OpenMI Association Technical Committee Meeting no 29

Participants:

Rob Knapen, Alterra, Wageningen UR (Rob.Knapen@wur.nl)
Standa Vanecek, DHI (s.vanecek@dhi.cz)
Adrian Harper, Wallingford Software (adrian.harper@wallingfordsoftware.com)
Stef Hummel, Deltares (stef.hummel@deltares.nl)
Unknown User (don), Deltares (gennadii.donchyts@deltares.nl)
Jesper Grooss, DHI (jgr@dhigroup.com)

Table of contents

- Participants:
- Table of contents
- Extendable OpenMI 2
  - Contributors
  - Tasks
    - Design Tasks
    - Code Tasks
    - Documentation Tasks
  - Source Code Changes
    - Main interface changes
    - Alternative way to define the extensions
    - Parallel Computing Extension
    - OGC Element Set Extension
  - Discussions
    - Naming of new interfaces
    - Add property to indicate lengths of the dimensions to IValueSet
    - Add ValueType to IValueSet
    - Get/set multiple values from IValueSet
    - IExchangeItem event
    - Namespaces for interfaces
    - Other issues

Extendable OpenMI 2

This is a discussion page on making the OpenMI 2 Standard more extendable, which was discussed during the Skype conference call. Decisions made during the meeting are marked as 👍 or 👎, with additional comments in italics where needed. You can find these (mostly) in the discussion section at the end of the page.

The general idea is to include a few base interfaces and separate the core OpenMI aspects from the time and space dimensional aspects. In OpenMI 1 both are combined in the interfaces making it difficult to create extensions that allow use of OpenMI in domains that have other dimensionality requirements. Another goal is to allow postponing of some of the more difficult and time consuming functionality (to define, build and test) like support for parallel computing and OGC compliant element sets, to OATC approved extensions to the basic OpenMI 2 Standard.

Contributors

List of OATC members who can contribute and amount of time available until July 2nd.

- Stef: 2-3 days, preferred focus is documentation (+ FRAMES work)
- Rob: 4-5 days, preferred focus is on Java source code (+ FRAMES work)
- Jesper: 4-5 days
- Peter: 4-5 days, preferred focus is on documentation (code & stand alone)
- Standa?
- Gena?
- Adrian?

Tasks
Overview of tasks required to add extendability to the current version of OpenMI 2. For each task please specify type of work, amount (hours), responsible person and deadline.

**Design Tasks**

Deadline **June 18**: All to help finalize the design of the extendable version by contributing to this WIKI page and to the Skype meeting scheduled for Friday 18, 9:00-10:00am.

**Code Tasks**

All work to be done in the SourceForge OpenMI-2.0.0-extendable branch: [http://openmi.svn.sourceforge.net/viewvc/openmi/branches/OpenMI-2.0.0-extendable/](http://openmi.svn.sourceforge.net/viewvc/openmi/branches/OpenMI-2.0.0-extendable/)

**Meeting Decision**: Code will be moved to the main trunk folder (Jesper).

<table>
<thead>
<tr>
<th>Component</th>
<th>Update?</th>
<th>update author</th>
<th>author deadline</th>
<th>reviewer</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>C# OpenMI 2 Standard</td>
<td>y/n</td>
<td>who</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>C# OpenMI 2 SDK</td>
<td>y/n</td>
<td>who</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>C# OpenMI 2 GUI</td>
<td>y/n</td>
<td>who</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>Samples to test extendability?</td>
<td>y/n</td>
<td>who</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>FRAMES Extension Concept</td>
<td>y/n</td>
<td>Stef</td>
<td>28. June</td>
<td>Rob</td>
<td>2. July</td>
</tr>
</tbody>
</table>

