Simulating Intracellular Protein Traffic and Cellular Stress using PyDEVS

Reehan Shaikh
Modelling, Simulation and Design Lab
School of Computer Science
McGill University, Montreal, QC, Canada
reehan.shaikh@cs.mcgill.ca

May 2007

Abstract

A simulation of intracellular protein traffic and how it affects cellular stress has been developed. Though still in its early stages, it has proven to be an easy way of simulating what proteins are doing in the cell. Following a protein's journey from the nucleus to the plasma membrane or lysosome and what it does in between has been the focus of this experiment, as well as to hypothesize of the outcome of such a journey, for example, how a cell is stressed.

Developed using PyDEVS, the Python implementation of the DEVS formalism, it can easily be extended to support larger and more complex models. Moreover, future hypotheses of protein behaviour within the cell can be easily tested using these extensions. Due to PyDEVS' hierarchical nature, adding new models to the existing model is a seamless task that involves declaring the model and patching it up to the existing models via channels for communication.

1 Introduction

A eucaryotic cell is intracately separated into little compartments known as *organelles*. Each organelle is surrounded by a membrane and carries out a distinct function within the cell. Proteins play a major role in these functions; they incite internal organelle reactions as well as transport molecules in and out of the compartment. Certain proteins also carry organelle-specific markers that help newly synthesized proteins direct themselves to their destined compartment. On average, a cell contains close to 10 billion proteins, divided into about 10,000 - 20,000 different types.

The complexity of the cell, the large number of proteins and the important functions that proteins aid in prove that a simulation of some sort is needed. To be able to visualize exactly what happens when, for example, the endoplasmic reticulum contains too many unfolded proteins, we can further test hypotheses in an extremely easy manner.

2 The Cell

2.1 Cell Organelles

Unlike a procaryotic cell, the animal cell is composed of compartments known as organelles. More than half of the cell's volume is taken up by the *cytosol*, the ogranelle responsible for all protein synthesis and degradation. Intracellular membrane systems further divide the cell into little aqueous pockets separate from the cytosol. These pockets make up the rest of the cell's organelles. The membrane's lipid bilayer is impermeable to most hydrophilic molecules, thus an organelle's membrane must carry proteins

that help in the import and export of molecules. Moreover, each membrane must also store proteins with organelle-specific receptor markers that help newly synthesized proteins find their destination organelle. These specific proteins give each organelle its uniqueness.

The cytosol also carries out the majority of the cell's intermediary metabolism - the reactions where small molecules are degraded and synthesized. These small molecules are the essential building blocks of the cell's macromolecules. The cytosol, along with the cytoplasmic organelles, make up the cytoplasm. The cytoplasm surrounds one of the cell's principal organelles, the *nucleus*. The nucleus contains the main genome of the cell and is entirely responsible for DNA and RNA synthesis.

Close to half of the total area of the membrane system in an animal cell is used to enclose the maze-like organelle known as the endoplasmic reticulum (*ER*). While other organelles translocate and import proteins only after the protein is completely synthesized, this special organelle is best known for translocating proteins into its interior **while** the protein is being synthesized. Consequently, the ER has many ribosomes attached to its cytosolic surface which help in the synthesis of these proteins. Once synthesized, these proteins are usually bound for secretion to the cell's exterior or to other organelles.

The ER sends most of its proteins to the *Golgi apparatus*. The Golgi is responsible for sending these proteins to the cell's various other organelles. *Mitochondria*, the second largest organelle of the cell, generates most of the cell's **ATP**. The *lysosome* consists of digestive enzymes that degrade obselete intracellular organelles and proteins. It also destroys particles and macromolecules that may have slipped into the cell via endocytosis. These particles and macromolecules, on their way to the lysosome, pass through a number of organelles known as *endosomes*. *Peroxisomes* are organelles that hold enzymes used in a diverse set of oxidative reactions within the cell. *Proteasomes* are abundantly dispersed throughout the cytosol and aid in the its degradation of proteins.

