Systems Biology Workbench C++ Programmer's Manual

Michael Hucka, Andrew Finney,Herbert Sauro,
Hamid Bolouri, Frank Bergmann

{frank_bergman,hsauro}@kgi.edu, mhucka@caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA
Keck Graduate Institute
535 Watson Drive, Claremont
CA 91711, USA
http://www.sys-bio.org

Principal Investigators: John Doyle and Hiroaki Kitano

October 21, 2004

Contents

1 Introduction

The aims of this manual are twofold: teaching programmers how to interface C++ applications to the ERATO Systems Biology Workbench (SBW), and providing a reference for the SBW C++ API. This manual complements the overview of the SBW system provided by Hucka et al. (2001a,c, 2002b) and the description by Sauro et al. (2001) of a prototype SBW implementation. This document assumes that the reader is familiar with C++ programming language including the C++ Standard Library particularly those parts derived from the Standard Template Library (STL; Musser and Saini, 1996). Documentation on programming SBW in C (Finney et al., 2001) and Java (Hucka et al., 2002a) is also available.

SBW is a software integration environment that enables applications (potentially running on separate machines) to learn about and communicate with each other. Applications can be written to be providers of software services, or consumers, or both. The SBW communications facilities allow heterogeneous packages to be connected together using a remote procedure call mechanism; this mechanism uses a simple message-passing network protocol and allows either synchronous or asynchronous invocations. The interfaces to SBW are encapsulated in client libraries for different programming languages (currently C, C++, Delphi, Java,MATLAB and Python, with more anticipated), but the protocol is open and small, and developers may implement their own interfaces to the system if they choose. The project is entirely open-source and all specifications and implementations are freely and publicly available.

Frameworks for integrating disparate software packages are certainly not new. Compared to other broker-based integration frameworks, SBW has the following combination of features:

- Free, open-source implementations available for all platforms;
- Availability today for Linux and Windows, with more platforms anticipated in the future;
- Comparatively simple APIs and data exchange protocol;
- Support for major programming and scripting languages and seamless interaction between modules written in different languages;
- No need for a separately-compiled interface definition language (IDL); and
- Resource management performed by underlying services (which means, for example, that there is no exposed object reference counting).

SBW is being developed as part of existing collaborations in the systems biology community with the following software development groups: *BioSpice* (Arkin, 2001), *DBsolve* (Goryanin, 2001; Goryanin et al., 1999), *E-CELL* (Tomita et al., 1999, 2001), *Gepasi* (Mendes, 1997, 2001), *Jarnac* (Sauro and Fell, 1991; Sauro, 2000), *ProMoT/DIVA* (Ginkel et al., 2000), *StochSim* (Bray et al., 2001; Morton-Firth and Bray, 1998), and *Virtual Cell* (Schaff et al., 2000, 2001). These collaborations have already successfully established SBML, the Systems Biology Markup Language (Hucka et al., 2001b), as an important emerging standard in this field.

This document describes the C API for a system which is in active development. This document will be revised as that development proceeds.

2 A Brief Tour of SBW

The primary goal of SBW is to allow the *integration* of software components performing a wide range of tasks common in computational biology, such as simulation, data visualization, optimization, and bifurcation analysis. SBW is not designed to be in the foreground of either the

user's or the programmer's experience, but instead, to allow existing systems biology software packages to easily and transparently access functionality from each other.

In more specific terms, SBW is a computational resource brokerage system. It allows the interactive discovery and use of software resources. In the SBW scheme of things, software resources are independent applications and are called *modules*. A module instance is a running application or process. A module can implement one or more *services*. *Services* are interfaces to the resources inside a module and consist of one or more *methods* (i.e., callable functions).

Broker architectures are relatively common and are considered to be a well-documented software pattern (Buschmann et al., 1996). In SBW, the remote service invocations are implemented using *message passing*, another well-known and proven software technology. Communications in message-passing systems take place as exchanges of structured data bundles—messages—sent from one software entity to another over a channel. Some messages may be requests to perform an action, other messages may be notifications or status reports. Because interactions in a message-passing framework are defined at the level of messages and protocols for their exchange, it is easier to make the framework neutral with respect to implementation languages: modules can be written in any language, as long as they can send, receive and process appropriately-structured messages using agreed-upon conventions. Figure 1 illustrates the overall SBW system organization.

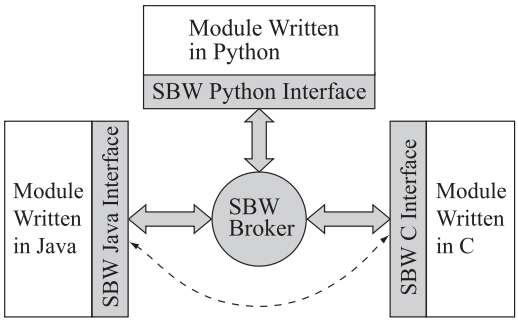

Figure 1: *The overall organization of the Systems Biology Workbench. Gray areas indicate SBW components (libraries and the broker). To individual modules, communications appear to be direct (dotted line), although they actually pass through the broker.*

2.1 Overview of SBW from a Programmer's Perspective

The SBW APIs provide the following facilities:

1. *Dynamic service and module discovery*: The SBW Broker keeps track of modules, services and service categories, and provides facilities for a module to learn about them.
2. *Remote method invocation*: The bread and butter of SBW is enabling one module to invoke a service method in another module. If necessary, the SBW Broker will automatically start an instance of a module whose services are requested.
3. *Data serialization*: Method invocations involve sending messages between modules, with arguments and data packed into message streams. For some languages such as Java, Perl and Python, the SBW library provides proxy objects that hide the message-passing, so that to client programs, remote services appear as local objects whose methods can be invoked like any other object method.
4. *Exception handling*: SBW provides facilities for dealing with exceptional conditions.
5. *Event notification*: Certain events in SBW, such as the startup or shutdown of an instance of a module, are announced to all modules upon their occurrence.
6. *Module, service and method registration*: In order for a module to advertise its services to others, it must first inform the Broker about them. The registration facilities allow a module to record with the Broker the services that the module provides, the command that should be used to start up the module on demand, and other information. The SBW Broker stores this in a disk file, so that the information provided by modules is persistent between start-up and shutdown of the modules and the Broker.

2.1.1 Service Categories

Each service is described by a unique name, a humanly-readable name, and a service category. Services in SBW are categorized hierarchically as a tree in which the leaves are services. Each level in the hierarchy is named and can be uniquely described via a text string very much like directory pathnames in Unix or Windows.

The purpose of this facility is to support a hierarchy of programming interfaces. Each level has an associated, documented interface. Descendants inherit or extend the interface of their parents. Categorizing services in this way allows other applications to find services with known interfaces without having to know about specific modules. New modules, by complying with a given interface, can extend the behavior of existing applications without requiring those applications to be rewritten.

To give an example, one could define a top-level category, "Analysis", with a service interface consisting of one method:

```
void doAnalysis(string SBML)
```

Then one could have a subcategory of "Analysis" called "Analysis/Simulation", with a service interface consisting of two methods:

```
void doAnalysis(String SBML)
void ReRun()
```

Service category names can consist of any sequence of non-control characters excluding '\n' and '/' but including space. Service category strings as used in the API are sequences of one or more service category level names separated by a forward slash ('/') character.

2.1.2 Adapting Applications to Use SBW

We strove to create APIs that provide a natural interface in each of the different languages for which we have implemented libraries so far. By "natural", we mean that it uses a style and features that programmers accustomed to that particular language should find familiar. We hope that application developers will find it relatively easy to introduce SBW interoperability into their software. To give some idea of what is involved, here is a summary of the steps a developer would follow to adapt a particular application for use as a module in SBW:

1. Decide on the services that will be provided by the module to clients.
2. (Optionally) Categorize each service, using as a starting point the existing SBW service hierarchy.
3. For each service, define its methods along with their parameters and return value(s).
4. If necessary, implement the methods as they are intended to be seen by other modules.
5. Add calls in the application's main routine to the SBW module registration methods as described below.
6. Compile the application with the SBW API library (if appropriate for the programming language in use).

The resulting application will be able to run in three modes: a regular, non-SBW mode; a *registration* mode; and a *module* mode. By convention, the registration mode is invoked by starting the module with the command-line argument `-sbwregister`; it tells SBW to contact the Broker once, register the services declared by the module (if the module's author so desires), and exit. The module mode is invoked by starting the module with the command-line argument `-sbwmodule`; it allows the module to run, connecting to SBW and making its services (if any) available to other SBW modules. The non-SBW mode is invoked by not supplying either of these arguments.

3 Foundations of SBW

This section introduces various features of SBW that are exposed through its programming APIs.

3.1 Message Block Types

The arguments and return values of methods on services are encoded in message or data blocks. Message blocks are heterogeneous collections of data using a simple set of data types consisting of 32-bit integers, double-precision IEEE floating-point numbers, strings, bytes, boolean values, arrays and lists.

The array types are essentially homogeneous collections of one type. The list types are heterogeneous collections of data or recursive substructures.

The message block types and their correspondence to Java, C, C++, Python, and Delphi types is shown in table 1.

