Systems Biology Workbench Perl Programmer's Manual

Andrew Finney, Michael Hucka, Herbert Sauro, Hamid Bolouri

{mhucka,afinney,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

July 3, 2002

Contents

1 Introduction

The goals of this manual are to explain how to write Perl scripts for interacting with the Systems Biology Workbench (SBW), and to provide a reference for the SBW Perl APIs. This manual complements the overviews 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.

SBW provides 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, Perl, 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 the 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 (SBML; Hucka et al., 2001b), as an important emerging standard in systems biology.

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.

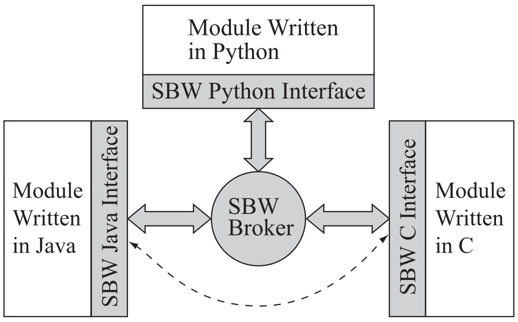

Figure 1: *The overall organization of the Systems Biology Workbench. Gray areas indicate SBW components (libraries and the broker). To individual modules, communications appear to be direct (dotted line), although they actually pass through the broker.*

From the application programmer's point of view, it is preferable to isolate the message-passing details from the application details. We provide two levels of programming interfaces in SBW: a low-level API consisting of the basic operations involved in sending and receiving messages, and a high-level API that hides the messaging level and provides ways for a module's services to be "hooked into" the messaging framework in an object-oriented fashion.

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, 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. The current version of the SBW Perl module does not provide these facilities.

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. The current version of the SBW Perl module does not provide these facilities.

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.

3 A Tutorial on Programming with SBW in Perl

We describe in this tutorial several programming scenarios that use the SBW Perl API. Section 4 provides a detailed reference manual for all of the methods and objects used in this section, plus others that are not discussed in this tutorial.

3.1 Method Signatures

SBW uses a textual notation for describing method names, their parameters and return values. A description of a method in this notation is called a *method signature string*. The syntax of method signatures is shown in Figure 2.

```
Letter     ::= 'a'..'z','A'..'Z'
Digit      ::= '0'..'9'
Space      ::= ( '\t' | ' ' )+
SName      ::= '_'* ( Letter | Digit ) ( 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”.*

In Figure 2 on the preceding page, the `Signature` element describes the signature of one method. The `Type` enumeration refers to the different possible data types in an argument or the method return value. Table 1 shows the correspondence between the values of `Type` shown here and actual programming language data types. The character sequence '`...`' is used in argument lists to indicate that the remainder of the arguments are variable (roughly equivalent to *varargs* in the C programming language). The `ListType` element indicates a list; it may be left empty to indicate a list whose contents are unspecified.

The following are some examples of method signatures. The following describes a method that takes a double parameter `x` as a parameter and returns a double value:

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

The following is a method that returns an integer given a one dimensional array of integers:

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

The following describes a method that takes a single string as argument and returns a list whose elements are a string and a double. This also demonstrates that the SBW method signature notation permits the inclusion of descriptive names such as `name` and `value`, to make signatures easier for humans to discuss.

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

The variable names are optional annotations and are ignored for purposes of comparing signatures; the signature above is functionally equivalent to the following one:

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

The following is a method that returns an array of lists. The method takes no arguments, and the content of the returned lists is undefined:

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

As a special feature, SBW takes an empty list specifier (i.e., `{}`) as a wild-card that matches any other list in another signature. Thus, the lists `{}` and (for example) `{x, int y}` are equal as far as signature parsing is concerned.

Methods can also be declared as returning no value, using the type `void`:

```
void f(byte x, int[] y)
```

Finally, here is a method that takes a variable number of arguments but returns no value:

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

3.2 Data Types

In order to allow an SBW module written in one language to communicate with another written in a different language, it is necessary in some instances for SBW to define its own data types. We have tried to minimize this and use each programming language's natural types as much as possible. Table 1 on the next page shows the mappings for a subset of languages currently supported in SBW. The following are noteworthy about the Perl mapping:

- Since Perl has no representation for scalars smaller than an integer, byte and boolean values are represented as integers;
- SBW homogeneous arrays are represented as Perl lists, even though Perl lists are heterogeneous; and
- Perl hash objects are used to represent SBW heterogeneous lists.

SBW Signature	Perl	Java	C++	C	Python
`string`	*string*	`String`	`std::string`	`char *`	`string`
`int`	*integer*	`int`	`Integer` [a]	`SBWInteger`	*int*
`double`	*double*	`double`	`Double` [a]	`SBWDouble`	*float*
`boolean`	*integer*	`boolean`	`bool`	`SBWBoolean`	*int*
`byte`	*integer*	`byte`	`unsigned char`	`unsigned char`	`string`
array	*array*	*array*	`std::vector` [b] or *array*	*array*	`array` [c]
list	*hash*	`java.util.List`	`DataBlockWriter` [a] `DataBlockReader` [a]	`SBWDataBlockWriter *` `SBWDataBlockReader *`	*list*

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 3.1. 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]*SBW symbols in the C++ library are defined in the namespace* `SystemsBiologyWorkbench`*.* [b]*In the C++ library,* `std::vector` *is used for 1-D arrays and raw C++ arrays are used for 2-D arrays.* [c]*In Python, 1-D and 2-D arrays are implemented using the* `array` *type from the* Numerical Python *package (Ascher et al., 2001).*

These rules only apply to the methods on objects which serve as proxies for services in other modules. These proxies are created dynamically by the Perl SBW library.

The form of Perl hash objects used to represent SBW heterogeneous lists depends on the signature of the SBW service method being called. The SBW Perl library expects a hash object in the place of SBW lists. The keys of the hash object are taken from field names in the SBW list definition in the signature. As discussed in in Section 3.1, the field names in a list are optional in SBW. Where these are not supplied by the method signature, the hash keys in the Perl API are of the form `element`*`n`*, where *n* is the position of the item in the list starting at 0.

For example, a method with the following signature,

