USER GUIDE

A Whole-Cell Computational Model Predicts Phenotype from Genotype

Jonathan R. Karr[1,*], Jayodita C. Sanghvi[2,*], Derek N. Macklin[2], Miriam V. Gutschow[2], Jared M. Jacobs[2], Ben Bolival[2], Nacyra Assad-Garcia[3], John I. Glass[3] & Markus W. Covert[2†]

[1]Graduate Program in Biophysics, Stanford University, Stanford CA 94305.
[2]Department of Bioengineering, Stanford University, Stanford CA 94305.
[3]J. Craig Venter Institute, Rockville MD 20850.

[*]These authors contributed equally to this work. [†]Correspondence and requests for materials should be addressed to M.W.C (mcovert@stanford.edu).

The model reported in the accompanying manuscript accounts for 28 essential cellular processes, represents the function of 401 genes, includes over 1,900 quantitative parameters, and is based on over 900 primary research articles, reviews, books, and databases. This document provides brief instructions on how to install and run the whole-cell knowledge base and model, including how to run and analyze simulations, how to add and modify state variables and process submodels, how to fit the model, and how to computationally validate the model. The whole-cell model source code is freely available at SimTK: simtk.org/home/wholecell. Please contact the authors with any questions about the whole-cell model software. See simtk.org/home/wholecell for updated contact information.

Contents

Chapter 1

Installation

Thank you for your interest in the *M. genitalium* whole-cell model. The following sections provide detailed instructions on how to install the whole-cell model and knowledge base software. Please contact the authors with any questions about the whole-cell model software. See simtk.org/home/wholecell for updated contact information.

1.1 Whole-cell model installation

Required software

The whole-cell model software requires only MATLAB (version R2009b or newer) and can be used on any platform. All other required software is included in the whole-cell model distribution.

The whole-cell software distribution also includes code to run the whole-cell model on a compute cluster. This feature requires several additional pieces of software:

- Apache web server
- Maui cluster scheduler
- MySQL database
- Perl programming language
- PHP programming language
- Torque resource manager

We recommend using a pre-configured compute cluster toolkit which includes all of the software packages listed above. In particular, we recommend the Rocks clustering toolkit.

Installation instructions

Please follow these six steps to install the whole-cell model software:

1. Obtain the whole-cell code from SimTK: simtk.org/home/wholecell
2. Extract the code
3. Open MATLAB (version R2009b or newer is required)
4. Change to the `<whole-cell root>/simulation` directory
5. Run `install.m` script to configure the whole-cell model software
6. Follow the on-screen instructions
7. Optionally, to configure the software for use with a whole-cell knowledge base and cluster type "y" when prompted
   (a) Enter the host name, schema, user name, and password for your MySQL knowledge base server
   (b) Enter a file path where you wish simulated dynamics to be stored

1.2 Knowledge base installation

Required software

The whole-cell knowledge base requires the three software packages listed below. All other required software is included in the whole-cell knowledge base distribution.

- Apache web server
- MySQL database
- PHP programming language

Installation instructions

Please follow these six steps to install the whole-cell knowledge base software:

1. Obtain the whole-cell code from SimTK, simtk.org/home/wholecell
2. Extract the code
3. Change to `<whole-cell root>` directory
4. Run `install.php` script to configure the whole-cell model knowledge base including creating a new MySQL database, populating the database with the content of the *M. genitalium* knowledge base, configuring the knowledge base web viewer, and optionally configuring the whole-cell model code for use with a Linux cluster.
5. Follow the on-screen instructions:
   (a) Enter the host name and root user and password of your MySQL server
   (b) Enter a schema name for the whole-cell database
   (c) Enter a name and password for a new database user with limited privileges for only the whole-cell database
   (d) Enter a name and password for a new knowledge base user. This user name and password is necessary to edit the knowledge base using the web interface.
   (e) Optionally, to setup the whole-cell model code for use with a Linux cluster, enter the configuration of your cluster as prompted.
6. Visit the knowledge base web interface: `<yourserver>/knowledgebase`

Chapter 2

Running simulations

Thanks for your interest in the *M. genitalium* whole-cell model. This chapter provides a brief introduction on how to run whole-cell simulations, including how to override the default parameter values and store simulated cellular dynamics to disk. Ch. 1 provides instructions to install the whole-cell model software. Ch. 3 discusses how to analyze simulated cellular dynamics. Please see the accompany supplementary text for more information about the implementation of the whole-cell model. All of the example code in this document is also contained in the file `simulation/userGuide.m`.

2.1 Running a simulation with default parameter values

To run your first whole-cell simulation enter the commands listed in Box 2.1 below. *Note:* ***`setWarnings`*** *and* ***`setPath`*** *must be set at the beginning of each MATLAB session. Note also: it may take a day or more for each simulation to complete.*

Box 2.1 | Simulating a cell using the default parameter values.

```
%1. Supress warnings, add whole-cell code to MATLAB path. These functions
%   only need to called once at the beginning of each MATLAB session.
setWarnings();
setPath();

%2. Run simulation
runSimulation();
```

Summaries of the simulated dynamics will be printed to the command window and to a summary figure as illustrated in Box 2.2 and Fig. 2.1 below.

Box 2.2 | Sample simulation summary output.

```
                          Metabolites                                        ...
                          ============================================== ...
Time  RT  Mass Growth     ATP       ADP       AMP       NTPs      Min NTP  ...
===== === ==== ========== ======== ======== ======== ======== ========= ...
    0   0 1.00 2.119e-005    36234      3623      1449     72468      0 CTP ...
    1   3 1.00 2.119e-005    39070       101      2137     77357      4 CTP ...
    2   2 1.00 2.119e-005    40428        98       797     80302    345 CTP ...
    3   2 1.00 2.119e-005    39805       119       890     80945    812 UTP ...
    4   2 1.00 2.119e-005    39225       106       934     79279    201 UTP ...
    5   1 1.00 2.119e-005    39138       109       837     78871     26 UTP ...
```

```
   6    2 1.00 2.119e-005     39184       106       776     79067     21 UTP ...
   7    2 1.00 2.119e-005     39240        97       742     79113      6 UTP ...
   8    2 1.00 2.119e-005     39298        85       697     79312     33 UTP ...
   9    2 1.00 2.119e-005     39301        93       683     79133      6 UTP ...
  10    1 1.00 2.119e-005     39275        93       713     79170     24 UTP ...
```

