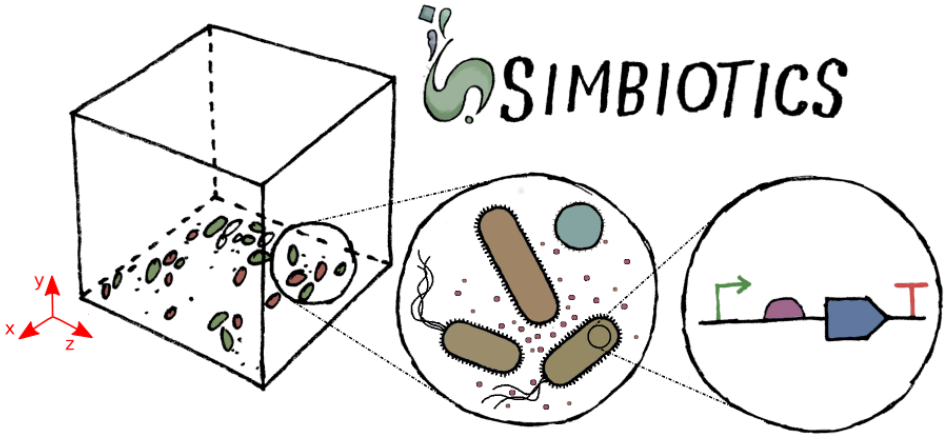
Simbiotics User Guide

Jonathan Naylor
ICOS, School of Computing Science, Newcastle University
j.r.d.naylor@ncl.ac.uk

Contents



1 Introduction

Welcome to the Simbiotics user guide! In these guide we'll take you through what the software does, how to install it, and how to use it. In brief Simbiotics is a java simulator which lets you construct models of multicellular systems, primarily populations of mixed bacterial species. Simbiotics can be used via a graphical user interface called Easybiotics. Once you have installed Simbiotics, you can go over to the *easybiotics_guide.pdf* if you wish to use that for model building/analysis.

You can also try Simbiotics in a Virtual Machine for easy out-of-the-box use, it can be found on the website along with video tutorials on how to use the software.

`https://bitbucket.org/simbiotics/simbiotics/wiki/Home`

Overview

Simbiotics is a 3D modelling platform which allows for the design, simulation and analysis of multicellular systems. The platform is focussed on modelling of bacterial populations, allowing for the representation of unique cellular species, where their individual behaviour and interactions can be defined. Through this one can simulate the emergent behaviours exhibited by the population, arising from the interplay between micorprocesses such as individual cell's genetic regulation, and macroscale processes such as dynamic spatial arrangement.

Simbiotics provides a standard modelling library for simulating typical processes of bacteria, such as growth, motility, gene regulation, metabolic activity and cell-surface appendages (receptors and adhesins). The library also provides some models of environmental factors such as a fluid mixing force, bouyancy/gravity, friction and a primitive flow chamber.

Additionally one can attach virtual devices to a model in order to probe or interact with it, allowing for a partial virtual lab experience. Data exporters can also be attached to a model in order to collate, format and write data to file. The inclusion of auxiliary programs to model specifications allows for initial conditions, repetitive tasks and desired interactions with the model to be automated.

In order to design a model in Simbiotics, modules from the Library can be attached to a model. This means that the modeller can fine tune the simulation content, designing a specification in a compositional manner in order to build bacterial and environmental models. In addition this means that only processes relevant to a model will be simulated.

License

Technical overview

Simbiotics is written in Java 1.7. It utilises the spatial representation and parallelised scheduler as implemented in Cortex3Dp **?**. Simbiotics was developed using LibSBML 5.13 and LibSBMLSim 1.3.

Terminology

To clarify some of the terminology used in this document, we list some keywords and their meaning.

Term	Meaning
Library module behaviour	Java classes within the Simbiotics library which describe model-specific
$SIMBIOTICS	The main Simbiotics folder, which contains the *src* folder

Table 1: User manual terminology

2 Getting Simbiotics

Simbiotics

Simbiotics is available at:

`https://bitbucket.org/simbiotics/simbiotics/wiki/Home`

Simbiotics is developed in Java 1.7, you must have the following installed:

- Open JDK >= 6 (GNU General Public Licence + classpath exception) or Oracle Java SE >= 6 (Oracle Binary Code Licence)

And optionally, if you wish to use SBML integration, you must have:

- libSBML - http://sbml.org/Software/libSBML (GNU LGPL)
- libSBMLSim - http://fun.bio.keio.ac.jp/software/libsbmlsim/ (GNU LGPL)

If you aren't using SBML integration you can skip to **3. Running Simbiotics** section.

Getting Dependencies

Simbiotics can be used in its minimal form without any dependencies, however if you wish to use the SBML integration you must have the following software packages installed, and linked to the project.

LibSBML

The first dependency is the Systems Biology Markup Language (SBML) library. SBML is a language format for representating computational models of biological processes, such as the metabolism and gene regulation of individual cells. Simbiotics can handle SBML through these libraries to represent populations of SBML models interacting with each other. LibSBMLj is a java interface for the SBML format, and can be downloaded at the link below:

Download LibSBML

LibSBMLsim

The second dependency is LibSBMLsim, a simulator which is used to run SBML models. It is written in C++ and LibSBMLsimj is the java interface for using the library, and can be downloaded from the link below:

Download LibSBMLsim

Linking dependencies

Once they are installed on your system (make sure you have the correct ones for your operating-system and cpu-architecture), locate *libsbmlj.jar/libsbmlsimj.jar* and *libsbmlj.so/libsbmlsimj.so* files on your system, and copy them into the *$SIMBIOTICS/jars* folder (overwrite any existing versions which are in that folder).

Compiling Simbiotics

The supplied Simbiotics source code must be compiled to an executable jar if you wish to run it from command-line. This is a simple one stage process; from command-line enter the $SIMBIOTICS root folder and run the *make* command, as such:

Listing 1: Compling Simbiotics source to executable jar

```
cd $SIMBIOTICS
make
```

3 Running Simbiotics

Simbiotics can be run in multiple ways allowing the user to choose which is most appropriate for them. Simbiotics can either by run via command-line, an IDE, or Easybiotics (see easybiotics_guide.pdf).

Using Simbiotics by command-line

Simbiotics can be launched by command line using the *jar* file, as seen below.

Listing 2: Minimal use of jar file

```
cd $SIMBIOTICS
java -jar simbiotics.jar
```

A specific configuration file may also be provided as a command line argument. The configuration file (config) contains the launch parameters for the software. Example configuration files can be found in *$SIMBIOTICS/examples/configs*. The configuration file is described in Section 6.1.

There are a range of command line parameters (arguments) for simbiotics to supply other information to simbiotics:

Listing 3: Parameters

```
-config      the configuration file
-model       the model file
-parameters  the model parameter file
-results     the target results directory
```

If a parameter which is already in the configuration file is provided as a command-line argument, it will override the value specific in the config.

