Chapter 17

Programming Juxtacrine-Based Synthetic Signaling Networks in a Cellular Potts Framework

Calvin Lam and Leonardo Morsut

Abstract

Synthetic development is a synthetic biology subfield aiming to reprogram higher-order eukaryotic cells for tissue formation and morphogenesis. Reprogramming efforts commonly rely upon implementing custom signaling networks into these cells, but the efficient design of these signaling networks is a substantial challenge. It is difficult to predict the tissue/morphogenic outcome of these networks, and in vitro testing of many networks is both costly and time-consuming. We therefore developed a computational framework with an in silico cell line (ISCL) that sports basic but modifiable features such as adhesion, motility, growth, and division. More importantly, ISCL can be quickly engineered with custom genetic circuits to test, improve, and explore different signaling network designs. We implemented this framework in a free cellular Potts modeling software CompuCell3D. In this chapter, we briefly discuss how to start with CompuCell3D and then go through the steps of how to make and modify ISCL. We then go through the steps of programming custom genetic circuits into ISCL to generate an example signaling network.

Key words Synthetic biology, Computational modeling, Cellular Potts, Tissue engineering, Morphogenesis, Self-organization, Juxtacrine signaling, Synthetic receptors, Synthetic development, Differential adhesion

1 Introduction

Synthetic development is a recent subbranch of synthetic biology that aims to control tissue formation and morphogenesis through custom signaling networks [1–6]. Efforts have recently expanded into engineering mammalian cells, but they proceed rather slowly due to reliance on trial-and-error iterations [7–10]. One possible solution is to perform parallel screens of numerous signaling networks. However, this is not practical in vitro as it is both time-consuming and costly. A computational analog could be an ideal solution here [11]. For example, an in silico framework with a simulated cell line that can be implemented with different signaling

Jeffrey Carl Braman (ed.), *Synthetic Biology: Methods and Protocols*, Methods in Molecular Biology, vol. 2760,
https://doi.org/10.1007/978-1-0716-3658-9_17,

networks would be modularly identical to that of the in vitro system. However, rather than requiring time-consuming modifications of the real cell's properties, it would be a rapid modification of code representing the property in the simulated cell. Such a framework enables cost-effective and rapid screening of numerous signaling networks, as it becomes possible to design, implement, alter, and test even within the same day.

We have developed one such framework and thus far have used it to study the synNotch differential adhesion system in Toda et al. [8, 12]. In this in vitro system, several signaling networks are implemented as genetic circuits into a mouse fibroblast cell line (L929). These genetic circuits consist of the synthetic Notch (synNotch) receptor that enables cells to sense a user-specified ligand and induce the expression of a fluorescent reporter and adhesion protein. Different signaling networks comprise cells programmed with different synNotch genetic circuits, enabling cells to communicate with one another and undergo organization into different structures. One signaling network (Fig. 1) from this system will serve as a demo in this chapter, and we will go through the steps on how to create this network. This signaling network served as a reference for others in the in vitro system and offers a good balance of generalizability and complexity. As we implemented our framework in a free cellular Potts software, CompuCell3D (CC3D) [13], we will show how to generate this network in CC3D. In general, however, this framework can be abstracted to other modeling software provided it can model the composing features.

2 Materials

Materials to use the framework include the CompuCell3D installation file for your operating system, the framework code on the GitHub repository, and a computer.

As the framework was built in CC3D version 3.7.8, we will show how to make the reference network in this version. Download the installation file for your operating system at: https://compucell3d.org/SrcBinOldReleases#A378 (*see* **Note 1**). The code for the reference signaling network is available at: https://github.com/calvinlamk/Book-Chapter-Demo. "ChapterDemo" is the project file containing all the code needed for the CC3D simulation. This signaling network corresponds to Fig. 4g in [12] (*see* **Note 2**).

A modern computer running either Windows, Mac, or Linux can be used with CC3D.

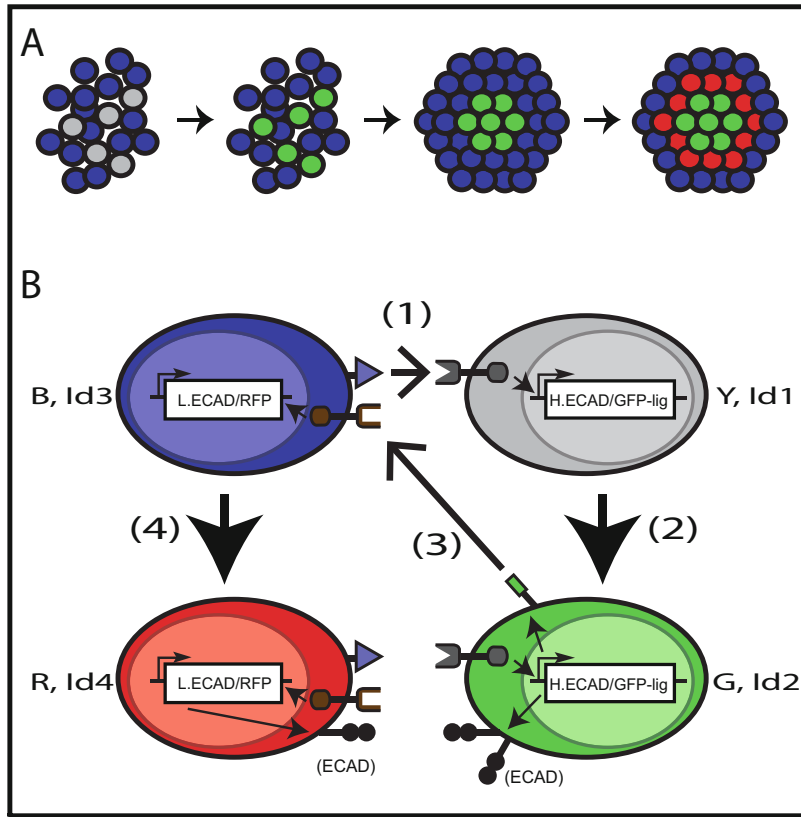

Fig. 1 Reference signaling network from [8] as modeled in [12]. (Redrawn from Toda 2018 with permission from AAAS). The reference signaling network contains two parts. (**a**) First, a developmental trajectory is designed consisting of a mixture of blue and gray cells. Blue cells then signal to gray cells turning them green and adhesive to other green cells. Green cells then signal to blue cells turning them red and adhesive to green cells more than other red cells. End point structure is shown. (**b**) To realize this developmental trajectory, cells were programmed with the following genetic circuits. Blue cells (in silico coded as type name B with type Id 3) signal with a triangle ligand to gray cells (in silico coded as type name Y with type Id 1) via a notched synNotch receptor (step 1). This causes gray cells to activate to green cells (in silico coded as type name G with type Id 2) (step 2). Green cells have high levels of E-cadherin (an adhesive protein denoted as circles on a stick) and a GFP reporter-ligand (GFP-lig) that can signal to blue cells via a square indented synNotch receptor (step 3). This signaling causes blue cells to activate, turning them red (in silico coded as type name R with type Id 4) via a red fluorescent reporter protein and expressing low levels of E-cadherin (step 4). As green cells and red cells still have the notched and square indented synNotch receptor, respectively, they can still receive signaling from the cognate ligand

3 Methods

3.1 Getting Started

After installing CC3D, we will open the built-in project editor called Twedit++ (*see* **Note 3**). Twedit++ can be found in the CC3D install directory, and in Windows OS, CC3D by default installs into the C directory. Be sure to open the run file and not the folder, which contains the backend. The program will pop up

along with a terminal, which serves to show the commands processed. Open the downloaded demo project file called "ChapterDemo," by navigating to the "CC3D Project" tab (top left of program), then clicking "Open CC3D Project. . .", and then opening the CC3D file named "ChapterDemo" in the project folder. Both the project folder and CC3D file are named "ChapterDemo." This will load all constitutive files in the project folder and will show up at the left of Twedit++. For this demo, you should see the CC3D icon (little green globe) with "ChapterDemo.cc3d" next to it. To see the code in the project, expand the project by clinking the > icons. Notice that the project comprises several files. In the demo, it is "ELUGM.py," "ELUGM.xml," "ELUGMSteppables.py," and "ParameterScanSpecs.xml." Click each of these individual files to load them into Twedit++ for viewing and editing. For the framework, the files we will discuss are the ELUGM xml file, ELUGM Steppables py file, and ParameterScanSpecs xml file (*See* **Note 4**).

