SIMULATION ENVIRONMENT FOR PROGRAMMABLE MICROORGANISMS

Technical Report UT-CS-06-585

Andrey A. Puretskiy

Department of Computer Science
University of Tennessee, Knoxville
www.cs.utk.edu/~puretski/PILOT

December 12th, 2006

Abstract

The purpose of this project was to build an engine that could be used to create various simulations involving programmable microorganisms. The project was written using an object-oriented language called *breve*. Separate object classes were created for the organisms, chemicals, sensors and emitters. Several different quorum sensing and chemotaxis simulations were created in order to test the simulation engine. The testing was performed on two different computer systems in order to evaluate the performance of different hardware as it relates to the simulation engine.

1 Background

Through the use of genetic engineering, it is possible to create microorganisms with a customized regulatory and coding "circuitry". Through the principles of self-organization and emergent complex behavior, these programmable microorganisms could fulfill a variety of useful functions. For instance, they could serve as extremely useful assistants in fields like robotics, where they could enable adaptive behaviors, such as self-repairing and self-reconfiguration. Because certain types of microorganisms are capable of surviving extreme environmental conditions, genetically engineering them for customized purposes could be a useful new approach for exploration and future development of such environments. Deep oceanic and space exploration are just some of the fields that could potentially benefit from the development of programmable microorganism technology. The goal of this project is to create a simulation environment that would enable the exploration of the general principles of self-organization by which programmable microorganisms could accomplish complex and useful tasks.

The necessity of being able to conduct experiments in a simulated, rather than a physical environment is clear. Simulations can significantly decrease the cost of a research study, as well as the total amount of time necessary for its completion. Additionally, a good simulation system will be capable of always producing replicable results from a given set of input variable and starting conditions. This is a significant benefit of simulations, since it is seldom possible to maintain a similarly high level of control over all variables in a real-life study.

2 Introduction

The primary aim of this PILOT has been the development of a simulation engine that would have two main capabilities. First, it would allow the user to easily customize the microorganism with a variety of pre-defined components. The user would not have to create these components, thus saving a considerable amount of time. Second, it would enable the customization of the environment in which the agents will operate, simulating the presence or absence of certain chemicals. Different chemicals can be easily programmed to cause various behavior patterns in the organisms. For example, a positive or negative chemotaxis simulation can easily be created using this engine. Because the simulation's potential applications are interdisciplinary, a secondary goal has been to make the simulation environment accessible and user-friendly for individuals without extensive computer programming experience.

In order to accomplish these goals, the simulation environment has been designed within the *breve* framework, which uses an easy to read, object-oriented language called *steve* to enable multi-agent simulations[1]. Each programmable microorganism was treated as an object consisting of a number of components. These components include other objects, as well as built in methods that control the simulated microorganism's behavior in various ways. All objects involved in this project have been stored in a well-documented

library. As a result, a user with relatively little programming experience will be able to construct diverse simulation instances with relatively little effort.

3 Discussion

The following sections describe software implementation details. First, the various classes that were created for this project are described. A more technical version of the documentation is located in the "doc" subdirectory within the project directory. Next, the testing of the simulation environment is described. Several test simulations were created as part of the testing process. The final section summarizes the limitations and recommended hardware specifications for this simulation environment.

Software Implementation

The following subsections describe the overall structure of the simulation engine code, as well as the details of each subclass. The methods and capabilities of each subclass are summarized here, but a more complete version of the documentation is available in the "doc" project subdirectory. Since the sections below frequently refer to specific methods implemented in the code, an explanation of the structure of a *breve* method is necessary. The following is a template for a *breve* method with two input arguments, an integer and a string:

+ to method-name keyword#1 variable-name#1 (int) keyword#2 variable-name#2(string):

For example, the following is the definition of a method that adds a number of sensors for a particular chemical to an organism:

+ to add-sensors of name (string) numbering num (int) with max (float):

In this example, "of", "numbering" and "with" are keywords, and "name", "num" and "max" are variable names. While the method definitions may seem needlessly complicated, the goal is to improve the readability of the methods when they are called in the code. The above method may be called as follows:

TestOrg add-sensor of ActivatorChemical numbering 4 with max 1.0.