```
void plotPoints({double x, double y}[])
```

can be called from Perl as follows:

```
$object->plotPoints([{ x => 0.1, y => 0.2 }, { x => 0.3, y => 0.4 }]);
```

A method with the following signature,

```
void plotEdges({double, double}[])
```

(note that it doesn't have named fields), can be called as follows:

```
$object->plotEdges(
    [{ element0 => 0.1, element1 => 0.2 }, { element0 => 0.3, element1 => 0.4 }]);
```

The latter form is also used when no internal type information is given for a list; i.e., when the SBW method signature is {}.

Data objects that are not scalar are returned from SBW method calls as references. All of the links within the returned structures between non-scalar objects are by reference. The objects passed to proxy methods can use references as the caller prefers.

3.3 Calling a Known Module

The simplest use of SBW consists of invoking services on a module. The simplest case of invoking services on a module is when the module identification name and service identification names are known by the programmer in advance.

We begin with a simple example that involves calling a service named "Trig" on a module named "edu.caltech.trigonometry". Let us assume that the interface for service "Trig" consists of two methods:

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

The following Perl program fragment illustrates making a call to one of these methods. In this example, the program executes one method on another module.

```
use lib 'SBW_HOME/lib'
use SBW;

$module = SBW::getModuleInstance("edu.caltech.trigonometry");
$service = $module->findServiceByName("Trig");
$trig = $service->getServiceObject();
$trig->sin(x);
```

(The term *SBW_HOME* above stands for the path to the SBW installation directory on your computer; it is not meant as a literal string. You have to substitute the appropriate pathname. If you installed SBW using the SBW installer, this directory is where you told the installer to place SBW.) Let us assume there is a module identified as "edu.caltech.trigonometry" and that it has a service named "Trig". We want to invoke the method `sin` on this service. The first step is to access an instance of the module from our program and obtain an object representing it. This is accomplished using `SBW::getModuleInstance` which returns an `SBW::Module` object. Then, we obtain an object representing the specific desired service on that module instance using `findServiceByName` on the `SBW::Module` object. The `findServiceByName` method returns a `SBW::Service` object. The SBW Perl interface makes accessing the SBW methods of this service convenient: the `getServiceObject` method returns a proxy object for the service (here assigned to the variable `trig`), and the proxy object permits calling the `sin` method directly.

The call to the `sin` method causes (underneath it all) messages to be sent to, and returned from, the implementing module. This is a blocking operation—the caller waits for a reply.

The Perl API uses the Perl `croak` exception handling mechanism to handle error states. Therefore its is possible to trap SBW errors using `eval`. Further data on the last SBW error can obtained using the `SBW::SBWException` module as documented in Section 4.5.

3.4 Calling a Known Service Having a Complex Return Value

The example above involved simple parameters and return values to and from service methods. However, SBW supports more elaborate data structures than those used so far. In this section, we present an example of calling a service method having a complex return value.

In order to allow method parameters and return values to be described in a language-independent manner, SBW uses the simple notation for method signatures described earlier in Section 3.1. For our purposes in this section, we explain the meanings of just those parts of the notation we actually need here.

The following is a signature definition for a method called `loadSBMLModel` that accepts a single string and returns an array of lists; each item in the array of lists consists of a string and a `double` floating-point value:

```
{string name, double x}[] loadSBMLModel(string sbml)
```

Let us also assume that there already exists in our program an instance of a `SBW::Service` object that implements this interface, assigned to a variable named `service`. As shown before, we can ask SBW to create a proxy object for this interface as follows:

```
$simulator = $service->getServiceObject();
```

Given this, we can call method `loadSBMLModel` on the proxy object and get the array of lists it returns. Obtaining the individual elements of each list is a simple matter of using standard Perl hash operators. To illustrate this the following code fragment shows one approach to iterating through the list returned by `loadSBMLModel` and extracting the two elements in each list. In this example, the result of the call to `$simulator->loadSBMLModel(sbml)` is stored in the variable `parameters`. A `foreach` loop then iterates over each element in the `parameters` array.

