The OpenMI Document Series

OpenMI Standard 2 Specification

For the OpenMI (Version 2.0)

Grant agreement number LIFE06 ENV/UK/000409

Title	OpenMI Document Series: OpenMI Standard 2 Specification for the OpenMI (Version 2.0)
Editor	Roger Moore, Centre for Ecology and Hydrology, Wallingford, UK
Authors	The OpenMI Association Technical Committee (OATC)
Current version	v1.0
Date	30/11/2010
Status	Final © The OpenMI Association
Copyright	All methodologies, ideas and proposals in this document are the copyright of the OpenMI Association. These methodologies, ideas and proposals may not be used to change or improve the specification of any project to which this document relates, to modify an existing project or to initiate a new project, without first obtaining written approval from those of the OpenMI-LIFE participants who own the particular methodologies, ideas and proposals involved.

Preface

The OpenMI stands for Open Modeling Interface, which aims to deliver a standardized way of linking environment-related models. This document describes the standardized OpenMI 2 interface specification in detail.

This is part of the OpenMI report series, which specifies the OpenMI interface standard, provides guidelines on its use and describes software facilities for migrating, setting up and running linked models.

Titles in the series include:

- Scope
- The OpenMI 'in a Nutshell'
- OpenMI Standard 2 Reference
- **OpenMI Standard 2 Specification** (this document)

The *Specification* is intended primarily for developers. For a more general overview of the OpenMI, see the *Scope* document.

The OpenMI is maintained by the OpenMI Association and this document, along with other more detailed documentation, can be obtained from www.openmi.org.

The official reference to this document is:

> The OpenMI Association (2010) *OpenMI Standard 2 Specification for the OpenMI (Version 2.0)*. Part of the OpenMI Document Series

Disclaimer

The information in this document is made available on the condition that the user accepts responsibility for checking that it is correct and that it is fit for the purpose to which it is applied.

The OpenMI Association will not accept any responsibility for damage arising from actions based upon the information in this document.

Acknowledgement

The OpenMI Association's members would like to acknowledge the contribution of the European Commission in co-funding the HarmonIT and OpenMI-LIFE projects. In particular, we would like to thank the Commission's staff for their sustained encouragement and support over many years.

Further information

Further information on OpenMI-LIFE can be found on the project website, www.OpenMI-Life.org.

Information on the OpenMI Association and the Open Modelling Interface (OpenMI) can be found on www.openmi.org.

Contents

1 Introduction

This chapter provides background information on the OpenMI and gives a summary of its architecture. It also specifies the expected readership of this document and their assumed expertise.

1.1 Background

A model application is the entire model software system that you install on your 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 and the user interface generates input files for the engine. The user can run the model simulation e.g. by pressing a button in the user interface, which will deploy the engine (see Figure 1). The engine will read the input files and perform calculations and finally the results are written 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 is the distinction between model application, engine, model, engine component and model component.

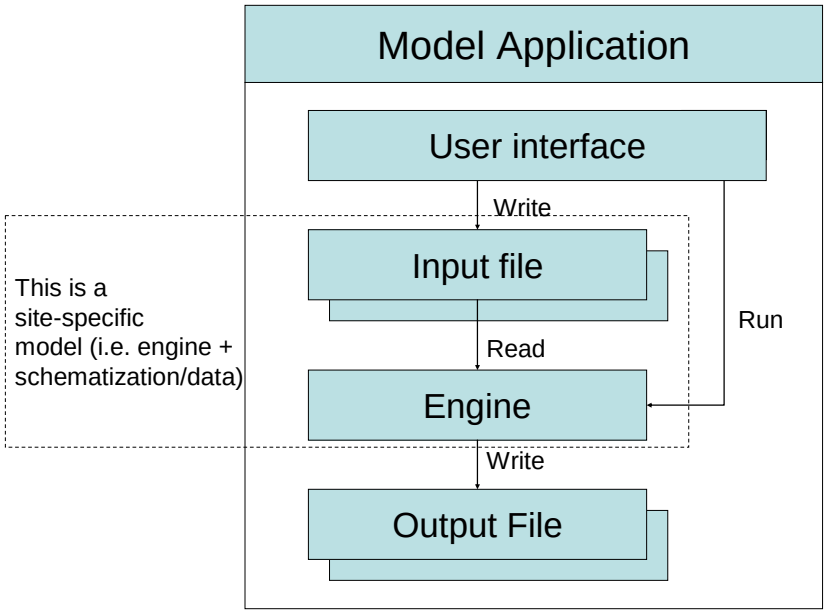


Figure 1 Model application pattern

Basically, 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 that OpenMI-compliant engine components must implement. When an engine component implements this interface it becomes a linkable component. A similar pattern can be applied for databases or other kinds of data sources. By turning them into components and implementing the OpenMI interface they become linkable components that provide direct access to its data at run time.

In summary, the OpenMI focuses on providing a complete protocol to explicitly define, describe and transfer (numerical) data between components on a time basis, including associated component access.

1.2 Overall architecture and layering

The interfaces of the OpenMI architecture – i.e. the Open Modelling Interfaces – are specified in the namespace OpenMI.Standard2. Software components that implement and use these interfaces properly are called OpenMI-compliant.

1.2.1 Base and extension interfaces

OpenMI 2.0 has become more versatile. Whereas OpenMI 1.4 was mainly restricted to models that progress in time, OpenMI 2 offers a set of base interfaces that are not aware of the type of a model. These interfaces can easily be extended, e.g. towards models that progress in time, or to models that express the exchange items and their values in terms of ontologies, instead of in terms of spatial and time scale definitions.

This implies that an OpenMI compliant component can now comply to the base interfaces (and indeed has to), but that it can also comply to one or more of these extension interfaces.

The current version of this document (November 2010) specifies the base interfaces and the extension supporting the time and space dependent component, the OpenMI.Standard2.TimeSpace extension. Future extensions could support parallel computing, compliance with the standards of the Open Geospatial Consortium OGC, and more. Implementing several extensions can combine the features of these extensions, e.g. a time and space dependent component that can run in parallel with other components.

1.2.2 Software Development Kit

To support the development of OpenMI-compliant components, a Software Development Kit (SDK) has been provided. This SDK consists of several parts, with a separate name space for each part (see Figure 2):

Oatc.OpenMI.Sdk.Backbone: a default implementation for the majority of the OpenMI.Standard2 interfaces.

Oatc.OpenMI.Sdk.DevelopmentSupport: some very generic support utilities.

Oatc.OpenMI.Sdk.Buffer: utilities for timestep buffering and time interpolation and extrapolation.

Oatc.OpenMI.Sdk.Spatial: utilities for spatial interpolation.

Oatc.OpenMI.Sdk.ModelWrapper: utilities to facilitate wrapping existing models.

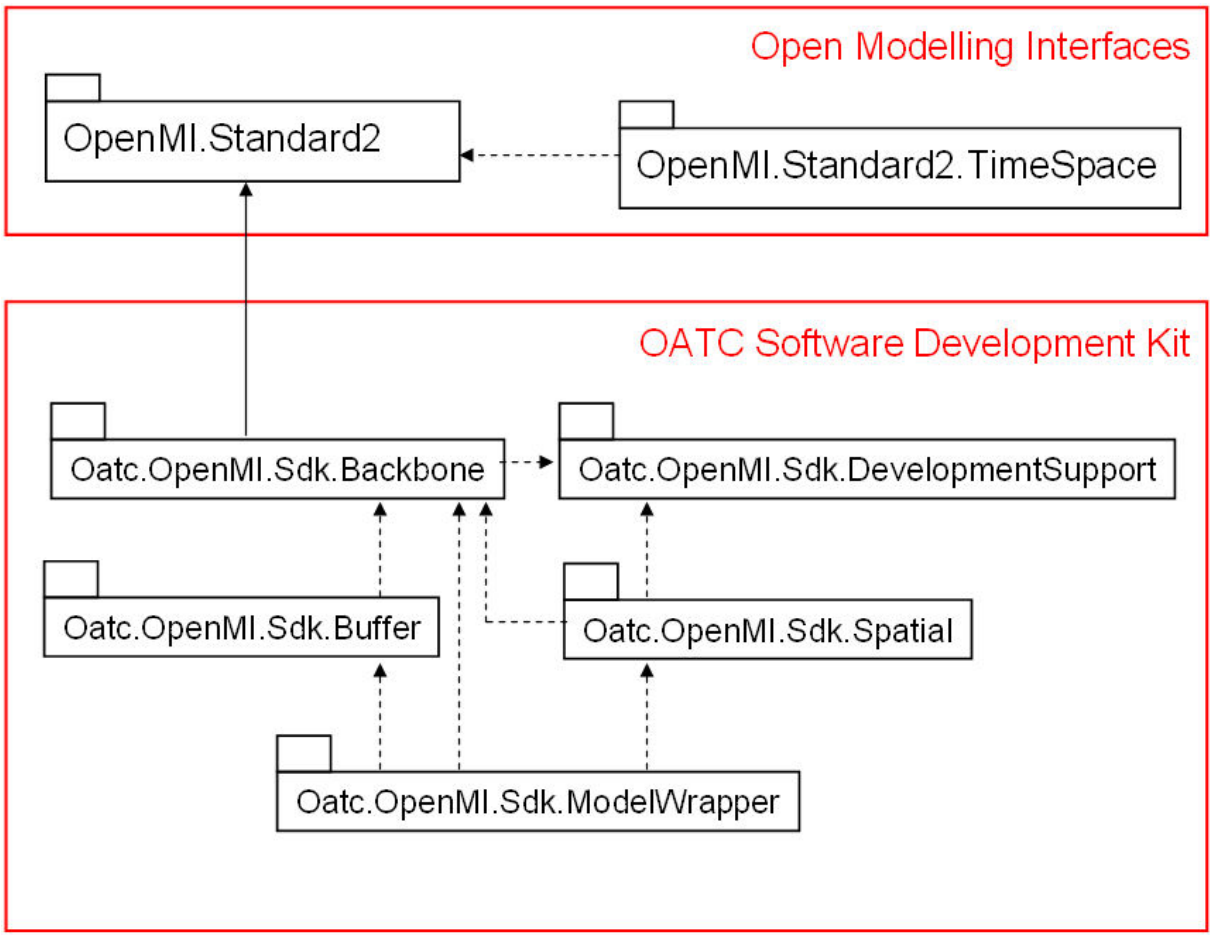


Figure 2 OpenMI standard and SDK namespaces

1.3 Document structure

Chapter 2 provides an explanation of the concepts underlying the Open Modelling Interface and its associated namespaces OpenMI.Standard2 and OpenMI.Standard2.TimeSpace. It highlights all issues that need to be addressed when defining an interface for linking models. Chapter 3 is the most important part of the document. It describes the formal specification of the interfaces in their static view (interface structure) and their dynamic view.

This formal specification is completed with the reference to the API description (see Appendix 1). Appendix 2 contains an overview of major changes since version 0.6 of the specification (May 2003).

1.4 Readership and expected expertise

This document is targeted at IT experts who would like to understand the concepts underlying the OpenMI as a means to formalize the linking of models and other components. It contains the formal interface specification, which needs to be adopted when you want to create your own OpenMI-compliant components. Other documents of the OpenMI report series can inform you about the scope of the OpenMI (OpenMI Association, 2010, Scope for the OpenMI), how to apply the OpenMI in

practice (OpenMI.org, 2010, http://www.OpenMI.org, 'How to …' pages) and how the OpenMI interfaces have been applied and implemented in the SDK (OpenMI Association, will be provided when the standard is formally released). Information about new extensions will also be given on http://www.OpenMI.org.

In order to understand this document, you need to have a basic understanding of model linking, object-orientation and UML-notation (particularly class diagrams and sequence diagrams).

Within the text, the following style-convention is applied:

- OpenMI interface
- OpenMI method
- OpenMI property
- OpenMI argument

2 OpenMI architecture

This chapter provides an explanation of the concepts underlying the Open Modelling Interface and its associated namespace, OpenMI.Standard2. It also highlights all issues that need to be addressed when defining an interface for linking models.

2.1 Introduction

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 model engine needs to be turned into a model engine component and the model engine component needs to implement an interface through which the data inside the component is accessible. The OpenMI defines a standard interface that such a component must implement to become an OpenMI-compliant model engine component. When a model engine component implements this interface it becomes a linkable component.

The OpenMI is based on the 'request & reply' mechanism. According to Buschmann et al. (1996), the OpenMI is a pull-based pipe and filter architecture that consists of communicating components (source components and target components) which exchange memory-based data in a pre-defined way and in a pre-defined format. The OpenMI defines the component interfaces as well as how the data is being exchanged. The components in the OpenMI are called linkable components to indicate that the process 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. In principle, data exchange in the OpenMI architecture is triggered by repeatedly invoking an update method on the component at the end of the component chain, until that component indicates it is done. In that update process, components can exchange data autonomously without any type of supervising authority. If necessary, components start their own computing process to produce the requested data. When output needs to converge to a certain criterion, a linkable component or another software component with controlling functionality might need to be incorporated. In that case, there need not to be an explicit end of the component chain.

Most important. however, is the fact that the OpenMI is not based on a framework; it only has linkable components.

2.2 Software perspective: Define, Configure, Deploy, Execute

Software development for developing a model-linking standard must start from the viewpoint of how the standard will be used. For running a linked simulation four main phases have been identified:

1. *Define*
   Definition of the data that can potentially be exchanged as well as the definition of available linkable components and models (linkable component + schematization).

2. *Configure*
   Configuration of the actual data that will be exchanged and the components (+ schematization) between which the data will be exchanged.

3. *Deploy*
   Construction of actual components (+ schematization) on a target system.

4. *Execute*:
   Actually running the linked computation, i.e. data exchange between components and computation by components.

To support all these phases, a model linking standard must explicitly define how the following aspects are covered:

1. *Generic model access*
   A common interface for all linkable components to allow the data exchange to be done in a generic way without requiring information on which linkable component types are used (see Section 2.3). It also involves the generic construction of a specific model without knowing its details.

2. *Data definition*
   To allow data exchange between models the data must be defined in some way. This includes the definitions of how quantities, space, time and the data itself are described (see Section 2.4).

3. *Definition of actually exchanged data*
   For this purpose, a provider/consumer relationship is established between one component's output and another component's input (see Section 2.5).

4. *Data transfer*
   This requires that the standard specifies the control flow of the overall system, and includes the mechanism to ask a component for values (see Section 2.6).

The OpenMI addresses all these issues but not more than this. The data being passed between models will typically be boundary conditions (e.g. fluxes) but transfer of model parameters and other data sets is possible. The OpenMI standard facilitates data exchange on both a time basis (either time stamp or time span) and a non-time dependent basis, being as complete as possible in describing the data being exchanged.

Apart from the core functionality of model linking, the standard also describes event handling mechanisms that can be used to implement generic tools for tracing, logging and online visualization, etc. (see Section 2.8). Assumptions underlying the architecture are discussed in Section 2.9, while Section 2.10 addresses considerations on efficiency.

2.3 Generic model access

Generic model access is essential for the component-based paradigm adopted in the OpenMI. At the core of the OpenMI is one basic interface to access a model component, the ILinkableComponent interface. This interface includes a section for initialization, a section for introspection and linkage configuration (description of exchange items and creation of links) and a section to exchange data at run-time.

Since all access to a component is through this interface, generic OpenMI implementations can be made independent of underlying type of engine or component being used. This approach allows the addition of new components to an existing OpenMI run-time environment without modifications to the environment.

The most important properties of the linkable component are its inputs and outputs, because they determine what can be exchanged and what will be exchanged. Once an input item is connected as a consumer to one of the outputs of another component, this connection determines that run-time data

can be retrieved; when an output item is connected as provider to one or more inputs of one or more other components, this connection states that run-time values have to be provided. The actual data exchange is done by invoking the GetValues() method on an output item. This method returns the data requested[1], if necessary after the linkable component has done some computation by invoking its Update() function. This function lets the component go to its next state (e.g. the next timestep for a time progressing component). It can be called repeatedly, until the component indicates it is complete.

Quite often, the values produced by a certain output item do not fit the way the input item requires them. In that case the result be adapted by adding an adapted output item to the output item. The adapted outputs usually are provided by the linkable component, but the can also be provided by other components.

For some situations – e.g. to enable iteration – it is useful if a linkable component can manage its state on request. A state management interface has been introduced for this purpose. This implementation of this interface, IManageState, is optional. It is up to the code developer to decide if states can be saved, and if so which state-related data is 'saved' and how it is done (e.g. in memory, in a file).

In other situations – e.g. in operational forecast systems – the system may need to store one ore more model states: for instance, to be able to restart from one of these states. Given the fact that in general the operational system will decide where to store all available states, the system needs to be able to retrieve a model state in a neutral, yet storable way. For this purpose the IByteStateConvertor has been introduced as an optional additional interface to the ILinkableComponent.

Essentially, all components exchanging (model) data are linkable components. Examples of linkable components are:

- Simulation engines for rivers, groundwater, general 3-D flow and rainfall-runoff processes.
- Measuring devices that need to be accessed online.
- Monitoring databases containing historic data.
- Data-driven models such as artificial neural networks.
- Analytical functions, used, for instance, to impose a certain boundary condition.

To locate and access the binary software unit implementing the interface, the OMI-file has been defined. The OMI-file is an XML file of a predefined XSD-format, which contains information about the class to instantiate, information about the assembly hosting the class and the arguments needed for initialization. Details of the OMI-file are provided in Section 3.2.5.

2.3.1 Wrapping legacy code

For legacy code, wrapping will most often be the technological choice to migrate to the OpenMI. An existing model engine (i.e. a computational core, often developed in Fortran 90) is encapsulated in a so-called wrapper that meets the interface specification of an OpenMI linkable component[2]. The OpenMI interface allows it to be treated in a generic way by an OpenMI run-time environment. Actually, the 'wrapper' turns the computational core into a linkable component for the OpenMI.

[1] This approach relies heavily on the generic definition of data described in Section 2.3.

[2] A default implementation is provided in the Oatc.OpenMI.ModelWrapper package.

2.4 Data definition

An essential part of standardizing data exchange is the metadata specifying which components, model schematizations and data sets are involved and what can be exchanged in terms of quantities (what it represents), geometry (where it applies) and time (when it applies). The availability of this metadata varies from fixed in the code or semi-fixed in a model script to highly variable, e.g. determined by the (run-time) settings of the model combination.

For most models, the model code determines which quantities can potentially be provided as output or are (potentially) needed as input. Often the code determines the location – i.e. geometry – where data can be exchanged (e.g. on the boundaries or on the full domain). In a site-specific model, the code is populated with site-specific schematization data (e.g. a network or grid with attribute data). With this data, the exact elements are known, including their position in the topology.

So-called 'exchange items' are defined to describe the data that a component can provide (the outputs) or accept (the inputs). Each input or output item that can be exchanged contains information on the quantity it represents, optionally the geometry where it applies, and optionally the time frame for which it applies. These three properties clearly determine which data produced by one model can be provided to which ingoing boundaries of another model (e.g. the ‘bottom’ of a river with the ‘top’ of a ground water model).

The list of exchange items, i.e. the list of inputs and the list of outputs, is specific for a combination of a model code with a model schematization. The exchange items are especially used during design/configuration time to set up the links between different models.

Of course, in many cases, the output of one component does not directly fit the needs of the receiving component. In that case, the output somehow has to be adapted to the needs of the requester.

So the data definition issues for standardized data exchange can be summarized as:

- *What* is exchanged?
- *Where* are the values defined?
- *When*, i.e. for what times, do the values hold?
- *How* are the values transformed, if needed.

These aspects are described in the following sections.

2.4.1 The OpenMI Standard and its extensions

The OpenMI Standard has been split in a base part and in extensions to this base.

The base interfaces are the core of the OpenMI 2.0 Standard, and define a Linkable Component (representing a model) in a very generic way. In fact it is so generic that it can only be used for the exchange of not described multi-dimensional data between components. Basically nothing more is known about the data than its programming language data type (e.g. double, integer or string). For a more meaningful exchange of data one or more of the available extensions need to be implemented too. In other words, the component needs to comply to the interfaces defined in such extension.

Putting it as the frequently used USB metaphor for OpenMI: The base interfaces only provide the connecting cable to exchange bits, while the extensions provide the protocols that give meaning to the bits.

The current packaging of the OpenMI 2.0 Standard already includes the TimeSpace Extension, that encapsulates the time / space specific interfaces of OpenMI. Most of these (e.g. ITime, IElementSet) are already known from OpenMI version 1.4. Future extensions could define support for parallel computing, compliance with the standards of the Open Geospatial Consortium (OGC), support for semantic data types, and more. Implementing several extensions can combine the features of these extensions, e.g. to create a time and space dependent component that can run in parallel with other components.

The following definitions about what is exchanged and where, when and how the exchange happens refer to both the base interfaces of the OpenMI Standard 2 and to the time and space dependent extension.

