DEVS-Based Dynamic Simulation of Deformable Biological Structures

By

Rhys Goldstein, B.A.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Biomedical Engineering

Ottawa-Carleton Institute for Biomedical Engineering (OCIBME)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

September 2009

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis

DEVS-Based Dynamic Simulation of Deformable Biological Structures

submitted by

Rhys Goldstein, B.A.Sc.

in partial fulfillment of the requirements for

the degree of Master of Applied Science in Biomedical Engineering

Chair, Professor Howard M. Schwartz, Department of Systems and Computer Engineering

Thesis Supervisor, Professor Gabriel Wainer

Carleton University

September 2009

Abstract

Computer simulations are gaining popularity among biologists and medical researchers seeking better understanding of biological systems and an ability to predict their behaviour. There are many problems facing designers of such simulations. We focus on one particular challenge: the design of simulation code that can be easily understood and modified despite the complexity of biological systems and the algorithms used to model them.

The Discrete Event System Specification (DEVS) is a general modeling formalism, a set of conventions for the formal description of a wide range of systems that vary in time. Using DEVS, one can address the complexity of a biological model by subdividing it into a hierarchy of simpler submodels. We demonstrate this technique by presenting DEVS-based simulations that capture the dynamics of deformable biological structures.

This thesis makes two main contributions. The first pertains to the DEVS formalism and its application to simulations of biological systems. We have designed the first biological DEVS models with continuous space, meaning that positions of objects are quantified but not restricted to discrete points on a lattice. The second contribution is the invention of a method, called the tethered particle system (TPS), which we use to simulate deformable structures. The TPS is unusual in that it may capture a gradual process of deformation using only instantaneous impulses that occur in response to particle collisions.

A new DEVS simulator, and a DEVS model describing the TPS, were used to simulate both vesicle clusters in nerve cells and deformable membranes. Our results suggest that the TPS is a promising method for small-scale self-assembling deformable biological structures, but is not yet practical for macroscopic deformable objects subject to sustained external forces. With regard to DEVS, we argue that the formalism is a compelling option for the design of simulations of complex biological systems. Upper levels of a DEVS model hierarchy can be exploited for the integration of different biological simulation algorithms, whereas lower levels can be used to partition space. A minimalistic formulation of DEVS is presented for the benefit of readers who wish to adopt the formalism.

Acknowledgments

I thank my supervisor, Dr. Gabriel Wainer, whose advice and ongoing support have made graduate studies and research a very rewarding experience for me over the past two years. I also thank Dr. James J. Cheetham, who provided expertise in the biology of presynaptic nerve terminals, and numerous suggestions on how to model them.

Contents

1 Introduction

1.1 Overview

Simulation is becoming an increasingly common tool among biologists and medical researchers, complementing traditional experimental techniques. As Kitano explains in [1], experimental data is first used to form a hypothesis, and that hypothesis may be investigated with a simulation. Predictions made by the simulation can then be tested using *in vitro* and *in vivo* studies, and the new experimental data may lead to new hypotheses. This iterative process can be applied to basic research on biological systems, as well the development of drugs and other treatments.

Many different types of simulations are used in the field of biomedicine. Some model the reaction and diffusion of chemicals, for example, as done in the simulated E. coli bacterium of [2]. Our interest lies in simulations that capture the dynamics of deformable structures, which are frequently targeted at surgical planning and training [3], as well as the analysis of prosthetics [4]. Models of smaller-scale deformable biological structures are rarer, but examples include the simulated deformation of 8-μm red blood cells [5], and that of membrane-sculpting proteins on the 10-nm scale [6].

The simulation of biological systems poses many technical challenges. Among these challenges are the selection of accurate model parameters, the validation of simulation results, and the optimization of simulation code for computational efficiency. The problem that interests us is the development of well-designed simulation software; "well-designed" in the sense that modelers can understand their code, modify it, and have confidence that the desired computations are described. This is difficult for two reasons. First, the systems that biologists and medical researchers wish to simulate are often extremely complex, and models of those systems tend to become complex as a consequence. Second, realistic simulations may require the integration of multiple complex algorithms. Several algorithms would be necessary if one wished to simulate, for example, the deformation of a cell membrane, surrounded by reacting and diffusing chemicals, in a changing electric field.

In [7], it is noted that in many engineering applications, one can address the complexity of a large system by partitioning it into simpler subsystems. The paper suggests that complex biological systems be conceptually modularized in an analogous manner, and recommends that modeling formalisms be adopted to support this approach. The Discrete Event System Specification (DEVS) is one such formalism. Using DEVS, a simulation program is partitioned into a simulator and a model. One can address the complexity of the model by subdividing it into simpler submodels, and those submodels can in turn be subdivided as many times as needed. A DEVS model may thus take on a hierarchical structure. Although extra code must be written to define this structure, the fact that each submodel can be designed independently often compensates for the additional overhead.

In this thesis, we assert that DEVS can be used to develop well-designed software for the simulation of biological systems, and demonstrate the technique by presenting DEVS-based simulations that capture the dynamics of deformable biological structures. The development of these simulations involved the following tasks:

1. The invention of a method, called the tethered particle system (TPS), for simulating deformable structures.
2. The development of a DEVS simulator, as well as several accompanying functions that facilitate the design of hierarchical DEVS models.
3. The design of a hierarchical DEVS model for the TPS method.
4. The application of the DEVS simulator, and DEVS TPS model, to the simulation of deformable structures found in the presynaptic terminals of nerve cells.

1.2 Contributions

This thesis makes two main contributions. The first is the application of the DEVS formalism to simulations of biological systems involving continuous-space models. The second contribution is the invention of an impulse-based method for simulating the dynamics of deformable structures.

The first contribution advances the state of the art in modeling methodology in the field of biomedicine. Pre-existing biological DEVS models have been either non-spatial, meaning that the positions of biological entities are not quantified, or cellular, meaning that spatial coordinates are restricted to discrete points on a lattice. We have designed the first biological DEVS models with continuous space, meaning that coordinates are not restricted to discrete points. Continuous-space biological models are used, among other things, to simulate molecules in a cell that move in any direction. Our work demonstrates that DEVS can be applied to these types of simulations. Continuous-space DEVS models have been designed outside of the biological domain, but our approach differs in that we avoid the more complicated variants of the formalism.

Looking at the first contribution from a practical perspective, we provide an example for other programmers to follow when designing their own simulation software. The mathematics of our DEVS simulator can be applied to models of any domain. For the design of hierarchical biological models, others can adopt our recommended approach: to partition different algorithms at upper levels of a DEVS model hierarchy, and to partition space at lower levels.

Looking at the second contribution, the most common methods for simulating the dynamics of deformable structures are mass-spring-damper systems and the finite element method. Our method, the TPS, differs in that it uses only impulses to alter motion. Impulse-based methods have previously been used to simulate rigid bodies, but are generally neglected or considered unsuitable for objects that deform. We demonstrate that impulse-based methods provide a relatively simple way to allow deformable biological structures to assemble themselves from rigid particles representing proteins and other biological entities. Also, with an impulse-based simulation, it is easy to incorporate the random motion exhibited by these small biological objects.

Aside from the novel application of DEVS and impulse-based simulation techniques, other contributions are made. As part of the TPS, we present a new approximation for resolving simultaneous and nearly-simultaneous collisions of rigid bodies. Also, our simulation code and presynaptic nerve terminal model is being used by Dr. James J. Cheetham, a biologist at Carleton University, to study neurotransmission.

Three papers that describe our work have been either published or accepted for publication. Listed below, these papers present the key formulas of the TPS, its application to presynaptic nerve terminals, and its design and implementation using DEVS.

- Rhys Goldstein and Gabriel Wainer. 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)*, Calgary, AB, Canada, 2009.
- Rhys Goldstein and Gabriel Wainer. Simulation of a Presynaptic Nerve Terminal with a Tethered Particle System Model, To appear in *Proceedings of the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Minneapolis, MN, USA, 2009.
- Rhys Goldstein and Gabriel Wainer. DEVS-Based Design of Spatial Simulations of Biological Systems, To appear in *Proceedings of the Winter Simulation Conference (WSC)*, Austin, TX, USA, 2009.

1.3 Organization

The first three sections of this document are introductory in nature. Section 3 describes the scope and purpose of the project, referring to the previous research outlined in Section 2.

Section 1	Introduction
Section 2	Review of the State of the Art
Section 3	Problem Statement

The next two sections describe how the problem stated in Section 3 was tackled.

Section 4	Model and Simulator Design
Section 5	Simulation of Deformable Biological Structures

Section 4.1 describes the TPS method, ignoring DEVS, whereas Section 4.2 focuses on DEVS and ignores the TPS. The DEVS formalism is applied to the TPS in Section 4.3. The Python code described in Section 4.4 was used to simulate various deformable biological structures and related processes. These simulations are presented in Section 5.

The remaining two sections discuss the project in hindsight, the dynamic simulation of deformable structures using impulses, the application of DEVS to biological models in general, and future work.

Section 6	Discussion
Section 7	Conclusion

Following a list of references, several formulas-intensive appendices are included for the benefit of those readers who would like to reproduce parts of our work in their own simulation code. A programmer who wishes to create their own TPS simulation, or implement their own DEVS simulator, will find many of the cases they need to consider explicitly handled. The appendices can also be referenced to clarify ambiguities in the informal descriptions of Sections 4 and 5.

Appendix A	Notation
Appendix B	Tethered Particle System Formulas
Appendix C	DEVS Formulas
Appendix D	DEVS TPS Model Formulas
Appendix E	Presynaptic Nerve Terminal Model Parameters

2 Review of the State of the Art

2.1 Simulation of Biological Systems

We use the term “model” to refer to a description of certain aspects of a real-world system. “Simulation”, by contrast, is a process by which a model is used to imitate a system. A model can be an informal description involving diagrams and prose, but in this document the term will generally refer to a formal description involving mathematical formulas. For our purposes, the task of simulating a system can be assumed to include the design of a formal model, and the use of computer technology to perform calculations based on the model’s formulas. The term “simulator” refers to a program designed to execute these calculations given a model as its input.

Models of biological systems can be designed as hierarchies [8]. Nodes in a “hierarchical model” typically reflect real-world entities in the represented system. Depicted in Figure 1 is a hierarchical model of a neuron, or nerve cell. The upper level of the hierarchy represents the neuron itself, and nodes in the lower level represent its components. Arrows indicate the direction in which a signal, or action potential, propagates through the neuron from one component to another. Note that the illustrated model is hypothetical, and that a typical nerve cell does not have exactly two terminals.

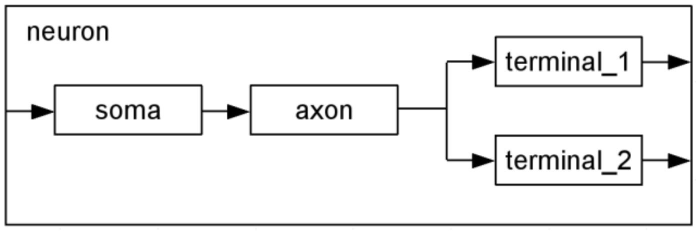


Figure 1: An illustration of a hypothetical, hierarchical, non-spatial model of a neuron.

We classify models as either “spatial” or “non-spatial”. Although the soma, axon, and terminals of a real-world neuron have positions, we would describe the Figure 1 model as non-spatial because positions are represented qualitatively. In the spatial model of Figure 2, the shape and position of a neuron can be represented quantitatively by sets of coordinates that identify cells in a lattice.

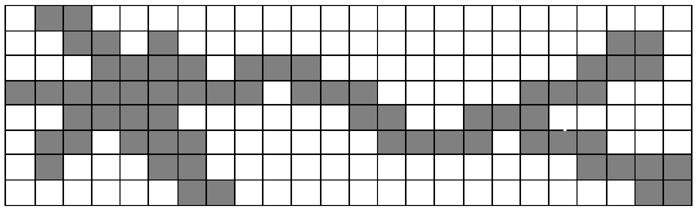


Figure 2: A hypothetical cellular model in which the shape of a neuron is captured by a set of shaded lattice cells.

There are a number of ways of representing space in biological models [9]. We will classify spatial models as either "cellular" models, like the one in Figure 2 above, or "continuous-space" models, like the one in Figure 3 below. In a continuous-space model, coordinates are not restricted to discrete points on a lattice.

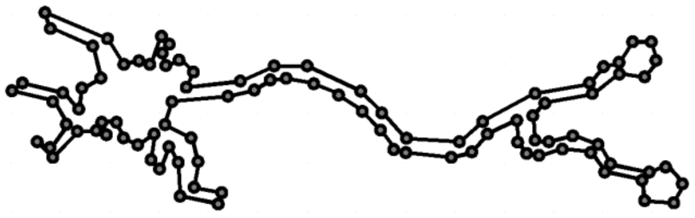


Figure 3: An example of a continuous-space model, in which the shape of a neuron is captured by a set of connected points.

Each type of model has its own advantages. With a non-spatial model such as that in Figure 1, it is easy to define a hierarchy based on anatomical features or other real-world entities. With spatial models, real-world objects can be harder to differentiate. Where is the exact boundary, for instance, between the soma and the axon in Figure 2? One advantage of the cellular model is that, by associating different values with different lattice cells, one can represent a property that changes along the length of a neuron's axon. Note, however, that a rotation or deformation of a neuron would be easiest to simulate by moving the points in the continuous-space model of Figure 3.

Examples of non-spatial biological models include those based on the Gillespie Algorithm

[10]. Popular in systems biology, this algorithm is used to simulate changes in the concentrations of various reacting chemicals over time. The key assumption is that a simulated chemical system is well mixed, in which case the probability distribution governing the next reaction is independent of the coordinates of any individual molecule. Coordinates are therefore excluded. After each simulated reaction, concentrations are updated, and the type and timing of the next reaction are randomized.

The Gillespie Algorithm has been adapted many times since its introduction in 1977. One recent non-spatial model, presented in [11], uses Gillespie's equations but treats a biological system like a factory in which machines (enzymes) take unprocessed items (chemical reactants) from a queue (of chemical reactions in progress) and deliver processed items (chemical products).

In 2004, the Gillespie Algorithm was extended to spatial cellular models [12]. In the Next Subvolume Method, the assumption is made that the chemical system in each lattice cell, or subvolume, is well mixed. Different subvolumes have different concentrations, however, and these concentrations affect not only reactions but also the diffusion of chemicals from one subvolume to another. The Next Subvolume Method is so-named because it requires a simulation to repeatedly identify the subvolume in which the next chemical reaction or diffusion will occur.

The Gillespie Algorithm and the Next Subvolume Method are examples of "population-based" methods for simulating chemical systems. Alternative methods are "particle-based", tracking the positions of individual particles. Population-based methods have an obvious advantage in that they accommodate the simulation of arbitrarily-large quantities of a chemical. They are inappropriate for biological systems with complex heterogeneity, however, due to the assumption that systems are well mixed. Population-based methods also suffer from "combinatorial explosion": if a molecule can exist in hundreds of different states, and if each state must be represented with a different chemical species, then the high number of possible reactions renders simulation intractable [13].

Particle-based methods require spatial models, as the positions of individual particles are quantified by sets of coordinates. GridCell is an example of a particle-based simulator that operates on cellular models [14, 15]. Each cell in a 3D cubic lattice contains at most one particle. At fixed time intervals, each particle may move randomly to one of its 26 neighboring cells, or undergo a reaction that causes itself and possibly a neighboring particle to be replaced with one or two new particles. One of GridCell's strengths is its inherent ability to simulate molecular crowding in a computationally efficient manner. Molecular crowding occurs when the density of particles in a particular region impedes particle motion and reactivity.

GridCell's most obvious weakness is its discretization of space. Other particle-based simulators have been developed for continuous-space models, with particle positions described by continuous coordinates. In the MCell program [16], particles move randomly in any direction through 3D space. When a particle encounters a surface, a cell membrane for instance, a reaction may occur. ChemCell and Smoldyn are similar to MCell, but allow reactions to occur between particles in solution [17, 18, 19].

Like GridCell, but unlike the Gillespie Algorithm, reactions in MCell, ChemCell, and Smoldyn all occur at fixed time steps. Green's Function Reaction Dynamics (GFRD) is an alternative particle-based method [20]. It also uses continuous-space models, but advances time in a sequence of irregular intervals. At any given simulated time, the length of the next time interval is calculated from reaction probability distributions that depend on the positions of each particle.

All of the simulation methods described above are examples of discrete event simulations, for in each case the state of a real-world system is represented in part by a set of values that change only at discrete points in time. A specific state change, occurring at a specific point in simulated time, is referred to as an "event". In the Gillespie Algorithm, for example, the recorded state would be the set of concentrations of each chemical. That state remains constant until a reaction event occurs, at which point the recorded concentrations change instantaneously. The phrase "discrete event simulation" usually refers to a method like Gillespie's or GFRD in which time is advanced by irregular intervals.

2.2 DEVS

With the conviction that a novel theory was needed for discrete event simulation, Bernard Zeigler invented the Discrete Event System Specification (DEVS). Described in detail in [21], DEVS is a general modeling formalism, a set of conventions for the formal description of a wide range of systems that vary in time. The formalism emphasizes the distinction between system states and constant states, the separation between simulators and models, and the design of hierarchical models. Many variants of DEVS have been proposed and implemented since the formalism was first introduced in 1976.

Note that the state of a real-world system may change continuously with time, whereas the recorded state in a discrete event simulation changes only at discrete times. DEVS addresses this discrepancy by representing the state of a real-world system with a "system state", which is distinct from the "constant state" stored by a computer. The system state has two components: the constant state and the time elapsed since the previous event. Because the "elapsed time" changes continuously with simulated time, so does the system state. In many other modeling formalisms, transition functions invoked at each event calculate new constant states from current constant states. The transition functions in DEVS calculate new constant states from current system states [22].

Consider, for example, a simulation of a ball bouncing on a floor. We would like to simulate the height of the ball above the floor, which of course changes continuously. We could do this by choosing as the constant state the upward speed of the ball immediately after each event. Events would coincide with impacts between the ball and the floor. Given the system state at any time, which consists of the constant state v and the elapsed time Δt_{el}, one can obtain the height of the ball (eg. $v \cdot \Delta t_{el} - (1/2) \cdot 9.81 \cdot \Delta t_{el}{}^2$).

Having explained the distinction between system states and constant states, will we from here on adopt the usual convention, drop the phrase "constant state", and use in its place the word "state". To summarize, the state remains constant between events in a DEVS-based simulation, but the system state varies continuously because it includes the elapsed time.

Recall from Section 2.1 that a model is a description of a system. A “DEVS model” is specific type of model: a formal description from which one can derive the mathematical sets and functions in the tuple of (1). If the variables in this tuple are defined explicitly, then the DEVS model is an “atomic model”.

$$\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle \tag{1}$$

A key principle of DEVS is that the model and the simulator remain separate. When designing an 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. At any given time during a simulation, a DEVS model has an associated state s ($s \in S$). After an event, the simulator evaluates the time advance function ta to determine the time when the next internal transition may occur. 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 of the model. If, however, an input value x ($x \in X$) is received before the time $ta(s)$ elapses, then the simulator applies the external transition function δ_{ext} instead to obtain the new state. A DEVS simulator is model-independent in the sense that it should carry out a simulation for any well-defined DEVS model, regardless of what the model represents.

