Systems Biology Workbench C Programmer's Manual

Andrew Finney, Michael Hucka, Herbert Sauro, Hamid Bolouri

{afinney,mhucka,hsauro,hbolouri}@cds.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
http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

February 26, 2002

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, 2002) and the description by Sauro et al. (2001) of a prototype SBW implementation. This document assumes that the reader is familiar with the C programming language. Documentation on programming SBW in the C++ (Finney et al., 2001a) and Java (Finney et al., 2001b) 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, 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. It 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.

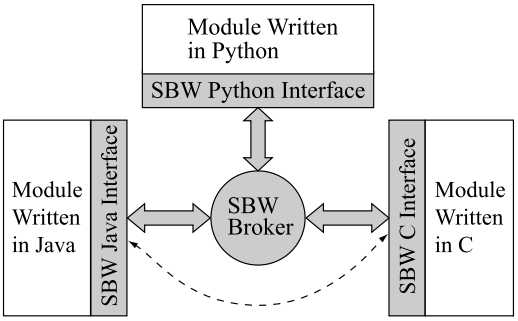

Figure 1: *The overall organization of the Systems Biology Workbench. Gray areas indicate SBW components (libraries and the broker). To individual modules, communications appear to be direct (dotted line), although they actually pass through the broker.*

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 dynamically querying SBW 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, SBW 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 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 the fly, 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 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 a known interface 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 either `\ or /` characters.

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. 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`

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
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.

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

This section describes how to use the Visual C++ Interactive Development Environment on Windows and `gcc` on Linux to develop SBW modules.

The C API is defined in the *sbwc.h* include file which should be included in C source files which use the C++ API.

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:

- Make sure to use the flags `-DLINUX -D_GNU_SOURCE` when compiling either C++ or C source code 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.*

- Use the `-I` flag to tell the compiler where to find the SBW include files.
- 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 and release versions of the SBW library. The release library is provided in the sbw.lib and sbw.dll files. The debug library is provided in the sbwd.lib and sbwd.dll files.

Visual C++ projects that use SBW should include *sbwc.h* and link with `sbw.lib` or `sbwd.lib`. This can be achieved first by changing 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 main “Project” menu.
2. the “Project Settings” dialog appears
3. in the drop down list “Settings For:” select “Win32 Debug”
4. select the “Link” tab
5. 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”.
6. in the drop down list “Settings For:” select “Win32 Release”
7. select the “Link” tab
8. 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”.
9. 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. 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.

The source code for these tutorials is available in the SBW installation and CVS directory in:

```
src/tutorials/C
```

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/TrigServerModule
```

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:

```
#ifdef WIN32
#include "windows.h"
#endif

#include "SBWC.h"
#include "math.h"

/*
* sinMethod
* sin method implementation for trig service on module edu.caltech.trig
* (all method implementations in C have the same C argument and result
* types as this function) this method has the SBW signature:
* double sin(double)
*/
void sinMethod(
    SBWInteger from,                /* id of module calling this method */
    SBWDataBlockReader *argObject,  /* pointer to object containing argument data */
    SBWDataBlockWriter *resultObject, /* pointer to object which will contain the
                                         result data */
    void *userData                  /* data supplied to SBW via
                                         SBWModuleImplSetHandler */)
{
    SBWDouble arg, result; /* variables to hold arguments and result */

    /* extract argument from argument data object */
    if (!SBWRead(argObject, "double", &arg))
        return;

    /* calculate result */
    result = sin(arg);

    /* store result into result data object */
    SBWWrite(resultObject, "double", result);
}

/*
* cserver
* implementation of SBW module edu.caltech.trig
* arguments: argc and argv same as those supplied to standard C main function
* result: int - 0 => SBW error has occurred, 1 => successful completion
*/
int cserver(int argc, char* argv[])
{
    if (!SBWCreateModuleImpl(
            "edu.caltech.trig",                 /* module identification */
            "trigonometry server C implementation", /* humanly readable name */
            SBW_SelfManagedModule,              /* management scheme */
            "does trigonometry"                 /* module help text */))
        return 0 ;

    if (!SBWModuleImplAddService(
            "trig",              /* service identification */
            "trigonometry",      /* service humanly readable name */
            "trig",              /* service category */
            "does trigonometry"  /* service help text */))
```

```
        return 0;

    if (!SBWModuleImplSetHandler(
            "trig",               /* service identification */
            sinMethod,            /* method implementation function */
            0,                    /* implementation data */
            "double sin(double)", /* method signature */
            0,                    /* synchronized 0 implies false */
            "sin"                 /* method help text */))
        return 0 ;

    /* connect to broker providing services */
    if (!SBWModuleImplRun(argc, argv, 1)) /* 1 => wait for disconnect */
        return 0 ;

    return 1 ;
}

/*
* 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 ;

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

    if (!cserver(argc, argv))
    {
#ifdef WIN32
        /* display error dialog */
        MessageBox(
            NULL, SBWExceptionGetMessage(), "Trig", MB_OK | MB_ICONEXCLAMATION);
#else
        fprintf(stderr, "cserver: %s\n", SBWExceptionGetMessage());
#endif
        return -1;
    }

    return 0;
}
```

5.1.1 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. The example code 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 ;

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

In the Windows case the function `SBWWindowsExtractCommandLine` extracts the command line. This function is only available on the windows platform.

5.1.2 Creating method implementations

First let us create an underlying implementation of the service functionality. In C each method is implemented in a separate function. For this example the sin method is implemented by a function with the following C signature:

```
void sinMethod(
    SBWInteger from,
    SBWDataBlockReader *argObject,
    SBWDataBlockWriter *resultObject,
    void *userData)
```

The `from` parameter gives the identity of the calling module instance. `argObject` contains argument data for the method. `resultObject` will contain the result data from the function. `userData` is arbitrary data that is passed to the SBW API with the function.

The first operation in this function is to extract the arguments to the method from the container containing the arguments:

```
SBWRead(argObject, "double", &arg)
```

In this call the second parameter is a string containing comma separated types. This sequence of types indicates the types of the arguments to the method. For each argument to the method a pointer should be passed to `SBWRead`. Each of these pointers points to a variable which will contain the argument data. These variables should match the types given in the type sequence. The type sequence string should conform to the `ArgumentList` production rule in the syntax given in section 3.3.

After calculating the method result value the function then stores the value in the result container:

```
SBWWrite(resultObject, "double", result)
```

The second parameter to this function is a string containing one type: the type of the return value. The third parameter is a value corresponding to that type. The `sinMethod` function completes by returning the pointer to the result container.

Both `SBWWrite` and `SBWRead` return the boolean value false if an error occurs during their execution. If an error occurs during the execution of `SBWRead` the `sinMethod` function simply returns immediately.

A function like `sinMethod`, that implements a SBW method can raise an error condition by calling `SBWSetException`. `SBWSetException` has 2 `char *` parameters. The first parameter is a message to be displayed to a user. The second parameter is a message for a developer to understand the background to the error.

In many situations, services need to invoke methods on a module instance which previously called the service. For example a simulator may wish to callback a module when a simulation has completed. This can be achieved by storing or using the `from` argument to method

implementation functions.

To disconnect a module from the broker from within a method function call use

```
SBWSignalDisconnect();
```

instead of

```
SBWDisconnect();
```

5.1.3 Creating a server module

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

The code to perform this in our example is contained in the function `cserver`. This function takes as arguments `argc` and `argv` arguments passed to the executable. `cserver` returns 0 if an error occurs, 1 if the function is successful. All the API calls in `cserver` return boolean results: false indicating an error and true indicating success.

`cserver` provides the code for both the register and module modes of operation for the module. The API does not require the developer to make a distinction between the two modes. `cserver` does not have any conditional code: it simply passes `argc` and `argv` which contain mode flags to the API.

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

```
SBWCreateModuleImpl(
           "edu.caltech.trig",
           "trigonometry server C implementation",
           SBW_SelfManagedModule,
           "does trigonometry")
```

In this call the first parameter is the identifier for the module which should be unique amoungst all modules installed. The SBW development group recommend that developers use the reversed internet domain name scheme used in this example. The second parameter is the humanly readable name for the module and can be potentially displayed to the user. The third parameter is the module type, which can be `SBW_SelfManagedModule` or `SBW_UniqueModule`. The last parameter is a help string which describes the module to users.

`SBW_UniqueModule` indicates that one module instance will be shared by all applications which access the module. `SBW_SelfManagedModule` indicates that a new module instance will be created every time a module is accessed through `SBWGetModuleInstance` 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 `SBWModuleShutdown` as shown in section 5.2.

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

```
SBWModuleImplAddService(
           "trig",
           "trigonometry",
           "trig",
           "does trigonometry")
```

The first argument is the identification name for the service and must unique amongst the services implemented by this module. The second argument is a humanly readable name for the service. The third argument is the category for the service. The last argument is a help string for the service.

The next step is to attach the `sinMethod` method function to the service:

```
SBWModuleImplSetHandler("trig", sinMethod, 0, "double sin(double)", 0, "sin")
```

The first parameter to the `SBWModuleImplSetHandler` function is the identification name of the service. The second parameter is a pointer to the function that implements the method. The third parameter is data to be passed to the method implementation function when it is called. The fourth parameter is the signature of the method using the syntax defined in Section 3.3. The fifth parameter is a boolean value. The methods that are registered with this value set true (1) will not be invoked concurrently with each other. The final parameter is a help string for the method.

The final part of `cserver` is the call:

```
SBWModuleImplRun(argc, argv, 1)
```

The `SBWModuleImplRun` function either registers the module with the broker or starts the provision of services by this module. The first 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.

The last parameter to `SBWModuleImplRun` is a boolean value which indicates whether this function, in module mode, blocks until the module is shutdown. This value is typically left true (1) 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 (0) and place the message loop after the `SBWModuleImplRun` 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.

5.1.4 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 itself 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

```
SBWGetModuleInstance("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. To achieve this a client application is required. Section 5.2 describes a simple C client for the module described here. The SBW Python extension can be used as a generic client for all modules. In fact for some modules specific clients will not be appropriate.

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 `SBW_SelfManagedModule` to `SBW_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 `SBW_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.trig". An implementation of this service is described in Section 5.1.

The complete code for this example is contained in the directory

```
src/tutorials/C/SimpleTrigClientModule
```

in the SBW installation and CVS tree.

This executable simply takes a single command line argument, a 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 the example follows:

```
#include "stdio.h"
#include "SBWC.h"

/* callTrig
* starts new instance of trig, fetches and stores identity of trig module,
* trig service and sin method.  Then calls the sin method on the trig service
* given the value in the given string argument.
* returns: SBWBoolean - 1 => successful, 0 => error occurred
*/
SBWBoolean callTrig(const char *valueStr)
{
    SBWInteger trigModuleId, trigServiceId, sinMethodId ;
    SBWDouble argument, result ;
    float x = 0, y ;

    /* connect to SBW Broker */
    if (!SBWConnect())
        return 0 ;

    /* fetch trig module identity */
    trigModuleId = SBWGetModuleInstance("edu.caltech.trig");

    if (trigModuleId == -1)
        return 0 ;

    /* fetch service identity */
    trigServiceId = SBWModuleFindServiceByName(trigModuleId, "trig");

    if (trigServiceId == -1)
        return 0 ;

    /* fetch method identity */
    sinMethodId = SBWServiceGetMethod(trigModuleId, trigServiceId, "sin");

    if (sinMethodId == -1)
        return 0;
```

```
    /* convert string to float */
    sscanf(valueStr, "%f", &x);
    argument = x ;

    /* call sin method */
    if (!SBWMethodCall(
            trigModuleId,
            trigServiceId,
            sinMethodId,
            "double sin(double)", /* method signature used to decode variable
                                     arguments and result */
            argument, /* variable number of arguments (only one in this case) */
            &result))
    {
        return 0;
    }

    /* print result float */
    y = result ;
    printf("%g\n", y);

    /* disconnect from trig module */
    if (!SBWModuleShutdown(trigModuleId))
        return 0 ;

    /* disconnect from SBW broker */
    SBWDisconnect();

    return 1;
}

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

    if (!callTrig(argv[1]))
    {
        fprintf(stderr, "cserver: %s\n", SBWExceptionGetMessage());
        return -1;
    }

    return 0;
}
```

In this example the `callTrig` function connects to SBW, using the `SBWConnect` function, executes one method on another module, and then disconnects from SBW, by invoking the `SBWDisconnect` 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.

In the C API certain values that API functions return indicate an error or exception has occurred during the function's execution. The actual value used depends on the function called. An error message corresponding to the exception can be obtained by calling

```
SBWExceptionGetMessage();
```

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

```
trigModuleId = SBWGetModuleInstance("edu.caltech.trig");
```

Whether or not a new module instance is started by these 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, if no such module instance exists a new instance is started.

