



SBW C# / .NET Bindings

Frank Bergmann

Version 1.2, 2005-09-21

This document gives a short overview of the C# / .NET language bindings for the Systems Biology Workbench. The language bindings allow any .NET project to access other SBW modules and / or to implement a new SBW module.

The language bindings have been tested in various modules (Bifurcation Discovery Tool¹, Oscill8², SimDriverNET and a new CSharpSimulator³) and found to be stable. We use the bindings with the following frameworks:

- .NET Framework 1.1
- .NET Framework 2.0 (Beta 2)
- Mono 1.1.8

Rather than describing each individual function, this document will present the features that seem noteworthy. For a generated API reference see also:

<http://public.kgi.edu/~fbergman/files/sbw/csharp/index.html>

¹ <http://public.kgi.edu/~fbergman/SBW.html>, Vijay Chickarmane et al.

² <http://sourceforge.net/projects/oscill8/>, Emery Conrad

³ <http://sourceforge.net/projects/jdesigner>, Herbert M. Sauro

Data types

SBW supports all basic data types along with one dimensional, two dimensional regular⁴ arrays and lists. Lists in SBW are recursively defined structures that can contain any SBW data type. The following list shows how these data types are mapped for C# / .NET.

| SBW Data type | C# Data type |
|------------------|-------------------------|
| boolean | bool |
| int | int |
| double | double |
| string | string |
| complex | SBWComplex ⁵ |
| array | array ⁶ |
| { } ⁷ | ArrayList |

Table 1: Mapping of data types

Since the list is the most complex data type here are some examples on how to use them. For further reference see also the MSDN documentation on System.Collections.ArrayList.

```
// If you are writing a SBW module and wonder how to
// add elements into an array list object ...
ArrayList list = new ArrayList();
list.Add(3); // adding an int
list.Add(new double[,] { // a 2d double
    { 1,2,3 },
    { 3,4,5 } } );
list.Add("test"); // and a string

// suppose in a previous call an array list object 'list'
// was returned containing: an integer, a 2D double array
// and a string

int      nInt      = (int)      list[0];
double[,] oDouble2D = (double[,]) list[1];
string   sString   = (string)   list[2];

// of course array list implements IList, ICollection
// and IEnumerable and so can be interrogated through
// these interfaces as well.
```

Code 1: Using ArrayLists

⁴ i.e.: no jagged arrays, but arrays with equal 2nd dimension.

⁵ SBWComplex is a basic class, allowing the user to get hold of real and imaginary part of a complex number. If oComplex is a complex number, use oComplex.Real to get/set the real part, and oComplex.Imag for the imaginary part.

⁶ Where 2 dimensional arrays are represented as [,] instead of [[]].

⁷ { } is the representation for lists in SBW.

The decision to use regular arrays instead of jagged arrays was made because they resemble more closely the SBW data type. Still some people might want to use jagged arrays instead. For this reason a utility function was implemented to convert the type. It will convert a [,] array into a [][] array. This function is:

```
SBW.HighLevel.convertArray()
```

It takes an object as an argument. If that object is a 2D array of type [,] it will return that array as []. If that object is a 2D array of type [][] it will return that array as [,] otherwise the object will be returned.

In most cases the language binding will perform the proper conversion for you. The only exception is if you extract a two dimensional array from an ArrayList. In that case you will find this array represented as [,].

Accessing a known SBW Module

This section describes how to access a known SBW module. It is also possible to use a Visual Studio 2003 Add-in to automatically wrap a SBW module. This wrapper will then be added to your current project. For more information about the Add-in see:

http://public.kgi.edu/~fbergman/sbw_c.htm

If you are not using the Add-in, a call to a SBW module could look like this:

```
using SBW; // using the SBW C# bindings (import the namespace)

...
// getting the "TestModule"
Module module = new Module("TestModule");
// getting the TestService
Service service = module.getService("TestService");
// getting the Method
Method method = service.getMethod("double Sin(double)");
// calling the Method
Console.WriteLine("Sin(3.14) is : " + method.Call(3.14));

// disconnect from SBW
LowLevel.SBWDisconnect();
```