Below are some other examples of launching Simbiotics from command-line.

Listing 4: Loading a model

```
#setting a custom java model
java -jar simbiotics.jar -config configs/default.json -model simbiotics.examples.Model1_Aggregation

#setting a custom java model and custom results directory
java -jar simbiotics.jar -config configs/default.json -model simbiotics.examples.Model1_Aggregation
    -results my_results/

#setting a custom JSON model and custom results directory
java -jar simbiotics.jar -config configs/default.json -model examples/models/1_aggregation.json -results
    my_results/
```

Using an Integrated Development Environment (IDE)

For this user manual the IDE we will use is Intellij 14.1, which can be downloaded at the link below.
Download Intellij 14.1

The following steps are how to open the project in Intellij, version 14.1 was used for this user guide.

1. File - New - Project from Existing Sources
2. Select the *simbiotics* main folder.
3. Create project from existing sources
4. Name the project
5. Make sure the *simbiotics* src folder path is selected
6. Make sure the libraries are selected
7. Finish

The dependencies may need to be manually linked in the IDE.

1. Navigate to File - Project Structure... (Ctrl+Shift+Alt+S)
2. Click on the Libraries tab on the left
3. Click New Project Library (Green +)
4. Choose Java
5. Navigate to the *$SIMBIOTICS/jars* folder
6. Choose one of the *.jar* or *.so* files
7. Choose to add it to the *simbiotics* module
8. Repeat this for all of the files in *$SIMBIOTICS/jars* (both *.jar* and *.so* files)

You can test that Simbiotics is running correctly by navigating the one of the example models, such as *"src\simbiotics\examples\Model1_Aggregation.java"* and run the java application with that class as the main entry.
Once you have verified that you can launch Simbiotics from the IDE, please see the section below entitled "Developing Simbiotics models in Java" for tutorials on how to build models.

4 Live visualisations of simulations

To run simulations with a live visualistion, set the ***gui*** variable in the configuration file to be ***true***. This loads the Simbiotics GUI, which renders a 3D scene which can be navigated with a camera. It also provides a tool bar with additional functions which are described below.

Visualisation layers

The renderer can be set to only display certain layers of the simulation.

Functions

Functions can be performed such as running a spectrophotometer scan on the system to take an optical density measurement.

Options

In the options menu you can pause/unpause the simulation. Additionally you may allocate more CPU threads.

View

The camera position can be modified/reset here.

Window

Popup windows can be shown to show the details of the simulation.

Recording

Both images and videos may be taken of the visualisation. Images work in the same way as having a *geometry_image* exporter attached to the model specification - it writes the properties of all the geometries in the simulation to a file.

Video recording generates an *.avi* which can be found in the *$SIMBIOTICS/results* folder.

5 Developing Simbiotics models in Java

To illustrate how to build Simbiotics models in Java, we run through some basic examples. The first tutorial is a step-by-step overview of how to create a basic model, withe the following building upon those ideas to develop more complex models.

Tutorial 1 - Creating your first model

In this first tutorial we will describe how to construct a basic model, followed by how to attach some library modules to describe model functionality and perform basic analysis and data collection.

The complete model can be found in the Simbiotics project at:
simbiotics.examples.Tutorial1_AggregationOpticalDensity

Creating a model class

First we define a new model class which extends Model. Make sure it this new class is in the Simbiotics source code folder. This class needs two functions to work, a Java *main* method to so you can start the simulation from the model class, and a *build* method in which the model specification definitions are. The *main* method should have a call to the *initialise* function, and should pass the *.class* variable of the model you are defining. The *build* method contains the model specification, and is used by calling desired *define* functions and passing in modules from the Simbiotics library. Additionally one may override the *prestep* and *poststep* methods, which are called before/after solving each iteration of the simulation, and can used for direct injection of commands as the simulation runs..

```
// define a new class which extends Model
public class MyModel extends Model {

   // define a main method in which this objects static class variable is passed into the initialise
       function
   public static void main(String[] args){
      initialise(MyModel.class);
   }

   // override the Model build method
   public void build(){
      // model definitions go here
   }

   // optionally override the Model prestep method
   public void prestep(){
      // custom modeller definitions
   }

   // optionally override the Model poststep method
   public void poststep(){
      // custom modeller definitions
   }
}
```

In the *build* function, the modeller is required to define the simulation domain size (world size). This is shown below where a world of size 100*50*100 micrometers is specified.

We also define boundary conditions which describe the behaviour at the domain boundaries. Here we set the X and Z axes to be cyclical (periodic) boundaries, such that agents which leave a face of the cuboidal domain on the X and Z axes enter from the opposing face of the domain. By default boundary conditions are set to be solid walls, in this case the Y axis (top and bottom faces of the cube) are impassable.

```
// define the world domain to be 100*50*100 micrometers (in form {x, y, z})
defineWorldSize(100, 50, 100);

// define the world X and Z boundaries to be cyclical
```

```
defineBoundary(Axis.X, new CyclicalBoundary());
defineBoundary(Axis.Z, new CyclicalBoundary());
```

Three solver systems for the model are required, namely the physical intergration solver, reaction-diffusion solver and the goemetry collisions solver. You may use different libraries modules for these, if none are loaded them the default solvers (standard solvers) are used. This is shown below, where we use the default library modules for each of the solvers.

The *StandardPhysics* module implements a verlet integrator which describes how forces are translated into velocities and positions for agent geometries. The *StandardDiffusion* module implements a finite-volume method of Fick's Law for solving the diffusion of chemicals in the world domain. The *StandardCollisions* module implements a mass-spring law to describe how intersecting agent geometries exert forces on each other.

```
// define the physics solver (StandardPhysics implements verlet integration) and add force components
definePhysics(new StandardPhysics());

// define the diffusion solver (StandardDiffusion implements a finite-volume method of Fick's law)
defineDiffusion(new StandardDiffusion());

// define the collision solver (StandardCollisions implements a mass-spring system)
defineCollisions(new StandardCollisions());
```

The modeller can define cell species using a *CellSpecies*, which describes the name and functionality of the species. Below we define two species, *"species_a"* which is red and is represented as a sphere of diameter 0.9 micrometers, and *"species_b"* which is green a sphere of 1.1 micrometers.

Populations of the two species are then defined, 300 *"species_a"* cells and 200 *"species_b"* cells by creating an initial condition.

```
// define the coccus morphology (spherical geometry)
defineMorphology(new CoccusMorphology(0.5), "coccus");

// define two species of cells
defineCellSpecies(new CellSpecies("species_a", Color.RED, "coccus");
defineCellSpecies(new CellSpecies("species_b", Color.GREEN, "coccus");

// define a population of the species
defineInitialCondition(new InitialPopulation("species_a", 300), "initial_species_a");
defineInitialCondition(new InitialPopulation("species_b", 200), "initial_species_b");
```

Loading the model in its current state results in a static scene with the inanimate cell populations suspended in the domain. This is the first step of building a typical model, providing the core components on which model functionality will be layered.

