Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, eds.

DEVS-BASED DESIGN OF SPATIAL SIMULATIONS OF BIOLOGICAL SYSTEMS

Rhys Goldstein
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada

ABSTRACT

The application of the DEVS formalism to spatial simulations of biological systems is motivated by a need to keep software manageable, even when faced with complex models that may combine algorithms for potential fields, fluid dynamics, the interaction of proteins, or the reaction and diffusion of chemicals. We demonstrate DEVS-based design by applying the formalism to a "tethered particle system" (TPS), a model we designed to capture the motion of deformable biological structures. The paper focuses on the design of DEVS models using hierarchies and layers, and describes a recently-developed simulator that supports our approach. The DEVS-based TPS model, which has been used to simulate certain interactions in nerve cells, demonstrates the formalism's potential as a means of addressing the complexity of spatial biological models.

1 INTRODUCTION

Simulation, in particular simulation involving spatial aspects, is attracting an increasing amount of interest among biologists and medical researchers seeking better understanding of biological systems and the ability to predict their behavior. The pursuit of ever more realistic models will inevitably require the integration of various algorithms; algorithms that simulate interrelated phenomena such as protein interaction, potential fields, fluid dynamics, and the reaction and diffusion of chemicals. If one lacks a strategy to address the complexity of such models, the result may well be large quantities of unmanageable simulation code. We contend, however, that if one can construct complex models from simpler submodels defined in layers, the code need not become unwieldy.

The Discrete Event System Specification (DEVS) is a general modeling framework in which models are defined as hierarchies of modular submodels that interact in a discrete-event fashion. Hierarchical model design, and other properties of DEVS such as the separation of model and simulator, have compelled us to investigate its application to spatial simulations of biological systems.

Motivated in part by an ongoing project to simulate vesicle-synapsin interactions in nerve cells (Goldstein, Wainer, Cheetham, and Bain (2008)), we defined a "tethered particle system" model (TPS) that captures the motion of deformable biological structures. In this paper we offer a brief description of the TPS algorithm, and focus on its DEVS-based design in an effort to address the following questions. How does one design hierarchical models of biological systems with spatial aspects? How does one parameterize those models? How does one implement them for simulation? Among other things, we suggest that distinct aspects of algorithms be separated at an upper level in a DEVS model hierarchy, and that the partitioning of space be handled at lower levels. We also recommend that DEVS models of biological systems be defined and parameterized in layers, and describe a recently-implemented DEVS simulator that supports this approach.

DEVS, and examples of its application to spatial simulation and biology, are described in Section 2. Sections 3, 4, and 5 address, respectively, hierarchical model design, layered model design, and implementation. The discussion in Section 6 outlines how the DEVS-based tethered particle system model might be combined with another model that captures the reaction and diffusion of chemicals. Whereas this last discussion explores just one example in which two algorithms are combined with DEVS, we hope to encourage experts in the field to consider DEVS for the integration of biological simulation algorithms in general.

2 BACKGROUND

DEVS was developed in the 1970s to provide a framework for the design of models for discrete-event simulation. One may refer to Zeigler, Kim, and Praehofer (2000) and Wainer (2009) for a detailed explanation of the theory and examples of its application. When applying the formalism, a distinction is generally made between indivisible DEVS models, described as atomic, and composite DEVS models, described as coupled. In the original formalism, atomic and coupled models are defined as tuples.

$$\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle \qquad \{atomic\ model\}$$

$$\langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \qquad \{coupled\ model\}$$

A key principle of DEVS is the separation of model and simulator. When designing a DEVS atomic model, for example, one defines functions such as δ_{ext}, δ_{int}, λ, and ta, but does not worry about invoking them. The evaluation of these functions is carried out by the simulator. After an event, the simulator evaluates the time advance function ta to determine the time until the next internal transition. Should this time elapse, the output function λ is invoked to obtain an output value y ($y \in Y$), and the internal transition function δ_{int} yields the new state s of the model ($s \in S$). If, however, an input value x is received before the calculated time elapses ($x \in X$), then the simulator applies the external transition function δ_{ext} instead. A DEVS simulator is model-independent in the sense that it should carry out a simulation for any valid DEVS model, regardless of what the model represents.

Atomic models can be linked to form coupled models. These coupled models can in turn be linked with other atomic models and/or other coupled models, and in this manner a model may take on a hierarchical structure. Suppose, hypothetically, that one has defined four atomic models identified by the labels *soma*, *axon*, $terminal_1$, and $terminal_2$. Each of these represents a part of a neuron, or nerve cell, and an entire neuron is represented by the coupled model *neuron* illustrated in Figure 1. The coupled model variable D, the set of components, would be defined as $\{soma, axon, terminal_1, terminal_2\}$. The set of components $\{M_i\}$, would be a set of four tuples, one for each atomic model. The arrows in the figure depict links through which the outputs of one model become inputs for others. These links would be formally defined by the set of influencees $\{I_i\}$. It is worth noting that any coupled model has an equivalent atomic model, a property referred to as "closure under coupling".

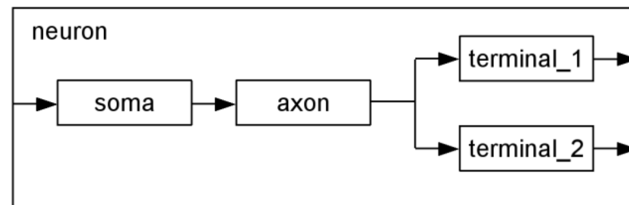

Figure 1: An illustration of a DEVS coupled model, called *neuron*, that links together four submodels. A typical nerve cell does not have exactly two terminals, but this model is only intended to serve as a hypothetical example.

We now turn our attention to models with spatial aspects. We do not consider the model in Figure 1 to be a spatial model, because although a neuron's soma, axon, and terminals can be associated with different regions, the shape of these structures is neglected. Consider, by contrast, the spatial cellular model illustrated in Figure 2, in which the shape of the neuron is represented by a set of shaded cells in a lattice.

