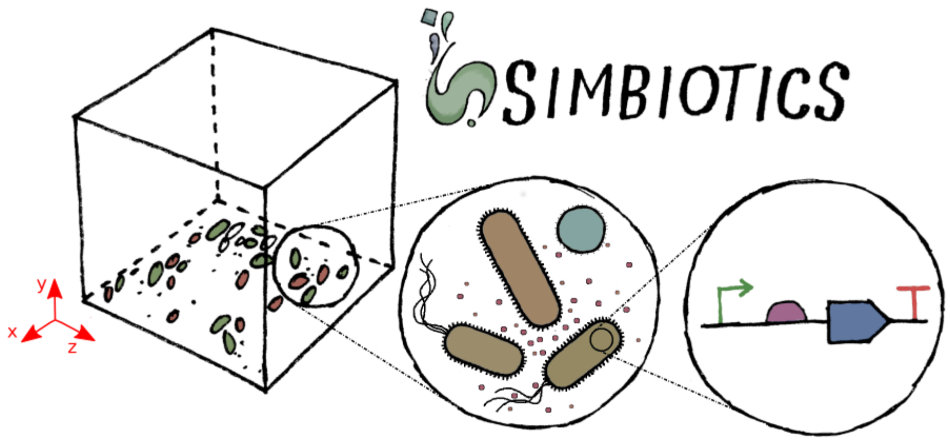
Simbiotics User Guide

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

Contents



1 Introduction

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, cellular species with individual behaviour and interactions with their environment can be defined. Large populations of multispecies communities can be simulated, with population dynamics emerging from the interplay between individual cells. Through this one can observe a colony self-organising based on cell metabolism, genetics and interactions. Simbiotics provides a standard modelling library of processes typical to bacteria, such as growth, motility, gene regulation, metabolic activity and cell-surface appendages such as 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.

Simbiotics is written in Java, and utilises a spatial representation and parallelised scheduler as implemented in Cortex3Dp.

1.1 License

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details: http://www.gnu.org/licenses/gpl.html

1.2 Terminology

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

Term	Meaning
Library module	Java classes within the Simbiotics library which describe model-specific behaviour, maybe also be called an implementation module.
$SIMBIOTICS	The main Simbiotics folder, which contains the *src* folder

Table 1: User manual terminology

2 Getting Simbiotics

Simbiotics is a Java project which can be run from command-line or opened in an integrated development environment (IDE). Simbiotics and the its dependencies must be downloaded, installed and linked to the project.

2.1 Simbiotics

Simbiotics is available free at:
Download Simbiotics

2.2 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

2.3 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

3 Running Simbiotics

Simbiotics can be run in multiple ways allowing the user to choose which is most appropriate for them. The two main routes for running Simbiotics are: running the software from command-line or opening the project in an IDE. Before Simbiotics can be run we must first link the dependecies (LibSBML and LibSBMLsim), and then compile the source code.

3.1 Linking dependencies

Once LibSBML and LibSBMLsim are installed on your system (make sure you have the correct ones for your OS and architecture), you must link them to the Simbiotics project. 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).

3.2 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
```

3.3 Using command line

Simbiotics can be launched by command line using the *jar* file. A configuration file must be provided as a command line argument, the config file contains the launch parameters for the software. A default configuration file is provided in the *$SIMBIOTICS/configs* folder. The configuration file is described in Section 4.1.

To run Simbiotics from command-line, change directory to the $SIMBIOTICS root directory (*simbiotics_1.0/*).

Linux/Mac

Where $\langle config \rangle$ is the Simbiotics configuration file path, $\langle model \rangle$ is an optional parameter specifying the path to a model file, and $\langle results \rangle$ is an optional parameter to set the root path for exported simulation results.

Running from jar

The jar file can be used to run a Simbiotics model in the following manner.

Listing 2: Jar file arguments

```
simbiotics.jar <config> <model> <results>
```

The configuration file is the only compulsory argument, thus Simbiotics can be run as simply as:

Listing 3: Minimal use of jar file

```
cd $SIMBIOTICS
java -jar simbiotics.jar configs/default.json
```

The target model, results directory and other parameters all exist in the configuration file, however one may wish to override the target model and results directory via command line, this can be done as much:

Listing 4: Loading a custom model

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

#setting a custom model and custom results directory
java -jar simbiotics.jar configs/default.json simbiotics.examples.Model1_Aggregation
    $CUSTOM_RESULTS_DIRECTORY/
```

Windows

...TBA

3.4 Opening in 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)

3.5 Developing models

To develop models in Simbiotics one can write their own Java class which extends the *Model* class, add it the source code folder, and then compile the source to a jar (as described in 3.1). Alternatively one may open the project in an IDE and develop classes in there, compilating and running can be done from within the IDE.