As long as the programmer chooses the keywords well, much of the *breve* code will have the appearance of regular English sentences. This can be of a significant advantage if the code needs to be modified by individuals who are not familiar with the standard programming languages, such as C, C++ and Java.

Overall Code Structure

This section describes the overall structure of the simulation engine. The chart below (Fig. 1) shows summarizes this structure.

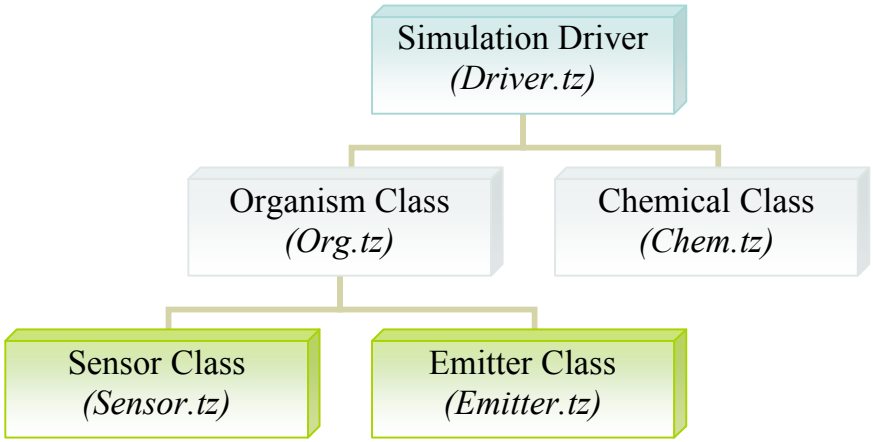

Figure 1. Simulation environment structure.

A simulation is initialized and created via the *Driver* class. The user manipulates various constants to alter the initial state of the simulation, as well as the behavior of the organisms during the course of the simulation. The number the organisms (*Org*-type objects) may be defined by the user here. Various visual qualities, such as the shape and color of the organisms may be altered here as well. Additionally, the user may create several chemicals based on the *Chem* template class. For the purposes of the simulation, a chemical is a non-physical object that diffuses within the user-specified environment. The way organisms interact with the chemicals determines the overall behavior of the simulation. The organisms are initialized to contain a number of *Sensor* and *Emitter* objects. Each sensor is set up to absorb some amount of a chemical. The chemical and the maximum amount the sensor can absorb at any given time are defined by the user. Each emitter produces some amount of a chemical. The chemical and the maximum amount the emitter can produces at any given time are defined by the user. The chart below (Fig. 2) shows the structure of a simple example simulation instance.

