Proc. of the Pacific Biocomputing Symposium 2002 (PSB 2002), ed. R. Altman et al., 2002.

THE ERATO SYSTEMS BIOLOGY WORKBENCH: ENABLING INTERACTION AND EXCHANGE BETWEEN SOFTWARE TOOLS FOR COMPUTATIONAL BIOLOGY

M. HUCKA[1,2], A. FINNEY[1,2], H. M. SAURO[1,2] H. BOLOURI[1,2,3,4], J. DOYLE[1,2], H. KITANO[1,2,5]

[1] *ERATO Kitano Symbiotic Systems Project, M-31 Suite 6A 6-31-15 Jingumae Shibuya-ku, Tokyo 150-0001, Japan*

[2] *Control and Dynamical Systems 107-81, California Institute of Technology, CA 91125, USA*

[3] *Science and Technology Research Centre, University of Hertfordshire, AL10 9AB, UK*

[4] *Division of Biology 216-76, California Institute of Technology, CA91125, USA*

[5] *Sony Computer Science Laboratories, 3-14-13 Higashi-gotanda Shinagawa-ku, Tokyo 141-0022, Japan*

Researchers in computational biology today make use of a large number of different software packages for modeling, analysis, and data manipulation and visualization. In this paper, we describe the ERATO Systems Biology Workbench (SBW), a software framework that allows these heterogeneous application components—written in diverse programming languages and running on different platforms—to communicate and use each others' data and algorithmic capabilities. Our goal is to create a simple, open-source software infrastructure which is effective, easy to implement and easy to understand. SBW uses a broker-based architecture and 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 the SBW architecture and the current set of modules, as well as alternative implementation technologies.

1 Introduction

The ERATO Systems Biology Workbench (SBW) is a framework for allowing both legacy and new application resources to share data and algorithmic capabilities. Our target audience is the computational biology community whose interest lies in simulation and numerical analysis of biological systems. Our work has been motivated by the desire to achieve interoperability between a set of tools developed by our collaborators: *BioSpice* [1], *DBsolve* [2], *E-Cell* [3], *Gepasi* [4], *ProMoT/DIVA* [5], *Jarnac* [6], *StochSim* [7], and *Virtual Cell* [8]. Since

these applications are written in a variety of languages and run on a variety of platforms, it was essential not to limit integration capabilities to resources implemented in a single language or platform. SBW allows communication between processes potentially located across a network on different hardware and operating systems. SBW currently has bindings to C, C++, Java, Delphi and Python, with more planned for in the future, and it is portable to both Windows and Linux.

We are aware that our target community is largely not composed of professional software programmers. Any software development carried out in this community tends to be secondary to the main research effort. As a result, we have endeavored to make integration of software components into SBW as straightforward as possible. SBW is also an open-source framework, to allow the community to evolve and grow SBW with their changing needs. In addition, many laboratories have budgetary constraints. Unlike the licensing terms of a number of other frameworks, our use of an open-source license (GNU Lesser General Public License, LGPL [9]) guarantees that SBW will remain available at no cost indefinitely, while simultaneously allowing developers the freedom to release closed-source modules that work with SBW.

SBW does not attempt to be more than a mechanism to enable the integration of applications. The architecture of SBW does not exclude its integration with other frameworks and integration technologies. In fact, we hope to integrate other frameworks to extend the functionality available to users and developers. In many configurations, SBW will be a small component in a larger system (size being quantified as CPU, disk and memory usage).

We begin by describing SBW from the user's perspective in Section 2, then from the developer's perspective in Section 3. In Section 4, we go on to compare its features to those of other tools for building interoperable software. Finally, in Section 5 we describe the various modules made available in the initial beta release of SBW in November 2001.