To aid in the development of models, there are a series of examples models (5) and modelling tutorials (6).

4 Input/output

4.1 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

4.2 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
a	Forces data exporters
q	Takes a 3D population snapshot
Spacebar	Toggles the colour scheme

Table 3: Simbiotics input commands

4.3 Inputs

4.3.1 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. Usage of the microscopy laoder module is as such:

```
MicroscopyLoader(<data file>, <image dimensions>, <loading dimensions>)
```

Where `data file` is the path to the target microscopy image data file, `image dimensions` is the resolution of the micropscopy image data, and `loading dimensions` describes the resolution where the image data cells will be loaded within.

This will be described in more detail through the following example:

Listing 6: Loading microscopy images into Simbiotics

```
defineWorld(200, 200, 200);

double[] image_dimensions = new double[]{1024, 1024, 256};
double[] loading_dimensions = new double[]{128, 128, 32};

PopulationEncoding my_population = MicroscopyLoader.generatePopulation("encoding.csv", image_dimensions,
    loading_dimensions);

definePopulation(my_population);
```

Here we create a new cellular population from a microscopy image data file called *encoding.csv*. This microscopy data file was generated from a z-stack of microscopy images, where each z-stack slice was 1024*1024*4px (in the order of x, y, z). In the z-stack there were 64 slices, therefore the final resolution of the z axis is not 4px, but infact $4 * 64 = 256$. Thus we have an image size of 1024*1024*256.

The simualtion dimensions must describe a cuboid which is within the defined simulation domain. It must also have the same proportions as the image dimensions. In our example we define a world which is 200*200*200μm, the image dimensions are 1024*1024*256 which do not fit into the world domain. We must set the `loading dimensions` to be the converted image dimensions which fit within the world domain. We set them to be 128*128*32, these dimensions have the same ratio as the image dimensions and fit within the world domain. When the microscopy image data is loaded, the encoded cell coordinates are mapped from the image dimensions to the loading dimensions.

Once the population is generated via the *MicroscopyLoader* class, that population must be defined to be a population in the model via the *definePopulation* function.

4.3.2 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. SBML module usage is as such:

```
SBMLModule(<SBML file>, <timestep>, <SBML timestep>)
```

where `SBML file` is the path to the target SBML model file, `timestep` is the time between SBML module runs, and `SBML timestep` is the internal timestep for the SBML solver (must be smaller than `timestep`). An example of how to use the SBML module can be seen below:

Listing 7: Loading SBML models into Simbiotics

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

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

Here we create an SBML module, which contains the SBML model contained within *my_sbml_file.xml*. The SBML model is run every 1 second, and has an internal time step of 0.1 seconds. The SBML module can then be defined as a behaviour and attached to a cell species definition, using the $defineBehaviour$ and $defineCellSpecies$ functions.

4.4 Outputs

4.4.1 Data exporting

4.4.2 Snapshot rendering

5 Specification

6 Example models

The Simbiotics source code contains a set of example models which utilise a range of library functionality. These models can be simulated and can be used as templates for creating your own models using the library functionality.

All models exist in the package *simbiotics.examples* (src/simbiotics/examples).

Model1_Aggregation
Caggregation between two species of bacteria, based on cell-surface receptors.

Model2_BoundaryConditions
Extended Model1$_{t}$*oincludeasoliddomainboundarysurfacetowhichbacteriamayadhere, andperiodic(cyclical)domainboundaries.*

Model3_Biofilm
Dualspeciesbiofilmformedfromaninitiallyplanktonicpopulation.Thegreenspeciescanadheretothesurface, andredspeciescana

Model4_BooleanNetworkGenetics

Coaggregation between two species of bacteria, based on cell-surface receptors.

Model5_NutrientDependence

Model6_DifferentialEqGenetics

Model7_MembraneTransport

Model9_SBMLIntegration

Model10_MicroscopyLoading

Model11_BoundaryInterfaces

Model12_BacterialDiffusionCoefficient

Model14_ConstantGrowth

Model15_DLVO_population

Model16_Gillespie

Model17_HertzianContact

Model18_Chemotaxis

Model19_LiveGraph

Tutorial1_AggregationOpticalDensity

Tutorial2_BiofilmHeight

7 Modelling tutorials

This section has two tutorials to demonstrate how Simbiotics models can be defined. The first tutorial is a step-by-step overview of how to create a basic model, and the second tutorial builds upon these ideas to develop a more complex model specification. These models can be run at the end of each subsection.