The next step is to obtain the identity of the service on the module instance, using the module instance identity obtained in the previous call. This is performed by the following code:

```
trigServiceId = SBWModuleFindServiceByName(trigModuleId, "trig");
```

Now we can consider calling the `sin` method on this service. First we obtain the identity of the method which we would like to call, using the service and module identities we have obtained:

```
sinMethodId = SBWServiceGetMethod(trigModuleId, trigServiceId, "sin");
```

All these functions to obtain the identities of SBW objects return `-1` if an error occurred during their execution.

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

```
SBWMethodCall(
    trigModuleId,
    trigServiceId,
    sinMethodId,
    "double sin(double)",
    argument,
    &result)
```

The first three parameters indicate the method which we would like to call. The fourth parameter is the signature of the method which is used to interpret the remaining parameters which can vary in number and type. The remaining parameters up to and including the penultimate parameter are the arguments to be passed to the method. In this example there is only one argument: `argument`. The last parameter is a pointer to a variable in which the method return value is to be stored.

The C API uses the type `SBWBoolean` to represent boolean values. `SBWBoolean` is just a type alias of `int`. As by C convention 0 represents false and any non zero value represents true. `SBWMethodCall` returns boolean value to indicate whether it was successful or not.

`SBWMethodCall` call can return false, i.e. indicate an error, if the method implementation on the called module raises an error during the execution of the method. These errors are called application exceptions.

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 code:

```
SBWModuleShutdown(trigModuleId)
```

This function returns a boolean value to indicate whether it was successful or not.

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

```
SBWDisconnect()
```

The above code shows how a method is called on a blocking basis; i.e., `SBWMethodCall` waits to receive a response from the method implementation. In SBW, it is possible to call a method

on a non-blocking basis; i.e., the following API function returns as soon as a message is sent to the method implementation:

```
SBWBoolean SBWMethodSend(
    SBWInteger moduleInstanceId,
    SBWInteger serviceId,
    SBWInteger method,
    const char *signature,
    ...);
```

The variable parameter list should not contain a parameter for the method return value.

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 shutdown 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 <string.h>
#include <malloc.h>
#include <stdio.h>
#include "SBWC.h"

SBWBoolean doit(int argc, char* argv[])
{
    FILE *sbmlSource;
    char *sbml = NULL;
    int size = 1000;
    int i = 0 ;
    int total = 0;
    SBWServiceDescriptor *serviceDescriptors;
    SBWInteger module, service, method, numOfServiceDescriptors;

    if (argc < 2)
    {
        printf("usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // read file
    sbmlSource = fopen(argv[1], "r");

    if (!sbmlSource)
    {
        printf("unable to open file %s\n", argv[1]);
        return 1;
```

```
}

while( !feof( sbmlSource ) )
{
    char *newBuffer = malloc(total + size);

    if (sbml != NULL)
        strncpy(newBuffer, sbml, total);

    sbml = newBuffer ;
    /* Attempt to read in 1000 bytes: */
    total += fread( sbml + total, sizeof( char ), size, sbmlSource );

    if (ferror( sbmlSource ) )
    {
        printf("Problem reading SBML");
        return 1;
    }
}

fclose(sbmlSource);
sbml[total] = '\0';

// connect to SBW broker
if (!SBWConnect())
    return 0;

// query broker for services in Analysis category
serviceDescriptors = SBWFindServices("Analysis", &numOfServiceDescriptors, 1);

if (serviceDescriptors == NULL)
    return 0;

// create menu
while (i != numOfServiceDescriptors)
{
    printf("%d %s\n", i, serviceDescriptors[i].serviceDisplayName);
    i++;
}

printf("\nType number to select service\n") ;

// get user response
scanf("%d", &i);

// check response
if (i < 0 || i >= numOfServiceDescriptors)
{
    printf("Incorrect selection\n");
    SBWDisconnect();
    return 1;
}

// locate/create module instance based on selected service
if (!SBWGetServiceInModuleInstance(serviceDescriptors + i, &module, &service))
    return 0;

SBWFreeServiceDescriptorArray(numOfServiceDescriptors, serviceDescriptors);

// locate method
method = SBWServiceGetMethod(module, service, "void doAnalysis(string)");

if (method == -1)
    return 0;
```

```
    // call method
    if (!SBWMethodCall(module, service, method, "void doAnalysis(string)", sbml))
        return 0;

    // disconnect from broker
    SBWDisconnect();

    return 1;
}

int main(int argc, char* argv[])
{
    if (!doit(argc, argv))
    {
        printf("Exception:\n");
        printf(SBWExceptionGetMessage());
        printf("\n");
        printf(SBWExceptionGetDetailedMessage());
        printf("\n");
    }
}
```

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 a string buffer pointed to by `sbml`. After connecting the following code fetches information on services in the "Analysis" category:

```
serviceDescriptors = SBWFindServices("Analysis", &numOfServiceDescriptors, 1);

if (serviceDescriptors == NULL)
    return 0;
```

where the argument variables are defined as

```
SBWServiceDescriptor *serviceDescriptors;
SBWInteger numOfServiceDescriptors;
```

`SBWServiceDescriptor` structures 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:

```
while (i != numOfServiceDescriptors)
{
    printf("%d %s\n", i, serviceDescriptors[i].serviceDisplayName);
    i++;
}
```

where `i` is initially 0.

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

```
scanf("%d", &i);

if (i < 0 || i >= numOfServiceDescriptors)
{
    printf("Incorrect selection\n");
    SBWDisconnect();
    return 1;
}
```

The variable `i` now contains the index of the service that the user has selected. We now obtain a numeric service identifier for the service represented by the selected `SBWServiceDescriptor` as follows:

```
if (!SBWGetServiceInModuleInstance(serviceDescriptors + i, &module, &service))
    return 0;
```

where the new argument variables are defined as

```
SBWInteger module, service;
```

The function `SBWGetServiceInModuleInstance` 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 array of `SBWServiceDescriptor` can now be destroyed as its no longer required:

```
SBWFreeServiceDescriptorArray(numOfServiceDescriptors, serviceDescriptors);
```

Now we have a reference to a service we can call the `doAnalysis` method on it as described in the previous tutorial:

```
method = SBWServiceGetMethod(module, service, "void doAnalysis(string)");

if (method == -1)
    return 0;

if (!SBWMethodCall(module, service, method, "void doAnalysis(string)", sbml))
    return 0;
```

The application finishes by disconnecting from the broker and terminating.

6 C API Reference

6.1 Types

The C API uses the types shown in tables 2, 3, 4, and 5.

6.2 Object and Array ownership

In the C API all the objects returned to the calling environment are owned by the calling environment. All arguments passed to the C API remain in the ownership of the calling environment. In short the API caller is responsible for recovering all memory allocated by the caller and returned to the caller by the SBW API.

The following functions to recover memory are part of the C API:

```
void SBWFree(void *)
```

this function recovers the memory used by a single object or a single dimension array returned by the C API. Single dimension arrays of char * should be recovered using `SBWFree2DArray`.