2.2 Protein Traffic

Most proteins' lives begin in the cytosol where ribosomes help in their synthesis. Depending on the *sorting signal* contained in the amino acid sequence, each protein is destined for a specific function in some organelle. Most proteins actually don't carry this signal and thus become resident cytosolic proteins. Proteins that carry a signal usually end up in the nucleus, ER, mitochondria or peroxisome. Moreover, there are protein signals which direct proteins from the ER to other organelles via the Golgi apparatus, which acts as a protein sorter. Most intracellular transport is guided via these signals and proteins bearing some sort of signal must have complementary receptor proteins to guide them to their destination. These receptors act as catalysts and once they have directed an incoming protein to its destination, the receptor goes back to where it came from in order to be reused. Moreover, a receptor protein can recognize more than one sorting signal, so it can aid in a variety of protein deliveries to different locations.

DNA and RNA systhesis occurs in proteins destined for the nucleus from the cytosol. Some of these proteins return to the cytosol after synthesis, others go on to the ER where they attempt to fold and assemble. Almost all proteins that must ultimately end up in one of the cell's organelles passes through the ER[7]. Certain proteins that carry the ER retention signal[2, p.701] will become ER resident proteins where they will aid newly translocated proteins in folding and assembling properly and until they end up in their proper and final state, these new proteins will not leave the ER. Thus, at some point, the ER will become flooded with unfolded proteins.

This extra load on the ER will trigger an **unfolded protein response**, *UPR*. This involves sending a signal to the nucleus to increase "the transription of genes encoding ER chaperones"[2, p.705]. Moreover, the nucleus will also increase the number of enzymes involved in ER-protein degradation. The reason for this is that despite all the help from ER chaperones, some proteins will never fold or assemble properly. This will result in **ER-associated degradation**, *ERAD*, whereby proteins are sent back to the cytosol for degradation by proteasomes.

If folded and assembled properly, proteins will go onto the Golgi apparatus where they will be sorted by destination. The Golgi is divided into two main sections, the *cis*, or entry, and *trans*, or exit, Golgi networks. Both Golgi networks act as a filter which first remove any escaped ER-resident proteins. These proteins are sent back to the ER. Otherwise, proteins that move further in their journey will go on to one of the cell's other organelles.

3 PyDEVS

The *D*iscrete *EV*ent system *S*pecification was first established in 1976 by Bernard P. Zeigler[1]. This formalism gave a detailed structure to discrete-event modelling. PyDEVS is an implementation of this formalism using *Python*[1], an extremely high-level, interpreted and object-oriented language. The PyDEVS package contains two files; **DEVS.py** gives the class architecture for hierarchical model definitions while **Simulator.py** implements the simulation engine. We will go into further detail about the modelling architecture but no description of the simulation engine is given as it doesn't fall into the scope of this paper. For those interested, please refer to [1].

There are two things to note of the simulation engine that are of importance to us. First, there is a notion of time, not real-time, but where events are timestamped in some chronological order that they occur. Secondly, the engine provides the modeller with an easy way to define how long an experiment will run via the use of a boolean value. This value can be any boolean expression such as when a certain model enters a specific state, when the simulation clock reaches a certain value or always return false so that the simulation will go on forever!

3.1 AtomicDEVS

An *atomic* DEVS model is used for describing a simple system and is primarily referred to by its *state*. Each atomic model starts in an initial state and has an associated set of states. The *time-advance* function is used to calculate when the next internal transition is scheduled for. The *internal* transition function allows an atomic model to change its state. Prior to the internal transition, the *output* function permits an atomic model to send messages to other models by the use of the *poke* method. These messages are sent over channels connected by *ports*. Ports are either input or output and there is a clear distinction between them (i.e. the same port cannot be used for input and output).

Messages are received via the *peek* method, exclusively used in the *external* transition function. If a message is received, an external transition is triggered. This also allows the atomic model to change state but based on external stimuli only. When a model is interrupted by a message, the elapsed time since the last transition is stored. Access to this value is also exclusive to the external transition funtion. Once there is an external transition, the time-advance function is called to schedule the next internal transition.