Note that the external transition function δ_{ext} accepts three arguments: the current state s, the elapsed time Δt_{el}, and the input value x. As stated earlier, it is a function of the system state because its arguments include both s and Δt_{el}. The internal transition function δ_{int} accepts only one argument: the state s. Effectively, if not technically, δ_{int} is also a function of the system state, for the elapsed time is guaranteed to be $ta(s)$.

Traditionally, a DEVS model is classified as being either an atomic model or a coupled model. “Coupled models” are composed of submodels, and these submodels are themselves either atomic or coupled. The nesting of coupled models within coupled models is the mechanism by which DEVS supports hierarchical model design. To design a coupled model, one defines the variables in the tuple of (2). These variables describe how messages are passed from one submodel to another during a simulation. Here “message” refers to a value that originates from one DEVS model as an output ($y \in Y$), and/or arrives at another DEVS model as an input ($x \in X$). It has been proven that for every coupled model expressed in the form of (2), one can define an equivalent atomic model in the form of (1). This property is referred to as "closure under coupling".

$$\langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle \tag{2}$$

Suppose we were to define the neuron model of Figure 1 in Section 2.1 as a DEVS coupled model. We could start by defining four atomic models to serve as submodels, one representing the soma, another the axon, and two more for the terminals. In the coupled model, representing the entire neuron, the variable D would be the set of model identifiers; $\{``soma", ``axon", ``terminal\_1", ``terminal\_2"\}$, for example. The set of models $\{M_d \mid d \in D\}$ would include the tuples of the four atomic models. The external input coupling set EIC would represent the arrow leading into the soma, the external output coupling set EOC would represent the arrows leading out of the terminals, and the internal coupling set IC would represent the arrows between submodels. Messages would be passed according to these links. A $Select$ function would be defined to determine which submodel undergoes an internal transition first in the event of a tie.

DEVS has been applied extensively, not only to non-spatial hierarchical models like the one in Figure 1, but also to cellular models. In [23], a watershed is described with a DEVS coupled model composed of submodels representing cells in a 3D lattice. Each submodel records the water retained in a certain region of the watershed, which accumulates as water influx messages are received from neighboring submodels. If the region's water capacity is reached, water outflow messages are sent to neighboring submodels.

Another way to apply DEVS to cellular models is offered by the Cell-DEVS formalism [24]. Cell-DEVS is an extension of DEVS, so the variables in (1) and (2) are not defined explicitly. For every Cell-DEVS coupled model describing a cellspace, however, one can define an equivalent DEVS coupled model. The advantage of Cell-DEVS over DEVS is that, in Cell-DEVS, the communication of a cell's state to neighboring cells is implicit in the formalism. A modeler must only define a neighborhood of cells, which is simpler than describing messages. Traditionally, simulations with cellular models are implemented with nested loops updating cell states at regular time intervals. Cell-DEVS allows cell transitions to occur asynchronously. Also, Cell-DEVS models can be integrated with other DEVS models, cellular or otherwise.

The application of DEVS to continuous-space models, like one in Figure 3, is rare. One such effort is described in [25], in which the front of a forest fire is represented by a set of connected points. For each point there is a separate DEVS atomic model. Because points are added as a fire grows, the simulation requires the use of a DEVS variant designed for models with dynamic structures [26]. The claimed advantage of the continuous-space fire spreading model is that, while a cellular model might distinguish between cells that are all on fire, the use of vectors eliminates this overhead. One disadvantage is that, in order to represent temperature variations within a fire as done in the cellular model of [27], multiple nested boundaries would be needed.

Another continuous-space model designed with DEVS is presented in [28]. Although it also represents a forest fire, it differs from the model of [25] in that vectors are used for the positions of fire fighters. The fire itself is represented by a set of cells in a lattice. Similar to [25], the model in [28] has a dynamic structure, but in this case the dynamic structure is used to create and delete links between DEVS fire fighter models and DEVS cell models.

2.3 DEVS-Based Simulation of Biological Systems

Despite the popularity of the DEVS formalism and the widespread use of simulation in the study of biological systems, the application of DEVS to biological models is relatively rare. One notable group, led by Adelinde M. Uhrmacher at the University of Rostock, applies DEVS to various non-spatial models of biological systems. The coupled model in [29] represents an enzyme called “RNA polymerase” as a message that is passed to a submodel called “promoter”. Upon receipt of the message, the state of the promoter depends on whether the RNA polymerase can bind to it. The model is non-spatial because, although the relationships between RNA polymerase and other biological structures are tracked, their positions in space are not.

Gabriel Wainer leads a group at Carleton University that has demonstrated the application of Cell-DEVS to cellular biological models [30]. One example features tumor growth impeded by immune cells [31].

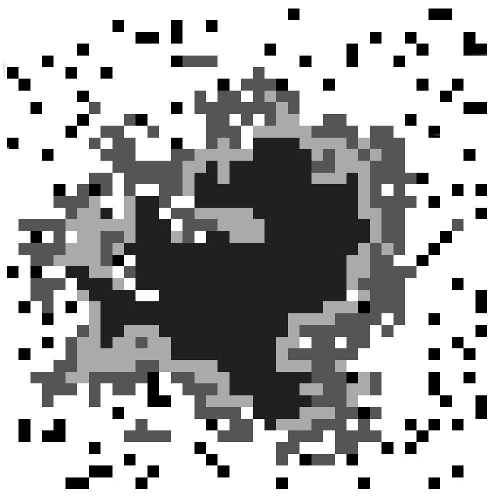


Figure 4: A snapshot of a Cell-DEVS simulation. Immune cells attack a growing tumor, which consists of necrotic cells, surrounded by dormant cells, surrounded in turn by proliferative cells.

It is easy to envision how continuous-space biological models, like those of MCell, could be designed with DEVS. One could alter the forest fire example in [28], for instance. Instead of fire fighters approaching a fire front, vectors could be used to track chemicals or proteins approaching a cell membrane. Nevertheless, a literature search did not yield any continuous-space DEVS models of the kind of biological system that might be found in a living organism.

When considering the use of DEVS for the design of a discrete event simulation, the question one must ask is not whether it is possible to use the formalism at all, but whether submodels can be chosen to effectively exploit its hierarchical nature. It is theoretically possible to represent a complex biological system with a single atomic model. This might well be preferable to not using any formalism at all, as the use of DEVS at least guarantees the separation of model and simulator. With only one atomic model, however, the δ_{ext} and δ_{int} functions might be thousands of lines long and contain many loops and data structures. To fully benefit from DEVS, coupled models must be used to subdivide a model into simpler submodels. The result is many smaller δ_{ext} and δ_{int} functions.

The design of suitable hierarchies is a challenge, particularly for models of biological systems that contain many interacting components. In [32], researchers at the University of Rostock compare the DEVS formalism to three of its alternatives: stochastic Petri Nets, stochastic π-Calculus, and StateCharts. They design hierarchies by coupling DEVS submodels that represent entities of biological systems, such as cell membranes and nuclei. Interactions between such entities are simulated by the passing of messages between the DEVS models. Because messages are passed independently of one another, the claim is made that DEVS is unsuitable for the modeling of interactions that involve three or more entities.

> "[When using DEVS or StateCharts,] the involved reactants and products become the focus of interest. In combination with an asynchronous interaction via events, this does not facilitate the representation of chemical reactions that involve several components at once. Although reactants and products can be considered as independent entities, they have to react in synchrony." (quoting Ewald et al. from [32])

The problem described above arises not only from the use of DEVS, but also from the decision to base model hierarchies on biological entities. However, as this happens to be a very intuitive design strategy, the point is worth noting. Several variants of DEVS have been been proposed to improve its efficacy for the modeling of biological systems, specific examples being ρ-DEVS [33] and ml-DEVS [34]. Inevitably, as DEVS is modified to simplify models of biological systems, the formalism itself becomes more complicated and more difficult to learn. Along with the usual sets of submodels and links, coupled models in ml-DEVS (MACRO-DEVS models) include a downward output function λ_{down}, a downward value coupling function v_{down}, an upward activation function act_{up}, and a structural change function sc.

2.4 Biological Simulation Algorithm Integration

Though accomplished without DEVS, a good example of biological algorithm integration is described in [35]. This works integrates the Next Subvolume Method with an algorithm that tracks the positions of individual molecules, resulting in a method that is both population-based and particle-based. The model, like that of the fire fighter simulation in [28], can also be seen as both cellular and continuous-space. Future models of complex biological systems will likely go beyond the integration of methods for the reaction and diffusion of chemicals, and attempt to incorporate algorithms for the propagation of electric fields, the motion of fluids, and the dynamics of deformable structures.

The desire to integrate biological algorithms, coupled with the belief that different algorithms are best described in different ways, has led to “multi-formalism modeling”. This approach involves the composition of a single model from submodels defined using different formalisms. The framework James II was developed to combine stochastic and spatial variants of the π-Calculus with variants of DEVS and a formalism called Beta Binders. It is described in a paper titled, in part, *One Modelling Formalism & Simulator Is Not Enough!* [36].

As explained in [37], there are different approaches to multi-formalism modeling. One strategy exploits the fact that models of certain formalisms can be transformed into models of certain other formalisms. It has been shown, for example, that a StateCharts model can be transformed into an equivalent DEVS model [38]. The strategy is therefore to transform all models, regardless of the formalism used to define them, into equivalent models of some common formalism. DEVS is often suggested as an appropriate common formalism [39]. An alternative strategy is to have a separate simulator for each modeling formalism, and ensure that all simulators can inter-operate during a simulation.

The advantage of multi-formalism modeling is that a modeler can select the formalism that best describes each integrated algorithm. One drawback is that, in order to understand the entire model, one must be familiar with all of the formalisms used. The E-Cell System demonstrates the integration of biological algorithms with a single general modeling formalism. Although E-Cell is described in [40] as a multi-formalism framework, the word “formalism” is interpreted as a mathematical model that must be adapted for simulation. What is referred to as a “meta-algorithm”, with tuples representing “models” and “steppers” and “processes”, is a novel modeling formalism analogous to DEVS.

2.5 Dynamic Simulation of Rigid Bodies

We use the phrase “dynamic simulation” to indicate the simulation of motion using laws of classical dynamics. We consider only those dynamic simulation methods that involve continuous-space models, as cellular models tend to restrict motion. This section reviews methods for the dynamic simulation of rigid bodies. In these methods, object deformation may be represented by a loss of kinetic energy, for example, or an overlapping of objects. If an object’s changing shape is modeled, however, then we classify it as a deformable structure instead of a rigid body.

“Impulse-based” methods are perhaps the most obvious approach to the dynamic simulation of rigid bodies. An impulse-based method involves two tasks: collision detection, the task of calculating the time at which any two objects come into contact, and collision response, the task of computing the new trajectories of two colliding objects. In response to a collision, the trajectory of an object changes instantaneously in simulated time. The instantaneous change in the momentum of the object is referred to as an “impulse”.

Because impulse-based methods assume instantaneous contacts, the approach seems inappropriate for the modeling of stable contacts. If a ball is rolling across a table, for example, it remains in contact with the table for a length of time. In his 1996 Ph.D. thesis, Brian Mirtich demonstrated that stable contacts could be modeled as sequences of independent collisions [41]. Consider an impulse-based simulation of an object bouncing along a horizontal surface. Provided each bounce was sufficiently short in height and duration, the model could accurately represent a ball rolling across a table.

Another well-known drawback to impulse-based methods is the possibility of simultaneous or nearly-simultaneous collisions. Consider a situation in which a small object is directly between two much larger approaching objects. After the first large object hits it, the small object may end up travelling back and forth between the larger objects in a long sequence of nearly-simultaneous collisions. This might require considerable computational effort. If kinetic energy is lost in each collision, then it is possible for the sequence of collisions to become infinite, slowing the simulation to a halt.

One advantage to impulse-based simulation is the simplicity of the method. If one is to implement an algorithm to detect collisions between pairs of objects, which is necessary in perhaps all of the competing methods, it is a simple matter to apply the law of conservation of momentum to give the two objects new trajectories. The constraint-based method of [42], by contrast, may compute new trajectories for more than two objects simultaneously. This is done by minimizing a linear function constrained by a system of linear inequalities. Depending on the model, this problem may be NP-hard, may have no solutions, or may have multiple different solutions.

Both impulse-based methods and the constraint-based method of [42] prevent the penetration of objects. "Penalty methods" differ in that they allow approaching objects to overlap slightly upon colliding. Typically, a spring is temporarily inserted between colliding objects; the more the objects overlap, the stronger the restoring force of the compressed spring [43].

Generally speaking, dynamic simulations of rigid bodies tend to be deterministic. An object with some velocity will continue moving in the same direction with the same speed until it is acted upon by a force or until it undergoes a collision. The particle-based simulations mentioned in Section 2.1 are nondeterministic, as particles move randomly through space unaffected by their own momentum.

2.6 Dynamic Simulation of Deformable Structures

We now review methods for the dynamic simulation of deformable structures, as opposed to rigid bodies. Note that phrase “dynamic simulation” excludes a wide range of methods for modeling deformable structures. An algorithm that fits a spline to a cross-sectional image of a human lung, for example, certainly models a deformable structure. But unless the motion of the lung is predicted from laws of physics, we would not consider it a dynamic simulation.

One way to model a deformable structure is with a set of point masses. Each mass is connected to its neighbors with a spring and possibly a damper. A spring applies a force that, depending on its present length, attracts or repels the masses on either end. A damper applies a force that decreases the relative speed of the masses on either end. These “mass-spring-damper” systems can be used to simulate the dynamics of deformable structures by predicting the acceleration of each mass, at regular time intervals, according to spring, damper, and external forces [44]. The mass-spring-damper method is essentially a penalty method like those described in Section 2.5 for rigid bodies. The difference is that the springs in Section 2.5 are inserted temporarily between detached colliding objects, whereas in this case the springs tend to be permanent and the point masses do not necessarily represent distinct objects.

Some mass-spring-damper models use spherical particles instead of point masses. This technique is used in [45] to simplify the detection of collisions between deformable objects. Each object is composed of several overlapping spherical particles. Instead of detecting collisions between the possibly-concave surfaces of these objects, only collisions between particles are considered. A similar approach is taken in [46], which incorporates friction, viscous forces, and the fracture of deformable objects.

Mass-spring-damper methods are used extensively in computer graphics. They are considered computationally efficient, but not particularly accurate. Incompressible deformable objects and nearly-rigid thin membranes are difficult to model, and appropriate spring parameters may be difficult to determine. Stiff objects, modeled using springs with large restoring forces, threaten the stability of mass-spring-damper simulations. Techniques have been developed to address the stiffness problem. The simplest solution is to decrease the time step, though this increases computational costs.

A popular alternative to mass-spring-damper systems is the “finite element method” (FEM) [47]. FEM actually refers to a more general mathematical technique, but we will refer to it as a dynamic simulation method for deformable structures. In an FEM model, a deformable object is represented as a set of adjacent polyhedra. Each polyhedron, or “element”, has a set of vertices, or “nodes”. Although recorded attributes are associated with each node, material properties can be obtained at every point in each element by interpolating the attributes of each node. Positions of each node may change at each time step. FEM simulations are considered to be more accurate, but also more computationally intensive, than those based on mass-spring-damper models. The FEM is most efficient with metals and materials that exhibit relatively little deformation. Highly deformable materials, like soft biological tissues, require frequent re-calculation of large mass and stiffness matrices that depend on the positions of the nodes.

Impulse-based methods, like those used for rigid bodies, tend to be either neglected or avoided for the dynamic simulation of deformable structures. A literature search revealed an “impulse response deformation model” [48], which does simulate the dynamics of deformable structures, but is not an impulse-based method despite its name. In this case the term “impulse” refers to an initial perturbation in an object’s shape. Convolution integrals are used to track the object’s shape after the perturbation.

The possibility of applying impulse-based methods to deformable objects is acknowledged in [49], but quickly dismissed with the assertion that “impulse-based methods assume short contacts only, and therefore they are not suitable for soft objects”. The argument is intuitive: impulses are instantaneous changes in momentum, whereas the deformation of an object is a gradual process that takes place over time. In [41], Mirtich states that the strongest restriction of impulse-based methods is that models are comprised of only rigid bodies.

The pre-existing method that most closely fits the phrase “impulse-based dynamic simulation of deformable structures" was developed recently to simulate inextensible cloth [50], as well as volume-conserving deformable objects [51]. In both cases, impulses are applied simultaneously to all particles in a structure at regular time intervals. Because the purpose of these impulses is to constrain either the distances or volumes between the particles, the method can be classified as constraint-based as well as impulse-based. The simulations of [50, 51] differ from impulse-based rigid body simulations in that, in the latter, impulses occur in response to collisions and not at regular intervals.

2.7 Simulation of Presynaptic Nerve Terminals

An action potential, a signal that propagates along the axon of a nerve cell, will ultimately arrive at a presynaptic nerve terminal. Inside this compartment are tens or hundreds of neurotransmitter-containing sacs called synaptic vesicles [52]. Some of those vesicles are docked to a region of the membrane called the active zone. When an action potential arrives, certain docked vesicles may release their neurotransmitters outside of the compartment. This may provoke another action potential in an adjacent nerve cell. When a docked vesicle releases its neurotransmitters in this fashion, it may undergo a process called exocytosis in which it fuses with the nerve cell membrane. Alternatively, the vesicle may separate from the membrane and drift back into the compartment.

Also present in a presynaptic nerve terminal are protein called synapsin [53]. Synapsins, which may number in the hundreds within the compartment, bind with vesicles to form clusters. An action potential triggers chemical reactions that cause synapsins to lose their affinity for vesicles. This disrupts the clusters, freeing vesicles. Clusters may reform before the next action potential arrives.

Figure 5 illustrates the clustering of vesicles and other features of a presynaptic nerve terminal.

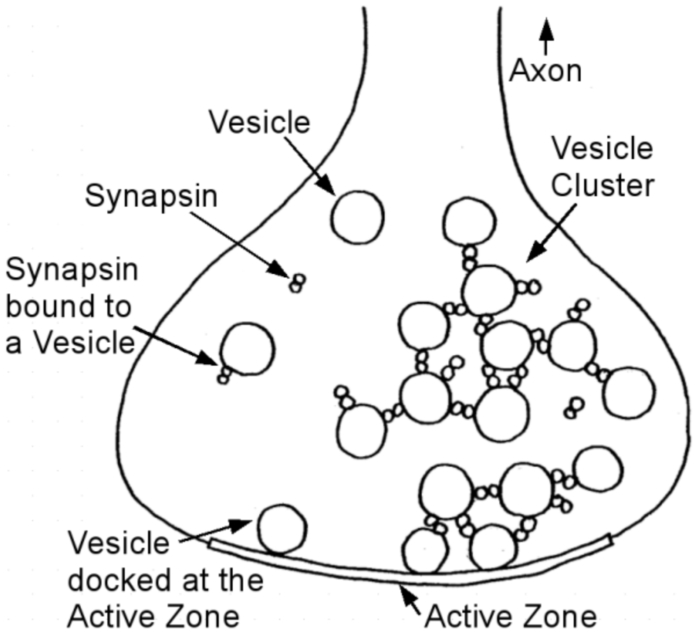

Axon
Vesicle
Vesicle Cluster
Synapsin
Synapsin bound to a Vesicle
Vesicle docked at the Active Zone
Active Zone

Figure 5: An illustration of a presynaptic nerve terminal.