```
void SBWFree2DArray(int xSize, void **)
```

This function recovers the memory used by a 2D array returned by the C API. Single dimension arrays of char * should be recovered using this function. `xSize` is the size of the first dimension of the array.

```
void SBWFreeServiceDescriptor(SBWServiceDescriptor *)
```

this function frees a single service descriptor structure

```
void SBWFreeServiceDescriptorArray(SBWInteger size, SBWServiceDescriptor *)
```

Type name	Definition
SBWModuleDescriptor	`typedef struct { char *name; /* for identification */ char *displayName; SBWModuleManagementType managementType ; char *commandLine ; /* to start the module */ char *help ; } SBWModuleDescriptor;` stores information contained in the registry for a given module
SBWServiceDescriptor	`typedef struct { char *serviceName; /* for identification */ char *serviceDisplayName; char *serviceCategory; SBWModuleDescriptor module; char *help; } SBWServiceDescriptor ;` stores information contained in the registry for a given module
SBWBoolean	32 bit integer holding a boolean value, 0 for false, true otherwise.
SBWInteger	This is an signed 32bit integer
SBWDouble	This is a 64 bit IEEE double precision floating point number
SBWDataBlockWriter	This is a container for data to be transmitted out of a module. Operations on `SBWDataBlockWriter` are defined in section 6.11.1.
SBWDataBlockReader	This is a container for data to be received by a module. Operations on `SBWDataBlockReader` are defined in section 6.11.1.
SBWSignature	Contains the structure of a signature
SBWSignatureElement	Contains type and possibly the name of an argument
SBWSignatureType	Contains information of a type in a signature

Table 2: *C API scalar and structure types*

this function frees an array of service descriptor structures; `size` is the number of descriptors in the array.

```
void SBWFreeModuleDescriptor(SBWServiceDescriptor *)
```

this function frees a single service descriptor structure

```
void SBWFreeModuleDescriptorArray(SBWInteger size, SBWServiceDescriptor *)
```

this function frees an array of service descriptor structures; `size` is the number of descriptors in the array.

```
void SBWFreeSBWDataBlockWriter(SBWDataBlockWriter *)
```

this function frees a SBWDataBlockWriter structure

Type name	Definition

SBWExceptionType	An enumeration of exception types
	`typedef enum sbwExceptionCode`
	`{`
	`SBWApplicationExceptionCode = 0,`
	`SBWRawExceptionCode = 1,`
	`SBWCommunicationExceptionCode = 2,`
	`SBWModuleStartExceptionCode = 3,`
	`SBWTypeMismatchExceptionCode = 4,`
	`SBWMethodSignatureNotCompatableExceptionCode = 5,`
	`SBWIncorrectModuleIdentifierSyntaxExceptionCode = 6,`
	`SBWIncorrectCategorySyntaxExceptionCode = 7,`
	`SBWServiceNameIsNotUniqueToModuleExceptionCode = 8,`
	`SBWServiceIsNotFoundExceptionCode = 9,`
	`SBWMethodTypeNotBlockTypeExceptionCode = 10,`
	`SBWMethodAmbiguousExceptionCode = 11,`
	`SBWUnsupportedObjectTypeExceptionCode = 12,`
	`SBWMethodNotFoundExceptionCode = 13,`
	`SBWSignatureSyntaxExceptionCode = 14,`
	`SBWModuleDefinitionExceptionCode = 15,`
	`SBWModuleNotFoundExceptionCode = 16,`
	`SBWUnknownExceptionCode = ~0`
	`} SBWExceptionType ;`

Table 3: *The C API Exception Enumeration type*

Type name	Definition

SBWModuleManagementType	An enumeration of the various management modes.
	`typedef enum sbwmoduleManagementType`
	`{`
	`SBW_UniqueModule = 0,`
	`SBW_SelfManagedModule = 1`
	`} SBWModuleManagementType;`

SBWDataBlockType	An enumeration of the various data block types.
	`typedef enum sbwDataBlockType {`
	`SBWIntegerType, SBWDoubleType,`
	`SBWStringType, SBWArrayType,`
	`SBWListType, SBWBooleanType,`
	`SBWByteType, SBWTerminateType,`
	`SBWErrorType, SBWVoidType`
	`} SBWDataBlockType;`
	where `SBWTerminate` marks the end of the data block and `SBWErrorType` is returned by functions when an error occurs.

Table 4: *The C API `SBWModuleManagementType` and `SBWDataBlockType` Enumeration Types*

```
void SBWFreeSBWDataBlockReader(SBWDataBlockReader *)
```

this function frees a SBWDataBlockReader structure

```
void SBWFreeSBWSignature(SBWSignature *)
```

this function frees a SBWSignature structure

```
void SBWFreeSBWSignatureElementArray(SBWInteger size, SBWSignatureElement **)
```

Type name	Definition

SBWHandler	`typedef void (*SBWHandler)(`
	`  SBWInteger from,`
	`  SBWDataBlockReader *args,`
	`  SBWDataBlockWriter *result,`
	`  void *userData);`
	This type defines the form of method handlers for the C API. `from` is the numeric identifier of the module calling this method. `args` contains the arguments to the method. `userData` is a value passed to the API when the `Handler` is attached to a service. Functions of this type are expected to put the result data into the container `result`. Exception data should be passed to the library by calling the `SBWSetException` function within the function.

SBWModuleListener	`typedef void (*SBWModuleListener)`
	`  (SBWInteger moduleInstanceIdentifier);`
	This type defines the form of module event callbacks for the C API. `moduleInstanceIdentifier` is the numeric identifier of a module. See section 6.13 for details of how to attach functions of this type to the SBW API.

SBWSimpleListener	`typedef void (*SBWSimpleListener)();` This type defines the form of simple event callbacks for the C API. See section 6.13 for details of how to attach functions of this type to the SBW API.

Table 5: *C API function types*

this function frees a SBWSignatureElement array; `size` is the number of signature arguments in the array.