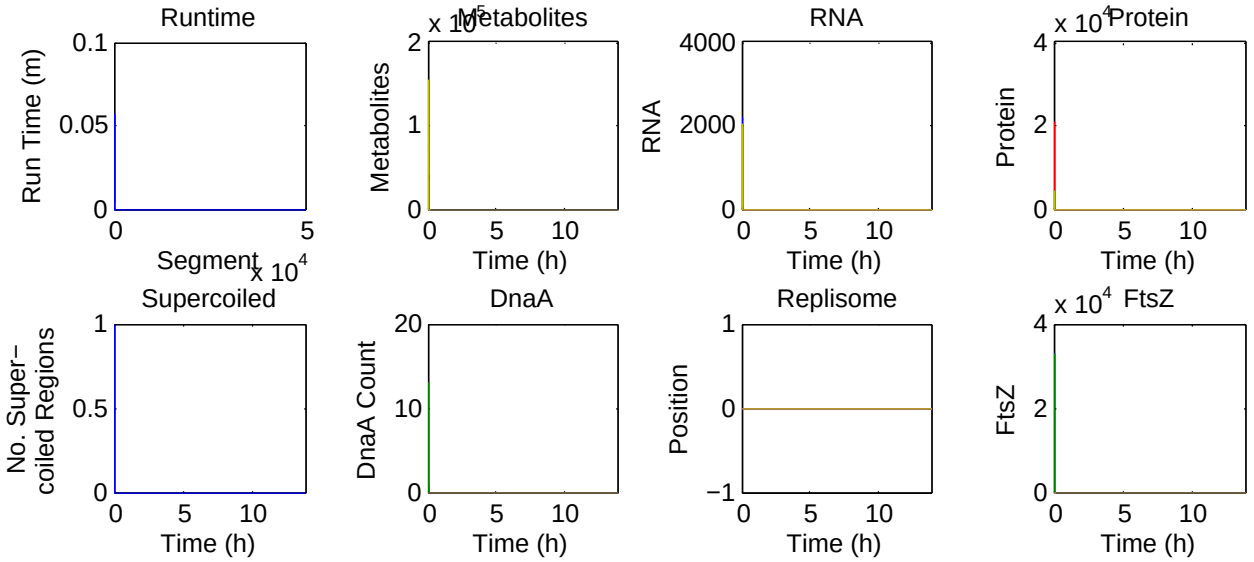

Figure 2.1 | Sample simulation summary output.

2.2 Running a simulation with modified parameter values

Now that you've run your first whole-cell simulation using the default parameter values, its time to explore the phenotypic consequences of alternative genotypes. There are two ways to override the default parameter values.

Setting parameter values using the online configurator

First, you can use the user-friendly configurator at wholecell.stanford.edu/simulation/runSimulations.php and the code listed in Box 2.3 to run a simulation with alternative parameter values. Box 2.4 provides a sample XML simulation configuration file generated by the online configurator. *Note: you will need to place the XML file you will generate at line 29 of Box 2.4 in the directory represented by the* `simDir` *variable.*

Box 2.3 | Setting parameter values using the online configurator and running a simulation.

```
%1. Supress warnings, add whole-cell code to MATLAB path. These functions
%   only need to called once at the beginning of each MATLAB session.
setWarnings();
setPath();

%2. Import classes
import edu.stanford.covert.cell.sim.util.SimulationDiskUtil;

%3. Select
%   a. simulation batch output directory and
%   b. simulation output directory
simBatch = datestr(now, 'yyyy_mm_dd_HH_MM_SS');
simIdx = 1;
```

```
simBatchDir = [SimulationDiskUtil.getBaseDir() filesep simBatch];
simDir = [SimulationDiskUtil.getBaseDir() filesep simBatch filesep ...
    num2str(simIdx)];

%4. Create simulation batch and simulation output directories
if ¬isdir(simBatchDir)
    mkdir(simBatchDir); %create simulation batch output directory
end
if ¬isdir(simDir)
    mkdir(simDir); %create simulation output directory
end

%5. Generate XML description of desired parameter values from
%   http://wholecell.stanford.edu/simulation/runSimulations.php, save XML
%   file to <simDir>/conditions.xml
%   - Select a short simulation length that is a multiple of 100, eg. 100

%6. Run simulation and save simulated dynamics to disk
runSimulation(simDir);
```

Box 2.4 | Sample parameter XML file generated by the online configurator.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
Condition set autogenerated by ...
    https://wholecell.stanford.edu/simulation/runSimulations.php at Mon, 13 Feb ...
    2012 19:57:43 GMT.
-->
<conditions
    xmlns="http://covertlab.stanford.edu"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://covertlab.stanford.edu runSimulations.xsd">
    <firstName>Jonathan</firstName>
    <lastName>Karr</lastName>
    <userName>jkarr</userName>
    <email>jkarr@stanford.edu</email>
    <affiliation>Stanford University</affiliation>
    <hostName>covertlab-jkarr.Stanford.EDU</hostName>
    <ipAddress>171.65.92.146</ipAddress>
    <revision>2378</revision>
    <differencesFromRevision><![CDATA[]]></differencesFromRevision>
    <condition>
        <shortDescription><![CDATA[test]]></shortDescription>
        <longDescription><![CDATA[test]]></longDescription>
        <replicates>1</replicates>
        <options>
            <option name="lengthSec" value="100"/>
        </options>
        <parameters>
            <parameter state="Mass" name="cellInitialDryWeight" value="4E-15"/>
            <parameter process="Transcription" ...
                name="rnaPolymeraseElongationRate" value="100"/>
        </parameters>
    </condition>
</conditions>
```

Lets review the code listed in Box 2.3. First, we ran the function `setWarnings` to turn off several distracting warning messages. Second, we ran the function `setPath` to add the whole-cell model code to the MATLAB path. `setWarnings` and `setPath` only need to be called once at the beginning of each MATLAB session. Third, we imported the `SimulationDiskUtil` class.

Fourth, we created a new directory to store the simulation. The `getBaseDir` method of the `SimulationDiskUtil` class returns the base directory under which all simulations are stored. Earlier you selected this path when you ran the installation script. Simulations are hierarchically organized under this path. First, simulations are organized into "batches", each of which corresponds to a single subdirectory under the path returned by the `getBaseDir` method; simulation batches are named by the date and time at which they are run. We recommend using simulation batches to organize groups of related simulations, for example replicates of a single condition. Second, each simulation is saved to a single subdirectory under a batch subdirectory; simulations are numbered starting at "1". By the end of each simulation, each simulation's directory will contain several mat files which store the complete simulated dynamics, as well as a summary of the simulated dynamics and meta data describing the simulation.

Fifth, we used the online configurator to generate an XML file which described our desired modifications to the default parameter values, and saved this XML file under the new directory we had just created to store the simulation. *Note: the `lengthSec` parameter must be set to a multiple of 100.*

Finally, we ran the `runSimulation` function to run a simulation. We passed the `simDir` parameter to the `runSimulation` function to specify where to store the simulated dynamics and where to find the XML file which describes the desired parameter values.

Programmatically setting parameter values

Alternatively, you can can programmatically override the default value of each option and parameter using the `setOptions` and `setParameters` methods of the `Simulation` class. Box 2.5 provides a complete example of how programmatically modify parameter values and run a whole-cell simulation. You can also use the `getOptions` and `getParameters` methods of the `Simulation` class to list all of the simulation parameters and their current values. For further information about the meaning of each parameter see the implementation of each state and process class.

Box 2.5 | Programmatically setting parameter values and running a simulation.