The xml file contains the basic parameters for the cellular Potts calculations and simple plugins for features such as cells, adhesion, and initialization of the simulation. The steppables.py file contains the majority of the framework and is organized by comments headed by the # symbol. These comments serve as section headers and offer a simple way to locate sections by simply searching via Ctrl F. The parameterscan xml file, though not necessary for running the simulation, is crucial for testing signaling network parameters.

3.2 Creating Cells in CC3D

To start, we will first determine the number of cell types needed in our signaling network (Fig. 1). In the reference signaling network, we have two cell types, gray and blue, that each have their own genetic circuit. With signaling, each cell type can change its adhesive state by expressing different levels of an adhesive protein. Gray cells can turn green indicating it has high E-cadherin, and blue cells can turn red indicating it has low E-cadherin (Fig. 1b). Therefore, we need to specify four cell types in total (*see* **Note 5**).

To create these cells in CC3D, we navigate to the xml file "ELUGM.xml" and find the line "<Plugin Name="CellType">" (Fig. 2a). Directly underneath this line is a block of code specifying the existence of the cell type along with its type Id and type name. For example, "<CellType TypeId="1" TypeName="Y"/>" tells CC3D that cells of type Y exist in this code and are recognized with a type Id of 1. The name here is arbitrary, and here, it is Y for gray. We strongly recommend naming cell types in a manner easily recognizable in the signaling network schematic. From this block of code, we can see we have four cell types: Y, G, B, R for gray, green, blue, and red, respectively. Respective Id number is 1, 2, 3, and 4. To add more cell types, just add in additional lines of "<CellType TypeId="X" TypeName="Z"/>," where X is a type Id number not previously used for another cell type and Z is the

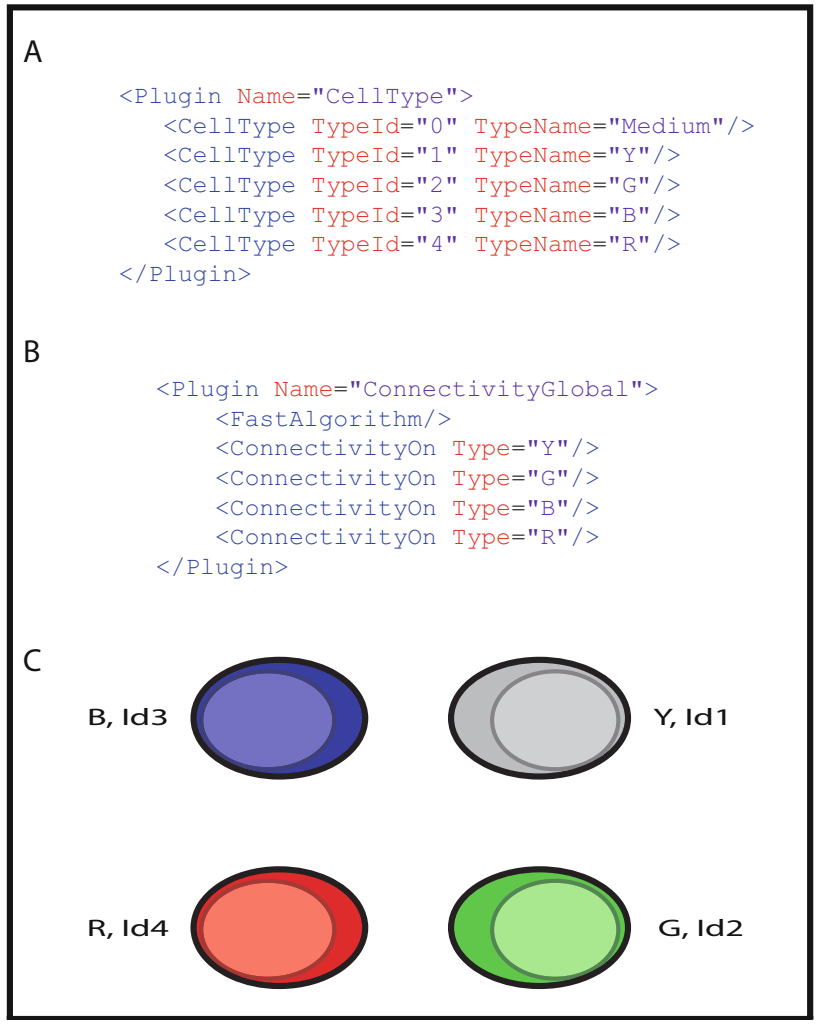

Fig. 2 Establishing the cells in CC3D. (**a**) CellType plugin allows specifying cells in CC3D through an Id number and a type name. (**b**) ConnectivityGlobal plugin prevents cells from fragmenting during the simulation. (**c**) With these lines specified, we have defined the cells needed in our reference signaling network in CC3D

name (*see* **Note 6**). For example, the line <CellType TypeId="5" TypeName="O"/> will create a cell type O with type number 5.

We have utilized a CC3D plugin that helps stop cells from fragmenting, and the added cell types will need to be included here as well. Find the line "<ConnectivityOn Type="R"/>" (Fig. 2b), and add your additional cell types under this by replacing the X in this line "<ConnectivityOn Type="X"/>" with the name specifying the type, and then copy the line to under "<ConnectivityOn Type="R"/>." In the example above, we would add the line "<ConnectivityOn Type="O"/>."

With this section completed, we have defined the cell types required to model the reference signaling network (Fig. 2c).

3.3 Adding Adhesion to the In Silico Cell Line (ISCL)

With cells defined in CC3D, we can start adding properties to these cells. We begin with adhesion, as it is a basic feature of all biological cells and is key in our reference signaling network. In the framework, we utilize the basic adhesion plugin called Contact Plugin, which is located in the same xml file, "ELUGM.xml" (*See* **Note 7**). Find the line "<Plugin Name="Contact">" (Fig. 3a).

A

```
<Plugin Name="Contact">
   <Energy Type1="Medium" Type2="Medium">0</Energy>
   <Energy Type1="Medium" Type2="Y">26.0</Energy>
   <Energy Type1="Medium" Type2="G">26.0</Energy>
   <Energy Type1="Medium" Type2="B">26.0</Energy>
   <Energy Type1="Medium" Type2="R">26.0</Energy>
   <Energy Type1="Y" Type2="Y">45.0</Energy>
   <Energy Type1="Y" Type2="G">40.0</Energy>
   <Energy Type1="Y" Type2="B">49.0</Energy>
   <Energy Type1="Y" Type2="R">40.0</Energy>
   <Energy Type1="G" Type2="G">20.0</Energy>
   <Energy Type1="G" Type2="B">49.0</Energy>
   <Energy Type1="G" Type2="R">35.0</Energy>
   <Energy Type1="B" Type2="B">49.0</Energy>
   <Energy Type1="B" Type2="R">49.0</Energy>
   <Energy Type1="R" Type2="R">40.0</Energy>
   <NeighborOrder>4</NeighborOrder>
</Plugin>
```

B

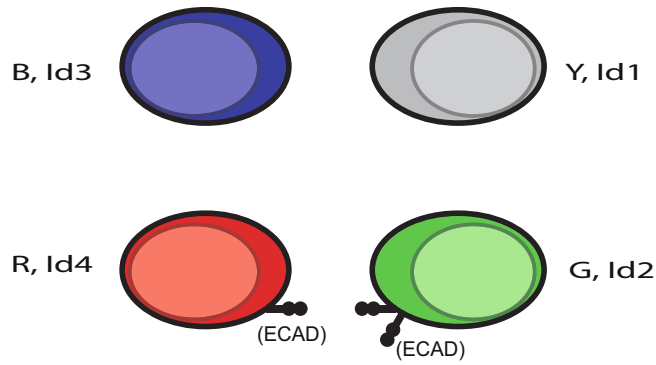

Fig. 3 Defining cell-cell adhesion in CC3D. (**a**) We have utilized a contact plugin that allows basic modeling of cell-cell adhesion as they differ between types of cells. The higher the value, the weaker the cell type-cell type adhesion. For example, G-R adhesion is 35, which is intermediate in our reference signaling network. Medium is a special double interface calculation; thus, as long as cell-cell adhesion is less than 52 (2 × 26), cells will prefer to adhere to one another rather than disperse. NeighborOrder controls the number of surrounding pixels used when calculating the effect of adhesion in moving cells. We find that four help cells sort at a reasonable rate for these signaling networks. (**b**) With the adhesion values implemented, we are beginning to piece our signaling network together

Underneath, you will notice a block of code following the pattern "<Energy Type1="X" Type2="Z">NUMBER</Energy>" where X and Z are names of cell types specified in the CellType Plugin we just examined. For example, the line "<Energy Type1="Y" Type2="Y">45.0</Energy>" specifies the adhesion values of cells of type Y to other cells of type Y as 45. In the Potts formulation, a smaller number corresponds to a stronger adhesion strength. Thus, G-G adhesion in this signaling network is much stronger than Y-Y adhesion ($20 < 45$).