2.4.2 What

The physical semantics of exchanged values is described by a so-called value definition. This value definition is either a quantity, combined with a unit in which the quantity is expressed, or a quality, combined with the list of categories that describe the possible values of the quality. The design is such that in the future, additional types of value definitions can be added if needed.

Examples of quantities are water level (m) and amount of flow (cubic metre/hour). The physical nature of a quantity can be related to the shape type of an element. For example, a water level can be given at a point or along a poly-line or over a plane, but not in a volume. Examples of qualities are land use (agriculture, recreation, housing, industry) and soil type (clay, sand, peat).

The OpenMI does not use a standardized data dictionary. However, to ensure that linkages between quantities are correct, two aspects of quantities received specific attention. First of all, for every quantity, units are defined as well as a conversion formula of the form a*x + b to enable unit conversion from the quantity's unit to standard SI units. This allows straightforward linking of quantities of different units without the performance penalty of unnecessary transformations. Second, the dimension needs to be provided for each quantity. By convention, this dimension is expressed as a combination of base SI quantities.

2.4.3 Where

The space where the values apply is indicated in a finite way by an ordered list of elements (the element set[3]), where conceptually each element consists of a number of connected vertices. An element holds an ordered set of vertices where the element shape type determines the minimum number of vertices of an element as well as the semantics of ordering. In this way topology is described. Within one element set all elements must have the same type. Elements have an ID and may, but do not need to be, geo-referenced. Elements described in more detail by vertices are geo-referenced, as the vertices contain the co-ordinates to locate the element in a geo-reference system

Every element set contains a string reference to a geo-reference system, which defines a co-ordinate system for the co-ordinates (e.g. WGS84[4]). This allows element sets for every model to be defined in the c-oordinate system of the model without transformation and allows an OpenMI implementation to perform the co-ordinate mappings between different geo-reference systems whenever needed. A geo-referenced element may be anything from a point, line, polyline, polygon (e.g. horizontal or vertical plane) to a polyhedron (i.e. a volume).

[3] In reality, the element set is not a set, as in an unordered collection, but a one-dimensional array of elements, where ordering is essential.

[4] World Geodetic System 1984 – see http:\\www.wgs84.com

Every element has an ID that is unique within the element set it belongs to. Components may offer data operations for geo-referenced mapping (i.e. based on co-ordinates) or ID-based mapping.

2.4.4 When

Time in the OpenMI is defined by a time set: a set of either time stamps or time spans. A time stamp is a single point in time, whereas a time span is a period from a begin time to an end time. A time is represented by a Modified Julian Day value and a duration. If the duration is zero, the time is a time stamp. If it is greater then zero it is a time span.

2.4.5 How

In the simplest case, the values returned are exactly the data as it is computed by a computational core of the linkable component. Nevertheless, under many conditions, these computed values have to be transformed in some way. In this case, an adapted output item can be created that takes the values from its adaptee (the output item that serves as a source for the adapted output) and transforms it into the required values.

The most common example of an adapted output will be one that performs temporal aggregation over a time span. This functionality can be used if one model, running a small timestep, needs to feed another model, which runs at a large time interval. The functionality may also be used to suppress a temporal interpolation method in case the timesteps of the source component and the target component do not coincide. In addition to temporal adapted outputs, spatial adapted outputs are likely, as well as a miscellaneous group of other adapted outputs.

2.4.6 Values

The values themselves are represented as a two-dimensional array of value objects contained in a so-called value set. Information about the values (space, time, quantity) is not part of the value set; it is assumed to be known by the code that uses them. The two dimensions are for time (the list of time stamps or time spans) and space (the elements in the element set). Generally the first index represents the time and the second index the space, but the value set has a flag that may indicate that the ordering is the other way around.

For time-independent data, the size of the time dimension in the value set is 1. For data that has no space definition, i.e. that has no element set, the size of the space dimension is 1.

The value objects contained in the value set can be of any type. The most commonly used type is the double precision value, but there are many quantities that can be expressed by an integer value, by a boolean value or by a user defined type, like a vector, or a list of possible categories.

2.5 Specification of actually exchanged data

This section provides information on the provider-consumer relation between outputs and inputs. There is also a description of adapted outputs and the setting of input values.

2.5.1 The provider-consumer relation between outputs and inputs

As indicated, a linkable component can retrieve data from another linkable component by invocation of the GetValues() method on one of the outputs of the other component. Which data will actually be exchanged during run-time is fully specified by establishing provider-consumer relationships. The fact

that an output item has been connected as provider to one or more inputs of one or more other components, and that at the same time these inputs have been registered as consumers, indicates that run-time values will be transferred from the output item to the input items.

If an input item has a provider, the input item will be fed with data during the computation. If it has not, the input item is not connected, i.e. is not involved in data exchange with other components (however, its values may still be set – see Section 2.5.3).

An output item can have zero, one or more consumers. Also, it can have zero, one or more adapted outputs that depend on it (see Section 2.5.2). If the output item has at least one consumer or at least one adapted output, the output item will be involved in the data exchange between components, i.e. will be asked to provide values.

2.5.2 Adapted outputs

If the quantity, the element set or the time set of a certain output does not fit the way the input item requires it, the output can be adapted by adding an adapted output to the output. As the name indicates, an adapted output in its turn is just an output again, so subsequent adapted output items can be added to the initially created adapted output.

Figure 3 shows an example of some sequences of adapted outputs. It may be clear that the adapted output approach offers great flexibility in defining the steps that have to be taken to transform that data from, for example, Output-1 to Input-2 and and Input-a.

The (adapted) output item that serves as a source for an adapted output is called the adaptee of the adapted output. It may be clear that a sequence of adapted outputs, as well as a direct provider-consumer connection, is unidirectional: from output item to input item.

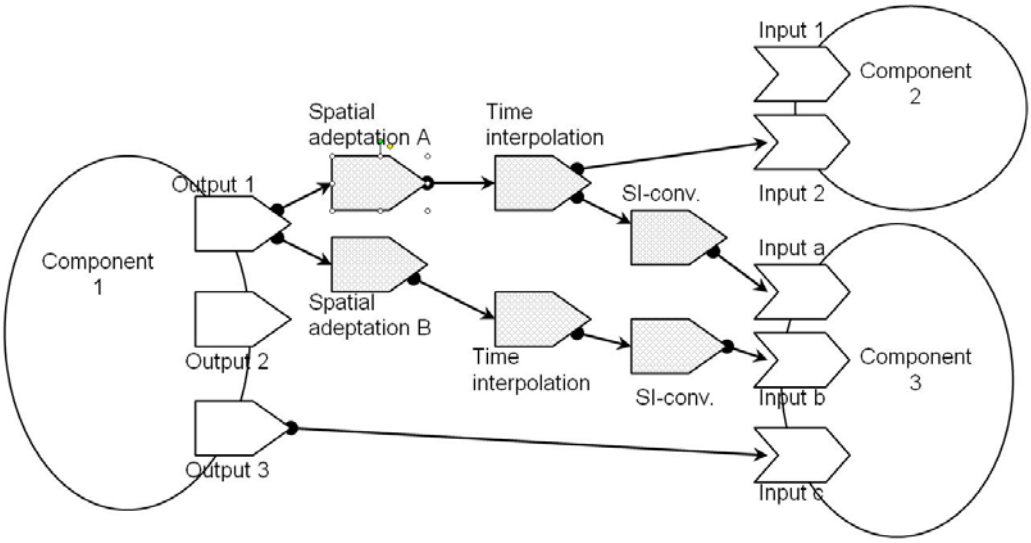


Figure 3 Example of adapted outputs

When creating an adapted output, arguments can be filled in to specify how the adapted output should do its work, for instance for the spatial adaptation: should it produce a weighted mean value when mapping to an element (e.g. for water level), or should it produce a weighted summed value (e.g. for a discharge).

Typical examples of adapted outputs are:

- Spatial interpolators
- Time interpolators and extrapolators
- Time integrators
- Unit convertors
- Quality to quantity translators

The adapted outputs for spatial and time issues may differ from extension to extension. The current extension for time and space dependent components defines a location in space by its x, y and z values, so these adapters have to implement this definition. Units however are more general applicable, and should therefore implement base interfaces instead of extension interfaces.

If components from different extensions exchange data, the key conversion will be done by an adapted output. In order to translate between the extensions, this adapted output has to be aware of the concepts in both extensions.

In general, adapted outputs are created by the adaptee's component. However, adapted outputs may also be created by any other component. The creation of adapted outputs is formalized by specifying a factory interface for it.

2.5.3 Setting input values

When a model is run in a calibration environment it is convenient to be able to set certain input values. For instance, if you want to calibrate a river model on bottom friction, you would like to adjust the friction a little before re-running the model. To facilitate this, OpenMI allows the values of input items to be set. Implementing setting the values of an input item is optional. If a component does not support it, an exception has to be thrown.

The values can only be set if the 2D-array with value objects contains exactly the right amount of values for the required number of elements and the required number of timesteps, as specified by the input's element set and time set.

The use of setting values in practice is limited to dedicated systems, such as a calibration tool. It is not used when running a 'regular' composition. Please note that, to really benefit from this functionality, you must be able to let the model re-compute; in other words, the linkable component must be able to re-initialize itself after having finished a computation (see Section 3.3 for these linkable component phases).

2.6 Data transfer

Data exchange, the core issue of OpenMI, is about linkable components that request data (i.e. the targets) and linkable components that provide this data (i.e. the sources). Every linkable component is a target component, a source component or both. Because of this request-and-provide approach, the OpenMI has been designed in principle as a pull-based pipe and filter system where the target component requests data from the one of the outputs of a source component and blocks (i.e. does not process any new call) until this data is returned.

An instance of a linkable component handles only one GetValues() request on one specific (adapted) output item at a time, before acting upon another request. By adhering strictly to this principle, the basic principles of the architecture are clear, leaving little room for interpretation errors. In particular,

the OpenMI architecture avoids the problems of multi-threading, thus allowing single-threaded thinking.

Since the target component pulls the data when needed, this one-GetValues()-at-a-time mechanism is called pull-driven. Details are explained in Section 2.6.2.

There are situations, however, for which the pull-driven approach might be too restrictive, e.g. in operation control systems, where it would be preferable to let every involved component check if new input has been provided and, if so, perform a computation as soon as all other required input is also available. Since such a system would typically loop over all components to let them check if they need to take action, this control flow is called the loop-driven approach. This extension is currently in November 2010 not available. But the basic interfaces already contain some methods and properties that will ease extension interfaces for the loop-driven approach.

2.6.1 Linkable component status

When a linkable component receives an Update() call to go to its next state, or a GetValues() call on one of its output items (which in its turn will usually lead to an Update() call on the component, the component enters a new ‘status’: it starts to produce output. This status in fact contains other recognizable sub-statuses. For instance, the component might first invoke a few subsequent GetValues() calls on the output items of other components and after that perform the actual computation. So it will pass states like ‘waiting for data’ and ‘updating’, ending as ‘updated’ when it has computed the required values.

The OpenMI explicitly specifies the possible states that a linkable component can be in. Before the first actual Update() call or GetValues() call can be performed, various states have been passed during the process of initialization and composing the various connections between outputs and inputs, e.g. ‘initializing’ and ‘validating’. Once one or more of the components have finished their computation (i.e. have reached that state ‘done’) a few last states are passed, such as ‘finishing’.

The linkable component status serves both an informative purpose (what is the component currently doing) and a control flow decision purpose. For instance, the actual computation can only be started if each component is in a valid state, which usually depends on whether all its required inputs have been connected to providing outputs. When two components are connected in such a way that they will retrieve values from each other, the distinction between the states ‘waiting for data’, ‘updating’ and ‘updated’ mentioned above plays an important role in avoiding deadlock during the mutual GetValues() calls – see Section 2.6.3.

The various states are described in Section 3.3.

2.6.2 Pull-driven communication

In OpenMI's pull-driven approach, linkable components can be connected in a chain, where invoking the Update() method on the last component in the chain triggers the entire stack of data exchange. Figure 4 illustrates three chain layouts:

- A unidirectional chain
- A logical switch to change from source component[5]
- A bidirectional chain

[5] Useful to implement a backup scenario in case a linkable component is down, or cannot provide the data requested.

The Update() method will be called repeatedly, until the states of all the components are ‘finished’.

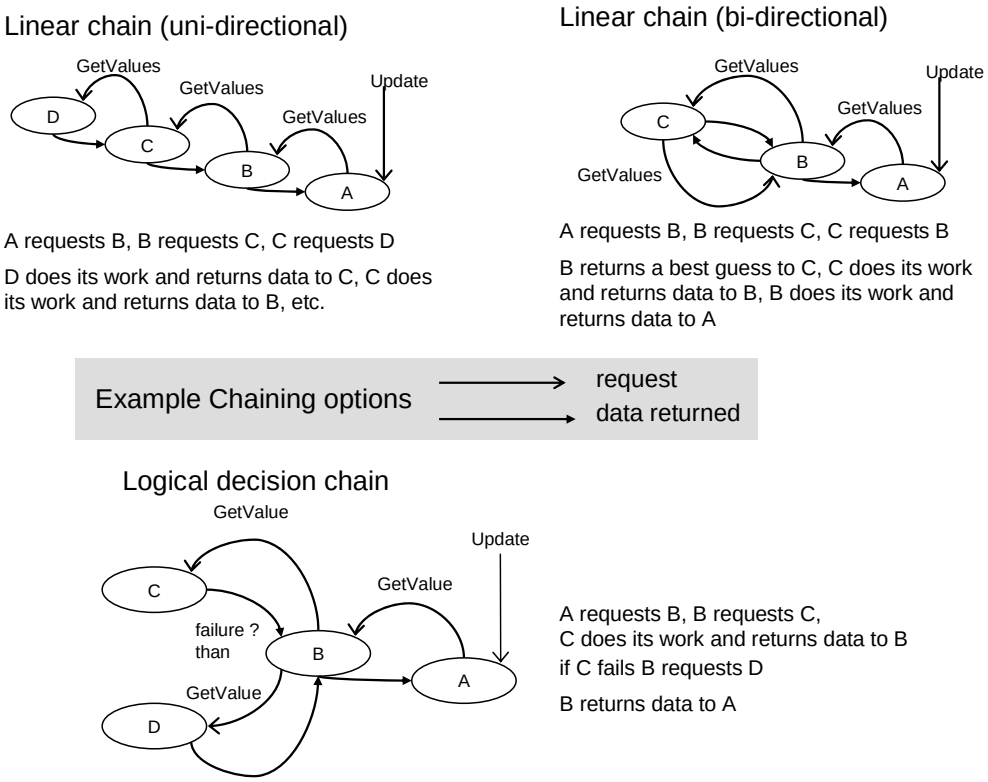


Figure 4 Different chain layouts with the pull-mechanism

A sequence of adapted outputs, as well as a direct provider-consumer connection, is unidirectional. Therefore, for bidirectional communication a provider-consumer relation is needed in each direction.

2.6.3 Bidirectional data exchange in pull-driven communication

The presence of a synchronous (blocking) GetValues() call requires some attention for the case of bidirectional data exchange or, more generally, in the case of cycles through the network of linkable components connected by provider-consumer relations. In this case, a straightforward implementation of the GetValues() call might lead to infinite recursion.

In the OpenMI, this problem is prevented by putting the obligation on a linkable component not to process additional GetValues() calls on its output items if it is already inside a GetValues() call on the same or another output item. This effectively prevents infinite recursion.

In case it is not possible to compute the requested values, either by a simulation or by interpolation, another solution must be implemented, such as returning an extrapolated value or returning the most recently computed value. The latter case may be useful for situations where iteration is implemented.

Bidirectional links might require specific iteration functionality to preserve numerically-stable solutions. The basis of this functionality is state management; let a linkable component store its state at some moment in the computation, and let it go back to that state if another iteration is needed. For this special case, the OpenMI requires a controlling mechanism that decides how to proceed. Simple solution mechanisms can be developed, implementing iteration-control functionality as a linkable component positioned in between the two computing models, or as some software component on top

of the involved components. Depending on the choice of the developers, this complex combination of linkable components might be encapsulated as a new linkable component.

2.6.4 Time synchronisation

Each GetValues() request initiates a processing activity (e.g. computation) when needed to respond properly. If models are involved that progress over time, the time specification in the GetValues() call is effectively the controlling variable for any processing activity. Linkable components that do not progress over time will not look at the time specification. They just do their work and return the data. However, they should be able to pass the time argument to another component if they invoke a GetValues() call themselves.

If the requested time stamp does not match the time-stepping in the computation and the computation is already ahead in its computation, an interpolated value must be delivered. Note that the model code developer decides how this interpolated value is computed and whether any buffering is to be applied to increase performance. When the computation has not yet reached the requested time stamp, two alternatives are available:

- Initiate a computation to compute the values at the requested time.
- Extrapolate the solution.

Under normal conditions, the first alternative is chosen. For bidirectional links, one of the components will need to extrapolate its solution in order to prevent deadlock situations. If a single entity in the value set cannot be provided, an invalid flag should be given. In the exceptional case that no value set can be provided, an exception needs to be thrown. Be aware that this is a serious exception as the entire computation chain might get stuck.

The above mentioned obligation has been formulated from the perspective of a simulation engine. However, (monitoring) databases and other linkable components, which do not progress in time, should also be able to return a value, whether it is the actual, interpolated or extrapolated one. It is up to the software developer to introduce an intelligent (or customizable) wrapper for this purpose. For those situations where the exact time information is required, an additional interface is provided to obtain the discrete time stamps available. Implementation of this interface, IDiscreteTimes, is optional.

2.7 Events

Within modern software systems, events are often applied for all types of messaging. In the OpenMI, the linkable component uses that event mechanism to tell the outside world that things have changed in the status and values of the linkable component.

Two types of events are distinguished: the ‘component status changed’ event and the ‘exchange item value changed’ event.

The event mechanism is also used to facilitate pausing and resuming the computation thread, as the computation process of an entire model chain is rather autonomous and is usually not controlled by any master controller (see Section 3.3.7). When a component receives the thread, it should preferably send an event, so listeners (e.g. a GUI) can grab and hold the thread, and thus pause the computation by not returning control. In normal conditions, the control is returned so the component can continue its computation. Of course the computation is also controlled at the level that triggers the first component of the chain by means of an Update() call. Stopping firing those calls will also result in a paused system, although it may take a while before an entire call stack completes its processing activity.

2.8 Assumptions underlying the OpenMI architecture

This section describes the assumptions made by the OpenMI, which need to be taken into account when developing OpenMI-compliant models.

2.8.1 Time is referenced

Time is considered essential in many of the applications for which the OpenMI is used, i.e. in the applications that use the OpenMI.Standard2.TimeSpace extension. While numerous models just run timesteps without knowledge of the associated calendar, this knowledge is required in order to link them to other models. Hence, all time information in the OpenMI is expressed as a Modified Julian Day and the offset from UTC time (see Section 0).

Generally, static models do not bother about time. However, to fit into an OpenMI system, static models need (i) to react when being invoked with a GetValues() call and (ii) be able to pass the time argument to another linkable component when appropriate.

2.8.2 The providing component knows best how to convert data in time or space

The OpenMI standard puts as much responsibility as possible at the level of the providing linkable component. This is because the providing linkable component has most knowledge about the data it provides (and the internal data it does not provide), so it is best equipped to perform any transformations. If data is not available at the requested time or element set, the providing linkable component knows best, in general, how the available data should be processed to deliver a value at the requested time and element set, so that is why the linkable component is asked for available adapted outputs. Of course the component is free to delegate this to third-party adapted output factories.

2.8.3 Knowledge gaps need to be filled with domain expertise

Even though the OpenMI specifies how data, space and time are represented, some domain knowledge may still be needed to actually link two model engines that have been developed independently. Figure 5 illustrates the various layout options for the example of a groundwater-river model linkage.

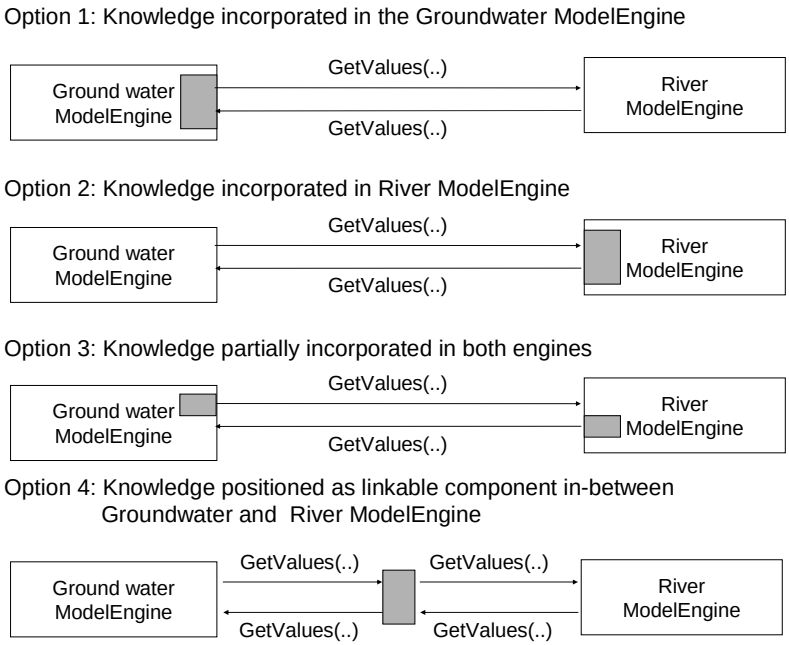


Figure 5 Example of different layouts to capture knowledge needed to link two water domains

Note that option 4 is the classical approach for model linkages, adopted by frameworks for model linking as well as by many dedicated linkages. Options 1, 2 and 3 anticipate, up to a certain extent, the capabilities of the outside world. Dependent on the software capabilities and the way a link is configured, the OpenMI accommodates all of them. Adapted outputs may be utilized to keep control over knowledge interpretations.

