# BioSim: An Integrated Simulation of an Advanced Life Support System for Intelligent Control Research
# Users Manual (draft)

Metrica Inc. and S&K Technologies
NASA Johnson Space Center/ER2
Houston TX 77058
http://www.traclabs.com/biosim

October 15, 2004

# Contents

# List of Figures

# Chapter 1

# Introduction

Advanced life support systems have multiple interacting subsystems, which makes control a particularly challenging task. The simulation described in this document provides a testbed for integrated control research. There have been other integrated life support simulations (e.g., [**?**]) and we have learned from those efforts. This simulation is designed exclusively for integrated controls research, which imposes different requirements. For example, the simulation is accessed through sensors and actuators, just as a real system would be. Noise and uncertainty are built in and controllable. Malfunctions and failures of subsystems are modeled and manifest themselves through anomalous readings in the sensors. Crew members are taskable and their tasks have purpose and meaning in the simulation. In essence, the simulation is a replacement for the Advanced Life Support (ALS) hardware and crew, allowing for testing of control approaches in advance of any integrated test. We want the simulation to be used to develop and evaluate integrated control techniques. There are still many open research questions with respect to controlling advanced life support systems. For example is a distributed or hierarchical approach better? What role does machine learning play in control? How can symbolic, qualitative control approaches be integrated with continuous, quantitative approaches? How can we evaluate different control philosophies? We hope that the controls community can use this simulation to begin to build systems which will provide us with answers to these kinds of questions.

The simulation is written in Java and is accessed using the Common Object Request Broker Architecture (CORBA). The distributed nature of the simulation allows for multiple instances to be run in parallel, making it ideal for testing advanced control concepts such as genetic algorithms or reinforcement learning, which require multiple trials. See [**?**] or [**?**] for a background on advanced life support control issues and previous work. This document begins with a brief introduction to advanced life support systems and provides details on installing, running and interacting with the simulation. The simulation is available from:

```
http://www.traclabs.com/biosim
```

# Chapter 2

# Background on advanced life support

An advanced life support system is one in which many resources are reused or regenerated. Such systems will be necessary for long duration human missions, such as those to Mars, where resupply opportunities are limited. Typically they have thin margins for mass, power and buffers, which requires optimization and tight control. Also, advanced life support systems consist of many interconnected subsystems all of which interact in both predictable and unpredictable ways (see Figure 2.1). Autonomous control of these systems is desirable. This section gives enough background so that users of the simulation can understand the major components of an advanced life support system. For detailed documentation on advanced life support systems see [**?**] or go to:

`http://advlifesupport.jsc.nasa.gov`

## 2.1 Modules

An advanced life support system consists of many interacting subsystems. While each is self-contained, they rely on each other for various resources. The simulation is built from a set of modules that each produce and consume different resources.

### 2.1.1 Environment

An environment contains air that is consumed by either people or crops. Air contains a mixture of gases – in our simulation these gases are oxygen (O2), carbon dioxide (CO2), nitrogen (N), water vapor (H20), and other gasses (trace). The initial composition of the gases is set by the simulation initialization file (see Section 7.2). As the simulation runs modules may consume air from the simulation and replace it with air of a different composition. Thus, the composition of gases and pressure in the air change over time and can be measured by environment sensors (Section 8.2.3). As with all modules, there can be multiple environments. For example, it is common for crew members and crops to have different air compositions and that is the default in this simulation.

### 2.1.2 Crew

The crew module is implemented using models described in [**?**]. The number, gender, age and weight of the crew are settable as input parameters – in the default configuration there are four crew members, two male and two female. The crew cycles through a set of activities (sleep, maintenance, recreation, etc.). As they do so they consume O2, food and water and produce CO2, dirty water and solid waste. The amount of resources consumed and produced varies according to crew member attributes and their activities. The crew's activities can be adjusted by passing a new crew schedule to the crew module. A default schedule can also be used. The crew module is connected to a crew environment that contains an atmosphere that they breathe. The initial size and gas composition (percentages of O2, CO2, H2O and inert gases) are input

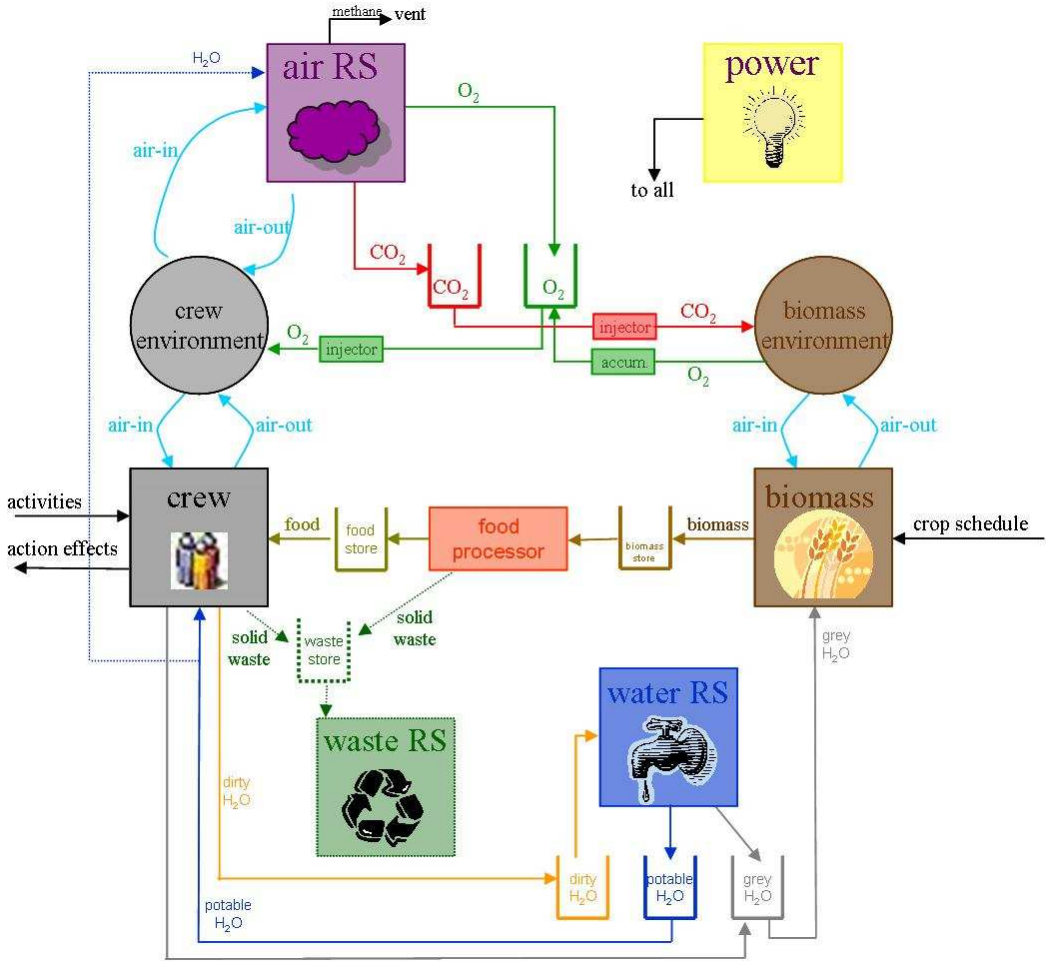


Figure 2.1: The various modules that comprise the BIOSim integrated simulation.

parameters and the default is $1.54893x10^6 liters$ (from [**?**]) with an atmosphere equivalent to sea level air. As the simulation progresses the mixture of gases in the atmosphere changes.

