OpenMI

The OpenMI Document Series

Part C - the org.OpenMI.Standard interface specification

For OpenMI (Version 1.4)

Title	OpenMI Document Series: Part C - the org.OpenMI.Standard interface specification for the OpenMI (version 1.4)
Editor	Peter Gijsbers, WL | Delft Hydraulics, Delft, The Netherlands
Authors	Jan Gregersen, DHI Water and Environment, Hørsholm, Denmark Stefan Westen, HR Wallingford Group, Wallingford, UK Stef Hummel, WL | Delft Hydraulics, Delft, The Netherlands Rob Brinkman, WL | Delft Hydraulics, Delft, The Netherlands
Document production	Peter Gijsbers, WL | Delft Hydraulics, Delft, The Netherlands
Current version	V1.4
Date	21/05/2007
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 the OpenMI Association who own the particular methodologies, ideas and proposals involved.
Acknowledgement	This document has been produced as part of the OpenMI-Life project. The OpenMI-Life project is supported by the european Commission under the Life Programme and contributing to the implementation of the thematic component LIFE-Environment under the policy area "Sustainable management of ground water and surface water managment" Contract no : LIFE06 ENV/UK/000409. The first version of this document has been produced as part of the HarmonIT project; a research project supported by the European Commission under the Fifth Framework Programme and contributing to the implementation of the Key Action “Sustainable Management and Quality of Water” within the Energy, Environment and Sustainable Development. Contract no: EVK1-CT-2001-00090.

Preface

OpenMI stands for Open Modeling Interface and aims to deliver a standardized way of linking of environmental related models. This document describes the standardized OpenMI interface specification in detail. It is the third in 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.

Other titles in the series include:

A. Scope

B. Guidelines

C. org.OpenMI.Standard interface specification (this document)

D. org.OpenMI.Backbone technical documentation

E. org.OpenMI Development Support technical documentation

F. org.OpenMI.Utilities technical documentation

The interface specification is intended primarily for developers. For a more general overview of the OpenMI, see Part A (Scope).

The official reference to this document is:

OpenMI Association (2007) *The org.OpenMI.Standard interface specification*. Part C 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.

Further information

Further information on the OpenMI Association and the Open Modelling Interface can be found on http://www.OpenMI.org.

Table of contents

List of figures

List of tables

1 Introduction

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 upon which 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.

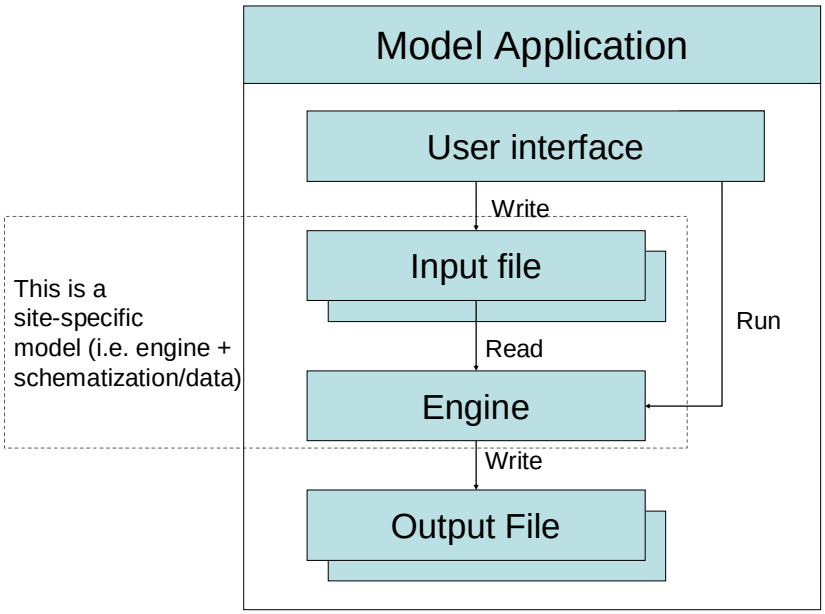


Figure 1 Model application pattern

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 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. 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, 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 org.OpenMI.Standard. Software components that implement and use these interfaces properly are called OpenMI compliant.

To support the development of OpenMI compliant components, a Software Development Kit (SDK) has been provided. This SDK contains a default implementation of these interfaces in the org.OpenMI.Backbone package (see Figure 2). In addition, the SDK provides utilities to support wrapping of legacy code to configure and deploy the components (all part of the org.OpenMI.Utilities namespace), a general support package that is not related to OpenMI (the org.OpenMI.DevelopmentSupport namespace) and front-end tools to enable interaction with its users (the org.OpenMI.Tools namespace). Utilization of the OpenMI SDK is not obligatory to develop OpenMI compliant components.

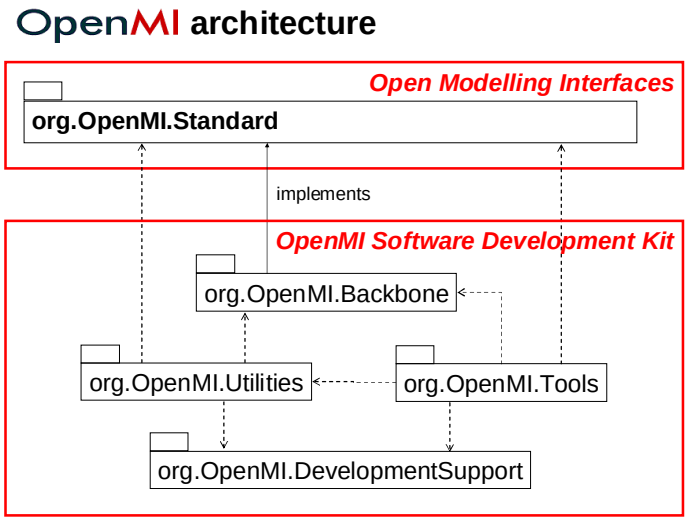


Figure 2 OpenMI architecture namespaces

1.3 Document structure

The document starts in chapter 2 with an explanation of the concepts underlying the Open Modelling Interfaces and its associated namespace org.OpenMI.Standard. 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 an API description (see Annex I). Annex II 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 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 the IT-expert wants to create its own OpenMI compliant components. Other documents of the OpenMI report series can inform the reader about the scope of OpenMI (OpenMI Association 2007a), how to apply OpenMI in practice (OpenMI Association,

forthcoming 2007b) and how the OpenMI interfaces have been applied and implemented in the SDK (OpenMI Association 2007d,e,f).

In order to understand this document, one needs to have 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: Concepts

2.1 Introduction

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 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. OpenMI defines a standard interface that engine components must implement to become OpenMI compliant engine components. When an engine component implements this interface it becomes a linkable component.

OpenMI is based on the 'request & reply' mechanism. According to Buschmann et al. (1996), OpenMI is a pull-based pipe and filter architecture which 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. OpenMI defines both the component interfaces as well as how the data is being exchanged. The components in OpenMI are called linkable components to indicate that it involves components that can be linked together.

From the data exchange perspective, OpenMI is a purely single-threaded architecture where an instance of a linkable component handles only one data request at a time before acting upon another request. Data exchange in the OpenMI-architecture is triggered by a component at the end of the component chain. Once triggered, components exchange data autonomously without any type of supervising authority. If necessary, components start their own computing process to produce the requested data. Only when output needs to converge to a certain criteria, a linkable component with controlling functionality might need to be incorporated. Most important however, is the fact that 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 view point of how the standard will be used. For running a linked simulation we have identified four main phases:

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.

Translating these four phases into tasks we arrive at the following five tasks that must be carried out:

1. *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 2.3.

2. *Generic model access*:
   This boils down to 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 2.5. It also involves the generic construction of a specific model without knowing its details.
3. *Meta data defining potentially exchangeable data:*
   This definition describes the data that can potentially be provided and accepted by a linkable component, see Section 2.4.
4. *Definition of actually exchanged data:*
   A link describes data that is actually transported between two different linkable components, see Section 2.6.
5. *Data transfer:*
   This requires that linkable components have a standard interface to allow data exchange, see Section 2.7.

OpenMI addresses these issues, but not more than this. The data being passed between models will be typically boundary conditions (e.g. fluxes), but also transfer of model parameters and other data sets is possible. The OpenMI standard is focused on data exchange on a time basis (either time stamp or time span), being as complete as possible in describing the data being exchanged. To keep its mandatory interfaces as lean and mean as possible, its interfaces are thus not necessarily optimized for all data exchange functions that can be imagined in a modelling system. However, the OpenMI interfaces provide various places where code developers can offer functionality to end-users to define their data exchange as desired, keeping full control on the process.

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 Data definition

To define the data that is exchanged, a distinction is made between *where*, *when*, *what* and the (numeric) values itself. The data is described by identifying the values (data) itself, the geometry on which the data is defined, the time(s) for which the data is valid and the actual quantity it represents.

2.3.1 Where

The space where the values apply is indicated in a finite element way by an ordered list of elements (the element set[1]), 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 coordinates to locate the element in a geo-reference system

Every element set contains a string reference to a geo-reference system, which defines a coordinate system for the coordinates (e.g. WGS84[2]). This allows element sets for every model to be defined in the coordinate system of the model without transformation and allows an OpenMI

[1] In reality the element set is, not an object which can be sorted, but a one-dimensional array of elements, where ordering is essential.

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

implementation to perform the coordinate 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).

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

2.3.2 When

Time in OpenMI is defined either by a timestamp or a time span. A time stamp is a single point in time, whereas a time span is a period from begin to end time. Each of these times is represented by the Modified Julian Date. Both interfaces are inherited from the time interface. This means that wherever time is expected, both a time stamp and a time span can be indicated.

2.3.3 What

The physical semantics of the values is described by a quantity, combined with a unit in which its value is expressed. Water level (m) and amount of flow (cubic metre/hour) are examples of quantities. The physical nature of a quantity is also related to the shape type of an element. E.g. a water level can be given in a point or along a poly-line or over a plane, but not in a volume.

OpenMI does not use a standardized data dictionary. However, to ensure that linkages between quantities are correct, two aspects 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 a performance penalty of unnecessary transformations. Secondly, the dimension needs to be provided for each quantity. By convention, this dimension is expressed as a combination of base quantities.

2.3.4 Values

The values themselves are represented as a one-dimensional array of scalars or vectors contained in a so-called value set. Information about the values (space, time, quantity) and their ordering is assumed to be known by the user. A single scalar or vector value exists for every element in the element set defined.

This approach is more flexible compared to approaches, which limit the data structures to be exchanged to grid-based computed fields or vector fields used as boundary conditions. Other types of data such as model parameters or numerical parameters (e.g. for optimization purposes), typically not geo-referenced, can also be represented by defining a non-geo-referenced element set. E.g. economic models may not address spatial variability, but their entity of analysis can easily be represented by an element set consisting of one (non)-geo-referenced element.

2.3.5 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 some conditions, specification of additional data-operations is desired to obtain the data in the way it is needed. Most common situation will be the need for temporal aggregation over a time span. This functionality can be used if one model, running a small time step, 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 time-steps of the source component and the target component do not coincide. In addition to temporal data operations, spatial data operations are foreseen as well as a miscellaneous group of data operations.

2.4 Meta data defining potentially exchangeable data

An essential part of standardizing data exchange is the meta-data telling which components, model-schematizations and data sets are involved and what can be exchanged in terms of quantities (what does it represent), element sets (where does it apply), time (when does it apply) and data operations (how should it be provided). The availability of this meta-data 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 element set 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 can be provided or accepted by a component. Each item that can be exchanged contains information on the role of the data (input or output), the quantity it represents and the element set where it applies. Output exchange items also contain information on the data-operations that can be provided by the delivering component. The combination of a quantity on an element set has been chosen as one typically links the boundaries of one model to the boundaries of another model (e.g. the ‘bottom’ of a river with the ‘top’ of a ground water model). Note that Section 2.6 describes the actual link between two linkable components, i.e. the data that actually will be exchanged.

The list of exchange items is specific for a combination of a model code with a model schematization. The exchange items are mainly used during design/configuration time to set up the links between different models.

2.5 Generic model access

Generic model access is essential for the component based paradigm as adopted in OpenMI. At the core of 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. To enable event publishing, a linkable component is inherited from an IPublisher-interface. 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.

GetValues() is the most important run-time method of a linkable component. It returns the data requested[3], if needed by invoking a computation. All other methods are supportive, e.g. during the initialization phase, to handle links, and to clean up. 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. Again 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).

Basically 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 device that need to be accessed online.

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

- Monitoring databases containing historic data.
- Data driven models such as Artificial Neural Networks

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 on the OMI file are provided in Section 3.2.6.

2.5.1 Wrapping legacy code

For legacy code, wrapping will most often be the technological choice to migrate to 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[4]. 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 OpenMI

2.6 Definition of actually exchanged data

2.6.1 The link: purpose

As indicated, one linkable component can retrieve data from another linkable component by invocation of the GetValues()-method. However, this is only possible if two components have information about each others existence and have a clear idea on the kind of data that is requested. In other words, it is only possible if (i) the accepting component can identify the providing component, (ii) the providing component knows what to deliver (expressed in quantity, location, time) and to whom (accepting component), and (iii) the providing component understands the relation with its internal data. This information is contained in the link. A link defines actual data exchange of a semantically similar quantity between two linkable components (the source component and the target component).

2.6.2 The link: content

The link specifies the following information:

- The source component and the target component.
- The source quantity and target quantity, i.e. two aliases for a semantically similar quantity. The source quantity may differ from the target quantity by name as well as by unit, only in that a unit conversion is required to transform the values. Of course the semantics and the dimension of both quantities should be similar.
  (The source quantity information may be redundant if a similar dictionary is applied)
- The list of data operations that must be applied by the source component before providing the data.
- The element set on which the quantity is requested by the target component (and thus delivered by the source component).
- The element set on which the quantity is defined internally in the source linkable component. (Dependent on the intelligence of the providing component, this information may be redundant).

Please note that links are unidirectional (i.e. from the source component to the target component). A link refers to a single semantic quantity only, which, however, may have two different names on each side of the link. This means that with bi-directional communication or with multiple quantities, multiple links must be configured.

[4] A default implementation is provided in the org.OpenMI.Utilities package.

In water related data exchange the spatial geometry underlying the data exchange typically is persistent over time. However, some advanced models (e.g. wave models) contain geometry that changes over time. Element sets therefore contain a property that indicates if it behaves dynamic over time. In addition, it is allowed to add and remove links at run-time in case the geometry should change.

2.6.3 The link as an argument

The link has a string ID which is passed as an argument in the GetValues()-call of the linkable component. This allows independent communication over different links from one source component to multiple target components. This means that data exchange takes place over entire (target) element sets. In case the target component is only interested in the information on a part of the element set, the target component should perform the selection itself or the link should be configured to contain only the appropriate part of the target element set. Both ways avoid the need to define generic geometric data selection algorithms, thus keeping the standard simple and allowing high performance implementations. Of course, such selection algorithms can be provided as utilities but they are not part of the OpenMI standard.

2.7 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). Therefore, OpenMI has been designed as a pull-based pipe and filter system where the target component requests data from the source component and blocks (i.e. does not process any new call) until this data is returned. Every linkable component is a target component, a source component, or both.

Characteristic to water resources modelling and management is the importance of time. Often, data is exchanged for the same quantity and spatial elements, but for different time slots. Therefore, a time-indication has been incorporated as a separate argument in the GetValues()-call.

An instance of a linkable component handles only GetValues() request over one specific link 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.

2.7.1 Pull driven communication

Since the ‘target’ pulls the data when needed, this mechanism is called pull-driven. Linkable components can be connected in a chain, where the last component in the chain triggers the entire stack of data exchange. Figure 3 illustrates three chain layouts:

- an 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.