2 SBW from the User's Perspective

When an application has been modified to interact with SBW, we describe it as being *SBW-enabled*. This means the application can interact with other SBW-enabled applications. The kinds of possible interactions depend on the facilities that have been exposed to SBW by the applications' programmers. Typical SBW-enabled applications also provide for ways of exchanging models and data using an XML-based common representation format, the Systems Biology Markup Language (SBML) [10]. SBW is not a controller in the system—the flow of control is entirely determined by what the individual modules and the user

do. SBW doesn't define any particular type of user interaction with modules: the user can control modules from either script language interpreter, GUIs or some hybrid of GUI and interpreter. The interpreter approach is inevitably more flexible enabling access to all modules in the SBW environment in a single application environment. In the remainder of this section, we present a scenario where the user is controlling events using via GUIs only.

A user will typically start up the first SBW-enabled application as they would any other program. The user doesn't need to do anything specific to start SBW itself. Figure 1 shows an example of using a collection of SBW-enabled software modules. The upper left-hand area in the figure (partly covered by other windows) shows an SBW-enabled version of JDesigner[11], a visual biochemical network layout tool. This module's appearance is nearly identical to that of its original non-SBW-enabled counterpart, except for the presence of a new item in the menu bar called **SBW**. This is typical of SBW-enabled programs: the SBW approach strives to be minimally intrusive. In this example, the user has created a network model in JDesigner, then has decided to run a time-series simulation of the model. To do this, the user has pulled down the **SBW** menu and selected one of the options listed, *Jarnac Analysis*, to invoke the SBW-enabled simulation program *Jarnac*[6]. This has brought forth a control GUI, shown underneath the plot window in the lower right-hand area of Figure 1; the user has then input the necessary parameters into the control GUI to set up the time-series simulation, and has finally clicked the **Run** button in the GUI to start the simulation.

In this example, the control GUI used SBW calls to instruct the simulation module (Jarnac) to run with the given parameters and send the results back to the controlling GUI module, which then sent the results to a plotting module. This example scenario illustrates the interactions involved in using SBW and four modules: the visual JDesigner, the computational module Jarnac, a time-series simulation control GUI, and a plotting module.

3 SBW from the Developer's Perspective

SBW uses a *broker-based, message-passing architecture* that allows dynamic extensibility and configurability. As mentioned above, software modules in SBW can 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. All remote calls run in their own threads. As shown in Fig. 2, 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

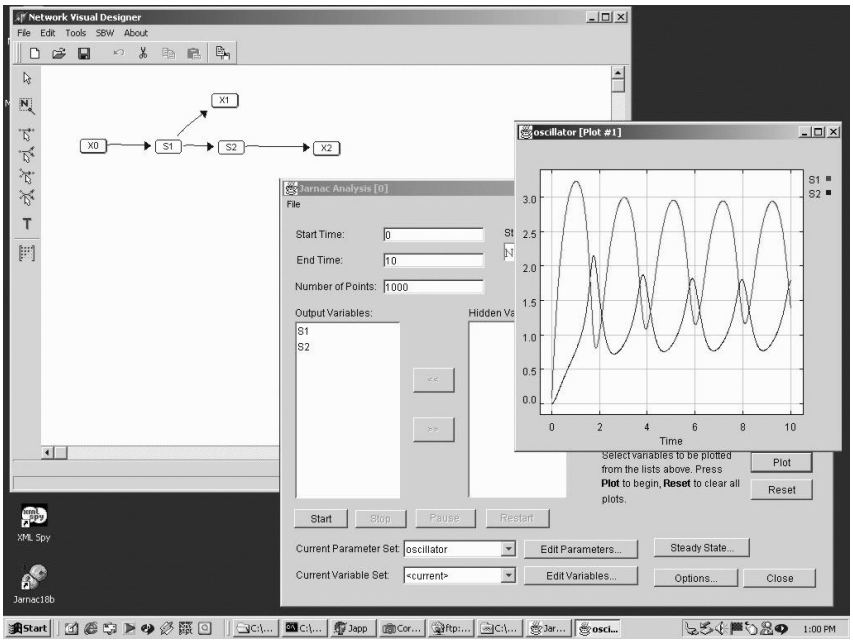

Figure 1. Example of applications interacting through SBW.

between them. Communications are implemented using a fast, lightweight system with a straightforward programming interface.

