Systems Biology Workbench Module Programmer's Manual

Andrew Finney, Michael Hucka, Herbert Sauro, Hamid Bolouri

{afinney,mhucka,hsauro,hbolouri}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA
http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

August 9, 2002

Contents

1 Introduction

The aim of this manual is provide programming reference documentation for the SBW modules supplied with the SBW download. This manual describes the SBW services exposed to SBW client modules by the supplied modules. This manual complements the SBW programming manuals for C (Finney et al., 2002a), C++ (Finney et al., 2002b) and Java (Hucka et al., 2002). These manuals provide an overview of the SBW system that is not provided here.

The SBW CORBA gateway is documented separately in Hucka and Finney (2002).

The following sections of this document describe the SBW services to the modules supplied with the SBW installation. Many of the supplied service are categorized i.e. they share common forms. These common forms are described in one section for each category. The remainder of the services are described in a section for each module.

Table 1 shows the modules supplied with the SBW installation.

Module Name	**Module Type**	**Comments**
edu.caltech.plot	Self Managed	appropriate for plotting any series data
edu.caltech.NOM	Self Managed	parses a given SBML string, resulting structure is accessible through service
edu.caltech.NOMClipboard	Unique	parses a given string, resulting structure is accessible to all clients
<simulator>.gui	Self Managed	<simulator> is the name of any other module with a service in the category *Simulation*. This module provides a GUI which drives the associated simulation service given a SBML string.
edu.caltech.gibson	Self Managed	Stochastic simulator
edu.caltech.MatlabTranslator	Unique	Translates given SBML strings into MATLAB (MathWorks, 1998) functions

Table 1: *Modules supplied with SBW installation*

Table 2 shows the services implemented by each supplied module together with their categorization and the section in which the service is documented.

Module Name	**Service Name**	**Category**	**Section**
edu.caltech.plot	MultipleViewsOfTimeSeriesData	/	5
edu.caltech.NOM	NOM	/	4
edu.caltech.NOMClipboard	NOM	/	4
edu.caltech.NOMClipboard	NOMClipboard	*Analysis*	2
<simulator>.gui	<simulator service>	*Analysis*	2
<simulator>.gui	SimulationCallback	/	3
edu.caltech.gibson	gibson	*Simulation*	3
edu.caltech.MatlabTranslator	MatlabTranslator	/	6
edu.caltech.MatlabTranslator	SaveModelAsMatlabSimulinkFunction	*Analysis*	2
edu.caltech.MatlabTranslator	SaveModelAsMatlabODEFunction	*Analysis*	2

Table 2: *Services implemented by Modules supplied with SBW installation*

2 The Generic Analysis Interface

Services in the *Analysis* category have the following simple interface:

void doAnalysis(string SBML)

Perform some function on the given SBML model. If the function generates an independent GUI the function should return as soon as that UI has been launched otherwise the function should block until the task is complete.

The services in the *Analysis* category are:

- **NOMClipboard** This service, on the unique module `edu.caltech.NOMClipboard`, takes the given SBML Model and parses it. The resulting structure is available via the `NOM` service on the same module, see section 4. This service thus provides a mechanism for transferring models between modules.

- **Simulator GUI** For each service in the *simulation* category it is possible using a supplied jar file to create a new module that provides a GUI for the service. These modules have a service, which has the same name as the original simulation service. This derived service is registered in the *Analysis* category. These *Analysis* services simply take the given SBML model and provide a GUI for launching simulations of that model. The simulations are then provided by the original simulation service.

  The JAR file, `SimDriver.jar`, when invoked with the `-sbwregister` flag registers a new module for each service registered in the category *Simulation*. The new module has a name of the form `<simulator>.gui` where `simulator` is the name of the module with the simulation service.

  The interface between the simulation service and the simulation GUI are described in more detail in section 3.

  In summary if you develop a new simulation service, you can expose an *Analysis* service providing a GUI for your service simply by registering your module then registering `SimDriver.jar`. A GUI module for the Gibson (Gibson and Bruck, 2000) simulator is registered in this way by the SBW Windows installer. A GUI module for the Jarnac (Sauro, 2000), which is available as a separate download can also be registered in this way.