The simulation of a presynaptic nerve terminal is motivated by a desire to better understand the physiology of the human brain. One may be able to use simulation results, for example, to predict how a decreased synapsin concentration affects the size of vesicle clusters. Such results could be used to investigate the theory that vesicle clusters form in order to ensure that, should action potentials arrive at a high frequency, vesicles remain available in the vicinity of the active zone [54].

In [55], the MCell program was used to simulate the reaction and diffusion of chemicals around vesicles docked on a presynaptic membrane. This simulation involved a continuous-space model in which vesicle positions were based on measurements of an actual presynaptic nerve terminal. Neither the vesicles nor the membrane could move, however.

In the 2D cellular model of [56], vesicles and synapsins were free to move and form clusters, and the clusters themselves could move as well. But because vesicle and synapsin positions were restricted to lattice cells, the clusters could neither rotate nor deform. The simulation was designed using the Cell-DEVS formalism mentioned in Section 2.2. A simulation snapshot is shown in Figure 6.

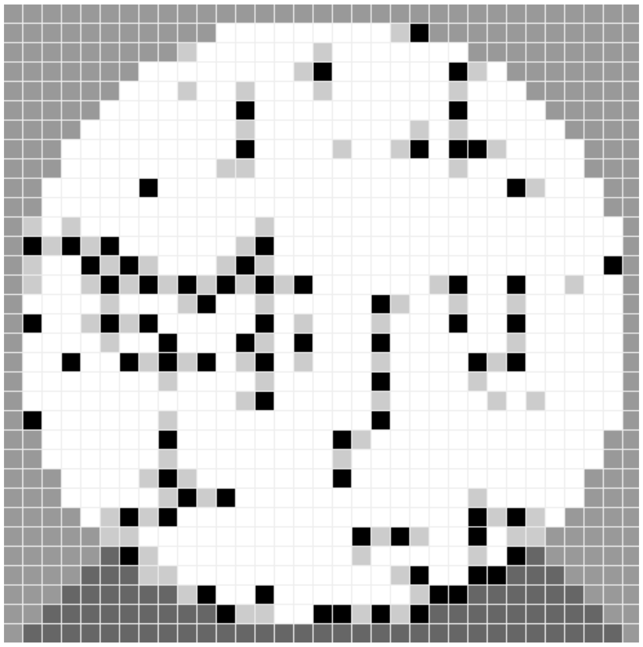


Figure 6: A cellular model of a presynaptic nerve terminal. Vesicles (black cells) form clusters with synapsin (light grey cells) inside a circular compartment.

Aside from the work described in this thesis, a literature search revealed no continuous-space models of presynaptic nerve terminals designed to capture the dynamics of vesicle clusters as deformable structures.

3 Problem Statement

The problem tackled by this thesis is the application of the DEVS formalism to the dynamic simulation of deformable biological structures. Three decisions were made at the outset of the project that narrowed its scope. First we had to choose whether to design our own DEVS simulator or use existing software. Because existing DEVS tools had not been developed with continuous-space biological models in mind, we decided to design our own. We also had to choose a method for the dynamic simulation of deformable structures. For reasons explained below, we chose to invent our own impulse-based method. Finally, although our objective was to develop simulation techniques that could be applied to a range of biological systems, we had to decide which deformable biological structures to focus on. Our priority was the vesicle clusters of presynaptic nerve terminals, a main interest of the biologist James J. Cheetham with whom we were collaborating. At a later stage we experimented with deformable membranes as well, like those that surround nerve cells. So a more specific description of the problem would be the following: the design of a DEVS simulator and a set of DEVS models that employ a novel impulse-based method to simulate the dynamics of deformable structures including vesicle clusters and biological membranes.

The biological models we are interested in are spatial models, as defined in Section 2.1. In this regard they differ from the non-spatial models of the Gillespie Algorithm described in [10], as well as the factory-influenced model of [11] in which enzymes were treated as chemical-processing machines. The type of spatial models that interest us are continuous-space models, as opposed to cellular models like those of the Next Subvolume Method [12]. We avoid the Next Subvolume Method also because it is population-based, and therefore unable to track the positions of each vesicle in a cluster. GridCell [14, 15], MCell [16], ChemCell [17, 18], Smoldyn [19], and GFRD [20] all use particle-based methods, and the latter four of the five employ continuous-space models as well. These simulators and methods might well be able to treat vesicles and synapsins as particles, and the binding of vesicles and synapsins as a type of reaction, but we also wish to simulate the deformation of vesicle clusters as they self-assemble from such reactions. We seek continuous-space biological models that combine interacting particles with deformable structure dynamics.

Although DEVS has been used for cellular models, as pointed out in Section 2.2, the application of the formalism to continuous-space models is rare. Our approach differs from that of the continuous-space forest fire models of [25] and [28] in that these projects used DEVS variants with dynamic structure. In pursuit of simplicity, we seek a relatively minimalistic formulation of DEVS that adheres closely to the original formalism.

Prior to this thesis, it is unlikely that DEVS had been applied to any continuous-space models of biological systems. The biological DEVS models mentioned in Section 2.3 were either non-spatial, as in [29, 32], or cellular, as in [30, 31]. By avoiding variants of DEVS like Cell-DEVS, ρ-DEVS, and ml-DEVS, we must address the concern raised in [32] that the original DEVS formalism is unsuitable for interactions of three or more biological entities. Our decision not to associate a DEVS model with each entity alleviates this problem.

One rationale for using DEVS is to support the integration of biological simulation algorithms, the topic of Section 2.4. Unlike [35], we do not actually integrate the population-based Next Subvolume Method with a particle-based method like our own. Nevertheless, our intent was to design a DEVS simulator that would accommodate this combination of algorithms and others. We avoid the multi-formalism modeling approach of James II [36], adopting a single formalism as done in E-Cell [40]. Unlike E-Cell, the single formalism we adopt is DEVS.

Perhaps the simplest way to simulate the dynamics of isolated vesicles is to treat them as rigid bodies. Collisions between vesicles could then be resolved using any of the rigid body simulation methods of Section 2.5. But because vesicles also form clusters that change in shape, the deformable structure simulations of Section 2.6 have to be considered as well. We avoided some methods, such as the constraint-based methods of [42] and the FEM [47], because we feared that the complexity of the mathematics would be a distraction from our focus on the application of DEVS. One very promising option would have been to combine penalty methods for rigid bodies, described in [43], with mass-spring-damper systems for deformable structures. We avoided this approach because of the need to choose appropriate time steps, address the stiffness problem, and incorporate random motion into these traditionally deterministic methods. In the end we decided on impulse-based methods, avoiding fixed time steps and allowing random motion to be modeled with impulses of randomized timing, magnitude, and direction. The drawback to impulse-based methods was that, while they had been used for rigid bodies [41], we would have to invent our own way to apply them to deformable structures. The constraint-based/impulse-based method of [50] and [51] would not have been ideal, as impulses affecting deformation were applied at regular time intervals. Also, the simulation of self-assembling deformable structures had yet to be demonstrated with this method.

Looking at the simulations of presynaptic nerve terminals described in Section 2.7, there had yet to be any continuous-space models capturing the dynamics of vesicle clusters as deformable structures. Recall that the continuous-space MCell model of [55] did not allow vesicles to move and form clusters. The cellular model of [56] did allow vesicle motion and cluster formation, but was not a continuous-space model and did not allow vesicle clusters to deform.

The "DEVS-based dynamic simulation of deformable biological structures", as we have described it, is a compelling problem in part because its solution would bring about a number of potentially useful inventions: a new DEVS simulator, an associated set of conventions for integrating biological simulation algorithms, a new method for simulating deformable structures, and a new continuous-space model of a presynaptic nerve terminal. More importantly, a successful solution would challenge two perceptions: that impulse-based methods are unsuitable for the dynamic simulation of deformable structures, and that the original DEVS formalism is unsuitable for biological models that involve many interacting entities. Both impulse-based methods, and the use of a single and relatively simple modeling formalism, are worth investigating because they have the potential to reduce the complexity of biological simulations in general. Even if the project were to reveal deficiencies with impulse-based methods or the chosen formalism, the knowledge gained may aid in the pursuit of more practical simulation methods and more effective variants of DEVS. Future researchers will be able to refer to this thesis for concrete examples of impulse-based deformable structure simulations and the application of DEVS to continuous-space biological models.

4 Model and Simulator Design

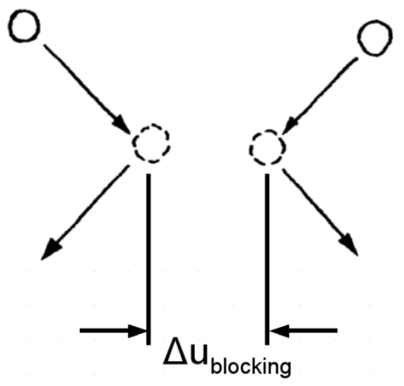
4.1 Tethered Particle System

Here we provide an informal overview of the tethered particle system (TPS). Formulas related to the TPS are defined in Appendix B. Note that on its own, the TPS has nothing to do with DEVS; we therefore ignore the formalism in this section.

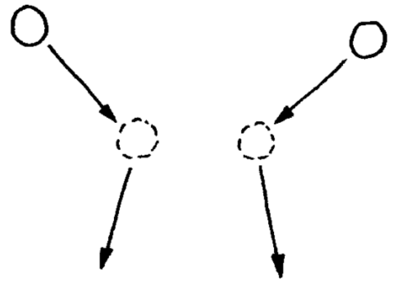
A "TPS model" tracks the positions and velocities of numerous particles, each with a fixed mass, that interact with one another via collisions. "The TPS" is a method in which the dynamics of deformable structures are simulated using a TPS model. The TPS is unusual in that it is an impulse-based method, meaning that any change in a particle's velocity is instantaneous, yet the method is designed for representing structures that deform over a length of time. The key idea is that a deformable structure may be represented by a group of particles; even though each individual particle changes velocity in a sequence of instantaneous impulses, the configuration of the particles in the group changes in a seemingly gradual process over time.

In order for a group of particles to exhibit any structure at all, the distances between certain pairs of particles in the group must be regulated or restricted. In a TPS model, the distance between a pair of particles is constrained by two types of collisions: "blocking collisions" and "tethering collisions".

What we refer to as a "blocking collision" is what one normally associates with the word "collision". As illustrated in Figure 7a, a blocking collision occurs when two approaching particles reach an inner limiting distance. This "blocking distance" is represented by $\Delta u_{blocking}$.



(a)



(b)

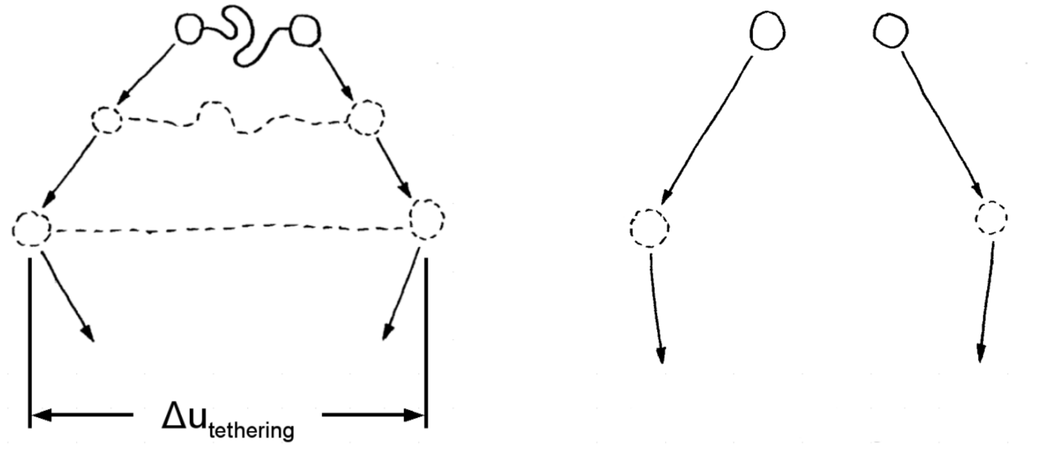
Figure 7: Illustrations of blocking collisions, in which pairs of approaching particles reach the blocking distance $\Delta u_{blocking}$ and rebound. In (a) the collision is elastic, whereas (b) depicts an inelastic collision.

Note that although we will frequently depict a particle as a circle or sphere of some radius, neither the shape nor the size of a particle is explicitly defined in a TPS model. The particles in Figure 7 could have been drawn as larger circles, for instance, perhaps overlapping with one another at the time of collision.

The particles in Figure 7a are shown rebounding at a similar angle to that at which they had been approaching. This indicates that the collision is elastic, meaning that no kinetic energy is lost. When real-world objects collide, they deform and absorb kinetic energy. Although individual particles in a TPS have no shape and do not explicitly deform, we may wish account for energy loss in particle collisions. The loss of kinetic energy in the inelastic collision of Figure 7b causes the particles to rebound at a smaller angle.

Energy loss due to collisions is generally modeled with a parameter called the "coefficient of restitution", which expresses the ratio of the post-collision relative speed of two particles to the pre-collision relative speed [42]. In a TPS model, different coefficients are used for different types of collisions. In a blocking collision, the coefficient of restitution is referred to as the "rebounding coefficient", and is represented by $c_{rebound}$. We will assume $0 \leq c_{rebound} \leq 1$, with $c_{rebound} = 0$ indicating maximum energy loss, and $c_{rebound} = 1$ indicating a perfectly elastic collision.

Any two specific particles may be tethered together at the start of a simulation. Also, when a blocking collision occurs between two particles, they may become tethered. If two particles are tethered and moving away from one another, and if they reach the "tethering distance" $\Delta u_{tethering}$, then one of two things may happen. The particles may become untethered, in which case they continue moving away from one another. Otherwise the particles remain tethered, undergo a tethering collision, and retract inwards. The phrase "tethering collision" is unintuitive because one normally expects a collision to occur only between approaching objects. We use the word "collision" for separating particles as well so that we can apply the phrases "collision detection" and "collision response" to either type of particle-particle interaction. Tethering collisions are illustrated in Figure 8.



(a) **(b)**

Figure 8: Illustrations of tethering collisions, in which pairs of separating tethered particles reach the tethering distance $\Delta u_{tethering}$ and retract. In (a) the tethering collision is elastic, whereas (b) depicts an inelastic collision.

A tethering collision may be envisioned as a situation in which a cord has completely unravelled, and therefore delivers an inward impulse to the particles attached to it on either end. Such a cord is illustrated in Figure 8a. Unlike a spring in a mass-spring-damper model, a cord in a TPS model has no effect on either particle before reaching its maximum length.

Figure 8a is meant to portray an elastic tethering collision, whereas Figure 8b illustrates a tethering collision in which energy is lost. To address energy loss in tethering collisions, we introduce another type of coefficient of restitution. We refer to it as the “retraction coefficient”, and represent it with $c_{retract}$ satisfying $0 \leq c_{retract} \leq 1$.

Suppose we have two particles, A and B. Their masses are m_A and m_B respectively, their current positions are $\vec{u}_A{}'$ and $\vec{u}_B{}'$ (as explained in Appendix B, $\vec{u}_A$ and $\vec{u}_B$ are past positions), and their velocities are $\vec{v}_A$ and $\vec{v}_B$. The distance Δu between the particles can be expressed as a function of the time Δt.

$$\Delta u = \sqrt{\sum \left(\left((\vec{u}_B{}' + \vec{v}_B \cdot \Delta t) - (\vec{u}_A{}' + \vec{v}_A \cdot \Delta t) \right)^2 \right)} \tag{3}$$

The basic procedure in a TPS simulation is to repeatedly solve (3) for Δt for all pairs of relatively close particles. Solving (3) with $\Delta u = \Delta u_{blocking}$ yields the time remaining before a blocking collision, whereas $\Delta u = \Delta u_{tethering}$ gives the time of a tethering collision. Time is advanced by the smallest calculated value of Δt, the time remaining before the next collision. When that collision occurs, the new velocities of the two particles involved are calculated from (4), and the process repeats.

$$\begin{aligned} \vec{v}_A{}' &\equiv \vec{v}_A + \frac{\Delta \vec{p}}{m_A} \\ \vec{v}_B{}' &\equiv \vec{v}_B - \frac{\Delta \vec{p}}{m_B} \end{aligned} \tag{4}$$

The vector $\Delta \vec{p}$ above is the impulse, the change in momentum of particle A as a result of the collision. If the particles rebound or retract, then $\Delta \vec{p}$ can be calculated from (5) below with $c_{restitute}$ being either $c_{rebound}$ or $c_{retract}$. The vector $\vec{v}_{\hat{u}}$ is the relative velocity of the particles projected onto the axis between them.

$$\Delta \vec{p} \equiv \left(\frac{1}{m_A} + \frac{1}{m_B} \right)^{-1} \cdot (1 + c_{restitute}) \cdot \vec{v}_{\hat{u}} \tag{5}$$

The actual computations performed are complicated by the possibility of revolution, simultaneous and nearly-simultaneous collisions, and random impulses. These concepts are described informally below. The TPS remains simpler than most deformable structure simulation methods in that all unknown variables can be calculated analytically from the explicit formulas given in Appendix B. There are no systems of equations or inequalities that need to be solved simultaneously or iteratively.

Note that it is the tethering collisions that distinguish the TPS from more traditional particle collision algorithms. They place potentially-useful outer limits on the distances between certain pairs of particles, but introduce a performance problem that must be addressed. The problem is illustrated in Figure 9. Particle A remains stationary because it has an infinite mass, whereas particle B is in motion with a finite mass. The two particles remain tethered, and undergo a sequence of elastic tethering collisions. Immediately after each collision, particle B approaches A at a relatively small angle θ.

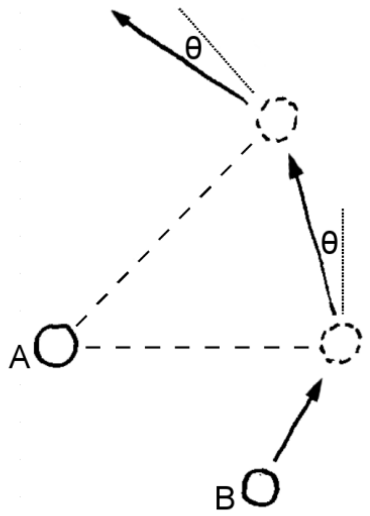


Figure 9: A scenario in which particle B revolves around A in a sequence of tethering collisions.

Note that any stage in a TPS simulation, time is advanced by an irregular interval to that of the next collision. The greater the frequency with which collisions occur, the slower the simulation progresses. The problem with the above scenario is that, if θ is small, the tethering collisions become extremely frequent and the simulation may become impractically slow. Worse, if $\theta = 0$, then time cannot be advanced at all without violating the constraint of $\Delta u_{tethering}$ on the distance between A and B. The problem still exists if the mass of A is finite, and is exacerbated by small values of $c_{retract}$.