```
%import classes
import edu.stanford.covert.cell.sim.util.CachedSimulationObjectUtil;
import edu.stanford.covert.cell.sim.util.DiskLogger;
import edu.stanford.covert.cell.sim.util.SimulationDiskUtil;
import edu.stanford.covert.cell.sim.util.SummaryLogger;

%Select
%(1) simulation batch output directory and
%(2) simulation output directory
simBatch = datestr(now, 'yyyy_mm_dd_HH_MM_SS');
simIdx = 1;
simBatchDir = [SimulationDiskUtil.getBaseDir() filesep simBatch];
simDir = [SimulationDiskUtil.getBaseDir() filesep simBatch filesep ...
    num2str(simIdx)];

%create simulation batch and simulation output directories
if ¬isdir(simBatchDir)
    mkdir(simBatchDir); %create simulation batch output directory
end
if ¬isdir(simDir)
    mkdir(simDir); %create simulation output directory
end
```

```
%load simulation object with default parameter values
[sim, kbWID] = CachedSimulationObjectUtil.load();

%set parameter values
sim.applyOptions('lengthSec', 100);

parameterValues = struct();
parameterValues.states = struct();
parameterValues.states.Mass = struct();
parameterValues.states.Mass.cellInitialDryWeight = 4e-15;
parameterValues.processes = struct();
parameterValues.processes.Transcription = struct();
parameterValues.processes.Transcription.rnaPolymeraseElongationRate = 100;
sim.applyParameters(parameterValues);

%verify that parameter values correctly set
sim.getParameters().states.Mass.cellInitialDryWeight
sim.getParameters().processes.Transcription.rnaPolymeraseElongationRate

%setup loggers
summaryLogger = SummaryLogger(1, 1); %print to command line
summaryLogger.setOptions(struct('outputDirectory', simDir)); %save to disk

diskLogger = DiskLogger(simDir, 10); %save complete dynamics to disk
diskLogger.addMetadata(...
    'shortDescription',         'test simulation', ...
    'longDescription',          'test simulation', ...
    'email',                    'jkarr@stanford.edu', ...
    'firstName',                'Jonathan', ...
    'lastName',                 'Karr', ...
    'affiliation',              'Stanford University', ...
    'knowledgeBaseWID',         kbWID, ...
    'revision',                 1, ...
    'differencesFromRevision',  [], ...
    'userName',                 'jkarr', ...
    'hostName',                 'hostname.stanford.edu', ...
    'ipAddress',                '10.0.0.0');

loggers = {summaryLogger; diskLogger};

%run simulation
sim.run(loggers);
```

2.3 Running simulations on a cluster

There are two ways to run simulations on your own Linux cluster. First, you can use the user-friendly online configurator at `<yourserver>/simulation/runSimulations.php` to specify the simulations you wish to run, including the desired number of replicates. This script will generate an XML file describing the desired simulations, compile the whole-cell model code using the MATLAB compiler, and submit simulations jobs to Torque to execute the whole-cell model code using the MATLAB MCR. Alternatively, you can use the command-line program `simulation/runSimulations.pl` to run simulations specified by an XML file.

Chapter 3

Analyzing simulations

In the previous chapter we used two loggers – `SummaryLogger` and `DiskLogger` – to save predicted dynamics to disk. In this chapter we briefly describe how to retrieve and analyze the predicted dynamics stored by these loggers.

3.1 Analyzing summary logs

`SummaryLogger` logs the dynamics of 60 important cellular properties. The output of `SummaryLogger` is contained in the mat file `<simDir>/summary.mat`. Each variable in the mat file represents the dynamics of one or more cellular properties; time is represented by the second dimension of each of the stored properties. To analyze a summary log, first load the summary log into memory. Next, plot the simulated dynamics. Box 3.1 provides a simple script to analyze the temporal dynamics of the cellular mass stored in the summary log. Fig. 3.2 illustrates the output of the script listed in Box 3.1. See `SummaryLogger` for more information about each of the 60 logged cellular properties.

Box 3.1 | Summary log analysis.