**Documentation Tasks**

<table>
<thead>
<tr>
<th>Document</th>
<th>Update?</th>
<th>update author</th>
<th>author deadline</th>
<th>reviewer</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>What’s new in OpenMI 2.0</td>
<td>yes</td>
<td>Stef, Peter</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>OpenMI in a nutshell</td>
<td>yes</td>
<td>Adrian, Rob</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>The OpenMI Standard2 interface specification</td>
<td>yes</td>
<td>Stef, Peter</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>How to download the most recent source code</td>
<td>no</td>
<td>nobody</td>
<td>28. June</td>
<td>nobody</td>
<td>2. July</td>
</tr>
<tr>
<td>How to upgrade from version 1.4 IEngine</td>
<td>y/n</td>
<td>Jesper</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>How to upgrade from version 1.4 using the upwards compatible wrapper</td>
<td>y/n</td>
<td>Jesper</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>How to get started with OpenMI 2.x and Java</td>
<td>yes</td>
<td>Rob</td>
<td>28. June</td>
<td>nobody</td>
<td>2. July</td>
</tr>
<tr>
<td>How to turn an ASCII file reader into a Linkable Component 2.0</td>
<td>yes</td>
<td>Peter</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
<tr>
<td>How to link models with different grids (spatial mapping)</td>
<td>yes</td>
<td>Peter</td>
<td>28. June</td>
<td>who</td>
<td>2. July</td>
</tr>
</tbody>
</table>

**Source Code Changes**

**Main interface changes**

Create base versions of all interfaces, and extend that base with a number of specific extensions.

The base part goes into the root Standard namespace, and contains:

* IBaseLinkableComponent
* IBaseExchangeItem
* IBaseInput
* IBaseOutput
* IBaseAdaptedOutput
* IBaseValueSet

An extension, like the TimeSpace extension that the 1.4 standard supports, can go into a separate namespace, alike OpenMI.Standard.TimeSpace. The interfaces there extend the base interfaces, and add TimeSpace specific functionality to the interfaces. Example, the TimeSpace linkable component will provide the time extent within which the component can provide data, example:
public interface ITimeSpaceLinkableComponent : IBaseLinkableComponent {
    ITime TimeExtent { get }
    ...

}

A TimeSpace namespace would contain the following interfaces:

ITimeSpaceLinkableComponent : IBaseLinkableComponent {
    [time-space specific stuff]
}

ITimeSpaceExchangeItem : IBaseExchangeItem {
    [time-space specific stuff]
}

ITimeSpaceInput : IBaseInput {
    [time-space specific stuff]
}

ITimeSpaceOutput : IBaseOutput {
    [time-space specific stuff]
}

ITimeSpaceAdaptedOutput : IBaseAdaptedOutput {
    [time-space specific stuff]
}

ITimeSpaceValueSet : IBaseValueSet {
    [time-space specific stuff]
}

In order to preserve names from 1.4 standard, we could rename above, removing the TimeSpace part of the name.

With this in place support for e.g. FRAMES Dictionaries could be built as derived interfaces as for the TimeSpace version (IDictionaryLinkableComponent : IBaseLinkableComponent etc.). AdaptedOutputs have to be provided to allow linking Dictionary based with Time/Space based linkable components.

**Alternative way to define the extensions**

A TimeSpace extension could also be defined by first adding a number of extension interfaces that supplies the new functionality, but not extends the base classes, and on top of that a number of interfaces that extends the base and the extension. I.e.

ITimeSpaceLinkableComponentExtension {
    [time-space specific stuff]
}

ITimeSpaceExchangeItemExtension {
    [time-space specific stuff]
}

ITimeSpaceInputExtension {
    [time-space specific stuff]
}

ITimeSpaceOutputExtension {
    [time-space specific stuff]
}

ITimeSpaceAdaptedOutputExtension {
    [time-space specific stuff]
}

ITimeSpaceValueSetExtension {
    [time-space specific stuff]
}

The extension versions does not extend the base version. The linkable components are:

ITimeSpaceLinkableComponent : IBaseLinkableComponent, ITimeSpaceLinkableComponentExtension {
    /*empty*/
}

ITimeSpaceExchangeItem : IBaseExchangeItem, ITimeSpaceExchangeItemExtension {
    /*empty*/
}

ITimeSpaceInput : IBaseInput, ITimeSpaceInputExtension {
    /*empty*/
}

ITimeSpaceOutput : IBaseOutput, ITimeSpaceOutputExtension {
    /*empty*/
}

ITimeSpaceAdaptedOutput : IBaseAdaptedOutput, ITimeSpaceAdaptedOutputExtension {
    /*empty*/
}

