GESSA Users Manual

Ludmila V. Danilova (ludmila.danilova@gmail.com)
Elana J. Fertig (ejfertig@jhmi.edu)

July 29, 2010

Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this document, we describe the Matlab code for a hybrid model of transcriptional reprograming due to cell signaling called GESSA (Graphically Extended Stochastic Simulation Algorithm). Specifically, this model combines (1) PPBN (Pooled Probabilistic Boolean Network) graphical models of cell signaling; (2) Stochastic Simulation Algorithm (SSA) models for transcription and translation processes; (3) diffusion of external signals (ligands); and (4) active transport of specified molecular species. Simulations of organisms defined a user-defined central driver script (Section 3.1) which evolves organisms defined through classes described in Section 4.1 with a central simulator class (Section 4.2). An example simulation for vulval development in *C. elegans* is released with this model as an example of its utility (Section 3.4).

The Matlab code for the GESSA model for cell signaling and transcriptional responses is freely available at `http://www.cancerbiostats.onc.jhmi.edu/GESSA.cfm`. The sofware is distributed under the GPL Version 3, as described in the Copyright section above. Installation instructions are provided in Chapter 2, running instructions in Chapter 3, and software overview for developers in Chapter 4.

We note that the functions `simulate.m` and `stoch.m` in `$GESSAPath/HybridModel/Classes/@TTUpdater/private` are adapted from the SSA code of Ullah M, Schmidt H, Cho KH, Wolkenhauer O: Deterministic modelling and stochastic simulation of biochemical pathways using MATLAB. Syst Biol (Stevenage) 2006, 153(2):53-60.

Please contact Ludmila V. Danilova `ludmila.danilova@gmail.com` or Elana J. Fertig `ejfertig@jhmi.edu` for GESSA citation information.

Chapter 2

Installation instructions

The archive `GESSA-1.0.0.zip` containing the GESSA code can be downloaded from `http://www.cancerbiostats.onc.jhmi.edu/GESSA.cfm`. This code requires at least Matlab Release 2008b. To install GESSA, this archive should be unzipped into a path (referred to as the system variable `$GESSAPath` throughout this User's Manual). The unzipped folder will contain the following folders:

`Celegans` Sample code, data files, and output from a simulation of wild type *C. elegans* vulval development.

`HybridModel` Classes and functions needed to run GESSA.

`UsersManual` This `GESSA.pdf` file and other supporting documents.

Chapter 3

Running instructions

In order to run GESSA to simulate cell signaling and transcription/translation events in an organism, we recommend that the user create a central driver script as specified in Section 3.1. This script will run GESSA in three major steps: (1) reading in organism and simulation specifications from configuration files (Section 3.2), (2) running the simulation, (3) and creating diagnostics to assess phenotypes from this simulation (Section 3.3). The first two steps are controlled through a central function `$GESSAPath/HybridModel/Functions/driveHybridModel` which inputs a central simulation file (Section 3.2.1) and outputs the results of the simulation. Although the user must define the appropriate diagnostics for the simulated organism, we have provided numerous tools for these diagnostics described in Section 3.3. An example of such a run used to model wild type vulval development in *C. elegans*, released in `$GESSAPath/Celegans` is provided in Section 3.4.

3.1 Driver script

In order to simulate an organism with GESSA, a user must create a driver script to implement this organism simulation. An example of such a script is provided for *C. elegans* in `$GESSAPath/Celegans/CElegansDriver.m` and will serve as a model for this description of construction of a driver script.

Any driver script should begin with the following commands

```
addpath('$GESSAPath/HybridModel/Classes')
addpath('$GESSAPath/HybridModel/Functions')
```

to make the GESSA classes and functions available to the simulation. In particular, the driver script will have access to the central GESSA simulation function `driveHybridModel`. This function accepts the following input arguments:

- Central simulation file that specifies the organism and simulation parameters. The format of this file is described in Section 3.2.1.
- (OPTIONAL) An output file name for the simulation results. If not specified, called "false" or by an empty string, results will not be saved.

This function will then read in the configuration files, initialize the organism and simulation, run the simulation for the specified time parameters and specified number of organism copies, and save the results and configuration files in the specified archive. After running, `driveHybridModel` will return the following variables:

1. `OrganismHistory` object (see Chapter 4) containing simulation results averaged across all organism copies.

2. Array of `OrganismHistory` objects containing simulation results for each organism copy.

3. Name of archive to which configuration files and simulation results were saved.

4. Start time of the simulation.

5. End time of the simulation.

We have developed several diagnostic tools (Section 3.3) which will input the `OrganismHistory` objects output by this simulation to create plots and assess phenotypes from specified species states within the organism. For example, `$GESSAPath/Celegans/CElegansDriver.m` uses the `plotSpeciesHistory` function to plot the protein history for the RAS pathway in p6.p cell and uses a user defined function `TF_snapshots` to infer transcription factor activity every half hour of simulation time (see Section 3.4). The *C. elegans* driver script adds these results to the archive created by the `driveHybridModel` function.

3.2 Model configuration files

The GESSA simulation requires several configuration files to specify simulation parameters, organism components, evolution processes, and process interactions. Wherever possible, these configuration files conform to SBML standards. However, for aspects including timing and geometry, which are not described by SBML, we developed appropriate formats described below.

Table 3.1 provides an overview of the GESSA configuration files, include references to each of the corresponding files in `$GESSAPath/Celegans/Data/WT_3cells` used for the *C. elegans* example described in (Section 3.4). The content and format of each of these files is described in further detail in the following subsections, included in the table for each file type. This section of the manual also describes modifications that should be made to these files to represent standard experimental conditions in the GESSA simulations.

Table 3.1: List and brief description of configuration files required by GESSA.

File	Format	Section	Example
Central configuration	txt	3.2.1	`simulationParameters.txt`
Organism geometry and lifecycle	xml	3.2.2	`3_cells_info.xml`
External signal propagation	xml	3.2.3	`diffusion.xml`
Cell signaling	SBML	3.2.4	`network_3cells.xml`
Transcription/translation	SBML	3.2.5	`celegans_TT_LAG2_EGL17.xml`; `celegans_TT_LIN39_UP1_LIN1a.xml`; `celegans_TT_LIP1.xml`
Process mapping	txt	3.2.6	`updaterMaping.txt`

3.2.1 Simulation specification

Central configuration file with simulation specification is a tab-delimited file with fields described in Table 3.2. Each line of the file represents a field and value. Fields in the file can be in arbitrary order. This file contains paths to all necessary configuration files, name of output archive, start and end time of simulation, the number of simulated organisms, and time steps for all updaters and averaging organism history. An example of the file is provided in the `$GESSAPath/Celegans/Data/WT_3cells/simulationParameters.txt`.

Table 3.2: List of fields for the central configuration file.

Field	Required/ Optional	Value	Comments
`modelFile`	Required	Path to file	Path to file(s) with organism geometry and lifecycle (Section 3.2.2), parameters for cell signaling (Section 3.2.4) and transcription/translation processes (Section 3.2.5). The values of properties for the object of the `Organism` class (Section 4.1) is read from the file(s).
`PPIinitFile`	Optional	Path to file	The file contains initiation values for cell signaling. The `PPIUpdater.initializer` method reads the file.
`TTinitFile`	Optional	Path to file	The file contains initiation values for transcription/translation process. The `TTUpdater.initializer` method reads the file.
`DiffusionInitFile`	Optional	Path to file	The file contains initiation values for external signal propagation. The `DiffusionUpdater.initializer` method reads the file.
`outputArchive`	Required	String	Specifies a name of archive to which configuration files and simulation results is saved.
`simulationStartTime`	Required	Time in seconds	The start time of simulation.
`simulationEndTime`	Required	Time in seconds	The end time of simulation.
`numberOfOrganisms`	Optional	Integer number	Number of organisms. Default value is 1.
`timeStepPPI`	Required	Time step in seconds	A time step for cell signaling (`PPIUpdater`).
`timeStepTT`	Required	Time step in seconds	A time step for transcription/translation process (`TTUpdater`).
`timeStepDiffusion`	Required	Time step in seconds	A time step for external signal propagation (`DiffusionUpdater`).
`timeStepAverage`	Required	Time step in seconds	A time step for averaging organism history.