Code 2: Calling a known Module

The oMethod.Call() function takes any number of arguments. Although they will have to be the SBW types described above. (With the exception of 2 dimensional arrays which can also be given in the form []). The type will be detected automatically. To call a method that takes a string, an integer and a double value could be invoked like this:

```
method.Call("string", 1, 1.0);
```

Implementing a simple SBW Module (Console Application)

Suppose we have a C# class and want to turn this class into a SBW module. The easiest way to do that is to just pass the type of the class to the SBW.DefaultModule together with the module name and the service name. This will create an instance of the class, register the module for a shutdown event (the application will terminate upon request of the Broker/or another Module) and start the module in its own event loop. Meaning the call will not return, until the module is shut down.

```
using SBW;
namespace CSharpTest
{
    public class ProvideService
    {
        // default constructor
        public ProvideService() { }
        // help provides the SBW help for the method
        [Help("Calculates the sine of the given value")]
        public double Sin(double value)
        {
            return Math.Sin(value);
        }
        [Ignore()]
        [STAThread]
        public static void Main(string[] args)
        {
            DefaultModule.Run(
                typeof(ProvideService), // the Type of the class
                "TestModule",           // the module name
                "TestService",          // the service name
                args);                  // the command line arguments
        }
    }
}
```

Code 3: Implementing a SBW Module (simple)

The `SBW.DefaultModule` is a convenient short form for the following code snippet:

```

// ignore() is used so that it will not be given as SBW method
[Ignore()]
[STAThread]
public static void Main(string[] args)
{
    // create a new service
    ProvideService oService = new ProvideService();
    // create an object reference to it
    Object oServiceObject = oService;
    // create SBW module
    ModuleImplementation oModule = new ModuleImplementation(
        "TestModule", // define the ModuleName
        "C# test module", // define ModuleDisplayName
        LowLevel.ModuleManagementType.UniqueModule,
        // define management type
        "small test module written in C#"); // help string

    // add the test service to it
    oModule.addService(
        "TestService", // define service name
        "TestService", // define display name
        "/test", // define service category
        "Test service written in C#", // help on the service
        ref oServiceObject); // only public and public
        // static methods will be provided

    oModule.run(args);
}

```

Code 4: Creating a SBW module (Extended form)

This way of creating a SBW module can be of advantage if more than one service should be provided by a SBW module. It also allows using events. (See also: page 7, SBW Module Events). In this case it is up to the user to register for the shutdown event.

SBW Attributes

Let us turn to the Attributes in this example. There are two attributes implemented. The first one is `[Help(string)]` which allows one to specify a help string for a SBW method. The other one is `[Ignore()]` which tells the object analyzer not to wrap this method into a SBW method. Note that only public and public static methods are automatically wrapped. (And yes, that includes properties. Hence it is advisable, to place `[Ignore()]` in front of a property.)

WinForm Applications

The approach outlined previously won't work for WinForm Applications. The reason is that these applications have their own message loop. For these kinds of applications the object representing your service class has to be created first, and then boxed into an "object" and this will be passed on to the SBW library.

The simplest case for such a WinForm Application would be a Main method like:

```
[Ignore()]
[STAThread]
public static void Main(string[] args)
{
    MyForm oForm = new MyForm();
    object oInstance = oForm;
    DefaultModule.EnableServices(
        ref oInstance, // the instance of the class
        "TestModule", // the module name
        "TestService", // the service name
        args); // the command line arguments
    Application.Run(oForm);
}
```

Code 5: Main method for WinForm Applications

Again it is possible to use this code without the `SBW.DefaultModule` (useful to add more services or to consume events). The corresponding Code snippet looks exactly like the one described in Code 4: Creating a SBW module (Extended form). The only difference is that instead of calling

```
“oModule.run(args);”
```

the method

```
“oModule.EnableServices(args);”
```

has to be called. After the call to `“EnableServices()”` the SBW Module will be up and running. All that is left to do is then to start the WinForm using `Application.Run();`