SBW Signature	**Java**	**C++**	**C**	**Python**	**Delphi**
`string`	`java.lang.String`	`std::string`	`char *`	`string`	`string`
`int`	`int`	`Integer`	`SBWInteger`	*int*	`integer`
`double`	`double`	`Double`	`SBWDouble`	*float*	`double`
`boolean`	`boolean`	`bool`	`SBWBoolean`	*int*	`boolean`
`byte`	`byte`	`unsigned char`	`unsigned char`	`string`	`byte`
array	*array*	`std::vector` [a] or *array*	*array*	`array` [b]	`SBWArray`
list	`java.util.List`	`DataBlockWriter` `DataBlockReader`	`SBWDataBlockWriter *` `SBWDataBlockReader *`	*list*	`SBWList`
`complex`	`SBWComplex`	`std::complex<double>`	`SBWComplex`	`complex`	`TSBWComplex`

Table 1: *Data types supported in SBW and their corresponding programming language types. The "SBW Signature" types are those permitted in service method signatures; they are described in Section 3.3. The italicized names* array *and* list *represent the natural versions of these data types; i.e., in Java, arrays are such things as "`int[]`", "`byte[]`", etc. `Integer` and `SBWInteger` are to defined to be a 32-bit signed integer (the same as the primitive `int` and `integer` types in the other languages). `Double` and `SBWDouble` are defined to be a double-precision floating-point number in IEEE 754 format (the same as the primitive `double` type defined in the other languages).* [a] *In the C++ library, `std::vector` is used for 1-D arrays and raw C++ arrays are used for 2-D arrays.* [b] *In Python, 1-D and 2-D arrays are implemented using the `array` type from the* Numerical Python *package (Ascher et al., 2001).*

3.2 Module Management Types

SBW manages various strategies for managing the lifetimes of module instances. Modules fall into the following categories or types with respect to lifetime management:

3.2.1 Unique

A module instance is started on the first request for a module instance object in the API. All subsequent requests for an instance are returned a reference to the existing module instance.

Typically these modules will provide simple functionality which does not require the module to contain much state information.

3.2.2 Self-Managed

A new module instance is started on every request for a module instance object in the API. Either the module instance itself or the requesting application decide when the broker should disconnect from the module instance. Normally, when the broker disconnect from a module instance, the module instance shuts down.

3.3 Method Signatures

The API uses strings called *Method Signature Strings* to define the names, arguments and return types of service methods. (In C, a subset of Method Signatures, *Argument Lists*, are used for the encoding and decoding of message blocks. The syntax of method signatures is shown in Figure 2.

```
Letter     ::= 'a'..'z','A'..'Z'
Digit      ::= '0'..'9'
Space      ::= ( '\t' | ' ' )+
SName      ::= '_'* Letter ( Letter | Digit | '_' )*
Type       ::= 'int' | 'double' | 'string' | 'boolean' | 'byte' | ArrayType | ListType |complexType
ArrayType  ::= Type Space? '[]'
ListType   ::= '{' Space? ArgList Space? '}'
ArgList    ::= ( Type [Space SName] ( Space? ',' Space? Type [Space SName] )* )?
ReturnType ::= 'void' | Type
VarArgList ::= (Space? ArgList [Space? ',' Space? '...'] Space? ) | Space? '...' Space?
Signature  ::= ReturnType Space SName Space? '(' VarArgList ')'
```

Figure 2: *Permissible syntax of method signatures, defined using the version of EBNF notation (Extended Backus-Naur Form) used in the XML specification (Bray et al., 1998). The meta symbols '(' and ')' group the items they enclose, '[' and ']' signify that the enclosed content is optional, '*' means "zero or more times", and '+' means "one or more times".*

`Signature` is the signature for one method. The `Type` enumeration refers to the different data block types. Table 1 shows the correspondence between these strings and programming language data types. The character sequence '...' indicates that the remainder of the data block is not of a predetermined format; i.e., equivalent to a variable parameter sequence. A `ListType` containing an empty `ListContents` indicates a list of arbitrary length and type.

Here are some examples of method signatures:

```
double f(double x)
```

a method that returns a double given a double value `x`.

```
int f(int[])
```

a method that returns an integer given a one dimensional array of integers.

```
{string name, double value} f(string)
```

a method that returns a string name and double value inside a list given a string argument.

```
{}[] f()
```

a method that returns an array of lists. The method takes no arguments. The content of the lists is undefined.

```
void f(...)
```

a method that returns nothing given a variable argument list.

The following is a method that returns a complex value given two double values:

```
complex f (double, double)
```

3.4 Module Names

Each module has two names: an identification name and a name for display. Module identification names are compared on a case-sensitive basis. The identification name should be given so that it is unlikely to be equal to another module identification name. To achieve, this the following syntax convention is proposed for these identification names: reverse domain name '.' module name. The reverse domain name is derived from a domain name owned or affiliated to by the module developer. The reverse domain name just reverses the sequence of names in the domain name string. An example of a module identification name would be "edu.caltech.gibson".

This scheme for module identification names is voluntary. The API will still work with other naming schemes. The objective is to make module names unique to one machine. A module name has to be combined with location information, e.g. IP address, to construct a module name unique to the world.

3.5 Service Names

Service identification names have the `SName` syntax as defined in Section 3.3. Service identification names are compared on a case sensitive basis. By convention service identification names start with a capital letter.

3.6 Method Names

Method names have the `SName` syntax as defined in Section 3.3. By convention method names start with a lower case letter. Methods signatures are compared using the same scheme as C++ and Java i.e. methods on the same service can have the same name but methods with the same name must have different parameter types. Method names are not compared outside the context of a signature.

4 Accessing the C++ API

The C++ API is defined in the `sbw.h` include file.

4.1 gcc on Linux

To compile SBW under Linux, you need a recent version of the Linux kernel (we use 2.4.x), and a recent version of GCC (we use 2.96-85).

Figure 3 shows a template Makefile that can be used as a starting point for a project Makefile. You may wish to copy this text into a file called `Makefile` in your module source code directory. Three things need to be modified in this file. First, the value of `sbw_root` needs to be changed to the path to the root of the SBW installation directory (the directory that contains the SBW "`lib`", "`include`", etc., directories). Second, the value of `module_name` needs to be changed to the desired name of your compiled module. Finally, the value of `objs` needs to be changed to a list of `.o` files that make up your module.

The template makefile in Figure 3 illustrates the main points about compiling SBW:

- Use the flags `-DLINUX -D_GNU_SOURCE` when compiling either C++ or C source code files.
- Use the `-I` flag to tell the compiler where to find the SBW include files.

```
# Configuration variables.

sbw_root       = /path/to/SBW
module_name    = modulename
objs           = list of .o files

# Common directives -- these are independent of the application.

libdir         = $(sbw_root)/lib
includedir     = $(sbw_root)/include

cflags_regular = -O
cflags_debug   = -g

ldflags_regular = -lpthread -lsbw
ldflags_debug = -g -lpthread -lsbw-debug

default:
       make regular

regular:
       make CFLAGS="$(cflags_regular)" LDFLAGS="$(ldflags_regular)" $(module_name)

debug:
       make CFLAGS="$(cflags_debug)" LDFLAGS="$(ldflags_debug)" $(module_name)

$(module_name): $(objs)
       g++ $(LDFLAGS) -L$(libdir) -Xlinker -rpath -Xlinker $(libdir) \textbackslash{}
           -o $(module_name) $(objs)

.cpp.o:
       g++ -Wall $(CFLAGS) -DLINUX -D_GNU_SOURCE -I. -I$(includedir) -c $<

.c.o:
       gcc -Wall $(CFLAGS) -DLINUX -D_GNU_SOURCE -I. -I$(includedir) -c $<

clean:;
    -rm -f $(objs)
```

Figure 3: *Template makefile for use with new SBW modules.*

- Make sure the the final link step includes the SBW library (using `-lsbw`) and the Pthreads library (using `-lpthread`), and also that the linker records the path to the SBW library so that at run-time, the module will find it.

The template makefile has two directives for building SBW, one for a regular (optimized, non-debug) version and one for a debugging version of the module. These can be invoked simply by running `make` to compile a regular version of the module and `make debug` to compile a debugging version of the module.

4.2 Visual C++ on Windows Platforms: 95, 2000, 98, ME and XP

There are debug (`sbwd`) and release (`sbw`) versions of the SBW library, with `.dll` and `.lib` files for both versions.

Visual C++ projects that use SBW should link with `sbw.lib` or `sbwd.lib`, include `sbw.h` and generate code for the multithreaded DLL version of the VC++ runtime. This can be achieved first by changing the Visual C++ options as follows:

1. Select the “Options...” menu item on the main “Tools” menu.

2. The “Options” dialog box appears.
3. Select the “directories” tab.
4. Select “include files” from the “show directories for” list.
5. Add a new directory: the directory `include` immediately below the SBW installation root directory.
6. Select “library files” from the “show directories for” list.
7. Add a new directory: the directory `lib` immediately below the SBW installation root directory.
8. Select the “OK” button.

When modifying a Visual C++ project to use SBW, perform the following operations:

1. Select the menu item “Settings...” on the “Project” menu.
2. In the drop down list “Settings For:” select “Win32 Debug”.
3. Select the “Link” tab.
4. In the “Object/library modules:” text field append “ sbwd.lib”. Ensure that there is a space between the existing contents of the list and “sbwd.lib”.
5. Select the “C/C++” tab.
6. In the “Category:” drop down list select “Code Generation”.
7. In the “Use run-time library:” drop down list select “Debug Multithreaded DLL”.
8. In the drop down list “Settings For:” select “Win32 Release”.
9. Select the “Link” tab.
10. In the “Object/library modules:” test field append “ sbw.lib”. Ensure that there is a space between the existing contents of the list and “sbw.lib”.
11. Select the “C/C++” tab.
12. In the “Category:” drop down list select “Code Generation”.
13. In the “Use run-time library:” drop down list select “Multithreaded DLL”.
14. Select the “OK” button.

5 Tutorials

In this section, we describe various programming scenarios that use the SBW C++ API. Section 5.1 describes how to implement a module that provides services and section 5.2 describes how to write an application which calls SBW services, in this case those services implemented in section 5.1. You will need to build and register the service provider before running the calling application. Section 5.3 describes how to create a simple menu of services from a service category.

5.1 Implementing a Service

This section describes the complete implementation of a simple module implementing the service called in Section 5.2.