To address the problem, we place a lower limit $\theta_{revolve}$ on the angle at which particles can approach after a tethering collision. In a collision where this restriction takes effect, we say that the particles "revolve" instead of "retract". We also introduce a "revolution coefficient"

$c_{revolve}$ that expresses the ratio of the new relative velocity to the old one after one complete revolution of the particles, allowing energy to be lost. We require $0 \leq c_{revolve} \leq 1$.

To summarize, between every pair of particles we have a blocking distance $\Delta u_{blocking}$ and a tethering distance $\Delta u_{tethering}$ that may constrain the distance between them. We also have three coefficients that affect the energy lost in a collision: the rebounding coefficient $c_{rebound}$, which pertains to blocking collisions; the retraction coefficient $c_{retract}$, which pertains to most tethering collisions; and the revolution coefficient $c_{revolve}$, which pertains to tethering collisions in which the resulting approach angle must be increased to $\theta_{revolve}$. In a TPS model, it is useful to define several distinct species of particles. The parameters $\Delta u_{blocking}$, $\Delta u_{tethering}$, $c_{rebound}$, $c_{retract}$, and $c_{revolve}$ are then chosen for each pair of species.

It is widely known that simultaneous and nearly-simultaneous collisions threaten the efficiency of impulse-based methods, potentially slowing simulations to a halt. We now describe this problem followed by our solution.

Consider the scenario in Figure 10. Assume particles A, B, and C are all of the same mass, and that collisions are elastic. At time t_{AB}, moving particle A collides with stationary particle B, transferring all of its momentum. Then at time t_{BC}, particle B collides with stationary C and transfers all of its momentum.

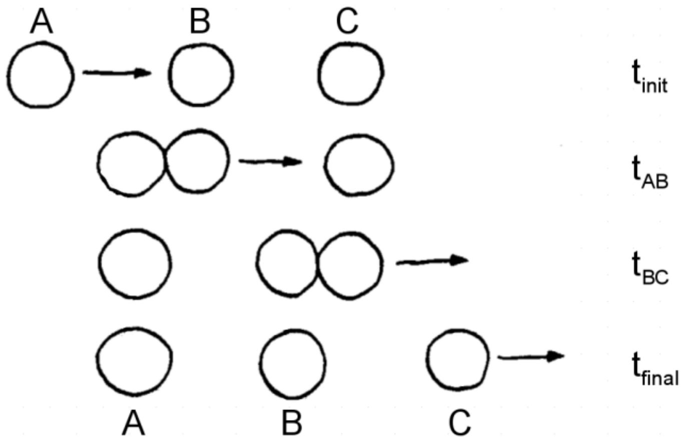


Figure 10: A scenario in which 2 collisions occur between three particles of equal mass.

An efficiency problem arises if the mass of particle B is reduced to a fraction of that of A and C, as illustrated in Figure 11. At time t_{AB_0}, particle A transfers only some of its momentum to B. Particle B reaches C and rebounds at t_{BC_0}, then meets A again at time t_{AB_1}. Because only a small amount of momentum is transferred in each collision, particle B must rebound back and forth in a sequence of nearly-simultaneous collisions. If the mass of B is one thousandth that of A and C, roughly 70 elastic collisions occur before enough momentum has been transferred to separate all three particles.

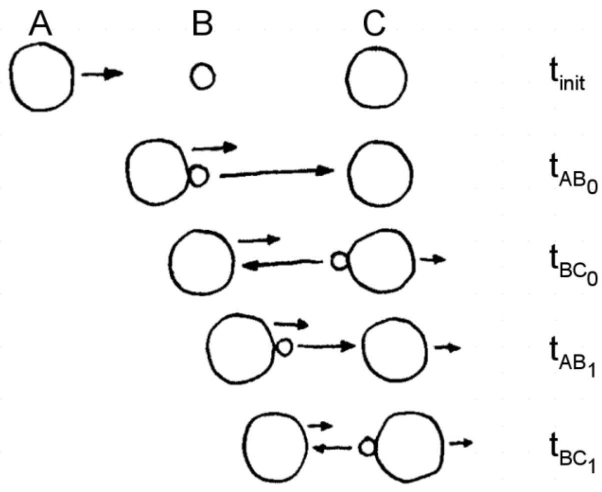


Figure 11: The same scenario as in Figure 10, but the center particle is now less massive. Numerous nearly-simultaneous collisions result.

The processing of 70 collisions is in itself a significant computational cost for such a simple scenario, but there are many situations in which the simulation will halt completely. When a simulation was performed with the Figure 11 scenario and a rebounding coefficient of 0.9, the momentum transferred on each collision eventually rounded to zero and the simulation stalled.

We propose a novel approximation that addresses the threat of simultaneous and nearly-simultaneous collisions. The idea is to separate each collision into a loading phase and a restitution phase, and to allow restitution to take place at a later time. When particles collide (the loading phase), they form loaded groups. A loaded group acts as a single body with the combined mass of all the particles in the group. A restitution delay time $\Delta t_{restitute}$

is introduced, after which the loaded particles separate (the restitution phase). Loaded particles may remain together longer than $\Delta t_{restitute}$ if necessary to ensure that the order in which particles separate is opposite that in which they loaded. Figure 12 illustrates the approximation.

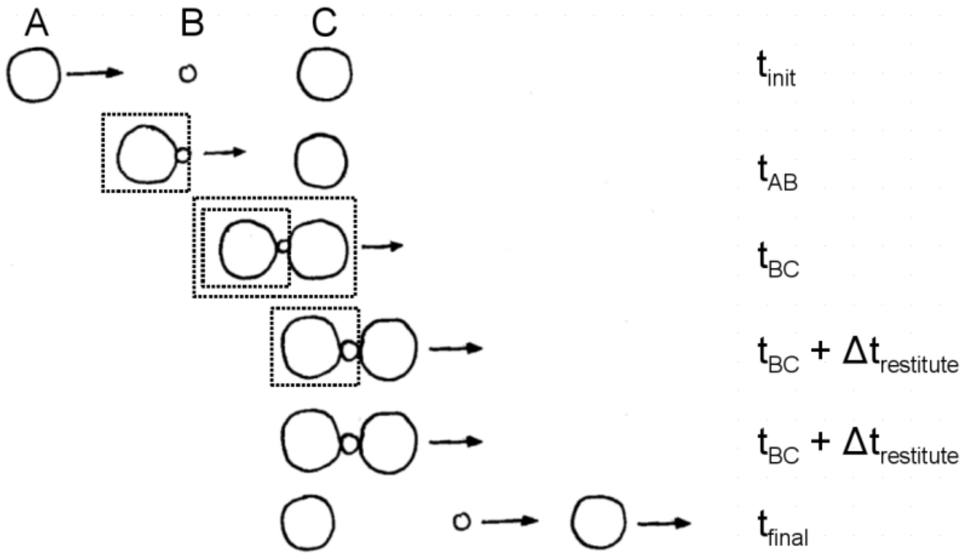

A
B
C
t_{init}
t_{AB}
t_{BC}
$t_{BC} + \Delta t_{restitute}$
$t_{BC} + \Delta t_{restitute}$
t_{final}

Figure 12: A scenario demonstrating an approximation that addresses the problem of simultaneous and nearly-simultaneous collisions. Particles form loaded groups for durations of $\Delta t_{restitute}$, during which time they act as single bodies.

At time t_{AB} in Figure 12, particles A and B collide, form a loaded group, and proceed with matching velocities. Suppose that this loaded group did not encounter any other particle. In that case, at time $t_{AB} + \Delta t_{restitute}$, particles A and B would separate or "restitute". But that does not happen, as at time t_{BC} while A and B are still loaded, they encounter particle C. The impulse delivered to C depends not on the mass of B, but rather on the mass of A and B added together. It is the temporary accumulation of mass that tends to increase the momentum transferred per collision, and thus reduces the number of collisions.

It is necessary that particles in a loaded group restitute in the opposite order from that in which they loaded. After all three particles form a loaded group at t_{BC}, particles A and B may no longer separate at time $t_{AB} + \Delta t_{restitute}$. The loaded group remains intact until $t_{BC} + \Delta t_{restitute}$ instead, at which point particle B separates from C. The result of the

B-C restitution is calculated with the masses of A and B still combined. After the B-C restitution is complete, but also at the simulated time $t_{BC} + \Delta t_{restitute}$, particles A and B finally separate.

The proposed approximation can dramatically reduce the number of collisions in a simulation, even if $\Delta t_{restitute}$ is very small. If $\Delta t_{restitute} = 0$, loading and restitution occur back-to-back and the approximation is effectively canceled.

Summarizing the TPS again, pairs of particles rebound as a result of blocking collisions, and either retract or revolve as a result of tethering collisions. Each collision is divided into a loading phase, in which particles form loaded groups, and a restitution phase occurring at least $\Delta t_{restitute}$ later, in which loaded groups separate.

We also include "random impulses", momentum changes of randomized magnitude and randomized direction applied to particles at randomized times. From a practical perspective, they are included to prevent the kinetic energy in a TPS model from converging to zero due to the energy losses of particle collisions. From a physical perspective, random impulses may represent Brownian motion, variability in electric potential fields or fluid pressure, or interactions with otherwise unrepresented objects.

4.2 Proposed DEVS Formulation

In this section we turn our attention away from the TPS, and focus on the DEVS formalism. While the formulation of DEVS presented here was designed with biological systems in mind, our conventions could be applied to simulations of artificial or environmental systems as well. Note that a complete, formal definition of our DEVS simulator can be found in Appendix C, along with several functions that aid in the design of DEVS models.

At the outset of the project, we desired a minimalistic formulation of DEVS that would adhere fairly closely to the original version of the formalism. Although we propose a new set of conventions, we do not consider these conventions to constitute a new DEVS variant. If a hierarchical DEVS model is designed using our formulation, the same hierarchy and messages could be defined using the original formalism. Variants, by contrast, generally aim to allow different types of hierarchies and message-passing patterns.

In the original DEVS formalism, models are classified as either atomic or coupled. As stated in Section 2.2, the closure under coupling property assures us that for every DEVS coupled model, there is an equivalent DEVS atomic model. Our perspective is a bit different. Exploiting closure under coupling, we require that the parameters used to define a coupled model be converted into the set of parameters that defines an atomic model. All of our DEVS models can therefore be considered atomic. When designing a simulator based on our conventions, the issue of coupling can safely be ignored.

Recall that in the original formalism, atomic models are generally defined as tuples of the form $\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$. In our formulation, all DEVS models are vectors of the form $[\delta_{ext}, \delta_{int}, ta]$. We omit the sets X, Y, and S for the sake of brevity, but note that modelers who adopt our conventions may decide to include these sets in the vector. The external transition function δ_{ext} and the time advance function ta are unchanged, but we absorb the output function λ into the internal transition function δ_{int}. Whereas the original δ_{int} results in the new state s', ours results in the vector $[s', Y]$, where Y is a vector of output values.

Combining λ and δ_{int} changes the formalism little, as the two functions are always invoked back-to-back. For any δ_{int} function defined with our conventions, an equivalent pair of original-DEVS $[\lambda, \delta_{int}]$ functions can be defined by invoking our δ_{int} twice. The advantage in combining λ and δ_{int} is that, in some situations, the repetition of computationally-intensive calculations can be avoided.

A DEVS simulator has three main inputs: a DEVS model of the form $[\delta_{ext}, \delta_{int}, ta]$, the initial state of that model, and a set of input times and values. The main output is the set of output times and values. The remainder of this section will focus on our approach for organizing definitions of models and initial states, two of the main simulator inputs.

We define DEVS models and initial states 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. Associated with the bottom layer of any model are the parameters $[\delta_{ext}, \delta_{int}, ta]$. Each additional layer requires a function that, by convention, has the subscript $_{DEVS}$ in its name. Each of these "DEVS model functions" takes a new set of parameters as its arguments, which are called "model parameters". The output of a DEVS model function is a vector of the form $[init_{model}, model]$, where $init_{model}$ is an "initialization function" and $model$ is a DEVS model of the form $[\delta_{ext}, \delta_{int}, ta]$. The initialization function takes "initialization parameters" as its arguments, and results in an initial state. If the initial state is a vector, then its components are called "state variables".

As an example, suppose we wished to define some hypothetical DEVS model named foo with model parameters a and b. We would then define a DEVS model function named foo_{DEVS} that performs the following tasks:

foo_{DEVS} :

1. Takes as its arguments the model parameters $[a, b]$
2. Defines the initialization function $init_{foo}$ using $[a, b]$
3. Defines the external transition function δ_{ext} using $[a, b]$
4. Defines the internal transition function δ_{int} using $[a, b]$
5. Defines the time advance function ta using $[a, b]$
6. Defines the DEVS model foo as $[\delta_{ext}, \delta_{int}, ta]$
7. Results in $[init_{foo}, foo]$

Note that we are treating functions as values, and we assume that functions can be defined within other functions. Aside from defining an initialization function, the main task of foo_{DEVS} is to transform its model parameters $[a, b]$ into the simulation-ready model $[\delta_{ext}, \delta_{int}, ta]$. We visualize foo as a two-layer DEVS model, with $[a, b]$ associated with the upper layer and $[\delta_{ext}, \delta_{int}, ta]$ with the lower.

Consider now the hypothetical three-layer DEVS model bar, defined below by the DEVS model function bar_{DEVS}. Associated with the third and uppermost layer of bar are the parameters c and d.

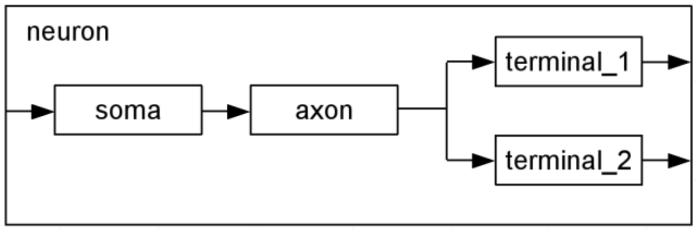
bar_{DEVS} :

1. Takes as its arguments the model parameters $[c, d]$
2. Defines the variable a using $[c, d]$
3. Defines the variable b using $[c, d]$
4. Invokes foo_{DEVS} using $[a, b]$ to obtain $[init_{foo}, bar]$
5. Defines the function $init_{bar}$, which invokes $init_{foo}$, but also uses $[c, d]$
6. Results in $[init_{bar}, bar]$

The main task of bar_{DEVS} is the transformation of $[c, d]$ into $[a, b]$. Although the result of bar_{DEVS} includes a model of the form $[\delta_{ext}, \delta_{int}, ta]$, the invocation of foo_{DEVS} alleviates the need to explicitly reference the transition functions or the time advance function.

One particular DEVS model function will be used many times over. Similar to foo_{DEVS}, the function $coupled_{DEVS}$ results in an initialization function and a DEVS model of the form $[\delta_{ext}, \delta_{int}, ta]$. Instead of $[a, b]$, $coupled_{DEVS}$ takes $[M, C, pr]$ as its vector argument. This is the parameterization we will use for all coupled models. It takes the place of the tuple $\langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select\rangle$ of the original DEVS formalism.

As an example, suppose we wish to define a DEVS model with the structure of Section 2's Figure 1. The image is shown for a second time below.

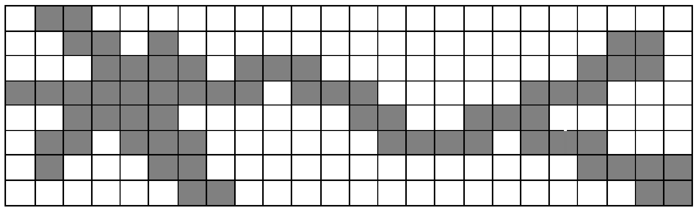


The parameter M is a function that maps submodel IDs, like "$soma$" or "$axon$", to DEVS models of the form $[\delta_{ext}, \delta_{int}, ta]$. A simplified description of the function C is that it maps source submodel IDs to destination submodel IDs. Invoking C with "$soma$" yields "$axon$", in part, reflecting the second arrow in the diagram. Invoking C with $\varnothing$ would yield

$“soma”$, reflecting the leftmost arrow. Invoking C with one of the terminal IDs would yield $\varnothing$, reflecting the two arrows on the far right.

Suppose that, according to their time advance functions, the $soma$ and $axon$ submodels are scheduled to undergo internal transitions at the same time. In that case, the priority function pr is evaluated. The result of $pr\left([“soma”, “axon”]\right)$ could be $“soma”$ or $“axon”$, the ID of the submodel that is to go first, or $\varnothing$, indicating that the order of the internal transitions is to be randomized.

Now we turn our attention to the layered design of cellular models like the one in Figure 2, which is shown below for the second time.



One way to simplify the design of such a model is to define the spatial relationships between lattice cells in a separate layer that has nothing to do with biology. In the two-dimensional model illustrated above, suppose we imposed the constraint that each cell can interact with only the four adjacent cells: the one to the left, the one to the right, the one above, and the one below. We would then describe this spatial configuration as a “rectangular lattice”. In 3D we would call this type of 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 cell on the inside of a “hypercubic lattice” can interact with its $2{\cdot}n_{dim}$ adjacent neighbors. Cells on an edge of a hypercubic lattice have, of course, fewer than $2{\cdot}n_{dim}$ neighbors.

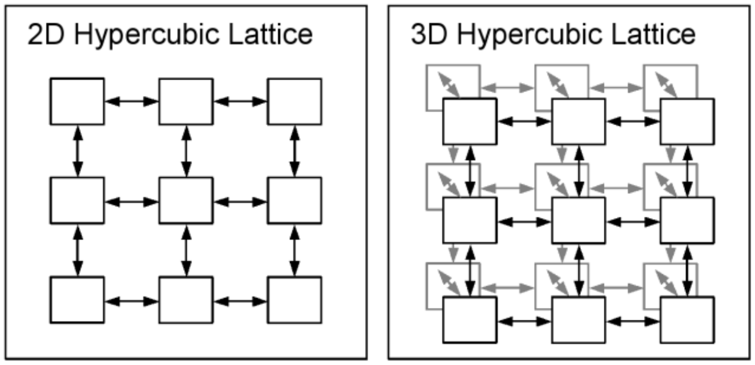


Figure 13: Two hypercubic lattices: one 3-by-3 and one 2-by-3-by-3.

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 13 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 submodels.

Hypercubic lattice models are defined by the DEVS model function HL_{DEVS}, which is similar in form to the hypothetical three-layer function bar_{DEVS} outlined above. Instead of mapping $[c, d]$ to $[a, b]$, the main task of HL_{DEVS} is to map $[N, HLm_{DEVS}, pr]$ to $[M, C, pr]$. After this conversion, $coupled_{DEVS}$ is invoked to obtain $[\delta_{ext}, \delta_{int}, ta]$.

In summary, the main idea behind our approach is that DEVS models can be parameterized in any way we wish. The coupled model parameterization $[M, C, pr]$ is only one of an infinite number of arbitrary ways to define a DEVS model. Whatever parameters are used, however, a function with the subscript $_{DEVS}$ must be defined to convert those parameters into the form $[\delta_{ext}, \delta_{int}, ta]$. These DEVS model functions can invoke other DEVS model functions, resulting in a layered model definition. Layers, as we describe them here, separate sets of model parameters. They are not to be mistaken with levels in a model hierarchy, which separate coupled models from the submodels they contain.

