Introduction to the Systems Biology Workbench

Michael Hucka, Andrew Finney, Herbert Sauro, Ben Kovitz,
Sarah Keating, Joanne Matthews, Hamid Bolouri

Systems Biology Workbench Development Group
ERATO Kitano Symbiotic Systems 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

August 8, 2003

1 Introduction

The goal of the *Systems Biology Workbench* (SBW) project is to create an open-source, integrated software environment for systems biology that enables sharing of models and resources between simulation and analysis tools. The project grew out of a desire to achieve interoperability between a number of independent software packages used in biochemical network simulation and analysis. The packages in particular were: *BioSpice* (Arkin, 2001), *Cellerator* (Shapiro and Mjolsness, 2001), *DBsolve* (Goryanin et al., 1999), *E-CELL* (Tomita et al., 2001), *Gepasi* (Mendes, 1997), *Jarnac* (Sauro, 2000), *StochSim* (Morton-Firth and Bray, 1998), and *Virtual Cell* (Schaff et al., 2001).

SBW uses a simple, portable architecture that currently supports Linux, FreeBSD and Windows, with support for other platforms expected in the future. SBW libraries are available for many different programming languages: C, C++, Delphi, Java, Perl, and Python. Software applications that implement different functions (such as GUIs, model simulation methods, analysis methods, etc.) can be connected to each other through SBW using a straightforward application programming interface (API). The SBW framework is implemented using well-established and transportable technologies such as network sockets and SSH. We deliberately avoided the use of platform- and language-specific technologies such as Java RMI and COM as well as more complicated systems such as CORBA. We also made all aspects of SBW open-source under the GNU LGPL.

In this document, we provide an overview of SBW and outline its general organization and architecture. Separate documents (Finney et al., 2002a,b,c; Hucka et al., 2002; Sauro et al., 2002, 2003) provide detailed explanations and tutorials on programming with SBW in the different languages supported.

1.1 Motivations

The tremendous amounts of data and research in molecular biotechnology have fueled an explosion in the development of computer tools by research groups across the world. This explosive rate of progress in tool development is exciting, but the rapid growth of the field has been accompanied by problems and pressing needs. One problem is that simulation models and results often cannot be compared, shared or re-used directly because the tools developed by different groups often are not compatible with each other. As the field of systems biology matures, researchers increasingly need to communicate their results as computational models rather than box-and-arrow diagrams. But they also need to reuse each other's published and curated models as library elements in order to succeed with large-scale efforts (e.g., the Alliance for Cellular Signaling, Gilman, 2000; Smaglik, 2000). These needs require that models implemented in one software package be portable to other software packages, to maximize public understanding and to allow building up libraries of curated computational models. In an attempt to define and establish a common exchange language for computational models in biology, we began the Systems Biology Markup Language [SBML] project, a separate effort from SBW that is still ongoing (Hucka et al., 2003, 2001b; Finney, 2003)

A second problem is that software developers often end up duplicating each other's efforts when implementing different packages. The reason is that individual software tools typically are designed to address a specific set of issues, reflecting the expertise and preferences of the originating group. As a result, most packages have niche strengths which are different from, but complementary to, the strengths of other packages. But because the packages are separate systems, developers end up having to re-invent and implement much general functionality needed by every simulation/analysis tool. The result is duplication of effort in developing software infrastructure.

No single package currently answers all the needs of the emerging systems biology community, despite an emphasis by many developers to continue adding more features to their software tools. Nor is such a scenario likely: the range of tools needed is vast, and new techniques requiring new tools are emerging far more rapidly than the rate at which any single package may be developed. For the foreseeable future, then, systems biology researchers are likely to continue using multiple packages to carry out their work. The best we can do is to develop ways to ease sharing and communication between such packages now and in the future.

These considerations lead us to believe that there is an increasingly urgent need to develop common standards and mechanisms for sharing resources within the field of systems biology. This is the motivation for the ERATO Systems Biology Workbench (SBW) project.