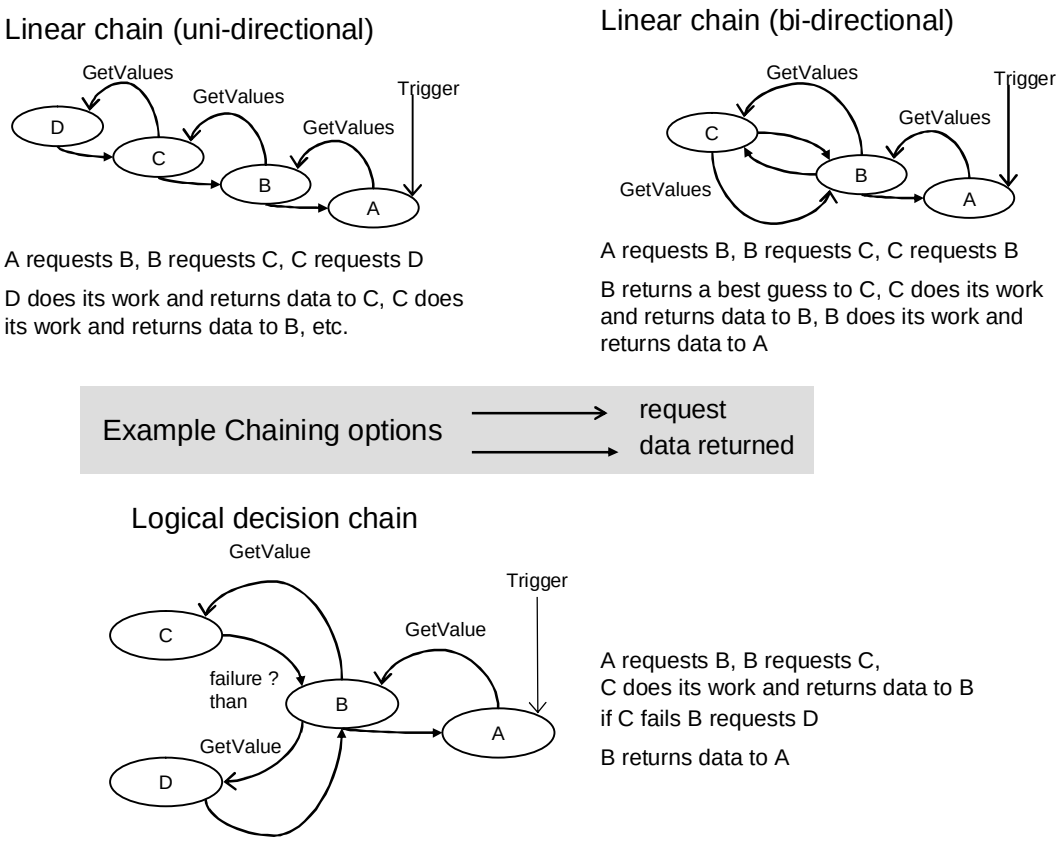


Figure 3 Different chain layouts with the pull-mechanism

As mentioned before, each link is unidirectional and describes only the communication of a single quantity. Therefore, for bidirectional communication at least two links are needed (e.g. in case of iterations). For more details, see Section 3.3.4.

2.7.2 Bi-directional links

The presence of a synchronous (blocking) GetValues() call requires some attention for the case of bi-directional links or more general in the case of cycles through the network of linkable components connected by links. In this case, one can imagine that a straightforward implementation of the GetValues() call might lead to infinite recursion.

In OpenMI, this problem is prevented by putting the obligation on a linkable component to not process additional GetValues() calls itself when it is already inside a GetValues() call. 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 cases where iteration is implemented.

Bi-directional links might require specific iteration functionality to preserve numerically stable solutions. The basis of this functionality is the combination of the pull-mechanism with state management. Only for this special case, 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. Dependent on

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

2.7.3 Time synchronisation

Each GetValues() request initiates a processing activity (e.g. computation) when needed to respond properly. As many water related models progress over time, the time argument is the controlling variable for any processing activity. Linkable components that do not progress over time can neglect the time-argument. 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.

In case the requested timestamp does not match the time stepping in the computation and the computation is already ahead in its computation, an interpolated value is to be delivered. Note that the model code developer decides how this interpolated value is computed and if any buffering is 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 bi-directional 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.

2.8 Events

Within modern software systems, events are often applied for all types of messaging. Within OpenMI a lightweight event mechanism is applied, using a generic Event interface and an enumeration of event types (see Table 3 in Section 3.2.5.6) to allow the implementation of generic tools that perform monitoring tasks such as logging, tracing, or online visualization. Linkable components can generate events to which other linkable components or tools can subscribe. In this way, it becomes possible to implement these generic tools without requiring any knowledge of the specific tools in the components themselves. By adopting the OpenMI event types, system developers can use those tools without additional effort. Note that the event mechanism should not be used to pass data sets. Data sets should be retrieved through the GetValues() call.

The event mechanism is also used to facilitate pausing and resuming of the computation thread, as the computation process of an entire model chain is rather autonomous and not controlled by any master controller (see Section 3.3.8). Once a component receives the thread, it preferably sends 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 a GetValues()-call. Stop 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.

2.9 Assumptions underlying the OpenMI architecture

2.9.1 Links are static or semi-static

It is assumed that in most modelling situations the components involved and quantities to be exchanged are static at run-time. While an element set will typically be static as well, advanced models may utilize changing coordinates for their elements (e.g. to describe waves). A version number has been introduced to the element set to assist code developers in identifying changing element sets over time. The interface of an element set can be queried at any moment.

Although the link handling methods of a linkable component do not forbid adding or deleting links at run-time, no specific method has been incorporated to simplify the task to update the properties of a link, except for an element set. If data needs to be exchanged for new quantities, a new instance of a link needs to be created, populated and added to the linkable component at run-time.

2.9.2 Time is referenced

Time is considered essential in OpenMI. While numerous models just run time steps without knowledge of the associated calendar, this knowledge is required in order to link them to other models. Hence, all time information in OpenMI is referenced and by convention is expressed as a Modified Julian Day (see Section 3.2.2.2).

Generally, static models do not bother about time. However, to fit in 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.

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

The OpenMI standard puts as much responsibility at the level of the providing linkable component. This is done since 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 or data operations. In case data is not available at the requested time or element set, the providing linkable component knows - in general - best how the available data should be processed to deliver a value at the requested time and element set.

2.9.4 Knowledge gaps need to be filled with domain expertise

Even while 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 4 illustrates the various lay-out options for the example of a groundwater-river model linkage.

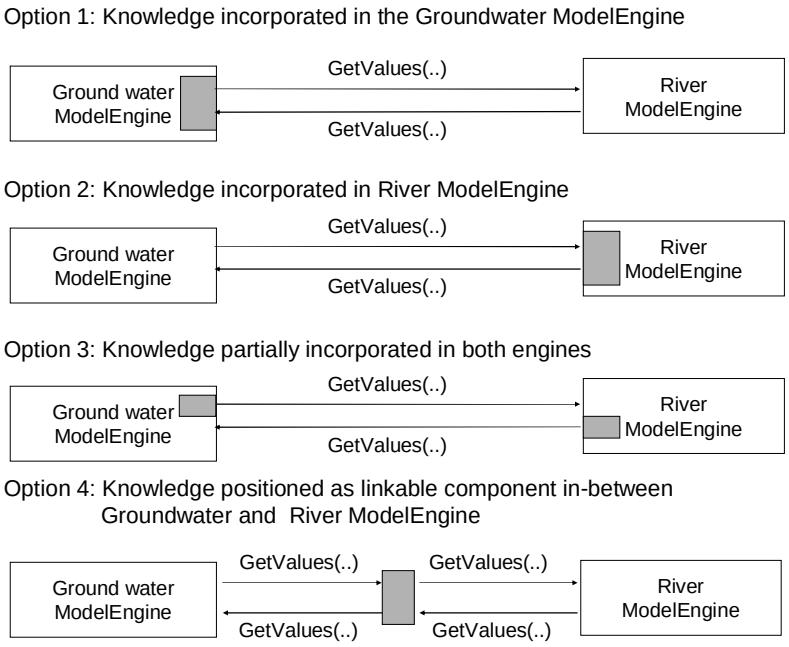


Figure 4 Example of different of layouts to capture knowledge needed to link two water domains

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

2.9.5 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 desired.

Nesting components

A linkable component in 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 the same vain, 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. Instead, guidance is given in

the Guidelines document (OpenMI Association, 2007b), by describing examples 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.

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. The guidelines describe in more detail how to implement controllers using the linkable component interface.

2.10 Miscellaneous issues

2.10.1 Efficiency considerations

Exposing meta-data

Although the methods to expose meta-data 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 provides a clear shift from establishing the connections towards the computational phase. Developers can choose whether they expose meta-data which is captured in files or whether they query engines. The former may be handy in multi-user environments, and in multi-processing jobs where one wants 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 for data through a GetValues()-call. In addition, its inclusive validation may save expensive CPU resources when some engines are not ready.

Transferring time series

The GetValues() method is defined to obtain a set of values which is valid for one time span or time stamp. Obtaining a time series requires multiple calls of this method. If supported by the providing component, a specific data operation flag may also be used to indicate that all available time stamps within a time span are requested.

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) time steps ahead in time. This effectively constitutes the method to do parallel computations in OpenMI.

2.10.2 Distributed computing

OpenMI has been designed in such way that the computational process calls of all linkable components (the GetValues-calls) are handled in one thread. OpenMI leaves the choice open to the

software developer whether the processing of a request is handled internally by starting multiple threads or 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.

3 org.OpenMI.Standard namespace

3.1 General description

3.1.1 Scope

The org.OpenMI.Standard namespace 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 and boolean values.

The usage of the org.OpenMI.Standard 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 proof that performance drops have become too significant, due to the lack of overload function calls.

Section 3.4 indicates which interfaces are mandatory to be OpenMI compliant.

3.1.2 Packages

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

3.1.3 Relationship to other namespaces

org.OpenMI.Standard is the independent interface specification. The OpenMI Software Development kit provides a default implementation of these interfaces through the org.OpenMI.Backbone package. Other packages within the org.OpenMI domain turn the standard in a useful software environment. They typically utilize the standard interfaces while some packages depend on the backbone implementation (see Figure 5).

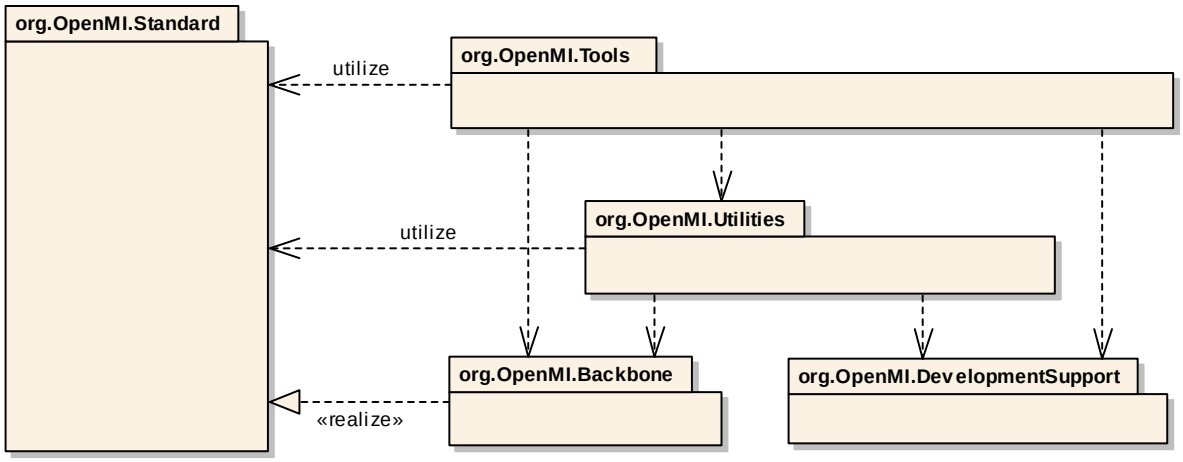


Figure 5 The namespaces of the OpenMI Software Development Kit

3.2 org.OpenMI.Standard: Static view

3.2.1 General convention

Some classes have a 1:n relation to other classes. Conceptually these relations are often lists or arrays. The general procedure to obtain each item is to ask for the total number of items via an {Item}Count method, and loop over the items with a Get{Item} type of call.

By convention the first item in a list starts with index 0, the last one with the count -1, where the count is returned by the Count method. .

3.2.2 Data definition interfaces

Correct interpretation of data being exchanged requires information on the semantics for each value passed over a link: what does it represent, where does it apply, when does it apply and how is it processed. Figure 6 illustrates these relations. The semantics of a value within a value set (as produced by a linkable component) is defined by the quantity (including the unit in which it is expressed), the elements (as defined in the element set) and the time (argument in the GetValues()-call). The combination of quantity and element set is contained in the link. DataOperations can be added to increase control on the delivery of data. Note that the ordering of the elements in the element set corresponds to the ordering of the values in the value set. In other words, the k-th value of the value set corresponds to the k-th element in the element set.

Each interface will be discussed in more detail in the remainder of this Section.

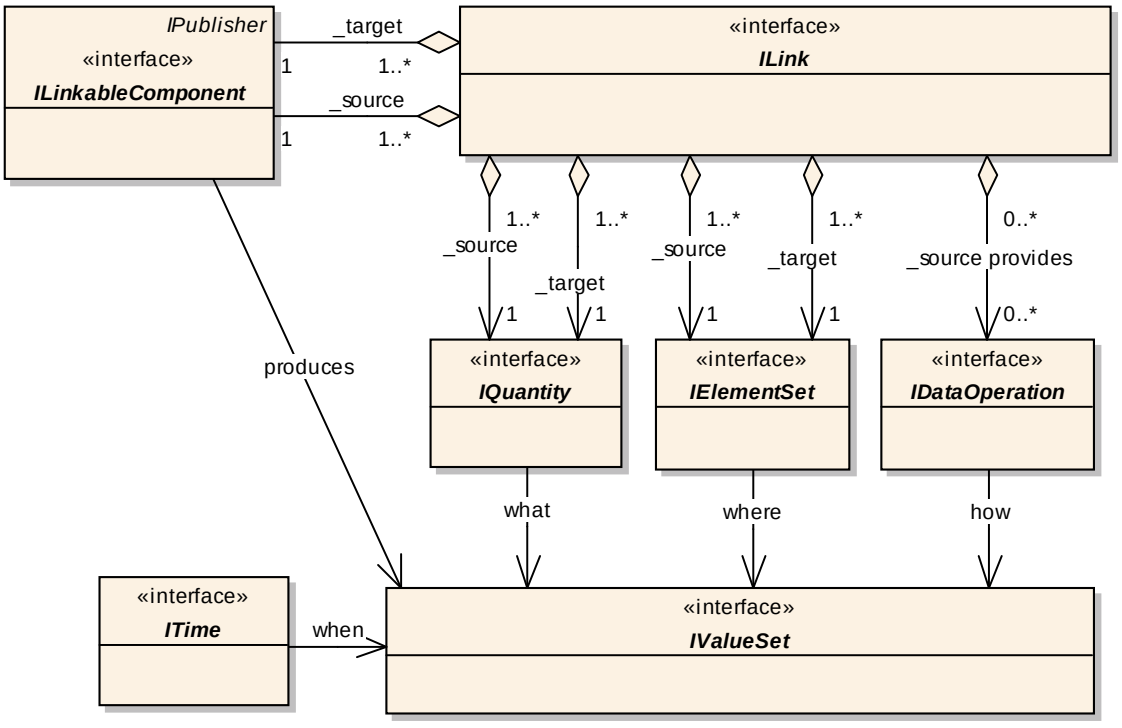


Figure 6 Relations between important data related interfaces

3.2.2.1 IElementSet

Data exchange between components in OpenMI is always related to one or more elements in a space, either geo-referenced or not. An element set in 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 coordinates) is also supported thus allowing exchange of arbitrary data that is not-related to space in any way.

The IElementSet interface (Figure 7) 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.