The initialization of an atomic model allows the modeller to set the initial state, set the starting elapsed time (so that the model can begin its timestamping of events at some time other than the default 0.00) and define the model's set of ports. The default behaviour of the atomic model's time-advance function is to return ∞(infinity). The internal and external transition functions both return the current state if not overridden. The default output function does absolutely nothing. The constructor will set the elapsed time to the default 0.00 and the model's state to None - Python's null object. No ports are defined.

In summary, an atomic model will first call its time-advance function to schedule the next internal transition at time x. It will then set itself to its initial state. If no external stimuli interrupts the model before time x is reached, then the model outputs its messages if any, an internal transition takes place and the time-advance function schedules the next internal transition at time y. If the model is interrupted before time x, no messages are sent, an external transition is triggered and the time-advance function is called to schedule the next internal transition at time z. Note that if a model continuously gets interrupted by external stimuli, it will never make an internal transition - this can be easily overcome as discussed

[1] www.python.org

in the following implementation section.

3.2 CoupledDEVS

A *coupled* DEVS model is used to describe complex systems. It consists of, possibly several, submodels that are either atomic or coupled. A coupled model has only one function of importance to the modeller, the *select* function. This method is used as a tie-breaker if and when two submodels of the coupled model have events scheduled at exactly the same time. It allows the modeller to choose a specific model whose events should be carried out first.

A coupled model may also have ports. The only difference is that the input (output) port of the coupled model must be connected to the input (output) port of one of the coupled model's submodels. Essentially, only atomic models can send messages to each other. If some atomic model A1 (that is a submodel of a coupled model C1) must send a message to atomic model A2 (that is a submodel of C2), then A1 must send the message from its output port to the output port of C1, C1 will send the message from its output port to the input port of C2 and finally C2 will deliver the message from its input port to the input port of A2. Hence, three separate channels must be defined for this communication to take place. The channel that connects C1 and C2 must be defined in a coupled model C3 which has as submodels C1, C2 and possibly other models. The channel that connects C1 and A1 must be defined in C1. Finally, C2 defines the channel that connects it to A1.

The initialization of a coupled model first instantiates all its submodels and then defines the channels between submodels as well as between itself and any of its submodels. The default select function will return the first element in the list of colliding submodels. When a model is instantiated, it is given a unique *myID* attribute and this list is lexicographically sorted based on this attribute. The default constructor for a coupled model will do absolutely nothing.

4 Implementation

The model is implemented as follows. All the cell's organelles are represented as atomic models. The entire cell is depicted as a coupled model with the organelles as its submodels. This section will go into precise detail of the implementation of each model and their overridden methods, as well as any difficulties and workarounds that were needed to allow for easier functionality. It is assumed that you have already installed the PyDEVS package and have access to its base classes and default interface methods. A readme file is included with the project for this purpose. The cytosol (and cytoplasmic) organelles aren't included in the model since they only produce the proteins and pass them onto the nucleus. They also pass proteins to the mitochondria and peroxisome but these organelles only use up the proteins and have no direct effect on stressing the cell. Hence, these organelles aren't modelled either. Moreover, the endosome only acts as a pit stop for proteins en route to the lysosome, so it is excluded as well.

4.1 Helper Classes

To allow for easier functionality, a *State* class is created that consists of two attributes - one to hold the actual current state and another to hold the elapsed time since the last transition. I had originally started with the state variable of each model to be a string which depicted the actual current state. There was no problem with such an approach since the state attribute of a model can take the form of any type or object. But further reading showed that each model's *total state* consists of the actual current state as well as the elapsed time since the last transition[1]. The difficulty of not keeping track of the elapsed time was that a model A's internal transition never took place if some other model B kept interrupting A with external messages.