Thread safe calls

In order to allow a SBW call to interact with the GUI process it is important to realize that SBW calls are invoked from a different thread than the GUI process. For a developer this means that he has to add the following lines into each WinForm method that gets called via SBW:

```
private delegate void delegateString(string sParameter);
[Help("This Method is called from SBW")]
public void setString (string sParameter)
{
    if (this.InvokeRequired)
    {
        this.Invoke(
            new delegateString (setString),
            new object[] { sParameter });
    }
    else
    {
        // the logic for this function
        // is placed here
    }
}
```

Code 6: Thread safe Invocation of a WinForm method

The first task is to define a delegate that has the same signature as the method that will be called from SBW. The first thing in the method should be a test, whether an invocation is required. This test is done by testing the property “`this.InvokeRequired`”. If this is not the case the actual logic of the method should be processed. Otherwise the method will be invoked via the delegate. The “`Invoke`” call takes all the arguments of the method. Of course “`Invoke`” can also return the results of the method call.

SBW Module Events

There are four events that a SBW module can listen to:

- `ModuleShutdown`
- `ModuleStartup`
- `OtherModuleShutDown`
- `RegistrationChange`

The **`ModuleShutdown`** event informs the SBW module that it should exit. If the application chooses not to exit it is important to know that from this point on it will be disconnected from SBW. To be a good module, the module should prompt the user to save any changes made (if any) and then quit.

The **`RegistrationChange`** event occurs whenever a new module was registered with SBW or if an existing module unregistered from SBW. This event is useful for applications that discover other modules based on their categories.

OtherModuleShutDown informs the SBW module that another SBW module just terminated. Each time a SBW module uses services from another SBW module it is important to register for this event. One example could be an application that uses a SBW module to simulate SBML models. Should the simulator terminate before the module is done with it will be notified of the shutdown via this Event. The event handler for this event looks like this:

```
static void OtherModuleShutDown(object sender, ModuleEventArgs e)
```

The `ModuleEventArgs` object contains a property “`ModuleId`” that identifies the module that just terminated.

ModuleStartup informs the module that a new SBW module has been started up and is available. Again the event handler is given a `ModuleEventArgs` object containing the module identifier of the new module.

In order to register for any of these events a `ModuleImplementation` object is required. This object is created as described previously in Code 4: Creating a SBW module (Extended form). An example of adding a module shutdown listener to the implementation object would look like:

```
oModule.ModuleShutdown+=new EventHandler(oModule_ModuleShutdown);
```

additionally a handler method has to be implemented in the class that could look like this:

```
private static void oModule_ModuleShutdown(object sender, EventArgs e)
{
    Environment.Exit(0);
    // for WinForm Applications this should be Application.Exit();
}
```

Exception Handling

All Exceptions that are thrown by SBW are derived from `SBWException`. This exception class inherits from `System.Exception`, and can be used like any other `Exception`. Since SBW allows specifying an error message along with a detailed error message `SBWException` and all derived classes have not only a `Message` property, but also a `DetailedMessage`.

For applications accessing SBW modules this means that potentially after each call to that module a `SBWException` might be thrown and has to be caught.

Module developers on the other hand should always throw `SBWApplicationException`'s in order for the clients to receive the most amount of information on how to remedy the problem.

A SBW method in C# should follow a pattern like this:

```
[Help("Describe Method, Input and Output values")]
public <type> myMethod(<arguments>)
{
    try
    {
        // test input arguments throw exception
        // if not valid
        ...

        // perform main calculation throw exception
        // if necessary
        ...

        // prior to returning the result check it for
        // validity if possible

        return result;
    }
    catch (SBWException)
    {
        throw;
    }
    catch (Exception e)
    {
        throw new SBWApplicationException(
            "Unexpected Error in myMethod",
            e.Message);
    }
}
```

Code 7: SBW Method pattern

If an application throws Exceptions other than those of type `SBWApplicationException` the binding library will wrap them into `SBWExceptions` as appropriate.