Extending the model

Defining environmental forces is done via the physics solver system. The *StandardPhysics* module can take a set of force component parameters, which describe the forces equations due to specific mechanisms. Force components are found in the *simbiotics.library.physics.components* package.

We define two force components, Brownian dynamics and friction dynamics, with force coefficients passed into their constructors.

```
// define the physics solver (StandardPhysics implements verlet integration) and add force components
definePhysics(new StandardPhysics(new Brownian(2.4), new Friction(2)));
```

Binding sites can also be used to represent targets for interactions, typically representing cell surface proteins and carbohydrates. We define two binding sites *"adhesin_a"* and *"adhesin_b"*. We then define an interaction called *"interaction_a_b"* which occurs between the two spcies of adhesin. An *InteractionTemplate* describes interaction parameters, here we set the interacton force coefficient to be 40 and the interaction rate to be 30.

```
defineBindingSite(new BindingSite("adhesin_a"));
defineBindingSite(new BindingSite("adhesin_b"));

// define the interaction and its mechanismwhich occur between adhesins
defineInteractionMechanism(new SpringMechanism(40, 30), "spring")
defineInteraction(new SpecificInteraction("interaction_a_b", new Pair("adhesin_a", "adhesin_b"),
    "interaction1"));
```

Now we have defined binding sites which have an interaction between them, we can add the binding sites our cell species definitions, this is achieved via adding a behaviour library module to the species. Below we define two behaviour modules, both instances of *CellAdhesion* which is a module implementing how cells detect binding site interactions with neighbouring cells. This module takes a parameter list of Strings, being the IDs of the binding sites which are present in that module. For our modules *"adhesion_a/b"* have their corresponding adhesin as their constructor parameter.

We then modify the cell species definitions we defined earlier; cell templates can take a parameter list of Strings after the cell geometry (sphere) parameter, these are the IDs of the behaviour modules as we defined above. Cell species *"species_a/b"* have their corresponding cell adhesion behaviour module attached to their definition, *"adhesin_a/b"* are then implicitly represented on the surface of *"species_a/b"*.

```
// define the cell behaviour module which implements cell-adhesin functionality
defineCellBehaviour(new CellAdhesion("adhesin_a"), "adhesion_a");
defineCellBehaviour(new CellAdhesion("adhesin_b"), "adhesion_b");
...

// add the new behaviour modules to the cell species templates using their unique keys
defineCellSpecies(new CellSpecies("species_a", Color.RED, "coccus", "adhesion_a");
defineCellSpecies(new CellSpecies("species_b", Color.GREEN, "coccus", "adhesion_b");
```

Binding stes can be used to define environmental structures such as binding targets on solid boundaries. We define a binding site called *"boundary_structure"*, and an interaction *"boundary_interaction"* which occurs between *"adhesin_a"* and the new boundary structure with a force coefficient of 100 and a rate of 100.

We then define a boundary condition on the Y axis, at the face of the cube where the Y coordinate is the maximum of the world domain (in Simbiotics Y max is the top face of the cuboid domain). The boundary is set to be a solid wall, and has a property object assigned to. In the property object we defined property called *"structures"*, which takes a String array of the binding sites which are present, in this case only the new binding site *"boundary_structure"*.

```
// define the new environmental binding site
defineBindingSite(new BindingSite("boundary_structure"));
...

// define the interaciton between species_a's adhesin, adhesin_a, and the environmental_structure
defineInteractionMechanism(new SpringMechanism(100, 100), "spring_2")
defineInteraction(new SpecificaInteraction("boundary_interaction", new Pair("adhesin_a",
    "boundary_structure"), "spring_2"));
...

//define the world Y boundaries to be solid, and the top substratum has a surface structure which
    interacts with species_a
defineBoundary(Axis.Y, AxisFace.MAX, new SolidBoundary(new BoundaryData(new Pair("structures", new
    String[]{"boundary_structure"}))));
```

Collecting data from the model

To collect data from the model we can define exporters, these are library modules which read desired model state information and writes it to file. Additionally the modeller can define devices, which are programs that perform built-in analysis on the model state such as measurements or interactions with the model, device data can then be

used by exporters.

For this model we can measure the aggregation of the bacterial population using a simulated spectrophotometer, emulating the process a biologist would go through to acquire such data. We first define the spectrophotometer module, then an exporter module which uses the data from this spectrophotometer. This is achieved by using the ID of the spectrophotometer in the constructor of the exporter. We take a spectrophotometer scan and export the data every 10 seconds, this sample period is the second parameter to the exporter.

```
// define the optical density device
defineDevice(new Spectrophotometer(), "spectrophotometer");

// define the optical density
defineExporter(new SpectrophotometerExporter("spectrophotometer", 10), "od600_data");
```

Tutorial 2 - Biofilm

In this tutorial we will develop a more advanced model, building on concepts we covered in the first tutorial. We first develop a primitive single species biofilm model, where planktonic cells can colonise a surface. We then extend the model, introducing a second bacterial species which performs chemotaxis towards a chemical which is produced by the first species biofilm, resulting in the second species adhering the the biofilm. Growth kinetics are introduced, as well as a boundary interface which describes a flux of new chemicals and bacteria into the world domain. Analysis is then performed to measure the biofilm height profile and this data is written to file.

The complete model can be found in the Simbiotics project at:
simbiotics.examples.Tutorial2_BiofilmHeight

Environment setup

We first define a world domain size of 100*50*100 micrometers followed by definition of cyclical (periodic) boundaries on the X and Z axes, as we did in the first tutorial. The domain boundary at the minimum value of the Y axis (bottom face of the cuboid domain) is then set to be solid with binding sites present.

We then define the solver systems for the physics, diffusion and collisions in the model. The physics system has three force components, namely forces due to gravity, Brownian dynamics and friction (viscous drag force).

```
// define a world domain of 100*50*100 micrometers
defineWorldSize(100, 50, 100);

// define the world X and Z boundaries to be cyclical
defineBoundary(Axis.X, new CyclicalBoundary());
defineBoundary(Axis.Z, new CyclicalBoundary());

// define the world Y boundaries to be solid, and the top substratum has a surface structure which
    interacts with species_a
defineBoundary(Axis.Y, AxisFace.MIN, new SolidBoundary(new BoundaryData(new Pair("structures", new
    String[]{"boundary_structure"}))));

// define the boundary structure binding site
defineBindingSite(new BindingSite("boundary_structure"));

// define the physics solver (StandardPhysics implements verlet integration) and add force components
definePhysics(new StandardPhysics(new Gravity(0.1), new Brownian(2.4), new Friction(2)));

// define the diffusion solver (StandardDiffusion implements a finite-volume method of Fick's law)
defineDiffusion(new StandardDiffusion());

// define the collision solver (StandardCollisions implements a mass-spring system)
defineCollisions(new StandardCollisions());
```