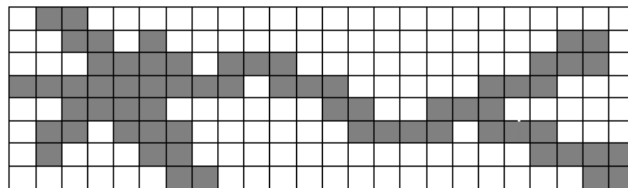

Figure 2: A hypothetical cellular model in which the shape of a neuron is captured by a set of shaded lattice cells

The application of the original DEVS formalism to cellular models is discussed in Zeigler and Vahie (1993) and elsewhere. Another approach is offered by the Cell-DEVS formalism, described in Wainer (2009), which is an extension of DEVS for cellular automata. As presented in Wainer, Jafer, Al-Aubidy, Dias, Bain, Dumontier, and Cheetham (2007) and Goldstein, Wainer, Cheetham, and Bain (2008), Cell-DEVS has been used to simulate a number of biological systems, including the vesicle-synapsin clusters mentioned in the introduction.

In one way or another, cellular models tend to introduce uniformity within each cell of the lattice. When using the Next Subvolume Method to model the reaction and diffusion of chemicals, for example, one introduces a cell space in which the concentration of each chemical changes from one lattice cell to the next, but is uniform within each cell (Elf and Ehrenberg (2004)). In this case the assumption of uniformity is advantageous in that it allows one to apply the Gillespie Algorithm, which predicts concentration changes in a homogeneous chemical system (Gillespie (1977)). In the case of the Figure 2 model, however, the uniform nature of each cell may be disadvantageous in that it complicates changes to the neuron's shape and position. Although it would be easy to translate the entire neuron up or down or left or right, moving it in an arbitrary direction and deforming it would be difficult. We are therefore interested not only in cellular models, but also continuous-space models such as the one illustrated in Figure 3. In this case the shape of a neuron is captured by a set of connected points that are not restricted to a lattice. The neuron may be translated by moving all points in the same direction, or deformed by moving each point in a slightly different direction than that of its neighbors.

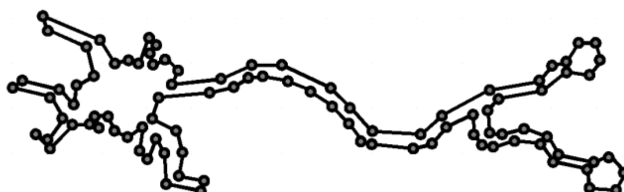

Figure 3: An example of a continuous-space model, in which the shape of a neuron is captured by a set of connected points

We now mention three challenges associated with the application of DEVS to spatial simulations of biological systems. First, we note that in Figure 1, we have a simple two-level hierarchy based on nerve cell anatomy: our neuron, at the upper level, is composed of a soma and an axon and two terminals, all at the lower level. In the spatial models of 2 and 3, however, we have abandoned this anatomical hierarchy and modeled the entire neuron as a whole. Our first challenge is this: if we are to neglect anatomical hierarchies in spatial models, how do we exploit the hierarchical nature of DEVS?

Assuming that we can find suitable hierarchies for DEVS models of biological systems, the second challenge is how to parameterize those models. A strategy is needed to define DEVS models as dramatically different from one another as those in Figures 1, 2, and 3, while at the same time encouraging the reuse of formulas between models.

Even if one has arrived at a sound mathematical definition of a DEVS model, the constraints of computer technology will motivate numerous changes and compromises. Implementation is therefore our third challenge.

3 HIERARCHICAL MODEL DESIGN

Here we begin with a brief description of a TPS (tethered particle system), then focus on its DEVS-based design to demonstrate how suitable hierarchies can be chosen for spatial biological models. The TPS models deformable biological structures such as the vesicle-synapsin clusters found in the terminals of nerve cells. This is accomplished by tracking the position of each particle in a set. If two approaching particles reach an inner limiting distance, they collide, rebound outwards, and may become "tethered". When two separating tethered particles reach an outer limiting distance, then provided they remain tethered, they retract inwards. By constraining the distances between pairs of particles in this manner, proteins and membranes and other biological structures may be modeled.

A more detailed description of the TPS can be found in Goldstein and Wainer (2009). We now consider the hierarchical structure of a DEVS model of a TPS. To simplify its presentation, we will focus on the handling of collisions between moving particles, as opposed to the tethering. As each particle can be at any location and move in any direction, the TPS is an example of a continuous-space model of the type illustrated in Figure 3 of Section 2.

When designing a particle system model, one must address two fairly distinct sub-problems: collision detection and collision response. Recognizing this, one can simplify matters by focusing on detection and response separately. We make this separation explicit in our hierarchical model design by defining the DEVS model TPS as a coupled model consisting of

a *detector* submodel and a *responder* submodel. The *detector* outputs a *collision* message when two particles collide with one another, but it is the *responder* that alters the trajectories of those particles after receiving the message. The two new trajectories are sent back to the detector as a pair of *response* messages. As shown in Figure 4a, the *response* messages are also used as outputs for the *TPS*.