ITimeSpaceValueSet : IBaseValueSet, ITimeSpaceValueSetExtension {
    /*empty*/
}

Hence, if a component is to provide exchange items from more than one extension, e.g. as well time-space exchange items as dictionary exchange items, it could be declared as:

MyDictionaryTimeSpaceComponent : IBaseLinkableComponent, ITimeSpaceLinkableComponentExtension,
IDictionaryLinkableComponentExtension {
    ...
}

In the first approach, this would be implemented as

MyDictionaryTimeSpaceComponent : ITimeSpaceLinkableComponent, IDictionaryLinkableComponent {
    ...
}

(Jesper) I am not sure that this actually makes a difference compared with the first approach.

**Parallel Computing Extension**
Probably a controlling component is required. Let’s call it IParallelController. This will have a number of methods available. A component also requires some extra functionality, which can be put in an IParallelComponent, which extends the IBaseLinkableComponent. Together these two can add the functionality required for supporting parallel computations.

**OGC Element Set Extension**

The idea is to have a base spatial axis, and have a number of extensions of this base.

```csharp
public interface ISpatialAxis {
    int NumberOfElements {get;}
    ... [more] ...
}
```

All other spatial axis extend this spatial axis:

```csharp
public interface IElementSet : ISpatialAxis
{ ...[all methods from the current IElementSet] ...}

public interface ISpatial2DGrid : ISpatialAxis
{
    int XCount { get; }
    double X0 { get; }
    double Dx { get; }
    int YCount { get; }
    double Y0 { get; }
    double DY { get; }
    double Orientation { get; }
}

public interface ISpatialLine : ISpatialAxis
{
    int Count { get; }
    double[] X { get; } // all x-coordinates
    double[] Y { get; } // all y-coordinates
    double[] Z { get; } // all z-coordinates, null if Z is not available
    double[] M { get; } // all m-coordinates, null if M is not available
}
```

There are many variations of how to implement the spatial axis. An alternative of the line definition is:

```csharp
public interface ICoordinate
{
    double X { get; }
    double Y { get; }
    double Z { get; }
    double M { get; }
}

public interface ISpatialLine : ISpatialAxis
{
    int Count { get; }
    ICoordinate GetCoordinate(int i);
    bool HasZ { get; }
    bool HasM { get; }
}
```

which is closer to the OGC feature definitions.

**Discussions**

**Naming of new interfaces**
To provide "name" compatibility with version 1.4 and articles written for IEMS, there are two alternatives:

- Rename TimeSpace versions, leaving out the TimeSpace part of the name.
- Adding empty interfaces.

Adding empty interfaces would be:

```csharp
ILinkableComponent : ITimeSpaceLinkableComponent {}
IExchangeItem : ITimeSpaceExchangeItem {}
IInput : ITimeSpaceInput {}
IOutput : ITimeSpaceOutput {}
IAdaptedOutput : ITimeSpaceAdaptedOutput {}
IValueSet : ITimeSpaceValueSet {}
```

And ILinkableComponent would be an empty interface provided for backward compatibility with OpenMI 1 and for ease of development. We have to decide if the advantages outweigh adding a number of empty interfaces. Maybe we can add them but put them in a "openmi1" extension (perhaps even marked as deprecated).

Adding empty interfaces has some unwanted side-effects:

- Two interfaces are meant to do the same.
- A ILinkableComponent is a ITimeSpaceLinkableComponent, but the reverse is not the case.
- The IInput Provider is still an ITimeSpaceOutput, if all interfaces are empty, hence full name compatibility will not be available anyway.

(Jesper) I would recommend to keep the TimeSpace name there, not adding any empty interfaces. I would also prefer to remove the Base part of the name from the base interfaces, i.e. IBaseLinkableComponent to just ILinkableComponent.

Meeting Decision: It was decided to go with the IBase... and ITimeSpace... interfaces names, and not have for example an empty ILinkableComponent interface. Although this requires more work on updating the documentation. Whether to use inheritance hierarchy or multi inheritance for defining the extension interfaces is still under discussion.