2.8.4 Compability of adapted outputs

OpenMI Version 2 allows for adapters tailored for specific applications, i.e. for specific extensions to the Standard's base interfaces. Hence it might happen that a combination of an adapted output and another (adapted) output is invalid. It can be reasonably expected that adapted output implementations from the same software source (SDK) should either be compatible or create sensible runtime warnings if not (if developed well). This might however not be the case when mixing adapters from different software sources. Hence, when using adapted outputs care should be taken to understand their functionality and maybe even implementation details.

2.8.5 Nesting, logical switches and other intelligence does not need additional classes

As the OpenMI Standard is based on the OO-concept of encapsulation, no specific classes are introduced to facilitate features such as nesting of components, logical switches, iteration controllers or optimization and calibration tools. All those features can be implemented in a linkable component, if required.

Nesting components

A linkable component in the OpenMI is not limited to representing just one (existing) model engine. In practice, it is possible that one linkable component represents one (composite) model component, which is able to simulate many types of (water-related) physics. In a similar way, new linkable

components can be created which consist internally of already available linkable components. Such a component would constitute a composite linkable component.

A composite component could be more easy to use than the individual components of which it consists. If a specific combination of components is often used in a particular company, this component would be ideally suited to become a predefined composite component.

Despite its practical use, many ways exist to develop a composite linkable component. For instance, for commercial reasons, the composite linkable component would preferably hide some functionality of the internal components that was not paid for. For these types of reasons the standard does not define a composite linkable component class that can be used.

It is recommended that developers check the OpenMI site (OpenMI.org, 2010) every now and then for examples of how to implement such composite linkable components using the default linkable component interface.

Implementing iterative methods

In the case of iterations between components, a separate controller is needed for the following reasons:

- It should be possible to modify the iteration method without modifying the components used in the iteration.

- Depending on the iterative method, the components should be triggered in different ways to compute (GetValues()) and/or save and restore state.

The OpenMI allows this to be done while staying within the concept of the pull-based pipe and filter architecture. This is done by putting the iteration algorithm inside a separate controller linkable component and connecting the controller to the individual components involved in the iteration.

An alternative will be to use the loop-driven approach, an extension which is currently in November 2010 not available. This approach provides more flexibility in steering which linkable component gets which value first. For example, the controller could take values from both involved components, feed them to the other and then let both components update themselves to their next state. Once again, it is recommended that the OpenMI site (OpenMI.org, 2010) be checked when the loop-driven extension will be available.

2.9 Miscellaneous issues

This section describes some efficiency considerations and provides information on distributed computing.

2.9.1 Efficiency considerations

There are a number of ways in which models can be made more efficient. Some of those considerations are given here.

Exposing metadata

Although the methods to expose metadata are in the same interface as the methods for run-time data exchange, a clear phasing can be maintained in the implementation as a method call to prepare for computation provides a clear shift from establishing the connections towards the computational phase. Developers can choose whether they expose metadata that is captured in files or whether they query engines. The former may be handy in multi-user environments and in multi-processing jobs where you want to reduce the demand for CPU capacity on the computational cluster.

Preparation before computation

The Prepare() method is designed to enable preparation of internal buffers and data mapping matrices just before computation time. This method reduces the performance loss at computation time, if compared to a situation where the mapping needs to be made each time the component is asked to perform its Update() step or is asked for data through a GetValues() call. In addition, its inclusive validation may save expensive CPU resources when some engines are not ready.

Pro-active computing

A straightforward implementation of the GetValues() call, where the GetValues() call initiates a computation for the requested value, can also be inefficient. In many cases, efficiency can be improved by a latency-hiding technique where a linkable component in the GetValues() call returns an already computed value and computes (one or more) timesteps ahead in time. This effectively constitutes a way of doing parallel computations in OpenMI.

2.9.2 Distributed computing

The general control flow mechanism in the OpenMI, the pull-driven approach, has been designed in such way that the computational process calls (the GetValues() calls on one of the outputs of a linkable component) are handled in one thread. The OpenMI leaves the choice open to the software developer as to whether the processing of a request is handled internally by starting multiple threads or by a parallel computing session. The OpenMI standard is well suited for distributed computing since it is defined in terms of objects and messages between these objects. The objects can run anywhere as far as the standard is concerned and the messages between these objects can be sent based on network communication.

A distributed computation can be implemented using established design patterns such as proxy-stub (see e.g. [1]) and using established middleware (e.g. Java, .NET, Web services) for the realization of the communication.

Distributed computation may also give rise to questions such as authentication, authorization and accounting (AAA). The specification and implementation of distributed computing (including AAA) are excluded from the OpenMI standard and are left to implementers of OpenMI linkable components.

OpenMI 2.0 intentionaly does not address the parallellism issue yet, but the intention is that by the combination of the linkable component's Update() call and Status property, and some limited extensions, parallel execution of components can be supported in a later version.

3 The OpenMI Standard2 namespace

This chapter describes the formal specification of the interfaces in their static view (interface structure) and their dynamic view.

3.1 General description

The scope of the namespace, its software package and its relationship to other namespaces are described here.

3.1.1 Scope

The OpenMI Standard2 namespace[6] specifies a platform and technology independent interface to describe, define, enable and troubleshoot data exchange between (model) components. Software components that adopt this interface are called OpenMI-compliant and can be linked (either hard coded or by a configuration utility) to other OpenMI-compliant components.

The interfaces are composed of low-level data types such as strings, integers, doubles, booleans and objects.

The usage of the OpenMI Standard2 namespace is the mandatory part of any OpenMI-compliant software component. Therefore it has been chosen to create a list of interfaces, which is as minimal and as complete as possible, to define exactly data that is being exchanged. To reduce the efforts of developing OpenMI compliance, it has been decided to keep convenience functions out of the standard, unless real-world applications prove that they are really needed.

Section 3.4 indicates which interfaces are mandatory to be OpenMI-compliant. The extensions to the standard, e.g. OpenMI.Standard2.TimeSpace for time and space dependent components, are optional.

3.1.2 Packages

The OpenMI Standard2 namespace is composed of one software package, the OpenMI Standard2 package containing the public interfaces of the standardized OpenMI.

3.1.3 Relationship to other namespaces

OpenMI Standard2 is the independent interface specification. The OpenMI Software Development Kit (SDK) provides a default implementation for the majority interfaces through the Sdk.Backbone package. Other packages within the SDK domain, like Sdk.Buffer, Sdk.Spatial and Sdk.ModelWrapper, provide utilities to facilitate creating linkable components and adapted outputs. They typically utilize the standard interfaces, and depend on the backbone implementation (see Figure 2).

[6] In .NET, the namespace is OpenMI.Standard2. In Java, it is org.openmi.standard2.

3.2 OpenMI Standard2: static view

This section provides a formal specification of the interfaces in their static view.

3.2.1 Identification and/or description of entities

Many of the OpenMI interfaces describe entities that will be shown in the user interface. Therefore, these entities need a caption and, if applicable, a more comprehensive description.

Besides of that some interfaces describe an entity of which the instances need to be identifiable, to be able to refer to these instances when needed. For example, when an existing composition is read from file, the connections between outputs and inputs can only be re-established if the involved exchange items can be selected from the list of outputs and inputs. Therefore an exchange item needs to be identifiable.

To unify the identification and description of all kinds of entity instances, two interfaces are provided: IDescribable and derived from that IIdentifiable (see Figure 6). The majority of the OpenMI.Standard2 interfaces are derived from one of these interfaces..

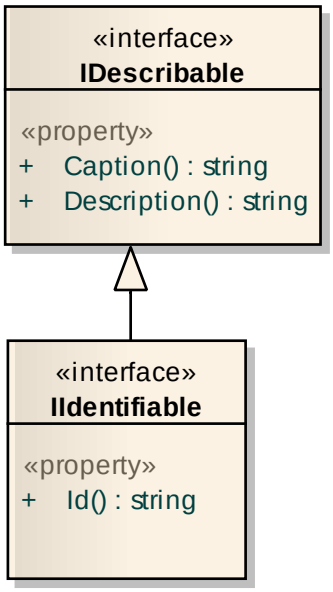


Figure 6 IIDentifiable and IDescribable interfaces

3.2.2 Data definition interfaces

Correct interpretation of data being exchanged requires information on the semantics for each value passed between components: what does it represent, where does it apply, when does it apply and how is it processed. This is information is expressed by the IBaseExchangeItem interface and the various interfaces that are derived from that (see Figure 7).

For a linkable component, an exchange item either is an input item (IBaseInput) or an output item (IBaseOutput). Time and space dependent linkable component will provide and recognize the ITimeSpaceExchangeItem, that is derived from the IBaseExchangeItem. A time space exchange time is either a ITimeSpaceInput or a ITimeSpaceOutput

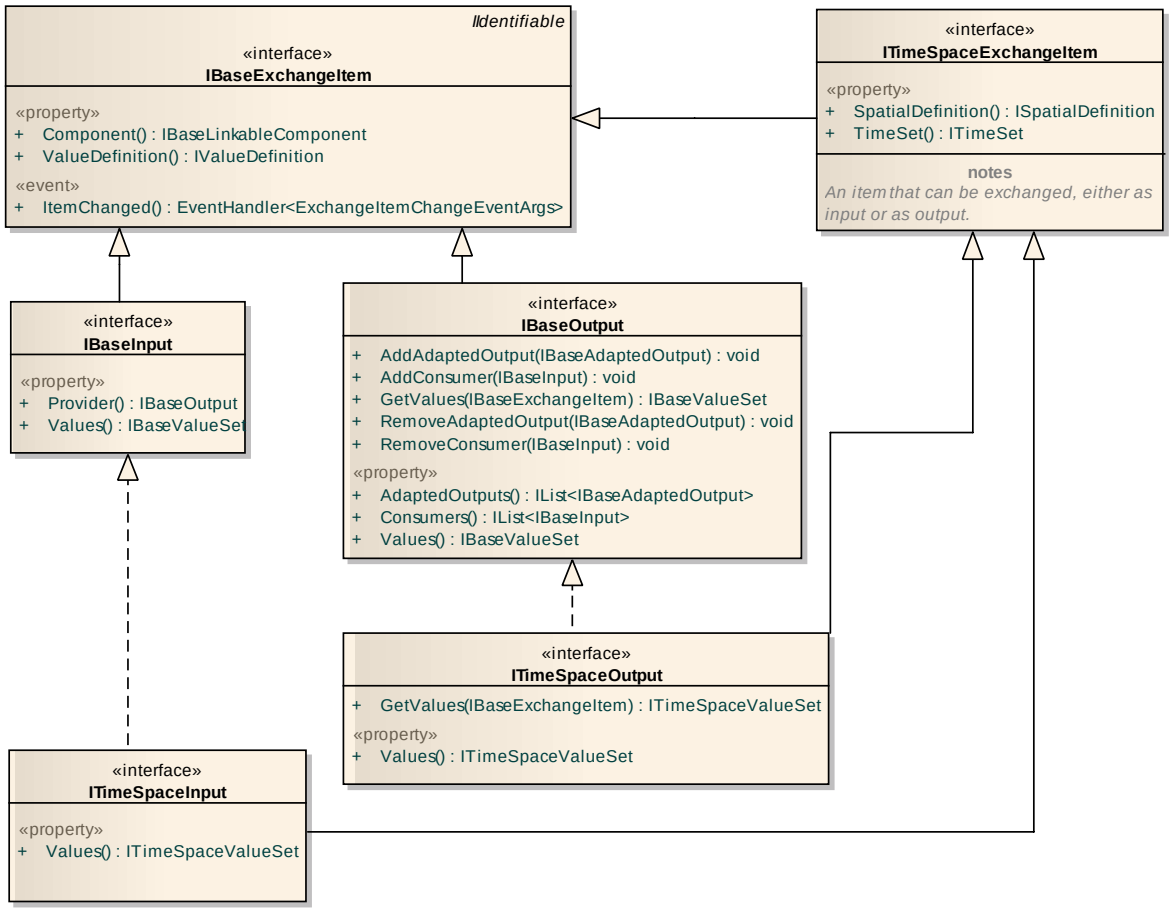


Figure 7 IExchangeItem interfaces

The semantics of the exchange data is defined by the what/where/when properties of the exchange item:

- The IBaseExchangeItem.ValueDefinition, i.e. a quantity, including the unit in which it is expressed, or a quality including its possible categories
- The ITimeSpaceExchangeItem.SpatialDefinition, defining the locations where the values are hold (usually an element set)
- The ITimeSpaceExchangeItem.TimeSet, defining the involved time stamps or time spans

These data definition interfaces are discussed in more detail in the remainder of this section.

Some properties and methods of the exchange items (Provider, AddConsumer, AdaptedOutputs etc.) are used for the connections between the outputs of one component and the inputs of another component. These methods are described in Section 3.2.3.

The Values property of the IBaseInput and its derived interfaces can be used for applications like calibrations and decision support systems. The current value of the input item can be retrieved, after which it can be slightly modified and be set.

The GetValues() method and Values property of the IBaseOutput and its derived interfaces are used during the actual computation. They will be discussed in Section 3.3.5, when the dynamic view of OpenMI is described.

IValueDefinition / IQuantity / IQuality

To enable proper linkage of data without a data dictionary, a metadata structure is needed that provides sufficient facilities to describe the semantics and enable automated checks on the semantics. Figure 8 illustrates the metadata structure defined in the OpenMI.

IValueDefinition defines the interface to indicate what the values represent, i.e. model variable or model parameter to be exchanged. A value definition is an IDescribable, in other words, it is presented by its caption (typically a short name) and is described by a description string (more extensive information for correct interpretation of the semantics, direction etc.)

The ValueType property indicates what type of object is returned when retrieving one of the values of the set, i.e. the value for a certain element and time.

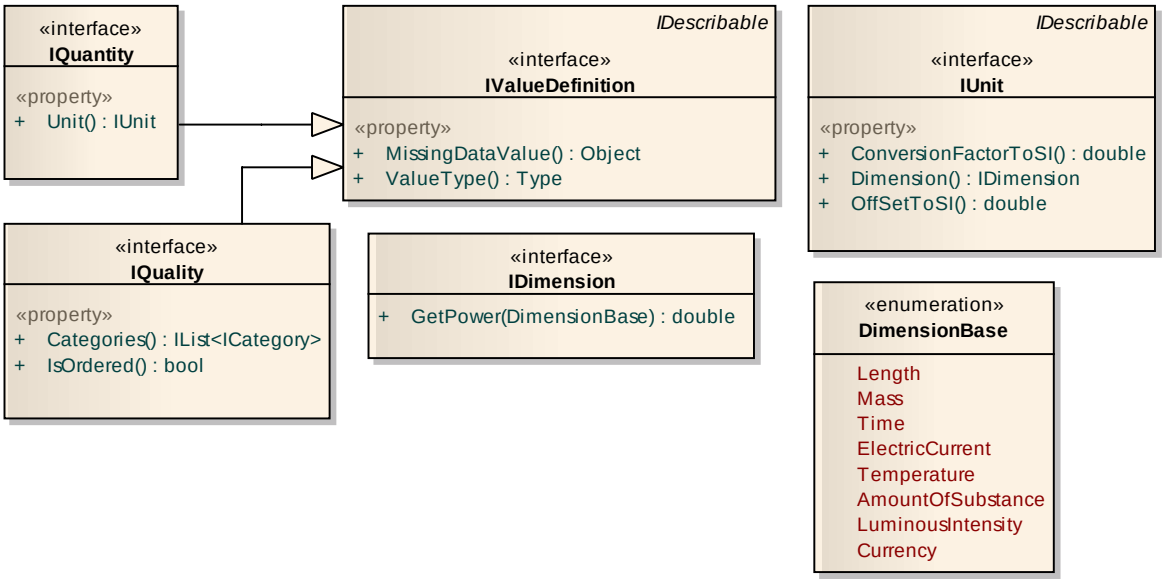


Figure 8 IQuantity and related interfaces

One of the inheritors of the IValueDefinition is the IQuantity. This interface defines how to specify quantifiable data: by means of the quantity's Unit.

The IUnit interface is defined to indicate the unit[7] in which a quantity is expressed. The IUnit interface contains sufficient information to facilitate unit conversions between quantities. For a given value v of a certain quantity, the conversion to the SI value s can be done using the following computation:

s = Unit.GetConversionFactorToSI() * v + Unit.GetOffsetToSI()

To enable (physical) dimension[8] checks between quantities, the IDimension interface has been defined. A dimension is expressed as a combination of base dimensions, derived from the SI system[9],

[7] A unit has a definite magnitude, and can be used as the basis for measuring other things. The inch is a unit. The foot is a different unit, because it has a different magnitude.

with a minor extension for currencies. This interface provides a method to obtain the power for each dimension base, as well as a method to check if two dimensions are equal.

For example, a discharge expressed in units of m3/s has dimension Length^3Time^-1. Table 1 illustrates the base quantities and the associated SI units.

Table 1 Base units and dimension base in the OpenMI (derived from SI)

Dimension base	SI base unit	symbol used
Length	meter	m
Mass	kilogram	kg
Time	second	s
ElectricCurrent	ampere	A
Temperature	kelvin	K
AmountOfSubstance	mole	mol
LuminousIntensity	candela	cd
Currency[10]	Euro	E

Note that some units are dimensionless, represent logarithmic scales or have other difficulties when expressed in SI. In such cases you should pay extra attention to the descriptive part of the unit, to ensure that the user who defines the link has a proper understanding of the quantity.

The other inheritor of the IValueDefinition is the IQuality. The IQuality specifies the list of Categories that describe the possible values of the quality, e.g. 'hot' and 'cold' for temperature indication, or 'sand', 'clay' and 'peat' for soil types.

The IsOrdered property indicates whether an ordering can be recognized in the categories or not. If IsOrdered is True, one category represents a 'higher' value than another category (e.g. 'very light', 'light', 'heavy', 'very heavy').

IElementSet

Data exchange between components in the OpenMI is nearly always related to one or more elements in a space, either geo-referenced or not. An element set in the OpenMI can be anything from a one-dimensional array of points, line segments, poly lines or polygons, through to an array of three-dimensional volumes. As a special case, a cloud of ID-based elements (without co-ordinates) is also supported, thus allowing exchange of arbitrary data that is not related to space in any way.

---

[8] A dimension describes the type of thing being measured, without specifying the magnitude. The inch and the foot both have dimensions of length.

[9] More information on the SI system can be found at the National Institute of Standards and Technology (http://physics.nist.gov/cuu/Units/).

[10] Currency has no base quantity in the SI system. Unfortunately, currency has conversion units that may vary over time.

The IElementSet interface (Figure 9) has been defined to describe, in a finite element sense, the space where the values apply, while preserving a course granularity level of the interface.

Please note that the more general part of the IElementSet has been put in a separate base interface, the ISpatialDefinition, to be able to let e.g. an OGC based OpenMI extension derive its geometrical spatial definitions from this base interface.