### 2.1.3 Water

Consumes:Power, Grey Water, Dirty Water Produces: Potable Water

The water recovery module consumes dirty water, grey water(i.e., water that can be used for washing but not drinking), and power and produces potable water. The water recovery module consists of four subsystems that process the water. The biological water processing (BWP) subsystem removes organic compounds. Then the water passes to a reverse osmosis (RO) subsystem, which makes 85% of the water passing through it grey. The 15% of water remaining from the RO (called brine) is passed to the air evaporation subsystem (AES), which recovers the rest. These two streams of grey water (from the RO and the AES) are passed through a post-processing subsystem (PPS) to create potable water. An external controller can turn on or off various subsystems. For example, all water can pass through the AES at a higher power cost. We based our water recovery module on a recently completed test at NASA Johnson Space Center [**?**].

### 2.1.4 Air

Consumes:Power, H2, Air, CO2, Potable Water Produces: Air (with less CO2), O2, H2, CO2, Potable Water

The air component takes in exhalant CO2 and produces O2 as long as there is sufficient power being provided to the system. This component is modeled on various Air Revitalization System (ARS) work at NASA JSC [**?**]. There are three interacting air subsystems: the Variable Configuration Carbon Dioxide Removal (VCCR) System in which CO2 is removed from the air stream; the Carbon Dioxide Reduction System (CRS), which also removes CO2 from the air stream using a different process and producing different gases than the VCCR; and the Oxygen Generation System (OGS) in which O2 is added to the air stream by breaking water down into hydrogen and oxygen. It is important to note that both the removal of CO2 and the addition of O2 are required for human survival. It is also important to note that the biomass component (next subsection) also removes CO2 and adds O2.

### 2.1.5 Biomass

Consumes:Power, Potable Water, Grey Water, Air Produces: Air (with more CO2), Biomass, Dirty Water, CO2, Potable Water

The biomass component is where crops are grown. It produces both biomass, which can be turned into food, and O2 and consumes water, power (light) and CO2. This component is based on models given in [**?**]. The system is modeled as shelves that contain plants, lights and water. Shelves are planted and harvested and there is growth cycle for each shelf. Currently, ten crops are modeled and can be planted in any ratio.

### 2.1.6 Food processing

Consumes:Power, Biomass Produces: Food

Before biomass can be consumed by the crew it must be converted to food. The food processing component takes biomass, power and crew time and produces food and solid waste. The crew needs to be involved in this process as it is labor intensive. See Section 2.1.2 for more information on scheduling crew activities.

### 2.1.7 Waste

Consumes:Power, Dry Waste, O2 Produces: CO2

The waste component consumes power, O2 and solid waste and produces CO2. It is modeled on an incinerator used in the Phase III test in 1997 [**?**]. Incineration can be scheduled.

### 2.1.8 Thermal control

Consumes: ? Produces: ?

The thermal control component regulates the air temperature in the habitat. This component is not implemented for this simulation and it is assumed that external controllers are maintaining chamber temperature.

### 2.1.9 Power

Consumes: Produces: Power

The power component supplies power to all of the other components that need it. There are two choices for power in the simulation. The first is nuclear power, which supplies a fixed amount throughout the lifetime of the simulation. The second is solar power, which supplies a varying amount (day/night cycle) of power to each component.

### 2.1.10 Accumulators

Consumes: All Produces: All

The accumulator component can take a resource from any store or environment and place it into another environment or store. It is functionally equivalent to an injector.

### 2.1.11 Injectors

Consumes: All Produces: All

The injector component can take a resource from any store or environment and place it into another environment or store. It is functionally equivalent to an accumulator.

# Chapter 3

# Simulation Properties

The simulation implements the modules outlined in the previous section. BioSim does *not* simulate at the level of valves, pumps, switches, etc. Instead, modules are implemented in a producer/consumer relationship, which is described in the next subsection. The simulation also provides additional functionality to help test and debug control programs.

## 3.1 Producer/Consumer Model

No component in BioSim directly interacts with other components. Instead, each component has a rigid set of resources that it consumes and produces. The resources are taken from stores/environments and put into stores/environments. The resources for BioSim currently consist of power, potable water, grey water, dirty water, air, H2, nitrogen, O2, CO2, biomass, food, and dry waste. At each simulation tick, each module takes resource from it's store (it's consumables), and puts resources into stores (it's products). Stores always report thier values (level and capacity) from the last tick and report new values after every component in the simulation has been ticked. Resource conflicts can occur when two components ask for a limited resource, i.e. two WaterRS's both need 20 liters of dirty water and there is only 20 liters in the dirty water tank. This is currently resolved on a winner-take-all basis that is for all practical purposes randomized.

## 3.2 Success criteria

In order to compare different control approaches there should be objective criteria for a successful advanced life support system mission. Several possibilities exist already in the simulation. For example, the length of the mission before consumables are gone is a success criterion. As is minimizing the starting levels of stores or minimizing the sizes of intermediary buffers. However, many of these fail to get at the true success of a mission, the productivity of the crew in performing their science objectives. Thus, the simulation includes an artificial productivity measure. As the crew cycles through their activities (see Section 3.5) the amount of time they spend doing "mission" tasks is accumulated. This number is also multiplied by a factor that takes into account the amount of sleep, exercise, water, food and oxygen they are getting to approximate crew effectivness at performing the task (a happy crew is a productive crew!). So each tick, for each crew member the productivity is given by the following equation:

[need to put the equation here] [some relevant code, derive equation later]

```
private static final float WATER_TILL_DEAD = 8.1f;
private static final float WATER_RECOVERY_RATE=0.01f;
private static final float OXYGEN_TILL_DEAD = 3f;
private static final float OXYGEN_RECOVERY_RATE=0.01f;
private static final float CALORIE_TILL_DEAD = 180000f;
private static final float CALORIE_RECOVERY_RATE=0.0001f;
private static final float DANGEROUS_CO2_RATION = 0.06f;
private static final float CO2_TILL_DEAD = 10f;
```

```
private static final float CO2_RECOVERY_RATE=0.001f;
private static final float LEISURE_TILL_BURNOUT = 168f;
private static final float LEISURE_RECOVERY_RATE=90f;
private static final float AWAKE_TILL_EXHAUSTION = 120f;
private static final float SLEEP_RECOVERY_RATE=12f;
consumedWaterBuffer = new SimpleBuffer(WATER_TILL_DEAD, WATER_TILL_DEAD);
consumedOxygenBuffer = new SimpleBuffer(OXYGEN_TILL_DEAD, OXYGEN_TILL_DEAD);
consumedCaloriesBuffer = new SimpleBuffer(CALORIE_TILL_DEAD, CALORIE_TILL_DEAD);
consumedCO2Buffer = new SimpleBuffer(CO2_TILL_DEAD, CO2_TILL_DEAD);
sleepBuffer = new SimpleBuffer(AWAKE_TILL_EXHAUSTION, AWAKE_TILL_EXHAUSTION);
leisureBuffer = new SimpleBuffer(LEISURE_TILL_BURNOUT, LEISURE_TILL_BURNOUT);
float caloriePercentFull = sigmoidLikeProbability(consumedCaloriesBuffer.getLevel() / consumedCaloriesBu
float waterPercentFull = sigmoidLikeProbability(consumedWaterBuffer.getLevel() / consumedWaterBuffer.get
float oxygenPercentFull = sigmoidLikeProbability(consumedOxygenBuffer.getLevel() / consumedOxygenBuffer
float CO2PercentFull = sigmoidLikeProbability(consumedCO2Buffer.getLevel() / consumedCO2Buffer.getCapaci
float sleepPercentFull = sleepBuffer.getLevel() / sleepBuffer.getCapacity();
float leisurePercentFull = leisureBuffer.getLevel() / leisureBuffer.getCapacity();
float averagePercentFull = (caloriePercentFull + waterPercentFull + oxygenPercentFull + CO2PercentFull +
myMissionProductivity += myCrewGroup.randomFilter(averagePercentFull);
```