4.3 DEVS Tethered Particle System Model

This section presents our DEVS TPS model, an application of the functions and conventions of Section 4.2 to yield a complete formal description of Section 4.1's tethered particle system method. The DEVS TPS model serves as a means of simulating deformable biological structures, and as an example of how one can design DEVS models of biological systems. While the formal description itself can be found in Appendix D, here we describe informally the model's hierarchical structure, as well as the role and layered definition of each submodel.

Our approach to biological model design can be summarized as follows: different algorithms or aspects of algorithms are separated at upper levels in a DEVS model hierarchy, and space is partitioned at lower levels. In general, we do not define separate DEVS models for separate biological entities. The partitioning of entity-specific data can instead be handled by a DEVS model's parameters and state variables.

For example, in a TPS model of a presynaptic nerve terminal, separate biological entities include vesicles and synapsins. A vesicle can be represented as a single particle, and a synapsin can be represented as a pair of tethered particles. We refrain from defining a "DEVS vesicle model" or a "DEVS synapsin model" or even a "DEVS particle model". We do wish to keep information about one particle separate from that of another, but we accomplish this with the state variable Ψ. As described in Appendix B, $\Psi\left(id_A\right)$ gives the past position and current velocity of the particle identified by id_A. We also wish to organize information about each particle species in some appropriate structure, but this we achieve with the model parameters Ω_ψ and $\Omega_{\psi\psi}$. Also explained in Appendix B, the mass of a species identified by spc_A is $\Omega_\psi\left(spc_A\right)\left(\text{“}m\text{”}\right)$, and the tethering distance between this species and one identified by spc_B is $\Omega_{\psi\psi}\left(\left[spc_A, spc_B\right]\right)\left(\text{“}\Delta u_{tethering}\text{”}\right)$. It is firstly the separation of collision detection, collision response, and random impulse generation, and secondly the division of space into subvolumes, for which we exploit the hierarchical nature of DEVS.

Figure 14 illustrates the structure of a DEVS coupled model called TPS, which can be considered the uppermost level of our hierarchy. The TPS model has three submodels: RI, which generates random impulses; $detector$, which detects collisions between particles; and $responder$, which calculates new trajectories of particles upon receiving notification of impulses or collisions.

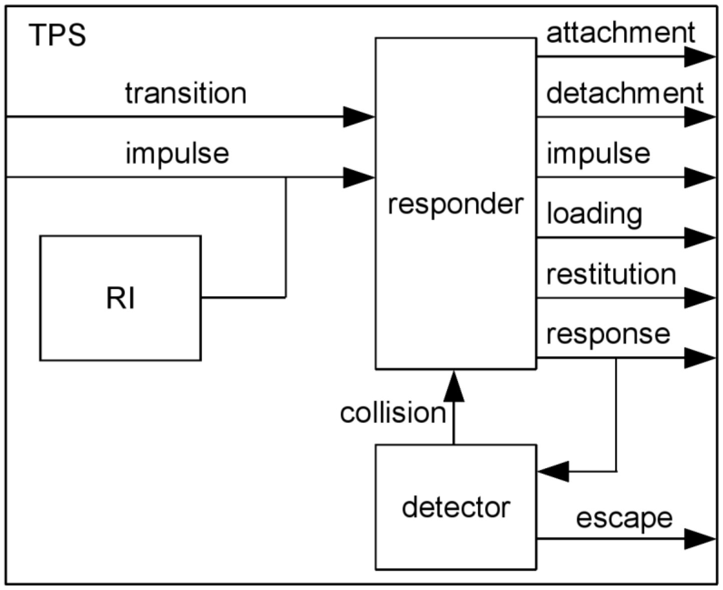


Figure 14: The structure of the TPS coupled DEVS model.

The key interaction of the TPS model is represented by the loop at the bottom-right of Figure 14. At the exact simulated time when a collision occurs between two particles, the $detector$ sends a $collision$ message to inform the $responder$. The $responder$ immediately outputs a $loading$ message to indicate that the two particles have become "loaded" (see Section 4.1), and then prepares a separate $response$ message for each particle affected by the collision. Each $response$ message describes the new trajectory of a particle, information that is sent back to the $detector$ so that future collisions times can be re-calculated. At some time after a collision in which two particles became loaded, those particles restitute. The responder then outputs a $restitution$ message along with a separate $response$ message for each affected particle. The new trajectories are again sent to the $detector$.

Other interactions include impulses that alter the trajectory of a particle. They originate from either the RI submodel or an input of the TPS. These messages enter the $responder$

and result in an $impulse$ output message and one or more $response$ messages. Particle collisions can cause two separate particles to become "tethered", in which case the $responder$ outputs an $attachment$ message, or they can cause two tethered particles to separate, in which case the $responder$ outputs a $detachment$ message. Input $transition$ messages can affect the nature in which particles attach or detach. Finally, the $detector$ may output an $escape$ message if a particle strays too far from the center of the region represented by the model.

The RI and $responder$ submodels can be considered DEVS atomic models, as the functions $[\delta_{ext}, \delta_{int}, ta]$ are defined directly. The $detector$, on the other hand, is defined as a DEVS coupled model with two submodels. The $lattice$ submodel partitions space into different regions called "subvolumes". Collisions are detected within each subvolume, as this is more efficient than detecting collisions in the entire space. If there are 100 particles, for example, then there are 5000 distinct pairs of particles that must be considered. But if those particles are partitioned into 5 subvolumes, then each subvolume has roughly 20 particles and 200 pairs. So with 5 subvolumes instead of 1, there are roughly 1000 pairs to check instead of 5000. The advantage is actually less than that, as several pairs of particles will be checked in multiple subvolumes.

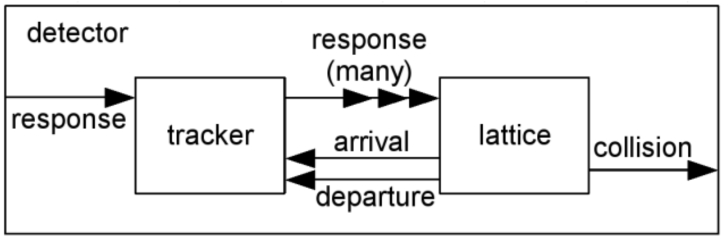


Figure 15: The structure of the $detector$ coupled DEVS model of the TPS.

When we say that a certain subvolume "is aware" of a particle, we mean that the detection of the next collision in that subvolume may depend on the position and velocity of that particle. In an informal sense, this generally means that the particle is situated either inside or near that subvolume. When a $response$ message is received by the $detector$, it must be redirected to each subvolume that is aware of the particle. This duplication of $response$ messages is performed by the $tracker$ submodel. The $tracker$ tracks, for each particle, the subvolumes that are aware of it. An $arrival$ message informs the $tracker$ that a particular subvolume has become aware of certain approaching particle, whereas a $departure$ message indicates that a certain subvolume is no longer aware of a certain departing particle.

The $lattice$ is a DEVS coupled model; or more specifically, a DEVS hypercubic lattice model. As shown in Figure 16, each subvolume of the $lattice$ has its own associated DEVS $subV$ model, and adjacent $subV$ models may pass adj messages to one another. Each individual $subV$ model may also receive $response$ messages, which initiate a recalculation of future collisions. When a collision occurs, a $collision$ message is output. When a particle gets sufficiently close to the subvolume associated with the DEVS $subV$ model, an $arrival$ message is output. When a particle gets too far from the subvolume, a $departure$ message is output.

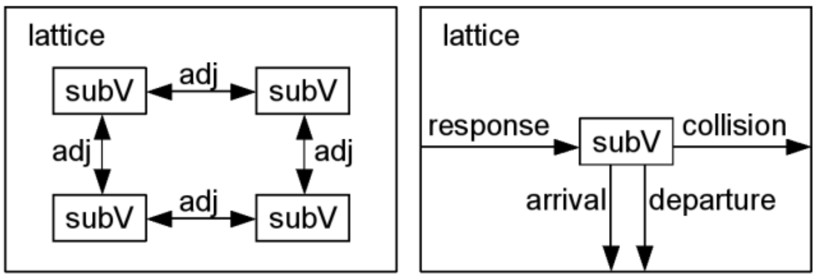


Figure 16: The structure of the $lattice$ coupled DEVS model of the $detector$. Although only a 2-by-2 configuration is shown on the left, one can have any number of subvolume models in 1, 2, or 3 dimensions.

Now that the hierarchical structure of the DEVS TPS model has been described, we present a scenario that illustrates how the various submodels interact. For this scenario, we will consider a system with only two particles. The two particles are to undergo only blocking collisions, not tethering collisions. The tethering distance can be assumed to be infinity ($\Delta u_{tethering} = \top$). Also, we will ignore for now the distinction between the loading and restitution phases of collisions.

The scenario begins with the situation depicted in Figure 17. Here we have a large particle A, a small particle B, and a 2-by-3 rectangular lattice ($N \equiv [2, 3]$). Each subvolume model, or $subV$, is identified by its coordinates. The subvolumes themselves are square regions. 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. Each subvolume is surrounded by two concentric circles that help determine when the associated model gains or loses awareness of an approaching or leaving particle. If a DEVS model is aware of a particle, the current state of that model includes the position and velocity of that particle.

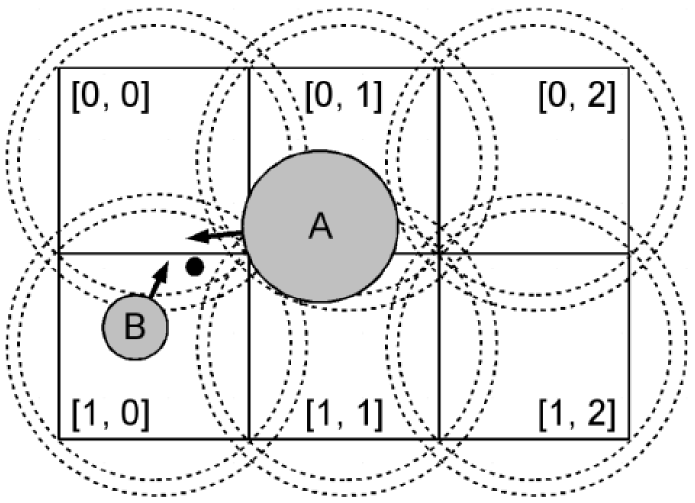


Figure 17: A scenario in which two particles approach one another.

Looking at Figure 17, 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 17, 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 17, 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 18. It is at this time that, having scheduled the collision, $subV$ $[1, 0]$ undergoes an internal transition. It outputs 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.

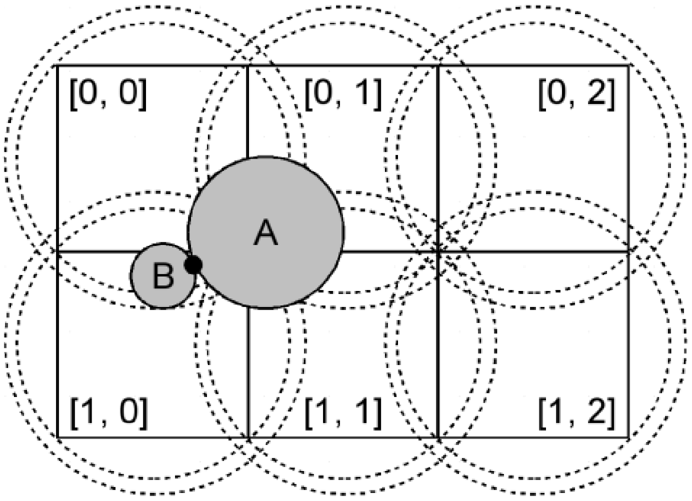


Figure 18: The same scenario as in Figure 17, but at the later time when the two particles collide.

If left alone, then despite having detected a collision, $subV$ $[1, 0]$ will 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 zero, 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.

That concludes the scenario. 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 moving circle enters or exits a square region, and worse to determine when a moving sphere enters or exits a cubic region. It turns out that with concentric circles or spheres around each subvolume, the formulas we define in Appendix B to detect blocking and tethering collisions can be reused to determine when subvolume models gain or lose awareness of particles.

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 the coupled models can be specified easily by defining the links illustrated in Figures 14, 15, and 16, the bulk of our efforts must go into the design of the external and internal transitions of the RI, $responder$, $tracker$, and $subV$ atomic models. These 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.

Having designed a hierarchical structure for our DEVS TPS model, we find ourselves with six distinct types of DEVS models: the TPS, the $detector$, the $lattice$, the $responder$, the $tracker$, and the random impulse model RI. Here we regard $subV$ models as part of the $lattice$. Following the conventions we proposed in Section 4.2, every DEVS model takes the form $[\delta_{ext}, \delta_{int}, ta]$, and several layers may to used to arrive at these three functions. The layers of each our six DEVS TPS models are shown below.

TPS	detector	lattice
$\begin{array}{c}\underbrace{\begin{bmatrix} N, a, \vec{u}_{center}, \omega_{\psi}, \omega_{\psi\psi}, \dots \\ attach, detach, \dots \\ \theta_{revolve}, \Delta t_{max}, \Delta t_{restitute}\end{bmatrix}}_{TPS_{DEVS}} \\ \downarrow \\ \underbrace{[M, C, pr]}_{coupled_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$	$\begin{array}{c}\underbrace{\begin{bmatrix} N, a, \vec{u}_{center}, \dots \\ \Omega_{\psi}, \Omega_{\psi\psi}, \Delta t_{max}\end{bmatrix}}_{detector_{DEVS}} \\ \downarrow \\ \underbrace{[M, C, pr]}_{coupled_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$	$\begin{array}{c}\underbrace{\begin{bmatrix} N, a, \vec{u}_{center}, \dots \\ \Omega_{\psi}, \Omega_{\psi\psi}, \Delta t_{max}\end{bmatrix}}_{lattice_{DEVS}} \\ \downarrow \\ \underbrace{[N, subV_{DEVS}, pr]}_{HL_{DEVS}} \\ \downarrow \\ \underbrace{[M, C, pr]}_{coupled_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$
responder	tracker	RI
$\begin{array}{c}\underbrace{\begin{bmatrix} \Omega_{\psi}, \Omega_{\psi\psi}, attach, \dots \\ detach, \theta_{revolve}, \Delta t_{restitute}\end{bmatrix}}_{responder_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$	$\begin{array}{c}\underbrace{[N, a, \vec{u}_{center}, \Omega_{\psi}]}_{tracker_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$	$\begin{array}{c}\underbrace{[n_{dim}, \Omega_{\psi}]}_{RI_{DEVS}} \\ \downarrow \\ [\delta_{ext}, \delta_{int}, ta]\end{array}$

The $detector$ model, for example, is a three-layer DEVS model with the six parameters $[N, a, \vec{u}_{center}, \Omega_\psi, \Omega_{\psi\psi}, \Delta t_{max}]$. The first three parameters give the dimensions of the lattice, the length of each of its subvolumes, and the position of its center. Properties of each particle species are provided by Ω_ψ and $\Omega_{\psi\psi}$, and Δt_{max} is a large duration value used to prevent divide-by-zero errors. The function $detector_{DEVS}$ converts these parameters into $[M, C, pr]$. It then invokes $coupled_{DEVS}$, mentioned in Section 4.2, to convert those parameters into $[\delta_{ext}, \delta_{int}, ta]$. The TPS, $detector$, and $lattice$ are all coupled models. Because it invokes the HL_{DEVS} function (see Section 4.2), the four-layer $lattice$ model is also a DEVS hypercubic lattice model. Note that we name HL_{DEVS}'s second parameter $subV_{DEVS}$, in this case, instead of HLm_{DEVS}. The $responder$, $tracker$, and RI models can be considered two-layer atomic models.

4.4 Simulation Code

Here we describe simulation code developed to demonstrate the TPS method of Section 4.1, the DEVS formulation of Section 4.2, and the DEVS TPS model of Section 4.3. Listings of selected functions are presented. We implemented all code in the Python programming language. Typically interpreted, Python is known more for its aesthetic syntax and convenient data structures than for its performance. The language satisfied our need, however, to develop a proof of concept. Other programmers may refer to the detailed formulas of Appendices B, C, and D to reproduce parts of our program in a faster compiled language.

Throughout our simulation code, we make use of various values and functions from the `numpy` package, a popular extension to Python designed to make the language suitable for numerical applications. Line 1000 below accesses `infty`, a value representing infinity, and `array`, a function that generates multi-dimensional arrays.

```
1000 from numpy import infty, array
```

Our function `position` calculates the current position of the particle identified by the integer `id_A`. Particle information is supplied to this function by the argument `Psi`, a Python dictionary similar to the Ψ parameter of Appendix B. The Python expression `Psi[id_A]` results in the item associated with the particle ID `id_A`, and this item also happens to be a dictionary. The dictionary `Psi[id_A]` maps the keys `"t"`, `"u"`, and `"v"`, respectively, to the time of the particle's last trajectory change, the position of the particle at that time, and the particle's velocity. The past position assigned to `u_A` on line 2002, and the velocity assigned to `v_A` on line 2003, are Python lists. On line 2004, we convert those lists to `numpy` arrays, perform element-wise operations on those arrays, then convert the resulting array back to a list. The resulting list, named `u_A_`, is returned on line 2005.

```
2000  def position(Psi, t, id_A):
2001      t_A = Psi[id_A]["t"]
2002      u_A = Psi[id_A]["u"]
2003      v_A = Psi[id_A]["v"]
2004      u_A_ = (array(u_A) + array(v_A)*(t - t_A)).tolist()
2005      return u_A_
```

We consider the close relationship between our Python code and mathematical formulas to be advantageous, as it allowed us to gain confidence in the formulas as we tested the software. Note the similarity between the coded version of `position` above, and the corresponding mathematical formula of Appendix B shown below.

$$\begin{aligned}
&position\left([\Psi, t, id_A]\right) := \vec{u}_A{}' \\
&\quad t_A := \Psi\left(id_A\right)\left(\text{“}t\text{”}\right) \\
&\quad \vec{u}_A := \Psi\left(id_A\right)\left(\text{“}\vec{u}_A\text{”}\right) \\
&\quad \vec{v}_A := \Psi\left(id_A\right)\left(\text{“}\vec{v}_A\text{”}\right) \\
&\quad \vec{u}_A{}' := \vec{u}_A + \vec{v}_A \cdot (t - t_A)
\end{aligned}$$

Recall from Section 4.2 that a DEVS simulator takes a DEVS model as one of its inputs. We implemented the simulator as the Python function `simulate`, listed on page 51. A detailed description of the corresponding mathematical function $simulate$ can be found in Appendix C. Note the `while` loop starting on line 3006, which repeats once for each processed event. An event is either an external transition, in which case the condition of the `if` statement on line 3025 is `True`, or an internal transition, in which case the `else` clause starting on line 3029 is executed.

```
def simulate(model, IO, Delta_n=infty, start=0, stop=infty, \
             t_suspend=infty, starve=False):
    [delta_ext, delta_int, ta] = model
    n = start
    [n_x, n_y, t, s] = IO.input_s(start)
    status = None
    while status == None:
        [t_ext, x] = IO.input_x(n_x)
        t_int = t + ta(s)
        t_event = min([t_ext, t_int])
        if t_event == infty:
            status = ”completed”
        elif t_event >= t_suspend:
            status = ”suspended”
        elif starve and (t_ext == infty):
            status = ”starved”
        elif n >= stop:
            status = ”stopped”
        if Delta_n == infty:
            checkpoint = False
        else:
            checkpoint = (n%Delta_n == 0)
        if (n > start) and (checkpoint or (status != None)):
            IO.output_s(n, n_x, n_y, t, s)
        if status == None:
            if t_ext <= t_int:
                s = delta_ext(s, t_ext - t, x)
                n_x = n_x + 1
                t = t_ext
            else:
                [s, Y] = delta_int(s)
                for y in Y:
                    IO.output_y(n_y, t_int, y)
                    n_y = n_y + 1
                t = t_int
            n = n + 1
    return [IO, n, status]
```

In the original DEVS formalism, and likely the majority of DEVS variants, the time advance function ta yields infinity to indicate that no internal transition is scheduled. Accordingly, the expression `ta(s)` on line 3008 can yield the value `infty`. If it does, then the internal transition time `t_int` is also `infty`. To be consistent, the external transition time `t_ext` obtained on line 3007 is `infty` if there are no more input values to process. We used the infinite value in a number of other places to avoid extra boolean variables and `if` statements. The parameter `stop` is the maximum number of events, for example. If `stop` is `infty`, its default value, then the number of events is not explicitly limited.