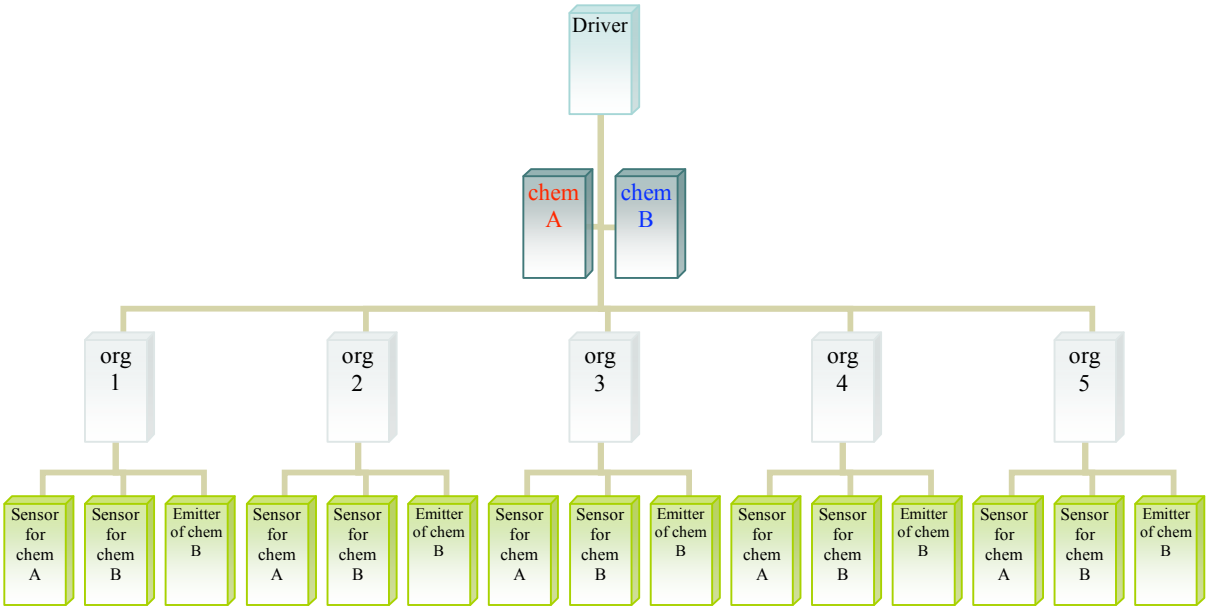

Figure 2. Initial structure of a simple example simulation.

While complexity can vary depending on how the user chooses to initialize the many various simulation attributes, this affects only the number of organisms, chemicals, sensors and emitters. The basic structure, as depicted above, remains the same.

Driver Class Implementation Details and Capabilities

The *Driver* class is contained within the *Driver.tz* file. The simulation engine was designed so that most, if not all of the user's interaction with the software would occur through modifications to the *Driver* class. Most of these modifications consist of altering various constants (*@define* statements in *breve*). The following tables include all of the constants, as well as a brief definition of each. A more complete explanation for those constants that require one is available in the documentation folder. Table 1 lists the constants that the user may utilize in order to determine the initial number of organisms, as well as the rate at which the organisms will reproduce.

Table 1. User defined constants related to the number of organisms.

Name in Code	What the constant defines or controls	Typical initial values range
ORG_COUNT	The number of organisms.	2-5000
REP_SCALE	Used with an activator-inhibitor chemical reproduction model. Controls how much influence the Inhibitor has (higher values = more influence).	0-100
REP_THRESHOLD	The minimum amount of a user-specified chemical that must be present in order for an organism to reproduce.	0-1000
REP_WAIT_TIME	The number of iterations (amount of simulation time) that must pass before an organism can divide (reproduce) following a division.	0-100000

Table 2 lists the constants that the user may alter to determine the how the organisms will move in the simulation environment. There are three choices: random motion, gradient motion and comparison motion.

If random motion is chosen, the organisms will simply select a nearby location at random and move there on the next iteration of the simulation.

If gradient motion is selected, the organisms will sample the nearby environment (all adjacent locations plus the organism's current location) for the highest concentration of a certain user-defined chemical. The organism will then either move to the location with highest concentration or, if the present location happens to be the one with the highest concentration, it will remain in place.

The third choice, comparison motion is somewhat similar in its effect to gradient motion, but it is computationally simpler. When the user selects comparison motion, the organisms will constantly compare two variables, *old_con* (concentration of the chemical at the organism's previous location) and *new_con* (concentration of the chemical at the organism's current location). This comparison determines whether the organism will keep traveling in the same direction as it is at the moment, will reverse direction, or will choose a random direction. Overall, comparison motion tends to produce more chaotic and less "perfect" organism motion patterns than gradient motion. When this motion type is selected, the organisms will sometimes exhibit "tumbling" behavior. Generally, this is way of implementing organism motion is more realistic for bacteria than pure gradient motion [2].

Table 2. User defined constants related to the movement of organisms.

Name in Code	What the constant defines or controls	Typical initial values range
MOVE_TYPE	The organisms' motion control type—this constant determines what internal method an organism will use to determine its next location.	1 = random 2 = gradient 3 = comparison
GRADIENT-MOVE-CHANCE	The chance that gradient motion will be used by the organism to determine its next location. If the value is less than 1.0, random motion is introduced with the probability of occurring equal to (1-GRADIENT-MOVE-CHANCE)	0.0-1.0
GRADIENT-MOVE-THRESHOLD	Used when gradient motion is selected by the user. This is the minimum amount of a chemical that would cause an organism to move.	0.0-10.0
COMPARE-MOVE-THRESHOLD	Used when comparison motion is selected by the user. This is the minimum amount of a chemical that would cause an organism to move.	0.0-10.0
SIGNIFICANT-DIFFERENCE	Used when comparison motion is selected, this constant symbolizes the difference between the chemical concentration at the organism's current and previous location that is considered to be significant (an insignificant difference will cause random organism motion.	