Also, notice that the cell-to-medium adhesion is specified as well. Here, it is 26, which CC3D interprets twice, one for the cell interface and one for the medium interface. This interpretation is unique for cell to medium only. Therefore, to have cells weakly aggregate to one another, we must set the adhesion to just a bit less than 2×26 (52). B to any cell types have the weakest adhesion in the signaling network and appropriately are set to 49 (*see* **Note 8**).

In general, adhesion values between cell types will depend on the signaling network modeled. As a broad rule, we find that the values themselves are a range and the hierarchy of type-to-type adhesion to be far more important. In the reference signaling network, we set a hierarchy based on the adhesion protein. High E-cadherin (H.ECAD) cells logically will prefer to adhere to high E-cadherin cells over low E-cadherin (L.ECAD). L.ECAD cells logically will prefer to adhere to other E-cadherin-bearing cells rather than non-cadherin-bearing cells. Therefore, our hierarchy should be G-G is more adhesive than R-R which is more adhesive than Y-Y or B-B. Cross-interacting values like G-Y will lie somewhere in between.

To add more cells, simply replace the X and Z in this line "<Energy Type1="X" Type2="Z">NUMBER</Energy>" with the appropriate type names and the number with the adhesion strength desired. Then, place the line under the Contact Plugin section with the other similar lines (*See* **Note 9**). With this, we have established how our cells adhere to one another (Fig. 3b).

3.4 Adding Motility to ISCL

With cell adhesion defined, we can move over to another basic cell feature, motility. In CC3D, motility can be specified on the individual cell level via the line "cell.fluctAmpl." These lines are found under the #Defining and Calculating Cell Physical Properties Code/Parameters section in the "ELUGMSteppables.py" file. The first instance of this line will be "cell.fluctAmpl=BASAL +SCF*(CtoM*CSAM+YtoY*CSAY+YtoG*CSAG+YtoB*CSAB +YtoR*CSAR)/cell.surface" appearing in the block of code in Fig. 4a. This block effectively states that, if the focal cell is of type 1 (type name Y/gray cells), then set its motility according to the given formula. Similar blocks of code exist for all the other cell types. This formula calculates motility based on a baseline motility BASAL and neighbors with weight SCF. The weight defines how

A

```
#Defining and Calculating Cell Physical Properties Code/Parameters
            if cell.type==1:
                cell.lambdaSurface=2.2
                cell.lambdaVolume=2.2
                NUMTY+=1
                cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+YtoY*CSAY+YtoG*CSAG+YtoB*CSAB+YtoR*CSAR)/cell.surface
                YGPTS+=cell.dict["PTS"][0]
```

B

```
#Iterating Over Neighbors Code
             for neighbor, commonSurfaceArea in self.getCellNeighborDataList(cell):
                if neighbor is None:
                    continue
                if neighbor.type==1:
                    CSAY+=commonSurfaceArea
                    PTSY+=0
                if neighbor.type==2:
                    CSAG+=commonSurfaceArea
                    PTSG+=commonSurfaceArea*neighbor.dict["PTS"][0]/(neighbor.surface)
                if neighbor.type==3:
                    CSAB+=commonSurfaceArea
                    PTSB+=commonSurfaceArea*(CONEXPSCF/(1+math.exp(-(mcs-THETA)/XI)))/neighbor.surface
                if neighbor.type==4:
                    CSAR+=commonSurfaceArea
                    PTSR+=commonSurfaceArea*(CONEXPSCF/(1+math.exp(-(mcs-THETA)/XI)))/neighbor.surface

            CSAM=cell.surface-(CSAY+CSAG+CSAB+CSAR)
```

C

```
#Calling Adhesion Values From the XML File Code
        global YtoY,YtoG,GtoY,YtoB,BtoY,YtoR,RtoY,GtoG,GtoB,BtoG,GtoR,RtoG,BtoB,BtoR,RtoB,RtoR
        YtoY=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','Y','Type2','Y']))
        YtoG=GtoY=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','Y','Type2','G']))
        YtoB=BtoY=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','Y','Type2','B']))
        YtoR=RtoY=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','Y','Type2','R']))
        GtoG=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','G','Type2','G']))
        GtoB=BtoG=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','G','Type2','B']))
        GtoR=RtoG=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','G','Type2','R']))
        BtoB=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','B','Type2','B']))
        BtoR=RtoB=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','B','Type2','R']))
        RtoR=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','R','Type2','R']))
```

Fig. 4 Adding motility to ISCL. (**a**) Motility is defined in section #Defining and Calculating Cell Physical Properties Code/Parameters of the steppable.py file "ELUGMSteppable.py." In this example, block of code for gray cells (determined by line "if cell.type==1:"), motility is defined as "cell.fluctAmpl" calculated with the formula shown. This formula calculates motility as a sum of baseline adhesion (BASAL) and motility affected by adhesive neighbors. (**b**) To determine the values of the parameters in the motility/cell.fluctAmpl formula, we must calculate the common surface area the focal cell shares with each cell type. This block of code, for a focal cell, iterates over its neighbors and tallies to total surface area shared with each cell type. For example, this is done for green neighbors via the line "CSAG+=commonSurfaceArea." (**c**) The other parameters (i.e., YtoY, YtoG, etc.) are determined by the cell-cell adhesion specified in the Contact Plugin of the xml file. To call these values from the xml plugin, we define several variables and assign them to the appropriate xml value through the lines shown. Once assigned, these variables can be referred to in the steppable.py file

well its neighbors' attenuate motility and we built the formula with the logic that cells with more adhesive neighbors are less likely to move. Dividing by the cell's surface area, cell.surface, normalizes the attenuation. BASAL, SCF, and CtoM (cell to medium) are constant values that are the same across all cells in this signaling network (*see* **Note 10**). They can therefore be found specified under the section #Motility Parameters at the top of the steppables.py file. The other variables are calculated on the individual cell level and will be examined subsequently.

The variables CSAM, CSAY, CSAG, CSAB, and CSAR refer to the common surface area the focal cell shares with the medium and neighbors of type gray, green, blue, and red, respectively. These variables are defined in the #Defining Per Step Per Cell Variables for Signaling and Motility Code section, and values are calculated in the #Iterating Over Neighbors Code section. In the former, we define each variable as 0 and then modify it from the calculations in the latter section. In the #Iterating Over Neighbors Code section, each focal cell has its neighbors iterated over and neighbor type identified (Fig. 4b). For instance, the surface area the focal cell shares with cells of type 1 is totaled by the line "CSAY+=commonSurfaceArea." Once calculated, the values are then used in the motility code. To add additional cell types, for example, cell type 5 with name O, the variable CSAO should be added to the #Defining Per Step Per Cell Variables for Signaling and Motility Code section with line CSAO = 0. Next, add this cell type to the #Iterating Over Neighbors Code section with lines such as

```
"if neighbor.type==5:
 CSAO+=commonSurfaceArea"
```

Also, modify the line "CSAM=cell.surface-(CSAY+CSAG +CSAB+CSAR)" to "CSAM=cell.surface-(CSAY+CSAG+CSAB +CSAR+CSAO)" as O cells can now also contribute to shared surface area.

The other variables YtoY, YtoG, YtoB, and YtoR refer to the focal cell's adhesion to the type of the neighbor. As our focal cell is of type 1 (Y), these variables refer to Y-Y, Y-G, Y-B, Y-R adhesion strengths we previously defined in the adhesion section of the xml file. To call these values from the xml file, we use the code located in the #Calling Adhesion Values from the XML File Code section. To begin, we define the names of the variables in the line "global YtoY, YtoG,GtoY,YtoB,BtoY,YtoR,RtoY,GtoG,GtoB,BtoG,GtoR, RtoG,BtoB,BtoR,RtoB,RtoR" (Fig. 4c). Global tells CC3D these variables are the same anywhere in the simulation. Next, we call the values directly from the xml file using the line "float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','X','Type2','Z']))." This line tells the steppable.py file to obtain the adhesion values between cells of type X and Z from the

Contact Plugin in the xml file. For example, the line "YtoG=GtoY=float(self.getXMLElementValue(['Plugin','Name','Contact'],['Energy','Type1','Y','Type2','G']))" defines the variables YtoG and GtoY as the same adhesion value of between types Y and G (40) in the xml file (*See* **Note 11**).

