Next Generation Simulation Tools: The Systems Biology Workbench and BioSPICE Integration

*Herbert M. Sauro[1,2], Michael Hucka[2], Andrew Finney[2], Cameron Wellock[1],
Hamid Bolouri[2,3], John Doyle[2] and Hiroaki Kitano[4]

[1]Keck Graduate Institute, 535 Watson Drive, Claremont, USA, 91711
[2] Control and Dynamical Systems 107-81,
California Institute of Technology, CA 91125, USA
[3] Institute of Systems Biology, Seattle
[4] ERATO Kitano Symbiotic Systems Project, M-31 Suite 6A 6-31-15
Jingumae Shibuya-ku, Tokyo 150-0001, Japan

July 14, 2003

*Address for Correspondence: Herbert M Sauro, Keck Graduate Institute, 535 Watson Drive, Claremont, USA, 91711 Phone: (909) 607 0377 Fax: (909) 607 8086 e-mail: hsauro@kgi.edu

Abstract

Researchers in quantitative systems biology make use of a large number of different software packages for modelling, analysis, visualization and general data manipulation. In this paper, we describe the Systems Biology Workbench (SBW), a software framework that allows heterogeneous application components – written in diverse programming languages and running on different platforms – to communicate and use each others' capabilities via a fast binary encoded-message system. Our goal was to create a simple, high performance, open-source software infrastructure which is easy to implement and understand. SBW enables applications (potentially running on separate, distributed computers) to communicate via a simple network protocol. The interfaces to the system are encapsulated in client-side libraries that we provide for different programming languages. We describe in this paper the SBW architecture, a selection of current modules, including Jarnac, JDesigner and SBWMetatool, and the close integration of SBW into BioSPICE which enables both frameworks to share tools and compliment and strengthen each others capabilities.

Introduction

The application of mathematics and computer science to understanding biochemical networks has a long history, going back in fact to the initial development of computers in the 30s and 40s (Chance *et al.*, 1962; Burns, 1971). More recently and especially since the development of high-throughput data collection and the completion of the human genome project there has been a renewed and vigorous interest in understanding the dynamic aspects of cellular networks (Endy & Brent, 2001; R. Rao & Arkin, 2002; Tyson *et al.*, 2003). Although it has been appreciated for many years that cellular networks were dynamic, intricate control systems, the molecular biology revolution of the last thirty years with its focus on DNA and protein structure has taken center stage in mainstream biology at the expense of other studies. It is only in the last few years that 'quantitative systems biology' is finally becoming a mainstream topic in biology.

One of the important techniques at the disposal of the quantitative systems biologist is computer modelling. This involves constructing kinetic models of the biochemical reaction networks, incorporating network as well as kinetic information. The models can vary in size from very small models comprising of only two reaction steps to whole cell models incorporating hundreds of reaction steps. The models are studied by computing the time-course behavior or the steady state. By these means hypotheses can be tested, new hypotheses developed and a general understanding of the network's behavior developed.

Almost from the earliest days of simulation it was realized that developing the necessary mathematical models was tedious and error prone. As a result specialized software was developed to help users input the models into the computer. This involved allowing users to enter reaction networks in a familiar form, often as a list of reactions and kinetic laws. This approach has been followed ever since. Interestingly, though perhaps not surprisingly, the software tools themselves have tended to progress in step with technological developments. In the early years of modelling, tools took a script-based approach to specifying models (Garfinkel, 1968; Park & Wright, 1973; Fell & Sauro, 1990; Sauro & Fell, 1991). With the beginning of the widespread use of graphical user interfaces in the 1980s simulation tools took a marked change in direction. Instead of specifying models using text-based script files, users could now specify models using much friendlier GUI-based user interfaces. The most famous of this new generation was and still is, Gepasi, developed by Pedro Mendes (Mendes, 1993). The development of Gepasi began a new episode in software development which continues to the present day and there are now numerous tools available that take a similar approach.

Easier you use, GUI-based simulators tend to be less flexible compared to script-based tools. In fact many general-purpose commercial simulation tools are script based for this very reason (Mathematica,

Matlab, MathCAD, etc). As a result, script based tools have continued to be developed, the most advanced example of this being Jarnac ?? which incorporates a full programming language as well as extensive libraries for numerical analysis. for In more recent years a second generation of GUI based tools has also emerged which take the user interface to an even more visual level. That is, models in the form of networks are drawn on a canvas and the diagrams converted into a mathematical representation for simulation. Examples of such tools include JDesigner, CellDesigner and KinCyte. At the last count there were over thirty-three different packages for simulating cellular networks. This proliferation of tools has resulted in a variety of capabilities and interfaces. However, though welcome in many respects, this proliferation has resulted in two unwelcome side effects:

1. Each tool uses its own format, often undocumented, to store models. The result is that a model saved in one tool cannot be loaded into another. This obviously hinders the free exchange of models from one tool to another.

2. The second problem is that many of the tools duplicate each other's capabilities. Writing simulation tools takes time and many of the projects are short-lived which means that the authors are unable to develop the tools much further than basic functionality. As a result many of the tools provide similar functionality.

Unlike other software development communities, there is little tradition of code reuse in the system biology community. As a result the community has seen much duplicated effort and little true novelty.

The first problem, that of model exchange has been addressed by introducing a standard format for all tools writers to employ. This standard is called SBML or the Systems Biology Markup Language (Hucka *et al.*, 2003). Along with CellML (Hedley *et al.*, 2001), the introduction of a standard format is beginning to make significant impact on tools writers and the majority of the most widely used tools now employ SBML as a means to exchange models.

The second issue is more difficult to address, that is how to encourage code reuse in the community. Our attempt to resolve this has been to develop a software framework called the System Biology Workbench. The workbench allows different tools to expose programmatic functionality to other tools. This means that a developer can now build on previous work without having to understand in detail the often intricate internal workings of other tools. All a developer need know is the interface that the tool exposes. Thus a particular tool may expose a time dependent simulation interface from a simulation tool, another tool developer rather than invent another simulation tool can exploit this capability and develop a new tool which can carry out additional functions. The workload for the second developer is greatly reduced and they can instead concentrate on novel functionality.

BioSPICE takes a very similar approach, so much so that both SBW and BioSPICE are becoming closely integrated. Like SBW, the goal of the BioSPICE project (BioSPICE, 2001) is to create an open source framework and toolset for modelling dynamic cellular network functions. The hope is that this will develop a user community committed to using and extending the tools. Clearly the SBW project has considerable overlap with the BioSPICE project. We are (Wellock) currently developing a software bridge that will allow modules in both SBW and BioSPICE to communicate with each other. At the moment, SBW and BioSPICE are to an large extent complimentary in functionality, whereas BioSPICE is more data centric, SBW's emphasis is on analysis. As a result SBW can provide a range of ready made modules to the BioSPICE program, including simulators, both stochastic and deterministic, model buiding tools, network analysis tools (based on METATOOL (Pfeiffer *et al.*, 1999)) and as part of the BioSPICE program, tools for optimization and bifurcation analysis. Such a bridge would therefore clearly benefit both communities.

A number of documents have been published in the past on SBW (Hucka *et al.*, 2002) but none have focused on the internal workings of SBW or on some of the applications that we have developed in conjunction with SBW. In this paper we will focus on these issues, in particular we will describe the data structures and the mode of operation of SBW, tools such as Jarnac, JDesigner and Metatool and how SBW will be integrated into BioSPICE.

Systems Biology Workbench

The Systems Biology Workbench is a computational resource sharing framework. It allows applications to communicate with each other efficiently and without loosing their identity. Applications can be written in a variety of different languages and can run on different operating systems across the internet. The entire workbench is open-source and vendor independent. SBW was designed to offer excellent performance and be geared specifically towards scientific applications.

SBW Architecture

In setting out the requirements for SBW, the following items were our highest priority:

- *Simplicity:* The framework must be simple enough that interested developers can use it in their projects with a minimum amount of learning and coding effort. We considered here the full range of developers, from experienced to novice.

- *Performance:* Since SBW will be used for scientific work, performance was an important issue. Moving data from module to module has to be efficient.

- *Component modularity:* As new tools and methods are developed, it must be possible to implement them as modules that can be hooked into the existing framework without having to modify the framework itself.

- *Language interoperability:* The framework must support the interaction of modules written in different programming languages.

- *Free distribution:* All interested users must be able to obtain both SBW and its source code for free. Any software that is incorporated into SBW and distributed with it, such as GUI widgets or object libraries, must itself be free of licensing fees or restrictions on redistribution. (This is only a requirement on SBW itself, and not on modules built for SBW or other software developed using SBW.)

- *Portability:* The framework must be portable to Microsoft Windows (NT, 2000, XP) and Linux initially, and clearly be portable to other platforms in the future.

- *On-demand plug-in loading:* Modules that implement particular capabilities should not have to be pre-loaded into SBW every time it is started; instead, the system should be data- and task-driven and dynamically load modules on an as-needed basis. This helps keep the size of the running system to a minimum.

- *Support for distributed computing:* The user should have control over where processes are executed and the ability to interact with remote services.

Given the requirements lists above, the question then arose, what software technology should we employ to build the framework. Some of the requirements immediately eliminated certain existing frameworks, including DCOM because it is limited to Microsoft Windows platforms, and Java JNI and Java RMI because this would limit the framework to Java.

Other frameworks such as XML-RPC (Winer, 2001) or SOAP (Box *et al.*, 2000) were also unsuitable because these frameworks did not meet our performance criteria. Some recent studies in particular (Olson & Ogbuji, 2002) indicate that SOAP and XML-RPC are orders of magnitude slower compared to CORBA or simple socket transmission.