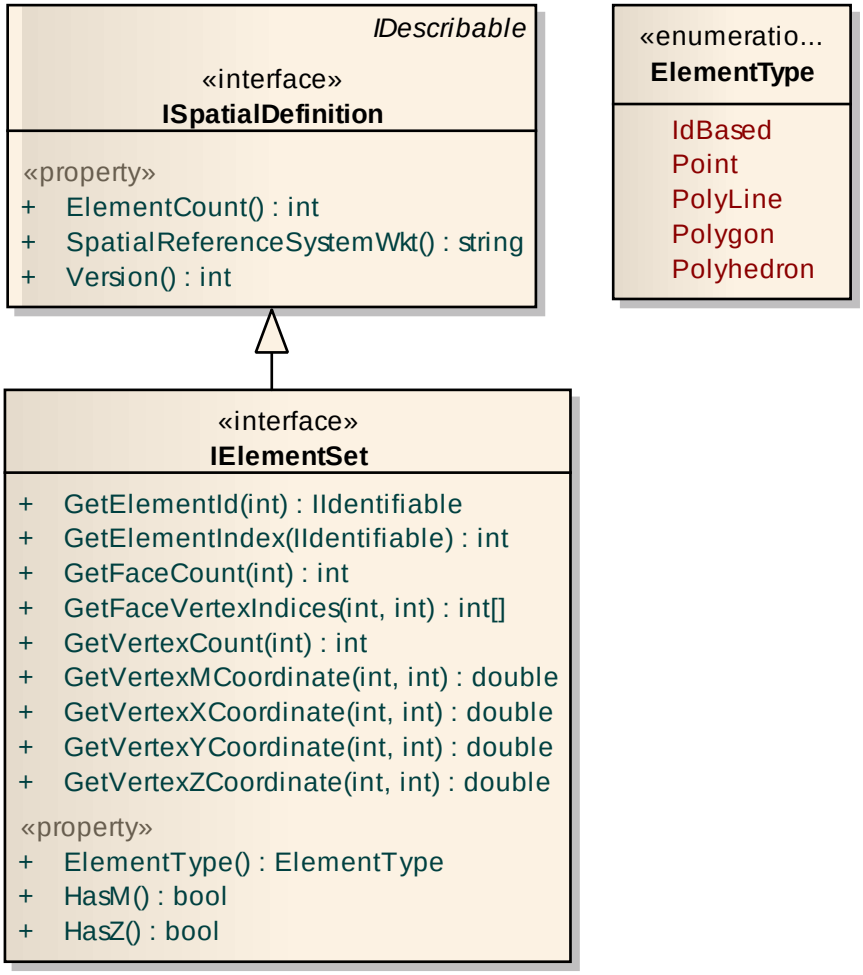


Figure 9 IElementSet and related interface

Conceptually, IElementSet is composed of an ordered list of elements having a common type. The geometry of each element can be described by an ordered list of vertices. The shape of three-dimensional elements (i.e. volumes or polyhedrons) can be queried by face. If the element set is geo-referenced (i.e. the SpatialReferenceSystemWkt is not an empty string), co-ordinates can be obtained for each vertex of an element.

The ElementType is an enumeration, listed in Table 2. Data not related to spatial representation can be described by composing an element set containing one (or more) ID-based elements, without any geo-reference.

If the HasZ property is True, the coordinates of the element vertices have X, Y and Z components. If it is False, the element set is defined in the horizontal (X, Y) space; i.e. it has no Z-coordinate.

If the HasM property is True, the element set supports M co-ordinates.

Note that IElementSet can be used to query the geometric description of a model schematization, but an implementation does not necessarily provide all topological knowledge on inter-element connections.

The Spatial Reference System Well Known Text (SpatialReferenceSystemWkt) is a string that speficies the OGC Well-Known Text representation of the spatial reference to be used in association with the co-ordinates in the ElementSet (SpatialReference.org, 2009)

The elements in an element set are identified by a string ID, and therefore are IIdentifiables. Of course you should let the element ID be useful to an end user. The element set needs not to be identifiable, because, from an OpenMI point of view, it is always part of an identifiable input or output exchange item. However, it is an IDescribable, meaning it has a caption and a description, thus supporting the end user in composing configurations.

The properties of an element (its vertices and/or faces) are obtained using an integer index (elementIndex, faceIndex and vertexIndex). This functionality has been introduced because an element set is basically an ordered list of elements, an element may have faces and an element (or a face) is an ordered list of vertices. The integer index indicates the location of the element/vertex in the array list.

Table 2 OpenMI enumeration of ElementType

ElementType	Convention
IDBased	ID-based (string comparison).
Point	geo-referenced point in the horizontal (XY)-plane or in in the 3-dimensional (XYZ)-space.
PolyLine	geo-referenced polyline connecting at least two vertices in the horizontal (XY)-plane or in in the 3-dimensional (XYZ)-space. The begin- and end-vertex indicate the direction of any fluxes. Open entity with begin- and end-vertex not being identical.
Polygon	geo-referenced polygons in the horizontal (XY)-plane or in the 3-dimensional (XYZ)-space. Vertices defined anti-clockwise. Closed entity with one face, begin- and end-vertex being identical.

ITime

Time in the OpenMI is defined by the ITimeSet interface (see Figure 10). A time set contains a list of times, where time is specified by the ITime interface, containing a Modified Julian Day value and a duration. If the duration is zero, the time is a time stamp. If it is greater then zero it is a time span.

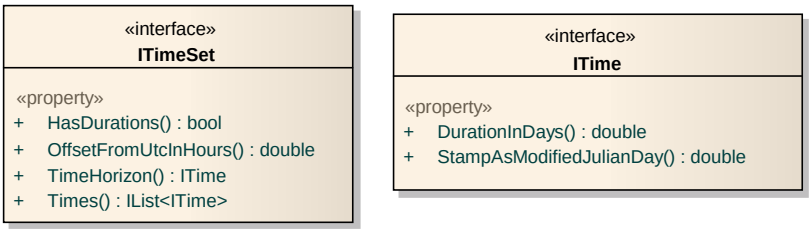


Figure 10 ITime and related interface

A Modified Julian day is the Julian Day minus 2400000.5. A Modified Julian Day represents the number of days since midnight November 17, 1858 Universal Time on the Julian Calendar[11]. The Modified Julian Day has been selected as a reference, since few models operate in a time horizon before 1858. Any date before November 17, 1858 will be represented as a negative value.

The ITimeSet interface contains additional information about the time stamps or spans that it contains or will contain. The OffsetFromUtcInHours property indicates the time zone in terms of an offset from UTC time (effectively the same as GMT, Greenwich Mean Time).

The value of HasDurations specifies whether the time sets' times are time stamps (in case of False) or time spans (in case of True).

The TimeHorizon property provides information on the timeframe during which an exchange item will interact with other exchange items. For an input item, the property specifies for what time span the input item can be expected to request values during the computation. This means that the providers of this input can assume that the input item never goes back further in time than the time horizons begin time, StampAsModifiedJulianDay. Also, it will never go further ahead than the time horizons end time, StampAsModifiedJulianDay + DurationInDays.

For an output item, and also for an adapted output item, the time horizon indicates the time span in which the output will be able to provide values.

To indicate that an input item may ask for values far back in time, or that an output item can provide values as far back in time as requested, the begin time of the time horizon should be set to infinitely back in time, i.e. the StampAsModifiedJulianDay should be set to the 'negative infinity value' of a double precision number. Comparably, if an input item may request values far in future, or if an output item can provide values far in the future, the end time of the time horizon should be set to infinitely far ahead in time, i.e. the DurationInDays should be set to the 'positive infinity value' of a double precision number.

IBaseValueSet and ITimeSpaceValueSet

To enable massive data exchange over a link, an interface structure has been defined that supports the exchange of multi-dimensional data sets (see Figure 11). The base interface for this, the (IBaseValueSet) represents an ordered N-dimensional list of values.

The ValueType property indicates the type of objects stored in the value set. This type must be the same as defined by the related exchange item's ValueDefinition.ValueType.

The NumberOfIndices property represents the number of dimensions in the value set.

Please note that the value set is not a matrix of values, but a list of lists (of lists, etc.). This means that every list can have its own number of values. To determine the number of values for a list that is specified by the i-th index, the GetIndexCount() method can be called with actual index-values for the indices up till i – 1.

To access individual values in the value set, the GetValue() and SetValue() methods can be called, with actual index-values for each index. This means that the length and the ordering of the array of index values passed as an argument must be according to the N-dimensional list of values.

Time and space dependent computational cores often compute per timestep. Therefore the time and space extension OpenMI.Standard2.TimeSpace contains a more specialized version of the IBaseValueSet, the ITimeSpaceValueSet. This interface represents an ordered two-dimensional list of values. The first dimension stands for the times for which values are available, whereas in the second dimension each value belongs to precisely one element in the corresponding element set (which was

[11] See http://tycho.usno.navy.mil/systime.html.

specified when asking for the values). In other words, the i-th value in that dimension of the value set corresponds to the i-th element in the element set.

Various methods are defined to access the values for a certain time and/or element (Set/Get-Value), for all times at once (Set/Get-TimeSeriesValuesForElement) or for all elements at once (Set/Get-ElementValuesForTime).

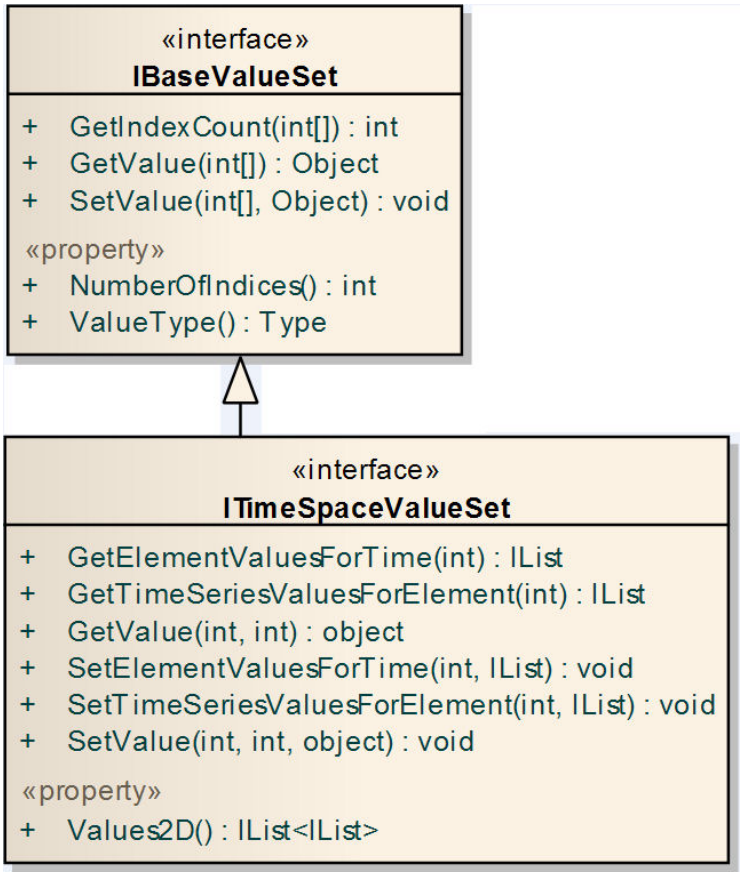


Figure 11 IBaseValueSet and ITimeSpaceValueSet interfaces

To prevent misunderstanding of positive and negative values, the following conventions are applied:

- Values are positive if the matter leaves the source component and enters the target component (this also is the case for volumes).
- The 'right-hand rule' applies for fluxes through a plane or polygon[12].
- The direction of fluxes along a polyline is defined as positive from the begin node to the end node.
- The right-hand rule applies for fluxes perpendicular to a polyline[13].

[12] Curl your right hand in the vertex order of the plane or polygon. The thumb points in the positive direction

[13] Put your hand along the line in the positive direction, turn your wrist clockwise. The thumb will point in the positive direction perpendicular to the (poly)line.

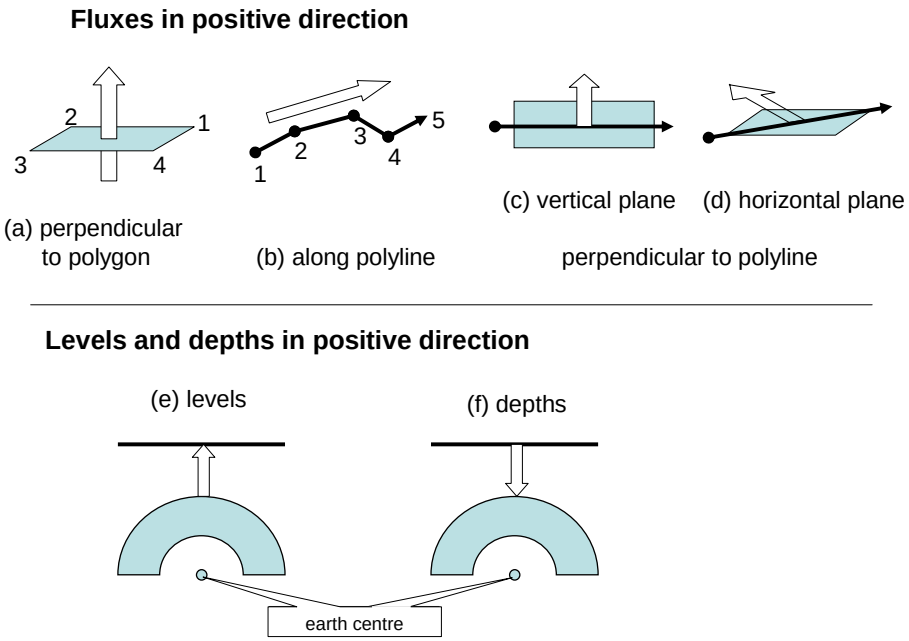


Figure 12 Illustration of directions to interpret positive values of fluxes, levels and depths

Software developers that do not comply with those conventions should make software users aware of this risk.

3.2.3 Specifying which data will be exchanged

The data to be exchanged is determined by the provider-consumer relationship, any adapted output and the supplied arguments.

Provider-consumer relationship

To specify which data will actually be exchanged during the computation, provider-consumer relationships are established between an output item and one or more input items. Figure 13 shows the methods and properties that are available for this purpose.

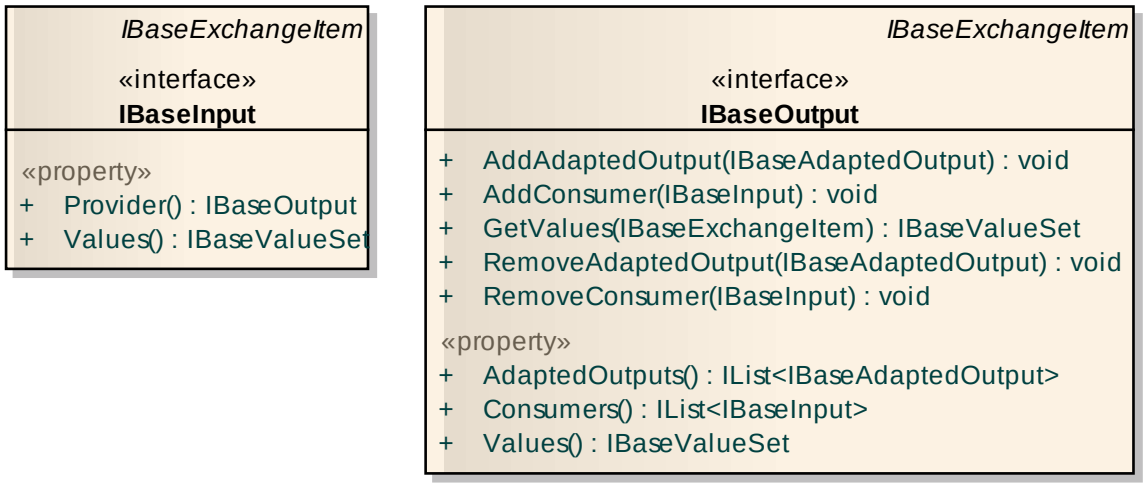


Figure 13 IBaseInput and IBaseOutput interfaces, used for provider/consumer connections

An input is connected to an output by calling the output item's AddConsumer() method. This method will take the internal actions needed to ensure that values can be provided once the computation starts, and will add the input it to the Consumers list. At the same time, this method sets the output item as the Provider of the input item.

If a connection is not needed anymore, the input is removed as a consumer by calling the output item's RemoveConsumer() method. This method may perform internal clean-up actions, and will remove the input from the Consumers list. At the same time, this method sets the Provider of the input item to Null.

Once the actual computation starts, the output items GetValues() method is invoked with a so-called query specification. The query specification argument will nearly always be a consumer of the output item.

There is a situation in which the query will be composed at run-time: when asking for values from an output item that represents an analytical function. In that case, an exchange item can be set up with the requested times in the time set and with the requested geometry in the element set. This exchange item is then passed to the GetValues() call as the query specification.

IBaseAdaptedOutput and ITimeSpaceAdaptedOutput

Many situations occur where the raw data available at the source component does not match the request of the target component. For instance, the units of a requested quantity might differ from the units that the quantity is provided in, or the discrete values of a quality may have to be translated into numerical quantity values. For time and/or space dependent components the location and time as produced by an output item may not match the specification given by the input item. Additional data operations may be required, varying from temporal averaging to spatial interpolation, etc. Especially when data is requested over a time span, it should be known how a single element value needs to be computed: by averaging, by accumulating, taking minimum or maximum, etc.

For these purposes, the IBaseAdaptedOutput interface and the ITimeSpaceAdaptedOutput (in OpenMI.Standard2.TimeSpace) have been defined (see Figure 14).

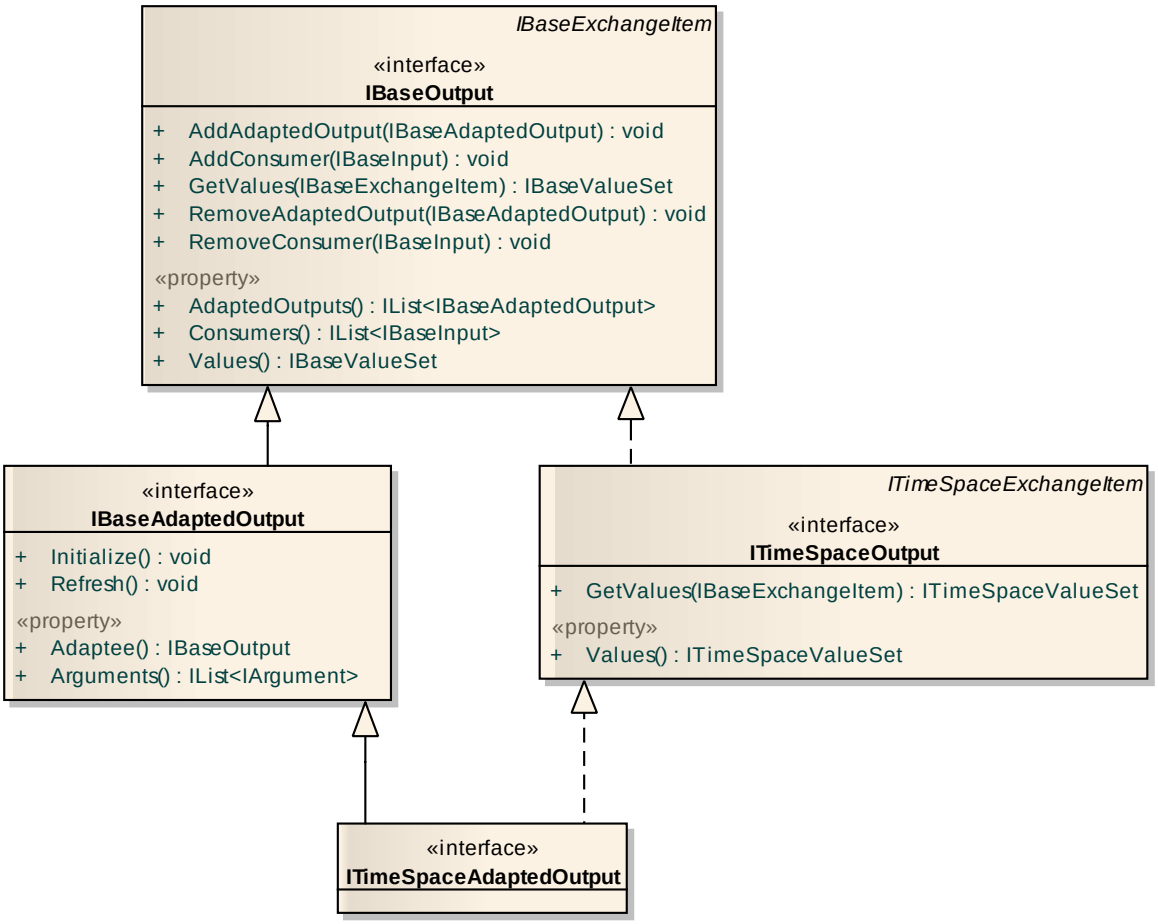


Figure 14 The IBaseAdaptedOutput and ITimeSpaceAdaptedOutput interface

Each adapted output may have a number of Arguments to manipulate the behaviour of the adapted output. Each argument is specified by means of the IArgument interface, a key-value pair (see below). During configuration time the arguments are specified. Before the actual computation starts, during the prepare phase, a linkable component calls the Initialize() method of all its adapted outputs. In case of stacked adapted outputs, the adaptee must be initialized first.The Adaptee is the output item from which the adapted output takes the source values that it needs to be able to perform its action.

If values of the adaptee have changed or may have been changed, the adapted output needs to take action. To enable this, IBaseAdaptedOutput contains a Refresh() method, that must be called each time the adaptee has been changed. The linkable component that owns an output item has the responsibility to call the Refesh() mehod of all the output item's AdaptedOutputs.

The ITimeSpaceAdaptedOutput makes the GetValues() method and the Values property return an ITimeSpaceValueSet instead of a IBaseValueSet.

Please note that the IBaseAdaptedOutput itself is an IBaseOutput, and that as a consequence it therefore can have adapted outputs itself. So when it has finished its own refresh action, it must call the Refesh() mehod of all its adapted outputs.