```
void SBWFreeSBWSignatureType(SBWSignatureType *)
```

this function frees a SBWSignatureType structure

6.3 Connecting and Disconnecting from the Broker

The following functions are part of the C API:

```
char *SBWGetVersion()
```

returns the version string of the C/C++ API library.

```
SBWBoolean SBWConnect()
```

connects this application to the SBW system as a module that does not provide services. returns false if an exception occurs, true otherwise.

```
void SBWDisconnect()
```

notifies the SBW system that this application process is no longer available to provide service and does not need to consume services. This function returns only when the library has disconnected from the broker.

```
void SBWWaitForDisconnect()
```

This function returns only when the library has disconnected from the broker.

```
void SBWSignalDisconnect()
```

notifies the SBW system that this application process is no longer available to provide, and does not need to consume services and then returns immediately.

6.4 Module and Service Discovery functions

The following functions are part of the C API:

```
SBWInteger SBWGetModuleInstance(const char *identificationName)
```

returns the module identification number for an instance of a module with a given identification name. returns -1 if an exception occurs.

Whether this function starts a new module instance depends on the management type of the module.

```
SBWInteger SBWGetThisModule()
```

returns the module identification number for this module.

```
SBWServiceDescriptor *SBWFindServices(
    const char *serviceCategory, SBWInteger *numberOfServices, SBWBoolean recursive)
```

returns an array of service records for all the services registered with the SBW system in the given category. returns null when an exception occurs. The integer pointed to by the `numberOfServices` argument, is set to the number of elements in the returned array.

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.

```
SBWBoolean SBWGetServiceInModuleInstance(
    SBWServiceDescriptor *service,
    SBWInteger *moduleIdentifier,
    SBWInteger *serviceIdentifier)
```

sets the integer pointed to by `moduleIdentifier` to the identifier of an instance of the module described in the service descriptor. This function sets the integer pointed to by the `serviceIdentifier` argument, to the service, on the module instance, that is described in the service descriptor. This function returns false if an exception occurs.

Whether this function starts a new module instance depends on the management type of the module.

```
char **SBWGetServiceCategories(
    const char *serviceCategory, SBWInteger *numberOfCategories)
```

returns an array of the service categories registered with the SBW system in the given category and subcategories. Returns null when an exception occurs. The integer pointed to by `numberOfCategories` is set to the number of elements in the returned array. Use the function `SBWFree2DArray` to recover the result.

```
SBWInteger *SBWGetExistingNamedModuleInstances(
    const char *moduleIdName, SBWInteger *numberOfModuleInstances)
```

returns an array of module identifiers for running module instances with identification name `moduleIdName`. The parameter `numberOfModuleInstances` is set to the number of identifiers in the array. Returns NULL if an exception has occurred.

```
SBWInteger *SBWGetExistingModuleInstances(SBWInteger *numberOfModuleInstances)
```

returns an array of module identifiers for all running module instances. The parameter `numberOfModuleInstances` is set to the number of identifiers in the array. Returns NULL if an exception has occurred.

```
SBWModuleDescriptor *SBWGetModuleDescriptors(
    SBWBoolean includeRunning, SBWInteger *numberOfModuleDescriptors);
```

returns information from the broker on all the modules known to the broker. The parameter `numberOfModuleDescriptors` is set to the number of descriptors in the returned array. This function returns data on all the modules registered with the broker and if the parameter `includeRunning` is true includes data on unregistered running modules in the returned array.

6.5 Accessing Application Command Line

```
char *SBWCalculateCommandLine(const char *command)
```

returns the command line for registration of the given command in module mode. Normally `command` is `argv[0]` passed to this application and the returned value is passed to `SBWModuleImplSetCommandLine`. Returns `NULL` if an exception occurs.

This function is only useful if you wish to modify the command line registered for a given module in the broker.

```
void SBWWindowsExtractCommandLine(int *argc, char ***argv)
```

Specific to Windows. Sets `argc` and `argv` to values corresponding to those passed to an ANSI `main` function for this application. To be used inside Windows non-console applications.

6.6 Module and Service registration functions

The following functions are in the API:

```
SBWBoolean SBWCreateModuleImpl(
    const char *uniqueName,
    const char *nameForDisplay,
    SBWModuleManagementType type,
    const char *help);
```

This function creates a new module implementation. `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 documentation for this module.

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

```
SBW_UniqueModule = 0,
SBW_SelfManagedModule = 1,
```

This function returns false if an exception occurs, true otherwise.

```
SBWBoolean SBWModuleImplRegister()
```

registers this module with the SBW Broker, adding a persistent record of how to start the module and list of services implemented by the module. Returns true if the function is successful and false if the function raises an exception.

```
SBWBoolean SBWModuleImplEnableServices()
```

notifies the SBW Broker that this module's services are operational and handles calls methods on these services. Returns true if the function is successful and false if the function raises an exception.

```
SBWBoolean SBWModuleImplSetCommandLine(const char *commandLine)
```

sets the command line required to start the module in module mode. Returns true if the function is successful and false if the function raises an exception.

```
public SBWBoolean SBWModuleImplSetHandler(
    const char *serviceName,
    SBWHandler handler,
    void *userData,
    const char *signature,
    SBWBoolean synchronized,
    const char *help);
```

notifies SBW that the given handler is the implementation of the given method on the given service. `userData` is passed to the handler on the method's invocation. `signature` is a method signature (see Section 3.3). `synchronized`, a boolean parameter, indicates whether the method must not be run concurrent with other methods that have been registered with this parameter set to 1. The `synchronized` parameter is equivalent to the Java `synchronized` keyword. `help` is documentation for the method.

This function returns false if an exception occurs, true otherwise.

```
public SBWHandler SBWModuleImplGetHandler(
    const char *serviceName, const char *methodName, void **userData);
```

returns a handler which is the implementation of the given method on the given service. `userData` is set to whatever was passed to `setHandler`. Returns null if SBW has not been notified of this handler or if another exception occurs

```
SBWBoolean SBWModuleImplAddService(
    const char *serviceName, const char *serviceDisplayName,
    const char *category, const char *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` defines the interface class that this service complies with. `help` is documentation for this service. Returns false if an exception has occurred, true otherwise.

```
SBWBoolean SBWModuleImplRun(int argc, char **argv, SBWBoolean waitForDisconnect)
```

`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` is true then the method will block until the module is shutdown. This function returns false if an exception occurs.