- **SaveModelAsMatlabSimulinkFunction** This service simply creates a dialog in which the user can select a directory and filename. This service then saves a MATLAB Simulink function equivalent to the supplied SBML model into a new file with the given name in the selected directory.

- **SaveModelAsMatlabODEFunction** This service simply creates a dialog in which the user can select a directory and filename. This service then saves a MATLAB ODE function equivalent to the supplied SBML model into a new file with the given name in the selected directory.

3 The Generic Simulation Interface

The form of services registered in the *Simulation* category is described in section 3.1. All modules which call this service must provide a callback service called `SimulationCallback`. The form of the `SimulationCallback` service is described in section 3.2.

The supplied module `edu.caltech.gibson` implements the service `gibson` which is registered in the *Simulation* category. Jarnac, which is available separately, also provides a service in this category.

The Simulator GUI module, described in section 2 provides a generic client GUI for services in the *Simulation* category and provides a `SimulationCallback` service.

3.1 Simulation service

Modules that implement services in the *Simulation* category are simulators. The form of these services is:

string[] optionsSupported()

This method returns an array of strings which indicate the features of the simulator. The returned array is a subset of the following strings:

- **variableTimeSteps** This string indicates that the number of time steps can't be set by the calling module.
- **performsSteadyState** This string indicates that the simulator has an implementation of the method `doSteadyStateAnalysis`.

string[] loadSBMLModel(string sbml)

Loads the given SBML model into the simulator i.e. parses the SBML. This method returns the set of variables that the simulator parsed from the model. The calling module can specify a subset of these variables, for which time series data will be returned.

void doTimeBasedSimulation(double startTime, double endTime, int noOfPoints, string[] filter)

Start a simulation on the loaded model. The simulation is performed from `startTime` to `endTime`. `filter` specifies the subset of variables for which time series data should be generated. `filter` is a subset of the variable set returned by `loadSBMLModel`. If the service does not return the option string *variableTimeStep* then `noOfPoints` specifies the number of data points in the time series data that should be generated, including data points for the start and end times. If the service does return the option string *variableTimeStep* then the simulation will generate an arbitrary number of data points from `startTime` to `endTime` or slightly beyond `endTime` and `noOfPoints` is ignored.
Data is passed back to the calling module via the `SimulationCallback` service which is described section 3.2. The simulation service will call the `onRowData` method on the `SimulationCallback` service once for each data point as soon as the data point has been computed. The order of the values passed to the callback service corresponds to the variable strings passed in the `filter` argument to `doTimeBasedSimulation`. If the simulation service does return the option string *variableTimeStep* then the simulation service will insert an additional value at the end of the array of data values which is the time of the data point.
If an error occurs the simulation service will call the `onError` method on the `SimulationCallback` service.

void stop()

Pauses the simulation started by the last call to `doTimeBasedSimulation`. The simulation can either be replaced by a new simulation by calling `doTimeBasedSimulation` or the simulation can be restarted by calling `restart`.

void restart()

Causes the simulation started by the last call to `doTimeBasedSimulation` to be restarted. Assumes that `stop` has been called since the call to `doTimeBasedSimulation`.

void doSteadyStateAnalysis(string[] filter)

This method is only present if the service returns the option string *performsSteadyState* from `optionsSupported`. This method starts the execution of steady state analysis on the parsed model. `filter` is the subset of variables for which values should be passed to the caller.
The simulation service will call the `onRowData` method once on the `SimulationCallback` service (see section 3.2) once the analysis is complete. The order of the values passed to the callback service corresponds to the variable strings passed in the `filter` argument to `doSteadyStateAnalysis`.
If an error occurs the simulation service will call the `onError` method on the `SimulationCallback` service.

3.2 `SimulationCallback` service

The form of the `SimulationCallback` service is:

void onError(string errorMessage)

This method is called if an error occurs during a simulation or analysis.

void onRowData(double[] data)

This method is called when a data point is computed by a simulator during or at the end of a simulation or analysis.