3.2.2 Organism geometry and cell lifecycle

Organism geometry and cell lifecycle are specified in xml file. This file should be referenced in the central configuration file in the `modelFile` field. An example of the file is provided in `$GESSAPath/Celegans/Data/WT_3cells/3_cells_info.xml`. This file contains the following tags and their attributes:

`Organism` The main tag of the xml document, containing the name of the organism being simulated. For example, it is set to `CElegans` in the sample file provided in the `Celegans` folder

`notes` String for identification of the file. The value must be "cells_info" to ensure that the program will read the information from the file.

`listOfCells` Container for cells of the organism that are represented by `cell` tags. One or more instances of the `cell` tag can be located in an instance of `listOfCells` tag.

`cell` Describes cell geometry, lifecycle and external signals received by a cell inside the `listOfCells` tag. The `cell` tag has the following attributes:

- `name` Name of the cell matching a name of cell in the `listOfCompartments` tag of the cell signaling file (Section 3.2.4).
- `alive` Boolean value that specifies whether this cell is alive at start time specified in the `startTime` attribute.
- `leftNeighbor` Name of a cell to the left.
- `rightNeighbor` Name of a cell to the right.
- `startTime` Time in seconds when this cell becomes alive.
- `divisionTime` Time in seconds when this cell divides.
- `growthRate` Rate of cell growth.

`signal` Describes a signal that the cell receives inside the `cell` tag. One or more instances of the `signal` tag can be located in an instance of `cell` tag. This tag has the following attributes:

- `id` String for identification of the signal. This attribute should correspond to the `id` attribute of the `signal` tag in the external signal propagation xml (Section 3.2.3).
- `distanceToSource` A distance to a source of the signal. This value is used for the external signal propagation modeling in a `DiffusionUpdater` object (Section 4.2.5).

Note, the simulation will start at the minimum of all `cell startTime` if it is greater than the `simulationStartTime` parameter of the central configuration file (Section 3.2.1) and end at the maximum of all `cell endTime` if it is less than the `simulationEndTime`. Also, in this version, the `cell` attribute is limited to a linear geometry with left and right neighbors. We will relax this geometric requirement in future versions. Finally, although specified, the `growthRate` attribute of `cell` is currently unused in this version, but is provided for future versions. This growth rate is also currently confined to a universal parameter for each cell in the current version. We will relax this constraint to allow for changing distances between each signal and source in future versions of the model.

3.2.3 External signal propagation

This configuration file that describes parameters for external signal propagation is in xml format. This file should be referenced in the central configuration file in the `DiffusionInitFile` field. A `DiffusionUpdater` object (Section 4.2.5) uses the parameters specified in this file to model signal propagation. An example of the file is provided in `$GESSAPath/Celegans/Data/WT_3cells/diffusion.xml`. This file contains the following tags and their attributes:

`Organism` The main tag of the xml document, containing the name of the organism being simulated. For example, it is set to `CElegans` in the sample file provided in the `Celegans` folder.

`annotation` Any user-defined string for description of the file.

`signalList` Container for signals that are represented by `signal` tags. One or more instances of the `signal` tag can be located in an instance of `signalList` tag.

`signal` Describes a signal in the `signalList` tag. This tag has the following attributes:

- `id` Character for identification of the signal. This attribute should correspond to the `id` attribute of the `signal` tag in the organism geometry and cell lifecycle xml (Section 3.2.2).

`ligand` Name of diffused species. It should correspond to the species name in cell signaling file (Section 3.2.4).

`type` Type of diffusion.

`speed` Speed of signal propagation. For diffusion, represents the diffusion coefficient.

`sourceGrowthRate` Rate at which source signal changes in time (positive for growth, negative for decay).

`sourceMagnitude` Initial magnitude of the signal at the source. If a point source, magnitude resulting from integrating the delta function describing that source.

`startTime` Start time in seconds of signal propagation from source.

`endTime` End time in seconds of signal propagation from source.

Note. The current version of the program supports only a single diffusion source for each species. We will modify this in future versions. Furthermore, the `sourceGrowthRate` is currently unused in our model, but serves as a placeholder for future propagation models along with the evolving geometry specified in the geometry and lifecycle xml (Section 3.2.2).

3.2.4 Cell signaling

The cell signaling configuration file is a file in SBML Level 2 Version 4 format (see `http://sbml.org/Main_Page`). We adapted this format in this model to enable the SBML format to capture all the necessary attributes to describe the state of signaling species and reactions. We describe our usage of the SBML tags and attributes in this model below. This file should be referenced in the central configuration file in the `modelFile` field. An example of the file is provided in `$GESSAPath/Celegans/Data/WT_3cells/network_3cells.xml`.

File Identifier

The `notes` tag of the file must be set to "network" to indicate that the file contains information about cell signaling to ensure proper file processing.

List of Compartments

In the `listOfCompartments` tag of the file, all cells of organism must be specified. Each cell must be describe in `compartment` tag inside the `listOfCompartments` tag. Cell names must correspond to the names in organism geometry and cell lifecycle file (Section 3.2.2).

List of Species

The list of species involved in signaling reactions are specified in the `listOfSpecies` tag. Each species is specified in `species` tag inside `listOfSpecies`. One or more instances of the `species` tag can be located in an instance of `listOfSpecies` tag. SBML format does not have attribute to refer to a species state (active, inactive, bound and unbound to scaffold), so we use the `id` attribute to specify the name and the state of the species. Each species has several `species` tags corresponding to active or inactive and bound or unbound to scaffold states. To specify species state, add the following letters to the species name:

`a` for active

`i` for inactive

`_b` for bound to scaffold

`_u` for unbound to scaffold

For example, setting the `id` attribute to `LIN45i_u` means that LIN45 species is inactive and unbound. These additional letters are necessary to initialize correctly the properties of the `CellProperties` object (Section 4.1.3) and they are removed from the species name after reading. The program uses the species names without the additional letters.

We use the `name` attribute of the `species` tag to specify the type of the species in the different properties of the `CellProperties` object (Section 4.1.3). See Table 3.3 for details.

Table 3.3: Naming convention for species types in the cell signaling configuration file

Value of the `name` attribute	**Property of the `CellProperties` object**
`activator`	`ProteinTypes`
`repressor`	`ProteinTypes`; `RepressorTypes`
`TF`	`ProteinTypes`; `canTransport`
`receptor`	`ReceptorTypes`
`external_signal`	`ExternalSignals`
`neighbor_signal`	`ExternalSignals`
`scaffold`	`ScaffoldTypes`

List of Reactions

The `listOfReactions` tag specifies the interactions between species in the signaling module. A reaction is described in the `reaction` tag inside the `listOfReactions` tag. We use the `name` attribute of the `reaction` tag to enable the SBML format to refer to the type of signaling reaction, as is required in our model. A list of the reaction types recognized by the cell signaling process is provided in Table 3.4.

Table 3.4: Naming convention for reactions in the cell signaling configuration file.

Value of the `name` attribute	**Description**	**Property of the `CellProperties` object**
`scaffold`	Binding and unbinding proteins to scaffold	`Scaffolds`
`ES_activation`	Activation of receptor by external signal	`ExternalSignals`
`activation_receptor`	Protein activation by receptor	`Receptor2Target`
`spontaneous_receptor`	Spontaneous activation and inactivation of receptor	`SpontaneousActivationReceptors`; `SpontaneousInactivationReceptors`
`activation`	Activation of signaling protein by signaling protein	`PPI`
`activation_bound`	Activation of bound protein by signaling or bound protein	`PPI`
`repression`	Repression of signaling protein by signaling protein	`PPI`; `Repressors`
`repression_bound`	Repression of bound protein by signaling protein	`PPI`
`spontaneous_unbound`	Spontaneous activation and inactivation of signaling and unbound proteins	`SpontaneousActivationUnboundProteins`; `SpontaneousInactivationUnboundProteins`
`spontaneous_bound`	Spontaneous activation and inactivation of bound protein	`SpontaneousActivationBoundProteins`; `SpontaneousInactivationBoundProteins`
`transport`	Transport to nucleus	`transportTime`

3.2.5 Transcription and translation

Configuration file for transcription and translation is a file in SBML Level 2 Version 4 format (see `http://sbml.org/Main_Page`). This file should be referenced in the central configuration file in the `modelFile` field. An example of the file is provided in `$GESSAPath/Celegans/Data/WT_3cells/celegans_TT_LIP1.xml`.