Bacterial species

A bacterial species is then defined; it's represented as a red sphere of diameter 0.9 micrometers, and has a binding site *"adhesin_a"* on its surface which may interact with the *"boundary_structure"* binding site. We then create 100 instances of the species.

```
// define the binding site
defineBindingSite(new BindingSite("adhesin_a"));

// define the interaction between species_a's adhesin (adhesin_a), and the boundary
defineInteractionMechanism(new SpringMechanism(100, 100), "spring_mechanism")
defineInteraction(new SpecificInteraction("interaction_a_boundary", new Pair("adhesin_a",
    "boundary_structure"), "spring_mechanism"));
```

```
// define the cell behaviour module which implements cell-adhesin functionality
defineCellBehaviour(new CellAdhesion("adhesin_a"), "adhesion_a");

// define the morphology
defineMorphology(new CoccusMorphology(0.5), "sphere");

// define the cell species
defineCellSpecies(new CellSpecies("species_a", new Color.RED, "sphere", "adhesion_a"));

// define cell population
defineInitialCondition(new InitialPopulation("species_a", 100), "initial_species_a");
```

Multiple bacterial species

To develop the biofilm model further we introduce a second species. We define *"species_b"*, which is represented by a blue sphere of diameter 1.1 micrometers, it has a binding site *"adhesin_b"* on its surface which may interact with *"adhesin_a"* on *"species_a"* cells.

```
// define the binding site
defineBindingSite(new BindingSite("adhesin_b"));

// define the interactions which occur between adhesins
defineInteractionMechanism(new SpringMechanism(50, 50), "spring_mechanism_2")
defineInteraction(new SpecificInteraction("interaction_a_b", new Pair("adhesin_a", "adhesin_b"),
    "spring_mechanism_2"));

// define the second morhology
 defineMorphology(new CoccusMorphology(0.65), "larger_sphere");

// define the second cell species
defineCellSpecies(new CellSpecies("species_b", Color.BLUE, "larger_sphere", "adhesion_b"));

// define second cell population
defineInitialCondition(new InitialPopulation("species_b", 50), "initial_species_b");
```

Bacterial growth

We use two forms of bacterial growth in this model. The first is a constant growth module which is not dependent on any factor, the second is a nutrient dependent growth which depends on an extracelluar nutrient. In order to represent an extracellular nutrient which undergoes reaction-diffusion dynamics, we must define the diffusion grid resolution and chemical species.

To define the diffusion grid resolution we pass a value of 3 to the StandardDiffusion constructor, this means a binary split will be recursively performed on the cuboidal domain 3 times. For our domain size of 100*50*100 micrometers, 3 binary splits mean our diffusion voxel resolution is 12.5*6.75*12.5 micrometers.

We then define the *"substance_b"* chemical which represents the nutrient, it has a diffusion rate of 50 and a degradation rate of 0.5.

We also define a *"chemotaxis"* behaviour module, which describes motility dynamics in order to ascend a chemical gradient. We set the chemoattractant to be *"substance_b"*.

```
// define the diffusion solver (StandardDiffusion implements a finite-volume method of Fick's law)
    and an integer of how many binary divisions to preform on the world domain
defineDiffusion(new StandardDiffusion(3));

// define substance_b with its diffusion and degradation rates
defineChemicalSpecies(new Chemical("substance_b", 50, 0.5));

// define species_b's oxygen chemotaxis module
```

```
defineCellBehaviour(new Chemotaxis("substance_b", 50, 50, 50), "chemotaxis");
```

We define two forms of growth in the model. For *"species_a"* a constant growth module is used, which has a growth rate of 0.0004 ± a variation of $0.0004 fgs^{-1}$.

For *"species_b"* a nutrient dependent growth module is used. We first create a reaction called *"growth_reaction"*, defining its as non-autocatalytic, then setting the maximum growth rate and reaction yield coefficient. We then add a kinetic factor describing the form of the reaction, using a MonodKinetic we set the depending substance to be *"substance_b"* and the half-saturation value to be 0.5 We then create a ReactionKineticGrowth behaviour module and attach the growth reaction we had defined. Then we set the stoichiometric yield coefficients of the reactants and products in the reaction. We set the yields to be *"substance_b"* decreasing by one unit as the cells *"biomass"* increases one unit.

Cells will divide (undergo mitosis) upon reaching twice the diameter they were at birth.

```
// define the species_a's constant growth module
defineCellBehaviour(new ConstantGrowth(0.0004, 0.0004), "growth_a");

// define the reaction kinetics for substrate-dependent growth
KineticReaction growth_reaction = new KineticReaction("growth_reaction");
growth_reaction.setAutocatalytic(false);
growth_reaction.setMaxRate(0.001);
growth_reaction.setYield(1.0);
growth_reaction.addKineticFactor(new MonodKinetic("substance_b", 0.5));

// define species_b's substance dependent growth module
ReactionKineticGrowth dependent_growth = new ReactionKineticGrowth();
dependent_growth.addReaction(growth_reaction);
dependent_growth.addYield("substance_b", -1.0);
dependent_growth.addYield("biomass", 1.0);
defineCellBehaviour(dependent_growth, "growth_b");
```

The new modules must then be added to the cell species definitions by their IDs. We modify the *"species_a"* definition to add the constant *"growth_a"* module, and modify *"species_b"* to have the nutrient-dependent *"growth_b"* module and *"chemotaxis"* module.

```
// define the cell species
defineMorphology(new CoccusMorphology(0.45), "sphere1");
defineCellSpecies(new CellSpecies("species_a", Color.RED, "sphere1", "adhesion_a", "growth_a")
);
defineMorphology(new CoccusMorphology(0.6), "sphere2");
defineCellSpecies(new CellSpecies("species_b", Color.BLUE, "sphere2", "adhesion_b", "growth_b",
    "chemotaxis")
);
```

Bacterial differentiation

To introduce bacterial differentiation to model we can embed some decision making into the cells. A cell can be in a set of discrete states, which can be turned on/off based on local environment factors. For this tutorial we represent this decision making at a high level of abstraction by using a single state, indicating whether the cell has adhered to the substratum. These states then effect the behaviour that the cell has, changing the way it interacts with its environment.

First, we will set up some cell behaviours which can be turned on when the cell attaches to the substratum. A secretor will be turned on which secretes *substance_b* at given rate. Extracellular-polymeric substances (EPS) also start being produced, EPS are represented as soft spheres.

```
// define the secretor which species_a has to secrete substance_b
defineCellBehaviour(new Secretor("substance_b", 100), "secrete_substance_b");

// define the species_a's constant growth module
```

```
defineCellBehaviour(new SecretingCapsule(0.002, 0.002, 0.05), "secreting_capsule");
```

Secondly we set the states of the species, in this instance both have one state *"SESSILE"* which is true if the cell is attached to the surface.