```
// Assume that something assigns a value to variable sbml.
...
$parameters = $simulator->loadSBMLModel($sbml);
...
foreach $parameter (@$parameters)
{
    $name = $parameter->{name};
    $value = $parameter->{x};

    // Do something with parameter data here...
}
```

The hash keys are taken from the method signature on the called module instance. If the items in the list are not labeled in the signature the Perl API module substitutes keys of the form `element`*n* where *n* is the integer position of the item in the list starting from 0. If the signature was:

```
{string, double}[] loadSBMLModel(string sbml)
```

then the following code would be used to recover the data:

```
$name = $parameter->{element0};
$value = $parameter->{element1};
```

4 SBW Perl API Reference

This section is a reference for the different components of the SBW Perl API. The classes in the API are summarized in Table 2 and described in detail in the paragraphs that follow.

Module/Class	Role	Section	Starting Page
`SBW`	Provides static methods for general SBW operations such as connecting to SBW and searching for services and modules	4.2	14
`SBW::Module`	Represents a running module instance and provides access to basic information about the module as well as ways of listing and searching the module's services	4.3	17
`SBW::ModuleDescriptor`	Describes a module that may or may not be running; the information corresponds to the static data recorded in the SBW Broker's registry	4.4	18
`SBW::SBWException`	The stores information on exceptions thrown by SBW methods and client-side service proxies	4.5	19
`SBW::Service`	Represents a service and provides access to the service methods implemented by a module instance	4.6	21
`SBW::ServiceDescriptor`	Describes basic information about a service, such as its name and category; also provides a way to get a `Service` object	4.7	23
`SBW::ServiceMethod`	Describes a method on a service and provides access to the method signature	4.8	24
`SBW::Signature`	Represents a method signature in a structured form and allows translation to and from text string form	4.9	25
`SBW::SignatureElement`	A component of a method signature consisting of a data type and optionally a variable name	4.10	26
`SBW::SignatureType`	Represents a data type in a method signature	4.11	27

Table 2: *The classes defined by the SBW Perl API, presented in alphabetical order.*

4.1 General Concepts in the SBW API

The SBW distribution and the code examples in this manual are designed for use with Perl distribution version 5.6 (Wall, 2002). Before describing the classes provided by the SBW Perl API library, we begin by discussing some general points about the Systems Biology Workbench framework and its implementation.

4.1.1 Relationships between the Different Elements in the SBW API

When dealing with the different objects listed in Table 2, it is useful to keep in mind the distinction between a local representation of a module, service or service method, and its implementation in an actual module instance.

The objects of class `SBW::Module` and `SBW::Service` are local representations (proxies) for other, concrete objects that exist elsewhere in the SBW system. Figure 3 illustrates how the proxies are related to the objects they represent.

Figure 3: *Relationships between proxy objects in a client and the corresponding objects in a module instance.*

SBW generally avoids caching all but the most basic information, to avoid problems that might arise if a remote module changed some part of its definition and the cached information fell out of date. However, some information is cached for efficiency and any method that may involve communicating with a remote module can potentially cause a Perl exception.

4.1.2 On Naming Modules, Services and Methods

Module Names. Each module has two names: a unique *module identification name*, and a name for display. Module identification names are used by SBW for identifying modules running in a session. They are compared by SBW on a case-sensitive basis. A module's identification name should be chosen so that it is unlikely to be equal to any other module identification name running on a particular computer. To achieve this, we suggest the following convention: use the reverse domain name, followed by a '.', followed by the module name, all in lower case. The reverse domain name is simply the reverse of an Internet domain name string affiliated with the module developer. (For the SBW team at Caltech, this is typically "edu.caltech".) An example of a module identification name following this convention is "edu.caltech.gibson".

This scheme for module identification names is voluntary. The API will still work with other naming schemes. The objective is simply to make module names likely to be unique to one host (or more specifically, to one running instance of the SBW Broker).

Service Names. Each service has two names: a unique *service identification name*, and a name for display. A service identification name must be unique to a given module when compared in a case-sensitive manner. Service identification names must follow the `SName` syntax as defined in Section 3.1, and by convention start with a capital letter.

Service display names are intended for use in user menus and other similar situations. They should be chosen to be fairly short and descriptive, but at the same time, fairly unique. Names that are too general, such as "Simulator", are usually not good choices.

Version numbers should *not* be included in service names (i.e., do not use a string such as "Stochastic Simulator 1.0"). Information about versions and similar attributes is best placed elsewhere such as the module's help string (see Section **??**).

Method Names. Method names must follow the `SName` syntax defined in Figure 2 on page 5. By convention, they should start with a lower case letter. Method names are not compared outside of the context of method signatures.

Method signatures are compared using the same scheme as in programming languages such as C++ and Java. This means, for example, that a service may define more than one method with the same name, as long as the total signatures for the methods are different. Note that in some situation this can make methods inaccessible to Perl. This issue is described in more detail in section 4.6.

4.1.3 Module Management Types

Depending on the kind of services a module implements and the style of use intended, it may be more suitable to have multiple copies (instances) of a module running, or to have only one instance running. Modules that do not keep state information are often best implemented as having only one copy running in an SBW system. For example, a module that performs data conversion operations could run as a unique instance in an SBW system, avoiding the resource usage that would result if a fresh copy were started every time that a client module requested its services.