7.1 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

7.1.1 Creating a model class

First we define a new model class which extends Model. This class needs two functions to work, a typical Java *main* method to start the application, and a *build* method in which the model 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 *step* method, which is called at each iteration of the simulation and can used for custom modeller defined uses.

```java
// 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 step method
   public void step(){
      // custom modeller definitions
   }
}
```

The modeller is required to define the world domain size which will be simulated. 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.

```java
// 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());
```

Defining three solver systems for the model is required, namely the physical intergration solver, reaction-diffusion solver and the goemetry collisions solver. 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 *CellTemplate*, 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. The *definePopulation* function randomly positions cells with a normal distribution throughout the world domain.

```
// define two species of cells
defineCellSpecies(new CellTemplate("species_a", Color.RED, new Sphere(0.9));
defineCellSpecies(new CellTemplate("species_b", Color.GREEN, new Sphere(1.1));

// define a population of the species
definePopulation("species_a", 300);
definePopulation("species_b", 200);
```

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.

7.1.2 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 interactions which occur between adhesins
defineInteraction(new PhysicalInteraction("interaction_a_b", new Pair("adhesin_a", "adhesin_b"), new
    InteractionTemplate(40, 30)));
```

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 CellTemplate("species_a", Color.RED, new Sphere(0.9), "adhesion_a");
defineCellSpecies(new CellTemplate("species_b", Color.GREEN, new Sphere(1.1), "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
defineInteraction(new PhysicalInteraction("boundary_interaction", new Pair("adhesin_a",
    "boundary_structure"), new InteractionTemplate(100, 100)));
...

//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"}))));
```

7.1.3 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");
```

7.2 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

7.2.1 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());
```

7.2.2 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
defineInteraction(new PhysicalInteraction("interaction_a_boundary", new Pair("adhesin_a",
    "boundary_structure"), new InteractionTemplate(100, 100)));

// define the cell behaviour module which implements cell-adhesin functionality
```

```
defineCellBehaviour(new CellAdhesion("adhesin_a"), "adhesion_a");

// define the cell species
defineCellSpecies(new CellTemplate("species_a", new Color.RED, new Sphere(0.9), "adhesion_a"));

// define cell population
definePopulation("species_a", 100);
```

7.2.3 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
defineInteraction(new PhysicalInteraction("interaction_a_b", new Pair("adhesin_a", "adhesin_b"), new
    InteractionTemplate(50, 50)));

// define the second cell species
defineCellSpecies(new CellTemplate("species_b", Color.BLUE, new Sphere(1.2), "adhesion_b"));

// define second cell population
definePopulation("species_b", 50);
```

7.2.4 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 $\pm$ a variation of 0.0004 fg $s^{-1.}$

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 diamater 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
defineCellSpecies(new CellTemplate("species_a", Color.RED, new Sphere(0.9), "adhesion_a", "growth_a")
);
defineCellSpecies(new CellTemplate("species_b", Color.BLUE, new Sphere(1.2), "adhesion_b",
    "growth_b", "chemotaxis")
);
```

7.2.5 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 CellTemplate(
   "species_a", Color.RED, states_a, links_a, new Sphere(0.9),
   "adhesion_a", "growth_a", "secrete_substance_b")
);
defineCellSpecies(new CellTemplate(
   "species_b", Color.BLUE, states_b, links_b, new Sphere(1.2),
   "adhesion_b", "growth_b", "chemotaxis", "secreting_capsule")
);
```

7.2.6 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<Flux> chemical_flux = new ArrayList<>();
chemical_flux.add(new Flux("substance_b", 0.001));

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

7.2.7 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");
```

8 Modelling library

Model specifications are composed of java classes which extend the Simbiotics' Model class.

Definition	Description	Interface	Base implementation
World Size	Simulation domain size	-	-
Physics	Physics solver	-	PhysicalSystem
Diffusion	Diffusion solver	-	DiffusionSystem
Collisions	Collisions solver	-	CollisionSystem
Chemical	Chemical species	-	Chemical
ChemicalInterface	Chemical fluxes at a given position	iChemicalInterface	-
BindingSite	Binding sites which which may interact	-	BindingSite
Interactions	Interactions between binding sites	iInteraction	PhysicalInteraction
Cell	Cellular (bacterial) species	iCell	Cell
States	Set of states an Agent can have	iStates	States
Behaviour	Behaviour module for a cellular agent	iBehaviour	Behaviour
Geometry	Geometry which physical represents an agent	iGeometry	Geometry
Devices	Device can interact/analyse the model state	iDevice	Device
Exporters	Exporter which can write data to file	iExporter	Exporter
Auxiliary	Program which can automate model events	iAuxiliary	Auxiliary

Table 4: Growth kinetic equations, were μ is the growth rate, S is a given substance concentration and K is the half-saturation constant of a given substance.

8.1 Model definitions

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)
```