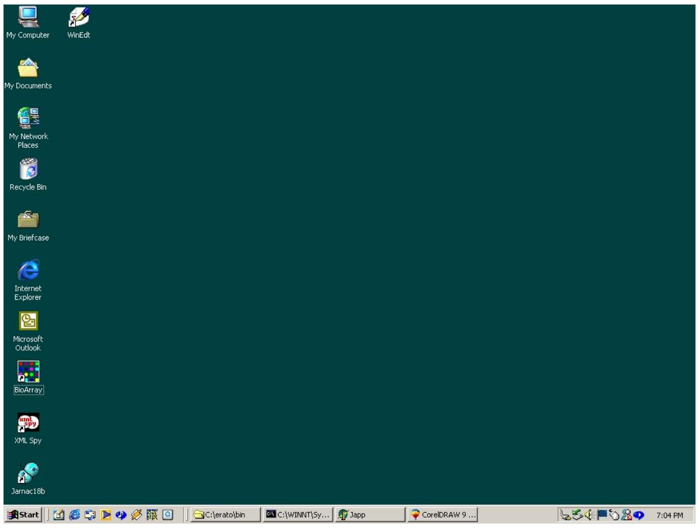
2 SBW from the User's Perspective

The Systems Biology Workbench is primarily a system for integrating resources. It provides infrastructure that can be used by software developers to interface software components together and enable them to communicate with each other. The components may be simulation codes, analysis tools, user interfaces, database interfaces, script language interpreters, or in fact any piece of software that conforms to a certain well-defined interface summarized in Section 3. From a user's point of view, then, SBW is very nearly invisible—it operates behind the scenes.

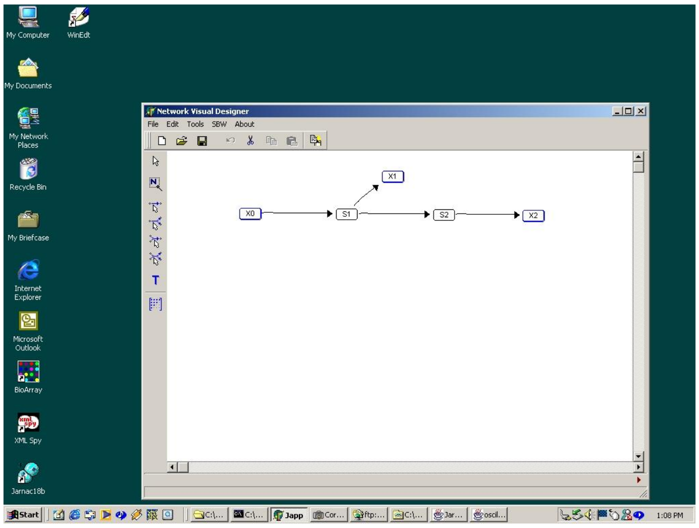
When an application has been designed to interact with SBW, we describe it as being *SBW-enabled*. From the user's perspective, this simply means that the application can interact with other SBW-enabled applications. The kinds of interactions that are possible depend on which facilities have been exposed to SBW by the application's programmers. Basic functionality includes finding out what other SBW-enabled applications are available and being able to start up applications on command; typical SBW-enabled applications also provide for ways of exchanging models and data.

In the following paragraphs and screen images, we present an example scenario involving SBW-mediated interactions between applications. The example described here comes from a PC running Microsoft Windows 2000, but essentially the same behavior is experienced when running SBW on other platforms.