Table 3 lists the constants that may be used to customize the organisms to absorb and/or produce certain chemicals. The more sensors for a chemical an organism has, the faster it will absorb that chemical. The more emitters of a particular chemical an organism has, the faster it will produce that chemical. This may affect the overall reproduction rate of the organisms.

Table 3. User defined constants related to the composition of the organisms.

Name in Code	What the constant defines or controls	Typical initial values range
SENSOR-COUNT-A	The number of sensors for one of the two built in chemicals (*ChemA*).	0-10
SENSOR-COUNT-B	The number of sensors for one of the two built in chemicals (*ChemB*).	0-10
EMITTER-COUNT-A	The number of emitters of one of the two built in chemicals (*ChemA*).	0-10
EMITTER-COUNT-B	The number of emitters of one of the two built in chemicals (*ChemB*).	0-10
MAX-SENSOR-ABSORBTION	The amount of a chemical that a single sensor can absorb during one simulation iteration.	1.0-100.0
MAX-EMITTER-PRODUCTION	The amount of a chemical that single emitter can produce during one simulation iteration.	1.0-100.0

The remaining constants are too few to warrant a category of their own. Therefore, they are all described in a single table (Table 4) below.

Table 4. Miscellaneous user defined constants.

Name in Code	What the constant defines or controls	Typical initial values range
NAME-A	The name the user chooses to give to one of the two built in default chemicals.	Any string
NAME-B	The name the user chooses to give to one of the two built in default chemicals.	Any string
RU	The diffusion scaling factor that can control the overall rate of chemical diffusion. Lower values will cause the chemicals to diffuse more slowly.	0.0000001-10.0
XSIZE	The size of the simulation environment's X-dimension.	1-32
YSIZE	The size of the simulation environment's Y-dimension.	1-32
ZSIZE	The size of the simulation environment's Z-dimension.	1-32

The methods included in the driver class can be divided into two categories. The first category consists of the mandatory *breve* methods. A driver class must extend the built in *Controller* class in order to run. It also must include an *init* and an *iterate* method. The former initializes the simulation state, while the latter describes the actions that will be taken during each iteration of the simulation. Although quite a bit of customization can be done by altering the constants described above, there may be times when further

customization is necessary. In these cases, the user may need to make changes to one or both of these methods.

The second category consists of internal methods that are called either from within the driver class, or from one of the other classes that constitute the simulation engine. These methods, though included in the documentation, are not meant to be altered or called in any fashion different from the current.

Organism Class Implementation Details and Capabilities

Most of the work done by the simulation engine takes place within the *Org* class. The methods that constitute this class can be divided into three main categories: initialization, chemical interaction and motion control.

1. **Initialization** methods**.** An *Org* object is created using the *init xsize x (float) ysize y(float) zsize z (float)*. The three arguments specify the size of the simulation environment. These arguments are assigned to the objects internal global variable for easy access by all motion control methods. The *init* method also initializes several internal variables, and sets the default shape and color of the organism. The methods *init-color* and *init-shape* can then be called by the user to alter the color and shape of the organism. Calling these methods, however, is not required. One more method is required to complete the initialization, and that is *init-location to V (vector)*. The organism's location is defined by the vector that serves as the input argument to this method. An organism can be assigned an ID number (unique or not), using the *set-id to inp (int)* method. An organism may also have a type that can be set using the *set-type to inp (string)* method. Both of these have corresponding *get-* methods.