CORBA was another possibility (OMG, 2001). However, CORBA is notorious for being difficult to master and requires highly skilled programmers to work with. Hence CORBA was not in line with our first requirement, that of simplicity. Since the development of SBW, Microsoft has released .NET which in some limited respects is similar to SBW. The .NET framework has many of the desirable features we sought in the requirements, however it has an uncertain future due to its availability on only a single platform although there is now an open-source, platform independent variant called Mono.

Since we couldn't find a suitable existing framework which satisfied all our requirements, it was decided to develop our own. During the period when we were considering the design, peer-2-peer technologies were becoming a fashionable and useful mode of communication (Oram, 2001). Peer-2-peer possessed many of the attributes which were attractive to us. The three main features which stood out were simplicity, performance and language independence. Most peer-2-peer frameworks were characterized by binary transmission of data over simple TCP/IP sockets. In addition, they were also characterized by simple APIs which helped ensure their rapid take up by third-party developers as witnessed by the plethora of peer-2-peer clients. As a result of these observations it was decided to base SBW on a binary messaging passing architecture over TCP/IP sockets.

Architecture

SBW uses a broker-based, message-passing architecture that allows dynamic extensibility and configurability. Software modules in SBW interact with each other as peers in the overall framework. Modules are started on demand through user requests or program commands. Modules are executables which have their own event loops and all remote calls run in their own threads. As shown in Fig. 1, interactions are mediated through the SBW Broker, a small program running on a user's computer. The Broker enables locating and starting other modules and establishing communications links between them. Communication is implemented using simple TCP/IP sockets which are fast, lightweight with a straightforward programming interface.

Broker-based architectures are a means of structuring a distributed software system with decoupled components that interact by remote service invocations. In SBW, the remote service invocations are implemented using message passing. 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 and platforms: modules can be written in any language, as long as they can send, receive and process appropriately-structured messages using agreed-upon conventions. The organization of SBW means that modules can be easily exchanged, added or removed, even at run-time, under user or program control.

We strove to develop an API for SBW that provides a natural and easy-to-use interface in each of the different languages for which we have implemented libraries. By "natural", we mean that it uses a style and features that programmers accustomed to that language would find familiar. For example, in Java, the high-level API is oriented around providing SBW clients with proxy objects whose methods implement the operations that another application exposes through SBW.

An SBW module provides one or more interfaces or services. Each service provides one or more methods. Modules register the services they provide with the SBW Broker. The module optionally places each service it provides into a category. By convention, a category is a group of services from one or more modules that have a common set of methods.

Supported Languages and Operating Systems

One of the key advantages of SBW is it's language and OS neutrality. At this point in time, we have support for Windows, and Linux operating systems (MacOS is scheduled for future development). The

languages we support, through language bindings, include Java, C, C++, Delphi, C#, VB.NET, Python and Perl. There are developments currently underway to create bindings for Matlab and Mathematica.

Capabilities

Here we summarize the capabilities of SBW.

- Dynamic service and module discovery: The SBW Broker keeps track of modules, services and service categories, and provides facilities for a module to learn about them.

- Remote method invocation: The bread and butter of SBW is enabling one module to invoke a service method in another module. If necessary, the SBW Broker will automatically start an instance of a module whose services are requested.

- Data serialization: Method invocations involve sending messages between modules, with arguments and data packed into message streams. For some languages such as Java, Delphi, C#, VB.NET, 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 in that language.

- Exception handling: SBW provides facilities for dealing transparently with exceptional conditions.

- 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.

- Module, service and method registration: Modules which are not running but wish nevertheless, to advertise their services, can do so by registering with the broker. This is accomplished by running the module once, in a special mode. The registration facilities allow a module to record with the Broker the services that the module provides, the command that should be used to start up the module on demand, and other information. The SBW Broker stores this in a disk file, so that the information provided by modules is persistent between start-up and shutdown of the modules and the Broker.

Messaging Protocols

At the heart of SBW is the messaging protocol used to exchange information between the different modules. For efficiency reasons, messages that are exchanged between modules are simple sequences of binary data. For each programming language there is a language binding library which takes care of much, if not all, of the housekeeping necessary to operate through SBW, including connection and transmission of data. In addition, issues such as little and big-endian byte ordering need not concern the developer as this is taken care of automatically by the binding libraries. Each binding also provides the necessary message packing and unpacking logic and exposes functionality in the form of an easy-to-use API (see example later on).

All modules that make a connection to the SBW Broker are assigned a numeric identification handle. The handle is generated when a module makes its initial connection with the SBW broker or when SBW starts a module and makes a connection. The Broker itself has its own publicly reserved handle which allows modules to make requests to services provided by the Broker. When a module wishes to communicate to another module, it does so by sending a message through the Broker. The message will contain the destination module handle which the Broker will use to route the message onto the appropriate module.
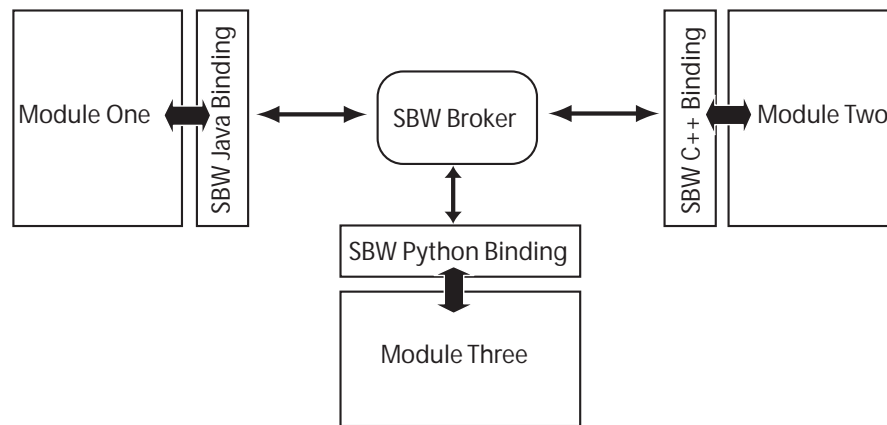
Figure 1: Connection between Broker, Modules and Binding Libraries

There are four basic message types: messages which represent blocking calls to methods in other modules or to the broker itself, messages which represent non-blocking calls to methods in other modules or the broker itself, messages which represent replies to earlier messages, and messages which represent error conditions as a result of poorly formatted messages or exceptions which occur in modules.

Call and Send Messages

These messages come in two varieties, `send` (non-blocking) and `call` (blocking). Both types of message have the same internal structure. What distinguishes the two is the value of the message type byte; see below.
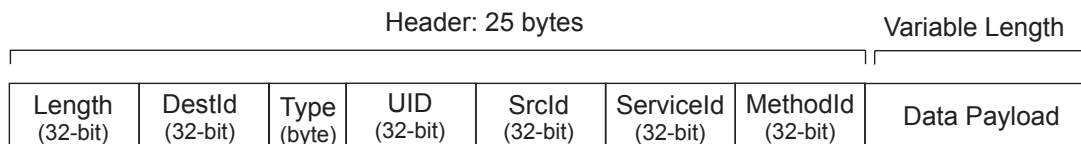
Figure 2: Structure of the Send/Call Message.

The fields in a call/send message have the following meanings:

Length	Length of the message in bytes, including the length integer itself.
DestId	A handle which indicates the destination module for this message.
Type	Indicates whether the message is a call, send, reply or an error condition.
UID	A unique identifier associated with this message. A corresponding reply will have the same UID (Unique identifier) and can be used to match a reply to the original sender.
SrcId	A handle which indicates the source module for this message.
ServiceId	Indicates the required service.
MethodId	Indicates the particular method in the service.
Data Payload	A data payload containing the arguments required by the method.

7

Reply Messages

A reply messages is sent in response to a call message. Its sole purpose is to deliver raw data to the recipient as a result of a method call. The format of the first thirteen bytes of a reply message is identical to a calling message except that the type byte is set to reply message type. All remaining data in the reply message is composed of data returned by the call.
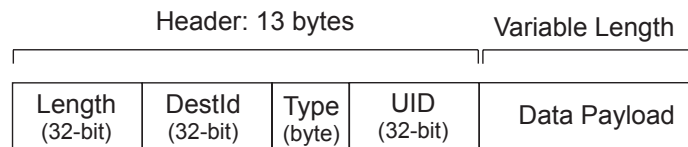

Figure 3: Structure of the Reply Message.

Error Messages

Error messages are sent in response to an error condition originating either as a result of a badly formatted message or as a result of an exception in the method which was meant to service the message. The error byte is a byte to indicate the type of error, these are defined in the developer documentation at the main SBW web site.
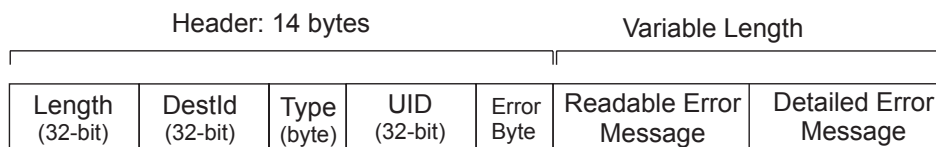
Figure 4: Structure of the Error Message

Data Types

In the previous section we described the structure for the four different SBW messages types. The call and send messages include an optional data payload which may be required by the recipient. Likewise, a reply message may also include a data payload for the recipient. In order for data to be easily exchangeable between modules we needed to decide on a collection of defined data types. Obviously it would not be possible to imagine every possible type of data type that a module might wish to package and send to another module, therefore we devised a set of data types, of sufficient generality, from which any other data type could be constructed. In the first version of SBW we defined seven basic data types. Five of these are fundamental data types, such as byte, boolean, integer, double and string. The remaining two are structured data types which provides most the flexility, these include arrays and lists.