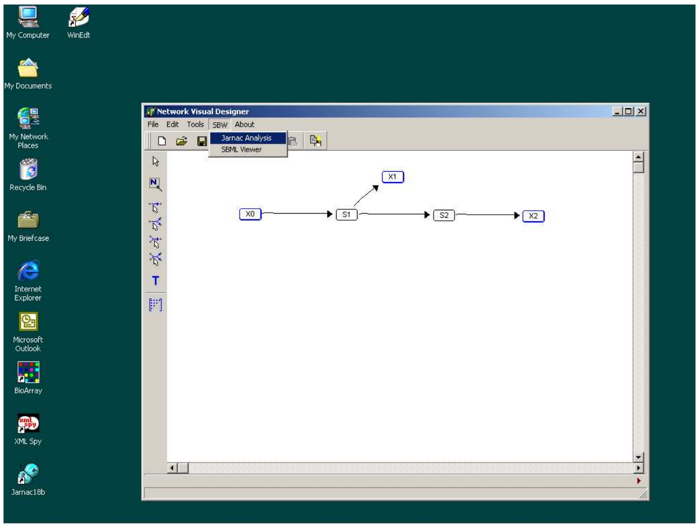
Step 1: A user will typically start up the first SBW-enabled application as they would any other program—by selecting it from a menu or double-clicking an icon on the desktop. The user does not need to do anything special to start up SBW. The SBW Broker is started automatically by the SBW-enabled application if the Broker is not already running on the user's machine. Under Windows, SBW also installs a small icon in the Windows system tray that provides access to a simple control interface for SBW configuration.



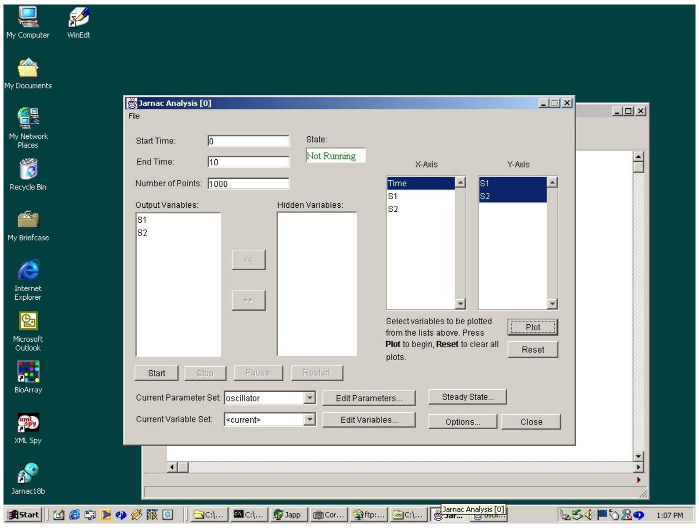
Step 2: A user will typically proceed to do some work in the first SBW-enabled application they start up. Here we show *JDesigner* (Sauro, 2001), a visual biochemical network layout tool. The SBW-enabled version of JDesigner appears nearly identical to its original non-SBW-enabled counterpart, except for the presence of a new menu item, **SBW**, in the menu bar. This is typical of SBW-enabled programs: the SBW design strives to be minimally intrusive.



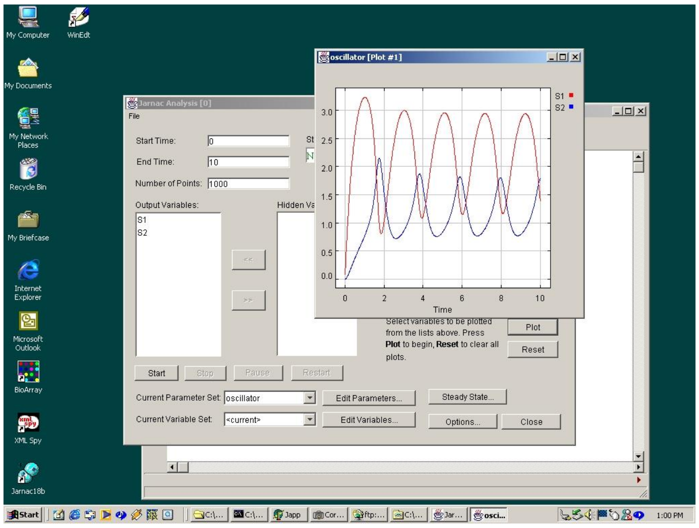
Step 3: The user may create a model of a network in JDesigner, then decide to run a time-series simulation of the model. To do this, she or he pulls down the **SBW** menu and selects one of the options listed. In this example, the user picks *Jarnac Analysis*, to invoke a simulation using the SBW-enabled program *Jarnac* (Sauro, 2000).