6.7 Functions for accessing module instances

```
SBWModuleDescriptor *SBWModuleGetModuleDescriptor(SBWInteger moduleInstanceId);
```

returns information from the registry on the given module instance. This function returns null if an exception occurs during its execution.

```
SBWInteger SBWModuleGetNumberOfServices(SBWInteger moduleInstanceId)
```

returns the number of services that exist on the the given module. returns -1 when an exception occurs. Valid service identifier numbers for this module exist in the range 0 to $n-1$ where n is the value returned by this function.

```
SBWInteger SBWModuleFindServiceByName(
    SBWInteger moduleInstanceId, const char *serviceName)
```

returns the service identifier number for the named service on the given module. returns -1 on an exception.

```
SBWInteger *SBWModuleFindServicesByCategory(
    SBWInteger moduleInstanceId, const char *category, SBWInteger *numberOfServices)
```

returns an array of service identifier numbers for the services on the given module in the given category. returns null when an exception occurs. The integer pointed to by `numberOfServices` is set to the number of elements in the returned array. This function returns null when an exception occurs.

```
SBWBoolean SBWModuleShutdown(SBWInteger moduleInstanceId)
```

requests that the broker disconnects from the given module. Returns false if an exception occurs. This function should always be called when an application no longer requires the services of a given self managed module.

6.8 Functions for accessing services

```
SBWServiceDescriptor *SBWServiceGetDescriptor(
    SBWInteger moduleInstanceId, SBWInteger serviceId)
```

returns a service descriptor corresponding to the given service. Returns null if an exception occurs.

```
SBWInteger SBWServiceGetNumberOfMethods(
    SBWInteger moduleInstanceId, SBWInteger serviceId)
```

returns the number of methods on the given service. returns -1 if an exception occurs. A method id for the given service is valid if the method id is in the range 0 to $n-1$ where n is the value returned by this function.

```
SBWInteger SBWServiceGetMethod(
    SBWInteger moduleInstanceId, SBWInteger serviceId, const char *signature)
```

returns the method id corresponding to the given signature. returns -1 if an exception occurs.

6.9 Functions for accessing and invoking methods

```
char *SBWMethodGetName(
    SBWInteger moduleInstanceId, SBWInteger serviceId, SBWInteger methodId)
```

returns the name of the given method. Returns null if an exception occurs.

```
char *SBWMethodGetSignatureString(
    SBWInteger moduleInstanceId, SBWInteger serviceId, SBWInteger methodId)
```

returns the signature of the given method. Returns null if an exception occurs.

```
char *SBWMethodGetHelp(
    SBWInteger moduleInstanceId, SBWInteger serviceId, SBWInteger methodId)
```

returns documentation for the given method. Returns null if an exception occurs.

```
public SBWBoolean SBWMethodCall(
```

```
    SBWInteger moduleInstanceId,
    SBWInteger serviceId,
    SBWInteger methodId,
    const char *signature,
    ...);

public SBWBoolean SBWMethodSend(
    SBWInteger moduleInstanceId,
    SBWInteger serviceId,
    SBWInteger methodId,
    const char *signature, ...);
```

Send a message to the service, either blocking until the method has completed on the service in the case of `SBWMethodCall` or returning immediately in the case of `SBWMethodSend`.

`signature` indicates the form of the variable argument sequence. The syntax for this string is defined in section 3.3.

The variable argument sequence is divided into two halves: the first half consists of the call arguments and the second half consists of a pointer to a variable that can receive the result of the call. Only call arguments should be passed to `SBWMethodSend`.

`signature` indicates the types of the parameters passed in the variable argument list. The correspondence between the types in the signature string and C types is shown in table 1.

Arrays are passed as a sequence of parameters: an integer for each dimension, indicating the size of the dimension, followed by a pointer to the array itself. Array return values are treated the same way except that the size parameters should be pointers to integers followed by a pointer to an array pointer.

Lists are passed as arguments as `SBWDataBlockWriter *`. `SBWDataBlockReader **` parameters should be passed as the last argument to fetch a list return value. Lists can be members of arrays. Refer to section 6.11 for more details on manipulating these structures.

An example of two consistent calls to `SBWMethodCall` would be:

```
SBWInteger arg = 0;
SBWDouble result ;
SBWInteger sizeIn = 3, SizeOut;
SBWDouble argArray[] = { 0.0, 0.1, 0.2 };

if (!SBWMethodCall(module, service, method1, "double f(int)", arg, &result))
{
    /* Handle error */
}

if (!SBWMethodCall(
        module,
        service,
        method2,
        "int[] f(double[])",
        sizeIn,
        argArray,
        &sizeOut,
        &resultArray);
{
    /* Handle error */
}
```

`SBWMethodCall` and `SBWMethodSend` return false if an exception occurs.

```
SBWSignature *SBWMethodGetSignature(
    SBWInteger moduleInstanceId, SBWInteger serviceId, SBWInteger methodId)
```

returns a parse structure for the signature of this method, returns null if an exception occurred

6.10 Signatures

```
SBWSignatureElement **SBWSignatureGetArguments(
    SBWSignature *, SBWInteger *numberOfArguments)
```

Sets the `numberOfArguments` argument to the number of arguments in the array. This function returns the `SBWSignatureElement` array for the arguments in the signature.

```
SBWSignatureType *SBWSignatureGetReturnType(SBWSignature *)
```

returns the result type of the give signature.

```
char *SBWSignatureGetName(SBWSignature *)
```

returns the name part of a signature

```
char *SBWSignatureElementGetName(SBWSignatureElement *)
```

returns the name of the given argument, null if its not present

```
SBWSignatureType *SBWSignatureElementGetType(SBWSignatureElement *)
```

returns the type of the argument

```
SBWDataBlockType SBWSignatureTypeGetType(SBWSignatureType *)
```

returns the type of the given object

```
SBWSignatureType SBWSignatureTypeGetArrayInnerType(SBWSignatureType *)
```

returns the type of the objects containing inside the parameter assuming that the parameter is an array. Returns SBWErrorType if parameter not an array type

```
SBWInteger SBWSignatureTypeGetArrayDimensions(SBWSignatureType *)
```

returns the number of dimensions of the parameter assuming that the parameter is an array. Returns -1 if this is not an array type

```
SBWSignatureElement **SBWSignatureTypeGetListContents(
    SBWSignatureType *, SBWInteger *numberOfArguments)
```

returns the `SBWSignatureElement` array for the contents of the parameter assuming that the parameter is a list . sets `numberOfArguments` to the number of objects in the array. Returns NULL if the parameter is not a list type.