## 3.3 Stochastic processes

Any sufficiently complex process, especially one with biological components, will not be deterministic. That is, given the same starting conditions and inputs it will not produce the exact same outputs each time it is run. A stochastic process is one in which chance or probability affect the outcome. This simulation offers both deterministic and stochastic operations. In the former case, running the simulation twice with the same inputs and initial conditions will produce the same results. This may be useful for quantitatively comparing different control approaches against each other. In the latter case, the user can control the amount of variability in the simulation (see Section 7.4). This is modeled using a Gaussian distribution. The deviation is determined by the stochastic intensity; a higher intensity will yield a higher deviation. The filter is then appropriately applied to certain variable of the model. This can be used to test a control systems ability to deal with stochastic processes.

## 3.4 Malfunctions

The simulation also has the ability to accept malfunction requests (see Section 7.5). These requests will change the operating regime of the simulation. For example, by causing a crew member to be sick, power supplies to drop, water to leak, plants to die, etc. Different modules will have different sensitivities, for example once plants die they cannot recover, but if the water module is damaged and then repaired it will operate normally. Of course, if levels of CO2, water or food reach hard-coded critical levels the crew will abandon the mission and return home.

## 3.5 Crew activity scheduling

Crew members in an advanced life support system do not just consume and produce – they do activities. These may be science, maintenance, exercise, sleep, etc. The different activities go for a specified length and for a specified intensity. The intensity is tied to how many resources the crew member will consume performing the activity (e.g., sleep takes less 02 to perform than exercise). The following shows the nominal daily routine for a crew member:

1. Sleep for 8 hours
2. Hygiene for 1 hour

3. Exercise for 1 hour
4. Eating for 1 hour
5. Mission tasks for 8 hours
6. Health for 1 hour
7. Maintenance for 1 hour
8. Leisure for 2 hours

This schedule can be interrupted by malfunctions (a crew member's activity changed to repair), sickness, or even mission end (if the crew member hasn't recieved proper resources).

## 3.6 Mass balance

Mass balanced means resources consumed by components are returned in totality (but usually in a different form) to stores, no mass is lost. This is important for closed loop simulations as it verfies the models (under perfect conditions) are keeping with the conversation of energy and mass. Our simulation has been mass balanced to a reasonable degree. There is a slight loss in mass from the crew (they don't gain weight from eating or produce heat) and from the plants (who don't produce heat). We plan to address these issues in the future, but the simulation is currently mass balanced enough to provide a very decent closed loop simulation.

# Chapter 4

# Simulation implementation

All of the simulation components are written in Java to make the simulation portable. Each component can be run as it's own CORBA server so that they can be distributed across computers for increased performance, or the entire simulation can be run on one server. We have tested the simulation on Windows, Unix and Macintosh computers. A user interface allows for insight into the simulation which provides real time graphing and more. There is also a logging facility that will automatically save all simulation values to a file (xml) or a database (in development). We have written Java and C++ interfaces to the simulation, though any language that supports CORBA should be able to access the simulation.

```
(BioSim) <-> Server ORB <-> Client ORB <-> (Client)
```

I think we should provide a link to a CORBA tutorial, but assume knowledge of basics of CORBA. Explain object oriented methodology and how it relates to BioSim. Would a figure be good here showing the relationship between servers, clients, CORBA, etc.?

# Chapter 5

# Installing the simulation

The simulation is written in Java and can run on most popular operating systems. We provide both an executable for Windows and source code for any operating system. This chapter deals with installing the code onto your computer.

## 5.1 Installing an executable

An executable that will run under most Windows operating systems is available. You can download the Windows executable from: `www.traclabs.com/biosim`.

At that site, go to Download and then download the Windows installer. Remember where you put it. Double-click on the Setup.exe icon, accept the agreement, choose the default installation, install it where you desire and then click "Finish". BIOSim will start automatically. NOTE: If you do not have Java installed on your PC then the installer will direct you to the WWW page to download it (free). Please e-mail Scott Bell (scott@traclabs.com) if you have trouble with installing the simulation.

## 5.2 Installing source code

Please read the README.txt file in the doc directory of your BIOSIM distribution.

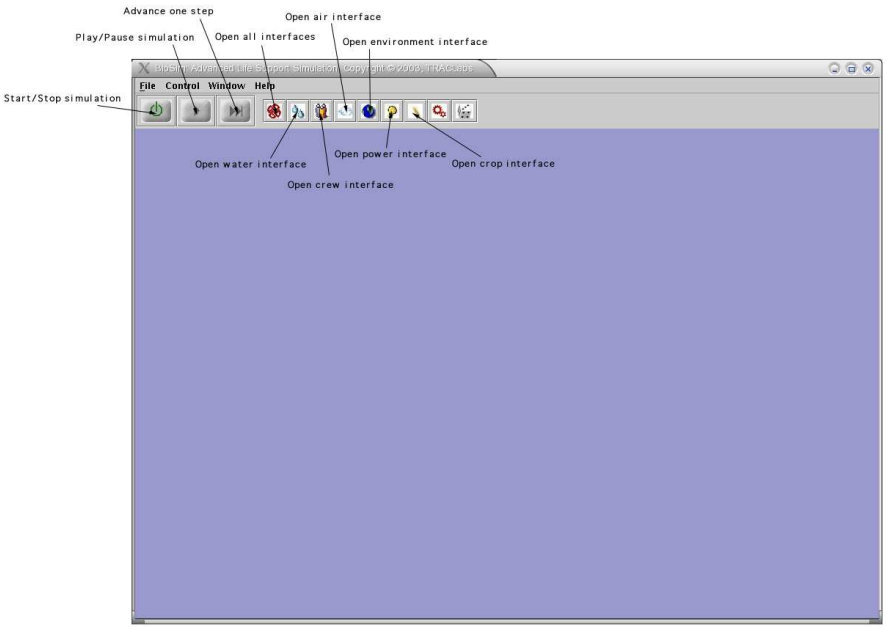


Figure 5.1: BioSims graphical user interface.