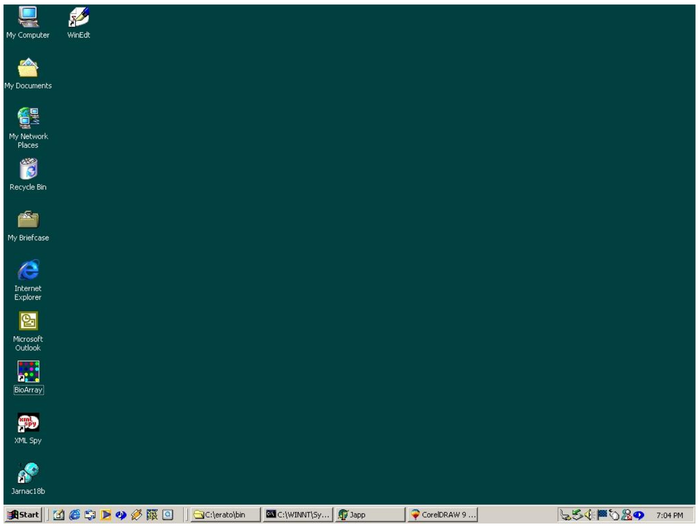
Step 4: SBW software components (called SBW *modules*) can come in many forms. A module may be primarily computational and lack a GUI, or it may be a computational module having its own GUI, or it may consist solely of a GUI designed to control other tools. Here we show an example of a GUI-only module: an interface for time-series simulations that controls Jarnac through SBW. The user sees only the graphical interface, but behind the scenes, Jarnac is also running as a third module started up by SBW.



Step 5: The user inputs the necessary parameters to set up the time-series simulation, then clicks the **Run** button in the GUI to start it. The control GUI uses SBW calls to instruct the simulation module (Jarnac) to run with the given parameters and send the results back to the controlling GUI module. In this example implementation, the control GUI buffers the results and allows the user to visualize them using a plotting facility.



Step 6: In the default configuration, when the last SBW-enabled application exits after the user finishes a particular work session, the SBW Broker remains resident in the background. Under Windows, the icon for SBW remains in the task bar. (The Broker is efficiently designed; it does not consume CPU cycles when it is not active. Leaving the Broker running provides easy access to the SBW configuration interface and it speeds up interactions between SBW-enabled applications.)



This example scenario illustrates the interactions involved in using SBW and three sample modules: the visual JDesigner, the computational Jarnac, and a generic simulation control GUI. The basic process described above can be extended to more modules; for example, the user could have chosen to send the results of the time-series simulation to an another module for analysis. This type of chaining is possible in SBW because SBW does not impose a particular sequence on the tasks to be performed.

The style of interaction described here, with different modules taking center stage in turn, is meant to provide context for what the user is doing at any given moment. The underlying assumption is that each module/application is itself best suited to providing that context, lending a sense of familiarity and *place* to the task it implements. This style of interaction involves a trade-off. On one hand, it is a good choice when users are interacting with multiple applications that were developed independently of SBW, especially when users are already familiar with those applications. On the other hand, it does not provide an overall integrated look-and-feel. Strictly speaking, SBW itself does not impose the style of interaction described here, but the modules we develop for SBW use this style of interaction because it appears to make the most sense when trying to bridge separate applications.

3 SBW from the Developer's Perspective

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. Interactions are mediated through the SBW Broker, a background program running on a user's computer; the Broker enables locating and starting other modules and establishing communications links between them. The Broker is started automatically the first time it is needed by a module. Communications are implemented using a fast, lightweight, message-passing system having a straightforward programming interface.

Developers who wish to use SBW to implement new software modules, or to modify existing modules and make them capable of interacting with other SBW-aware components, need to know the following:

- The overall architecture of SBW and how modules fit in; and
- The SBW APIs and how to use them.