For example, suppose the time-advance function for the nucleus and the ER returns a constant value x and y, respectively. Also suppose that x < y. Once the simulation is started, the nucleus will send a message to the ER via its output function, perform an internal transition and schedule its next internal transi-

tion at time $2x$. The ER will then receive an external message, perform an external transition and schedule its next internal transition at time $x + y$. There will be another message sent from the nucleus to the ER at time $2x$ (since $2x < x + y$) and the ER, after performing an external transition, will schedule its next internal transition at time $2x + y$. This cycle of events will never allow the ER model to take its internal transition. Consequently, the ER will never output any messages either, so the model will never propogate information further than the ER (moreover, the ER can never send a reply message, if need be, back to the nucleus).

For this reason, we opt for a variable time-advance function. When the ER is interrupted by some external stimuli, the current elapsed time x is stored. When the time-advance function is called, it will return the difference between the actual time that the next internal transition would have taken place y and the elapsed time x. Hence, if x time units have passed and the ER gets interrupted by the nucleus, we would set the next internal transition of the ER to be at y - x, allowing for the transition to take place at time y, as originally scheduled.

There is also a *Protein* class. Proteins are the actual messages passed from the nucleus to the ER, as well as from the ER to the Golgi and from the Golgi to the lysosome and plasma membrane. Like the state attribute of a model, a message can be of any type or object. A protein has two types, "p" or "c", which stand for a regular protein or a chaperone protein, respectively. All proteins are instantiated in the nucleus. They have three attributes - a boolean to check if the protein folded properly, another boolean to check if the protein should be discarded and an integer to keep track of how many times the protein tried to fold. A chaperone protein is just a helper protein. A regular protein starts off as unfolded. It is passed to the ER where it tries to fold randomly with a fifty percent success rate. After five unsuccessful tries to fold, it is set to be discarded. If successful, the protein is set to folded and passed onto the Golgi, where it goes onto either the lysosome or the plasma membrane.

4.2 The Model

There are five modelled organelles - the nucleus, ER, Golgi, lysosome and plasma membrane. Implemented as five separate classes, they all inherit from *AtomicDEVS*. Common to all models is the time-advance function. Currently, the "time" at which all events occur is at intervals of 30. Recall that this is not real-time, but just a timestamp. Hence, when looking at the trace of a simulation, all timestamps are multiples of 30. The time-advance function will always return the difference between 30 and the elapsed time of the total state of the model. It will store this difference in a temporary variable and set the total state's elapsed time to 0.0. The reason for this is if no external interrupt occurs in the next 30 time units, we would like the time-advance function to return the correct value when it is called subsequently. Finally, the difference is returned.

The total state's elapsed time is set in the external transition function, which is also common to all models. If a model is externally interrupted, the external transition function is triggered. This function, having exclusive access to the elapsed time since the model's last transition, will store a copy of this value in the total state's elapsed time attribute so that it may be accessible to the time-advance function. Also common to all models is an associated *State* class. The constructor of each model sets the model's initial state to a new instance of the State class with the actual state being "regular" and the total state's elapsed time being 0.0. Thus, each model has its own State class instance and in our case, the experiment will have five instances of the State class.

The constructor for the nucleus doesn't override the default elapsed time of the model, so the model starts at 0.0. Also declared is an input port and an output port for communication with the ER. Finally, because proteins start off in the nucleus, a protein counter is started as well. This counter will serve as the ID for each instantiated protein. The nucleus expects input from the ER and this input, an external interrupt, is exclusive to the external transition function. The messages from the ER can either be "help" or "okay". If the ER asks for help, the nucleus changes its state to "stressed". If the ER says that

it is okay, the nucleus changes its state to regular. The internal transition function isn't overriden because the nucleus only makes state transitions based on external events from the ER, so it just returns the current state. The output function will send different messages depending on the state of the nucleus. If in regular state, a new Protein is instantiated, its type is set to a regular protein and it is sent to the ER. If stressed, the instantiated Protein's type is set to chaperone and then sent to the ER. Finally, the protein ID counter is incremented.