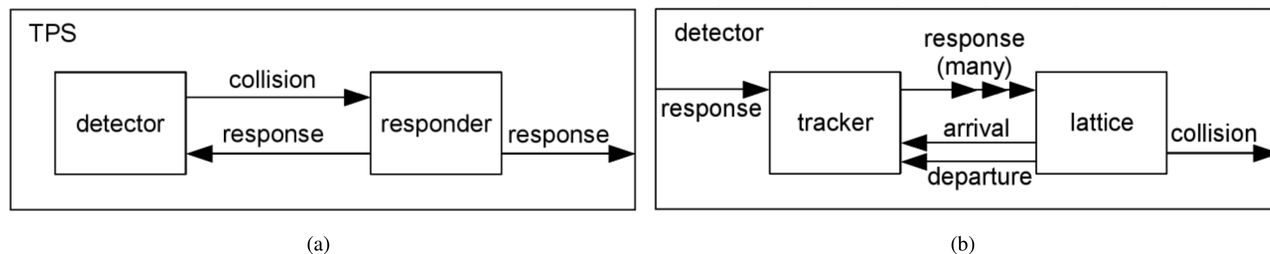

Figure 4: The hierarchical structures of the *TPS* and *detector* coupled DEVS models

Whereas the *responder* is simply an atomic model, we are motivated in part by computational efficiency to adopt a coupled model for the *detector*, and thereby add another level to our hierarchy. Note that a very simple detection algorithm would compare each particle with every other particle, and calculate the future time at which each pair collides. It is considerably more efficient, however, to compare only pairs of particles that are close to one another. This is achieved by introducing a lattice of subvolumes, represented by the *lattice* submodel. The *lattice* is responsible for sending the *collision* messages that ultimately go to the *responder*. However, when the *responder* sends a *response* message to the detector to update a particle's trajectory, the message must be re-directed to each subvolume of the lattice that is "aware" of that particle. This re-direction is accomplished by the *tracker* submodel. The *tracker*-*lattice* relationship, depicted in Figure 4b, also involves *arrival* and *departure* messages, which will be clarified by a scenario described further below.

For the *tracker* we use an atomic model. But as the *lattice* model represents a set of subvolumes, we take the obvious approach and represent each subvolume with its own DEVS atomic model. The *lattice* is thus a coupled model. As illustrated in Figure 5, each subvolume model *subV* may receive *response* messages and output *collision*, *arrival*, and *departure* messages. Additionally, each *subV* may send *adj* messages to the *subV* models that neighbor it according to their lattice configuration. The *adj*, *arrival*, and *departure* messages will be explained in the scenario.

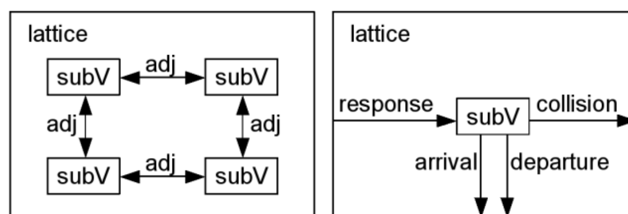

Figure 5: The *lattice* coupled model of the *detector*. Although only a 2-by-2 configuration is shown on the left, one can have any number of subvolume models in 1, 2, or 3 dimensions.

Arguably the most intuitive way to design a hierarchical DEVS model is to create spatial divisions of some sort, though we have only introduced space-based partitioning at the third and lowest level of our hierarchy. At upper levels in the hierarchy, we separate distinct aspects of the algorithm such as collision detection and collision response.

Now we consider a simple scenario involving a large particle *A* and a small particle *B* on a 2-by-3 lattice. The subvolume models of the lattice are identified by their coordinates. The subvolumes themselves are square regions, but as shown in Figure 6, for each subvolume there are two concentric circles that surround it. Each subvolume model is aware of certain nearby particles, but not other more distant particles. Model $[0,1]$, for example, is aware of particle *A* but not particle *B*. If a DEVS model is aware of a particle, the current state of that model includes the position and velocity of that particle.

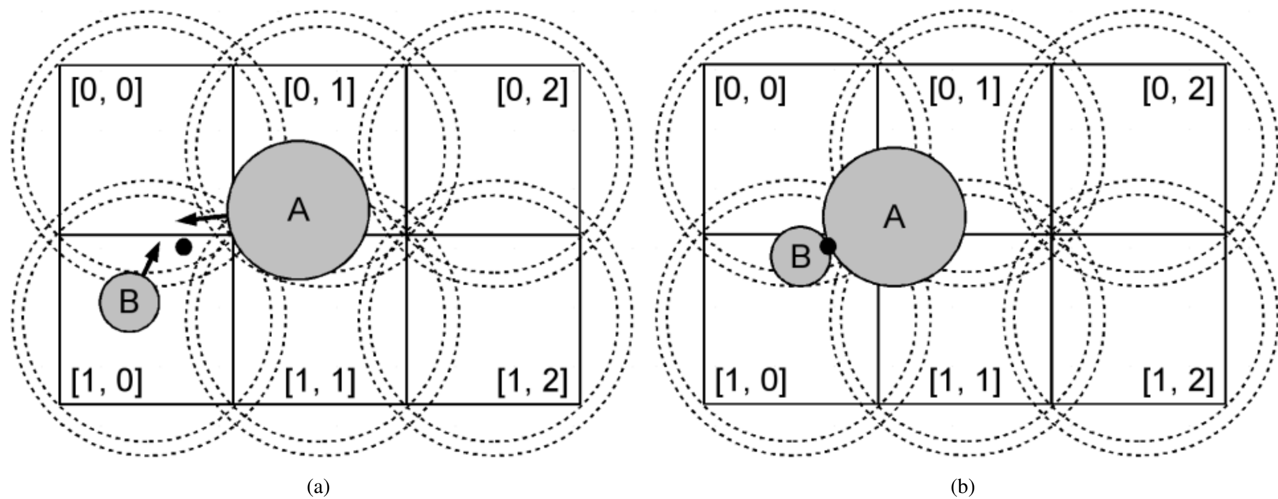

Figure 6: A scenario in which two particles first approach one another (a), then at a later time collide (b). The dot in both cases indicates the location at which the particles meet.

Looking at Figure 6a, we are going to assume that the subvolume model $[0,2]$ in the top-right corner is currently aware of particle *A*. Because the particle is moving away from that subvolume, and because its backside is just touching the outer circle, the associated *subV* undergoes an internal transition in order to lose awareness of the particle. The particle's position and velocity are removed from the subvolume model's state, and a *departure* message is output. The *departure* message leaves the *lattice* and is sent to the *tracker*. The *tracker*, which maintains a record of which subvolume models are aware of each particle, updates itself accordingly.

