



**The OpenMI Document Series**

# **Part B - Guidelines**

**For the OpenMI (Version 1.4)**

<b>Title</b>	OpenMI Document Series: Part B - Guidelines for the OpenMI (version 1.4)
<b>Editor</b>	Isabella Tindall, Centre for Ecology and Hydrology, Wallingford, UK
<b>Authors</b>	Peter Gijbbers, WL Delft Hydraulics, Delft, The Netherlands Jan Gregersen, DHI Water and Environment, Hørsholm, Denmark Stefan Westen, HR Wallingford Group, Wallingford, UK Flip Dirksen, RIZA, Lelystad, The Netherlands Costas Gavardinas, National Technical University of Athens, Greece Michiel Blind, RIZA, Lelystad, The Netherlands
<b>Document production</b>	Stephen Morris, Butford Technical Publishing Ltd., Pershore, UK
<b>Current version</b>	V1.4
<b>Date</b>	21/05/2007
<b>Status</b>	Final © The OpenMI Association
<b>Copyright</b>	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.
<b>Acknowledgement</b>	<p>This document has been produced as part of the OpenMI-Life project.</p> <p>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 management" Contract no : LIFE06 ENV/UK/000409.</p> <p>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.</p>

## Preface

OpenMI stands for the Open Modeling Interface and aims to deliver a standardized way of linking of environmental related models. This document provides Guidelines for using the Open Modelling Interfaces and supporting software, the OpenMI. It is the second in the OpenMI report series, which specifies the OpenMI interface standard, provides guidelines on its use and describes facilities for migrating, setting up and running linked models. Other titles in the series include:

- A. Scope
- B. Guidelines** (this document)
- C. org.OpenMI.Standard interface specification
- D. org.OpenMI.Backbone technical documentation
- E. org.OpenMI Development Support technical documentation
- F. org.OpenMI.Utilities technical documentation

The Guidelines are divided into five books:

- Book 1 provides an introduction to the OpenMI and describes the processes involved in creating an OpenMI-compliant model or migrating an existing model.
- Book 2 describes the way in which data are exchanged between OpenMI-compliant systems.
- Book 3 provides information on how to develop an OpenMI system.
- Book 4 gives instructions for migrating an existing model so that it becomes OpenMI-compliant.
- Book 5 illustrates the way in which data can be exchanged with software other than models (e.g text files, spreadsheets and databases).

The Guidelines are 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) *Guidelines*. Part B 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>.

## Contents

<b>BOOK 1</b>	<b>INTRODUCING THE OPENMI.....</b>	<b>1-1</b>
<b>Chapter 1.1</b>	<b>Introduction to the OpenMI .....</b>	<b>1-3</b>
1.1.1	Background .....	1-4
1.1.2	The OpenMI objectives and benefits .....	1-5
1.1.2.1	Aims and objectives.....	1-5
1.1.2.2	Performance and error handling.....	1-6
1.1.2.3	Benefits .....	1-6
1.1.3	Use cases.....	1-8
1.1.4	Terminology.....	1-9
<b>Chapter 1.2</b>	<b>Object oriented programming and UML.....</b>	<b>1-13</b>
1.2.1	Object oriented programming.....	1-14
1.2.1.1	Objects .....	1-14
1.2.1.2	Classes .....	1-14
1.2.1.3	Inheritance.....	1-15
1.2.2	UML .....	1-16
1.2.2.1	Class diagrams.....	1-16
1.2.2.2	Sequence diagrams.....	1-17
1.2.2.3	Interfaces .....	1-21
<b>Chapter 1.3</b>	<b>Linking models .....</b>	<b>1-23</b>
1.3.1	Linking models at run-time .....	1-24
1.3.2	The request-reply mechanism .....	1-26
1.3.2.1	The pull mechanism.....	1-26
1.3.2.2	Other features.....	1-27
1.3.3	The GetValues method.....	1-28
<b>Chapter 1.4</b>	<b>Developing OpenMI systems .....</b>	<b>1-31</b>
1.4.1	Overview of an OpenMI-compliant system .....	1-32
1.4.1.1	OpenMI systems.....	1-32
1.4.1.2	The OMI file.....	1-32
1.4.2	Deployment phases .....	1-33
1.4.3	Migrating existing models.....	1-34
1.4.3.1	Criteria for becoming a linkable component.....	1-34
1.4.3.2	The migration process .....	1-35
1.4.4	Implementations of the OpenMI .....	1-37
<b>BOOK 2</b>	<b>EXCHANGING DATA.....</b>	<b>2-1</b>
<b>Chapter 2.1</b>	<b>Data exchange at run-time .....</b>	<b>2-3</b>
2.1.1	The data exchange mechanism .....	2-4
2.1.2	The role of element sets in data exchange .....	2-6
2.1.3	Bidirectional links.....	2-7
2.1.3.1	Example 1: Linkage of two dynamic river flow models .....	2-7
2.1.3.2	Example 2: Linkage of a river model with a plant growth model .....	2-7
2.1.3.3	Example 3: Linkage of a river model with a weir control module .....	2-8
<b>Chapter 2.2</b>	<b>Describing the exchange data.....</b>	<b>2-9</b>

2.2.1	Introducing the use case .....	2-10
2.2.2	What to describe.....	2-11
2.2.3	Defining what the values represent .....	2-12
2.2.4	Defining where the values apply .....	2-15
2.2.4.1	OpenMI ElementSets.....	2-15
2.2.4.2	Using different types of elements .....	2-17
2.2.4.3	Choosing an ElementType.....	2-20
2.2.4.4	Dynamic ElementSets.....	2-21
2.2.5	Using data operations to describe how data can be mapped .....	2-22
2.2.6	Grouping into Exchangeltems .....	2-26
2.2.6.1	Exchangeltems.....	2-26
2.2.6.2	Initially unknown Exchangeltems .....	2-27
2.2.7	An advanced example .....	2-31
<b>Chapter 2.3</b>	<b>Configuring links and compositions .....</b>	<b>2-35</b>
2.3.1	Configuring a single link.....	2-36
2.3.2	Building a composition.....	2-38
<b>Chapter 2.4</b>	<b>Using the OpenMI configuration editor .....</b>	<b>2-39</b>
2.4.1	Starting the configuration editor .....	2-40
2.4.2	Adding models to the composition.....	2-41
2.4.3	Establishing connections between the models .....	2-43
2.4.4	Configuring the connections.....	2-44
2.4.5	Adding a trigger .....	2-45
2.4.6	Running the composition .....	2-46
<b>BOOK 3</b>	<b>DEVELOPING OPENMI SYSTEMS .....</b>	<b>3-1</b>
<b>Chapter 3.1</b>	<b>OpenMI-compliant systems.....</b>	<b>3-3</b>
3.1.1	What is an OpenMI system? .....	3-4
3.1.2	Locating the components.....	3-5
<b>Chapter 3.2</b>	<b>Establishing OpenMI systems.....</b>	<b>3-7</b>
3.2.1	Phases in using the linkable component interface .....	3-8
3.2.2	Phase I: Instantiation and initialization.....	3-9
3.2.3	Phase II: Inspection and configuration.....	3-10
3.2.4	Phase III: Preparation .....	3-12
3.2.5	Phase IV: Computation/execution .....	3-13
3.2.6	Phase V: Completion .....	3-14
3.2.7	Phase VI: Disposal .....	3-15
<b>Chapter 3.3</b>	<b>Hard-coded systems.....</b>	<b>3-17</b>
3.3.1	An example of a hard-coded system .....	3-18
<b>Chapter 3.4</b>	<b>Support for configurable systems .....</b>	<b>3-23</b>
3.4.1	Main aspects of a configurable system.....	3-24
3.4.2	Configuring and sustaining a component combination.....	3-25
3.4.3	Deploying and running the system .....	3-31
<b>Chapter 3.5</b>	<b>Graphical user interfaces .....</b>	<b>3-37</b>
3.5.1	Building visual tools .....	3-38
3.5.2	OmiEd, a simple front end of the OpenMI SDK.....	3-39

<b>BOOK 4</b>	<b>MIGRATING OPENMI MODELS .....</b>	<b>4-1</b>
<b>Chapter 4.1</b>	<b>Introduction .....</b>	<b>4-3</b>
4.1.1	OpenMI compliance.....	4-4
4.1.2	The Simple River example.....	4-6
<b>Chapter 4.2</b>	<b>Planning the migration .....</b>	<b>4-7</b>
4.2.1	Use cases.....	4-8
4.2.1.1	Use case 1: Connecting to other rivers.....	4-8
4.2.1.2	Use case 2: Inflow from geo-referenced catchment database.....	4-10
4.2.2	Defining exchange items.....	4-12
<b>Chapter 4.3</b>	<b>Wrapping.....</b>	<b>4-13</b>
4.3.1	A general wrapping pattern .....	4-14
4.3.2	The LinkableEngine.....	4-15
<b>Chapter 4.4</b>	<b>Migration – step by step.....</b>	<b>4-17</b>
4.4.1	Step 1: Changing your engine core.....	4-18
4.4.2	Step 2: Creating the .NET assemblies.....	4-20
4.4.3	Step 3: Accessing the functions in the engine core.....	4-22
4.4.4	Step 4: Implementing MyEngineDotNetAccess.....	4-24
4.4.5	Step 5: Implementing the MyEngineWrapper class.....	4-26
4.4.6	Step 6: Implementing MyModelLinkableComponent .....	4-28
4.4.7	Step 7: Implementation of the remaining IEngine methods .....	4-29
<b>Chapter 4.5</b>	<b>Migration of the Simple River.....</b>	<b>4-31</b>
4.5.1	The Simple River wrapper.....	4-32
4.5.2	Implementation of the Initialize method .....	4-33
4.5.3	Implementation of the SetValues method.....	4-36
4.5.4	Implementing the GetValues method .....	4-37
4.5.5	Implementation of the remaining methods.....	4-38
<b>Chapter 4.6</b>	<b>Testing the component.....</b>	<b>4-41</b>
4.6.1	Unit testing .....	4-42
<b>Chapter 4.7</b>	<b>Implementing IManageState.....</b>	<b>4-45</b>
4.7.1	The IManageState interface.....	4-46
<b>Chapter 4.8</b>	<b>The OMI file .....</b>	<b>4-49</b>
4.8.1	Structure of the OMI file .....	4-50
<b>Chapter 4.9</b>	<b>Design patterns for model migration.....</b>	<b>4-51</b>
4.9.1	Design patterns for ISIS.....	4-52
4.9.2	Design patterns for InfoWorks RS .....	4-53
4.9.3	Design patterns for Mike11 .....	4-54
4.9.4	Design patterns for SOBEK .....	4-57
<b>Chapter 4.10</b>	<b>Performance issues.....</b>	<b>4-59</b>
4.10.1	Memory consumption.....	4-60
4.10.2	System processes .....	4-61

<b>BOOK 5</b>	<b>NON-MODEL COMPONENTS .....</b>	<b>5-1</b>
<b>Chapter 5.1</b>	<b>Desktop and database applications.....</b>	<b>5-3</b>
5.1.1	ASCII files.....	5-4
5.1.2	Spreadsheets .....	5-8
5.1.2.1	Generating an ASCII file .....	5-8
5.1.2.2	Accessing Excel using Visual Studio Tools for the Microsoft Office system..	5-8
5.1.3	Report engines .....	5-12
5.1.4	Databases .....	5-13
5.1.4.1	Accessing databases.....	5-13
5.1.4.2	Accessing databases using ADO.NET .....	5-13
5.1.5	GIS.....	5-15
5.1.5.1	Accessing GIS through software libraries.....	5-15
5.1.5.2	Accessing GIS through ASCII files.....	5-15
<b>Chapter 5.2</b>	<b>Visualization .....</b>	<b>5-17</b>
5.2.1	The OpenMI DataMonitor.....	5-18
<b>Chapter 5.3</b>	<b>Advanced controllers .....</b>	<b>5-19</b>
5.3.1	Iteration .....	5-20
5.3.2	Optimization .....	5-26
5.3.3	Calibration .....	5-30
5.3.4	UML diagrams.....	5-32



# Book 1 Introducing the OpenMI

<b>BOOK 1</b>	<b>INTRODUCING THE OPENMI.....</b>	<b>1-1</b>
<b>Chapter 1.1</b>	<b>Introduction to the OpenMI .....</b>	<b>1-3</b>
1.1.1	Background .....	1-4
1.1.2	The OpenMI objectives and benefits .....	1-5
1.1.2.1	Aims and objectives.....	1-5
1.1.2.2	Performance and error handling.....	1-6
1.1.2.3	Benefits .....	1-6
1.1.3	Use cases.....	1-8
1.1.4	Terminology.....	1-9
<b>Chapter 1.2</b>	<b>Object oriented programming and UML .....</b>	<b>1-13</b>
1.2.1	Object oriented programming.....	1-14
1.2.1.1	Objects .....	1-14
1.2.1.2	Classes .....	1-14
1.2.1.3	Inheritance.....	1-15
1.2.2	UML .....	1-16
1.2.2.1	Class diagrams.....	1-16
1.2.2.2	Sequence diagrams.....	1-17
1.2.2.3	Interfaces .....	1-21
<b>Chapter 1.3</b>	<b>Linking models .....</b>	<b>1-23</b>
1.3.1	Linking models at run-time.....	1-24
1.3.2	The request-reply mechanism.....	1-26
1.3.2.1	The pull mechanism.....	1-26
1.3.2.2	Other features.....	1-27
1.3.3	The GetValues method.....	1-28
<b>Chapter 1.4</b>	<b>Developing OpenMI systems .....</b>	<b>1-31</b>
1.4.1	Overview of an OpenMI-compliant system .....	1-32
1.4.1.1	OpenMI systems.....	1-32
1.4.1.2	The OMI file.....	1-32
1.4.2	Deployment phases .....	1-33
1.4.3	Migrating existing models.....	1-34
1.4.3.1	Criteria for becoming a linkable component.....	1-34
1.4.3.2	The migration process .....	1-35
1.4.4	Implementations of the OpenMI .....	1-37



## Chapter 1.1 Introduction to the OpenMI

The OpenMI standard has been designed to allow data to be exchanged between independent models running simultaneously. Any models that are designed to comply with the requirements of the standard will be able to exchange data, as will any existing models that are modified to be OpenMI-compliant.

These Guidelines provide developers and users with the information needed to make models OpenMI-compliant and then to link and run them. Book 1 of the Guidelines provides an introduction to the OpenMI and describes the processes involved in creating an OpenMI-compliant model or migrating an existing model.

The book refers to the OpenMI supporting software, i.e. the Software Development Kit and the Graphical user interface, which contains a number of tools to ease the task of converting an existing model engine into an OpenMI linkable component. There is no requirement to use these tools when creating an OpenMI-compliant component but they may make the task simpler.

This chapter introduces the OpenMI, giving details of its objectives, the models that can be linked using the OpenMI, the examples against which the standard has been tested and the terminology used throughout the Guidelines.

### 1.1.1 Background

The *Water Framework Directive* (WFD) calls for *integrated water management* to be put into practice and identifies *whole catchment modelling* as a key part of integrated management. The challenge that this presents is not only that individual catchment processes be modelled but also their interactions. Constructing a single model of all catchment processes is not a feasible option, does not make good use of existing models and doesn't provide the flexibility to try alternative models of individual processes. The only realistic mechanism for whole catchment modelling is *integrated modelling*. This approach links models of different processes and hence allows process interactions to be simulated.

Within the FP5 project *HarmonIT*, co-funded by the European Commission, the *Open Modelling Interface and supporting software (the OpenMI)* has been developed. The *OpenMI Interface* is a standard interface that enables *OpenMI components* to exchange data as they run. An OpenMI component is a piece of software that complies with the OpenMI requirements.

The *OpenMI supporting software* comprises a Software Development Kit and a Graphical User Interface. These tools facilitate making new and existing model codes *OpenMI-compliant* and they offer facilities to combine OpenMI-compliant components into integrated modelling systems and then run them.

OpenMI components are not restricted to being models. The interface can also be used for data exchange with, for example, databases, text files, GUIs, report writers and visualization aids.

## 1.1.2 The OpenMI objectives and benefits

The OpenMI has been developed with a number of objectives in mind. Adopting the OpenMI can produce a variety of benefits.

### 1.1.2.1 Aims and objectives

The aim of the OpenMI is to provide a mechanism by which physical and socio-economic process models can be linked to each other, to other data sources and to a variety of tools at run-time, hence enabling process interactions to be better modelled.

Specific objectives are that the mechanism's design should allow the linking of:

- Models from different domains (hydraulics, hydrology, ecology, water quality, economics etc.) and environments (atmospheric, freshwater, marine, terrestrial, urban, rural etc.)
- Models based on different modelling concepts (deterministic, stochastic etc.)
- Models of different dimensionality (0, 1, 2, 3D)
- Models working at different scales (e.g. a regional climate model to a catchment runoff model)
- Models operating at different temporal resolutions (e.g. hourly to monthly or even annual)
- Models operating with different spatial representations (e.g. networks, grids, polygons)
- Models using different projections, units and categorizations
- Models that link to other data sources (e.g. databases, user interfaces, instruments)
- Models running on different platforms (e.g. Windows, Unix and Linux)

Note that linked models can be run on different computers, as long as the components are OpenMI-compliant and an appropriate distributed technology or technology combination is chosen for implementation; for example, remoting protocols of .NET, Java and WebServices could be applied, as well as proxy-stub patterns.

More general objectives are that the mechanism:

- Be applicable to new and existing models
- Impose as few restrictions as possible on the modeller's freedom
- Be applicable to most, if not all, time-based simulation techniques
- Require the minimum of change to the program code of existing applications
- Keep the cost, skill and time required to migrate an existing model to a minimum so that these factors are not a deterrent to the OpenMI's use

- Be easy to use
- Not unreasonably degrade performance

### 1.1.2.2 Performance and error handling

Many models are computationally intensive and for these the maintenance of performance is an important issue. Particular care has therefore been taken in the design of the OpenMI to minimize any reduction in performance.

Factors that are likely to affect performance are:

- The complexity of the models
- The amount of data exchanged
- The complexity of the links
- The location of the models
- The communications network over which links run
- The efficiency of the code

The handling of errors in a way that enables the source of the errors to be identified quickly has also been given careful attention. The OpenMI uses exceptions for error handling and an event message system to pass progress and debugging information. A number of conventions, some mandatory and some voluntary, are either required or recommended to be adopted by code developers.

### 1.1.2.3 Benefits

The discussion above has explained the need for the OpenMI created by the adoption of the WFD. What benefits does it bring to the designated authorities, basin managers, regulators, consultants, modellers and model developers responsible for implementing the WFD? Some of the arguments for adopting the OpenMI put forward by organizations that have already adopted or are considering adopting the OpenMI are:

- Protection and enhancement of existing investment in model development (i.e. it is not necessary to rewrite them completely in order for them to become OpenMI-compliant)
- The simplification of the model-linking process, leading to an improved ability to model process interactions
- The ability to use appropriate model combinations and to swap between different models of the same process, assisting sensitivity analyses and benchmarking
- A reduction in development time and hence cost for decision support systems
- An increased choice for model users, in that they will be able to 'mix and match' models from different sources

- Increased opportunities for model developers, in that individual models become more saleable because they can be linked to established systems, enhancing the value of both
- Increased opportunities for the creation of Small and Medium Enterprises (SME), especially from the academic sector
- Increased opportunities to contribute to the implementation and evolution of EU policies
- The opportunity for model developers to concentrate their core business (e.g. computational cores) because they will be able to buy in OpenMI-compliant tools such as GUIs and post-processing tools
- The OpenMI Software Development Kit and Graphical User Interface, which offer tools for migrating and linking models and monitoring linked model runs (the tools are available free under an Open Source licence and would otherwise have to be written by the developer)
- The small cost of conversion compared with the cost of writing a whole catchment model from scratch or redeveloping existing models
- The ability for model users to run third-party computational cores in their own environments
- No need to understand other organizations' I/O procedures
- The ability to change a model's code without affecting the linking process or interface

### 1.1.3 Use cases

A range of scenarios or 'use cases' were identified to check that the requirements were correctly expressed and to ease the development of an architecture for the OpenMI. Some examples from the full list of cases are shown below:

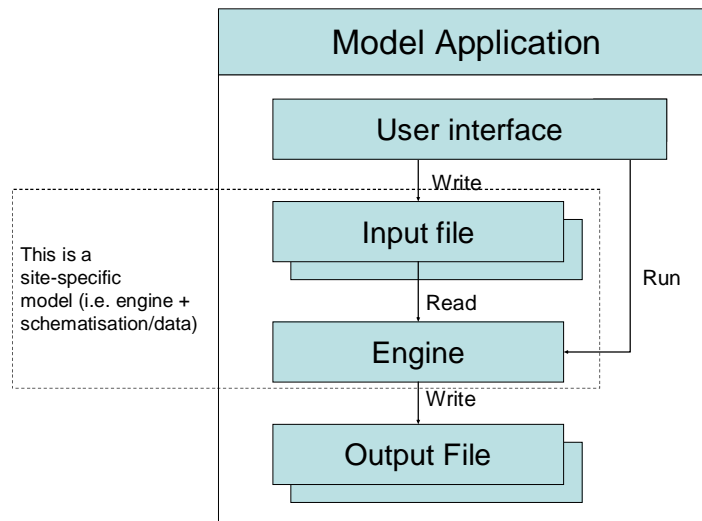
- Connect two 1D hydrodynamic river models.
- Connect a 1D hydrodynamic model with a water quality transport model.
- Connect a 1D river model with a 3D groundwater model.
- Connect a 1D hydrodynamic river model to vegetation and habitat models.
- Connect a 3D coastal model to a 1D river model.
- Connect a 2D polygon-based root zone model to a 3D regular grid groundwater model.
- Calibrate a rainfall runoff model linked to a hydrodynamic sewerage model.
- Model the propagation of uncertainty through a chain of models.
- Use different units of measurement for the data to be exchanged between models.
- Connect to an agent-based model.



## 1.1.4 Terminology

A number of terms are used when describing the OpenMI standard.

As shown in Figure 1-1, the term *model application* encompasses all parts of the modelling system software that is installed on a computer: for example Mike11, PHABSIM or InfoWorks-RS.



**Figure 1-1 The general structure of a model application**

Typically, such systems consist of a *user interface* and an *engine*. Usually, the engine is a generic representation of a process and is where the calculations for simulating or modelling that process take place. The user supplies information through the user interface and this is converted into the input data for the engine.

The data describe a specific scenario in which the process is to be simulated: for example the Rhine during a time of extreme rainfall. The user runs the engine by selecting an option or pressing a button on the user interface. The engine reads the input, performs the calculations and outputs the results to files or displays.

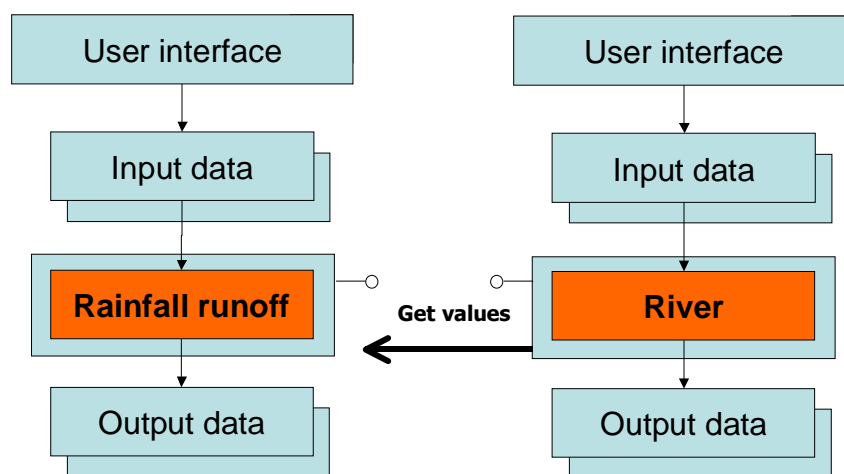
When an *engine* has read its input it becomes a *model*. For example, an engine may represent the generic process of water flowing in an open channel. When it has read in the data describing the channel network of the Rhine, along with any boundary conditions and rainfall data, it becomes a model of the Rhine in the scenario to be simulated.

If the code for an engine can be instantiated separately and has a well-defined *interface* through which it can *accept* and *provide* data, then it is an *engine component*. The engine's *interface* is the part of the code that handles the transfer of data to and from the engine; it should not be confused with the *user interface*, which is the part of the application that the user sees. The key to enabling models to exchange data lies in standardizing the design of the engine interface. When an engine component implements such a standard interface, it becomes a *linkable component*. An engine that implements the OpenMI interface is called *OpenMI-compliant*. Note that the engine is controlled from outside by applications that call the functions in the engine's interface.

The OpenMI defines a standard interface that has three functions:

- *Model definition:* To allow other linkable components to find out what items this model can exchange in terms of quantities simulated and the locations at which the quantities are simulated.
- *Configuration:* To define what will be exchanged when two models have been linked for a specific purpose.
- *Run-time operation:* To enable the model to accept or provide data at run-time.

Figure 1-2 shows two model applications whose engines have been made OpenMI-compliant. Their overall structure remains unchanged but each engine is now a component with an OpenMI interface.



**Figure 1-2 Two applications after migration to the OpenMI standard**

Figure 1-3 illustrates some of the information held in the model definition about the variables or *quantities* that two models can either *accept* or *provide*. The arrow represents a *link* between the two models and indicates that, in this particular case, runoff produced by the Rainfall Runoff Model will be used to represent lateral inflow in the River Model. There is no requirement to harmonize the terminology; the linking process creates the appropriate cross-reference table.

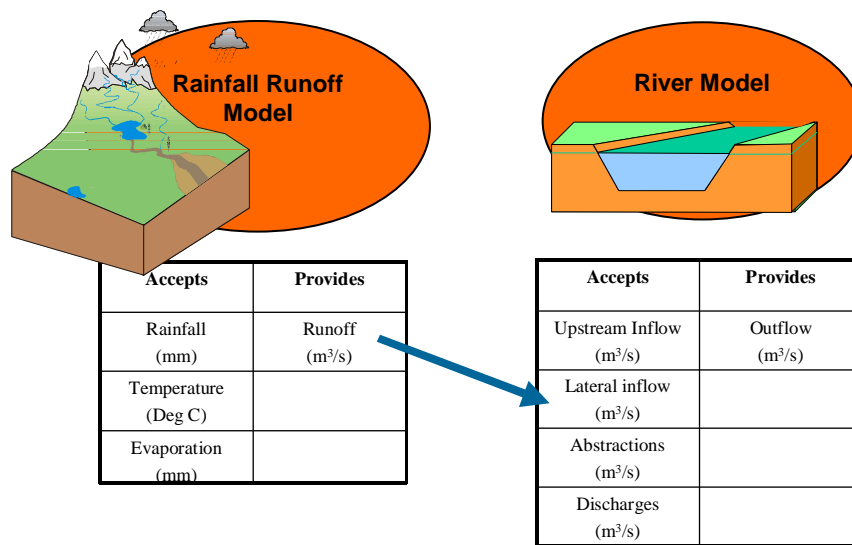


Figure 1-3 Showing and linking quantities

An *element* is an object or location for which quantities are computed over time by a model.

Figure 1-4 shows the geographical matching of *elements* in a river model to those in a groundwater model. The river model is a vector model and each element represents a single stretch; the groundwater model is grid-based, each node being an element. Therefore, in order to link the two models, each element in the river model will usually be linked to several elements in the groundwater model. The OpenMI supporting software provides tools for aggregating and disaggregating values passing between models based on different spatial representations, such as vectors, areas or grids.

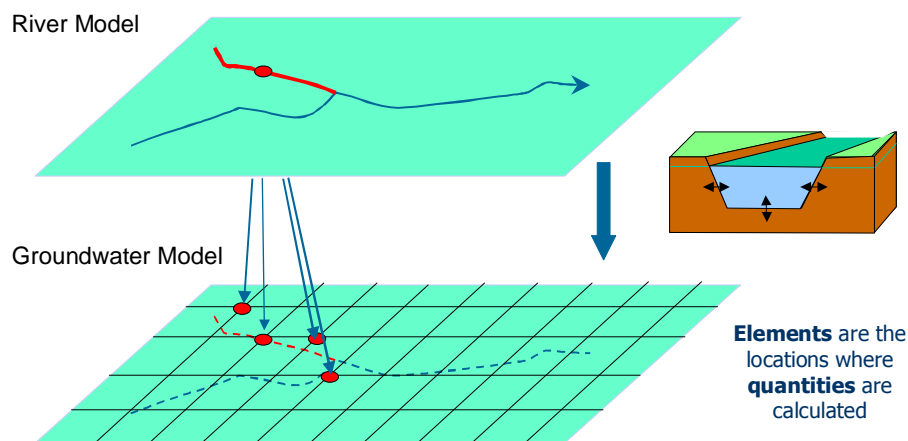


Figure 1-4 Linking element sets



## Chapter 1.2 Object oriented programming and UML

The construction of an OpenMI-compliant model requires a basic understanding of object oriented programming and in particular the relationship between objects and classes and the principle of inheritance. Some knowledge of the Unified Modelling Language (UML) is also necessary.

This chapter explains the principles of object orientated programming and UML. The information given here is of particular interest to Fortran programmers wishing to migrate existing Fortran models to the OpenMI.

## 1.2.1 Object oriented programming

Object oriented programming (OOP) provides the basis for the languages most commonly used when developing models: for example C#, C++ and Java. This section gives a brief introduction to the philosophy and terminology of object oriented programming.

### 1.2.1.1 Objects

*Objects* are the basis of object oriented technology since they represent real-life entities. Objects have a *state* and *behaviour*. An object maintains its state through *variables*, whereas its behaviour is expressed in terms of its *methods*. (Note that the term varies: methods are also called *functions*, *procedures* or *subroutines* depending on the terminology used).

Consider as an example that you want to model real-world bicycles. A bicycle's state is expressed in terms of the variables that represent the bicycle's speed, direction, pedal cadence and current gear. You may also want to consider the bicycle's colour, brand and so on. A bike would also need methods to express its behaviour, such as a method to apply the break, change the pedal cadence or change gears.

Therefore an *object* is a software bundle of *variables* and related *methods*.

Typically, methods surround and hide the object's state from other objects in the program. Packaging an object's variables within the protective custody of its methods is called *encapsulation*. In many cases an object may wish to expose some of its variables or hide some of its methods.

In brief, the two primary benefits of object oriented programming are:

- *Modularity*: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.
- *Information hiding*: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. (You don't need to understand the gear mechanism on your bike to use it.)

### 1.2.1.2 Classes

Using the previous example, imagine different sorts of bicycle in a race. Each has its own speed, direction etc. In OOP terminology, all these bicycle objects are *instances* of the *class* of objects known as bicycles. All these objects, despite having different states, being independent and being different to each other, share the same structure. This commonality is expressed by the *class* bicycle. A class is a prototype that defines the variables and the methods common to all objects of a certain kind.

The difference between *classes* and *objects* is often the source of some confusion, as frequently the terms are wrongly used interchangeably. Always keep in mind that the *class* bicycle refers to the general blueprint or description of a bicycle, which does not 'exist', whereas bicycle *objects* or *instances* refer to specific bicycles that exist, have a specific speed, colour etc.

### 1.2.1.3 Inheritance

Probably, the most important feature of OOP is *inheritance*. Inheritance allows classes to be defined in terms of other classes. Consider for instance, mountain bikes and racing bikes. They are both bicycles and share all the common characteristics of bicycles, such as being able to break, having a speed etc. At the same time though, there are things that differ between them; a mountain bike might need to represent its suspension unit whereas a racing bike might need to express its aerodynamic factor.

It would be a waste of time to have to rewrite the classes representing racing bikes and mountain bikes. OOP languages avoid this by introducing inheritance. Mountain bikes and racing bikes *are* bicycles and their respective classes inherit from the bicycle *class*. This automatically enables them to inherit the state (the variables) as well as the behaviour (the methods) of the bicycle class.

Depending on the OOP language there are different terms to express the relationship between the classes. In C# and C++ the mountain bike class is a *derived class* and inherits from the *base class* bicycle. However, these terms are also described as *child class* as opposed to the *parent class* or *subclass* as opposed to *superclass*.

Subclasses can also *override* inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the 'change gears' method so that the rider could use those new gears. This is often useful in real situations. Imagine that somebody has developed a library that defines the class bicycle and some functions that take bicycle objects as arguments. You could inherit from the bicycle class to define the mountain bike class. Although the instances of mountain bike class might have more functionality than the simple bike class you can still send an instance of mountain bike to a function that expects a bicycle object, since a mountain bike *is* a bicycle. When the 'change gear' method is called in the function, the version of the rewritten mountain bike class will be called!

## 1.2.2 UML

The Unified Modelling Language (UML) is a family of graphical notations that help in describing and designing software systems, particularly software systems built using the object oriented programming (OOP) methodology. It is an open standard, controlled by an open consortium of companies and was born out of the unification of other OOP graphical modelling languages in 1997.

UML officially describes a total of 13 types of diagrams, each with its own characteristics, but just two of these are described here: class diagrams and sequence diagrams. The following sections briefly explore the main properties of these diagrams and illustrate their use by simple examples. The code that accompanies these examples is written in C# but can be easily understood if you are familiar with other OOP languages such as Java or C++.

### 1.2.2.1 Class diagrams

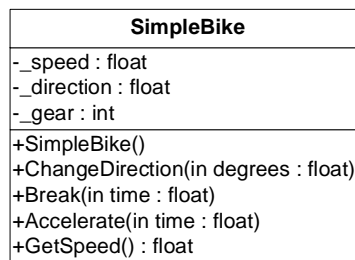
Class diagrams are the most commonly used UML diagrams. A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected.

Consider the following code (in C#) for a simple class that represents a bike:

```
class SimpleBike
{
    private float _speed;
    private float _direction;
    private int _gear;
    public void ChangeDirection(float degrees)
    {
        // Do stuff to change direction
    }
    public void Break(float time)
    {
        // Break for some time
    }
    public void Accelerate(float time)
    {
        // Accelerate for some time
    }
    public void ChangeGear(int gear)
    {
        // Change gear
    }
    public float GetSpeed()
    {
        return _speed;
    }
}
```

Figure 1-5 shows the UML class diagram corresponding to that class.





**Figure 1-5 UML Class diagram**

A class is represented by a single box divided into three sections. The upper section contains the name of the class.

The middle section contains the *attributes*, which represent the structural features of the class (its fields). The attribute notation form is:

*visibility name : type = default*

The *visibility* marker indicates whether the attribute is public (+), private (-) or protected (#). The *name* typically corresponds to the name of the field in a programming language. The *type* following the colon declares the data type of the attribute. Finally, if there is a *default* value for the attribute, it is written after the = sign.

The bottom section of the class box contains the *operations* of the class (its *methods* or *functions* in Java and C++/C# terminology respectively). The notation used to describe operations is:

*visibility name (parameter-list) : return-type*

In this case, *visibility* and *name* are the same as for attributes. The *return-type* shows the type of object that the operation returns. The *parameter-list* is in the form:

*direction name : type = default value*

The *direction* indicates whether the parameter is input (*in*), output (*out*) or both (*inout*). If no direction is shown, it is assumed to be *in*. There are several other syntax modifiers you may encounter in UML class diagrams.

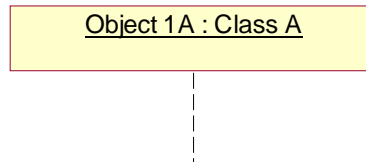
### 1.2.2.2 Sequence diagrams

In OOP analysis and design, the real objective is to devise ways in which groups of objects can collaborate to complete some useful task. Different kinds of objects perform different and specialized functions that when orchestrated correctly produce the desired overall results. In OOP terms, the objects interact with each other by sending *messages*. While class diagrams describe the static structure of a system, *sequence diagrams* describe interactions among classes in terms of an exchange of messages over time.

Sequence diagrams display objects, not classes, showing how these objects interact with each other via messages (method calls and events) over time. To understand the use and the functionality of these diagrams some sequence diagram symbols and notations are needed.

## Class roles

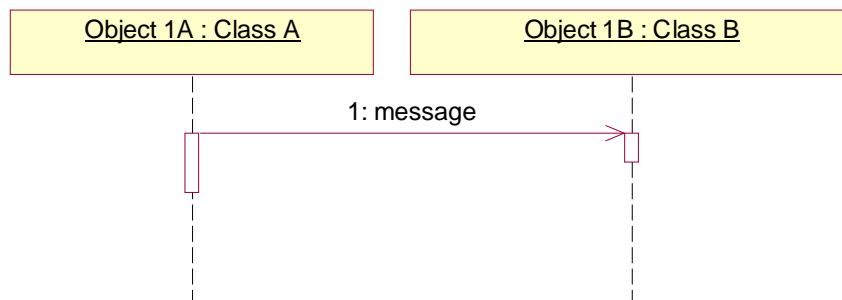
*Class roles* describe the way an object will behave in context. Class roles are illustrated by the UML object symbol (rectangles with the name and type (class) of the instance underlined), without its attributes and methods. There is a dashed line beneath each object. This line shows the lifetime of the object. Time flows from the head of the timeline, where the object is shown, down to the tail at the other end (Figure 1-6).



**Figure 1-6 Example of a class role**

## Activation

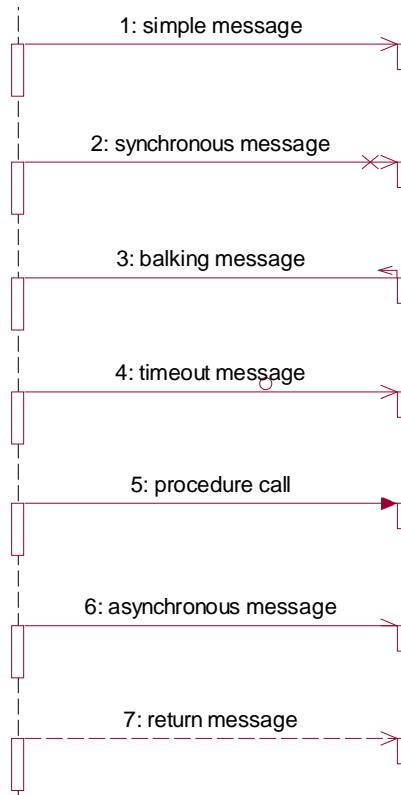
A rectangle on the timeline of an object means that the object has the focus of control. These rectangles are called *activation* boxes and represent the time an object needs to complete a task (Figure 1-7).



**Figure 1-7 Example of an activation box**

## Messages

Messages are displayed by arrows and represent communication between objects. There are seven types of message, shown in Figure 1-8. Any information described in a sequence diagram must conform to other diagrams involving the same object types. Every message must have a corresponding operation on that class.



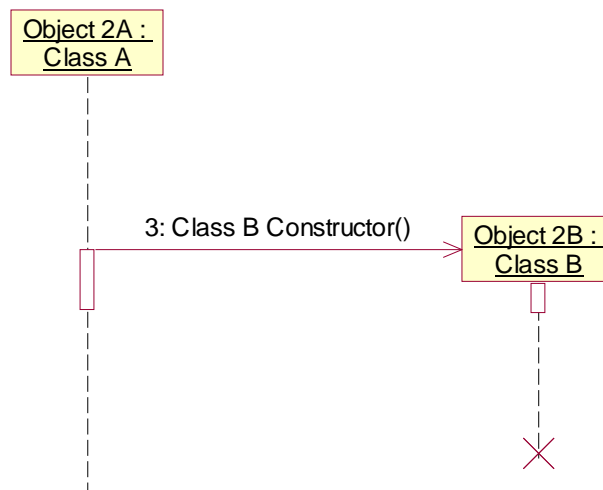
**Figure 1-8 The seven different types of messages**

### Creation and destruction

In a sequence diagram, not all of the objects involved exist from the beginning. During the execution of the use case, objects can be created and then released for garbage collection (Figure 1-9).

To show that an object is created, a message is drawn from the creator object to the head of the created object's timeline.

Objects can be terminated (ready to be garbage-collected) by drawing an X at the point where an object's timeline stops.



**Figure 1-9 Creating and terminating objects**

## Loops

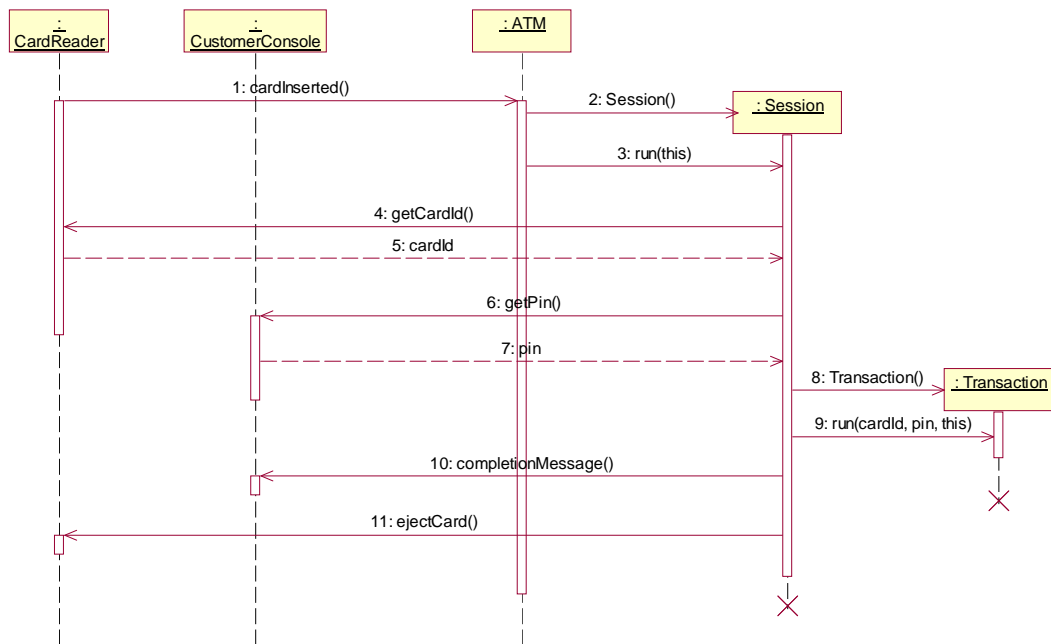
UML provides a notation for describing iterations and loops. A repetition or loop within a sequence diagram is depicted as a rectangle. The condition for exiting the loop is usually placed at the bottom left corner in square brackets [ ].

## Example

The following example illustrates the use of sequence diagrams. Assume that a bank customer wants to withdraw money from an ATM. A number of classes must be defined:

- Class CustomerConsole: The objects of this class represent the console of an ATM. The customer should interact with the user interface to enter his PIN.
- Class ATM: The objects of this class represent the ATM Controller.
- Class Session: Every time a customer uses the ATM an instance of this class is created, to service the customer.
- Class CardReader: CardReader objects just read the cards of the customers.
- Class Transaction: To get the transaction completed a Transaction instance must be created and destroyed

A sequence diagram based on the above classes describes how these objects interact with each other (Figure 1-10).



**Figure 1-10 Sequence diagram for a customer withdrawing money from an ATM**

This sequence diagram depicts only the scenario in which the transaction is successful. For every possible scenario there should be a sequence diagram.

### 1.2.2.3 Interfaces

The previous sections have shown that classes define the state and behaviour of their respective objects. Often, though, objects need to interact with other objects on different terms. Consider for example the inventory program of a retail store. In general, that program does not and should not care about what kind of objects it contains as long as the objects can provide information about their price. It can provide a protocol of communication, though, that classes have to implement in order to be compatible with the inventory. This protocol comes in the form of a set of method definitions contained within an *interface*, such as the 'get price' method. The interface defines but does not implement these methods. It is the job of other classes to implement these methods.

In general, an interface is used to define a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class



## Chapter 1.3 Linking models

The OpenMI standard defines an interface that allows models to exchange data at run-time. Before an exchange can take place, the participating models must be made OpenMI-compliant and the quantities that are to be exchanged must be identified and matched. The models can then be linked at run-time.

This chapter describes the way in which models can be linked and the changes that need to be made to existing models to make them OpenMI-compliant: it also introduces the request-reply mechanism, by which data are transferred between models at run-time and describes the `GetValues` method, which is at the heart of the data exchange process.

### 1.3.1 Linking models at run-time

Computational models are often viewed as software entities that transform input data into output data: for example, rainfall data into runoff data. This has been taken as the starting point for the OpenMI, which regards a model as an entity that can accept data and/or provide data. Linking models is thus interpreted as exchanging data between two model engines, taking care that output from one model fits the input requirements of the other model. This 'fit' should address both the data format as well as the scientific semantics.

Most current models receive data by reading input files and provide data by writing output files. This procedure is often adopted when creating a sequential link between models; that is, one model computes an entire time series and passes this series as input to the next model. However, to enable process interaction and feedback loops, models must run simultaneously and exchange data on a time basis. This is not possible with sequential links. Although a file-based approach could still be used at the timestep level of exchange, in most cases it would lead to unacceptable performance. Therefore, another approach has been selected for the OpenMI.

The OpenMI has adopted a component-based approach, in which the model is accessed directly at run-time, without using files for data exchange. This is achieved by making all models, databases and tools into components that support the same minimum set of properties, methods and events.

In designing the OpenMI the challenge has been to provide a standard, generic interface that allows models to exchange data when required. To meet this challenge, the process of linking and running linked models has been analyzed and broken down into four steps. They are:

- *Define*: Defining the components that can be linked and the data that they can *potentially* exchange (accept or provide – the receiving model only has to accept the quantities that it needs).
- *Configure*: Configuring the components (populated with input data) to be linked and specifying the *actual* data that will be exchanged between the components.
- *Deploy*: The run-time creation of components (populated with model data) in the memory of the target computer systems.
- *Execute*: Running the linked computation (i.e. simulating the interacting processes represented by the linked models).

To enable these four steps to be taken, the following functions and facilities are required:

- *Data definition*: In order that links can be established between models by users, the quantities that each model can potentially accept and provide must be defined and those definitions must be publicly accessible at both the program and user levels. The definition must include sufficient information so that the user can make a scientifically valid link and so that the programs can effect the transfer.
- *Generic model access*: In order that any model can pass data to any other model, the interface must be independent of the model's domain or the concept upon which it is based. Therefore, there must be a common generic interface that can be used for all linkable components.
- *Metadata*: Metadata is needed to inform others of the data that can potentially be provided and accepted by a linkable component.



- *Definition of exchanged data:* To establish a connection between two models a link mechanism is required to describe the data that will actually be transported between the two linkable components
- *Troubleshooting:* Facilities are needed to monitor the information flow and identify problems and their causes when something goes wrong.

The Open Modelling Interface and Open Modelling supporting software address all these requirements.

Having defined what the OpenMI does and is, it is perhaps useful to state what the OpenMI is not. It is *not* a common data-model specification, it does *not* contain scientific knowledge on process interactions and it certainly is *not* an integrated modelling system. However, the OpenMI can be used to create such integration.

## 1.3.2 The request-reply mechanism

The solution chosen for the OpenMI is a request-reply mechanism: i.e. a model 'replies to a request'. Therefore, to be OpenMI-compliant, a model needs to be transformed into an object or component that can reply to different questions. By implementing a number of relevant methods and properties this component can become a linkable component. For existing models, this is achieved by embedding the engine code within a standard wrapper. New models or codes can be developed directly as a component with the appropriate interface.

Linkable components can exchange data through this request-reply mechanism; a model that requires input asks a providing model for a set of *values* for a given *quantity* at a set of locations or *elements* for a given time. The providing model calculates these values and returns them. This section explains the mechanism in more detail.

### 1.3.2.1 The pull mechanism

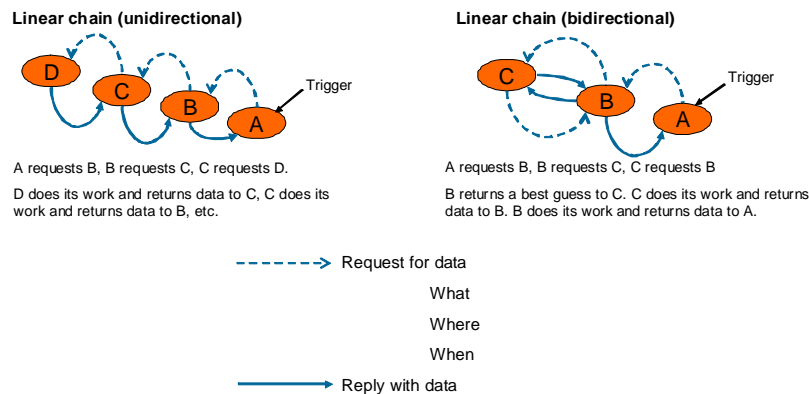
The OpenMI allows one component (e.g. a model) to 'pull' data that it needs from another component across a link. This is a very simple mechanism – just two components connected by a single link. Complex sets of interconnected components can be constructed by chaining several components using a number of links. Each component in the chain 'pulls' the data it needs from the model at the other end of its link. Models usually generate data at many points in space and time. The arguments of the pull method specify the particular point for which data are being requested.

The data that pass across the link are the output data or results of the providing model and form the input data or boundary conditions of the receiving model. 'Results' could be rainfall, water level, bottom level, water quality concentrations, fish population counts, vegetation covers or cost of water. It is worth noting here that the same mechanism can also be used to obtain data from or store data in a database – this will be explained in more detail later.

The OpenMI enables model engines to compute and exchange data at their own timestep, without any external control mechanism. Deadlocks are prevented by the obligation of a component always to return a value whatever the situation. When a model is asked for data, it decides how to provide them. The model may already have the data in a buffer because it has previously run the appropriate simulation; it may need to run its own simulation or calculation; it may make a best estimate by interpolation or extrapolation; or it may not be able to provide the requested data and so will raise an exception. The exchange of data at run-time is automated and driven by the pre-defined links, with no human intervention.

An important feature is that components always deal with requests in order of receipt. The possibility of the calculation sequence becoming confused therefore does not arise. This approach has much strength but, in particular, it is simple and gives freedom to the developer both at the domain and at the IT levels. Having freedom also implies accepting responsibility. The developer will be responsible for deciding how situations are handled where the requested data are either not available or not available at the requested time, at the requested location or in the requested unit of measurement. It is considered that the data supplier will, in general, know best how the available data should be processed to deliver a value for the requested time and place. The code developer thus decides the level of sophistication that is appropriate when interpolating or extrapolating to obtain the required value.

The examples in Figure 1-11 show how models can be chained and feedback loops accommodated using bidirectional links.



**Figure 1-11 Different chain layouts with the pull mechanism**

### 1.3.2.2 Other features

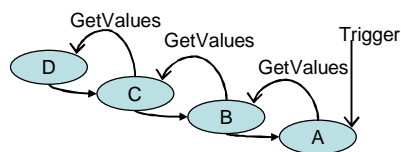
Certain other features of the linking mechanism should be noted:

- In some situations, components do not want to invoke computation, but they just want to monitor or visualize what happens. At such moments, a 'listener' mechanism is utilized. Models send events to signal when new values are available and based on such signals generic tools use the pull mechanism to retrieve the desired data. For example a graphing package could use this facility to update a graph.
- Before run-time, the links between components need to be created (either hard-coded or configured). For each model, you need to know what inputs the model requires and what outputs it can make available. During the configuration process, the links are defined between particular pairs of models, including an indication of which data will be exchanged across the link and in which direction. The information on the configured links is passed to the run-time processes. Human involvement will be needed when the links are being specified (to decide what should be connected to what); however, some tools are available to support this process.
- The OpenMI describes both sides of a link in terms of the source component and the target component. Hence, different quantity or variable names can be applied on either side, as long as their meanings, in scientific terms, are the same. The dimensions of a quantity need to be described to reduce the risk of different semantics and unit inconsistencies.
- An OpenMI-compliant model may support the ability to hold its own status. The OpenMI provides methods by which models can be asked to save their current state and revert to a previously saved state. This feature creates opportunities for iteration and optimization. Note that, given the implications for the underlying code, support of this feature is optional. An exception can be expected if this method is called but not supported.
- The OpenMI supporting software allows you to combine collections of links in a *composition*, which can be created, modified, stored, executed and applied for scenario analysis.