Byte Bytes start with a byte code (`dtByte`) indicating a byte type. This is then followed by an 8 bit byte value.

Data Type	Type Code	Description
Byte	dtByte	Simple Byte
Integer	dtInteger	32 bit integer
Double	dtDouble	IEEE 754 double 64 bit format
Boolean	dtBoolean	Byte indicating true or false (0 represents false)
String	dtString	Sequence of characters, the first unsigned integer indicates the length of the string
Array	dtArray	Homogeneous array of data (n dimensional)
List	dtList	Heterogenous, nested list structure

Table 1: Data Types.

Integers   Integers start with a byte code (`dtInteger`) indicating an integer type. This is then followed by a signed 32 bit integer value in Intel-byte order which has the range -2147483648 to 2147483647.

Double   Double values start with a byte code (`dtDouble`) indicating a double type. This is then followed by a floating-point value stored in standard **IEEE standard 754 double 64-bit** format, that is 1-bit sign, 11-bit base 2 exponent and 52-bit fraction in Intel-byte order.

Figure 5: Integer and Double Data Types

Boolean   Boolean values start with a byte code (`dtBoolean`) indicating a boolean type. This is then followed by a further byte indicating the value of the boolean. A byte value of zero indicates `False` and a value of one indicates `True`.

String   String values start with a byte code (`dtString`) indicating a string type. This is then followed by an unsigned 32-bit integer denoting the number of bytes in the string. The remainder of the data consists of the sequence of characters that make up the string. Note that the string is also null terminated.
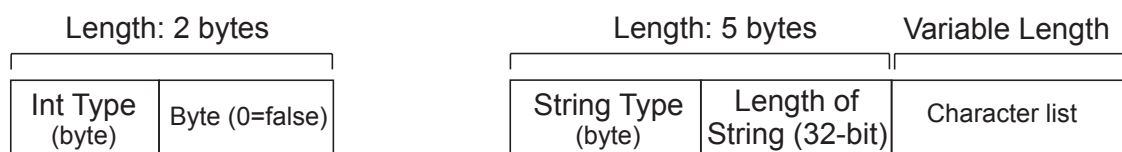

Figure 6: Boolean and String Data Types

Arrays   Arrays are multi-dimensional objects of arbitrary size containing homogeneous data. Arrays start with a header made up of one byte indicating the data type stored in the array, and an integer indicating the number of dimensions, followed by a sequence of integers, one for each dimension, denoting the number of elements in each dimension. The header is therefore $(2 + 4 + 4d)$ bytes long, where d equals the number of dimensions of the array. Array access can be optimized at the module if it is known that the data type has a fixed size. This is especially the case for simple types such as integers and doubles. In these cases, the application can carry out block copies of the data in order to greatly improve performance.

Lists   Lists are recursively defined structures for storing heterogeneous data. This means that lists can be used to store other lists which allows complex relationships to be represented.

A list is a much simpler structure that an array. A list starts with a list type byte, followed by a 32-bit integer indicating the number of items in the list. Each items in the list can be any of the data types previously described, including a list.

For example, the following are legal list structures:

```
[ 1, 2, 3, 4 ]
[ 1, "ATP", 3.1415, {1, 2, 3} ]
[ [ "S1", "S2", "S3", [ 4, 5, 6 ] ], "k1", "k2" ]
[ ["J1", [ ["X0"], ["S1"] ], "k1S1" ], ["J2", [ ["S1", "S2" ], [ "S3" ] ],
          "k2S2" ], .... ]
```

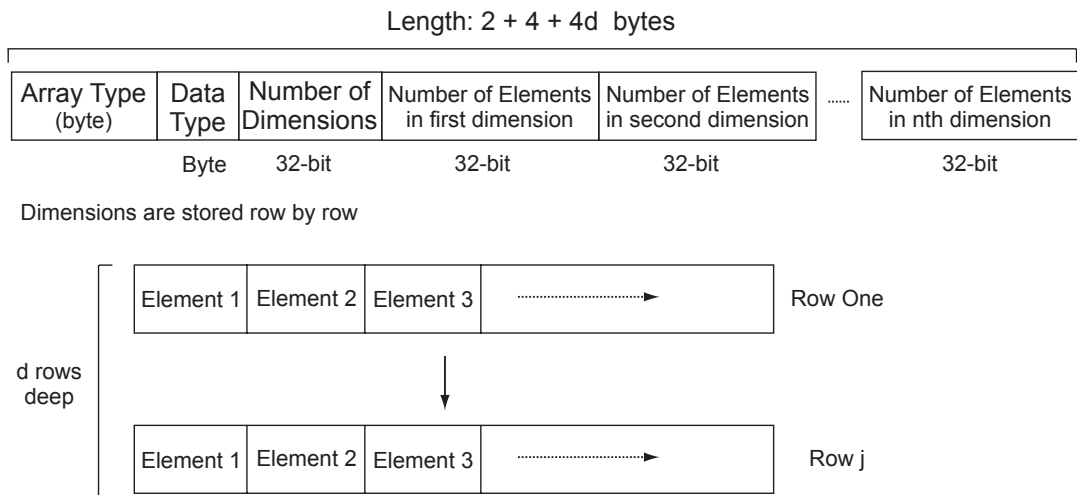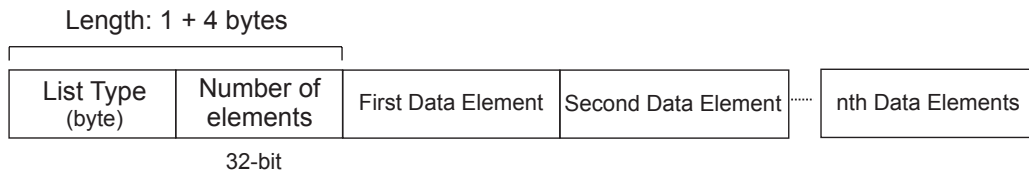Note the nested lists in the third and fourth examples.

Figure 7: Array Data Type.

Length: 1 + 4 bytes

List Type (byte)	Number of elements	First Data Element	Second Data Element	⋯	nth Data Elements

32-bit

The size of each data element is data type dependent

Data elements can be any of the standard types, byte, boolean, integer, double, string, arrays and lists

Figure 8: List Data Type.

Event Support

SBW supplies a number of special method calls to modules which are sent when certain events occur. These include:

void onModuleShutdown(Module module) This method on the listener is called every time a module instance somewhere in the SBW system disconnects from the SBW Broker. The module passed to the method represents the module instance that has just shut down.

void onModuleStart(Module module) This method on the listener is called every time a module instance starts up or connects to the SBW Broker. The module passed to the method represents the module instance that has just started.

void onRegistrationChange() This method on the listener is called whenever a registration change for a module occurs in the SBW Broker. Registration changes are: a module registering itself with the Broker, a module registering a service with the Broker, or a module being unregistered.

Simple Example

Having described in some detail the internal structures and design of SBW, it is now worth showing a simple example to illustrate how it might be used. The following example shows how to set up a module which provides two math services, trig and log, and another module which uses these services. The code is shown using Java but similar code would apply to other languages.

We first declare the classes which represent the services that the module is going to provide. In this case we will provide two services, one that offers basic trigonometric functions, and another that provides basic logarithmic functions.

```
class TrigClass {
    public double sin(double x) throws SBWApplicationException {
        return Math.sin(x);
    }

    public double cos(double x) throws SBWApplicationException {
        return Math.cos(x);
    }
}

class LogClass {
    public double log(double x) throws SBWApplicationException {
```

```
            return Math.log(x);
        }

        public double exp(double x) throws SBWApplicationException {
            return Math.exp(x);
        }
}
```

In the main program we create a new instance of an object that represents the object that other modules will see. Into this object we register the two services that we wish to provide. Finally we call the run method of the module object which initiates the connection to the SBW broker. Once the broker receives the connection request it transmits a startup event to all connected modules indicating that a new module is available. At this point modules may now interrogate the new module and use the services that the module provides.

```
ModuleImpl module = new ModuleImpl("edu.caltech.math", "math", ModuleImpl.UNIQUE);

module.addService("Trigonometry", "trig functions", TrigClass.class);
module.addService("Logarithmic",  "log functions",  LogClass.class);

module.run(args);
```

The argument args in the run method is the command line argument that invoked the module. If a remote module wishes to use the services provided by math, it would use the following code:

```
Interface ITrigService {
        double sin(double x) throws SBWException;
        double cos(double x) throws SBWException;
    }

Interface ILogService {
        double log(double x) throws SBWException;
        double exp(double x) throws SBWException;
}

Module module = SBW.getModuleInstance("edu.caltech.math");

// Get the individual services

Service trig = module.findServiceByName("Trigonometry");
Service log  = module.findServiceByName("Logarithmic");

// Create proxy with this interface and call it:

ITtrigService trigService = trig.service.getServiceObject(ITrigService.class);
ILogService   logService  = log.service.getServiceObject(ILogService.class);

Double result;
Double x = 12.2;

result = trigService.sin(x);
result = logService.log (x);
```

The first two sections declare interfaces of the services which will be used. In this case the services are hard-coded but SBW also allows runtime reflection on a remote module and thus allows methods and services to be used dynamically if need be. Under Java the easiest approach to access remote services is to use interfaces.

Once the interfaces have been declared a call is made to obtain a handle of the module. If the module has been registered with the Broker but is not currently running, this call will cause the module to be automatically started up.

Finally, the objects representing the individual services are created using the `getServiceObject` method call. Last but not least, the methods are called through the service object returned previously.