Add property to indicate lengths of the dimensions to IValueSet

(Rob) Not sure what the lengths of the dimensions would indicate and why you would have to specify them all the time?

(Gena) The lengths of the dimensions

```csharp
int[] dimensionSizes = { timeSet.Count, elementSet.Count }
```

is required in order to implement code which knows only about IValueSet, otherwise IValueSet is useless without information about specialized implementation. Once we have instance of the IValueSet - we can call GetValues / SetValues without need to know about specific implementation. In the current version you are forced to extract number of times and number of elements from the specialized version, and only after that it is possible to call GetValue / SetValue.

(Jesper) I am not quite sure that an int[] will be able to describe the data correctly. As I understand the FRAMES stuff, then a dimension may vary in size depending on another dimension, i.e. assuming the underlying data is of the form (simulation-index, quantity-index, time-step-index):

```csharp
double[][][] values
```

then

```csharp
values[0][0].Length
```

need not equal

```csharp
values[1][0].length
```

- it may describe two different simulations that may not include the same time-steps and even the same quantities. We will at least have to check that with the FRAMES people...

But I believe without these size, the IValueSet is not very useful in its general form, so it would be nice to have some indications of the sizes.

(Gena) Ah, then I misunderstood requirements a little bit. Then it makes sense to have:
int Rank { get; } - how many dimensions are possible
int GetLength(int[] indices) - actual length of the dimension with specific indices

We can still have

int[] MaxLengths { get; }

...but it can be also obtained from the above.

(Rob) I have added the following methods to the IBaseValueSet (Java):

```java
/**
 * Returns the number of possible indexes (dimensions) for the value set.
 * @return number of indexes, zero based
 */
public int getNumberOfIndexes();

/**
 * Returns the length (max index count) of the dimension specified by the
 * given indexes.
 * @param indexes of the dimension to get the length of
 * @return length of the specified dimension
 */
public int getIndexCount(int[] indexes);
```

**Meeting Decision:** All agreed to adding this to the IValueSet. C# code will be updated following what was already added to the Java code.

**Add ValueType to IValueSet**

(Rob) Would be a good thing, I agree (has been discussed before).

(Jesper) But you have that already from the exchangeItem, is it not a bit redundant to have it in the ValueSet as well?

(Gena) But it is not accessible when you have an instance of the IValueSet. The question is if we want to know type of the values stored in the IValueSet when we get instance of it or not?

It should be possible to know at least value type without ExchangeItem.ValueDefinition.ValueType, otherwise we will be forced to always pass ExchangeItem when IValueSet is used.

(Rob) Agree, let's add only ValueType (not ValueDefinition) to avoid returning to old discussions 😊👍

(Jesper) But then you do not know the size of it either, hence you would need the spatial definition and time definition as well? I think a better solution would be to include the ExchangeItem within the value set, if this kind of information is required. 😐

**Meeting Decision:** All agreed to the solution of adding only the ValueType to the IValueSet. Adding a reference to the ExchangeItem would force the developer to always create copies of the ExchangeItem since it is only a reference that can be updated.

**Get/set multiple values from IValueSet**

(Gena) Make it possible to set/get multiple values instead of working with a single value, otherwise performance will be extremely bad, checking all indices [] on every call is not that nice and will make OpenMI perform terrible.

(Rob) Perhaps we could leave this to specialised high performance versions of IValueSet? 👍

(Gena) The use of interfaces without methods / properties is a very bad code smell AFAIK. If we introduce IValueSet and define it as a generic way to access values - it should provide way to do so.

(Jesper) In principle I agree with Gena. However, the methods needs to be "implementable" in all specialized cases. If they are not, then they are better left to their specializations. I did not put much effort into the general IValueSet, so it needs a review...

(Gena) 2 more methods to implement in a new specialization is not much. I can't imagine that implementing a new specialization is something to be done every day. There will be 2-5 max. But it will make it possible to use OpenMI in high-performance scenarios without knowing details about specialized implementation.
Alternatively these can be implemented once and for all as C# extensions, which will be able to utilize high performance implementations, but does not require to be a part of the public interface.