### 1.3.3 The GetValues method

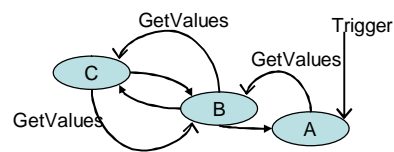
The key to run-time data exchange is the *GetValues* method illustrated in Figure 1-12. When one model requires data from another model, the first calls the *GetValues* method of the second. The illustration shows the application of the *GetValues* method in a variety of typical modelling situations. The examples show how models can be chained and feedback loops accommodated. In a linked model run, one model will be nominated to act as the trigger that starts the run. When the calculation reaches a point where data are required from another model, the *GetValues* method of that model is used to request the required data – see the Uni-directional Linear Chain in the illustration.

Linear chain (uni-directional)

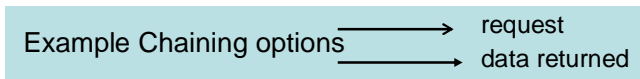


A requests B, B requests C, C requests D  
 D does its work and returns data to C, C does its work and returns data to B, etc.

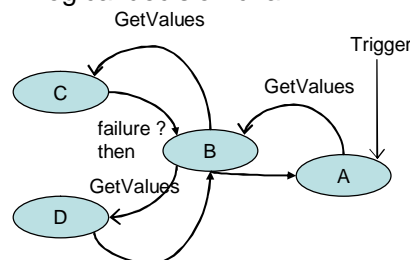
Linear chain (bi-directional)



A requests B, B requests C, C requests B  
 B returns a best guess to C, C does its work and returns data to B, B does its work and returns data to A



Logical decision chain



A requests B, B requests C,  
 C does its work and returns data to B  
 if C fails B requests D  
 B returns data to A

**Figure 1-12 Data exchange between models**

Several situations can now arise. If the model has already computed the requested data, it will return them to the requesting model. If it has not computed them, then the model will run until it can return them, obtaining data from other models in the process as necessary. However, it may be that the model cannot run because it is waiting for data, for example from an instrument in the field. In this case, the model must extrapolate in order to return a value.

A similar situation arises when a model answering a request needs data from the requesting model. This can occur, for example, in backwater calculations where, in order to compute the flow out of a river reach, it is necessary to know the level in the downstream reach. However, the level in the downstream reach is dependent on the flow from the upstream reach. In such cases, an iterative process is required to reach a solution. OpenMI-compliant models are able to perform such iterations because there is a requirement that they should be able to save their status at any point and be able to revert to any previously saved status upon request.

An important situation which might arise is that the requesting model asks for data at a point in space and time that does not match the calculation points in the requested model; for

example, one model could be running on an hourly timestep while the other is on a daily timestep. In this case, the requested model must interpolate and return the required values. In all cases, the returned values will be qualified so that the requesting model can assess their reliability.

Before returning the values, the requested model will make any necessary unit conversions. It will also map the data from the elements of the requested model to those of the requesting model. The possibility of the calculation sequence becoming confused does not arise, as the GetValues method always deals with requests in order of receipt. Figure 1-13 shows a schematic representation of the processes described.

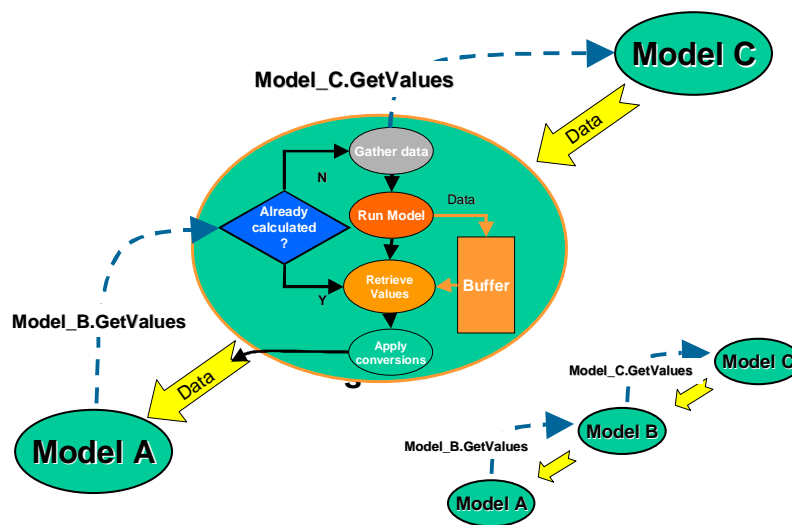


Figure 1-13 Handling the GetValues process



## Chapter 1.4 Developing OpenMI systems

For a model to become OpenMI-compliant, its engine must be transformed into a component supporting the OpenMI standard interface. This chapter introduces the steps in developing new OpenMI models and migrating existing models to the OpenMI standard. The steps will be explained in detail in Books 3 and 4.

## 1.4.1 Overview of an OpenMI-compliant system

This section provides an introduction to OpenMI systems and OMI files, which are used to store information about OpenMI components.

### 1.4.1.1 OpenMI systems

An *OpenMI system* is any software application that includes a set of one or more OpenMI-compliant components. Such systems can link to other OpenMI-compliant models through their standard interface. In order to do this, the OpenMI system must incorporate the following knowledge and functionality:

- The system must know where it can find linkable components.
- The system must know what links exist between linkable components.
- The system must be able to instantiate, deploy and run a combination of linkable components.

A *configurable* OpenMI system is one that is able to inspect the exchange items in a linkable component and hence, for example, provide drag-and-drop style facilities for model linking.

### 1.4.1.2 The OMI file

The knowledge identified above is stored in the OpenMI components' *OMI files*. An OMI file is an XML file that contains the information needed to instantiate the component and populate it with input data.

Figure 1-14 shows a simple example of an OMI file. At this stage, it is only necessary to be aware of the existence of OMI files; it is not important to understand them.

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

<LinkableComponent Type="wldelft.OpenMI.WLinkableComponent" Assembly="wldelft.OpenMI,
  Version=1.4.0.0, Culture=neutral, PublicKeyToken=8384b9b46466c568"
  XMLns="http://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 1-14 OMI file example**



## 1.4.2 Deployment phases

The deployment of an OpenMI linkable component includes the following phases:

- *Instantiation and initialization.* The application reads the OMI file and constructs the linkable component. The component is then populated with input data.
- *Inspection and configuration.* Available exchange items are examined. Links and other objects are created and added to the component. The status of the component and the status of its links are validated.
- *Preparation.* This phase completes any preparatory work before the main computation process starts. For example, database and network connections are established, output files are opened and buffers are organized.
- *Computation/execution.* This phase consists of a loop that is executed for each timestep. For each pass through the loop, any necessary calculations are performed and data are exchanged with other linkable components.
- *Completion.* This phase is invoked when the computation/execution loop has been completed. Files and network connections are closed, memory is cleaned up and so on.
- *Disposal.* This phase is entered when the application is closed. Remaining objects are removed and memory is de-allocated.

Full details of these phases are given in Book 3.

### 1.4.3 Migrating existing models

This section specifies the criteria that an existing model must satisfy in order to become an OpenMI linkable component. The steps in the migration process are also outlined.

#### 1.4.3.1 Criteria for becoming a linkable component

As described earlier, the OpenMI defines an interface for data exchange between models. In order to make this possible, the original *engine* needs to be turned into an *engine component* and the engine component needs to implement the OpenMI interface so that the quantities calculated by the component become accessible to other components. The engine component then becomes an OpenMI-compliant linkable component.

A similar pattern can be applied for databases or other kinds of data sources. By turning them into components that implement the OpenMI interface, they become linkable components that provide direct access to their data at run-time.

To become an OpenMI linkable component, a model must satisfy the following criteria:

- The model must be structured in such a way that initialization is separate from computation, with boundary conditions being collected in the computation phase and not during initialization.
- The model must be able to expose information to the outside world on the modelled quantities it can provide.
- The model must be able to provide the values of the modelled quantities for any requested point in time and space.
- The model must be able to respond to a request, even when the component itself is time-independent; if the response requires data from another component, the component must be able to pass on the time in its own request.
- The model must be able to submit to run-time control by an outside entity.

For components progressing in time, the requirement 'always' to return values when requested imposes the following conditions:

- The delivering component must know what time it has reached. It must recognize whether it has not yet reached the requested time, it is at the requested time or it has passed the requested time. Depending on the model and the context, the model will thus know whether to extrapolate, to compute up to the requested time or to search its buffer (if available).
- Components must be able to interpolate if the requested time is not in their own timestep or space frame.
- Components must know when they are waiting for data, in which case they will have to return an extrapolated value.

The easiest way to make a model compliant with the OpenMI is to contain it in a suitable wrapper. The wrapper controls the run-time activity of pulling data across links. The OpenMI Software Development Kit provides a 'smart wrapper' that already handles most of the tedious (and difficult) tasks to be performed.

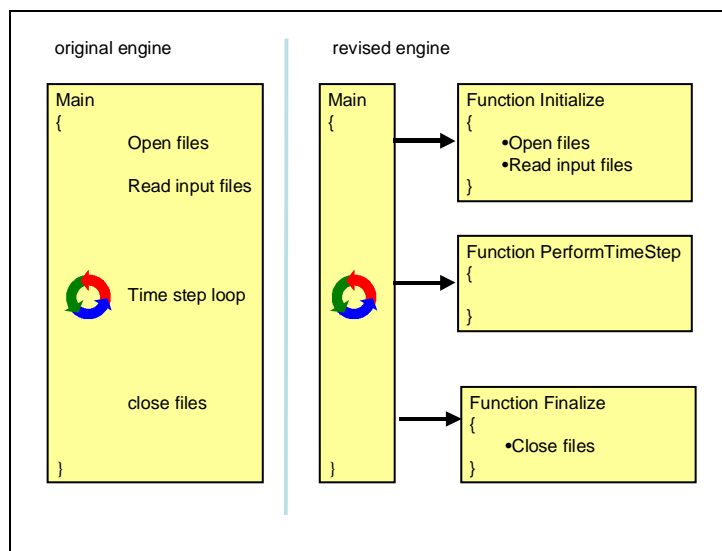
### 1.4.3.2 The migration process

The OpenMI Software Development Kit contains a number of utilities to ease the task of converting an existing model engine into an OpenMI linkable component. You do not have to use these utilities when creating an OpenMI-compliant component but they may make the task simpler.

Before starting the migration process, you should have a clear idea of how your model will be used and how it may be linked to other OpenMI components. In particular, you should define the exchange items for your component: these are the input data that it will require and the output data that will be made available to other components.

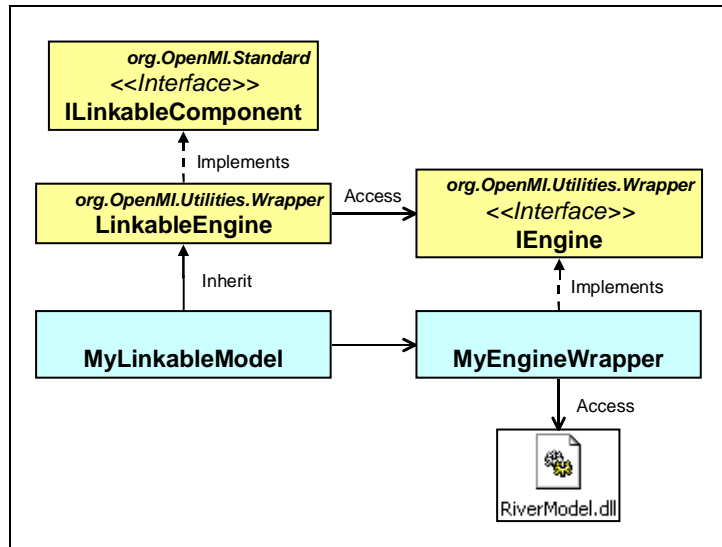
The migration process consists of seven steps:

1. *Change the engine core.* The model engine should be converted from an EXE file to a DLL, which can be accessed by other components (Figure 1-15).



**Figure 1-15 Changing the engine core**

2. *Create the .NET assemblies.* After installing the OpenMI Software Development Kit, create assemblies for wrapper classes and test classes in the .NET development environment (Figure 1-16).



**Figure 1-16 Wrapper classes**

3. *Access the functions in the engine core.* The engine needs to be accessible from .NET. The MyEngineDLLAccess class makes a one-to-one conversion of all exported functions in the engine core code to public .NET methods.
4. *Implement MyEngineDotNetAccess.* This class changes the calling conventions to C# conventions and converts error messages into .NET exceptions.
5. *Implement the wrapper class.* The MyEngineWrapper class implements the ILinkableEngine interface.
6. *Implement the linkable component.* The MyModelLinkableComponent class must be implemented. This class defines the linkable component that is accessed by other models.
7. *Implement the remaining IEngine methods.* The remaining methods in the MyEngineWrapper class must be implemented. In some cases you may need to make changes to the engine core as well as adding code to the IEngine methods.

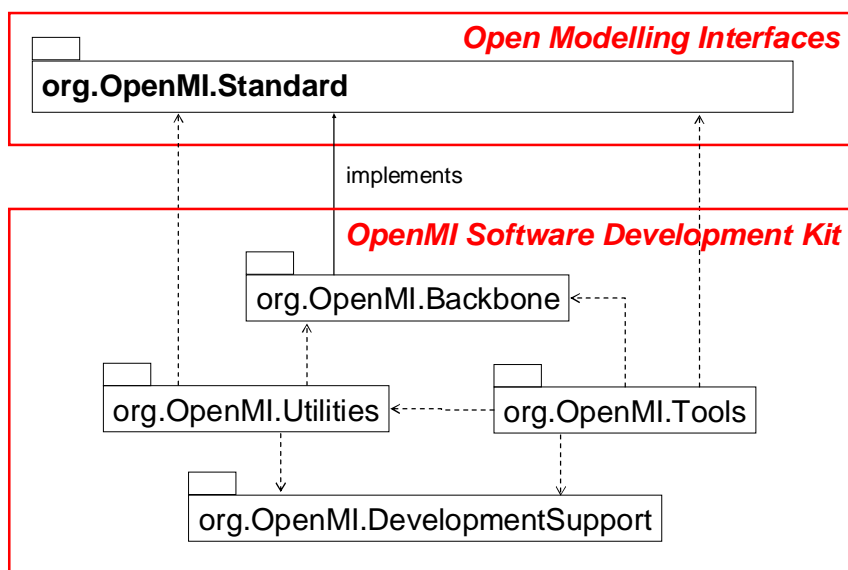
A full description of this process, with examples, is given in Book 4.

## 1.4.4 Implementations of the OpenMI

The OpenMI is defined as an interface in the `org.OpenMI.Standard` namespace. This allows anyone to implement the OpenMI in their own way. Software components that implement and use these interfaces are called *OpenMI-compliant*.

However, to make it easier to be OpenMI-compliant there is default implementation in the .NET framework using C#. This implementation is being released as open source under Lesser GPL license conditions. A less comprehensive implementation in Java will also be available.

### OpenMI architecture



**Figure 1-17 OpenMI architecture namespaces**

This default implementation, called the *OpenMI Software Development Kit*, is composed of a number of software packages (see Figure 1-17):

- The `org.OpenMI.Backbone` package provides the minimum set of classes required to implement the standard interface.
- The `org.OpenMI.Utilities` namespace provides utilities to support the wrapping of legacy code, to manipulate data sets and to configure and deploy the components.
- The `org.OpenMI.DevelopmentSupport` namespace contains generic software to enable the parsing of XML-files containing compositions of OpenMI components.
- The `org.OpenMI.Tools` namespace contains a group of front-end tools to enable interaction with the system.

However, it is emphasized that there is no requirement to use the OpenMI Software Development Kit to develop OpenMI-compliant components.



# Book 2 Exchanging data

<b>BOOK 2</b>	<b>EXCHANGING DATA.....</b>	<b>2-1</b>
<b>Chapter 2.1</b>	<b>Data exchange at run-time .....</b>	<b>2-3</b>
2.1.1	The data exchange mechanism .....	2-4
2.1.2	The role of element sets in data exchange .....	2-6
2.1.3	Bidirectional links.....	2-7
2.1.3.1	Example 1: Linkage of two dynamic river flow models .....	2-7
2.1.3.2	Example 2: Linkage of a river model with a plant growth model .....	2-7
2.1.3.3	Example 3: Linkage of a river model with a weir control module .....	2-8
<b>Chapter 2.2</b>	<b>Describing the exchange data.....</b>	<b>2-9</b>
2.2.1	Introducing the use case .....	2-10
2.2.2	What to describe.....	2-11
2.2.3	Defining what the values represent .....	2-12
2.2.4	Defining where the values apply .....	2-15
2.2.4.1	OpenMI ElementSets.....	2-15
2.2.4.2	Using different types of elements .....	2-17
2.2.4.3	Choosing an ElementType.....	2-20
2.2.4.4	Dynamic ElementSets.....	2-21
2.2.5	Using data operations to describe how data can be mapped .....	2-22
2.2.6	Grouping into ExchangeItems .....	2-26
2.2.6.1	ExchangeItems.....	2-26
2.2.6.2	Initially unknown ExchangeItems .....	2-27
2.2.7	An advanced example .....	2-31
<b>Chapter 2.3</b>	<b>Configuring links and compositions .....</b>	<b>2-35</b>
2.3.1	Configuring a single link.....	2-36
2.3.2	Building a composition.....	2-38
<b>Chapter 2.4</b>	<b>Using the OpenMI configuration editor .....</b>	<b>2-39</b>
2.4.1	Starting the configuration editor .....	2-40
2.4.2	Adding models to the composition.....	2-41
2.4.3	Establishing connections between the models .....	2-43
2.4.4	Configuring the connections.....	2-44
2.4.5	Adding a trigger .....	2-45
2.4.6	Running the composition .....	2-46





## Chapter 2.1 Data exchange at run-time

The GetValues function is the essence of the OpenMI data exchange mechanism. This function allows the exchange of data between two linkable components.

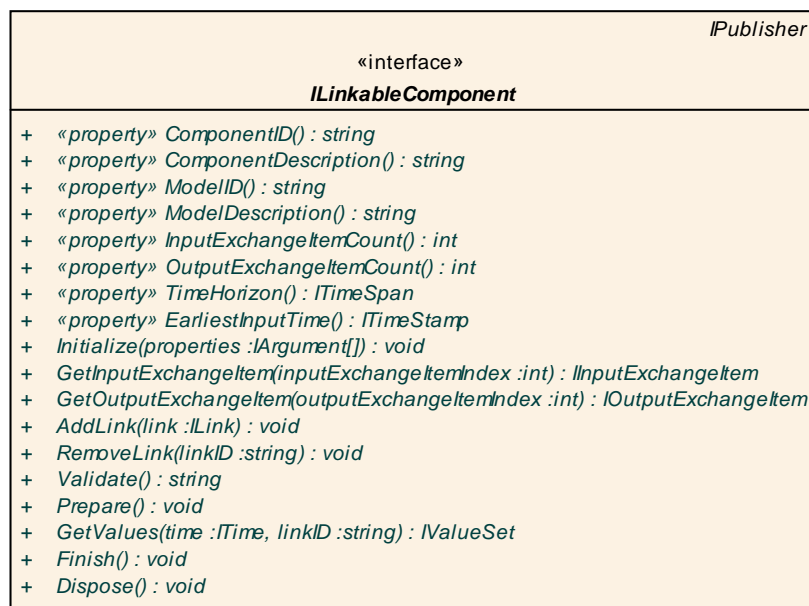
This chapter discusses the syntax of the GetValues function and provides details of the information needed in order to link two OpenMI-compliant models. In particular, it discusses the role of element sets, which define the locations for which data is to be exchanged. The way in which bidirectional links can be handled is also covered.

## 2.1.1 The data exchange mechanism

Connecting two linkable components is a complex procedure, requiring the fulfilment of several conditions in order to be successful and efficient. One of the key issues in this process is the triggering of the data exchange operation and the specification of the information to be exchanged. This forms the fundamental operation principle of the OpenMI: the *pull mechanism*.

There are three parties involved in each data transfer operation: the requesting component, the providing component and the link between them. According to the OpenMI specification, when the requesting component comes to a point in its work where it needs information that has to be delivered by the providing component, it issues a *GetValues* call. The providing component will then proceed in calculating the required value and pass it over the link.

The *GetValues* function is defined in the *ILinkableComponent* interface specification (Figure 2-1) as *GetValues(time: ITime, linkID: string): IValueSet*. Its syntax is both simple and precise and can be translated as: 'Provide the value(s) of the required quantity for the required timestep (or time span) at the required location(s)'.



**Figure 2-1 The OpenMI interface definition of the Linkable Component**

The elements of the *GetValues* function are as follows:

- The timestep or time span is described by the *time* parameter. According to the architectural specification, its data type is *ITime*, which can be either a timestamp expressed in the Modified Julian Date format or a time span, specified by the start time and end time, also expressed in the Modified Julian Date format.
- The components involved as well as the location and the quantity of the required value(s) are hidden inside the link specification (*ILink*).

- The components involved are contained in the SourceComponent and the TargetComponent. Note that the target requests the source for data while the numerical values flow from the source to the target.
- The quantity to be exchanged is contained in the SourceQuantity and TargetQuantity properties of the link.
- The location of the values is described by the SourceElementSet and the TargetElementSet properties of the link.
- The returned values are in IValueSet format, which is effectively one or more scalar or vector values.

It is apparent that the syntax of the GetValues command is straightforward, requiring only the necessary parameters and hiding any inherent technicalities within the definition of the link.

In any linkage the providing component needs to take care that it can deliver the data as requested by the acceptor. In addition to the identification of components, the TargetQuantity and the TargetElementSet are essential. The link object also contains the SourceQuantity and the SourceElementSet to ensure that the end user has control of the information source. By including the source information, an alias table is created to map semantics without the need for agreement on variable names.

## 2.1.2 The role of element sets in data exchange

The element sets have a key role in a coupled model simulation as they represent the points of information exchange. This section describes the way in which element sets are defined.

A link between two linkable components in the OpenMI context is not just a pipe of information but rather an intelligent data path between two precisely defined locations. These locations are described using the `IElementSet` interface. But what does an element set really stand for?

The first thing to consider is the physical aspect of an element set. In real life problems the interaction between two (or more) physical entities is usually not limited to a single exchange point but rather stretches across several locations, varying in dimension (point, line, surface) and exchanged quantity. For example, a groundwater model can provide the groundwater level at a specific point (single value) or as the average value over a specific polygon. Therefore, when defining interactions using model abstractions, one has to specify not only the location but also the type and other properties of this interaction.

The OpenMI `IElementSet` aims to provide a flexible descriptor for data exchange locations. To this effect, an element set is an ordered collection of elements (one or more), described by an ID and possibly by a text description. Each element can be either a simple node without any geometrical properties or it can be a point, a poly-line, a polygon or even a three-dimensional shape. Furthermore, it may have a spatial reference or not. However, an element set can only contain elements of the same type.

The element set does not only fulfil the need for describing a complex physical interaction but also provides a significant computational optimization: within one call of the `GetValues` method, you can exchange values of a quantity across multiple locations at once and not iteratively for each location. This is where the `ValueSet` comes in: for each `ElementSet` there is a corresponding `ValueSet` and its values are ordered in the same order as the `ElementSet`. Using this one-to-one mapping, references to quantities at a specific location become easy and efficient.

### 2.1.3 Bidirectional links

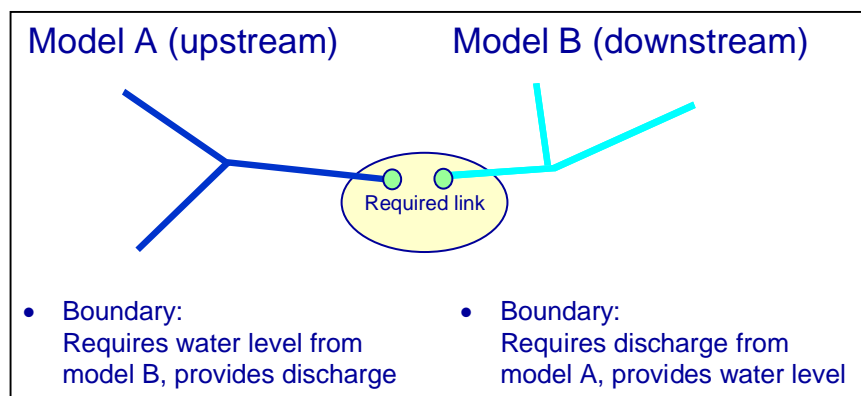
Although OpenMI links are unidirectional, many simulations require *bidirectional* data exchange. This section presents some typical examples using bidirectional links and briefly discusses the OpenMI way of handling them.

In the OpenMI architecture the link between two linkable components is considered as a unidirectional data path between the source and target components. However, there are many cases where two linked components may need to pass data to each other in both directions.

#### 2.1.3.1 Example 1: Linkage of two dynamic river flow models

Dynamic river flow models are models that simulate the flow in rivers (discharge volume and water level), usually based on the Saint-Venant equations. Developing such models is a complex task and many models have been developed for parts of rivers. Instead of trying to expand such models to cover larger parts of river basins in a single model, linking the different regional models is an option, which can be accomplished using the OpenMI. The benefit of such a linkage is that regional models remain available for separate use and new or improved regional models can simply be integrated in the larger scale model.

In a standalone regional model a discharge timeseries is usually imposed at the upper boundaries, which are the upstream 'inlets' of the model. The water level is calculated. At the 'outlets' of the model a water level timeseries is commonly imposed (lower boundary). If two models are linked, the water level timeseries of the lower boundary of the upstream model is replaced by the calculated values of the downstream model and the discharge timeseries of the downstream model is replaced by the upstream model results. This means that at the same time the upstream model requires a value for the water level, and the downstream model requires a value for the discharge (Figure 2-2).



**Figure 2-2 Linking two river flow models**

Such interdependence at the same timestep results in a bidirectional link.

#### 2.1.3.2 Example 2: Linkage of a river model with a plant growth model

A key feature of a dynamic river model is the bed roughness, which is the resistance of the river bed or floodplains. This roughness depends on plant growth, which itself depends on

flow velocities, among other things. Linking these two types of model thus implies that at the same timestep in the calculations the flow model requires roughness, while the plant-growth model requires velocities.

Such an interdependence at the same timestep results in a bidirectional link.

### 2.1.3.3 Example 3: Linkage of a river model with a weir control module

Dynamic river models are used for real-time flood prevention. Based on the computations of the models preventative measures are taken, such as lowering or setting up weirs. A control module receives the water levels from the river model. The river model receives the level to which weirs are set. If the data used are instantaneous, this is another example of a bidirectional link.

A precise definition of a bidirectional link can be formulated as a link between two components where at the same timestep each component requires the other component's output in order to complete its calculation.

Clearly a bidirectional link leads to a deadlock in the numerical calculations. However, the OpenMI architecture provides a solution to this issue: using extrapolation, one of the involved components can calculate the quantity it has been asked for and thus end the deadlock.

More information about the data exchange pattern for bidirectional links can be found in Part C Section 3.3.5.2.

## Chapter 2.2 Describing the exchange data

To use the `GetValues` function you must be able to define the data that is to be exchanged, its location and the time period for which data is required. This chapter gives a detailed description of how these parameters can be determined.

The chapter also introduces the concept of `ExchangeItems`, which are a combination of the quantities to be exchanged and the locations (element sets) for which the data is exchanged.

## 2.2.1 Introducing the use case

The example described in this chapter is based on a common use case (Figure 2-3). A lumped rainfall-runoff model is applied to calculate lateral inflows in a river system. The rainfall-runoff model receives precipitation from a monitoring database. The river model computes the water levels and stages along the river. Throughout this book, the example is extended with data coming from a weather forecasting system and a groundwater model that interact with the rainfall-runoff and river models.

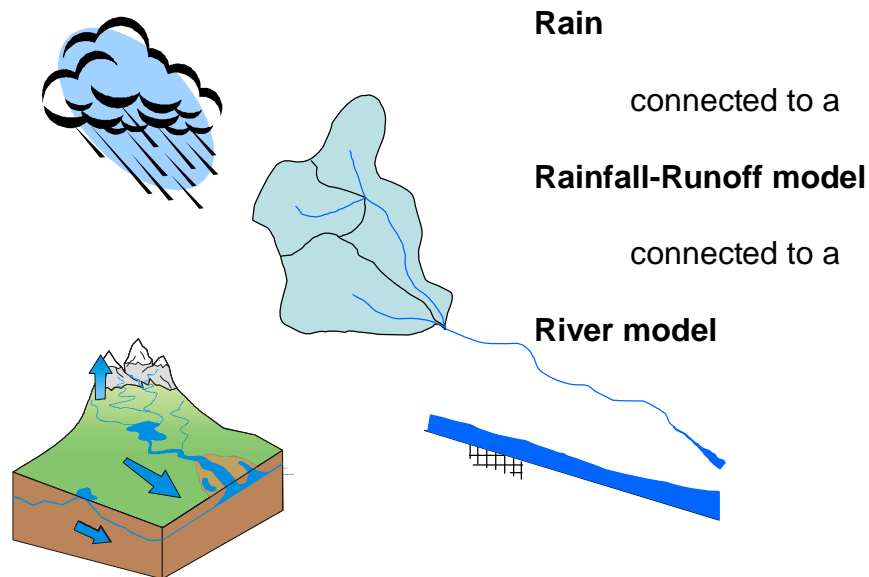
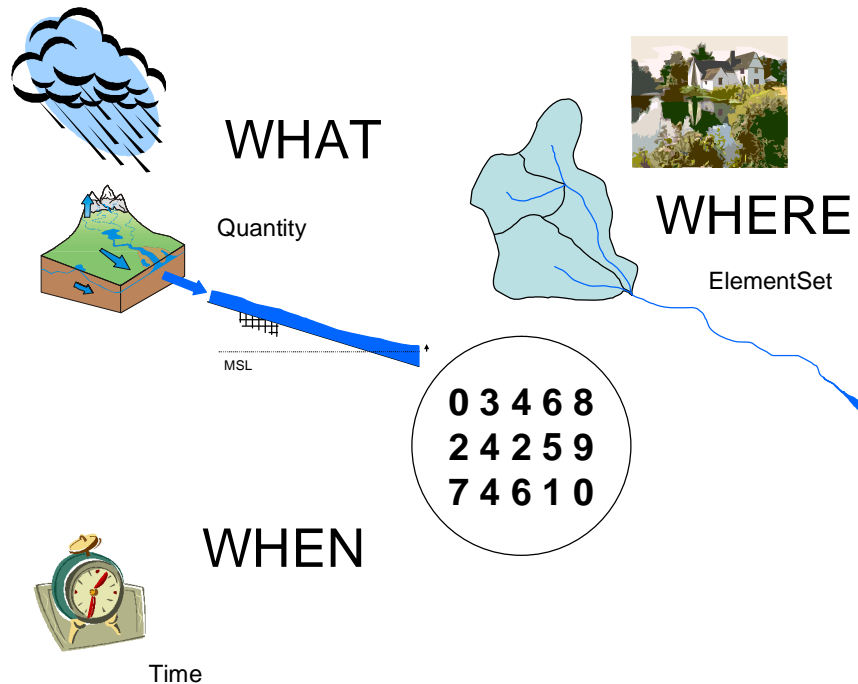


Figure 2-3 A common use case



## 2.2.2 What to describe

Model linkage by means of data exchange requires that both the providing and accepting models have the same understanding of the values that are being exchanged. Exchanging a set of values without understanding their meaning is useless. Figure 2-4 illustrates the basic entities in the OpenMI used to describe the meaning of a set of values.



**Figure 2-4 Values and their semantics**

The values themselves are scalars or vectors. Their meaning is identified by three axes, namely what, where and when:

- *What* the values represent and in what unit they are expressed is indicated by the *quantity* and its *unit*. A *dimension* is added to enable validation of this aspect.
- *Where* these values apply is indicated by the *ElementSet* class, which contains an ordered set of *elements*. Each element is defined by an ordered set of *nodes*. These nodes may be geo-referenced with *co-ordinates* (but do not need to be).
- *When* the values apply is indicated by the *time*, either expressed as an instantaneous moment in time (a *timestep*) or a period over time (a *time span*).

## 2.2.3 Defining what the values represent

Within the world of modelling, a wide range of terms is used to indicate 'what' the values represent. Typical terms applied are *variable* or *parameter*. As different views exist on the meaning of those terms and when they can be applied, the OpenMI has chosen to use the term *quantity*, as the values typically represent a quantity (whether this is a decision variable, an input variable, an input parameter or an output variable). An extensive metadata structure has been defined to enable a complete and explicit description of what the values represent (Figure 2-5). The metadata structure has been made extensive as the OpenMI wants to allow scientifically sound linkages of semantically similar quantities without forcing a 'standard' data dictionary. Although the expertise of the person setting up the link will also be needed, features such as a dimension check have been included.

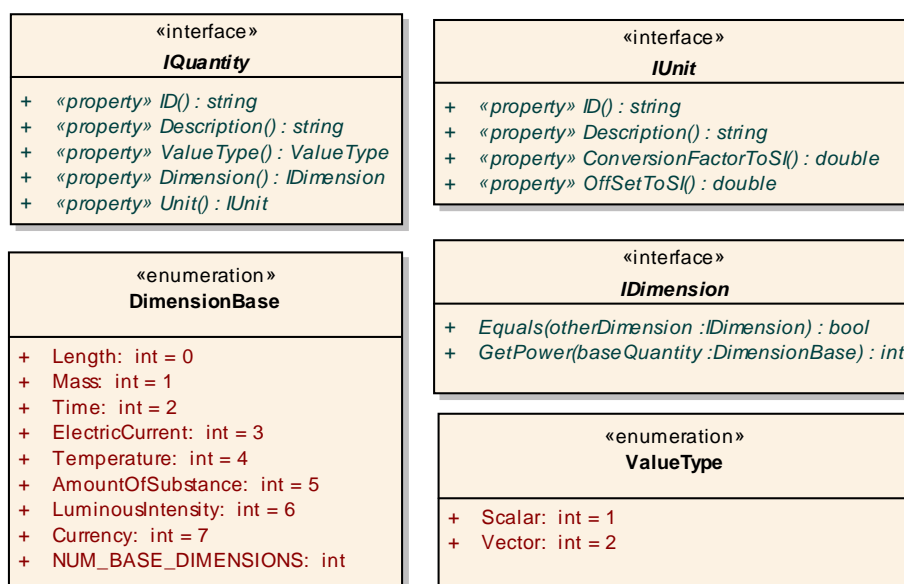


Figure 2-5 The OpenMI interface definition of the Quantity and associated metadata

The string ID of the quantity and the unit are meant to contain the short name or an understandable abbreviation of it. (A *unit* has a definite magnitude and can be used as a basis for measuring other things. The millimetre is a unit. The metre is a different unit, because it has a different magnitude.) A more extensive explanation of the quantity or unit can be provided using the Description property.

Currently, the OpenMI supports two types of values: scalars (expressed as doubles) and vectors (expressed in terms of X, Y and Z components).

Unit conversions are commonly needed when exchanging data between models. Within the OpenMI a methodology has been chosen which enables unit conversion when needed, while it does not force unit conversion at all times. The Unit object 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:

$$s = \text{Unit.GetConversionFactorToSI}() * v + \text{Unit.GetOffsetToSI}()$$

To enable (physical) dimension checks between quantities, an explicit definition of the dimension is incorporated. (A *dimension* describes the type of thing being measured, without

specifying the magnitude. The millimetre and the metre both have dimensions of length.) A dimension is expressed as a combination of base quantities, derived from the SI system, with a minor extension for currencies.



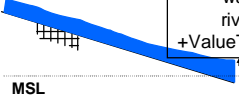
This interface provides a method to obtain the power of each base quantity, as well as a method to check if two dimensions are equal. For example, a discharge expressed in unit  $m^3/s$  has dimension  $Length^3Time^{-1}$ . Table 2-1 illustrates the base quantities and the associated SI units.

Note that some units are dimensionless, represent logarithmic scales or have other difficulties when expressed in SI. In that case you should pay extra attention to the descriptive part of a unit, to ensure that the user defining the link has a proper understanding of the quantity. Dimensionless units can be represented by power '0' for all base dimensions.

**Table 2-1 Base units and base quantities in the OpenMI (derived from SI)**

Base quantity	SI base unit	Symbol used
Length	metre	m
Mass	kilogram	kg
Time	second	s
ElectricCurrent	ampere	A
Temperature	Kelvin	K
AmountOfSubstance	mole	mol
LuminousIntensity	candela	cd
Currency	Euro	E

Figure 2-6 provides an example of how to use the classes to describe various quantities of the use case.

Quantity	Unit	Dimension
 +ID = "Rainfall" +Description = "Incoming rainfall on a catchment" +ValueType = Scalar	+ID = "mm/h" +Description = "millimeters per hour" +ConversionFactorToSI = "2.778e-7" +OffsetToSI = "0" (mm/h=0.001 m/3600 s)	-Length = 1 -Mass = 0 -Time = -1 -ElectricCurrent = 0 -Temperature = 0 -AmountOfSubstance = 0 -LuminousIntensity = 0 -Currency = 0
 +ID = "Outflow" +Description = "Outflow from catchment" +ValueType = Scalar	+ID = "m3/s" +Description = "cubic meters per second" +ConversionFactorToSI = "1" +OffsetToSI = "0"	-Length = 3 -Mass = 0 -Time = -1 -ElectricCurrent = 0 -Temperature = 0 -AmountOfSubstance = 0 -LuminousIntensity = 0 -Currency = 0
 +ID = "WaterLevel" +Description = "water level along river" +ValueType = Scalar	+ID = "m+MSL" +Description = "meters above Mean Sea Level" +ConversionFactorToSI = "1" +OffsetToSI = "0"	-Length = 1 -Mass = 0 -Time = 0 -ElectricCurrent = 0 -Temperature = 0 -AmountOfSubstance = 0 -LuminousIntensity = 0 -Currency = 0

Note: public properties are preceded by a '+' sign, private properties by '-' sign

Figure 2-6 Description of quantities for the use case

## 2.2.4 Defining where the values apply

This section explains how to define 'where' the values apply and which direction conventions apply for the understanding of positive and negative values. It introduces the concept of the ElementSet.

### 2.2.4.1 OpenMI ElementSets

Understanding where the values apply is crucial for appropriate linkages. Typically, models have their representation of space. Some models do not vary over space at all; others apply a set of unrelated calculation units, while the majority incorporate some topology by representing the spatial geometry as a network, a structured grid or an unstructured grid/mesh. A *structured grid* can be represented as a 2D or 3D matrix, where an element in the matrix can always determine the position of its neighbouring element using its own position (i,j) in the matrix. An *element* in an unstructured grid does not have such an (i,j) position in a matrix. Typically, a lookup table, holding topological information (IDs or co-ordinates) needs to be queried to find the neighbouring element.

Instead of fixing data structures for these kinds of common spatial representations in computational modelling, a more flexible method has been applied, accounting for the fact that most data is exchanged to a subset of the boundaries of the model or to the full geometry.

The method takes the list of calculation units, called an ElementSet, as a starting point. Within an ElementSet, all individual elements need to have the same type. The following element types have been identified: ID-based, XYPoint, XYLine (i.e. line segments), XYPolyLine, XYPolygon, XYZPoint, XYZLine, XYZPolyLine, XYZPolygon, XYZPolyhedron. ID-based elements may be non-geo-referenced, while elements based on a GIS-primitive type are geo-referenced, requiring co-ordinates in a spatial reference system. (GIS-primitive based elements may have (internal) IDs as well but that is not their primary way of identification within the element set.)

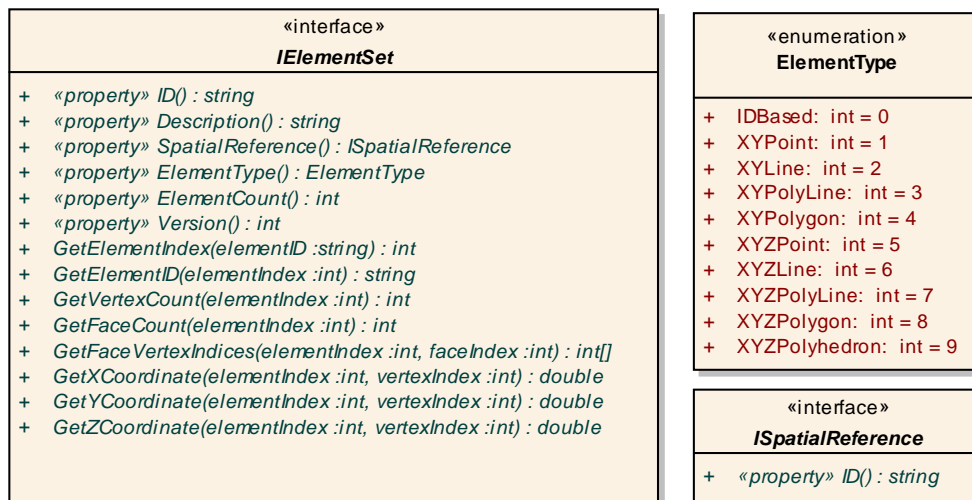
Table 2-2 describes in more detail the conventions that apply with regard to the ordering of vertices (i.e. nodes) that contain the co-ordinates of an element.

**Table 2-2 OpenMI enumeration of element types**

<b>ElementType</b>	<b>Convention</b>
IDBased	ID-based (string comparison)
XYPoint	Geo-referenced point in the horizontal (XY)-plane
XYLine	Geo-referenced line-segment connecting two vertices (nodes) in the horizontal (XY)-plane (start and end vertices indicate the direction of any fluxes)
XPolyLine	Geo-referenced polyline connecting at least two vertices in the horizontal (XY)-plane, open (start and end vertices are not identical and indicate the direction of any fluxes)
XPolygon	Geo-referenced polygons in the horizontal (XY)-plane, vertices defined anti-clockwise, closed (start and end vertices are identical)
XYZPoint	Geo-referenced point in 3-dimensional space (XYZ)
XYZLine	Geo-referenced line-segment connecting two vertices (nodes) in 3-dimensional space (XYZ) (start and end vertices indicate the direction of any fluxes)
XYZPolyLine	Geo-referenced polyline connecting at least two vertices in 3-dimensional space (XYZ), open (start and end vertices are not identical and indicate the direction of any fluxes)
XYZPolygon	Geo-referenced polygons in 3-dimensional space, vertices defined anti-clockwise, closed (start and end vertices are identical)
XYZPolyhedron	Geo-referenced polyhedron (closed volume of any shape) in 3-dimensional space, vertices for each face defined anti-clockwise

Note that the XY-ElementTypes are a simplified version of the XYZ-ElementTypes.

The exact OpenMI interface with associated interfaces is displayed in Figure 2-7. 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 of inter-element connections. Therefore you cannot assume by default that the IElementSet interface enables complete inheritance of a model grid for all purposes.

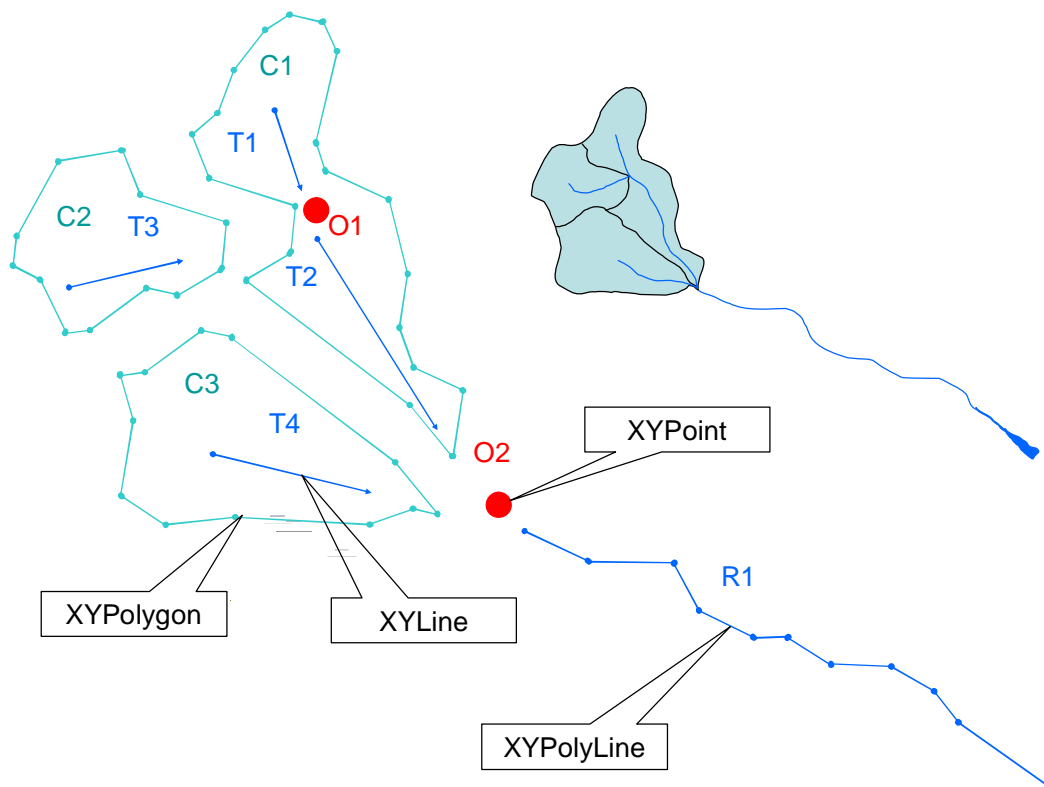


**Figure 2-7 The OpenMI interface definition of an Element Set**

#### 2.2.4.2 Using different types of elements

Elements of the same type can be combined in an ElementSet. A network can thus be defined as a collection of elements of type XYLine or XYPolyLine, while grids and meshes can be represented as a collection of XYPolygon elements. A hybrid-schematization, mixing networks with grids, has to be exposed in at least two ElementSets (i.e. one for the network and one for the grid). A non-geo-referenced model can be represented as a set of ID-based elements.

Figure 2-8 illustrates how various types of elements can be applied to provide information on the spatial representation of the catchment, its internal tributaries and the river.



**Figure 2-8 Illustration of various types of element set**

The subcatchments of the rainfall-runoff model are represented by an ElementSet of three XYPolygon elements, namely C1, C2 and C3. The outlets of the subcatchments form an ElementSet of two XYPoint elements, namely O1 and O2. Although not necessary for linkages, the tributaries of the subcatchments (T1,T2,T3 and T4) have been displayed as an ElementSet of type XYLine. The river model is composed of an ElementSet with only one XYPolyLine element.

Figure 2-9 and Figure 2-10 illustrate how the properties of these ElementSets might look. In Figure 2-9 the exact co-ordinates in (x,y) are replaced by references to the vertex co-ordinates for illustrative purposes only. Figure 2-10 illustrates another way of internal representation of an XYPolygon.



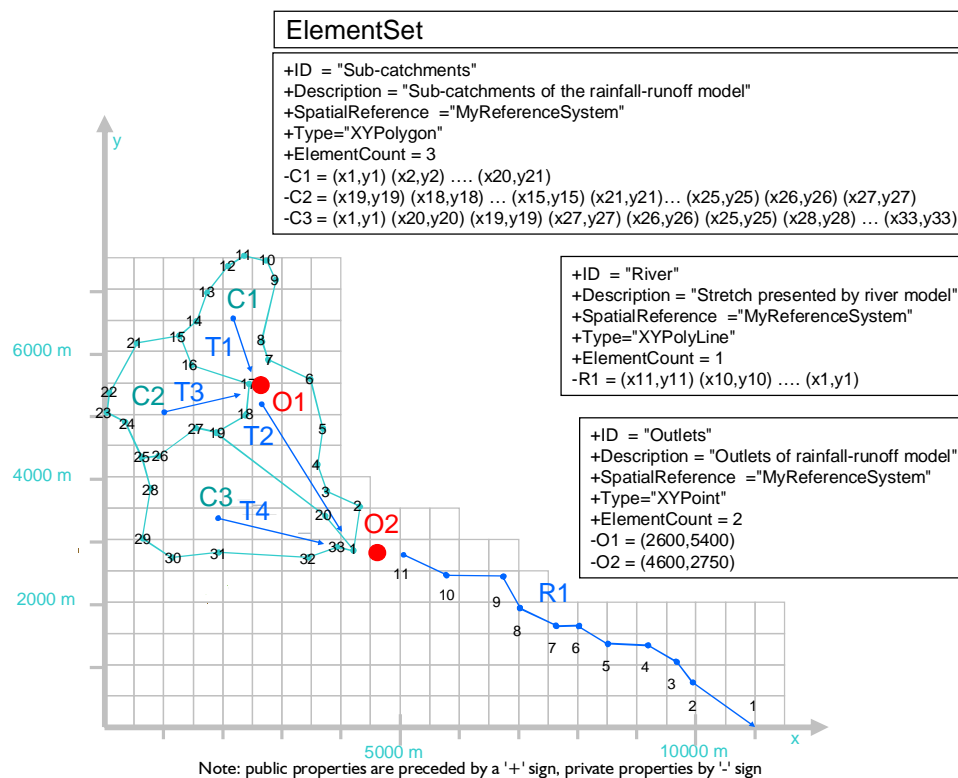


Figure 2-9 Illustration of ElementSet properties (example 1)

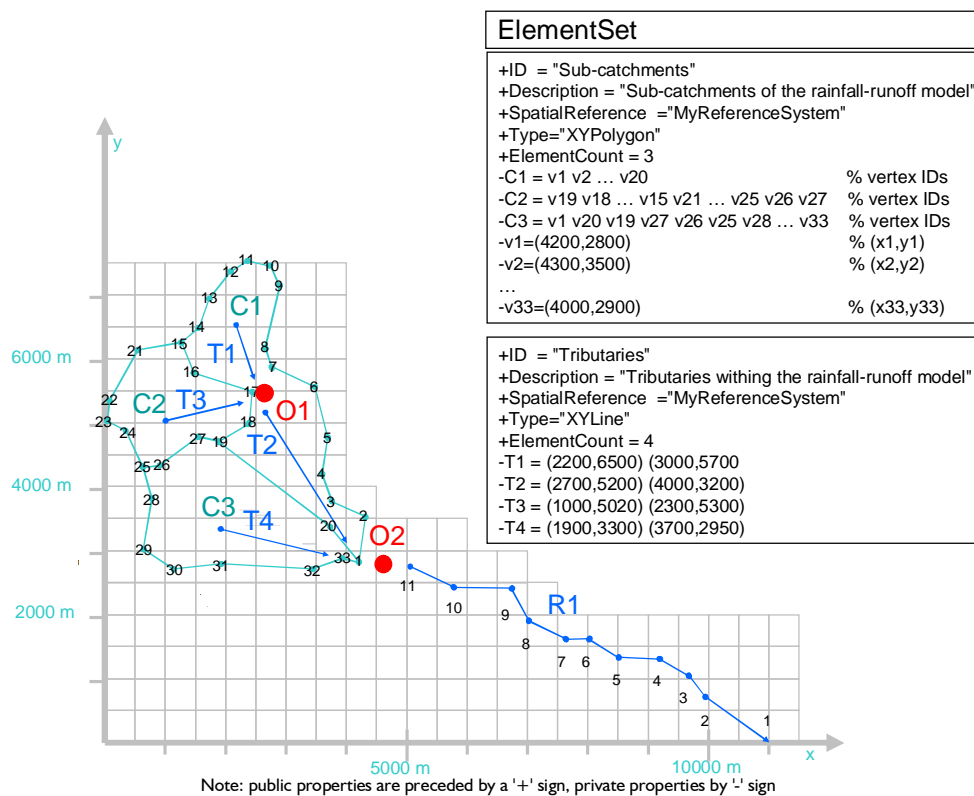


Figure 2-10 Illustration of ElementSet properties (example 2)

Regular grids will again use other internal representations (e.g. number of rows, number of columns, cell size in x and y direction). By using an interface definition that should be implemented, the OpenMI leaves developers their own choice of intelligent storage and representation.