The rest of this section addresses these points at an introductory, summary level. The SBW API manuals provide greater detail (Finney et al., 2002a,b,c; Hucka et al., 2002; Sauro et al., 2002, 2003).

3.1 The SBW Architecture Design

The Systems Biology Workbench borrows ideas from a number of other software systems, including such things as Java RMI and the Linux KDE window environment (Sweet, 2001). Succinctly put, SBW is a *dynamically-extensible, broker-based, message-passing architecture.*

Broker architectures are relatively common and are considered to be a well-documented software pattern (Buschmann et al., 1996); 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 software technology (e.g., Farley, 1998; Gropp et al., 1999; Lee, 1998). Message-passing systems implement inter-component communications 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.

Broker-based systems are appropriate when the software components that need to interact are decoupled and independent, and the components must be able to access services through remote, location-transparent invocations (Buschmann et al., 1996). The *dynamically-extensible* 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.

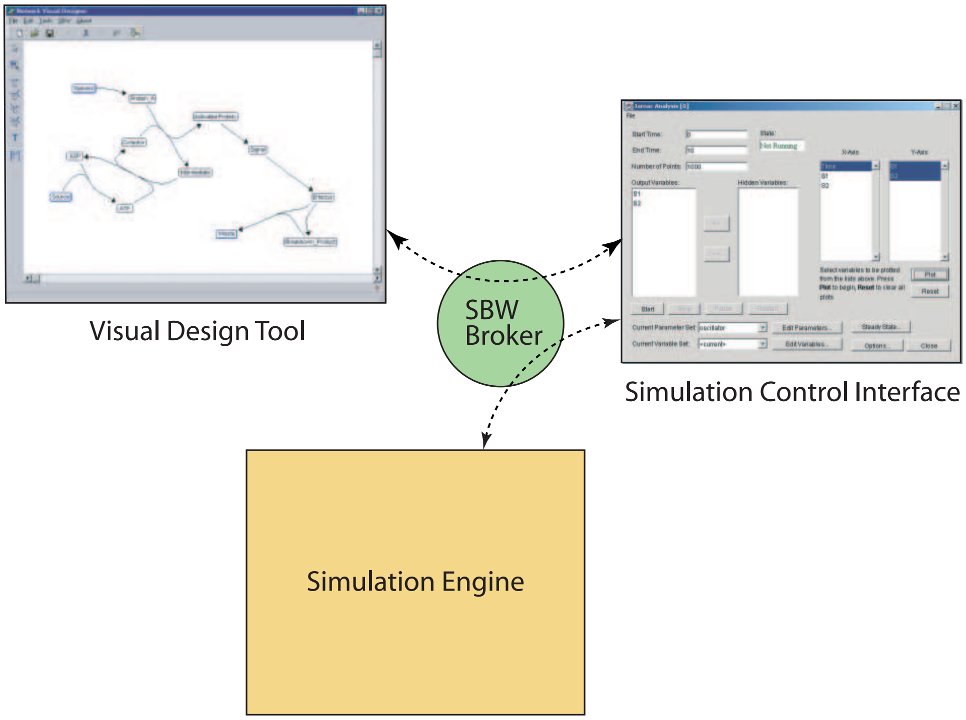


Figure 1: *Illustration of the relationship between the SBW Broker and SBW modules. Programmers can interface their modules to the SBW system by using the SBW interface libraries; the libraries implement the necessary APIs that allow a module to communicate with other modules via the SBW framework. To the modules themselves, communications appear to be direct (i.e., module-to-module); underneath, however, messages pass back-and-forth through the SBW infrastructure.*

The goals of this design are to maximize flexibility, maintainability and changeability.