File Identifier

The `notes` tag of the file must be set to "TT" value as it is specified in the `celegans_TT_LIP1.xml` file, for example. This value means that the file contains information about transcription and translation process, and the program calls corresponding function to read it and create a `SSA_Properties` object (Section 4.1.3). This object is added to the `TT_Species` property of the `CellProperties` object (see Section 4.1.3).

Species Name Agreement

The list of species involved in transcription and translation reactions are specified in the `listOfSpecies` tag. Each species is specified in `species` tag inside `listOfSpecies`. One or more instances of the `species` tag can be located in an instance of `listOfSpecies` tag. We append either "nucleus_" or "cytoplasm_" to the species name referring to compartment where the species is located. If a species can be located in both compartments, it has two different names; and the program treats these names as different species. For example, if LIN39 species can be in both cytoplasm and nucleus, it has two species in the file with "cytoplasm_LIN39" and "nucleus_LIN39" names.

If several files for transcription and translation processes are loaded in the program, the species with the same names in different files are treated as the same species and the populations of these species are synchronized across objects in the `TT_Species` property of the `CellProperties` object.

3.2.6 Updater interactions

SBML files for cell signaling and transcription/translation may use different naming conventions to refer to the same species. For proper updating of population for species that has different names in the `PPIUpdater` and `TTUpdater` objects, we need a mapping between the species in these objects. The process mapping file contains list of species names that should be updated. This is a tab-delimited text file with the two fields: `[networkToTT]` for updating the number of species from cell signaling to transcription/translation and `[TTtoNetwork]` for reversed updating. All species names should correspond to the `id` attribute of the `species` tag in cell signaling and transcription/translation xml's. For species defined in cell signaling xml, their type ("protein" for signaling proteins, "externalsignal" for external signals, etc as specified in Section 4.1.3) also should be specified in the format "name, type" (see the `updaterMapping.txt` file in the `$GESSAPath/Celegans/Data/WT_3cells` folder as an example). This file should be referenced in the central configuration file in the `TTinitFile` field.

3.2.7 Encoding biological / experimental conditions

Users may wish to run several simulations for organisms under a range of biological or experimental conditions to perform *in silico* experiments to predict the effect of these conditions on phenotypes. To explore these variations, users must define configuration files that encode these conditions as described above. In this case, users should first create a set of files for the standard wild-type organism and then create new configuration files based on the original files that encode the desired conditions. Table 3.5 describes the appropriate modifications that the user should make to the wild-type configuration files to encode some standard experimental conditions.

Table 3.5: Modifications to wild-type organism configuration files to standard biological / experimental conditions. The "file" column lists the configuration file which should be modified with the modification in the "modification" column.

Condition	File	Modification
Gain of function (GOF)	Cell signaling	Set inactive population of GOF species to 0 and active population to full species concentration in from the wild type organism. Also must set spontaneous activation rate for this species to a small number to make probability of activation one.
Loss of function (LOF)	Cell signaling	Set inactive and active populations of LOF species to 0.
Partial GOF	Cell signaling	Increase initial active population, decrease initial inactive population, and increase spontaneous deactivation rate by an amount based upon the degree of partial GOF.
Partial LOF	Cell signaling	Reduce initial inactive and active populations by an amount based upon the degree of partial LOF.
Knock in	Transcription / Translation	Increase initial population of DNA for knock in species.
Knock out	Transcription / Translation	Set population of DNA for knock out species to 0.

3.3 Model output and diagnostics

By default, the `driveHybridModel` function will create a `zip` archive with the name specified in the central configuration file (Section 3.2.1) containing the following:

- `SimulationResults` folder
  - Matlab data file (`.mat`) titled by the name specified in call to `driveHybridModel` (Section 3.1) containing `OrganismHistory` (Section 4.3) objects with average state and each state for simulated organisms. Note that by default, `driveHybridModel` will not create this file leaving this folder empty in the archive.
- `ConfigurationFiles` folder
  - Model configuration files (Section 3.2).

After running the model, the user may wish to examine the state of specific cell species to assess the cellular state and organism phenotype over the simulation period. GESSA facilitates such simulation analysis following the support functions inside `$GESSAPath/HybridModel/Functions` that act on the `OrganismHistory` (Section 4.3) objects returned by the `driveHybridModel` (Section 3.1):

`getSpeciesNumber` Returns population of specified species from specified cells at a given simulation time.

`plotSpeciesHistory` Plots population of specified species from specified cells throughout the simulation.

`speciesPerOrg` Creates a table of the population of specified species from specified cells at given simulation time(s) for multiple simulations stored in an array of `OrganismHistory` objects.

Unless otherwise specified, each of these support functions acts only on a single `OrganismHistory` object, typically on the average simulated organism or on a single simulation copy stored in elements of the array of `OrganismHistory` objects returned by `driveHybridModel`. We describe sample implementations of these functions for *C. elegans* in Section 3.4 and provide further details on these functions in Section 4.3.

3.4 Example: *C. elegans* VPC

The `$GESSAPath/Celegans` folder contains a sample application of GESSA. This example simulates development of the three central vulval precursor cells (VPCs; p5.p-p7.p) in a wild type *C. elegans* organism due to the interaction of signaling in the RAS and Notch pathways. The `CElegansDriver.m` script contains a central driver script, as described in Section 3.1. First, this script uses the `driveHybridModel` function to run GESSA for parameters specified in files (listed here in Table 3.1) that are indicated the central configuration file `$GESSAPath/Celegans/Data/WT_3cells/simulationParameters.txt` (format described in Section 3.2.1). As described in Section 3.1, calling `driveHybridModel` runs GESSA, places the simulation files in the specified zip archive (here called `output`), and returns `OrganismHistory` objects with the average state (here called `avgHistory`) and state from each simulation (here called `allOrgHistory`). As described in Section 3.2.7, this same driver script can be run with configuration files modified from the wild type as described in Table 3.6 to simulate standard mutations in *C. elegans* VPC experiments.

Table 3.6: Modifications to wild type *C. elegans* configuration files to simulate standard experimental mutations.

Condition	**File**	**Modification**
LIN-12 LOF	`network_3cells.xml`	Set `initialAmount` attribute of the `species` tag with `id` attribute "LIN12i" to "0".
LET-60 GOF	`network_3cells.xml`	Set `initialAmount` attribute of the `species` tag with `id` attribute "LET60i" to "0" and with `id` attribute "LET60a" to "100". Set the spontaneous activation rate of the `reaction` tag with `id` containing the phrase "spontaneous_activation_LET60" contained in the `listOfParameters` tag to "1".
AC Ablation	`diffusion.xml`	Set `sourceMagnitude` attribute of `signal` tag with attribute `id` of "AC_LIN3" to "0".
LIP-1 Knock out	`celegans_TT_LIP1.xml`	Set `initialAmount` attribute of `species` tag with attribute `id` of "nucleus_DNAlip1" to "0".
KSR LOF	`network_3cells.xml`	Set `initialAmount` attribute of `species` tag with `id` attribute "KSR" to "0".

In this sample driver script, we perform additional *C. elegans* specific diagnostics from support functions and classes provided with this example and demonstrate how to save the results with the specified archive.

After running GESSA, the driver script first unzips the output archive into a specified folder using the `unzip` command. The driver script then calls the `speciesPerOrg` support function (Section 4.3) to create a table of the specified species in the specified output file place in the path of the unzipped archive. Similarly, the driver script uses the `plotSpeciesHistory` function to plot the average state of proteins in the RAS pathway during the simulation. This plot may also be saved into an `eps` file for output if the user uncomments the `saveas` command. After creation of these additional diagnostic files, we recreate the output archive using the `zip` command and remove the temporary folder to which these files were output using the `rmdir` command.

We have included additional *C. elegans* specific diagnostics for release with this example, some of which are called in the driver script (e.g., `plotmRNA`). We describe these utilities in the following subsections.