### 2.2.4.3 Choosing an ElementType

The question of which ElementType to choose is highly relevant to the way models are connected. The code developer and/or model builder need to decide how its data can be provided, and thus how its elements are exposed.

Straightforward decisions have to be made:

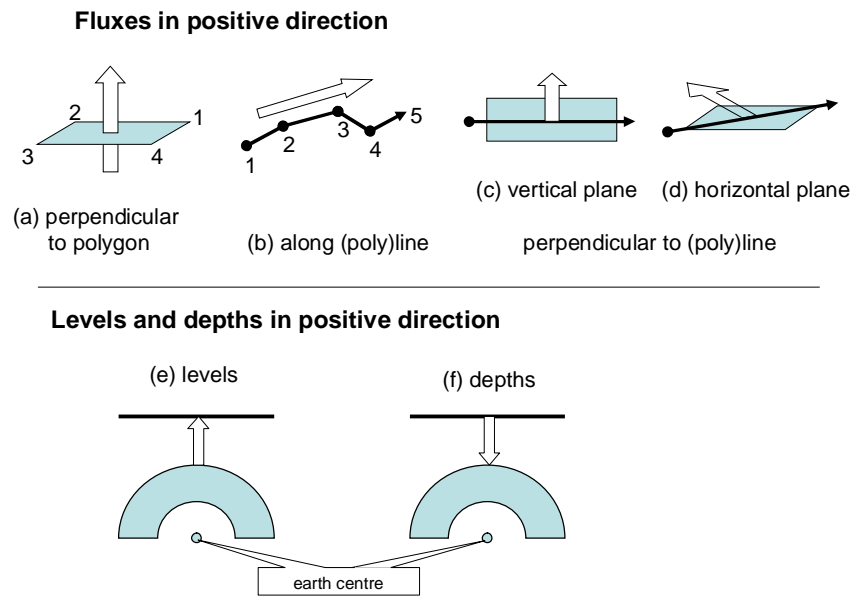
- Do you provide an ID-based ElementSet (non-geo-referenced information) or a geo-referenced ElementSet (thus enabling spatial mapping methods)?
- Do the geo-referenced ElementSets include topological information (using XYLine/XYPolyLine/XYPolygon) or not include topological information (using XYPoint)?

An even more important issue in relation to the ElementType is the question of where the quantity applies and how to interpret positive and negative values.

To explain this issue, a distinction will be made between quantities representing a flux (e.g. discharge) and quantities that do not represent fluxes (e.g. water level).

In general, values are positive if the matter leaves the source component and enters the target component:

- For fluxes through a plane or polygon, the 'right-hand rule' applies (Figure 2-11a). (Curl your right hand in the vertex order of the plane or polygon. The thumb points in the positive direction.)
- The direction of fluxes along a line or polyline is defined as positive from the start to the end vertex (Figure 2-11b).
- The right-hand rule applies for fluxes perpendicular to a line or polyline (see Figure 2-11c for the vertical plane and Figure 2-11d for the horizontal plane). (Put your hand flat (vertical) along the line in the positive direction and turn your wrist clockwise. When passing the horizontal plane, the thumb will point in the positive direction perpendicular to the line or polyline.)
- For volumes, the value is positive if the flux 'leaves' the source and 'enters' the target.
- Levels are positive in the direction that moves away from the earth centre (Figure 2-11e), depths are positive in the direction that moves towards the earth centre (Figure 2-11f).



**Figure 2-11 Interpreting positive values of fluxes, levels and depths**

#### 2.2.4.4 Dynamic ElementSets

Typically, a model is based on a known grid which remains static throughout the lifetime of the computation. However, advanced models might use adaptive grids. Examples may be wave models having an adaptive vertical dimension or bank-erosion models where the grid of the channel adapts to the shape of the bank.

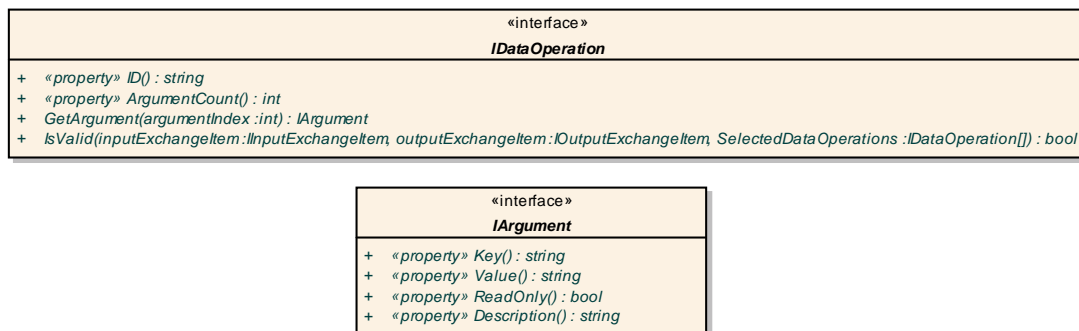
To accommodate these advanced features, a version number has been introduced in the `IElementSet` interface. `SourceComponents`, which have to deliver on this adaptive `ElementSet`, can use this property to determine if their `TargetElementSet` has changed over time. If so, the `SourceComponent` has to re-query the `ElementSet` to update its own data mapping.

## 2.2.5 Using data operations to describe how data can be mapped

This section introduces the methodology that the OpenMI provides to describe available data operations.

In many situations, data transformations will be needed to map the available data of the source component to the request format of the target component. Data transformations might address spatial aspects, temporal aspects or other aspects. Data transformations are the responsibility of the component that provides the data (i.e. the source component). Every component developer may take this responsibility in a different way; for example, one component may offer advanced interpolation algorithms, while another can only provide the nearest available value and a third component may offer both methods (or even more).

One or more parameters can be introduced to specify exactly the settings of the data operation. Figure 2-12 introduces the concept chosen. During configuration, a value is assigned to the argument to specify the settings of the data operation.



**Figure 2-12 Definition of a Data Operation and associated arguments**

For illustrative purposes, a number of potential data operations have been described in Table 2-3 (temporal aspects), Table 2-4 (spatial aspects) and Table 2-5 (miscellaneous aspects). Note that this table reflects just one way of describing data operations; this is not necessarily the only way to do it.

**Table 2-3 Temporal data operations**

DataOperation		Argument			
ID	Arg. Count	Key	Description	Value	Read Only
NoDataOperation	0	Null			
ProvideException	0	Null			
ProvideAllValues	0	Null			
TimeSpanAggregationbyAveraging	0				
TimeSpanAggregationbyAccumulation	0				
TemporalMappingByTakeNearest	0				
TemporalMappingByBeginValue	0				
TemporalMappingByEndValue	0				
TemporalMappingByLinearInterpolation	0				
TemporalCompletionByMissingValue	1	MissingValue	Define missing value		
TemporalCompletionByLinear Extrapolation	2	Multiplier	Define multiplier		
		Offset	Define offset		
TemporalCompletionByExtrapolateWith LastGradient	0				

**Table 2-4 Spatial data operations**

DataOperation		Argument			
ID	Arg. Count	Key	Description	Value	Read Only
NoDataOperation	0				
ProvideException	0				
ProvideAllValues	0				
SpatialAggregationbyAccumulation	0				
SpatialAggregationbyAveraging	0				
SpatialMappingByKriging	0				
SpatialMappingByInterpolation	0				
SpatialMappingByInverseDistance	0				
SpatialMappingByTakeNearest	0				
SpatialCompletionByMissingValue	1	MissingValue	Define numerical missing value		
SpatialCompletionByLinear Extrapolation	2	Multiplier			
		Offset			
UseDefinedSpatialMappingMatrix	1	File name	Reference to mapping matrix		

**Table 2-5 Miscellaneous data operations**

DataOperation		Argument			
ID	Arg. Count	Key	Description	Value	Read Only
ApplyVerticalShift	1	VerticalShift	Define numerical value (negative is towards earth centre)		

As can be seen from the names, data operations may be addressing the issues of data aggregation, data transformation/mapping (when the surrounding values are known) or data completion (in other cases). Not all data operations can be combined with each other, while

some data operations may transform the unit and dimension of data: for example, temporal aggregations may affect the time dimension.

User interfaces should be able to support end users in the selection of a valid combination of data operations.

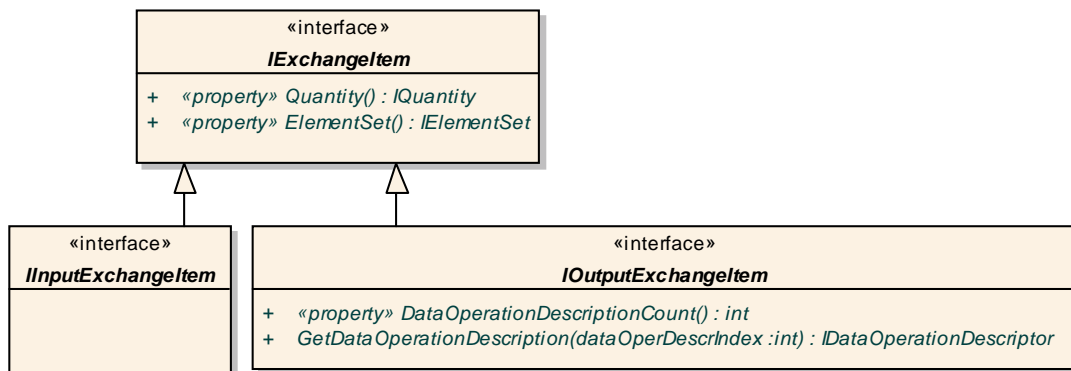
The `IsValid(InputExchangeItem, OutputExchangeItem, SelectedDataOperations)` method enables an integrated check on the combination of data operations. Modification of the unit and dimension due to temporal aggregations can be accommodated, as well as other advanced checks.

## 2.2.6 Grouping into Exchangeltems

As you typically exchange a Quantity on an ElementSet, this combination is grouped into an *Exchangeltem*. A model can have Exchangeltems as input (InputExchangeltem) or can provide them as output (OutputExchangeltem). The list of all Exchangeltems of a model is sometimes referred as an *exchange model*, although this is a not part of the OpenMI standard.

### 2.2.6.1 Exchangeltems

The previous sections discussed the various individual pieces that are necessary to expose the exchangeable data to the outer world. Quantities are typically exchanged for certain locations (i.e. the ElementSet) and the combination forms an Exchangeltem. Such an Exchangeltem can either act as an input for a specific model or as an output (Figure 2-13).



**Figure 2-13 Exchangeltems: combination of a Quantity on an ElementSet**

As the providing component has to perform the data operations, its Exchangeltems will need to describe the available data operations.

One model, related to a specific software component, will typically be able to exchange several Exchangeltems. For organizational purposes, such a group of Exchangeltems may sometimes be referred to as an *exchange model*.

Taking the use case of the river and rainfall-runoff models, Figure 2-14 illustrates the Exchangeltems of the various components involved. MyRainGrid in the rainfall model is a 500x500 m grid (defined as a collection of XYPolygon elements). Laterallnlets is a collection of XYPoint elements at the calculation nodes of the river model.



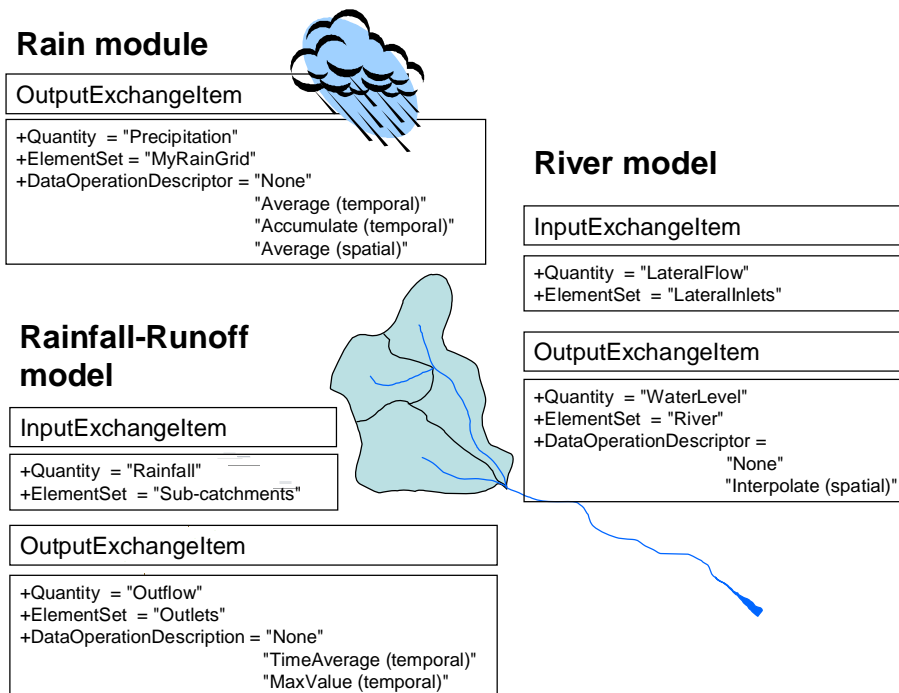


Figure 2-14 Exchangeltems in the use case

### 2.2.6.2 Initially unknown Exchangeltems

While many linkable components are models that have a-priori knowledge of the complete content of their Exchangeltems, this is not always the case. Six basic use cases are presented where the InputExchangeltems and OutputExchangeltems range from a-priori known to fully a-priori unknown (and thus completely dynamically dependent on the input link). These are shown in Table 2-6.

**Table 2-6 Examples of ExchangeItems****Case 1: Data source – e.g. database**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
Null	Null	A-priori known	A-priori known

**Case 2: Model (engine + schematization) – e.g. SOBEK for the Rhine**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
A-priori known (encapsulated in engine)	A-priori known (via input data)	A-priori known (encapsulated in engine)	A-priori known (via input data)

**Case 3: Model engine (no schematization loaded) – e.g. SOBEK-CF**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
A-priori known (encapsulated in engine)	A-priori unknown ElementType known	A-priori known (encapsulated in engine)	A-priori unknown ElementType known

**Case 4: Configurable model engine (schematization to be inherited) – e.g. SOBEK-WQ**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
A-priori known (via input configuration)	A-priori unknown, depends on link  ElementType known	A-priori known (via input configuration)	A-priori unknown, depends on input ElementSet  ElementType known

**Case 5: Configurable data processing engine (schematization is irrelevant) – e.g. frequency analysis package or mathematical script**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
A-priori unknown, depends on link	A-priori unknown, depends on link ElementType might be configurable	Known derivation, depends on input quantity	A-priori unknown, depends on input element set ElementType might be configurable

**Case 6: Data sink – e.g. visualization or output file**

InputExchangeItem		OutputExchangeItem	
Quantity	ElementSet	Quantity	ElementSet
A-priori unknown depends on link	A-priori unknown depends on link	Null	Null

From Case 4 onwards, the components inherit their ElementSet from a connected source component. Therefore, the model can only deliver fully populated ExchangeItems by dynamically asking its SourceComponents for their ElementSets and, from Case 5, their Quantities.

However, as long as no link is established, it is still useful for a component to be able to specify whether it can accept input and provide output. Table 2-7 gives recommendations on the ExchangeItems to be returned in the initial stage of configuration.

**Table 2-7 Recommended ExchangeItems**

	<b>Quantity</b>	<b>ElementSet</b>
<b>Case 3 Model engine (unpopulated)</b>	(List of quantities)	Choose from: <ul style="list-style-type: none"> <li>Unknown element set</li> <li>Unknown element set of type &lt;specific ElementType&gt;</li> </ul>
<b>Case 4 Configurable model engine (unpopulated)</b>	(List of quantities)	Choose from: <ul style="list-style-type: none"> <li>Unknown element set</li> <li>Unknown element set of type &lt;specific ElementType&gt;</li> </ul>
<b>Case 5 Data processing engine</b>	Any quantity (input), derived quantity (output)	Choose from: <ul style="list-style-type: none"> <li>Unknown element set /</li> <li>Unknown element set of type &lt;specific ElementType&gt;</li> </ul>
<b>Case 6 Data sink</b>	Any quantity	Choose from: <ul style="list-style-type: none"> <li>Any element set</li> <li>Any element set of type &lt;specific ElementType&gt;</li> </ul>

Recommended keywords are:

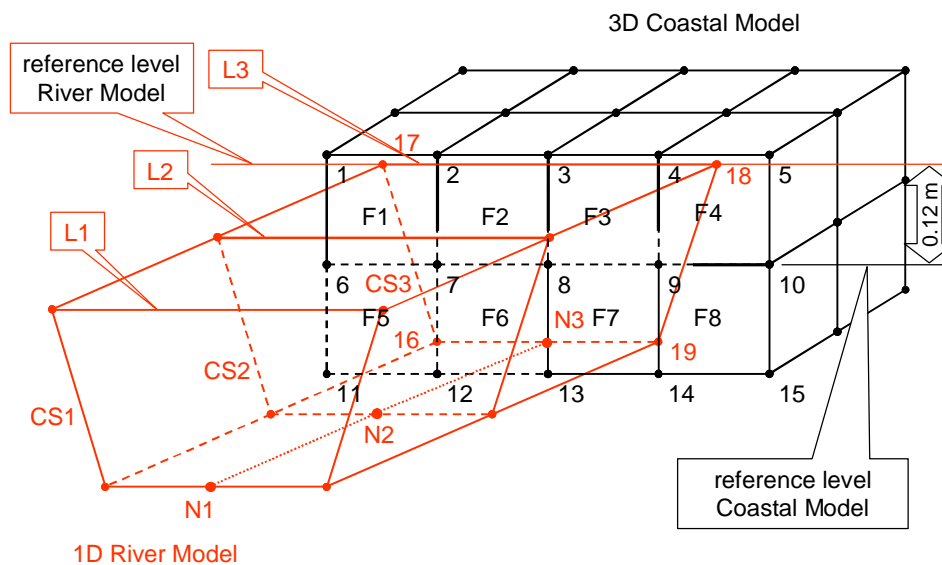
- For Quantities: any quantity.
- For ElementSets: any element set / any XYPoints / any XYZPoints / any XYLines / any XYZLines / any XYPolyLines / any XYZPolyLines / any XYPolygons / any XYZPolygons / any XYZPolyhedrons / unknown

Note that the keyword 'nothing' is not needed as a Null is returned if no ExchangeItem can be accepted or provided.

Finally, returning an empty ExchangeItem may also be acceptable if 'anything' or 'unknown' can be accepted or provided but the use of keywords will make it easier for users to understand.

## 2.2.7 An advanced example

This section illustrates how an advanced link between a 1D hydrodynamic river model and a 3D hydrodynamic coastal model might be developed. Figure 2-15 shows the models to be linked. The river model contains cross-sections, which are shown in red. The coastal model contains a regular grid of cells. Both models will require boundary conditions from each other. The 3D model requires (mass conservative) inflow conditions for the cell faces, which are indicated by F1, F2, F3, F4 etc. The 1D river model requires a water level boundary condition for the last cross-section, which in this case is the average water level between vertex 17 and vertex 18. Both models expose their boundary face in the vertical plane. The reference level of the two models differs by 12 centimetres.



**Figure 2-15 Example exposing a 1D river model to a 3D coastal model**

The model components of this example are very powerful, as they can provide all kinds of data on various types of ElementSet. Those ElementSets are illustrated in Figure 2-16 and Figure 2-17.

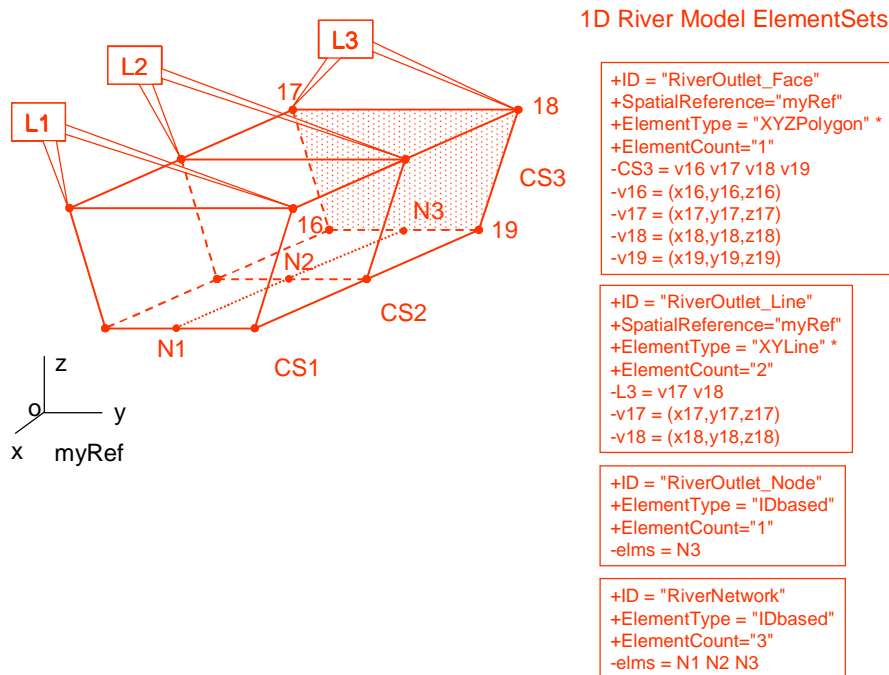


Figure 2-16 Element sets of a 1D river model

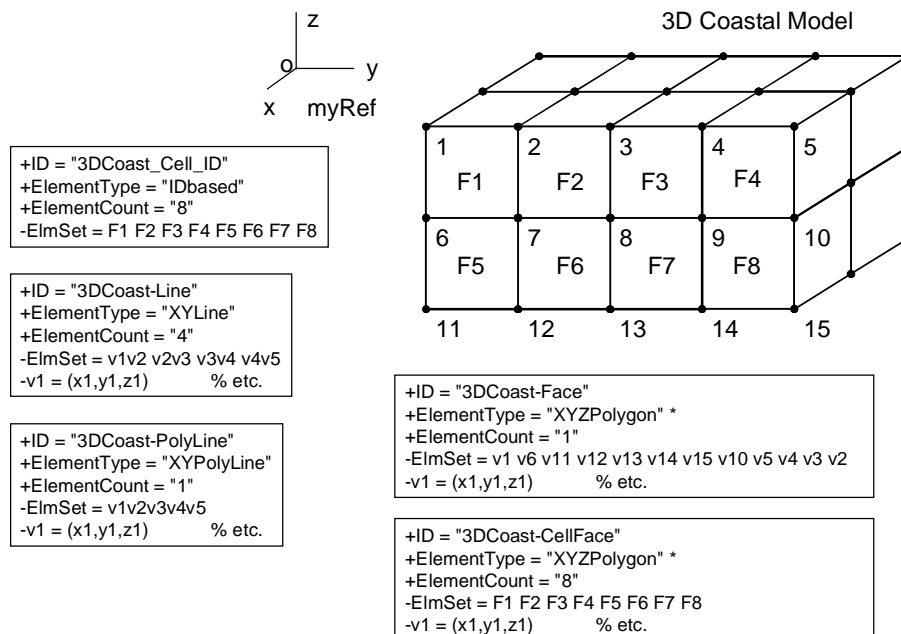


Figure 2-17 Element sets of a 3D coastal model

In this example, the models share the same name for the same semantic quantity. Table 2-8 indicates the quantities that can be exchanged.

**Table 2-8 Quantities for the 1D river/3D coast model**

ID	Description	Value type	Dimension	Unit
WaterLevel	Height of the open water surface above a certain reference level	Scalar	Length=1	m to ref.
TidalInflow	Tidal flow in the upstream direction of the river outlet	Scalar	Length = 3 Time = -1	m <sup>3</sup> /s
Outflow	Outflow (i.e. combination of river discharge and tidal outflow) of the river into the estuary/sea	Scalar	Length = 3 Time = -1	m <sup>3</sup> /s
Velocity	Velocity (vertically averaged) through a vertical plane	Vector	Length = 1 Time = -1	m/s

Many quantities are available on all types of ElementSets, as is shown in the tables indicating the InputExchangeItems and OutputExchangeItems (Table 2-9 and Table 2-10 for the 1D river and Table 2-11 and Table 2-12 for the 3D coastal model).

**Table 2-9 InputExchangeItems for the 1D river model**

Quantity	ElementSet
WaterLevel	RiverOutlet_Node
WaterLevel	RiverOutlet_Line
WaterLevel	RiverOutlet_Face

**Table 2-10 OutputExchangeItems for the 1D river model**

Quantity	ElementSet	DataOperation (available)
Outflow	RiverOutlet_Node	None
TidalInflow	RiverOutlet_Node	None
Velocity	RiverOutlet_Node	None
Outflow	RiverOutlet_Face	SpatialInterpolation (mass conservative)
TidalInflow	RiverOutlet_Face	SpatialInterpolation (mass conservative)
Velocity	RiverOutlet_Face	SpatialInterpolation (mass conservative)

**Table 2-11 InputExchangeItems for the 3D coastal model**

Quantity	ElementSet
Outflow	3DCoast_CellFaces
TidalInflow	3DCoast_CellFaces
Velocity	3DCoast_CellFaces
Outflow	3DCoast_Cell_ID
TidalInflow	3DCoast_Cell_ID
Velocity	3DCoast_Cell_ID

**Table 2-12 OutputExchangeItems for the 3D coastal model**

Quantity	ElementSet	DataOperation (available)
WaterLevel	3DCoast_Cell_ID	VerticalShift MissingValue
WaterLevel	3DCoast_Line	VerticalShift MissingValue SpatialAveraging
WaterLevel	3DCoast_PolyLine	VerticalShift MissingValue SpatialAveraging



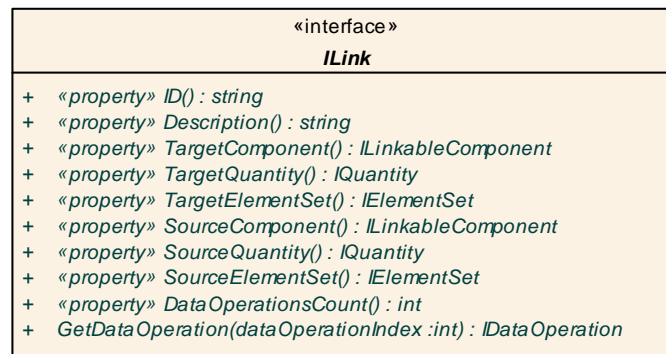
## Chapter 2.3 Configuring links and compositions

A link is the way in which two models are coupled. When two or more OpenMI-compliant models are linked together, they form a composition. The various links within the composition must be configured for the models that are included in the system.

This chapter describes the OpenMI link definition and the properties that have to be configured.

### 2.3.1 Configuring a single link

A link is the data path connecting two linkable components. It is uniquely identified by an ID and can also be given a textual description. It connects one (and only one) *source component* with one (and only one) *target component*. Furthermore, using this link, one (and only one) quantity is exchanged in one direction. As this quantity may be named differently by each linkable component, the link specifies both the *source quantity* and the *target quantity* (Figure 2-18).



**Figure 2-18 The OpenMI interface definition of the Link**

To enable communication between models using different units, spatial references or timesteps, the link can also specify the data transformation operations that have to be performed by the providing component before delivering the values to the accepting component.

Finally, you have to specify the locations where data will be exchanged: i.e. the SourceElementSet and the TargetElementSet. Between these two sets exists a mapping relation, which varies according to the nature of the elements sets (e.g. it can be a one-to-one mapping of nodes or a function mapping a line to a polygon).

The link configuration can be a complex procedure and usually requires human intervention. The OpenMI specification does not imply any compatibility and logical checks on the linked quantities, relying on the modeller to identify the appropriate quantities and locations. This means that you have to know beforehand both the available locations and the available quantities.

To make the link configuration easier, the OpenMI provides some support tools that can speed up the linking process, for example by automatically specifying the source and target elements in the case of geo-referenced data. Furthermore, using a graphical configuration editor, you can visualize the links and specify their properties using intuitive dialog boxes.

To summarize, link configuration requires the following steps:

- Select source and target linkable component.
- Select source and target quantity.
- Select source and target element set.
- Select data operations.

Taking the use case of the river and rainfall-runoff models, Figure 2-19 illustrates the links to be made for the connection between the rain module, the rainfall-runoff model and the river model. Figure 2-20 provides the exact definition of the associated link properties.

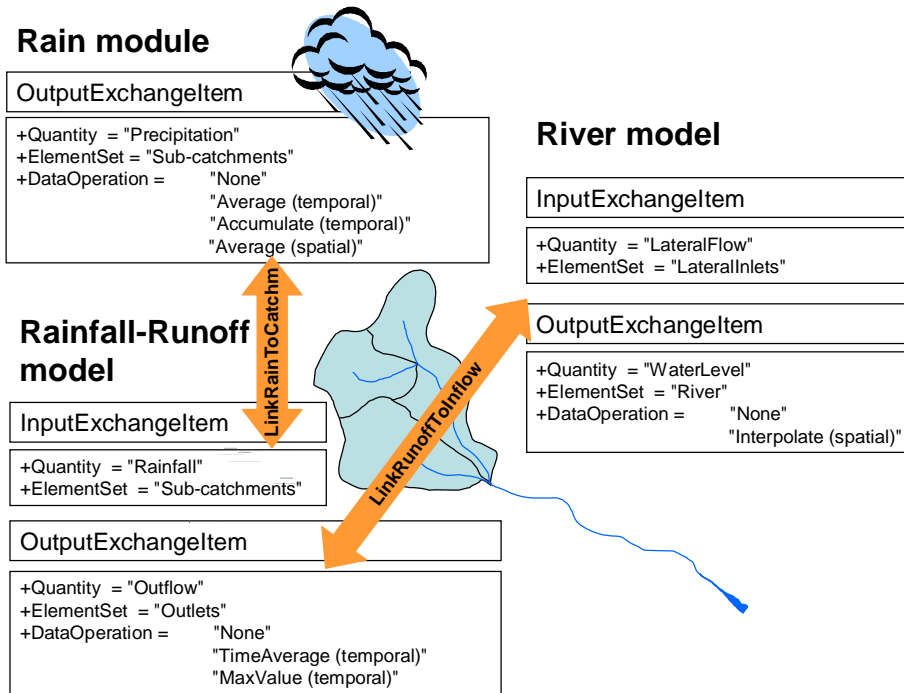


Figure 2-19 Links to be made in the river/rainfall-runoff use case

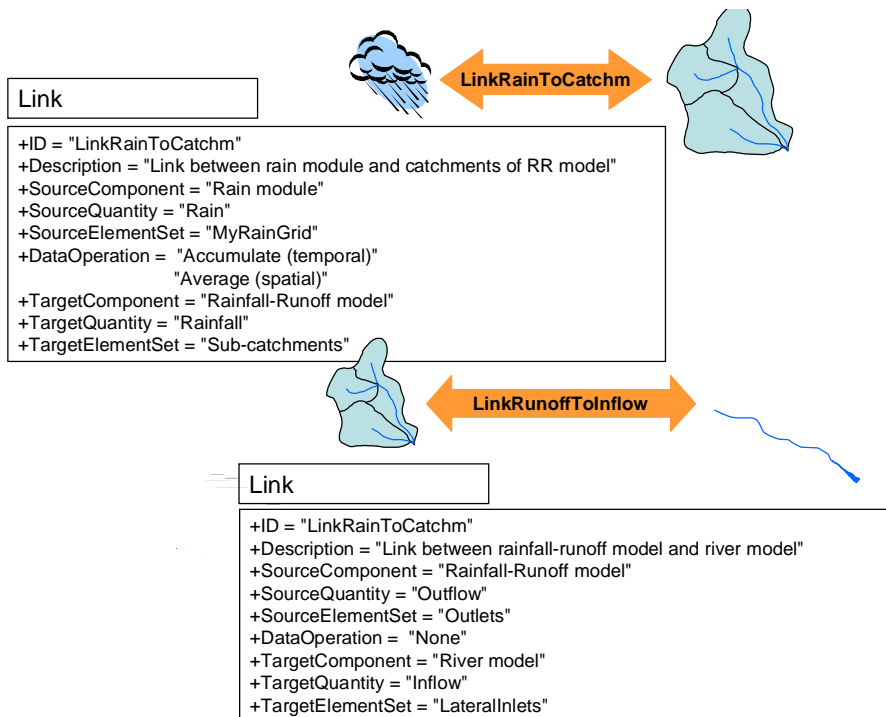


Figure 2-20 Full link definition of the river/rainfall-runoff use case

## 2.3.2 Building a composition

This section investigates the steps in building a complete integrated simulation from linkable components and links and representing it as a composition.

In the OpenMI terminology, a *composition* is a set of linkable components, possibly populated with model data and interconnected with links. Conceptually it represents the final stage before running the integrated simulation. Physically, it is the input to a utility, usually referred to as the *Deployer*, which instantiates all involved models and links, sets all necessary parameters and finally initiates the simulation.

There are four steps in building a composition:

- Select all involved linkable components or schematizations.
- Create the links between the interacting components.
- Configure the links.
- Set any other simulation parameters.

The easiest way to build a composition is via a graphical utility, the OmiEd configuration editor, described in Chapter 2.4.

First the linkable components involved in the simulation have to be selected and be positioned on the working area of the configuration editor. The selection procedure is facilitated by built-in repositories that store all descriptive information about available components and their properties.

The second step is linking the components. This is done simply by using the mouse to draw a connecting line between the two models. If two components exchange more than one quantity or exchange data in both directions, separate links have to be set up.

After defining each link, its properties have to be set. Double-clicking on the link will bring up the link properties dialog box where available quantities, element sets and data operations are listed.

Finally, any other model-specific or simulation-related parameters have to be set up.

The composition should be saved for the following reasons:

- Setting up a complex composition usually takes up a lot of effort.
- A simulation is frequently run repetitively, each time with different parameters, until the desired results are achieved.
- An integrated model simulation may use model engines running remotely or use data that do not exist on the local system.

All linkable components are represented by an XML file which refers to the software unit and the data to populate it. By calling and initializing this component, it can supply metadata on the quantities it can provide or accept and the locations where it can provide or accept data. Based on these metadata, the modeller can establish the links between the components and build a composition, which can be then saved into an XML file. Of course, this file can then be opened and edited each time the simulation has to be run or modified.

## Chapter 2.4 Using the OpenMI configuration editor

The OpenMI Software Development Kit includes OmiEd, a visual tool for building and running OpenMI systems. The details of the system are stored as a composition.

The stages in building an OpenMI system include:

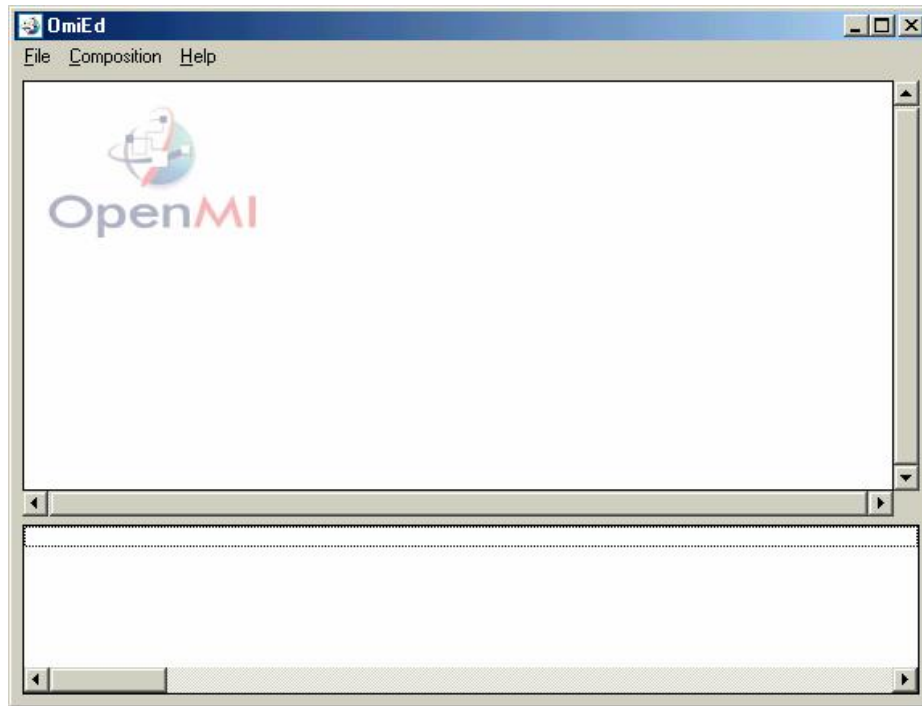
- Start the configuration editor.
- Add models to the composition.
- Establish connections between the models.
- Configure the connections.
- Add a trigger.
- Run the composition.

This chapter gives a brief introduction to the editor and its use.

## 2.4.1 Starting the configuration editor

The OmiEd application is installed in the Program Files directory, using the standard Windows installation program.

To run the application, from the Windows Start menu select **Program Files / OpenMI / OpenMI Configuration Editor**. The OmiEd window is displayed (Figure 2-21).



**Figure 2-21** The OmiEd display

The editor has three menus:

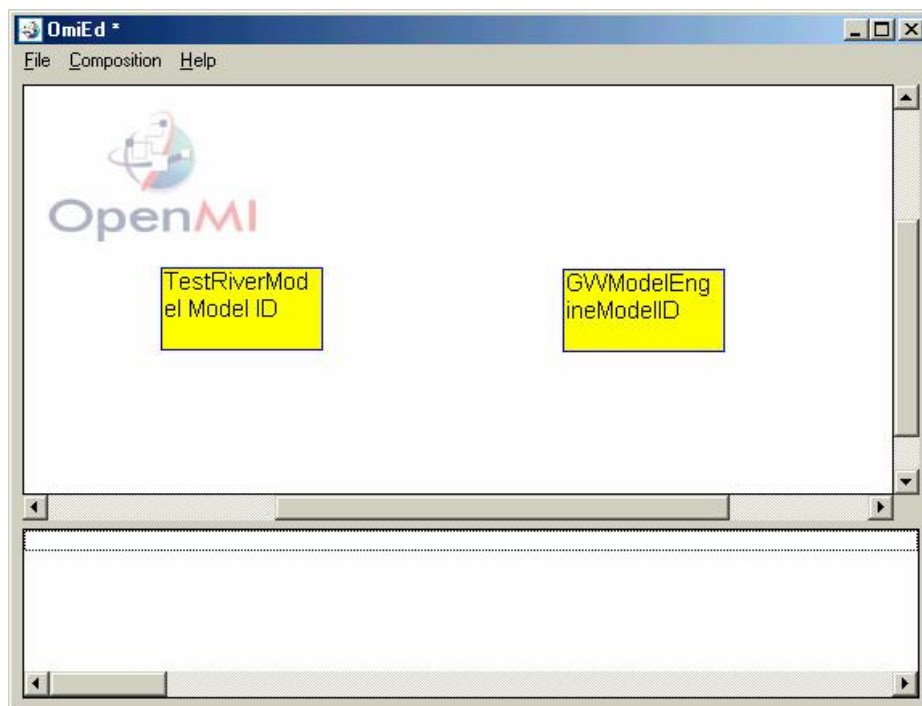
- The **File** menu has options to create a new composition; reload a composition (losing any changes made since the last save); open an existing composition; save the current composition (with its existing name or a new name); and exit the program. Compositions are saved with an OPR extension.
- The **Composition** menu lets you add models, connections and triggers; edit connection and model properties; and run the composition.
- The **Help** menu provides instructions for using OmiEd and displays information about the program.

As you add models, these are displayed in the top part of the window.

## 2.4.2 Adding models to the composition

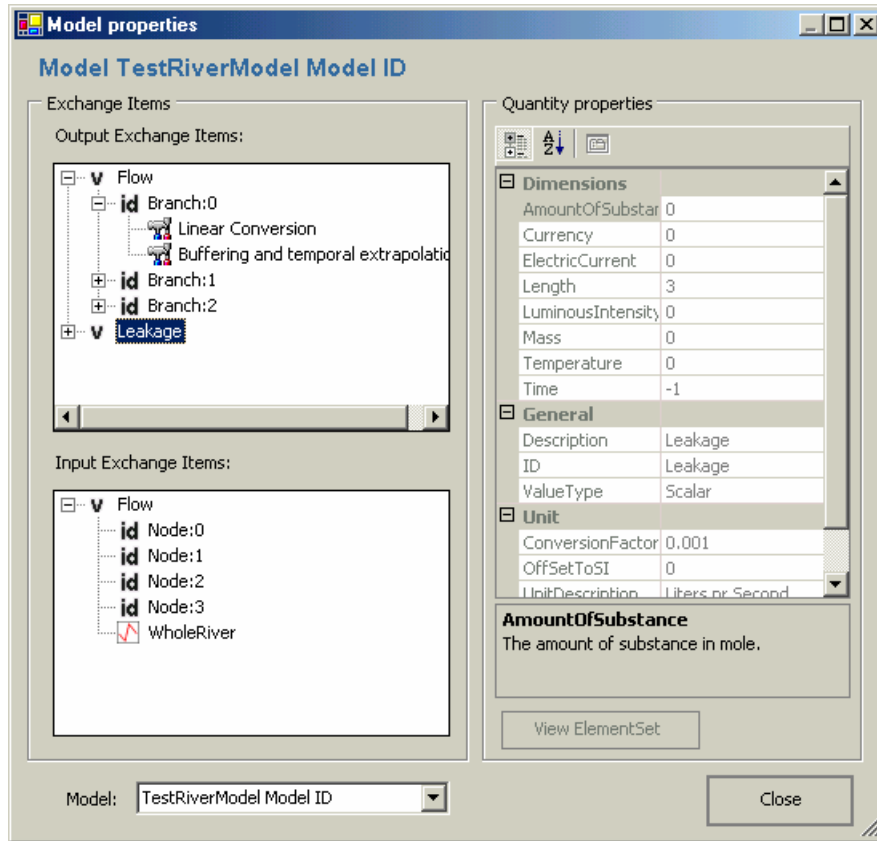
The composition must consist of two or more OpenMI-compliant models, each of which will have a corresponding OMI file. To add a model to the composition:

1. From the **Composition** menu, select **Add Model**.
2. Locate the OMI file for the model and click on **Open**.
3. A box containing the model name is added to the editor window. Drag the box to a suitable position in the window.
4. Add any other models required in the composition (Figure 2-22).



**Figure 2-22 Adding models to the configuration**

You can inspect a model's properties by right-clicking on the model and selecting **Model Properties**. The properties dialog provides details of the model's exchange items. The top box on the left-hand side lists the output quantities; the bottom box lists the input quantities. The lists can be expanded to show the element sets that are available for each item; the element sets can be expanded to show the data operations that are available. Clicking on any item displays the corresponding properties on the right (Figure 2-23).



**Figure 2-23 Model properties dialog**

You can view the properties for any other model by selecting it from the drop-down list at the bottom of the dialog. Save the composition after adding the models.



### 2.4.3 Establishing connections between the models

The models are linked together by adding connections between them:

1. From the Composition menu, select **Add Connection**.
2. Drag the pointer from one model to another. A link is added between the models (Figure 2-24)
3. Repeat for any further connections.

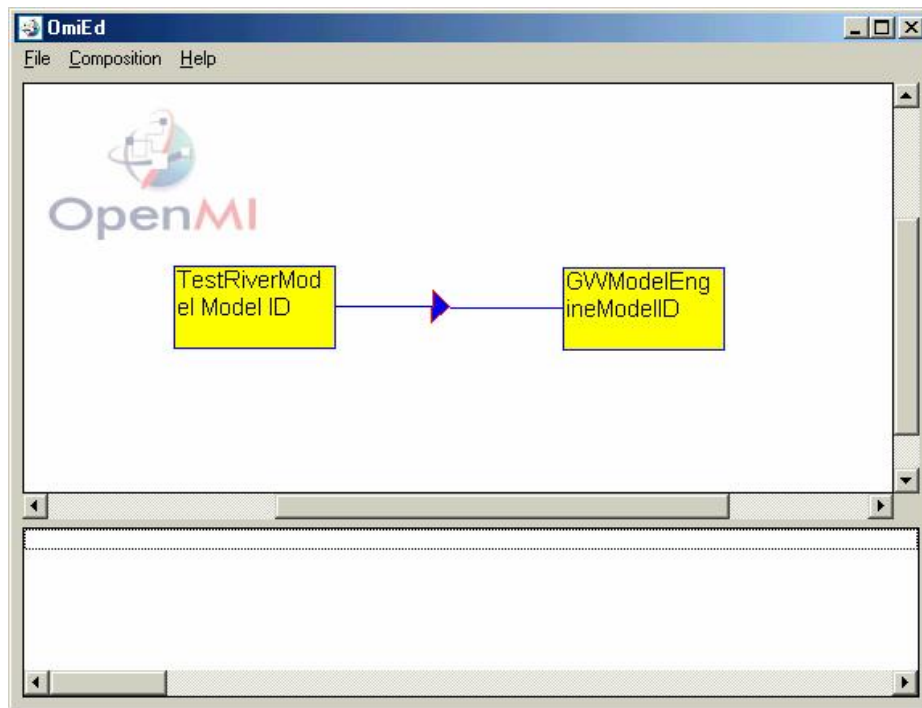
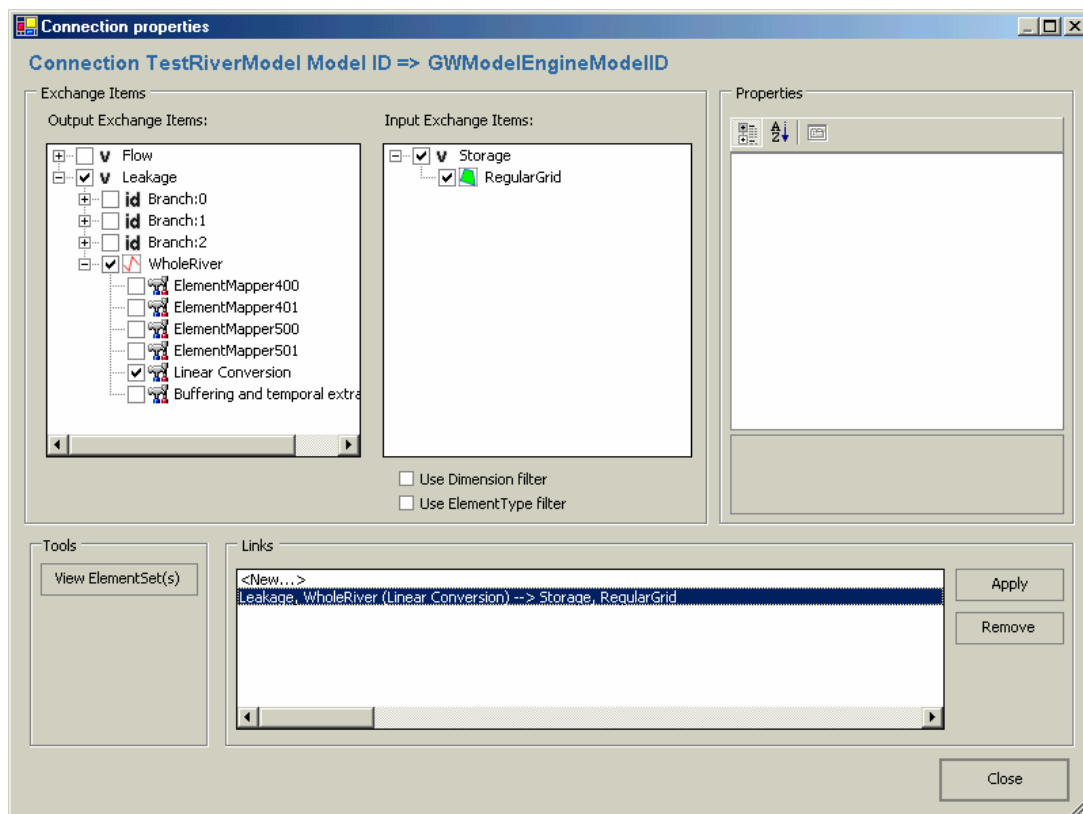


Figure 2-24 Adding connections between models.

## 2.4.4 Configuring the connections

The connection properties must be set for each link:

1. Right-click on the arrow in the middle of the connection and select **Connection Properties**. The properties dialog is displayed.
2. In the Output Exchange Items box, expand the required output quantity and element set. Click on the required data operation. (This determines what data are output, where they are located and how they are presented.)
3. In the Input Exchange Items box, expand the required input quantity and click on the element set that is to receive the data.
4. Click on the **Apply** button. The new link is added to the list at the bottom of the dialog. Click on the link to redisplay its exchange items (Figure 2-25).
5. Click on **Close**.



**Figure 2-25 Connection properties**

In the model properties dialog, you can view the properties for any quantity, element set or data operation by clicking on it. The properties are shown in the box on the right.

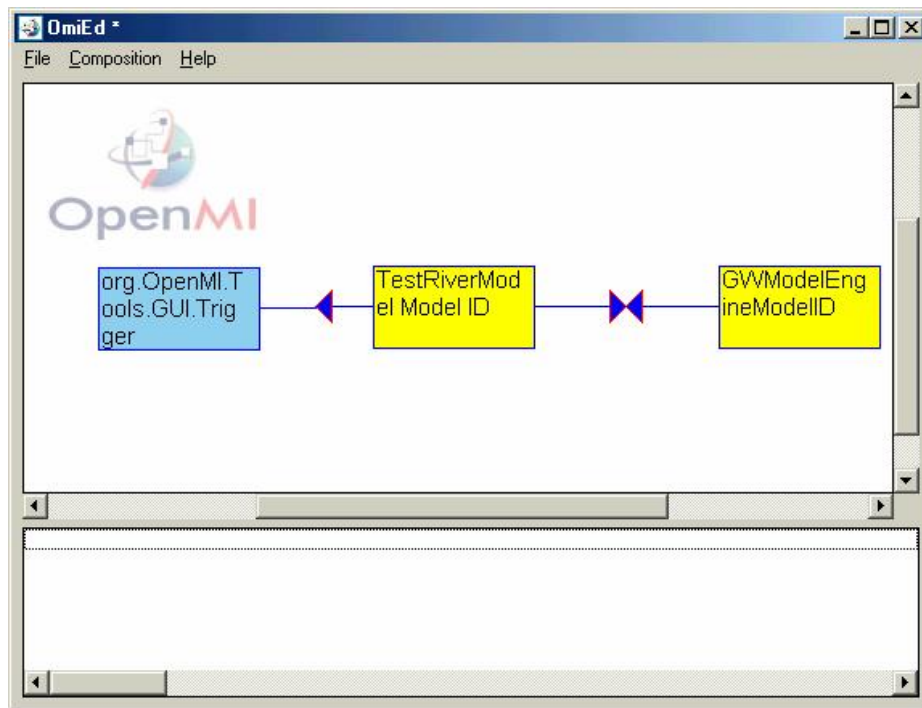
You can remove any connection by clicking on it in the Links box and clicking on **Remove**.

Remember to save the composition after adding links.

## 2.4.5 Adding a trigger

Each composition needs a **trigger** to start the process running. This is added as follows:

1. From the **Composition** menu, select **Add Trigger**. A blue trigger box is added to the composition; this can be moved to a suitable position.
2. Add a connection from the model that is to be run first to the trigger (Figure 2-26).
3. Set the trigger connection's properties.