The code for this example is contained in the directory

```
src/tutorials/C++/TrigPlusServerModule
```

This code follows the typical pattern of implementing an executable which can both provide services (module mode) and register those services with the broker (register mode). The module presented in this example provides one service "trig". The interface for this service is as follows:

```
double sin(double)
```

The code is as follows:

```
#include <math.h>
#include "SBW.h"

using namespace SystemsBiologyWorkbench ;

/*
* Trignometry
* this class provides an implementation of the Trignometry service
*/
class Trignometry
{
public :
    /// this method provides the implementation of an SBW method
    /// - in this case the sin method
    /// arguments:  from - module calling this method
    ///             reader - object containing argument data
    /// returns:    object containing result data
    DataBlockWriter sin(Module from, DataBlockReader reader)
    {
        Double arg ;

        // extract double argument from argument data
        reader >> arg ;

        // create result data object, calculate result, store result in result object
        // and return result object
        return DataBlockWriter() << (Double) ::sin(arg);
    }

    /**
     * register methods of this class with a module implementation represented by a
     * given MethodTable
     * argument - table object that provides a simple interface to a module implementation
     */
    static void registerMethods(MethodTable<Trignometry> &table)
    {
        table.addMethod(&Trignometry::sin, "double sin(double)");
    }
};

/*
* module entry point WinMain/main
* On windows using WinMain as an entry point means that the executable can be an
* invisible process rather than a console application.
* arguments: in windows case all are ignored, see windows documentation
* for details of arguments.  In linux case standard main arguments
* result: int : -1 if error occurred 0 otherwise.
*/
```

```
#if defined(WIN32)
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int nCmdShow)
{
    int argc ;
    char **argv ;

    ModuleImpl::windowsExtractCommandLine(&argc, &argv); /* get windows commandline */
#elif defined(linux)
int main(int argc, char* argv[])
{
#endif

    try
    {
        // in the following calls help text is an optional argument which has not
        // been supplied in this example

        ModuleImpl modImpl(
            "edu.caltech.trigplus", // module identification
            "trig written in C++", // humanly readable name
            SelfManagedModule); // management scheme

        modImpl.addServiceObject(
            "trig",               // service identification
            "trig written in C++", // humanly readable name
            "trig",               // category
            new Trignometry());   // service implementation

        // connect to broker providing services
        modImpl.run(argc, argv);
    }
    catch (SBWException *e)
    {
#if defined(WIN32)
        MessageBox(
            NULL, e->getMessage().c_str(), "TrigPlus", MB_OK | MB_ICONEXCLAMATION);
#elif defined(linux)
        fprintf(stderr, "TrigPlus exception: %s\n", e->getMessage().c_str());
#endif
        return -1;
    }

    return 0;
}
```

5.1.1 Accessing the API

The SBW C++ API is contained within the `SystemsBiologyWorkbench` namespace. The example above uses the line

```
using namespace SystemsBiologyWorkbench;
```

to enable the SBW API to be used without qualification in the rest of the code.

5.1.2 Portable Application Entry Point

Typically a module will either have a graphical user interface or will be invisible. In either case, on Windows the module will not have the ANSI standard application entry point function `main` but use `WinMain` instead. Our current example includes some portable code to ensure that the command line for the module is available in standard form on Linux and Windows platforms:

```
#if defined(WIN32)
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                    LPSTR lpCmdLine, int nCmdShow)
{
    int argc ;
    char **argv ;

    ModuleImpl::windowsExtractCommandLine(&argc, &argv); /* get windows commandline */
#elif defined(linux)
int main(int argc, char* argv[])
{
#endif
```

In the Windows case, the method `ModuleImpl::windowsExtractCommandLine` extracts the command line. This method is only available on the Windows platform.

5.1.3 Creating Method Implementations

First part of the example code creates an underlying implementation of the service. In C++, each service is implemented as class. Each method to be exposed to other SBW modules and clients (SBW methods) should have the following C++ signature:

```
DataBlockWriter method_name(Module from, DataBlockReader argObject)
```

The `from` parameter represents the calling module. `argObject` contains argument data for the service method. Each method returns a `DataBlockWriter` object containing the result of the service method.

The first operation of each SBW method is to extract the arguments from the container containing the arguments. An example is the code from the `sin` method of the `Trigonometry` class:

```
reader >> arg ;
```

The `>>` operator is defined so that `DataBlockReader` can be used like a standard input stream. Thus, when a service method has more than one argument, the form `reader >> arg1 >> arg2 ... >> argn;` can be used.

The method then returns the result value using the following code:

```
return DataBlockWriter() << (SBWDouble) ::sin(arg);
```

This line of code implements the following operations in order of execution:

- `::sin(arg)` : calculates the result
- `DataBlockWriter()` : constructs a container for the result
- `<<` : stores the result into the container
- `return` : returns the result

An SBW method can raise an error condition by throwing a pointer to a new object created, using the `new` operator, of type `SBWApplicationException`. The `SBWApplicationException` constructor takes a message string parameter. An additional optional parameter is a detailed message string containing information useful to a developer programming an application using the module, for example a stack trace and/or source code location information.

In many situations, services need to invoke service methods on a module instance which previously called the service. For example, a simulator may wish to call back a module when a simulation has completed. This can be achieved by using the `Module from` argument to SBW methods. The `from` argument can be used to locate a specific service on the calling module. The calling module can be called back to notify it of events occurring during or after any calls it makes.

To disconnect from the broker an SBW method should use `SBW::signalDisconnect` instead of `SBW::disconnect`.

5.1.4 Registration of Methods

Simply adding methods with a given signature doesn't expose those methods to SBW. Classes that implement service functionality should contain a method with the following C++ signature:

```
void registerMethods(MethodTable<service_implementation_class> &table)
```

`registerMethods` registers the SBW methods to be exposed using the given `MethodTable` parameter object. The `MethodTable` object has one method `addMethod` which performs a single SBW method registration. So the `registerMethods` method of the service implementation class `Trigonometry` is as follows:

```
static void registerMethods(MethodTable<Trignometry> &table)
{
    table.addMethod(&Trignometry::sin, "double sin(double)");
}
```

The first parameter of `addMethod` is a pointer to a SBW method. The second parameter is the signature of the method using the syntax defined in Section 3.3. In this case the signature indicates that the method will be called `sin` in SBW and will accept a single `double` argument and return a `double` result. There are two more optional arguments to `addMethod` described in section 6.6.

`addMethod` is called for all the SBW methods to be exposed. The `registerMethods` method can be either a class method or an object method i.e. it does not have to be `static`.

5.1.5 Creating a Server Module

To connect the previously defined service to SBW, we need to create a module implementation containing the service implementation and attach a method object to the service.

The code to perform this in our example is contained in the `main` function. `main` provides the code for both the register and module modes of operation for the module. The API doesn't require the developer to make a distinction between the two modes. `main` doesn't have any conditional code: it simply passes `argc` and `argv` to the API.

The first part of `main` is the creation of a module implementation object:

```
ModuleImpl modImpl(
    "edu.caltech.trigplus",
    "trig written in C++",
    SelfManagedModule);
```

In this call, the first parameter is the identifier for the module which should be unique among all modules installed. As described in section 3.4, the SBW development group recommend that developers use the reversed internet domain name scheme used in this example. The second parameter is the human-readable name for the module. This string may be displayed to the user and so should be chosen concise, informative and should clearly identifiy the module from other modules. The third parameter is the module type, which can be `SelfManagedModule` or `UniqueModule`.

`UniqueModule` indicates that one module instance will be shared by all applications which access the module. `SelfManagedModule` indicates that a new module instance will be created every time a module is accessed through `SBW::getModuleInstance` and that either the module or the application accessing the module will decide when to shutdown the module. An application shuts down a module using `Module::shutdown` as shown in section 5.2.

Next, we need to add the service to the `ModuleImpl` object:

```
modImpl.addServiceObject(
    "trig",
    "trig written in C++",
    "trig",
    new Trignometry());
```

The first argument is the identification name for the service and must be unique among the services implemented by this module. The second argument is a human-readable name for the service. The third argument is the category for the service. The fourth argument is a pointer to an object that implements the service. As described above the only requirement of this object is that it has a `registerMethods` method.

In the example code, the final part of the `main` function is the call:

```
modImpl.run(argc, argv)
```

The `ModuleImpl::run` method either registers the module with the broker or starts the provision of services by this module. The two parameters contain data on the arguments passed to the application on the command line and enables the API to determine whether the module is registering or providing services.

5.1.6 Running and Debugging a Module

The executable described here takes a single command line argument which can either be "`-sbwregister`" or "`-sbwmodule`". Any other arguments are ignored. If the argument is "`-sbwregister`" the module simply registers the module and terminates. Before calling this module the module should be first registered with SBW by running with the "`-sbwregister`" argument on the command line. The module should be re-registered whenever the following changes:

- module executable location
- module display name
- module management mode
- service names
- service display names
- service categorization

As this module is self-managed, the module will now be automatically executed whenever another module executes the API function

```
SBW::getModuleInstance("edu.caltech.trig")
```

The broker launches the module executable with "`-sbwmodule`" as the single command line argument and the module runs in module mode providing services. A client application is required to launch the module. Section 5.2 describes a simple C++ client for the module described here. For some modules specific clients will not be appropriate: they can be invoked from a scripting language interpreter such as Python or the module will implement services of a specific category which has a standard interface which can be invoked by a standard client.

If the module is registered as running in the unique management mode then the broker will launch the module once, if it has not already connected to the broker, and all subsequent requests refer to that first instance of the module.

To debug a module which is designed to be self-managed, temporarily change the management mode from `SelfManagedModule` to `UniqueModule` and re-register the module. Then set

breakpoints in the method implementation functions and run the executable from the debugger passing the argument "`-sbwmodule`". For normal use, restore the management mode to `SelfManagedModule` and re-register.

5.2 Calling a Known Module

The simplest use of SBW is to invoke services on a module. And the simplest case of invoking services on a module in SBW is when the module and service identification names are known by the programmer in advance. A module identification name is unique to a given computer. A service identification name is unique to the module. This section gives an example of calling service "trig" on module "edu.caltech.trigplus". The interface for this service is as follows:

```
double sin(double)
double cos{double)
```

This executable simply takes a single command line argument, floating point number, and passes it to the "trig" service to compute the sin of the argument. The executable then prints the computed value.