Thus it only takes a few lines of code to access and call remote methods. For more details of the API the reader is referred to the API manual available the the SBW web site. Note that services and methods on remote modules are also available via a number of interactive scripting tools, in particular Python and Perl. In these cases, interaction is even simpler, as the scripting tools will automatically wrap remote services in to Python and Perl objects. Thus under Python, to the access the trigonometric method, `sin`, only requires a single line:

```
print edu_caltech_math.Trigonometry.sin (30.0)
```

The BioSPICE/SBW Bridge

One of BioSPICEs' main interfaces is based on the Netbeans IDE. This has enabled the developers of BioSPICE to implement a data flow GUI which allows users to direct data from one module to another through a GUI interface. This allows users to chain a series of processes in whatever fashion that suits their needs. The BioSPICE modules themselves are made accessible to the Netbeans IDE either via OAA (Open Agent Architecture) or directly into the Netbeans IDE itself.

Work is nearing completion to construct a software bridge between SBW and OAA, which will enable clients of either system to access the services provided by the other system. These services will be made to appear as native services, identical to any other. This is made possible by a special generic translation layer.

OAA is based on a Prolog programming model, and is organized in terms of agents which provide specific functions. Parameters are untyped, and may be numbers, strings, lists, and several other Prolog-specific data types. Prolog functions do not have return values; rather they use a system called "unification" wherein unbound variables are replaced with results. For example, to get the sum of two numbers in Prolog one might call a function as follows:

sum(2,5,X)

and get the result

sum(2,5,7).

SBW in contrast uses a more common model for methods, which take typed parameters and have a single typed result, i.e.

int sum(int,int).

The bridge handles the conversion of the method signatures between OAA and SBW automatically. If there was a method in OAA such as get_some_data_3 (the name, 'get_some_data,' and three parameters), this would be translated into an appropriate SBW method signature:

get_some_data()

The most recent version of OAA provides for typed, directional parameters. If get_some_data_3 had two input parameters of integer type, and one output parameter of double type, then the SBW method signature would be

double get_some_data(int,int)

In the other direction, an SBW method such as

int get_just_an_int(int)

would be translated by the bridge into an OAA method get_just_an_int_2-the second parameter is just the SBW function's return value.

Incompatible data types are mapped whenever possible to equivalent types. As an example, arrays, which are used by SBW, are not recognized in OAA. These are translated into lists or lists-of-lists, which OAA can handle. Services in SBW appear as separate agents in OAA, and agents in OAA are represented as services of a single module (the "oaabridge") module in SBW.
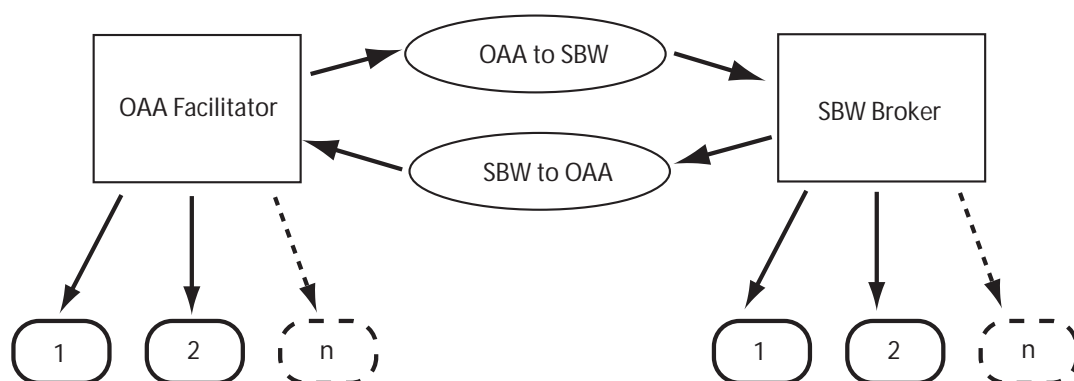
Figure 9: Structure of the BioSPICE/SBW Bridge

Applications of SBW and BioSPICE via the SBW/BioSPICE Bridge

In this section we will describe an application of SBW which utilizes three SBW-compliant tools, Jarnac, JDesigner and METATOOL.

Jarnac

Jarnac is a script-based simulation tool that can operate either interactively via a console window or as a simulation server for SBW. Details of Jarnac can be found in (Sauro, 2000). Here we will just describe the SBW interface.

The Jarnac SBW interface supports four services: `modelServices`, `msim`, `mat` and `asim`. `asim` is used to interface to the SBW GUI, details of which can be found at the SBW web site `http://www.sbw-sbml.org`. The interface provided by `msim` is more extensive than `asim` and is the one described here.

`msim` supports multiple simulation instances, that is more than one simulation can be active at any one time. All the methods in `msim` require a model handle to indicate the current model instance. Model instances can be created and destroyed through the `modelServices` service. `msim` provides a range of methods to control, interrogate and simulate either continuous (ordinary differential equation based), or probabilistic (based on the Gillespie method) models.

Models are loaded into a model instance in the form of SBML Level 1 (Hucka *et al.*, 2003) via the loadSBML() method. A range of access methods are provided to interrogate the currently loaded model.

For example, remote modules can request the number of reactions, the rates of change of species, the reaction velocities etc. In addition, methods are provided for a remote application to access the model equations, including the list of differential equations, the rate law expressions (both in infix format) and the list of any conservation laws in the model. Methods are also provided to allow a remote application to modify parameters and initial conditions. Finally there is a range of analysis methods, include time course simulation, Gillespie stochastic simulation, steady-state analysis, sensitivity analysis and specialist information such as the Jacobian matrix.

The remaining service, `mat` supplies two matrix related methods, one method to compute the eigenvalues for a matrix and a second method to compute the inverse for a matrix.

Jarnac can be run in two modes, either interactively, where a users has access to the model capabilities through a console window and via the SBW interface, or in server mode where Jarnac runs invisibly as a background service. The only way to access Jarnac in server mode is via the SBW interface.

Note that we provide a GUI based browser tool that allows a users to inspect the services and methods available from a particular module.

JDesigner

JDesigner is a model design tool for editing biochemical networks visually. It has no simulation capabilities itself but it can interface to the Jarnac SBW interface. Unlike Jarnac or SCAMP (Sauro & Fell, 1991; Sauro, 2000), where models are entered in the form of a script describing the chemical reactions and rate laws, under JDesigner, models are entered visually as reaction networks. JDesigner stores models in the form of SBML Level 1 (Hucka *et al.*, 2003) with specific annotation added to support layout information. Details of this and other information on JDesigner can be found at the web site, `www.sys-bio.org`. The figure below illustrates a basic screen shot from JDesigner.
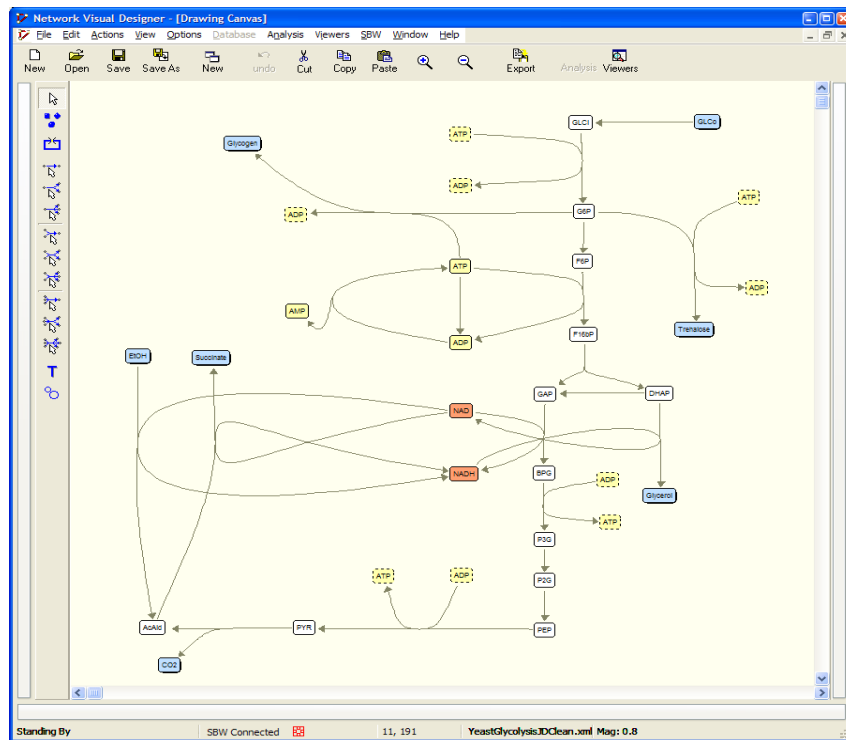
Figure 10: JDesigner, illustrating a model of Glycolysis taken from (Pritchard & Kell, 2002)