We developed SBW's simple message-passing infrastructure after evaluating and rejecting several other alternatives, including CORBA (OMG, 2001; Seetharaman, 1998; Vinoski, 1997), XML-RPC (McLaughlin, 2000; UserLand Inc., 2001; Winer, 2001), Java RMI (Steflik and Sridharan, 2000; Sun Microsystems, 2001), MPI (Gropp et al., 1999), and other technologies. We judged these alternatives either too complex to work with, or too slow. Our test results comparing SBW's message-passing mechanisms to XML-RPC are available in a separate document (Hucka et al., 2001a).

Message-passing schemes traditionally suffer from one drawback: as the messaging scheme becomes more elaborate, it becomes more complicated and error-prone to program at the application level. From the application programmer's point of view, it is preferable to isolate communications details from application details. For this reason, we provide two levels of programming interfaces in SBW: a low-level API that consists of the basic operations for constructing and sending messages, and a high-level API that hides some of the details of constructing and sending messages and provides ways for methods in an application to be "hooked into" the messaging framework at a higher level.

3.2 Adapting Software to Use SBW

We strove to develop a high-level API for SBW that provides a natural interface in each of the different languages for which we have implemented libraries so far. By "natural", we mean that it uses a style and features that programmers accustomed to that language would find familiar. For example, in Java, the high-level API is oriented around providing SBW clients with objects whose methods implement the operations that another application exposes through SBW. We hope that by using the libraries and high-level API, application developers will find it relatively easy to introduce SBW interoperability into their software.

To give some idea of what is involved, here is a summary of the steps a developer would follow to adapt a particular application for use as a module in SBW:

1. Decide on the operations that will be provided to clients. These operations are the *services* provided by a module in the SBW framework.
2. Categorize each service, using as a starting point the hierarchy of services in existing SBW modules.
3. For each operation/service, determine the parameters and return value(s) that are appropriate. Express these in terms of method (function) signatures using the notation described in the SBW API documentation (Finney et al., 2002a,b,c; Hucka et al., 2002; Sauro et al., 2002, 2003).
4. Implement each method or function implied by the signatures collected from the previous step.
5. Add calls in the application's main routine to the SBW module registration methods. These provide the information needed by SBW to "hook" the application into SBW. The registration methods are summarized in Section 3.3 and fully documented in the SBW API documentation.
6. Compile the application and link it with the SBW API library.

3.3 General Overview of the SBW APIs

In general, SBW modules are stand-alone executable applications. A module may make a set of operations available programmatically to other software modules. In SBW, sets of methods or functions are grouped into one or more *services*. A service is an interface to a resource inside a module. A useful analogy is to interfaces in Java: an interface collects together a set of methods (among other things) that encapsulate access to some functionality in a program. We use the name *services* to represent interfaces in SBW because the framework is not exclusively Java-based, and because it deals with resources made available in a distributed system.

A given module may implement zero or more services. More than one module may provide similar services, so it is useful to be able to group services into categories. SBW supports *service categories* of this kind explicitly in the APIs, but it is entirely up to applications to decide how to structure and manage the categories.

For passing data around in messages, SBW supports the following common and useful data types:

- Byte: An 8-bit quantity, equivalent to a Java `byte`.
- Boolean:: A true or false value.
- Integer: A 32-bit signed number, equivalent to a Java `int`.
- Double: A double precision floating-point number in IEEE 754 standard format.
- String: A sequence of characters, equivalent to `char *` in C.
- Array: A multidimensional homogeneous collection of data of arbitrary size.
- List: A sequence of heterogeneous data elements; the elements can be of any other types, including for example other lists.

3.3.1 High-Level APIs

As discussed above, SBW's high-level API is designed to isolate many of the communications details from an application and provide a more direct programming interface to services. This is accomplished, in those languages that permit it, by constructing proxy methods or proxy objects on the client side. The proxies represent the remote services and hide some of the details involved in sending messages between modules.