# Chapter 6

# Running the simulation

Doubleclicking on the "BioSim" icon in Windows will start the simulation. In Linux, run the `run-biosim.sh` shell script in the `/bin` directory. When BioSim starts you'll see a blank interface as shown in Figure 5.1. Holding your mouse over the buttons on the top reveals what they do. As noted in the figure, the first three buttons control the simulation. The next series of buttons will display various subsystem views. Some of the views, like the crew, do not contain anything until the simulation is started. It's often easiest to open up the main subsystem views before starting the simulation.

The open interface is shown in Figure 6.1. The view can be changed between text, chart and schematic by clicking on the tabs. The simulation will run until resources (air, water, food, etc.) drop below critical levels. Then the simulation will stop. Pressing the start button will restart the simulation from the beginning. If you want to examine the interfaces closely press the "Pause" button or go through the simulation one step at a time.

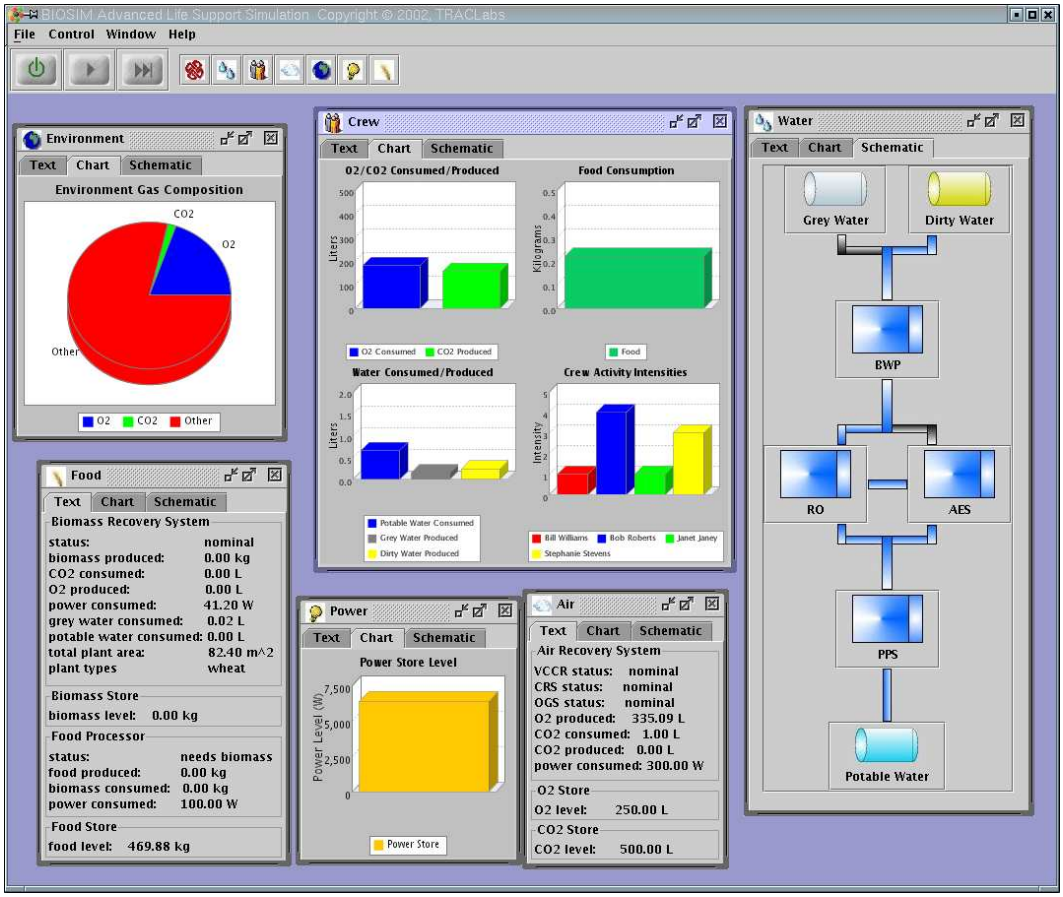


Figure 6.1: A full view of BioSims graphical user interface

# Chapter 7

# Configuring the simulation

The previous chapter discussed installing and running the default BioSim. This section will discuss configuring BioSim to address specific user interests. This includes changing the underlying producer/consumer relationship, changing the stochastic nature of BioSim and changing the logging of data. We start with some basic CORBA infrastructure requirements.

## 7.1 CORBA, Orbs and namesevers

BioSim is a server. Client programs connect to BioSim via CORBA – the Common Object Request Broker Architecture. In order to connect to the BioSim server will you will need to install an Object Request Broker (ORB) on your machine. We recommend the JacORB for Java (http://www.jacorb.org) and the ACEOrb for C++ (http://www.cs.wustl.edu/ schmidt/TAO.html), although any CORBA-compliant ORB will work. When the BioSim server is started it registers with a nameserver, which tells client programs where to find it. The ORBs we recommend include a name server. This manual will not cover the details of CORBA, we refer you to any book on CORBA for your language or look at the sample programs in Appendix A and B. We also assume that you have already installed one of the ORBs.

## 7.2 Initial conditions

BioSim is almost infinitely reconfigurable – from the number of crew members to the size and number of the different modules and should be able to provide scenarios ranging from transit vehicles to Mars colonies. The start-up configuration of BioSim is controlled by an eXtensible Markup Language (XML) file that is read in during BioSim initialization. If you want to change BioSim's configuration you need to change the XML file. The default XML file is located in:

```
BIOSIM/resources/biosim/server/framework/DefaultInit.xml
```

where `BIOSIM` is the directory where you installed BioSim.

You can copy this file, change it's name, modify it and then give its full path name as a parameter to BioSim on startup. The flag when you start BioSim is `-xml` followed by the full path name of the XML file. If you do not wish to change the initial configuration of BioSim you can skip the rest of this section.

### 7.2.1 XML file format

A snippet from the default initialization XML file is shown below:

```
<SimBioModules>
  <crew>
    <CrewGroup name="CrewGroup">
      <potableWaterConsumer maxFlowRates="100" desiredFlowRates="100" inputs="PotableWaterStore"/>
```

```
    <airConsumer maxFlowRates="0" desiredFlowRates="0" inputs="CrewEnvironment"/>
    <foodConsumer maxFlowRates="100'' desiredFlowRates="100" inputs="FoodStore"/>
    <dirtyWaterProducer maxFlowRates="100" desiredFlowRates="100" outputs="DirtyWaterStore"/>
    <greyWaterProducer maxFlowRates="100" desiredFlowRates="100" outputs="GreyWaterStore"/>
    <airProducer maxFlowRates="0" desiredFlowRates="0" outputs="CrewEnvironment"/>
     .
     .
     .
    <crewPerson name="Bob Roberts" age="43" weight="77" sex="MALE">
      <schedule>
        <activity name="sleep" length="8" intensity="1"/>
        <activity name="hygiene" length="1" intensity="2"/>
        <activity name="exercise" length="1" intensity="5"/>
        <activity name="eating" length="1" intensity="2"/>
        <activity name="mission" length="9" intensity="3"/>
        <activity name="health" length="1" intensity="2"/>
        <activity name="maintenance" length="1" intensity="2"/>
        <activity name="leisure" length="2" intensity="2"/>
      </schedule>
    </crewPerson>
     .
     .
     .
  </crew>
```

This file implements the producer/consumer relationship at the heart of the simulation. Each module (only the crew module is shown above) has a name and a list of resources that it consumes and produces along with max and desired flow rates and what source(s) produce(s) that resource (the `inputs`) or from where the resource is drawn (`outputs`) More than one source or sink is allowed, but the module tries to draw as much as it can from the first. In the above snippet some flow rates are 0 because the user has no control over them (e.g., crew air consumption is based on crew physiology and can't be directly controlled). As shown above, you can have as many (or as few) crew members as you want of various ages, weights and sexes. You can give a default schedule for their day (which can be changed via a CORBA API call during simulation runs).