```
%load summary log into memory
log = load([simDir filesep 'summary.mat']);

%plot data
subplot(2, 2, 1);
plot(log.time, log.mass * 1e15);
xlabel('Time (s)');
ylabel('Mass (fg)');

subplot(2, 2, 2);
plot(log.time, log.ploidy);
xlabel('Time (s)');
ylabel('Chr Copy No.');

subplot(2, 2, 3);
plot(log.time, log.rnas(1, :));
xlabel('Time (s)');
ylabel('RNA');

subplot(2, 2, 4);
plot(log.time, log.proteins(1, :));
xlabel('Time (s)');
ylabel('Protein');
```

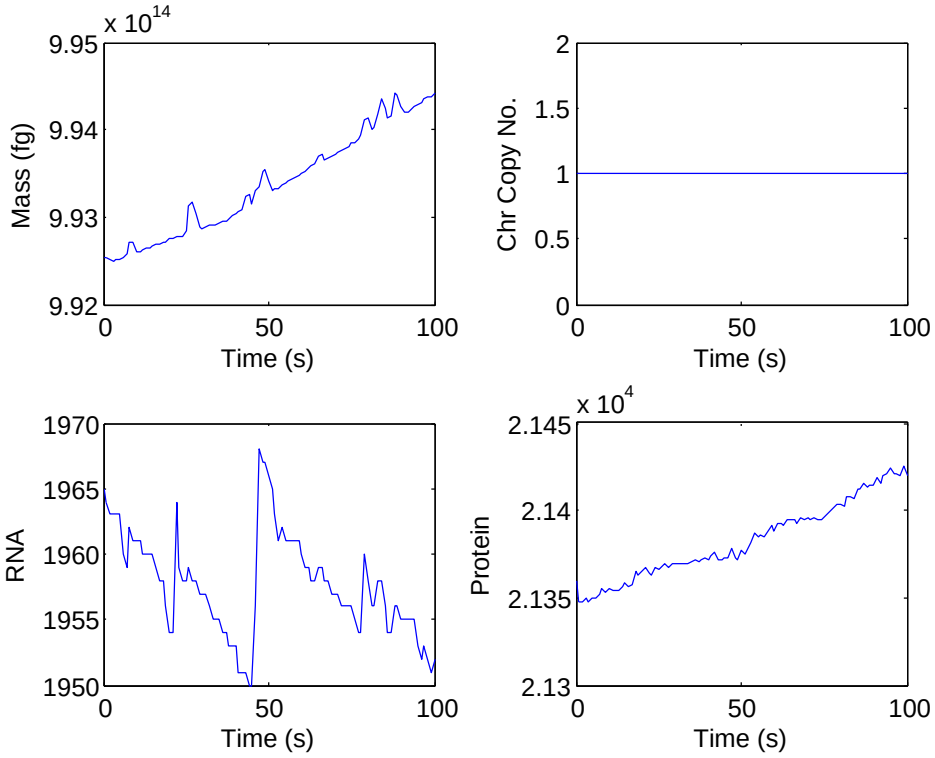

Figure 3.1 | Summary log analysis.

3.2 Analyzing the complete predicted dynamics

`DiskLogger` completely logs the simulated dynamics. The output of `DiskLogger` is organized into several mat files in the `<simDir>/` directory: `metadata.mat` contains simulation metadata including a textual description of the simulation; `options.mat`, `parameters.mat`, and `fittedConstants.mat` contain the value of each model option and parameter used in the simulation; `state-*.mat` contain the simulated dynamics divided into 10 s segments; and `randStreamStates.mat` contains the state of each rand stream at each simulated time point. To analyze the complete simulated dynamics, first load one or more cellular properties into memory using the `load` method of the `SimulationEnsemble` utility class. Second, instantiate the `Simulation` object which contains the detailed labels of the logged cellular properties. Finally, plot specific entries of the stored properties.

Box 3.2 provides an example of how to analyze the contribution of acetate kinase (*ackA*, MG357) to the cellular growth using the simulated dynamics logged by `DiskLogger`. Fig. 3.2 illustrates the output of the script listed in Box 3.2. The `edu.stanford.covert.cell.sim.analysis` package several additional examples of how to analyze the simulated dynamics stored by `DiskLogger`. See the implementation of each cellular state variable and process submodel for further information about the meaning and organization of each stored property.

Box 3.2 | Analysis of the complete simulated dynamics.

```
%import classes
import edu.stanford.covert.cell.sim.util.CachedSimulationObjectUtil;
import edu.stanford.covert.cell.sim.util.SimulationEnsemble;

%load simulaton object
sim = CachedSimulationObjectUtil.load();
comp = sim.compartment;
```

```
met = sim.process('Metabolism');
pc = sim.state('ProteinComplex');
pm = sim.state('ProteinMonomer');
rna = sim.state('Rna');
fluxIdx = met.reactionIndexs('AckA');
cpxIdx = pc.matureIndexs(pc.getIndexs('MG_357_DIMER'));
monIdx = pm.matureIndexs(pm.getIndexs('MG_357_MONOMER'));
rnaIdx = rna.matureIndexs(rna.getIndexs('TU_260'));

%load data
stateNames = {
    'Time'              'values'
    'Mass'              'cell'
    'MetabolicReaction' 'fluxs'
    'ProteinComplex'    'counts'
    'ProteinMonomer'    'counts'
    'Rna'               'counts'
    };
states = SimulationEnsemble.load(simBatchDir, stateNames, [], [], 1, ...
    'extract', simIdx);

%plot
subplot(5, 1, 1);
plot(permute(states.Time.values, [1 3 2]), permute(sum(states.Mass.cell, 2), ...
    [1 3 2]) * 1e15);
ylabel('Mass (fg)');

subplot(5, 1, 2);
plot(permute(states.Time.values, [1 3 2]), ...
    permute(states.MetabolicReaction.fluxs(fluxIdx, :, :), [1 3 2]) * 1e-3);
ylabel({'Flux' '(10^3 rxn s^{-1})'});

subplot(5, 1, 3);
plot(permute(states.Time.values, [1 3 2]), ...
    permute(states.ProteinComplex.counts(cpxIdx, comp.cytosolIndexs, :), [1 ...
    3 2]));
ylabel('Complex');

subplot(5, 1, 4);
plot(permute(states.Time.values, [1 3 2]), ...
    permute(states.ProteinMonomer.counts(monIdx, comp.cytosolIndexs, :), [1 ...
    3 2]));
ylabel('Monomer');