Another consequence of the fact that the adapted output is an output itself is that it also needs to provide its ValueDefinition, its ElementSet and its TimeSet. The value definition can usually be taken from the adaptee, unless the adapted output is for instance a ‘quantity to quality translater’, in which case it exposes the quality and the categories where the values are converted to. For adapted outputs

that perform spatial operations, the time set can be taken from the adaptee, while adapted outputs that perform temporal operations can take the adaptee's element set.

For adapted outputs that perform a temporal operation, it is less clear what the content of its time set should be. After all, it is the querying input that specifies the required time set. Therefore either a time set with zero times should be provided (representing 'time-dependent item but no values available yet', like an output item), or the times of the adapted should be provided. The OpenMI compliancy information for the component should describe which approach has been chosen.

Similarly, for adapted outputs that perform a spatial operation, it is not clear what its element set should look like. Such an adapted output converts values to the required elements specified by the query. If the adapted output modifies the element type, an empty element set of the target type should be provided. If it does not modify the element type, either an empty element set or the adaptee's element set may be provided. Once again the OpenMI compliancy information for the component should describe which approach has been chosen.

Adapted output operations may cover various aspects. For example:

- Temporal data operations
  e.g. for time stamps: interpolation and extrapolation (linear, quadratic, by regression function) for time spans: averaging, aggregate, accumulation, moving average, minimum value, maximum value, first value, last value, all values.

- Spatial data operations
  e.g. interpolations: kriging, inverse distance, all kinds of averaging, maximum value, minimum value.

- Miscellaneous data operations
  e.g. perform vertical shift.

To create adapted outputs, the IAdaptedOutputFactory has been defined (see Figure 15). First the factory can be asked what types of adapted outputs are available, given a certain output item and a target input item. This is done by calling GetAvailableAdaptedOutputIdentifiers(). This methord returns a list of identiers for the available types.

Once the user has chosen which type of adapter will be used, an instance of this adapted output is created by calling the CreateAdaptedOutput() method.

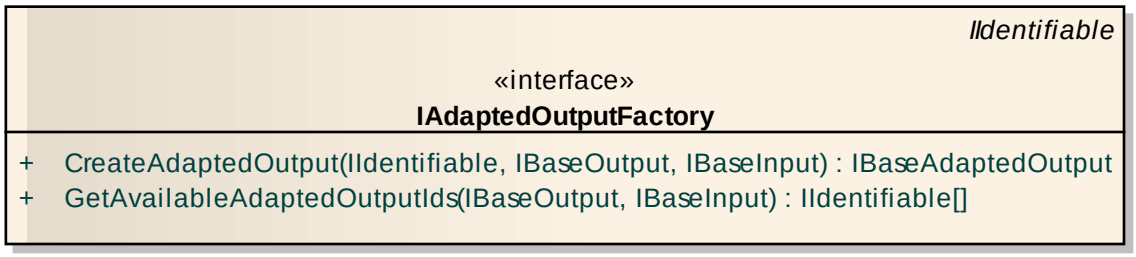


Figure 15 The IAdaptedOutputFactory interface

IArgument

Both the adapted output and the linkable component contain arguments that are used to provide information to let the adapted output do its work. This is done by means of the IArgument interface (see Figure 16).

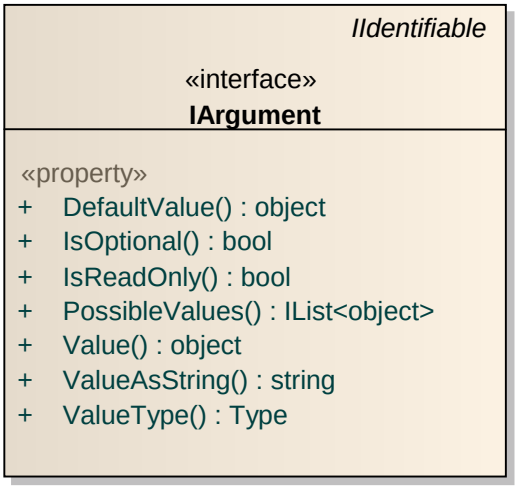


Figure 16 The IArgument interface

The ValueType property describes the type of the value of the argument, e.g. an integral type like string, integer or double precision number, or a non-integral type, such as a time series object.

Value contains the current value of the argument. If no value has been set yet, a default value is provided. So if the value property is Null, this actually means that the default value is Null.

The DefaultValue specifies the value to be used by default. This value can be used to reset the value of the argument, when it has been changed and the changes have to be undone.

IsOptional specifies whether the argument is optional or not. If the Value property returns null and IsOptional is False, a value has to be provided.

The IsReadOnly property defines whether the Values property may be edited. This is used to let a linkable component or an adapted output present the actual value of an argument that cannot be changed by the user, but is needed to determine the values of other arguments, or is informative in another way.

PossibleValues provides a list of possible allowed values for this argument. If for integral types or component-specific types all possible values are allowed, Null is returned. A list with length 0 indicates that there is a limitation on the possible values but that currently no values are possible. Effectively this means that the values will not and cannot be set.

ValueAsString contains the argument's value, represented as a string. If ValueType indicates that the argument's value is not of type string, the ValueAsString property offers the possibility to treat it as a string, e.g. to let the GUI persist the value in the composition file.

3.2.4 Interfaces for component access

Three interfaces provide generic access to a component (IBaseLinkableComponent), create feedback loops (IManageState) and provide the ability to restart from a certain state (IByteStateConverter). The question of throwing exceptions must also be considered.

IBaseLinkableComponent

OpenMI components need to have one interface that defines the generic access to the component. As linking components is the most important functionality of the OpenMI, the IBaseLinkableComponent

interface, which is the interface for enabling this linkage, is the entry point of an OpenMI component and thus the key interface of the OpenMI. Figure 17 displays the interface. The various methods are described below, while Section 3.3 provides additional insight into the call sequence to be adopted.

The current IBaseLinkableComponent is with respect to the scope a subset of the OpenMI 1.4 ILinkableComponent interface, which supported time and space dependent components implicitly. In order to get their functionality, an OpenMI 2.0 compliant linkable component would have to implement the IBaseLinkableComponent and the interfaces of the OpenMI.Standard2.TimeSpace extension.

Functionality to initialize the computational core:

- Arguments
  The linkable component provides a list of arguments that should be set before the component is initialized. These arguments, which are of the type IArgument, are key-value pairs. An argument may be optional and it may have a default value that will be used if it value has not been specified explicitly. See Section 3.2.3 for a detailed description of IArgument.

- Initialize()
  After instantiation of the LinkableComponent object by its constructor, and after the arguments have been specified, the Initialize() method is called to populate the object with specific information. After this method call, the LinkableComponent can be inspected for its inputs and outputs. At this stage, code developers may choose (but are not forced) to instantiate the computational core or database and populate it with site-specific data. The Initialize() method allows instantiation of a computational core without having specific knowledge of the component.

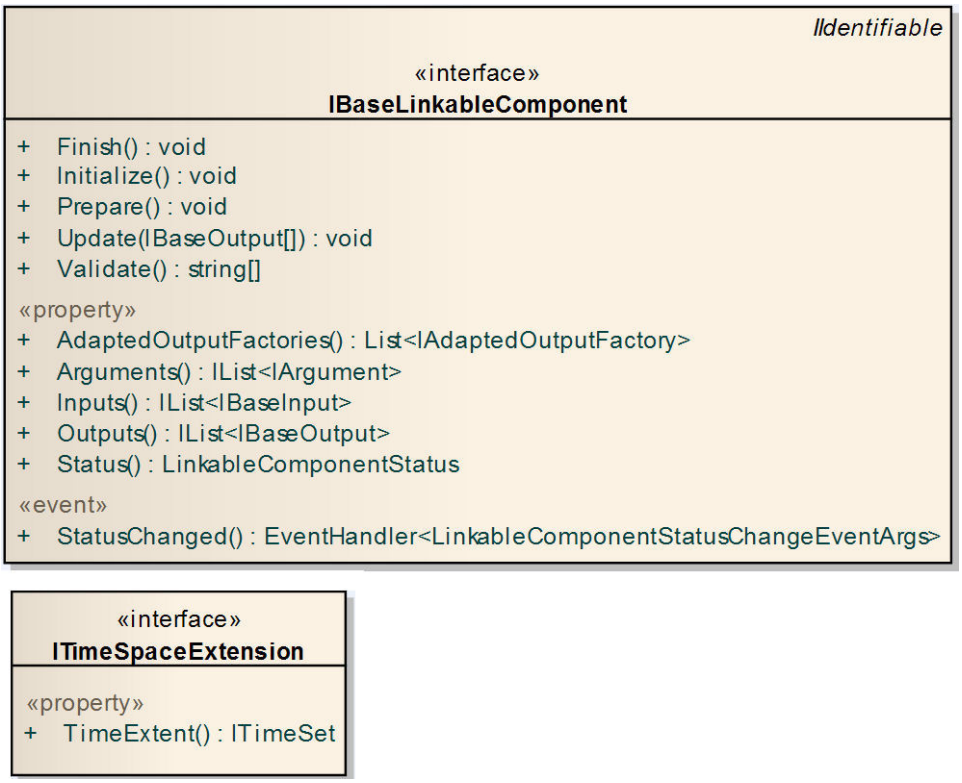


Figure 17 IBaseLinkableComponent interface

Functionality to accommodate inspection of the component, its content and exchangeable data:

- Caption
  Logical naming in terms of model/schematization/application area of the component instance.

- Description
  Additional description of the model/schematization/application area.

- Id
  A string that has to be unique within its context, e.g. within the collection of instances of available linkable components

- Inputs
  The data sets that can be accepted as input.

- Outputs
  The data sets that can be provided as output.

ITimeExtension

Optionally, a linkable component may implement the time extension, thus indicating that it is a model that is aware of time. The extension contains only one propery.

- TimeExtent
  The time span over which a component can provide data (either discrete or continuous).

Functionality to create and validate the output/input connections:

- Inputs[m] and Outputs[n]
  The individual exchange items are accessed to establish the provider-consumer relationship, as described in Section 3.2.3.

- AdaptedOutputFactories
  This property is accessed by the GUI at configuration time, when the values that are provided by an output do not directly fit the way the input needs it, so that an adapted output is required. The list of returned factories will be queried for applicable adapted output types, as explained in the IAdaptedOutputFactory interface in Section 3.2.3.

- Validate()
  This method is typically called by the GUI at configuration time when output/input connections have been added. This call enables validation of the current state of the model and its connections, i.e. a check whether all input data is available, either a-priori or at run-time, and consistent (as far as the a-priori data is concerned). The method returns an empty string if the component is in a valid state; otherwise it returns a message describing the problem. Displaying the message in the User Interface enables the user to correct the error (e.g. inconsistent quantities connected).

Run-time section – preparation, computation/data transfer/retrieval, completion:

- Prepare()
  This method is invoked just before the first GetValues() call. It enables the component to prepare itself: e.g. instantiate the engine (if needed), set up the network connections, do a model validation, prepare internal buffers and data mapping matrices. If something goes wrong, an exception will be thrown, as opposed to Validate() where a message is returned allowing the user to correct the error.

- Update()
  This function lets the linkable component progress to its next state. For a time-stepping component this would typically perform one timestep; for a time-independent component this

would typically re-evalute its state. Both types will, as a first action in the update call, let all their connected input items get the latest values from their providers and the output items of other components.
The Update() call can be repeated until the component has reached the status Done.

- Outputs[m].GetValues() / Outputs[m].Values
  During the computation, the GetValues() call will be performed on one of the component's output items or on one of the adapted outputs of such an output item.

- Finish()
  The Finish method is intended to be used for closing files. This means that a GUI can, for example, inspect some results before the components are disposed.

Status and events:

- Status
  While going through the above methods' phases, the component will switch between various states. The possible states are formalized in a LinkableComponentStatus enumeration. Section 3.3.1 describes these states and the usage and benefits of the Status property.

- Events
  When going from one status to another, the component has to fire a LinkableComponentStatusChange event. Also, if during computation the content of an exchange item changes, the component has to fire an ExchangeItemChange event. Section 3.2.5 describes these events.

IManageState

Some process interactions require feedback loops and even iterations over time. To support this type of functionality, the IManageState interface (Figure 18) has been defined, containing three methods. By calling the KeepCurrentState() method, a component is requested to store its internal state and return a string identifier that enables future restoring. The state data itself is not of interest to the OpenMI, so the component developer can decide what needs to be stored and how (e.g. in memory or on disk in a file or database). The only item to be returned is the key to identify the state, which will be used as the argument for the RestoreState() method (or the ClearState() method).

Note that implementing the business logic of this interface is not obligatory to provide an OpenMI-compliant linkable component. However, you should be able to throw an exception if the logic is not implemented.

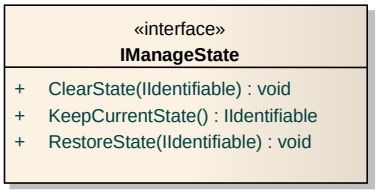


Figure 18 IManageState interface

IByteStateConverter

In operational systems it may be required to be able to restart from a certain state that was stored some time earlier. This requires the facility to make a linkable component's state persistent. One of

the ways to do this would be to let the component store the states and let it keep track of its own persistent states. However, this would lead to a management problem: how does a component know that a state will never be used again? In general, it will be the overall system that keeps track of all availabe states. That is why the IByteStateConverter interface has been designed.

Given a state identifier as retrieved from the KeepCurrentState() method in the IManageState interface, this system can pass the state identifier to the IByteStateConverter's ConvertToByteArray() method, which will transform the state into a byte stream. The system can, for instance, write this byte stream to file. By providing the byte stream to the ConvertFromByteArray() method, a handle to the original state is retrieved back.

The most common way to implement this interface is to let a component access its engine's restart file as a stream of bytes.

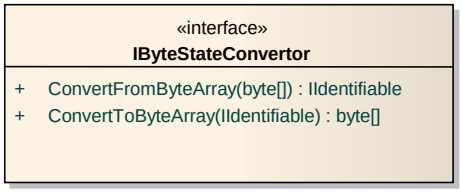


Figure 19 IByteStateConverter interface

The IByteStateConverter interface is optional. If a component does not implement it, it should throw an exception if one of the methods is called.

Exceptions

By convention a linkable component has to throw an exception if an internally irrecoverable error occurs. Before throwing an exception, the Linkable Component must set its Status to *Failed*.

The exception should be based on the Exception class, as provided by the development environment. The exception might be caught by a deployer or user interface, which finally handles the exception properly (if required, with user interference).

3.2.5 Events

When a component changes from one state to another, it should fire a LinkableComponentChanged event. Figure 20 shows the arguments that are passed with this event:

- LinkableComponent
  The component that fired the event.

- OldStatus
  The state that the component has left.

- NewStatus
  The state that the component has entered.

- Message
  If useful, on additional message on the status change.

When the component starts updating, it usually updates all the values of its connected inputs as its first action. When such an input item has been updated, it should fire an ItemChanged event for each

updated input. When the component is at the end of the updating process, right before it enters the *Updated* state, it usually has updated its connected output items, so each updated output should also fire an ItemChanged event. Figure 20 shows the arguments for this event:

- ExchangeItem
  The exchange item (input or output) that fired the event.

- Message
  If useful, on additional message on the content change.

Figure 20 Linkable component statuses and events

3.2.6 Where to start the component access: the OMI-file

All interfaces of the OpenMI.Standard have been discussed above. OpenMI-compliant components need to implement those interfaces. However, once implemented, you still cannot get started as long as the software unit has not been identified properly. Therefore, the information on the assembly and class to be instantiated has been combined in one registration file, called the OMI-file, which can be located anywhere on disk. This file also holds the arguments to populate the component at the initialization phase. In addition to its interfaces, the OpenMI Standard therefore also defines an Xml Schema Definition for the OMI-file. Figure 21 provides a graphical view of the file structure according

to the Xml Schema Definition. Figure 22 provides an example XML-file while Appendix 1B contains the XSD.

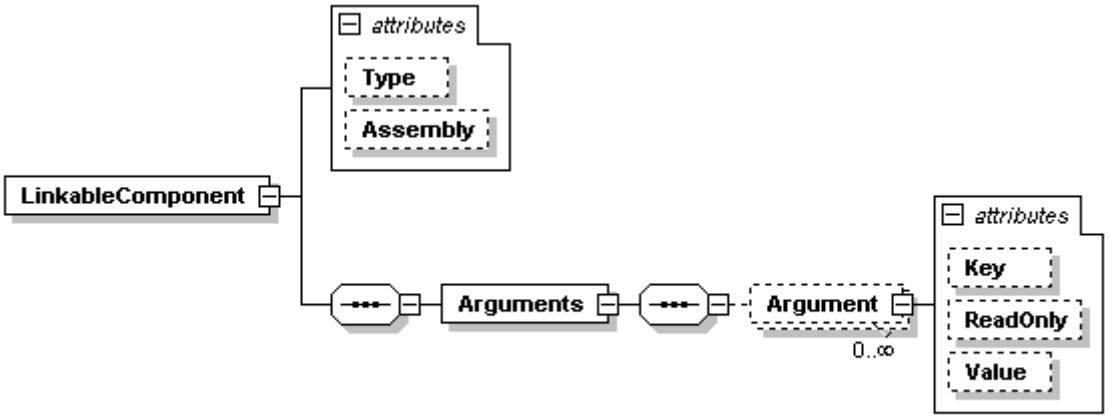


Figure 21 Graphical view of the OMI-file structure

```
<?xml version="1.0"?>

<LinkableComponent  Type="wlDelft.OpenMI.WLLinkableComponent"  Assembly="wlDelft.OpenMI,  Version=1.2.0.0,
Culture=neutral, PublicKeyToken=8384b9b46466c568" xmlns="http://www.openmi.org/LinkableComponent.xsd">

   <Arguments>

      <Argument Key="Model" ReadOnly="true" Value="RR" />

      <Argument Key="Schematization" ReadOnly="true" Value="D:\Rain-RR-CF\Model\Cmtwork\sobek_3b.fnm" />

   </Arguments>

</LinkableComponent>
```

Figure 22 Illustrative example of the OMI-file content

3.3 org.OpenMI.Standard: dynamic view

This section provides a formal specification of the interfaces in their dynamic view.

3.3.1 Phases in utilizing the linkable component interface

An OpenMI linkable component provides a variety of services that can be used in various phases of deployment. Figure 23 provides an overview of the phases that can be identified, and the methods that might be (logically) invoked at each phase. While the sequence of phases is prescribed, the sequence of calls within each phase is not prescribed.

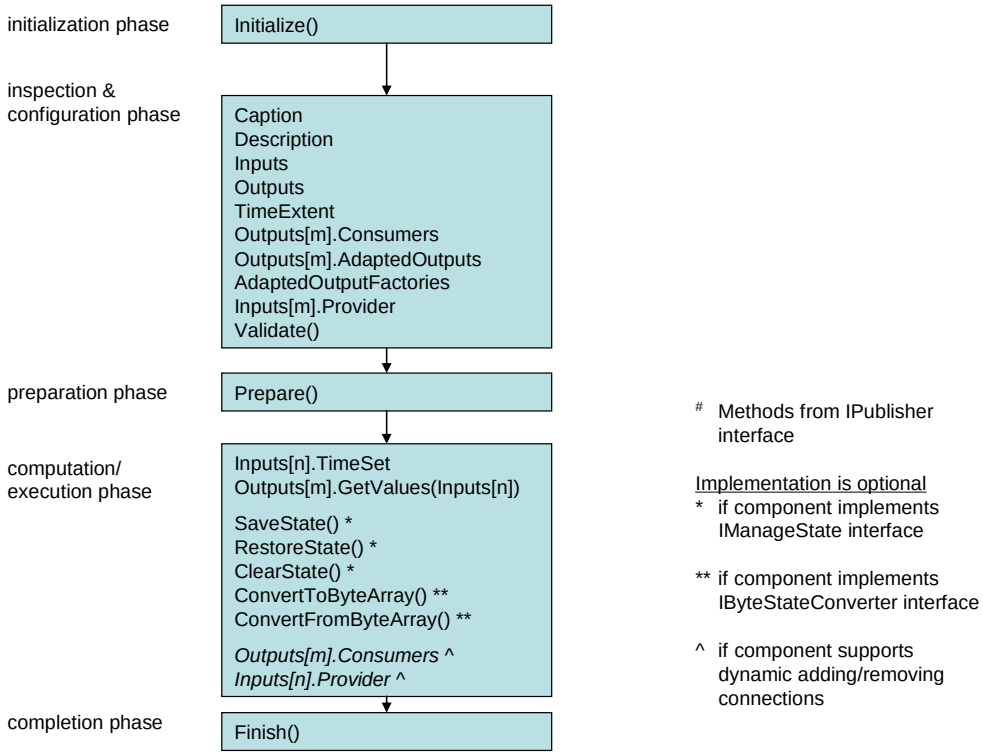


Figure 23 Deployment phases of OpenMI linkable cComponents

While going through the various phases illustrated in Figure 23 the Status property of the IBaseLinkableComponent specifies the state that a linkable component currently is in. The state diagram in Figure 24 shows the possible states and their transitions. The named arrows represent the methods that can be called. They will let the component take action (hence the ***…ing*** names of the entered states); once the component has completed the action, it will enter one of the ***…ed*** states.