To add additional cell types, again with cell type 5 with name O as an example, its adhesion can be called into the steppables.py file with similar lines. Do not forget to first define the variable name in the global line before assigning it a value.

With the common surface area calculated and the adhesion values called from the xml file, cell motility is calculatable. For adding additional cell types, we will give an example with cell type 5 with name O. For example, in the #Defining and Calculating Cell Physical Properties Code/Parameters, for cells of type 1 (denoted by line "if cell.type==1:"), "cell.fluctAmpl" can be changed to "cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+YtoY*CSAY +YtoG*CSAG+YtoB*CSAB+YtoR*CSAR+YtoO*CSAO)/cell. surface" if example cell type 5 with name O was added. This change will need to be repeated for each cell type defined in this section #Defining and Calculating Cell Physical Properties Code/Parameters. We will also need to specify a block for cell type O in this section as well. For example, a block such as

"if cell.type==5:

cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+OtoY*CSAY +OtoG*CSAG+OtoB*CSAB+OtoR*CSAR+OtoO*CSAO)/cell. surface". In the later sections, we go over the other lines in the block (Fig. 4a) and give examples on how they are added to this block.

3.5 Adding Size, Growth, and Division to ISCL

In CC3D, cells can be programmed at the individual cell level to target a surface area and volume through the variables "cell.targetSurface" and "cell.targetVolume." For example, a "cell.targetSurface" of 113 and a "cell.targetVolume" of 113 tell the cells to try to maintain a surface area and volume of 113 pixels. How well the cell adheres to these targets is dependent on the parameters "cell.lambdaSurface" and "cell.lambdaVolume," respectively. These lambda parameters can be thought of as spring constants and dictate how well the cell can deviate from its assigned target values. A higher value constrains the cell closer to its target surface/volume, while a lower value allows the cell to deviate more from the target surface/volume.

In this framework, we chose to create and specify a target radius for each cell, and targetSurface and targetVolume are calculated from this target radius through the classic formulas for the surface area and volume of a sphere, respectively. This can be found in the section #Initialization of Cells Parameters with the lines: "cell.dict ["RDM"]=RNG.gauss(RADAVG,RADDEV)," "cell.

A

```
#Initialization of Cells Parameters
        for cell in self.cellList:
            cell.dict["RDM"]=RNG.gauss(RADAVG,RADDEV)
            cell.lambdaSurface=2.5
            cell.targetSurface=4*math.pi*cell.dict["RDM"]**2
            cell.lambdaVolume=2.5
            cell.targetVolume=(4/3)*math.pi*cell.dict["RDM"]**3
            cell.dict["PTS"]=[0]
```

B

```
#Mitosis Codes
from PySteppablesExamples import MitosisSteppableBase

class MitosisSteppable(MitosisSteppableBase):
    def __init__(self,_simulator,_frequency=1):
        MitosisSteppableBase.__init__(self,_simulator, _frequency)
        self.setParentChildPositionFlag(0)
    def step(self,mcs):
        cells_to_divide=[]
        for cell in self.cellList:
            cell.dict["RDM"]+=RNG.uniform(MTFORCEMIN,MTFORCEMAX)
            cell.targetSurface=4*math.pi*cell.dict["RDM"]**2
            cell.targetVolume=(4/3)*math.pi*cell.dict["RDM"]**3
            if cell.volume>2*(4/3)*math.pi*RADAVG**3:
                cells_to_divide.append(cell)

        for cell in cells_to_divide:
            self.divideCellRandomOrientation(cell)

    def updateAttributes(self):
        self.parentCell.dict["RDM"]=RNG.gauss(RADAVG,RADDEV)
        self.parentCell.targetVolume=(4/3)*math.pi*self.parentCell.dict["RDM"]**3
        self.parentCell.targetSurface=4*math.pi*self.parentCell.dict["RDM"]**2
        self.cloneParent2Child()
```

C

```
#Defining and Calculating Cell Physical Properties Code/Parameters
            if cell.type==1:
                cell.lambdaSurface=2.2
                cell.lambdaVolume=2.2
                NUMTY+=1
                cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+YtoY*CSAY+YtoG*CSAG+YtoB*CSAB+YtoR*CSAR)/cell.surface
                YGPTS+=cell.dict["PTS"][0]

            if cell.type==2:
                cell.lambdaSurface=1.0
                cell.lambdaVolume=1.0
                NUMTG+=1
                cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+GtoY*CSAY+GtoG*CSAG+GtoB*CSAB+GtoR*CSAR)/cell.surface
                YGPTS+=cell.dict["PTS"][0]

            if cell.type==3:
                cell.lambdaSurface=2.2
                cell.lambdaVolume=2.2
                NUMTB+=1
                cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+BtoY*CSAY+BtoG*CSAG+BtoB*CSAB+BtoR*CSAR)/cell.surface
                BRPTS+=cell.dict["PTS"][0]

            if cell.type==4:
                cell.lambdaSurface=1.0
                cell.lambdaVolume=1.0
                NUMTR+=1
                cell.fluctAmpl=BASAL+SCF*(CtoM*CSAM+RtoY*CSAY+RtoG*CSAG+RtoB*CSAB+RtoR*CSAR)/cell.surface
                BRPTS+=cell.dict["PTS"][0]
```

Fig. 5 Adding physical dimensions and division into ISCL. (**a**) In the #Initialization of Cells Parameters section, we initialize cells with a target radius, which is then used to calculate the target surface area and target volume. Starting cells with a distribution in radius prevents mitosis from synchronizing and can represent a cell line with cells in various stages of the cell cycle. The two lambda values here are temporary and are reassigned on the cell type level in (**c**) below. Cells are also initialized with a dictionary named "PTS" short for points. This dictionary is used to track reporter levels. (**b**) In the mitosis section of the steppable.py file, cells undergrow random fluctuations in their target radius, with bias toward increasing to cause cells to grow. When

targetSurface=4*math.pi*cell.dict["RDM"]**2," and "cell.targetVolume=(4/3)*math.pi*cell.dict["RDM"]**3" (Fig. 5a). The first line assigns each cell a radius drawn from a Gaussian distribution with mean RADAVG and standard deviation RADDEV. These parameters are specified in the #Cell Size and Division Parameters section at the top of the steppables.py file. The two subsequent lines calculate the targetSurface and targetVolume based on this target radius.

With target radius defined, we can also add in a growth and division model. Navigate to the #Mitosis Codes section in the steppables.py file, which effectively contains all the growth and division codes (Fig. 5b). Here, cells experience changes in their target radius through the line cell.dict["RDM"]+=RNG.uniform (MTFORCEMIN,MTFORCEMAX). These changes are drawn from a uniform distribution with minimum MTFORCEMIN and maximum MTFORCEMAX. These parameters can be found in the #Cell Size and Division Parameters section. Once target radius is adjusted, the new targetSurface and targetVolume are calculated, and if the cell exceeds twice the volume calculated using the mean (RADAVG), it undergoes division. Upon dividing, cells are assigned a new target radius and repeat the cycle.

"Cell.lambdaSurface" and "cell.lambdaVolume" are specified in the section #Defining and Calculating Cell Physical Properties Code/Parameters. They are also defined in the section #Initialization of Cells Parameters, but the value there is rapidly overwritten. They are included, so CC3D does not throw an error when trying to initialize the simulation. Now, navigate to the #Defining and Calculating Cell Physical Properties Code/Parameters section, where the actual values are specified via the lines "cell.lambdaSurface" and "cell.lambdaVolume" (Fig. 5c). Notice that their values can be specified on a per cell type basis but like the "cell.fluctAmpl" motility line, which could also be specified on an individual cell level via a formula if desired (*see* **Note 12**).

In this implementation of size, growth, and division, there are minimal changes to make here if an additional cell type is added. Again, with example cell type 5 with name O, cell.lambdaSurface and cell.lambdaVolume would need to be specified in the #Defining and Calculating Cell Physical Properties Code/Parameters

Fig. 5 (continued) cells reach twice the expected cell volume, calculated as the volume using the mean of the population, cells undergo division. They are then reassigned a new target radius along with a new calculated target surface area and target volume. (**c**) Here, cells have their lambdaSurface and lambdaVolume specified on the per cell type level. For example, as G cells are adhesive and change their morphology when adhesive [12], we relax their lambdas to 1. Notice this is the same section for the motility code. In general, this section is useful for specifying parameters on a per cell type level

section. Then, a complete block for a cell type, excluding quantification and signaling, will have cell.fluctAmpl (added in the motility section above), cell.lambdaSurface, and cell.lambdaVolume in this implementation. Figure 4a serves as an example.