subplot(5, 1, 5);
plot(permute(states.Time.values, [1 3 2]), permute(states.Rna.counts(rnaIdx, ...
    comp.cytosolIndexs, :), [1 3 2]));
ylabel('RNA');
xlabel('Time (s)');
```

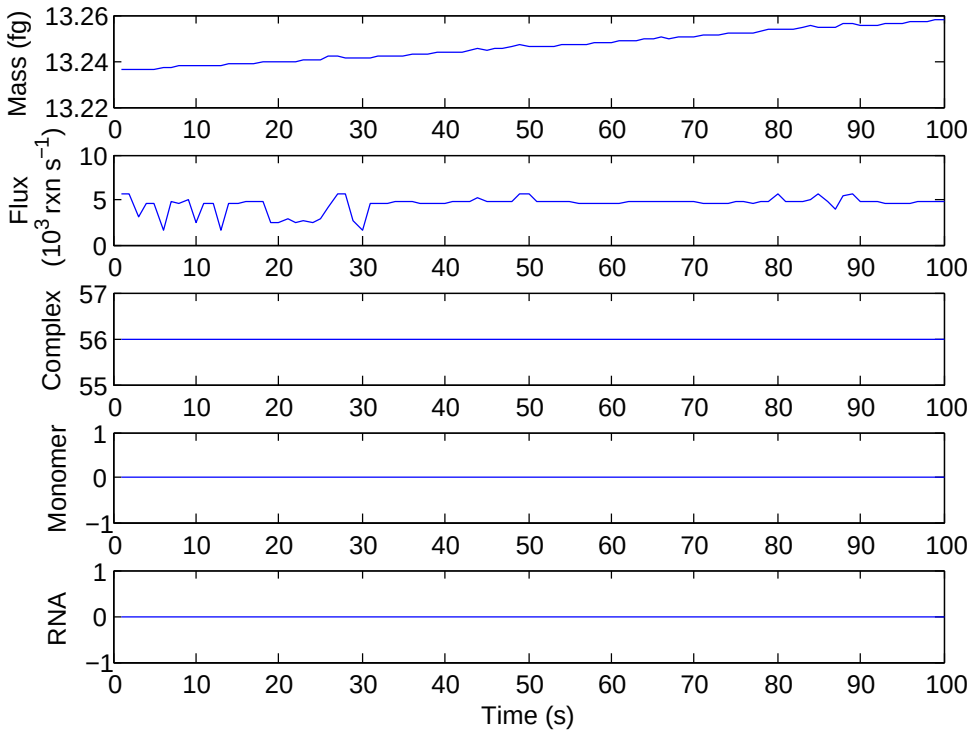

Figure 3.2 | Summary log analysis.

Chapter 4

Advanced topics

Now that you've mastered the basics of how to run and analyze whole-cell simulations, its time to get your hands dirty and explore the consequences of alternative submodels on the predicted phenotype. This chapter provides brief instructions on how to (1) modify the implementations of the states and processes, (2) add new states and processes, and (3) instantiate and fit modified models. Please contact the authors for help modifying and fitting the model.

4.1 Modifying states

As an example of how to modify a state, lets revise the cell shape model of the `Geometry` state. In particular, lets revise the cell shape model to reflect *M. genitalium*'s flask shape. First lets open the `Geometry` class, `simulation/src/+edu/+stanford/+covert/+cell/+sim/+state/CellGeometry.m`. Second, lets add two properties to the `Geometry` state to represent the terminal organelle diameter and length as illustrated in Box 4.1.

Box 4.1 | Adding state properties.

```
%fixed biological constants
properties
    density                 %g/L [PUB_0553]
    termOrgDiameter = 80e-9  %m [PUB_0091]
    termOrgLength   = 300e-9 %m [PUB_0091]
end
```

Third, lets annotate these new properties as constants as illustrated in Box 4.2.

Box 4.2 | Annotating state properties.

```
fixedConstantNames      = {
    'density'
    'termOrgDiameter'
    'termOrgLength'
    };
```

Fourth, lets modify the `calculateGeometry` method to account for the terminal organelle as illustrated in Box 4.3.

Box 4.3 | Editing state methods.

```
function [cylindricalLength, surfaceArea, totalLength] = ...
    calculateGeometry(width, pinchedDiameter, volume)
    if pinchedDiameter > 0
        %calculate dimensions of cell whose shape is a cyclinder with
        %hemispherical caps

        w = width;
        s = (w - pinchedDiameter)/2; %m
        termOrgVolume = (...
            + pi * (this.termOrgDiameter / 2)^2 * (this.termOrgLength) ...
            + (1/2) * 4/3 * (this.termOrgDiameter / 2)^3 ...
            ) * 1e6;

        cylindricalLength = ...                    %(m)
            max(0, ...
            + (volume - termOrgVolume) / 1000 ... %total - term org vols
            - 1/6 * pi * width^3 ...               %spherical caps
            - pi/2*(s*(8*s^2-4*s*w+w^2) ...        %septum
            + 2*s^2*(-2*s+w)*pi/2 - 4/3*s^3) ...
            ) * 4/(pi * width^2);
        surfaceArea = ...                                        %(m^2)
            + pi * width^2 ...                                   %spherical caps
            + pi * width * cylindricalLength ...                 %cylinder
            + pi * 4*s*(w-2*s+s) ...                             %septum
            + pi * this.termOrgDiameter * this.termOrgLength %term org cylinder
            + 1/2 * 4 * (this.termOrgDiameter/2)^2               %term org cap
            - pi * (this.termOrgDiameter/2)^2                    %term org base
        totalLength = ...            %m
            + cylindricalLength ...
            + width ...
            + 2 * s ...
            + this.termOrgLength;
    else
        % cell has divided
        warning('WholeCell:warning', 'Cell has divided. Cell shape undefined');
        cylindricalLength = -1;
        surfaceArea = -1;
        totalLength = -1;
    end
end
```

Finally, for completeness we should update the `Geometry` state test cases. We leave this as an exercise for the reader.

4.2 Modifying processes

You just read a seminal paper in *Nature* which illuminated the previously misunderstood roles of the five enzymes associated with chromosome segregation. In particular, the new study shows that the proteins CobQ, MraZ, Obg, Era, and topoisomerase IV are required in 1:2:3:4:5 stoichiometry for chromosome segregation, and that chromosome segregation is coupled to the hydrolysis of 15 GTP molecules. Lets revise the chromosome segregation submodel as an example of how modify a cellular process submodel. First, lets open the `ChromosomeSegregation` submodel class, `simulation/src/+edu/+stanford/+covert/+cell/+sim/+process/ChromosomeSegregation.m`. Second, lets edit the `evolveState` method which implements the dynamic model to require the observed enzyme stoichiometries as illustrated in Box 4.4.

Box 4.4 | Editing `evolveState` methods.