The complete code for this example is contained in the directory

```
src/tutorials/C++/SimpleTrigPlusClientModule
```

in the SBW installation and CVS tree. An implementation of this service is described in Section 5.1.

The complete code for the example follows:

```
#include <string>
#include <iostream>
#include <sstream>
#include "SBW.h"

using namespace SystemsBiologyWorkbench ;

int main(int argc, char* argv[])
{
    if (argc == 1)
    {
        std::cerr << "cserver: needs one argument";
        return -1;
    }

    try
    {
        // connect to broker
        SBW::connect();

        // fetch module, service and method identities
        Module trigModule = SBW::getModuleInstance("edu.caltech.trigplus");
        Service trigService = trigModule.findServiceByName("trig");
        ServiceMethod sinMethod = trigService.getMethod("double sin(double)");

        Double argument, result ;
        std::stringstream stringStream(argv[1]);

        // extract argument from commandline
        stringStream >> argument ;

        // call sin method and marshal arguments and return values
        sinMethod.call(DataBlockWriter() << argument) >> result ;

        // print result on standard output
```

```
        std::cout << result ;

        // close connection to trig module
        trigModule.shutdown();

        // disconnect from broker
        SBW::disconnect();
    }
    catch (SBWException *e)
    {
        std::cerr << e->getMessage() << " " << e->getDetailedMessage();

        // disconnect from broker
        SBW::disconnect();
        return -1;
    }

    return 0;
}
```

In this example, the `main` function connects to SBW using the `SBW::connect` method, executes one method on another module, and then disconnects from SBW using the `SBW::disconnect` function. This is artificial; typically an application will either connect to SBW for its entire runtime or for at least a much larger sequence of calls on other modules.

The C++ API all methods, apart from `SBW::disconnect`, can potentially throw exceptions of the type `SBWException *`. The `main` function catches exceptions of this type and displays messages contained within the exception object on the standard error stream.

In this code, we are assuming that there is a module identified as "edu.caltech.trigplus" which has a service identified as "trig". We are invoking the `sin` method on this service. The first step is to access an instance of the module and the identity of the module instance. This is performed by the following code fragment:

```
Module trigModule = SBW::getModuleInstance("edu.caltech.trigplus");
```

Whether or not a new module instance is started by this calls depends on the type of the module. If the module is *self-managed* then a new module instance is always started. If the module is *unique*, then the identity of the existing module instance is returned and if no such module instance exists, a new instance is first started.

The next step is to obtain the service object corresponding to the service on the module instance, using the module instance object created in the previous call. This is performed by the following code:

```
Service trigService = trigModule.findServiceByName("trig");
```

Now we can consider calling the `sin` method on this service. First we obtain the method object for the `sin` method using the service object we have obtained:

```
ServiceMethod sinMethod = trigService.getMethod("double sin(double)");
```

We can now invoke the method. To invoke the method in C++, we use the following code:

```
sinMethod.call(DataBlockWriter() << argument) >> result ;
```

This line of code is comprised of four parts in order of execution:

- `DataBlockWriter()` : the construction of a `DataBlockWriter` object to contain the arguments for the call.
- `<<` : storing `argument` in the `DataBlockWriter` object

- `sinMethod.call()` : calling the module method
- `>>` : extracting `result` from the `DataBlockReader` object returned from `sinMethod.call()`.

There is no reason why these operations can't be split into multiple lines; however the above form is easier to read. The `<<` operator is defined so that `DataBlockWriter` can be programmed as if it were an output stream. This means that several arguments can be stored in a single `DataBlockWriter` using the form `DataBlockWriter() << arg1 << arg2 << ... argn`.

When a module instance does not have its own free-standing GUI frame and the module is self-managed, the broker connection to the module instance should be terminated when a client application has finished using the module instance. This applies in our example configuration. In our example code we terminate the broker connection to the module instance after calling the `sin` method, using the following call:

```
trigModule.shutdown();
```

After this, the `main` function finishes correctly by terminating the connection to the broker using the call:

```
SBW::disconnect();
```

The above code shows how a method is called on a blocking basis; i.e., `sinMethod.call()` waits to receive a response from the method implementation. In SBW, it is also possible to call a method on a non-blocking basis. The following method returns as soon as a message is sent to the method implementation:

```
void ServiceMethod::send(DataBlockWriter args)
```

5.3 Creating a Menu of Services from a Service Category

The following tutorial describes a simple application which displays a menu of services to the user, inputs the user's selection and a method on the selection. The services displayed are all in the same category. The application assumes that all the services in this category support the same method.

In this example we have to use the service "Analysis" which is used by a number of existing modules. All services in this category implement the method `void doAnalysis(string sbml)` which takes a string containing SBML and performs some function on the given model. It is assumed that the service display name for the service will indicate to the user what that functionality is. It is also assumed that the module instances providing the selected service is a free standing GUI that should not be shut down by the menu application after calling the `doAnalysis` method.

The complete code for this example is contained in the directory

```
src/tutorials/C++/MenuClientModule
```

in the SBW installation and CVS tree.

The complete code for this example is as follows:

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include "sbw.h"

using namespace SystemsBiologyWorkbench;

int main(int argc, char* argv[])
```

```
{
    try
    {
        // check file name has been supplied
        if (argc < 2)
        {
            std::cerr << "usage: " << argv[0] << " <filename>\n";
            return 1;
        }

        std::string sbml;

        // read file
        std::ifstream sbmlSource(argv[1]);

        if (!sbmlSource)
        {
            std::cerr << "unable to open file " << argv[1] << '\n';
            return 1;
        }

        char c ;

        while (sbmlSource.get(c))
            sbml += c ;

        if (!sbmlSource.eof())
        {
            std::cerr << "unable to read file " << argv[1] << "correctly\n";
            sbmlSource.close();
            return 1 ;
        }

        sbmlSource.close();

        // connect to SBW broker
        SBW::connect();

        // query broker for services in Analysis category
        std::vector<ServiceDescriptor> *serviceDescriptors =
            SBW::findServices("Analysis");

        int i = 0 ;

        // create menu
        while (i != serviceDescriptors->size())
        {
            std::cout << i << ' ' ;
            std::cout << (*serviceDescriptors)[i].getDisplayName() << '\n';
            i++;
        }

        std::cout << "\nType number to select service\n" ;
        std::cout.flush();

        // get user response
        std::cin >> i ;

        // check response
        if (i < 0  i >= serviceDescriptors->size())
        {
            std::cerr << "Incorrect selection\n";
            SBW::disconnect();
            return 1;
        }
```

```
        // locatecreate module instance based on selected service
        Service service = (*serviceDescriptors)[i].getServiceInModuleInstance();

        delete serviceDescriptors ;

        // locate method
        ServiceMethod method = service.getMethod("doAnalysis(string)");

        // call method
        method.call(DataBlockWriter() << sbml);

        // disconnect from broker
        SBW::disconnect();

        return 0;
    }
    catch (SBWException *e)
    {
        std::cerr << e->getMessage() << '\n' << e->getDetailedMessage();
        return 1;
    }
}
```

The code up to the point at which the application connects to the broker simply loads the contents of a file, given as a command line argument, into the string `sbml`. After connecting the following code fetches information on services in the "Analysis" category:

```
std::vector<ServiceDescriptor> *serviceDescriptors = SBW::findServices("Analysis");
```

`ServiceDescriptor` objects provide information on the services potentially provided by registered modules and don't represent services on module instances. The application uses these objects to create a menu, on standard output, of the display names of the described services as follows:

```
int i = 0 ;

while (i != serviceDescriptors->size())
{
    std::cout << i << ' ' << (*serviceDescriptors)[i].getDisplayName() << '\n';
    i++;
}
```

The application then inputs and checks the users response as follows:

```
std::cout << "\nType number to select service\n" ;
std::cout.flush();

// get user response
std::cin >> i ;

// check response
if (i < 0  i >= serviceDescriptors->size())
{
    std::cerr << "Incorrect selection\n";
    SBW::disconnect();
    return 1;
}
```

The variable `i` now contains the index of the service that the user has selected. We can now obtain a `Service` object represented by the selected `ServiceDescriptor` as follows:

```
Service service = (*serviceDescriptors)[i].getServiceInModuleInstance();
```

The method `getServiceInModuleInstance` creates a module instance that implements the described service and returns a reference to that service on the new module instance. In short it starts with a description of a service and returns an instance of that service.

The vector of `ServiceDescriptors` can now be destroyed as its no longer required:

```
delete serviceDescriptors ;
```

Now we have a `Service` object we can call the `doAnalysis` method on it as described in the previous tutorial:

```
ServiceMethod method = service.getMethod("doAnalysis(string)");

method.call(DataBlockWriter() << sbml);
```

The application finishes by disconnecting from the broker and terminating.

5.4 Using Class Inheritance with Service Implementations

The coding of service implementations which use a class hierarchy requires a variation of the technique shown in section 5.1.

The following example shows how a simple class hierarchy can be used to create a service implementation:

```
#include "SBW.h"

using namespace SystemsBiologyWorkbench;

// simple base class for SBW service
class TestService
{
public:
    // creates simple test service implementation initializes counter to zero.
    TestService() : i(0) {}

    // SBW method - return the counter
    DataBlockWriter getCount(Module ignored, DataBlockReader ignoredToo)
    {
        return DataBlockWriter() << i;
    }

    // SBW method -  increment the counter
    DataBlockWriter increment(Module ignored, DataBlockReader ignoredToo)
    {
        i++;

        return DataBlockWriter();
    }

    // register methods of this class with a module implementation
    template <class T> static void registerMethods(MethodTable<T> &table)
    {
        table.addMethod(&T::getCount, "int getCount()");
        table.addMethod(&T::increment, "void increment()");
    }

protected:
    // counter
    Integer i;
};