The rest of the XML file specification proceeds in a similar way. The best way to change the configuration is to start with the default file and edit it to reflect your requirements. The default configuration is based on BIO-Plex [**?**], a NASA JSC ground testbed for advanced life support. As such, if all you want is a "typical" advanced life support configuration, then just use the default XML file.

### 7.2.2 Initializing BioSim

Talk about alternative initialization via an API. Again, capacities, levels, logging, etc. can be set from here.

## 7.3 Enabling stochastic processes

BioSim provides the ability to change the stochastic nature of the underlying modules via API calls or the XML initialization file. If you want a purely deterministic simulation you can do that. If you want a very unpredictable simulation you can do that as well. This section provides an overview of the stochastic controls available in BioSim. The type `StochasticIntensity` includes the following values:

- `HIGH_STOCH`
- `MEDIUM_STOCH`
- `LOW_STOCH`

- NONE_STOCH

Each of these controls the width of the Gaussian function that provides the variations in output. Setting the stochastic intensity to NONE_STOCH means that the simulation is deterministic. All noise is Gaussian. The outputs of a module as determined by the module's underlying equations are run through a Gaussian filter before being output by the simulation. The Gaussian filter's parameters are controlled by the stochastic intensity, namely the higher the stochastic intensity, the higher the deviation.

### 7.3.1 Setting the stochastic intensity via the XML configuration file

To change the stochastic intensity of a module in XML, simply add setStochasticIntensity as an attribute to the module (if one isn't listed, it defaults to NONE_STOCH). For example:

```
<AirRS name="AirRS" setStochasticIntensity="HIGH_STOCH">
```

The stochastic level for the entire simulation (which is equivalent to setting each module's stochastic intensity) can by adding setStochasticIntensity to the Globals section at the head of the XML document. For example:

```
<Globals crewsToWatch="CrewGroup" setStochasticIntensity="LOW_STOCH">
```

### 7.3.2 Setting the stochastic intensity via the API

Each module of BioSim (i.e., the `SimBioModule` class) has the following methods:

- `getStochasticIntensity()`: returns type `StochasticIntensity`, which is the current stochastic level of the module.
- `setStochasticIntensity(StochasticIntensity pIntensity)`: Sets the current stochastic intensity of the module to `pIntensity`.

The stochastic level for the entire simulation (which is equivalent to setting each module's stochastic intensity) can be set via the `BioDriver` class using the `setStochasticIntensity` method with the same parameter (`(StochasticIntensity`)

## 7.4 Logging

BioSim includes a facility for logging all data produced by the simulation. This facility can be used to debug your controller or to create displays. The data is saved as an XML file in `$BIOSIM_HOME/biosim-log.xml` This section discussed controlling data logging in BioSim.

### 7.4.1 Setting logging via the XML configuration file

Logging in BioSim is done through log4j, a popular logging utility for Java. By default, BioSim logs to the console at the "INFO" level. With log4j, output can be outputted to files, consoles, databases, sockets, etc. Individual components of the simulation can have their logging turned on. For example, to turn on debugging of air sensors, the following change must be added to the Globals section of the BioSim XML configuration file:

```
<log4jProperty name="log4j.logger.com.traclabs.biosim.server.sensor.air" value="DEBUG"/>
```

You can also globally enable logging for the senors by adding:

```
<log4jProperty name="log4j.logger.com.traclabs.biosim.server.sensor" value="DEBUG"/>
```

See log4j's documentation for more sophisticated logging options here:

`http://logging.apache.org/log4j`

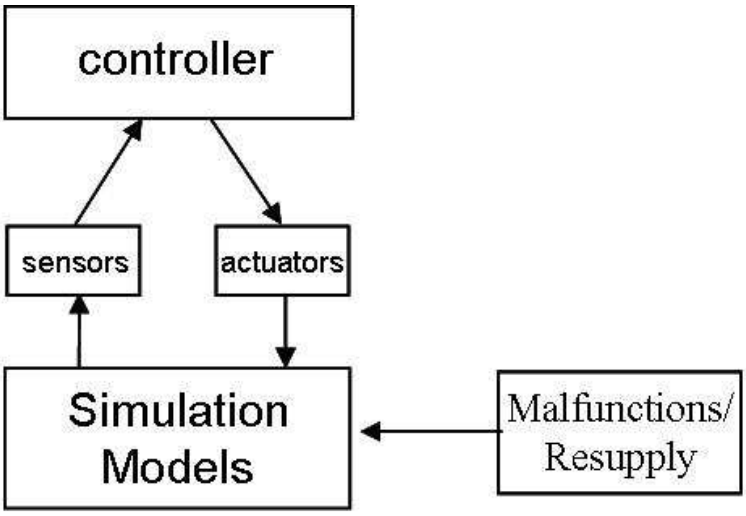


Figure 7.1: Interaction with the simulation.

### 7.4.2 Setting logging via the API

Each module of BioSim (i.e., the `SimBioModule` class) has the following methods:

- `isLogging()`: returns type `boolean`, whether the module is logging.
- `setLogging(boolean pLogging)`: Sets whether the module should log.

# Chapter 8

# Controlling the simulation

The previous two chapters showed you how to run BioSim, monitor its internal state via a graphical user interface and configure it to your specifications. While interesting, this is not useful for control system research. This chapter will show you how to connect an external control program to BioSim and change its underlying operation. Note that this chapter will not present an exhaustive list of all of the objects and methods in BioSim, but only the most important for new users. Complete documentation of all objects and method calls is available at:

`http://www.traclabs.com/biosim/doc/api/index.html`

Figure 7.1 shows the basic control strategy for the simulation. The simulation models represent the underlying equations that govern the producer/consumer relationships (see Section 3.1). Sensors and actuators are provided to access the underlying simulations. The user is responsible for writing the controller, which uses the sensors and actuators to control the simulation. The user can also inject malfunctions into the underlying simulation models and resupply consumables.

## 8.1 Controlling simulation runs

The root class of the simulation is the `BioDriver` class, which contains the methods used to control the simulation. The basic control unit is a **tick**, which advances the simulation exactly one hour. After creating your own BioDriver object the following methods are available:

1. `void advanceOneTick()`: Moves all modules of the simulation ahead exactly one hour.
2. `void endSimulation()`: Ends the simulation entirely.
3. `boolean isDone()`: Returns true if the simulation has met an end condition and has stopped.
4. `void reset()` : Returns the simulation to its initial state.