```
function evolveState(this)
    c = this.chromosome;

    if ...
            ¬c.segregated && ...
            collapse(c.polymerizedRegions) == c.nCompartments * ...
                c.sequenceLen && ...
            collapse(c.supercoiled) == c.nCompartments && ...
            this.enzymes(this.enzymeIndexs_cobQ)  ≥ 1 && ...
            this.enzymes(this.enzymeIndexs_mraZ)  ≥ 2 && ...
            this.enzymes(this.enzymeIndexs_obg)   ≥ 3 && ...
            this.enzymes(this.enzymeIndexs_era)   ≥ 4 && ...
            this.enzymes(this.enzymeIndexs_topIV) ≥ 5 && ...
            this.substrates(this.substrateIndexs_gtp) ≥ this.gtpCost && ...
            this.substrates(this.substrateIndexs_water) ≥ this.gtpCost

        c.segregated = true;

        this.substrates(this.substrateIndexs_gtp) = ...
            this.substrates(this.substrateIndexs_gtp) - this.gtpCost;
        this.substrates(this.substrateIndexs_water) = ...
            this.substrates(this.substrateIndexs_water) - this.gtpCost;
        this.substrates(this.substrateIndexs_gdp) = ...
            this.substrates(this.substrateIndexs_gdp) + this.gtpCost;
        this.substrates(this.substrateIndexs_phosphate) = ...
            this.substrates(this.substrateIndexs_phosphate) + this.gtpCost;
        this.substrates(this.substrateIndexs_hydrogen) = ...
            this.substrates(this.substrateIndexs_hydrogen) + this.gtpCost;
    end
end
```

Third, lets set the default value of the GTP cost of chromosome segregation as illustrated in Box 4.5.

Box 4.5 | Editing process properties.

```
properties
    gtpCost = 15 %number of GTP required for chromosome segregation
end
```

Fourth, as illustrated in Box 4.6, lets edit the `calcResourceRequirements_LifeCycle` method to ensure that the five enzymes will be sufficiently expressed to support chromosome segregation. See Sec. 4.4 for further discussion.

Box 4.6 | Editing process resource requirement calculations.

```
function [bmProd, byProd, minEnzExp, maxEnzExp] = ...
    calcResourceRequirements_LifeCycle(this, ¬, ¬)
    %% initialize
    bmProd = zeros(size(this.substrateWholeCellModelIDs));
    byProd = zeros(size(this.substrateWholeCellModelIDs));
    minEnzExp = zeros(size(this.enzymeWholeCellModelIDs));
    maxEnzExp = Inf(size(this.enzymeWholeCellModelIDs));

    %% substrate and byproducts: GTP required to decatenate chromosomes
    bmProd(this.substrateIndexs_gtp)       = this.gtpCost;
    bmProd(this.substrateIndexs_water)     = this.gtpCost;
```

```
        byProd(this.substrateIndexs_gdp)       = this.gtpCost;
        byProd(this.substrateIndexs_phosphate) = this.gtpCost;
        byProd(this.substrateIndexs_hydrogen)  = this.gtpCost;

        %% Segregation requires at least 1 copy of every enzyme
        minEnzExp(this.enzymeIndexs_cobQ)  = 1;
        minEnzExp(this.enzymeIndexs_mraZ)  = 2;
        minEnzExp(this.enzymeIndexs_obg)   = 3;
        minEnzExp(this.enzymeIndexs_era)   = 4;
        minEnzExp(this.enzymeIndexs_topIV) = 5;
    end
```

Finally, for completeness we should update the `ChromosomeSegregation` process test cases. We leave this as an exercise for the reader.

4.3 Adding states and processes

Now that we've mastered how to edit states and processes, its time to add new states and processes to the model. *Note: adding states and processes requires installation of the whole-cell knowledge base.*

First, lets add new state and process classes to the `simulation/src/+edu/+stanford/+covert/+cell/+sim/+state` and `simulation/src/+edu/+stanford/+covert/+cell/+sim/+process` directories following the templates provided in Boxes 4.7 and 4.8. Second, lets implement unit tests of the new states and processes following the templates in Boxes 4.9 and 4.10 and place these new tests in the `simulation/src_test/+edu/+stanford/+covert/+cell/+sim/+state` and `simulation/src_test/+edu/+stanford/+covert/+cell/+sim/+process` directories. Third, we need to register the new states and processes with your whole-cell knowledge base by completing the web forms at `<yourserver>/knowledgebase/index.php?Method=Edit&TableID=states` and `<yourserver>/knowledgebase/index.php?Method=Edit&TableID=processes`. Finally, we need to rebuild and fit the simulation as discussed in Sec. 4.4.

Box 4.7 | State template.

```
%StateTemplate
%   Description of state class.
%
%   References
%   ==========
%   1. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_xxxx]
%   2. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_yyyy]
%   3. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_zzzz]
%
% Author: Jonathan Karr, jkarr@stanford.edu
% Affilitation: Covert Lab, Department of Bioengineering, Stanford University
% Last updated: 2/13/2012
classdef StateTemplate < edu.stanford.covert.cell.sim.CellState
    %property annotations
    properties (Constant)
        optionNames             = {   %names of properties that are options
            'verbosity';
            'seed';
            };
        fixedConstantNames      = {   %names of fixed constant properties
            'constant1';
            'constant2';
            };
        fittedConstantNames     = {}; %names of fitted constant properties
        stateNames              = {   %names of state properties
```

```
            'state1'
            'state2'
            };
        dependentStateNames     = {   %names of dependent state properties
            'dependentState1'
            'dependentState1'
            };

    end

    %constants -- for example, used for enumeration or indexing
    properties (Constant)
    end

    %fixed biological constants, set to values store in knowledge base by the
    %base class initializeConstants method or by this class'
    %initializeConstants method
    properties
        constant1
        constant2
    end