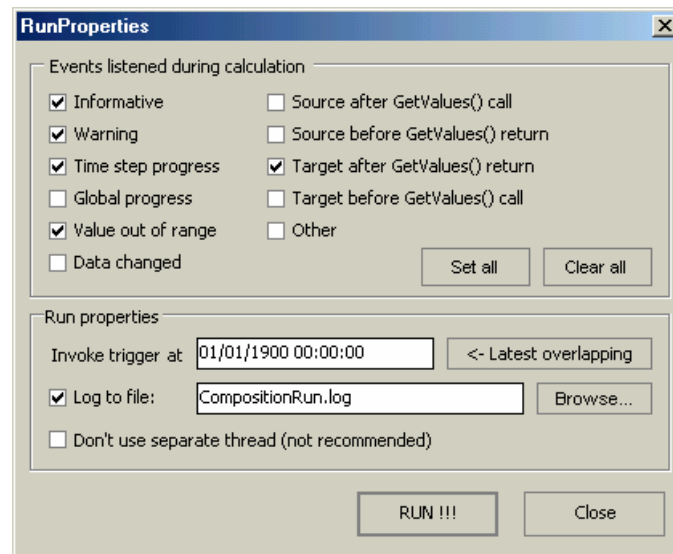
**Figure 2-26 Adding a trigger**

The composition is now complete and ready to be run.

## 2.4.6 Running the composition

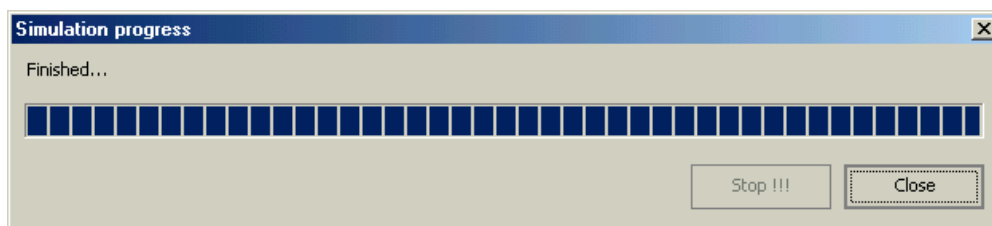
To run the composition:

1. From the **Composition** menu, select **Run**.
2. In the Run Properties dialog, select the events that you want to monitor during the run. You can also specify the time at which the trigger is to be invoked and the name of the log file (Figure 2-27).



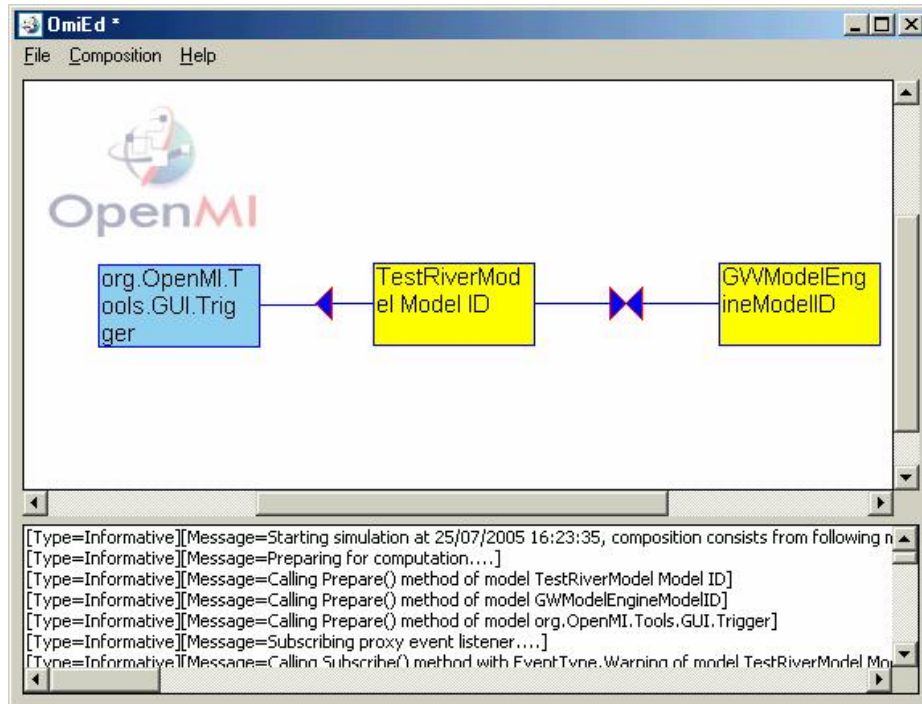
**Figure 2-27 Run properties**

3. Click on **Run**. A dialog is displayed, showing the progress of the run (Figure 2-28).



**Figure 2-28 Simulation progress**

4. When the simulation is complete, click on the **Close** button and confirm when asked that you wish to reload the project. The bottom pane in the OmiEd window shows the run log (Figure 2-29). This log is also available as a text file.



**Figure 2-29 The OmiEd display after a run**

Note that you can increase the size of the OmiEd window when viewing the run log.

Following a run, you can make further changes to the composition and then run it again.



# Book 3    Developing OpenMI systems

<b>BOOK 3</b>	<b>DEVELOPING OPENMI SYSTEMS .....</b>	<b>3-1</b>
<b>Chapter 3.1</b>	<b>OpenMI-compliant systems.....</b>	<b>3-3</b>
3.1.1	What is an OpenMI system? .....	3-4
3.1.2	Locating the components.....	3-5
<b>Chapter 3.2</b>	<b>Establishing OpenMI systems.....</b>	<b>3-7</b>
3.2.1	Phases in using the linkable component interface .....	3-8
3.2.2	Phase I: Instantiation and initialization.....	3-9
3.2.3	Phase II: Inspection and configuration.....	3-10
3.2.4	Phase III: Preparation .....	3-12
3.2.5	Phase IV: Computation/execution .....	3-13
3.2.6	Phase V: Completion .....	3-14
3.2.7	Phase VI: Disposal .....	3-15
<b>Chapter 3.3</b>	<b>Hard-coded systems.....</b>	<b>3-17</b>
3.3.1	An example of a hard-coded system .....	3-18
<b>Chapter 3.4</b>	<b>Support for configurable systems .....</b>	<b>3-23</b>
3.4.1	Main aspects of a configurable system.....	3-24
3.4.2	Configuring and sustaining a component combination.....	3-25
3.4.3	Deploying and running the system .....	3-31
<b>Chapter 3.5</b>	<b>Graphical user interfaces .....</b>	<b>3-37</b>
3.5.1	Building visual tools .....	3-38
3.5.2	OmiEd, a simple front end of the OpenMI SDK.....	3-39





## Chapter 3.1 OpenMI-compliant systems

OpenMI systems are software systems that combine two or more OpenMI-compliant components. This chapter provides an introduction to OpenMI systems and describes the OMI files, through which individual OpenMI components are identified.

### 3.1.1 What is an OpenMI system?

OpenMI systems can be considered software systems that combine a set of OpenMI-compliant components, possibly in addition to non-OpenMI-compliant components. Such a system can deploy and run OpenMI components by accessing them through their standard interface.

In order to do this, the following functionality should be incorporated:

- The OpenMI system needs to know where (i.e. at what resource location) it can find linkable components.
- The OpenMI system needs to know which linkable components are joined together and how; i.e. it needs to know the links.
- The OpenMI system needs to be able to instantiate, deploy and run a combination of linkable components.

OpenMI systems can come in two types:

- Hard-coded systems
- Configurable systems

While the hard-coded system addresses only the functionality above, the configurable system also addresses the inspection of OpenMI linkable components for their exchange items. However, before going into depth on hard-coded and configurable systems, a complete overview is given on all dynamic aspects related to establishing links and running OpenMI components.

### 3.1.2 Locating the components

An OpenMI component can be identified through its OMI file. This is an XML file which contains sufficient information to identify a component, instantiate the binary unit on your machine (i.e. find the assembly and the class to instantiate) and populate it with input data. An XML schema definition has been created to enable default tools to parse the information.

In principle, OMI files can reside anywhere on your system. Users are therefore free to organize their own repository, as long as they can find the relevant OMI files themselves when configuring their model combination.

An example OMI file is given in Figure 3-1. The underlying schema definition (XSD) is provided in Figure 3-2.

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

<LinkableComponent Type="wxDelft.OpenMI.WLinkableComponent" Assembly="wxDelft.OpenMI,
  Version=1.0.0.0, Culture=neutral, PublicKeyToken=8384b9b46466c568"
  XMLns="http://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 3-1 OMI file example**

```

<?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>

```

**Figure 3-2 XML schema definition of the OMI file**

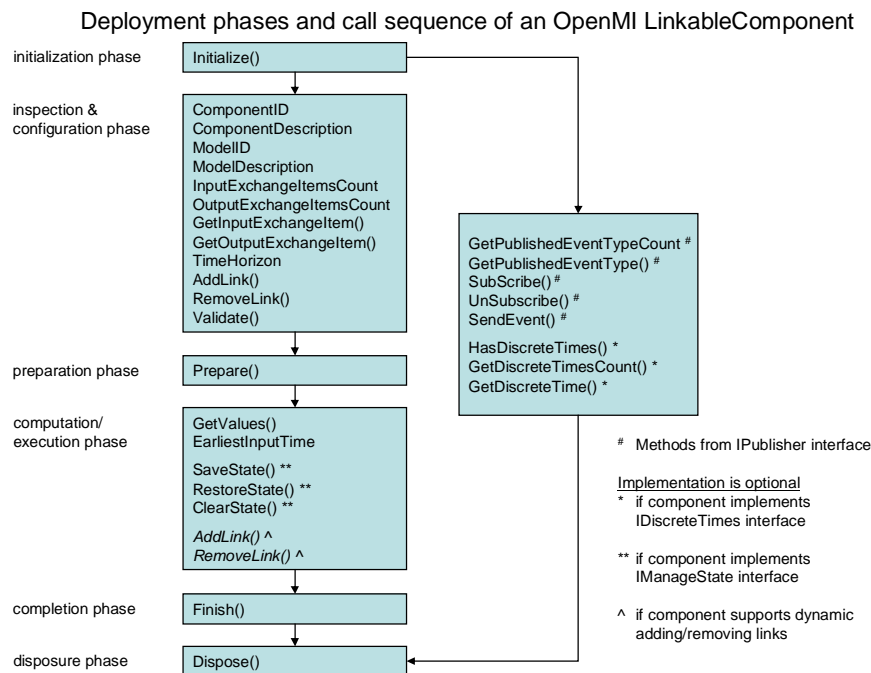
## Chapter 3.2 Establishing OpenMI systems

The OpenMI Standard identifies six phases that are required when establishing and running a combination of OpenMI linkable components.

This chapter addresses each of these phases in turn.

### 3.2.1 Phases in using the linkable component interface

OpenMI systems are composed of OpenMI components that can be used in specific ways. The OpenMI standard identifies a number of phases in the use of OpenMI linkable component. Figure 3-3 provides an overview of the phases that can be identified and the methods that might be invoked at each phase. While the sequence of phases is prescribed, the sequence of calls within each phase is not fixed.



**Figure 3-3 Deployment phases of OpenMI linkable components**

The dynamic behaviour of the various phases is discussed in more detail below.

### 3.2.2 Phase I: Instantiation and initialization

This phase is the entry point of the process to establish an OpenMI system. At the end of the phase, 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.

To instantiate and prepare the river and groundwater model combination the following steps are needed (Figure 3-4):

- *Instantiation:* In this phase the application reads the OMI file, which refers to the software unit (i.e. assembly) that implements the LinkableComponent. Using this reference the LinkableComponent will be constructed.
- *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 contain references to data files.

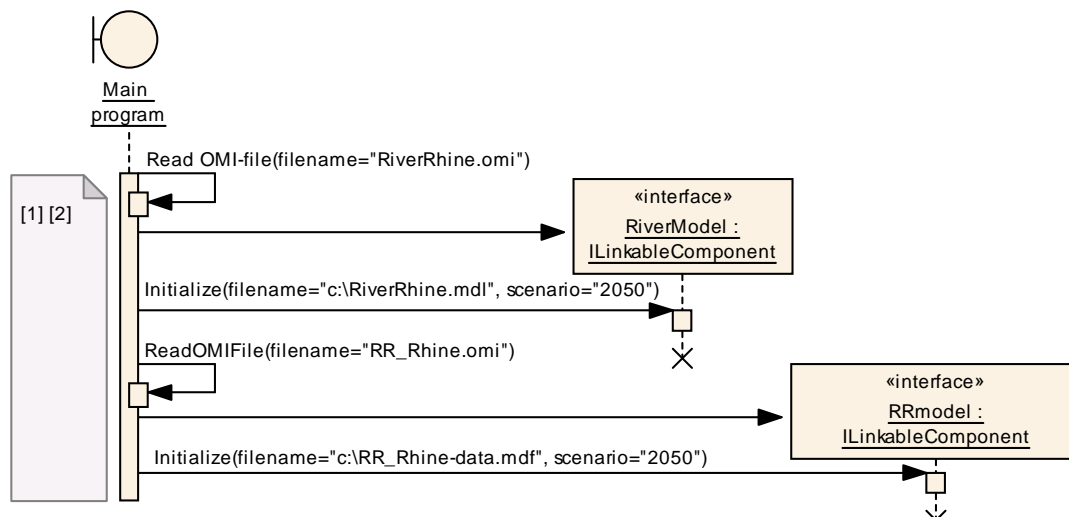
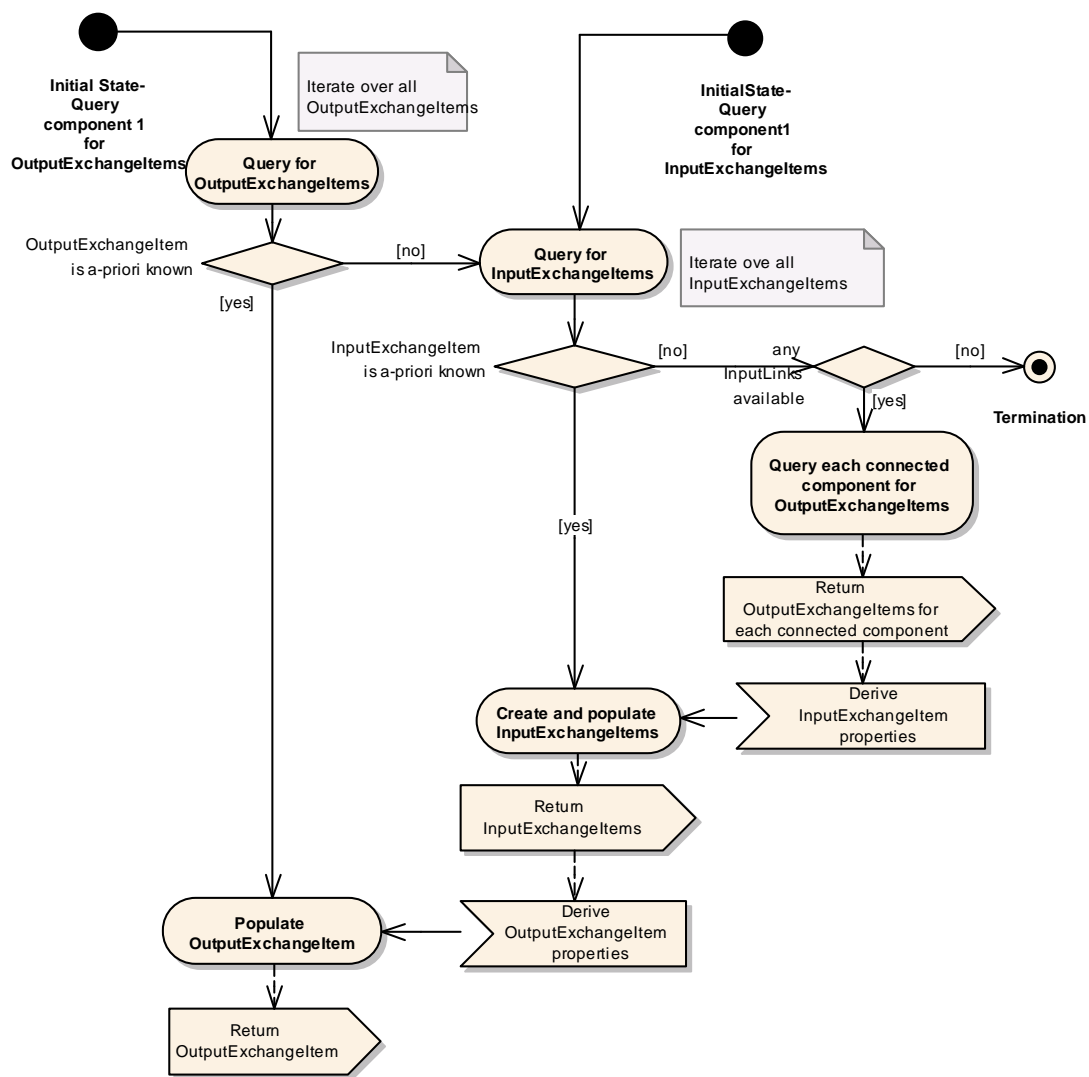


Figure 3-4 Object instantiation and initialization

### 3.2.3 Phase II: Inspection and configuration

At the end of this phase, the links will have been defined and added and each component will have validated its status.

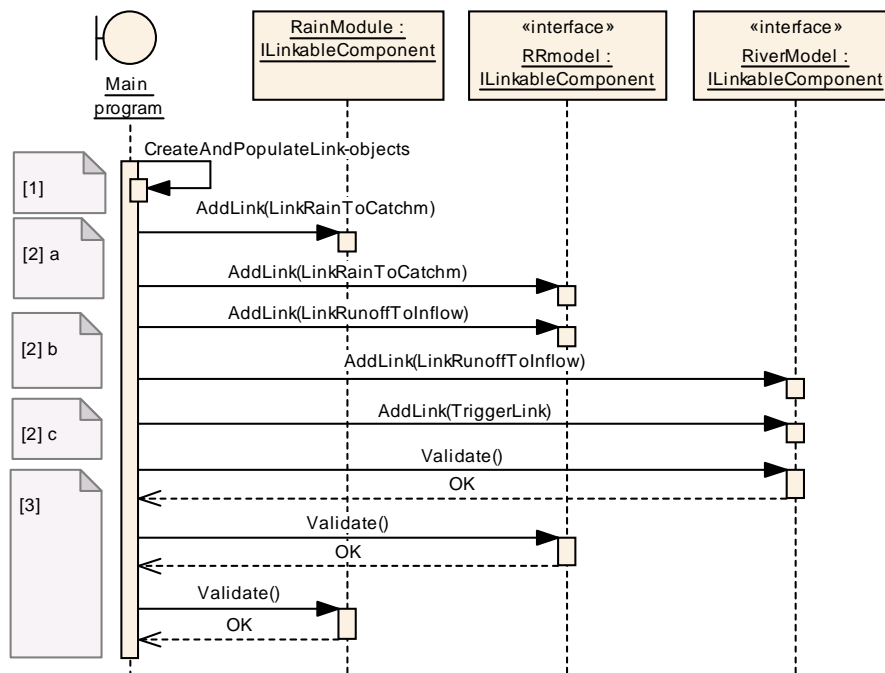
In some cases, this phase might be very straightforward; this is the case for a hard-coded system, for example. In an 'open' system, however, the examination of available exchange items plays a crucial role. The simplest way to retrieve exchange items is to ask for the number of exchange items and loop through the list. The providing component can only implement this directly if the exchange items are static and known a-priori. In those cases where exchange items are a-priori unknown (i.e. they depend on the connected components) a dynamic query process will take place. Figure 3-5 shows how requests for exchange items are resolved (for example, by examining other components for their exchange items).



**Figure 3-5 Obtaining derived exchange items (activity diagram)**

Once the ExchangeItems have been examined the links can be prepared and added to the component. Figure 3-6 illustrates this process for the rainfall-runoff/river model example.





**Figure 3-6 Adding links in the rainfall-runoff/river use case (sequence diagram)**

1. Create the relevant objects (links, quantities, element sets, data operations). Check the validity of selected data operations and populate the objects.
2. Add the links between the components:
  - LinkRainToCatchm (links rainfall to the catchment): The target component (RRmodel) needs to know from which component to obtain the rainfall. The source component (RainModule) needs to know the target element set where it has to deliver the rainfall.
  - LinkRunoffToInflow: The target component (RiverModel) needs to know from which component to obtain the lateral inflow. The source component (RRmodel) needs to know the target element set where it has to deliver lateral inflows. Note that a flux towards the target destination (RiverModel) is positive.
  - TriggerLink: A trigger link to the river model – the downstream component in the data chain – is created and populated. The purpose of this link is to enable the first GetValues() call to the RiverModel. This call triggers the calculation chain.
3. Validate the status of the components and their links using the Validate() method.

### 3.2.4 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 workload starts. This phase contains only one method: Prepare().

During this phase database or network connections (or both) might be established, monitoring stations might be called or model engines might prepare themselves 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 that this phase *must* include a final validation of the status of the linkable component.

### 3.2.5 Phase IV: Computation/execution

During this phase, the heavy workload 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. Section 3.3 of the standard explains how data exchange takes place for unidirectional data exchange, bidirectional data exchange and iteration.

The computation starts by invoking the `GetValues` method via a link. This link can be connected to a *trigger* object (an 'empty' implementation of a `LinkableComponent`) or it might be connected to 'real' components such as a visualization tool or an output file.

While OpenMI linkable components sort out their own data communication and time synchronization, system developers should still pay attention to the question of process control.

System developers can easily develop an application that invokes the component chain with the end time as an argument in its `GetValues` call. The consequence is, however, that the model components do not return control to the main application until they are finished, thus making any smooth interruption of the process difficult. In addition, the 'overall' step size of the computation remains 'out-of-sight' as it is decided by an unknown component.

Given those disadvantages it is therefore recommended that you implement some execution facility that turns the entire simulation period into a loop of timesteps. (Within the `org.OpenMI.Configuration` package this execution facility is called the *deployer*.) While progressing through its simulation period, the `GetValues` call will be invoked for each step. Between each call, the application may react to external events, while the execution facility keeps control over the process. Figure 3-7 illustrates the function of such a loop.

```
//--- Run ---
try
{
    MyRainModule.Prepare();
    MyRRModel.Prepare();
    MyRiverModel.Prepare();
    int nrsteps=100;
    DateTimestep = (end - start)/nrsteps;
    DateTime stop = end + 0.000001;
    DateTime _time = start;
    while (_time <= stop)
    {
        Application.DoEvents();
        IValueSet Values = _trigger.GetValues(new TimeStamp(_time),
            _triggerLink.ID);
        _time = _time.AddSeconds(step);
    }
    GermanRhineModel.Finish();
    NethRhineModel.Finish();
} // end try
```

**Figure 3-7 A time loop**

### 3.2.6 Phase V: Completion

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

### 3.2.7 Phase VI: Disposal

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.



## Chapter 3.3 Hard-coded systems

Hard-coded systems are those systems where the establishment of links and the deployment and execution of a combination of components is completely encapsulated in the source code.

This chapter covers all phases: instantiating the components, establishment of links and run-time deployment and execution.

### 3.3.1 An example of a hard-coded system

When developing a hard-coded system, you usually know the underlying OpenMI components and their capabilities in terms of input and output. Hard-coding the system then comes down to the following steps:

1. Read the OMI files.
2. Instantiate the LinkableComponents and initialize them with the proper input data sets.
3. Create the Quantity and ElementSet objects by querying ExchangeItems.
4. Create the Link objects and populate them with the Quantity and ElementSet objects.
5. Add the Link objects to the components and validate them.
6. Prepare the event listener to catch events.
7. Run the simulation (including preparation and finish).
8. Dispose of the components.

You must also continuously catch exceptions on all steps.

Figure 3-8 provides a (pseudo) C#-code example that includes all phases in the utilization of OpenMI components. Note that this example does not include data operations and associated validations, while the calendar conversion between the DateTime data type of C# and the OpenMI Time object is omitted.



```

// --- Read the OMI files ----
// Note for namespaces: the abbreviation MC refers to Model Components

// --- MyRain-module hard-coded OMI file information ----
string myRainModuleAssemblyFile =
    "C:\\OpenMI\\Examples\\MC\\SimpleRain\\org.OpenMI.Examples.MC.DLL";
string myRainModuleClass      = "org.OpenMI.Examples.MC.SimpleRain";
string myRainArguments       = new Argument[1];
myRainArguments[0]          = new Argument("FilePath",
    "C:\\OpenMI\\Examples\\MC\\SimpleRain\\Data",true,"");

// --- MyRainfallRunoff-model hard-coded OMI file information ----
string myRRModelAssemblyFile =
    "C:\\OpenMI\\Examples\\MC\\SimpleRR\\org.OpenMI.Examples.MC.DLL";
string myRRModelClass        = "org.OpenMI.Examples.MC.SimpleRainfallRunoff";
string myRRArguments        = new Argument[1];
myRRArguments[0]            = new Argument("FilePath",
    "C:\\OpenMI\\Examples\\MC\\SimpleRainfallRunoff\\Data",true,"");

// --- MyRiverModel hard-coded OMI file information ----
string myRiverModelAssemblyFile =
    "C:\\OpenMI\\Examples\\MC\\SimpleRiver\\org.OpenMI.Examples.MC.DLL";
string myRiverModelClass     = "org.OpenMI.Examples.MC.SimpleRiver";
string myRiverArguments     = new Argument[1];
myRiverArguments[0]        = new Argument("FilePath",
    "C:\\OpenMI\\Examples\\MC\\SimpleRiver\\Data",true,"");

// -- Create LinkableComponents ---
ILinkableComponent MyRainModule = new SimpleRainModuleWrapper();
ILinkableComponent MyRRModel    = new SimpleRREngineWrapper();
ILinkableComponent MyRiverModel = new SimpleRiverEngineWrapper();
ILinkableComponent trigger      = new Trigger();
MyRainModule.Initialize(myRainArguments);
MyRRModel.Initialize(myRRArguments);
MyRiverModel.Initialize(myRiverArguments);

// --- Query Quantities and ElementSets

// Outputs
IQuantity Rain_Precipitation = (MyRainModule).GetOutputExchangeItem(0).Quantity;
IQuantity RR_Outflows        = (MyRRModel).GetOutputExchangeItem(0).Quantity;
IQuantity Riv_WaterLevels    = (MyRiverModel).GetOutputExchangeItem(0).Quantity;
IElementSet Rain_MyRainGrid = (MyRainModule).GetOutputExchangeItem(0).ElementSet;
IElementSet RR_Outlets      = (MyRRModel).GetOutputExchangeItem(0).ElementSet;
IElementSet Riv_RiverNetwork = (MyRiverModel).GetOutputExchangeItem(0).ElementSet;
IDataOperation Rain_TmAvg    = (MyRainModule).GetOutputExchangeItem(0).DataOperation(0);
IDataOperation Rain_TmAccu   = (MyRainModule).GetOutputExchangeItem(0).DataOperation(1);
IDataOperation Rain_SptAvg   = (MyRainModule).GetOutputExchangeItem(0).DataOperation(2);
IDataOperation RR_TmAvg      = (MyRRModel).GetOutputExchangeItem(0).DataOperation(0);
IDataOperation RR_TmMaxVal   = (MyRRModel).GetOutputExchangeItem(0).DataOperation(1);
IDataOperation Riv_SpatIntp  =
    (MyRiverModel).GetOutputExchangeItem(0).DataOperation(0);

//Inputs
IQuantity RR_Rainfall        = (MyRRModel).GetInputExchangeItem(0).Quantity;
IQuantity Riv_LateralInflows = (MyRiverModel).GetInputExchangeItem(0).Quantity;
IElementSet RR_SubCatchments = (MyRRModel).GetInputExchangeItem(0).ElementSet;
IElementSet Riv_LateralInlets = (MyRiverModel).GetInputExchangeItem(0).ElementSet;

```

```

// TimeHorizon
DateTime start = MyRainModule.TimeHorizon.Start;
if (start < MyRRModel.TimeHorizon.Start)
{
    start = (DateTime)MyRRModel.TimeHorizon.Start;
}
if (start < MyRiverModel.TimeHorizon.Start)
{
    start= (DateTime)MyRiverModel.TimeHorizon.Start;
}
DateTime end = MyRainModule.TimeHorizon.End;
if (end < MyRRModel.TimeHorizon.End)
{
    end = (DateTime)MyRRModel.TimeHorizon.End;
}
if (end < MyRiverModel.TimeHorizon.End)
{
    end= (DateTime)MyRiverModel.TimeHorizon.End;
}
DateTime stop = end + 0.000001;

// --- Create Links ---
// create data operations here
IDataOperation Rain_DataOperations[] = new Rain_DataOperation[1];
Rain_DataOperation(0)           = new Rain_TmAccu;
Rain_DataOperation(1)           = new Rain_SpatAvg;

// check validity: to be done raise exception
// Rain_DataOperation(1).IsValid((MyRainModule).GetOutputExchangeItem(0),
// (MyRRModel).GetInputExchangeItem(0), Rain_DataOperation[0])
ILink triggerLink = new Link(MyRiverModel, Riv_RiverNetwork, Riv_WaterLevels, trigger,
    "", "", "RiverModel to Trigger Link", "RiverModelToTrigger", new ArrayList());
ILink LinkRunoffToInflow = new Link(MyRRModel, RR_Outlets, RR_Outflow, MyRiverModel,
    Riv_LateralInlets, Riv_LateralInflows, "Link Runoff to River Inflow",
    "LinkRunoffToInflow", new ArrayList());
ILink LinkRainToCatchm = new Link(MyRainModule, MyRainGrid, Rain_Precipitation,
    MyRRModel, RR_SubCatchments, RR_Rainfall, "Link rainfall to catchment",
    "LinkRainToCatchm", Rain_DataOperations[]);

// --- Add Links ---
MyRiverModel.AddLink(triggerLink);
trigger.AddLink(triggerLink);
MyRainModule.AddLink(LinkRainToCatchm);
MyRRModel.AddLink(LinkRainToCatchm);
MyRRModel.AddLink(LinkRunoffToInflow);
MyRiverModel.AddLink(LinkRunoffToInflow);

// --- Validate Components ---
trigger.Validate();
MyRainModule.Validate();
MyRRModel.Validate();
MyRiverModel.Validate();

//--- Prepare Event listener ---
org.OpenMI.Standard.IListener myListener = new EventListener();
for (int i = 0; i < myListener.GetAcceptedEventTypeCount(); i++)
{
    for (int n = 0; n < MyRainModule.GetPublishedEventTypeCount(); n++)
    {

```

```

        if (myListener.GetAcceptedEventType(i) ==
            MyRainModule.GetPublishedEventType(n))
        {
            MyRainModule.Subscribe(myListener,
                myListener.GetAcceptedEventType(i));
        }
    }
    for (int n = 0; n < MyRRModel.GetPublishedEventTypeCount(); n++)
    {
        if (myListener.GetAcceptedEventType(i) ==
            MyRRModel.GetPublishedEventType(n))
        {
            MyRRModel.Subscribe(myListener, myListener.GetAcceptedEventType(i));
        }
    }
    for (int n = 0; n < MyRiverModel.GetPublishedEventTypeCount(); n++)
    {
        if (myListener.GetAcceptedEventType(i) ==
            MyRiverModel.GetPublishedEventType(n))
        {
            MyRiverModel.Subscribe(myListener,
                myListener.GetAcceptedEventType(i));
        }
    }
}

//--- Run ---
try
{
    MyRainModule.Prepare();
    MyRRModel.Prepare();
    MyRiverModel.Prepare();
    int nrsteps=100;
    DateTimestep = (end - start)/nrsteps;
    DateTime stop = end + 0.000001;
    DateTime _time = start;
    while (_time <= stop)
    {
        Application.DoEvents();
        IValueSet Values = _trigger.GetValues(new TimeStamp(_time),
            _triggerLink.ID);
        _time = _time.AddSeconds(step);
    }
    GermanRhineModel.Finish();
    NethRhineModel.Finish();
} // end try
//--- Clean up ---
MyRainModule.Dispose();
MyRRModel.Dispose();
MyRiverModel.Dispose();
}
//--- Exception Handling ---
catch (Exception e)
{
    // write exception to screen;
    Console.WriteLine(e);
}
}

```

**Figure 3-8 Hard-coded system using OpenMI linkable components**



## Chapter 3.4 Support for configurable systems

In situations where modelling is a 'production' facility, model combination requires as minimal resources as possible. Standardized interfaces are an important step forward but to minimize operational costs it is desirable to prevent the need for hard-coding each time a new model combination is made. The OpenMI becomes powerful if tools are provided that make full use of the metadata exposure when developing model combinations.

This chapter discusses the main aspects of a configurable system, with some details of the tools provided in the OpenMI Software Development Kit.

### 3.4.1 Main aspects of a configurable system

If you are developing tools that use metadata exposure when developing model combinations, you should keep in mind that organizations might have organized their working procedures in such way that different people are responsible for different jobs (or different computer systems are used). Jobs that might be distinguished are the development of model schematizations, composing and configuring model combinations, and deploying and running models. Each of these jobs might put different requirements on the tools. The ability to run jobs in a batch process, or run different jobs on different machines, requires that the associated administrative part (i.e. the model configurations) is not tied to a specific user interface.

Therefore it is recommended that you take this separation of concerns into account when developing any supportive tools. The following functional distinction is desired:

- *Component definition*: ability to identify and locate components
- *Configuration* (administration/composition of linked components): ability to create, configure and validate model combinations; ability to save and load those configurations to a persistent store
- *Deployment*: ability to instantiate the components in a generic way and start running them
- *User interface components* that implement the above functions

All these items play an important role in improving the usability of the OpenMI.

In essence, the component definition is determined by the OMI file (see Section 3.1.2). All other items are discussed in the remainder of this chapter. Note however that the open source OpenMI configuration editor (OmiEd) described in Chapter 2.4 combines all functionality in one package. Sections 3.4.2 and 3.4.3 address the separation of business logic and GUI underlying the `org.OpenMI.Utilities.Configuration` package.

### 3.4.2 Configuring and sustaining a component combination

Configuration is the task in which links are defined between the components involved. To enable re-use of such configurations, a persistent storage facility is required. This section describes the facilities provided in the OpenMI Software Development Kit by the `org.OpenMI.Utilities.Configuration` package.

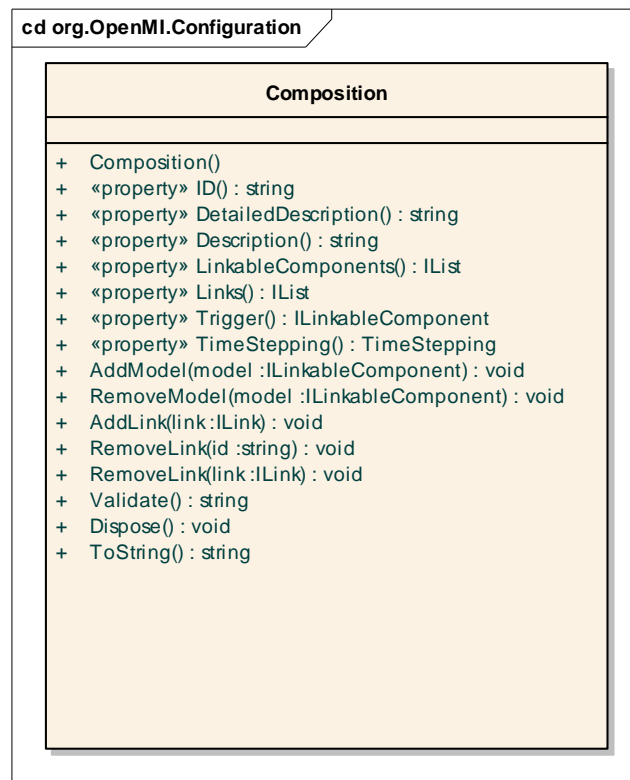
To enable persistent storage and exchange of a model configuration between modellers, the details of the configuration of the components involved (i.e. OMI files) and the links between the components are saved. The saved configuration is called a *composition*. To deploy (i.e. run) such a composition, the linkable components need to be created and the links should be added.

The composition process consists of the following steps:

1. Create a new composition object or load an existing composition.
2. Select the linkable components involved (e.g. by loading the OMI files). If required, modify their properties (and save).
3. Create the links and add them to the components; define the links' properties.
4. Populate the link properties, while paying specific attention to the validation of the selected data operations.
5. Validate the components and the links.
6. Specify the time frame (begin, end, step size).
7. Save the composition.
8. If required, deploy the composition.

Validation is an important step in the configuration phase. Typically, the validation will generate a string message, informing the user of the validity of the component and its links.

Within the OpenMI Software Development Kit, the `Composition` class is responsible for the composition details (Figure 3-9).



**Figure 3-9 Composition class (org.OpenMI.Utilities.Configuration package)**

This composition can have a persistent representation in an XML file (Figure 3-10 and Figure 3-11). The XML file contains two pieces of information:

- Involved components (references to OMI files)
- Link definitions

This file can be generated automatically from the class or it can be created manually and manipulated.

The OpenMI Software Development Kit contains a generic, customizable XML parser to parse (load) and serialize (store) this file. This XML parser allows XML elements of the file (e.g. components, links, quantities, element sets) to be defined 'in-line' or 'by reference'. The reference can be to an XML element which has been defined earlier in the file, or it can refer to an object that can be generated by instantiating an associated class. This reference feature allows various components to share the same element sets or quantities and, if needed, provide the information on-the-fly.



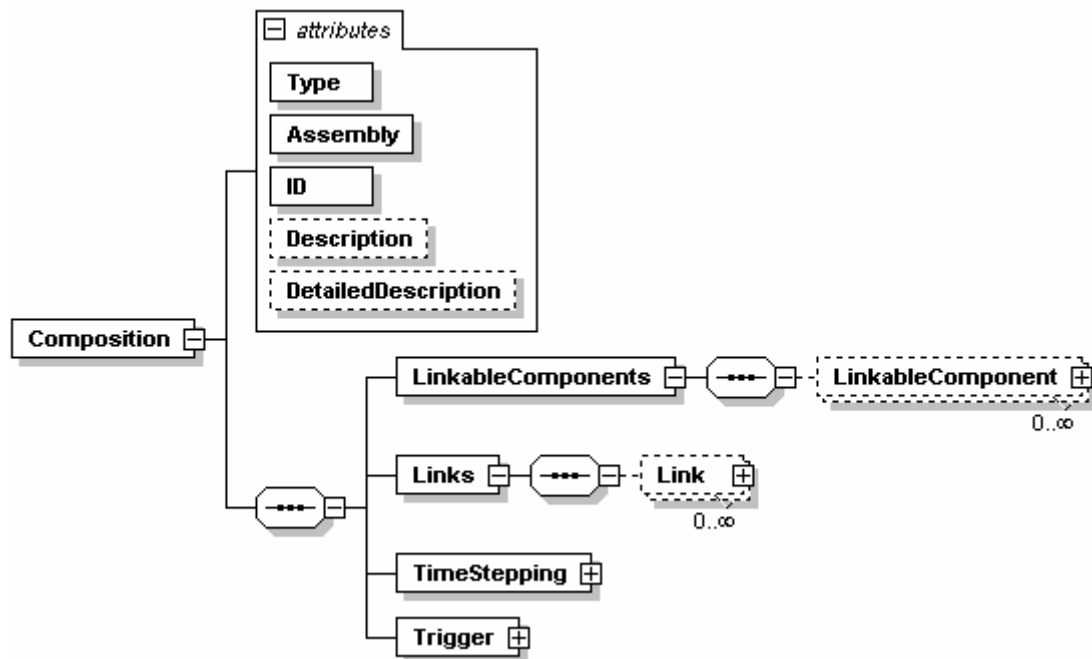


Figure 3-10 Visual representation of the composition

```

<?XML version="1.0"?>

<Composition XMLns="http://www.openmi.org/Composition.xsd"
  Type="org.OpenMI.Utilities.Configuration.Composition"
  Assembly="org.OpenMI.Utilities.Configuration, Version=1.4.0.0, Culture=neutral,
  PublicKeyToken=8384b9b46466c568" Description="Example Composition"
  DetailedDescription="" ID="65adab37-ac7d-423a-8c41-c42dc216c0a3">
  <LinkableComponents>
    <LinkableComponent Type="org.OpenMI.Examples.MC.SimpleRain"
      Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
      File="..\data\SimpleRain.omi" />
    <LinkableComponent Type="org.OpenMI.Examples.MC.SimpleRR"
      Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
      File="..\data\SimpleRR.omi" />
    <LinkableComponent Type="org.OpenMI.Examples.MC.SimpleRiver"
      Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
      File="..\data\SimpleRiver.omi" />
  </LinkableComponents>
  <Links>
    <Link Description="Link Rainfall to Catchment">
      <DataOperations>
        <DataOperation Type="org.OpenMI.Examples.MC.Interpolator"
          Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
          ID="Rain_TmAccu">
          <Arguments />
        </DataOperation>
        <DataOperation Type="org.OpenMI.Examples.MC.Interpolator"
          Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
          ID="Rain_SpatAvg">
          <Arguments />
        </DataOperation>
      </DataOperations>
      <SourceComponent Type="org.OpenMI.Examples.MC.SimpleRain"
        Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
        File="..\data\SimpleRain.omi" />
      <SourceElementSet Type="org.OpenMI.Backbone.ElementSet"
        Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
        PublicKeyToken=8384b9b46466c568" File="..\data\SimpleRain.omi"
        RefID="Rain_MyRainGrid" />
      <SourceQuantity Type="org.OpenMI.Backbone.Quantity"
        Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
        PublicKeyToken=8384b9b46466c568" Description="Rainfall /
        Precipitation"
        ID="Rain_Precipitation" ValueType="Scalar">
        <Dimension AmountOfSubstance="0" Currency="0"
          ElectricCurrent="0" Length="1" LuminousIntensity="0" Mass="0"
          Temperature="0" Time="-1" />
        <Unit ConversionFactorToSI="2.778e-7" Description="mm per hour"
          ID="mm/h" OffSetToSI="0" />
      </SourceQuantity>
      <TargetComponent Type="org.OpenMI.Examples.MC.SimpleRR"
        Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
        File="..\data\SimpleRR.omi" />
      <TargetElementSet Type="org.OpenMI.Backbone.ElementSet"
        Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
        PublicKeyToken=8384b9b46466c568" File="..\data\SimpleRR.omi"
        RefID="RR_SubCatchments" />
      <TargetQuantity Type="org.OpenMI.Backbone.Quantity"
        Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,

```

```

        PublicKeyToken=8384b9b46466c568 "
        Description="Rainfall on catchment"
        ID="RR_Rainfall" ValueType="Scalar">
            <Dimension AmountOfSubstance="0" Currency="0"
                ElectricCurrent="0" Length="1" LuminousIntensity="0" Mass="0"
                Temperature="0" Time="-1" />
            <Unit ConversionFactorToSI="2.778e-7" Description="" ID="m3/s"
                OffSetToSI="0" />
        </TargetQuantity>
    </Link>
    <Link Description="Link Runoff To River Inflow" ID="LinkRunoffToInflow">
        </DataOperations>
        <SourceComponent Type="org.OpenMI.Examples.MC.SimpleRR"
            Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
            File="..\data\SimpleRR.omi" />
        <SourceElementSet Type="org.OpenMI.Backbone.ElementSet"
            Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
            PublicKeyToken=8384b9b46466c568" File="..\data\SimpleRR.omi"
            RefID="RR_Outlets" />
        <SourceQuantity Type="org.OpenMI.Backbone.Quantity"
            Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
            PublicKeyToken=8384b9b46466c568" Description="Runoff Outflows"
            ID="RR_Outflows " ValueType="Scalar">
            <Dimension AmountOfSubstance="0" Currency="0"
                ElectricCurrent="0" Length="3" LuminousIntensity="0" Mass="0"
                Temperature="0" Time="-1" />
            <Unit ConversionFactorToSI="1" Description="m3/s" ID="m3/s"
                OffSetToSI="0" />
        </SourceQuantity>
        <TargetComponent Type="org.OpenMI.Examples.MC.SimpleRiver"
            Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
            File="..\data\SimpleRiver.omi" />
        <TargetElementSet Type="org.OpenMI.Backbone.ElementSet"
            Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
            PublicKeyToken=8384b9b46466c568" File="..\data\SimpleRiver.omi"
            RefID="Riv_LateralInlets" />
        <TargetQuantity Type="org.OpenMI.Backbone.Quantity"
            Assembly="org.OpenMI.Backbone, Version=1.4.0.0, Culture=neutral,
            PublicKeyToken=8384b9b46466c568"
            Description="Lateral Inflows into River"
            ID="Riv_LateralInflows" ValueType="Scalar">
            <Dimension AmountOfSubstance="0" Currency="0"
                ElectricCurrent="0" Length="3" LuminousIntensity="0" Mass="0"
                Temperature="0" Time="-1" />
            <Unit ConversionFactorToSI="1" Description="" ID="m3/s"
                OffSetToSI="0" />
        </TargetQuantity>
    </Link>
</Links>
<TimeStepping End="46097" Start="46066" Step="10800" />
<Trigger Type="org.OpenMI.Examples.MC.SimpleRiver"
    Assembly="C:\OpenMI\Examples\MC\org.OpenMI.Examples.MC.DLL"
    File="..\data\SimpleRiver.omi" />
</Composition>

```

**Figure 3-11 Example XML file for a composition (org.OpenMI.Utilities.Configuration)**

To ensure the validity of the file and its contents, an XML schema definition has been created (details are given in the technical documentation of the Configuration package). Using this schema, the file is validated during parsing to ensure that a proper Composition object can be instantiated and populated.

Figure 3-12 provides a code example to illustrate how a composition can be loaded, manipulated and saved.

```
// -- Create LinkableComponents ---
// ... see hard-coded example ...

// Reads composition from file
Composition MyComposition = (Composition) XMLFile.GetRead (new
    FileInfo("mycomposition.XML"));

// -- alternative: Create Composition ---
// Composition MyComposition = new Composition();

// -- Add LinkableComponents to Composition ---
MyComposition.AddModel(MyRiverModel);
MyComposition.AddModel(MyRRModel);
MyCompositon.AddModel(MyRainModule);

// -- Create Links ---
// ..... see hard-coded example ...

// -- Add Links to Composition ---
MyComposition.AddLink(triggerLink);
    // Composition updates internal administration,
    // calls MyRiverModel.AddLink(triggerLink)
    // and calls trigger.AddLink(triggerLink);
MyComposition.AddLink(LinkRainToCatchm);
    // Composition updates internal administration,
    // calls MyRainModule.AddLink(LinkRainToCatchm)
    // and calls MyRRModel.AddLink(LinkRainToCatchm)
MyComposition.AddLink(LinkRunoffToInflow);
    // Composition updates internal administration,
    // calls MyRRModel.AddLink(LinkRunoffToInflow)
    // and calls MyRiverModel.AddLink(LinkRunoffToInflow)
// -- Create and add Time information -
TimeStepping MyTimeInfo = new TimeStepping();
MyTimeInfo.Start = 46066.0; //01-01-1985
MyTimeInfo.End = 46097.0; //01-02-1985
myTimeInfo.Step = 60; // 60 seconds
MyComposition.timeStepping = MytimeInfo;

// -- Assigning the trigger --
MyComposition.Trigger = MyRiverModel;

// -- Write Composition to file ---
XMLFile.Write (MyComposition, new FileInfo("mycomposition.XML"));
```

**Figure 3-12 Manipulating the composition**

### 3.4.3 Deploying and running the system

In order to run a set of integrated models, a deployer component can be used. The task of the deployer is to instantiate the linkable components, to call the Initialize methods on the linkable components, to call the AddLink method on all the linkable components, to prepare the components and finally start the computation/data transfer by calling one or more GetValues methods on the linkable components. The deployer is also responsible for connecting event publishers and event listeners to each other. Event listeners can include a graphical display of values or an event logger. After the simulation has finished, the deployer calls the Finalize methods on all linkable components.

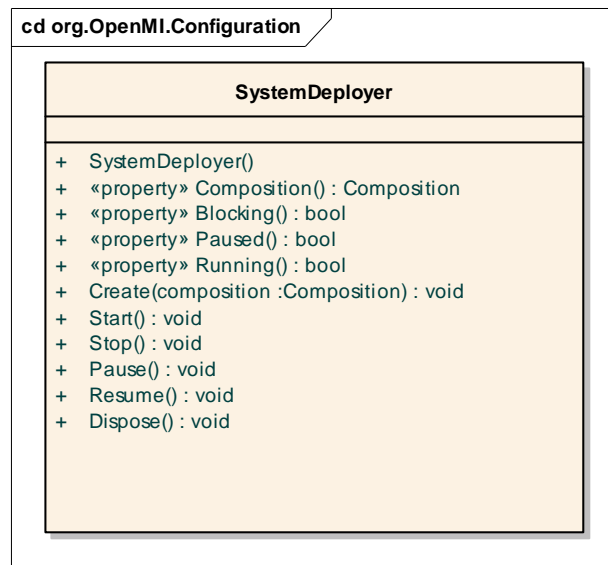
Preferably, the deployer component is designed in such way that it can operate either with or without a user interface. Taking a Composition object as input, the essential functions actually are:

- 'Create' to establish an in-memory representation of the populated components: methods to be invoked after instantiation are Initialize and AddLinks.
- 'Start' and 'Stop' to run the computation process: methods to be invoked are Prepare, GetValues and Finish.
- 'Pause' and 'Resume' to enable interruptions from the outside world on the computation process: catch events and hold/return control when needed.
- Catch exceptions, if they cannot be resolved, and pass them to the user.

When implementing such functionality, you should keep in mind that the OpenMI has been designed so that the data transfer process (i.e. the GetValues call stack) is in the same thread. This does not mean that a linkable component should execute its computation in the same thread, however.

Within the documentation of the Standard various ways are described to stop a computation or pause and resume. As indicated in Section 3.2.5, it is recommended that a timestep loop is applied to manage the process. Irresolvable exceptions will eventually pop up at the deployment level, where the user interface can be activated for manual assistance.

The Deployer component of the OpenMI Software Development Kit has been designed with the functionality described above (Figure 3-13).