Looking again at Figure 6a, let us say that particle *B* is just touching the inner circle around the subvolume in the top-left corner. Subvolume model $[0,0]$ must therefore become aware of particle *B*. This process begins when *subV* $[1,0]$ detects the circle-particle intersection, and sends *subV* $[0,0]$ an *adj* message. The *adj* message triggers a transition in which *subV* $[0,0]$ adds particle *B*'s position and velocity to its state, then sends an *arrival* message to update the tracker.

Taking one last look at Figure 6a, and assuming the departure and arrival events described above have already taken place, we note that the two subvolumes on the left are both aware of both particles. Thus the imminent collision between the two particles is detected twice. It is only scheduled once, by *subV* $[1,0]$, for the location at which the particles will meet is in the subvolume of that model.

The DEVS simulator now advances time to the point at which particles *A* and *B* meet. This new situation is shown in Figure 6b. It is at this time that, having scheduled the collision, *subV* $[1,0]$ undergoes an internal transition. It sends a *collision* message, which is directed out of the *lattice* model, then out of the *detector* model, then to the input of the *responder* model. If left alone, the subvolume model will then allow the colliding particles to continue their current motion and pass through one another. This will not happen, however, as after a simulated delay time of 0, the *responder* will calculate new velocities for the particles and send two *response* messages to the *detector*. In the *detector*, the messages go first to the *tracker* submodel. When the *tracker* receives the *response* message for particle *A*, it sends copies of the message to the four subvolume models that are aware of it: $[0,0]$, $[0,1]$, $[1,0]$, and $[1,1]$. These four *subV* models then update their own recorded velocities of particle *A*. When the *tracker* receives the *response* message for particle *B*, only subvolume models $[0,0]$, $[1,0]$ need to be notified.

Note that one could dispense with the concentric circles and use instead the square subvolume boundaries to determine which subvolume models are aware of each particle. It is somewhat cumbersome, however, to determine when a circular particle enters or exits a square region, and worse to determine when a spherical particle enters or exits a cubic region.

We now reflect on what has been gained and lost by adopting this DEVS-based hierarchical model. Starting with a loss, we note that a more traditional programming approach would require us to store only a single position vector and velocity vector for each particle. With our hierarchy of models, identical copies of the position and velocity vectors of a single particle are stored several times over: once in the *responder* model, and again in each subvolume model aware of that particle. In order to change a particle's velocity, we must first replace it in the *responder*, then pass messages to change it in the subvolume models.

So what have we gained? Our complex *TPS* model has been divided into simpler submodels, each of which addresses a small part of the overall problem. As our three coupled models can be specified easily by defining the links illustrated in Figures 4 and 5, the bulk of our efforts must go into the design of the external and internal transitions of the *responder*, *tracker*, and *subV* atomic models. These six transition functions are all independent of one another, and each performs a relatively specific task. And so by adopting the DEVS-based hierarchy, we introduce seemingly-redundant values and messages, but benefit in that a complex routine has been reduced to a set of simpler functions.

4 LAYERED MODEL DESIGN

We now address the second challenge from Section 2, which seeks a strategy for defining dramatically different DEVS models while at the same time encouraging the reuse of formulas. In the previous section we addressed the issue of complexity by dividing complex DEVS models into linked submodels. Here we further reduce complexity by defining DEVS models, hierarchical or otherwise, in layers. Each layer has an associated set of parameters, and the parameters of one layer are used to define those of the underlying layer.

In the original DEVS formalism, every model is either an atomic model or a coupled model. Adopting a different perspective, we will state that every DEVS model is simply an atomic model. Instead of defining an "atomic model" with the tuple $\langle X,Y,S,\delta_{ext},\delta_{int},\lambda,ta\rangle$, we define a "DEVS model" with the vector $[\delta_{ext},\delta_{int},ta]$. We consider these three parameters to be associated with the bottom layer of any model. Almost every model should have at least a second layer with a different set of parameters, however, and the simplest way to add that layer is with a function. To illustrate, the function a_model_{DEVS} below introduces a second layer with the parameters a, b, and c, taking these variables as arguments and resulting in a vector $[\delta_{ext},\delta_{int},ta]$. For the time being, the reader may assume that any function with the subscript $_{DEVS}$ results in a DEVS model of this form. Note that indentation is used to control the scope of certain variables. Because δ_{ext}, δ_{int}, and ta are all indented relative to a_model_{DEVS}, they may use a, b, and c. Because the two definitions of s' are indented, they do not conflict with one another. Certain expressions are replaced with comments in braces (eg. $\{a\ comment\}$).

$$a_model_{DEVS}([a,b,c]) := [\delta_{ext},\delta_{int},ta]$$

$$\delta_{ext}([s,\Delta t_{el},x]) := s'$$
$$s' := \{some\ expression\ that\ may\ depend\ on\ s,\ \Delta t_{el},\ x,\ a,\ b,\ c\}$$

$$\delta_{int}(s) := [s',Y]$$
$$[s',Y] := \{some\ expression\ that\ may\ depend\ on\ s,\ a,\ b,\ c\}$$

$$ta(s) := \Delta t_{int}$$
$$\Delta t_{int} := \{some\ expression\ that\ may\ depend\ on\ s,\ a,\ b,\ c\}$$

Our external transition function δ_{ext} is the same as in the original DEVS formalism, calculating the final state s' from the initial state s, the time Δt_{el} elapsed since the previous transition, and the input value x. The time advance function ta has not been altered either. It results in the time Δt_{int} remaining until the next internal transition, assuming that no input messages are received in the meantime. We have changed things, however, by absorbing the output function λ into δ_{int}. The internal transition function now takes the initial state s as before, but results in a vector Y of output values in addition to the final state s'. This change is possible because the original λ and δ_{int} function were to be invoked back-to-back anyhow. The advantage in combing them is that, in certain cases, we alleviate the need to repeat the same calculations twice over. We omit the sets X, Y, and S for the sake of brevity, but note that they could be incorporated into the layering scheme.