4 The Network Object Model (NOM) Service

This service is provided by the `edu.caltech.NOM` and `edu.caltech.NOMClipboard` modules where the latter is self managed type module and the former is a unique type module. In both cases the service is called `NOM` and has the exactly the same form. These services parse an SBML model and enables the caller to browse the resulting structure.

The form of these services is as follows:

void loadSBML(string sbml)

parses the given SBML model. The methods below have no effect unless this method is called first.

string getSBML()

returns the loaded SBML.

string getModelName()

returns the name of the loaded model.

double getValue(string symbol)

returns the initial value of the given symbol. The symbol can be a parameter, species or compartment. Throws an exception if symbol has no initial value in the model.

boolean exists(string symbol)

returns whether symbol exists in the loaded model. The symbol can be a parameter, species or compartment.

boolean hasValue(string symbol)

returns whether symbol has a value in the model. The symbol can be a parameter, species or compartment. Throws an exception if symbol is not in the model.

int getNumCompartments()

returns the number of compartments in the model.

int getNumReactions()

returns the number of reactions in the model.

int getNumFloatingSpecies()

returns the number of species whose concentration can vary during a simulation.

int getNumBoundarySpecies()

returns the number of species whose concentration is fixed for the duration of a simulation.

int getNumGlobalParameters()

returns the number parameters whose scope is over the whole model.

string getNthCompartmentName(int nthCompartment)

returns the name of a compartment in the loaded model. `nthCompartment` is the index of the compartment, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumCompartments`.

string getNthFloatingSpeciesName(int nthSpecies)

returns the name of a species in the loaded model, whose concentration can vary during a simulation. `nthSpecies` is the index of the species, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumFloatingSpecies`.

string getNthBoundarySpeciesName(int nthSpecies)

returns the name of a species in the loaded model, whose concentration is fixed for the duration of a simulation. `nthSpecies` is the index of the species, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumBoundarySpecies`.

string getNthFloatingSpeciesCompartmentName(int nthSpecies)

returns the name of the compartment containing a species in the loaded model. The species concentration can vary during a simulation. `nthSpecies` is the index of the species, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumFloatingSpecies`.

string getNthBoundarySpeciesCompartmentName(int nthSpecies)

returns the name of the compartment containing a species in the loaded model. The species concentration is fixed for the duration of a simulation. `nthSpecies` is the index of the species, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumBoundarySpecies`.

string getNthReactionName(int nthReaction)

returns the name of a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`.

int getNumReactants(int nthReaction)

returns the number of reactants in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`.

int getNumProducts(int nthReaction)

returns the number of products in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`.

string getNthReactantName(int nthReaction, int nthReactant)

returns the name of the species that is a reactant in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthReactant` is the index of the reactant, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumReactants` for reaction `k`.

string getNthProductName(int nthReaction, int nthProduct)

returns the name of the species that is a product in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthProduct` is the index of the product, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumProducts` for reaction `k`.

string getKineticLaw(int nthReaction)

returns the kinetic law or rate equation of a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`.

int getNthReactantStoichiometry(int nthReaction, int nthReactant)

returns the stoichiometry of a product in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthReactant` is the index of the reactant, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumProducts` for reaction `k`.

int getNthProductStoichiometry(int nthReaction, int nthProduct)

returns the stoichiometry of a reactant in a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthProduct` is the index of the product, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumReactants` for reaction `k`.

int getNumParameters(int nthReaction)

returns the number of parameters that are local to a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`.

string getNthParameterName(int nthReaction, int nthLocalParameter)

returns the name of a parameter that is local to a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthLocalParameter` is the index of the local parameter, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumParameters` for reaction `k`.

double getNthParameterValue(int nthReaction, int nthLocalParameter)

returns the value of a parameter that is local to a reaction. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthLocalParameter` is the index of the local parameter, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumParameters` for reaction `k`.

boolean getNthParameterHasValue(int nthReaction, int nthLocalParameter)

returns whether a parameter that is local to a reaction, has a value. `nthReaction` is the index of the reaction, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumReactions`. `nthLocalParameter` is the index of the local parameter, and should be between 0 and `j - 1` inclusive, where `j` is the result of `getNumParameters` for reaction `k`.