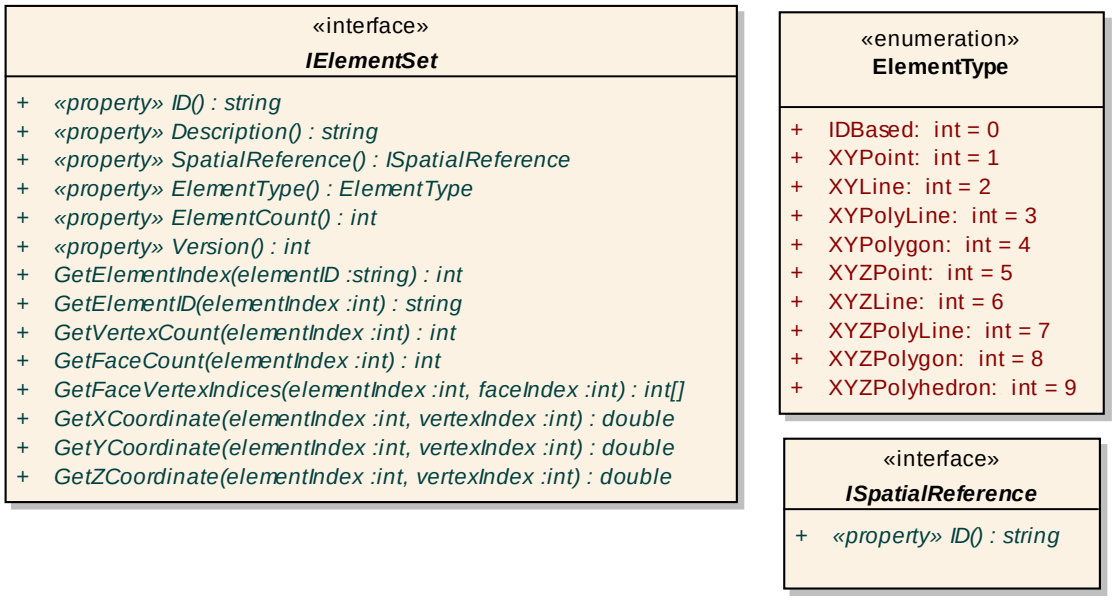


Figure 7 IElementSet and related interfaces

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 SpatialReference is not Null), coordinates (X,Y,Z) can be obtained for each vertex of an element. The ElementType is an enumeration, listed in Table 1. 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.

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 interface of a spatial reference (ISpatialReference) only contains a string ID. No other properties and methods have been defined as the OpenGIS SpatialReferenceSystem specification (OGC 2002) provides an excellent standard for this purpose.

The element set and the element are identified by a string ID. The ID is intended to be useful in terms of an end user. This is particularly useful for configuration as well as for providing specific logging information. However, the properties of an element (its vertices and/or faces) are obtained using an integer index (elementIndex, faceIndex and vertexIndex). This functionality is introduced as an element set basically is 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.

While most models encapsulate static element sets, some advanced models might contain dynamic elements (e.g. waves). A version number has been introduced to enable tracking of changes over time. If the version changes, the element set might need to be queried again during the computation process.

Table 1 OpenMI enumeration of ElementType

ElementType	Convention
IDBased	ID-based (string comparison)
XYPoint	geo-referenced point in the horizontal (XY)-plane
XYLine	geo-referenced line-segment connecting two vertices in the horizontal (XY)-plane. The begin- and end-vertex indicate the direction of any fluxes.
XYPolyLine	geo-referenced polyline connecting at least two vertices in the horizontal (XY)-plane. The begin- and end-vertex indicate the direction of any fluxes. Open entity with begin- and end-vertex not being identical.
XYPolygon	geo-referenced polygons in the horizontal (XY)-plane. Vertices defined counter clockwise. Closed entity with one face, begin- and end-vertex being identical
XYZPoint	geo-referenced point in the 3-dimensional space (XYZ)
XYZLine	geo-referenced line-segment connecting two vertices in the 3-dimensional space (XYZ). The begin- and end-vertex indicate the direction of any fluxes.
XYZPolyLine	geo-referenced polyline connecting at least two vertices in the 3-dimensional space (XYZ). The begin- and end-vertex indicate the direction of any fluxes. Open entity with begin- and end-vertex not being identical.
XYZPolygon	geo-referenced polygons in the 3-dimensional space (XYZ). Vertices defined counter clockwise. Closed entity with one face, begin- and end-vertex being identical
XYZPolyhedron	geo-referenced polyhedron in the 3-dimensional space (XYZ). Closed volume/entity with at least four faces Vertices for each face defined counter clockwise

3.2.2.2 ITime

Time in OpenMI is defined either by a ITimeStamp interface or a ITimeSpan interface, both interfaces inherited from the abstract ITime interface (see Figure 8). A time stamp is a single (instantaneous) point in time whereas the time span is a period from a begin time to end time.

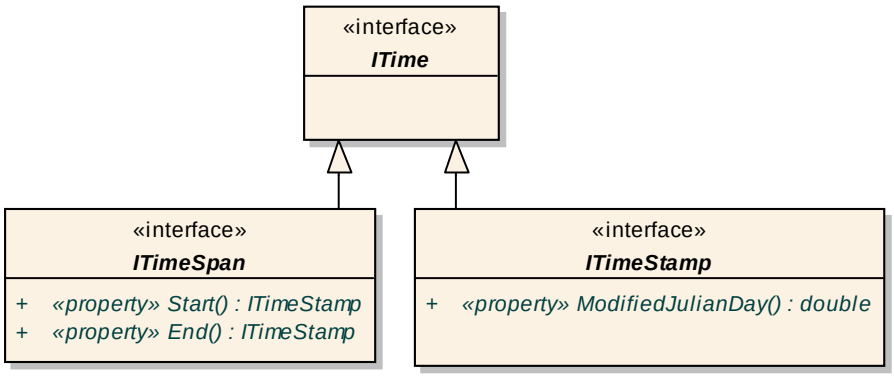


Figure 8 ITime and related interface

By convention, each of these times is represented by the Modified Julian Day. A Modified Julian day is the Julian date minus 2400000.5. A Modified Julian Day represents the number of days since midnight November 17, 1858 Universal Time on the Julian Calendar[6]. 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.

[6] see http://tycho.usno.navy.mil/systime.html

3.2.2.3 IQuantity

To enable proper linkage of data without a data dictionary, a meta data structure is needed which provides sufficient facilities to describe the semantics and enable automated checks on he semantics. Figure 9 illustrates the meta-data structure defined in OpenMI.

IQuantity defines the interface to indicate what the values represent, i.e. model variable, or model parameter to be exchanged. A quantity is identified by a string ID (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 whether the values are scalar or vector data.

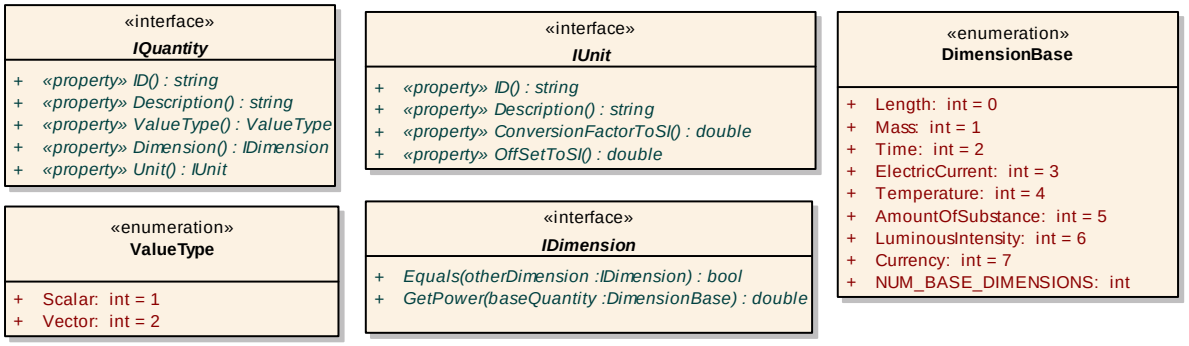


Figure 9 IQuantity and related interfaces

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 SI-system[9] , 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 unit m3/s has dimension Length^3Time^-1. Table 2 illustrates the base quantities and the associated SI units.

Table 2 Base units and dimension base in OpenMI (derived from SI)

Dimension base	SI base unit	symbol used
Length	meter	m
Mass	kilogram	kg

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

[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/)

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 difficulty when expressing in SI. In that case it is recommended to pay extra attention to the descriptive part of a unit, to ensure that the user which defines the link has proper understanding of the quantity.

3.2.2.4 IValueSet

To enable massive data exchange over a link, an interface structure has been defined that tightly matches to the typical (flat array) implementation of computational cores. This interface (IValueSet) represents an ordered list of values, where each value belongs to precisely one element in the corresponding element set (as indicated in the link definition). In other words, the i-th value in the value set corresponds to the i-th element in the element set.

Currently, two types of values are supported, namely scalar (IScalarSet) and vector (IVectorSet), see also ValueType[11]. While a scalar set is composed of doubles, the vector set is composed of vectors (IVector interface) with an X-, Y- and Z-component of the value. Figure 10 illustrates the associated static structures.

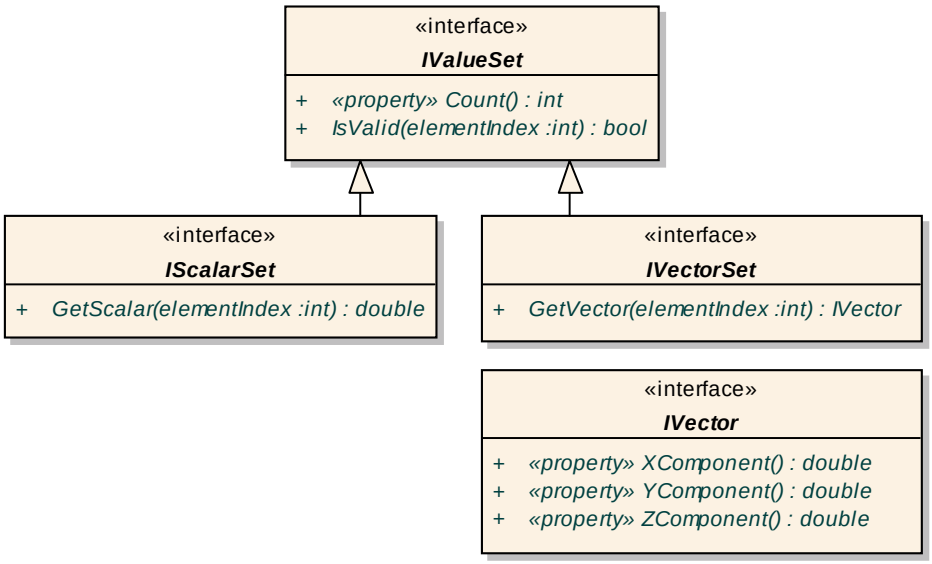


Figure 10 IValueSet and related interfaces

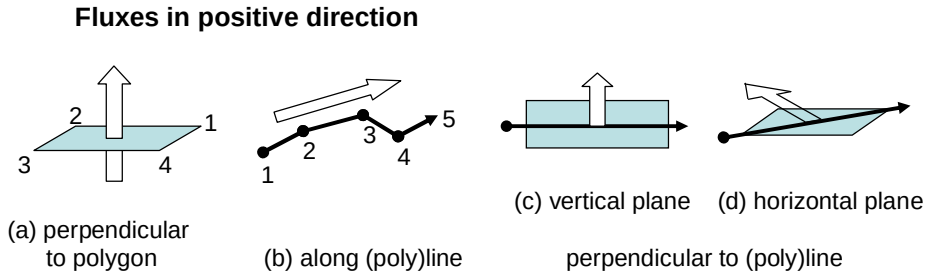
To prevent misunderstanding of positive and negative values, the following conventions are applied:

- values are positive if the matter leaves the source component and enters the target component (this also is the case for volumes).

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

[11] As soon as real world applications show the need for additional ValueTypes, this enumeration might be extended.

- The 'right hand rule' applies for fluxes through a plane or polygon, [12]
- The direction of fluxes along a (poly)line is defined positive from the begin- to the end node.
- The right hand rule applies for fluxes perpendicular to a (poly)line[13]



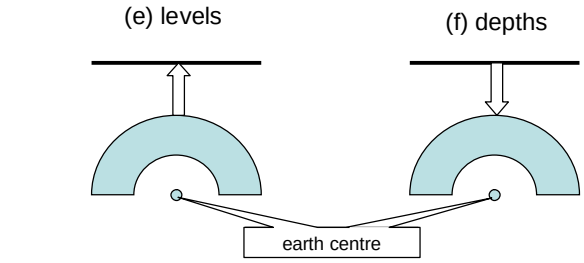


Figure 11 Illustration of directions to interpret positive values of fluxes, levels and depths

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

3.2.2.5 IDataOperation

Many situations occur where the raw data available at the source component does not match the request in location and time of the target component. 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, e.g. by averaging, by accumulating, taking minimum or max. etc.

By convention, the source component has the responsibility to perform data operations, using the intelligence as incorporated by the developer. Which data operations are available depends on the developer. The IDataOperation interface is defined for this purpose (see Figure 12).

[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 wrest clockwise. The thumb will point in the positive direction perpendicular to the (poly)line.

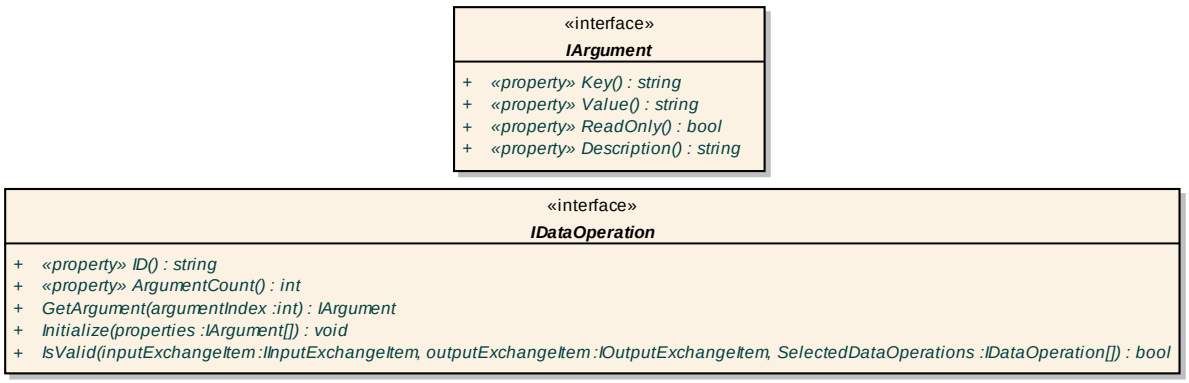


Figure 12 The IDataOperation and IArgument interfaces

Each data operation may have a number of arguments to manipulate the behaviour of the data operation. The ReadOnly property of an argument indicates whether the value may be changed, e.g. in a user interface. Note that the IArgument interface is also applied at other places, sometimes having fixed values.

Please note that the Value-property of IArgument is the only property of all interfaces which can be 'set'. All other properties of all other interfaces only accommodate a 'get' function (see API-spec)..

A source component may execute several data operations, e.g. a spatial data operation and a temporal one. However not all combinations are allowed. To enable checks on those combinations without the need to define validation rules, the IsValid()-method has been introduced. This method can test if the data operation is valid for this input/output combination given the combination with other data operations that already have been selected. The latter accommodates a component to test whether a time series accumulation is well perceived in the change of dimension (from time dependent to a time independent dimension).

The Initialize()-method has been introduced to feed the providing component (who executes the data operation) with the selected arguments.

Data operations may cover various aspects, e.g.

- temporal data operations
  e.g. for time stamps: interpolation and extrapolation (linear, quadratic, by regression function)
  e.g. for time spans: averaging, aggregate, accumulation, moving average, minimum value, maximum value, first value, last value, all values
- spatial data operations
  e.g. interpolations :krigging, inverse distance, all kinds of averaging, maximum value, minimum value, etc.
- miscellaneous data operations
  e.g. perform vertical shift, etc

3.2.3 Meta data interfaces to express what data can be exchanged

To set up a chain of linked components, information is needed on the existence of components and the data they can exchange. Each component that can become part of the component chain has to provide meta-data which describes the input and output data that potentially can be exchanged, including the data operations it can provide. While in theory, this information may be given in a user manual, in practice this information should be digitally 'open' to the outside world via a generic interface.