3.6 Adding Genetic Circuits to ISCL: Assigning and Tracking Reporter Levels in Cells

With ISCL and its physical properties defined, we can now add the genetic circuits that control the changes in adhesion in the reference signaling network. To start, navigate to the #Initialization of Cells Parameters section (Fig. 5a). Here, we specify that each cell has a dictionary list attached to it. For example, through the line "cell.dict["PTS"]=[0]," we assign each cell a dictionary called "PTS" (for points) with one numerical value that will keep track of the level of reporter (and thus effect of signaling) in each cell. For our gray Y and green G cells, this dictionary will track GFP reporter levels, while for our blue B and red R cells, it will track RFP reporter levels. The value 0 here reflects that cells begin with 0 reporter as in the reference signaling network (*See* **Note 13**).

3.7 Adding Genetic Circuits to ISCL: Defining the Signal from Ligands on Neighbor Cells

With reporter tracking enabled and initial reporter condition defined, we will now calculate the ligand amount a focal cell is exposed to per simulation timestep. We first define these variables in the section #Defining Per Step Per Cell Variables for Signaling and Motility Code. PTSY, PTSG, PTSB, and PTSR refer to points from gray, green, blue, and red cells, respectively. Note that this is similar to the code we are using to calculate motility for each cell.

Next, navigate to the #Iterating Over Neighbors Code section (Fig. 6a). Just as how we totaled the surface area the focal cell shares with cells of a specific type, we now also total the ligands the focal cell is exposed to from cells of a specific type. In the reference signaling network (Fig. 1b), gray cells do not have ligands that signal to neighbors. Therefore, they contribute 0 points via the code "PTSY+=0" (*see* **Note 14**).

Now, turn to the line under "if neighbor.type==2:" (recall that type 2 is green cells with type name G), and look at the green points code "PTSG+=commonSurfaceArea*neighbor.dict["PTS"][0]/(neighbor.surface)." In the signaling network, green cells signal via a reporter that doubles as a ligand (GFP-lig) and is thus tracked by our "PTS" dictionary. To calculate how much reporter-ligand our focal cell is exposed to, we call upon the total reporter level of the neighboring cell via "neighbor.dict["PTS"][0]," and then, divide it by the neighbor's surface area to obtain a density. We then multiply by the focal cell's shared surface area to effectively get the number of ligands that contact our focal cell. The += operation continuously adds to PTSG for each neighbor that is a green cell.

A similar line occurs in the subsequent code for PTSB and PTSR. However, as blue and red cells signal via a constitutive ligand (not induced or affected by the genetic circuit), we instead have

A

```
#Iterating Over Neighbors Code
            for neighbor, commonSurfaceArea in self.getCellNeighborDataList(cell):
                if neighbor is None:
                    continue
                if neighbor.type==1:
                    CSAY+=commonSurfaceArea
                    PTSY+=0
                if neighbor.type==2:
                    CSAG+=commonSurfaceArea
                    PTSG+=commonSurfaceArea*neighbor.dict["PTS"][0]/(neighbor.surface)
                if neighbor.type==3:
                    CSAB+=commonSurfaceArea
                    PTSB+=commonSurfaceArea*(CONEXPSCF/(1+math.exp(-(mcs-THETA)/XI)))/neighbor.surface
                if neighbor.type==4:
                    CSAR+=commonSurfaceArea
                    PTSR+=commonSurfaceArea*(CONEXPSCF/(1+math.exp(-(mcs-THETA)/XI)))/neighbor.surface
```

B

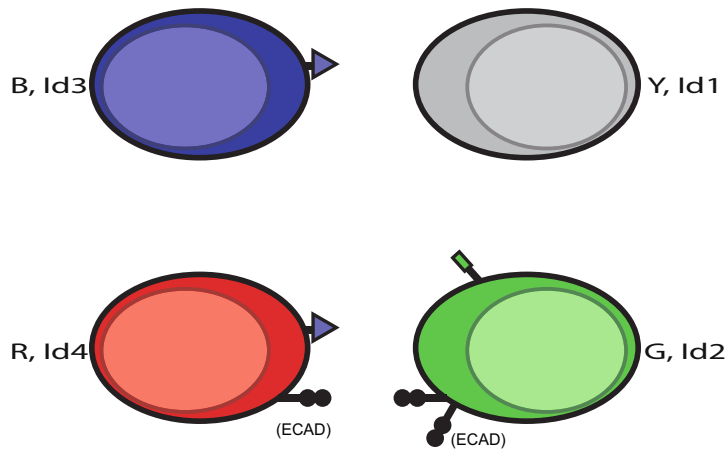

Fig. 6 Determining the level of ligand, each cell is exposed to. (**a**) To determine what ligand and how much of that ligand the focal cell is exposed to, we turn to the #Iterating Over Neighbors Code. Here, each cell in the signaling network is iterated over and its neighbors assessed. For neighbors of each cell type (gray, green, blue, red), the variables PTSY, PTSG, PTSB, and PTSR are calculated via the formula shown. Effectively, each variable totals the ligand level from one cell type by assessing the total ligand on a neighboring cell of the specified type, dividing by the neighbor's surface area, and then by multiplying by the focal cell's contacted surface area. This is repeated on all neighboring cells of that type. This process is then repeated for each cell type. (**b**) With this section of code, the ligand type on each cell is specified along with the type and level of ligands each cell in the simulation is exposed to

their total ligand calculated by the sigmoid equation "(CONEXPSCF/(1+math.exp(-(mcs-THETA)/XI)))." These parameters are constant and are defined in the #Constitutive Ligand Parameters section at the top of the steppables.py file (*see* **Note 15**). Just like with neighbors of type 2 (green cells), the amount of ligand contacting our focal cell is again calculated by dividing by neighbor surface and multiplying by the amount of surface contacting our focal cell.

If additional cell types with signaling ligands are to be added, this is the section to do it. Again, using the example cell type 5 with

name O, the "if neighbor.type==5:" line would be added along with a PTSO line underneath (*see* **Note 16**). This completes defining ligands on cell types for our reference signaling network (Fig. 6b).

3.8 Adding Genetic Circuits to ISCL: Determining How the Focal Cell Responds to Signal from Ligands on Neighbors

Now that we know the type and how much ligand our focal cell is exposed to from each cell type, the next step is to determine how the focal cell responds to them. Navigate to the section #Changing Reporter as a Result of Signaling Changes. Here, we encode which and how cells process the ligands as a signal. We first notice, from our reference signaling network (Fig. 1b), that gray and green cells receive signal from blue and red cells (triangle ligand signaling to notched receptor) to express H.ECAD (high E-cadherin) and GFP-lig (GFP-ligand). We code this underneath "if (cell.type==1 or cell.type==2):" with the indented line: "DTRES=(1/(ALPHAYG+math.exp(-((PTSR+PTSB)-BETAYG)/EPSILONYG)))-(1/KAPPAYG)*cell.dict["PTS"][0]" (Fig. 7a). This line calculates DTRES, the change (delta) in reporter (GFP-ligand in this case) as a result of signaling (*see* **Note 17**). Note that DTRES is first defined in the same code block as PTSY. The constant parameters (ALPHAYG, BETAYG, EPSILONYG, KAPPAYG) for this formula are found in the section #YG Signaling Parameters at the top of the steppable.py file.

Once DTRES is calculated, it is added to the PTS dictionary via the line "cell.dict["PTS"][0]+=DTRES," effectively documenting the change in total reporter as a result of signaling at a timestep. Notice that we only specify PTSB and PTSR in this DTRES formula for gray and green cells. This is because in our reference signaling network, only blue and red cells signal to gray and green cells. Thus, this formula for DTRES not only calculates the change in reporter because of signaling but also specifies what ligands the cell should respond to.

We repeat this code for the blue and red cells, first with the "if (cell.type==3 or cell.type==4):" line. However, the change in reporter, DTRES, is calculated only from PTSG as only green cells can signal to blue or red cells (Fig. 1b). Once DTRES is calculated, the cell reporter level is updated in the PTS dictionary. Notice that the parameters are different per genetic circuit. Gray and green cells have their own parameters, while blue and red cells have their own parameters (i.e., BETAYG vs BETABR; *see* **Note 18**).

With this, we complete the ligand-receptor interactions in our signaling network (Fig. 7b). If desired, additional cell types that respond to a ligand in the signaling network can be added in this #Changing Reporter as a Result of Signaling Changes section via similar lines.