Interaction of JDesigner with Jarnac, or any other compatible simulator, is automatic. Figure 10 illustrates a simulation displayed by JDesigner but actually carried out by Jarnac via SBW.
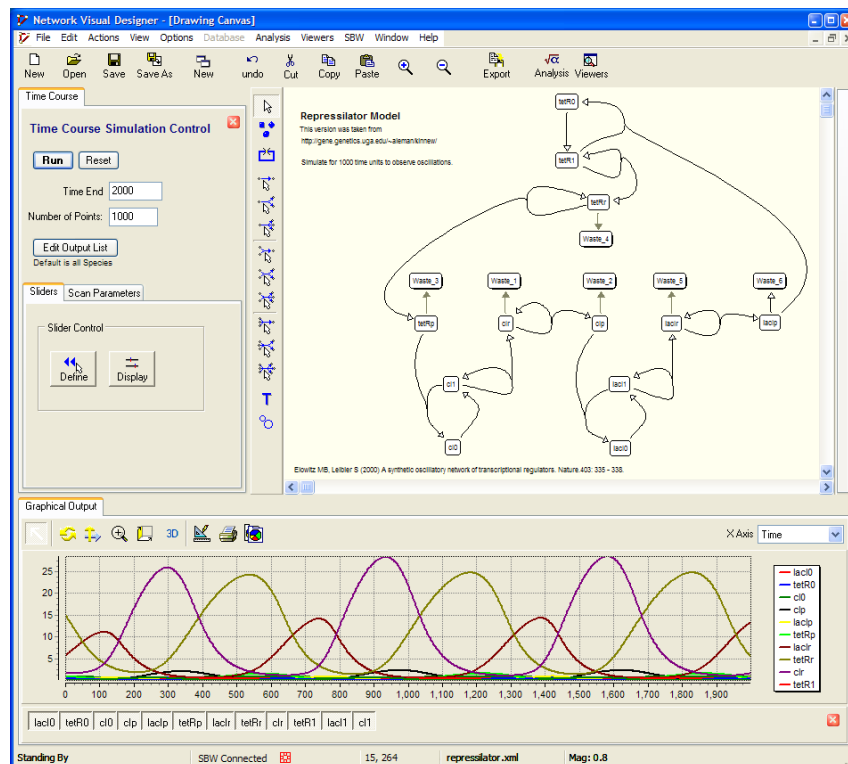
Figure 11: JDesigner and Jarnac working together to carry out and display the results from a simulation. The model was taken from (Elowitz & Leibler, 2000) which illustrates a synthetic oscillatory circuit that was constructed in *Escherichia coli*
.

SBWMetatool

METATOOL (Pfeiffer *et al.*, 1999) is an application developed by Stefan Shuster, Thomas Pfeiffer and more recently by Ferdinand Moldenhauer and Juan Carlos Nuno (http://www.bioinf.mdc-berlin.de/projects/metabolic/metatool/). It's primary task is the determination of elementary modes (Schuster *et al.*, 2000) but it also has a variety of other functions, including null space computation and conservation analysis. It easily runs on Linux or Windows, or for that matter any platform which can compile standard C code. METATOOL generates a multitude of information, including but not exclusively, the null space of the stoichiometry matrix, conservation relations, and what METATOOL was specifically designed to generate, elementary modes. Generating elementary modes is a non-trivial exercise and other packages, such as the interactive simulator, Jarnac (Sauro, 2000), employ METATOOL for this task.

To make METATOOL available to SBW we wrote a small controlling application that has an interface to SBW and controls the running of METATOOL.

Figure 11 illustrates the interaction of METATOOL with JDesigner. JDesigner acts as the model editor from which users can initiate simulation and METATOOL analysis. The figure illustrates two aspects, the lower panel shows the SBWMetatool interface, this displays all the elementary modes that METATOOL found for the displayed model (Calvin Cycle). Note that one of the elementary modes in the lower panel is highlighted. The main canvas shows the Calvin reaction network and the selected elementary mode is displayed on the reaction network by highlighting the appropriate reactions. This

17

allows a user to easily visualize each elementary mode in turn. The example illustrates the ability of SBW to combine two unrelated applications (JDesigner and METATOOL) and deliver completely new functionality. The other point to make is that METATOOL was not modified in this project, we only wrote a small separate SBW based module which could control the running of METATOOL.
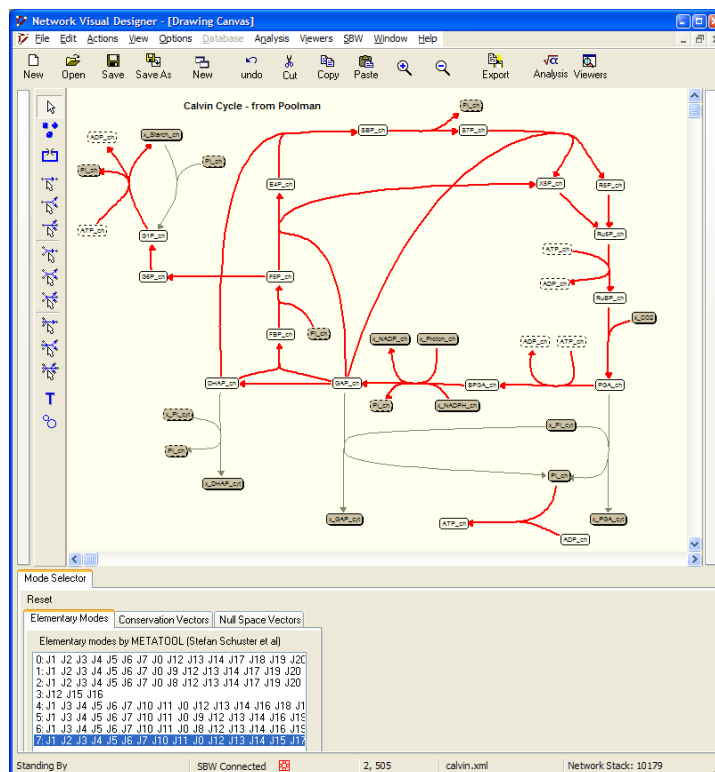

Figure 12: Operation of METATOOL with JDesigner illustrating the visualization of elementary modes in the Calvin Cycle(Model curtesy of M Poolman, D Fell and C Raines (Poolman *et al.*, 2003)

Conclusion

Software reuse is considered a well-known technique for increasing development productivity but the promise often falls short of the expectation. There are some success stories, in particular the number of reusable components for Delphi (Borland) and Visual Basic (Microsoft) run into the thousands (www.torry.com) and probably many more for Visual Basic. The question is why have some development environments been more successful than others at encouraging a vibrant community of code reuse? One of the distinguishing features of VB and Delphi development is the ease with which it is possible to develop stand-alone reusable components. Other environments such as CORBA or basic COM have a much steeper learning curve and thus the number of people actively engaged in supporting code reuse is correspondingly smaller. For code reuse to be a actively supported, code development should be correspondingly easy to accomplish.

In terms of SBW, we have tried to achieve this situation by making the development of reusable SBW modules as easy as development under VB or Delphi. Developing reusable modules in Java or Delphi is particulary straight forward under SBW.

18

In terms of our own development, code reuse was most successful. In developing JDesigner, we did not need to write yet-another-simulator, instead we leveraged the existing simulator Jarnac. As an example of third-party component use, we were able to wrap METATOOL into a SBW-compliant tool thus making available the sophisticated algorithms present in METATOOL to all SBW modules.

Some of the key advantages of SBW over other technologies is performance, simplicity of implementation, and language and platform neutrality. With the rapid rise in interest in Systems Biology, it is fair to say that it it probably impossible for one person or even a group to attempt to write the all singing, all dancing software tool for Systems Biology, simply because the breadth of the field is too wide.

One great advantage of SBW is that it does not constrain developers to a single platform or even a single language, it eliminates language and platform wars at a stroke which means we can concentrate on functionality instead.

Future Directions

The future direction of SBW is in two places, enhancements to the core SBW technology, that is enhancements to the Broker and/or binding libraries and enhancements to modules.

Module Development

Module development is taking place on two levels, enhancements to existing modules and development of new modules. The existing modules, in particular JDesigner will continue to be enhanced. One of the most interesting projects is the development of library based model construction. That is, models can be developed in parts and combined at a later date.

As for new modules, two are currently planned for development, this includes an optimization module and a bifurcation module. Both modules are being primarily developed for BioSPICE and will be made available to BioSPICE via the SBW/BioSPICE bridge.

Core Development - Broker, Language Bindings and BioSPICE Integration

The first version of SBW is complete and in production. The current plans for the development of the core are fairly limited. There are a couple of items that we would like to include in a future version. For example, we would like to add an additional type to the core data types that can be transmitted from module to module, this type being the complex number type. Since SBW is primarily aimed at the scientific community, complex numbers would prove a very useful addition. One of the primary applications of complex numbers in systems biology is stability analysis and data analysis such as principal component analysis. Of course, in the current version, complex numbers can be transmitted by combining existing types; however, since complex numbers are fundamental to quantitative science there is no reason why they should not have 'first class' status as one of the fundamental types.

A second addition we would like to make is to give the binding libraries the ability to decide whether to compress messages before transmission. Some messages especially those containing XML data can be very large. These messages, by their nature are also highly compressible. It would seem sensible therefore to be able to compress such messages automatically before they get transmitted to the receiver. Depending on where the threshold is set to compress a message according to its size, performance increases could easily be achieved.

The most immediate project however to the SBW core is the development of a bridge between SBW and BioSPICE. This is currently underway and should be completed very soon. As previously mentioned

the bridge will allow modules in both SBW and BioSPICE to communicate with each other. At the moment, SBW and BioSPICE are complimentary in functionality, and such a bridge would therefore greatly benefit both communities.

Acknowledgements

References

BioSPICE (2001). Home page. The BioSPICE Development Project, `http://www.biospice.org/`.

Box, D., Ehnebuske, D., Kakivayat, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S. & Winer, D. (2000). Simple object access protocol (SOAP) 1.1: W3C note 08 May 2000. http://www.w3.org/TR/SOAP/.

Burns, J. A. (1971). *Studies on Complex Enzyme Systems*. PhD thesis, University of Edinburgh. url: http://www.cds.caltech.edu/ hsauro/Burns/jimBurns.pdf.

Chance, B., Higgins, J. J. & Garfinkel, D. (1962) Analogue and digital computer representations of biochemical processes. *Proc. Fedn. Am. Socs. exp. Biol.,* **21**, 75–86.

Elowitz, M. B. & Leibler, S. (2000) A synthetic oscillatory network of transcriptional regulators. *Nature,* **403**, 335–338.