Below is an outline of the definition of the *responder* model of the *TPS*. Its parameters include Ω_ψ, which provides the mass of each particle; $\Omega_{\psi\psi}$, which includes the inner and outer limiting distances referred to in Section 3; *attach*, which determines when particles become tethered; and *detach*, which determines when tethered particles separate.

$$responder_{DEVS}\left(\left[\Omega_\psi,\Omega_{\psi\psi},attach,detach\right]\right) := [\delta_{ext},\delta_{int},ta]$$
$$\delta_{ext}([s,\Delta t_{el},x]) := \{an\ expression\ that\ depends\ on\ s,\ \Delta t_{el},\ x,\ \Omega_\psi,\ \Omega_{\psi\psi},\ attach,\ detach\}$$
$$\delta_{int}(s) := \{an\ expression\ that\ depends\ on\ s,\ \Omega_\psi\}$$
$$ta(s) := \{an\ expression\ that\ depends\ on\ s\}$$

Because we do not wish to abandon coupled models altogether, we introduce them as atomic models but add a second layer using the function $coupled_{DEVS}$. Its parameters include M, which defines the submodels, and C, which defines the links between submodels. In the event that two internal transitions take place at the same simulated time, the priority function pr selects the model with priority. The "closure under coupling" property assures us that $coupled_{DEVS}$ can be defined.

$$\begin{aligned}
&coupled_{DEVS}([M,C,pr]) := [\delta_{ext}, \delta_{int}, ta]\\
&\quad \delta_{ext}([s, \Delta t_{el}, x]) := \{an\ expression\ that\ depends\ on\ s,\ \Delta t_{el},\ x,\ M,\ C,\ pr\}\\
&\quad \delta_{int}(s) := \{an\ expression\ that\ depends\ on\ s,\ M,\ C,\ pr\}\\
&\quad ta(s) := \{an\ expression\ that\ depends\ on\ s\}
\end{aligned}$$

Because the TPS model is coupled, we define it using $coupled_{DEVS}$ but add a third layer. As shown below, the function M yields the *detector* and *responder* submodels when given their corresponding IDs, "*detector*" and "*responder*". We assume that one has a means of obtaining the domain of a function, and can therefore obtain these IDs from M. Consistent with the links in Figure 4a of Section 3, the function C maps *collision* messages from the *detector* to the *responder*, and *response* messages from the *responder* to both the *detector* and the output of the TPS. If both the *detector* and the *responder* are to undergo internal transitions at the same simulated time, then the *responder* is to go first. This ordering is expressed by the *order* vector, which is passed as an argument to pr_{order}, which in turn is used to define the priority function pr. The parameter N gives the dimensions of the collision detection lattice, and a is the length of each subvolume.

$$TPS_{DEVS}\left([N, a, \Omega_{\psi}, \Omega_{\psi\psi}, attach, detach]\right) := coupled_{DEVS}([M,C,pr])$$

$$M(id_m) := \begin{pmatrix} id_m = \text{“}detector\text{”} & \rightarrow detector_{DEVS}\left([N, a, \Omega_{\psi}, \Omega_{\psi\psi}]\right) \\ id_m = \text{“}responder\text{”} & \rightarrow responder_{DEVS}\left([\Omega_{\psi}, \Omega_{\psi\psi}, attach, detach]\right) \end{pmatrix}$$

$$C(src) := \begin{pmatrix} src = \begin{bmatrix} \text{“}detector\text{”} \\ \text{“}collision\text{”} \end{bmatrix} & \rightarrow \left[\begin{bmatrix} \text{“}responder\text{”} \\ \text{“}collision\text{”} \end{bmatrix}\right] \\ \\ src = \begin{bmatrix} \text{“}responder\text{”} \\ \text{“}response\text{”} \end{bmatrix} & \rightarrow \left[\begin{bmatrix} \text{“}detector\text{”} \\ \text{“}response\text{”} \end{bmatrix}, \begin{bmatrix} \varnothing \\ \text{“}response\text{”} \end{bmatrix}\right] \end{pmatrix}$$

$$\begin{aligned}
&pr([id_A, id_B]) := pr_{order}([id_A, id_B, order])\\
&\quad order := [\text{“}responder\text{”}, \text{“}detector\text{”}]
\end{aligned}$$

Now we turn our attention to the layered design of spatial models. The *lattice* model of Figure 5, which detects collisions between particles, appears to be one of the more complex submodels in our TPS hierarchy. One way to simplify it is to define the spatial relationships between subvolumes in a separate layer that has nothing to do with biology or physics. In two dimensions, the relationship between subvolumes can be described as a "rectangular lattice" in which each subvolume can interact with up to four adjacent subvolumes: one to the left, one to the right, one above, and one below. In 3D we would call this type of geometric configuration a "cubic lattice", and add two more adjacent subvolumes: one in front and one behind. Generalizing this type of configuration to an arbitrary number of dimensions n_{dim}, each subvolume of a "hypercubic lattice" interacts with its, at most, $2 \cdot n_{dim}$ adjacent neighbors.

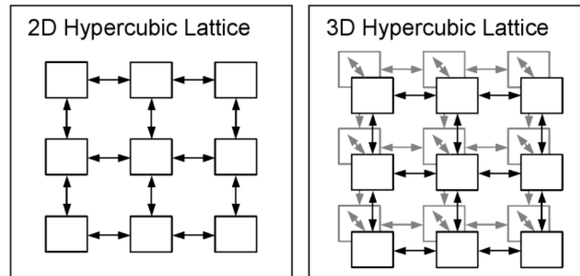

Figure 7: Two hypercubic lattices: one 3-by-3 and one 2-by-3-by-3

Recall that in a DEVS coupled model, the submodels are defined by M and the links by C. If the submodels of a coupled model have a hypercubic lattice configuration, then we can define C given only the lattice dimensions N. Note that for Figure 7 we have $N=[3,3]$ on the left and $N=[2,3,3]$ on the right. Given N, we also know the identities of each submodel; if $N=[2,3]$ for example, then the IDs of the submodels are the sets of coordinates $[0,0]$, $[0,1]$, $[0,2]$, $[1,0]$, $[1,1]$, and $[1,2]$. What we are missing is the DEVS model associated with each ID. We therefore require the function HLm_{DEVS}, which maps coordinates to models. In summary, we define a hypercubic lattice model with N and HLm_{DEVS}, which take the place of M and C. The new layer of parameters is introduced by the function HL_{DEVS}, the definition of which is outlined below.