2. **Chemical Interaction** methods make up the second category of *Org* class methods. In turn, these can be divided into chemical interaction initialization and active interaction methods. The former consist of *add-sensors of name (string) numbering num (int) with max (float)* and *add-emitters of name (string) numbering num (int) with max (float)*. These methods are called from the *Driver* class as part of the simulation initialization. They initialize and add a user defined number of sensors and emitters to every organism. The latter consist of *use-sensors of Name (string)* and *use-emitters of Name (string)*. Both of these are called by the *Driver* class's *iterate* method as part of the simulation execution. The *use-sensors* method absorbs a chemical (as defined by its input argument) from the environment, while *use-emitters* releases a chemical (as defined by its input argument) into the environment.

3. The third category consists of the organism **Motion Control** methods. There are three main methods that may be called by the *Driver* class, depending on the user's choice of the motion model: *update-location-based-on-comparison of*

new_con (float) with thresh1 (float) and-with thresh2 (float), *update-location-based-on-gradient of Name (string) with chance (float) and-with thresh (float)* and *update-location-randomly*. The first method uses a comparison between the concentration of a chemical at the organism's current location and its concentration at the organism's previous location. The results of this comparison determine whether the organism will: (1) continue moving in its current direction; (2) move in the opposite direction; (3) move randomly. The thresholds that serve as the input arguments to this method affect this choice. The second method surveys the organism's surrounding environment for the highest concentration of a certain chemical (as defined by the input arguments). The organism then moves in the direction of the highest concentration. The third method selects a nearby location randomly. The organism then moves to that location. This method is called by the first two under certain conditions--for example, if the user would like to introduce a random error into the gradient motion method. It can also be called from the *Driver* class, if the simulation requires the organisms to use pure random motion.

In addition to the above, there are several miscellaneous internal methods within the *Org* class. Since these methods are used primarily for code modularity within the class, they are not described here. However, their descriptions are available in the documentation.

Chemical Class Implementation Details and Capabilities

The *Chem* class represents a diffusing chemical. It uses *breve's* built in *Patch Grid* class to simulate this process visually. A chemical's initial appearance and behavior are determined by the following four methods:

+ to init-size xsize XSIZE (float) ysize YSIZE (float) zsize ZSIZE (float):

+ to init-value to val (float) at-x x (float) at-y y (float) at-z z (float):

+ to init-color to choice (string):

+ to set-name to some-name (string):

The first one sets the size of the environment in which the chemical will diffuse. The second one initializes the concentration of the chemical at a given location. The third one sets the color that will represent the chemical in the simulation. The fourth method is used to name the chemical. Several other methods use the chemical's name for various purposes, so it is important that the user keep track of the names of the chemicals in the simulation.

There are two methods that other classes use to interact with a *Chem* object. The first is *get-value at-x x (float) at-y y (float) at-z z (float)*. This method simply returns the concentration of the chemical at the given location back to the calling object. The second method is *update-chem rate RU (float) tstep TIMESTEP (float) on-grid grid (object)*.

This method diffuses the chemical and re-draws its concentrations on the provided *Patch Grid* object. Typically, it is called by the *iterate* method of the *Driver* class only.

Sensor Class Implementation Details and Capabilities

Each sensor is tailored to absorb one particular chemical, up to a certain maximum value. This is done using the *init-to chem name (string) with-maximum val (float)* method. An *Org* object's *use-sensors* method calls the *Sensor* object's *sense at-x x (float) at-y y (float) at-z z (float)* method. This method works through the *Driver* to modify the concentration of a chemical in the environment.

Emitter Class Implementation Details and Capabilities

Each emitter is tailored to produce one particular chemical, up to a certain maximum value. This is done using the *init-to chem name (string) with-maximum val (float)* method. An *Org* object's *use-emitters* method calls one of the following *Emitter* object methods:

+ to emit-constant at-x x (float) at-y y (float) at-z z (float):

+ to emit amount val (float) at-x x (float) at-y y (float) at-z z (float):

+ to emit-percent percent p (float) at-x x (float) at-y y (float) at-z z (float):

These methods work through the *Driver* to modify the concentration of a chemical in the environment. The first one simply produces 1.0 units of some chemical. The second one allows the calling class to specify the amount that will be produced. The third one allows the called class to specify the percentage of the internal chemical concentration that will be emitted into the environment.