Endy, D. & Brent, R. (2001) Modeling cellular behavior. *Nature,* **409**, 391–395.

Fell, D. A. & Sauro, H. M. (1990) Metabolic control analysis by computer: progress and prospects. *Biomed. Biochim. Acta,* **8/9**, 811–816.

Garfinkel, D. (1968) A machine-independent language for the simulation of complex chemical and biochemical systems. *Comput. Biomed. Res.,* **2**, 31–44.

Hedley, W. J., Melanie, N. R., Bullivant, D. P. & Nielson, P. F. (2001) A short introduction to CellML. *Phil. Trans. Roy. Soc. London A,* **359**, 1073–1089.

Hucka, H. M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. & Kitano, H. (2002) The ERATO Systems Biology Workbench: enabling interaction and exchange between software tools for computational biology. In *Pac Symp Biocomput.* pp. 450–461 World Scientific Pub Co.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J.-H., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Novère, N. L., Loew, L. M., Lucio, D., Mendes, P., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J. & Wang, J. (2003) The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Boinformatics,* **19**, 524–531.

Mendes, P. (1993) Gepasi: a software package for modelling the dynamics, steady states and control of biochemical and other systems. *Comput. Applic. Biosci.,* **9**, 563–571.

Olson, M. & Ogbuji, U. (2002). The Python web services developer: messaging technologies compared. http://www-106.ibm.com/developerworks/webservices/library/ws-pyth9/.

OMG (2001). CORBA specication. http://www.omg.org.

Oram, A. (2001) *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates.

Park, D. J. M. & Wright, B. E. (1973) Metasim, a general purpose metabolic simulator for studying cellular transformations. *Comput. Progm. Biomed.,* **3**, 10–26.

Pfeiffer, T., Sanchez-Valdenebro, I., Nuno, J. C., Montero, F. & Schuster, S. (1999) Metatool: for studying metabolic networks. *Bioinformatics,* **15**, 251–257.

Poolman, M. G., Fell, D. A. & Raines, C. A. (2003) Elementary modes analysis of photosynthate metabolism in the chloroplast stroma. *Eur. J. Biochem,* **270**, 430–439.

Pritchard, L. & Kell, D. K. (2002) Schemes of flux control in a model of saccharomyces cerevisiae glycolysis. *Eur. J. Biochem,* **269**, 3894–3904.

R. Rao, D. M. W. & Arkin, A. P. (2002) Control, exploitation and tolerance of intracellular noise. *Nature,* **420**, 231–237.

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

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

Schuster, S., Fell, D. A. & Dandekar, T. (2000) A general definition of metabolic pathways useful for systematic organization and analysis of complex metabolic networks. *Nature Biotechnlogy,* **18**, 326–332.

Tyson, J. J., Chen, K. C. & Novak, B. (2003) Sniffers, buzzers, toggles and blinkers: dynamics of regulatory and signaling pathways in the cell. *Current Opinion in Cell Biology,* **15**, 221–231.
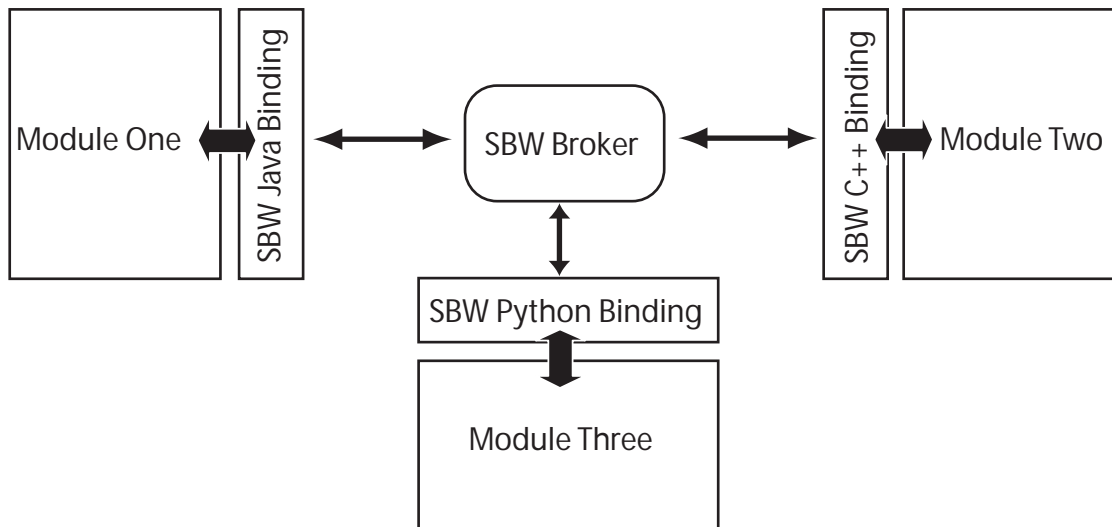
Winer, D. (2001). XML-RPC. http://www.xmlrpc.com/spec/.

Figure 1: Connection between Broker, Modules and Binding Libraries
Sauro et al.
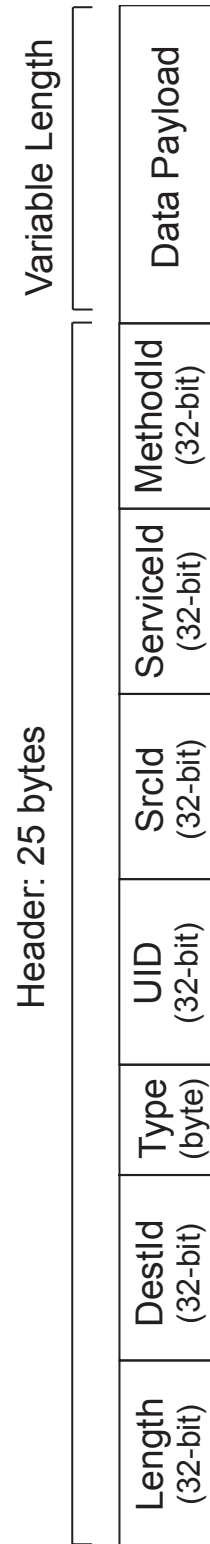
Figure 2: Structure of the Send/Call Message.
Sauro et al.

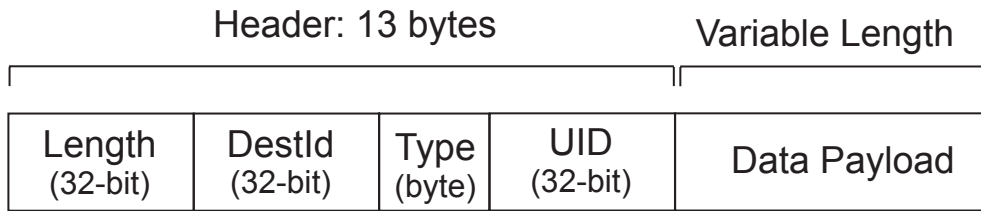| Length (32-bit)	DestId (32-bit)	Type (byte)	UID (32-bit)	Data Payload

Figure 3: Structure of the Reply Message.

Sauro et al.
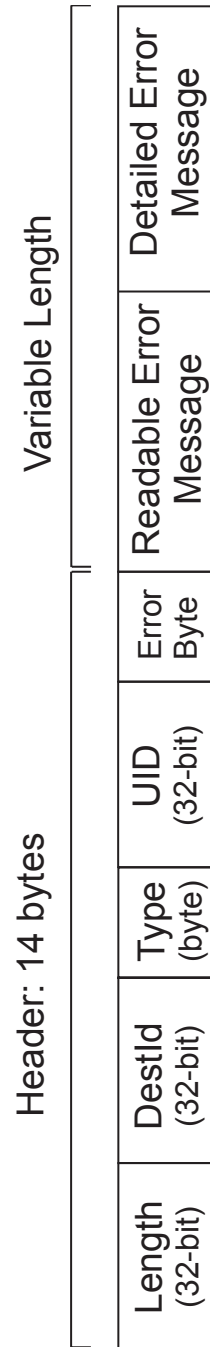
Figure 4: Structure of the Error Message.
Sauro et al.

```
┌─────────── Length: 5 bytes ───────────┐          ┌─────────── Length: 9 bytes ───────────┐

┌──────────────┬──────────────────┐          ┌──────────────┬──────────────────┐
│   Int Type   │                  │          │ Double Type  │ Double IEEE 754  │
│   (byte)     │  32 bit Integer  │          │   (byte)     │    (64-bit)      │
└──────────────┴──────────────────┘          └──────────────┴──────────────────┘
```
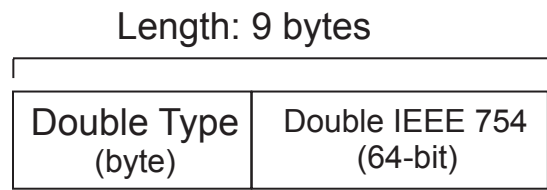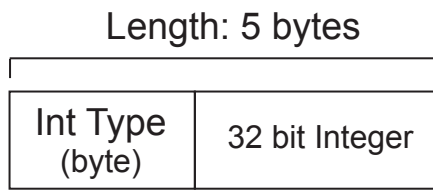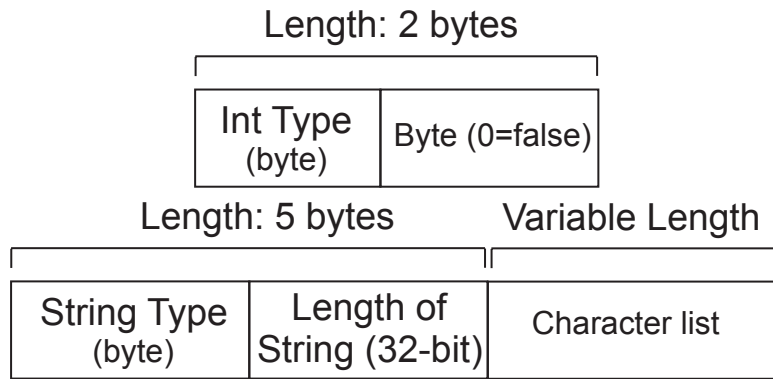
Figure 5: Integer and Double Data Types
Sauro et al.

