10. Component Implementation Techniques

10.2 Common implementation interfaces for Managed Code

In order to permit a component developed by one organization to be used by simulation software from other organizations, the following interfaces must be provided by protocol-compliant components implemented using managed code under the Microsoft .net or Mono frameworks. These interfaces should be regarded as a prescription for protocol implementations on multiple operating systems.

10.2.1 Main namespace and interface class

For the CMP compliant simulation engine to communicate to any CMP component it is required that each component includes a common namespace and class name so that reflection can be used to interact with the component.

The class library will include a class containing the required component functions. From this class a child class can be declared with a standard name. Below is an example:

```
using System;
using CMPServices;  //for definition of MessageFromLogic
using output;       //for definition of TextOutInstance


namespace CMPComp
{
    //=========================================================================
    /// <summary>
    /// Standard interface to the CMP engine. This class must be derived from the
    /// user's custom model class.
    /// </summary>
    //=========================================================================
    public class TComponentInstance : TextOutInstance
    {
        //=========================================================================
        /// <summary>
        /// Main constructor for the component class.
        /// </summary>
        /// <param name="compID">The component ID that has been allocated for
        /// the component.</param>
        /// <param name="parentCompID">ID of the parent component.</param>
        /// <param name="messageCallback">Delegate function for callback into
        /// the engine.</param>
        //=========================================================================
        public TComponentInstance(uint compID, uint parentCompID, MessageFromLogic messageCallback)
            : base(compID, parentCompID, messageCallback)
        {

        }
    }
}
```

The required namespace is **CMPComp**. The required class name is **TComponentInstance**. Using these standard names enables the engine to find the interface for the component logic in each component class library.
In the above example, the main component logic is found in **TextOutInstance**. This class needs to include the public functions as shown in the next section.

10.2.2. Interface for simulation design and construction

Protocol-compliant components implemented as class libraries must implement the following functions.

(a) Component description interface

```
public string description(string context)
```

Returns the string containing the component description as set out in section 9.2

The *context* parameter is designed to allow polymorphic components specify a description based on details within the character string. The contents of this string are implementation specific and if not used it should be an empty string.

(b) Initialisation script interface

As described in section 7, the initialisation information provided to a component in the SDML document (the “initialisation script” for each component) is in a format that is not known to the rest of the simulation. These routines form an interface for building and parsing the component-specific initialisation scripts from name, type and value data.

Initialisation scripts are understood to be composed only of initialised properties, each of which is composed of a name, a type and a value. Note the use of character strings to denote multiple instances of initialisation scripts for a particular component.

```
public void createInitScript(string sScriptName)
```

Create a new initialisation script.

```
public void deleteInitScript(string sScriptName)
```

Delete a previously created script.

```
public void textToInitScript(string sScriptName,
                             string sScriptText)
```

Sets the contents of an initialisation script using a component specific format. The properties in this text will be appended to the list of any existing ones in this script.

```
public String textFromInitScript(string sScriptName)
```

Returns the contents of an initialisation script.

```
public void valueToInitScript(string sScriptName,
                              string sPropertyName,
                              string sTypeDDML,
                              Byte[] pValueData)
```

Sets an initial value within an initialisation script. `sTypeDDML` denotes the type using DDML; `pValueData` is value data laid out as for message value data (section 9.1).

```
public int valueFromInitScript(string sScriptName,
                               string sPropertyName,
                               ref string sTypeDDML,
                               ref Byte[] pValueData)
```

Returns an initial value from an initialisation script. `sTypeDDML` denotes the type using DDML; `pValueData` is value data laid out as for message value data (section 9.1). The return value is the number of bytes in the pValueData array.

(c) Name of the wrapper DLL

As described in section 10.1.2, each component DLL may use a “wrapper” DLL that implements an interface, between the simulation software and the component DLL, for passing messages.

```
public string wrapperDll()

e.g.

return
System.Reflection.Assembly.GetAssembly(this.GetType())
.Location;
```

Returns the name of the “wrapper” DLL. If the returned value is zero length, the simulation assumes that the component DLL acts as the “wrapper”. It is valid to return the name of the component dll.

10.2.3. Component wrapper DLLs

Refer to section 10.1.2

Providing communications between the simulation engine and the logic components.

Sending messages to the component class library is accomplished using a defined function name in the main interface class. A callback facility is provided in the simulation engine for messages sent to it from any logic component.

The component class library must export the following routines:

```
public TComponentInstance(uint compID,
                          uint parentCompID,
          MessageFromLogic messageCallback)
          : base(compID,
              parentCompID,
              messageCallback)
```

Creates an instance of a component.

`compID`	is the instance's registration ID (input);
`parentCompID`	is the registration ID of the system that manages the component (input);
messageCallback	delegate function for the entry into the engine for messages sent back into the engine.

The delegate function is defined as:

```
public delegate void
MessageFromLogic(TMsgHeader inMsg);
```

```
public void handleMessage(TMsgHeader msg)
```

Passes a message to the component from the simulation engine.

- Because multiple simulations may be running concurrently and using the same wrapper DLL, the component registration ID (which is only unique within a single simulation) is insufficient as a unique instance identifier.
- The definition of `TMsgHeader` follows the layout of the message header in section 9.1 and is shown below.

```
[StructLayout(LayoutKind.Explicit, Pack = 1)]
public struct TMsgHeader
{
    [FieldOffset(0)]
    public UInt16 version;
    [FieldOffset(2)]
    public UInt16 msgType;
    [FieldOffset(4)]
    public UInt32 from;
    [FieldOffset(8)]
    public UInt32 to;
    [FieldOffset(12)]
    public UInt32 msgID;
    [FieldOffset(16)]
    public UInt32 toAck;
    [FieldOffset(20)]
    public UInt32 nDataBytes;
    [FieldOffset(24)]
    public byte[] dataPtr;
}
```