Here's an example of using one of these methods after a BioDriver class has been created (called `myBioDriver`) in a Java program:

```
myBioDriver.advanceOneTick();
```

There are a number of other methods available in the BioDriver class that are less commonly used than those listed above. These are available via the JavaDoc facility. After you have installed your source code you can make documentation via the `make-docs.sh` (for Linux) or `make-docs` (for Windows) executables in the `bin` directory. The BioDriverImpl class is under the biosim.server.framework package.

## 8.2 Accessing sensors and actuators

### 8.2.1 Generic sensors

Sensors are objects that connect to the underlying simulation. Each module has its own set of sensor objects, but they all inherit the same method calls. These method calls are:

- `getMax:` Returns the maximum allowed value of the sensor
- `getMin:` Returns the minimum allowed value of the sensor
- `getValue`: Returns the current setting of the sensor

#### Additional sensor methods

Several other generic methods related to sensor noise, malfunctions and logging are available:

In addition, there will be some sensor-specific methods, which we will discuss with the appropriate sensor.

### 8.2.2 Generic actuators

Actuators are objects that connect to the underlying simulation. Each module has its own set of actuator objects, but they all inherit the same method calls. These method calls are:

- `getMax`: Returns the maximum allowed value of the actuator
- `getMin`: Returns the minimum allowed value of the actuator
- `getValue`: Returns the current value of the actuator
- `setValue`: Sets the actuator

#### Additional actuator methods

Several other generic methods related to actuator noise, malfunctions and logging are available:

In addition, there will be some actuator-specific methods, which we will discuss with the appropriate actuator.

### 8.2.3 Environments

Environments contain the air that crew and crops use. As there can be multiple environments in the simulation, there can be multiple sensors tied to each environment.

#### Sensors

The following sensors are available in the environment:

1. `AirInFlowRateSensor`: Flow rate in moles/tick of air into the environment
2. `AirOutFlowRateSensor`: Flow rate in moles/tick of air out of the environment
3. `CO2AirConcentrationSensor`: Percentage of CO2 in the environment.
4. `CO2AirStoreInFlowRateSensor`: Flow rate in liters/tick of CO2 from a store (paired with CO2AirEnvironmentInFlowRateSensor).
5. `CO2AirEnvironmentInFlowRateSensor`: Flow rate in liters/tick of CO2 into the environment (paired with CO2AirStoreInFlowRateSensor)
6. `CO2AirPressureSensor`: Pressure in kPA of CO2 in the environment

7. `CO2AirStoreOutFlowRateSensor`: Flow rate in liters/tick of CO2 from a store. (paired with CO2AirEnvironmentOutFlowRateSensor)
8. `CO2AirEnvironmentOutFlowRateSensor`: Flow rate in liters/tick of CO2 out of the environment (paired with CO2AirStoreOutFlowRateSensor)
9. `NitrogenAirConcentrationSensor`: Percentage of Nitrogen in the environment
10. `NitrogenAirEnvironmentInFlowRateSensor`: Flow rate in liters/tick of nitrogen into the environment (paired with NitrogenAirStoreInFlowRateSensor)
11. `NitrogenAirStoreInFlowRateSensor`: Flow rate in liters/tick of nitrogen into a store (paired with NitrogenAirEnvironmentInFlowRateSensor)
12. `NitrogenAirEnvironmentOutFlowRateSensor`: Flow rate in liters/tick of nitrogen out of an environment (paired with NitrogenAirStoreOutFlowRateSensor)
13. `NitrogenAirStoreOutFlowRateSensor`: Flow rate in liters/tick of nitrogen out of a store (paired with NitrogenAirEnvironmentOutFlowRateSensor)
14. `NitrogenAirPressureSensor`: Pressure in kPA of nitrogen in the environment
15. `O2AirConcentrationSensor`: Percentage of O2 in the environment.
16. `O2AirEnvironmentInFlowRateSensor`: Flow rate in liters/tick of O2 into the environment (paired with O2AirStoreInFlowRateSensor)
17. `O2AirStoreInFlowRateSensor`: Flow rate in liters/tick of O2 out of the store (paired with O2AirEnvironmentInFlowRateSensor)
18. `O2AirEnvironmentOutFlowRateSensor`: Flow rate in liters/tick of O2 out of the environment (paired with O2AirStoreOutFlowRateSensor)
19. `O2AirStoreOutFlowRateSensor`: Flow rate in liters/tick of O2 out of the store (paired with O2AirEnvironmentOutFlowRateSensor)
20. `O2AirPressureSensor`: Pressure in kPA of O2 in the environment
21. `OtherAirConcentrationSensor`: Percentage of trace gases in the environment.
22. `OtherAirPressureSensor`: Pressure in kPA of trace gases in the environment
23. `WaterAirConcentrationSensor`: Percentage of water (humidity) in the environment
24. `WaterAirEnvironmentInFlowRateSensor`: Flow rate in liters/tick of water vapor into the environment (paired with WaterAirStoreInFlowRateSensor)
25. `WaterAirStoreInFlowRateSensor`: Flow rate in liters/tick of water vapor into the store (paired with WaterAirEnvironmentInFlowRateSensor)
26. `WaterAirEnvironmentOutFlowRateSensor`: Flow rate in liters/tick of water vapor out of the environment (paired with WaterAirStoreOutFlowRateSensor)
27. `WaterAirStoreOutFlowRateSensor`: Flow rate in liters/tick of water vapor out of the store (paired with WaterAirEnvironmentOutFlowRateSensor)
28. `WaterAirPressureSensor`: Pressure in kPA of water in the environment

**Actuators**

1. `AirInFlowRateActuator`: Flow rate in moles/tick of air into the environment
2. `AirOutFlowRateActuator`: Flow rate in moles/tick of air out of the environment
3. `CO2AirStoreInFlowRateActuator`: Flow rate in liters/tick of CO2 from a store (paired with CO2AirEnvironmentInFlowRateActuator)
4. `CO2AirEnvironmentInFlowRateActuator`: Flow rate in liters/tick of CO2 into the environment (paired with CO2AirStoreInFlowRateActuator)
5. `CO2AirStoreOutFlowRateActuator`: Flow rate in liters/tick of CO2 from a store. (paired with CO2AirEnvironmentOutFlowRateActuator)
6. `CO2AirEnvironmentOutFlowRateActuator`: Flow rate in liters/tick of CO2 out of the environment (paired with CO2AirStoreOutFlowRateActuator)
7. `NitrogenAirEnvironmentInFlowRateActuator`: Flow rate in liters/tick of nitrogen into the environment (paired with NitrogenAirStoreInFlowRateActuator)
8. `NitrogenAirStoreInFlowRateActuator`: Flow rate in liters/tick of nitrogen into a store (paired with NitrogenAirEnvironmentInFlowRateActuator)
9. `NitrogenAirEnvironmentOutFlowRateActuator`: Flow rate in liters/tick of nitrogen out of an environment (paired with NitrogenAirStoreOutFlowRateActuator)
10. `NitrogenAirStoreOutFlowRateActuator`: Flow rate in liters/tick of nitrogen out of a store (paired with NitrogenAirEnvironmentOutFlowRateActuator)
11. `O2AirEnvironmentInFlowRateActuator`: Flow rate in liters/tick of O2 into the environment (paired with O2AirStoreInFlowRateActuator)
12. `O2AirStoreInFlowRateActuator`: Flow rate in liters/tick of O2 out of the store (paired with O2AirEnvironmentInFlowRateActuator)
13. `O2AirEnvironmentOutFlowRateActuator`: Flow rate in liters/tick of O2 out of the environment (paired with O2AirStoreOutFlowRateActuator)
14. `O2AirStoreOutFlowRateActuator`: Flow rate in liters/tick of O2 out of the store (paired with O2AirEnvironmentOutFlowRateActuator)
15. `WaterAirEnvironmentInFlowRateActuator`: Flow rate in liters/tick of water vapor into the environment (paired with WaterAirStoreInFlowRateActuator)
16. `WaterAirStoreInFlowRateActuator`: Flow rate in liters/tick of water vapor into the store (paired with WaterAirEnvironmentInFlowRateActuator)
17. `WaterAirEnvironmentOutFlowRateActuator`: Flow rate in liters/tick of water vapor out of the environment (paired with WaterAirStoreOutFlowRateActuator)
18. `WaterAirStoreOutFlowRateActuator`: Flow rate in liters/tick of water vapor out of the store (paired with WaterAirEnvironmentOutFlowRateActuator)