Figure 6: Boolean and String Data Types

Sauro et al.

Length: 2 + 4 + 4d bytes

Array Type (byte)	Data Type	Number of Dimensions	Number of Elements in first dimension	Number of Elements in second dimension		Number of Elements in nth dimension
	Byte	32-bit	32-bit	32-bit	.....	32-bit

Dimensions are stored row by row

Element 1	Element 2	Element 3	·······>	Row One

→

Element 1	Element 2	Element 3	·······>	Row j

d rows deep
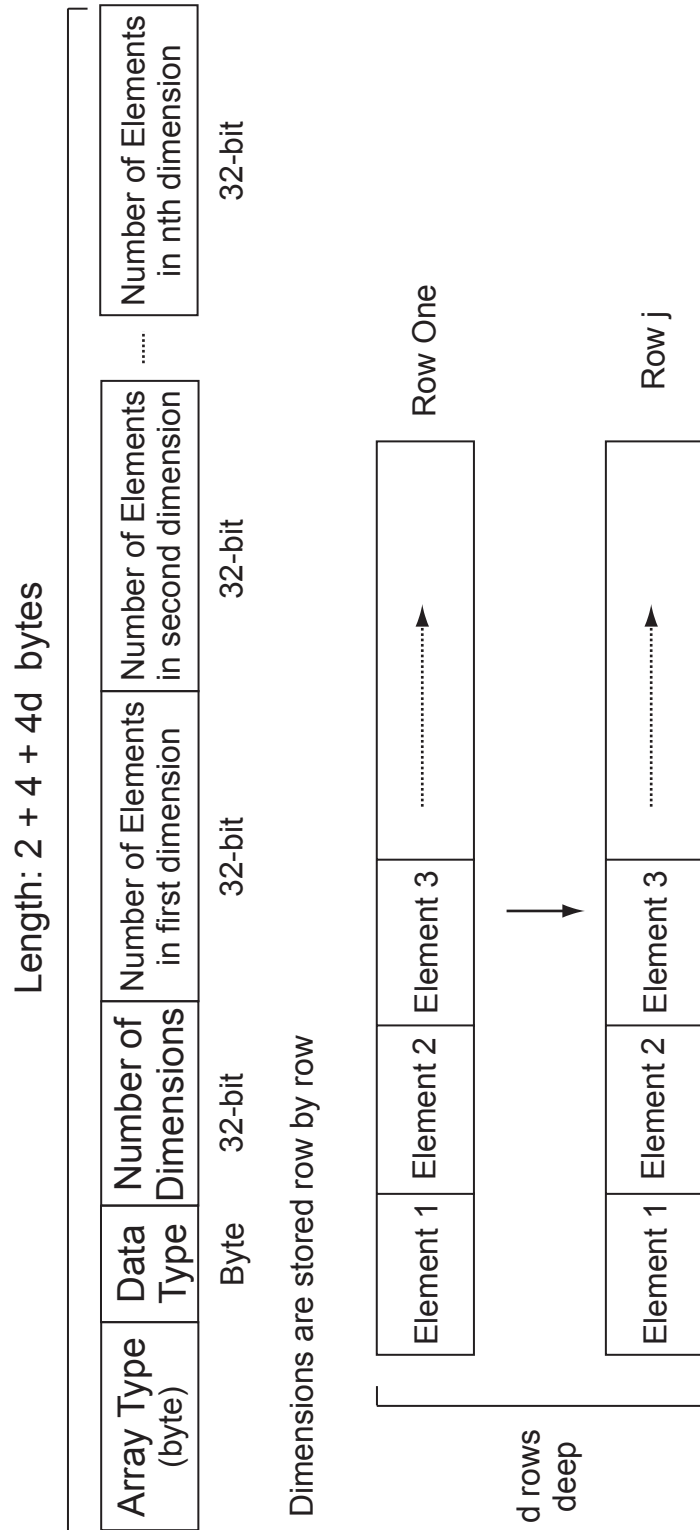
Figure 7: Array Data Type.
Sauro et al.
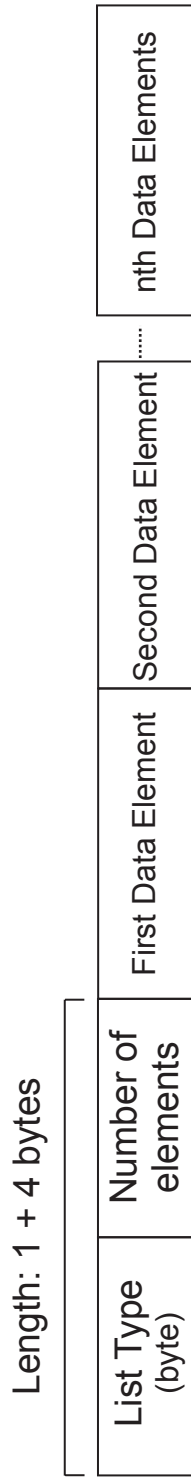
29

Length: 1 + 4 bytes

List Type (byte)	Number of elements	First Data Element	Second Data Element	......	nth Data Elements

32-bit

The size of each data element is data type dependent

Data elements can be any of the standard types, byte, boolean, integer, double, string, arrays and lists

Figure 8: List Data Type.
Sauro et al.
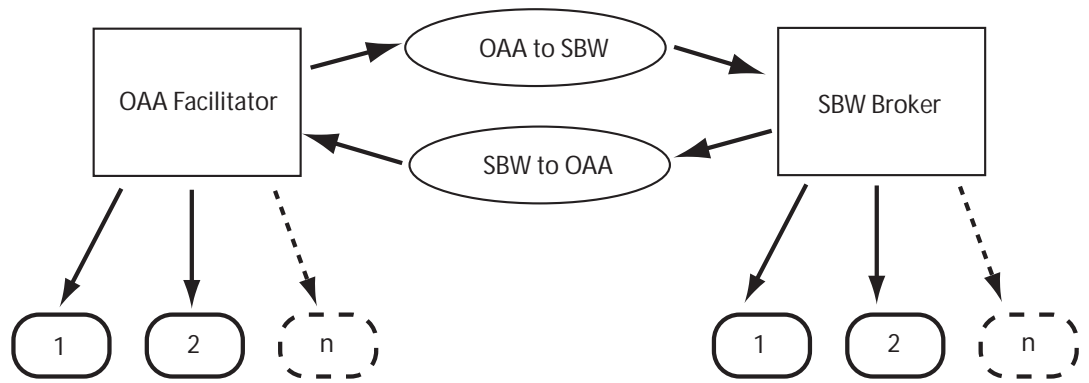
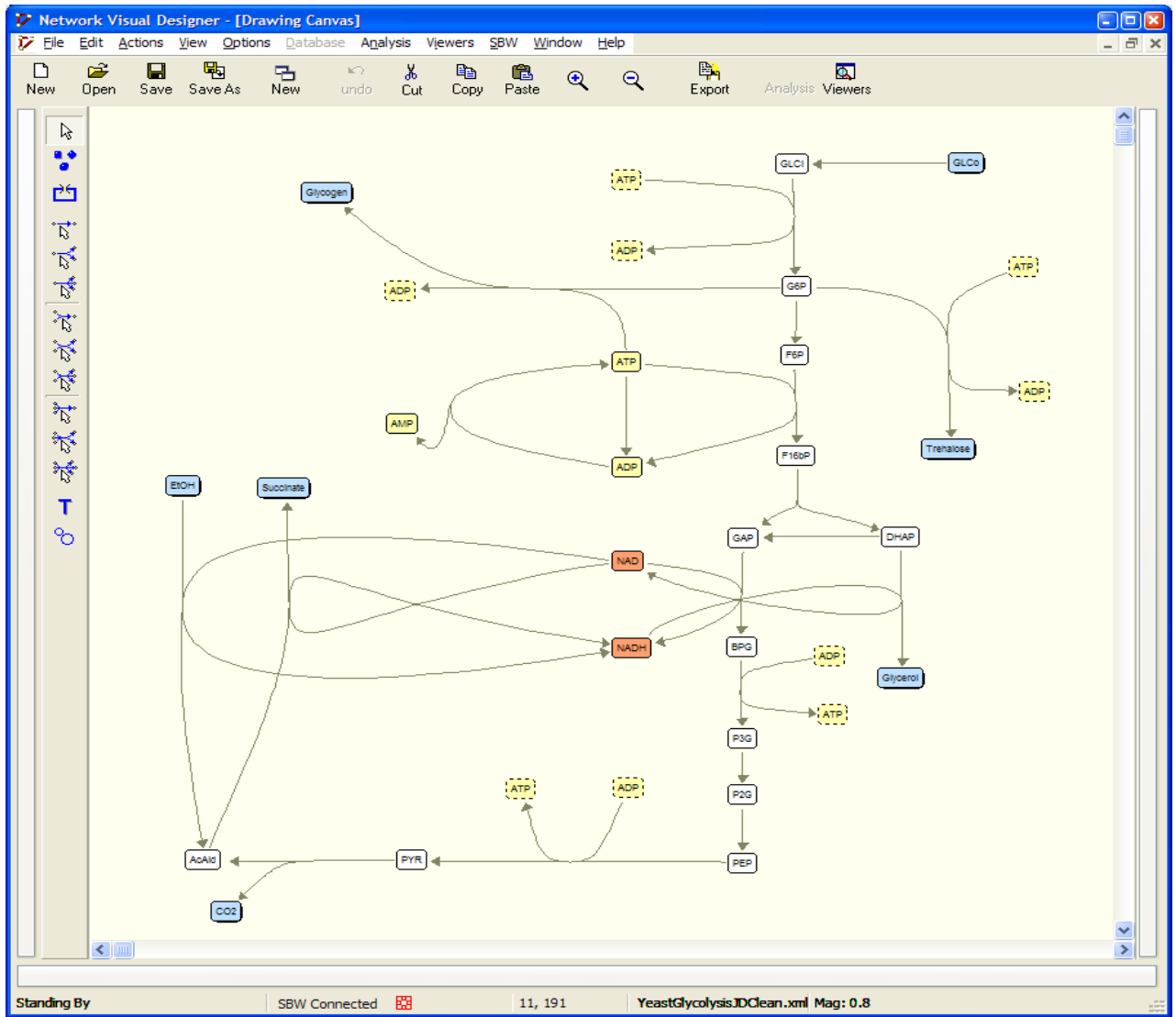Structure of the BioSPICE/SBW Bridge

Sauro et al.