The ER is the driving model of this simulation. It dictates if the cell is stressed and keeps track of folded, unfolded and chaperone proteins as well as proteins to be discarded. The ER's constructor will first declare lists to hold the different sets of proteins. Two sets of input and output ports are also declared, one set for communicating with the nucleus and the other with the Golgi. The ER can expect two external interrupts, one from the nucleus and another from the Golgi. The nucleus will always send messages that hold a protein object. If it is a regular protein, it is added to the unfolded list, otherwise it is a chaperone protein and it is added to the chaperone list. The Golgi will always send back the chaperone proteins that the ER sent to it, so those will be added to the chaperone list as well. The internal transition function will first iterate over all unfolded proteins. Each protein will try to fold. If folded, it will be removed from the unfolded list and put into the folded list. If the protein doesn't fold, we make sure it isn't ready to be discarded. If so, we remove it from the unfolded list and append it to the discarded list. Finally, we check if there are too many unfolded proteins, currently the maximum is three. If this is the case, we change our state to stressed. If not, we change our state to regular. The output function will randomly, with a twenty percent rate, send a chaperone protein to the Golgi. Otherwise, with an eighty percent chance, it will send a folded protein. As for the nucleus, if the state of the ER is regular, it will send a string message saying "okay". If the ER is stressed, it will send a message saying "help".

The Golgi's constructor will define three lists to keep track of proteins. A list for those that will end up in the plasma membrane, another for those whose destination is the lysosome and the third for those that need to return to the ER. An input port and an output port are defined for communication with the ER. Two other output ports are defined as well, one for sending messages to the plasma membrane and the other for the lysosome. The plasma membrane and lysosome don't send messages to the Golgi, so no input ports are required. The external transition function only expects input from the ER, in the form of a Protein object. If it is a regular protein, it is put into the plasma membrane list fifty percent of the time, otherwise it goes into the lysosome list. If it is a chaperone protein, it gets appended to the ER list. The internal transition function is not overridden since the Golgi doesn't perform any activities when the cell is stressed. The output function will check if its ER-proteins list has any elements. If so, the chaperone protein is sent back to the ER. Then the lists for the plasma membrane and lysosome are checked in that order for elements and if any contents are found, they are sent to their respective destinations.

The plasma membrane and lysosome are identical but separate models since either organelle is the last stop in a protein's journey. The constructor declares a list to store received proteins as well as an input port so that the Golgi can send messages to it. The external transition function expects input from the Golgi, in the form of a protein object. This regular protein is appended to the list. The internal transition checks if there are any elements in the list. If so, the first element is removed. In the case of the plasma membrane, this can be regarded as protein secretion whereas for the lysosome, it can be seen as protein degradation. The output function is not overridden since both these organelles don't output any messages.

Finally the class Cell, which inherits from CoupledDEVS, acts as the coupled model holding all of the above-mentioned atomic models together. The constructor first instantiates each submodel and then creates the necessary channels by connecting its submodels' ports. In the case of colliding models, the select function is overridden to return the nucleus, then the ER, otherwise the Golgi. If none of these models is found, then the last model in the list of

colliding models is returned. This was done because the nucleus needs to pass proteins to the ER so that something interesting can become of the experiment. We choose the ER over the Golgi so that the ER can dictate if the cell is stressed.

4.3 The GUI

The GUI is driven by the events that take place within the simulation. The GUI has a helper class called *Writer* which is actually embedded in the model itself. When an experiment is executed, the Writer class is instantiated only once and used throughout the simulation. Its constructor first creates a file called **output.sim**. If the file already exists, it is cleared of any content. The file handle is closed and a command counter is started. Everytime there is some type of communication between two models, the communication is recorded into the output.sim file. This is a fairly easy task since all communication starts in a model's output function. Within the output function of each model just before the message is sent, we call the Writer to record the communication. To allow for easy implementation, the Writer class also has a method called *write*. It takes as input the originating model, the message that will be passed and the destination model. The output.sim file is opened, a list of the form

[command, origin, message, destination]

is recorded and the file is closed. Finally, the command counter is incremented.