**Malfunctions**

State exactly what the malfunctions do. How do the method calls work. What is the malfunction. What is the leak rate, etc. for each of the different malfunction classes. Since method calls are probably identical may want to move this up to top and then use this section only for module-specific malfunction information.

### 8.2.4 Crew

**Sensors**

1. `CrewGroupActivitySensor`: Returns average activity level of crew
2. `CrewGroupAnyDeadSensor`: Returns whether any crew member has died
3. `CrewGroupDeathSensor`: Return whether the entire crew has died
4. `CrewGroupProductivitySensor`: Total productivity of the crew

**Actuators**

1. `CrewGroupActivityActuator`:

**Malfunctions**

### 8.2.5 Air

**Sensors**

1. `CO2InFlowRateSensor`: Reports the CO2 flow rate (in moles) into a module
2. `CO2OutFlowRateSensor`: Reports the CO2 flow rate (in moles) out of a module
3. `CO2StoreLevelSensor`: How much CO2 (in moles) is contained in this store.
4. `H2InFlowRateSensor`: Reports the H2 flow rate (in moles) into a module
5. `H2OutFlowRateSensor`: Reports the H2 flow rate (in moles) out of a module
6. `H2StoreLevelSensor`: How much (in moles) is contained in this store.
7. `NitrogenInFlowRateSensor`: Reports the nitrogen flow rate (in moles) into a module
8. `NitrogenOutFlowRateSensor`: Reports the nitrogen flow rate (in moles) out of a module
9. `NitrogenStoreLevelSensor`: How much nitrogen (in moles) is contained in this store.
10. `O2InFlowRateSensor`: Reports the O2 flow rate (in moles) into a module
11. `O2OutFlowRateSensor`: Reports the O2 flow rate (in moles) out of a module
12. `O2StoreLevelSensor`: How much O2 (in moles) is contained in this store.

**Actuators**

1. `CO2InFlowRateAcutator`: Changes how much CO2 (in moles) goes into the module
2. `CO2OutFlowRateAcutator`: Changes how much CO2 (in moles) goes out of the module
3. `H2InFlowRateAcutator`: Changes how much H2 (in moles) goes into the module
4. `H2OutFlowRateAcutator`: Changes how much H2 (in moles) goes out of the module
5. `NitrogenInFlowRateAcutator`: Changes how much nitrogen (in moles) goes into the module
6. `NitrogenOutFlowRateAcutator`: Changes how much nitrogen (in moles) goes out of the module
7. `O2InFlowRateAcutator`: Changes how much O2 (in moles) goes into the module
8. `O2OutFlowRateAcutator`: Changes how much O2 (in moles) goes out of the module

**Malfunctions**

Same as others

### 8.2.6 Water

**Sensors**

1. `DirtyWaterInFlowRateSensor`: Reports the dirty water flow rate (in liters) into a module
2. `DirtyWaterOutFlowRateSensor`: Reports the dirty water flow rate (in liters) out of a module
3. `DirtyWaterStoreLevelSensor`: How much dirty water (in liters) is contained in this store.
4. `GreyWaterInFlowRateSensor`: Reports the grey water flow rate (in liters) into a module
5. `GreyWaterOutFlowRateSensor`: Reports the grey water flow rate (in liters) out of a module
6. `GreyWaterStoreLevelSensor`: How much grey water (in liters) is contained in this store.
7. `PotableWaterInFlowRateSensor`: Reports the potable water flow rate (in liters) into a module
8. `PotableWaterOutFlowRateSensor`: Reports the potable water flow rate (in liters) out of a module
9. `PotableWaterStoreLevelSensor`: How much potable water (in liters) is contained in this store.

**Actuators**

1. `DirtyWaterInFlowRateActuator`: Changes how much dirty water (in liters) goes into the module
2. `DirtyWaterOutFlowRateActuator`: Changes how much dirty water (in liters) goes ouf of the module
3. `GreyWaterInFlowRateActuator`: Changes how much grey water (in liters) goes into the module
4. `GreyWaterOutFlowRateActuator`: Changes how much grey water (in liters) goes ouf of the module
5. `PotableWaterInFlowRateActuator`: Changes how much potable water (in liters) goes into the module
6. `PotableWaterOutFlowRateActuator`: Changes how much potable water (in liters) goes ouf of the module

**Malfunctions**

### 8.2.7 Biomass

**Sensors**

1. `BiomassInFlowRateSensor`: Reports the biomass flow rate (in kilograms) into a module
2. `BiomassOutFlowRateSensor`: Reports the biomass flow rate (in kilograms) out of a module
3. `BiomassStoreLevelSensor`: How much biomass (in kilograms) is contained in this store.
4. `FoodInFlowRateSensor`: Reports the food flow rate (in kilograms) into a module
5. `FoodOutFlowRateSensor`: Reports the food flow rate (in kilograms) out of a module
6. `FoodStoreLevelSensor`: How much food (in kilograms) is contained in this store.

**Actuators**

1. `BiomassInFlowRateActuator`: Changes how much biomass (in kilograms) goes into the module
2. `BiomassOutFlowRateActuator`: Changes how much biomass (in kilograms) goes out of the module
3. `FoodInFlowRateActuator`: Changes how much food (in kilograms) goes into the module
4. `FoodOutFlowRateActuator`: Changes how much food (in kilograms) goes out of the module

**Malfunctions**

### 8.2.8 Dry Waste

**Sensors**

1. `DryWasteInFlowRateSensor`: Changes how much dry Waste (in kilograms) goes into the module
2. `DryWasteOutFlowRateSensor`: Changes how much dry Waste (in kilograms) goes out of the module
3. `DryWasteStoreLevelSensor`: How much dry Waste (in kilograms) is contained in this store.