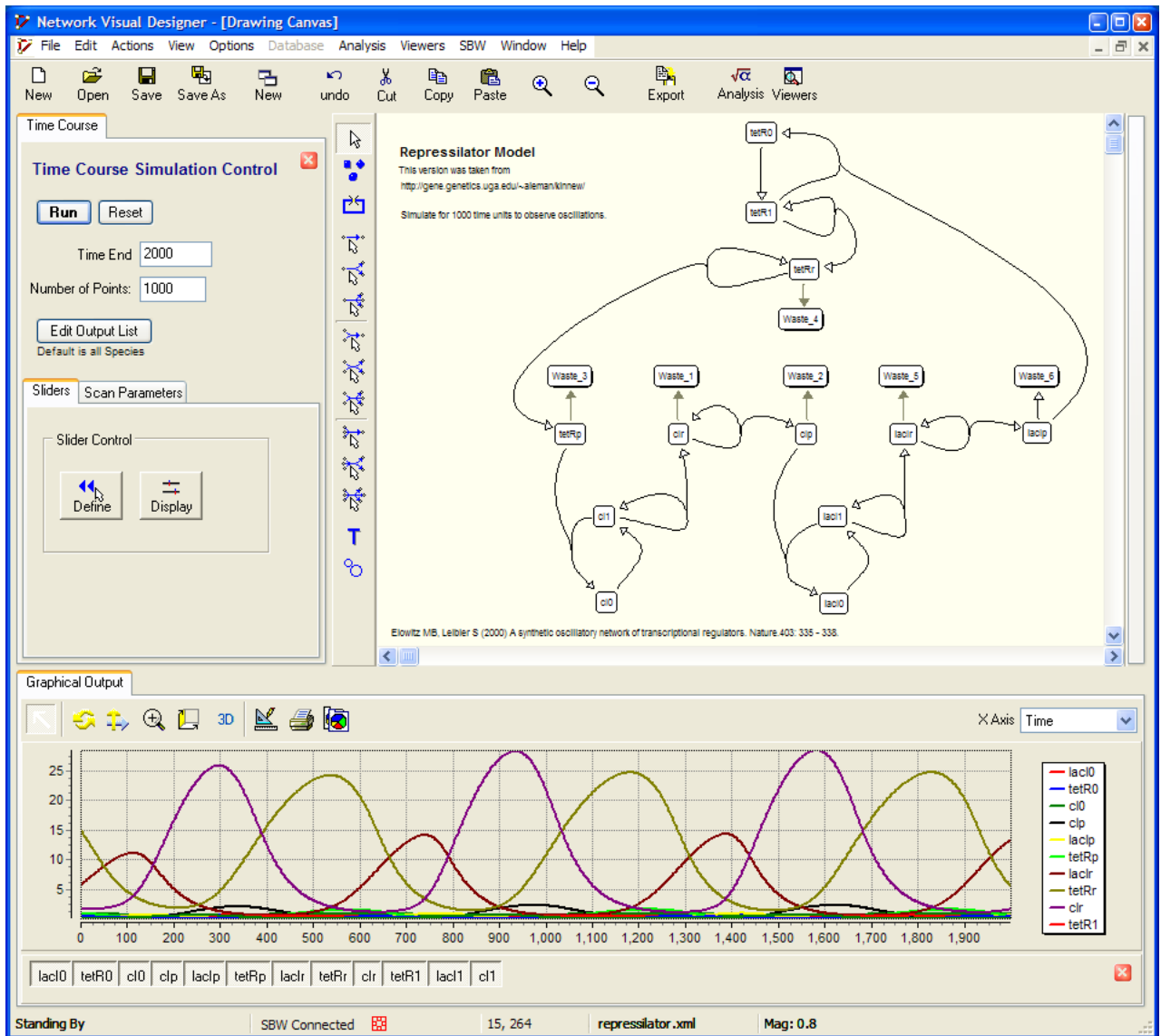Figure 9: JDesigner, illustrating a model of Glycolysis taken from (Pritchard & Kell, 2002) Sauro et al.

Figure 10: JDesigner and Jarnac working together to carry out and display the results from a simulation. The model was taken from (Elowitz & Leibler, 2000) which illustrates a synthetic oscillatory circuit that was constructed in *Escherichia coli*.
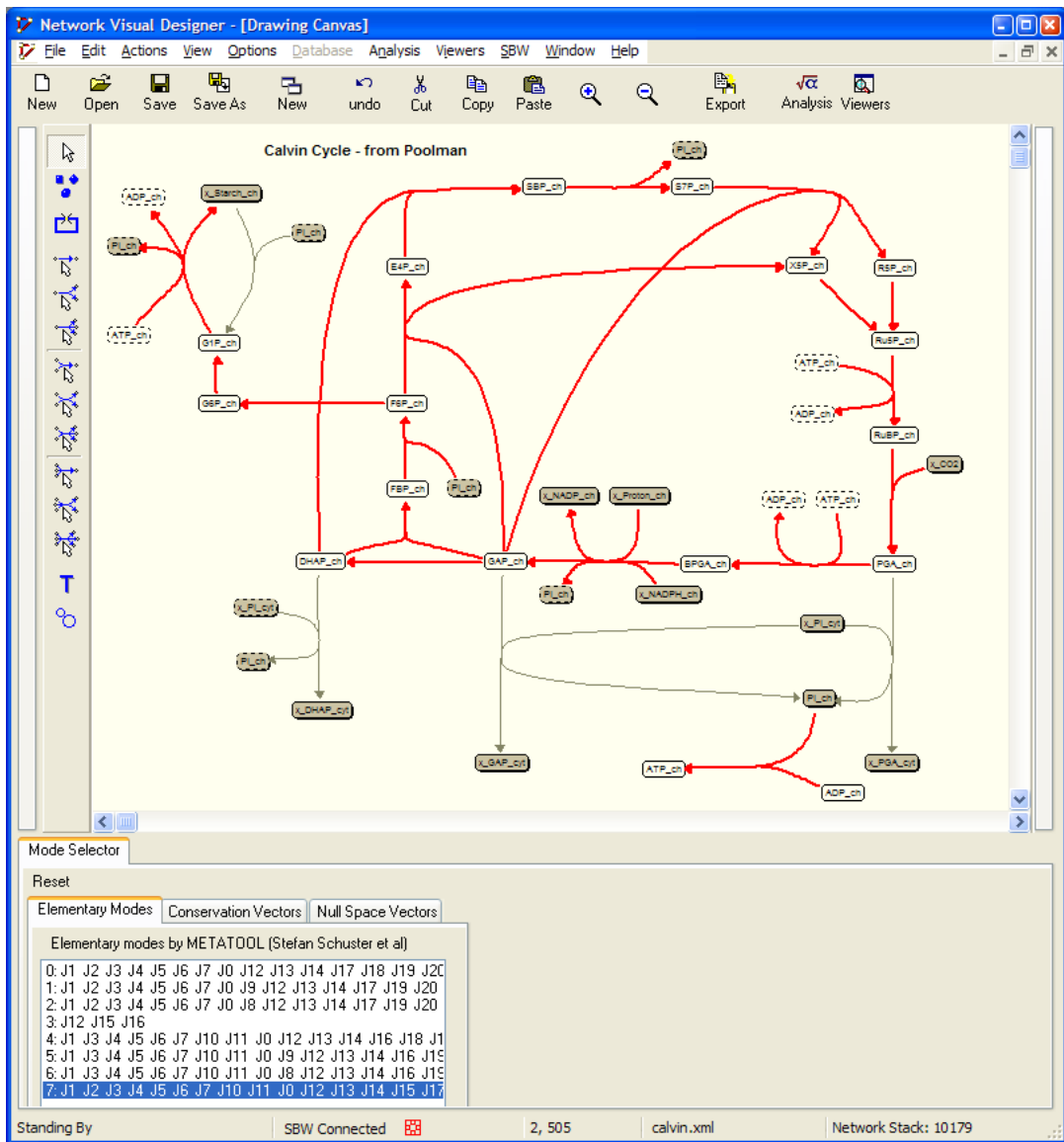
Sauro et al.

Figure 11: Operation of METATOOL with JDesigner illustrating the visualization of elementary modes in the Calvin Cycle(Model curtesy of M Poolman, D Fell and C Raines (Poolman *et al.*, 2003)

Sauro et al.