**Figure 3-13 Deployer class of the org.OpenMI.Utilities.Configuration package**

Figure 3-14 provides a code example of how to deploy a composition. Since the Deployer catches events during the time loop, the computation can be paused (or stopped) by simply calling the Pause method.

```
// pre-condition: the MyComposition-object is available
// this example communicates with the user via a console-object
// -- Validate composition --
if (!MyComposition.Validate().Equals(""))
{
    Console.Write ("Composition cannot be run: ");
    Console.WriteLine (MyComposition.Validate());
    Wait();
    return;
}

// -- Create system deployer --
SystemDeployer MyDeployer = new SystemDeployer();
MyDeployer.Blocking = true;

// -- Assign Composition --
MyDeployer.Create (MyComposition);

// -- Start Computation --
// Composition will run for period as indicated in timestepping object
try
{
    MyDeployer.Start();
}
catch (Exception e)
{
    // raise exception
    Console.WriteLine (e.Message);
    Wait();
    return;
}
```

**Figure 3-14 Example using the deployer**

Figure 3-15 provides a code example of a fully functioning executable that runs a composition from the command line.

```
using System;
using System.IO;
using org.OpenMI.Utilities.Configuration;
using org.OpenMI.Utilities.Configuration.XML;
using org.OpenMI.Standard;
using org.OpenMI.DevelopmentSupport;

namespace org.OpenMI.Configuration.RunOpenMI
{
    /// <summary>
    /// Console application to run an OpenMI composition from the command line
    /// </summary>

    class RunOpenMI
    {

        private static bool _wait = false;
        /// <summary>
        /// Console application for running an OpenMI composition
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            if (args.Length == 0)
            {
                Console.WriteLine ("Usage : RunOpenMI <file> [-wait]");
                Console.WriteLine ("");
                Console.WriteLine ("<file> = full path to XML file containing
                    composition");
                Console.WriteLine ("-wait = After run application waits to
                    terminate until user presses <Enter>");
                Console.WriteLine ("");
                _wait = true;
                Wait();
                return;
            }
            for (int i = 1; i < args.Length; i++)
            {
                if (args[i].Equals ("-wait"))
                {
                    _wait = true;
                }
            }

            // -- Check file availability --
            FileInfo file = new FileInfo (args[0]);
            if (!file.Exists)
            {
                Console.WriteLine ("File {0} not found", file.FullName);
                Wait();
                return;
            }

            // -- Initialize XML-parser --
            XMLConfiguration.Initialize();

            // -- Parse XML file --
            object fileObject;
```



```
try
{
    fileObject = XMLFile.GetRead (file);
}
catch (Exception e)
{
    Console.WriteLine (e.Message);
    Wait();
    return;
}
if (!(fileObject is Composition))
{
    Console.WriteLine ("File {0} does not contain a composition",
        file.FullName);
    Wait();
    return;
}

// -- Create Composition-object from file and validate composition --
Composition composition = (Composition) fileObject;
if (!composition.Validate().Equals(""))
{
    Console.Write ("Composition cannot be run: ");
    Console.WriteLine (composition.Validate());
    Wait();
    return;
}

// -- Create Deployer-object and assign composition to be deployed --
SystemDeployer deployer = new SystemDeployer();
deployer.Blocking = true;

try
{
    deployer.Create (composition);
}
catch (Exception e)
{
    Console.WriteLine (e.Message);

    Wait();
    return;
}

// -- Start Computation --
Console.WriteLine ("Starting composition {0}", file.FullName);
try
{
    deployer.Start();
}
catch (Exception e)
{
    Console.WriteLine (e.Message);
    Wait();
    return;
}
Console.WriteLine ("Finished composition {0}", file.FullName);
Wait();
}
```

```
private static void Wait()
{
    if (_wait)
    {
        Console.WriteLine ("Press <Enter> to continue");
        int key = Console.Read();
    }
}
}
```

**Figure 3-15** Example of an executable to run a composition from the command line

## Chapter 3.5 Graphical user interfaces

Visual tools help in the configuration phase. Visual tools can be standalone, very basic, very sophisticated or fully embedded in the default environment of a software product. Preferably they should support all the configuration steps discussed in the previous chapter.

The OpenMI has been designed in such way that its functionality can be completely separated from any visual tool. The `org.OpenMI.Tools` namespace is the only namespace containing visual forms; none of the other namespaces include any visual forms. This is also true for the code examples in this book.

However, visual tools are convenient to assist in configuration, deployment and visualization. This chapter briefly discusses how to connect a user interface to the composition. In addition, it discusses the user interface being shipped with the OpenMI Software Development Kit.

### 3.5.1 Building visual tools

The `org.OpenMI.Utilities.Configuration` package has been developed in such way that it can be re-used in many ways, without being forced to use the visual tools of the OpenMI Software Development Kit.

First of all, a default composition, persistently stored in an XML file, can be used as a template for actual model runs. Such a default composition could contain a list of software units, with references to OMI files and possibly a default list of links that are described with default names for quantities and element sets (e.g. inflow at lateral inlets). For instance, the composition file in Figure 3-11 could act as a template. Typically the attributes in the OMI file will be used during initialization and hence determine which schematization and which calculation points are associated with this element set.

A default composition can provide a good starting point for your code. Via code you can add (or delete) components, modify input data references and modify links and the timestep information. For example, your code might manage scenarios that directly adapt the composition based on the user selection. The actual code will be a variation of the code example that manipulates a composition (see Figure 3-12 in Section 3.4.2).

### 3.5.2 OmiEd, a simple front end of the OpenMI SDK

The first OpenMI configuration editor utilized the `org.OpenMI.Utilities.Configuration` package, which is based on persistent storage of all descriptive information.

The second editor – OmiEd, being released in open source – was developed as a straightforward tool built from scratch. Figure 3-16 shows a sample OmiEd display. This lightweight tool depends much more on the run-time inspection capabilities of linkable components. The associated configuration file (Figure 3-17) has therefore become much more readable.

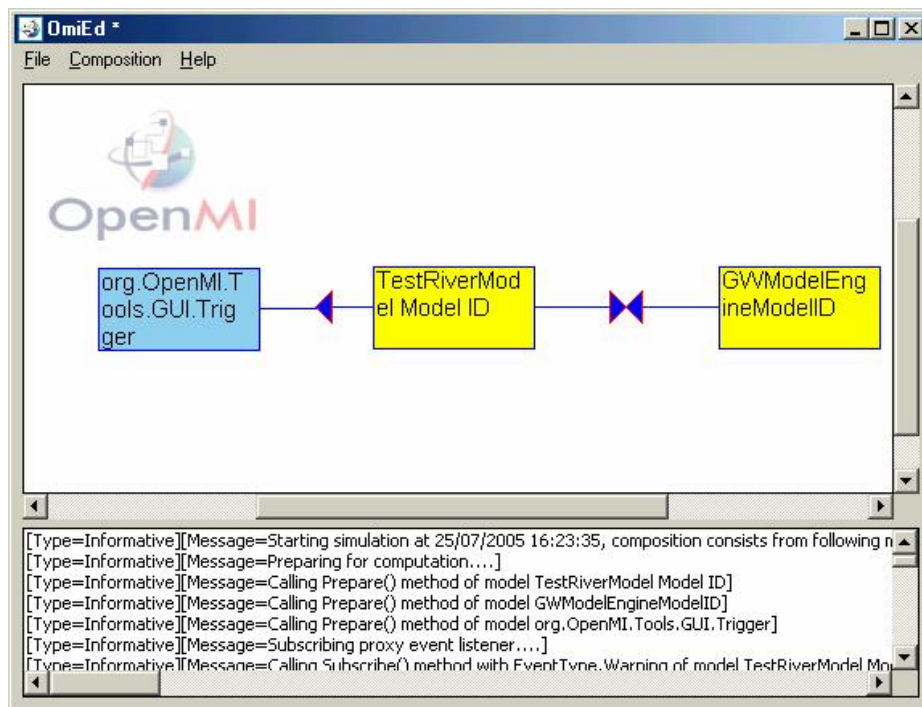


Figure 3-16 OmiEd – front-end application of the OpenMI SDK

```
<guiComposition version="1.4">
  <models>
    <model omi="D:\openmi-demo\omi\SobekRR-hills.omi" rect_x="30" rect_y="30"
rect_width="100" rect_height="51" />
    <model omi="D:\openmi-demo\omi\SobekCF-river_remote.omi" rect_x="231"
rect_y="33" rect_width="100" rect_height="51" />
    <model omi="org.OpenMI.Tools.GUI.Trigger" rect_x="436" rect_y="59"
rect_width="100" rect_height="51" />
  </models>
  <links>
    <uilink model_providing="RR-D:\openmi-demo\data\sobek-rr-hills\sobek_3b.fnm"
model_accepting="CF-D:\openmi-demo\data\sobek-cf-river\sobeksim.fnm">
      <link id="1" source_elementset="RR-Boundaries" source_quantity="Flow"
target_elementset="Laterals" target_quantity="Discharge" />
    </uilink>
    <uilink model_providing="CF-D:\openmi-demo\data\sobek-cf-river\sobeksim.fnm"
model_accepting="org.OpenMI.Tools.GUI.Trigger">

```

```
<link id="2" source_elementset="HBoundaries" source_quantity="Discharge"
target_elementset="TriggerElementID" target_quantity="TriggerQuantityID" />
</uilink>
</links>
<runproperties listenedeventtypes="1111111111" triggerinvoke="1/2/2000 12:00:00
AM" runinsamethread="0" showeventsinlistbox="1" logfile="CompositionRun.log" />
</guiComposition>
```

**Figure 3-17 Sample configuration file for OmiEd**

# Book 4 Migrating OpenMI models

<b>BOOK 4</b>	<b>MIGRATING OPENMI MODELS .....</b>	<b>4-1</b>
<b>Chapter 4.1</b>	<b>Introduction .....</b>	<b>4-3</b>
4.1.1	OpenMI compliance.....	4-4
4.1.2	The Simple River example .....	4-6
<b>Chapter 4.2</b>	<b>Planning the migration .....</b>	<b>4-7</b>
4.2.1	Use cases.....	4-8
4.2.1.1	Use case 1: Connecting to other rivers.....	4-8
4.2.1.2	Use case 2: Inflow from geo-referenced catchment database.....	4-10
4.2.2	Defining exchange items.....	4-12
<b>Chapter 4.3</b>	<b>Wrapping.....</b>	<b>4-13</b>
4.3.1	A general wrapping pattern .....	4-14
4.3.2	The LinkableEngine .....	4-15
<b>Chapter 4.4</b>	<b>Migration – step by step.....</b>	<b>4-17</b>
4.4.1	Step 1: Changing your engine core.....	4-18
4.4.2	Step 2: Creating the .NET assemblies.....	4-20
4.4.3	Step 3: Accessing the functions in the engine core.....	4-22
4.4.4	Step 4: Implementing MyEngineDotNetAccess.....	4-24
4.4.5	Step 5: Implementing the MyEngineWrapper class.....	4-26
4.4.6	Step 6: Implementing MyModelLinkableComponent .....	4-28
4.4.7	Step 7: Implementation of the remaining IEngine methods .....	4-29
<b>Chapter 4.5</b>	<b>Migration of the Simple River.....</b>	<b>4-31</b>
4.5.1	The Simple River wrapper.....	4-32
4.5.2	Implementation of the Initialize method .....	4-33
4.5.3	Implementation of the SetValues method .....	4-36
4.5.4	Implementing the GetValues method .....	4-37
4.5.5	Implementation of the remaining methods.....	4-38
<b>Chapter 4.6</b>	<b>Testing the component.....</b>	<b>4-41</b>
4.6.1	Unit testing .....	4-42
<b>Chapter 4.7</b>	<b>Implementing IManageState.....</b>	<b>4-45</b>
4.7.1	The IManageState interface.....	4-46
<b>Chapter 4.8</b>	<b>The OMI file.....</b>	<b>4-49</b>
4.8.1	Structure of the OMI file .....	4-50
<b>Chapter 4.9</b>	<b>Design patterns for model migration.....</b>	<b>4-51</b>
4.9.1	Design patterns for ISIS.....	4-52
4.9.2	Design patterns for InfoWorks RS .....	4-53
4.9.3	Design patterns for Mike11 .....	4-54

4.9.4	Design patterns for SOBEK .....	4-57
<b>Chapter 4.10</b>	<b>Performance issues.....</b>	<b>4-59</b>
4.10.1	Memory consumption.....	4-60
4.10.2	System processes .....	4-61



## Chapter 4.1 Introduction

Although it may appear a huge challenge to turn a model engine into an OpenMI-compliant linkable component, it may not be as difficult as it seems. The OpenMI Software Development Kit provides a large number of software utilities that make migration easier. These tools and utilities can be used by anyone migrating a model but are not required in order to comply with the OpenMI standard. The utilities can be used as a whole or you can select only a few of them; alternatively, you can use the utilities as the basis for your own implementations.

This book assumes that you will use the OpenMI utilities to the full extent. Step-by-step instructions are given for the whole migration process, from defining the requirements for an OpenMI component, through design and implementation to testing.

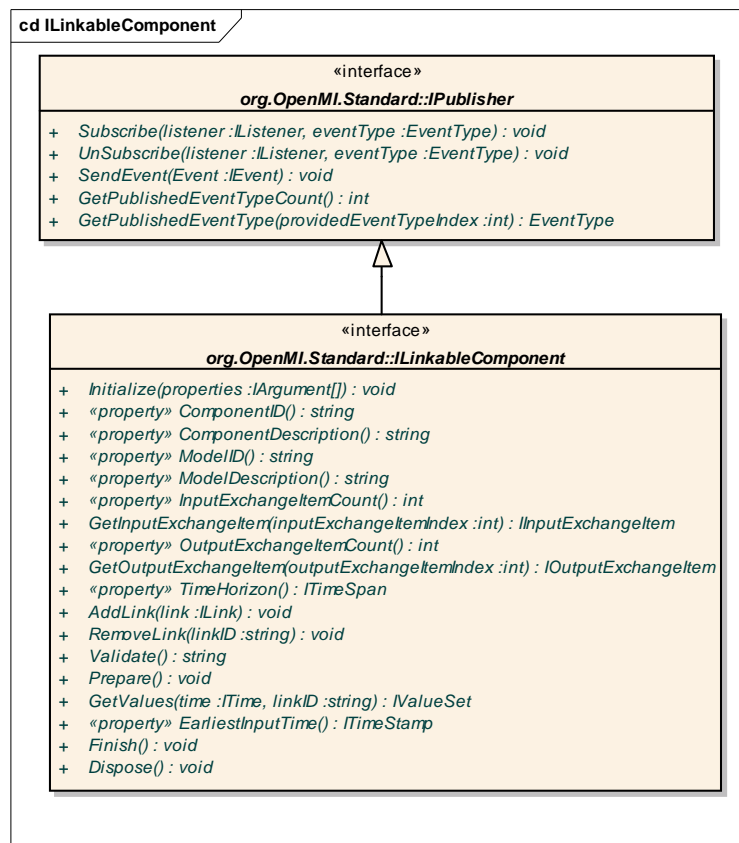
This chapter describes the requirements for OpenMI-compliance and introduces the Simple River model, which is used to illustrate the migration process.

### 4.1.1 OpenMI compliance

The official requirements for OpenMI compliance are given in Part C, *org.OpenMI.Standard interface specification*. The OpenMI utilities take care of most of the requirements for compliance.

There are three basic requirements for compliance:

1. The component must implement the `org.OpenMI.Standard.ILinkableComponent` interface (Figure 4-1).



**Figure 4-1 ILinkableComponent interface**

2. The component must be associated with an XML file containing information needed for deployment. The XML file must follow the LinkableComponent schema (Figure 4-2).

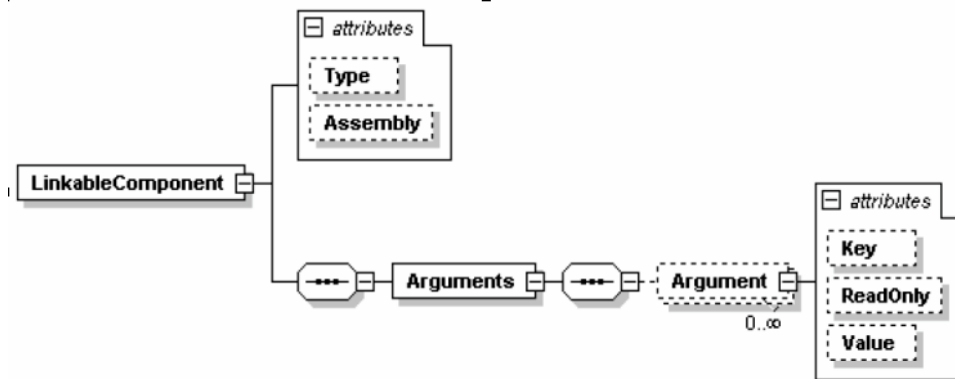


Figure 4-2 Schema for the LinkableComponent XML file

3. The component must be able to handle invocation of methods in the sequence shown in Figure 4-3.

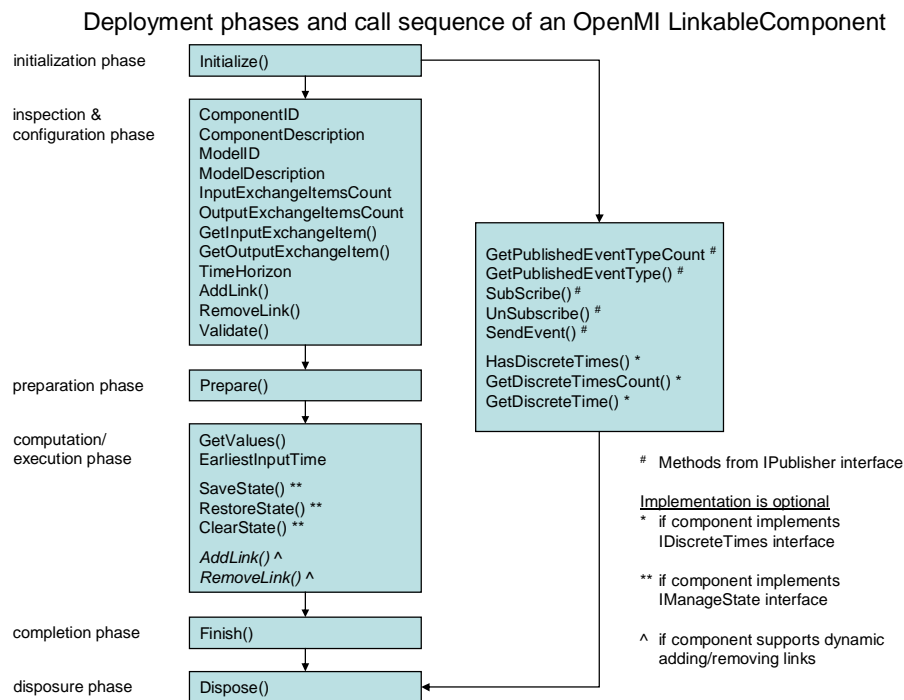
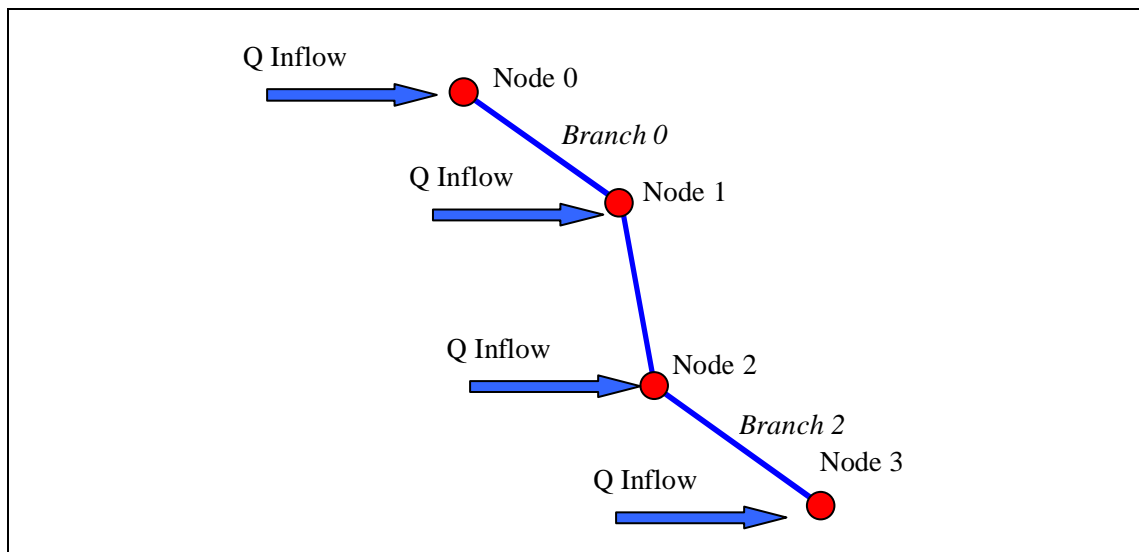


Figure 4-3 Required sequence for invocation of methods in the LinkableComponent

### 4.1.2 The Simple River example

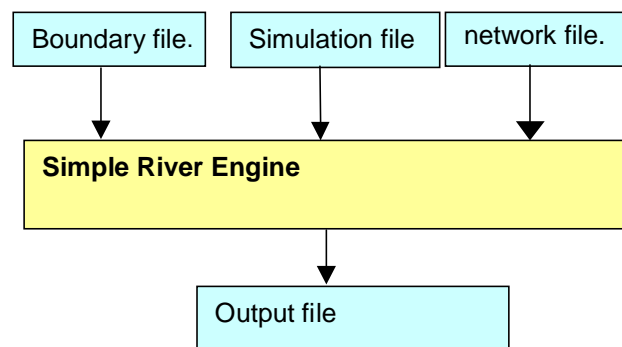
A Simple River model engine was developed as an example of model migration. The model engine is programmed in Fortran and is a very simple conceptual river model.

The Simple River consists of nodes and branches, as shown in Figure 4-4. For each timestep, the inflow to each node is obtained from a boundary-input file. These flow rates are multiplied by the timestep length and added to the storage in each node. Then, starting from the upstream end, the water is moved to downstream nodes and the flow rate in each branch is calculated.



**Figure 4-4 Simple River network**

The Simple River engine reads data from three input files, which contain information about the inflow to the river nodes (boundary file), the simulation period and timestep length (simulation file) and the river network (network file) – see Figure 4-5.



**Figure 4-5 Simple River input and output files**

The full source code for the Simple River model, the associated wrappers and the test classes used to migrate the model is available via [www.OpenMI.org](http://www.OpenMI.org).

## Chapter 4.2 Planning the migration

Before you start migrating a model it is important that you have a precise idea about how your model is intended to be used when it is running as an OpenMI component. Think about any situation where it will be useful to run your model linked to other OpenMI components. Such components could be other models, data providers, optimization tools or calibration tools. You may even find it useful to run two instances of your model component in the same configuration.

This chapter suggests ways in which you can plan the migration of a model, including the development of use cases and the definition of exchange items.

## 4.2.1 Use cases

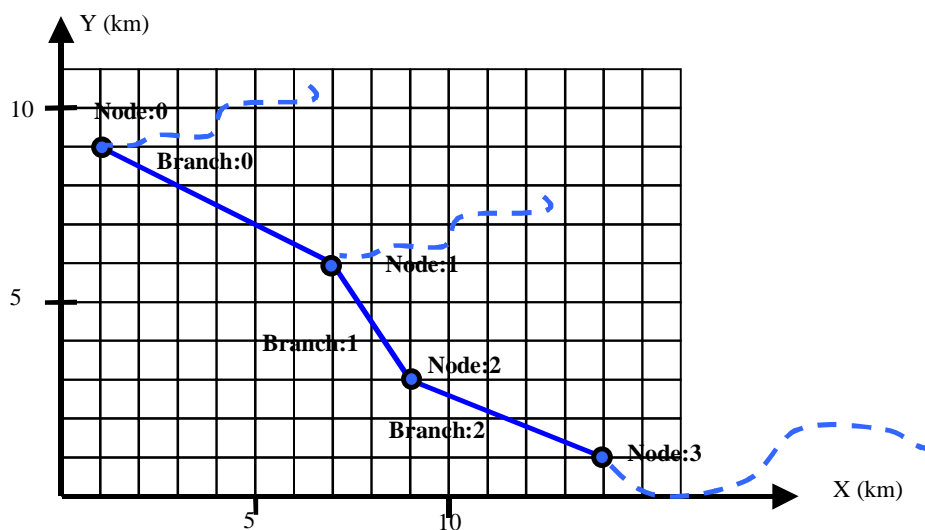
Use cases (examples of how software is used) have become very popular in software development. There are no formal requirements for defining a use case. However, what makes a use case different from an example is that a use case is more detailed and well defined. Most importantly, a use case must be formulated in such a way that, after completion of software development, you can unambiguously determine whether the use case is covered or not. The big advantage of use cases is that they are easily understood both by the software developer and the software user.

At the beginning of the development process, a number of use cases should be defined. It is important that the repository of use cases at any time, in all areas of the software development, reflects the current target. If a particular use case cannot be fulfilled it should be modified or removed.

Two use cases for the migrated Simple River model are given below. The use cases give a step-by-step description of how a user will use the models.

### 4.2.1.1 Use case 1: Connecting to other rivers

In the first use case, the Simple River model is connected to another OpenMI-compatible river model (Figure 4-6).



**Figure 4-6 Use case 1: Connecting to other rivers**

Preconditions:

- The model user has the OpenMI-compliant Simple River model installed on his PC.
- The model user has input files for the Simple River model available on his PC.
- The model user has an OpenMI configuration user interface installed on his PC.
- The model user has another OpenMI-compliant river model (including required data files) available on his PC.

Success guarantee (postconditions):

- All models have generated correct results.

Main success scenario:

1. The model user loads the OpenMI GUI on the PC.
2. The model user uses the GUI to browse for available LinkableComponents.
3. The model user finds the Simple River OMI file and the OMI file for the other river model.
4. The model user loads the two files (components) into the GUI.
5. The model user creates a unidirectional and ID-based link from the downstream node in the other river model to the upstream node in the Simple River.
6. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').
7. The model user defines the simulation period.
8. The model user runs the simulation.

Extensions to the use case provide alternative flows. Here, the flow splits from step 5 into two alternatives.

First alternative:

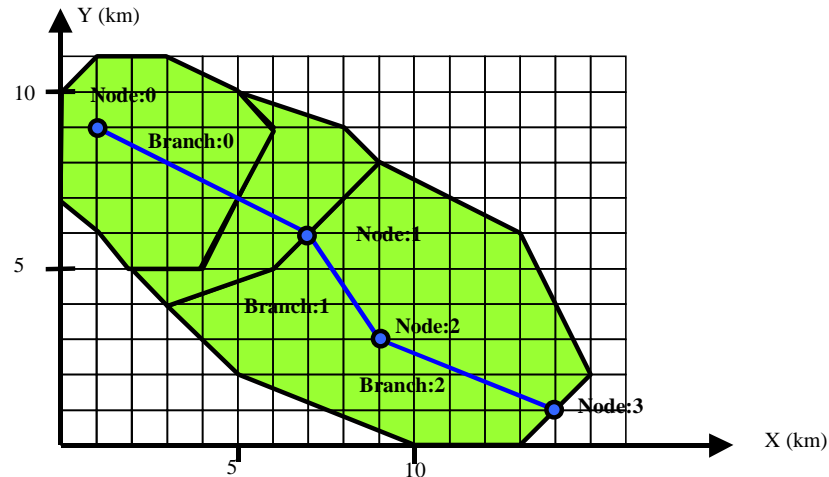
5. The model user creates a unidirectional and ID-based link from the downstream branch in the Simple River model to the upstream node in the other river model.
6. The model user selects input and output exchange items for the link (output quantity for the Simple River is 'flow').
7. The model user defines the simulation period.
8. The model user runs the simulation.

Second alternative:

5. The model user creates a unidirectional and ID-based link from the downstream branch in the other river model to an internal node in the Simple River model.
6. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').
7. The model user defines the simulation period.
8. The model user runs the simulation.

#### 4.2.1.2 Use case 2: Inflow from geo-referenced catchment database

In the second use case, the inflow for the Simple River model comes from an OpenMI-compliant runoff database (Figure 4-7).



**Figure 4-7 Use case 2: Inflow from catchments**

Preconditions:

- The model user has the OpenMI-compliant Simple River model installed on his PC.
- The model user has input files for the Simple River model available on his PC.
- The model user has an OpenMI configuration user interface installed on his PC.
- The model user has an OpenMI-compliant runoff database (including required data files) available on his PC.

Success guarantee (postconditions):

- All models have generated correct results.

Main success scenario:

1. The model user loads the OpenMI GUI on the PC.
2. The model user uses the GUI to browse for available LinkableComponents.
3. The model user finds the Simple River OMI file.
4. The model user finds the OMI file for the runoff database.
5. The model user loads the two files (components) into the GUI.
6. The model user creates a unidirectional and geo-referenced link from the runoff database to 'All Branches' input exchange item in the Simple River model.



7. The model user selects input and output exchange items for the link (input quantity for the Simple River is 'Inflow').
8. The model user defines the simulation period.
9. The model user runs the simulation.

Note that the runoff for a particular polygon is distributed on the river branches depending on how large a portion of a branch is included in each polygon. This type of boundary condition, where water is added to branches, was not possible in the original Simple River engine. The Simple River engine is (as a result of the migration) extended with this feature, simply because such a boundary condition becomes a possibility when running in combination the OpenMI.

## 4.2.2 Defining exchange items

*Exchange items* are combined information about what can be exchanged and where the exchanged item applies. An input exchange item could define that inflow can be accepted on nodes or river branches. An output exchange item could specify that flow can be provided on branches. The Quantity ID identifies what can be exchanged (e.g. 'Flow') and the ElementSet ID identifies where this quantity applies (e.g. 'Node:1').

The next step is to define input and output exchange items. The exchange items that are required in order to run the use cases are listed in Table 4-1.

**Table 4-1 Required exchange items for use cases 1 and 2**

ElementSet.ID	Type	Quantity.ID	Unit	IsInput	IsOutput	Use case
'Branch:0'	Polyline	Flow	M3/sec	No	Yes	1
'Branch:1'	Polyline	Flow	M3/sec	No	Yes	1
'Branch:2'	Polyline	Flow	M3/sec	No	yes	1
'Node:0'	IDBased	Inflow	M3/sec	Yes	No	1
'Node:1'	IDBased	Inflow	M3/sec	Yes	No	1
'Node:2'	IDBased	Inflow	M3/sec	Yes	No	1
'Node:3'	IDBased	Inflow	M3/sec	Yes	No	1
'Branch:0'	Polyline	Inflow	M3/sec	Yes	No	
'Branch:1'	Polyline	Inflow	M3/sec	yes	No	
'Branch:2'	Polyline	Inflow	M3/sec	yes	No	
'All Branches'	Polyline	Inflow	M3/sec	yes	No	2

Naturally, the exchange items should not be limited to a particular network, but for the purpose of planning the migration it is easier to start out with a specific case and then generalize this case when it comes to the more detailed design.

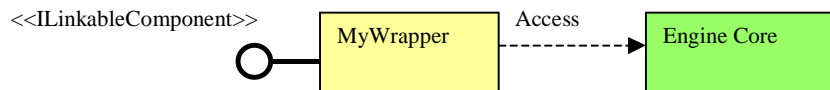
## Chapter 4.3 Wrapping

The OpenMI standard was designed to allow easy migration of existing model engines. The standard is implemented in C# running under the .NET framework. Almost all existing model engines are implemented in other programming languages, such as Fortran, Pascal, C and C++. In order to bridge the gap between the different technologies and to minimize the amount of changes needed to be made to the engine core a *wrapping pattern* will be the most attractive choice in most cases.

This chapter describes the process of wrapping and the generic wrapper that is provided by the OpenMI Software Development Kit.

### 4.3.1 A general wrapping pattern

*Wrapping* basically means that you create a C# class that implements the ILinkableComponent interface. This wrapper will communicate internally with your engine core. The wrapper will appear to the users as a 'black box', which means that all communication will take place through the ILinkableComponent interface (Figure 4-8).



**Figure 4-8 OpenMI wrapping pattern**

One further advantage of using the wrapping pattern is that you can keep the OpenMI-specific implementations separated from your engine core. Typically, the engines will also be used as standalone applications where OpenMI is not used and it is naturally an advantage to be able to use the same engine in different contexts. This means that even in situations where new engines are built the wrapping pattern may still be the best choice.

### 4.3.2 The LinkableEngine

Model engines that are doing timestep-based computations have many things in common. It is therefore possible to develop a generic wrapper that can be used for these engines. This wrapper is called *LinkableEngine* and is located in the `org.OpenMI.Utilities.Wrapper` package. Basically, the *LinkableEngine* provides a default implementation of the `ILinkableComponent` interface. Naturally, the *LinkableEngine* cannot know the specific behaviour of your model engine; this information is obtained through the `IEngine` interface.

The recommended design pattern for model engine migration when using the *LinkableEngine* is shown in Figure 4-9. The design includes the following classes:

- The `MyEngineDLL` class is the compiled core engine code (e.g. Fortran).
- The `MyEngineDLLAccess` class is responsible for translating the Win32Api from `MyEngineDLL` to .NET (C#).
- Calling conventions and exception handling are different for .NET and Fortran. The `MyEngineDotNetAccess` class ensures that these operations follow the .NET conventions.
- The `MyEngineWrapper` class implements the `IEngineAccess` interface, which means that it can be accessed by the *LinkableEngine* class.
- The `MyLinkableEngine` class is responsible for the creation of the `MyEngineWrapper` class and for assigning a reference to this class to a protected field variable in the *LinkableEngine* class, thus enabling this class to access the `MyEngineWrapper` class.

More details of these classes are provided in the following sections.

The OpenMI standard puts a lot of responsibilities on the *LinkableComponents*. The main idea is that when the `GetValues` method is invoked the providing component must be able to deliver the requested values so that these apply to the requested time and the requested location. To be able to do this the *LinkableComponent* may have to interpolate, extrapolate or aggregate both in time and space. These and other things are handled by the *LinkableEngine*.

The *LinkableEngine* class includes the following features:

- *Buffering*: When a model is running as an OpenMI component it may be queried for values that correspond to a time that is before the current time of the model. Most models will only keep values for the current timestep and the previous timestep in memory. It is therefore necessary to store data associated with the OpenMI links in a buffer. The *LinkableEngine* handles the buffering for you.
- *Temporal interpolation and extrapolation*: Most models are only capable of delivering results at times that correspond to their internal timesteps. The *LinkableEngine* class handles all the temporal operations that are required for *LinkableComponents*.
- *Spatial operations*: The *LinkableEngine* provides a range of spatial data operations.
- *Link book-keeping*: The *LinkableEngine* handles book-keeping for links added to your component.

- *Event handling:* The LinkableEngine sends events that enable an event-listener to monitor the progress of the linked system when running.

More details about how the LinkableEngine works is given in Part F: *org.OpenMI.Utilities technical documentation*.

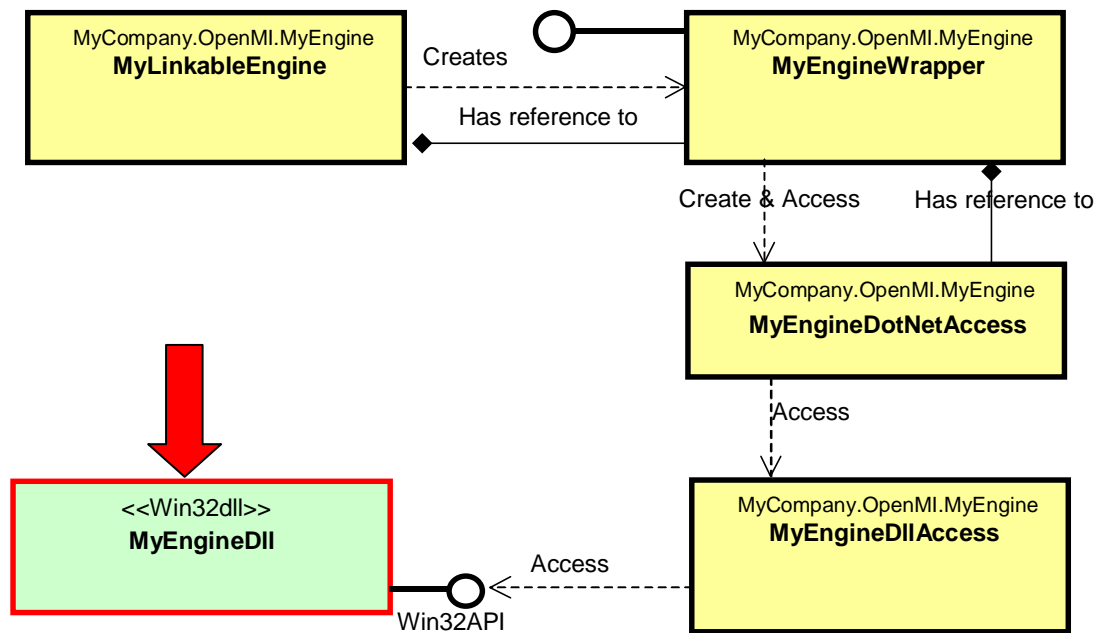
## Chapter 4.4 Migration – step by step

The best strategy when migrating a model is to split the process into a number of steps; at the end of each step you can compile your code and run a small test.

The steps needed for migration are described in this chapter.

### 4.4.1 Step 1: Changing your engine core

The aim of the migration is to develop a class that implements the IEngine interface. As shown in Figure 4-9, the class that implements the IEngine interface is supported by other classes and the engine DLL.



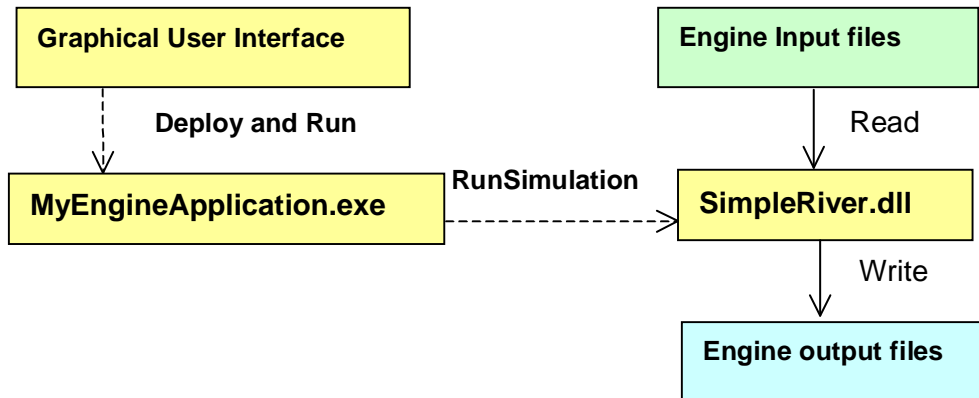
**Figure 4-9 Wrapper classes and engine core DLL**

Model engines are typically compiled into an executable file (EXE). Such executable files are not accessible by other components and as such are not very suitable as a basis for OpenMI components. It is therefore necessary for your engine to be compiled into a dynamic link library file (DLL).

Ideally you should make modifications to your engines so that the same engine can be used both when running as an OpenMI component and when running as a standalone application. Having two versions of the same engine leads to unnecessary maintenance work. Therefore you could make a new application (EXE) that calls a function in the engine core DLL which, in turn, makes your engine perform a full simulation.

Figure 4-10 illustrates the software required to run an engine as a standalone application. The SimpleRiverApplication.EXE file is never used when running in an OpenMI setting.





**Figure 4-10 Running an engine as a standalone application**

The following steps are required in the conversion of the engine core:

1. Change the engine core so that it can be compiled into a DLL.
2. Add a function to the engine core that will run a full simulation:

```
logical function RunSimulation()
```

3. Create an engine application (EXE) that from its main program calls the RunSimulation function in your engine core DLL.
4. Run your engine by deploying the engine application and check that the engine is still producing correct results.

When your engine is running in the OpenMI Software Development Kit it must be able to initialize, perform single timesteps, finalize and be disposed as separate operations. This means that your engine core may need to be reorganized. You can do this in any way you like but one logical approach is to create four functions:

```
logical function Initialize()
(Open files and populate your engine with initial data)
```

```
logical function PerformTimeStep()
(Perform a single timestep)
```

```
logical function Finish()
(Close files)
```

```
logical function Dispose()
(De-allocate memory)
```

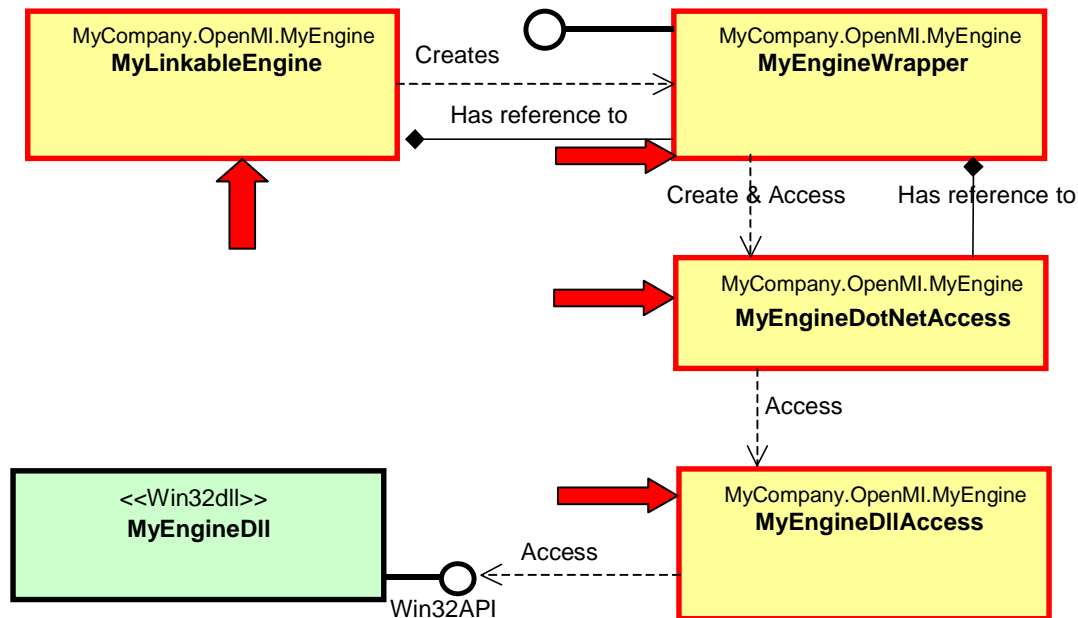
The RunSimulation function should now be changed so that it calls the Initialize function, then repeatedly calls the PerformTimeStep function until the simulation has completed, and finally calls the Finish and Dispose functions.

At this point you should run your application again and check that the engine is still producing the correct results.

You have now completed the restructuring of the engine. The remaining changes that you need to make to the engine will be much smaller. The nature of the changes will be dependent on the particular engine. For now, you can move on to creating the wrapper code.

## 4.4.2 Step 2: Creating the .NET assemblies

The next step is to create the wrapper classes (Figure 4-11). For this stage, make sure that the OpenMI Software Development Kit is installed on your PC.



**Figure 4-11 C# wrapper classes**

Load the .NET development environment. You should create one assembly for your wrapper classes and it is strongly recommended that you also create one assembly for the corresponding test classes.

You should use the following naming conventions for your wrapper assembly:

Assembly name:	MyOrganisation.OpenMI.MyModel
Assembly DLL name:	MyOrganisation.OpenMI.MyModel.DLL
Namespace:	MyOrganisation.OpenMI.MyModel
Class names:	MyModelEngineWrapper MyModelEngineDotNetAccess MyModelEngineDLLAccess MyModelLinkableComponent

Naming conventions for the test assembly:

Assembly name:	MyOrganisation.OpenMITest.MyModel
Assembly DLL name:	MyOrganisation.OpenMITest.MyModel.DLL
Namespace:	MyOrganisation.OpenMI.MyModel

Class names:                   MyModelEngineWrapperTest  
                                  MyModelEngineDotNetAccessTest  
                                  MyModelEngineDLLAccessTest  
                                  MyModelLinkableComponentTest

Now install the NUnit test software (see Chapter 4.6)

To the wrapper assembly, add the following references:

Org.OpenMI.Standard  
Org.OpenMI.Backbone  
Org.OpenMI.Utilities.Wrapper

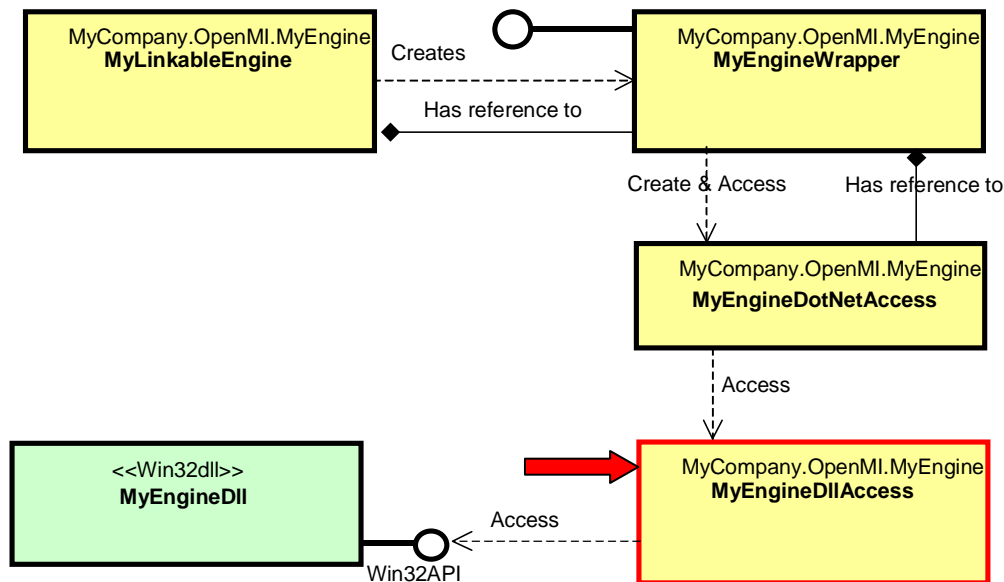
To the test assembly, add the following references:

Org.OpenMI.Standard  
Org.OpenMI.Backbone  
Org.OpenMI.Utilities.Wrapper  
NUnit.framework  
MyOrganisation.OpenMI.MyModel

After creating the assemblies and the classes, you can start working on the first class, MyEngineDLLAccess. Details are given in the next section.

### 4.4.3 Step 3: Accessing the functions in the engine core

The third step is to implement the MyEngineDLLAccess class (Figure 4-12).



**Figure 4-12 MyEngineDLLAccess class**

Because you are using a C# implementation of OpenMI, your engine needs to be accessible from .NET. In the pattern shown above this is handled in two wrappers, MyEngineDLLAccess and MyEngineDotNetAccess. The MyEngineDLLAccess class will make a one-to-one conversion of all exported functions in the engine core code to public .NET methods. The MyEngineDotNetAccess class will change some of the calling conventions.

The specific implementation of the MyEngineDLLAccess class depends on the compiler you are using. Start by implementing export methods for the Initialize, PerformTimeStep, Finish and Dispose functions.

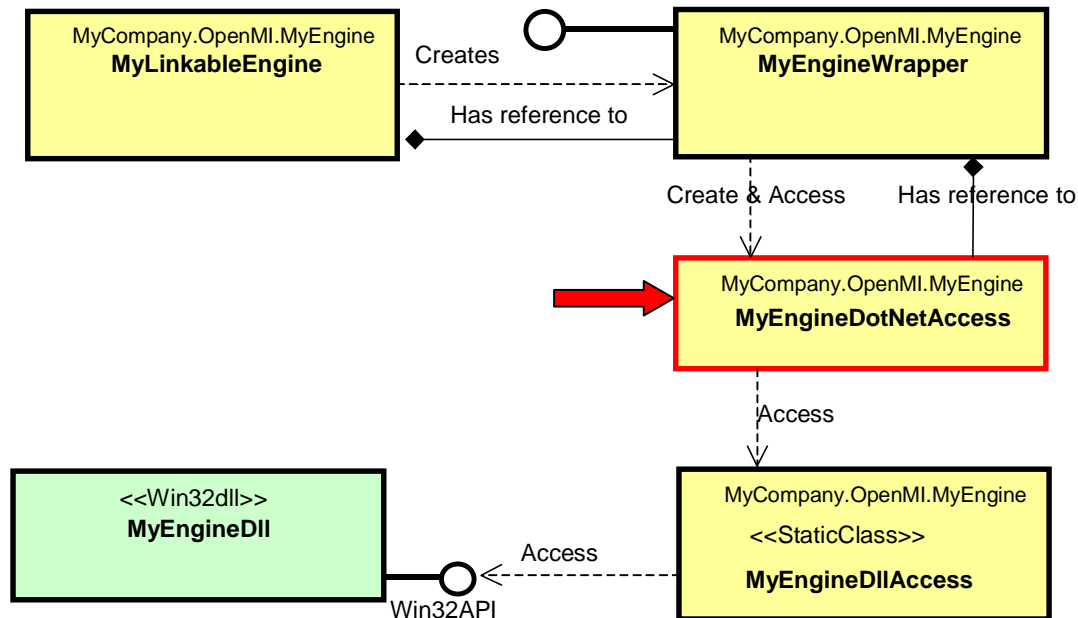
Figure 4-13 shows an example of such an implementation for the Simple River Fortran engine. Note that this implementation corresponds to a particular Fortran compiler; the syntax may vary between compilers.

```
using System;
using System.Run-time.InteropServices;
using System.Text;
namespace MyOrganisation.OpenMI.MyModel
{
    public class MyEngineDLLAccess
    {
        [DllImport(@"C:\MyEngine\bin\MyEngine.DLL",
            EntryPoint = 'INITIALIZE',
            SetLastError=true,
            ExactSpelling = true,
            CallingConvention=CallingConvention.Cdecl)]
        public static extern bool Initialize(string filePath, uint length);
        [DllImport(@"C:\MyEngine\bin\MyEngine.DLL",
            EntryPoint = 'PERFORMTIMESTEP',
            SetLastError=true,
            ExactSpelling = true,
            CallingConvention=CallingConvention.Cdecl)]
        public static extern bool PerformTimeStep();
        [DllImport(@"C:\MyEngine\bin\MyEngine.DLL",
            EntryPoint = 'FINISH',
            SetLastError=true,
            ExactSpelling = true,
            CallingConvention=CallingConvention.Cdecl)]
        public static extern bool Finish();
    }
}
```

**Figure 4-13 Implementing the Simple River Fortran engine**

#### 4.4.4 Step 4: Implementing MyEngineDotNetAccess

The fourth step is to implement the MyEngineDotNetAccess class (Figure 4-14).



**Figure 4-14 MyEngineDotNetAccess class**

The MyEngineDotNetAccess has two purposes: to change the calling conventions to C# conventions and to change error messages into .NET exceptions.

Figure 4-15 shows the Simple River example code for a MyEngineDotNetAccess class that implements the Initialize method, the PerformTimeStep method and the Finish method. In each of these methods the corresponding method in the MyEngineDIIAccess class is called and, if this method returns false, the error message from the engine is queried through the GetMessage method (following which an exception is created and thrown).

Note that the MyEngineDIIAccess class is not instantiated. All the methods in this class are static and can be accessed directly by referencing the class. (Note that to make the example simpler, not all DLL import statements are shown.)

The normal convention for Fortran DLLs is that values are returned through the function (effectively, they are passed by reference); for C# results are passed by value. Therefore there may be a need to state explicitly that C# parameters passed to a Fortran routine are to be passed by reference. In Fortran, arrays normally start from index 1 whereas in C# the convention is to start from zero. These differences can be handled in the MyEngineDotNetAccess class.