string getNthGlobalParameterName(int nthGlobalParameter)

returns the name of a parameter which has scope over the whole of the loaded model. `nthGlobalParameter` is the index of the parameter, and should be between 0 and `k - 1` inclusive, where `k` is the result of `getNumGlobalParameters`.

string[] getBuiltinFunctionInfo(string functionName)

returns a sequence of strings describing a built-in SBML function. `functionName` is the name of a built-in function. The result consists of a descriptive string for the function, followed by the parameters to the function which is finally followed by the expression implemented by the function. This method returns an empty array if the given name is not of a built-in SBML function.

string[] getBuiltinFunctions()

returns the set of built-in SBML function names.

5 The Plot Service

The service `MultipleViewsOfTimeSeriesData` is provided by the module `edu.caltech.plot`. Despite its name this service is capable of plotting any series data i.e. matrix data where you wish to compare values of different columns over a set of rows. This module is designed so that data can be plotted while it is being generated. Any number of views of the data can be created as the data is arriving. Views are updated as data arrives.

`edu.caltech.plot` is a self managed type module. Each instance of `edu.caltech.plot` stores one set of data but allows several different views of that set.

The service `MultipleViewsOfTimeSeriesData` has the following form:

void addDataPoints(double commonAxis, double[] vector)

pass a vector of values for plotting. `vector` is a set of data values. `commonAxis` is a value nominally indicating the position of all the values in `vector` however this value is arbitrary: for example it can always be 0. Typically `commonAxis` is the time at which the values in `vector` occurred.
It is assumed that every call to `addDataPoints` is consistent. `commonAxis` and each element at the same position of `vector` should consistently represent the same entity.
Calls to `addDataPoints` can occur at any time and are independent of the other calls on this service.

void deleteViews()

destroys all the current views of the data set.

The following methods should be called in the sequence that is given here to create one view of the data. Any number of views can be created.

void setTitle(string)

set the title of the new view. This method is optional.

void newView()

create the new view.

void setCurrentVariableNames(string xAxisName, string[] variableNames)

supply names for the variables. `xAxisName` is the legend to be given to the x axis. `variableNames` is the sequence of variable names corresponding to the data vectors passed in calls to `addDataPoints`. `variableNames` is used to create the y axis legend.

void setCurrentPlottedVariables(int xVariable, boolean[] yVariableFilter)

identify what data should be plotted. `xVariable` is the index value of the variable to be plotted on the x axis. `xVariable` indexes the data vectors passed in calls to `addDataPoints`. If `xVariable` is -1 then the `commonAxis` values are used on the x axis. The array `yVariableFilter` corresponds to the data vectors passed in calls to `addDataPoints`. Each entry indicates whether the variable should be plotted in the view or not.

void startView()

display the view.

6 The MATLAB Translator Service

The module `edu.caltech.MatlabTranslator` provides the service `MatlabTranslator` in addition to the *Analysis* category services that it provides (see section 2). The Service `MatlabTranslator` has the form:

string translate(string sbml)

returns a MATLAB Simulink function corresponding to the given SBML model.

string translate(string sbml, string modelName)

returns a MATLAB Simulink function corresponding to the given SBML model. The name of the returned Simulink function is `modelName`.

string translate(string sbml, boolean ODEFunction)

returns a MATLAB function corresponding to the given SBML model. If `ODEFunction` is true the returned function is an ODE function, otherwise the returned function is a Simulink function.

string translate(string sbml, string modelName, boolean ODEFunction)

returns a MATLAB function corresponding to the given SBML model. If `ODEFunction` is true the returned function is an ODE function, otherwise the returned function is a Simulink function. The name of the returned function is `modelName`.

References

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.cds.caltech.edu/erato/sbw/docs/`.

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.cds.caltech.edu/erato/sbw/docs/`.

Gibson, M. and Bruck, J. (2000). Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889.

Hucka, M. and Finney, A. (2002). The SBW-CORBA Gateway.

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.cds.caltech.edu/erato/sbw/docs`.

MathWorks, T. (1998). *Using* MATLAB. MATLAB: The Language of Technical Computing. The MathWorks, Inc., Natik, MA.

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.