A

```
#Changing Reporter as a Result of Signaling Changes
            if (cell.type==1 or cell.type==2):
                DTRES=(1/(ALPHAYG+math.exp(-((PTSR+PTSB)-BETAYG)/EPSILONYG)))-(1/KAPPAYG)*cell.dict["PTS"][0]
                cell.dict["PTS"][0]+=DTRES
            if (cell.type==3 or cell.type==4):
                DTRES=(1/(ALPHABR+math.exp(-((PTSG)-BETABR)/EPSILONBR)))-(1/KAPPABR)*cell.dict["PTS"][0]
                cell.dict["PTS"][0]+=DTRES
```

B

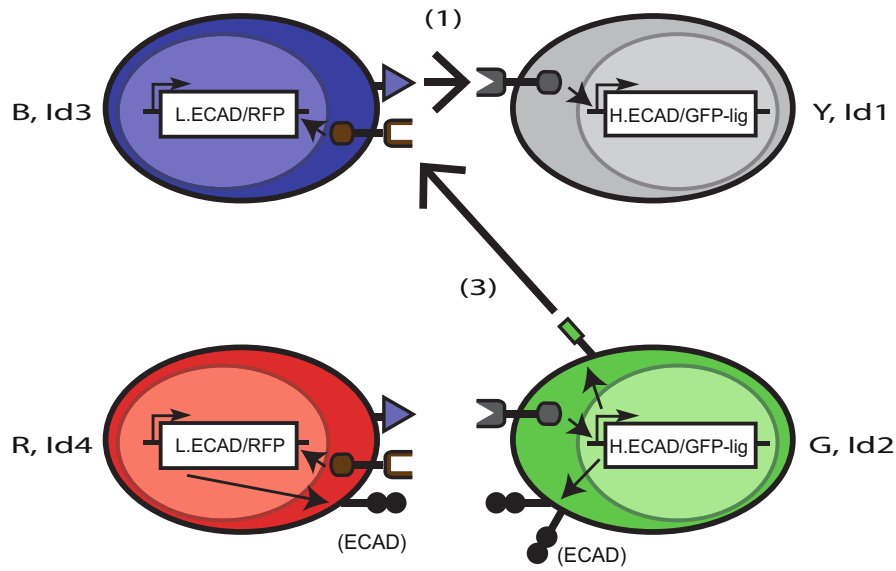

Fig. 7 Specifying ligand-receptor signaling and calculating the change in reporter due to signaling. (**a**) The section #Changing Reporter as a Result of Signaling Changes specifies which cells receive signal from what cell types. In this block, cells of type 1 or type 2 (gray or green), receive signal (PTSB+PTSR) from cells of type 3 or type 4 (blue or red), as determined by the notched synNotch receptor and triangle ligand, respectively. A similar process occurs for cells of type 3 or type 4, except they only receive signaling from green cells, hence PTSG. This signal is then fed into the formula shown to calculate DTRES, the change in reporter at this timestep as a result of signaling. This change is then added to the "PTS" dictionary as it tracks reporter level. (**b**) This section completes the ligand-receptor signaling and links it to the reporter levels crucial for tracking change in adhesion

3.9 Adding Genetic Circuits to ISCL: Changing the State of the Cell as a Result of Reporter Accumulation

Our signaling network is nearly complete. We can finish it by adding the last piece, the state transition code. In our framework, cells transition discretely to the activated state (gray to green and blue to red), representing changing from weakly/nonadhesive to adhesive via their representative cadherins. They can only transition when they have enough reporter, as it is a proxy for tracking their adhesion protein levels. We code this in the section #Changing State as a Result of Reporter Changes (Fig. 8a). As gray cells can

A

```
#Changing State as a Result of Reporter Changes
            if cell.type==1:
                if cell.dict["PTS"][0]>=THRESHOLDUPYG:
                    cell.type=2
            if cell.type==2:
                if cell.dict["PTS"][0]<THRESHOLDDOYG:
                    cell.type=1
            if cell.type==3:
                if cell.dict["PTS"][0]>=THRESHOLDUPBR:
                    cell.type=4
            if cell.type==4:
                if cell.dict["PTS"][0]<THRESHOLDDOBR:
                    cell.type=3
```

B

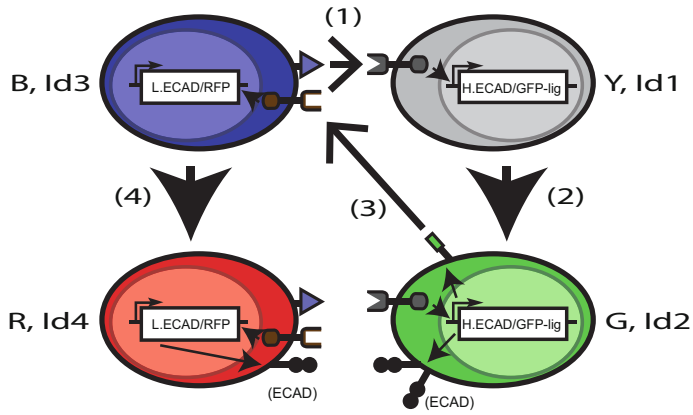

Fig. 8 Changing cell type/state as a result of reporter accumulation. (**a**) As reporter expression is linked to adhesion protein expression in our reference signaling network, once cells have accumulated enough reporter, they activate to the adhesive state. Here, the adhesive states are specified as cell types 2 and 4 (green and red) as previously defined in the cell-cell adhesion values. Therefore, for cells of type 1 or type 3 (gray or blue), if their reporter level exceeds the threshold, then change cell type to type 2 or type 4, respectively. Conversely, cells of type 2 or type 4 could lose reporter without signaling and revert to the nonadhesive state, 1 or 3, respectively. To code this, we specify that if these cells are less than the threshold, then they revert. (**b**) This is the last piece needed to complete the genetic circuits in the reference signaling network. All that remains is to initialize the cells and run the simulation

become green, we specify with the code "if cell.type==1:", and if the PTS dictionary (tracking reporter level) is greater than a specified threshold (parameter THRESHOLDUPYG), then change the cell type to 2 (green), effectively switching it from weak adhesion to H.ECAD expressing with GFP-lig. Likewise, a green cell could deactivate, and this is coded in the lines below. "if cell.type==2:",

and if the PTS dictionary is less than a specified threshold (parameter THRESHOLDDOYG), then change the cell type to 1, effectively removing its H.ECAD and GFP-lig. We repeat this for blue and red cells as they can transition between each other and note that these threshold parameters can be different between cells of different genetic circuits. These parameters can be found in the #YG Signaling Parameters and #BR Signaling Parameters sections. Additional thresholds for added cell types can be specified here. With this, we have coded the genetic circuits that comprise the reference signaling network (Fig. 8b).

3.10 Setting the Initial Simulation State

With the genetic circuits complete, we can now code the initial condition of the developmental trajectory. Navigate back to the xml file and find the line: "<Steppable Type="BlobInitializer">" (Fig. 9). We use this initializer as cells initialize as a mixture of both gray and blue cells in the reference signaling network (Fig. 1a). Here, the blob initializes centered at coordinates $x = y = x = 50$ (*see* **Note 19**). With a blob radius of 20 pixels, 0 pixels as gaps between cells, and a width of 5, cells initialize as a cube of $5 \times 5 \times 5$ pixels (*See* **Note 20**). The line "<Types>B,Y,B,B,B</Types>" controls the ratio of cells. Here, we initialize with a 4B to 1Y cell which generates cells with a ratio similar to that of the reference experiment (*See* **Note 21**). This generates the initial cell mixture required for the reference signaling network (Fig. 9).

```
<Steppable Type="BlobInitializer">
   <Region>
      <Center x="50" y="50" z="50"/>
      <Radius>20</Radius>
      <Gap>0</Gap>
      <Width>5</Width>
      <Types>B,Y,B,B,B</Types>
   </Region>
</Steppable>
```

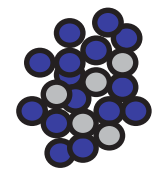

Fig. 9 Initializing the initial simulation conditions. The "Blobinitializer" steppable in the xml file allows specifying an initial blob of cells with a chosen radius for the blob. Additional options include the gap between cells and the initial cell dimensions. Cells here initialize as a cube of $5 \times 5 \times 5$ pixels and, with the target radius implementation, converge to spherical rapidly. The cell types are initialized, and the ratio of the types can also be specified here

3.11 Technical Simulation Parameters

Before starting the simulation, a few technical parameters need to be examined. Navigate back to the steppable file and look at the parameters ENDMCS under #EndTime Parameters and RESOL, USEDNODEs under #Sampling and Computational Threads Parameters. The ENDMCS parameter calls the end of the simulation here directly via the line: "self.stopSimulation()." Here is set to end the simulation at 20000 timesteps and write the quantification files at this timestep (*see* **Note 22**).