Testing

The simulation engine was tested extensively using two types of simulations: quorum sensing and chemotaxis. Multiple simulation instances were created for each type. The behavior of the simulation engine was monitored for any abnormalities.

Quorum Sensing Simulations

In a quorum sensing simulation test, the goal was to demonstrate that the organisms are capable of determining that a certain number (quorum) of organisms has been reached. All quorum sensing simulations were set up with the following qualities:

- The organisms are capable of reproducing.
- Reproduction is triggered by the presence of an Activator chemical (chemical A).
- Reproduction can be stopped by a certain level of an Inhibitor chemical (chemical B).

- The organisms are capable of moving.
- The concentration of chemical A affects organism motion.
- The organisms absorb both chemical A and chemical B via sensor(s).
- The organisms emit chemical B, but not chemical A.
- At the start of the simulation, a certain amount of chemical A is present in the environment.

In order to test the simulation engine thoroughly, the following parameters were varied across different quorum sensing simulation instances:

- The overall rate of reproduction, as controlled by several user-defined constants (described above).
- The motion model used by the organisms.
- The motion thresholds (minimum amount of chemical that must be present in order for motion to occur).
- The initial number of the organisms.
- The initial concentration of chemical A.
- The initial locations of the organisms and the spatial distribution of chemical A.
- The diffusion rate of the chemicals.
- The number of sensors and emitters for chemical A and chemical B that each organism would have.
- The maximum amount of chemical that a sensor could absorb.
- The amount of chemical that an emitter would emit.

In all of the testing simulations, the organisms were to indicate that a quorum had been reached by a cessation of motion. In all instances, the organisms eventually demonstrated quorum sensing behavior.

Chemotaxis Simulations

In the chemotaxis simulation testing, the goal was to demonstrate that the organisms were capable of using either gradient or comparison motion to converge upon a high concentration of a diffusing chemical. All chemotaxis simulations were set up with the following qualities:

- The organisms are capable of moving.
- The concentration of chemical A affects organism motion.
- The organisms use either the comparison or the gradient motion model. In either case, there is no chance of a random move.
- The organisms absorb both chemical A and chemical B via sensor(s).
- The organisms emit chemical B, but not chemical A.
- At the start of the simulation, the environment contains a slowly diffusing, centrally located quantity of chemical A.

In order to test the simulation engine thoroughly, the following parameters were varied across different chemotaxis simulation instances:

- Whether the organisms reproduce.
- The overall rate of reproduction, as controlled by several user-defined constants (described above).
- The initial location of the organisms.
- The initial location of the highest chemical A concentration point.
- The rate of chemical A's diffusion.
- The number of the sensors for chemical A that each organism would have.

As expected, in all test instances the organisms eventually converged upon the region containing the highest concentration of chemical A.

Hardware Requirements and Limitations

Two different computer systems were used to test the simulation engine, with the purpose of determining the maximum simulation complexity that each would handle. On both systems, it was determined that the engine does not process an environment larger than 32x32x32 well. The following table summarizes the maximum number of organisms and chemicals that each system processed without excessive lag:

Table 5. System Performance Summary

	System 1	**System 2**
Operating System	Mac OS X version 10.4.8	Windows XP Home Edition SP2
Processor	2.16 GHz Intel Core Duo	1.6 GHz AMD Athlon XP 1900+
Memory	1 GB 667 MHz DDR2 SDRAM	768 MB
Maximum Number of Organisms	2000	1200
Maximum Number of Chemicals	10*	10*

For the maximum number of organisms testing, the number of chemicals was set to two. For the maximum number of chemicals testing, the number of organisms was set to a constant (non-reproducing) ten. The maximum number of chemicals was not tested beyond ten. This is because ten chemicals did not seem to place significantly more demand on the system than just one or two. It appears that the number of organisms is the primary limiting factor on the simulation engine's performance.

4 Information Sources

1. Klein, Jon. "*breve* a 3d Simulation Environment for Multi-Agent Simulations and Artificial Life." http://www.spiderland.org/node/291.
2. Hartl, Daniel L. (1994). Genetics, 3rd ed. Boston & London: Jones & Bartlett, pp. 254-263.