/// simple test service implementation - checking that derivation can work
class DerivedTestService : public TestService
```

```
{
public:
    // SBW set the counter
    DataBlockWriter set(Module ignored, DataBlockReader args)
    {
        args >> i ;

        return DataBlockWriter() ;
    }

    // register methods of this class with a module implementation
    static void registerMethods(MethodTable<DerivedTestService> &table)
    {
        TestService::registerMethods(table);
        table.addMethod(&DerivedTestService::set, "void set(int)");
    }
};

#if defined(WIN32)
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int nCmdShow)
{
    int argc ;
    char **argv ;

    ModuleImpl::windowsExtractCommandLine(&argc, &argv); /* get windows commandline */
#elif defined(linux)
int main(int argc, char* argv[])
{
#endif
    try
    {
        ModuleImpl moduleImpl(
            "NativeObjectOrientedServer",
            "NativeObjectOrientedServer",
            UniqueModule);

        moduleImpl.addServiceObject("test", "test service", "", new TestService());
        moduleImpl.addServiceObject(
            "derivedTest", "derived test service", "", new DerivedTestService());
        moduleImpl.run(argc, argv);
    }
    catch (SBWException *e)
    {
        std::string text = e->getMessage().append("\n");

        text = text.append(e->getDetailedMessage());
        printf(text.c_str());
        return 1;
    }

    return 0;
}
```

This code is a specific example which demonstrates the general principles of using service implementation classes in a class hierarchy. A super class, such as `TestService`, can contain a `registerMethods` method which registers its methods. If this method is to be called from derived classes then it needs to defined as a template method i.e. have the form:

```
template <class T> void registerMethods(MethodTable<T> &)
{
...
}
```

As this is a template method the method must be `inline` as shown in the example. This method can then be called from within the `registerMethods` method on a derived class as shown on the class `DerivedTestService`. Objects of the super class as defined can be registered as service implementations as shown.

This module, `NativeObjectOrientedServer`, exposes the following services:

```
test
    int getCount()
    void increment()

derivedTest
    int getCount()
    void increment()
    void set(int)
```

6 C++ API Reference

6.1 Overview

6.1.1 The `SystemsBiologyWorkbench` Namespace

The entire C++ API is contained in the `SystemsBiologyWorkbench` namespace. i.e. all the type identifiers described in this section are contained with in that namespace.

6.1.2 Classes

The classes in the C++ API are as follows:

- `SBW` enables the discovery of services and modules (see Section 6.4)
- `ModuleImpl` enables the registration and implementation of module services (see Section 6.7)
- `Module` enables access to the services provided by a module instance (see Section 6.8)
- `Service` enables access to the methods on a service in a module instance (see Section 6.9)
- `ServiceMethod` a method on a service on a module instance (see Section 6.10)
- `ServiceDescriptor` a component of a query result on the registered services (see Section 6.11)
- `ModuleDescriptor` the static data stored in the registry associated with a module (see Section 6.12)
- `DataBlockReader` enables access to the arguments sent with a message and the results returned from a call (see Section 6.17)
- `DataBlockWriter` enables encoding of arguments sent with a message and the results returned from a call (see Section 6.17)
- `MethodTable` stores information on methods on service implementations
- `Handler` service method implementation interface
- `Signature` representation of (or parse output from) a signature string (see section 6.13)
- `SignatureElement` representation of an argument in a signature string (see section 6.14)
- `SignatureType` representation of a type in a signature string (see section 6.15)

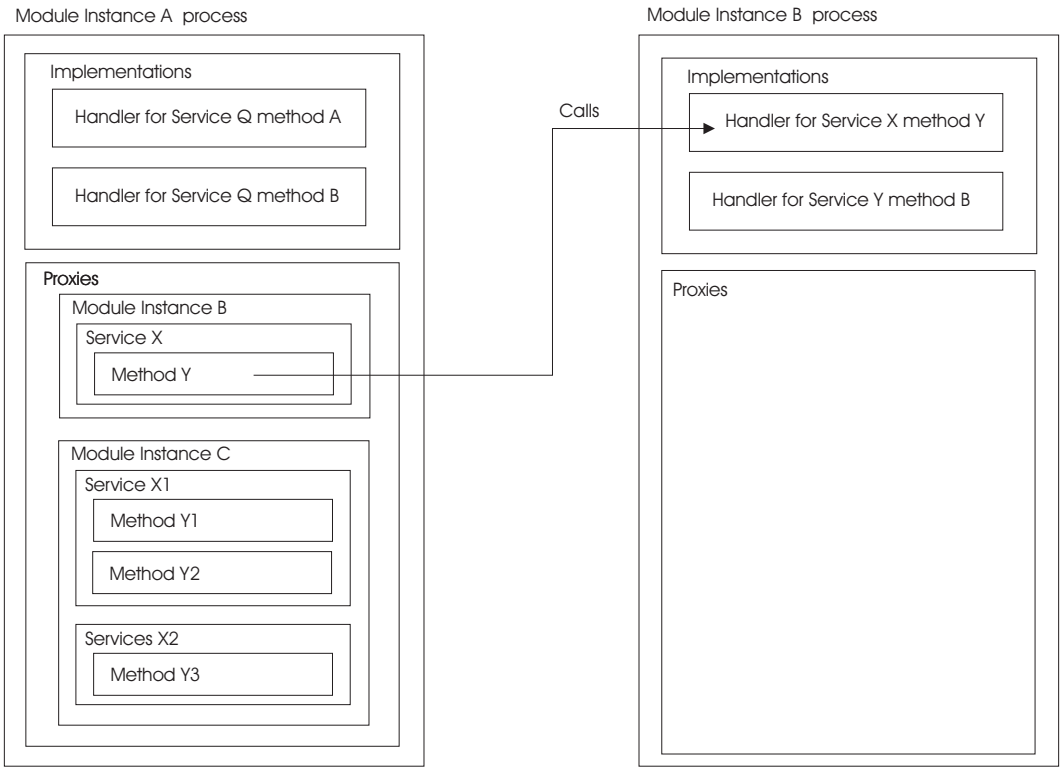

Figure 4: *Overview of SBW proxy objects.*

- `SBWListener` a class which can be derived from to provide an event notification in a module (see section 6.16)

These classes can be divided into two groups: those that are local representations or proxies for other concrete objects that exist elsewhere in the SBW system and those that exist only within a calling application. `Module`, `Service` and `ServiceMethod` objects are proxies for external objects. The remaining objects exist only within a calling application.

Figure 4 shows how the proxy API objects are grouped.

6.1.3 `std::vector` objects

The `std::vector` objects returned by API functions are owned by the calling environment and should be destroyed using the `delete` operator.

6.2 Scalar Types

Integers

The type `Integer` is used in place of all integers and is defined to be a signed 32-bit integer.

Doubles

The type `Double` is used in place of all floating point numbers and is defined to be a IEEE double precision floating point number (64-bit).

6.3 Enumeration Types

Table 2 lists the enumeration types used in the C++ API.

Type name	Definition

ModuleManagementType	An enumeration of the various management modes.
	`typedef enum moduleManagementType`
	`{`
	`  UniqueModule = 0, SelfManagedModule = 1`
	`} ModuleManagementType;`

DataBlockType	An enumeration of the various data block types.
	`typedef enum DataBlockType {`
	`  IntegerType, DoubleType, StringType,`
	`  ArrayType, ListType, BooleanType,`
	`  ByteType, TerminateType, ErrorType,`
	`  VoidType`
	`  } DataBlockType;`
	where `Terminate` marks the end of the data block.

Table 2: *C++ API enumeration types*

6.4 SBW Class

SBW enables an application to connect and disconnect from the SBW Broker as well as locating modules and services. The SBW class has the following methods:

static void connect()

connects this application to the SBW system indicating that this application does not provide any services.

static void connect(string hostNameOrIPAddress)

connects this application to a remote SBW system indicating that this application doesn't provide any services. `hostNameOrIPAddress` can be either a fully qualified host name e.g. `erato4.cds.caltech.edu` or an IP address e.g. `131.215.142.99`.

static Module getModuleInstance(std::string identificationName)

returns an object representing an instance of a given module. The module is identified by its unique identification name. Creates a new module instance (launches a new process) if a module instance doesn't exist or if the module is not of the *Unique* type.

static std::vector<ServiceDescriptor> *findServices(
std::string serviceCategory, bool recursive = true)

returns service records for all the services registered with the SBW system in the given category and subcategories. This is a simple query for services. An example might be to use the category "Analysis" to query for all the services which provide analyses of SBML models.
If `recursive` is true then this function returns all the services in this category and its subcategories. If `recursive` is false then this function returns only the services in this category and not those in its subcategories.

static std::vector<std::string> *getServiceCategories(std::string serviceCategory)

returns all the immediate subcategories of a given category.

static void disconnect();

notifies the SBW system that this application process is no longer available to provide, and does not need to consume, services. Terminates only when all service calls being performed by this instance have completed and the SBW module library is in its shutdown state. This function should not be called inside a service method implementation.

static std::vector<Module> *getExistingModuleInstances()

returns all the module instances that are currently running. This function would normally be used in SBW explorer, scripting and debug environments.

static std::vector<Module> getExistingModuleInstances(
std::string moduleIdentificationName)

returns all the module instances with the given module identification name that are currently running. This function would normally be used in SBW explorer, scripting and debug environments.

static std::vector<ModuleDescriptor> *getModuleDescriptors(
bool includeRunningModules = false,
bool localOnly = false)

returns descriptions of all the modules registered with the SBW system. Includes modules which are not registered with the broker but are running only if `includeRunningModules` is set. Excludes remote modules if `localOnly` is set.

static ModuleDescriptor getModuleDescriptor(
String moduleUniqueName, bool includeRunning = false)

returns the module descriptor for a module given the module's unique name includes modules which are not registered with the broker but are running only if `includeRunningModules` is set.

static void addListener(SBWListener *)

adds a listener to receive events as they occur in the module.

static void removeListener(SBWListener *)

removes the given `SBWListener` object so that it no longer receives events.

static void signalDisconnect()

notifies the SBW system that this application process is no longer available to provide services and does not need to consume. This function may terminate before service calls being performed by this instance have completed. This function will terminate before the SBW module library is in its shutdown state. This function can be called inside a service method implementation (`Handler::receive` overload).

static void waitForDisconnect()

terminates once all service calls being performed by this instance have completed and the SBW module library is in its shutdown state. This function should not be called inside a service method implementation (`Handler::receive` overload).

static std::string getVersion()

returns a version string for the C library.

static std::string getBrokerVersion()

returns a version string for the broker.

static Module getThisModule()

returns an object representing the module implemented by this application.

6.5 Handler Class

The C++ API has an abstract class, `Handler`. A class derived from `Handler` implements a SBW service method. It has the following method and a virtual destructor.

virtual DataBlockWriter receive(Module from, DataBlockReader args) = 0

The `receive` method should be overloaded with the implementation of a service method. `from` is a proxy for the calling module instance. `args` contains the argument data for the service method. The `receive` method should return a `DataBlockWriter` containing the result data.

6.6 MethodTable Class

The C++ API has a template class, `MethodTable<class T>` where `T` is a service implementation class. Objects of `MethodTable` class expose methods of class `T` as SBW methods. `MethodTable` objects are constructed by the API and passed to the `registerMethods` method of the service implementation class. `MethodTable` objects are created by `addServiceObject` on `ModuleImpl`.

`MethodTable` has one method:

void addMethod(
Method method,
std::string signature,
bool synchronized = false,
std::string help = "")

`addMethod` exposes `method` as an SBW method. `signature` is the SBW signature for `method` (see Section 3.3). `synchronized` indicates whether the method must not be run concurrent with other methods on the service that have been registered with this parameter set `true`. This means the default (false value) behavior is to allow the method to run concurrently with all methods. The `synchronized` parameter is roughly equivalent to the Java `synchronized` keyword. `help` is the documentation for this method.
`method` is a pointer to a method on the class `T` and has the type