3.4.1 Support functions in `$GESSAPath/Celegans/Functions`

plotmRNA

The `plotmRNA` function plots the total mRNA (top subplot) and protein (bottom subplot) population of LIN39, LAG2, EGL17, and LIP1 for a specified *C. elegans* VPC cell over the simulation period in an `OrganismHistory` object. The function is called as follows:

```
plotmRNA(orgHist, cellName)
```

Input arguments:

`orgHist` `OrganismHistory` object from which to plot the species states.

`cellName` Name of cell for which to plot the species states. Must be a single *C. elegans* VPC cell or function call will throw an error.

TF_snapshots

The `TF_snapshots` function creates a table in the output file `TF_numbers.txt` containing the total population of LIN31, CEH20-LIN39 complex, and NICD-LAG1 complex bound and unbound to the DNA in the nucleus for a specified cells in each input simulation. The function is called as follows:

```
fileName = TF_snapshots( orgHistory, cellName, startTime, endTime, timeStep)
```

Input arguments:

`orgHistory` Array of `OrganismHistory` objects containing results from GESSA simulations of *C. elegans*.

`cellName` Cell array containing names of *C. elegans* VPC cells from which to extract the species populations.

`startTime` First time at which to extract the species populations.

`endTime` Last time at which to extract the species populations.

`timeStep` Interval at which to extract the species populations. (Optional; default: every 1800 s)

Return variable:

`fileName` File name to which results were output (`TF_numbers.txt`).

3.4.2 Support classes in `$GESSAPath/Celegans/Classes`

CelegansPlotter

The `CelegansPlotter` contains several static methods which call the `plotSpeciesHistory` support function in order to plot the state of key species in all three VPC cells in the example *C. elegans* simulation.

Chapter 4

Software overview

GESSA consists of classes and support functions designed to describe organism components and simulate the multi-cellular model specified through the simulation configuration files described above (Section 3.2). In this section, we describe several of the classes in GESSA. For clarity, we focus these descriptions largely on the objects and methods within the classes that are most likely to be accessed by users for other organisms. We also note some places in these classes where developers may wish to make modifications to explore hypotheses about mechanisms underlying biological processes not modeled here. Any developers desiring further information about GESSA's structure should contact Ludmila V. Danilova `ludmila.danilova@gmail.com` or Elana J. Fertig `ejfertig@jhmi.edu`.

GESSA classes are divided into three major categories: (1) classes used to describe the organism and its components (Section 4.1), classes used to perform the simulation of modeled cellular processes (Section 4.2), and classes used for simulation results and diagnostics (Section 4.3). Figure 4.1 diagrams the interaction between these core classes. We describe this subset of classes in further detail in the following subsections.

4.1 Organism classes

4.1.1 `Organism`

An object of class `Organism` (source in `$GESSAPath/HybridModel/Classes/@Organism`) describing the properties and states of cells at a single timestep used for the simulation. `Organism` objects have the following properties with public get access:

`CellTypes` Array of `CellProperties` objects defining the set of possible cell types in the `Organism`.

`NCells` Integer containing the number of cells in the `Organism`.

`OrgCells` Array of `Cells` objects defining the cell types and states of each of the `NCells` in the `Organism`.

`CellNetwork` `NCells` by `NCells` matrix defining the topology of the `Cells` in the `Organism`.

`Progeny` `NCells` by `NCells` matrix whose ij entry is 1 if cell i is a progeny of cell j and 0 otherwise.

`CellStartTime` Numeric array of starting active time for each of the `NCells` as defined in Section 3.2.2.

`DivisionTime` Numeric array containing division time for each of the `NCells` as defined in Section 3.2.2.

These properties are set by methods defined in the `Organism` class called when reading in the simulation configuration specified in Section 3.2.

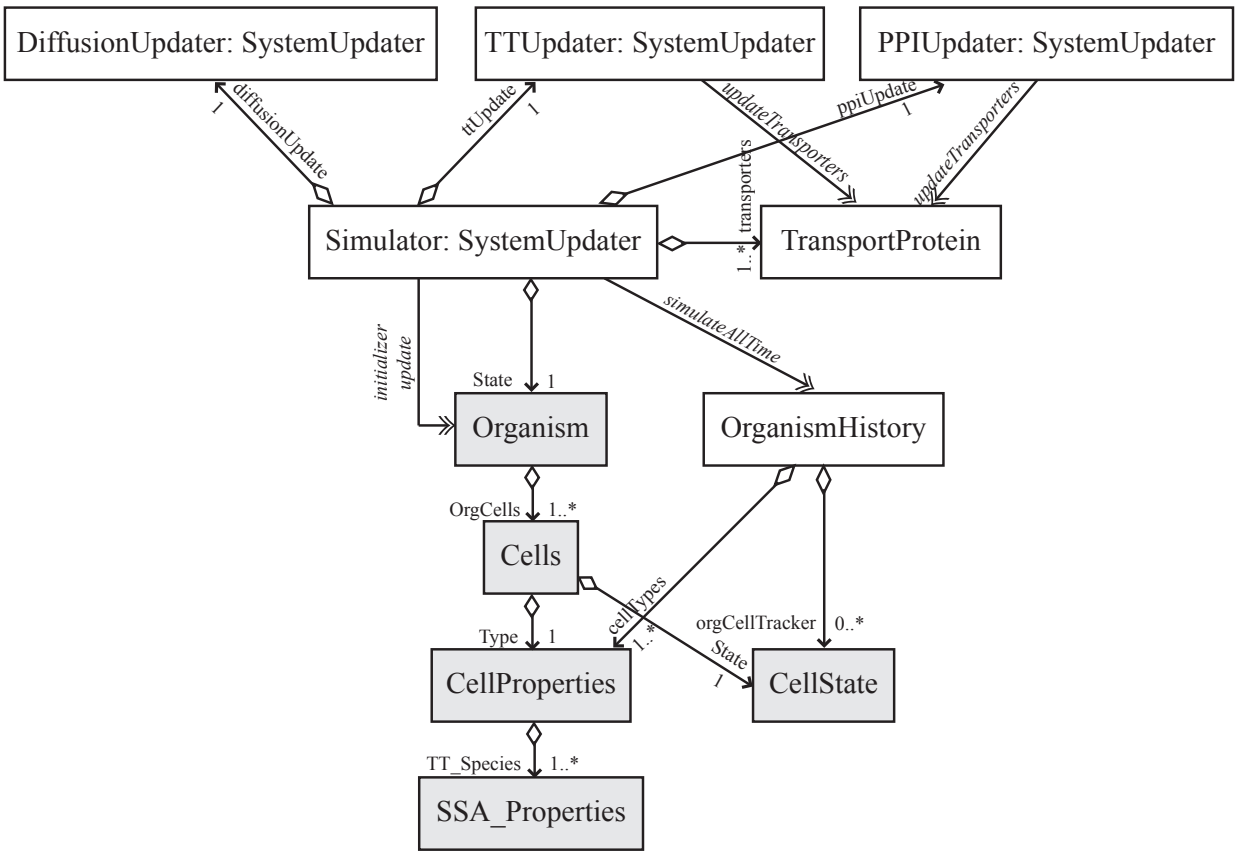

Figure 4.1: Diagram of key components of and key classes in GESSA. A single arrow with a diamond at the end denotes an object of specified class (diamond side) containing a set of object(s) with indicated number and property name of the specified class (arrow side). A double arrow indicates that the specified method (name in italics) acts on an object of the specified class (arrow side) internal to that object (if contained in the object) or as a return argument. Highlighted classes are the organism classes described in Section 4.1.

4.1.2 `Cells`

An object of class `Cells` (source in `$GESSAPath/HybridModel/Classes/@Cells`) stores the properties and state of a cell at a single timestep used for the simulation. `Cells` objects have the following properties with public get access:

`Name` Character for name of the `Cells` object.

`Type` `CellProperties` object describing cell type for the `Cells` object.

`State` `CellState` object containing the state of species for the `Cells` object.

4.1.3 `CellProperties`

An object of class `CellProperties` (source in `$GESSAPath/HybridModel/Classes/@CellProperties`) describes the species contained within each cell. `CellProperties` objects have the following properties with public get access:

`ProteinTypes` Character array of signaling proteins in the cell.