SBW allows a module developer to designate the type of management that SBW should assume for a module. In the Perl API, this is indicated by the value returned by the method `$moduleDescriptor->getManagementType()`. The possible management types are:

- `$SBW::UniqueModule`: Indicates that only one instance of this module should be started up by SBW, and that it should be shared among all requests for the services it implements.
- `$SBW::SelfManagedModule`: Indicates that multiple copies of the module may run and that the module manages itself.

4.1.4 Propagating Exceptions in SBW

Exceptions that occur in a module during the execution of a service method are propagated back by SBW to the module that invoked the service. If the exception was thrown deliberately by the module, for example to indicate that the caller has made a mistake in its invocation, the caller will receive an exception of type `SBWApplicationException`. The latter is the only kind of SBW exception that a module can create. Otherwise, if the exception involves a problem in SBW or incorrect use of SBW, the caller will receive an instance of an SBW exception specific to the problem. Section 4.5 discusses SBW exceptions in more detail and lists the kinds possible.

4.1.5 Distributed Computing with SBW

SBW supports the ability to run modules on multiple computers. It does this by starting an SBW Broker on each remote computer involved in response to certain forms of invocations of `SBW.getModuleInstance` (described in Section 4.2). At the time of this writing, SBW uses the Secure Shell (SSH; Barrett and Silverman, 2001) to start remote Brokers and provide a secure communications channel between Brokers running on separate computers.

In order to use this facility, you must set up SSH for password-free operation between the computers involved. Appendix 5 describes how to do this.

Once SSH is set up in this manner, remote SBW operations are transparent: modules communicate with each other in exactly the same way as when running on the same host computer. The Brokers on each host take care of routing messages to their proper destinations.

4.2 Module: `SBW`

The `SBW` module provides the following functions for various fundamental operations. There are no object methods in this module.

SBW::connect()

Connect to SBW on the local computer as a client. The connection is anonymous. Other modules querying the Broker about which modules are connected to SBW will be informed that a module is connected, but without a name or other information.
A client can only be connected to one SBW Broker at a time; attempting to call `connect()` twice (without calling `disconnect()` in the interim) will result in an exception. Connection attempts that end in failure (e.g., due to a network interruption) will also result in an exception.
Note that this function does not normally need to be called. All the other functions in the SBW Perl API will automatically connect to the local SBW Broker if the client is not already connected. The function is nevertheless provided in case client modules need finer control over their connections to SBW.

SBW::connect(hostNameOrAddress)

Deprecated method. The introduction of proper networked distributed operation in SBW version 1.0 makes this method obsolete. Programs should always connect to their local Broker using `connect()` and connect multiple Brokers together using `link(hostNameOrAddress)`.
Connects (as a client) to the SBW Broker running on the computer identified by the host name or IP address in parameter `hostNameOrAddress`. `hostNameOrAddress` should be supplied as a string. The connection is anonymous. Other modules querying the Broker about which modules are connected to SBW will be informed that a module is connected, but without a name or other information.
This is a variation on the plain `connect()` method described above. The same exception conditions apply.
Note that, as with the plain `connect()`, this function does not normally need to be called. All the other functions in the SBW Perl API will automatically connect to the local SBW Broker if the client is not already connected. The function is nevertheless provided in case client modules need finer control over their connections to SBW.

SBW::disconnect()

Notifies SBW that this application will no longer be consuming services. Invoking this function will first send a notification message to the Broker, then close the network socket and terminate the thread handling message processing.
A client can only be connected to one SBW Broker at a time; therefore, unlike the two versions of the `connect` method, there is no variant of `disconnect` that takes a host name or address as argument.

static void SBW::link(String hostNameOrAddress)

Tells the local SBW Broker to connect to another SBW Broker running on a remote computer, starting the remote Broker first if necessary. The remote host is identified by the host name or IP address in `hostNameOrAddress`.
It should not normally be necessary to call this method when an application simply wants to start modules on remote hosts. The `SBW::getModuleInstance()` call described below will start remote Brokers and modules if the given module name has a specific form.

The following method provide ways of discovering services and service categories.

SBW::getServiceCategories(parentCategory)

Returns an array of strings listing all of the immediate subcategories of the given category `parentCategory`. An empty string as the value of `parentCategory` stands for the root of the hierarchy tree; thus, `getServiceCategories("")` returns the top-level categories known to SBW.

SBW::findServices(category)

Returns an array of `SBW::ServiceDescriptor` objects corresponding to all of the services registered with SBW in the given `category` and subcategories of `category`. An example use of this method might be to use the category "Analysis" to query for all the services which provide analyses of SBML models.

SBW::findServices(category, recursive)

Returns an array of `SBW::ServiceDescriptor` objects corresponding to all of the services registered with SBW in the given `category`. The boolean flag `recursive` indicates whether the search should be performed recursively through the service hierarchy. If `true`, the string `category` is matched against all categories and subcategories. An example use of this method might be to use the category "Analysis" to query for all the services which provide analyses of SBML models.

The following methods are concerned with obtaining module instances from the SBW Broker.

SBW::getModuleInstance(moduleName)

Returns an instance of the given module represented by an `SBW::Module` object. If the module instance has management type `$SBW::UniqueModule` (see Section 4.1.3 on page 12), then a `SBW::Module` object corresponding to an existing module instance is returned. Otherwise, a new module instance is launched by the SBW Broker. A new instance is always launched if no existing instance exists.