**Meeting Decision:** All agreed to leave this out of the IValueSet for OpenMI 2.

### IExchangeItem event

(Gena) What is our goal with ItemChanged event in IExchangeItem? Do we want to get this event fired only when values of the exchange item are changed? If yes - then let’s rename it to ValuesChanged and add indices to it (indicating which values were changed). That should make it much more useful.

(Rob) Let’s keep the event for IExchangeItem as simple as possible and not add the burden here of passing around (lots of) indices.

(Gena) I can agree to this, but then only rename it to ValuesChanged, then we will know that Values of ExchangeItem have been changed. Now the name of the event is far too abstract.

(Jesper) Or ValuesUpdated - if that would give the impression of that the values may not actually have changed? No big deal though...

(Gena) Agree, it is minor, but for new users it may save time when the name is more intuitive.

(Rob) I have updated the method name in IBaseExchangeItem as follows:

```csharp
/**
 * Returns an Observable to be used to receive notifications from the
 * exchange item about content changes. This notification is sent when
 * the exchange item values have been updated. It is the responsibility
 * of the observer to figure out which actual values have changed and
 * how to act on these changes.
 * The argument passed in the Observer update call must be an instance of
 * the ExchangeItemValuesUpdatedEventArgs class.
 * @return Observable for receiving exchange item change notifications
 */
public Observable getValuesUpdatedObservable();
```

Also renamed the event args class to: ExchangeItemValuesUpdatedEventArgs.

**Meeting Decision:** All agreed to this change, the C# code will be updated accordingly. It was noted that some additional events / notifications should be added to the ITimeSpaceExchangeItem for changes to the time set and element set.

### Namespaces for interfaces

(Rob) I prefer a namespace split, e.g. in openmi.standard.base and openmi.standard.timespace packages. Provides structure and leaves room for the future.

(Gena) ... then let’s move ITime and IElementSet into it and keep top-level interfaces strictly time/space independent.

(Jesper) I think the way that Rob did it is quite nice, though whether the base classes should be moved to a base namespace or left in the root, is not that important for me. The extension name space is at least not very nice, it should be renamed to TimeSpace or similar.

(Jesper) I would then probably make a spatial/geometry namespace or similar, that holds the ISpatialAxis, IElementSet, and the future ISpatialLine or whatever types of geometries that are required to be supported. Because most likely the TimeSpace version is not the only type of component that requires a spatial definition.

**Meeting Decision:** Agreed to create the timespace namespace and put relevant interfaces there. Base interfaces are to be put in the root (standard) namespace.

### Other issues

- (Peter) What happens if MyDictionaryComponent gets linked to MyLinkableComponent? Does one class have to derive from both extensions? (MyLinkableDictionaryComponent : IBaseLinkableComponent, IDictionaryComponent, ExtensionTimeSpace) Is there more than we can properly think through in three weeks?

(Jesper) It does not have to extend both specializations, on contrary, the intension is that it needs not to. This is to be handled by adapters. So if you have an ITimeSpaceLinkableComponent that you want to link with an IDictionaryLinkableComponent, i.e. you want to connect an ITimeSpaceInput to an IDictionaryOutput. Then there must be a decorator for the IDictionaryOutput that converts it to an ITimeSpaceInput, i.e.
public class WrapDictionaryInTimeSpaceAdaptedOutput : ITimeSpaceAdaptedOutput
{
    public Adaptee { get { return (_myDictionaryOutput); } };
}

That must be provided by the DictionaryComponent, or by 3rd party.

- When is a component OpenMI 2.0 compliant:
  - when it implements IBaseLinkableComponent or
  - when it implements one of the official extensions, e.g.
    ILinkableComponent : IBaseLinkableComponent, ITIMEspaceLinkableComponent

(Jesper) When it implements the IBaseLinkableComponent, then it is OpenMI 2.0 compliant. The rest is in principle up to adaptors. We could consider to require that components also specify which extensions it supports, i.e. a 2.0 TimeSpace component is base compliant and time-space-extension compliant.

Meeting Decision: Adrian pointed out that with this support for extensions we should be more clear about what compliancy means. Even with OpenMI 1.4 it is not clear to a user what he can expect when a model is OpenMI compliant. We agreed to work on a better definition of OpenMI compliance and add it to the documentation.