$$
\begin{aligned}
&HL_{DEVS}\left([N, HLm_{DEVS}, pr]\right) := coupled_{DEVS}\left([M, C, pr]\right)\\
&\quad M\left(id_m\right) := \{an\ expression\ that\ depends\ on\ id_m,\ N,\ HLm_{DEVS}\}\\
&\quad C\left(src\right) := \{an\ expression\ that\ depends\ on\ src,\ N\}
\end{aligned}
$$

When it comes time to define the *lattice* model, our task is simplified in that the invocation of HL_{DEVS} takes care of the links between subvolume models. We may therefore focus our efforts on the scheduling of events within each subvolume. The scheduling is defined within $subV_{DEVS}$, which is passed to HL_{DEVS} as the second argument. Note that our *lattice* model has four layers: $lattice_{DEVS}$ at the top uses HL_{DEVS}, which uses $coupled_{DEVS}$, which defines $[\delta_{ext}, \delta_{int}, ta]$ at the bottom.

$$
lattice_{DEVS}\left(\left[N, a, \Omega_{\psi}, \Omega_{\psi\psi}\right]\right) := HL_{DEVS}\left([N, subV_{DEVS}, pr]\right)
$$

$$
\begin{aligned}
&subV_{DEVS}\left(coords\right) := [\delta_{ext}, \delta_{int}, ta]\\
&\quad \delta_{ext}\left([s, \Delta t_{el}, x]\right) := \{an\ expression\ that\ depends\ on\ s,\ \Delta t_{el},\ x,\ coords,\ N,\ a,\ \Omega_{\psi},\ \Omega_{\psi\psi}\}\\
&\quad \delta_{int}\left(s\right) := \{an\ expression\ that\ depends\ on\ s,\ coords,\ N,\ a\}\\
&\quad ta\left(s\right) := \{an\ expression\ that\ depends\ on\ s\}
\end{aligned}
$$

$$
pr\left([id_A, id_B]\right) := \varnothing
$$

Recall that the use of hierarchies in the previous section possibly increased the amount of code, yet aided the design of the model by facilitating its division into simpler parts. The use of layers brings analogous costs and benefits. Each extra layer may introduce overhead in the form of function calls and parameter transformations, but modelers will likely have an easier time working with several simple layers than one complex layer.

We now mention two design complications we have thus far neglected. The first complication relates to the priority of simultaneous events. If two events occur at the same simulated time in two different submodels of a coupled model, which submodel goes first? In the case of the non-spatial model TPS, we defined pr such that the *responder* would undergo a transition before the "*detector*". In this case the ordering was essential, as otherwise two collisions could be detected in a situation where one collision prevents the other. In the case of the *lattice* model, however, we refrained from ordering the subvolumes. Unlike the *detector* and the *responder*, the subvolume models all have the same role. Ordering them is at best unnecessary, and may even contribute to a bias in simulation results. Therefore, instead of resulting in a submodel ID, the priority function of the *lattice* model yields $\varnothing$. Our $coupled_{DEVS}$ layer interprets $pr\left([id_A, id_B]\right) = \varnothing$ as an instruction to order the internal transitions randomly. This convention may prove useful in various spatial simulations of biological systems.

The other complication is initialization. We must parameterize not only DEVS models, but their initial states as well. And the parameters of the initial states should also be grouped in layers. Although we neglected this complication in the formulas above, we addressed the issue in the actual model by including an initialization function in the result of each function with the subscript $_{DEVS}$. This is demonstrated by the following example, a definition of the DEVS *tracker* model accompanied by the initialization function $init_{tracker}$. The parameter Ψ provides the initial locations of all particles.

$$
tracker_{DEVS}\left(\left[N, a, \Omega_{\psi}\right]\right) := [init_{tracker}, tracker]
$$

$$
\begin{aligned}
&init_{tracker}\left(\Psi\right) := s\\
&\quad s := \{an\ expression\ that\ depends\ on\ \Psi,\ N,\ a,\ \Omega_{\psi}\}
\end{aligned}
$$

$$
\begin{aligned}
&tracker := [\delta_{ext}, \delta_{int}, ta]\\
&\quad \delta_{ext}\left([s, \Delta t_{el}, x]\right) := \{an\ expression\ that\ depends\ on\ s,\ \Delta t_{el},\ x\}\\
&\quad \delta_{int}\left(s\right) := \{an\ expression\ that\ depends\ on\ s\}\\
&\quad ta\left(s\right) := \{an\ expression\ that\ depends\ on\ s\}
\end{aligned}
$$

5 IMPLEMENTATION

The usefulness of a DEVS-based model design depends largely on the extent to which an implementation adheres to it. We chose the interpreted Python programming language because the ease with which functions can be passed as values, and the representation of infinity provided by the `numpy` extension, promised to simplify the task of adapting our mathematical formulas to code. With the Python code serving as proof-of-concept, a similar high performance simulator could be developed in a compiled language.