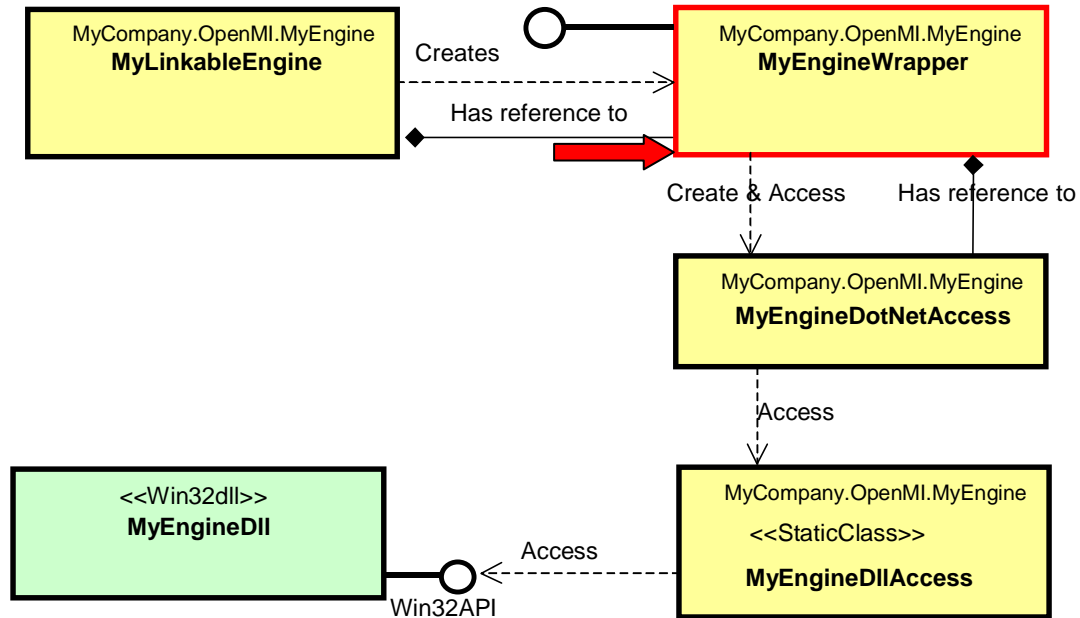
When you have completed the implementation of the methods shown below you can create and implement the corresponding test class and run the unit test.

```
using System;
using System.Text;
namespace MyOrganisation.OpenMI.MyModel
{
    public class MyEngineDotNetAccess
    {
        public void Initialize(string filePath)
        {
            if(!(MyModelDLL.Initialize(filePath, ((uint)
                filePath.Length))))
            {
                CreateAndThrowException();
            }
        }
        public void PerformTimeStep()
        {
            if(!(MyModelDLL.PerformTimeStep()))
            {
                CreateAndThrowException();
            }
        }
        public void Finish()
        {
            if(!(MyModel.Finish()))
            {
                CreateAndThrowException();
            }
        }
        private void CreateAndThrowException()
        {
            int numberOfMessages = 0;
            numberOfMessages = MyModelDLL.GetNumberOfMessages();
            string message = 'Error Message from MyModel `';
            for (int i = 0; i < numberOfMessages; i++)
            {
                int n = i;
                StringBuilder messageFromCore
                    = new StringBuilder(`
                    `);
                MyModelDLL.GetMessage(ref n, messageFromCore,
                    (uint) messageFromCore.Length);
                message += ` `;
                message += messageFromCore.ToString().Trim();
            }
            throw new Exception(message);
        }
    }
}
```

**Figure 4-15 Code for the MyEngineDotNetAccess class**

### 4.4.5 Step 5: Implementing the MyEngineWrapper class

The fifth step is to implement the MyEngineWrapper class (Figure 4-16).



**Figure 4-16 The MyEngineWrapper class**

The MyEngineWrapper class must implement the ILinkableEngine interface (org.OpenMI.Utilities.Wrapper.ILinkableEngine). The easiest way to get started is to make your development environment auto-generate the stub code for this interface.

The first step is to implement the Initialize method and the Finish method.

The MyEngineWrapper has a private field variable (\_myEngine) that holds a reference to the MyEngineDotNetAccess class. The Initialize method will instantiate the MyEngineDotNetAccess object and assign a reference to this object to the \_myEngine variable.

Example code for this is shown in Figure 4-17.



```
using System;
using System.Collections
namespace MyOrganisation.OpenMI.MyModel
{
    public class MyEngineWrapper : org.OpenMI.Utilities.Wrapper.IEngine
    {
        private MyEngineDotNetAccess _myEngine;

        public void Initialize(Hashtable properties)
        {
            _myEngine = new MyEngineDotNetAccess();
            _myEngine.Initialize((string)properties['FilePath']);
        }
        public void Finish()
        {
            _simpleRiverEngine.Finish();
        }
    }
}
```

**Figure 4-17 Example code for the wrapper classes**

#### 4.4.6 Step 6: Implementing MyModelLinkableComponent

The sixth step is to implement the MyModelLinkableComponent class (Figure 4-18).

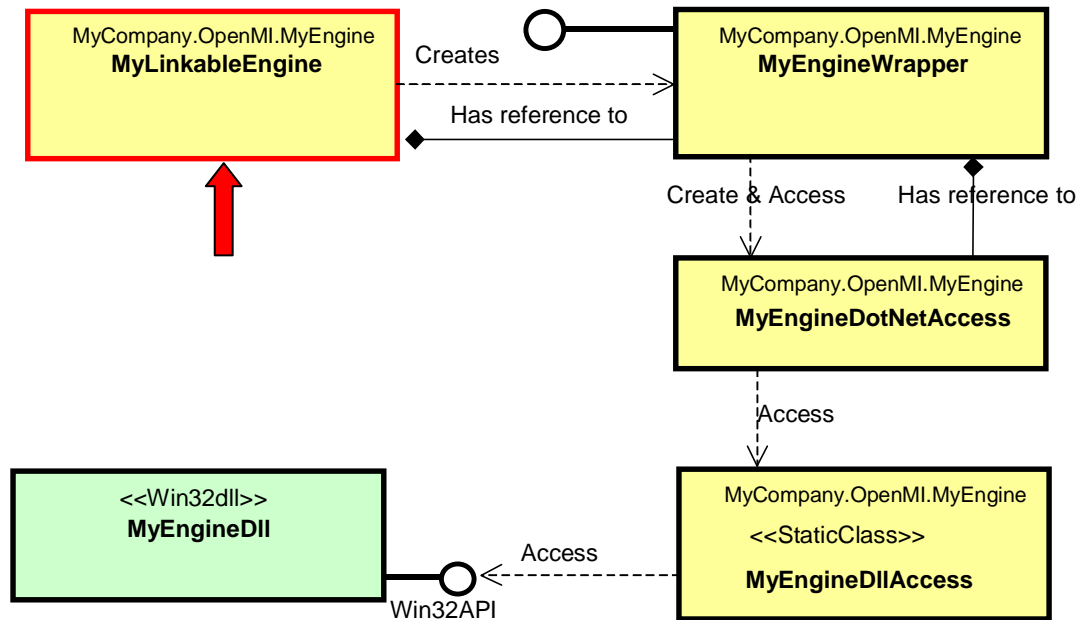


Figure 4-18 MyLinkableEngine class

The MyModelLinkableComponent is the OpenMI-compliant linkable component that is going to be accessed by other models. Implementation of this class is very simple. The example shown below (Figure 4-19) is the complete implementation for the Simple River model.

```

using System;
namespace MyOrganisation.OpenMI.MyModel
{
    public class MyModelOpenMIComponent :
        org.OpenMI.Utilities.Wrapper.LinkableEngine
    {
        protected override void SetEngineApiAccess()
        {
            _engineApiAccess = new MyEngineWrapper();
        }
    }
}
  
```

Figure 4-19 Complete implementation for the Simple River model

This class inherits from the LinkableEngine class. The class creates the EngineWrapper and assigns it to the protected field variable `_engineApiAccess`.

### 4.4.7 Step 7: Implementation of the remaining IEngine methods

The basic structure of your engine and wrapper code is now in place. The task is now to go through the MyEngineWrapper class and complete the implementation of the methods that are currently auto-generated stub code. Some of these methods can be completed only by changing the code in the MyEngineWrapper; for others, changes also need to be made to the other classes and the engine core (MyEngineDLL). After completion of each method you should update the test classes and run the unit test.

For each method you must decide if the bulk of implementation should be located in the MyEngineWrapper class or in the engine core. There is no general answer to this question. Placing the bulk of implementation in the engine core could be advantageous from the perspective of maintenance because you have most things located in one place. On the other hand, you may want to keep the engine core as free as possible of OpenMI-related code and therefore put the bulk of the implementation into the MyEngineWrapper class. Finally, there may also be considerations about the preferred programming language; the engine core may be programmed in Fortran, C or Pascal, whereas the MyEngineWrapper class is programmed in C#.

Implementation of the IEngine interface depends on the engine core, so it is not possible to give a general explanation of how each individual method should be implemented. However, the next chapter gives a description of how the IEngine interface has been implemented for the Simple River model.

The IEngine interface is shown in Figure 4-20.

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

**Figure 4-20 The IEngine interface**



## Chapter 4.5 Migration of the Simple River

The previous chapter described the steps involved in migrating a model to the OpenMI. This chapter shows how the migrated code is developed for the Simple River example.

### 4.5.1 The Simple River wrapper

The Simple River model uses the migration pattern shown in Figure 4-9. Figure 4-21 gives a detailed explanation of how the Simple River wrapper works in terms of the wrapper classes.

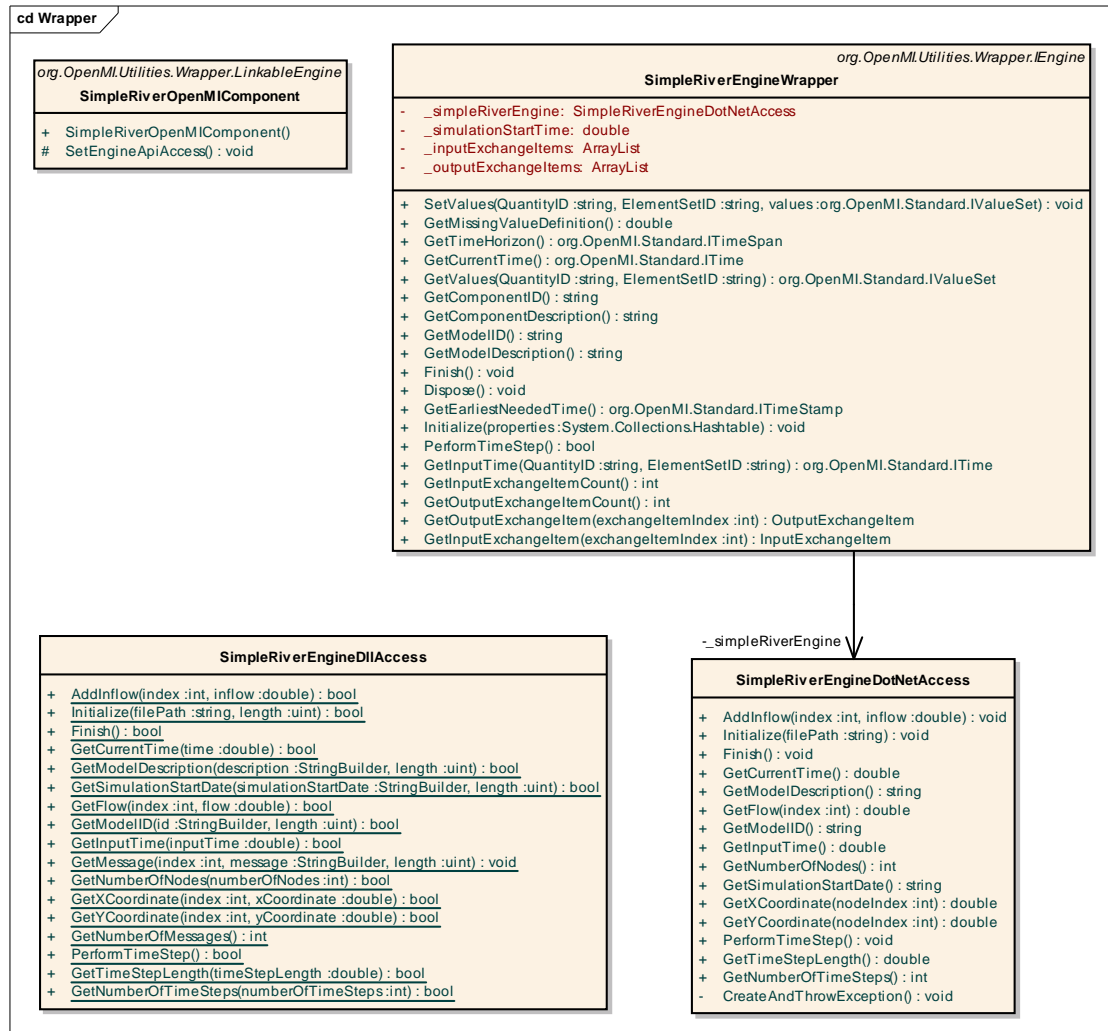


Figure 4-21 Simple River wrapper classes

## 4.5.2 Implementation of the Initialize method

The SimpleRiverEngineWrapper has two private field variables:

```
ArrayList _inputExchangeItems;
ArrayList _outputExchangeItems;
```

The \_inputExchangeItems is a list of org.OpenMI.Backbone.InputExchangeItem objects and the \_outputExchangeItems is a list of org.OpenMI.Backbone.OutputExchangeItem objects. These arraylists are populated in the Initialize method.

The Simple River wrapper must fulfil the requirements defined by the use cases described in section 4.2.1. This means that the input and output exchange items are based on the list of items in Table 4-1.

The source code for implementation of the Initialize method in the SimpleRiverEngineWrapper is shown in Figure 4-22.

```
public void Initialize(System.Collections.Hashtable properties)
{
    _inputExchangeItems = new ArrayList(); //ArrayList of
        org.OpenMI.Backbone.InputExchangeItem objects
    _outputExchangeItems = new ArrayList(); //ArrayList of
        org.OpenMI.Backbone.OutputExchangeItem objects

    // -- Create and initialize the engine --
    _simpleRiverEngine = new SimpleRiverEngineDotNetAccess();
    _simpleRiverEngine.Initialize((string)properties['FilePath']);

    // -- Simulation start time -
    // The start time is obtained from the engine core as a string. This string is
    // passed and converted to a System.DateTime. Then the
    // org.OpenMI.DevelopmentSupport.CalendarConverter class is used to convert
    // this time into the ModifiedJulianDay (this is the OpenMI standard time)
    char [] delimiter = new char[]{'-', ' ', ':'};
    string[] strings = _simpleRiverEngine.GetSimulationStartDate().Split(delimiter);
    int StartYear = Convert.ToInt32(strings[0]);
    int StartMonth = Convert.ToInt32(strings[1]);
    int StartDay = Convert.ToInt32(strings[2]);
    int StartHour = Convert.ToInt32(strings[3]);
    int StartMinute = Convert.ToInt32(strings[4]);
    int StartSecond = Convert.ToInt32(strings[5]);
    DateTime startDate = new DateTime(StartYear, StartMonth, StartDay, StartHour,
        StartMinute, StartSecond);
    _simulationStartTime = org.OpenMI.DevelopmentSupport.CalendarConverter.
        Gregorian2ModifiedJulian(startDate);
    // -- Build exchange items ---
    Dimension flowDimension = new Dimension();
    Unit flowUnit = new Unit('m3/sec', 1, 0, 'm3/sec'); //The Simple River only uses
        // quantities with the unit m3/sec.

    Quantity flowQuantity = new Quantity(flowUnit, 'description', 'Flow',
        org.OpenMI.Standard.ValueType.Scalar, flowDimension);
    Quantity inFlowQuantity = new Quantity(flowUnit, 'description', 'InFlow',
        org.OpenMI.Standard.ValueType.Scalar, flowDimension);
}
```

```

int numberOfNodes = _simpleRiverEngine.GetNumberOfNodes();
for (int i = 0; i < numberOfNodes - 1; i++) //For each branch
{
    OutputExchangeItem flowFromBranch = new OutputExchangeItem();
    InputExchangeItem inFlowToBranch = new InputExchangeItem();
    // One ElementSet is created for each branch. The ElementID's are
    // Branch:<Branch number>. E.g. 'Branch:3'
    ElementSet branch = new ElementSet('description', 'Branch:' +
        i.ToString(), ElementType.XYPolyLine, new SpatialReference('ref'));
    branch.AddElement(new Element('Branch:' + i.ToString()));
    branch.Elements[0].AddVertex(new Vertex(_simpleRiverEngine.
        GetXCo-ordinate(i), _simpleRiverEngine.GetYCo-ordinate(i), 0));
    branch.Elements[0].AddVertex(new Vertex(_simpleRiverEngine.
        GetXCo-ordinate(i+1), _simpleRiverEngine.GetYCo-ordinate(i+1), 0));

    flowFromBranch.ElementSet = branch;
    flowFromBranch.Quantity = flowQuantity;
    inFlowToBranch.ElementSet = branch;
    inFlowToBranch.Quantity = inFlowQuantity;

    _outputExchangeItems.Add(flowFromBranch);
    _inputExchangeItems.Add(inFlowToBranch);
}
for (int i = 0; i < numberOfNodes; i++) //For all nodes
{
    InputExchangeItem inflowToNode = new InputExchangeItem();
    // Each node is a ID-based ElementSet. The ElementSet ID are
    // Node:<node number>. E.g. 'Node:3'
    ElementSet node = new ElementSet('description', 'Node:' +
        i.ToString(), ElementType.IDBased, new SpatialReference('ref'));
    node.AddElement(new Element('Node:' + i.ToString()));
    inflowToNode.Quantity = inFlowQuantity;
    inflowToNode.ElementSet = node;
    _inputExchangeItems.Add(inflowToNode);
}
ElementSet Branches = new ElementSet('description', 'AllBranches',
    ElementType.XYPolyLine, new SpatialReference('ref'));
for (int i = 0; i < numberOfNodes - 1; i++) //Create an InputExchangeItem that
// has all branches in one ElementSet
{
    Element branch = new Element('Branch: ' + i.ToString());
    branch.AddVertex(new Vertex(_simpleRiverEngine.
        GetXCo-ordinate(i), _simpleRiverEngine.GetYCo-ordinate(i), 0));
    branch.AddVertex(new Vertex(_simpleRiverEngine.
        GetXCo-ordinate(i+1), _simpleRiverEngine.GetYCo-ordinate(i+1), 0));
    Branches.AddElement(branch);
}
InputExchangeItem inFlowToBranches = new InputExchangeItem();
inFlowToBranches.ElementSet = Branches;
inFlowToBranches.Quantity = inFlowQuantity;
_inputExchangeItems.Add(inFlowToBranches);
}

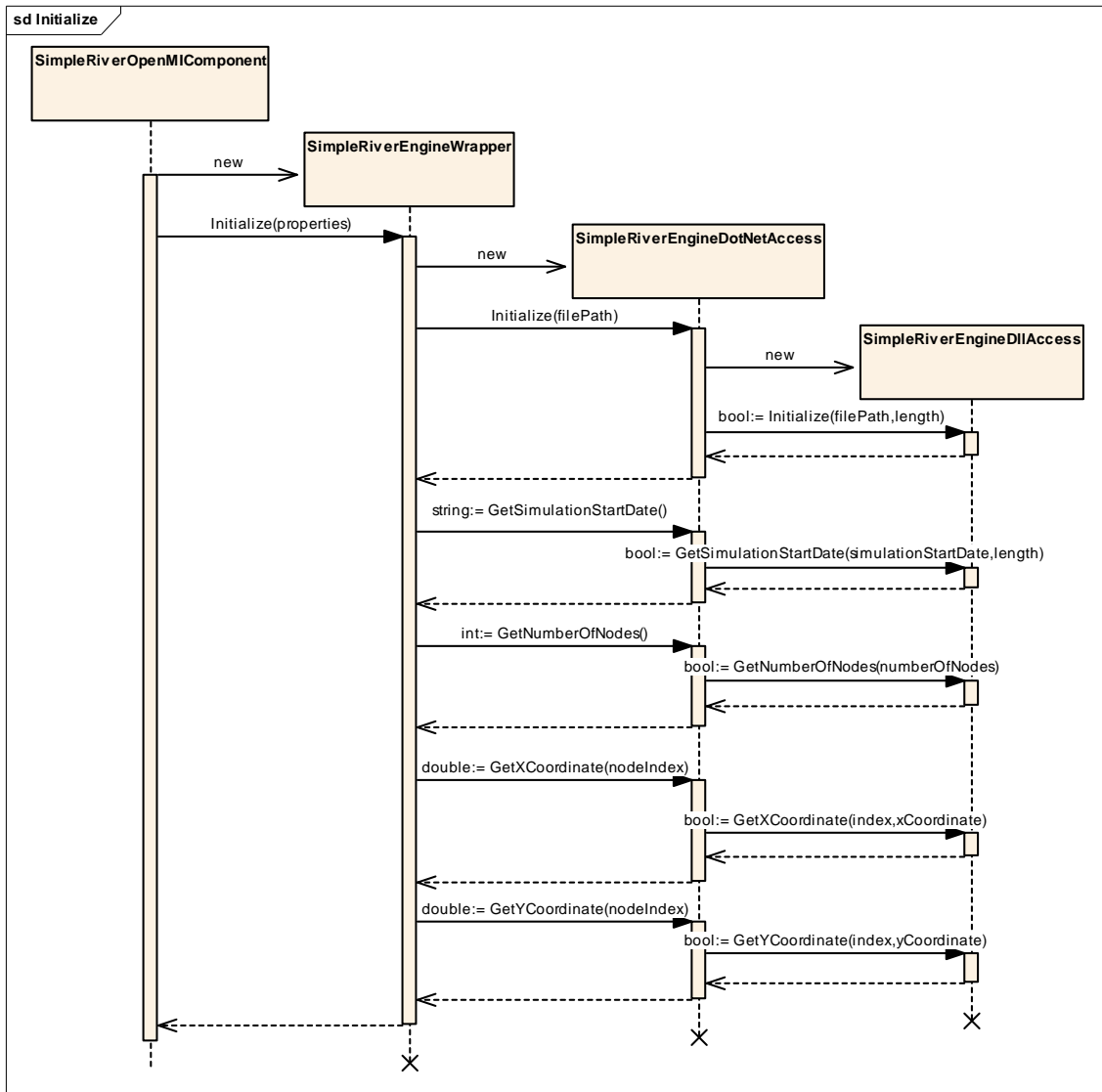
```

**Figure 4-22 Implementation of the Initialize method**

As you can see from the implementation of the Initialize method, some methods need to be implemented in the MyEngineDotNetAccess class, the MyEngineDLLAccess class and the engine core.



The sequence diagram in Figure 4-23 illustrates the communication with the other wrapper classes when the Initialize method is invoked. The EngineDLL is not included in the diagram since there is a one-to-one communication between the EngineDLL and the EngineDLLAccess classes. In other words, each time a method is called in the EngineDLLAccess the corresponding function is called in the EngineDLL.

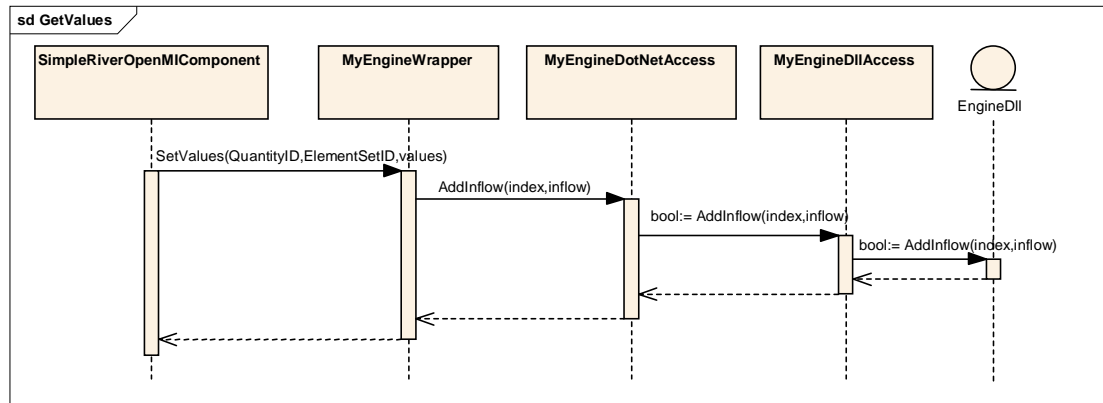


**Figure 4-23 Calling sequence for the initialize method**

Note that no DataOperations are added to the OutputExchangeItems. The LinkableEngine class will complete the OutputExchangeItems for you by adding spatial and temporal data operations to your OutputExchangeItems. You can still add your own data operations as well.

### 4.5.3 Implementation of the SetValue method

The calling sequence for the SetValue method is shown in Figure 4-24.



**Figure 4-24** Calling sequence for the SetValue method

The EngineWrapper class decides what has to be done, based on the QuantityID and the ElementSetID. In the Simple River engine core there is only one possible variable that can act as input, which is the storage of water in the nodes. For the Simple River model, inflow is interpreted as additional inflow, which means that the inflow already received from other sources (the boundary inflow) is not overwritten. The inflow is added to the current storage in the nodes. The ElementSetID is parsed and the node number to which the water is going is determined.

If the inflow is going to the branches, the water is added to the downstream node for each branch. If the inflow is going to the nodes, the water is simply added to the storage of the node. Understanding the role of the QuantityID and the ElementSetID is very important when you are migrating your model. For each ExchangeItem you define your ElementSetID and QuantityID. These IDs will be included in the link when a user is configuring a system with your model. When the system is running, the same ElementSetID and QuantityID will get back to the ModelComponent when the GetValues method is invoked. Your component will then use this information to navigate to the correct variables in the engine.

You can use any convention for naming these IDs. In the Simple River ElementSetID, the convention used was Branch:<branch number> (e.g. 'Branch:3' ) or 'AllBranches', and the Quantity IDs were 'Flow' and 'InFlow'.

## 4.5.4 Implementing the GetValues method

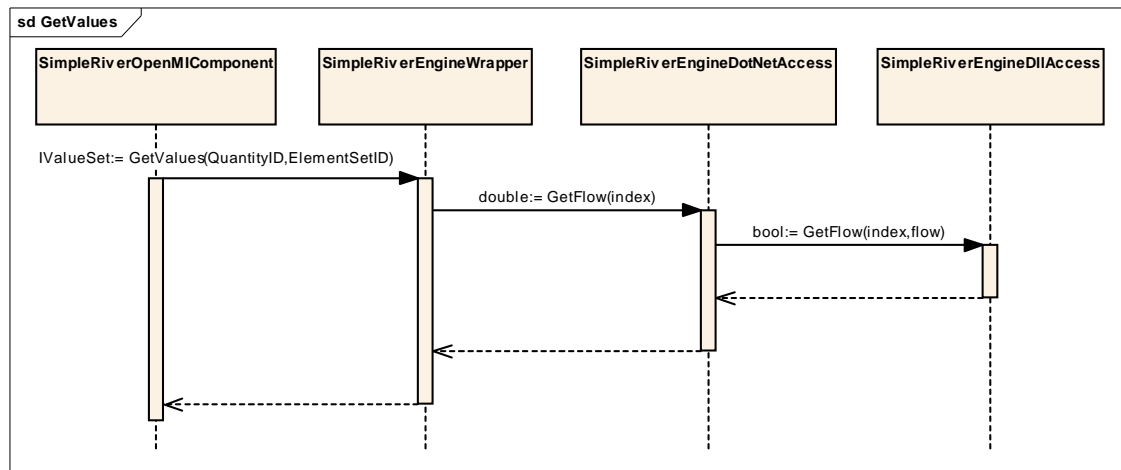
The source code for the IEngine GetValues implementation is shown in Figure 4-25.

```
public org.OpenMI.Standard.IValueSet GetValues(string QuantityID, string ElementSetID)
{
    double[] returnValues;
    Char[] separator = new char[]{' ':' '};
    if (QuantityID == 'Flow')
    {
        int index = Convert.ToInt32((ElementSetID.Split(separator))[1]);
        returnValues = new double[1];
        returnValues[0] = _simpleRiverEngine.GetFlow(index);
    }
    else
    {
        throw new Exception('Illegal QuantityID in GetValues method in
            SimpleRiverEngine');
    }
}
```

**Figure 4-25 Implementation of the GetValues method**

The branch number is extracted from the ElementSetID and used as an index in the GetValues call to the SimpleRiverDotNetAccess class.

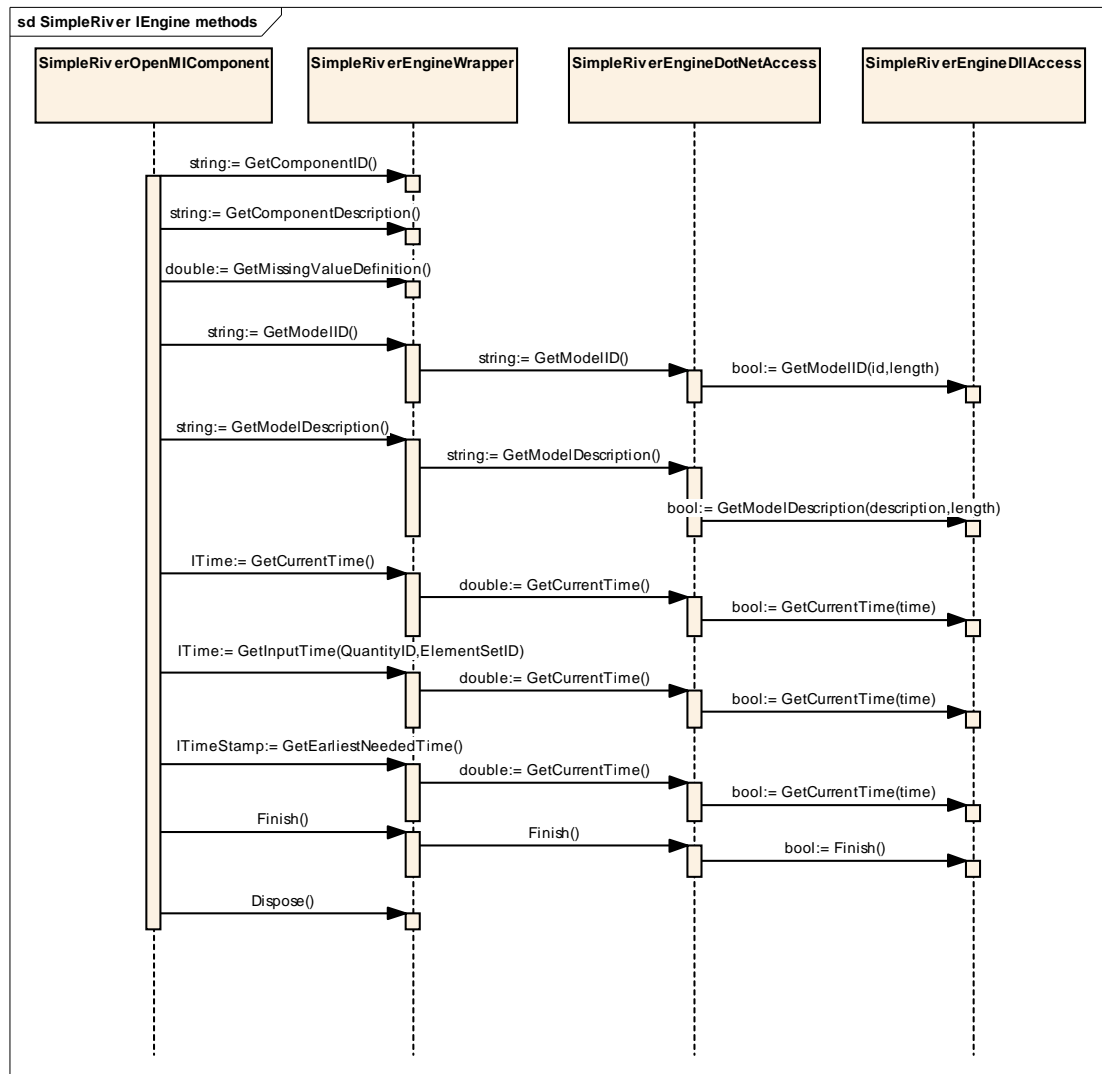
The calling sequence for the GetValues method is shown in Figure 4-26.



**Figure 4-26 Calling sequence for the GetValues method**

### 4.5.5 Implementation of the remaining methods

Implementation of the remaining methods in the IEngine interface is not complicated. On the sequence diagram in Figure 4-27 you can see how each method is accessing the other engine wrapper classes.



**Figure 4-27 Calling sequence for Simple River**

The calling sequence for methods not shown in Figure 4-27 is given in Figure 4-23, Figure 4-24 and Figure 4-26. Note that for some of the methods the full implementation is done in the SimpleRiverEngineWrapper class. The methods GetCurrentTime, GetInputTime and GetEarliestNeededTime are all invoking the GetCurrentTime method in the SimpleRiverDotNetAccess class. The returned time is the engine local time. This time is converted to the ModifiedJulianTime in the SimpleRiverEngineWrapper (Figure 4-28).

```
public org.OpenMI.Standard.ITime GetCurrentTime()
{
    double time = _simulationStartTime + _simpleRiverEngine.GetCurrentTime() /
((double)(24*3600));
    return new org.OpenMI.Backbone.TimeStamp(time);
}
```

**Figure 4-28 Conversion of time to Modified Julian time**



## Chapter 4.6 Testing the component

It is important to test the component to check that it is working correctly. Traditionally, the procedure has been to complete implementation and then run the engine to see if it produces the correct results. However, in recent years new methodologies have been developed for testing. The dominant testing method for object oriented programs is *unit testing*. Unit testing is done in parallel with the implementation. This means that you will be able to find errors earlier and thus save time on debugging your code later.

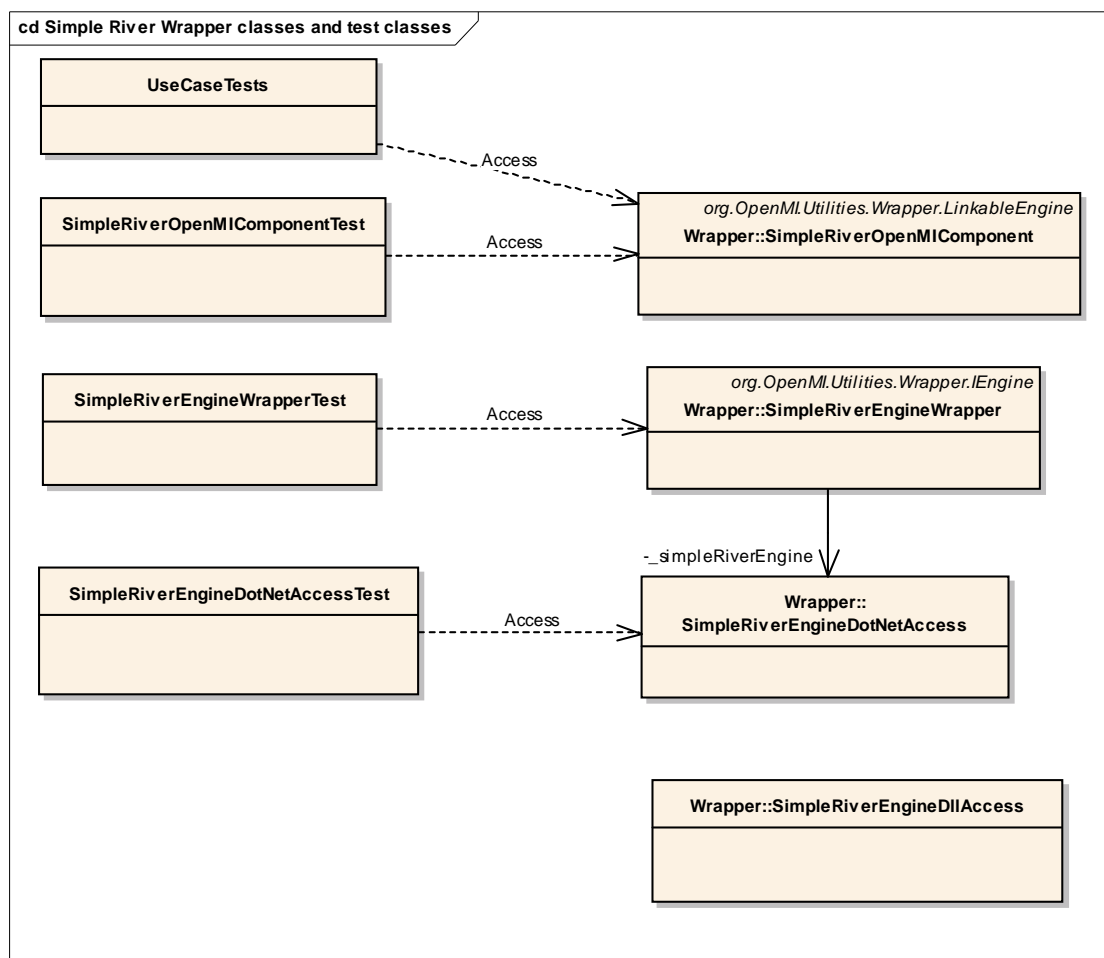
This chapter discusses the testing of migrated components.

### 4.6.1 Unit testing

The testing procedure described here assumes you are using the NUnit test tool. You can download the NUnit user interface and libraries from <http://www.NUnit.org>. This web page also gives more information about NUnit. Basically, you create a test class for each of the wrapper classes; in the test classes you implement a test method for each public method in the class.

This chapter focuses on OpenMI-specific testing. The NUnit home page gives detailed information about Unit testing.

Section 4.4.2 describes how to create test assemblies. The test classes used for the Simple River example are shown in Figure 4-29.



**Figure 4-29 Wrapper and test classes for the Simple River model**

There is a one-to-one relation between the wrapper classes and the test classes, with two exceptions. There is no test class for the SimpleRiverEngineDIIAccess class and there is one additional class called UseCaseTests. The test class for the SimpleRiverEngineDIIAccess class was left out because every method in the SimpleRiverEngineDotNetAccess class will invoke the corresponding method in the SimpleRiverEngineDIIAccess class; therefore testing all methods in the SimpleRiverEngineDotNetAccess class will be sufficient.



The main idea of unit testing is to create very simple code that will test each method in the classes. However, it can also be useful to make some more advanced tests that are actually running full simulations. This is done in the `UseCaseTests` class. This class ensures that the use cases described in section 4.2.1 run without errors.

As described earlier, the model is migrated by implementing the `IEngine` interface. For every `IEngine` method in the `MyEngineWrapper` class you decide which method needs to be implemented in the remaining wrapper classes. You will implement the methods or functions that are needed in the engine core and then implement the corresponding methods in the `MyEngineDLLAccess` class and `MyEngineDotNetAccess` class. Each time you have completed the implementation of a method you can create a test method in the `MyEngineDotNetAccessTest` class and run the unit test. You can then move on to implement the method in `MyEngineAccess` and the associated test methods.

Figure 4-30 contains sample test code for the `GetModelID` method implementation in the Simple River model. Figure 4-31 shows the NUnit interface.

```
using System;
using org.OpenMI.Examples.ModelComponents.SimpleRiver.Wrapper;
using NUnit.Framework;
namespace org.OpenMITest.Examples.ModelComponents.SimpleRiver.Wrapper
{
    [TestFixture]
    public class SimpleRiverEngineDotNetAccessTest
    {
        [Test]
        public void GetModelID()
        {
            SimpleRiverEngineDotNetAccess _simpleRiverEngineDotNetAccess;
            String _filePath;
            _simpleRiverEngineDotNetAccess = new SimpleRiverEngineDotNetAccess();
            _filePath = 'C:\\SimpleRiver\\UnitTest\\Data\\Rhine';
            _simpleRiverEngineDotNetAccess.Initialize(_filePath);
            Assert.AreEqual('The river Rhine',
                _simpleRiverEngineDotNetAccess.GetModelID());
            _simpleRiverEngineDotNetAccess.Finish();
        }
    }
}
```

**Figure 4-30 Test code for the `GetModelID` method implementation**

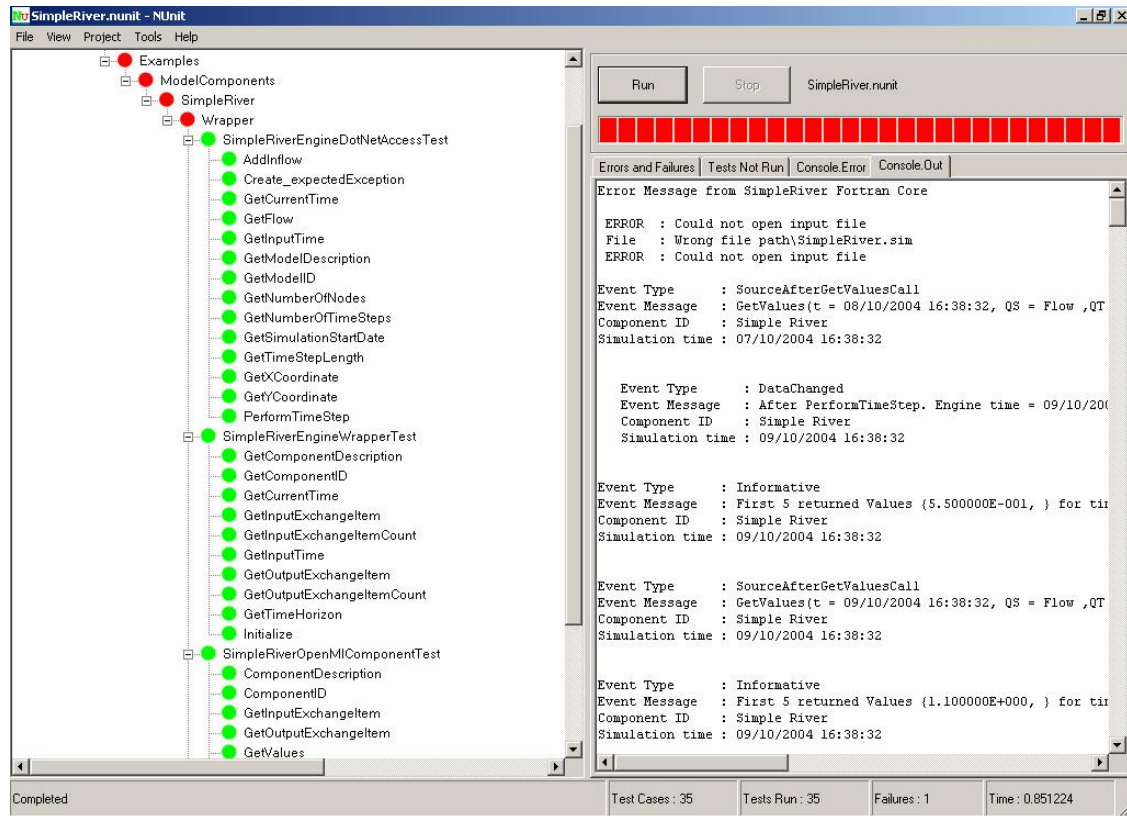


Figure 4-31 NUnit User interface with Simple River wrapper classes loaded

## Chapter 4.7 Implementing IManageState

Implementation of the IManageState interface is not required in order to claim OpenMI compliance. However, if you want to use your model in configurations where iterations are needed or you want to use calibration or optimization controllers, the implementation of the IManageState interface is required. Normally, you should put the bulk of the implementation into the engine core and save the data required in order to restore a state in memory.

### 4.7.1 The IManageState interface

Implementation of the IManageState interface is shown in Figure 4-32.

The LinkableEngine class already implements the required methods for the IManageState interface but it is not specified in this class that it implements the IManageState interface.

To implement the IManageState interface when using the LinkableEngine, the procedure is as follows:

1. In MyLinkableEngine, specify that it implements the IManageState interface.
2. In MyEngineWrapper, specify that it implements the IManageState interface.
3. Implement the IManageState methods in all wrapper classes. The implementation will typically be very simple code that redirects the call to the next wrapper class and finally to the engine core, where the bulk of the implementation is located.

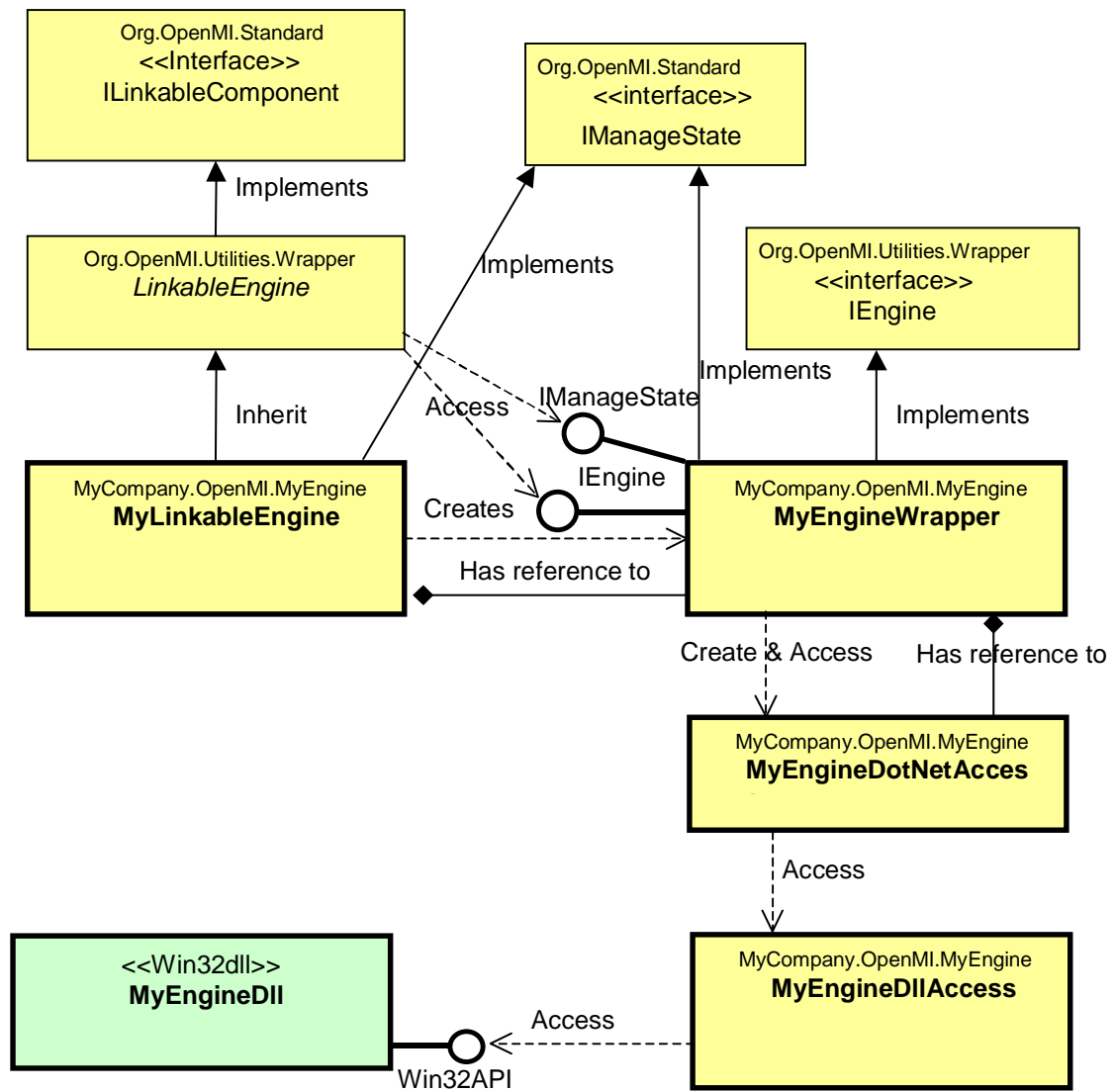


Figure 4-32 IManageState implementation



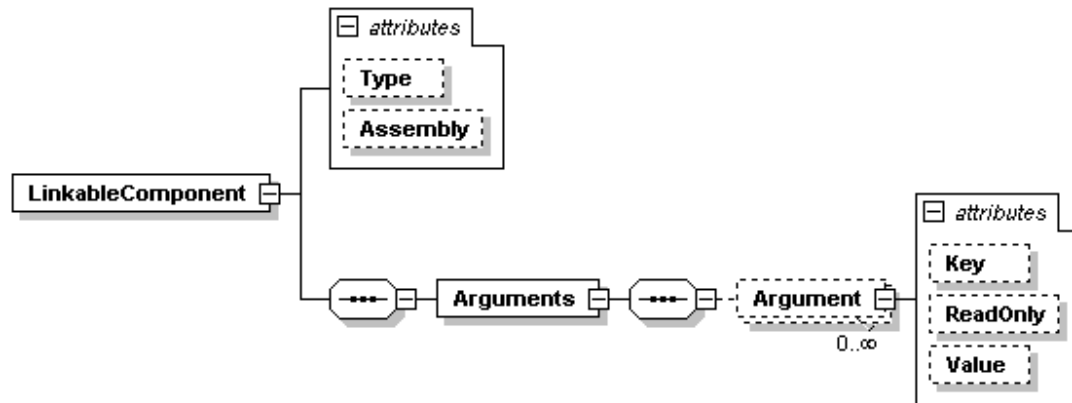
## Chapter 4.8 The OMI file

The OMI file defines the entry point to a LinkableComponent. It contains information on the software unit to instantiate and the arguments to provide at initialization. This file makes it possible for a user interface to deploy your model.

This chapter describes the OMI file.

## 4.8.1 Structure of the OMI file

The structure of the OMI file is defined in `org.OpenMI.Standared.LinkableComponent.XSD`. Figure 4-33 provides a visual representation of the schema definition; Figure 4-34 provides an example of an OMI file.



**Figure 4-33** Visual representation of the LinkableComponent XML schema definition

```
<?XML version='1.0'?>
<LinkableComponent Type='org.OpenMI.Examples.MC.SimpleRain' Assembly=
  'org.OpenMI.Examples.MC, Version=1.4.0.0, Culture=neutral, PublicKeyToken=
  8384b9b46466c568' XMLns='http://www.openmi.org/LinkableComponent.xsd'>
  <Arguments>
    <Argument Key='Data' ReadOnly='true'
      Value='c:\OpenMI\Examples\Data\SimpleRain.txt' />
  </Arguments>
</LinkableComponent>
```

**Figure 4-34** An example OMI file of the LinkableComponent XML schema definition



## Chapter 4.9 Design patterns for model migration

A wide variety of software packages have been migrated to the OpenMI so far. This chapter outlines the processes involved in the migration in various cases.

### 4.9.1 Design patterns for ISIS

ISIS is a modelling package for open channel systems, including loops, branches, floodplain conveyance and storage. The software incorporates standard equations and modelling techniques for structures including weirs, sluices, bridges, culverts, pumps, syphons, orifices and outfalls; it also provides logical control for moving structures. Systems can be simulated either in full unsteady mode or as an advanced steady-state simulation.

The OpenMI wrapper for ISIS is directly derived from the LinkableComponent and does not use any of the utilities. This means that the ISIS wrapper itself handles the link handling, interpolation, buffering and spatial mapping. The reason for this is that the first version of the ISIS wrapper was written early in the development process of the OpenMI (v.1.0) before any of the utilities had been developed. When designing the wrapper, it was decided to keep the changes to the ISIS Fortran code to a minimum and to write most of the wrapper functionality in C#. The ISIS design pattern is shown in Figure 4-35.

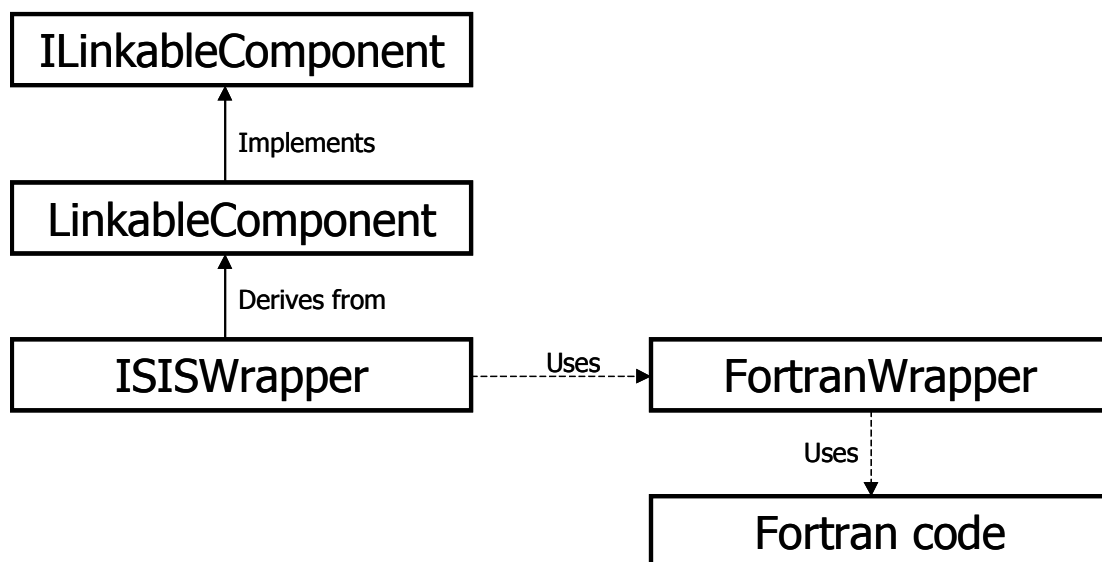


Figure 4-35 The ISIS design pattern

Note that the FortranWrapper is not strictly necessary and the ISISWrapper could have talked to the Fortran code directly. However, this would mean that all code would have to run in one process. Like many Fortran programs, ISIS has a large number of global (common) variables, which means that only one instance of the Fortran code can run at any time. It is not possible to link two ISIS models and keep the Fortran code in the same process. To solve this problem, the FortranWrapper can run in a different process from the ISISWrapper and it is now possible to link many ISIS models together. The communication between ISISWrapper and FortranWrapper is done using .NET remoting; however, any inter-process communication mechanism could have been used.