3.2.3.1 IExchangeItem

Typically a linkable component exchanges 'a quantity on an element set', i.e. always as a combination. Although the base interfaces are known, an additional interface (IExchangeItem) has been introduced to stress the combination of the two. For a linkable component, an exchange item either is an input or an output. As discussed in Section 3.2.2.5, the source component can perform data operations to tailor the data to the needs of the target component. The person setting up the links selects the data operations desired, based on the list of operations offered by the source component. Since the IExchangeItem interface informs this person about the available output data available (i.e. quantity on an element set), this seems a logical place to describe the available data operations. The distinction between input (quantity on an element set) and output (quantity on an element set supported by additional data operations) result in two interfaces, namely IInputExchangeItem and IOutputExchangeItem. Figure 13 illustrates those interfaces in a class diagram.

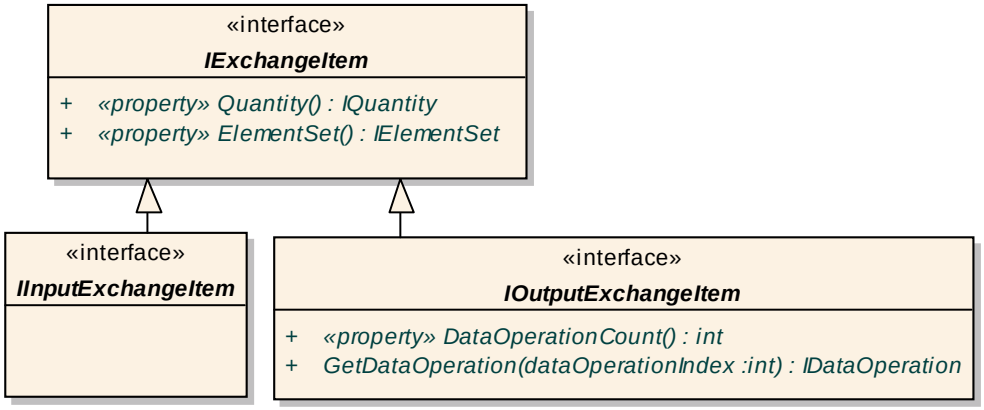


Figure 13 IExchangeItem interfaces

The descriptive IExchangeItem interfaces have been incorporated in the main linkable component interface of OpenMI, since most OpenMI-components exchange data in some way. If a component has no ExchangeItems, it returns a Null when asked for its ExchangeItems.

Section 3.3.2 will discuss the dynamic behaviour of populating ExchangeItems in case they are not known a-priori.

While these meta-data interfaces are embedded in the main linkable component interface, they might sometimes be referred to as being the 'exchange model' of a linkable component.

3.2.4 Interface to define the link

3.2.4.1 ILink

The ILink interface (Figure 14) captures all information about the link between two linkable components. Every link represents one quantity to be exchanged from the SourceComponent to a TargetElementSet on the TargetComponent. The SourceElementSet is included to enable queries on the data source. In the GetValues() call the link id and a time-object is passed (see Section 3.2.5.1). If data operations are needed, e.g. to reduce the computed values in a time span into one value passed back, they can be obtained from the link by the GetDataOperation() method for the number of operations specified. By convention, the index order of the data operations determines the order in which data operations are handled by the linkable component

The link object is a semi-static object, i.e. the element set reference and quantity information for one link object will remain constant after construction. Note however, that a dynamic element set keeps the same reference but may change its content over time. Run-time changes in data operation settings or new quantities being attached must be dealt with by removing and adding links.

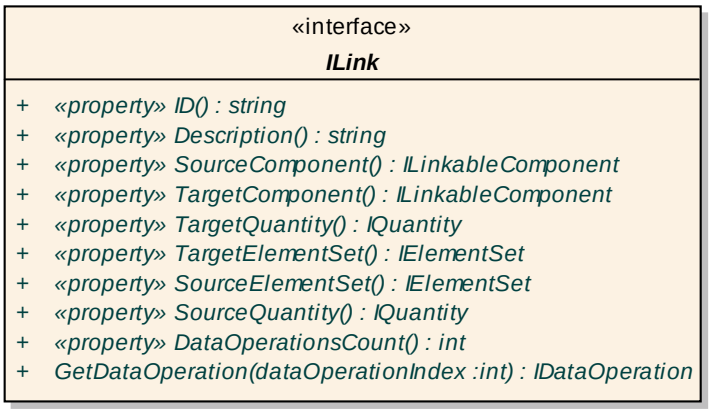


Figure 14 ILink and associated interfaces

3.2.5 Interfaces for component access

3.2.5.1 ILinkableComponent

OpenMI-components need to have one interface which defines the generic access to the component. As linking components is the most important functionality of OpenMI, the ILinkableComponent interface, the interface enabling this linkage and data exchange, is the entry point of an OpenMI-component and thus the key interface of OpenMI. Figure 15 displays the interface.

Functionality to initialize the computational core:

- <method> Initialize (arguments: IArgument[]): void
  After instantiation of the LinkableComponent-object by its constructor, the Initialize method is called to populate the object with specific information. After this method-call, the LinkableComponent can be inspected for its meta data. Code developers may choose (but are not forced) to instantiate the computational core or database and populate it with to site-specific data. the Initialize() method allows instantiation of a computational core without having specific knowledge about the component

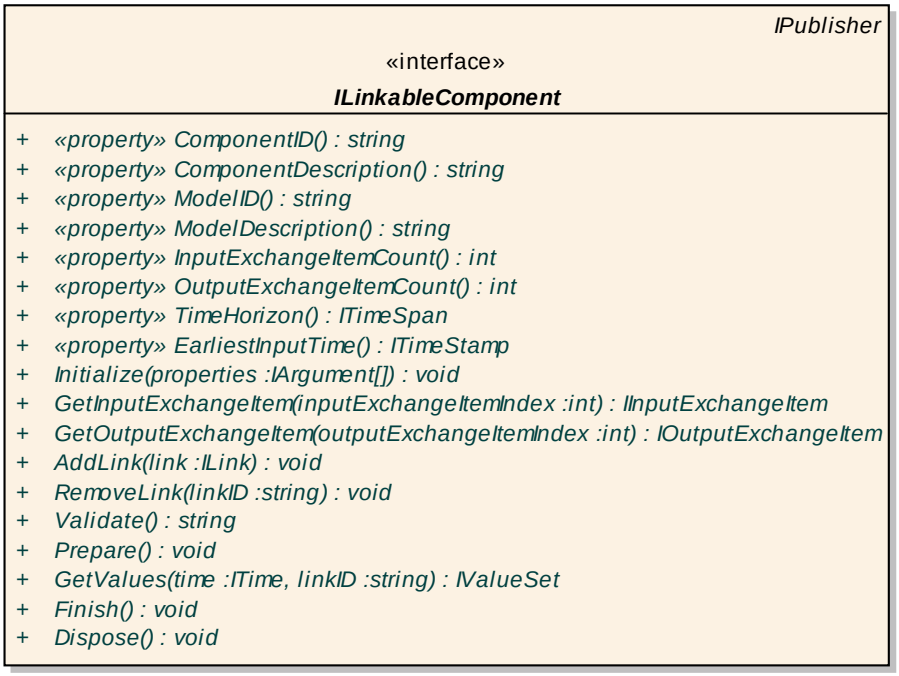


Figure 15 ILinkableComponent interface

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

- <property {get}> ComponentID() : string
  Logical identification of the software unit (typically not populated with data), e.g SOBEK-CF, Mike11-HD, Isis Flow etc.
- <property {get}> ComponentDescription() : string
  Description of the software unit (typically not populated with data), may include version numbering etc.
- <property {get}> ModelID() : string
  Logical identification of the content in terms of model/schematization/application area of the software unit once populated with data
- <property {get}> ModelDescription() : string
  Description of the content in terms of model/schematization/application area of the software unit once populated with data (e.g. River Rhine)
- <property {get}> InputExchangeItemCount() : int
  The number of different data sets that can be accepted as input (InputExchangeItems)
- <method> GetInputExchangeItem(inputExchangeItemIndex: int): IInputExchangeItem
  Method to obtain data on the potential input, by looping over all InputExchangeItems.
- <property {get}> InputExchangeItemCount() : int
  The number of different data sets that can be provided as output (InputExchangeItems)
- <method> GetOutputExchangeItem(outputExchangeItemIndex: int): IOutputExchangeItem
  Method to obtain data on the potential output, by looping over all OutputExchangeItems.
- <property {get}> TimeHorizon: ITimeSpan
  The time span over which a component can provide data (either discrete or continuous). If a component does not know time at all it returns a Null.

Note that the IDiscreteTimes interface (Section 3.2.5.3) is an optional interface to provide more detailed information on the temporal discretization of available data.

Functionality to establish and validate the links:

- <method> AddLink(link: ILink): void
  Every link must be added to its SourceComponent and its TargetComponent. By informing both the source and target linkable component about their link, they can prepare themselves for efficient data exchange at run-time.
- <method> RemoveLink(linkID : string): void:
  Removes a link with the given ID.
- <method> Validate(): string
  Method typically called by the GUI at configuration time when links have been added. This call enables validation of the current state of the model and its links, 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). Returns an empty string if the component is in a valid state otherwise if will return a message describing the problem. Displaying the message in the User Interface will enable the user to correct the error (e.g. inconsistent quantities connected).

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

- <method> Prepare (): void
  This method is invoked just before the first GetValues()-call. It enables the component to prepare itself, e.g. instantiate the engine (if needed), setup the network connections, do a model validation, prepare internal buffers and data mapping matrices, etc. 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.
- <method> GetValues(time : ITime, linkID : string): IValueSet:
  This method is the core method of OpenMI. It is applied to retrieve values from the (linkable) component over a certain link. A GetValues()-call incorporates either a time stamp or a time span. The component will respond to this call values by providing all values on the requested elements for this specific time instance.
- <property {get}> EarliestInputTime() : ITimeStamp
  This property enables an outside (source) component to detect the earliest time stamp at which a (target) component requires input data. The (source) component can use this information to manage its internal buffers (if any) or to prepare itself in another way.
- <method> Finish(): void
  The finish method is intended to be used for closing files. This means that a GUI can e.g. inspect some results before the components are disposed.

Functionality to dispose the component:

- <method> Dispose(): void
  Dispose is intended for de-allocation of memory (from unmanaged code). It is not required that LinkableComponents can be initialised after invocation of Dispose. Dispose typically is called when deployment is completed.

The ILinkableComponent interface is inherited from IPublisher interface. Any linkable component thus must implement this interface (See 3.2.5.4 for more details). If a linkable component cannot provide events it should indicate this by returning ‘zero’ when asked for the number of events it can publish.

Section 3.3 provides more insight in the call sequence to be adopted.

3.2.5.2 IManageState

Some process interactions require feedback loops and even iterations over time. To support this type of functionality, the IManageState interface (Figure 16) has been defined, containing three methods. By calling the KeepCurrentState() method, a component is requested to store its internal state and return string identifier that enables future restoring. The state data itself is not of interest to OpenMI, so the component developer can decide himself what needs to be stored and how (e.g. in memory, 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).

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

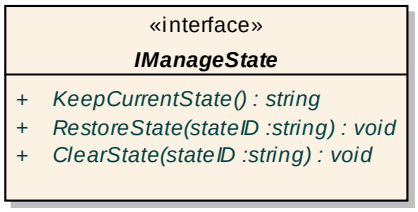


Figure 16 IManageState interface

3.2.5.3 IDiscreteTimes

Within and outside modelling exercises, many situations occur where 'raw' data is desired at the (discrete) time stamp as it is available in the source component. A typical example is the comparison of computation results with monitoring data, or a computational core that wants to adhere to the time stepping of its data source. To keep the values fixed to the discrete times as they are available in the source component, the IDiscreteTimes interface has been defined (see Figure 17). This interface can provide a list of time stamps for which values of a quantity on an element set are available.

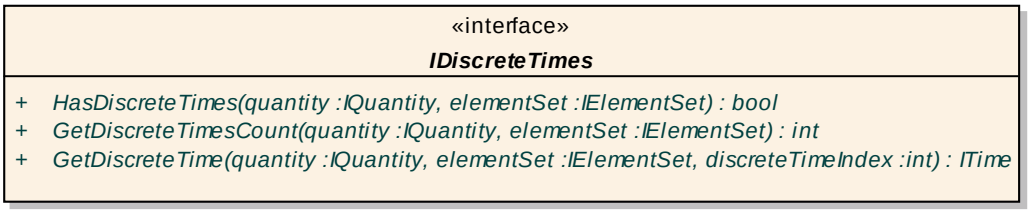


Figure 17 IDiscreteTimes interface

3.2.5.4 IPublisher

A publish-subscribe pattern is introduced in combination with a high-level, property-based event definition to enable troubleshooting and to facilitate development of monitoring and visualisation tools. The IPublisher interface (Figure 18) defines that a component should be able to 'publish' to others which events it can throw. By definition, the IPublisher interface is inherited by the ILinkableComponent interface . If a linkable component does not publish events, it returns a '0' when asked for its PublishedEventTypesCount.

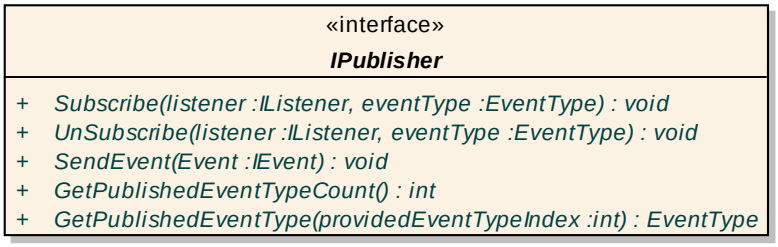


Figure 18 IPublisher interface

3.2.5.5 IListener

The IListener interface (Figure 19) enables components to catch events and act upon them. Events are only published to listeners that have subscribed themselves at a publisher. Linkable components can, but are not obliged to implement the IListener interface.

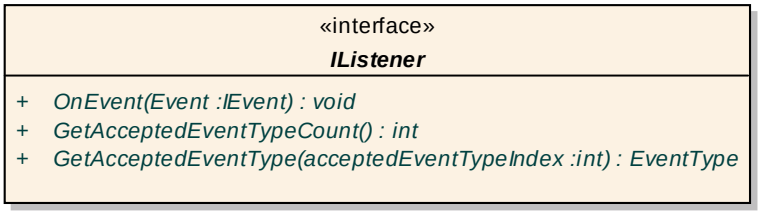


Figure 19 IListener interface

3.2.5.6 IEvent

By convention, a linkable component is obliged to throw an exception when an internal irresolvable error occurs (see Section 3.2.5.7). In all other error and warning situations an event is thrown. OpenMI has standardized a high level interface for event (IEvent, see Figure 20) with some default properties to be incorporated in any event. An enumeration of EventTypes has been defined to enable handling of common events by various types of monitoring and support tools.

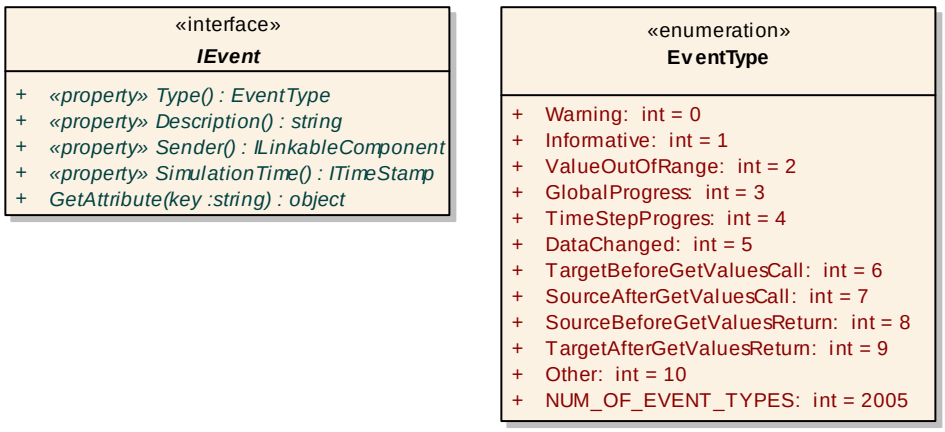


Figure 20 IEvent interface

Note that events can also be grabbed and held to interrupt a computation. As this is the only mechanism, in addition to stop firing GetValues()-calls at the highest level, to interrupt computations, it is highly recommended to throw events on a regular basis. Section 3.3.8 will explain the dynamics of interrupting, the conventions to be applied as well as the logic behind the interrupting event. Table 3 provides an overview of the EventType enumeration of OpenMI.