`NProteins` Number of distinct signaling proteins.

`PPI NProteins` by `NProteins` by 2 array with entry `PPI`$(:, i, j)$ indicating the rate at which proteins activate (if > 0) or repress (if < 0) the i^{th} protein unbound to a scaffold ($j = 1$) or bound to a scaffold ($j = 2$).

`canTransport NProteins` Boolean array indicating whether active copies of the proteins can be transported to the nucleus with the `TransportProtein` class.

`transportTime NProteins` numeric array providing transport rate for transportable species (negative values indicate non-transportable species).

`ExternalSignalTypes` Character array of names of external signals propagated to the cell (see `DiffusionUpdater`).

`NExternalSignals` Number of external signals propagated to the cell.

`ExternalSignals NReceptors` by `NExternalSignals` numeric matrix whose ij entry provides the rate at which the j^{th} external signal activates the i^{th} receptor.

`isExternalNeighbor NExternalSignals` Boolean array indicating whether the external signal acts on neighboring cells.

`SignalIDs` Character array of names of signal propagation reactions for any propagated external signals.

`DistanceToSource` Numeric array with distance of external signal to source for each signal propagation reaction in `SignalIDs`.

`RepressorTypes` Character array of names of external repressors available to the cell.

`NRepressors` Number of external repressors.

`Repressors NProteins` by `NRepressors` by 2 array with entry `Repressors`$(:, i, j)$ indicating the rate at which repressor i represses all proteins if they are unbound to a scaffold ($j = 1$) or bound to a scaffold ($j = 2$).

`ScaffoldTypes` Character array of names of scaffolds.

`NScaffolds` Number of scaffolds.

`Scaffolds NProteins` by `NScaffolds` by 2 array with entry `Scaffolds`$(:, i, j)$ indicates the rate at which proteins are bound to ($j = 1$) or unbound from ($j = 2$) scaffold i.

`NeighborCellTypes` Character array naming cells neighboring this cell.

`NNeighborCells` Number of neighboring cells.

`isNeighborCell` Boolean array with length equal to the number of cells in the organism indicating whether each cell neighbors this cell.

`GrowthRate` Rate at which cell moves from signal source due to organism growth (currently ignored).

`ReceptorTypes` Character array naming receptors in the cell.

`NReceptors` Number of receptors.

`Receptor2Target NProteins` by `NReceptors` by 2 array with entry `Receptor2Target`$(:, i, j)$ indicating the rate at which receptor i activates each protein if it is unbound to a scaffold ($j = 1$) or bound to a scaffold ($j = 2$).

`SpontaneousActivationReceptors NReceptors` numeric array giving rate at which receptors are spontaneously activated.

`SpontaneousActivationUnboundProteins NProteins` numeric array giving rate at which proteins not bound to scaffolds are spontaneously activated.

`SpontaneousActivationBoundProteins NProteins` numeric array giving rate at which proteins bound to scaffolds are spontaneously activated.

`SpontaneousInactivationReceptors NReceptors` numeric array giving rate at which receptors are spontaneously inactivated.

`SpontaneousInactivationUnboundProteins NProteins` numeric array giving rate at which proteins not bound to a scaffold are spontaneously inactivated.

`SpontaneousInactivationBoundProteins NProteins` numeric array giving rate at which proteins bound to a scaffold are spontaneously inactivated.

`TT_Species` Array of `SSA_Properties` objects describing properties for transcription and translation events.

In the following subsections, we describe the `SSA_Properties` class and access of the properties inside `CellProperties` objects.

`SSA_Properties`

An object of class `SSA_Properties` (source in `$GESSAPath/HybridModel/Classes/@SSA_Properties`) describes the parameters for stochastic simulation. `SSA_Properties` objects have the following properties with public get access:

`V` The volume of the compartment where simulated reactions occur. This parameter is important for bimolecular reactions.

`Names` Column vector of names of species involved in simulated process.

`Population` Column vector corresponding to `Names` vector with initial populations of species.

`ReactionRates` Row vector of reaction constants.

`StoichMatrix` Stoichiometry matrix with rows corresponding to species and columns corresponding to reactions. ij entry provides the number of molecules of i^{th} species that take part in the j^{th} reaction. The positive value corresponds to a reactant of the reaction; the negative value correspond to a product of the reaction.

The values of properties correspond to values in a transcription/translation SBML file described in Section 3.2.5.

Accessing `CellProperties` properties

Although the properties of species in a `CellProperties` object has direct get access, we have developed several functions designed to access these objects in a controlled way. In particular, special care must be taken in accessing particular species defined in a `CellProperties` object as the indexing used in these arrays also defines the indexing used to store their population in `CellState` objects. We, therefore, recommend use of the following public methods an object `cellType` of the `CellProperties` class:

`cellType.getNState(stateName, ttName)` Returns the number of species of type `stateName`. To reflect properties in the cell, `stateName` should be one of the following strings: "protein", "externalsignal", "repressor", "scaffold", "receptor", "tt_species". The number of elements in the transcription and translation process depends on the specific process. We select this process by specifying the name of one of the species types `ttName` stored in the `SSA_Properties` object.

`cellType.getStateNames(stateName)` Returns the list of species of type `stateName` as described above. If "tt_species" is selected, this function will return all of the states stored inside all of the transcription and translation processes.

4.1.4 CellState

An object of class `CellState` (source in `$GESSAPath/HybridModel/Classes/@CellState`) stores the population of the species defined in the `CellProperties` object in `Cells` at a single simulation time. `CellState` objects have the following properties with public get access:

`Proteins` Population of signaling proteins.

`ActiveProteins` Population of active signaling proteins.

`ExternalSignals` Population of external signals.

`Repressors` Population of repressors.

`Scaffolds` Population of scaffolds.

`Receptors` Population of receptors.

`ActiveReceptors` Population of active receptors.

`TT_Species` Population of species involved in transcription and translation. Indexes match those of appended array obtained from call to `cellType.getStateNames(''tt_species'')`.

`NeighborSignals` Population of external signals acting on this cell from neighboring cells received by this cell.

`Active` Boolean indicating if the cell is currently active (within lifetime described in Section 3.2.2).

Unless otherwise specified, all of the properties above are indexed by the same index used to describe that species in a corresponding `CellProperties` object.

As described for `CellProperties`, although publicly get accessible, we recommend accessing properties through public support methods. For example, the population of any species type for an object `state` of class `CellState` can be accessed as follows

```
outState = state.getState(stateName, index),
```

where

`stateName` Name of species type to return. To reflect properties in the cell, `stateName` should be one of the following strings: "protein", "active protein", "externalsignal", "neighbor signal", "repressor", "scaffold", "receptor", "active receptor", or "tt_species".

`index` If "tt_species" is selected for `stateName`, the index of the transcription and translation process of interest should be specified. This index should be selected with care prior to calling the `getState` method based on the array of `SSA_Properties` objects in `CellProperties`.

If the user wishes to access the population of a `CellState` object in an `OrganismHistory` object for diagnostics, we recommend using the `getSpeciesNumber` support function to ensure consistency in referencing indexes from corresponding `CellProperties` and `CellState` objects (see Section 4.3).

The `CellProperties` class has an additional function, `divideCells`, which is used to facilitate cell division inside the central GESSA `Simulator` class. This function divides cells by assuming that its contents are matched for transcription and translation species and halved for other protein types. Users should modify this function if they wish to model alternative division processes.

4.2 Simulation classes

4.2.1 SystemUpdater

The SystemUpdater class (source in $GESSAPath/HybridModel/Classes/@SystemUpdater) is an abstract parent class for all the processes used to evolve an organism in GESSA. This parent class requires that all of its subclasses define an initializer method to configure the update process and its properties from the Organism object created by the configuration files (Section 3.2) and specified process-specific input files. For a SystemUpdater object called updater, this method is invoked as follows:

```
[state, time] = updater.initializer(organism, initFile, initTime),
```

where the input arguments are

organism An Organism object defining the organism to simulate.

initFile File(s) used to configure the SystemUpdater object.

initTime Initial simulation time for this updater process.

and output arguments are

state Updated description of the system state (usually in the form of an Organism object or array of Cells objects) based on the configuration performed in this method.

time Updated estimate of the current system time based on the configuration.