Links are set up, which connect cell behaviours to cell states. For both *"species_a/b"* there is a *BiofilmSensor* link, which connects their *"adhesion_a/b"* to the *"SESSILE"* state, setting the state to be true if the cell has adhered to the substratum (boundary structure) or to a cell is already sessile.

If a *"species_a"* cell is sessile it has the following behaviour:

- Turns on secretion of *"substance_b"* (StateToBehaviourLink)
- Increases its growth rate and variation (VariableChanger)

If a *"species_b"* cell is sessile is has the following behaviour:

- Turns on secretion of EPS (StateToBehaviourLink)
- Decreases its chemotaxis propel speed (VariableChanger)

```
// define the cell states
States states_a = new States();
states_a.add("SESSILE", false);
States states_b = new States();
states_b.add("SESSILE", false);

// define links
Links links_a = new Links();
links_a.add(new BiofilmSensor("adhesion_a", "SESSILE"));
links_a.add(new StateToBehaviourLink("SESSILE", "secrete_substance_b"));
links_a.add(new VariableChanger(new Pair("SESSILE", "growth_a"), new Pair("growth_rate", 0.00125)));
links_a.add(new VariableChanger(new Pair("SESSILE", "growth_a"), new Pair("deviation", 0.0005)));
Links links_b = new Links();
links_b.add(new BiofilmSensor("adhesion_b", "SESSILE"));
links_b.add(new StateToBehaviourLink("SESSILE", "secreting_capsule"));
links_b.add(new VariableChanger(new Pair("SESSILE", "chemotaxis"), new Pair("run_force", 1)));
```

We must then attach the newly defined behaviours, states and links to the cell species definitions, modify the original definitions.

```
// define the cell species
defineCellSpecies(new CellSpecies(
   "species_a", Color.RED, states_a, links_a, new Sphere(0.9),
   "adhesion_a", "growth_a", "secrete_substance_b")
);
defineCellSpecies(new CellSpecies(
   "species_b", Color.BLUE, states_b, links_b, new Sphere(1.2),
   "adhesion_b", "growth_b", "chemotaxis", "secreting_capsule")
);
```

Chemostat and bactostat

We define a flux of new bacteria and chemicals into the system. This is achieved by defining a chemostat (for chemical fluxes) and a bactostat (for bacterial fluxes), and assigning them an environment interface which describes which domain boundary they operate on.

Below we define two lists of Fluxes, one for chemicals representing a flux of acid into the system, and one for bacteria representing the flux of the two species into the domain. Flux declarations have the flux rate as their second parameter.

For chemicals we have flux of *"substance_b"* at a rate of 0.01 $\mu M s^{-1}$ μm^2. For bacteria we have a flux of *"species_a"* at a rate of 0.6 cells s^{-1}, and of *"species_b"* at 0.4 cells s^{-1}.

We then define an environment interface, describing which domain boundary this flux occurs at. Here we specific that the *MAX* boundary of the *Y* axis is where the fluxes occur, meaning that cells and chemicals are introduced from the top face of the cuboid simulation domain.

We then define the two devices, a Chemostat and a Bactostat, passing their constructors the corresponding fluxes and the target environment interface. They are also identifiable by their unique device IDs, *"chemostat"* and *"bactostat"*.

```
// set up the fluxes used for the chemostat
ArrayList<ChemicalFlux> chemical_flux = new ArrayList<>();
chemical_flux.add(new ChemicalFlux("substance_b", 0.001));

// set up the fluxes used for the bactostat
ArrayList<BacterialFlux> bacteria_flux = new ArrayList<>();
bacteria_flux.add(new BacterialFlux("species_a", 0.6));
bacteria_flux.add(new BacterialFlux("species_b", 0.4));

// define the environment interface
EnvironmentInterface environment_interface = new EnvironmentInterface(Axis.Y, AxisFace.MAX)

// define up the chemostat and bactostat devices with their respective fluxes
defineDevice(new Chemostat(chemical_flux, environment_interface), "chemostat");
defineDevice(new Bactostat(bacteria_flux, environment_interface), "bactostat");
```

Biofilm height measurements

To analyse the model we take measurements of the biofilm height. This gives us both the average and standard deviation of the biofilm height, as well as a 2D heatmap which encodes the biofilm height profile.

First we define the biofilm height measuring device which samples the height of the biofilm across the entire world domain. Its scan resolution is defined in its constructor by as X and Z resolution, here we set that resolution to be 2 micrometers on both the X and Z axes. We give it a device ID of *"biofilm_height_measurer"*.

We then define a data exporter specifically for this device. We pass the ID of the device we defined above to instruct the exporter to use data collected from this device. The second parameter is the sample period of data collection, it's set to export the data every 25 seconds. The exporter unique ID *"biofilm_height_data"* is the name of the file which will hold this default, it can be found in the results directory which is defined in the Simbiotics configuration.

```
// define the biofilm height measuring device
defineDevice(new BiofilmHeight(2, 2), "biofilm_height_measurer");

// define the biofilm height exporter
defineExporter(new BiofilmHeightExporter("biofilm_height_measurer", 25), "biofilm_height_data");
```

6 Input/output

Configuration file

The configuration file is the first argument when loading Simbiotics from command-line, it is the only compulsory argument. It describes the parameters for Simbiotics which can be seen in Table 2 below. When developing in an IDE, the configuration parameters exist in the SimbioticsConfig class.

Listing 5: Simbiotics configuration file

```
{
  "model_file": "simbiotics.examples.Model1_Aggregation",
  "results_dir": "results/",
  "duration": 0,
  "simple_workers": 1,
  "complex_workers": 4,
  "max_nodes_per_pm": 20000,
  "node_depth": 0,
  "slot_resolution": 20,
  "balance_round": 300
  "verlet_update": 10,
  "view_width": 1280,
  "view_height": 800,
  "parallel": true,
  "profiling": false,
  "gui": true
}
```

Parameter	Description	Type
model_file	The path to the model class/file to be simulated	String
results_dir	The default results directory for data exporting	String
duration	Number of simulated seconds before exiting, 0 means indefinite	Double
simple_workers	Number of simple worker threads	Integer
complex_workers	Number of complex worker threads	Integer
max_nodes_per_pm	Number of agent geometries in partition before it is split into subpartitions	Integer
node_depth	Number of binary splits of the cuboid domain into the diffusion grid of subdomains	Integer
slot_resolution	Number of voxels in each subpartition	Integer
balance_round	Number of iterations before the domain is checked if it should be split into subdomains	Integer
verlet_update	Number of iterations before updated a cells verlet list (nearest neighbours)	Integer
view_width	Width of the GUI frame in pixels	Integer
view_height	Height of the GUI frame in pixels	Integer
parallel	Whether the simulation should be run in a parallelized manner	Boolean
profling	Whether the simulation profiling data should be displayed	Boolean
gui	Whether the simulation should be run with a GUI	Boolean

Table 2: Simbiotics configuration parameters

Keyboard/mouse interactivity

There are some default key bindings provided in Simbiotics. These can only be run when the Simbiotics GUI is also loaded ($gui = true$ in configuration file).