Broker-based architectures are a common software pattern [12]. They 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*, another tried and proven approach [13,14]. 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 *dynamic extensibility and configurability* quality of SBW is that components—i.e., SBW modules—can be easily exchanged, added or removed, even at run-time, under user or program control.

From the application programmer's point of view, it is preferable to isolate communications details from application details. For this reason, we provide an

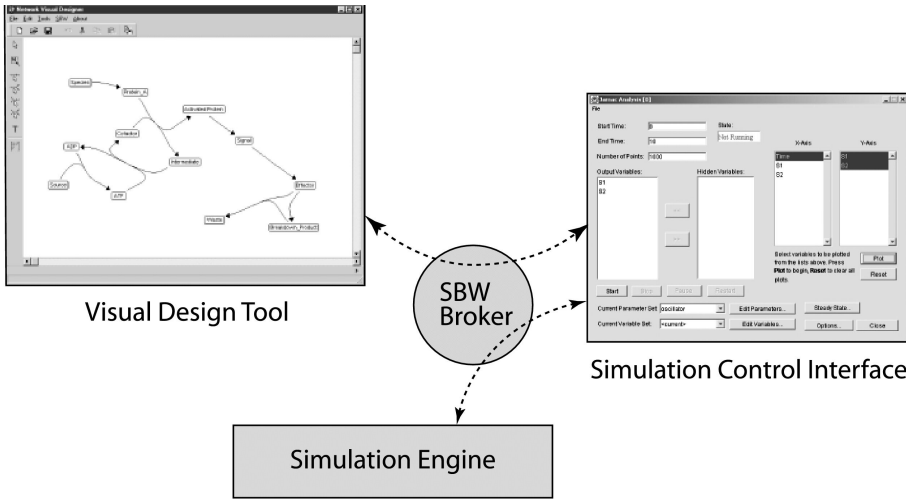

Figure 2. Illustration of the relationship between the SBW Broker and SBW modules. At the API level, communications appear to be direct (i.e., module-to-module); however, the implementation passes messages back-and-forth through the SBW Broker.

Application Programming Interface (API)[15] that hides the details of constructing and sending messages and provides ways for methods in an application to be "hooked into" the messaging framework.

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.

As an example of how simple the high-level API is to use in practice, the following is Java code demonstrating how one might invoke a simulator from a hypothetical module:

```
// Define the interface for the Java compiler.
interface Simulator
{
  void loadSBML(string);
  void setTimeStart(double);
  void setTimeEnd(double);
  void setNumPoints(integer);
  double[] simulate();
}

double[] runSimulation(String modelDefinition, double startTime,
                       double endTime, integer numPoints)
{
  try
  {
    // Start a new instance of the simulator module.
    Module module
      = SBW.getModuleInstance("edu.caltech.simulator");

    // Locate the service we want to call in the module.
    Service srv = module.findServiceByName("simulation");
    Simulator simulator
      = (Simulator) srv.getServiceObject(Simulator.class);

    // Send the model to the simulator and set parameters.
    simulator.loadSBML(modelDefinition);
    simulator.setTimeStart(startTime);
    simulator.setTimeEnd(endTime);
    simulator.setNumPoints(numPoints);

    // Run the simulation and return the result.
    return simulator.simulate()

  } catch (SBWException e) {
    // Handle problems here.
  }
}
```

As the example above shows, using an SBW-enabled resource involves getting a reference to the module that implements a desired service and invoking methods on that service.