Table 3 EventType enumeration

EventType	Description
Warning	general warning message
Informative	general information message
ValueOutOfRange	indicates value out of range message
GlobalProgress	indicates progress as % of global time horizon
TimeStepProgress	indicates progress as % of time step
DataChanged	indicates changes of (exchangeable) data within a component
TargetBeforeGetValuesCall	enables tracing call stacks, send by target component before it invokes a GetValues()- call
SourceAfterGetValuesCall	trace event, send by source component after it receives a GetValues()- call
SourceBeforeGetValuesReturn	trace event, send by source component before it returns the result from a GetValues()- call
TargetAfterGetValuesReturn	trace event, send by target component after it receives the result from a GetValues()- call
Other	all other types of events

3.2.5.7 Exceptions

By convention a linkable component has to throw an exception if an internally irrecoverable error occurs. This 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 (e.g. if required with user interference).

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

In the above mentioned sections, all interfaces of the OpenMI.Standard have been discussed. OpenMI compliant components need to implement those interfaces. However, once implemented, one 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 Annex I-B contains the XSD.

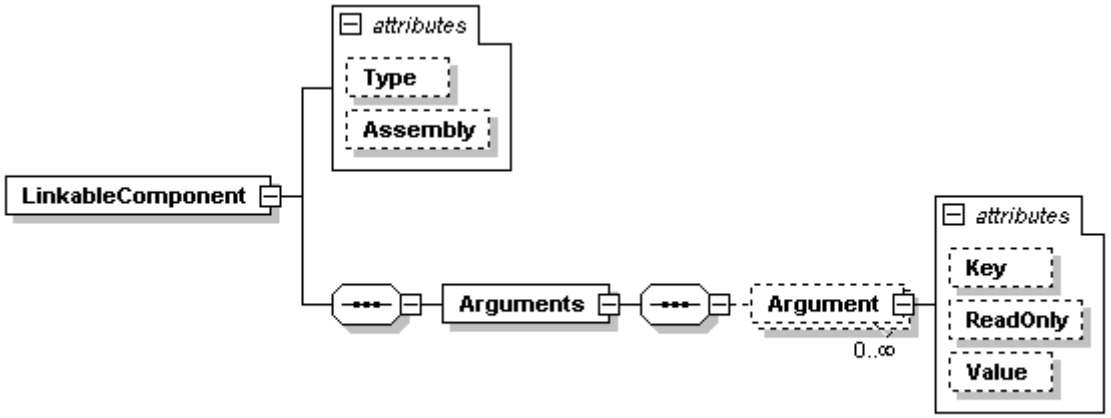


Figure 21 Graphical view of the OMI-file structure

```
<?xml version="1.0"?>
<LinkableComponent Type="wlDelft.OpenMI.WLLinkableComponent" Assembly="wlDelft.OpenMI, Version=1.4.0.0,
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

3.3.1 Phases in utilizing the linkable component interface

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

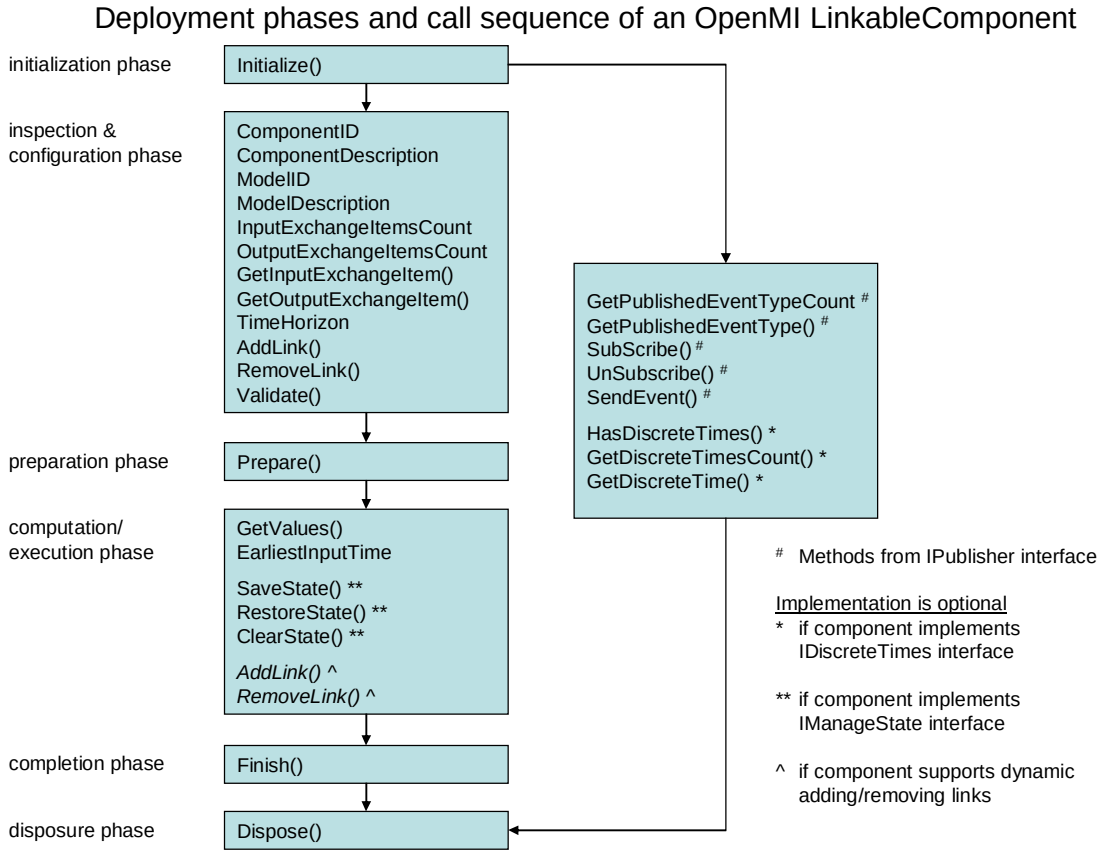


Figure 23 Deployment phases and associated call sequence of OpenMI Linkable Components

The various phases will be discussed briefly, except for the computation phase. The dynamic behaviour of the computation phase will be discussed in more detail with a use case involving three linkable components, namely a rainfall-runoff mode, a river model and a groundwater model.

The chapter will be completed 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

This phase ends by the situation where 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 will be 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

Dependent on the setting this phase might be very static and straightforward or very dynamic. The end situation of this phase is the following: The links have been defined and added and the component has validated its status[14].

The following steps can be identified:

3. Request for the exchange items
   Ask the LinkableComponents for its OutputExchangeItems and InputExchangeItems; using the methods {item}Count/Get[item]
4. Create and add links
   Instantiate the link-objects
   populate them
   check the validity of its data-operation(s) in combination with other selected data operations; using IDataOperation.IsValid, and
   add the links to the components; using method ILinkableComponent.AddLink
5. Validation
   Validate the status of the components and their links; using the ILinkableComponent.Validate method

Hard coded systems will not require step 3. All use cases require step 4, while step 5 is highly recommended. Use cases having linkable components with a-priori knowledge on exchange items can easily respond to step 3. Use cases 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

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

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

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

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

During this phase, the heavy work load will be executed and associated data transfer will get bulky.

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

In addition, the call sequence of advanced linkages based on state management, e.g. iteration, will be discussed as well.

3.3.5.1 Unidirectional data transfer (one way)

Figure 24 illustrates the calling sequence between two linkable components in 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 uni-directional (from RR model to RiverModel).

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

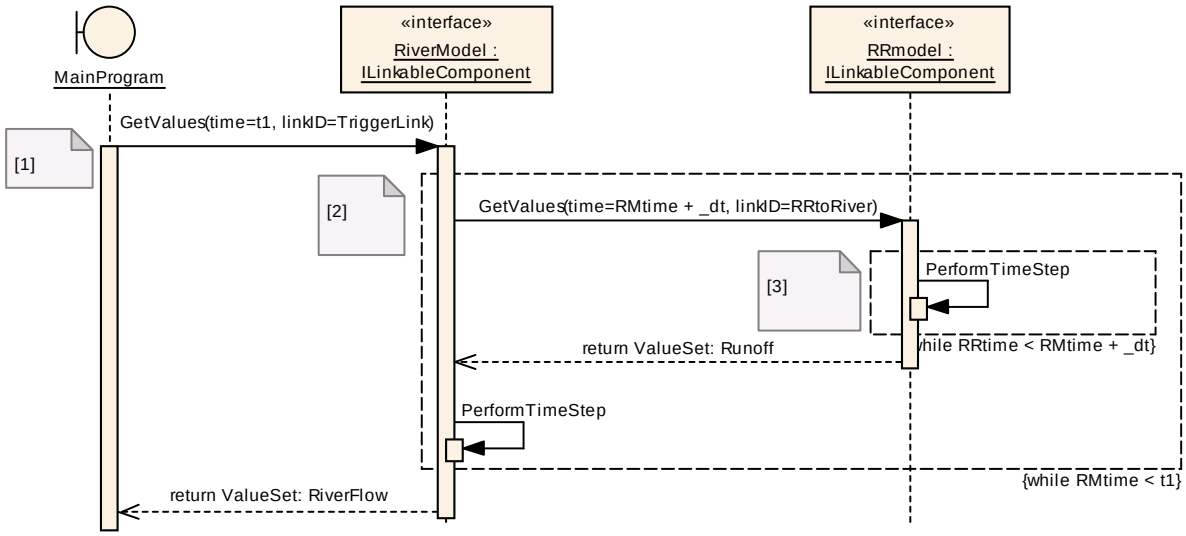


Figure 24 Unidirectional data transfer (sequence diagram)

1. Invocation
   The computation is started by triggering the last component in the chain. The RiverModel component is invoked, where the time argument defines the time for which results are expected from the RiverModel.
2. Computation RiverModel
   the RiverModel will evaluate whether its internal time (RMtime) is before or after the requested time (t1). While RMtime is less than t1, the RiverModel will perform time steps. The RiverModel will, before each time step, retrieve the runoff from the RR model by invoking the GetValues() method in the RR model.
3. Computation RR model
   The RR model will perform as many time steps as necessary in order to calculate the requested value.
   Note that the two models do not need to have matching time steps. It is the responsibility of the

delivering model to do any interpolation (or extrapolation) required in order to return a value that represents the time argument in the GetValues() call.

3.3.5.2 Bidirectional data transfer (two way)

Figure 25 illustrates the calling sequence between two linkable components in 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 GroundWater model providing a lateral inflow to the RiverModel. The two components have been instantiated and prepared in Section 3.3.3 and 3.3.4.

In the diagram, it is assumed that both the RiverModel and GroundWaterModel have not yet the requested data available, but have sufficient information to compute the data requested. For illustration purposes, the time step of the GroundWaterModel is has been set to two times the time step of the RiverModel. The iteration of the RiverModel (2 time steps) has been written out.

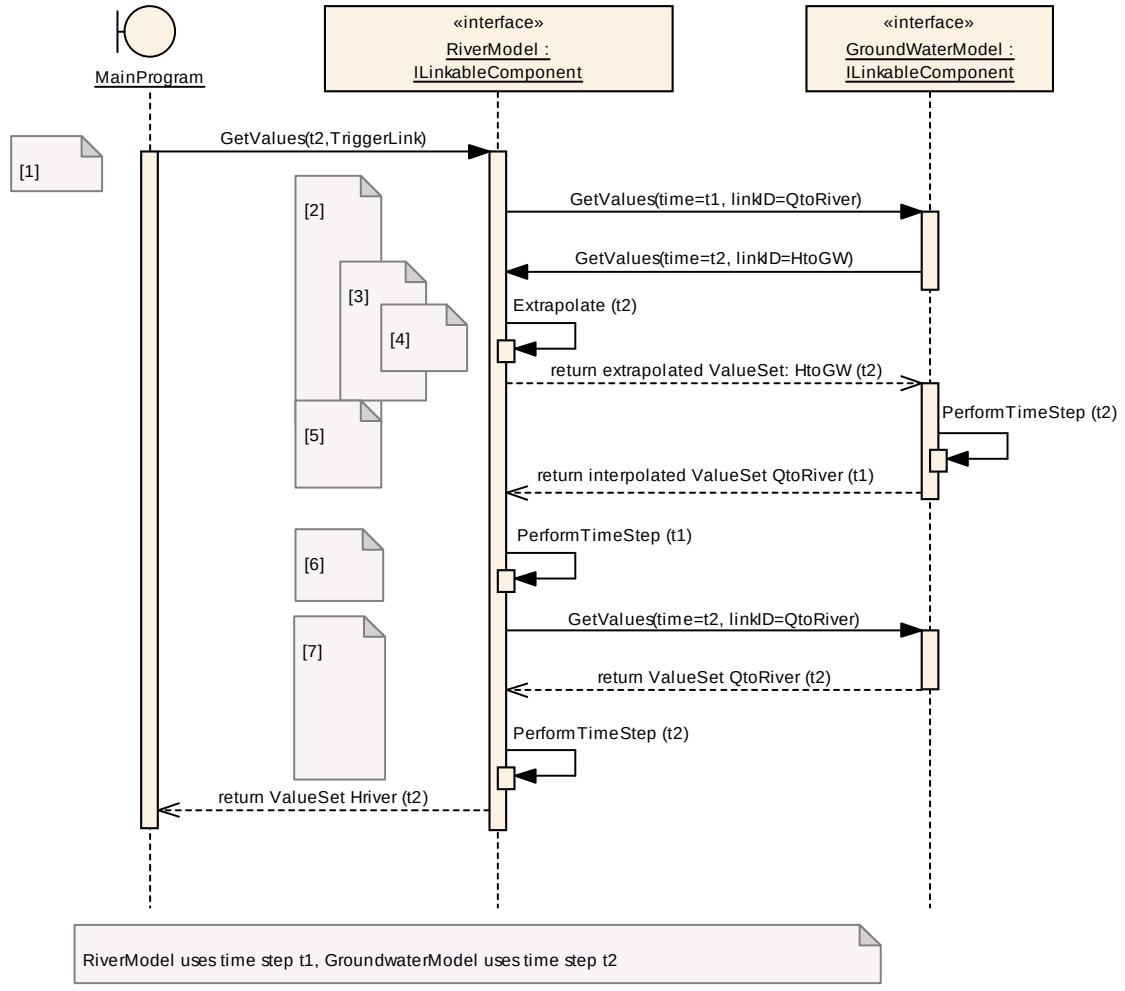


Figure 25 Bidirectional data transfer (sequence diagram)

1. Invocation
   The computation is started by using the TriggerLink to trigger the last component in the chain. The RiverModel component is invoked, where the time argument defines the time for which results are expected from the RiverModel.

2. Computation RiverModel
   (the RiverModel will evaluate whether its internal time is before or after the requested time (t2)). The RiverModel will, before each time step, retrieve the lateral inflow from the GroundWaterModel by invoking the GetValues() method in the GroundWaterModel.
3. Computation GroundWaterModel
   (the GroundWaterModel will evaluate whether its internal time is before or after the requested time (t1)). 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.
4. Extrapolation RiverModel
   After receiving the GetValues() call for t2, the RiverModel determines that it already is computing 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.