Input	Action
Left click + drag	Translates the model visualisation
Right click + drag	Rotates the model visualisation
‘	Toggles pause
a	Saves data for all exporters
q	Takes a 3D snapshot of all geometric agents (for post rendering)
Spacebar	Toggles the colour scheme

Table 3: Simbiotics input commands

Inputs

Microscopy images

Microscopy images can be processed and loaded into Simbiotics to specify the initial spatial arrangement of bacteria. This is achieved by using the *MicroscopyLoader* class, which is available in the *simbiotics.loader* package.

Calling the *generatePopulation* function requires 3 parameters, in the following form:

Listing 6: Loading microscopy images into Simbiotics

```
MicroscopyLoader.generatePopulation(image_file, image_dimensions, world_dimensions);
```

image_file is a csv file encoding the microscopy image. *image_dimensions* is 3D double array containing the size of the image file in pixels. *world_dimensions* is the size the image will be scaled down to in the simulation.

Listing 7: Loading microscopy images into Simbiotics

```
PopulationEncoding my_population = MicroscopyLoader.generatePopulation("encoding.csv", new
    double[]{1024, 1024, 1024}, new double[]{92, 92, 92});

definePopulation(my_population);
```

SBML models

SBML models can be embedded in agents in Simbiotics. This is achieved by using the *SBMLModule* behaviour class, which is available in the *simbiotics.library.behaviour.sbml* package.

Listing 8: Loading SBML models into Simbiotics

```
SBMLModule my_sbml = new SBMLModule("my_sbml_file.xml", 1, 0.1);
```

The SBML module can then be defined as a behaviour and attached to a cell species definition as such:

```
defineCellBehaviour(my_sbml, "sbml_metabolism");
defineCellSpecies(new CellSpecies("my_species", Color.BLUE, new Sphere(1.0), "sbml_metabolism"));
```

Outputs

Data exporting

Data exporters output files to the $results_dir$ defined in the configuration file, unless their $file\_path$ variable is set, in which case that specific exporter outputs data to that folder, whilst the results outputs to the main results directory. Simulations also copy a version of the model used to run the simulation.

7 Modelling library

world

2D_world 2D simulation domain
3D_world 3D simulation domain

boundaries

solid solid domain boundary for agents
cyclical cylical domain boundary for agents
no_return no return domain boundary for agents

surface_properties

adhesive adhesive structure on a solid domain boundary

forces

gravity force of gravity on agents (-Y axis force)
brownian force of brownian motions agents (random walk)
friction force of friction on agents (drag force)
interactions_force force of interactions on agents (the defined specific interactions)
collisions force of collisions between only spherical (coccus) agents
collisions_extended force of collisions between only rod-shaped (bacillus) agents
collisions_complete force of collisions between mixed spherical (coccus) and rod-shaped (bacillus) agents
collisions_hertzian force of collisions modelled as hertzian interaction between agents
non_specific force of non-specific interactions (van der Waals and electrostatic appropximation)
dlvo force of non-specific interactions according to DLVO theory

chemicals

chemical a chemical that can diffuse in the extracellular space
intracellular_only_chemical a chemical that can only exist in an intracellular compartment

interactions

specific_interation a specific interaction between two binding sites on agents surfaces

interaction_mechanisms

spring_mechanism a specific interaction is modelled as a Hookian spring forming between the interacting agents

states

state a qualitative intracellular state (boolean)
quantitative_state a quantitative intracellular state (continuous value)

links

state_to_state connect two states, such that state 2 always updates to be state 1's value
state_to_behaviour connect a state and a behaviour, such that the behaviour's activity (on or off) is equal the the states boolean value
behaviour_to_state connect a state to a behaviour, such that the states boolean value is equal to the behaviours activity variable (on or off)
state_is_external_concentration connect a state (quantitative state) to an external chemical, such that the states value is equal to the extracellular concentration of that chemical

surface_sensor connect a state to a surface sensor, such that the state is true if the agent is interacting with a solid boundary

conditions

general_condition define a custom condition
concentration_threshold check if a chemical concentration in relation to a set threshold
touched_cell check if the agent is touching a cell of a specific species
world_time check if a certain duration of global simulation time has elapsed
concentration_vs_concentration check the concentration of two chemicals against each other
has_interactions check if the cell has an interaction of a specific type
touched_surface_with check if the agent is touching a boundary with a specific surface property

actions

general_action define a custom action
change_state change the value of a state
new_behaviour add a new behaviour module to the agent
remove_behaviour remove a behaviour module from the agent
behaviour_activity set the activity of a behaviour module to be on or off
change_colour change the colour of the agent
kill_cell kill the agent
divide trigger the agent to divide (mitosis)
produce_child trigger the agent to create a child agent
delayed_action do an action after a given duration of time
probabilistic action do an action with a given probability
break_interactions remove all interactions (specific interactions) of a given type

morphpologies

coccus representation of a coccus (spherical) cell morphology
bacillus representation of a bacillus (rod-shaped) cell morphology

species

cell representation of a cell agent, which has states, links, behaviours and a morphology
eps representation of an eps agent, which has a morphology

behaviours

periodic_action an action that occurs periodically
trigger a list of conditions and actions, where once all conditions are met, all actions are executed
mitosis a cell divides upon reaching twice of its original mass
eps_secretion cells secrete EPS (agents, represented as small spheres) at some rate
conjugation models conjugation between physically contacting bacteria
cell_adhesion models membrane surface structures and specific interactions between cells
sbml models intracellular dynamics as SBML models which are solved with LibSBMLsim
differential_equations models intracellular dynamics as sets of ordinary differential equations
chemotaxis models chemotaxis of bacteria - run and tumble dynamics ascending chemical gradients
reporter changes the colour of the cell based on the value of a state
toxicity kills the cell upon it experiencing over a threshold of a specific chemical
membrane models membrane transport of a cell (active or passive mechanisms can be defined)
random_walk models a random walk of a cell (similar to brownian motion force)
pressure_death kills the cell upon it experiencing more than a define threshold of physical pressure
gillespie models intracellular dynamics as a stochastic Gillespie model
constant_growth models a cell growing at a constant rate
boolean_reporter agent changes between two colours based on a boolean state

boolean_grn models intracellular dynamics as a boolean network
run_tumble models flagellar based run and tumble motility dynamics