The GUI's constructor declares the widget that will hold the image which depicts the current command. Also, a status bar is added for convenience as well as a menu with basic simulation controls. These controls include opening a simulation file with a .sim extension and exiting the program, both under the File menu. The Simulation menu includes Start, Pause and Stop controls for running a visual simulation. The Help menu contains an About feature with minimal information. A reference to every type of image is created so that we can easily swap images at once. There are two sets of images - those that show a normal cell and those that show a stressed cell. Booleans to check if a simulation has started and if the cell is stressed are also declared, as well as a string to hold the simulation file name and a list to hold all of the simulation commands obtained from that file. An attribute named *after* is also declared. Its use will be discussed shortly.

Opening a simulation file only gives the GUI the absolute path to the file. Once a user opts to start the simulation, the file is opened and all the commands are read at once into a list. Then the *runSim* method is called. It stores and removes the first command in the list (in turn, this moves the second command to the first position), parses it and calls the *updatePhoto* method. Depending on the command, certain images are displayed. Most commands are associated with two images, so we needed a way to let the simulator show an image, sleep for a certain amount of time, show the second image, sleep for a certain amount of time and finally continue with the rest of the commands by calling *runSim* again. First, the Tkinter (Python's widely used GUI library) sleep method was used. This posed a problem because the application became completely dormant; users that tried to interrupt the current simulation couldn't. So, if you wanted to interrupt it, you had to click the mouse at exactly that millisecond where the application was processing a command and wasn't sleeping!

So, I looked into the library and found a very nifty method called *after*. It allows you to schedule an event to be executed at a certain time from now, usually a function call. While waiting for the event to occur, the application doesn't sleep and can be interrupted. Moreover, a scheduled event can also be cancelled. Thus, every photo change was put into a function that could be called by this *after* method. Everytime an event was scheduled, its handle was stored in the *after* attribute described above. If the application is interrupted, the *after* handle is cancelled and the event won't take place. So, for example, the *updatePhoto* method calls the method *zeroa* to display the first image which shows a protein about one-third the distance from the nucleus to the ER. This method will schedule an event to call the method *zerob*, which shows the protein about two-thirds the distance from the nucleus to the ER, **x** milliseconds from now. The method *zerob* will schedule an event to call the *runSim* method in **x** milliseconds. Finally,

runSim will process the next command and this cycle of events will occur again. Currently, **x** is set to 750 milliseconds.

5 Conclusion

A simulation of intracellular protein traffic and how it impacts cellular stress has been developed. At the present moment, certain internal cellular behaviour has been assumed to attain the current result. For example, if the ER has too many proteins within it, the cell becomes stressed. My first goal was to develop a working model. With this, I will further the model and tweak certain aspects of it to obtain a more realistic set of results.

Near future work includes fiddling with the probability functions that dictate where proteins end up. Also, the current implementation doesn't simulate the effects of ERAD as well as the help that chaperone proteins provide in the ER. Both these issues have been started but yet to be finished. Once this is achieved, then the visual simulator needs to be tweaked to show the results visually.

References

[1] Jean-Sébastien Bolduc and Hans Vangheluwe. A modelling and simulation package for classical hierarchical DEVS. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001.

[2] Bruce Alberts et al. *Molecular Biology of the Cell*. Garland Science, New York, 4th edition, 2002.

[3] Sean Munro. Lipid rafts: Elusive or illusive? *Cell*, 115(4):377–88, November 2003.

[4] Sean Munro. Organelle identity and the organization of membrane traffic. *Nature Cell Biology*, 6(6):469–72, June 2004.

[5] Hugh R.B. Pelham and James E. Rothman. The debate about transport in the golgi–two sides of the same coin? *Cell*, 102(6):713–9, September 2000.

[6] Enrique Rodriguez-Boulan and Anne Musch. Protein sorting in the golgi complex: shifting paradigms. *Biochim et Biophys Acta*, 1744(3):455–64, May 2005.

[7] James E. Rothman. The golgi apparatus: Two organelles in tandem. *Science*, 213(4513):1212–19, September 1981.