5. Continue computation groundwater level
   Utilizing the data returned, the GroundWaterModel is able to compute the lateral flow for t2. Based on the outcome it can return an interpolated value for t1.
6. Continue computation river model, iteration step t1
   The RiverModel has received all requested data for t1 and can compute do its iteration step over t1.
7. Continue computation river model, iteration step t2
   For the second iteration step of the RiverModel (t1 to t2), the model 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 time step t2 and return the requested data to the main program.
   Note that the software developer of the GroundWaterModel may choose for an implementation that does recomputed the lateral flow based on an update and hopefully more accurate, water level extrapolation for t2.

3.3.5.3 Managing states using IManageState

The IManageState interface has been introduced to accommodate the development of advanced controllers for iteration and optimization purposes. Figure 26 illustrates how the IManageState interface can be utilized to enable iteration between a GroundWaterModel and a RiverModel. Both the GroundWaterModel and the RiverModel implement the IManageState interface. A separate linkable component has been introduced to supervise the iteration, i.e. the IterationController. This controller holds internal links to the GroundWaterModel and the RiverModel, while it hides the two models to the ‘outside’ world (i.e. the main program).

The procedure can be as follows:

1. Invocation
   The main program calls the IterationController to provide the surface water level. The IterationController starts it iteration by saving the state of the models at the beginning of the time step.
2. Initial guess
   The IterationController than produces its first guess of the flux between the RiverModel and GroundWaterModel
3. Restore states
   The iteration is entered with restoring the state of the GroundWaterModel and the RiverModel to their begin situation.

4. Groundwater level requested
   The GroundWaterModel is requested for its groundwater level. To answer this question, the GroundWaterModel requires the flux from the iteration controller. 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 requested 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.

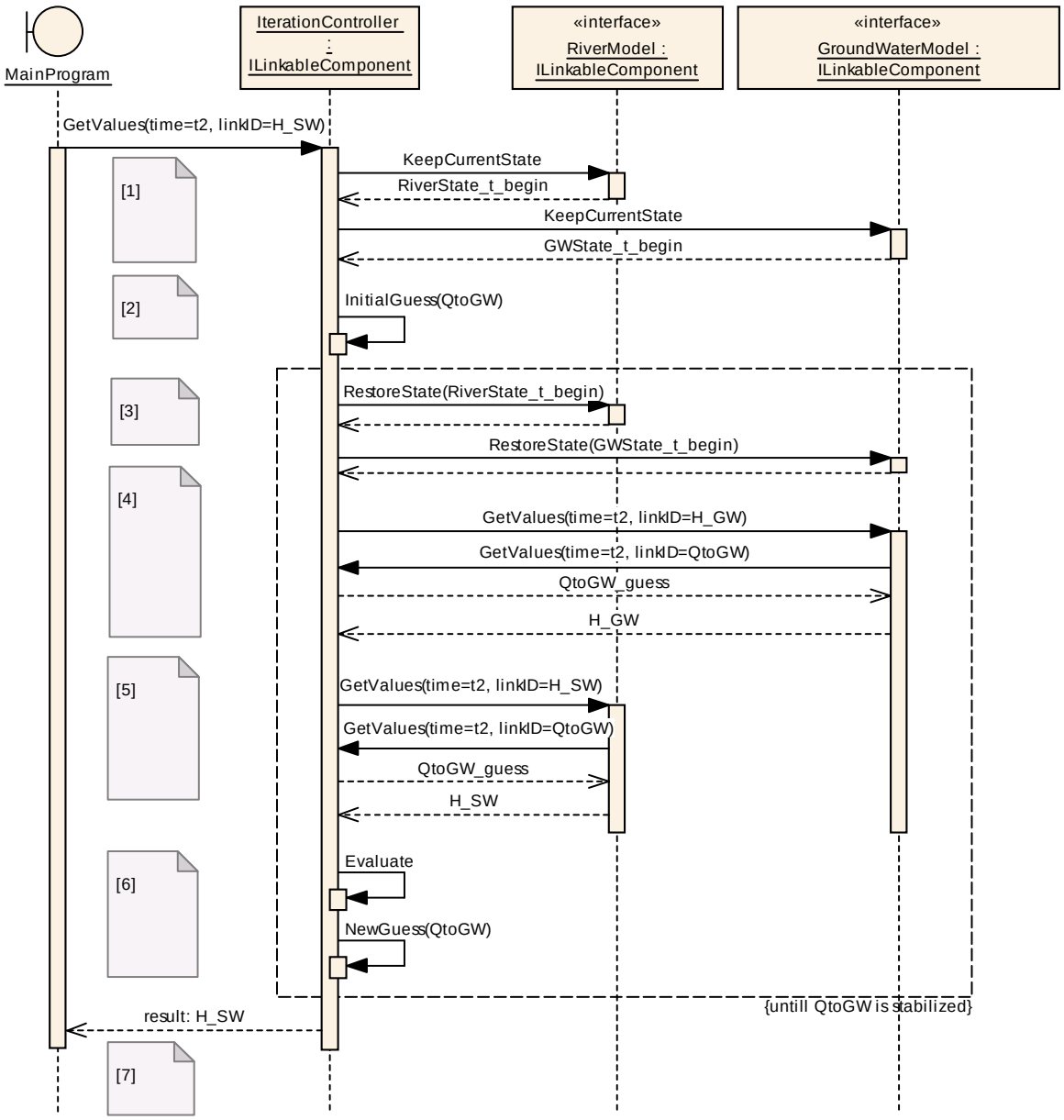


Figure 26 Illustration how IManageState can be used for iterations (sequence diagram)

3.3.5.4 Managing internal buffers using EarliestInputTime

Within a model combination involving several components, it may happen that several components ask one component for data at the same time stamp. To prevent the need of re-computation for each request, such 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 which still is needed, the EarliestInputTime-property has been implemented.

The example of Figure 27 is an extension of the iteration between the GroundWaterModel and the RiverModel. It is an extension of Note [5], since the RiverModel requires inflows from the RRmodel to compute its water level. In this example, the RR model investigates whether its buffer can be cleared.

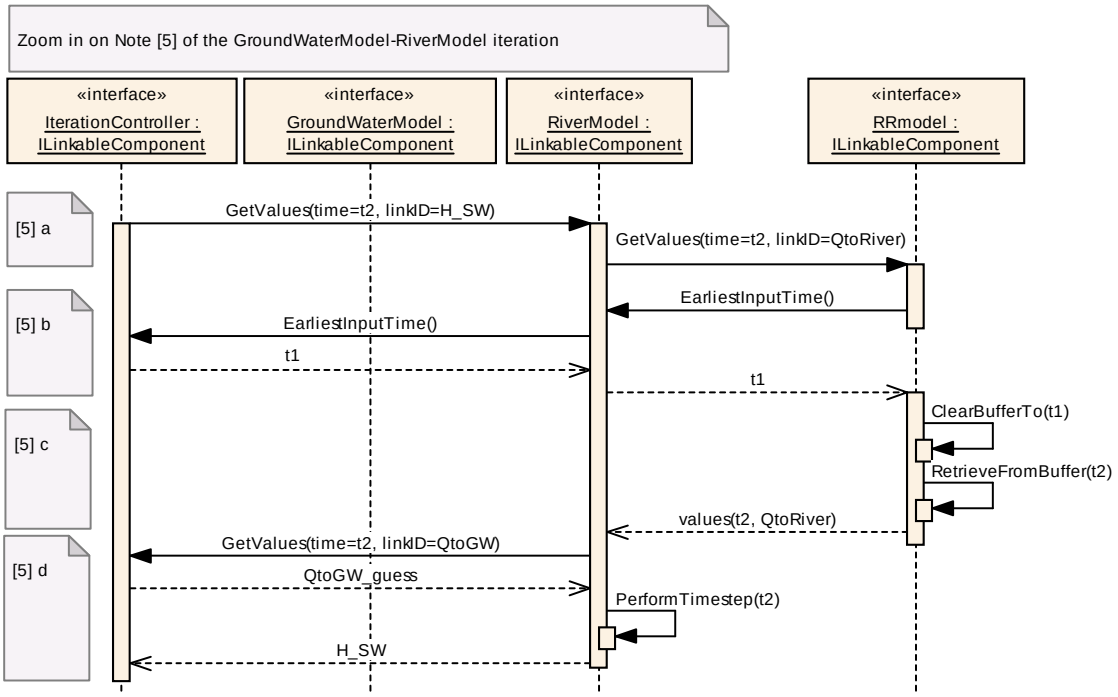


Figure 27 Using EarliestInputTime to clear internal buffers (sequence diagram)

5. a) As part of the iteration, the IterationController asks the RiverModel to provide the surface water level. In order to compute this value, the RiverModel requires inflow data from the RRmodel. In this case, a direct link had been defined between the RiverModel and the RRmodel.
   b) The RRmodel already has computed a number of time steps and has put them in a buffer. To keep the size of the buffer within range, the RRmodel first wants to clear the buffer as far as possible. Therefore, it asks its target component(s), i.e. the RiverModel for the earliest input time that will be requested by this model.
   To answer this question, the RiverModel needs to interrogate its target component(s). Therefore it asks the IterationController (its target component) for the same information. The IterationController is aware for what period the iteration takes place, so its passes back the begin time of the iteration. The RiverModel compares the answers from its target components with its own knowledge and returns the earliest time in its list.

c) Based on this information the RRmodel clears its buffer as far as possible, retrieves the requested data from its buffer and returns this to its target component, the RiverModel.
d) The RiverModel now can proceed with the remaining steps of note 5. It retrieves the groundwater-river flux from the IterationController, computes its own time step and returns the requested values to the IterationController.

Please note that many use cases can be identified where components use internal buffers. In all those situations, providing components need to be able to interrogate their target components.

3.3.6 Phase V: Completion

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

3.3.7 Phase VI: Disposure

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

3.3.8 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 OpenMI is defined as single thread of synchronous GetValues() calls. As GetValues is the mechanism that starts the process and keeps it going, it has been decided that any more direct interruption mechanism should be incorporated in the same thread.

The event mechanism is applied for this purpose, using a listener (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 trace events (the ones surrounding the GetValues() call) are preferred mostly. 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.

Both mechanisms work similar. The ‘Pause’ variant has been illustrated in Figure 28. In the example, two linkable components are involved, i.e. a Rainfall Runoff model and a RiverModel.

1. The user passes the start event to the main program. The main program triggers the RiverModel by a GetValues() call. The RiverModel sends a trace event after it received the GetValues call. The component waits until it receives the thread back.
2. Once the thread is back, it performs a GetValues() call to the RR model. The RR model sends a trace event after it received the GetValues call. The component waits until it receives the thread back.
3. The compute performs a time stepping loop, sending a data changed-event after each internal time step. Before the data is returned a trace event is send to inform the listener that the values will be passed back

4. In the mean while, the end user requested the main program for a pause. The main program waits until it receives the next event from the computation thread, in this case a trace event, and holds the thread.
5. The end user has asked 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 trace event that it received the control from the GetValues() call stack. After the thread is returned by the main program, the RiverModel can compute its time step.
7. After computation the RiverModel send a data changed event. When the thread is returned, it sends a trace event informing that the values are passed back.