    %cell state properties -- store dynamic state of cell
    properties
        state1
        state2
    end

    %dependent state -- views of the cell state which can
    %be calculated from other cell state properties
    properties (Dependent = true, SetAccess = protected)
        dryWeight
        dependentState1
        dependentState2
    end

    %references to other cell state objects
    properties
        stateReference1
        stateReference2
    end

    %constructor -- called by constructStates method of simulation class during
    %construction of the simulation object
    methods
        function this = StateTemplate(wholeCellModelID, name)
            this = this@edu.stanford.covert.cell.sim.CellState(...
                wholeCellModelID, name);
        end
    end

    methods
        %build object graph -- this method should store references to other
        %parts of the cell state and set the properties, for example
        %stateReference1 and stateReference2. the method is called during
        %construction of the simulation object, just after construction
        %of all the state and process objects
        function storeObjectReferences(this, simulation)
            this.stateReference1 = simulation.state('stateReference1');
```

```
            this.stateReference2 = simulation.state('stateReference2');
        end
    end

    methods
        function initializeConstants(this, knowledgeBase, simulation)
            this.initializeConstants@edu.stanford.covert.cell.sim.CellState(...
                knowledgeBase, simulation);
        end
    end

    %allocate memory for state -- called before initializing state to
    %allocate memory to represent cell state
    methods
        function allocateMemory(this, numTimePoints)
            this.state1 = zeros(1, 1, numTimePoints);
            this.state2 = zeros(1, 1, numTimePoints);
        end
    end

    %model
    methods
        %initialize state -- this method should initialize
        %the state variable of this class (eg. properties state1,
        %state2)
        %- called by the initializeState method of the
        %  simulation class prior the start of the simulation
        %- also called by the FitConstants class
        function initialize(this)
        end
    end

    %public interface methods exposed to processes and other states
    methods
    end

    %getters
    methods
        %calculates dryWeight value
        function value = get.dryWeight(this)
            value = this.calcDryWeight();
        end

        %calculates dependentState1 value
        function value = get.dependentState1(this)
            value = this.calcDependentState1();
        end

        %calculates dependentState2 value
        function value = get.dependentState2(this)
            value = this.calcDependentState2();
        end
    end
end
```

Box 4.8 | Process template.

```
%ModuleTemplate
```

```
%
% @wholeCellModelID Module_ModuleTemplate
% @name             ModuleTemplate
% @description
%   Biology
%   =================
%
%   Knowledge Base
%   =================
%
%   Representation
%   =================
%
%   Initialization
%   =================
%
%   Simulation
%   =================
%
%   Algorithm
%   ==================
%
%   References
%   =================
%   1. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_xxxx]
%   2. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_yyyy]
%   3. Authors (Year). Title. Journal. Volume (Number): Pages. [PUB_zzzz]
%
% Author: Jonathan Karr, jkarr@stanford.edu
% Affiliation: Covert Lab, Department of Bioengineering, Stanford University
% Last updated: 8/12/2010
classdef  ModuleTemplate < edu.stanford.covertlab.wholecell.simulation.Module
    %property annotations
    properties
        optionNames__               = { %names of option properties
            'option1'};
        fixedConstantNames__        = { %names of fixed constant properties
            'fittedConstant1'};
        fittedConstantNames__       = { %names of fitted constant properties
            'fittedConstant1'};
        localStateNames__           = { %names of state properties
            'localState1'};
    end

    %options
    properties
        option1
    end

    %enumerations
    properties (Constant = true)
    end

    %IDs, names, and local indices
    properties
        stimuliWholeCellModelIDs = {};     %stimuli whole cell model IDs

        substrateWholeCellModelIDs = {     %subsate whole cell model IDs
            'substrate1';
```

```
            'substrate2'};
        substrateIndexs_substrates12 = (1:2)';

        enzymeWholeCellModelIDs = {...       %enzyme whole cell model IDs
            'enzyme1'};          %name of enzyme 1
        enzymeIndexs_enzyme1  = 1;
    end

    %fixed biological constants
    properties
        fittedConstant1
    end

    %local state
    properties
        localState1
    end

    %references to cell state
    properties
        stateReference1
        stateReference2
    end

    %constructor
    methods
        function this = ModuleTemplate(idx, wholeCellModelID, name)
            this = this@edu.stanford.covertlab.wholecell.simulation.Module(...
                idx, wholeCellModelID, name);
        end
    end

    %communication between module/simulation
    methods
        %set references to state objects
        function storeObjectReferences(this, simulation)
            this.storeObjectReferences@edu.stanford.covert.cell.sim.Process(...
                simulation);
            this.stateReference1 = simulation.state('stateReference1');
            this.stateReference2 = simulation.state('stateReference2');
        end

        %initialize constants
        function initializeConstants(this, knowledgeBase, simulation, varargin)
            this.initializeConstants@edu.stanford.covertlab.wholecell....
                simulation.Module(knowledgeBase, simulation, varargin{:});

            this.fittedConstant1 = knowledgeBase.fittedConstant1;
        end

        %retrieve state from simulation
        function copyFromState(this)
            this.copyFromState@edu.stanford.covertlab.wholecell....
                simulation.Module;

            this.localState1 = simulation.globalState1;
        end

        %send state to simulation
```

```
        function copyToState(this)
            this.copyToState@edu.stanford.covertlab.wholecell....
                simulation.Module;

            simulation.globalState1 = this.localState1;
        end
    end

    %allocate memory for state
    methods
        function allocateMemoryForState(this, numTimePoints)
            this.allocateMemoryForState@edu.stanford.covertlab.wholecell....
                simulation.Module(numTimePoints);

            numComps = 1;
            this.localState1  = zeros(localState1_size,  numComps, ...
                numTimePoints);
        end
    end

    %model
    methods
        %Calculate
        %(1) contribution to FBA objective
        %(2) minimum expression consistent with cell cycle length given
        %    current estimates of
        %    - Cell weight
        %    - Cell cycle length, area under cell growth curve
        %    - Composition: dNMP, NMP, AA, other
        %    - Expression: RNA, gene, protein monomers
        %    - Decay rates: RNA, protein monomers
        function [biomassProduction, byproducts, ...
            minimumEnzymeExpression, maximumEnzymeExpression] = ...
            calcResourceRequirements_LifeCycle(this, constants, states)

            biomassProduction = zeros(size(this.substrates));
            byproducts = zeros(size(this.substrates));

            minimumEnzymeExpression = zeros(size(this.enzymes));
            maximumEnzymeExpression = zeros(size(this.enzymes));
        end

        %Calculate demand for metabolite resources
        function result = calcResourceRequirements_Current(this)
            result = zeros(size(this.substrates));
        end

        %initialization
        function initializeState(this)
        end

        %simulation
        function evolveState(this)
            %do some random operation
            randomWeightIndexs = this.randStream.randw(weights, 10);
        end
    end

    %model helper functions
```

```
        methods
        end
180 end
```

Box 4.9 | State test template.