Various unusual conditions may lead to exceptions. These include: null module names, inability to contact the SBW Broker, inability to start the requested module, etc.

SBW::getExistingModuleInstances()

Returns an array of `SBW::Module` objects corresponding to every module connected to the system at this time.

SBW::getExistingModuleInstances(name)

Returns an array of `SBW::Module` objects corresponding to every running module that has the given module unique name. (Multiple instances of a module may be running if the module does not have management type `$SBW::UniqueModule` and multiple client modules have requested instances.)

SBW::getModuleDescriptor(name)

Returns a `SBW::ModuleDescriptor` object corresponding to the named module.

SBW::getModuleDescriptors()

Returns an array of `SBW::ModuleDescriptor` objects corresponding to all modules known to the local SBW Broker and remote brokers connected to the local Broker. Only registered modules are included.

SBW::getModuleDescriptors(includeRunning)

Returns an array of `SBW::ModuleDescriptor` objects corresponding to all modules known to the known to the local SBW Broker and remote Brokers connected to the local Broker. If the boolean flag `includeRunning` is `false`, only registered modules will be included. If the flag is `true`, this method additionally includes running module instances. The difference in behavior is relevant when there are unregistered modules connected to SBW: when `includeRunning` is `false`, unregistered modules are not included in the array of module descriptors returned.

SBW::getModuleDescriptors(includeRunning, localOnly)

Returns an array of `SBW::ModuleDescriptor` objects corresponding to all modules known to the local Broker as well as any remote Brokers connected to the local one.
If the boolean flag `includeRunning` is `false`, only registered modules will be included. If the flag is `true`, this method additionally includes running module instances. The difference in behavior is relevant when there are unregistered modules connected to SBW: when `includeRunning` is `false`, unregistered modules are not included in the array of module descriptors returned.
If the boolean flag `localOnly` is `true`, then only modules registered with or connected to the local SBW Broker are included in the set of descriptors returned. If `localOnly` is `false`, all modules known to all Brokers are included in the set.

SBW::getVersion()

Returns the version of the SBW C API library on which the Perl extension is based.

SBW::getBrokerVersion()

Returns the version of the SBW Broker.

4.3 Module: `SBW::Module`

The `SBW::Module` class represents an instance of a module. A `SBW::Module` class object *stands for* an actual module instance, meaning the corresponding application, that is running on the user's computer or perhaps another computer. Having a reference to a `SBW::Module` object in a program implies that the SBW Broker has started up at least one instance of the corresponding real module. (This is in contrast to having an instance of a `SBW::ModuleDescriptor` object, described in the next section; a `SBW::ModuleDescriptor` describes a module but does not imply that an instance of the module is necessarily running.)

This module has the following methods:

getDescriptor()

Returns a `SBW::ModuleDescriptor` object (Section 4.4) that describes the module. The descriptor can be used to obtain the module's display name, management type, etc.

A `Module` object contains zero or more `Service` objects which represent the services implemented by the module. These services can be listed or searched by name and category using the three methods `getServices`, `findServiceByCategory` and `findServiceByName`.

getServices()

Returns an array of `SBW::Service` objects (cf. Section 4.6) representing the services implemented by this module.

findServiceByName(serviceName)

Returns a `SBW::Service` object (Section 4.6) representing the service with the given name implemented by this module instance. The name matching performed is case-sensitive and exact.

findServicesByCategory(serviceCategory)

Returns an array of `SBW::Service` objects (Section 4.6) corresponding to all the services in the given category implemented by this module instance. Returns an empty array if there is no service with a category that matches the string `serviceCategory`. The name matching performed is case-sensitive and exact.

The lifetime of a given module X is determined by a combination of two factors: (1) the management type of the module and (2) the number of modules that are actively communicating with X.

shutdown()

Tells the SBW Broker to disconnect from this module instance; this normally causes the module instance to shut down.

4.4 Module: `SBW::ModuleDescriptor`

A *module descriptor* describes a module. It embodies the static data recorded by the SBW Broker in the module registry (see Section 2.1). The corresponding real module instance may or may not be running.

It may seem confusing at first that there may be no running module instance corresponding to a module descriptor returned by SBW. However, the utility of this becomes clear when one considers the need to list all modules known by the system. SBW provides methods, such as `SBW::getModuleDescriptors`, that return module descriptors for all modules known by the Broker. It would be inefficient, and in some cases downright impractical, if every query for a module descriptor resulted in a module instance being started by SBW. Thus, module descriptors are decoupled from the running state of a module.

Module descriptors are useful for finding out a module's basic characteristics, such as its name, without necessarily starting up an instance of the module.

`SBW::ModuleDescriptor` has the following methods:

getName()

Returns the unique identification name of the module. This name typically follows a notational convention such as that outlined in Section 4.1.2; for example, a module's unique name might be "edu.caltech.trigonometry".

getDisplayName()

Returns the display name of the module. This is suitable for showing to users in a menu or list. Although there is no fixed limit on the length of display names, they should be kept short.

getCommandLine()

Returns the command line used to start up this module. Depending on how the module is implemented, the command line may have been constructed automatically by SBW or it may have been explicitly supplied by the programmer.

getManagementType()

Returns an integer indicating the *management type* of the module (Section 4.1.3). The management type indicates how the lifetime of the module is managed by SBW.

getHelp()

Returns a help string for the module. This is an optional description of the module provided by the module's author(s). It is informational only and not actually required in the definition of a module.

getServiceDescriptors()