The code below is an implementation of the *tracker* atomic model. Note the similarity between this example and the mathematical formulas outlined at the end of the previous section. In general, a model's parameters may be used in its transition functions and its time advance function. In this case, `N`, `a`, and `Omega_psi` happen to appear in only the initialization function. The definition of `init_CL`, used on line 4 to initialize the state variable `CL`, is omitted.

```
1   def tracker_DEVS(N, a, Omega_psi):
2
3       def init_tracker(Psi):
4           CL = init_CL(Psi, N, a, Omega_psi)
5           RL = []
6           s = [CL, RL]
7           return s
8
9       def delta_ext(s, Delta_t_el, x):
10          [CL, RL] = s
11          [port, msg] = x
12          if port == "response":
13              RL_ = RL
14              RL_.append(msg)
15              s_ = [CL, RL_]
16          elif port == "arrival":
17              [id_A, coords] = msg
18              CL_ = CL
19              if coords not in CL[id_A]:
20                  CL_[id_A].append(coords)
21              s_ = [CL_, RL]
22          elif port == "departure":
23              CL_ = CL
24              [id_A, coords] = msg
25              if coords in CL[id_A]:
26                  CL_[id_A].remove(coords)
27              s_ = [CL_, RL]
28          return s_
29
30      def delta_int(s):
31          [CL, RL] = s
32          s_ = [CL, []]
33          Y = []
34          for response in RL:
35              [id_A, spc_A, t_A, u_A, v_A] = response
36              for coords in CL[id_A]:
37                  port = ["response", coords]
38                  Y.append([port, response])
39          return [s_, Y]
40
41      def ta(s):
42          [CL, RL] = s
43          if len(RL) > 0:
44              Delta_t_int = 0.0
45          else:
46              Delta_t_int = infty
47          return Delta_t_int
48
49      tracker = [delta_ext, delta_int, ta]
50
51      return [init_tracker, tracker]
```

Recall that a *response* message describes the new trajectory of a particle, and that the role of the *tracker* is to send these messages to various subvolumes. Accordingly, the external transition function `delta_ext` receives *response* messages and stores them in the state variable `RL`. Upon receipt of *arrival* and *departure* messages, it also updates the state variable `CL` that tracks the subvolume coordinates associated with each particle. The internal transition function `delta_int` takes *response* messages from `RL`, and duplicates them for each set of coordinates in `CL`. Also shown, the time advance function `ta` yields 0 if there are any response messages to output, and infinity otherwise.

The greatest problem we encountered during implementation was possibility of multiple variables referring to the same block of memory. The code above is written as if the operator = copies all data on the right to the variable on the left. In fact the assignments on lines 13, 18, and 23 copy references, and the new state `s_` returned on line 28 shares memory with the old state `s` passed in on line 9. We addressed old/new state conflicts in the simulator. In the simplified simulator below, the assignments on lines 65 and 69 abandon old states by replacing them. Note that `TX` and `TY` represent input and output values with associated times.

```
52  def simulate(model, s, TX):
53      [delta_ext, delta_int, ta] = model
54      [t, i, TY] = [0.0, 0, []]
55      while t < infty:
56          if i < len(TX):
57              [t_ext, x] = TX[i]
58          else:
59              [t_ext, x] = [infty, None]
60          t_int = t + ta(s)
61          if (t_ext == t_int == infty):
62              t = infty
63          else:
64              if t_ext <= t_int:
65                  s = delta_ext(s, t_ext - t, x)
66                  t = t_ext
67                  i = i + 1
68              else:
69                  [s, Y] = delta_int(s)
70                  for y in Y:
71                      TY.append([t_int, y])
72                  t = t_int
73      return [s, TY]
```

The new Python simulation code allowed us to implement the *TPS* DEVS model with the same hierarchical structure and layered design as outlined in Sections 3 and 4. The simulation results obtained to date demonstrate the application of this software to the interaction of vesicles and synapsins in neurons. As shown in Figure 8, particles representing vesicles and synapsins are tethered together in a cluster. Obeying conservation of momentum, the cluster reacts realistically to impacts from surrounding particles. The example demonstrates that DEVS can be used for spatial simulations of biological systems not only with cellular models, as previously shown, but with continuous-space models as well.

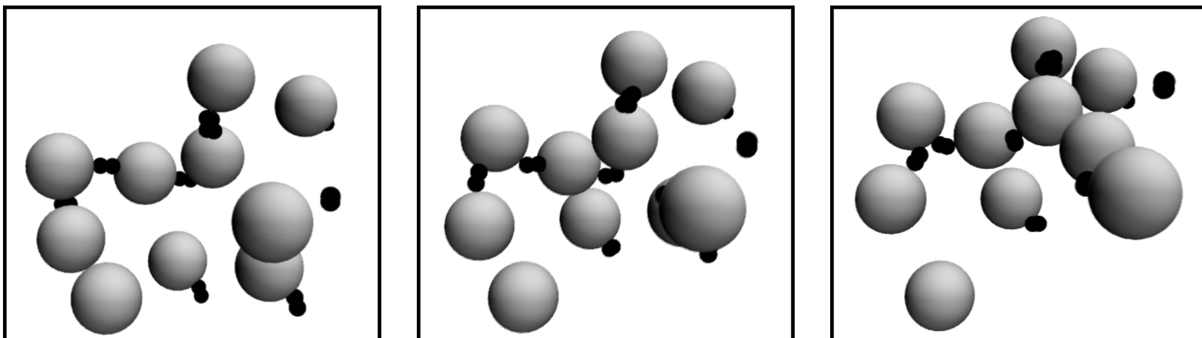

Figure 8: Three sequential snapshots of a simulation using the DEVS *TPS* model. The large spheres represent vesicles, neurotransmitter-containing structures found in nerve cells. Shown in black, synapsins are proteins that bind with vesicles.

6 EXAMPLE OF ALGORITHM INTEGRATION

We have described the DEVS-based design and implementation of a fairly complex model of a biological system. But arguably the TPS consists of a single algorithm, whereas at the outset we stated that one of the motivations for DEVS was the need to combine multiple algorithms.

Though accomplished without DEVS, a good example of algorithm integration is described in Jeschke and Uhrmacher (2008). This recent work combines the Next Subvolume Method, which tracks the concentrations of chemicals in various subvolumes, with an algorithm that, like the TPS, tracks the positions of relatively large individual particles. Looking at the diagram in Figure 9, the concentration of a chemical in subvolume $[0,0]$ may change due to a reaction within the subvolume, or a diffusion of the chemical to or from subvolume $[1,0]$ or subvolume $[0,1]$. The Next Subvolume Method handles this type of scenario. In the case of subvolume $[1,2]$, however, reaction and diffusion are complicated by the presence of the large particle. This requires an algorithm we will refer to as the Modified Next Subvolume Method.