Once initialized, running of the processes controlled by subclasses of the SystemUpdater class are performed through the required update method as follows:

```
[newState, delTReal] = updater.update(state, previousTime, delT),
```

where the input arguments are

state The current state of the Organism object, typically as the Organism object itself or as an array of Cells or CellState objects.

previousTime The time at the start of this update process.

delT The expected timestep of this update process.

and output arguments are

newState The state of the Organism object after the update process. Typically of the same classes as the state input argument.

delTReal The true timestep of this update process.

If the update process also controls passing of species between the nucleus and cytoplasm as in TTUpdater and PPIUpdater, the update method becomes a dummy method inside the subclasses of SystemUpdater. This update method is then replaced with a new method called updateTransport which has an additional input argument of an array of TransportProtein objects called transporters as follows:

```
[newState, delTReal] = updater.updateTransport(state, previousTime, delT, transporters)
```

Due to present limitations in the Matlab class structure, this modification is manually coded into the corresponding SystemUpdater classes rather than overriding the standard update method.

4.2.2 Simulator

The driveHybridModel function (Section 3.1) creates an object of the Simulator class (source in $GESSAPath/HybridModel/Classes/@Simulator) to facilitate in running the GESSA simulation in the simulateOrganisms function (source in $GESSAPath/HybridModel/Functions/simulateOrganisms.m). As this class evolves the state of the organism, it is a subclass of the generic updater class SystemUpdater. The update method in the Simulator object required by the SystemUpdater updates the state of the Organism state in the Simulator object at each timestep based on the SystemUpdater processes called in the simulateAllTime method. This simulateAllTime method runs modules for each of the update processes stored as private access SystemUpdater and TransportProtein objects as specified by the configuration files (Section 3.2). The Simulator object controls cell division module internally with the division method. While running these modules, the simulateAllTime method also regulates their relative timing throughtout the simulation time $t0$ to tf with the scheduler method as shown in the flow chart in Figure 4.2. We note that both the cell signaling reaction PPIUpdater and transcription translation TTUpdater objects have access to the TransportProtein objects in the Simulator object to facilitate their shuttling of species between the cytoplasm and nucleus.

If the user wishes to run their own module for a biological process, they would include a user-developed SystemUpdater object inside the simulator. They should ensure that this object is properly initialized by the simulateOrganisms function and Simulator initializer method with its own configuration files specified in the central input file (Section 3.2.1). Likewise, they should modify the simulateAllTimes method to invoke this module and track its timing appropriately as demonstrated in Figure 4.2. If this class will distribute species between the cytoplasm and nucleus, it should have access to the TransportProtein objects in the Simulator object similarly to the PPIUPdater and TTUpdater objects. Finally, GESSA assumes that each process occurring over a given time step only makes moderate changes to the organism state. Therefore, the Simulator object may safely update the system state resulting from all of the processes by adding the corresponding change in state at the appropriate simulation time (see Figure 4.2), with an additional check to ensure that no species population ever goes negative due to interactions of these changes. We recommend the user be careful to likewise ensure small changes in the system state from their process to avoid introducing falsely negative populations when it is implemented.

4.2.3 DiffusionUpdater

A DiffusionUpdater object (source in $GESSAPath/HybridModel/Classes/@DiffusionUpdater) propagates external signals to the cells in the organism as specified in the configuration file description in Section 3.2.3. The private properties in this class are used to store the information about the source and propagation process (typically a diffusion process) for each signal from that configuration file with the initializer routine. Currently, this initializer checks to ensure that the configuration file conforms to the standard that there is only one source for each signal, which will be relaxed in future releases of GESSA. Once initialized, the update method in the DiffusionUpdater object loops over each source and propagates the signal according to the propagation model specified in the configuration file (see Section 3.2.3) and stored in the diffusionType property of this class. Currently, the DiffusionUpdater recognizes the "3DPoint" propagation process, which diffuses the signal in three dimensions from a point source which is either sustained throughout the simulation period (in method diffuseSustain3D) or instantaneous (in method diffuse3D), as specified by the start and end time of the source saved in sourceInitialTime and sourceEndTime properties. The user may add additional signal propagation processes, including more complex diffusion models, by modifying the central switch in the update process to refer to a function that implements that process when named as the type of signal propagation in the diffusionType property.

4.2.4 PPIUpdater

A PPIUpdater object (source in $GESSAPath/HybridModel/Classes/@PPIUpdater) evolves the state of the organism due to cell signaling processes with a PPBN graphical model. Specifically, it propagates the external signal distributed by the DiffusionUpdater through the signaling network to alter the activation status of receptors, signaling proteins, and finally transcription factors. At each step, the scaffold reaction is also implemented to bind and unbind proteins from the scaffolds and incorporate that binding information in the signal propagation analogously, but more directly and faster than for unbound proteins. These signal propagation processes are implemented in the updateTransporters method, which calls several methods in the PPIUpdater object to modularize the numerous reactions involved in the cell signaling reactions.

The model in the PPIUpdater considers that any active transcription factor (canTransport property in CellProperties) may be transported to the nucleus if it remains active until the transport time (expected value given in the transportTime property in CellProperties). To achieve this, any newly activated transcription factor is added to the list of transported proteins in the TransportProtein object using the addTransportProtein method in that class. After calling this method, active transcription factors are removed from the population as stored in CellState to avoid double counting any species in transition. At the simulation time, the PPIUpdater also extracts the list of transcription factors marked for transport with the getTransportList method of the TransportProtein, where the logicalDirection argument is set to 1 to extract only those copies of the transcription factors that have not yet reached the transport time. The PPIUpdater then evolves the state of these transcription factors according to the signaling model. If any of the candidate transported transcription factors is inactivated during this process, it is removed from the list of transported proteins in the TransportProtein object using the inactivateProt method of TransportProtein and returning these transcription factors to the population in the CellState object.

4.2.5 TTUpdater

A TTUpdater object (source in $GESSAPath/HybridModel/Classes/@TTUpdater) uses the Stochastic Simulation Algorithm (SSA) to perform the transcription and translation processes that create proteins in the organism in response to transcription factors activated in the PPIUpdater. In order to work with active transcription factors in the nucleus, the TTUpdater must first transport active transcription factors to the nucleus. Similar to the PPIUpdater, the TTUpdater uses the transportProteins method in the TransportProtein object to transport only those copies of the transcription factor that have remained active until the transport time. Once transported to the nucleus, the TTUpdater implements SSA to perform the transcription and translation process based on the available transcription factors and reactions specified in the configuration files described in Section 3.2.5. Upon completion of the transcription and translation process, the TTUpdater object moves any inactivated transcription factors, created signaling proteins, or ligands from the nucleus into the cytoplasm after a characteristic transport time to make them available to the other simulation processes. Because this transport back into the cytoplasm does not require further modification of the species state over the transport period, the TTUpdater object performs it directly without use of the TransportProtein object. If a user defined process required more complex balancing of species transport into the cytoplasm, that process could use the methods in the TransportProtein object as described above and in the following section to facilitate that transport.

An additional complication in properly moving species between the nucleus and cytoplasm in the TTUpdater arises from the different naming conventions used in the TTUpdater and PPIUpdater. I.e., the TTUpdater utilizes primarily species defined in the TT_species property of CellProperties to facilitate implementation of SSA, which the PPIUpdater uses other directly named properties in CellProperties. To avoid any conflicts, the TTUpdater object contains the private property mapping defined in the configuration file described in Section 3.2.6 to appropriately name equivalent species in the processes. The TTUpdater when performing all of the above described transport processes between the cytoplasm and nucleus to ensure proper naming when retrieved from or returned to the cytoplasm. If any user-defined process similarly uses a different naming convention, we recommend likewise specifying the mapping between names of the processes in the configuration file as described in Section 3.2.6.