## 4.9.2 Design patterns for InfoWorks RS

InfoWorks RS is a one-dimensional river modelling software package. Combining the ISIS flow simulation engine, geographical analysis and a relational database within a single environment, InfoWorks RS integrates survey and time-series data with detailed and accurate modelling. InfoWorks RS includes full solution modelling of open channels, floodplains, embankments and hydraulic structures. Rainfall-runoff simulation is available using both event-based and conceptual hydrological methods. The underlying data can be accessed from any graphical or geographical view.

Running an RS model via OpenMI is essentially a three-stage process:

1. Retrieve input and output quantities from RS.
2. Run the RS engine for the model, setting and getting the required quantities.
3. Import the results back into RS.

RS C++ code had to be modified to expose the functions needed for the first and third steps. These functions were then called via the RSLinkableRunEngine, which is derived from LinkableRunEngine.

RS engine Fortran code had to be modified so that the engine ran via a set of timesteps, with an initialize step and a finalize step. A set of functions was written so that these steps were exposed and could be called from a DLL. RSRunEngine implements the IRunEngine interface, which was used to wrap the RS engine and make these calls.

RSLinkableRunEngine creates the RSRunEngine object by .NET remoting so that RS runs in its own process. This is necessary so that global variables in the RS engine do not conflict with each other when two RS models are linked and run together. The InfoWorks RS design pattern is shown in Figure 4-36.

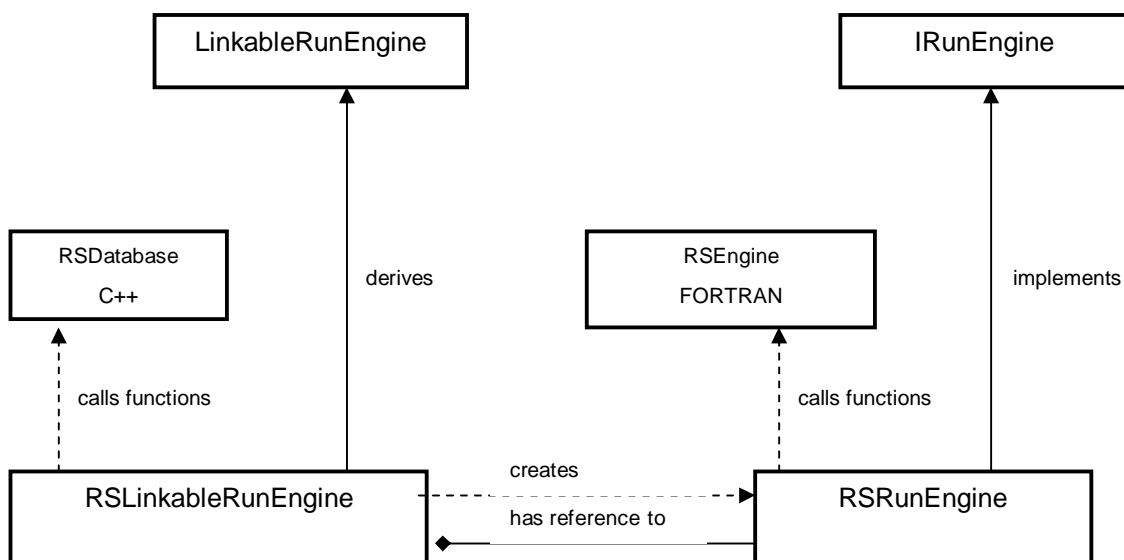


Figure 4-36 The InfoWorks RS design pattern

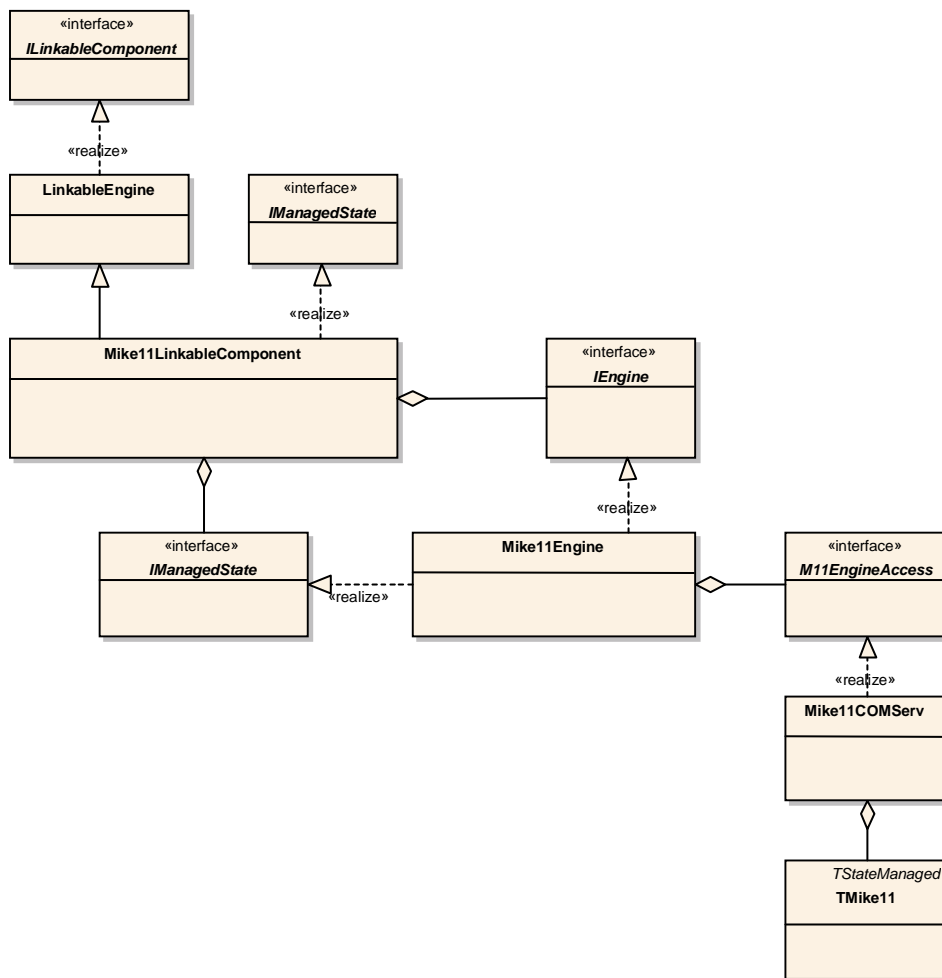
### 4.9.3 Design patterns for Mike11

Mike11 is a modelling system that covers a number of hydraulic areas that are relevant for river modelling and management. The major components of Mike11 are:

- 1D hydrodynamic river modelling in river networks
- 1D advection dispersion modelling in river networks
- 1D water quality modelling in river networks
- 1D sediment transport modelling and morphological modelling in river networks
- Eight built-in rainfall-runoff models for catchment modelling
- A large suite of structures ranging from simple culverts and weirs to advanced control structures and dam structures tailored for dam break modelling

The engine code for Mike11 is written in Delphi and is object oriented.

Migration of the Mike11 components has been made using the wrapper pattern included in `org.OpenMI.Utilities Wrapper`, including both the `org.OpenMI.Utilities.Buffer` and `org.OpenMI.Utilities.Spatial` packages. The pattern used is shown in Figure 4-37.



**Figure 4-37 The Mike11 design pattern**

The migration is made by wrapping Mike11 into a COM object with an interface (M11EngineAccess) very similar to the IEngine interface. The COM interface is again made as a wrapper around the main class of Mike11 itself. The COM interface, the COM wrapper and the main class of Mike11 are kept within Delphi. There is no Mike11-specific code in the logic for everything above the TMike11 class. Therefore the migration has been accomplished without transferring the Mike11 logic out of the engine core.

Table 4-2 and Table 4-3 contain the quantities and element sets that are available.

**Table 4-2 Exchangeable quantities and element sets**

Quantity	ElementSets	Role
Discharge	All Q-points in set-up Q-points in specific branch Single Q-point Discharge boundaries	Input and output
Water Level	All H-points in set-up H-points in specific branch Single H-point Water level boundaries	Input and output
Resistance No.	All H-points in set-up H-points in specific branch Single H-point	Input and output
Groundwater Base Flow	All H-points in set-up H-points in specific branch Single H-point	Output
Velocity		
Surface Slope		
Flow Area		
Flow Width		
Radius		
Conveyance		
Froude No		
Water Level Slope		
Energy Level		
Energy Level Slope		
Bed Shear Stress		
Volume		
Lateral Inflow	All H-points in set-up H-points in specific branch Single H-point Point source boundaries	Input
Groundwater level	All H-points in set-up H-points in specific branch Single H-point	Input
Time-series specified control structure settings	Single Q-point	

**Table 4-3 Quantities and element sets**

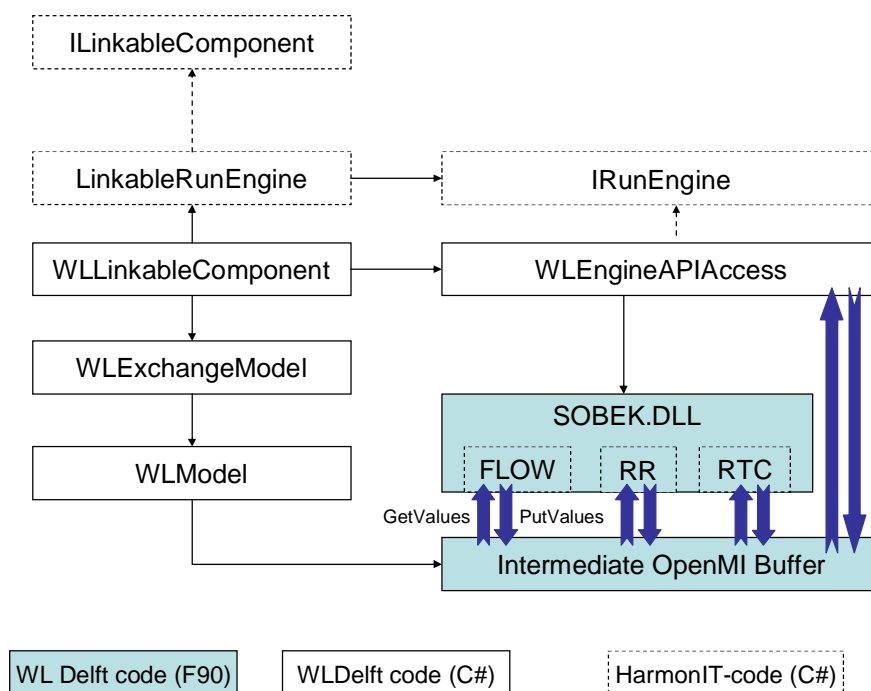
Quantity	ElementSets	Role
Run off	ID for each catchment	Output
Net rainfall		

### 4.9.4 Design patterns for SOBEK

At the heart of the SOBEK modelling system is a powerful hydraulic engine for combined flow computation applicable to sewers, open channels and overland flow. This is supported by engines for real-time system control, rainfall-runoff processes, water quality processes, sediment transport and morphology. Since all modelling engines of WL Delft Hydraulics are developed in Fortran90, a solution has been developed that can easily be applied to all engines.

The solution chosen is an intermediate buffer, implemented in F90/C, where model engines can get their data from external sources and place their data for exchange. The buffer contains the logic to sort out which data needs to be kept until used. Computational engines do not need to be concerned with this data issue or any other time synchronization. Both engines and OpenMI wrappers can transfer data to and from this intermediate buffer.

The class structure illustrated in Figure 4-38 shows a design pattern on the C# side which is based on a previous stage of OpenMI development. The WExchangeModel handles the InputExchangeItems and OutputExchangeItems and interacts with the model schematization through the WModel class. This functionality is an integrated part of the ILinkableEngine interface.



**Figure 4-38 Class structure of the migrated SOBEK engines**

The WEngineAPIAccess class implements the IRunEngine interface and provides the runtime data exchange interface between the F90 code and the OpenMI wrapper. Figure 4-39 illustrates the main methods available and the interaction between the wrapper classes and the F90 engine module.

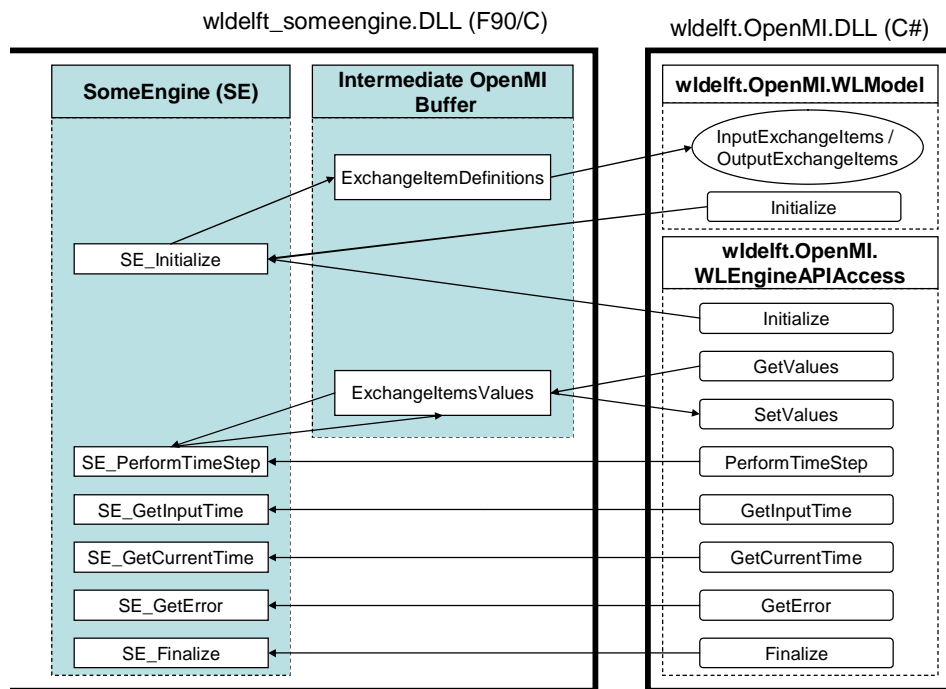


Figure 4-39 Main interaction between wrapper classes, intermediate buffer and engine



## Chapter 4.10 Performance issues

This chapter discusses some of the issues that may affect performance when migrating models to the OpenMI.

### 4.10.1 Memory consumption

When running a set of linked models on one computer it is important to realize that several models will be kept in memory at the same time and that the overall computation time is the sum of the computation time for the individual models. Therefore it is crucial that the individual models should consume as little memory as possible. When the amount of memory used by all programs exceeds the amount of physical memory in the computer, the computer starts swapping chunks of memory to disk. This is very slow and can lead to severe performance degradation.

## 4.10.2 System processes

Although all linkable components run in one system process, the actual computation can either be run in the same process or in a different process. Running the computation in the same process is preferable because communication between processes can be up to a hundred times slower than in-process communication. However, this is not always possible, especially when Fortran code with many global variables is used. In that case there is no other option than to run the Fortran code in a separate process.



# Book 5 Non-model components

<b>BOOK 5</b>	<b>NON-MODEL COMPONENTS .....</b>	<b>5-1</b>
<b>Chapter 5.1</b>	<b>Desktop and database applications.....</b>	<b>5-3</b>
5.1.1	ASCII files.....	5-4
5.1.2	Spreadsheets .....	5-8
5.1.2.1	Generating an ASCII file .....	5-8
5.1.2.2	Accessing Excel using Visual Studio Tools for the Microsoft Office system..	5-8
5.1.3	Report engines .....	5-12
5.1.4	Databases .....	5-13
5.1.4.1	Accessing databases.....	5-13
5.1.4.2	Accessing databases using ADO.NET .....	5-13
5.1.5	GIS.....	5-15
5.1.5.1	Accessing GIS through software libraries.....	5-15
5.1.5.2	Accessing GIS through ASCII files.....	5-15
<b>Chapter 5.2</b>	<b>Visualization .....</b>	<b>5-17</b>
5.2.1	The OpenMI DataMonitor.....	5-18
<b>Chapter 5.3</b>	<b>Advanced controllers .....</b>	<b>5-19</b>
5.3.1	Iteration .....	5-20
5.3.2	Optimization .....	5-26
5.3.3	Calibration .....	5-30



## Chapter 5.1 Desktop and database applications

Previous books have discussed the use of OpenMI in model systems. Although linking models is the primary aim of the OpenMI standard, the OpenMI doesn't exclude the linking of other systems. Online measuring systems, databases and decision support systems can be linked to each other and to models using the OpenMI interfaces.

The OpenMI interfaces are designed to enable data exchange between components. A component is not necessarily a model system; as long as the application is OpenMI-compliant, it will be able to run in combination with other OpenMI.

This chapter illustrates the possible implementations of different kinds of OpenMI-compliant applications. The illustrations should be considered as inspiration for your own developments.

## 5.1.1 ASCII files

The input for model systems is often contained in simple ASCII files. Wrapping these files allows you to make the data available to OpenMI-compliant systems.

For example, you can wrap the output file from one component so that it serves as the input for another component. When such a file is wrapped, it can be accessed in the same way as any other OpenMI component. The wrapped output file makes it easy to test the OpenMI component that uses the file as input. It will also be possible to run the component using the data without having to run the delivering component.

To be OpenMI-compliant, a component needs to implement two interfaces: the IPublisher and ILinkableComponent interfaces. Therefore the ASCII reader needs to implement these interfaces. As with model systems, this can be done by inheriting from org.OpenMI.Utilities Wrapper.LinkableEngine and implementing specific methods.

The difference between the wrapping of ASCII files and the approach used for model components lies in the implementation of the ExchangeItems (GetOutputExchangeItem) and the GetValues call. In the case of a model component, the ExchangeItems will usually be provided by a populated model kernel. When wrapping an ASCII file, the information about possible ExchangeItems must be contained in the ASCII file itself.

```
// Lines starting with // are comment
// The first uncommented line defines the quantity
// The second uncommented line defines the locations
// All next lines consist of a timestamp and values of the quantity for each location
"Flow"
"Loc1";"Loc2";"Loc3"
"2005-01-01 00:00";"15.4";"18.2";"22.4"
"2005-01-01 03:00";"15.3";"18.5";"22.5"
"2005-01-01 06:00";"15.4";"18.9";"22.4"
"2005-01-01 09:00";"15.2";"19.0";"22.6"
"2005-01-01 12:00";"15.1";"18.8";"22.7"
"2005-01-01 15:00";"14.8";"18.6";"22.9"
"2005-01-01 18:00";"14.7";"18.4";"23.4"
"2005-01-01 21:00";"14.7";"18.2";"23.8"
"2005-01-02 00:00";"14.6";"18.2";"23.9"
"2005-01-02 03:00";"14.1";"18.1";"24.0"
"2005-01-02 06:00";"13.8";"18.2";"23.5"
"2005-01-02 09:00";"13.9";"18.0";"23.6"
"2005-01-02 12:00";"13.6";"17.8";"23.5"
"2005-01-02 15:00";"13.2";"17.3";"23.3"
"2005-01-02 18:00";"12.8";"17.4";"23.2"
"2005-01-02 21:00";"12.7";"17.2";"23.1"
"2005-01-03 00:00";"12.6";"17.2";"22.9"
"2005-01-03 03:00";"12.1";"16.8";"22.7"
"2005-01-03 06:00";"12.2";"16.6";"22.5"
"2005-01-03 09:00";"11.4";"16.5";"22.4"
"2005-01-03 12:00";"11.6";"16.4";"22.2"
"2005-01-03 15:00";"11.7";"16.3";"21.8"
"2005-01-03 18:00";"11.6";"16.0";"21.5"
"2005-01-03 21:00";"11.2";"15.8";"21.4"
"2005-01-04 00:00";"11.0";"15.6";"21.3"
```

**Figure 5-1 An example ASCII file**



Figure 5-1 is an example of an ASCII file; there are four lines of comments, followed by a line specifying the quantity and another containing the three locations. This information defines the possible OutputExchangeItems.

The OutputExchangeItems could look like the following:

```
OutputExchangeItem[0] = { "Flow", "Loc1"}
OutputExchangeItem[1] = { "Flow", "Loc2"}
OutputExchangeItem[2] = { "Flow", "Loc3"}
```

Implementing an OpenMI-compatible linkable component that uses this ASCII file as input and makes the contents of the file available through OutputExchangeItems requires the following steps:

- Start developing the component by implementing `org.OpenMI.Utilities.Wrapper.LinkableEngine`.
- Write code for methods such as `Initialize`, `Finalize`, `GetModelID` and `GetModelDescription`.
- Write code for the `SetValues`, `GetInputExchangeItem` and `GetInputExchangeItemCount` methods; these methods are empty because the ASCII reader does not accept data.
- Write code for the `GetOutputExchangeItem` method; make sure the ASCII file is read, the ExchangeItems are extracted from the file and the OutputExchangeItems are defined.
- Write code for the `GetValues` method; this should extract the correct data for the requested timestamp/ExchangeItem combination from the ASCII file.

Prototype code for `GetOutputExchangeItem` is given in Figure 5-2.

```
public IOutputExchangeItem GetOutputExchangeItem(int outputExchangeItemIndex)
{
    return _output;
}

private void ReadFile()
{
    StreamReader reader = new StreamReader(_inputFile);
    bool quantityRead = false;
    bool elementsRead = false;
    _timeStamps.Clear();
    _elementValues.Clear();

    string line;
    while ((line = reader.ReadLine()) != null)
    {
        if (line.StartsWith("//"))
        {
            // ignore
        }
        else if (!quantityRead)
        {
```

```
        _quantity = new Quantity(line.Trim(' ', ''));
        _output = new OutputExchangeItem();
        _output.Quantity = _quantity;
        quantityRead = true;
    }
    else if (!elementsRead)
    {
        string[] elements = line.Split(';');
        _elementSet = new ElementSet();
        _elementSet.ID = "File Contents";

        _output.ElementSet = _elementSet;

        for (int i = 0; i < elements.Length; i++)
        {
            _elementSet.AddElement (new Element(elements[i].Trim('')));
        }
        elementsRead = true;
    }
    else
    {
        string[] values = line.Split(';');
        DateTime timestamp = Convert.ToDateTime (values[0].Trim(''),
            _culture);
        double modifiedJulianDateTime =
            CalendarConverter.Gregorian2ModifiedJulian (timestamp);

        double[] locationValues = new double[values.Length - 1];
        for (int i = 1; i < values.Length; i++)
        {
            locationValues[i - 1] = Convert.ToDouble (values[i].Trim(''),
                _culture);
        }

        _timeStamps.Add (modifiedJulianDateTime);
        _elementValues.Add (locationValues);
    }
}
}
```

**Figure 5-2 Code for GetOutputExchangeItem**

A prototype of the GetValues method is shown in Figure 5-3.

```
public IValueSet GetValues(ITime time, string linkID)
{
    ILink outgoingLink = (Link) _links[linkID];

    if (outgoingLink == null)
    {
        throw new Exception ("Unknown link or quantity");
    }

    if (time is TimeStamp)
    {
        TimeStamp timestamp = (TimeStamp) time;

        double[] results;
        for (int i = 0; i < _timeStamps.Count; i++)
        {
            if ( (double)_timeStamps[i] + _delta > timestamp.ModifiedJulianDay)
            {
                results = (double[]) _elementValues[i];
                return new ScalarSet(results);
            }
        }

        throw new Exception ("No appropriate values found");
    }

    throw new Exception ("Time should be of type TimeStamp");
}
```

**Figure 5-3 Example GetValues method**

## 5.1.2 Spreadsheets

There are a number of ways to make a connection to spreadsheet applications such as Excel. This section discusses some ways of implementing this.

### 5.1.2.1 Generating an ASCII file

One way of connecting to a spreadsheet is to modify the OpenMI DataMonitor by redirecting the output to a file that can be read by desktop applications: for example, the comma separated value (CSV) file used to export information to Excel.

### 5.1.2.2 Accessing Excel using Visual Studio Tools for the Microsoft Office system

Using Visual Tools for Office enables the developer to access office documents such as Word or Excel. The example below illustrates the way in which you can export data to Excel by generating an Excel document from within Visual Studio .NET 2003.

As well as Microsoft Visual Studio .NET 2003 and Microsoft Office 2003, you must also install Visual Studio Tools for the Microsoft Office System.

The first step is to create a reference in the project to the Excel x.x Object Library (the version number depends on the installed version of Microsoft Office). Right-click on the References folder in the Solution Explorer and choose Add Reference. Choose the COM tab and pick Microsoft Excel x.x Object Library (Figure 5-4).

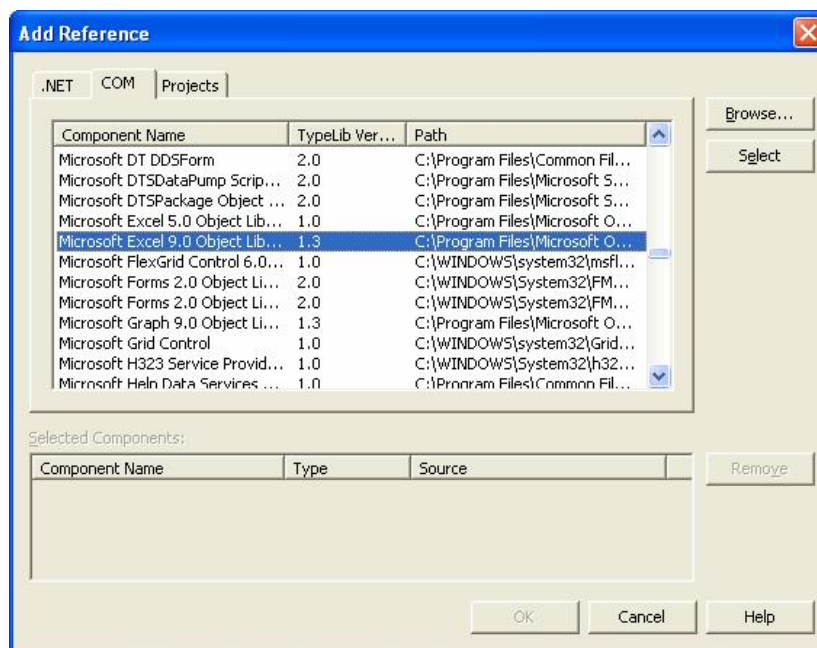


Figure 5-4 Adding an Excel reference

In this example, the DataMonitor application is modified so that it generates an MS-Excel document. Taking the DataMonitor (OpenMI.Tools.Datamonitor.csproj) as a starting point, you need to change some of the methods:

- Add 'using' statements.

```
using System;
using System.Windows.Forms;
using org.OpenMI.Standard;
using org.OpenMI.Backbone;
using System.Diagnostics;
using org.OpenMI.DevelopmentSupport;
using org.OpenMI.Tools.DataMonitor;
using System.Collections;
using System.Threading;
using System.Reflection;
using System.Runtime.InteropServices;
using Excel=Microsoft.Office.Interop.Excel;
```

- Declare workbooks, sheets etc.

```
namespace riza.OpenMI.Tools.DataMonitorPlus
{
    /// <summary>
    /// The DataMonitorPlus class
    /// </summary>
    public class DataMonitorPlus:LinkableComponent, IListener
    {
        private Excel.Application ExcelObj = new Excel.Application();
        private Excel.Workbook _workbook;
        private Excel.Sheets _sheets;
        private Excel.Worksheet _sheet;
```

- Add source to the Initialize method in order to create a workbook and a worksheet.

```
        if (ExcelObj == null)
        {
            throw new Exception("Excel object not loaded");
        }
        // Make Excel object visible and
        // add a workbook to populate
        ExcelObj.Visible=true;
        _workbook = ExcelObj.Workbooks.Add(Type.Missing);
        // Get the sheets collection from the workbook
        _sheets =_workbook.Worksheets;
        // Get the first (and only) sheet
        _sheet = (Excel.Worksheet)_sheets.get_Item(1);
```

- Modify the OnEvent code for the DataMonitor so that it writes the values to the spreadsheet.

```

    /// <summary>
    /// OnEvent
    /// </summary>
    /// <param name="Event">Event</param>
    public static int rownumber;
    public void OnEvent (IEvent Event)
    {
        ILink[] links = GetAcceptingLinks();
        Excel.Range range;

        foreach (ILink link in links)
        {
            if (link.SourceComponent==Event.Sender)
            {
                IValueSet values =
                    Event.Sender.GetValues(Event.SimulationTime,link.ID);

                if (values is IScalarSet)
                {
                    IScalarSet scalarSet = (IScalarSet)values;

                    string[] subitems = new string[4+scalarSet.Count];

                    subitems[0] =
                        CalendarConverter.ModifiedJulian2Gregorian(
                            Event.SimulationTime.ModifiedJulianDay).
                            ToString();
                    subitems[1] = Event.Sender.ModelID;
                    subitems[2] = link.SourceQuantity.ID;
                    subitems[3] = link.SourceElementSet.ID;

                    rownumber++; // need rownumber in sheet

                    for (int i=0;i<scalarSet.Count;i++)
                        subitems[i+4] =
                            scalarSet.GetScalar(i).ToString();

                    ListViewItem item = new ListViewItem(subitems);
                    _form.listView1.Items.Add(item);

                    // write all cells to next row in Excel Worksheet
                    for (int i=0;i<scalarSet.Count;i++)
                    {
                        char rangeChar = (char)(i+65);
                        string strCell = rangeChar +
                            Convert.ToString(rownumber);
                        range = _sheet.get_Range(strCell,strCell);
                        range.Value2=subitems[i];
                    }
                }
            }
        }
    }

```

The difficult part is to determine the cell co-ordinates for the spreadsheet. Excel uses ranges such as (A1:C1). In our example, such a range is being created for each value in the ValueSet (B3:B3) based on the row number and position in the 'subitems' array.

The code shows how to incorporate Excel documents in .NET code. This code needs to be completed by saving the Excel document to a specified filename (either using an argument from the OMI file or by user input) and closing the file.

### 5.1.3 Report engines

You can generate reports by developing an OpenMI-compatible tool that incorporates the Crystal Reports Engine (integrated in the .NET environment). With this engine you can define a standard report that can be exported to the following industry standards:

- Adobe Acrobat (.pdf)
- Crystal for Visual Studio .NET (.rpt)
- HTML 3.2 and 4.0 (.html)
- Microsoft Excel (.xls)
- Microsoft Rich Text (.rtf)
- Microsoft Word (.doc)



## 5.1.4 Databases

In the same way as for the ASCII reader, you can expose data contained in a database via the OpenMI by replacing the methods that implement the `ExchangeItems` and the `GetValues` calls. Furthermore it is possible to store data in the database through the `SetValues` call. One way of implementing this is to use SQL statements.

### 5.1.4.1 Accessing databases

In the OpenMI, the delivering component is obliged to return a set of values for the requested time. Each `GetValues` request initiates a processing activity. As many water-related models progress in time, the time argument is the controlling variable for any processing activity. Linkable components that do not progress in time can neglect the time argument; they do the required processing and return the data. However, they must be able to pass the time argument to another component if they invoke a `GetValues` call themselves.

If the requested time does not match the timestep in the computation and the computation is already ahead, an interpolated value must be delivered. If the computation has not yet reached the requested time, two alternatives are available:

- Calculate the values at the requested time.
- Extrapolate the solution.

Under normal conditions the first alternative should be chosen. For bidirectional links, one of the components will need to extrapolate its solution in order to prevent deadlock situations.

Components that do not progress in time, such as databases, should also be able to return a value, whether it is the actual, interpolated or extrapolated one. For those situations where the exact time information is required, an additional interface is provided to obtain the discrete timestamps available. Implementation of this interface, `IDiscreteTimes`, is optional.

### 5.1.4.2 Accessing databases using ADO.NET

The .NET environment offers a set of classes to access different kinds of database (MS-Access, SQL-Server, Oracle). Developing a tool that saves data from a `GetValues` call to a database is not difficult. The steps are as follows:

1. Connect to an SQL server (Figure 5-5).

```
// C#
this.sqlConnection1 = new System.Data.SqlClient.SqlConnection();
this.sqlConnection1.ConnectionString = "data source=riz02\NETSDK; " +
    "initial catalog=OpenMI;" +
    "user id=User;password=Open;" +
    "persist security info=True;" +
    "workstation id=riz02;" +
    "packet size=4096";
```

**Figure 5-5 Connecting to an SQL server**

## 2. Execute an SQL command (Figure 5-6).

```
//  
// SQL command Select  
//  
this.sqlSelectCommand.CommandText = " select * from measurement";  
this.sqlSelectCommand.Connection=this.sqlConnection1;  
//  
// SQL command Insert  
//  
this.sqlInsertCommand.CommandText= " insert into OpenMI.measurement  
                                (Quantity, Time, Value)  
                                VALUES (QuantityID, Timestamp, Values[0])  
this.sqlSelectCommand.Connection=this.sqlConnection1;
```

**Figure 5-6 Executing an SQL Command**

This is an example of accessing an SQL-server database. The .NET environment provides classes to access many types of database, including MS-Access.

## 5.1.5 GIS

Geographic information systems (GIS) are widely used to generate model input or present and process model output. This section gives some suggestions on how to link GIS systems to OpenMI components.

### 5.1.5.1 Accessing GIS through software libraries

Most GIS systems support incorporating part of their functionality into custom-built applications.

For example, with the ESRI GIS software suite, a developer can use MapObjects or ArcObjects depending on the functionality that is needed. MapObjects is a software library with functions to display maps within the .NET environment. With ArcObjects (which is platform-independent), it becomes possible to incorporate complete GIS functionality into a custom application such as an OpenMI component. Therefore with these libraries you can build an OpenMI-compatible component with full GIS functionality.

### 5.1.5.2 Accessing GIS through ASCII files

A very easy way to exchange data between an OpenMI component and a GIS system is by generating or importing an ASCII file in the GIS environment. Two examples are given below.

#### Exporting data from GIS to an ASCII file

By exporting data from a GIS system to an ASCII file you can include this file in an OpenMI composition using a tool like the ASCII reader discussed in section 5.1.1.

ArcInfo is one of the GIS systems used. Figure 5-7 shows ArcInfo aml code combining three quantities into one grid, sorting this grid by 'krwdeelstr' (one of the quantities) and then exporting the generated table to a comma separated value (ASCII) file.

```
krwmaxima = combine(..\..\krwdeelstr, krwmax74, krwmax74gt0)
setmask off
arc tables
select krwmaxima.vat
sort krwdeelstr
unload krwmaxima.csv krwdeelstr krwmax74 krwmax74gt0 delimited init
q
```

**Figure 5-7 ArcInfo aml (export)**

#### Importing data from an ASCII file to GIS

If you have developed a tool to generate ASCII files that are based on the results from an OpenMI component, you can use this file to import the data into a GIS system. Figure 5-8 shows the ArcInfo code used to import data from an ASCII file that has been formatted as shown in Figure 5-9.

```

&echo &brief

setwindow 0 300000 300000 625000
setcell 500

&do p &list g3 d3 b3
/* &do p &list f1 f2 f3 f4 f5

arc w m:\national_calculation_2005\%p%\

&r kill gvg2-%p% glg2-%p% ghg2-%p% dgvg2-%p% dglg2-%p% dghg2-%p%

setmask m:\national_calculation_2005\g3\gws-recharge\ghg2
gvg2-%p% = ( float(reclass ( m:\mona\data\droogtebasis\plotcode_500 , gvg.rmp )) /
  1000000 )
glg2-%p% = ( float(reclass ( m:\mona\data\droogtebasis\plotcode_500 , glg.rmp )) /
  1000000 )
ghg2-%p% = ( float(reclass ( m:\mona\data\droogtebasis\plotcode_500 , ghg.rmp )) /
  1000000 )

/*dgvg2-%p% = gvg2-%p% - m:\national_calculation_2005\%p%\gws-recharge\gvg2
/*dglg2-%p% = glg2-%p% - m:\national_calculation_2005\%p%\gws-recharge\glg2
/*dghg2-%p% = ghg2-%p% - m:\national_calculation_2005\%p%\gws-recharge\ghg2

&end

```

**Figure 5-8 ArcInfo aml (import)**

```

16612 : 1398699
16613 : 3730433
16614 : 4181700
16615 : 3039099
16616 : 1606600
16617 : 3452666
16618 : 3256366
16619 : 2974466
16620 : 2237799
16621 : 1531333
16622 : 1693533
16623 : 2394033
16624 : 1874899
etc.

```

**Figure 5-9 Imported ASCII file**

The ASCII file contains two columns; the first column identifies the grid number and the second column identifies the quantity value multiplied by 1000000 (in order to get an integer value, which is obligatory).

The command 'reclass' imports the specified ASCII file. The imported value is divided by 1000000 and converted to 'float'.

The example above is just one of the many possible ways to import or export data to and from GIS systems.

## Chapter 5.2 Visualization

This chapter provides background information on how to include a new data display function.

## 5.2.1 The OpenMI DataMonitor

As the users of OpenMI will certainly need applications to present results or apply statistical analysis, a variety of applications can be developed. The OpenMI DataMonitor is a good starting point to understand how OpenMI can be used in combination with visualization tools.

One important design aspect needs consideration. In the OpenMI, components are triggered by a GetValues call. Applications like the DataMonitor also need data, which they can obtain by another GetValues call. Care must be taken when deciding the point at which to place this call. Furthermore, post-processing applications are not meant to trigger model applications to start a new computation.

This issue is covered by the DataChanged event. The DataMonitor is linked to a linkable component and is registered as an Event Listener to this component. As soon as the delivering component has answered a GetValues call from another linkable component, it will throw a DataChanged event. The DataMonitor will respond to this event by issuing a GetValues call to the delivering component. This component is able to return the GetValues call immediately because it has the requested data in its buffer. This is illustrated in Figure 5-10, where the Visualization component is the DataMonitor.

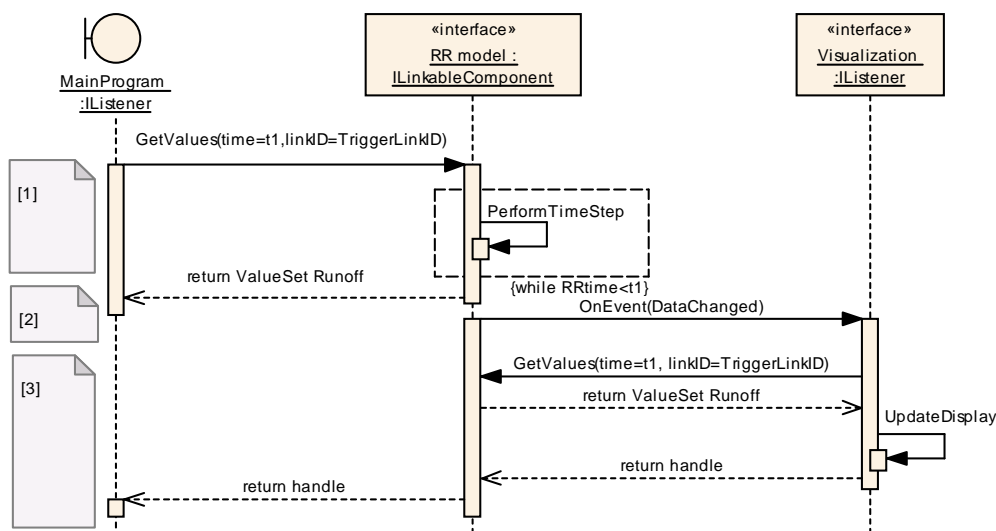


Figure 5-10 The DataChanged event

Another issue to keep in mind is the fact that linkable components may have the IManageState interface implemented. This means that these components may go back in time, so the data received through the GetValues call from these components is not sequential. A way to handle these situations is by implementing a buffer and checking whether the received data is to overwrite data in the buffer or be appended to the buffer.

## Chapter 5.3 Advanced controllers

OpenMI can be used to address iteration, optimization or calibration issues. To illustrate these applications, a number of advanced controllers have been developed as part of the org.OpenMI.Utilities package. Their usage is described in this chapter.

### 5.3.1 Iteration

Before using an iterated link, it is strongly recommended that you try using a bidirectional link first. Only when there are strong backwater effects are iterated links really needed. Usually a bidirectional link with a small timestep will work; in the example below a bidirectional link with no iterations gives the same result. A bidirectional link is much easier to set up and will run much faster. Also, to support iterations the models used will have to be able to restore their state to an earlier time, something that not all OpenMI-compliant models will be able to do. You should check with the model supplier to ensure that the model supports the ability to restore its state before trying to use any of the controllers.

The iteration controller should be placed between any components that are to be iterated. Any access to these components should not be done directly but through the iteration controller. The iteration controller has a number of data slots, numbered 1, 2, 3 etc. In order to connect quantities through the iteration controller, the output exchange item of the providing component should be connected to a certain slot (e.g. 1), and the input exchange item of the receiving component should be connected to the same slot (i.e. 1). One slot should only be used for one link.

Figure 5-11<sup>1</sup> shows an example of two river models connected with the iteration controllers. Note that there are no direct connections between the two model components; all connections are through the iteration controller. Figure 5-12, Figure 5-13, Figure 5-14 and Figure 5-15 show the links that were used. The iteration controller should be selected as the start-up component in this case.

The start and end time and timestep should be chosen in the OpenMI user interface and the simulation can be run as normal.

---

<sup>1</sup> The user interface as shown is not operational anymore



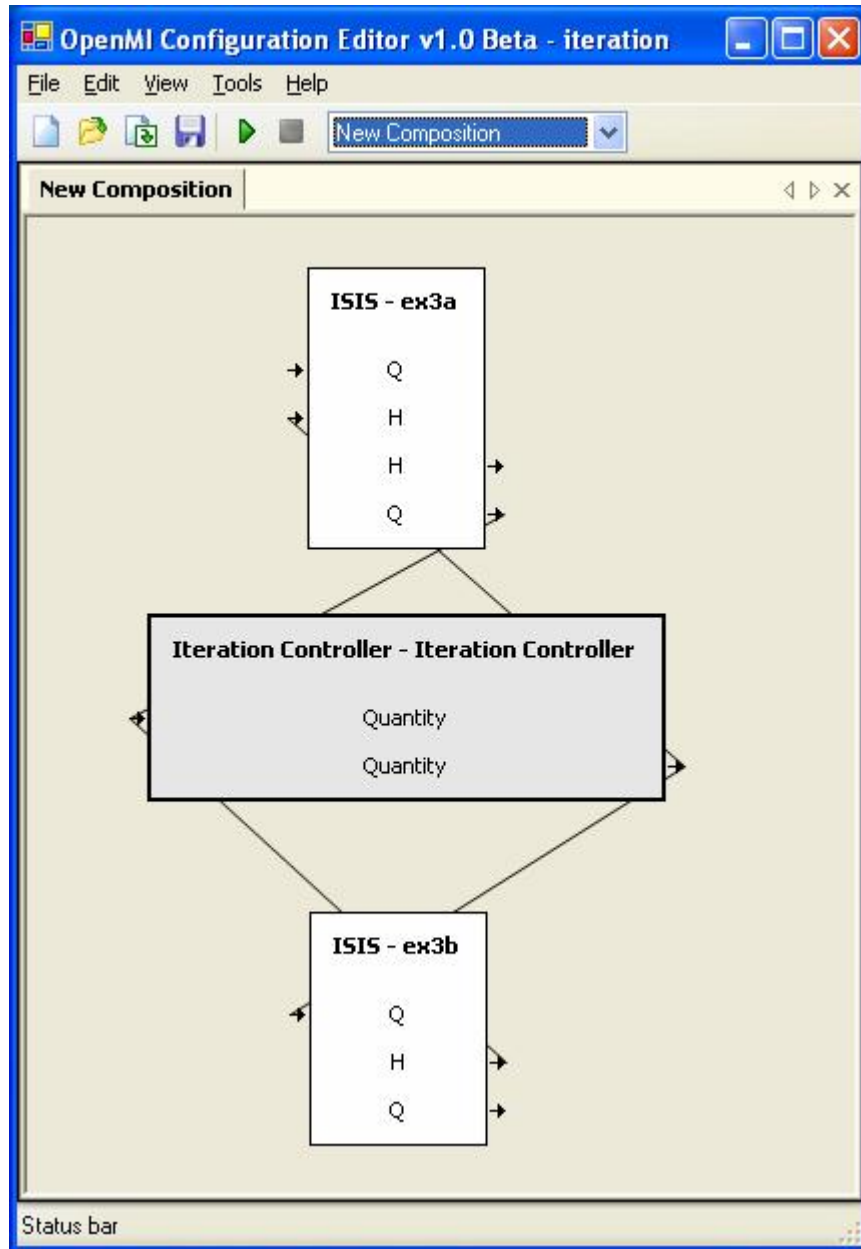


Figure 5-11 Two river models linked by the iteration controller

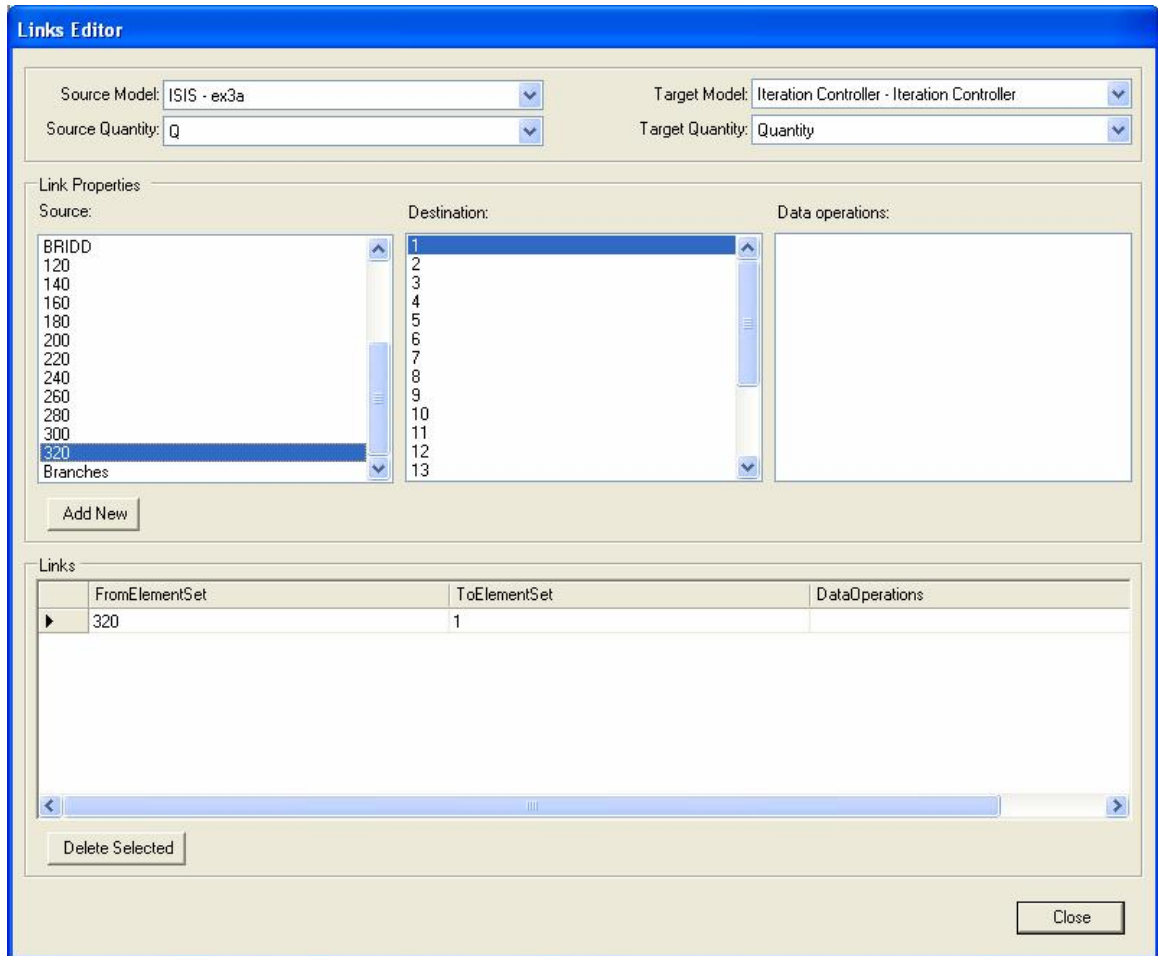


Figure 5-12 Linking the outflow from model 1 to slot 1

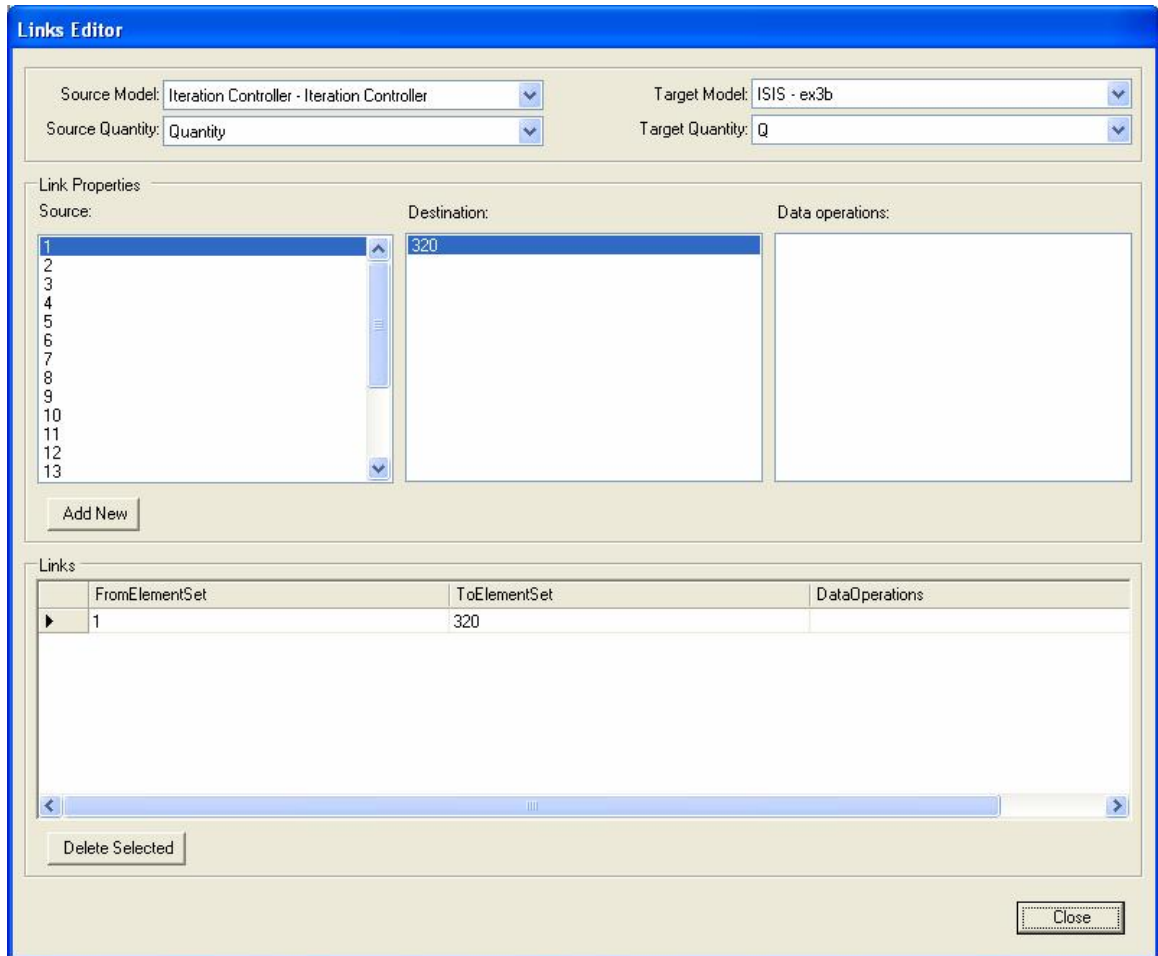


Figure 5-13 Linking the output from slot 1 to the input of the second river model