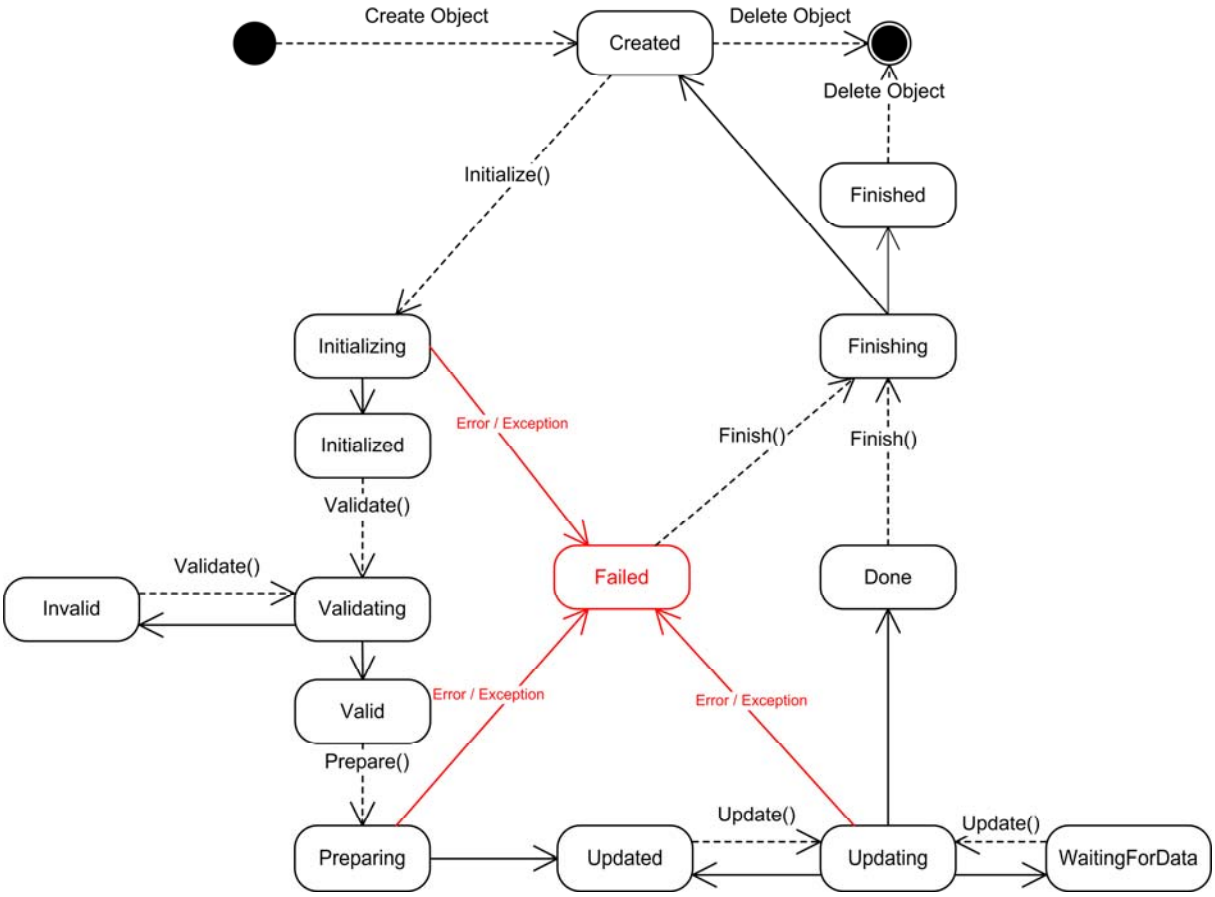


Figure 24 State diagram for IBaseLinkableComponent.Status

The left side of the diagram shows the possible states during the initialization, inspection and configurations phases, the bottom part the run-time states, and the right side the states of the completion phases.

The various phases and the linkable component states in those phases are discussed briefly in the next sections. The dynamic behaviour of the computation phase is discussed in detail with a use case involving three linkable components, namely a rainfall-runoff mode, a river model and a groundwater model.

The chapter concludes with a few sections on event and exception handling, the interruption of the computation process and other dynamic behaviour issues.

3.3.2 Phase I: Instantiation and initialization

***States**: Initializing/Initialized, Validating/Invalid/Valid*

This phase ends when a linkable component has sufficient knowledge to populate itself with model data and expose its exchange items. Whether the linkable component has been populated with model data depends on the solution chosen by the code developer.

The phase is composed of two steps:

1. Instantiation
   A LinkableComponent is constructed using the software unit which has been referred to in the OMI-file

2. Initialization
   The LinkableComponent can be populated with input data by calling the Initialize() method with the arguments as listed in the OMI-file. The arguments typically should contain references to data files. In situations where the initialization is not completed successfully, an exception should be thrown with sufficient information to solve the problem.

3.3.3 Phase II: Inspection and Configuration

Depending on the setting, this phase might be very static and straightforward or very dynamic. At the end of this phase the provider-consumer connections between outputs and inputs have been defined and the component has validated its status[14].

The following steps can be identified:

3. Request for the exchange items
   Ask the LinkableComponent for its outputs items and input items. Ask the LinkableComponent for its adapted output factories.

4. Establish provider-consumer connections between outputs and inputs, by using the IBaseOutput.AddConsumer() call and setting IBaseInput.Provider.
   If required, create adapted output using the adapted output factories, and establish the relations by the AddAdaptedOutput() call on the adaptee and the AddConsumer() call on the adapted output and by setting the input item's Provider to the adapted output.

5. Validation
   Validate the status of the components and their links using the IBaseLinkableComponent.Validate method.

Hard-coded systems do not require step 3. All systems require step 4, while step 5 is strongly recommended. Systems with linkable components with a-priori knowledge of exchange items can easily respond to step 3. Systems where the exchange items depend on connected components will require a dynamic querying process to reply with proper information.

3.3.4 Phase III: Preparation

***States**: Preparing/Updated*

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

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

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

[14] Note that any model combination is not persistent unless tools are used to save the configuration. The org.OpenMI.Utilities.Configuration package provides this type of facilities.

3.3.5 Phase IV: Computation/execution (including data transfer)

***States**: Updating/WaitingForData/Updated*

During this phase, the heavy workload is executed and associated data transfer gets bulky.

The data transfer mechanism of the OpenMI is defined as a request-reply service mechanism, having direct interaction between the output and input items of linkable components, without any involvement of external facilities. Two types of data transfer are distinguished: unidirectional data transfer and bidirectional data transfer.

In addition, the call sequence of advanced linkages based on state management (e.g. iteration) is also discussed.

Unidirectional data transfer (one way)

Figure 25 illustrates the calling sequence between two linkable components in the case of unidirectional data transfer. The data transfer is illustrated by the link between a Rainfall-Runoff model and a RiverModel, as this type of link typically is unidirectional (from RR model to RiverModel).

In the diagram, it is assumed that the RR model does not yet have the requested data available but has sufficient information to calculate the runoff on request. Note that the diagram peers into the private handling of the GetValues() call, typically by looping over its owntime step until the requested time has been reached.

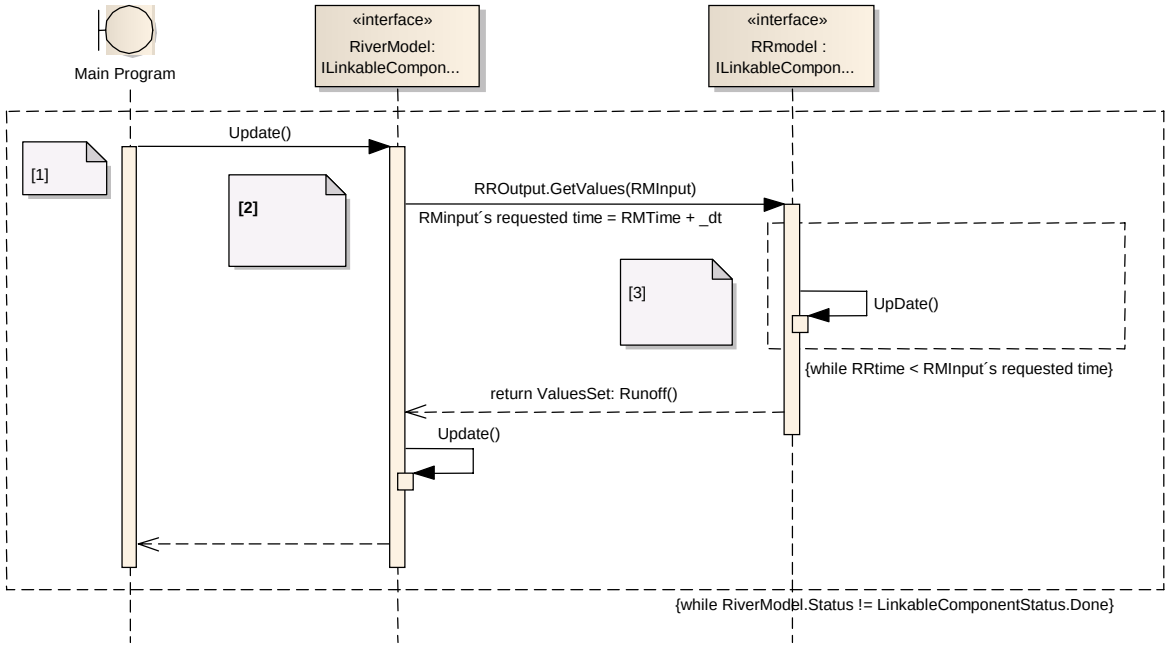


Figure 25 Unidirectional data transfer (sequence diagram)

Annotation: The fact, that the RiverModel and the RRModel in the figure above both implement ILinkableComponent is a simplification. Actually they implement IBaseLinkableComponent and ITimeSpaceComponentExtension.

6. Invocation
   The computation is repeatedly invoked by calling the Update() method on the last component in the chain. The Update() call will be repeated until the RiverModel has reached the status *Done*.

7. Computation – RiverModel
   The RiverModel component performs one timestep during the Update() call. As a first action in that timestep, the RiverModel will retrieve the runoff from the RR model by invoking the GetValues() method in the RR model. The argument to the GetValues() method is the RiverModel's input item for the incoming runoff. Before performing the call, the time set of the input item is set to one time stamp, specifying the requested time for which the RiverModel needs the runoff (which is its current time plus its timestep size).

8. Computation – RR model
   The RR model will perform as many timesteps as necessary in order to calculate the requested value. This is done by invoking the Update() method as often as needed to reach the requested time.
   Note that the two models do not need to have matching timesteps. It is the responsibility of the delivering model to do any interpolation (or extrapolation) required in order to return a value that represents the requested time in the GetValues() call's query specification.

Bidirectional data transfer (two way)

Figure 26 illustrates the calling sequence between two linkable components in the case of bidirectional data transfer. The data transfer is illustrated by the link between a RiverModel and a GroundWaterModel, with the RiverModel providing a surface water level to the GroundWaterModel, and the GroundWaterModel providing a lateral inflow to the RiverModel. The two components have been instantiated and prepared as described in Sections 3.3.3 and 3.3.4. The fact that both implement ILinkableComponent is a simplification. Actually they implement IBaseLinkableComponent and ITimeSpaceComponentExtension.

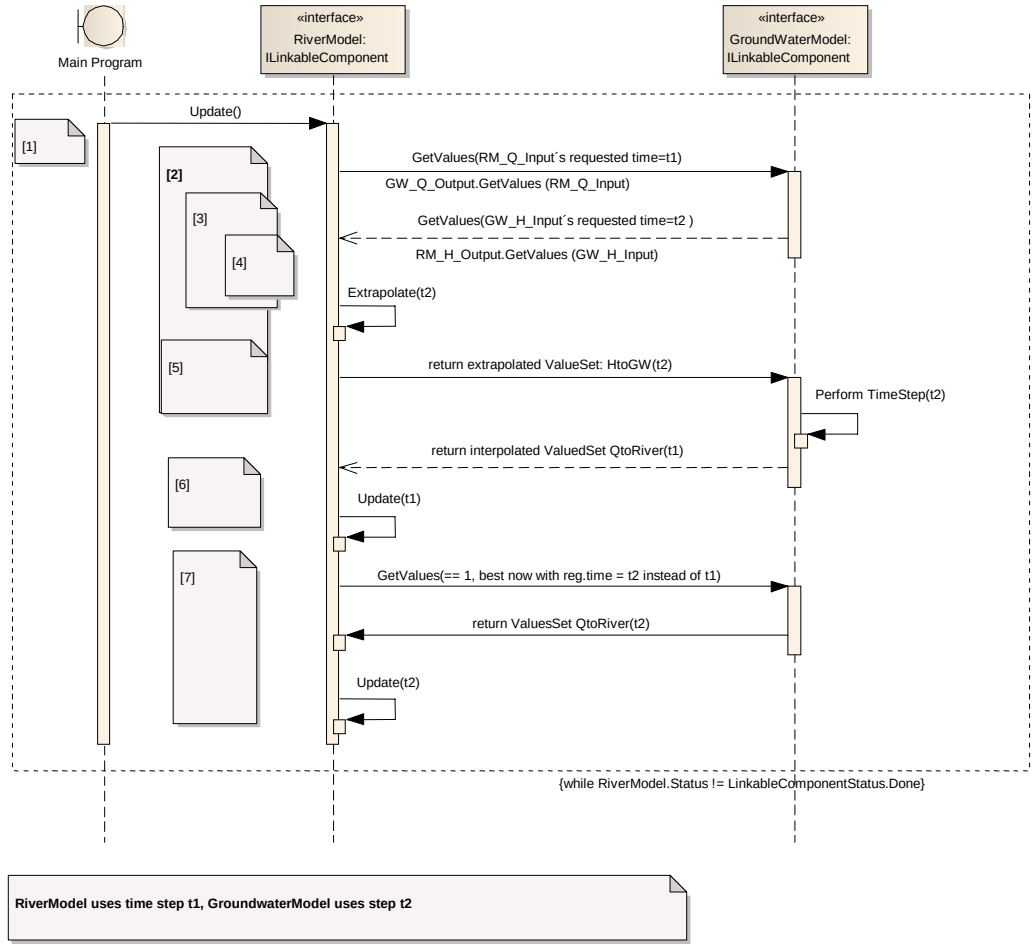


Figure 26 Bidirectional data transfer (sequence diagram)

In the diagram, it is assumed that both the RiverModel and GroundWaterModel do not yet have the requested data available but have sufficient information to compute the data requested. For illustration purposes, the timestep of the GroundWaterModel has been set to twice the timestep of the RiverModel. The iteration of the RiverModel (two timesteps) has been written out.

6. Invocation
   The computation is repeatedly invoked by calling the Update() method on the last component in the chain, until the RiverModel has reached the status *Done*. The RiverModel component performs one timestep during this Update() call. The figure shows two of these subsequent Update() calls.

7. Computation – RiverModel
   Before performing the timestep, the RiverModel retrieves the lateral inflow from the GroundWaterModel by invoking the GetValues() method in the GroundWaterModel (see the unidirectional data transfers section above for the explanation of how the requested time is set).

8. Computation – GroundWaterModel
   The GroundWaterModel, currently at t0, is only able to compute a lateral flow at t2. A request for t1 requires interpolation. To compute a lateral flow at t2 it requires the water level in the river at t2. For this purpose, the RiverModel is invoked by the GetValues() method with time t2.

9. Extrapolation – RiverModel
   After receiving the GetValues() call for t2, the RiverModel determines from its Status (WaitingForData) that it already is in a GetValues() call (for t1). A new computation process thus cannot be started. The deadlock between the two components, waiting for each other, needs to be broken by returning the best guess of a water level.

10. Continue computation – groundwater level
    Using the returned data, the GroundWaterModel is able to compute the lateral flow for t2. Based on the outcome it can return an interpolated value for t1.

11. Continue computation – river model, towards t1
    The RiverModel has received all requested data for t1 and can compute towards t1.

12. Continue computation – river model, towards t2
    The second iteration step of the RiverModel (t1 to t2), is invoked by the next Update()-call.
    The RiverModel again asks the GroundWaterModel for a lateral flow at t2. As this has been computed before (in step 5), the GroundWaterModel can directly return the result. The RiverModel can perform its timestep towards t2.
    Note that the software developer of the GroundWaterModel may choose an implementation that recalculates the lateral flow based on an updated and hopefully more accurate water level extrapolation for t2.

Managing states using IManageState

The IManageState interface has been introduced to accommodate the development of advanced controllers for iteration and optimization purposes. Figure 27 illustrates how the IManageState interface can be used to enable iteration between a GroundWaterModel and a RiverModel. Both the GroundWaterModel and the RiverModel implement the IManageState interface as well as the IBaseLinkableComponent and ITimeSpaceComponentExtension.

A separate linkable component has been introduced to supervise the iteration: the IterationController. This controller holds internal links to the GroundWaterModel and the RiverModel, while it hides the two models from the ‘outside’ world (i.e. the main program).

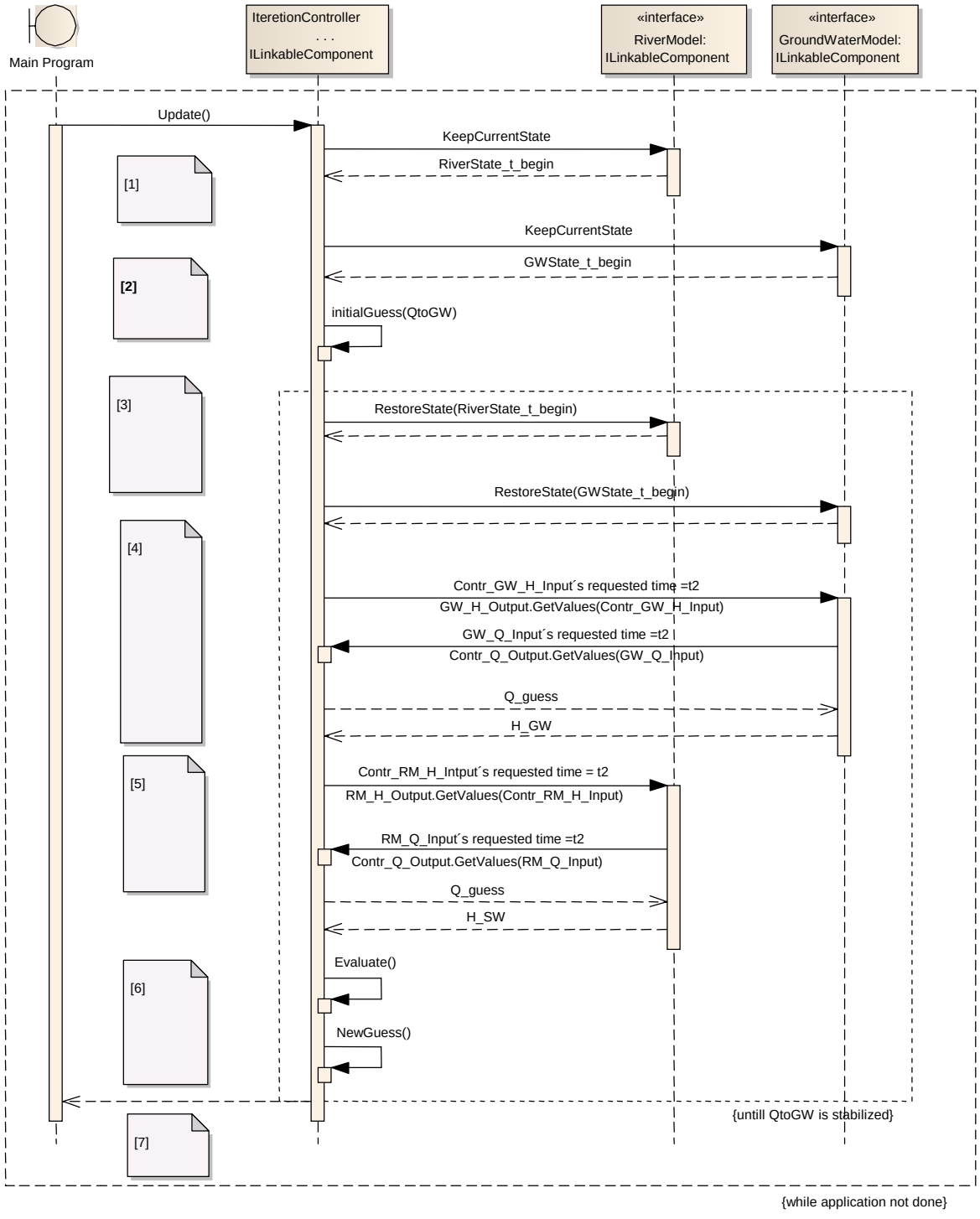


Figure 27 Illustration of how IManageState can be used for iterations (sequence diagram)

The procedure may be as follows:

1. Invocation
   The main program calls the IterationController to provide the surface water level. The IterationController starts its iteration by saving the state of the models at the beginning of the timestep.