4.2.6 TransportProtein

As described in the PPIUpdater and TTUpdater classes, TransportProtein objects (source in $GESSAPath/HybridModel/Classes/@TransportProtein) facilitate species propagation between the nucleus and cytoplasm. Transport using these objects is particularly important when that transport process may be disrupted due to an alternative reaction such as transcription factor deactivation in the PPIUpdater. TransportProtein objects contain the following properties

transportProteinTypes Character array with names of species that can be transported with the TransportProtein object.

expectedTransportTime Numeric array with expected transport times corresponding to each of the species in transportProteinTypes.

proteinNames Cell array containing names of species marked for transport.

activationTime Numeric array of times at which species in proteinNames are placed into the TransportProtein object.

transportTime Numeric array of times at which species in proteinNames would be transported.

Currently, the transportProteinTypes and expectedTransportTime properties are set in the TransportProtein constructor using the canTransport, transportTime, and ProteinTypes properties of the input CellProperties object. As a result, the TransportProtein class is currently limited to transporting transcription factors from the cytoplasm to nucleus. However, this requirement could be relaxed easily by modifying the constructor to initialize the transportProteinTypes and expectedTransportTime properties from other input arguments. This modification would also enable the TransportProtein class to facilitate transport from the nucleus to the cytoplasm and between other user defined compartments. In this case, we recommend the user create other copies of the TransportProtein objects in the Simulator to avoid conflicts in referring to transport of the same species across different compartments.

Although the properties in TransportProtein objects have public read access, we strongly caution the user against such direct access to ensure that methods access only those species prior to the transport time if modifying their state or after the transport time if performing the transport process. A transProt object of class TransportProtein, therefore, has several public support methods that we recommend accessing instead as described in the following subsections. It is important to note that these methods **will not** update the species population stored in CellState objects in the organism. Therefore, the user must take care to properly increment or decrement these populations when extracting or adding a species to the TransportProtein list as described in the PPIUpdater and TTUpdater classes above.

```
[numProteins, protList] = transProt.clearTransporter(protsToClear)
```

The clearTransporter method removes all specified proteins from the list of transported proteins, without regard for system time.

Input arguments:

protsToClear Cell array containing names of species to remove from the transporter. (Optional; default all proteins).

Return variables:

numProteins Numeric array containing counts of each protein type which were removed.

protList Cell array containing names of proteins types removed.

`transProt.addTransportProtein(names, times)`

The `addTransportProtein` method marks a protein for transport in the `TransportProtein` object. This method will also determine and store the transport time for that protein using the private `getTransportTime` method in the `TransportProtein` class.

Input arguments:

`names` Cell array containing names of proteins that are transported. Note, this array will contain duplicate names if multiple copies of the same protein type are transported.

`times` Numeric array with corresponding time at which each protein in the `names` list is first marked for transport.

`numRemoved = transProt.inactivateProt(currentTime, name, actTime, transTime)`

The `inactivateProt` method will remove any copies of the species marked for transport specified by the name, activation time, and transport time from the transporter list in the `TransportProtein` object. If there are multiple proteins with the same name, activation time, and transport time, the `inactiveProt` function will only remove a single copy of that protein. Multiple copies can be specified by repeats in the input arguments (see below). Note, the method will only perform this removal for proteins with transport times before the current time to avoid removing a protein which has already been transported.

Input arguments:

`currentTime` Time at which species are to be removed.

`name` Cell array containing name(s) of proteins to be removed. Note, this array will contain duplicate names if multiple copies of the same species type are transported.

`actTime` Numeric array containing the time(s) at which the proteins to be removed were marked for transport corresponding to species names specified in `name`. Note, this array will contain duplicate entries if removing multiple species marked for activation at the same time.

`transTime` Numeric array containing the transport time(s) of the species to be removed corresponding to the species names specified in `name`. Note, this array will contain duplicate entries if removing multiple species transported at the same time. Moreover, specification of any species with `transTime` after the `currentTime` will be disregarded, and may cause this function to throw an error message.

Return variable:

`numRemoved` Total number of species removed from the `TransportProtein` object.

`numTransport = transProt.transportProteins(currentTime, name)`

The `transportProteins` method removes any specified species with transport times prior to the current time from the list of proteins in the `TransportProtein` object. The process calling this method **must** add the removed species to the population in the appropriate compartment, or they are lost for future simulations.

Input arguments:

`currentTime` Time at which to perform transport.

`name` Name of species to transport.

Return variable:

`numTransport` Number of copies of species of type `name` with transport time before the current time which were stored in the `TransportProtein` object and removed from the object by this method.

```
[nameList, actTime, transTime] = transProt.getTransportList(name, time, logicalDirection)
```

The `getTransportList` gets a list of all species in the `TransportProtein` object available for removal or transport based upon timing specified.

Input arguments:

`name` Name of species to query.

`time` Time at which to query.

`logicalDirection` 0 to find species with transport times equal to `time`, positive for transport times greater than `time`, and negative for transport times less than `time`.

Return variables:

`nameList` Cell array containing the names of the species satisfying the above described query for the `TransportProtein` object.

`actTime` Numeric array containing the times at which species were marked for transport for species satisfying the above described query for the `TransportProtein` object.

`transTime` Numeric array containing the transport times for species satisfying the above described query.

4.3 Simulation result classes and support functions

4.3.1 `OrganismHistory`

Simulation of GESSA creates an object of class `OrganismHistory` (source in `$GESSAPath/HybridModel/Classes/@OrganismHistory`). The resulting `OrganismHistory` objects store the properties of cells defined in the simulation as `CellProperties` objects and their corresponding state at each simulation time in an array of `CellState` objects. Specifically, the following properties have public get access:

`timeTracker` Numeric array of stored simulation times.

`orgCellTracker` Array of `CellState` objects storing simulation results for cells (rows) at each simulation time (columns).

`cellNames` Character array of names of stored cells.

`cellTypes` Array of `CellProperties` objects describing the stored cells.

`startActiveTime` Numeric array of first active time for each cell (Section 3.2.2).

`endActiveTime` Numeric array of last active time for each cell (Section 3.2.2).

The `OrganismHistory` class contains public methods to facilitate storage of the simulation results and to extract the system state for later diagnostics. We recommend that users access these latter methods (including notably `getState`, `sumSetTimes`, `times`) as described below for creating organism-specific diagnostics.

`getState`

The `getState` method returns the simulation history of a given cell and optionally for a given species type and name if specified from the `OrganismHistory` object. Call to `getState` method from an `OrganismHistory` object called `orgHistory` is as follows:

```
[state, time] = orgHistory.getState(cellName, speciesType, speciesName)
```

Input arguments:

`cellName` Name of cell(s). If specific species are requested in the following arguments, this may contain only the name of a single cell.

`speciesType` Type of species (see `CellProperties`) for which state will be returned. (OPTIONAL; Default: all species types from specified cell(s).)

`speciesName` Name of species (defined in configuration files as described in Section 3.2) for which state will be returned. (OPTIONAL; Requires specification of `speciesType`; Default: all species of type `speciesType`).

Return variables:

`state` Array containing state of selected cell(s) and species (if specified) (rows) for each simulation time (columns). The array is of `CellState` objects if only the `cellName` argument is specified. Otherwise it is a numeric array.

`time` Numeric array containing simulation times of each column in the state array.

`sumSetTimes`

The `sumSetTimes` method returns the sum of the states of the `OrganismHistory` object (`orgHist1` to the `orgHist2` `OrganismHistory` object at a set of specified timesteps (`timeList`). This function can be used in combination with the `times` function (overloading the multiplication operator) to compute the average state in these two histories. If the state of either `orgHist1` or `orgHist2` is not defined at a time in `timeList`, the state is interpolated to that time. Any times before the first time in either `orgHist1` or `orgHist2` or after the last time in either object is excluded. This function is called as follows:

```
sumHist = orgHist1.sumSetTimes(orgHist2, timeList)
```

Input arguments:

`orgHist2` `OrganismHistory` object to which to add to the `orgHist1` `OrganismHistory` object.

`timeList` Numeric array with time steps at which sum of states in `orgHist1` and `orgHist2` will be computed. (OPTIONAL; Default: all timesteps in `orgHist1` and `orgHist2` objects.)

Return variables:

`state` Array containing state of selected cell(s) and species (if specified) (rows) for each simulation time (columns). The array is of `CellState` objects if only the `cellName` argument is specified. Otherwise it is a numeric array.

`time` Numeric array containing simulation times of each column in the state array.