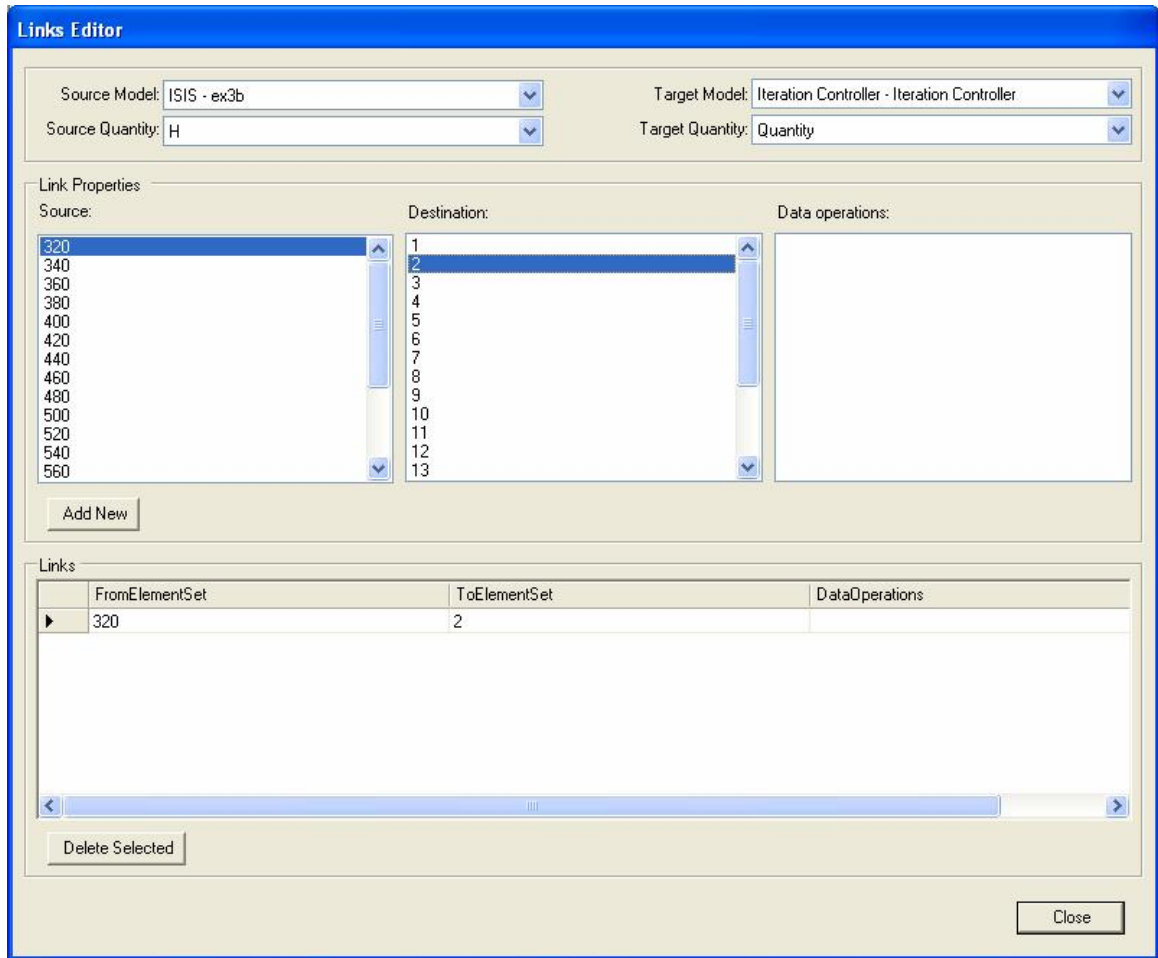


Figure 5-14 Linking the output (stage) of the second model to slot 2

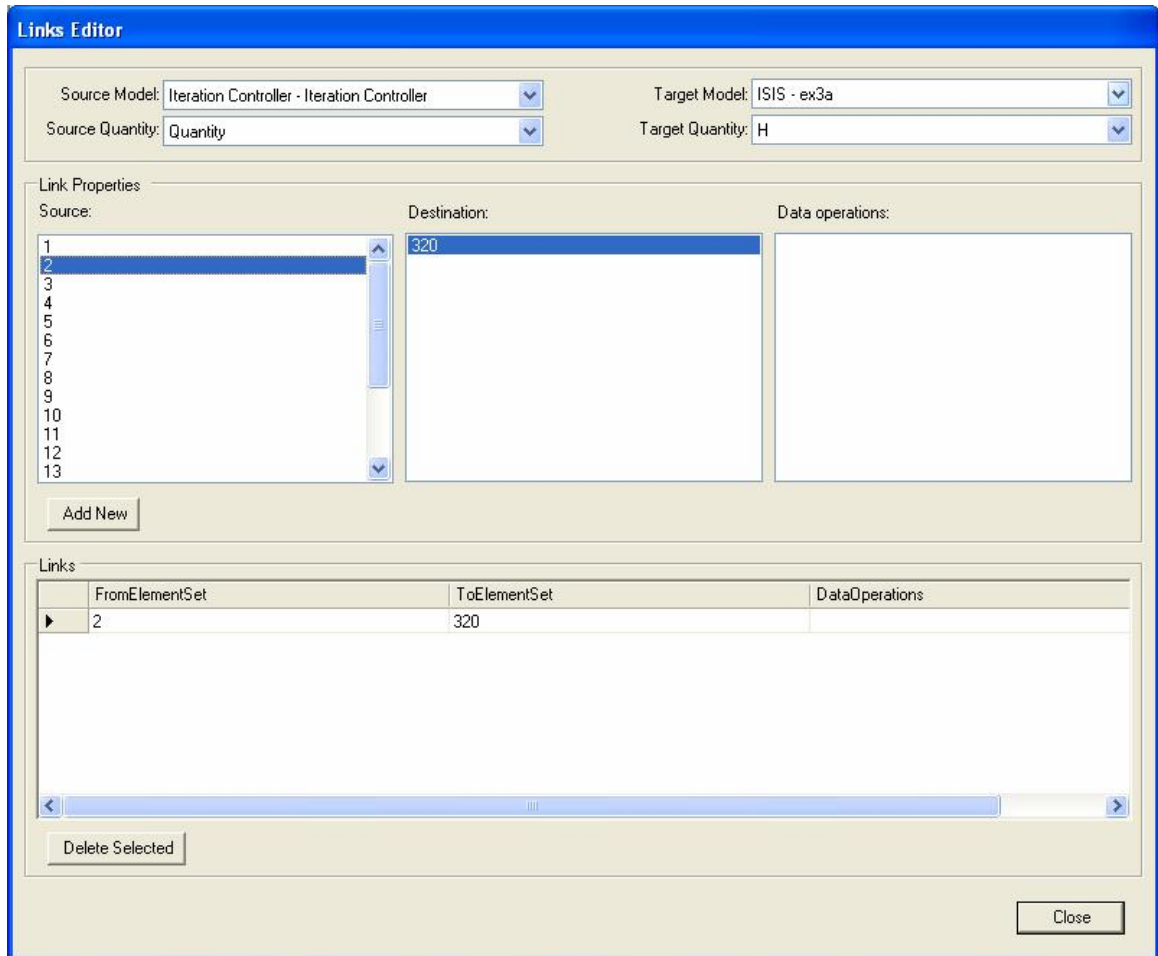


Figure 5-15 Linking the output of slot 2 to the stage input of the first model

### 5.3.2 Optimization

For optimization an objective function is needed. This objective function takes a number of parameters and calculates a value. The optimizer provides parameter values to the objective function and the objective function returns values to the optimizer. The optimizer tries to minimize or maximize the objective function.

The number of parameters and the minimum, maximum and starting value for the optimization are specified in the OMI file. Figure 5-16 shows an example OMI file for two parameters, which both have a minimum of  $-100$ , a maximum of  $100$  and a starting value of  $3.5$ . There is no limit on the maximum number of parameters, although the optimization might take a very long time with a large number of parameters.

```
<?XML version="1.0"?>
<LinkableComponent Type="org.OpenMI.Utilities.AdvancedControl.OptimizationController"
  Assembly="org.OpenMI.Utilities.AdvancedControl.DLL">
  <Arguments>
    <Argument Key="Parameter" ReadOnly="true" Value="P0,-100,100,3.5" />
    <Argument Key="Parameter" ReadOnly="true" Value="P1,-100,100,3.5" />
  </Arguments>
</LinkableComponent>
```

**Figure 5-16 An example OMI file for the optimization controller with two parameters**

The next step is to connect the objective function and the optimizer in the OpenMI GUI (Figure 5-17). Figure 5-18 and Figure 5-19 show how to connect the objective function to the optimizer.

The simulation is run as usual and the message window in Figure 5-17 shows the result. The real minimum for the objective function is  $(0, 0)$  and the real minimum cost is  $-1$  so the optimizer has been effective in this case.

The optimizer is based on a genetic algorithm and should produce reasonable results with most objective functions.

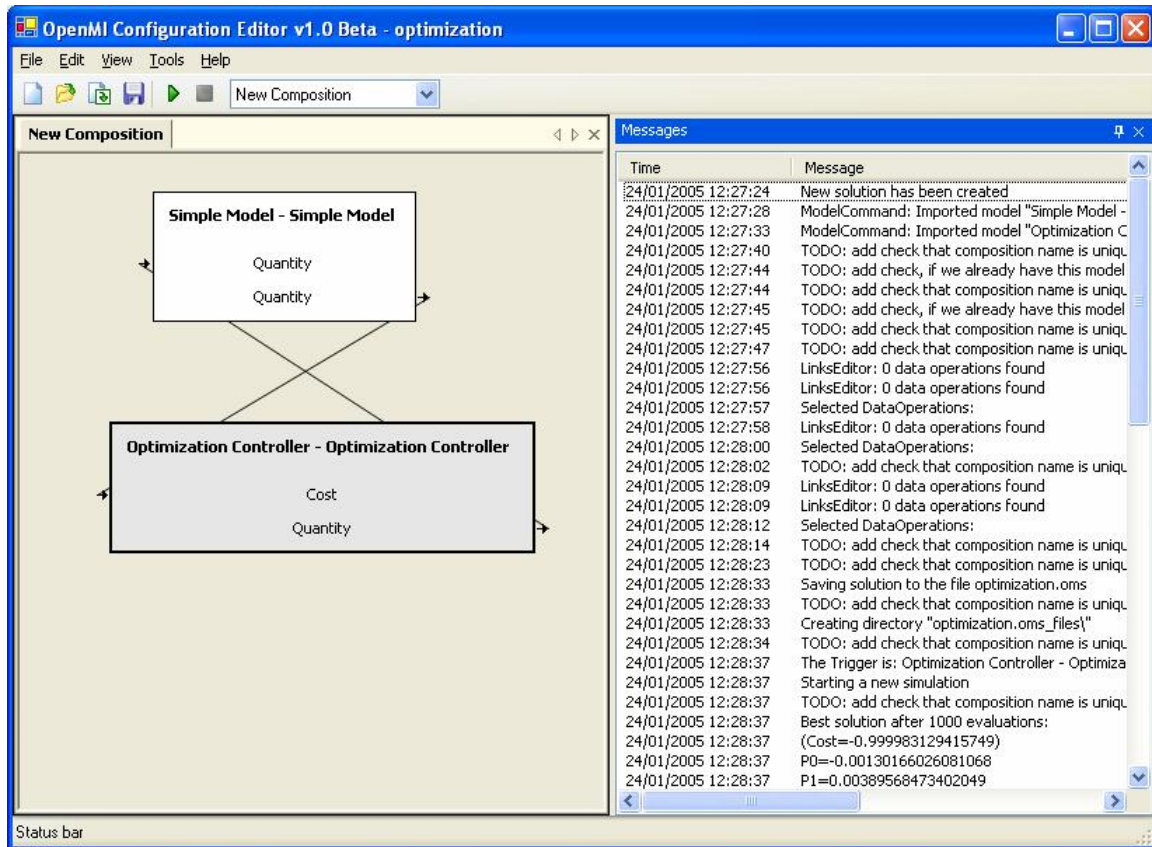
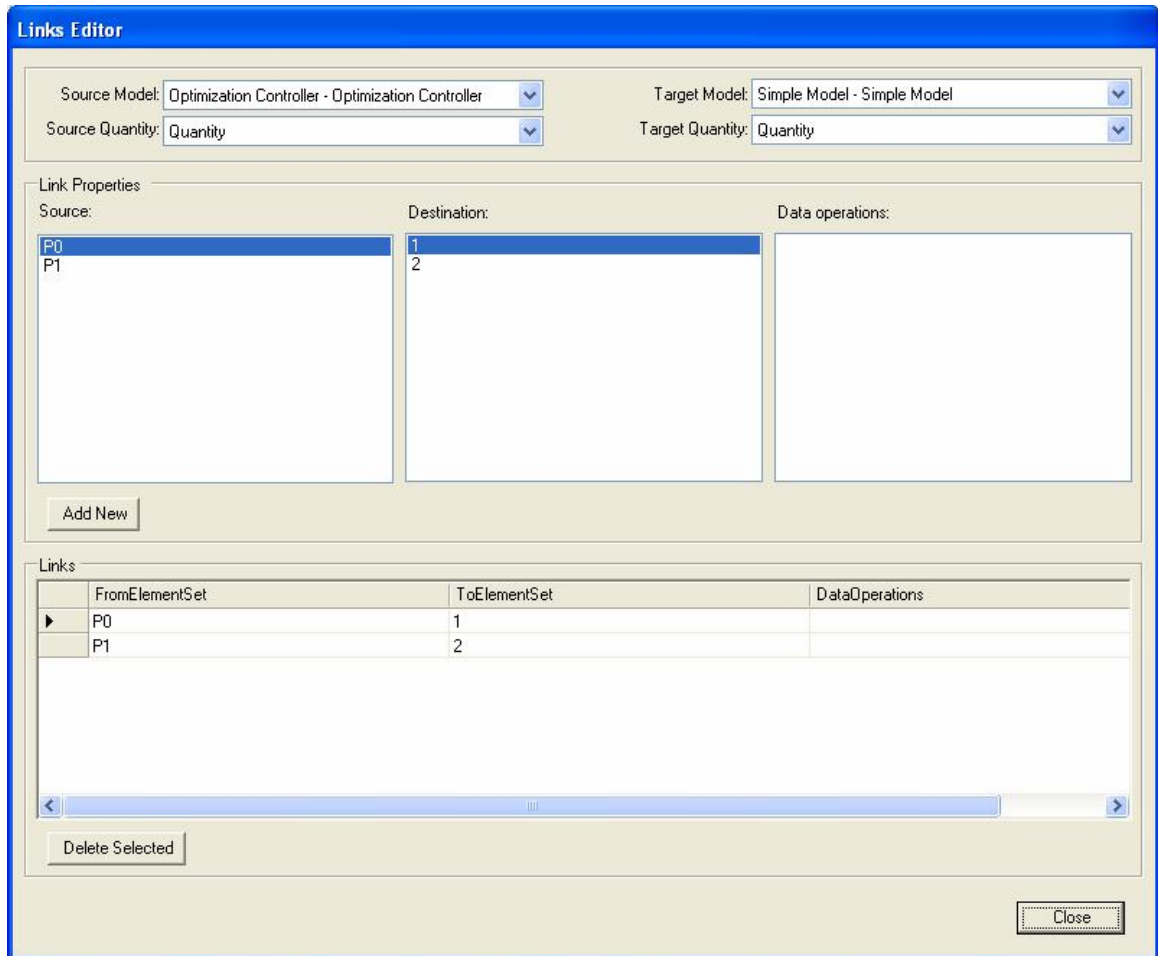
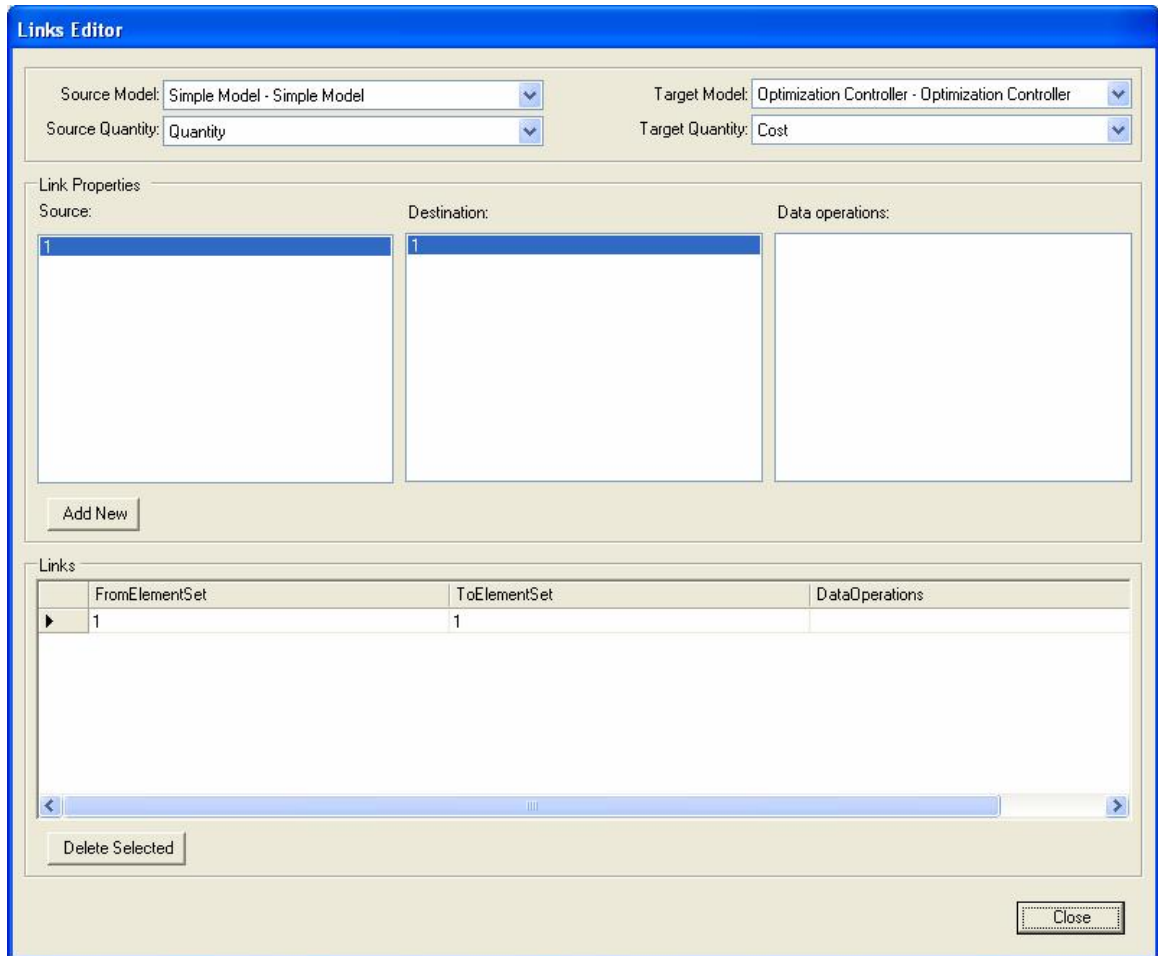


Figure 5-17 Linking the optimizer to the objective function



**Figure 5-18 The link from the optimizer to the objective function**





**Figure 5-19 The link from the objective function to the optimizer**



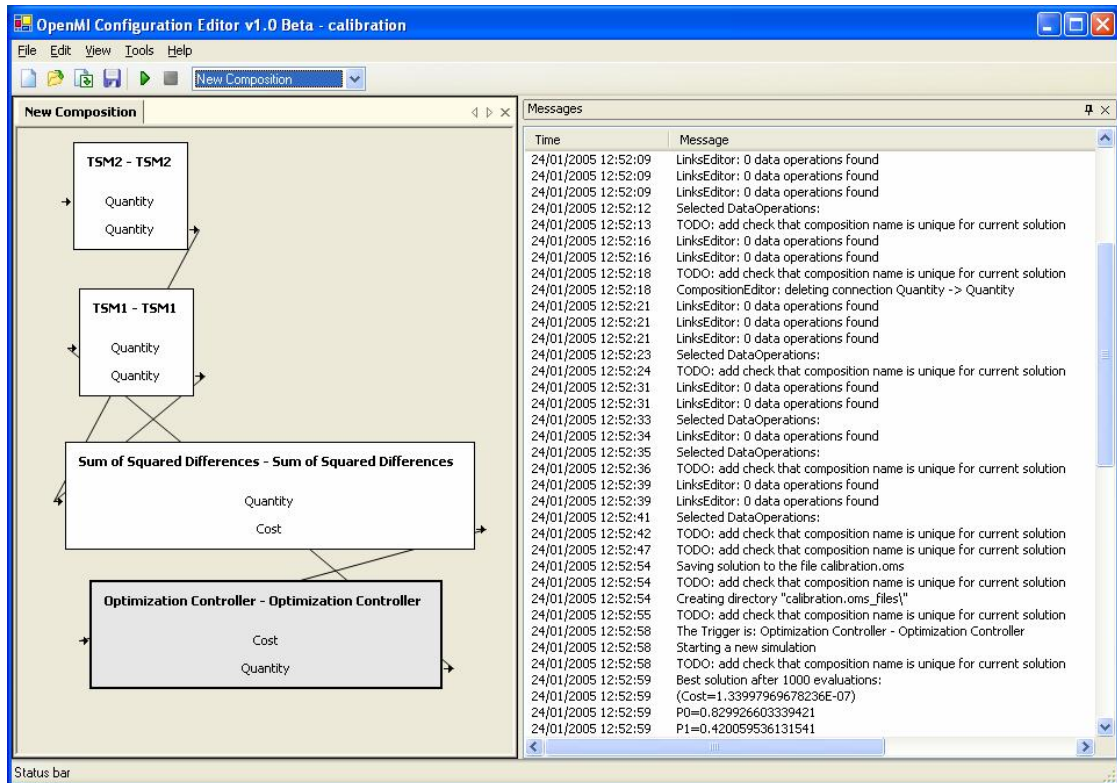
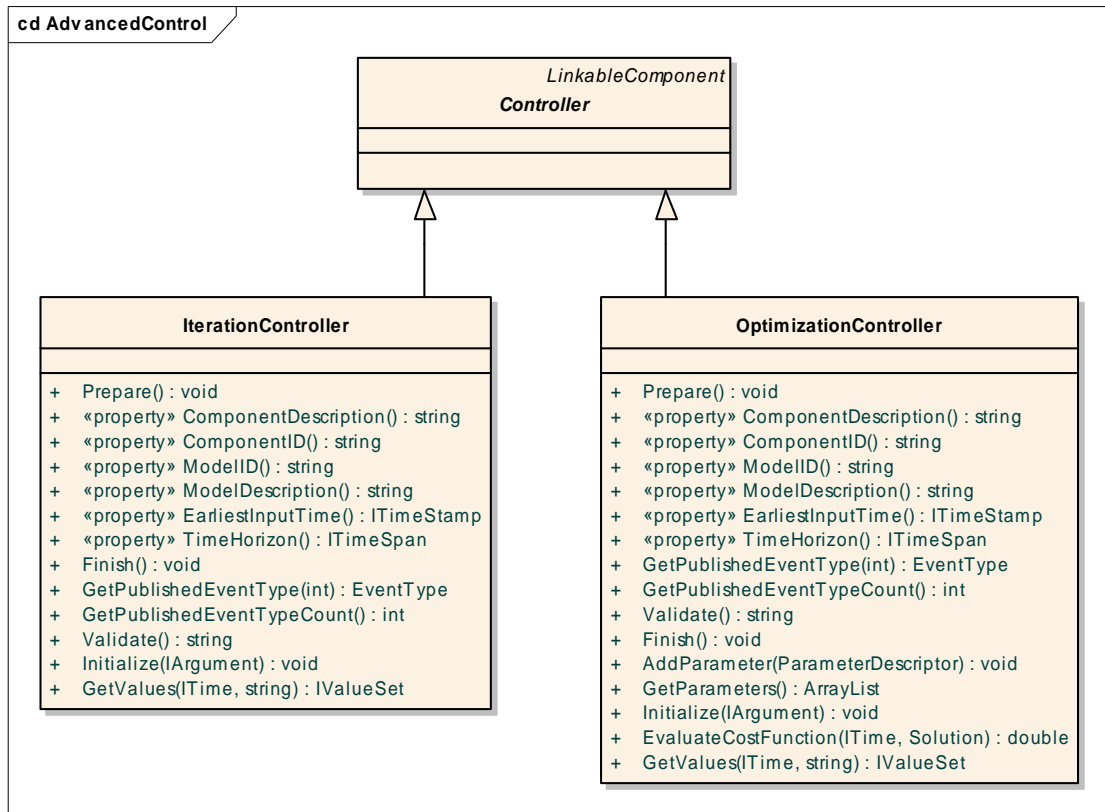


Figure 5-21 Setting up the calibration

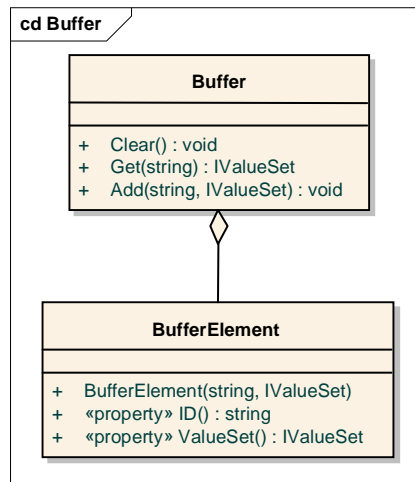
### 5.3.4 UML diagrams

Figure 5-22 shows the UML diagram for the advanced controllers. The generic Controller class is derived from LinkableComponent and both the IterationController and OptimizationController classes derive from the Controller class.



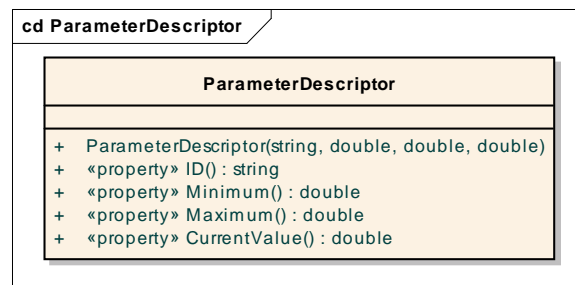
**Figure 5-22 The advanced controllers**

The advanced controllers contain an internal buffer, the UML diagram for which is shown in Figure 5-23.



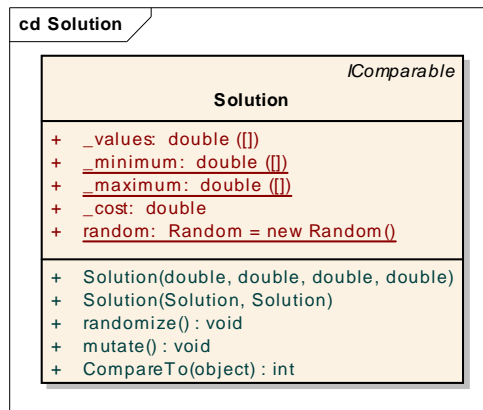
**Figure 5-23 The internal buffer**

The optimization and calibration controllers use a parameter descriptor to describe the minimum, maximum and default values for parameters. Figure 5-24 shows the UML diagram for the parameter descriptor.



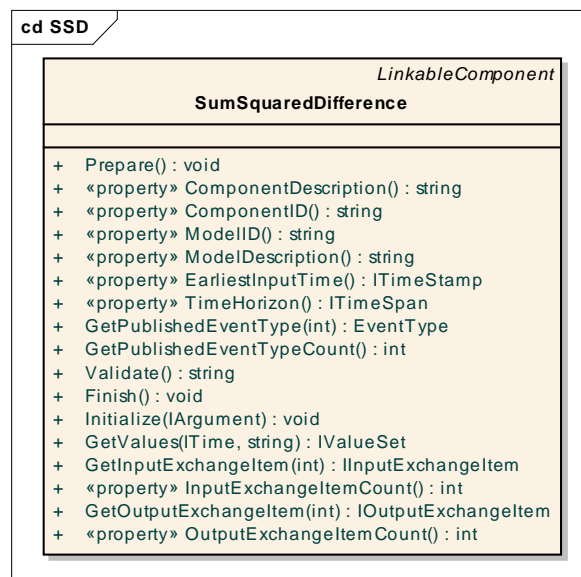
**Figure 5-24 The parameter descriptor**

The optimization and iteration controllers use a Solution class to store candidate solutions. Figure 5-25 shows the corresponding UML diagram.



**Figure 5-25 The Solution class**

Figure 5-26 shows the UML diagram for the 'sum of squared differences' component.



**Figure 5-26 The SumSquaredDifference class**

Figure 5-27 shows an example of two models linked together using the iteration controller.

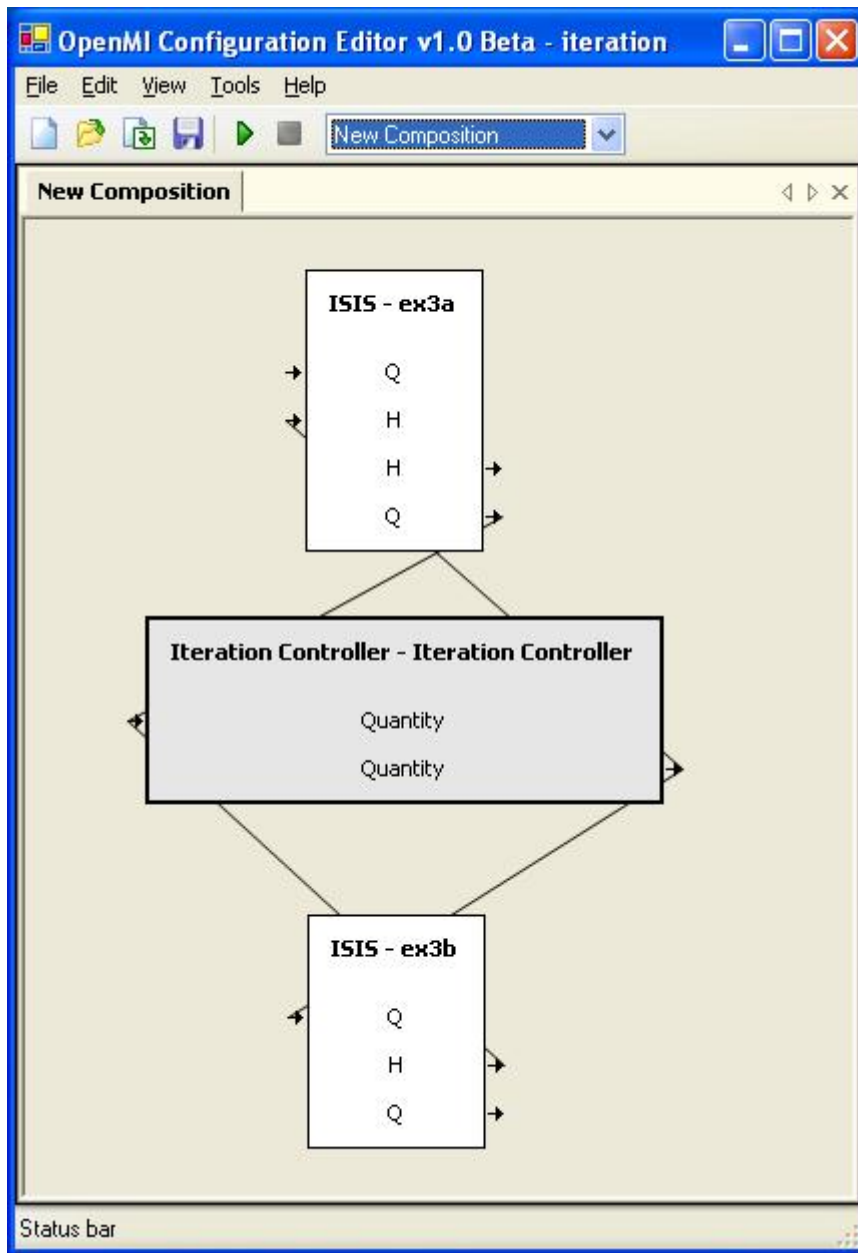


Figure 5-27 Two models linked together by the iteration controller

Figure 5-28, Figure 5-29 and Figure 5-30 show the sequence diagrams for the iteration controller, optimisation controller and calibration controller respectively. Note that Q' and H' are values after relaxation.

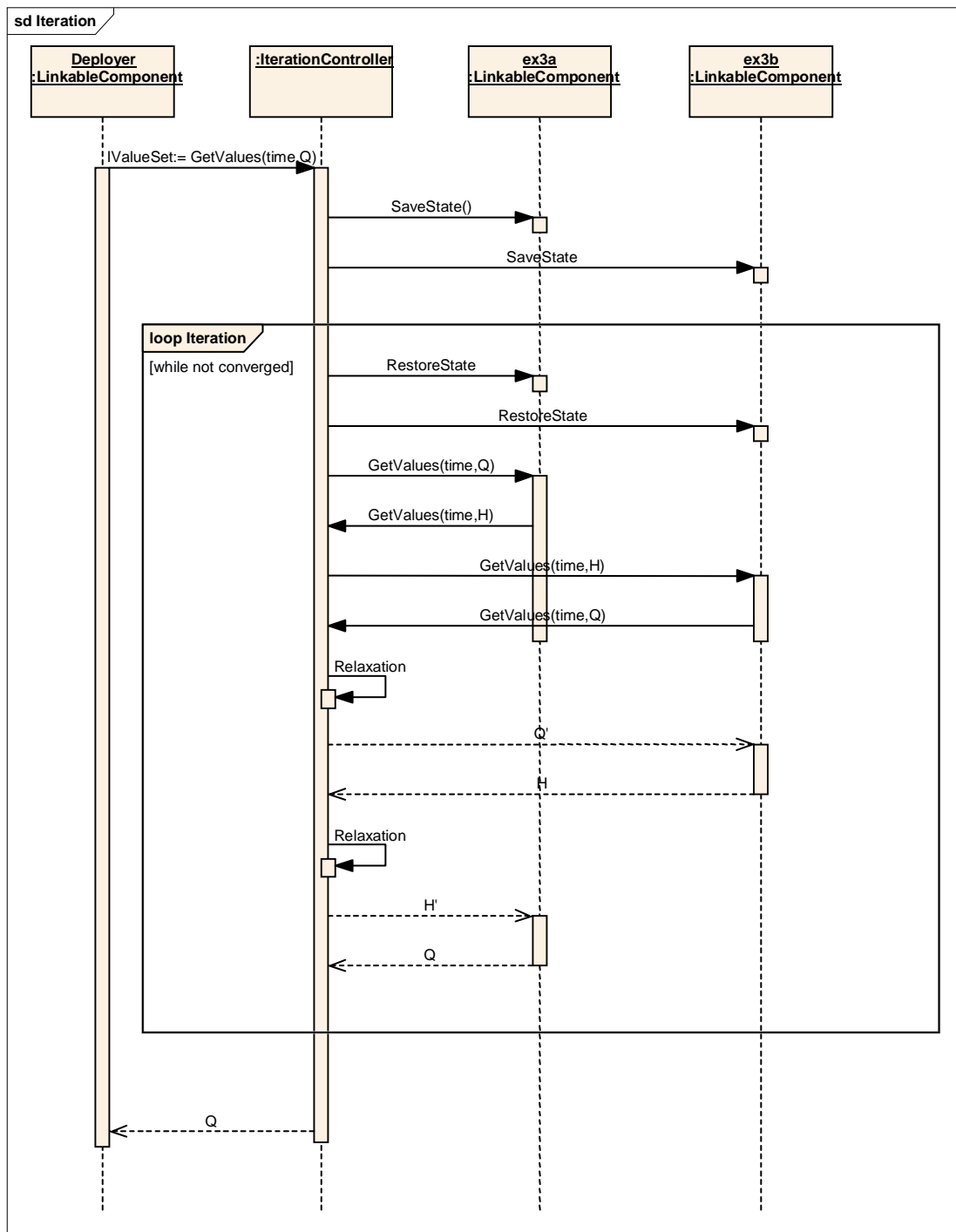


Figure 5-28 Sequence diagram for the iteration controller



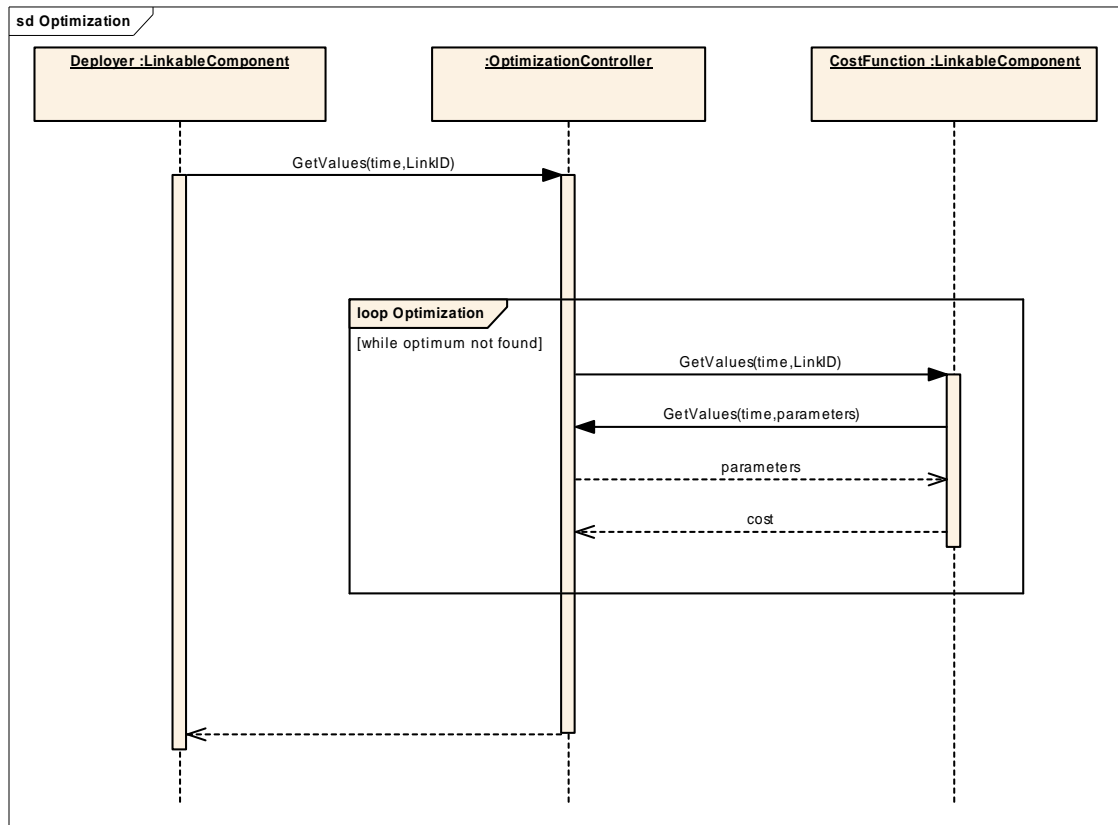


Figure 5-29 Sequence diagram for the optimization controller

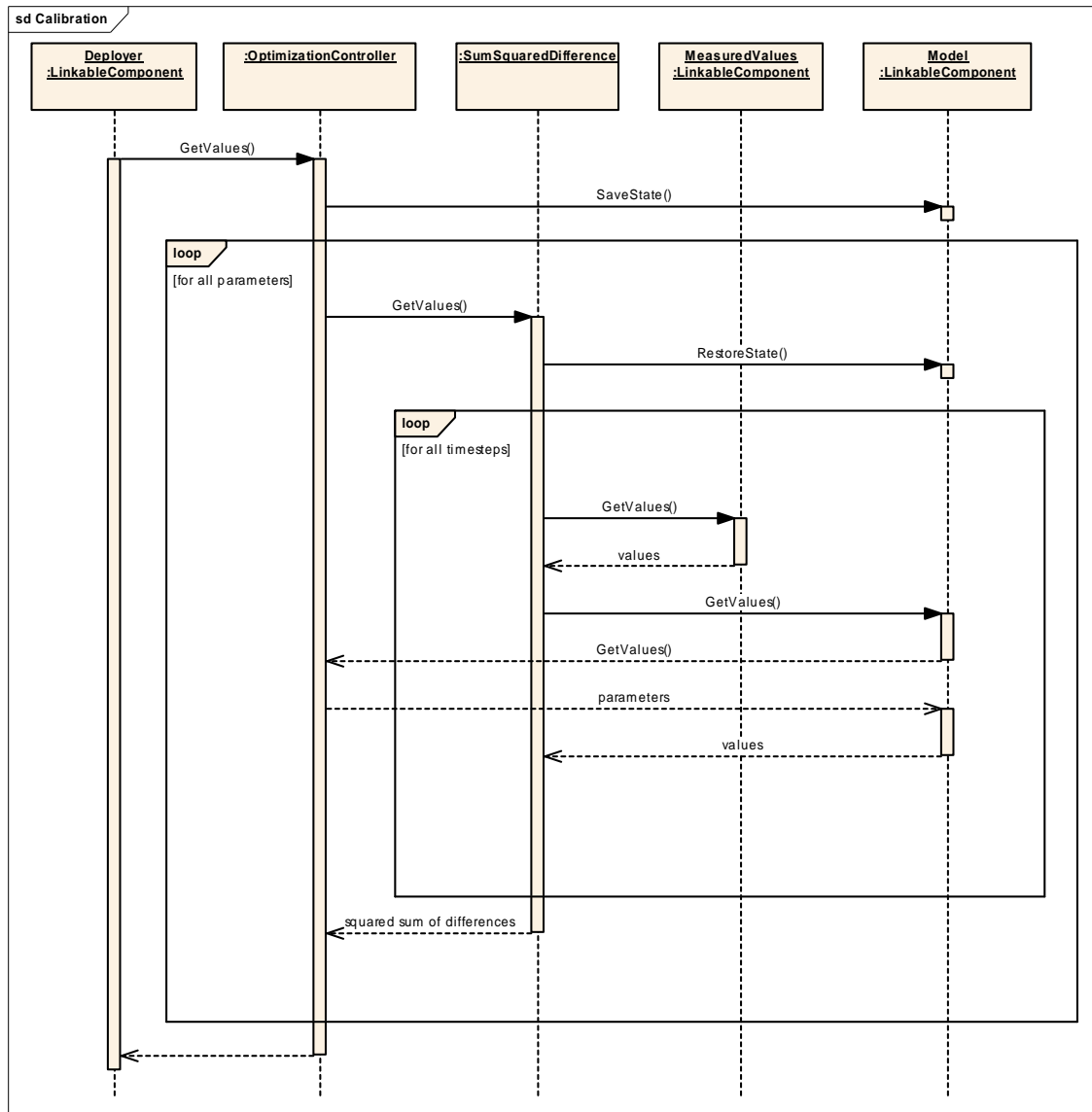


Figure 5-30 Sequence diagram for the calibration controller

## Index

- .
  - .NET assemblies, 4-20
  - .NET development environment, 4-20
- A**
- Accessing databases, 5-13
  - Accessing engine core functions, 4-22
  - Accessing GIS, 5-15
  - Activation, 1-18
  - ADO.NET, 5-13
  - Advanced controllers, 5-21
    - calibration, 5-32
    - iteration, 5-22
    - optimization, 5-28
    - UML diagrams, 5-34
  - ASCII files, 5-4
    - input to spreadsheets, 5-8
    - using with GIS, 5-15
  - Attributes, 1-17
- B**
- Base classes, 1-15
  - Behaviour of objects, 1-14
  - Benefits of OpenMI, 1-6
  - Bidirectional links, 2-7
  - Building visual tools, 3-38
- C**
- Calibration, 5-32
  - Changing the engine core, 4-18
  - Child classes, 1-15
  - Class diagrams, 1-16
  - Class roles, 1-18
  - Classes, 1-14
  - Completion phase, 3-14
  - Components
    - locating, 3-5
  - Compositions, 2-38
    - adding models, 2-41
    - running, 2-46
    - triggers, 2-45
    - XML files, 3-26
  - Computation phase, 3-13
  - Configurable OpenMI systems, 1-32
  - Configurable systems, 3-23
  - Configuration, 3-25
    - validation, 3-25
  - Configuration editor, 2-39
  - Configuration phase, 3-10
  - Configuring connections, 2-44
  - Configuring links, 2-36
  - Connecting models, 2-43
  - Connections between models, 2-43
    - configuring, 2-44
  - Creating .NET assemblies, 4-20
  - Creation of objects, 1-19
  - Crystal Reports Engine, 5-12
- D**
- Data
    - exchanging, 2-9
    - mapping, 2-22
  - Data exchange mechanism, 2-4
  - Database applications, 5-3
  - Databases
    - accessing, 5-13
  - Databases (OpenMI compliance), 5-13
  - DataMonitor, 5-20
  - Deployer, 2-38
  - Deployer component, 3-31
  - Deploying systems, 3-31
  - Deployment phases, 1-33, 3-8
  - Derived classes, 1-15
  - Design patterns, 4-51
  - Desktop applications, 5-3
  - Destruction of objects, 1-19
  - Dimensions, 2-12
  - Disposal phase, 3-14
  - Dynamic ElementSet, 2-21
- E**
- Element sets, 2-6, 2-15
  - Elements, 2-15
    - combining types, 2-17
  - ElementSet, 2-15
    - dynamic, 2-21
  - ElementType
    - choosing, 2-20
  - Encapsulation, 1-14
  - Engine components, 1-9, 1-34
  - Engine core
    - accessing functions, 4-22
    - changing, 4-18
  - Engine interfaces, 1-9
  - Engines, 1-9
  - Error handling, 1-6
  - Excel (OpenMI compliance), 5-8
  - Exchange items, 4-12
    - defining, 4-12
  - Exchange models, 2-26
  - ExchangeItem, 2-26
  - Exchanging data, 2-9
    - use case, 2-10
    - values, 2-11
  - Execution phase, 3-13
- F**
- Flux (quantities), 2-20

Functions, 1-14, 1-17

## G

Geographic information systems, 5-15

GetValues method, 1-28, 2-4  
implementation, 4-37

## GIS

accessing, 5-15  
using ASCII files, 5-15

GIS (OpenMI compliance), 5-15

Graphical user interfaces, 3-37

GUIs, 3-37

## H

Hard-coded systems, 3-17

## I

IElementSet, 2-6

IEngine interface, 4-18

implementation of methods, 4-38

ILinkableComponent, 2-4

ILinkableEngine

implementing, 4-26

IManageState, 4-46

implementing, 4-45

Implementing IManageState, 4-45

Implementing MyEngineDotNetAccess, 4-24

Implementing MyEngineWrapper, 4-26

Implementing MyModelLinkableComponent, 4-28

Information hiding, 1-14

InfoWorks RS migration, 4-53

Inheritance, 1-15

Initialization phase, 3-9

Initialize method, 4-33

Inspection phase, 3-10

Instances of classes, 1-14

Instantiation phase, 3-9

Integrated water management, 1-4

Interfaces (objects), 1-21

ISIS migration, 4-52

Iteration controller, 5-22

## L

Linkable component interface, 3-8

Linkable components, 1-9, 1-34

phases for use, 3-8

LinkableEngine, 4-15

Linking models, 1-23, 2-43

Links, 2-35

configuring, 2-36

Loops, 1-20

## M

Mapping data, 2-22

Memory consumption, 4-60

Messages, 1-17, 1-18

Methods, 1-14, 1-17

Migrating models, 1-34, 4-7

Migration

InfoWorks RS, 4-53

ISIS, 4-52

Mike11, 4-54

planning, 4-7

SOBEK, 4-57

steps in process, 4-17

use cases, 4-8

Migration process, 1-35

Mike11 migration, 4-54

Model application, 1-9

Model properties, 2-41

Models, 1-9

adding to composition, 2-41

linking, 1-23, 2-43

migrating, 1-34, 4-7

Modularity, 1-14

MyEngineDLLAccess class, 4-22

MyEngineDotNetAccess class, 4-24

MyEngineWrapper class, 4-26

completion, 4-29

MyModelLinkableComponent class, 4-28

## N

Non-model components, 5-3

NUnit software, 4-42

installing, 4-21

## O

Object oriented programming, 1-14

Objectives of OpenMI, 1-5

Objects, 1-14

creation, 1-19

destruction, 1-19

OMI files, 1-32, 3-5, 4-49

structure, 4-50

OmiEd, 2-39, 3-39

adding triggers, 2-45

configuring connections, 2-44

connecting models, 2-43

running compositions, 2-46

starting, 2-40

OOP, 1-14

OpenMI, 1-3

benefits, 1-6

configuration editor, 2-39

objectives, 1-5

OpenMI architecture, 1-37

OpenMI compliance, 1-9, 1-37, 4-4

OpenMI compliant systems, 3-3

OpenMI components

locating, 3-5

OpenMI DataMonitor, 5-20

- OpenMI interface, 1-10
- OpenMI Software Development Kit, 1-37
- OpenMI systems, 1-32, 3-4
  - configurable, 1-32, 3-23
  - deploying, 3-31
  - developing, 1-31
  - establishing, 3-7
  - hard-coded, 3-17
  - migrating, 1-34
  - running, 3-31
- Operations of classes, 1-17
- Optimization, 5-28
- Overriding methods, 1-15

## P

- Parent classes, 1-15
- Performance, 1-6
- Performance issues, 4-59
- Persistent storage, 3-25
- Phases for using components, 3-8
- Preparation phase, 3-12
- Prepare method, 3-12
- Procedures, 1-14
- Pull mechanism, 1-26, 2-4

## Q

- Quantities, 2-12
  - flux, 2-20
  - source, 2-36
  - target, 2-36

## R

- Report engines, 5-12
- Request-reply mechanism, 1-26
- Running compositions, 2-46
- Running systems, 3-31

## S

- Scalars, 2-11
- Sequence diagrams, 1-17
- SetValues method, 4-36
- Simple River example, 4-6
  - migration, 4-31
  - wrapper, 4-32
- SOBEK migration, 4-57
- Source quantity, 2-36
- Spreadsheets (OpenMI compliance), 5-8
- State of objects, 1-14

- Structured grids, 2-15
- Subclasses, 1-15
- Subroutines, 1-14
- Superclasses, 1-15
- System processes, 4-61

## T

- Target quantity, 2-36
- Terminology, 1-9
- Test software, 4-42
- Testing units, 4-42
- Triggers, 2-45, 3-13

## U

- UML, 1-16
- UML diagrams
  - advanced controllers, 5-34
- Unified Modelling Language, 1-16
- Unit conversions, 1-29
- Unit testing, 4-42
- Units, 2-12
- Use cases, 1-8
  - migration, 4-8
- User interfaces, 1-9

## V

- Validation (configuration), 3-25
- Values, 2-11
- Variables, 1-14
- Vectors, 2-11
- Visual tools, 3-37
  - building, 3-38
- Visual Tools for Office, 5-8
- Visualization, 5-19

## W

- Water Framework Directive, 1-4
- WFD, 1-4
- Whole catchment modelling, 1-4
- Wrapping, 4-13
- Wrapping pattern, 4-14

## X

- XML files, 3-26