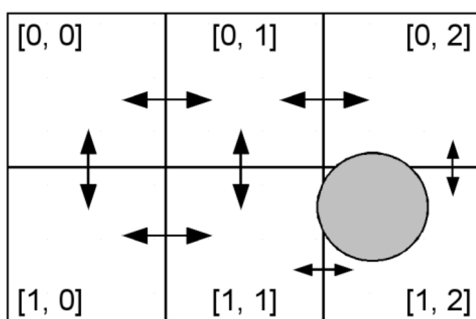

Figure 9: A model in which chemicals react within each subvolume, diffuse between subvolumes as shown by the arrows, and avoid the large particle

Now suppose we designed a DEVS model, named *MNSM*, that uses the Modified Next Subvolume Method to output the concentration of each chemical in each subvolume. We could then combine it with the *TPS*. Assuming that *MNSM* requires as an input the positions of all particles at regular time intervals, we would need a third DEVS model. The TPS sequencing DEVS model named *TPSS*, which was in fact implemented for visualization purposes, inputs *response* messages at the irregular times when collisions occur. Particle information is updated accordingly, and at regular time intervals the positions of all particles are output in *frame* messages. To integrate the TPS with the Modified Next Subvolume Method, we would link all three DEVS models as shown in Figure 10.

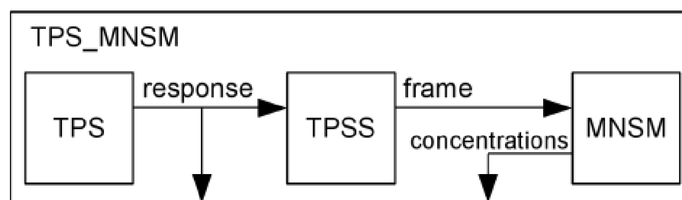

Figure 10: A hypothetical DEVS coupled model that combines two algorithms

The example outlined here is hypothetical. A DEVS-based approach to the work of Jeschke and Uhrmacher (2008) may or may not simplify matters. Furthermore, we do not rule out the possibility that in this case, the pi-calculus or another formalism may be more appropriate than DEVS. Our point is that, in all probability, it would be feasible to integrate these algorithms and others using DEVS and the techniques demonstrated in this paper, and that in general this approach may help one design realistic yet manageable biological models.

7 CONCLUSION

A hierarchy of DEVS models can be used to take what would be a complex routine, and reduce it to a set of simpler functions. Additionally, DEVS models can be defined in layers, and simulation code has been developed to support this approach. The presented DEVS-based TPS model has been successfully applied to vesicle-synapsin clusters, supporting the idea that DEVS is among the formalisms useful for the design of both cellular and continuous-space spatial simulations of biological systems. Future work may include the modeling of membranes and other biological structures with the TPS; the pursuit of more accurate physical parameters in order to improve the validity of TPS models; the DEVS-assisted integration of the TPS with other modeling algorithms; the implementation of a high performance version of our DEVS simulator; and a comparison of DEVS with the pi-calculus and other formalisms for the design of spatial biological models.

ACKNOWLEDGMENTS

We thank Dr. James J. Cheetham (Department of Biology, Carleton University), who aided in the design and validation of various models capturing the interaction of vesicles and synapsins in nerve cells.

REFERENCES

Elf, J., and M. Ehrenberg. 2004. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems Biology, IEE Proceedings* 1 (2): 230–236.

Gillespie, D. T. 1977. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry* 81 (25).

Goldstein, R., and G. Wainer. 2009. Simulation of Deformable Biological Structures with a Tethered Particle System Model. In *Proceedings of the 32nd Conference of the Canadian Medical and Biological Engineering Society (CMBEC)*.

Goldstein, R., G. Wainer, J. J. Cheetham, and R. S. Bain. 2008. Vesicle-Synapsin Interactions Modeled with Cell-DEVS. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, 813–821. Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, Inc.

Jeschke, M., and A. M. Uhrmacher. 2008. Multi-Resolution Spatial Simulation for Molecular Crowding. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, 1384–1392. Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, Inc.

Wainer, G., S. Jafer, B. Al-Aubidy, A. Dias, R. Bain, M. Dumontier, and J. Cheetham. 2007. Advanced DEVS models with application to biomedicine. In *Artificial Intelligence, Simulation and Planning (AIS)*.

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press.

Zeigler, B. P., T. G. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation*. Academic Press.

Zeigler, B. P., and S. Vahie. 1993. DEVS formalism and methodology: unity of conception/diversity of application. In *Proceedings of the 1993 Winter Simulation Conference*, ed. G. W. Evans, M. Mollaghasemi, E. C. Russell, and W. E. Biles, 573–579. Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, Inc.

AUTHOR BIOGRAPHIES

RHYS GOLDSTEIN received a B.A.Sc. degree (2003) from the Engineering Physics Department at the University of British Columbia (Vancouver, BC, Canada). He then worked in the mining industry, developing data analysis software and leading geophysical surveys in various parts of the world. He is now pursuing a M.A.Sc. in Biomedical Engineering from the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada). His email and web addresses are <rhys@sce.carleton.ca> and <www.sce.carleton.ca/˜rhys>.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the University of Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor and an investigator at the Centre for Advanced Studies in Visualization and Simulation (VSIM). He has held positions at the Computer Science Department of the University of Buenos Aires, and visiting positions at the University of Arizona, LSIS (CNRS), the University of Nice, and INRIA Sophia-Antipolis (France). He is an author of three books and over 190 research articles, and has helped organize over 70 conferences. His current research interests include modeling methodologies and tools, parallel/distributed simulation, and real-time systems. His email and web addresses are <gwainer@sce.carleton.ca> and <www.sce.carleton.ca/faculty/wainer>.