RESOL is short for resolution and determines how often the quantification values are plotted. At 100, the values are plotted at every 100th timestep. Quantifications are optional and are discussed below. USEDNODES dictates how parallelized your simulation will be. For the lattice size here ($100 \times 100 \times 100$), specified in the xml line "<Dimensions x="100" y="100" z="100"/>," eight threads offer a good balance of speed and computational use. You will want to change this depending on your computer. For example, if your computer only has four cores and eight threads, CC3D will try to use all threads, leaving none for your background processes and OS on your computer. In this example, two or four may be a better parameter for USEDNODES.

3.12 Running the Simulation

Finally, it is time to configure our CC3D player. Click the CC3D globe icon in Twedit++ (the big globe under the tab CC3D Project). The CC3D Player should open up. Navigate to the "Tools" tabs and then click "Configuration…". Now, click the "Output" tab. If you want to save cross-sectional images, click the "save images every Nth MCS" and specify at what timesteps you want to save images at. For example, 100 would save images every 100th timestep. If you want to save lattice files, which allow you to reexamine the full 3D structure, click the "Save lattice every Nth MCS" and select how often you want to save it. For the reference signaling network, saving the lattice every 500 timestep is a good parameter. This will output vtk files in the output folder as the simulation runs. You can play these files in the CC3D player by navigating to the "File" tab in the player, then "Open Lattice Description Summary File…", and then open, in the "LatticeData" folder in the output folder, the "StepLDF.dml" file.

To change the color of our simulated cells to match what we want in the reference signaling network, navigate to the "Tools" tab in CC3D player and then click "Configuration…". Instead of clicking the "Output" tab, click the "Cells/Colors" tab. You'll notice the cell type number (i.e., 1, 2, 3, 4) are specified along with a color. Change the color to match the type. Type 1 is gray, type 2 is green, type 3 is blue, and type 4 is red in our reference signaling network. This is a nice option as you can recolor cells if you decide to change colors later. Just make sure to change it in the signaling network as well.

To run the simulation, go to "File" tab in the player, then "Open Simulation File(.cc3d)," then go to our project folder which will be "ChapterDemo," and then open the CC3D file (also named "ChapterDemo"). Then, click the triangle play button. You will also notice, in addition to the structure cross-section window, that two additional plots pop up (*see* **Note 23**). See the below sections as these are additional quantifications we have left in the code. They are useful for monitoring the signaling network but are not necessary and can be skipped if not needed.

3.13 Optional: Quantifications for Monitoring the Signaling Network

In the code, there are a few quantifications that should help track and troubleshoot the signaling network. These quantifications are plotted per the rate of RESOL via the code in the section #Graph the Quantification Code (Fig. 10a). In the first block of lines, self.pw1 specifies the creation of a plot number 1 with title "Types" and *x*-axis and *y*-axis labels as written. Below, the line "self.pW1.addPlot ('Y', _style='Dots', _color='gray', _size=3)" tells CC3D to add to plot number 1 the value of label Y. Y here is the number of gray cells at the timestep and the logic follow for the other letters. We show how a value is assigned to label Y below. Similar to plot number 1, plot number 2 specifies a graph labeled Point System and plots the values of the labels YG and BR.

To determine how these labels are linked to plot values, navigate to the section #Defining Per Step Variables Required for Quantifications Code. Here, we define some variables, initializing them with value 0, and changing them as cells are counted in the timestep. For example, NUMTY is the number of type Y cells, and at the beginning of each timestep, it is reset to 0. The same occurs for the other variables like YGPTS as well.

Now, navigate to section #Defining and Calculating Cell Physical Properties Code/Parameters (Fig. 5c). We see if cell type is 1, it tallies 1 for NUMTY, and if cell type is 3, it tallies 1 for NUMTB. This is effectively a counter-totaling the number of each cell type at each timestep. Likewise, for gray and green cells (type 1 and type 2, respectively), we total the GFP-lig reporter levels via the variable YGPTS, and for blue/red cells, we total the RFP points (BRPTS). After tallying over all the cells in the simulation, the values are either directly plotted or undergo further calculations before plotting. This is in the section #Adding Data Points to Graphs Code (Fig. 10b).

The line "if mcs%RESOL==0:" means if the timestep number (i.e., 10,000), which is divisible by 100, plot the value to the corresponding graph. For example, "self.pW1.addDataPoint("Y", mcs, NUMTY)" adds for the label Y in plot number 1, the point (timestep, value of NUMTY). This repeats for all the cell types that we have and allows tracking the number of cells of each type. The second plot tracks average reporter levels. For example, "self.pW2.addDataPoint("YG", mcs, YGPTS/(NUMTY+NUMTG)" adds

A

```
#Graph the Quantification Code
        self.pW1 = self.addNewPlotWindow(
            _title='Types',
            _xAxisTitle='MonteCarlo Step (MCS)',
            _yAxisTitle='Count',
            _xScaleType='linear',
            _yScaleType='linear',
            _grid=True)
        self.pW1.addPlot('Y', _style='Dots', _color='gray', _size=3)
        self.pW1.addPlot('G', _style='Dots', _color='green', _size=3)
        self.pW1.addPlot('B', _style='Dots', _color='blue', _size=3)
        self.pW1.addPlot('R', _style='Dots', _color='red', _size=3)

        self.pW2 = self.addNewPlotWindow(
            _title='Point System',
            _xAxisTitle='MonteCarlo Step (MCS)',
            _yAxisTitle='Count',
            _xScaleType='linear',
            _yScaleType='linear',
            _grid=True)
        self.pW2.addPlot('YG', _style='Dots', _color='green', _size=3)
        self.pW2.addPlot('BR', _style='Dots', _color='blue', _size=3)
```

B

```
#Adding Data Points to Graphs Code
        if mcs%RESOL==0:

            self.pW1.addDataPoint("Y", mcs, NUMTY)
            self.pW1.addDataPoint("G", mcs, NUMTG)
            self.pW1.addDataPoint("B", mcs, NUMTB)
            self.pW1.addDataPoint("R", mcs, NUMTR)

            self.pW2.addDataPoint("YG", mcs, YGPTS/(NUMTY+NUMTG))
            self.pW2.addDataPoint("BR", mcs, BRPTS/(NUMTB+NUMTR))
```

C

```
#Exporting the Graphs as Data in Text Files Code
            if mcs==ENDMCS:
                fileName = "FOU" + str(mcs) + ".txt"
                self.pW1.savePlotAsData(fileName)
                fileName = "SIG" + str(mcs) + ".txt"
                self.pW2.savePlotAsData(fileName)
                self.stopSimulation()
```

Fig. 10 Optional quantifications. (**a**) This block of code initializes plot windows that can be viewed as the simulation progresses. Here, two plots are initialized, one plotting labels Y, G, B, and R which will be used to track the number of each respective cell type. The second plot labels YG and BR will be used to track average GFP and RFP levels, respectively. (**b**) Here, the plotting variables are assigned values. Notice that the variables can be a variable within the code or a calculation of several variables. (**c**) At the ENDMCS, which is a parameter defined in the steppable.py file, the simulation writes the plots as text files with the name specified. These text files can be used for further analyses

for the label YG in plot number 2 the point (timestep, value of YGPTS/(NUMTY+NUMTG). This effectively allows us to track at every 100th timestep, the average level of GFP-ligand over all gray and green cells.

The #Exporting the Graphs as Data in Text Files Code section (Fig. 10c) outputs these plots as data files at the end of the simulation, determined by the parameter ENDMCS. The names of these files can be specified per plot. For example, for plot number 1, it is FOU, and for plot number 2, it is SIG.