**Actuators**

1. `DryWasteInFlowRateActuator`: Changes how much dry waste (in kilograms) goes into the module
2. `DryWasteOutFlowRateActuator`: Changes how much dry waste (in kilograms) goes out of the the module

**Malfunctions**

### 8.2.9 Power

**Sensors**

1. `PowerInFlowRateSensor`: Changes how much power (in watts) goes into the module
2. `PowerOutFlowRateSensor`: Changes how much power (in watts) goes out of the module
3. `PowerStoreLevelSensor`: How much power (in watts) is contained in this store.

**Actuators**

1. `PowerInFlowRateActuator`: Changes how much power (in watts) goes into the module
2. `PowerOutFlowRateActuator`: Changes how much power (in watts) goes out of the module

**Malfunctions**

# Chapter 9

# Writing a simple Java controller

This Java example uses the JacOrb to connect to the BioSim. You can cut and paste this code and follow the compiling and running instructions. This code is also included in source code distributed with BioSim.

To compile this code you need to do the following:

```
% run make-client.sh
% javac -classpath /
        .:$BIOSIM_HOME\lib\jacorb\jacorb.jar:$BIOSIM_HOME\generated\client\classes /
        TestBiosim.java
```

where `javac` is the Java compiler, `jacorb.jar` is the library that has the ORB and various CORBA utilities, `generated`
`client`
`classes` are the client stubs generated from the IDL, and `TestBiosim` is the file described in this chapter.

To run this code you will need to do the following:

```
% run run-nameserver.sh
% run run-server.sh
% java -classpath /
       .:$BIOSIM_HOME\lib\jacorb\jacorb.jar:$BIOSIM_HOME\lib\jacorb:$BIOSIM_HOME\generated\client\cl
       -Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB /
       -Dorg.omg.CORBA.ORBSingletonClass=org.jacorb.orb.ORBSingleton /
       -DORBInitRef.NameService=file:$BIOSIM_HOME/generated/ns/ior.txt TestBiosim
```

/noindent `run-nameserver.sh` and `run-server.sh` are in the `$BIOSIM_HOME/bin` directory and start the CORBA name server and the BioSim server respectively. The -D options tell Java that we are using JacOrb and where to look for the ior file that the nameserver creates.

The code to connect to the BioSim server is as follows:

```
public class TestBiosim{
public static void main(String[] args){
try{
System.out.println("TestBiosim begin");
//The ORB
ORB myOrb = null;
//The naming context reference
NamingContextExt myNamingContextExt = null;
// create and initialize the ORB (with no arguments)
```

```
String[] nullArgs = null;
System.out.println("Initializing ORB");
myOrb = ORB.init(nullArgs, null);
System.out.println("Getting a reference to the naming service");
org.omg.CORBA.Object nameServiceObject = myOrb.resolve_initial_references("NameService");
System.out.println("Narrowing name service");
myNamingContextExt = NamingContextExtHelper.narrow(nameServiceObject);

//Let's get the BioDriver
System.out.println("Getting BioDriver");
BioDriver myBioDriver = BioDriverHelper.narrow(myNamingContextExt.resolve_str("BioDriver0"));
System.out.println("Invoking method on BioDriver");
//Now let's call a method on BioDriver
myBioDriver.spawnSimulationAndRunTillDead();
System.out.println("Invoking another method on BioDriver");
//Now let's call another method on BioDriver, this time with a result
int numberOfTicks = myBioDriver.getTicks();
System.out.println("Result was: "+numberOfTicks);
//All done!
System.out.println("TestBiosim end");
}
catch (Exception e){
System.out.println("Something went wrong!");
e.printStackTrace();
}
}
}
```

# Chapter 10

# Writing a simple C++ controller

This C++ example uses the ACEOrb to connect to the BioSim server. You will need to install the ACEOrb on your machine. The example also assumes a Linux operating system using the g++ compiler.

## 10.1 Makefile

We will use a Makefile to compile the C++ executable. The Makefile consists of the following:

```
INCLUDES = -I. -I$(TAO_ROOT) -I$(ACE_ROOT) -I$(TAO_ROOT)/orbsvcs/orbsvcs

biosim_client: biosimC.cpp biosim_client.cpp
       g++ -o biosim_client biosimC.cpp biosim_client.cpp /
                $(INCLUDES) /
       -L$(TAO_ROOT)/ACE_wrappers/TAO/tao  /
               -L$(TAO_ROOT)/ACE_wrappers/TAO/orbsvcs/orbsvcs /
               -L$(TAO_ROOT)/ACE_wrappers/TAO/tao/PortableServer /
               -lTAO -lTAO_PortableServer -lTAO_CosNaming /
```

Where TAO_ROOT and ACE_ROOT are environment variables that contain the pathname where TAO and ACE are installed on your machine and where biosimC.cpp is a file generated from the IDL and biosim_client.cpp is the code from this chapter.

## 10.2 Client code

```
#include "biosimC.h"
#include <CosNamingC.h>

static const char* CorbaServerName = "BioDriver0";
int
main(int argc,char *argv[])
{
  try {
    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc,argv,"TestClient");

    cerr << "initialized orb" << endl;

    if (argc < 2) {
```

```
      cerr << "Usage: " << argv[0]
           << " IOR_string" << endl;
      return 1;
    }

// Use the first argument to create the object reference,
    // in real applications we use the naming service, but let's do
    // the easy part first!

    CORBA::Object_var obj =
      orb->string_to_object (argv[1]);

    cout << "After resolve_initial_references" << endl;


    /*   CORBA::Object_var obj =
         orb->string_to_object (argv[1]); */

    CosNaming::NamingContext_var inc = CosNaming::NamingContext::_narrow(obj);

    cout << "After NameService narrow" << endl;

    if(CORBA::is_nil(inc)) {
      cerr << "Unable to narrow NameService." << endl;
      throw 0;
    }  // end if couldn't get root naming context

    CosNaming::Name name;
    name.length(1);
    name[0].id = CORBA::string_dup(CorbaServerName);
    name[0].kind = CORBA::string_dup("");

    CORBA::Object_var tmp = inc->resolve(name);

    cout << "After resolve name" << endl;

    if(CORBA::is_nil(tmp)) {
      cerr << "Unable to resolve " << CorbaServerName << " from NameService" << endl;
      throw 0;
    }  // end if couldn't find object

    cerr << "got object" << endl;

    // Now downcast the object reference to the appropriate type

    biosim::idl::framework::BioDriver_var myBiodriver = biosim::idl::framework::BioDriver::_narrow(tmp)


    cerr << "narrowed" << endl;

    // example of a call to the simulation

    myBiodriver->spawnSimulationTillDead();
```

```
    cerr << "spawned" << endl;

    orb->destroy ();
  }
  catch (CORBA::Exception &ex) {
    std::cerr << "CORBA exception raised! " << std::endl;
  }
  return 0;
}
```

## 10.3 Running the client code

To run this code you will need to do the following:

```
    % run run-nameserver.sh
    % run run-server.sh
    % ./biosim_client
```

/noindent run-nameserver.sh and run-server.sh are in the $BIOSIM_HOME/bin directory and start the CORBA name server and the BioSim server respectively.

# Chapter 11

# Control Examples

## 11.1 JSC genetic algorithm

## 11.2 Rice reinforcement learner

## 11.3 Texas Tech reinforcement learner

# Chapter 12

# Conclusions

# Index

# Index