Python is not considered a purely-functional programming language, but it does treat functions as values that can be passed in and out of functions in the same manner as numbers and data structures. Note that the first argument of `simulate` is the DEVS model. Our DEVS models take the form $[\delta_{ext}, \delta_{int}, ta]$, as explained in Section 4.2, and so the functions `delta_ext`, `delta_int`, and `ta` are extracted from `model` on line 3002.

Treating Python functions as values, it was straightforward to implement DEVS models in layers as described in Section 4.2. In many popular object-oriented languages, functions cannot easily be defined within other functions. It may also be difficult to pass functions as arguments or return values. In a object-oriented language, one can implement a DEVS model using a subclass that is required to include implementations of `delta_ext`, `delta_int`, and `ta`. Depending on the language, it may still be possible to adopt our layering technique in the constructors of those subclasses. Incidentally, Python does have object-oriented features, but the only class we defined was the one used to instantiate the `simulate` input/output argument `IO`. Note that `IO` methods are invoked on lines 3004, 3007, 3023, and 3032.

Of the DEVS models used to construct the hierarchical DEVS TPS model of Section 4.3, the random impulse model RI is the simplest. While the DEVS model function RI_{DEVS} is defined in Appendix D, we give the corresponding Python code on page 53. The random impulse model parameters include the number of dimensions `n_dim` and a dictionary of particle properties `Omega_psi`.

```
def RI_DEVS(n_dim, Omega_psi):

    def init_RI(Psi):
        t = 0
        SPC = {}
        FEL = empty_FEL()
        for id_A in Psi.keys():
            SPC[id_A] = Psi[id_A]["spc"]
            spc_A = SPC[id_A]
            t_A = t + detect_RI(spc_A, Omega_psi)
            FEL = delta_FEL(FEL, id_A, t_A, pr_None)
        s = [t, SPC, FEL]
        return s

    def delta_ext(s, Delta_t_el, x):
        raise ValueError("RI␣must␣not␣receive␣inputs")

    def delta_int(s):
        [t, SPC, FEL] = s
        [id_A, t_] = event_FEL(FEL)
        spc_A = SPC[id_A]
        t_A = t_ + detect_RI(spc_A, Omega_psi)
        FEL_ = delta_FEL(FEL, id_A, t_A, pr_None)
        s_ = [t_, SPC, FEL_]
        Delta_p = impulse_RI(n_dim, spc_A, Omega_psi)
        Y = [["impulse", [id_A, Delta_p]]]
        return [s_, Y]

    def ta(s):
        [t, SPC, FEL] = s
        [id_A, t_A] = event_FEL(FEL)
        Delta_t_int = max([0.0, t_A - t])
        return Delta_t_int

    RI = [delta_ext, delta_int, ta]

    return [init_RI, RI]
```

The random impulse initialization function defined on line 4002 prepares the initial values of the state variables. The state variables include the time `t`, the dictionary `SPC` that records the species of each particle, and the future events list `FEL` that records the randomized times when randomized impulses are to be applied to each particle.

The DEVS model `RI` is constructed on line 4034 from the transition functions and the time advance function. It is then returned on line 4036 along with the initialization function. Because the model accepts no inputs, it effectively has no external transition function. While we do include a function `delta_ext` on line 4014, it does nothing more than raise an exception should it be mistakenly invoked.

The internal transition function `delta_int` is invoked when the time elapsed since the previous event reaches `ta(s)`. The time advance function `ta`, defined on line 4028, simply results in the time remaining until the next event in the `FEL`. Defined on line 4017, `delta_int` changes the first state variable from the old time `t` to the time `t_` of the next event, which is the time of the present impulse. The `FEL` is also updated with a new future impulse time `t_A`. The present impulse itself is added to the list of outputs `Y`.

Looking at `delta_int` from a technological perspective, it turns out that the old state `s` shares references with the new state `s_`. The old and new state variables `t` and `t_` are completely independent, but the second element of `s` and `s_` refer to the same location in memory. Also, because we decided not to accept the performance penalty of reconstructing a new future events list with each update, `FEL` and `FEL_` share data. The fact that old and new states share memory is addressed by the function `simulate`, which replaces the old state every event on either line 3026 or line 3030.

The other place in the code where shared references is a potential problem is in the DEVS model function `coupled_DEVS`. The DEVS coupled model code partitions each submodel output into a port component and a message component. We ensured that all data in the message component of any output is copied before it is received by another submodel. This prevents shared references from occurring between state variables of different submodels.

5 Simulation of Deformable Biological Structures

5.1 Simulation of Vesicle Clusters

Recall from Section 2.7 that vesicles in a presynaptic nerve terminal bind with synapsin protein to form clusters. Vesicles can also become docked to the active zone on the membrane of the compartment. When an action potential arrives, these docked vesicles may release neurotransmitters and trigger an action potential in an adjacent neuron. Our focus in this section is on simulations that capture the dynamics of vesicle clusters as deformable structures, the formation of these clusters, and the manner in which they congregate at the active zone. The TPS model parameters associated with the simulation presented here are defined formally in Appendix E.

Consider a TPS model consisting of particles of three different species: V, S, and D. Each V particle represents a vesicle. Synapsins, being dimers, are represented by pairs of tethered S particles. A D particle is a docking site, a mobile location in the active zone of the membrane on which a vesicle may become docked. Such a model was used in the simulation of Figure 19, which shows two vesicles surrounded by synapsins and docking sites. The vesicles are both tethered to opposite ends of a synapsin.

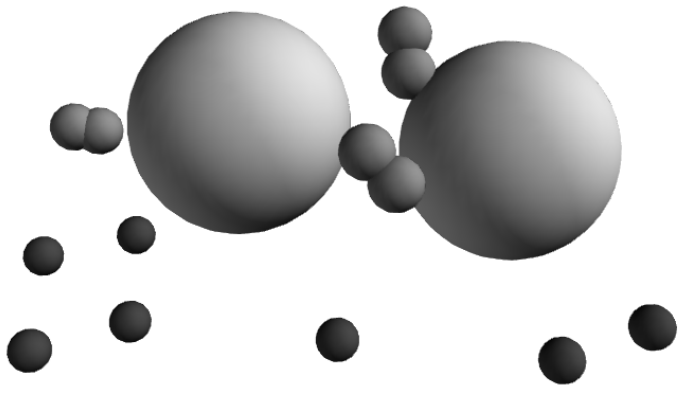


Figure 19: A snapshot of a simulation showing two vesicles (V particles), three synapsins (pairs of S particles) and seven docking sites (D particles).

The tethering of particles is governed by the following rules.

- A V particle and another V particle may never be tethered (vesicles do not bind to one another directly).
- An S particle and another S particle may be tethered at the start of a simulation; if they are not tethered at the beginning, they will never become tethered, and if they are initially tethered, they will never detach (an S particle and its tethered counterpart represent one synapsin).
- An S particle and a V particle will become tethered if they collide, if the S particle is not already tethered to a vesicle (at most two vesicles may bind to a two-particle synapsin), and if the V particle is not already tethered to the S particle's counterpart (we do not allow both ends of a synapsin to bind to the same vesicle).
- A D particle and another D particle may never be tethered.
- A D particle and a V particle will become tethered if they collide, if the D particle is not already tethered to another V particle, and if the V particle is not already tethered to another D particle (vesicles and docking sites pair up).
- A D particle and an S particle may never be tethered.

Table 1 lists the blocking and tethering distances for V, S, and D particles in our presynaptic nerve terminal model.

Particle Species Pair	Blocking Distance $\Delta u_{blocking}$ (nm)	Tethering Distance $\Delta u_{tethering}$ (nm)
V-V	40	$\top$
S-S	2.5	7.5
S-V	22.5	25
D-D	10	$\top$
D-V	25	30
D-S	0	$\top$

Table 1: Blocking and tethering distances for particles representing vesicles, synapsins, and docking sites.

As indicated in the table, approaching docking site and vesicle particles collide and rebound at 24 nm. If tethered and separating, they retract at 28 nm. The distances are chosen to reflect the sizes of actual structures. The diameter of a vesicle is roughly 40 nm, for example, the value used for the vesicle-vesicle blocking distance. Note that a blocking distance of zero indicates that blocking collisions never occur between those species, whereas a tethering distance of infinity ($\top$) indicates that tethering collisions never occur.

The masses of V and S particles are chosen to be roughly proportional to their volumes, whereas each D particle is assigned a relatively high mass for its size to account for resistance in the membrane. The rebounding, retraction, and revolution coefficients are selected such that a considerable amount of kinetic energy is lost when vesicles, synapsins, and docking sites collide. Random impulses are applied to all three of these types of particles to maintain a certain level of kinetic energy in the entire system.

In order to model the formation, disruption, and motion of vesicle clusters, it is necessary to constrain the V, S, and D particles to a region representing the presynaptic nerve terminal compartment. The simplest way to achieve this is to model the nerve cell membrane as a rigid sphere. This is done by adding two particles to the model, one with species M and one with species Z. Both of these particles are given infinite mass, which ensures that they remain stationary. The particle of species M, representing the membrane, is tethered to all V (vesicle), S (synapsin), and D (docking site) particles.

For the sake of convenience, we introduce the parameter r_M to approximate the radius of the compartment, r_V to approximate that of a vesicle, and r_S to approximate the radius of half of a synapsin. As illustrated in Figure 20, an M-V (membrane-vesicle) tethering distance of $r_M - r_V$ keeps vesicles in the compartment, and an M-S (membrane-synapsin) tethering distance of $r_M - r_S$ does the same for synapsins. Because vesicles and synapsins move freely within the compartment, the M-V and M-S blocking distances are both zero.

The D particles, representing docking sites, must be constrained to the spherical surface representing the cell membrane. The M-D blocking and tethering distances are therefore both chosen to be near r_M, with $\Delta u_{tethering}$ slightly greater than $\Delta u_{blocking}$. Another constraint on the docking sites is that they must all be located in that region of the membrane known as the active zone. Hence, all D particles are tethered to the Z particle shown in Figure 20. With the exception of this tethering, the Z particle has no influence on any other particle.

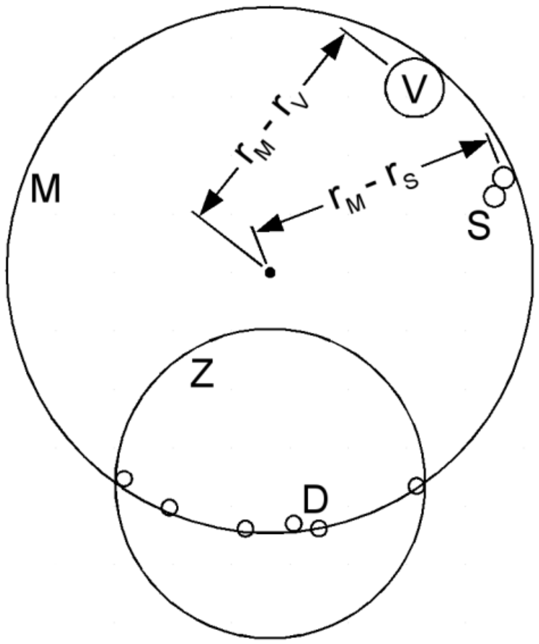
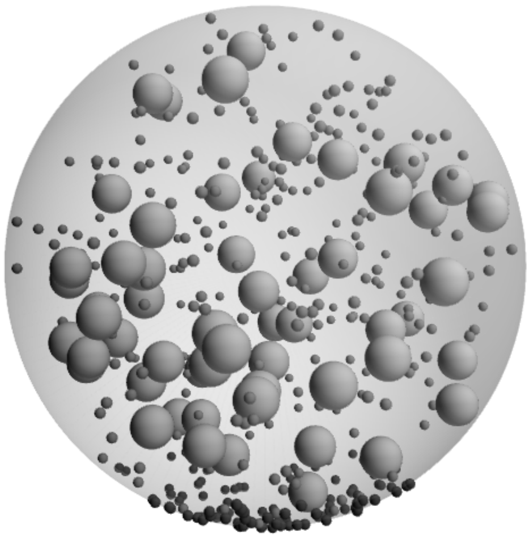
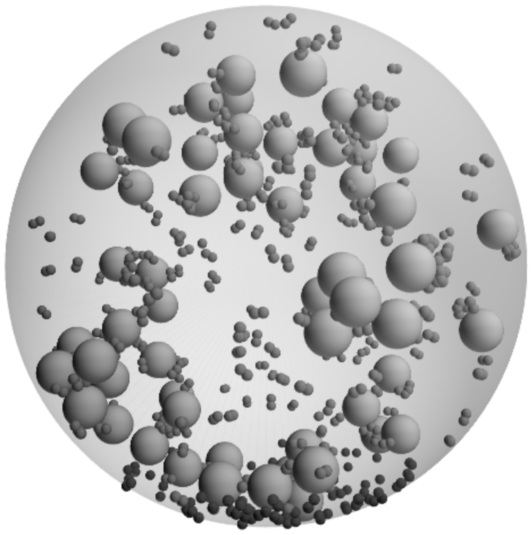


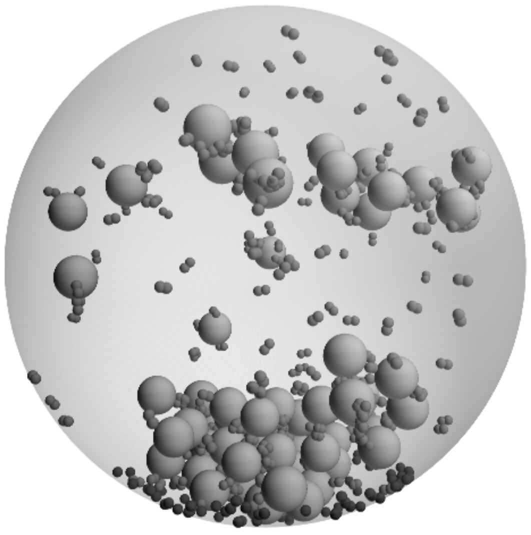
Figure 20: A diagram illustrating the relationships between the five particle species in the presynaptic nerve terminal model.

Figure 21 shows four snapshots of a simulation of a presynaptic terminal of a nerve cell. Vesicles and smaller synapsins move inside the semi-transparent M particle, while docking sites move slowly along the bottom of the membrane. The Z particle that constrains the docking sites is invisible.

Initially, the location of each vesicle and synapsin is randomized within the spherical compartment. None of the vesicles are initially tethered to synapsin. If we were to start the simulation with vesicle clusters or more complex structures, we might have to ensure that no two structures occupy the same space. As it is, we allow vesicles and synapsins to intersect in the initial configuration shown in Figure 21a. Any initial violation of the blocking distance constraints will be resolved by a sequence of blocking collisions occurring in the first few time units. For each synapsin, each of the two particles is initially given the same position. Although the distance between these synapsin particles is obviously less than the blocking distance initially, it is sure to increase after the simulation starts.


(a) $t = 0$


(b) $t = 100$


(c) $t = 200$

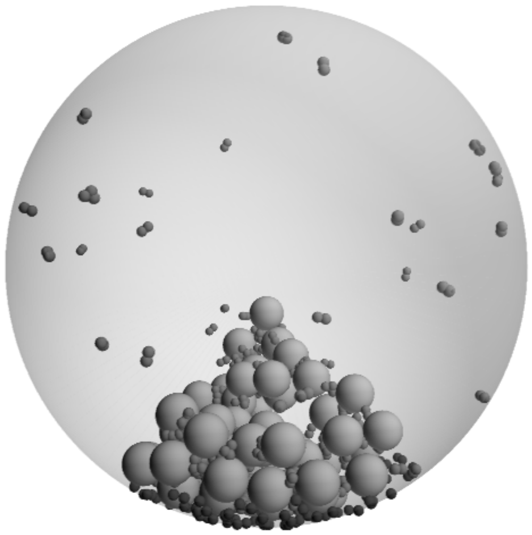

(d) $t = 600$

Figure 21: Four snapshots of a simulation of a presynaptic nerve terminal with a rigid spherical membrane. With a randomized initial distribution, vesicles form clusters that eventually congregate at the active zone at the bottom of the membrane.

The tethering of V and S particles leads to the formation of vesicle clusters. These clusters, which begin to take shape in Figures 21b, and 21c, grow fewer in number but larger in size as the simulation progresses. The tethering of V and D particles constrains some of these clusters to the membrane. In Figure 21d, all of the vesicles have gathered in a single cluster at the active zone.

5.2 Simulation of Deformable Membranes

Although the rigid spherical membrane of Section 5.1 is likely adequate for a number of investigations involving vesicle clusters, the representation of deformable membranes may help capture the dynamics of a presynaptic nerve terminal on a larger scale. Deformable membranes may also prove useful for models of entire nerve cells, networks of nerve cells, tissues, blood vessels, and possibly even large organs.

A simple way to represent a membrane with a TPS is illustrated in Figure 22. Particles are positioned on a surface, and each particle is tethered the nearest neighboring particles. To avoid excessively-sharp folds and other anomalous features, a membrane should have at least two layers; that is, there should be two or more parallel surfaces of particles, and corresponding particles on adjacent surfaces should be tethered together.

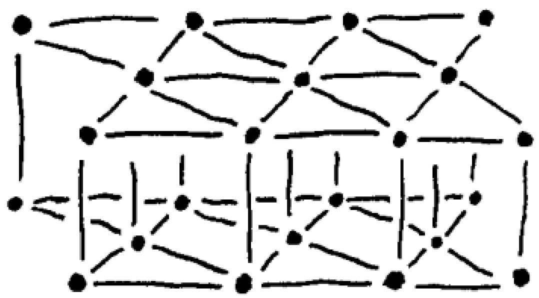


Figure 22: An illustration of how deformable membranes may be represented. Dots are particles, and lines indicate pairs of tethered particles.