8.2 Library modules

Chemical

Represents a chemical species in the world domain.

```
Chemical(String id, double diffusion_constant, double degradation_constant, boolean diffusable, Color
    colour)
```

ChemicalSource

Represents a flux of chemicals at a position into the source domain. Could be used to represent a pipette or point source of a chemical.

```
ChemicalSource(double[] position, Flux... fluxes)
ChemicalSource(double[] position, ArrayList<Flux> fluxes)
ChemicalSource(double[] position, boolean active, Flux... fluxes)
ChemicalSource(double[] position, boolean active, ArrayList<Flux> fluxes)
```

ChemicalPool

Represents a set chemical concentration at a point in the simulation. Could be used to set a maximum concentration point which can diffuse to form a gradient in the domain.

```
ChemicalPool(double[] position, Flux... concentrations)
ChemicalPool(double[] position, boolean active, Flux... concentrations)ChemicalPool(double[] position,
    ArrayList<Flux> concentrations)
ChemicalPool(double[] position, boolean active, ArrayList<Flux> concentrations)
```

ChemicalSink

Represents a flux of chemicals a specific point out of the domain.

```
ChemicalSink(double[] position, Flux... fluxes)
ChemicalSink(double[] position, ArrayList<Flux> fluxes)
ChemicalSink(double[] position, boolean active, Flux... fluxes)
ChemicalSink(double[] position, boolean active, ArrayList<Flux> fluxes)
```

CellTemplate

Represents a species definition for a cell in the domain.

```
CellTemplate(String id, Color color, Geometry geometry)
CellTemplate(String id, Color color, States states, Geometry geometry)
CellTemplate(String id, Color color, States states, Links links, Geometry geometry)
CellTemplate(String id, Color color, Geometry geometry, String... module_ids)
CellTemplate(String id, Color color, States states, Geometry geometry, String... module_ids)
CellTemplate(String id, Color color, States states, Links links, Geometry geometry, String... module_ids)
CellTemplate(String id, Color color, Geometry geometry, iBehaviour... modules)
```

BindingSite

Represents a binding target/active site in the simulation domain. Could represent a receptor or adhesin on the surface of a cell, or an adhesive structure on a substratum.

```
BindingSite(String site_id)
```

InteractionTemplate

Represents the dynamics of binding site interactions, modelled as a spring connecting the two cels.

```
InteractionTemplate(double spring_constant)
InteractionTemplate(double spring_constant, double rate)
InteractionTemplate(double spring_constant, double rate, double extension_percentage)
InteractionTemplate(double spring_constant, double rate, double extension_percentage, double
    spring_offset)
```

StandardPhysics

Implements the physics solver to be force-based dynamics with Verlet-strömer integration.

```
StandardPhysics(Force... forces)
```

Collisions

```
Collisions()
```

PhysicalBonds

```
PhysicalBonds()
```

Random

```
Random(double force_constant)
```

Friction

```
Friction(double force_constant)
```

Gravity

```
Gravity(double force_constant)
```

StandardDiffusion

Implements the diffusion solver to be a finite-volume method of Fick's law.

```
StandardDiffusion(int binary_splits)
```

StandardCollisions

Implements the collision solver to be a soft-sphere implementation with mass-spring dynamics dictating forces geometries exert on each other.

```
StandardCollisions()
```

States

Represents state information of a cell. Can be used to describe modeller-specific detail, such as a high level state describing whether a cell has adhered to a substratum, or a low level detail such as the MRNA concentration of a particular gene.

```
public States(String states_id, iState... states)
```

Links

Represents some form of relationship between a cell *state* and a cell *behaviour*. Could be used to set a bacterial behaviour only to become active if the cell is experiencing specific stimulii.

```
public Links(String links_id, iLink... links)
```

Geometry

Represents the physical geometry of an agent in the world domain. Currently Simbiotics only has sphere representations.