4 Comparison to Related Efforts

The idea of creating a framework that enables the integration of disparate software packages is not new. When we began this project, we considered using an existing framework and simply augmenting it with additional facilities. But after examining a number of other options, we were forced to conclude that none of the existing systems provided an adequate combination of sim-

plicity, support for major programming and scripting languages, support for dynamically querying modules for services they offer, support for distributed computing on Windows and Linux (with a clear ability to be ported to other platforms), and free availability of open-source implementations for Windows and Linux.

4.1 Frameworks for Computational Biology

One of the projects most similar to SBW is ISYS [16]. This system provides a generalized platform into which components may be added in whatever combination the user desires. The system provides a bus-based communications framework that allows components to interoperate without direct knowledge of each other, by using a publish-and-subscribe approach in which components place data on the bus and other components can listen for and extract the data when it appears. ISYS components include graphical visualization tools and database access interfaces. This style of interoperability is an alternative to the more direct communications in SBW, but could be used to the same ends.

The main drawbacks of ISYS for our goals is that it is not freely available and distributable. Moreover, it is largely Java-based and does not offer direct support for components written in other languages.

4.2 General-Purpose High-Level Frameworks

In terms of communications frameworks, SBW has many similarities to Java RMI [17] and CORBA [18]. Both of the latter technologies enable a programmer to tie together separate applications potentially running on different computers, and both offer directory services so that modules can dynamically query and discover the services being made available by other modules. Unfortunately, Java RMI is only truly practical when all applications are written in Java, conflicting with our goal of supporting as many languages as possible. Although RMI-over-IIOP [19] is an option, this simply means that the non-Java components would have to use CORBA.

CORBA is the industry standard for broker-based application object integration. We decided against using CORBA as the basis of SBW primarily because of issues of standards compliance, complexity and maintenance. CORBA is a large and complicated standard and has a steep learning curve. We felt it would have been too much to ask of most researchers, whose time is limited and main goals are in developing domain-specific applications, to acquire CORBA development skills. Further, there are no open-source, standard-compliant implementations of CORBA that support sufficiently many languages in the same implementation. The implication is that SBW modules written in dif-

ferent languages would have to interact with CORBA implementations from different open-source projects. We were concerned about the difficulties of managing not only compatibility of different CORBA packages, but also the installation process, user documentation, and long-term maintenance.

Notwithstanding these issues, we are not in principle opposed to using CORBA. Indeed, we plan to design an interface that will provide a CORBA bridge to SBW for those developers who prefer to use this technology.

4.3 Low-Level Communications Frameworks

SBW uses a custom message-passing communications layer with a simple tagged data representation format and a specialized protocol layered on top of TCP/IP sockets. We examined several alternatives before implementing the scheme used in SBW.

Two attractive, recent alternatives were SOAP [20] and XML-RPC [21]. The latter is essentially a much-simplified version of the former; both provide remote procedure calling facilities that use HTTP as the protocol and XML as the message encoding. We performed an in-depth comparison of XML-RPC and SBW's messaging protocols [22] and concluded that XML-RPC and SOAP would not work for the goals of SBW. The HTTP and XML layers impose a performance penalty not present in SBW's simpler protocol and encoding scheme. Further, the HTTP protocol is not bidirectional: HTTP is oriented towards client-server applications in which a client initiates a connection to a server listening on a designated TCP/IP port. The implication of using XML-RPC for SBW is that each module would have to listen on a different TCP/IP port. This would add needless complexity to SBW.

Another alternative for the message-passing functionality in SBW is MPI [14]. We declined using MPI primarily because at this time there does not appear to be a standard Java interface, and because MPI is considerably more complex than the simple message-passing scheme used in SBW. However, MPI remains an option for reimplementing the communications facility in SBW if it proves useful to do so in the future.

5 SBW Modules

In this section, we describe a variety of different modules that we have implemented and released with the SBW beta release in November 2001.

5.1 Inspector Module

The inspector module is a GUI based tool which allows a user to explore the SBW environment. It enables other modules and their services and methods to be inspected. In the future we hope to extend the inspector module to enable individual methods of a module service to be executed, in which case the inspector will provide an excellent tool for testing new modules.