2. Initial guess
   The IterationController then produces its first guess of the flux between the RiverModel and GroundWaterModel.

3. Restore states
   The iteration begins by restoring the state of the GroundWaterModel and the RiverModel to their original states.

4. Groundwater level requested
   The GroundWaterModel is asked for its groundwater level. To answer this question, the GroundWaterModel requires the flux from the IterationController. The guessed value of the controller is returned, after which the GroundWaterModel can compute and deliver its groundwater level.

5. Surface water level requested
   The RiverModel is asked for a better estimate of the surface water level. This estimate can be delivered after the RiverModel received an updated value for the flux (i.e. from the controller).

6. Update guess
   Based on the new information (surface and groundwater level), the IterationController can determine a new estimate for the flux and start another iteration (if the flux has not stabilized yet).

7. Once the flux has stabilized, the surface water level can be returned to the main program.

Managing internal buffers using the consumer's time set

Within a model combination involving several components, it may be that several components ask one component for data at the same time stamp. To prevent the need for re-computation for each request, such a source component will possibly have an internal buffer to store data for its connected components, even when they haven't asked for it yet. However, buffers get filled and need to be emptied. To prevent flushing data that is still needed, the time sets of the consumers can be used.

The time set of a connected input item (i.e. a consumer) contains a time horizon. The begin stamp of this time horizon indicates how far back in time the consumer may go when asking the provider for values. It is the task of the linkable component that owns the input item to ensure that this 'earliest needed input time' specification is up to date. Please note that to do this, the component may need to query the input items of other components, to which it delivers data, to check how far these consumers may go back in time. Of course there is no need to do this if the component itself in its turn buffers all its computed output, because in that case it will never ask for input further back in time than its current time. It is strongly recommended that the linkable component updates this time information right at the beginning of its Update() call.

It is up to the related provider to decide at which moment(s) it wants so check its consumers' earliest needed input times. The most logical moment to do this, however, is right before handling the GetValues() call, because it can assume that the component that invokes the GetValues() call is at the start of a new Update() step, and therefore has just updated its input time specifications.

3.3.6 Phase V: Completion

***States**: Done/Finishing/Finished/Created*

This phase comes when the computation and/or data retrieval process has been completed. Code developers can use this phase to close their files and network connections, clean up memory etc. This phase contains only one step with one method-call: Finish(). If the component supports being re-initialized the status switches back to *Created*; otherwise it can be released from memory.

3.3.7 Pausing and stopping computations

In many situations, the end user would like to keep more direct control over the computation process in order to stop or pause the computation. The data transfer mechanism of the OpenMI is defined as a single thread of synchronous GetValues() calls. As GetValues() is the mechanism that starts the process and keeps it going, any more direct interruption mechanism should be incorporated in the same thread.

The event mechanism is applied for this purpose, using an event handler that can grab and hold the computation thread if required. Pausing can be done by an 'event handler' that grabs and holds the computation thread, and returns the thread when resuming. Stopping can be done by an 'event handler' that grabs and holds the 'computation' thread, calls the Finalize functions of all involved components and kills the 'computation' thread. Any event type is suitable for this purpose, but status changed events (the ones surrounding the GetValues() call) are generally preferred. Note that the linkable component should be in a consistent state when it throws the event, as the user may want to save this state for future use.

The two mechanisms work in similar way. The 'Pause' variant is illustrated in Figure 28. In the example, two linkable components are involved: a Rainfall Runoff model and a RiverModel. They implement IBaseLinkableComponent and ITimeSpaceComponentExtension. The ILinkableComponent term in the figure is a simplification.

1. The user passes the start event to the main program. The main program triggers the RiverModel by an Update() call. The RiverModel sends a 'status changed' event after it receives the Update() call. The event mechanism takes care that the component gets the thread back after the event has been handled (if there are any listeners).

2. Once the thread is back, it performs a GetValues() call to one of the outputs of the RR model. If a computation is needed, the output item calls the Update() call of the RR model.

3. The Update() call of the RR model enters a time-stepping loop, in which at the start of each timestep a 'status changed' event is sent to indicate that the RiverModel is waiting for data, followed by a status change to *Updating*, indicating that all input data is available and that the component is computing.

4. Meanwhile, the end user requests the main program for a pause. The main program waits until it receives the next event from the computation thread, in this case a 'status changed' event, and holds the thread.

5. The end user asks the main program to continue, so the main program returns the thread to the RR model, which can continue by returning the results to the RiverModel.

6. The RiverModel sends a 'status changed' event when it receives control from the GetValues() call stack. After the thread is returned by the main program, the RiverModel can compute its timestep.

7. After computation, the RiverModel sends a 'data changed' event. When the thread is returned, it sends a 'status changed' event informing that the values are passed back.

8. When the thread is returned it passes the value set back.

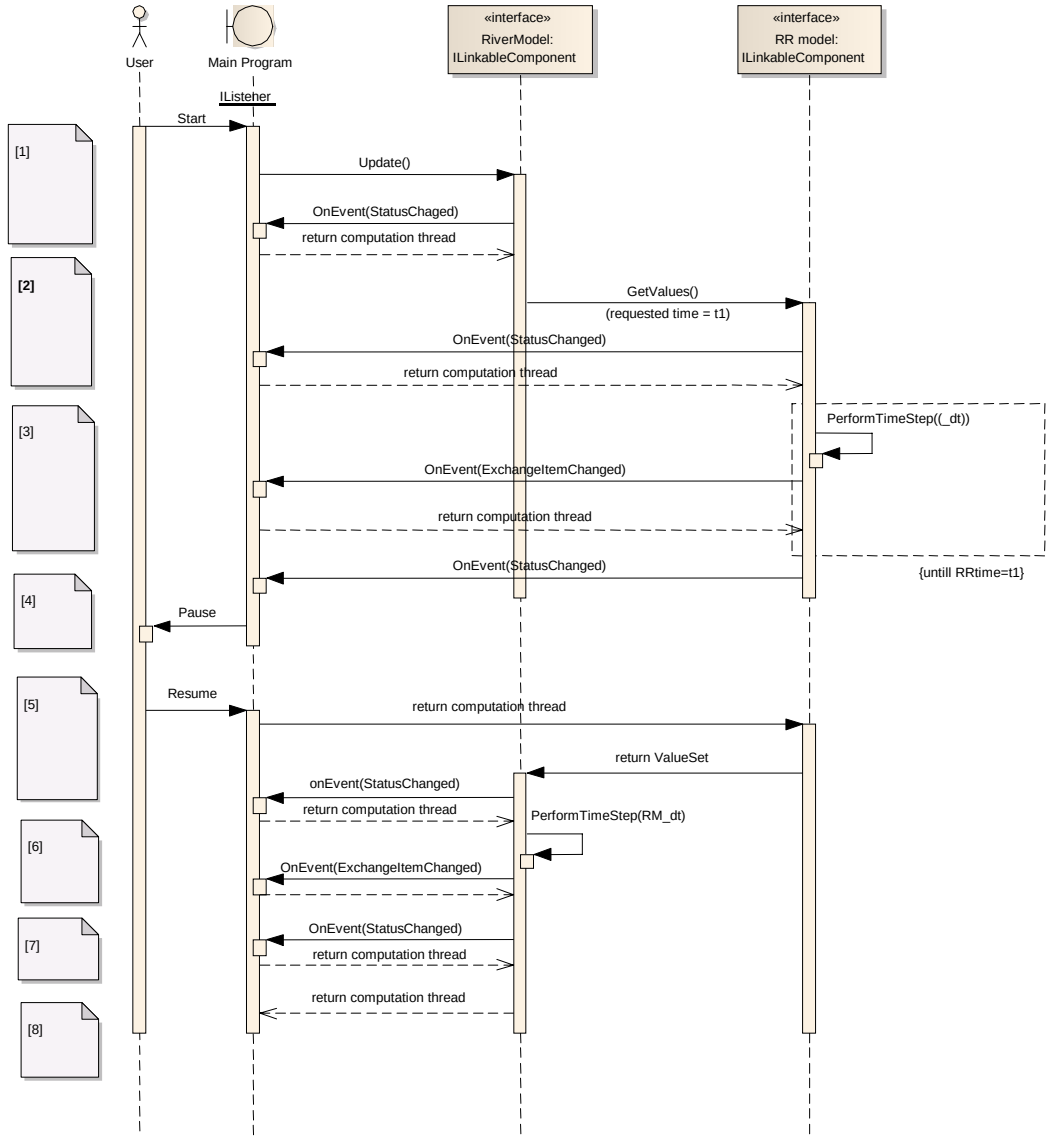


Figure 28 Pause and resume of a computation process (sequence diagram)

3.3.8 Miscellaneous issues

Using events for logging and visualization

The event mechanism is also used to enable on-line visualization. As soon as the content of an input or output exchange item has been changed, an ExchangeItemValueChanged event should be generated. Various other compononents, or more particularly the application in which the linkable component is running, can then react to the event. For instance:

1. Acting upon a GetValues() request on one of its outputs, the RR model starts computing until the request is met.

2. After that, the data is returned to the requesting component. Simultaneously, listeners that have subscribed to the ExchangeItemValueChanged vent type are notified that data has changed.

3. The receiving component handles the returned values and sends an ExchangeItemValueChanged event to notify listeners that the input item has received new values.

4. The visualization component is such a listener. It can ask the input item and/or the output item for the current values.

3.4 OpenMI compliance

3.4.1 Meaning of OpenMI compliance for developers

Each OpenMI-compliant component is available as a software unit that in short must implement the IBaseLinkableComponent interface and can implement one or more extensions of the OpenMI Standard2 namespace. This software unit should adhere to the calling phases as grouped in Section 3.3.1.

A developer can choose to make a component not compliant to the TimeSpace extension, but instead build a different extension (e.g. TimeOnly, SpaceOnly, DataDictionary, Ontology, etc.) and implement that for a more meaningful and customized data transfer. When finished the extension could be proposed to the OpenMI Association and become part of the OpenMI Standard as an endorsed extension.
Since extensions can be updated and distributed in libraries separate from the Standard library (which contains the base and the TimeSpace extension at the moment) they can evolve more quickly while the base functionality remains the same.

A few special situations may occur regarding a component's compliancy:

- IManageState interface is optional.
  If state management is not supported by a linkable component, you cannot implement the logic of the IManageState interface. Therefore you should not implement this interface or throw an exception.

- IByteStateConverter interface is optional.
  If state conversion to and from a byte stream is not supported by a linkable component, you should not implement this interface or throw an exception.

- No input exchange item.
  OpenMI-compliant data sources not acting as data destinations (e.g. input databases or monitoring stations) must return a list of length zero for the Inputs property.

- No output exchange item.
  OpenMI-compliant data destinations not acting as data sources (e.g. visualization) must return a list of length zero for the Outputs property.

The compliant component must have an associated registration file (the OMI-file).

3.4.2 Meaning of OpenMI compliance for users

Potential users of an OpenMI compliant component should understand the following point

"Being compliant is no guarantee that it can be usefully linked to another OpenMI compliant component. It is a necessary but not sufficient condition."

To be usable the user must also consider the following factors

1. Do both components target the same system infrastructure? Currently Java and .Net components cannot be mixed within the same composition.

2. Do both components implement the same OpenMI version? Development is under way to allow 1.4 components to be loaded alongside version 2 components within version 2 of the configuration editor; however, currently this is not possible. Even when possible, there will still be restrictions as version 2 of the standard exposes new (additional) data exchange concepts which are incompatible with version 1.4

3.Do both components provide and consume all the required data interactions for the underlying physics for the two models interaction? Breaking the underlying physics will cause instabilities (at best) or wrong answers (at worst). The underlying issue here is that OpenMI provides the means to couple models together, but cannot make judgements on the advisability of doing so. The responsibility for this lies with the user; it is expected that they should have competence in the use and application of both models.

References

Buschmann et.al., *Pattern-Oriented Software Architecture, a System of Patterns*, John Wiley & Sons, 1996.

OpenMI Association (2010) *Scope for the OpenMI (version 2.0)*. Part of the OpenMI report series.

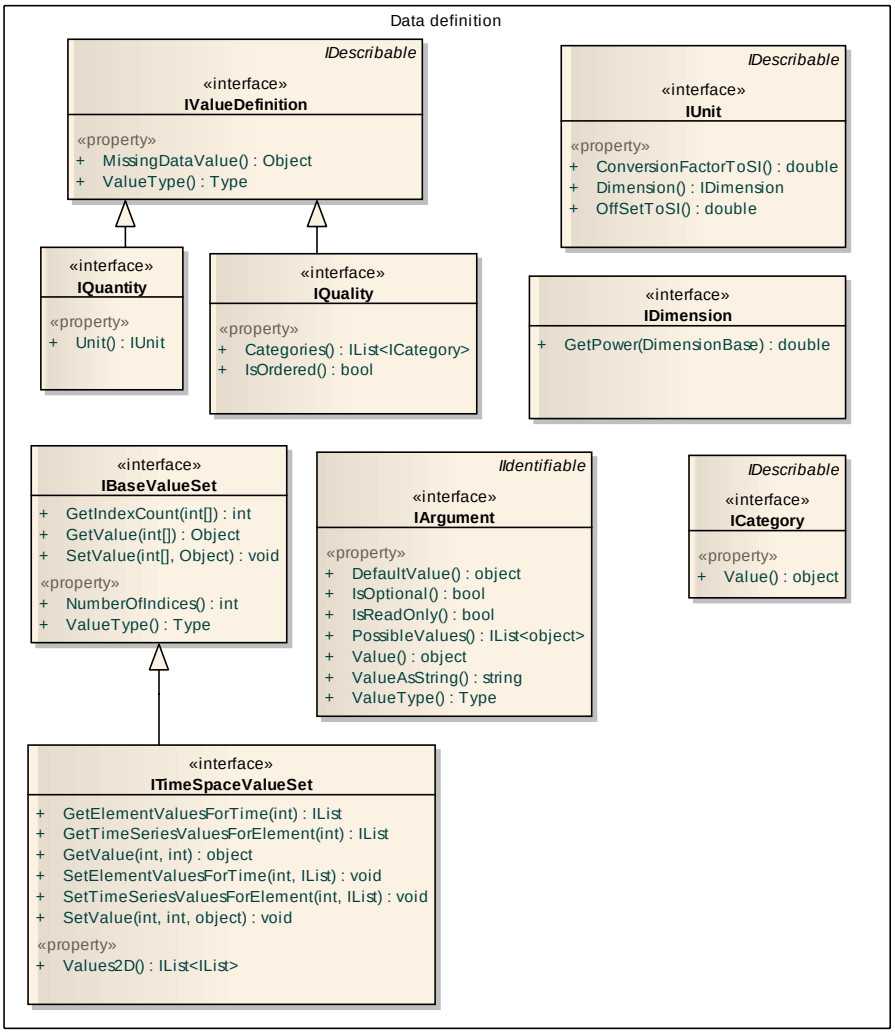
OGC (2002) *The OpenGIS Abstract Specification Topic 2: Spatial referencing by Coordinates* OGC 01-063r2, OpenGIS Consortium Inc.

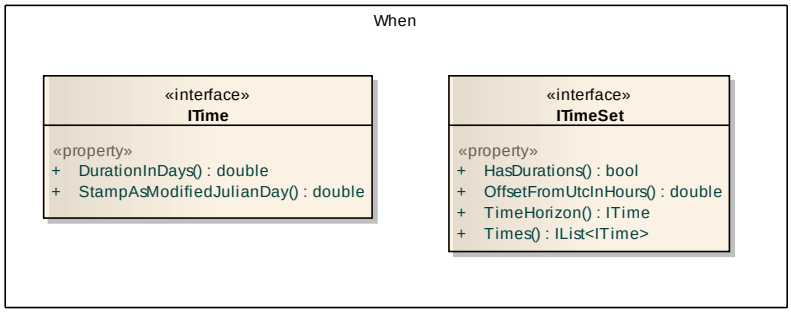
SpatialReference.org (2010) *About Spatial References*. http://spatialreference.org

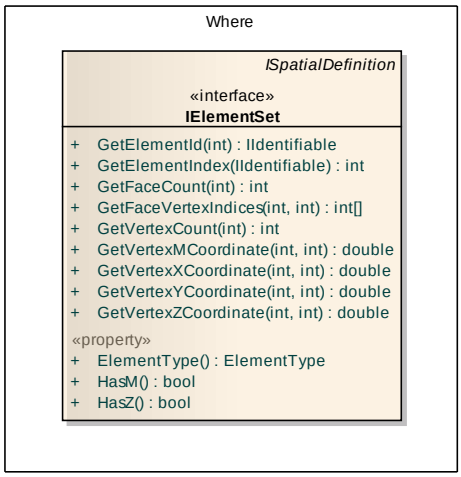
OpenMI.org (2010) *OpenMI Association*. http://www.openmi.org

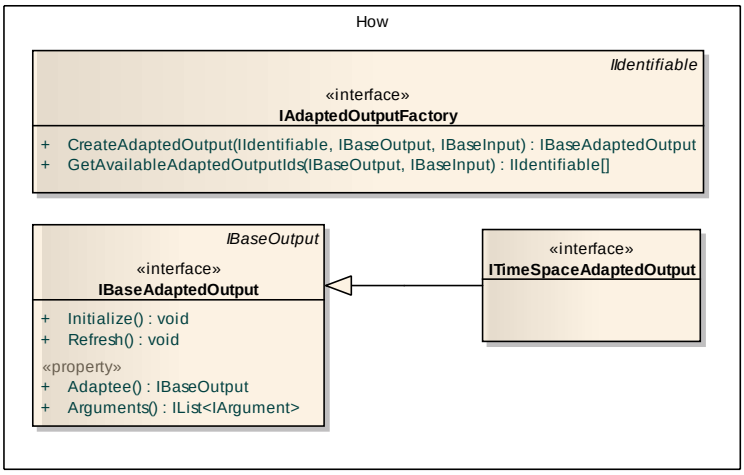
Appendix 1 org.OpenMI.Standard2 in short