As an example of how simple the API is to use in practice, the following is C++ code demonstrating how one might invoke a simple trigonometry function in a hypothetical trigonometry module:

```
SBWDouble doTrigonometry(SBWDouble x)
{
    try
    {
        // Start a new instance of the trigonometry module.
        Module module = SBW::getModuleInstance("edu.caltech.trigonometry");

        // Locate the service that we want to call in the module.
        Service service = module.findServiceByName("trig");
        Method method = service.getMethod("sin");

        // Make the call and get the result back.
        DataBlockReader resultData = method.call(DataBlockWriter() << x);

        // Extract the result data.
        SBWDouble result;
        resultData >> result;

        return result;
    }
    catch (SBWException e)
    {
        e.HandleWithDialog();
    }
    return 0;
}
```

As the example above shows, the first step in using an SBW-enabled resource involves getting a reference to the module that implements a desired service. The next step is getting a reference to a particular method within the service, and then the final step is invoking that method. A few additional details surround the handling of arguments and result values.

The facilities provided by the high-level SBW API can be summarized as follows:

1. *Service and module discovery*: These are facilities for querying SBW about the services, service categories, and modules known to the system.

2. *Service method invocation*: As mentioned above, SBW provides different ways of invoking a service method in different programming language, in an attempt to provide the most natural kind of interface for each language.

3. *Module, service and method registration*: These are facilities for defining services and their implementations within modules.

4. *Data serialization*: All data and method invocations involve passing messages between modules, which requires packing data into message streams. (In Java, it is possible to abstract these details completely and introduce proxy objects that completely encapsulate operations on remote modules. Unfortunately, this is not possible in all languages; in those cases, some aspects of data serialization must be exposed to the application.)

5. *Exception handling*: These are facilities for dealing with exceptional conditions signaled by modules.

3.3.2 Low-Level APIs

The SBW low-level API provides direct access to messages and basic SBW functionality. The operations at this level center around blocking and non-blocking remote procedure calls. The blocking version (`call`) invokes a specific method of a specific service in a specific module, handing it arguments serialized into a message data stream. The call waits until the method on the remote module returns a value. The non-blocking version (`send`) is similar, except that it does not wait for a return value.

Additional methods in the low-level API are concerned with registration, discovery and invocation, much as in the high-level API but at a more basic level.

4 Summary

While some modelers write their own numerical software using common programming languages, and others use general-purpose software such as MATLAB, Mathematica, and XPPAUT (Ermentrout, 2001), many researchers find dedicated analysis and simulation tools more effective. These packages provide efficient facilities for handling systems of differential equations generated from biochemical network models, enabling users to express problems using constructs specialized for the domain. Common applications of these tools include time-series simulation, stochastic simulation, metabolic control analysis, and parameter-fitting. Almost without exception, these different tools do not interoperate with each other. Our hope in creating the Systems Biology Workbench is that it will provide the scaffolding on which interoperability can be achieved. By providing a common framework for linking different packages together, we hope to enable a new synergy of functionality. We hope that as more tools are integrated into the framework, powerful new capabilities will emerge, such as: the combination of stochastic and differential-equation based approaches to simulating complex models; the ability to construct models using components drawn from databases and experimental devices, then apply a variety of different analysis and simulation procedures; automatic derivation of models from proteomic and genomic data; and more. The development of such systems will require major theoretical advances that will have important applications outside of biology as well.

5 Acknowledgements

The Systems Biology Workbench project was conceived at an ERATO-sponsored workshop held at the California Institute of Technology, USA, in December, 1999. Following that meeting, Hamid Bolouri organized a team of three of us (Andrew Finney, Herbert Sauro, Michael Hucka), outlined the requirements for the system, and lead the group through the production of version 1.0 of SBW. The project later grew to include additional staff: Ben Kovitz, Jo Matthews and Sarah Keating. All funding until September 2003 came from the Japan Science and Technology Corporation via the ERATO Kitano Symbiotic Systems Project.

Early in the project, Herbert Sauro implemented a prototype of SBW that served as a testbed for many concepts. The document *The Systems Biology Workbench Concept Demonstrator: Design and Implementation* (Sauro et al., 2001) describes the implementation of this initial version of SBW.