initial_conditions

initial_chemical_concentration define an initial chemical concentration at a position
initial_chemical_quantity define an initial chemical quantity at a position
initial_chemical_concentration_everywhere define an initial chemical concentration throughout the whole domain
initial_chemical_quantity_everywhere define an initial chemical quantity throughout the whole domain
initial_intracellular_chemical_concentration define an initial chemical concentration inside cells of a specific species
initial_intracellular_chemical_quantity define an initial chemical quantity inside cells of a specific species
initial_cell_state_activity define the initial value of a specific state of a specifc species
initial_cell_position define a cell (agent) at a specific position
initial_population define a well mixed population of cells
initial_population_in_area define a well mixed population of cells within a specific area (or volume)
initial_population_image define the initial spatial arrangement of cells from a previous simulation state
initial_population_microscopy_image define the initial spatial arrangement of cells from a processed microscopy image encoding
initial_grid_of_cells define a uniform grid of cells of a specific species

devices

chemostat define a chemostat attached to a domain boundary
bactostat define a bactostat attached to a domain boundary
chemical_pool define a chemical pool at a specific point
chemical_source define a chemical source at a specific point
chemical_sink define a chemical sink at a specific point
canera define a camera to record the simulation

exporters

sampler samplers collection custom dave from the simulation
geometry_image geometry images (population images) export the spatial arrangement of cells (can be used to initialise models)
timers exports the timers profiling the Simbiotics integrator
microsensor exports the chemical concentration at a given position
positional_cell_chemical exports a spatial description of intracellular chemical quantities (for heatmaps etc)
orientation exports the orientation of agents

schedules

save_and_exit saves all data collection to file and exits the simulation
export_periodically flushes and saves data exporters to file periodically
pipette_event schedules a pipette event (adding chemicals or agents to the domain)
camera_rotate schedules the camera to rotate at a given rate
camera_pan schedules the camera to pan at a given rate

Model definitions

Java functions

defineWorldSize

Defining the world size sets the simulation domain dimensions.

```
void defineWorldSize(double world_size)                                    (1)
void defineWorldSize(double world_x, double world_y, double world_z)       (2)
```

Where *world_size* is the length of a cubic domain. Alternatively one can have a cuboidal domain, where*world_x* is the length of the domain along the X axis, *world_y* the length of the domain along the Y axis, and *world_z* the length of the domain along the Z axis.

In Simbiotics, the X axis is right/left, the Y axis is up/down and the Z axis is back/front, with the positive/negative values being the respective direction for each axis.

defineBoundary

Defining boundaries sets the behaviour of agent geometries when they interact with the sides of the cuboidal world domain. Specific boundary behaviours can be set to particular faces of the domain by specifying the Axis and AxisFace parameters (2), if no AxisFace parameter is passed (1) then the boundary condition is applied to both the minimum and maximum faces of the given axis.

```
void defineBoundary(Axis axis, BoundaryCondition boundary_condition)                    (1)
void defineBoundary(Axis axis, AxisFace axis_face, BoundaryCondition boundary_condition) (2)
```

Where *axis* is the target axis (X, Y, Z), *axis_face* is which face of the cube along that axis (MIN, MAX) and *boundary_condition* is an implementation module describing boundary mechanics.

definePhysics

Defining the physics solver sets the integration method for calculating how agent geometries positions change due to forces.

```
void definePhysics(PhysicsSolver physics_solver)
```

Where *physics_solver* is an implementation module of the physics solver.

defineDiffusion

Defining the diffusion solver sets the method used for calculating chemical fluxes between domain subvolumes.

```
void defineDiffusion(DiffusionSolver diffusion_solver)
```

Where *diffusion_solver* is an implementation module of the diffusion solver.

defineCollisions

Defining the collision solver sets the method used for calculating the forces geometries which are colliding exert on each other.

```
void defineCollisions(CollisionSolver collision_solver)
```

Where *collision_solver* is an implementation module of the collision solver.

defineChemicalSpecies

Defines a chemical species to be part of the model with given ID and properties.

```
void defineChemicalSpecies(Chemical chemical)
```

Where *chemical* is an implementation module of a chemical, which can be present in extracellular and intracellular compartments.

defineChemicalInterface

Defines a flux of chemicals at a point position in the domain, which can be identified with an ID.

```
void defineChemicalInterface(ChemicalInterface chemical_interface, String id)
```

Where *chemical_interface* is an implementation module of a chemical interface, and *id* is the name of that interface.

defineBindingSite

Defines a binding site which can represent a physical binding location on the surface of cellular geometries and boundary interfaces.

```
void defineBindingSite(BindingSite binding_site)
```

Where *binding_site* is an implementation module of a binding site.

defineInteraction

Defines an interaction which can represent the physical mechanism between two binding sites.

```
void defineInteraction(PhysicalInteraction interaction)
```

Where *interaction* is an implementation module of a PhysicalInteraction.

defineCellBehaviour

Defines a behaviour module to be identified by its ID and key, which can then be bound to cell species definitions to describe cell dynamics.

```
void defineCellBehaviour(iBehaviour behaviour, String module_id, String module_key, Boolean active)
```

Where *behaviour* is an implementation module of an iBehaviour, *module_id* is its unique identifier, *module_key* is the type of behaviour corresponding to the Simbiotics library keys, and *active* is a boolean whether the behaviour is active (on) or inactive (off).

defineCellSpecies

Defines a cell species with a particular implementation, such as their spatial representation, behaviour and state information.

```
void defineCellSpecies(CellSpecies cell_species)
```

Where *cell_species* is an implementation module of CellSpecies.

definePopulation

Defines the initial population size of the cell species, their positions are distributed normally throughout the cubic domain.

```
void definePopulation(String species_id, int population_size)
```

Where *species_id* is the target species ID, and *population_size* is the number of cells .

defineCellAtPosition

Defines a cell of the given species at a position, can also have a unique cell name to track an individual cell throughout the simulation.

```
void defineCellAtPosition(String species_id, double[] position)
void defineCellAtPosition(String species_id, double[] position, String cell_name)
```

Where *species_id* is the target species ID, *position* is the coordinates of the cell, and *cell_name* is the unique name of that cell.

defineInitialVelocity

Defines the initial velocity for all cells in at the initial state of the model with some random deviation.

```
void defineInitialVelocity(double velocity, double standard_deviation)
```

defineDevice

Defines a device which may interact with or probe the model state, indentifiable by its ID.

```
void defineDevice(iDevice device, String device_id)
```

defineExporter

Defines an exporter to write model data to file, it's identifiable by its ID and has an optional file path of where to write the data to. If no file path is supplied then the default results folder as defined in the Simbiotics configuration will be used.

```
void defineExporter(Exporter exporter, String exporter_id)
void defineExporter(Exporter exporter, String file_path, String exporter_id)
```

defineAuxiliary

Defines an auxiliary program which may automate interactions or events in the model, identifiable by a unique ID.

```
void defineAuxiliary(iAuxiliary auxiliary, String auxiliary_id)
```

defineDrawer

Defines a model component to visual for 3D rendering output.

```
void defineDrawer(Drawer drawer)
```

defineConstant

Defines a constant for the simulation engine, such as the global *"TIME_STEP"*.

```
void defineConstant(String id, double value)
```

8 Building new modules

The Simbiotics library can be extended by designing new modules in Java. This is achieved by meeting the requirements of one of the Simbiotics interfaces. The interfaces are all stored in the same Java package, found in: *simbiotics.plugs.interfaces*. The base classes implementing these are found in: *simbiotics.plugs.base*.