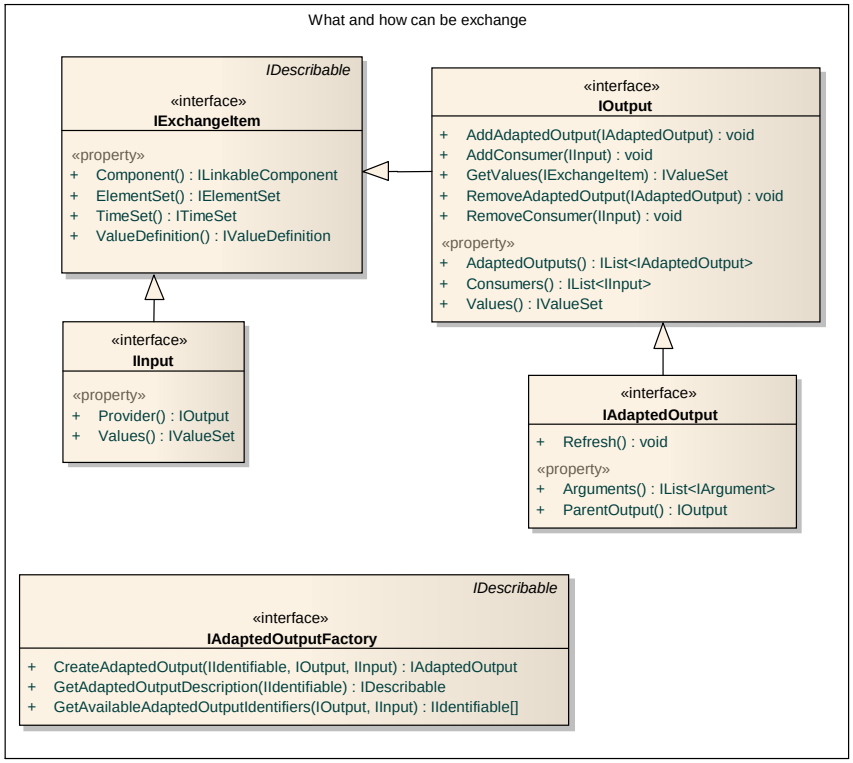
Appendix 1A The interface definitions

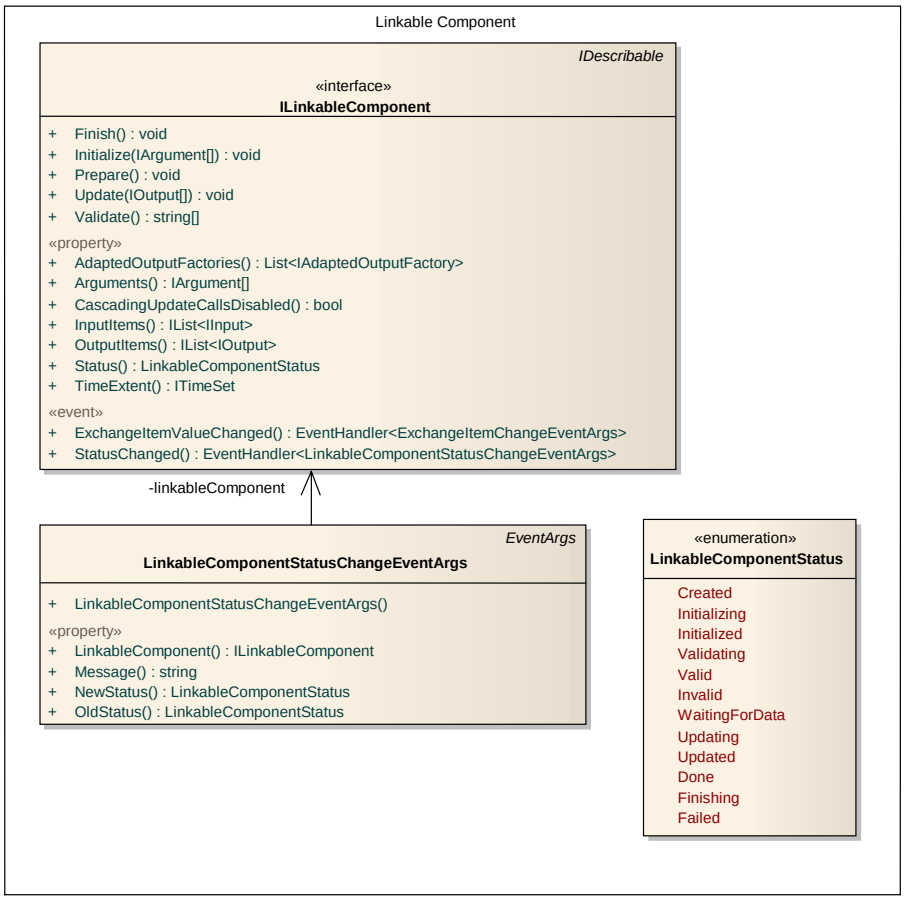





Where
ISpatialDefinition
«interface»
IElementSet
+ GetElementId(int) : IIdentifiable
+ GetElementIndex(IIdentifiable) : int
+ GetFaceCount(int) : int
+ GetFaceVertexIndices(int, int) : int[]
+ GetVertexCount(int) : int
+ GetVertexMCoordinate(int, int) : double
+ GetVertexXCoordinate(int, int) : double
+ GetVertexYCoordinate(int, int) : double
+ GetVertexZCoordinate(int, int) : double
«property»
+ ElementType() : ElementType
+ HasM() : bool
+ HasZ() : bool

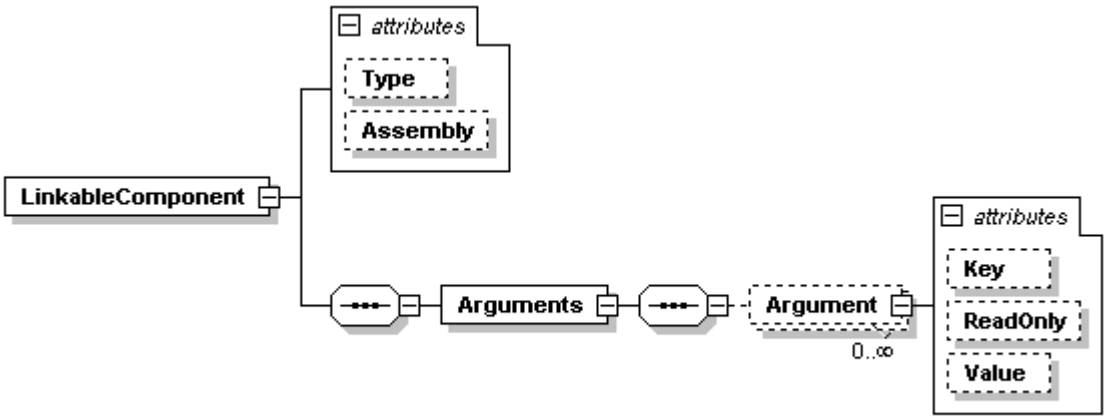

How
IIdentifiable
«interface»
IAdaptedOutputFactory
+ CreateAdaptedOutput(IIdentifiable, IBaseOutput, IBaseInput) : IBaseAdaptedOutput
+ GetAvailableAdaptedOutputIds(IBaseOutput, IBaseInput) : IIdentifiable[]
IBaseOutput
«interface»
IBaseAdaptedOutput
+ Initialize() : void
+ Refresh() : void
«property»
+ Adaptee() : IBaseOutput
+ Arguments() : IList<IArgument>
«interface»
ITimeSpaceAdaptedOutput


What and how can be exchange
IDescribable
«interface»
IExchangeItem
«property»
+ Component() : ILinkableComponent
+ ElementSet() : IElementSet
+ TimeSet() : ITimeSet
+ ValueDefinition() : IValueDefinition
«interface»
IOutput
+ AddAdaptedOutput(IAdaptedOutput) : void
+ AddConsumer(IInput) : void
+ GetValues(IExchangeItem) : IValueSet
+ RemoveAdaptedOutput(IAdaptedOutput) : void
+ RemoveConsumer(IInput) : void
«property»
+ AdaptedOutputs() : IList<IAdaptedOutput>
+ Consumers() : IList<IInput>
+ Values() : IValueSet
«interface»
IInput
«property»
+ Provider() : IOutput
+ Values() : IValueSet
«interface»
IAdaptedOutput
+ Refresh() : void
«property»
+ Arguments() : IList<IArgument>
+ ParentOutput() : IOutput
IDescribable
«interface»
IAdaptedOutputFactory
+ CreateAdaptedOutput(IIdentifiable, IOutput, IInput) : IAdaptedOutput
+ GetAdaptedOutputDescription(IIdentifiable) : IDescribable
+ GetAvailableAdaptedOutputIdentifiers(IOutput, IInput) : IIdentifiable[]


Linkable Component
IDescribable
«interface»
ILinkableComponent
+ Finish() : void
+ Initialize(IArgument[]) : void
+ Prepare() : void
+ Update(IOutput[]) : void
+ Validate() : string[]
«property»
+ AdaptedOutputFactories() : List<IAdaptedOutputFactory>
+ Arguments() : IArgument[]
+ CascadingUpdateCallsDisabled() : bool
+ InputItems() : IList<IInput>
+ OutputItems() : IList<IOutput>
+ Status() : LinkableComponentStatus
+ TimeExtent() : ITimeSet
«event»
+ ExchangeItemValueChanged() : EventHandler<ExchangeItemChangeEventArgs>
+ StatusChanged() : EventHandler<LinkableComponentStatusChangeEventArgs>
-linkableComponent
EventArgs
LinkableComponentStatusChangeEventArgs
+ LinkableComponentStatusChangeEventArgs()
«property»
+ LinkableComponent() : ILinkableComponent
+ Message() : string
+ NewStatus() : LinkableComponentStatus
+ OldStatus() : LinkableComponentStatus
«enumeration»
LinkableComponentStatus
Created
Initializing
Initialized
Validating
Valid
Invalid
WaitingForData
Updating
Updated
Done
Finishing
Failed

Appendix 1B The OMI-file definition

Graphical view of the OMI-file structure



The Xml Schema Definition of an OMI-file

```
<?xml version="1.0" ?>

<xs:schema id="LinkableComponent" targetNamespace="http://www.openmi.org/LinkableComponent.xsd"

   xmlns:mstns="http://www.openmi.org/LinkableComponent.xsd"
xmlns="http://www.openmi.org/LinkableComponent.xsd"

   xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"

   attributeFormDefault="qualified" elementFormDefault="qualified">

   <xs:element name="LinkableComponent">

     <xs:complexType>

       <xs:sequence>

         <xs:element name="Arguments" minOccurs="1" maxOccurs="1">

           <xs:complexType>

             <xs:sequence>

               <xs:element name="Argument" minOccurs="0" maxOccurs="unbounded">

                 <xs:complexType>

                   <xs:attribute name="Key" form="unqualified" type="xs:string" />

                   <xs:attribute name="ReadOnly" form="unqualified" type="xs:boolean" use="optional" />

                   <xs:attribute name="Value" form="unqualified" type="xs:string" />

                 </xs:complexType>

               </xs:element>
```

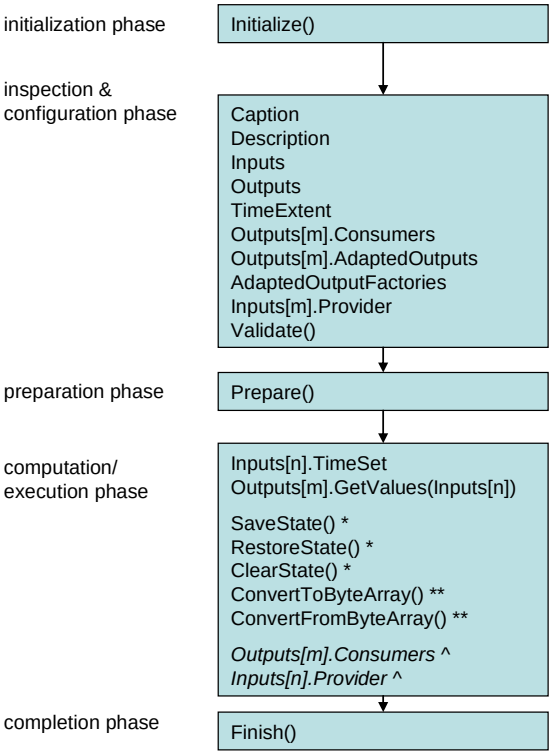
```
            </xs:sequence>

          </xs:complexType>

        </xs:element>

      </xs:sequence>

      <xs:attribute name="Type" form="unqualified" type="xs:string" />

      <xs:attribute name="Assembly" form="unqualified" type="xs:string" use="optional" />

    </xs:complexType>

  </xs:element>

</xs:schema>
```

XML-look of the OMI file

```
<?xml version="1.0"?>

<LinkableComponent                                          Type="Deltares.OpenMI.Wrapper.DLinkableComponent
     Assembly="..\..\bin\Deltares.OpenMI.Wrapper.dll"
         xmlns="http://www.openmi.org/LinkableComponent.xsd">

  <Arguments>

    <Argument Key="Model" ReadOnly="true" Value="CF" />

    <Argument Key="Schematization" ReadOnly="true" Value=".\CmtWork\sobesim.fnm" />

  </Arguments>

</LinkableComponent>
```

Appendix 1C The states in dynamic utilization

Deployment phases of an OpenMI LinkableComponent, and the involved methods and properties:



Methods from IPublisher interface

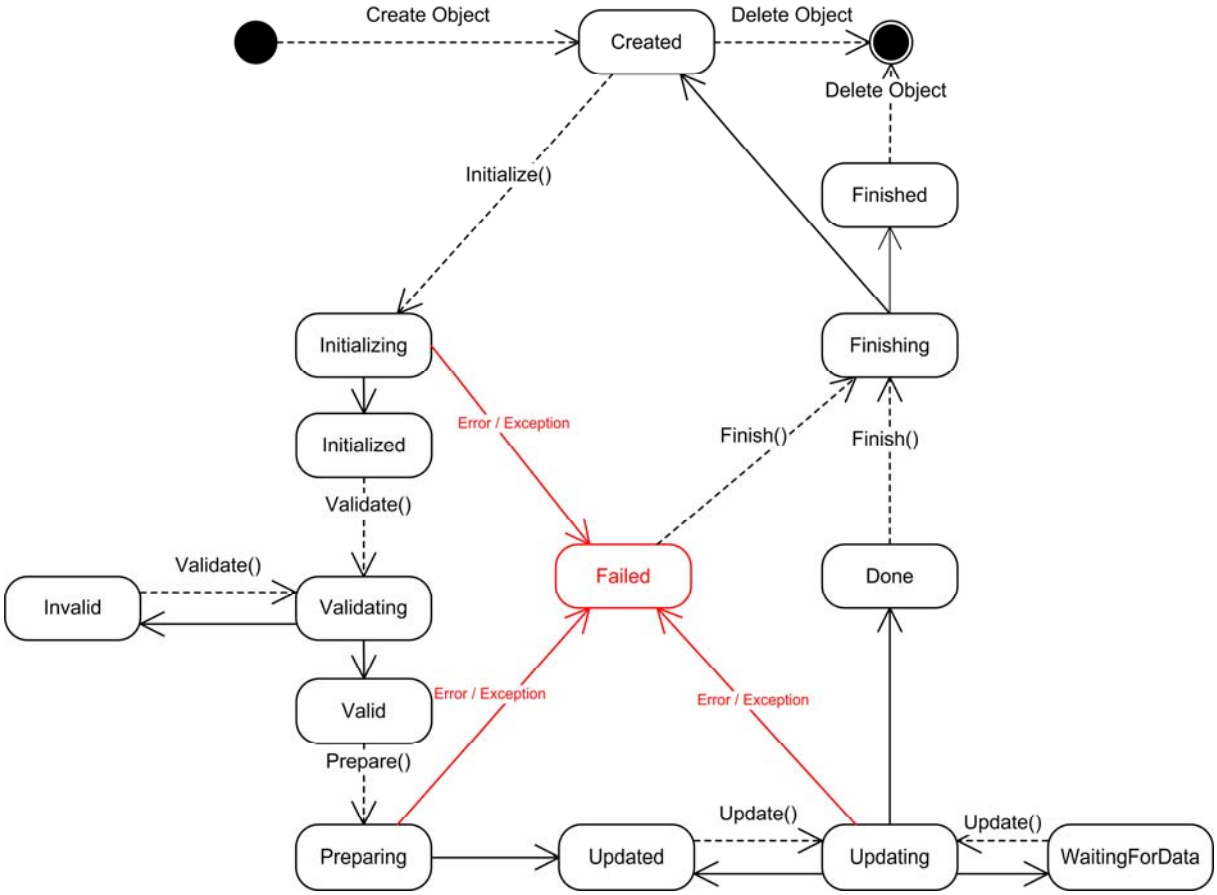
Implementation is optional

* if component implements IManageState interface

** if component implements IByteStateConverter interface

^ if component supports dynamic adding/removing connections

State diagram for IBaseLinkableComponent.Status:



Appendix 2 org.OpenMI.Standard2 API-specification

The specification is fully described in the *OpenMI Standard 2.0 References*, which is available from www.openmi.org.

Appendix 3 Overview of changes

Appendix 3A Changes from version 1.4.0 (September 2007) to version 2.00 (Juli 2010)

General changes

- Document rewritten according to the new OpenMI.Standard2 architecture

Architectural changes

- More symmetry in the handling of space and time
- Splitting the standard into a base part and extensions to the standard
- GetValues() moved from ILinkableComponent to IBaseOutput
- Replacement of the link by a sequence of adapted outputs
- Introduction of IQuality next to IQuantity
- Redesign of IValueSet
- Values property on IBaseOutput and IBaseInput
- Values property on IBaseInput has set access
- Introduction of IBaseLinkableComponent.Update() and IBaseLinkableComponent.Status
- Many other details, arising from one or more of the changes above

OMI-file changes

- None

Appendix 3B Changes from version 1.0.0 (May 2005) to version 1.4.0 (September 2007)

General changes

- None

Architectural changes

- None

OMI-file changes

- None

Reason for change: versioning issues with .NET framework and introduction of signature file.

Appendix 3C Changes from version 0.99 (November 2004) to version 1.0.0 (May 2005)

General changes

- None

Architectural changes

Data definition changes

- Added NUM_OF_EVENT_TYPES field to EventTypes enumeration
- Modified return type of method IDimension.GetPower

OMI file changes

- Changed namespace into www.openmi.org

Appendix 3D Changes from version 0.91 (June 2004) to version 0.99 (November 2004)

General changes

- Merged IExchangeModel and ILinkableComponent interfaces
- Introduced OMI-file and associated XSD as entry point for systems utilizing OpenMI LinkableComponents

Architectural changes

Data definition changes

- Introduced EventTypes as fixed enumeration instead of a recommended convention
- Modified/reduced IEvent interface
- Extended enumeration of ElementTypes to include the 3-dimensional space
- Extended IElementSet with methods to obtain faces of 3D-objects
- Extended IDataOperation with a validation method
- Extended IValueSet with a validation method

LinkableComponent changes

- Moved methods from IExchangeModel to ILinkableComponent
- Removed Finalize method
- Renamed PrepareForComputation() method into Prepare() method
- Renamed EarliestNeededTime property into EarliestInputTime property
- Added TimeHorizon property
- Added Validate() method and Finish() method

Appendix 3E Changes from version 0.9 (May 2004) to version 0.91 (June 2004)

General changes

- None

Architectural changes

Data definition changes

- Introduced IArgument to replace IDataOperationParameter and IComponentArgument
- Reduced complexity of data operation related interfaces
  Modified IDataOperation
  Removed IDataOperationDescriptor, DataAspectType, SourceTargetRole, IDataOperationDescriptorParameter, IDataOperationParameter, ParameterSpecificationType
- Renamed enumeration BaseQuantity into DimensionBase
- Removed IVisibleComponent
- Removed IException as an OpenMI-specific class. Exceptions are based on the Exception class as provided in the development environment
- Combined the functionality of IExchangeModel and IComponentDescriptor into one 'metadata interface', namely IexchangeMode; this interface includes a Create() method to create and populate an exchange model
- Combined the construction and run-time access from ILinkableComponentFactory and ILinkableComponent into one 'run-time interface', namely ILinkableComponent

Appendix 3F Changes from version 0.6 (May 2003) to version 0.9 (May 2004)

General changes

- Moved the Standard from a mixture of (abstract) class implementation and XML to an 'interface only' specification
- Introduced a new namespace for this purpose (OpenMI Standard2) and skipped the old ones (org.OpenMI.System and org.OpenMI.System.Components)
- The HarmonIT project provides a default implementation with the namspace org.OpenMI.Backbone, org.OpenMI.Utilities, org.OpenMI.Configuration and org.OpenMI.Tools; however, other implementations of the standard interfaces are welcome
- Reformulated the methods into a combination of properties, {get} only, and methods. Properties introduced for those items that typically are implemented as properties Methods are applied for those items that require internal data processing/querying
- Introduced and applied a general pattern to query an internal list (array) of items

Architectural changes

Data definition changes

- ElementSet:
  - Introduction of a rigid ElementSet interface, still based on concepts of elements and vertices (geo-referenced)
  - Underlying items (elements and nodes) are skipped from the Standard; they may still be useful in an implementation
  - Incorporated ElementType (shape type) as enumeration in the standard
  - Note: The IElementSet interface can be queried only; the ElementSet object can be populated at instantiation, or an implementation can be provided with convenience functions to add and remove elements
- SpatialReferenceSystem:
  - Skipped the entire spatial reference system specification part, leaving only a string pointer to the Spatial Reference System applied
- Time:
  - Time reference system is fixed to the ModifiedJulianDate
- Quantity:

- Introduced the dimension interface (IDimension) to enable (physical) dimension checks

Run time interfaces

- ImanageState:
  - Renamed ModelEngine class into IManageState
  - Separated IManageState interface from its former base-class, ILinkableComponent
- Trigger:
  - Skipped the Trigger as separate class. When adding links to the component chain, one link needs to be added/appointed as the trigger link
- IlinkableComponent:
  - Moved functionality of GetPotentialOutputTimes, into a separate interface (IDiscreteTimes)
  - Skipped Validate method; validation is to be done at Initialization time; an exception needs to be thrown if an error occurs
  - Reshuffled link-handling methods
- IEvent and Iexception:
  - Slight modification of properties and methods

Meta data interfaces

- IExchangeModel and associated interfaces:
  - Renamed LinkableDataDescriptor into ExchangeModel
  - Introduced IExchangeItem as a grouping entity including derived interfaces to describe (potential) input/output combinations for a linkable component
  - Preserved ‘Input/Output’ as the key identifier for data exchange directions from the component perspective
  - Skipped ‘Potential’ in the naming of the various methods
  - Skipped the factory associated to the ExchangeModel (the former LinkableDataDescriptorFactory)
  - Skipped ComponentFactoryType
  - Introduced IComponentDescriptor interface and IComponentArgument interface; the latter replaces the factory arguments
- IDataOperationDescriptor and IDataOperationDescriptorParameter
  - Introduced a set of interfaces and enumerations to describe data operations and associated parameters

Configuration-related interfaces

- ILink:
  - Transformed methods into properties
  - Renamed link properties from Provider/Acceptor into Source/Target (accounting for the direction of data transfer through the link)
- Miscellaneous:
  - Skipped hints (in the Scenarios part) to possible implementations of a configuration utility