We are grateful for comments, support, advice and help from John Doyle and Hiroaki Kitano as well as fellow ERATO Kitano Symbiotic Systems Project members Bruce Shapiro, Ben Bornstein, Mark Borisuk, Mineo Morohashi, Akira Funahashi, Tau-Mu Yi and Eric Mjolsness. We also thank the many testers and users of SBW who continue to send feedback and make contributions to the project. Finally, we thank the authors of BioSpice, CellDesigner, Cellerator, CellML, COPASI, DBSolve, E-Cell, Gepasi, Jarnac, JDesigner, JigCell, MCell, NetBuilder, Promot/DIVA, StochSim, Virtual Cell, as well as the members of the `sysbio` mailing list, for additional discussions, comments, help and advice.

References

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

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

Ermentrout, B. (2001). XPPAUT 5.0. Available via the World Wide Web at `http://www.math.pitt.edu/~bard/xpp/xpp.html`.

Farley, J. (1998). *Java Distributed Computing*. O'Reilly & Associates.

Finney, A. (2003). Systems Biology Markup Language (SBML) Level 2: Structures and facilities for model definitions. Available via the World Wide Web at `http://www.sbml.org`.

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

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

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

Gilman, A. (2000). A letter to the signaling community. Alliance for Cellular Signaling, The University of Texas Southwestern Medical Center. Available via the World Wide Web at `http://afcs.swmed.edu/afcs/Letter_to_community.htm`.

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

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001a). A comparison of two alternative message-passing approaches for SBW. Available via the World Wide Web at `http://www.sbw-sbml.org/`.

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

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

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., 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., Le Novère, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., 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., and Wang, J. (2003). The Systems Biology Markup Language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531.

Lee, E. A. (1998). Overview of the Ptolemy project. ERL Technical Report UCB/ERL No. M98/71, University of California, Berkeley. Available via the World Wide Web at `http://ptolemy.eecs.berkeley.edu/publications/papers/98/Overview/`.

McLaughlin, B. (2000). *Java and XML*. O'Reilly & Associates.

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

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

OMG (2001). *CORBA*. Specification documents available via the World Wide Web at `http://www.omg.org`.

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., 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/`.

Sauro, H. M., Hucka, M., Finney, A., and Bolouri, H. (2002). Systems Biology Workbench Delphi/Kylix programmer's manual. Available via the World Wide Web at `http://www.cds.caltech.edu/erato/sbw/docs`.

Sauro, H. M., Kovitz, B., Hucka, M., Finney, A., and Bolouri, H. (2003). Systems Biology Workbench Python programmer's manual. Available via the World Wide Web at http://www.cds.caltech.edu/erato/sbw/docs.

Sauro, H. S. (2001). JDesigner: A simple biochemical network designer. Available via the World Wide Web at http://members.tripod.co.uk/sauro/biotech.htm.

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.

Seetharaman, K. (1998). The CORBA connection. *Communications of the ACM*, 41(10):34–36.

Shapiro, B. E. and Mjolsness, E. D. (2001). Developmental simulations with Cellerator. In Yi, T.-M., Hucka, M., Morohashi, M., and Kitano, H., editors, *The Proceedings of the Second International Conference on Systems Biology (ICSB2001)*. Omnipress, Inc.

Smaglik, P. (2000). For my next trick... *Nature*, 407:828–829.

Steflik, D. and Sridharan, P. (2000). *Advanced Java Networking*. Prentice-Hall.

Sun Microsystems (2001). Java Development Kit 1.3. Available via the World Wide Web at http://java.sun.com/j2se/1.3/.

Sweet, D. (2001). *KDE 2.0 Development*. Sams.

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/.

UserLand Inc. (2001). XML-RPC home page. Available via the World Wide Web at http://www.xml-rpc.com/.

Vinoski, S. (1997). CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communication*.

Winer, D. (2001). XML-RPC specification. Available via the World Wide Web at http://www.xmlrpc.com/spec/.