Sphere

```
Sphere(double birth_diameter)
Sphere(double birth_diameter, double birth_mass)
Sphere(double birth_diameter, Adherence adherence)
Sphere(double birth_diameter, double birth_mass, Adherence adherence)
```

Flux

Represents a flux of a species at some rate. Could be used to represent a flux of chemicals or bacteria into the simulation domain.

```
Flux(String species_id, double rate)
```

Pair

Represents a pair of data which may be used in a multitude of ways. Could represent a pair of binding sites in an interaction. This module is mainly used in other module constructors in order to cluster related parameter data.

```
Pair(K key, V value)
```

CellAdhesion

Represents the adhesive behaviour that a cell has.
Could be used to model adhesins/receptors on the surface of a cell which can bind with those on the surface of other cells or surfaces.

```
CellAdhesion()
CellAdhesion(String... sites)
CellAdhesion(BindingSite... sites)
CellAdhesion(ArrayList<BindingSite> sites)
```

NonSpecific

Represents the electrostatic interactions between geometries.

```
NonSpecific(double p_adhesion, double range_factor, double spring_constant)
```

SecretingCapsule

Represents the production of extracellular polymeric substances (EPS) which are secreted as particulates into the local extracellular space of the cell.

```
SecretingCapsule(double volume, double volume_threshold, double growth_rate)
```

RunTumble

Represents the motility behaviour of a cell due to an active flagellar. Alternates between a run (clockwise flagellar rotation) and a tumble (counter-clockwise flagellar rotation) to perform a random walk.

```
RunTumble(double p_end_run, double p_end_tumble)
```

Chemotaxis

Represents a cell ascending a chemical gradient. Uses run and tumble dynamics which are modulated by chemical concentration memories. Could be used to represent a cell searching for a nutrient.

```
Chemotaxis(double p_end_run, double p_end_tumble)
Chemotaxis(String chemoattracant, double p_end_run, double p_end_tumble)
```

BooleanGRN

Represents cell decision making or processing. Could be used to represent state transitions that a cell undergoes, such as gene regulation or phenotype changes, depending on the level of abstraction the modeller wishes to capture.

```
BooleanGRN(double update_period)
BooleanGRN(double update_period, States states)
BooleanGRN(double update_period, BooleanNetwork network)
```

DifferentialGRN

Represents a set of differential equations which may describe a cells gene regulation.

ProteinData

```
ProteinData(double protein)
```

MrnaProteinData

```
MrnaProteinData(double mrna, double protein)
```

Conjugation

Represents the transfer of genes between adjacent cells.

```
Conjugation(String gene_id)
```

ConstantGrowth

Represents a constant growth of the behaviour with some random variations. Could represent an autotroph which can be said to grow at a specific rate.

```
ConstantGrowth(double growth_rate)
ConstantGrowth(double growth_rate, double deviation)
```

ReactionKineticGrowth

Represents a nutrient-dependent growth. Can model multiple reactions utilising multiple chemicals in order to produce biomass. Yields can be set in a stoichiometry matrix. Additionally one can set the growth to be autocatalytic if needed.

```
ReactionKineticGrowth()
ReactionKineticGrowth(HashMap<String, iReaction> reactions, HashMap<String, Double> yields)
```

iReaction

Represents a chemical reaction in the system. It consists of kinetic factors which describe the dynamics of the reaction. Could be used to represent metabolic processes of a cell, converting nutrients to biomass.

```
addKineticFactor(iKineticFactor kinetic_factor)
```

KineticReaction

```
KineticReaction()
KineticReaction(String reaction_id)
KineticReaction(String reaction_id, iKineticFactor... kinetic_factors)
```

iKineticFactor

Represents a kinetic factor of a reaction.

```
setChemicalId(String chemical_id)
```

Kinetic

MonodKinetic

InhibitingKinetic

MembraneTransport

Represents the membrane of a cell geometry. Can be used to model active or passive transport mechanisms which can take chemicals into or out of the cell.

```
MembraneTransport()
```

ActiveTransport

```
ActiveTransport()
```

PassiveTransport

```
PassiveTransport()
```

SBMLModule

Represents an SBML model in the world domain. Can be used to represent a cells internal dynamics, and can be connected with links to existing Simbiotics definitions such as states, behaviours and chemicals.

```
SBMLModule(String sbml_model, double time_step, double sbml_time_step)
```

8.3 Model configuration

Constants

Represents the constants used in the model. Can set the integration time step and other default parameters.

SimbioticsConfig

Describes the input parameters to the Simbiotics simulation core.

9 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. For example, when developing a new module for bacterial behaviour, the *Behaviour* class could be extended. If greater control is required than extending the existing base class, one may implement the interface *iBehaviour* directly.

Below is a schematic showing all of the Simbiotics interfaces which may be customised.

9.1 States

9.2 Behaviours

9.3 Links

9.4 Geometries

9.5 Agents

9.6 Devices

9.7 Exporters

9.8 Auxiliary

10 Software architecture