8. When the thread is returned in passes the value set back

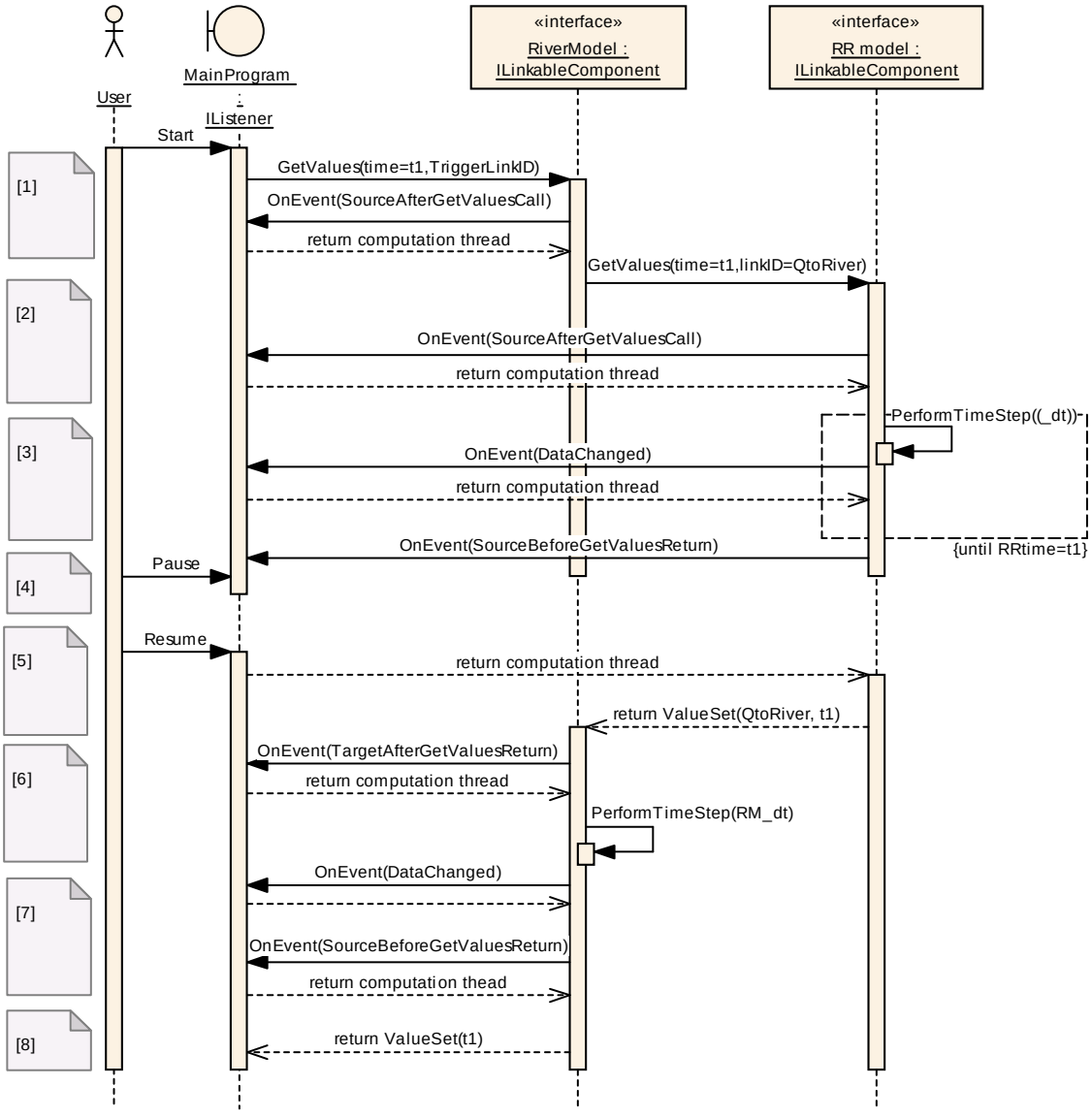


Figure 28 Pause and resume of a computation process (sequence diagram)

This example illustrates a situation where almost all event types (of Table 3) are applied. Whether this is a real case depends on the way the linkable component applies events. Although many messages pass along, the traffic of events does not take many resources as the events are 'lightweight'

information carriers. Clearly the response time is very fast, even in multiple component chains. Pausing 'the natural way' (a dying call stack) would result in a reaction of the system by the time the results of the RiverModel are returned.

Typically, the component having the task of the main program will register for many event types send by all linkable components in a computation chain.

Within OpenMI various kinds of event types are defined for usage by any supporting environment or tools (see Table 3, Section 3.2.5.6). Although usage of events is not mandatory, they are highly recommended to create user friendly systems for both end-users (which want quick responses) as well as developers (which want debug facilities).

3.3.9 Miscellaneous issues

3.3.9.1 Using events for logging and visualization

The event mechanism is also applied to enable on-line visualization. Figure 29 illustrates how a DataChanged-event can be used to update a visualization, by retrieving the computed data using a GetValues() call. In a similar way, other listeners can be implemented for logging all kinds of data traffic and progress.

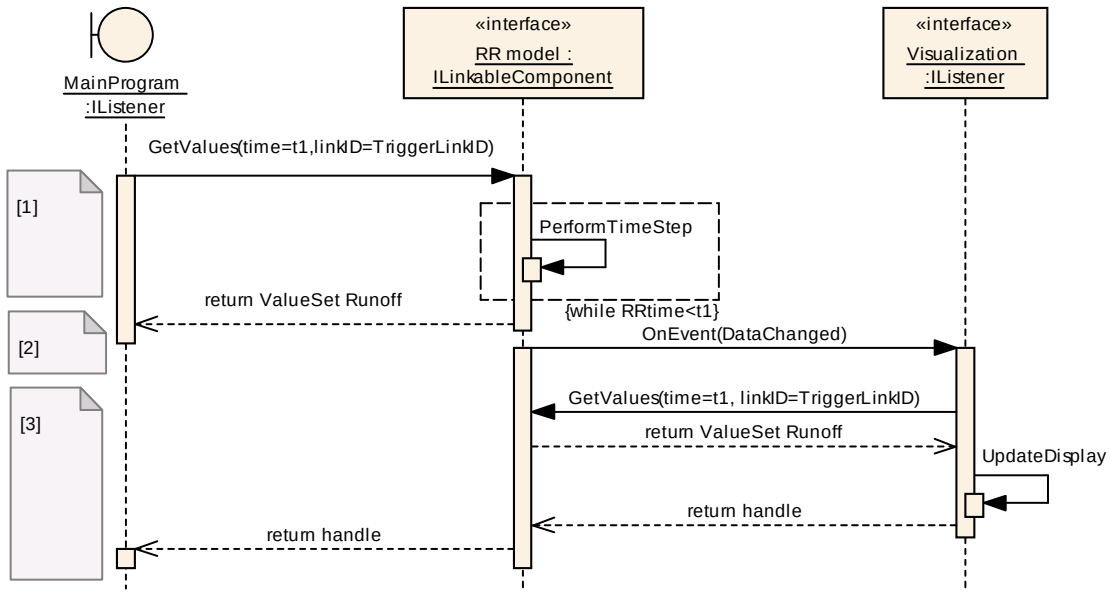


Figure 29 On-line visualization using a DataChanged-event (sequence diagram)

1. Acting upon a GetValues() request, the RR model starts computing until the requested time.
2. Once the time stamp is reached, the data is returned to the requesting component. Simultaneously, listeners that have subscribed to the DataChanged-event type are notified that data has changed.
3. The Visualization component is such listener and decides to retrieve the data and update its display. All handles are returned to the GUI.

3.3.9.2 Exceptions

By convention, a Linkable Component is obliged to throw an Exception when an internal irresolvable error occurs. This Exception is caught by the component that called the linkable component. If this component cannot react properly, it has to pass the exception until the level that a component can handle the exception. Figure 30 provides a simple illustration where the only computing component generates an exception. The main program cannot handle it properly and asks the user for a manual intervention.

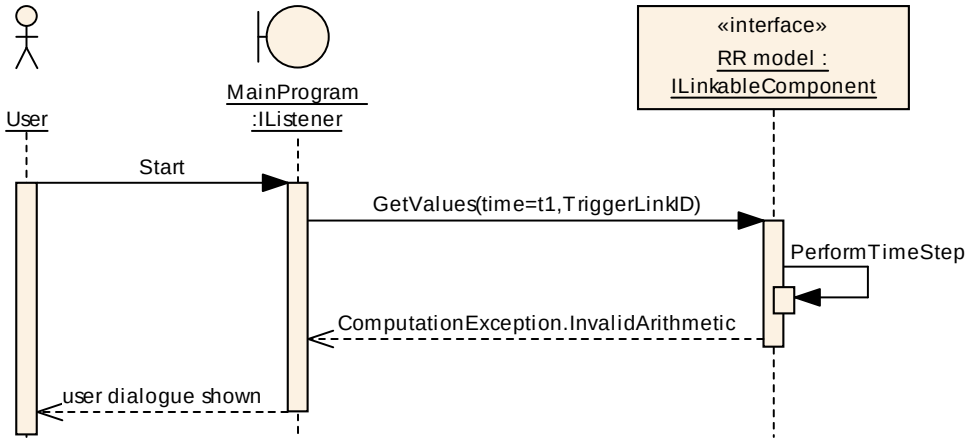


Figure 30 Sequence diagram: exception

3.3.9.3 Obtaining listed items

Within the OpenMI interface definition, many classes contain other classes in a list wise sense. Figure 31 illustrates the common method to obtain listed items. Note, by convention, the first item in the list starts with index 0.

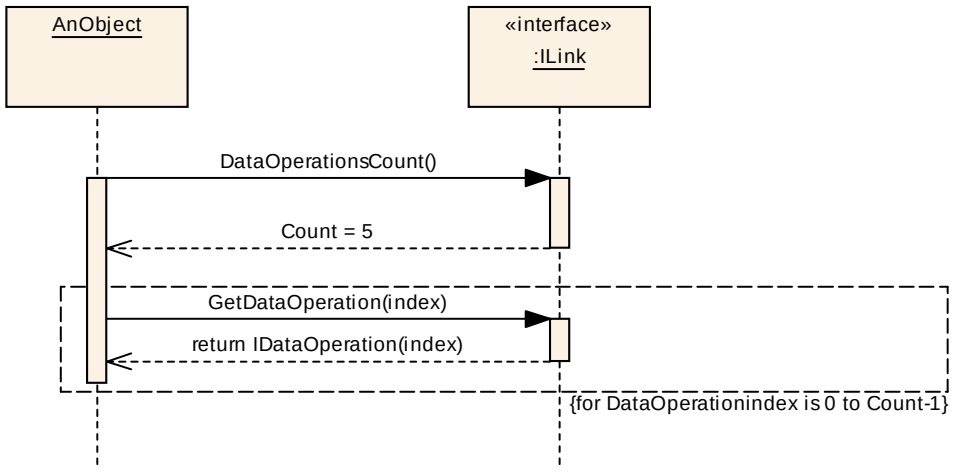


Figure 31 Sequence diagram: obtaining listed items

3.4 OpenMI compliance

Each OpenMI compliant component is available as a software unit which must implement the ILinkableComponent interface, including all underlying interfaces of the org.OpenMI.Standard namespace, and have an associated registration file (the OMI-file). This software unit should adhere to the calling phases as grouped in Section 3.3.1.

A few special situations may occur:

- IListener interface is optional
  The IListener interface is only needed if a component wants to register for specific events and handle accordingly.
- IManageState interface is optional.
  If state management is not supported by a linkable component, one cannot implement the logic of the IManageState interface. Therefore one should not implement this interface or throw an exception.
- IDiscreteTimes interface is optional
  If a component does not know time at all (TimeHorizon is Null), it should not implement the logic of the IDiscreteTimes interface. In all other case it is recommended to implement IDiscreteTimes. In case IDiscreteTimes interface is not implemented, the component should reply with False on the HasDiscreteTimes() method and should throw an exception for all other method calls.
- No input exchange item
  OpenMI compliant data sources not acting as data destination (e.g. input databases or monitoring stations) must return a 'zero' for the item count and a Null for any GetInputExchangeItem()-call
- No output exchange item
  OpenMI compliant data destinations not acting as data source role (e.g. visualization) must return a 'zero' (0) for the item count and a Null for any GetOutputExchangeItem()-call
- No events to publish
  OpenMI compliant linkable components that do not publish events must still implement the IPublisher-interface. They must return a 'zero' (0) for the PublishedEventTypesCount and throw an exception for all other method calls.

References

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

Gregersen, J.B.G. et al. (2002), *Requirements Analysis*. Report of Work Package 2. IT Frameworks (HarmonIT) EC Framework 5. Energy, Environment and Sustainable Development. Contract EVK1-CT-2001-00090

OpenMI Association (2007a) *Scope of the OpenMI architecture*. Part A of the OpenMI report series.

OpenMI Association (2007b) *OpenMI Guidelines*. Part B of the OpenMI report series.

OpenMI Association (2007d) *org.OpenMI.Backbone technical documentation*. Part D of the OpenMI report series.

OpenMI Association (2007e) *org.OpenMI.DevelopmentSupport technical documentation*. Part E of the OpenMI report series.

OpenMI Association (2007f) *org.OpenMI.Utilities technical documentation*. Part F of the OpenMI report series.

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

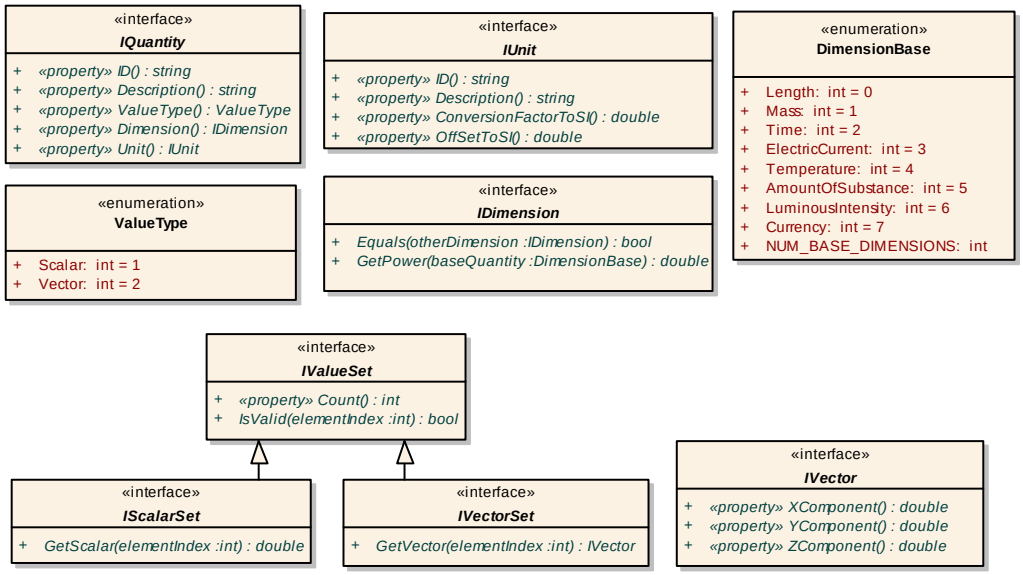
WGS84 (1984) World Geodetic system. http://www.wgs84.com

Annex I org.OpenMI.Standard in short

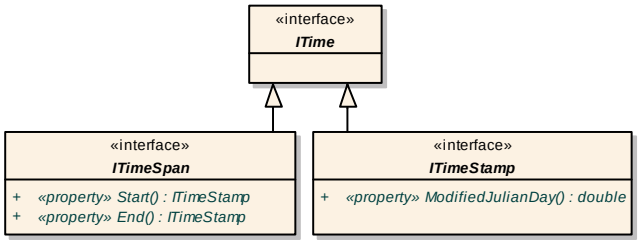
Annex I-A The interface definitions

data definitions

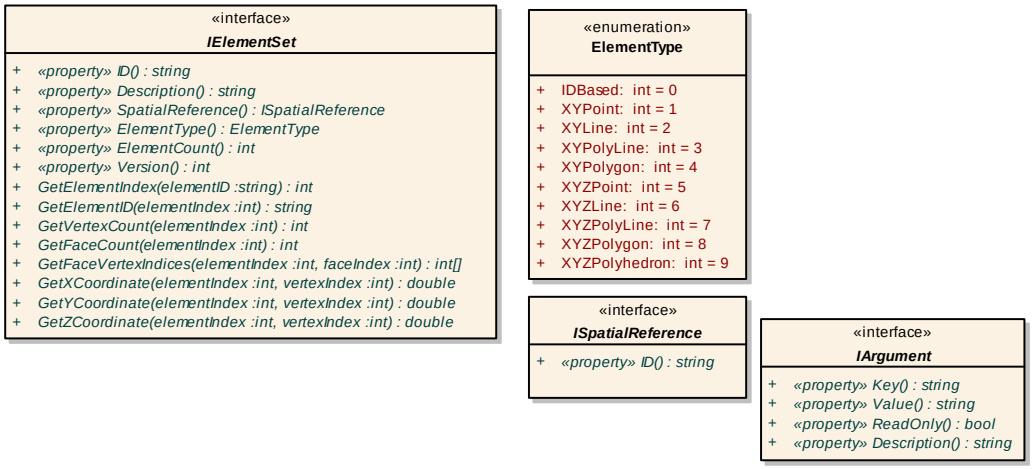
What



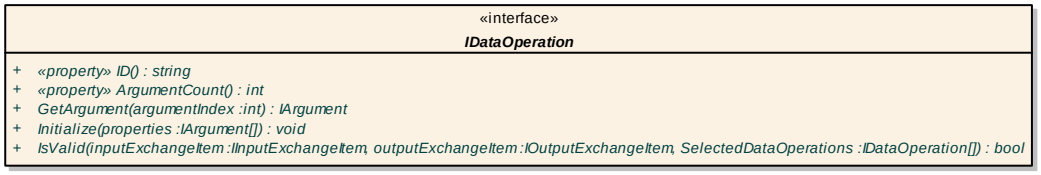
When



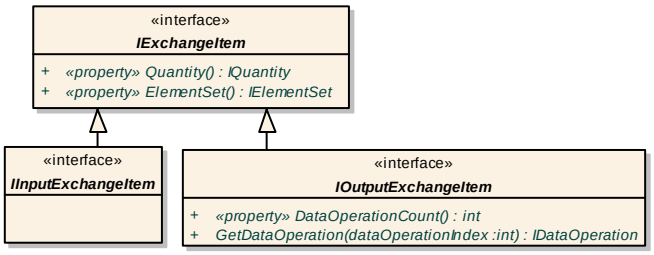
Where



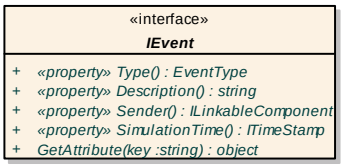
How

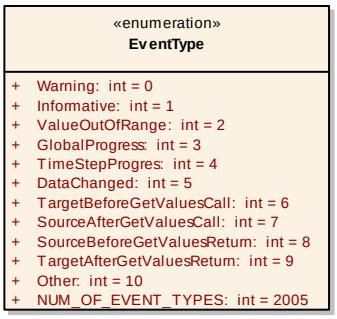


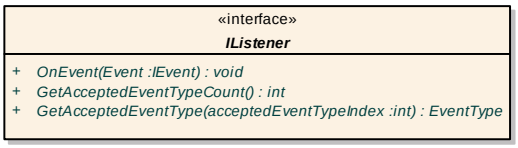
meta data to express what can be exchanged



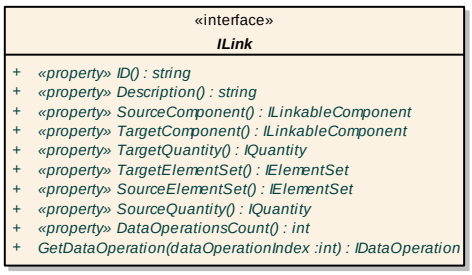
message definition



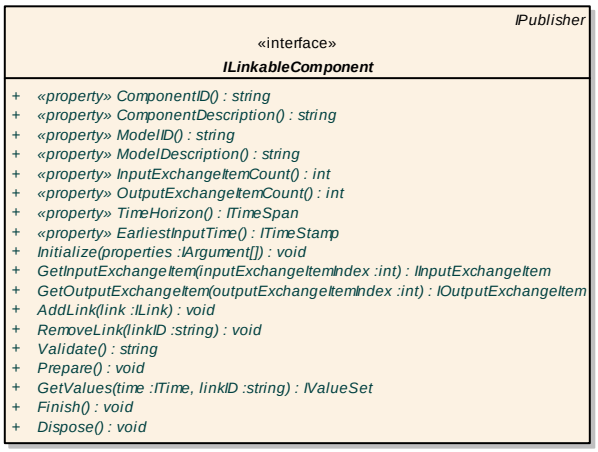




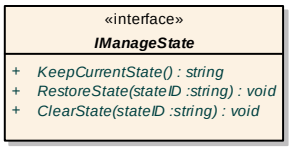
specification what will be exchanged and how

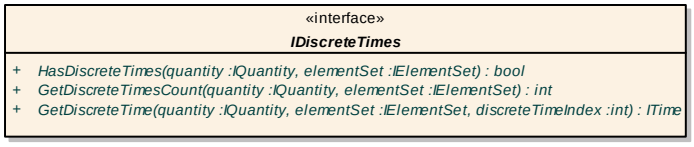


component interfaces for generic component access



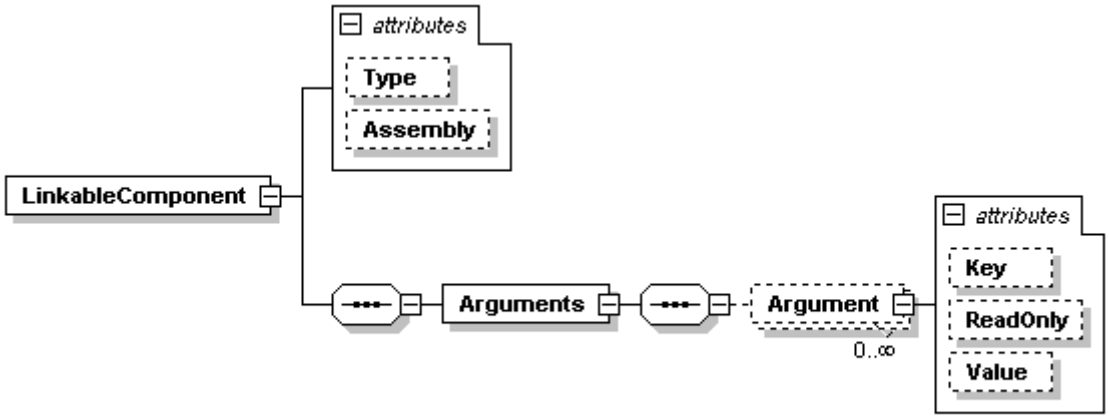
advanced component interface extensions (optional)





Annex I-B The OMI file definition

Graphical view on 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="wlDelft.OpenMI.WLLinkableComponent" Assembly="wlDelft.OpenMI, Version=1.4.0.0,
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>
```