```
DataBlockWriter (T:: * Method) (Module from, DataBlockReader args)
```

where `from` is a proxy for the calling module instance. `args` contains the argument data for the service method. The method should return a `DataBlockWriter` containing the result data.

6.7 ModuleImpl Class

Enables the registration and the creation of module service implementations.

`ModuleImpl` constructor has the following signature:

ModuleImpl(
std::string uniqueName,
std::string nameForDisplay,
ModuleManagementType type,
std::string help = "")

The `type` parameter is one of the following values:

```
UniqueModule
SelfManagedModule
```

`uniqueName` is the unique identification string for the module. `nameForDisplay` is the humanly readable name for the module. `moduleCommand` is the command that should be run in the current environment to invoke the module. `help` is the documentation for this module.

`ModuleImpl` has the following methods:

void addService(
std::string serviceName,
std::string serviceDisplayName,
std::string category,
std::string help = "")

Notifies the SBW system that the service `serviceName` is provided by the given module i.e. adds `serviceName` to the list of services provided by this module in the module registry. `category` is the service category for this service. `help` is the documentation for this service.
Throws `SBWServiceNameIsNotUniqueToModule` if `serviceName` is already in the list of services for this module.

template<class T> void addServiceObject(
std::string serviceName,
std::string serviceDisplayName,
std::string category,
T *serviceImplementation,
std::string help = "")

Notifies the SBW system that the service `serviceName` is provided by `serviceImplementation` as part of the given module. `T` is the type of `serviceImplementation`. `category` is the service category for this service. `help` is the documentation for this service. This method adds `serviceName` to the list of services provided by this module in the module registry.
`T` must be a class which has a member function, static or otherwise, with the C++ signature:

```
void registerMethods(MethodTable<T> &)
```

`addServiceObject` throws `SBWServiceNameIsNotUniqueToModule` if `serviceName` is already in the list of services for this module.

void setHandler(
std::string serviceName,
Handler *handler,
std::string signature,
bool synchronized = false,
std::string help = "")

notifies SBW that `handler` is the implementation of the given method on the service identified by `serviceName`. `signature` is a method signature (see Section 3.3). `synchronized` indicates whether the method must not be run concurrent with other methods that have been registered with this parameter set `true`. This means the default (false value) behaviour is to allow the method to run concurrently with all methods. The `synchronized` parameter is roughly equivalent to the Java `synchronized` keyword. `help` is the documentation for this method. `Handler` objects attached to the API using `setHandler` are destroyed when a module terminates.

void run(int argc, char **argv, bool waitForDisconnect = true)

`argc` and `argv` contain the argument data for this module. If this data contains the argument `-sbwmodule` then the method provides services to other modules. If the argument data contains the argument `-sbwregister` then the method registers this module with the broker. If the method provides services to other modules and `waitForDisconnect` indicates whether `SBWModuleImplRun`, in module mode, blocks until the module is shutdown. This value is typically left true however if the module, for example, needs to run a message loop to service a UI, then it might be appropriate to set this parameter to false and place the message loop after the `run` call so that the module can process both UI and SBW method calls. In this case it is important to note that service provision will end as soon as the `main` or `WinMain` function terminates.

Handler *getHandler(std::string serviceName, std::string methodSignature)
throws SBWServiceNotFoundException

returns a handler which is the implementation of the given method on the given service. Returns null if SBW has not been notified of this handler. This function uses the same algorithm for matching the given signature as that used in the method `Service::getMethod`.

void registerModule()

registers this module with the SBW Broker, adding a persistent record of how to start the module and a list of the services implemented by the module.

void enableModuleServices()

implements the module services. Notifies the SBW Broker that this module's services are operational and enables handlers for calls to methods on these services. This method returns as soon as this notification and setup process is complete.

void setCommandLine(std::string commandLine)

sets the command line to start the `module` in module mode. This string is stored in the SBW Broker registry file when the module is registered.

static std::string calculateCommandLine(std::string moduleCommand)

returns the command line for the registration of a module. Normally `moduleCommand` is the `argv[0]` of the given module and the return value of `calculateCommandLine` is passed to
`ModuleImpl::setCommandLine`.

static void windowsExtractCommmandLine(int *argc, char ***argv)

This method is specific to Windows platforms and sets `argc` and `argv` to contain the arguments of passed on the command line to this application. This method is essential when using `ModuleImpl::run` inside a Windows non-console application.

6.8 Module Class

The `Module` class represents an instance of a module. A `Module` object contains zero or more `Service` objects which represent the services implemented by the module instance.

The `Module` class has a default constructor. Using a `Module` object which has not been assigned from an API return value will result in an exception. The `Module` class has the following methods:

std::vector<Service> *getServices()

returns the services implemented by the module instance

Service findServiceByName(String serviceName)

returns the service with the given name implemented by the module instance. Returns null if such a service doesn't exist.

std::vector<Service> *findServicesByCategory(String serviceCategory)

returns the services in the given category implemented by the module instance.

ModuleDescriptor getDescriptor()

returns a description of the module.

void shutdown()

instructs the broker to close its connection with the module instance. The normal behaviour, especially with non GUI self managed modules, is for the module to terminate. Applications which have finished using a non GUI self managed module should call this function.

bool operator==(Module other)
bool operator!=(Module other)

The equality and inequality operators are overloaded to enable the comparison of `Module` objects. Two `Module` objects are equal if they refer to the same module instance.

6.9 Service Class

The `Service` class represents one interface that is available on a given module instance.

An application sends messages (invokes methods) on services to access the resources in a module instance. The mechanism for doing this depends on the language employed. In C++ messages are sent using the method class, see section 6.10.

The `Service` class has a default constructor. Using a `Service` object which has not been assigned from an API return value will result in an exception. The `Service` class has the following methods:

void getDescriptor(ServiceDescriptor &x)

sets `x` to information about the service

void getModule(Module &x)

sets `x` to be the module instance containing this service.

std::vector<ServiceMethod> *getMethods()

returns the methods on this service

ServiceMethod getMethod(String signature)

returns the method with the given signature on this service. The given signature string can be either be just the method name or should have the following syntax with respect to the definition in Section 3.3:

```
[ReturnType Space] SName Space? '(' VarArgList ')'
```

where the arguments list is used to locate specific methods with the same name. The return type is ignored when matching signatures. When matching types, empty list syntax, , will match with any list type irrespective of its content. When matching types, if two list types specify a sequence of types, for example `int, double` then the type sequences should match exactly. For example `void foo(int, double)` matches both `void foo(int a, double b)` and `void foo()`.

bool operator==(Service other)
bool operator!=(Service other)

The equality and inequality operators are overloaded to enable the comparison of `Service` objects. Two `Service` objects are equal if they refer to the same service on the same module instance.

6.10 ServiceMethod

This class is specific to C++ and represents a method on a service on a module instance. An application sends messages (invokes methods) on services to access the resources in a module instance. In C++ messages are sent using the method class.

The `ServiceMethod` class has a default constructor. Using a `ServiceMethod` object which has not be assigned from an API return value will result in an exception. The `ServiceMethod` class has the following methods:

std::string getName()

returns the method name

std::string getHelp()

returns documentation on the method

std::string getSignatureString()

returns the method signature for this method (see Section 3.3).

Signature getSignature()

return the Signature object corresponding to the signature of this method.

DataBlockReader call(DataBlockWriter args)

Send a message to the service, blocking until the method has completed on the service. The return value is that which was returned by the service implementation. The `DataBlockWriter` argument is the arguments to the method on the service.

void send(DataBlockWriter args)

Send a message to the service, returning immediately. The DataBlockWriter argument is the arguments to the method on the service.

void getService(Service &x)

assigns the service containing this method to `x`

6.11 ServiceDescriptor Class

Objects of this class contain information derived from the SBW registry file for one service. These objects can be used as handles to potential services that may be used in the future. For example if a list of services in a given category is to be displayed to the user before those services are invoked a representation is required of those services outside the context of a module instance.

The `ServiceDescriptor` class has the following methods:

Service getServiceInModuleInstance()

returns an service object corresponding to the service record. This service object is part of an existing module instance or a newly created module instance depending on the the module type and whether a module instance currently exists.

std::string getName()

returns the name of the service

std::string getHelp()

returns documentation on the service

std::string getDisplayName()

returns the humanly readable name of the service

std::string getCategory()

returns the category of the service

ModuleDescriptor getModuleDescriptor()

returns the description of the module that implements the service.

6.12 ModuleDescriptor Class

this class represents the data stored in the registry for a module

std::string getName()

returns the unique identification name of the module

std::string getDisplayName()

return the display name of the module

std::string getHelp()

returns documentation on the module

std::string getCommandLine()

returns the commandline required to launch the module

std::vector<ServiceDescriptor> *getServiceDescriptors()

returns service descriptors for the service implemented by the module

SBWModuleManagementType getManagementType()

returns the management type of the module. the return value is one of:

```
UniqueModule
SelfManagedModule
```

6.13 Signature Class

This class represents the structure of a signature or the result of a parse of a signature. This class has the following methods:

std::vector<SignatureElement> *getArguments()

returns a representation of the arguments in a signature

SignatureType getReturnType()

returns a representation of the result in a signature

std::string getName()

returns the method name part of a signature

6.14 SignatureElement Class

This class represents the structure of an argument in a signature or the result of a parse of such an argument. This class has the following methods:

std::string getName()

returns the name of the argument

void getType(SignatureType &x)

sets `x` to the type of the argument

6.15 SignatureType Class

This class represents the structure of a type in a signature or the result of a parse of such an type. This class is thus a union of the methods required to access information on different types including lists and arrays. This means that some methods will throw exceptions if the method is inappropriate to the type represented by the object. This class has the following methods:

DataBlockType getType()

returns the type of this object which is one of the following values:

```
SBWByteType = 0 ,
SBWIntegerType = 1,
SBWDoubleType = 2,
SBWBooleanType = 3,
SBWStringType = 4,
SBWArrayType = 5,
SBWListType = 6,
SBWVoidType = 7,
SBWComplexType = 8
```

SignatureType getArrayInnerType()

returns the type of this array if this is an array, throws `TypeMismatchException` otherwise.

std::vector<SignatureElement> *getListContents()

returns the contents of this list if this is a list, throws `TypeMismatchException` otherwise.

Integer getArrayDimensions()

returns the number of dimensions of the array type represented by this object

6.16 SBWListener Class

By registering objects of classes derived from this class using the `SBW::addListener` method modules can be notified of events occurring in the SBW system.

virtual void onModuleShutdown(Module module)

This method is called every time a module instance disconnects from the broker. `module` represents the module instance that has just shutdown.

virtual void onModuleStart(Module module)

This method is called every time a module instance starts up. `module` represents the module instance that has just shutdown.

virtual void onRegistrationChange()

This method is called whenever the registration data in the broker changes.

virtual void onShutdown()

This method is called when the broker disconnects from this application

6.17 Serialization of Messages

This section describes how to construct and deconstruct message blocks in C++. Message blocks are used as containers for service method arguments and results.

In the API `DataBlockWriter` is used to set the arguments to a method and to encode the data returned from a method in its implementation. In the API `DataBlockReader` is used to extract the arguments to a method, in its implementation, and to extract the data returned from a method.

In all cases `DataBlockWriter` and `DataBlockReader` act like output and input streams i.e. they track a cursor through the data block. This means that a data block can constructed or deconstructed in one or more calls.

`DataBlockWriter` and `DataBlockReader` objects are smart handles for an underlying data representation. This means that if you, for example, declare two `DataBlockWriter` objects, assign one to the other and then insert data into only one of the objects the result would be that both objects refer to the inserted data. Similarly if you, for example, declare two `DataBlockReader` objects, assign one to the other and then extract data from only one of the objects the result would be that both objects have had the data extracted.

One side effect of this implementation is the effect of

```
std::vector<DataBlockWriter>::resize
```

and the

```
std::vector<DataBlockWriter>
```

constructor. These methods result in a vector in which all the elements in the array refer to the same underlying implementation object. To avoid this effect assign separately constructed `DataBlockWriter` objects to each element of the vector. For example:

```
std::vector<DataBlockWriter> x(2);

x[0] << 1 << 2 ;
x[1] << 3 << 4 ;
```

will result in both elements referring to the same data block containing the sequence 1, 2, 3 and 4. To overcome this problem new `DataBlockWriter` objects are assigned to each element:

```
std::vector<DataBlockWriter> x(2);

x[0] = DataBlockWriter() << 1 << 2 ;
x[1] = DataBlockWriter() << 3 << 4 ;
```

Datablocks are recursive data structures: a datablock can contain nested datablocks. These

nested datablocks represent heterogeneous lists. This can be used, for example, to return complex data structures as return values from a method with a signature like:

```
{string name, int age } getPerson(int id)
```

and to represent arrays of complex structures for a method with a signature like:

```
void insertPeople({string name, int age }[])
```

The contents of these nested structures are not fixed by type, for example in a method with a signature like:

```
{} getData()
```

The caller can examine the types of the content of the returned nested datablock.

6.17.1 DataBlockWriter

This class has the following operators and methods. All these methods add data to the `DataBlockWriter` in sequence.

DataBlockWriter()

Constructs an empty data block.

DataBlockWriter& operator<<(const char *)
DataBlockWriter& operator<<(std::string)
DataBlockWriter& operator<<(Integer)
DataBlockWriter& operator<<(Double)
DataBlockWriter& operator<<(bool)
DataBlockWriter& operator<<(unsigned char)
DataBlockWriter& operator<<(std::complex<double >)
DataBlockWriter& operator<<(DataBlockWriter)
DataBlockWriter& operator<<(const std::vector<const char *> &)
DataBlockWriter& operator<<(const std::vector<std::string> &)
DataBlockWriter& operator<<(const std::vector<Integer> &)
DataBlockWriter& operator<<(const std::vector<Double> &)
DataBlockWriter& operator<<(const std::vector<bool> &)
DataBlockWriter& operator<<(const std::vector<unsigned char> &)
DataBlockWriter& operator<<(const std::vector<const DataBlockWriter> &)
DataBlockWriter& operator<<(const std::deque<const char *> &)
DataBlockWriter& operator<<(const std::deque<std::string> &)
DataBlockWriter& operator<<(const std::deque<Integer> &)
DataBlockWriter& operator<<(const std::deque<Double> &)
DataBlockWriter& operator<<(const std::deque<bool> &)
DataBlockWriter& operator<<(const std::deque<unsigned char> &)
DataBlockWriter& operator<<(const std::deque<const DataBlockWriter> &)
DataBlockWriter& operator<<(const std::list<const char *> &)
DataBlockWriter& operator<<(const std::list<std::string> &)
DataBlockWriter& operator<<(const std::list<Integer> &)
DataBlockWriter& operator<<(const std::list<Double> &)
DataBlockWriter& operator<<(const std::list<bool> &)
DataBlockWriter& operator<<(const std::list<unsigned char> &)
DataBlockWriter& operator<<(const std::list<const DataBlockWriter> &)

These `<<` operators add their arguments to the `DataBlockWriter` and are designed to be used in the same way as the standard library output stream operators.

void add(int size, const char **)
void add(int size, const std::string *)
void add(int size, const Integer *)
void add(int size, const Double *)
void add(int size, const bool *)
void add(int size, const std::complex<double> *)
void add(int size, const unsigned char *)
void add(int size, const DataBlockWriter *)

These `add` methods add C type 1 dimensional arrays of the given size to the `DataBlockWriter`.

void add(int sizeX, int sizeY, const char *)**
void add(int sizeX, int sizeY, const std::string **)
void add(int sizeX, int sizeY, const Integer **)
void add(int sizeX, int sizeY, const Double **)
void add(int sizeX, int sizeY, const std::complex<double> **)
void add(int sizeX, int sizeY, const bool **)
void add(int sizeX, int sizeY, const unsigned char **)
void add(int sizeX, int sizeY, const DataBlockWriter **)

These `add` methods add C type 2 dimensional arrays of the given size to the `DataBlockWriter`.

6.17.2 DataBlockReader

`DataBlockReader` has the following operators and methods.

DataBlockReader()

creates an empty data block reader.

DataBlockReader& operator>>(std::string)
DataBlockReader& operator>>(Integer)
DataBlockReader& operator>>(Double)
DataBlockReader& operator>>(bool)
DataBlockReader& operator>>(unsigned char)
DataBlockReader& operator>>(std::complex<double>)
DataBlockReader& operator>>(DataBlockReader)
DataBlockReader& operator>>(std::vector<std::string> &)
DataBlockReader& operator>>(std::vector<Integer> &)
DataBlockReader& operator>>(std::vector<Double> &)
DataBlockReader& operator>>(std::vector<bool> &)
DataBlockReader& operator>>(std::vector<unsigned char> &)
DataBlockReader& operator>>(std::vector< DataBlockReader > &heterogeneousListArray)
DataBlockReader& operator>>(std::deque<std::string> &)
DataBlockReader& operator>>(std::deque<Integer> &)
DataBlockReader& operator>>(std::deque<Double> &)
DataBlockReader& operator>>(std::deque<bool> &)
DataBlockReader& operator>>(std::deque<unsigned char> &)
DataBlockReader& operator>>(std::deque< DataBlockReader > &heterogeneousListArray)
DataBlockReader& operator>>(std::list<std::string> &)
DataBlockReader& operator>>(std::list<Integer> &)
DataBlockReader& operator>>(std::list<Double> &)
DataBlockReader& operator>>(std::list<bool> &)
DataBlockReader& operator>>(std::list<unsigned char> &)
DataBlockReader& operator>>(std::list< DataBlockReader > &heterogeneousListArray)

These **>>** operators extract their arguments from the `DataBlockWriter` and are designed to be used in the same way as the standard library input stream operators.

void get(int &size, std::string *&)
void get(int &size, Integer *&)
void get(int &size, Double *&)
void get(int &size, bool *&)
void get(int &size, unsigned char *&)
void get(int &size, std::complex<double> *&)
void get(int &size, DataBlockReader *&)

These `get` methods extract one dimensional arrays from the `DataBlockReader`. The `size` parameter is set to the size of the extracted array and the other parameter is set to point to the newly allocated array. All arrays created by these methods are owned by the caller and should be deleted using the C++ `delete[]` operator.

void get(int &sizeX, int &sizeY, std::string **&)
void get(int &sizeX, int &sizeY, Integer **&)
void get(int &sizeX, int &sizeY, Double **&)
void get(int &sizeX, int &sizeY, bool **&)
void get(int &sizeX, int &sizeY, unsigned char **&)
void get(int &sizeX, int &sizeY, std::complex<double> **&)
void get(int &sizeX, int &sizeY, DataBlockReader **&)

These `get` methods extract two dimensional arrays from the `DataBlockReader`. The size parameters are set to the size of the extracted array and the other parameter is set to point to the newly allocated array. All arrays created by these methods are owned by the caller and should be deleted using the methods:

```
void free2DArray(int xsize, std::string **)
void free2DArray(int xsize, Integer **)
void free2DArray(int xsize, Double **)
void free2DArray(int xsize, bool **)
void free2DArray(int xsize, unsigned char **)
void free2DArray(int xsize, DataBlockReader **)
```

The following `DataBlockReader` methods allow a data block of an unknown format to be decoded:

DataBlockType getNextType()

Returns the type of the next item in the reader.

DataBlockType getNextArrayType()

Assuming that the next object in the reader is an array returns the type of item in the array.

int getNextArrayDimensions()

Assuming that the next object in the reader is an array returns the number of dimensions of the array.

6.18 Exceptions

All methods in the API apart from `SBW::disconnect` potentially throw pointers to objects of classes derived from `SBWException`. When a `SBWException *` is caught it should eventually be deleted or thrown.

`SBWException` has the following methods.

std::string getMessage()

This is a message that can be displayed to a user.

std::string getDetailedMessage()

This is a message that can be displayed to a developer and may include a stack trace.

6.18.1 Derived exceptions

Table 3 shows the exceptions derived from `SBWException`.

6.18.2 Raising Application Exceptions

Service methods can raise application error conditions. Service method implementation classes (derived from `Handler`) should throw pointers to `SBWApplicationException` objects from within overloaded `receive` methods to indicate error conditions. `SBWApplicationException` has the following constructors:

public SBWApplicationException(const char *message)
public SBWApplicationException(const char *message; const char *detailedMessage)
public SBWApplicationException(std::string message)
public SBWApplicationException(std::string message; std::string detailedMessage)

`message` is a message for a user and `detailed message` is a message for a developer.

Exception Class	Meaning
SBWApplicationException	Application-specific exception, thrown deliberately by a module. This is the only type of exception that can be created by a module in SBW.
SBWRawException	Throw by API when platform exceptions occur and the context of the error has been lost
SBWCommunicationException	Communications between a caller and receiver failed, possibly due to a lost connection.
SBWModuleStartException	An attempt to start a new module failed.
SBWTypeMismatchException	The type of data element that was attempted to be read from a message was not the type found. This often indicates a mismatch between the messages expected by a caller and receiver.
SBWIncompatibleMethodSignatureException	An interface or class definition uses method signatures that don't correspond to the signatures on the corresponding service.
SBWModuleIdSyntaxException	Indicates that a supplied module instance identifier has incorrect syntax.
SBWIncorrectCategorySyntaxException	A supplied category string has incorrect syntax. This is thrown by methods that find or search services by categories.
SBWServiceIsNotFoundException	Requested service is not present on this module.
SBWMethodTypeNotBlockTypeException	The supplied class uses types which are not SBW message data block types.
SBWMethodAmbiguousException	Thrown by search methods such as `getMethod` on the `Service` class to indicate that the given signature or signature component matches with more than one method on the given service.
SBWUnsupportedObjectTypeException	The library encountered an object of a type that it cannot encode or decode from a message block.
SBWMethodNotFoundException	The given method identification number, signature or partial signature does not match any of the methods that exist on a given service. This can occur if a caller uses the low-level SBW API to attempt to invoke a method on a service and the service has no method with that index.
SBWSignatureSyntaxException	Thrown if a signature string does not contain a valid SBW signature.
SBWModuleDefinitionException	Thrown by the `ModuleImpl` constructors if any aspect of a module definition is incorrect.
SBWModuleNotFoundException	Thrown if the given module identification name doesn't match any module known to SBW.
SBWBrokerStartException	Thrown if a problem occurred starting a broker.

Table 3: *Exception Classes*

References

Arkin, A. P. (2001). *Simulac* and *Deduce*. Available via the World Wide Web at `http://gobi.lbl.gov/~aparkin/Stuff/Software.html`.

Ascher, D., Dubois, P. F., Hinsen, K., Hugunin, J., and Oliphant, T. (2001). Numerical Python. Available via the World Wide Web at `http://www.numpy.org`.

Bray, D., Firth, C., Le Novère, N., and Shimizu, T. (2001). *StochSim*. Available via the World Wide Web at `http://www.zoo.cam.ac.uk/comp-cell/StochSim.html`.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). Extensible markup language (XML) 1.0, W3C recommendation 10-February-1998. Available via the World Wide Web at `http://www.w3.org/TR/1998/REC-xml-19980210`.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

Finney, A., Hucka, M., Sauro, H. M., and Bolouri, H. (2001). Systems Biology Workbench C programmer's manual. Available via the World Wide Web at `http://www.sbw-sbml.org/`.

Ginkel, M., Kremling, A., Tränkle, F., Gilles, E. D., and Zeitz, M. (2000). Application of the process modeling tool ProMot to the modeling of metabolic networks. In Troch, I. and Breitenecker, F., editors, *Proceedings of the 3rd MATHMOD*, pages 525–528.

Goryanin, I. (2001). *DBsolve*: Software for metabolic, enzymatic and receptor-ligand binding simulation. Available via the World Wide Web at `http://homepage.ntlworld.com/igor.goryanin/`.

Goryanin, I., Hodgman, T. C., and Selkov, E. (1999). Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics*, 15(9):749–758.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001a). Introduction to the Systems Biology Workbench. Available via the World Wide Web at `http://www.sbw-sbml.org/`.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001b). Systems Biology Markup Language (SBML) Level 1: Structures and facilities for basic model definitions. Available via the World Wide Web at `http://www.sbml.org`.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2002a). Systems Biology Workbench Java™ programmer's manual. Available via the World Wide Web at `http://www.sbw-sbml.org/`.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., and Kitano, H. (2001c). The ERATO Systems Biology Workbench: Architectural evolution. In Yi, T.-M., Hucka, M., Morohashi, M., and Kitano, H., editors, *Proceedings of the Second International Conference on Systems Biology*, pages 352–361. Omnipress, Inc.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., and Kitano, H. (2002b). The ERATO Systems Biology Workbench: Enabling interaction and exchange between software tools for computational biology. In Altman, R. B., Dunker, A. K., Hunker, L., Lauderdale, K., and Klein, T. E., editors, *Pacific Symposium on Biocomputing 2002*. World Scientific Press.

Mendes, P. (1997). Biochemistry by numbers: Simulation of biochemical pathways with Gepasi 3. *Trends in Biochemical Sciences*, 22:361–363.

Mendes, P. (2001). Gepasi 3.21. Available via the World Wide Web at `http://www.gepasi.org`.

Morton-Firth, C. J. and Bray, D. (1998). Predicting temporal fluctuations in an intracellular signalling pathway. *Journal of Theoretical Biology*, 192:117–128.

Musser, D. R. and Saini, A. (1996). *STL Tutorial and Reference Guide*. Addison Wesley.

Sauro, H. M. (2000). Jarnac: A system for interactive metabolic analysis. In Hofmeyr, J.-H. S., Rohwer, J. M., and Snoep, J. L., editors, *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*. Stellenbosch University Press.

Sauro, H. M. and Fell, D. A. (1991). SCAMP: A metabolic simulator and control analysis program. *Mathl. Comput. Modelling*, 15:15–28.

Sauro, H. M., Hucka, M., Finney, A., and Bolouri, H. (2001). The Systems Biology Workbench concept demonstrator: Design and implementation. Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs/detailed-design/`.

Schaff, J., Slepchenko, B., and Loew, L. M. (2000). Physiological modeling with the Virtual Cell framework. In Johnson, M. and Brand, L., editors, *Methods in Enzymology*, volume 321, pages 1–23. Academic Press, San Diego.

Schaff, J., Slepchenko, B., Morgan, F., Wagner, J., Resasco, D., Shin, D., Choi, Y. S., Loew, L., Carson, J., Cowan, A., Moraru, I., Watras, J., Teraski, M., and Fink, C. (2001). Virtual Cell. Available via the World Wide Web at `http://www.nrcam.uchc.edu`.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J. C., and Hutchison, C. (1999). E-Cell: Software environment for whole cell simulation. *Bioinformatics*, 15(1):72–84.

Tomita, M., Nakayama, Y., Naito, Y., Shimizu, T., Hashimoto, K., Takahashi, K., Matsuzaki, Y., Yugi, K., Miyoshi, F., Saito, Y., Kuroki, A., Ishida, T., Iwata, T., Yoneda, M., Kita, M., Yamada, Y., Wang, E., Seno, S., Okayama, M., Kinoshita, A., Fujita, Y., Matsuo, R., Yanagihara, T., Watari, D., Ishinabe, S., and Miyamoto, S. (2001). E-Cell. Available via the World Wide Web at `http://www.e-cell.org/`.