6.11 Serialization of Data

SBW uses data serialization to transmit argument and result data between callers and method implementations. Data is serialized into SBWDataBlockWriter structures and serialized out of SBWDataBlockReader structures. These structures are used recursively: SBW list data is stored into SBWDataBlockWriter structures and copied out of SBWDataBlockReader structures.

6.11.1 SBWDataBlockWriter

The interface to SBWDataBlockWriter consists of the following functions:

```
SBWDataBlockWriter *SBWCreateSBWDataBlockWriter();
```

Returns a pointer to a new empty `SBWDataBlockWriter` object.

```
SBWBoolean SBWWrite(SBWDataBlockWriter *, const char *argumentList, ...);
```

The ... refers to a variable argument list of values and objects to be appended to the SBWDataBlockWriter. The argument argumentList, which must in the syntax defined by the ArgumentList production in section 3.3, indicates what arguments have been supplied in variable argument list. Arrays should be added as a sequence of arguments: an integer for each dimension, indicating the size of the dimension, followed by a pointer to the array itself. Lists are passed, recursively, as SBWDataBlockWriter *.

SBWWrite returns false when an exception occurs.

For example the following is a consistent call to SBWWrite:

```
SBWInteger i;
SBWDouble d;
char *s = "hello";
SBWInteger a[4] = { 0, 1, 2, 3 };

if (!SBWWrite(SBWDataBlockWriter, "int, double, string, int[]", i, d, s, 4, a))
    /* handle error */
```

6.11.2 SBWDataBlockReader

The C interface to SBWDataBlockReader consists of the following function:

```
SBWBoolean SBWRead(SBWDataBlockReader *, const char *argumentList, ...);
```

The ... refers to a variable argument list of values and objects to be extracted from the SBWDataBlockReader. In the argumentList (see Section 3.3) the types, 'int', 'double', 'string', 'boolean', 'byte' and list, refer to the scalar types SBWInteger *, SBWDouble *, char **, SBWBoolean, unsigned char * and SBWDataBlockReader ** respectively in the variable argument list. The objects pointed to by the arguments in the variable argument list section are set by SBWRead.

SBWRead returns false when an exception occurs.

Arrays are extracted by supplying a sequence of arguments: an integer pointer for each dimension followed by a pointer to a pointer for the array. The integer values are set to the size of the array dimensions. The array pointer is set to point to the extracted array.

For example the following is a consistent call to SBWRead:

```
SBWInteger i;
SBWDouble d;
char *s = "hello";
SBWInteger *a;
SBWInteger sizeOfA;

if (!SBWRead(
        SBWDataBlockReader,
        "int, double, string, int[]",
        &i, &d, &s, &sizeOfA, &a))
    /* handle error */
```

The following functions allow a data block of an unknown format to be decoded.

```
SBWDataBlockType SBWGetNextType(SBWDataBlockReader *);
```

returns the type of the next item in the reader.

```
DataBlockType SBWGetNextArrayType(SBWDataBlockReader *);
```

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

```
SBWInteger SBWGetNextArrayDimensions(SBWDataBlockReader *);
```

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

6.12 Exceptions

The C library stores exceptions as they occur. The best information concerning an error will be stored in the pending exception after the function call where that error occurs. Subsequent API calls will replace that error information with exceptions that do not have the same contextual information as the original error. The functions below allow the currently stored exception to be examined. Use `SBWExceptionGetCode()` to find out if an exception is pending.

The following functions are in the C API to support exception handling:

```
char *SBWExceptionGetMessage();
```

returns the message associated with the currently stored exception

```
char *SBWExceptionGetDetailedMessage();
```

returns the detailed message associated with the currently stored exception

```
void SBWSetException(const char *userMessage, const char *developerMessage);
```

creates and stores an application exception condition. This function should be called from within a method implementation to indicate that an exception has occurred.

```
int SBWExceptionGetCode();
```

returns the exception type code for the pending exception. This function returns -1 if no exception is pending. The API provides an enumeration `SBWExceptionType` which provides identifiers for the various exception codes, 0 to 16, returned by this function (see Tables 6 and 7).

6.13 Events

The following API functions allow callback functions to be attached to the SBW library to allow SBW applications to respond to events occurring in the broker:

```
void SBWRegisterShutdownListener(SBWSimpleListener function)
```

Attaches the given function to the SBW library. Any number of functions can be attached. `function` will be called when the broker disconnects from this module.

```
void SBWRegisterModuleShutdownListener(SBWModuleListener function);
```

Attaches the given function to the SBW library. Any number of functions can be attached. `function` will be called whenever the broker disconnects from other modules.

```
void SBWRegisterModuleStartupListener(SBWModuleListener function);
```

Attaches the given function to the SBW library. Any number of functions can be attached. `function` will be called whenever the broker connects to other modules.

```
void SBWRegisterRegistrationChangeListener(SBWSimpleListener function);
```

Attaches the given function to the SBW library. Any number of functions can be attached. `function` will be called whenever the broker registry changes.

```
void SBWRemoveShutdownListener(SBWSimpleListener function);
```

Removes `function` that was attached to the API by `SBWRegisterShutdownListener`

#	Exception Code Identifier	Meaning
0	SBWApplicationExceptionCode	This is an Application-specific exception, thrown deliberately by a module. This is the only type of exception that can be created by a module in SBW.
1	SBWRawExceptionCode	Throw by API when platform exceptions occur and the context of the error has been lost
2	SBWCommunicationExceptionCode	Communications between a caller and receiver failed, possibly due to a lost connection.
3	SBWModuleStartExceptionCode	An attempt to start a new module failed.
4	SBWTypeMismatchExceptionCode	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.
5	SBWIncompatibleMethodSignatureExceptionCode	An interface or class definition uses method signatures that don't correspond to the signatures on the corresponding service.
6	SBWModuleIdSyntaxExceptionCode	Indicates that a supplied module instance identifier has incorrect syntax.