Particles on a membrane surface need not be coplanar, and need not be arranged in the triangular grid pattern shown in Figure 22. One alternative is demonstrated in Figure 23, which shows an initially spherical membrane deforming in response to an impact with an initially downward-moving particle. The particles in the membrane were arranged in two concentric icosahedral grids, each constructed by iteratively interpolating the edges of a 20-sided regular polyhedron.

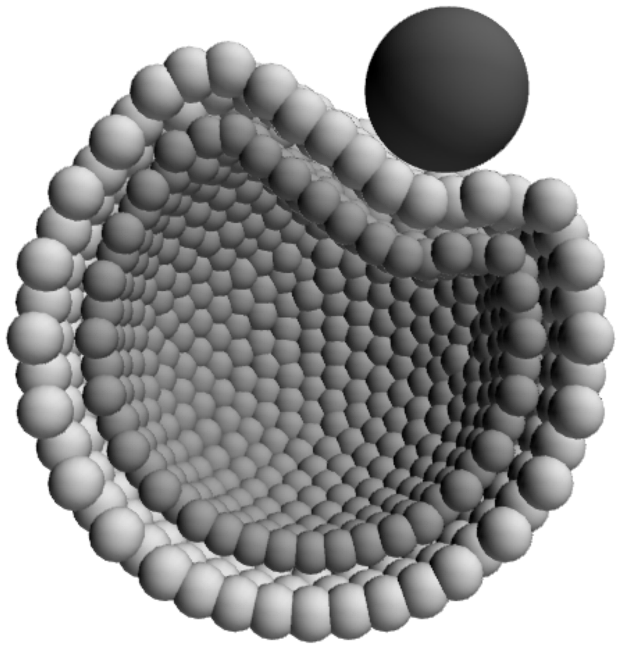


Figure 23: A simulation in which an initially-spherical deformable membrane suffers an impact. The membrane was closed, but the front half is not shown.

Because we composed each of the two concentric icosahedral grids out of a single species of particle, the blocking and tethering distances along the surface of the membrane were all the same. The actual distances between adjacent particles vary, however. As a consequence, we observed undesirable protrusions of the edges of the underlying 20-sided polyhedron as the Figure 23 simulation progressed. Some of these edges are also apparent in Figure 24, a snapshot of a simulation in which we used a deformable surface to model the membrane of a presynaptic nerve terminal. Although the membrane was initially spherical, we assigned initial velocities in an effort to coerce it into the pear-like shape of Section 2.7's Figure 5.

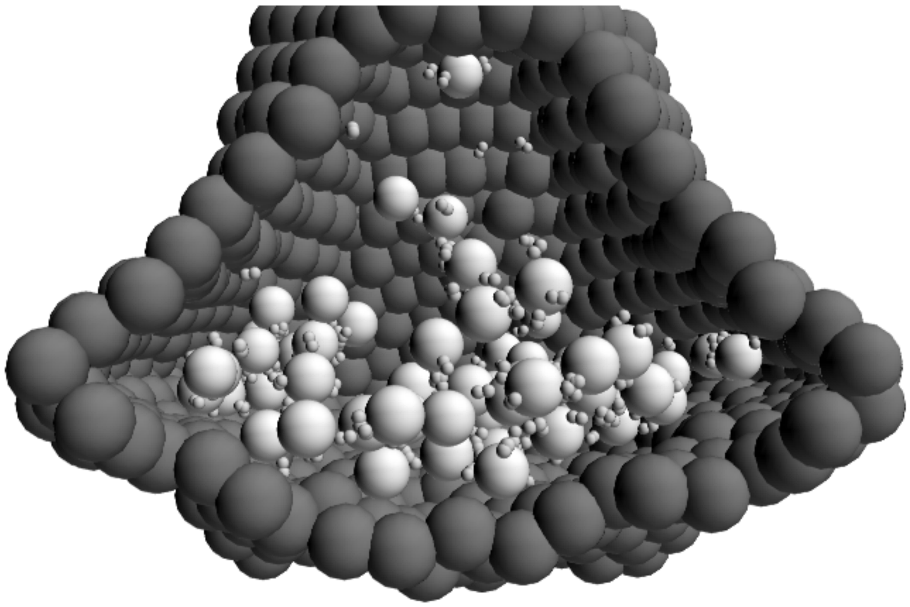
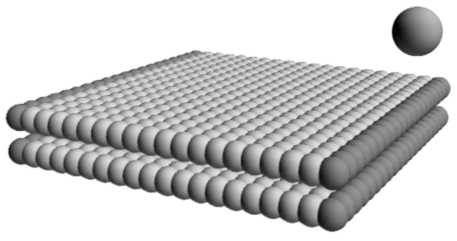
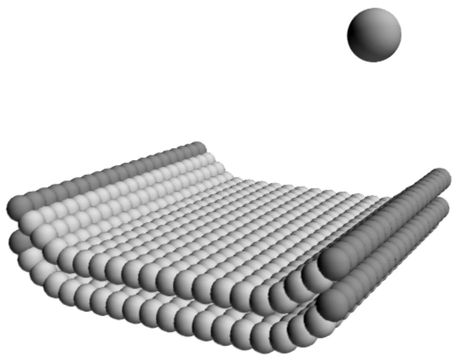


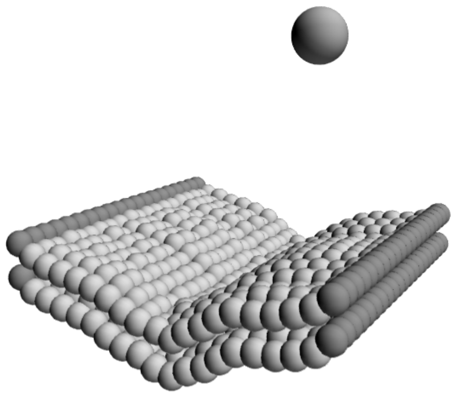
Figure 24: A simulated presynaptic nerve terminal with a deformable membrane. The outer layer and front of the membrane are not shown.

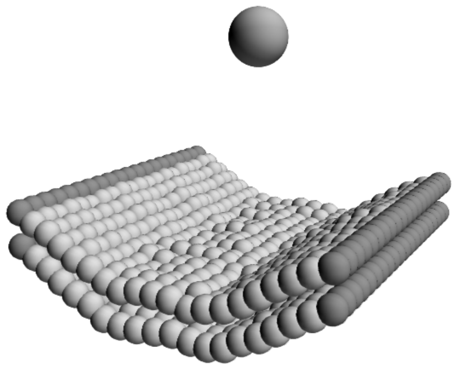
Figure 25 shows an effort to simulate a square biological membrane or soft tissue clamped along two opposing edges. The membrane has two layers of particles, and each particle on the inside is tethered to the four adjacent particles in the same layer and one particle in the opposite layer. All particles along the two clamped edges are assigned a mass of infinity and an initially-zero velocity, rendering them immobile.

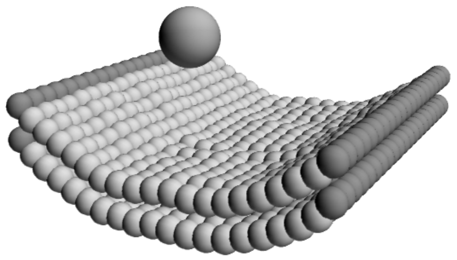
Recall from Section 4.3 that the DEVS TPS model receives $impulse$ messages as inputs. Gravity is incorporated in the Figure 25 model via downward impulses applied to each mobile particle at regular time intervals. As a result of these impulses, the initially flat membrane in Figure 25a is starting to sag in Figure 25b. In Figure 25c the membrane exhibits a wave-like pattern as it responds to internal tethering collisions triggered by the initial fall. Small non-deterministic ripples appear in the membrane as a consequence of two sources of randomness; the order in which particles receive gravitational impulses is randomized, as is the order in which simultaneous collisions are resolved. The gravity-induced waves have mostly subsided after 48 time units, as shown in Figure 25e. Shortly after, a falling particle impacts the membrane and produces the small ridge in Figure 25f.


(a) $t = 0$


(b) $t = 12$


(c) $t = 24$


(d) $t = 36$


(e) $t = 48$

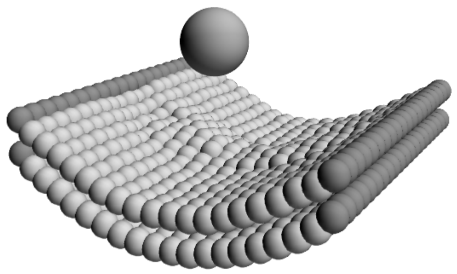

(f) $t = 60$

Figure 25: A square membrane clamped along two edges responds first to gravity, then to an impact with a falling object.

The simulation of Figure 25 reveals two weaknesses in the TPS method. First, while the strain of a macroscopic soft tissue will increase with the applied force, a TPS deformable membrane will reach its maximum length and stop stretching. If, for whatever reason, the force of gravity were to increase by an order of magnitude, the membrane in Figure 25 would sag faster but no further. The other weakness is common to impulse-based methods in general: the computational inefficiency of resolving stable contacts. After the membrane in Figure 25 has sagged, many pairs of particles remain near or at their tethering distances. The effect of this is that collisions occur at an extremely high rate, slowing the simulation to a crawl.

The weaknesses described above may render the TPS, as it is currently defined, unsuitable for macroscopic soft tissues subject to sustained external forces. It is possible, however, that future enhancements to the method could alleviate these problems. Even without any modifications, the TPS may still be useful for modeling cell walls and other small-scale deformable biological membranes that need not be subjected to gravity or continuously stretched.

5.3 Simulation of Action Potentials and Exocytosis

Recall from Section 2.7 that an action potential arriving at a presynaptic nerve terminal has a few notable effects on vesicles. First, through a series of chemical reactions, an action potential may cause synapsins to lose their affinity for vesicles. This may free some vesicles from their clusters. Second, an action potential may trigger exocytosis, causing certain vesicles that are docked at the active zone to fuse with the membrane. Third, in some cases, a docked vesicle may be released from the active zone without fusing with the membrane.

There are several reasons to simulate action potentials and exocytosis. One reason is to identify conditions under which docked vesicles are likely to become depleted, a situation that would prevent an action potential from being transmitted from one neuron to the next.

Although an action potential is a complex process involving numerous types of chemical reactions and dramatic electric field changes, we model it as a period of time delimited by precise, pre-determined start and end times. We model exocytosis by selecting certain docked vesicles and removing them from the spherical membrane of Section 5.1. We attempt to capture neither the deformation of a vesicle as it fuses, nor the neurotransmitters that it releases.

In Section 4.3, a DEVS coupled model named TPS was described as the uppermost level in a DEVS model hierarchy. In order to extend the presynaptic nerve terminal model of Section 5.1 with action potentials and exocytosis, another level was added above the TPS. The PNT model, shown in Figure 26, is a DEVS coupled model with three submodels: $compartment$, $action$, and $fusion$.

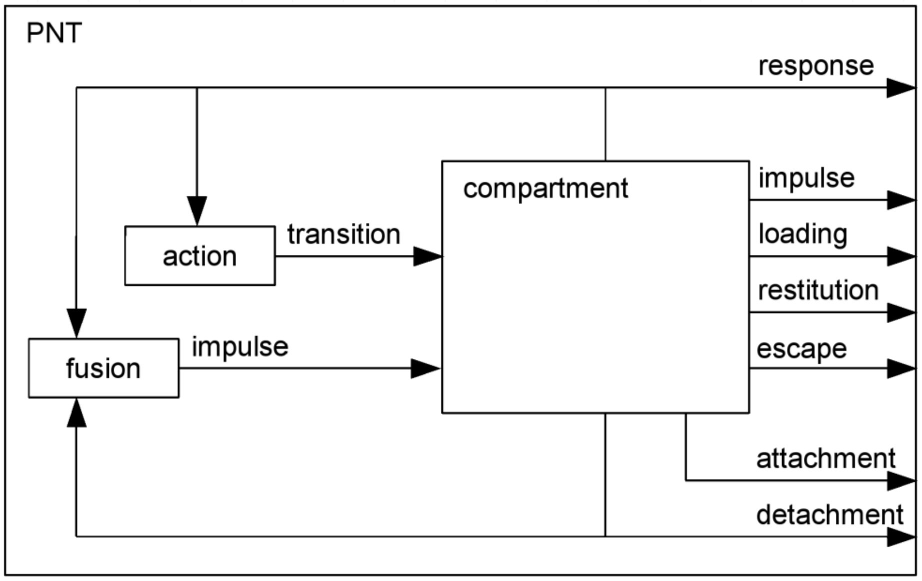


Figure 26: The structure of the PNT coupled DEVS model.

The $compartment$ submodel is a TPS model with an extra layer of model parameters. It includes the V (vesicle), S (synapsin), D (docking site), M (membrane), and Z (active zone) particle species of Section 5.1. The $action$ submodel receives $response$ messages from the $compartment$ in order to keep track of the locations of all of the particles. It sends a $transition$ message to the $compartment$ each time an action potential either begins or ends, and the information in these messages affects which particles in the $compartment$ become tethered or untethered. Like $action$, the $fusion$ submodel also receives $response$ messages to track particle locations. It also receives $detachment$ messages, and takes notice if a V particle and the M particle become detached. A V-M detachment is an indication that a vesicle is fusing, and $fusion$ responds by sending a sequence of $impulse$ messages to the $compartment$ to push the vesicle outwards well beyond the membrane.

At the beginning of an action potential, the $action$ submodel randomizes several time values that determine when vesicles are to detach from synapsins, docking sites, and the membrane. In reality, these effects are the result of a complex sequence of chemical reactions; a set of randomized times provides a simple means to model this process. The time values are recorded in a variable named Φ that accompanies the $transition$ messages sent from $action$ to $compartment$. Each individual time value is given by an expression of the form $\Phi\left(reaction\right)\left(id_A\right)$. Here $reaction$ is one of the following: "SV", representing a synapsin losing its affinity for vesicles during an action potential; "DV", representing a docking site no longer accepting vesicles; and "VM", representing the fusing of a vesicle with the membrane. The integer id_A identifies the synapsin particle, the docking site, or the vesicle in question. In the equation below, t is the time at which the action potential starts, and $exponential\left(\tau\right)()$ is a positive real number selected randomly from an exponential distribution with mean value of τ.

$$\Phi\left(reaction\right)\left(id_A\right) = t + exponential\left(\tau\right)()$$

The average time τ is calculated using one of the three model parameters $T_{AP}\left(\text{"}SV\text{"}\right)$, $T_{AP}\left(\text{"}DV\text{"}\right)$, and $T_{AP}\left(\text{"}VM\text{"}\right)$, depending on the type of reaction. There is also a geometry-dependent factor, explained further below.

$$\tau = T_{AP}\left(reaction\right) \cdot \left(\frac{\Delta u_{ac} + \Delta u_{bc}}{\Delta u_{ab}}\right)^{g_{AP}}$$

The value $\Phi\left(\text{"}SV\text{"}\right)(139)$, for example, is the simulated time at which the synapsin particle with ID 139 loses its affinity for vesicles. If that time arrives before the action potential ends, then the synapsin identified by 139 will detach from any vesicle tethered to it. It will also remain detached from any other vesicles until the end of the action potential.

Similarly, $\Phi\left(\text{“}DV\text{”}\right)(84)$ is the time when docking site particle 84 detaches and remains detached from any vesicle, and $\Phi\left(\text{“}VM\text{”}\right)(427)$ is the time when vesicle 427 is scheduled to detach from the membrane to simulate exocytosis.

Recall from Section 5.1 that all vesicles are tethered to the M particle representing the membrane. When a V-M tethering collision occurs, it appears to an observer as if a vesicle simply rebounded off the inside of the spherical membrane. A V-M detachment allows a vesicle to escape the compartment by passing through the membrane (this is how we simulate exocytosis, though in the real system the vesicle fuses and becomes part of the membrane). For vesicle 427 to detach from the membrane, we not only require the time $\Phi\left(\text{“}VM\text{”}\right)(427)$ to arrive before the end of the action potential, but we also require vesicle 427 to be tethered to a D particle at the time when the V-M tethering distance is reached. This guarantees that the vesicle is docked at the active zone before fusing.

We now explain the geometry-dependent factor used in the calculation of τ. When an action potential reaches a presynaptic nerve terminal, calcium ions enter the compartment in the vicinity of the active zone. The calcium ions do not directly cause synapsin to lose their affinity for vesicles, but they do trigger a sequence of chemical reactions that ultimately has this effect. A synapsin located near the active zone is more likely to take part in these reactions, and separate from vesicles, than a synasin located further away. For this reason, the average time τ increases with distance from the active zone.

Because the active zone is a region and not a point, we calculate the distance from any position to the active zone using the triangle in Figure 27 instead of a straight line. The top of this triangle, point c, is the position of a synapsin particle at the beginning of an action potential. The points a and b at the bottom are located at the top of the active zone as determined by the radius r_Z associated with the Z particle. Points a and b are also chosen to be in the plane that includes point c, and the center the M particle representing the membrane, and the center of the Z particle delimiting the active zone. The distance between points a and c is Δu_{ac}, the distance between b and c is Δu_{bc}, and the distance between a and b is Δu_{ab} which equals $2{\cdot}r_Z$. As indicated by the equation above, the average time τ before the synapsin at c loses its affinity for vesicles is determined in part by the ratio of Δu_{ac} plus Δu_{bc} to Δu_{ab}. This ratio is raised by g_{AP}, an exponent that parameterizes the gradient of the chemical reactions triggered by the calcium influx at the active zone. If g_{AP} is large (eg. $g_{AP} = 3$ or $g_{AP} = 4$), synapsins at a distance are much less affected by an action potential than those that are close to the active zone.