5.2 JDesigner

JDesigner [11], developed by Herbert Sauro, allows users to draw biochemical networks on screen. It can save models in SBML [10] format. We provide an SBW interface to JDesigner which allows other modules connected to SBW to gain access to the functionality of JDesigner. In particular, it is possible for a remote module to request SBML code from JDesigner. In addition, we also provide an interface which allows remote modules to control many details of JDesigner, for example providing the ability to rearrange the network on-screen.

JDesigner has a menu option "SBW" which lists the services registered with the SBW Broker in the "Analysis" category (e.g. the MATLAB Model Generator, described below). JDesigner passes the SBML representing the drawn model to the selected service.

5.3 Network Object Model

The most frequently requested module is some means of parsing and interpreting SBML. SBML is defined in terms of XML and for many developers in our community it is a non-trivial task to code a parser for SBML. We have therefore written a module, called the Network Object Model or NOM with methods that load and generate SBML as well as methods for accessing and modifying the object model constructed from the loaded SBML. The NOM module can be used as a SBML clipboard for moving data between applications.

5.4 Optimization Module

A frequently need in modeling is the ability to fit parameters to a model. This problem is recast as the minimization of some predefined fitting function by adjusting model parameters. We have collaborated with Pedro Mendes to allow user access to the extensive optimization algorithms in Gepasi.

5.5 Plotting Module

This module provides a 2D graph plotting service.

5.6 MATLAB Model Generator

This translation module creates either ODE or Simulink models for MATLAB[23] from SBML. This modules provides services in the "Analysis" category and thus can be invoked from JDesigner or similar modules. We anticipate integrating the MATLAB application itself as a module in later releases enabling SBW modules to be invoked from the MATLAB command line and scripts.

5.7 Simulation Control GUI

The simulation control GUI is a non-scripting interface to a simulation server such as Jarnac. The simulation control GUI service is in the "Analysis" category. The GUI enables users to set up simulation runs, edit parameters or variables and plot the resulting runs. In addition the GUI can also be used to compute the steady state and carry out metabolic control analysis. Any simulator in the "Simulation" category can be controlled from this GUI interface. The Gillespie, Gibson and Jarnac modules (see below) provide services in that category.

5.8 Gillespie Stochastic Simulator

The stochastic simulator module is based on the Gillespie algorithm[24]. The code which forms the basis of this module was provided by Baltazar Aguda[25]. Once a model is loaded, the module allows a user to change parameters and variables in addition to graphing of results and collection of run data.

5.9 Gibson Stochastic Simulator

This stochastic simulator uses an algorithm[26], developed by Gibson and Bruck, based on the Gillespie algorithm which includes optimizations to reduce simulation run times.

5.10 Jarnac Simulator

Jarnac[6] is an ODE-based biochemical network simulator. Simulations are controlled via a scripting language. Services supported by Jarnac include matrix manipulation, time-course simulation, steady-state analysis and metabolic

control analysis. Adding an SBW interface to Jarnac permits two types of interaction. In one mode, Jarnac can act as a server for carrying out simulations. This allows users access to the capabilities of Jarnac without having to interact with a scripting interface.

The second mode of operation is from the scripting interface itself. In this mode, the user is able to explore and use SBW modules from a command line. Interaction is achieved by requesting Jarnac to create a Jarnac object interface to the desired module. This allows a user to use a module as if it were part of Jarnac itself.

6 Summary

The SBW is a very flexible and straightforward system to integrate a range of heterogeneous software components written in a variety of languages and running on a variety of platforms. At the time of this writing, we have completed the implementation of the SBW Broker and the libraries that implement the SBW protocol in Delphi, C, C++, and Java. Full documentation of the SBW design is available from the project web site [27]. A beta release of the SBW software and several sample modules was made in November, 2001, and is available from the project web site.