When developed a new module, one may wish to *extend* one of the base classes, or for finer control, they may wish to directly *implement* the interface. For example, when developing a new module for bacterial behaviour, the *Behaviour* class could be *extended*, or one may *implement* the *iBehaviour* interface directly.

We exemplify this through how the *Mitosis* behaviour is implemented. We start by overriding the *iBehaviour* interface, which can be seen below.

```
public interface iBehaviour extends Serializable, CustomSerializable {

   /** Execute (run) the behaviour module **/
   void execute();

   /** Apply the changes of the behaviour module **/
   void apply();

   /** Get a copy of the module**/
   iBehaviour getCopy();

   /** Divide the module (with a given ratio) **/
   iBehaviour divide(double ratio);

   /** If returns true, this module is copied to the child cells during division **/
   boolean isCopiedWhenCellDivides();

   /** Return the agent this module belongs to **/
   iAgent getAgent();

   /** Set the agent this module belongs to **/
   void setAgent(iAgent cell);

   /** Get the probability of this behaviour module being inherited during division**/
   double getInheritanceProbability();

   /** Set the probability of this behaviour module being inherited during division **/
   void setInheritanceProbability(double inheritance_probability);

   /** Get behaviour module id (String name) **/
   String getBehaviourId();

   /** Set the behaviour module id (String name) **/
   void setBehaviourId(String module_id);

   /** Set the behaviour module type (String type) **/
   void setBehaviourType(String module_type);

   /** Get the behaviour module type **/
   String getBehaviourType();

   /** Get whether the behaviour module is active (switched on) **/
   boolean isActive();

   /** Set the behaviour module activity to be on or off **/
   void setActive(boolean active);

   /** Returns true if the module has an associated volume (biomass) **/
   boolean hasVolume();
```

```
   /** Get the volume of this module **/
   double getVolume();
}
```

This interface has a set functions which have to be implemented in order for it to be treated as a valid behaviour module. The base implementation of this interface is the *Behaviour* abstact class, which can be seen below:

```
public abstract class Behaviour implements iBehaviour {

   public String behaviour_id = "";
   public String behaviour_type = "";

   public boolean active = true;
   public double inheritance_probability = 1.0;
   protected iAgent agent;

   /** Behaviour Constructor(s) **/
   public Behaviour() {
      this(true, 1.0);
   }
   public Behaviour(String behaviour_type){
      this.behaviour_type = behaviour_type;
      this.behaviour_id = behaviour_type;
   }
   public Behaviour(boolean active){
      this(active, 1.0);
   }
   public Behaviour(boolean active, double inheritance_probability){
      setActive(active);
      setCopyOnDivide(inheritance_probability);
   }

   /** Execute (run) the cell behaviour - toggled by 'active' variable **/
   public void execute(){
      if(active)
         run();
   }

   /** Run the module at each iteration **/
   public abstract void run();

   /** Apply the module at each iteration **/
   public abstract void apply();

   /** Get a copy of the module */
   public abstract iBehaviour getCopy();

   /** Get a copy of the module */
   public iBehaviour divide(double ratio){
      return getCopy();
   }

   /** Get the cell this module is associated with **/
   public iAgent getAgent() {
      return agent;
   }

   /** Set the cell this module is associated with **/
   public void setAgent(iAgent cell) {
      this.agent = cell;
   }
```

```
    /** Set whether the module is copied on divide **/
    public void setCopyOnDivide(double copy_on_divide){
        this.inheritance_probability = copy_on_divide;
    }

    /** Returns true if this module is copied upon mitosis (for child cell) **/
    public boolean isCopiedWhenCellDivides() {
        return inheritance_probability > 0;
    }

    /** Set the module ID **/
    public void setBehaviourId(String module_id){
        this.behaviour_id = module_id;
    }

    /** Set the module type **/
    public void setBehaviourType(String module_type){
        this.behaviour_type = module_type;
    }

    /** Get the module type **/
    public String getBehaviourType(){
        return behaviour_type;
    }

    /** Get the module ID **/
    public String getBehaviourId(){
        return behaviour_id;
    }

    /** Returns true if the module is currently active **/
    public boolean isActive(){
        return active;
    }

    /** Set whether the module is currently active or not **/
    public void setActive(boolean active){
        this.active = active;
    }


    public double getInheritanceProbability(){
        return inheritance_probability;
    }
    public void setInheritanceProbability(double inheritance_probability){
        this.inheritance_probability = inheritance_probability;
    }
    public boolean hasVolume(){
        return false;
    }

    public double getVolume(){
        return 0;
    }
}
```

This is an abstract class and thus can not be instantiated - it simply provides the general implementation that most behaviour modules need so that code doesn't have to be repeated in those classes. Often this *Behaviour* class is sufficient to extend, and one can override all the methods in the class as they please - however at some point it makes more sense to implement the *iBehaviour* class directly to reduce any efficiency costs associated with

instantiating many instances of a class with a chain of super constructors.

In the case that the *Behaviour* class is sufficient to build on top of, it can be used to specific specific behaviour of an agent. Below we show a *Mitosis* class implemented to divide a coccus (spherical) cell upon reaching twice of its original volume:

```
public class Mitosis extends Behaviour {

    protected boolean divide;

    /** Constructor(s) and copy **/
    public Mitosis(){
        super("mitosis");
        setActive(true);
    }
    public Mitosis getCopy(){
        return new Mitosis();
    }

    /** Run this cell internal (store results for a synchronous update) **/
    public void run(){
        divide = false;
        if(agent.getMass() >= agent.getBirthMass() * 2)
            divide = true;
    }

    /** Update all of the cells synchronously **/
    public void apply(){
        if(divide)
            agent.divide();
    }
}
```

And here is an example of a variation of the *Mitosis* class which implements a bacillus (rod-shaped) cell dividing upon twice its original length:

```
public class Mitosis extends Behaviour {

    protected boolean divide;

    /** Constructor(s) and copy **/
    public Mitosis(){
        super("mitosis");
        setActive(true);
    }
    public Mitosis getCopy(){
        return new Mitosis();
    }

    /** Run this cell internal (store results for a synchronous update) **/
    public void run(){
        divide = false;
        CapsularBody body = agent.getBody();
        if(body.getLength() >= 2 * body.getBirthLength())
            divide = true;
    }

    /** Update all of the cells synchronously **/
    public void apply(){
        if(divide)
            agent.divide();
    }
```

```
}
```

Please note: neither of these *Mitosis* class implementations are the one actually used in the platform, they are simply examples of how a new module can be built.