3.14 Optional: Parameterscan

Being able to scan parameters is key to testing signaling networks and fortunately, these are simple to add in CC3D. The project code already has one added, but if it is deleted, it can be added back by right-clicking a file name (i.e., "ELUGM.xml") in the left panel where all file names are displayed and then clicking "Add parameter scan." To scan a variable (or multiple variables), right-click the file name in that same left panel and click "Open Scan Editor." The file opened should contain the variable you want to be scanned. For example, to scan Y-Y adhesion, right-click the "ELUGM.xml" file and click "Open Scan Editor." This will create a temporary copy to prevent modification of the original file. It is denoted by "tmp" in the copy name. In this temporary copy, right-click the variable to scan and click "Add To Scan. . .". A window will pop up with the variable selected shown. Next, click "Edit. . .", and input the values desired in the "Values" field. Alternatively, you can have CC3D decide by choosing the minimum, maximum, steps in between, and then click "generate." Regardless of the method, once values are placed into the "Values" field, click "OK." A ParametersScanSpecs.xml file will be created, and you can see what the output folder, which contains all the simulations, one for each scanned value, will be named in the "OutputDirectoy" line. Here, results will be outputted in the folder "ChapterDemo_ParameterScan." You can also see the parameter being scanned and the values scanned. In this code, it is "RandomSeed" with the values "39212,12300,94354, 75020,39844,3312,90755,75759,76434,93784."

4 Notes

1. The framework is implemented in version 3 which is incompatible with the current version of CC3D (version 4). It is important to download the correct version of CC3D to use the code provided. Alternatively, if version 4 CC3D is desired, the framework implementation can be downloaded at https://github.com/calvinlamk/CC3D-4-vs-CC3D-3. It does, however, contain additional codes and labeling, which makes it

difficult to start with. It is recommended to try the version 3 framework with the settings given here to first learn how the framework works.

2. The codes for the other signaling networks in the publication can be found at https://github.com/lmorsut/Lam_Morsut_GJSM. These codes will contain the same framework but also numerous additional lines for quantifications.
3. Here, we use Twedit++ as it conveniently contains access to all files in the code and access to the CC3D simulation player as well. However, other editors like Notepad++ will work just fine to view and edit files as well.
4. The "ELUGM.py" is where steppables can be registered, but this is advanced coding outside of the framework given here.
5. Four cell types were chosen, one per color, but it is possible to introduce additional cell types to capture intermediate states. For example, you can designate a light green cell type to model intermediate levels of high E-cadherin.
6. In general, it is better to have extra cell types established than less cell types. For example, you may want to modify this demo for a simulation with only two cell types Y and G. It is recommended to keep B and R coded in CC3D but not used in the actual simulation. This way, if additional cell types are needed in the future, they can easily be called upon and re-enabled.
7. Alternative plugins for adhesion exist. For example, it is possible to specify the strength on a per adhesion protein level, but this may increase the parameters required for modeling.
8. Cell dispersion, rather than aggregation, may be a desired behavior. For example, signaling could lead to dispersion in cell type R, so in this case, R to cell adhesion would be greater than 52.
9. To check if all the interactions between each cell type are specified, the number of adhesion values should be the summation of the number of cell types. For example, with five cell types (Y, G, B, R, medium), we have 5 + 4 + 3 + 2 + 1 interactions, thus 15 values.
10. Modifying these parameters will enable modeling different cell lines. We built ours to model L929 mouse fibroblasts cells as they were used in the corresponding in vitro experiment [8]. However, modeling less motile cells, neighbor adhesion insensitive cells may be desired and can be adjusted here.
11. We chose to use two variables for the same adhesion interaction as it is easier to keep track of the code when standardized.
12. Like the motility lines, modifying the parameter mentioned in this section will allow modeling different cell lines. For

example, cell division rate can be increased or slowed by modifying MTFORCEMIN or MTFORCEMAX. Size of cells can be modified as well via RADAVG.

13. Many different dictionaries can be added or even extended to track additional quantifications of interest. For example, in a signaling network with three reporters, the dictionary could be defined as cell.dict["PTS"]=[0,0,0]. Initial level of reporter can also be changed as well.
14. This line can be removed, but we kept it here to demonstrate that signaling can be coded several ways.
15. The use of a sigmoid equation here allows tuning the initial ligand level and how quickly steady state/saturation ligand levels are reached on the signaling cells. Alternative equations may be used depending on the modeler's choice.
16. One cell type may have multiple ligands. If this is the case, two ligands can be defined here with different variables names. For example, PTSOONE for ligand 1 on orange cells and PTSOTWO for ligand 2.
17. We use a simple sigmoid equation here as it captures the synNotch juxtacrine signaling well. If other models of signaling dynamics are desired, it should be implemented here.
18. Cells with different genetic circuits can be engineered with different parameters. For example, blue cells may be more difficult to activate than gray cells. Differing parameters can model this, and splitting the parameters per genetic circuit can offer increased flexibility in testing signaling networks.
19. Where the blob of cells is initialized should be chosen carefully. For example, if the cell blob has a radius of 20 pixels and is initialized with its center 10 pixels away from the edge of the lattice, 10 pixels of the radius will be outside the lattice, resulting in cells missing or the simulation crashing.
20. Choose the initial size of your cube cells to be similar in surface area and volume to the target surface area and target volume to have cells quickly get to the desired spherical shape.
21. The assigned ratio may not be as close as desired to the target ratio. It is recommended to test different initial conditions.
22. The end timestep can also be defined in the xml file via the line "<Steps>100000</Steps>." We recommend defining it in the steppables.py file as it can be directly called for when to output the quantification files. If done this way, just set the endpoint in the xml file to some very large number, so it doesn't take precedent over the value defined in the steppables.py file.
23. Simulation speeds can be improved by simply minimizing the CC3D player window, so it does not have to render the images.

References

1. Cheng AA, Lu TK (2012) Synthetic biology: an emerging engineering discipline. Annu Rev Biomed Eng 14:155–178. https://doi.org/10.1146/annurev-bioeng-071811-150118
2. Davies J (2017) Using synthetic biology to explore principles of development. Development 144:1146–1158. https://doi.org/10.1242/dev.144196
3. Hoffman T, Antovski P, Tebon P, Xu C, Ashammakhi N, Ahadian S, Morsut L, Khademhosseini A (2020) Synthetic biology and tissue engineering: toward fabrication of complex and smart cellular constructs. Adv Funct Mater 30:1909882. https://doi.org/10.1002/adfm.201909882
4. Johnson MB, March AR, Morsut L (2017) Engineering multicellular systems: using synthetic biology to control tissue self-organization. Curr Opin Biomed Eng 4:163–173. https://doi.org/10.1016/j.cobme.2017.10.008
5. Santos-Moreno J, Schaerli Y (2019) Using synthetic biology to engineer spatial patterns. Adv Biosyst 3:1800280. https://doi.org/10.1002/adbi.201800280
6. Zarkesh I, Kazemi Ashtiani M, Shiri Z, Aran S, Braun T, Baharvand H (2022) Synthetic developmental biology: engineering approaches to guide multicellular organization. Stem Cell Rep 17:715–733. https://doi.org/10.1016/j.stemcr.2022.02.004
7. Morsut L, Roybal KT, Xiong X, Gordley RM, Coyle SM, Thomson M, Lim WA (2016) Engineering customized cell sensing and response behaviors using synthetic notch receptors. Cell 164:780–791. https://doi.org/10.1016/j.cell.2016.01.012
8. Toda S, Blauch LR, Tang SKY, Morsut L, Lim WA (2018) Programming self-organizing multicellular structures with synthetic cell-cell signaling. Science (1979) 361:156–162. https://doi.org/10.1126/science.aat0271
9. Toda S, McKeithan WL, Hakkinen TJ, Lopez P, Klein OD, Lim WA (2020) Engineering synthetic morphogen systems that can program multicellular patterning. Science (1979) 370:327–331. https://doi.org/10.1126/science.abc0033
10. Yokobayashi Y, Weiss R, Arnold FH (2002) Directed evolution of a genetic circuit. Proc Natl Acad Sci 99:16587–16591. https://doi.org/10.1073/pnas.252535999
11. Santorelli M, Lam C, Morsut L (2019) Synthetic development: building mammalian multicellular structures with artificial genetic programs. Curr Opin Biotechnol 59:130–140. https://doi.org/10.1016/j.copbio.2019.03.016
12. Lam C, Saluja S, Courcoubetis G, Yu D, Chung C, Courte J, Morsut L (2022) Parameterized computational framework for the description and design of genetic circuits of morphogenesis based on contact-dependent signaling and changes in cell–cell adhesion. ACS Synth Biol 11:1417–1439. https://doi.org/10.1021/acssynbio.0c00369
13. Swat MH, Thomas GL, Belmonte JM, Shirinifard A, Hmeljak D, Glazier JA (2012) Chapter 13 - Multi-scale modeling of tissues using CompuCell3D. In: Asthagiri AR, Arkin APBT-M in CB (eds) Computational methods in cell biology. Academic, pp 325–366