Returns an array of service descriptors containing information for the registered services provided by this module.

4.5 Module: `SBW::SBWException`

This module allows access to the error state of the SBW API. Exceptions raised by the Perl library independently of the underlying C implementation will not set a pending exception accessible by this class. `SBW::SBWException` has no object methods. `SBW::SBWException` has the following functions:

SBW::SBWException::getMessage()

Returns the message string associated with the pending exception. This is a string that can be displayed to the user. This message is identical to the message passed to the `croak` command which throws a Perl exception.

SBW::SBWException::getDetailedMessage()

Returns additional details about the pending exception. This is typically at a level best understood by developers and may include a stack trace.

SBW::SBWException::getCode()

Returns the exception code of the pending exception. Table 3 on the next page lists each exception code and its meaning. This function returns -1 if there is no pending exception.

`Exception`	**Meaning**
`$SBW::ApplicationExceptionCode`	Application-specific exception, thrown deliberately by a module. This is the only type of exception that can be created by a module in SBW.
`$SBW::RawExceptionCode`	Wrapper around any exceptions inadvertently thrown by a module.
`$SBW::CommunicationExceptionCode`	Communications between a caller and receiver failed, possibly due to a lost connection.
`$SBW::ModuleStartExceptionCode`	An attempt to start a new module failed.
`$SBW::TypeMismatchExceptionCode`	The type of data element that was attempted to be read from a message was not the type found. This often indicates a mismatch between the messages expected by a caller and receiver.
`$SBW::IncompatibleMethodSignatureExceptionCode`	An interface or class definition uses method signatures that don't correspond to the signatures on the corresponding service.
`$SBW::ModuleIdSyntaxExceptionCode`	Indicates that a supplied module instance identifier has incorrect syntax.
`$SBW::IncorrectCategorySyntaxExceptionCode`	A supplied category string has incorrect syntax. This is thrown by methods that find or search services by categories.
`$SBW::ServiceNotFoundExceptionCode`	Requested service is not present on this module.
`$SBW::MethodTypeNotBlockTypeExceptionCode`	The supplied class uses types which are not SBW message data block types.
`$SBW::MethodAmbiguousExceptionCode`	Thrown by search methods such as `getMethod` on the `Service` class to indicate that the given signature or signature component matches with more than one method on the given service.
`$SBW::UnsupportedObjectTypeExceptionCode`	The library encountered an object of a type that it cannot encode or decode from a message block.
`$SBW::MethodNotFound`	The given method identification number, signature or partial signature does not match any of the methods that exist on a given service. This can occur if a caller uses the low-level SBW API to attempt to invoke a method on a service and the service has no method with that index.
`$SBW::SignatureSyntaxError`	Thrown if a signature string does not contain a valid SBW signature.
`$SBW::ModuleDefinitionExceptionCode`	Thrown by the `ModuleImpl` constructors if any aspect of a module definition is incorrect.
`$SBW::ModuleNotFoundExceptionCode`	Thrown if the given module identification name doesn't match any module known to SBW.
`$SBW::BrokerStartExceptionCode`	Thrown if a problem occurs starting an SBW Broker

Table 3: *Exceptions in SBW.*

4.6 Module: `SBW::Service`

The `SBW::Service` class represents a *service*, an interface to resources inside a module. Obtaining an instance of a `SBW::Service` object implies that a module implementing the service has been launched by the SBW Broker. This class has the following methods:

getDescriptor()

Returns a `SBW::ServiceDescriptor` object (see Section 4.6) corresponding to this service. The service descriptor holds information about this service's unique name, display name, and other attributes.

getModule()

Returns a `SBW::Module` class object corresponding to the module that implements this service. Since the module instance must already be running, this action does not result in a new instance being launched; however, it does create a new object of class `SBW::Module` referring to the same instance.

The most powerful methods on class `Service` are the following two; they return objects that serve as proxies for the service. Most applications access other SBW modules by first obtaining a proxy object using one of these methods.

getServiceObject()

Returns a proxy object for this service. The returned object will have an interface defined by the signatures exposed by the service implementation. Section 3.2 describes the mapping between Perl and SBW types.
In SBW it is possible to have methods with the same name on a service. These methods must have different argument types. The Perl library uses the types of the supplied arguments to determine the method to be called. As some SBW types have no direct correspondence in Perl, it is possible that some methods will be inaccessible from Perl—in particular, this will occur when a method has the same name and similar arguments to another method. This is a limitation due to programming language differences.

getServiceObject(warnAboutInaccessableMethods)

Returns a proxy object for this service. The returned object will have an interface defined by the signatures exposed by the service implementation. Section 3.2 describes the mapping between Perl and SBW types.
In SBW it is possible to have methods with the same name on a service. These methods must have different argument types. The Perl library uses the types of the supplied arguments to determine the method to be called. As some SBW types have no direct correspondence in Perl, it is possible that some methods will be inaccessible from Perl—in particular, this will occur when a method has the same name and similar arguments to another method. This is a limitation due to programming language differences. If argument `warnAboutInAccessableMethods` is `true`, then a message is printed on the standard output stream when this masking occurs.

The following two methods return information about the methods defined on this service. The information is represented using `SBW::ServiceMethod` class objects (see Section 4.8 on page 24), which only serve to describe service methods at the level of name, signature string, signature structure, and help string.

getMethods()

Returns an array of `SBW::ServiceMethod` objects, one for each of the methods on this service.

getMethod(nameAndArgs)

Returns a `ServiceMethod` object corresponding to the single method described by the partial signature string `nameAndArgs`. The parameter may consist of only a method name, or a name and arguments or complete signature written in the form discussed in Section 3.1 on page 5. Specifically, the allowable syntax is:

```
SName
```

or

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

where the optional arguments list is used to locate specific methods with the same name. The return type is ignored when matching signatures. If the syntax is incorrect, this method will set the exception code to `$SBW::SignatureSyntaxException`.
The given name (and optional arguments) is compared to the parsed signatures of the methods on the service. If more than one method on the service matches the given argument, this method will set the exception code to `$SBW::MethodAmbiguousException`.

4.7 Module: `SBW::ServiceDescriptor`

A *service descriptor* is used to describe a single service provided by a resource available through SBW. The information is obtained from the SBW Broker.

A `SBW::ServiceDescriptor` object is similar to a `SBW::ModuleDescriptor` (Section 4.4 on page 18) in that it provides basic information about a service, but does not necessarily imply that a module instance implementing the service has been started up. This decoupling of services from service descriptors allows service descriptor objects to be used as handles to potential services. 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.

`SBW::ServiceDescriptor` has the following methods:

getServiceInModuleInstance()

Returns a `SBW::Service` object (Section 4.6 on page 21) corresponding to this service. Since `SBW::Service` objects are tied to running module instances, invoking this method will either create a new module instance or reuse an existing instance (depending on the module's management type and whether an instance already exists).

getName()

Returns the unique identification of the service. This name must follow the conventions outlined in Section 4.1.2. In particular, the name must use the `SName` syntax as defined in Figure 2 on page 5, and should start with a capital letter. An example of a service name is "Simulation".

getDisplayName()

Returns the display name of the service. This is suitable for showing to users in a menu or list. Although there is no fixed limit on the length of display names, they should be kept short.

getCategory()

Returns the category of this service. Service categories 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. An example service category is "Analysis/Bifurcation".

getHelp()

Returns a help string for the service. This is an optional description of the service provided for informational purposes only.

getModuleDescriptor()

Returns a `SBW::ModuleDescriptor` object for the module that implements this service.

4.8 Module: `SBW::ServiceMethod`

This class is used to describe a method on a service. (In the Perl SBW API, its role is limited to being informational only, although in some of the other language bindings the `ServiceMethod` class has additional functionality.)

getName()

Returns the name of this method.

getSignatureString()

Returns the signature of this method as a string. Contrast this to the next method, `getSignature()`, which returns a tree-structure representation of the method signature.

getSignature()

Returns a `SBW::Signature` object (Section 4.9 on the next page) representing the signature for this service method. The `SBW::Signature` object provides access to a parsed representation of the individual components of a method signature.

getHelp()

Returns the documentation string for this service method.

4.9 Module: `SBW::Signature`

The `Signature` class is used to represent signatures in a parsed, normalized format. This allows more precise matching of method signatures than using simple string forms of signatures.

getName()

Returns the name of this method.

getArguments()

Returns a reference to an array of `SBW::SignatureElement` class objects, one for each of the arguments for this method. The order of the elements in the array is the order of the arguments in the method signature.

getReturnType()

Returns a single `SBW::SignatureElement` class object representing the return type of this method.

toString()

Converts this method signature to string form.

4.10 Module: `SBW::SignatureElement`

A *signature element* is a name-type pair in a method signature, with the name portion being optional. This provides a single common structure for representing both method arguments (where an argument may be either a data type or a data type followed by a name) and list content types. For example, the method signature `"void foo(string a, int b)"` contains two signature elements for its arguments, `string a` and `int b`.

This class has the following methods:

getName()

Returns the name portion of this signature element. This may be null.

getType()

Returns a `SBW::SignatureType` class object (Section 4.11 on the following page) representing the data type of this signature element.

4.11 Module: SBW::SignatureType

A `SBW::SignatureType` class object represents a data type in a signature. The allowable types are those of the element `Type` in the syntax definition for method signatures in Figure 2 on page 5.