`times` **(Overloading the multiplication operator)**

The `times` function is defined to overload the multiplication operator to define scalar multiplication with the states stored in an `OrganismHistory` object. It is called automatically with either of the following commands for multiplication of the `orgHist` `OrganismHistory` object with scalar `c` as follows:

```
c*orgHist
orgHist*c
```

4.3.2 getSpeciesNumber

The `getSpeciesNumber` function (source in `$GESSAPath/HybridModel/Functions/getSpeciesNumber`) returns the population of a specified species in given cell(s) at a specified simulation time from an `OrganismHistory` object. The function is called as follows:

```
nMol = getSpeciesNumber(orgHist, speciesName, speciesType, cellName, time)
```

Input arguments:

`orgHist OrganismHistory` object from which to extract the species state.

`speciesName` Name of species (defined in configuration files as described in Section 3.2) for which state will be returned.

`speciesType` Type of species (see `CellProperties`) for which state will be returned.

`cellName` Cell array containing strings with names of cell(s) for which to extract the species population.

`time` Time at which to return the species state. If not in the `timeTracker` list stored in `orgHist`, `time` refers to the closest time on `timeTracker` to the specified time.

Return variables:

`nMol` Numeric array with population of specified species at a specified time for cells in `cellName` (array).

4.3.3 plotSpeciesHistory

The `plotSpeciesHistory` function (source in `$GESSAPath/HybridModel/Functions/plotSpeciesHistory`) plots the population of a specified species in specified cell(s) over the simulation period in an `OrganismHistory` object. The function is called as follows:

```
plotSpeciesHistory(orgHist, speciesType, speciesName, plotSymbols, inHours, ...
                   showLegend, plotColors, plotCells)
```

Input arguments:

`orgHist OrganismHistory` object from which to plot the species state.

`speciesType` Type of species (see `CellProperties`) for which state will be plotted.

`speciesName` Array containing name(s) of species (defined in configuration files as described in Section 3.2) for which state will be plotted.

`plotSymbols` Cell array specifying the plot colors/symbols used for each species specified in `speciesName`. To use only cell specific symbols, specify as `' '`.

`inHours Boolean` to scale time axis in hours if true and in seconds in false.

`showLegend Boolean` to specify whether or not to include the plot legend.

`plotColors` Cell array specifying the plot colors/symbols used for each cell. To use only species specific symbols, specify as `' '`.

`plotCells Boolean` array specifying which of the cells contained in the `OrganismHistory` object should be plotted. (Optional; Default: all cells)

4.3.4 speciesPerOrg

The `speciesPerOrg` function (source in `$GESSAPath/HybridModel/Functions/speciesPerOrg`) outputs the population of specified species in specified cell(s) in an `OrganismHistory` object over the specified time period. The function is called as follows:

```
speciesPerOrg(orgHist, cellName, speciesName, speciesType, startTime, ...
              endTime, timeStep, fileName)
```

Input arguments:

`orgHist` Array of `OrganismHistory` objects containing results from multiple simulations of the same organism.

`cellName` Cell array containing the name(s) of cells for which to output species populations.

`speciesName` Cell array containing name(s) of species whose populations should be output.

`speciesType` Cell array of same length as `speciesName` containing types of each specified species (as described in the `CellProperties` class).

`startTime` Time at which to start output.

`endTime` Time at which to end the output.

`timeStep` Length of time between outputs. (Optional; default: 100s).

`fileName` Name of file to which to output the results. (Optional; Default: `speciesTable.txt`)

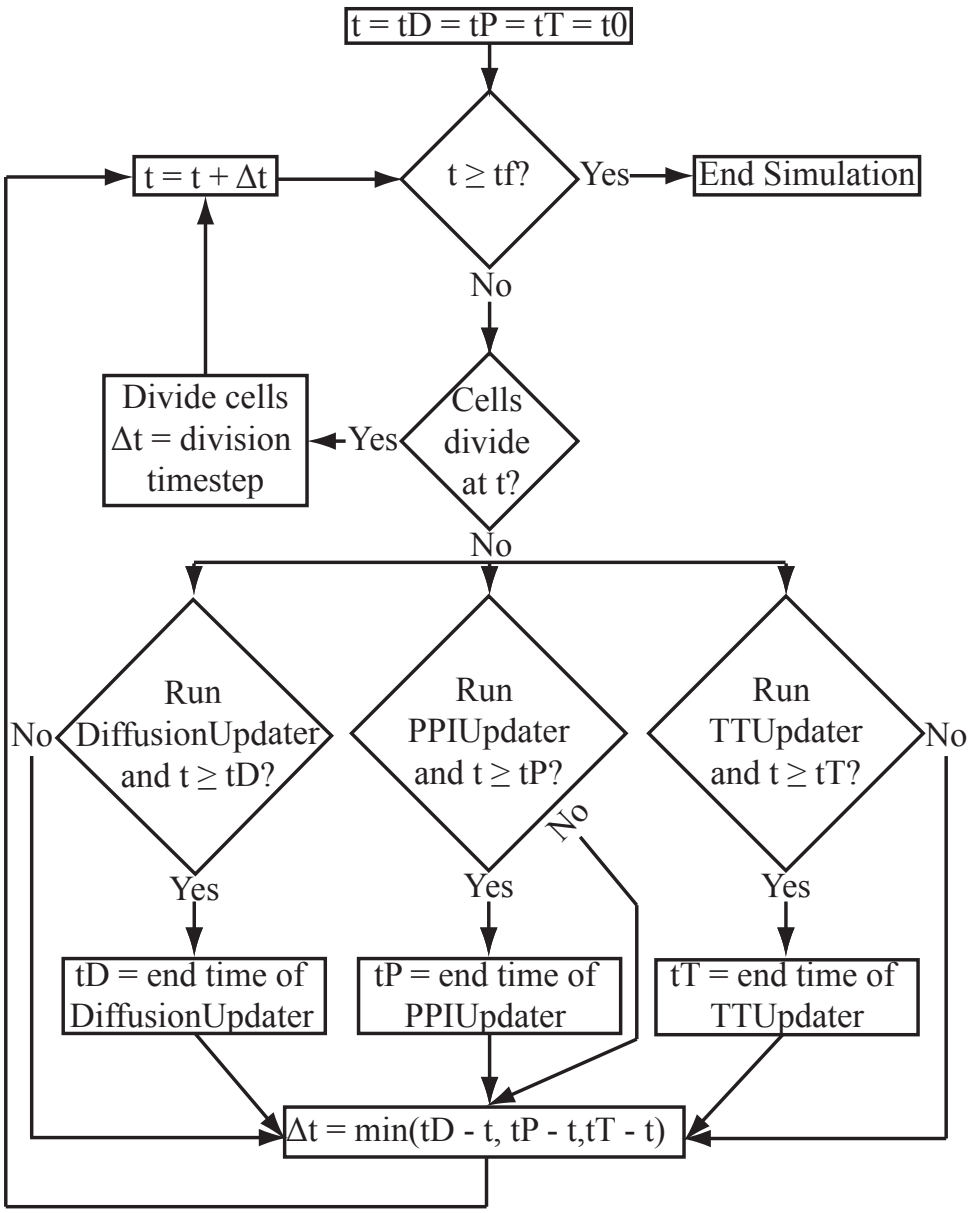

Figure 4.2: Flow chart demonstrating regulation of relative timing of simulation processes at time from simulation start time $t0$ to simulation end time tf by `simulateAllTimes` method in the `Simulator` objects. Here, t is the current simulation time, tD the time for the next external signal propagation reaction (by the `DiffusionUpdater` object), tP the time for the next cell signaling reaction (by the `PPIUpdater` object), and tT the time for the next transcription translation reaction (by the `TTUpdater` object).

Chapter 5

Feedback

Please send feedback to Ludmila Danilova `ludmila.danilova@gmail.com` or Elana Fertig `ejfertig@jhmi.edu`.

If you want to send a bug report, it must be reproducible. Send the configuration files, describe what you think should happen, and what did happen.

Chapter 6

Acknowledgments

Thanks to paper co-authors Alexander V. Favorov and Michael F. Ochs for advice in developing and testing the GESSA software. The authors of this work were funded by the Maryland Cancer Restitution Fund (MCRF) and NIH-NLM (LM009382 and LM008932).