Annex I-C The phases in dynamic utilization

Call sequence of an OpenMI LinkableComponent

Annex II org.OpenMI.Standard API-specification

org.OpenMI.Standard.IArgument

Argument Interface, key-value pairs, applied as arguments for a data operation or arguments to populate components at instantiation with (model specific) data

Key property {get}

Argument identification ('Key' in: Key=Value pair)

Value property {get} {set}

Argument value ('Value' in: Key=Value pair) ; set only allowed in case ReadOnly = “False”

Description property {get}

Description of the argument

ReadOnly property {get}

Boolean determines if argument value can be modified by the user

org.OpenMI.Standard.DimensionBase

Enumeration for base dimensions

Length field

Dimension base length

Mass field

Dimension base mass

time field

Dimension base time

ElectricCurrent field

Dimension base electric current

Temperature field

Dimension base temperature

AmountOfSubstance field

Dimension base amount of substance

LuminousIntensity field

Dimension base luminous intensity

Currency field

Dimension base currency

NUM_BASE_DIMENSIONS field

Total number of base dimensions involved

org.OpenMI.Standard.IDataOperation

DataOperation Interface, identifies data operation and contains associated argument values

GetArgument(System.Int32) method

Get n-th argument, key

Returns

argument object

Initialize(org.OpenMI.Standard.IArgument[]) method

Initialize (called to populate data operation with specific – i.e. selected - information)

Arguments

properties

Array of argument objects to be used for initialization

IsValid(org.OpenMI.Standard.IInputExchangeItem, org.OpenMI.Standard.IOutputExchangeItem, org.OpenMI.Standard.IDataOperation[]) method

Indicates if the combination of data operations is valid for the selected input/output combination

Arguments

inputExchangeItem

The inputExchangeItem as selected in a GUI (i.e. the content will be at the target side of the link)

outputExchangeItem

The outputExchangeItem as selected in a GUI (i.e. the content will be at the source side of the link)

SelectedDataOperations

The array of already selected data operations (excluding the current one)

Returns

Boolean

ID property {get}

Identification string

ArgumentCount property {get}

Number of arguments for this data operation

org.OpenMI.Standard.IDimension

Dimension interface, describes the dimension, expressed in base dimensions, of a Quantity

GetPower(org.OpenMI.Standard.DimensionBase) method

Get power for the selected base dimension

Arguments

baseDimension

baseDimension instance to compare with

Returns

double, power of selected base dimension

Equals(org.OpenMI.Standard.IDimension) method

Check if a Dimension instance equals to another Dimension instance.

Arguments

otherDimension

Dimension instance to compare with

Returns

Boolean, Dimension instances are equal

org.OpenMI.Standard.IDiscreteTimes

DiscreteTimes Interface, to obtain discrete time information associated to the quantity-element set combination of a linkable component

HasDiscreteTimes(org.OpenMI.Standard.IQuantity,org.OpenMI.Standard.IElementSet) method

Is Quantity/Elementset defined on discrete time steps?

GetDiscreteTimesCount(org.OpenMI.Standard.IQuantity,org.OpenMI.Standard.IElementSet) method

Get number of discrete time steps for Quantity/Elementset

GetDiscreteTime(org.OpenMI.Standard.IQuantity,org.OpenMI.Standard.IElementSet,System.Int32) method

Get n-th discrete time stamp or span for Quantity/Elementset

Arguments

quantity

The quantity

elementSet

The element

discreteTimeIndex

index of timeStep

Returns

Discrete time stamp or span

org.OpenMI.Standard.ElementType

Shape Type of an ElementSet

IDBased field

Identifier based

XYPoint field

Points in the (horizontal) XY-plane

XYLine field

Lines/Line segments in the (horizontal) XY-plane

XYPolyLine field

Polylines in the (horizontal) XY-plane

XYPolygon field

Polygons in the (horizontal) XY-plane

XYZPoint field

Points in the 3-dimensional space

XYZLine field

Lines/Line segments in the 3-dimensional space

XYZPolyLine field

Polylines in the 3-dimensional space

XYZPolygon field

Polygons in the 3-dimensional space

XYZPolyhedron field

Polyhedron (volume) in the 3-dimensional space

org.OpenMI.Standard.IElementSet

ElementSet Interface , rigid interface to query an element set for its content in terms of elements, optionally composed of vertices

GetElementIndex(System.String) method

Index of element 'ElementID' in the elementset

GetElementID(System.Int32) method

ID of 'ElementIndex'-th element in the elementset

GetVertexCount(System.Int32) method

Number of vertices for the element specified by:

Arguments

ElementIndex

element index in element set

Returns

Number of vertices in element with ElementIndex

GetXCoordinate(System.Int32,System.Int32) method

X-coord for the vertex with VertexIndex of the element with ElementIndex

Arguments

ElementIndex

element index

VertexIndex

vertex index in the element with index ElementIndex

GetYCoordinate(System.Int32,System.Int32) method

Y-coord for the vertex with VertexIndex of the element with ElementIndex

Arguments

ElementIndex

element index

VertexIndex

vertex index in the element with index ElementIndex

GetZCoordinate(System.Int32,System.Int32) method

Z-coord for the vertex with VertexIndex of the element with ElementIndex

Arguments

ElementIndex

element index

VertexIndex

vertex index in the element with index ElementIndex

ID property {get}

Identification string

Description property {get}

Additional descriptive information

SpatialReference property {get}

Spatial reference system for the element set

ElementType property {get}

Shape Type of the element set

ElementCount property {get}

Number of elements in set

org.OpenMI.Standard.EventType

Shape Type of an ElementSet, extensions for shape types in Z-direction need to be defined

Warning field

Warning event

Informative field

Informative event

ValueOutOfRange field

ValueOutOfRange event

GlobalProgress field

Indicates progress as percentage of global time horizon

TimeStepProgress field

Indicates progress as percentage of requested time step

DataChanged field

Indicates change of (computed) data in the component

TargetBeforeGetValuesCall field

Call by target component to inform source component on an upcoming request. Useful while debugging.

SourceAfterGetValuesCall field

Call by source component to indicate that request has been received

SourceBeforeGetValuesReturn field

Call by source component to indicate that is about to return the valueset

TargetAfterGetValuesReturn field

Call by target component to indicate that values have been received

Other field

Other not predefined event type

NUM_OF_EVENT TYPES field

Total number of event types involved

org.OpenMI.Standard.IEvent

IEvent. Interface, generic meta-data structure to pass event information

GetAttribute(System.String) method

Get the value of a Key=Value pair, containing additional information on the event

Type property {get}

Type of event

Description property {get}

Additional descriptive information

Sender property {get}

LinkableComponent that generated the event

SimulationTime property {get}

Current SimulationTime ("-" if not applicable)

org.OpenMI.Standard.IExchangeItem

ExchangeItem Interface , holds Quantity-ElementSet combinations that can be exchanged

Quantity property {get}

Quantity

ElementSet property {get}

ElementSet

org.OpenMI.Standard.IInputExchangeItem

InputExchangeItem Interface, sub-class of IExchangeItem, holding quantity –element set combinations that act as input to a specific linkable component

org.OpenMI.Standard.ILink

Link Interface, holds the actual information of the link, including the components, quantity and element set on both the source and target side, as well as the selected data operations to be executed by the source component

GetDataOperation(System.Int32) method

Get n-th data operation item

Returns

Data operation

ID property {get}

Identification string

Description property {get}

Additional descriptive information

DataOperationsCount property {get}

Number of data operations

SourceComponent property {get}

Souce linkable component

SourceQuantity property {get}

Source quantity

SourceElementSet property {get}

Source elementset

TargetComponent property {get}

Target linkable component

TargetQuantity property {get}

Target quantity

TargetElementSet property {get}

Target elementset

org.OpenMI.Standard.ILinkableComponent

LinkableComponent Interface, implements org.OpenMI.Standard.IPublisher. Interface for generic model access to the component and the data it can exchange

Initialize(org.OpenMI.Standard.IArgument[]) method

Initialize (called to populate component with specific information)

Arguments

properties

array of IArguments to be used for initlization

GetInputExchangeItem(System.Int32) method

Get n-th input exchange item

Returns

Input exchange item

GetOutputExchangeItem(System.Int32) method

Get n-th output exchange item

Returns

Output exchange item

AddLink(org.OpenMI.Standard.ILink) method

Add an Input or Output Link (Called when initialize has been called for all components)

Arguments

link

Link to be added

RemoveLink(System.String) method

Remove a link

Arguments

LinkID

Link to be added

Validate method

Validation of the component status and its links

Returns

Returns an empty string if the component is valid otherwise returns a message string

Prepare method

Prepare for computation (Called just before computation starts, called when all links have been added)

GetValues(org.OpenMI.Standard.ITime,System.String) method

Get Values for a certain TimeStamp or TimeSpan, for a certain Link (= Quant./Elm.set)

Arguments

time

timestamp or timespan

LinkID

involved link

Returns

ValueSet (scalar or vector)

Finish method

Finish (Called when computation is done, close files, network connectiosn, de-alloate memory where easible)

Dispose method

Dispose (i.e. cleanup; called when deployment stops)

ComponentID property {get}

Identification string of the component/engine

ComponentDescription property {get}

Additional descriptive information of the component/engine

ModelID property {get}

Identification string of the site specific model/study area

ModelDescription property {get}

Additional descriptive information of the site specific model/study area

InputExchangeItemCount property {get}

Number of input exchange items

OutputExchangeItemCount property {get}

Number of output exchange items

TimeHorizon property {get}

time horizon (begin and end date for which data can be retrieved)

EarliestInputTime property {get}

Earliest needed input time (Queried by providing component when they want to clear their buffers)

org.OpenMI.Standard.IListener

Listener Interface, enables components to grab events

OnEvent(org.OpenMI.Standard.IEvent) method

Method called when event is raised

Arguments

Event

Event that has been raised

GetAcceptedEventTypeCount method

Get number of accepted event types

Returns

Number of accepted event types

GetAcceptedEventType(System.Int32) method

Get accepted event type with index acceptedEventTypeIndex

Arguments

acceptedEventTypeIndex

index in accepted event types

Returns

Accepted event type

org.OpenMI.Standard.IManageState

Manage State Interface, to preserve, restore and clear the internal state of a component. (To be implemented optionally, in addition to the linkable component interface.)

KeepCurrentState method

Store the linkable component's current State

Returns

State identifier

RestoreState(System.String) method

Restore the linkable component's current State

Arguments

stateID

State identifier

ClearState(System.String) method

Clears a state from the linkable component's memory

Arguments

stateID

org.OpenMI.Standard.IOutputExchangeItem

OutputExchangeItem Interface, holds the output combinations of a quantity-element set, including a description of data operations that can be offered on the associated value set

GetDataOperation (System.Int32) method

Get n-th data operation

Returns

Data operation description

DataOperationCount property {get}

Get number of data operations

org.OpenMI.Standard.IPublisher

Publisher Interface, provides meta data on available events, accommodates subscription to those events and publishes the events

Subscribe(org.OpenMI.Standard.IListener,System.String) method

Subscribes a listener

Arguments

Listener

The listener

EventType

The event type

UnSubscribe(org.OpenMI.Standard.IListener,System.String) method

Unsubscribes a listener

Arguments

Listener

The listener

EventType

The event type

SendEvent(org.OpenMI.Standard.IEvent) method

Sends an event to all subscribed listeners

Arguments

Event

The event

GetPublishedEventTypeCount method

Get number of published event types

Returns

Number of provided event types

GetPublishedEventType(System.Int32) method

Get provided event type with index providedEventTypeIndex

Arguments

providedEventTypeIndex

index in provided event types

Returns

Provided event type

org.OpenMI.Standard.IQuantity

Quantity Interface, describes the (physical) quantity that can be exchanged

ID property {get}

Identifier

Description property {get}

Additional descriptive information

ValueType property {get}

Quantity's value type (vector, scalar)

Dimension property {get}

Quantity's Dimension

Unit property {get}

Unit in which quantity is expressed

org.OpenMI.Standard.IScalarSet

ScalarSet Interface, holds an array of doubles for a certain quantity on a certain element set (ordering corresponds to elements in element set). Implements org.OpenMI.Standard.IValueSet interface

GetScalar(System.Int32) method

Value for one of the elements in the set

Arguments

ElementIndex

index in the scalar set

Returns

double scalar value

org.OpenMI.Standard.ISpatialReference

SpatialReference Interface, holds a string reference to (known) spatial reference systems

ID property {get}

Identifier indicating which spatial reference to use

org.OpenMI.Standard.ITime

Time Interface, 'Abstract' interface, base for TimeStamp and TimeSpan

org.OpenMI.Standard.ITimeSpan

TimeSpan Interface, describes a continuous period over time

Start property {get}

Time span's begin time stamp

End property {get}

Time span's begin time stamp

org.OpenMI.Standard.ITimeStamp

TimeStamp Interface, describes an instantaneous moment in time

ModifiedJulianDay property {get}

Get TimeStamp expressed as ModifiedJulianDateAndTime (JulianDateAndTime - 2400000.5) Number of days since 1858/11/17 12:00:00.00, and fraction of 24hr. See for example http://aa.usno.navy.mil/data/docs/JulianDate.html

org.OpenMI.Standard.IUnit

Unit Interface, describes the unit in which a quantity is expressed

ID property {get}

Identification string

Description property {get}

Additional descriptive information

ConversionFactorToSI property {get}

Conversion factor to SI ('A' in: SI-value = A * quant-value + B)

OffSetToSI property {get}

OffSet to SI ('B' in: SI-value = A * quant-value + B)

org.OpenMI.Standard.IValueSet

ValueSet Interface holds an array of doubles for a certain quantity on a certain element set (ordering corresponds to elements in element set). Base for VectorSet and ScalarSet

Count property {get}

Number of elements in the set

org.OpenMI.Standard.IVector

Vector Interface, containing the values of the X,Y and Z component of a vector

XComponent property {get}

Vector component in X-direction

YComponent property {get}

Vector component in Y-direction

ZComponent property {get}

Vector component in Z-direction

org.OpenMI.Standard.IVectorSet

VectorSet Interface, holds an array of vectors for a certain quantity on a certain element set. Implements org.OpenMI.Standard.IValueSet interface

GetVector(System.Int32) method

Vector for one of the elements in the set

Arguments

ElementIndex

index in the vector set

Returns

vector

org.OpenMI.Standard.ValueType

Value(Set)Type for Quantity

Scalar field

Scalar

Vector field

Vector

Annex III Overview of changes

Annex III-A 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.

Annex III-B 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

Annex III-C 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

Annex III-D 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 ‘meta data interface’ namely IExchangeModel. 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.

Annex III-E 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: org.OpenMI.Standard 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 are 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 if 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

INDEX