This class differs from the equivalent Java class in that multiple array dimensions are modeled on the same `SBW::SignatureType` object rather than by nested `SBW::SignatureType` objects of type `$SBW::ArrayType` (hence the need for the additional `getArrayDimensions` method).

This class has the following methods:

getType()

This method returns the integer encoding of this type which can be one of:

- `$SBW::ByteType`
- `$SBW::IntegerType`
- `$SBW::DoubleType`
- `$SBW::BooleanType`
- `$SBW::StringType`
- `$SBW::ArrayType`
- `$SBW::ListType`
- `$SBW::VoidType`

getArrayInnerType()

If this `SBW::SignatureType` object represents an array type, this method returns the data type of the elements in the array.

getArrayDimensions()

If this `SBW::SignatureType` object represents an array type, this method returns the number of dimensions of the array.

getListContents()

If this `SBW::SignatureType` object represents an list type, this method returns a reference to an array of `SBW::SignatureElement` class objects corresponding to the types of each element in the list.

toString()

Returns a string corresponding to the data type. The possible types are `byte`, `int`, `double`, `boolean`, `string`, `void`, `[]` to indicate an array, and to indicate a list.

5 Setting Up SSH for Distributed SBW Operation

Although many situations involve all modules running on the same host computer, SBW supports the ability to run modules on remote hosts as well. Each remote host runs a separate Broker process; modules on that host connect to the Broker (the local Broker for that host), and each Broker Broker arranges to route messages between modules whether they are local or remote.

When a remote Broker is started, each existing Broker is informed of the modules that are registered with it. Every Broker thus knows about every other Broker's modules, and offers an amalgamated view to any module that queries for available modules and services. Figure 4 illustrates the organization of modules and Brokers when multiple Brokers are running.

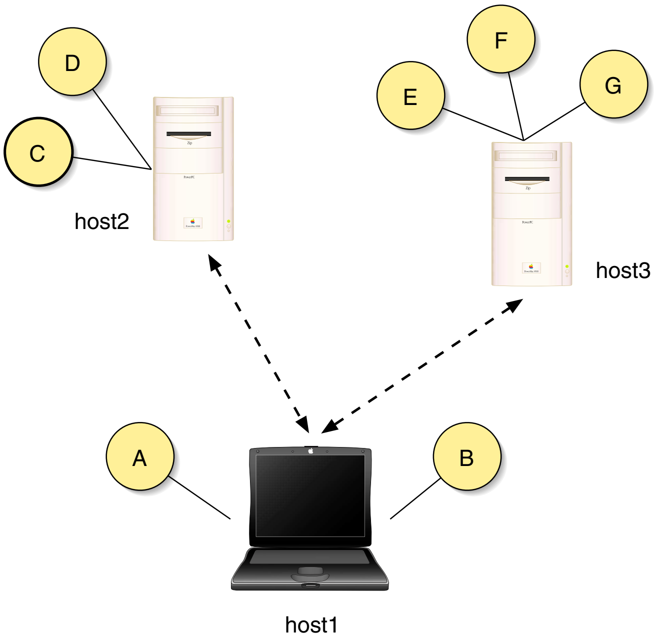

Figure 4: *Example of multiple Brokers and multiple modules running on separate computers. Each module (represented by the yellow circles) connects only to the Broker on the local host; the Brokers on separate hosts connect to each other and route messages between modules as appropriate.*

Brokers are started automatically when a module is requested on a remote host. There are two API calls that can cause a remote Broker to be started:

- `SBW.getModuleInstance(moduleName)`, described in Section 4.2, accepts module names of the form `hostName:moduleName`. If the argument has this form, then the `hostName` part is used as the name of the remote host where the module should be started.
- `SBW.link(hostNameOrAddress)`, described in Section 4.2, allows a program to connect explicitly to a Broker running on the host indicated by parameter `hostNameOrAddress`. This is useful when a module wishes to browse the list of modules and services known to a remote Broker. However, in practice, most remote Brokers and modules are started using the `getModuleInstance` call described above.

SBW uses SSH (Barrett and Silverman, 2001) to start remote Brokers and establish a secure tunnel connection between Brokers on separate hosts. In order for this to work, the following conditions must be met:

- The remote host must be running the SSH daemon and allow remote SSH connections. This is the case for many Unix/Linux hosts; if SSH is not enabled on the target host, you must either enable it yourself (if you have administrative control over the computer), or ask your systems administrator to enable SSH for you. If the remote host is running Microsoft Windows, you will first have to install an SSH daemon package, and then enable incoming SSH connections on that host. The steps for doing this are beyond the scope of this API manual. [Since SSH is a very popular system, it is easy to find instructions on the internet and in books such as the one by Barrett and Silverman (2001).]
- You must have set up your SSH accounts in such a way that you can use SSH to run commands on the remote host without having to supply a password. A good test to verify that this is the case (at least in the case of a remote Unix/Linux host) is to try typing a command such as the following in a command shell:

  ```
  ssh remotehost date
  ```

  The `date` command simply returns a one-line reply indicating the current date. If SSH is set up properly for password-less execution, then the command above should *not* prompt you for a password. If it prompts you for a password, your SSH account or other elements are not set up properly.

Once SSH is set up properly, it is possible to start remote modules and Brokers seamlessly using the two API calls listed above.

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.

Barrett, D. J. and Silverman, R. E. (2001). *SSH, the Secure Shell*. O'Reilly & Associates, Sebastopol, CA, USA.

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.

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/`.

Wall, L. (2002). Perl 5.6. Available via the World Wide Web at `http://www.cpan.org`.