```
1  %StateTemplate test cases
   %
   % Author: Jonathan Karr, jkarr@stanford.edu
   % Affiliation: Covert Lab, Department of Bioengineering, Stanford University
5  % Last updated: 2/13/2012
   classdef StateTemplate_Test < edu.stanford.covert.cell.sim.CellStateTestCase
       %constructor
       methods
           function this = StateTemplate_Test(name)
10             this = this@edu.stanford.covert.cell.sim.CellStateTestCase(name);
           end
       end

       %tests
15     methods
           %test #1
           function test1(this)
               assertion1();
               assertion2();
20             assertion3();
           end

           %test #2
           function test2(this)
25             assertion1();
               assertion2();
               assertion3();
           end
       end
30 end
```

Box 4.10 | Process test template.

```
1  %Template module test case
   %
   % Author: Jonathan Karr, jkarr@stanford.edu
   % Affilitation: Covert Lab, Department of Bioengineering, Stanford University
5  % Last updated: 7/13/2010
   classdef Module_TestTemplate < ...
       edu.stanford.covertlab.wholecell.simulation.ModuleTestCase
       %constants
       properties (Constant = true)
           expected_essentialGenes = {};
10     end

       %constructor
       methods
           function this = Module_TestTemplate(name)
15             this = this@edu.stanford.covertlab.wholecell.simulation. ...
                   ModuleTestCase(name);
           end
```

```
    end

    %hard coded fixture
    methods
        function testSimpleNetwork(this)
            %module
            module = this.module;
        end

        function loadSimpleNeworkTestFixture(this)
            %module
            module = this.module;
        end
    end

    %tests
    methods
        function testNoStimuli(this)
            %module
            module = this.module;
        end

        function testNoSubstrates(this)
            %module
            module = this.module;
        end

        function testNoEnzymes(this)
            %module
            module = this.module;
        end

        function testGeneEssentiality(this)
            %module
            module = this.module;

            %super class method
            this.testGeneEssentiality@edu.stanford.covertlab.wholecell....
                simulation.ModuleTestCase();
        end

        % Gene essentiality
        function testGeneEssentiality(this)
            m = this.process;

            this.helpTestGeneEssentiality({
                'EssentialGene1'
                'EssentialGene2'
                'EssentialGene3'}, ...
                @this.isCellProperlyFunctioning);
        end
    end

    %helper methods
    methods
        function cellIsProperlyFunctioning = isCellProperlyFunctioning(...
            this, initial_timeCourses)
            module = this.module;
```

```
                cellIsProperlyFunctioning = true;
            end
        end
80  end
```

4.4 Instantiating and fitting the model

Finally, to incorporate new states and processes into the model, we must construct an instance of the `Simulation` class and fit the model as illustrated in Box 4.11. First, we must download the content of the knowledge base. Second, we must construct an instance of the `Simulation` class using the knowledge base content. Third, we must fit the model using the `FitConstants` class. Fourth, we must initialize the simulation object using the `initializeState` method and cache the fitted simulation instance to disk using the `CachedSimulationObjectUtil` utility class. The `generateTestFixtures` script in the `simulation` directory provides additionally functionality beyond the script listed in Box 4.11 (1) to seed the random streams contained inside the new simulation instance to ensure reproducibility and (2) regenerate the test fixtures based on the new simulation instance.

Box 4.11 | Instantiating and fitting simulations.

```
1   % import classes
    import edu.stanford.covert.cell.kb.KnowledgeBaseUtil;
    import edu.stanford.covert.cell.kb.KnowledgeBase;
    import edu.stanford.covert.cell.sim.Simulation;
5   import edu.stanford.covert.cell.sim.util.FitConstants;
    import edu.stanford.covert.db.MySQLDatabase;

    % initialize
    dbParams = config();
10  db = MySQLDatabase(dbParams);

    % construct latest knowledge base from database
    knowledgeBaseWID = KnowledgeBaseUtil.selectLatestKnowledgeBase(db);
    kb = KnowledgeBase(db, knowledgeBaseWID);
15
    % construct simulation and initialize its constants
    simulation = Simulation(kb.states, kb.processes);
    simulation.initializeConstants(kb);
    fitter = FitConstants(simulation);
20  fitter.run();

    % write simulation data
    save('data/Simulation_fitted.mat', 'simulation', 'knowledgeBaseWID');

25  % clean up
    db.close();
```

4.5 Testing the model

The whole-cell model was computationally validated using over 1,000 unit tests. The tests were divided into the eight builds listed in Tab. 4.1. Each build was implemented as a MATLAB script. These scripts are located in the `simulation` directory. Each build script produces a JUnit-style XML report of the successful and unsuccessful tests. The unit tests were implemented in object-oriented MATLAB and are organized under the `simulation/src_test` directory. Most of these tests use test fixtures which represent a fully fitted and initialized simulation, or parts thereof. These test fixtures are produced by the `generateTestFixtures`

script in the `simulation` directory.

Table 4.1 | Testing builds.

Build	Description
`runSmallTests`	Tests each individual state, process, and utility class
`runMediumTests`	Tests assembles of states and process classes
`runMediumDNATests`	Tests the chromosome state and related processes
`runMediumProteinTests`	Tests the protein synthesis and maturation processes
`runLargeTests`	Tests all of the states and processes together
`runSimulationTests`	Tests one full-length simulation
`runAnalysisTests`	Tests of each analysis script
`runCoverageTests`	Assesses the code coverage of all of the above tests