Acknowledgments

This work has been funded by the Japan Science and Technology Corporation under the ERATO Kitano Systems Biology Project. The Systems Biology Workbench has benefitted from the input of many people. We wish to acknowledge in particular the authors of BioSpice, DBsolve, Cellerator, E-Cell, Gepasi, ProMoT/DIVA, StochSim, and Virtual Cell, and the members of the `sysbio` mailing list. We also thank Mark Borisuk, Mineo Morohashi and Tau-Mu Yi for support, comments and advice.

References

1. A. P. Arkin, *Simulac* and *Deduce*, `http://gobi.lbl.gov/~aparkin/Stuff/Software.html` (2001).
2. I. Goryanin, T. C. Hodgman, and E. Selkov, *Bioinformatics* **15**(9), 749–758 (1999).
3. M. Tomita et al., *Bioinformatics* **15**(1), 72–84 (1999).
4. P. Mendes, *Trends in Biochemical Sciences* **22**, 361–363 (1997).
5. M. Ginkel et al., in *Proceedings of the 3rd MATHMOD*, ed. I. Troch and F. Breitenecker, 525–528 (2000).

6. H. M. Sauro in *Animating the Cellular Map*, ed. J-H. S. Hofmeyr, J. M. Rohwer, and J. L. Snoep (Stellenbosch University Press, 2000).
7. D. Bray, C. Firth, N. Le Novère, and T. Shimizu, *StochSim*, `http://www.zoo.cam.ac.uk/comp-cell/StochSim.html` (2001).
8. J. Schaff, B. Slepchenko, and L. M. Loew, in *Methods in Enzymology*, ed. M. Johnson and L. Brand, **321**, 1–23 (Academic Press, 2000).
9. Free Software Foundation, GNU Lesser General Public License, `http://www.gnu.org/copyleft/lesser.html` (1991).
10. M. Hucka, A. Finney, H. M. Sauro, and H. Bolouri, Systems Biology Markup Language (SBML) Level 1, `http://www.cds.caltech.edu/erato` (2001).
11. Herbert S. Sauro, JDesigner: A simple biochemical network designer, `http://members.tripod.co.uk/sauro/biotech.htm` (2001).
12. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns* (John Wiley & Sons, 1996).
13. Jim Farley, *Java Distributed Computing* (O'Reilly & Associates, 1998).
14. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface* (MIT Press, 1999).
15. A. Finney, H. Sauro, M. Hucka, and H. Bolouri, Programmer's manual for the Systems Biology Workbench (SBW), `http://www.cds.caltech.edu/erato/sbw/docs/api/` (2001).
16. A. Siepel et al., *Bioinformatics* **17**(1), 83–94 (2001).
17. M. Hughes, M. Shoffner, and D. Hamner, *Java Network Programming* (Manning Publications Co., 1999).
18. OMG, *CORBA* Specification, `http://www.omg.org` (2001).
19. Sun Microsystems, RMI over IIOP, `http://java.sun.com/products/rmi-iiop/` (2001).
20. D. Box et al., Simple Object Access Protocol (SOAP) 1.1: W3C note 08 May 2000, `http://www.w3.org/TR/SOAP/` (2000).
21. D. Winer, XML-RPC, `http://www.xmlrpc.com/spec/` (2001).
22. M. Hucka, A. Finney, H. M. Sauro, and H. Bolouri, A comparison of two alternative message-passing approaches for SBW, `http://www.cds.caltech.edu/erato/sbw/docs/xml-rpc-comparison/` (2001).
23. Mathworks, Matlab, `http://www.mathworks.com/products/matlab/` (2001).
24. D. Gillespie, *J. Comput. Phys.* **22**, 403–434 (1976).
25. Baltazar Aguda, personal communication.
26. M. Gibson and J. Bruck, *J. Phys. Chem.* **104**, 1876–1889 (2000).
27. The ERATO Systems Biology Workbench Development Group, `http://www.cds.caltech.edu/erato/` (2001).