Table 6: *Exception Codes 0 to 6*

```
void SBWRemoveModuleShutdownListener(SBWModuleListener function);
```

Removes `function` that was attached to the API by `SBWRegisterModuleShutdownListener`

```
void SBWRemoveModuleStartupListener(SBWModuleListener function);
```

Removes `function` that was attached to the API by `SBWRegisterModuleStartupListener`

```
void SBWRemoveRegistrationChangeListener(SBWSimpleListener function);
```

Removes `function` that was attached to the API by `SBWRegisterRegistrationChangeListener`

A Standard Low Level Interfaces

This appendix describes the standard interfaces that are used by the libraries to implement the API described in this document.

A.1 Module System Service

All modules, including the broker, implement the service with index -1 which implements services which are not directly accessible by the API but are used to implement the API. The

#	Exception Code Identifier	Meaning
7	SBWIncorrectCategorySyntaxExceptionCode	A supplied category string has incorrect syntax. This is thrown by methods that find or search services by categories.
9	SBWServiceNotFoundExceptionCode	Requested service is not present on this module.
10	SBWMethodTypeNotBlockTypeExceptionCode	The supplied class uses types which are not SBW message data block types.
11	SBWMethodAmbiguousExceptionCode	The function SBWServiceGetMethod throws this exception to indicate that the given signature or signature component matches with more than one method on the given service.
12	SBWUnsupportedObjectTypeExceptionCode	The library encountered an object of a type that it cannot encode or decode from a message block.
13	SBWMethodNotFoundExceptionCode	The supplied 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.
14	SBWSignatureSyntaxExceptionCode	Thrown if a signature string does not contain a valid SBW signature.
15	SBWModuleDefinitionExceptionCode	Thrown if any aspect of a module definition is incorrect by the SBWCreateModuleImpl function .
16	SBWModuleNotFoundExceptionCode	Thrown if the given module identification name does not match any module known to SBW.

Table 7: *Exception Codes 7 to 16*

methods on this service have predefined method identification numbers.

```
string[] getServices()
```

returns the names of services implemented on the module. The index of each service gives the service identification number for the service. This method has method identification number 0.

```
string[] getMethods(int serviceId)
```

returns the method signatures of the given service. This method has method identification number 1.

```
void onOtherModuleInstanceShutdown(int moduleInstanceId)
```

called when another module shuts down. This method has method identification number 2.

```
void shutdown()
```

terminates the module. This method has method identification number 3.

```
string getMethodHelp(int serviceId, int methodId)
```

returns documentation for the given method. This method has method identification number 4.

```
void onOtherModuleInstanceStartup(int moduleInstanceId)
```

called when another module instances starts. This method has method identification number 5.

```
void onRegistrationChange(int moduleInstanceId)
```

called when the registration data in the broker changes. This method has method identification number 6.

A.2 Broker System Service

The SBW Broker behaves as ubiquitous module -1 which implements a service "SYSTEM" to support the module API. This service, which does not use hard coded method identification numbers:

```
int getModuleInstance(string moduleIdentificationName)
```

returns the module instance identifier to a instance of the given module, if the module instance is static then an existing instance identifier is returned otherwise an new instance launched. A new instance is always launched if no existing instance exists.

The Broker records that the caller is taking a reference to the returned module.

```
{
    string moduleIdentificationName,
    string moduleDisplayName,
    string serviceIdentificationName,
    string serviceDisplayName,
    string serviceCategory
}[] findServices(string category, bool recursive)
```

returns the service information for those services listed in the registry file in the given category.

```
int getLocalModuleInstanceId(string moduleInstanceIdentifier)
```

returns the local module identifier for a given module instance. the Broker records that the caller is taking a reference to the given module

```
string[] getServiceCategories(string category)
```

returns all the immediate subcategories of the given category (category can be empty to indicate the top level).

```
void release(int moduleId)
```

disconnects the broker from the module

```
{
    string moduleIdentificationName,
    string moduleDisplayName,
    int managementTypeCode,
    string commandLine,
    string help
```

```
} getModuleDescriptor(string moduleName, boolean includeRunning)
```

returns the module descriptor for the named module.

```
{
    string moduleIdentificationName,
    string moduleDisplayName,
    int managementTypeCode,
    string commandLine,
    string help
} getModuleDescriptor(int moduleId)
```

returns the module descriptor for the given module instance.

```
{
    string moduleIdentificationName,
    string moduleDisplayName,
    int managementTypeCode,
    string commandLine,
    string help
}[] getModuleDescriptors(boolean includeRunning)
```

returns the module descriptor for all the modules known to SBW.

```
{
    string moduleIdentificationName,
    string serviceIdentificationName,
    string serviceDisplayName,
    string serviceCategory,
    string help
} getServiceDescriptor(int moduleId, string serviceName)
```

returns the service descriptor for the named service on the given module instance

```
{
    string moduleIdentificationName,
    string serviceIdentificationName,
    string serviceDisplayName,
    string serviceCategory,
    string help
} getServiceDescriptor(int moduleId, int serviceId)
```

returns the service descriptor for the given service on the given module

```
{
    string moduleIdentificationName,
    string serviceIdentificationName,
    string serviceDisplayName,
    string serviceCategory,
    string help
}[] getServiceDescriptors(int moduleId)
```

returns the service descriptors for all the services on the given module

```
string getVersion()
```

returns the version of the broker

```
int[] getExistingModuleInstanceIds()
```

return the module instances that are currently connected to the broker

```
void unregisterModule(string moduleIdentificationName)
```

removes all the information for the given module from the registry file.

```
void registerService(
    string moduleIdentificationName,
    string serviceIdentificationName,
    string serviceDisplayName,
    string category,
    string help)
```

adds a record for the given service for the given module to the registry file

```
void registerModule(
    string moduleIdentificationName,
    string nameForDisplay,
    int moduleType,
    string commandLine,
    string helpString)
```

adds a record for the given module to the registry file. The int parameter indicates the type of the module according to the following encoding:

- 0 Unique
- 1 self managing

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. (2001a). Systems Biology Workbench C++ programmer's manual. Available via the World Wide Web at http://www.cds.caltech.edu/erato/sbw/docs/api/.

Finney, A., Hucka, M., Sauro, H. M., and Bolouri, H. (2001b). Systems Biology Workbench Java programmer's manual. Available via the World Wide Web at http://www.cds.caltech.edu/erato/sbw/docs/api/.

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://websites.ntl.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.cds.caltech.edu/erato/sbw/docs/intro.

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.cds.caltech.edu/erato.

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. (2002). 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.

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/.