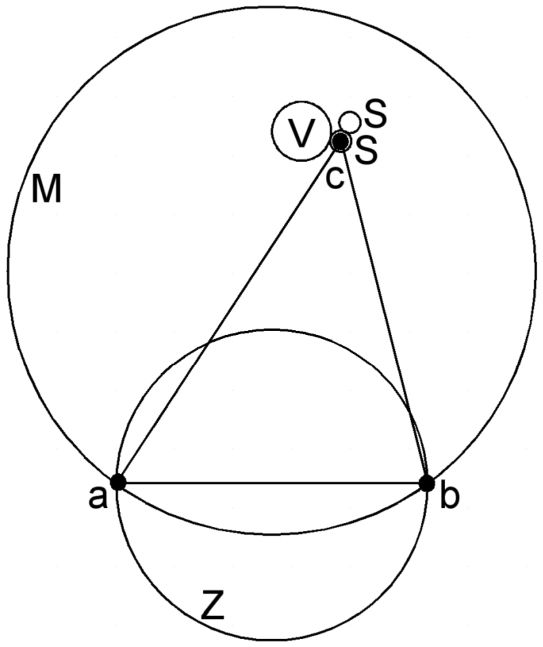
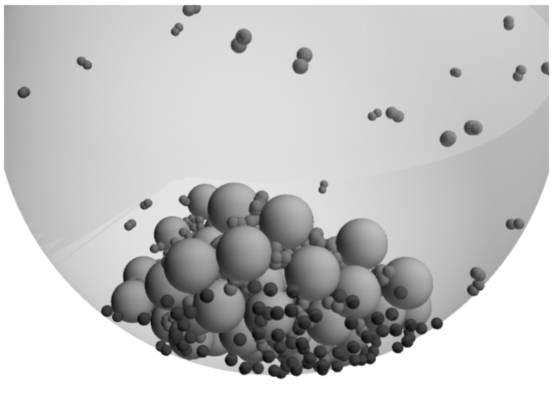
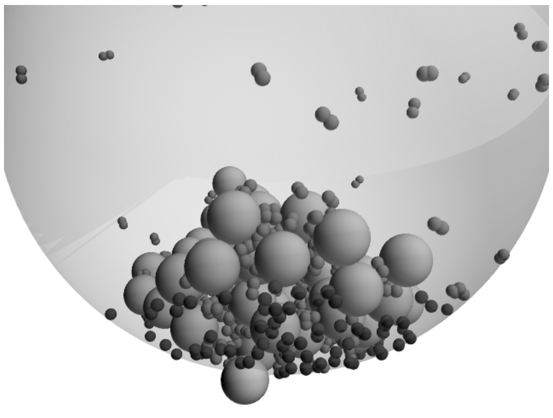


Figure 27: The triangle used to quantify distances from the active zone.

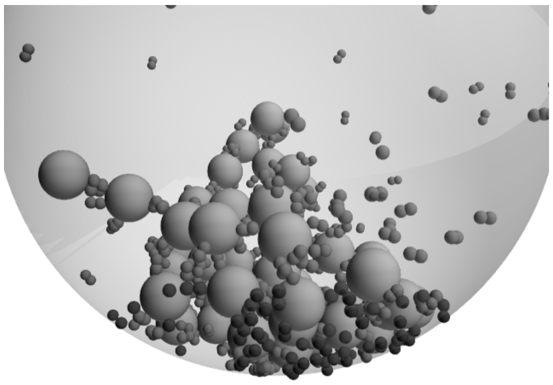
In Section 5.1, vesicle clusters were shown forming in a simulated presynaptic nerve terminal. At time 600 in Figure 21, the vesicles had congregated in a single cluster at the active zone. Figure 28 shows the continuation of that simulation, with an action potential starting at time 600 and lasting 30 time units. The vesicle cluster in Figure 28a is exactly the same as the one in Figure 21d, but viewed from a different angle. Figure 28b is a snapshot taken 20 time units into the action potential. At this time, a vesicle can be seen escaping the compartment in a simulated process of exocytosis. In Figure 28c, taken 20 time units after the end of the action potential, recently-freed synapsins can be seen immediately to the right of the cluster. As there are fewer synapsins binding with vesicles, the cluster loosens and flanges emerge. A single isolated vesicle can be seen at the top of Figure 28d, having completely detached from the cluster. As shown in Figures 28e and 28f, the cluster consolidates as time progresses and new vesicle-synapsin bonds are formed.



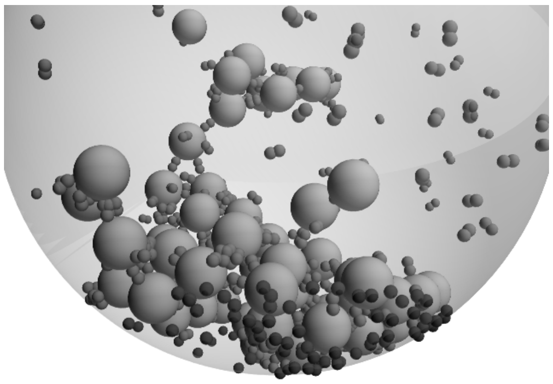
(a) $t = 600$



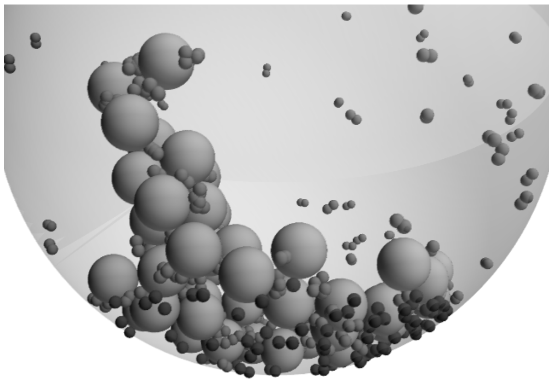
(b) $t = 620$



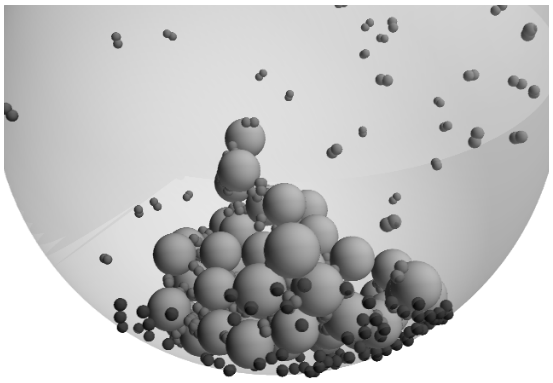
(c) $t = 650$



(d) $t = 680$



(e) $t = 750$



(f) $t = 900$

Figure 28: Six snapshots of a vesicle cluster reacting to an action potential in a simulated presynaptic nerve terminal. Here $g_{AP} = 3$, which indicates that during an action potential, the synapsins that are close to the active zone at the bottom are the most likely to lose their affinity for vesicles.

Figure 29 shows a real vesicle cluster situated on the membrane of a presynaptic nerve terminal. This particular cluster is considerably larger than the one simulated in Figure 28a, though the overall shape is not entirely dissimilar.

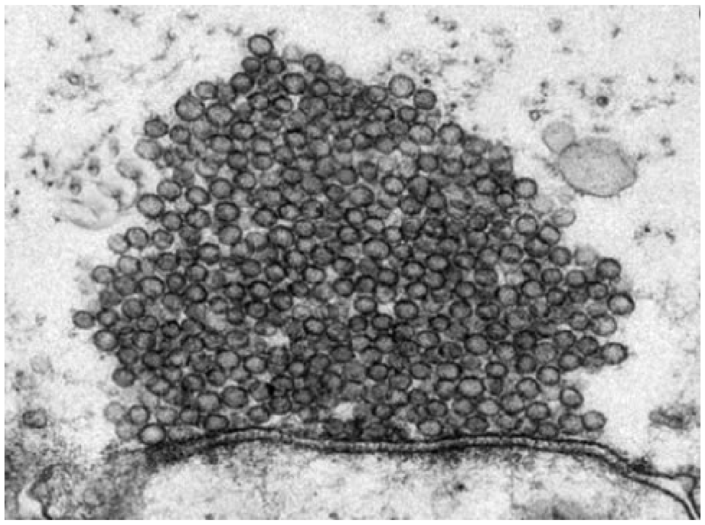


Figure 29: An electron microscopy image of a real vesicle cluster in a lamprey spinal neuron (reprinted with permission from Dr. Oleg Shupliakov).

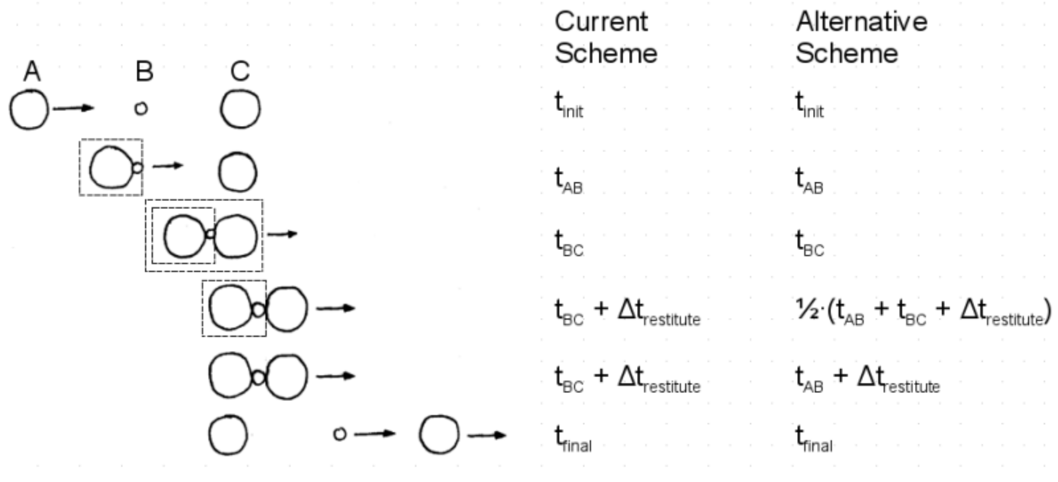
Simulations like the one in Figure 28 may help biologists investigate the role of synapsin in the human brain. Experimental results presented in [54] suggest that synapsin helps to maintain a number of vesicles in the vicinity of the active zone, which in turn increases the chance that a sequence of action potentials will be transmitted from one neuron to the next. Simulations can be used to quantify the relationship between synapsin concentration and the availability of docked vesicles. An iterative research process, involving both simulation and experimentation, could potentially lead to a better understanding of neurotransmission coupled with an ability to predict the behaviour of presynaptic terminals under various conditions.

6 Discussion

6.1 In Hindsight

Two design decisions were made during the project that, in hindsight, could have been improved. One improvement pertains to the TPS method described in Section 4.1, and the other pertains to the DEVS TPS model of Section 4.3. While we recommend these changes to others, we were unable to adopt them because they were first considered at a late stage in the project.

In the TPS method, a collision between two particles results in an immediate loading phase and a subsequent restitution phase. In Section 4.1, the parameter $\Delta t_{restitute}$ was introduced as the minimum time between loading and restitution. In hindsight, the same parameter should have been used as the maximum time between loading and restitution. Figure 30 contrasts the "current scheme" we adopted with the "alternative scheme" we now recommend. In both schemes, we have a loading of particles A and B, followed by a loading of B and C, followed by a restitution of B and C, followed by a restitution of A and B. The difference is that, in the alternative scheme, the restitution of B and C is scheduled at an earlier time to allow the A-B restitution to occur at a duration of $\Delta t_{restitute}$ after the A-B loading. In the current scheme, the A-B restitution is delayed.



A
B
C
Current Scheme
Alternative Scheme
t_{init}
t_{init}
t_{AB}
t_{AB}
t_{BC}
t_{BC}
$t_{BC} + \Delta t_{restitute}$
$\frac{1}{2} \cdot (t_{AB} + t_{BC} + \Delta t_{restitute})$
$t_{BC} + \Delta t_{restitute}$
$t_{AB} + \Delta t_{restitute}$
t_{final}
t_{final}

Figure 30: Current and alternative schemes for calculating restitution times.

In the alternative scheme, if two particles A and B collide to form a two-particle loaded group, the restitution occurs after a duration of $\Delta t_{restitute}$ as in the current scheme. If, however, the A-B collision results in a loaded group of three or more particles, then the A-B restitution time is the average of the current time and the minimum restitution time in the remainder of the group.

The alternative scheme has two advantages. First, it discourages the formation of excessively large loaded groups that occur with the current scheme if $\Delta t_{restitute}$ is too high. Although small loaded groups of two of three particles can dramatically speed up a simulation, loaded groups of 100 or more particles are detrimental to both performance and validity. The second advantage of the alternative scheme is that it is easier to implement. The current scheme requires future restitution times to be repeatedly re-scheduled. Also, the priority function pr passed into a future events list of restitution events must be continually updated to resolve simultaneous restitutions. The alternative scheme would alleviate the need for rescheduled restitutions and changing priority functions. It would also render unnecessary the state variables $N_{loading}$ and $n_{loading}$ of the DEVS $responder$ submodel defined in Appendix D.

We now turn our attention to the other improvement. In Section 4.3, we presented the DEVS $detector$ model as a coupled model consisting of a $tracker$ submodel and a $lattice$ submodel. In hindsight, we would likely have divided the $lattice$ in three, introducing a submodel named $network$ and another named $scheduler$.

Recall that the $lattice$ is itself a coupled model with submodels named $subV$. Each $subV$ is associated with a subvolume of space, detecting and scheduling collisions that occur in or near that subvolume. If one collision is scheduled by one $subV$, and another collision is scheduled for the same simulated time by another $subV$, then each collision ends up with a 50% chance of occurring first. A problem arises, however, if one collision is scheduled by one $subV$, and two other collisions are scheduled for the same simulated time by another $subV$. Of course we would like each of the three collisions to have a 1 in 3 chance of occurring first. Unfortunately, each $subV$ would gain priority with a 50% probability, and consequently the two collisions in the latter $subV$ would each end up with a 1 in 4 chance of occurring first. From a practical perspective, simultaneous collisions are sufficiently rare that a slight bias in their randomized order should not undermine the validity of an entire simulation. But the problem remains troubling aesthetically.

To ensure an unbiased random ordering of simultaneous collisions, all particle-particle collisions should be scheduled by a single DEVS model. This is the rationale for adding a $scheduler$ submodel to the $detector$. The $scheduler$ would receive $blocking$ or $tethering$ messages containing collision information for pairs of particles anywhere in the simulated space. Each of these messages would be sent at the time when a collision is detected, not the time when the collision actually occurs. The $scheduler$ would schedule the collisions itself, and output the $collision$ messages.

In addition to the separation of collision detection and scheduling, the $detector$ could be modified such that the $lattice$ would detect only blocking collisions and not tethering collisions. Tethering collisions would instead be detected by a submodel named $network$, which would search for these collisions by considering each pair of tethered particles. This modification is logical because proximity is a better predictor of blocking collisions than tethering collisions.

The structure of the alternative $detector$ model, with the $scheduler$ and $network$ submodels, is illustrated in Figure 31.

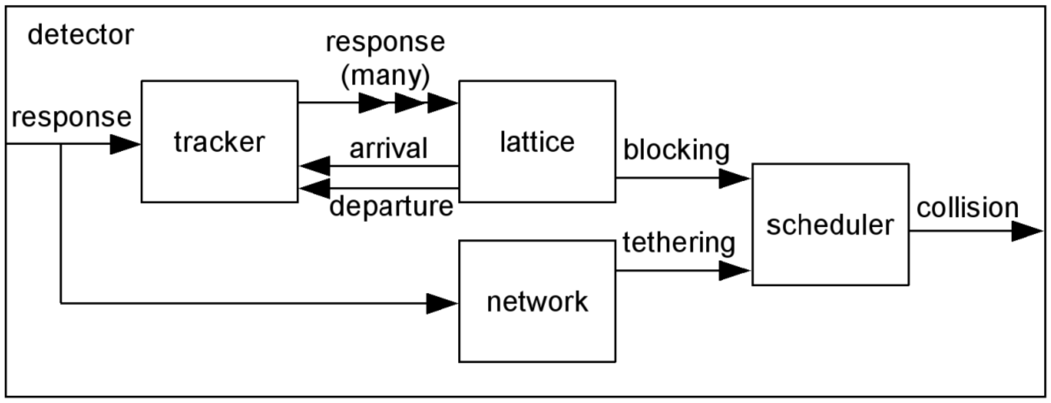


Figure 31: The structure of the recommended alternative DEVS $detector$ model.

6.2 On Impulse-Based Dynamic Simulations of Deformable Structures

The TPS method described and demonstrated in this thesis provides, arguably, the most convincing evidence to date that impulse-based methods can be used to simulate the dynamics of deformable structures. The TPS is very similar to the impulse-based method of [41], yet contradicts the assertion that such methods require models to be comprised of only rigid bodies. The key concept behind the TPS is that, in place of forces and differential equations, it is the net effect of many instantaneous collisions that produces a gradual process of deformation. In this thesis we have applied the TPS to deformable vesicle clusters and membranes, and students Sanaa Lissari and Nada Farran have recently modeled deformable cylindrical and helical proteins using our code. Another student, Hamel Yigang, has simulated a fluid by adding small isolated particles to a TPS model.

The mathematics of the TPS presented in Appendix B allow chemical reactions to be represented by the tethering or untethering of certain pairs of particles. We have not allowed particles to be added or removed during a simulation, though such an enhancement remains possible. Had we allowed particles to be removed during a simulation, we could have made the fusing vesicles in Section 5.3 disappear instead of detaching from the membrane and exiting the compartment. If one were to allow particles to be either introduced or removed, one could simulate a chemical reaction by replacing two colliding particles A and B with some particle C, or replacing some particle C with particles A and B. With this enhancement, the TPS would be a compelling option for simulations capturing the reaction and diffusion of chemicals in combination with the deformation of larger structures.

Simultaneous and nearly-simultaneous collisions have posed a problem for impulse-based dynamic simulations in general. The approximation we introduce, involving the formation of loaded groups of particles, can be used in rigid body simulation methods as well as the TPS. The approximation is compelling in that it is relatively simple, obeys the law of conservation of momentum, and does not allow energy to increase. Further investigation would be required to determine whether the approximation is sufficiently accurate for models involving small numbers of large rigid bodies moving in a deterministic fashion. In the case of a biological system, we expect large numbers of small particles exhibiting a considerable degree of random motion. In such models, the error introduced by temporarily combining particle masses seems like a small price to pay to prevent a simulation from stalling.

As mentioned in Section 5.2, simulations of deformable membranes have revealed two weaknesses of the TPS method. The strain on a deformable TPS structure will reach its limit given a sustained external force, regardless of the magnitude of that force. A mass-spring-damper model, by contrast, has the desirable potential to stretch further as the external force is increased. Also, in situations involving stable contacts, the TPS will suffer from poor performance due to a high rate of particle collisions. These weakness are most apparent in models of macroscopic structures subject to external forces. The simulation of a clamped membrane subject to gravity, shown in Section 5.2's Figure 25, is a good example of an application requiring either an alternative method, or perhaps some future enhanced version of the TPS. In the case of small-scale self-assembling biological structures subject to random motion, however, stable contacts are far less likely to occur. Dr. James J. Cheetham, a biologist a Carleton University, is currently using the TPS to run and analyze simulations of vesicle clusters similar to those of Sections 5.1 and 5.3. The TPS may well prove useful for other small-scale biological systems.

A detailed comparison of the TPS with alternative dynamic simulation methods remains important future work. Here we speculate that the FEM would be difficult to apply to small-scale self-assembling biological structures, as rapid deformation and re-structuring would require frequent re-calculation of mass and stiffness matrices. Mass-spring-damper systems suffer from the threat of instability, though it is possible to address this problem with constraints as done in the method of [50] and [51]. Recall from Section 2.6 that, although it is described as "impulse-based", the [50, 51] method requires new trajectories to be computed for each node in a deformable structure at regular time intervals. The TPS simulates deformation with impulses applied not at regular intervals, but rather in response to collisions. The [50, 51] method seems to be the more computationally efficient, as the number of collisions in a TPS simulation can be extremely high. The TPS is appealing in that all calculations are analytic; there is no need for the iterative algorithm of [50, 51] that repeatedly re-calculates trajectories until all constraints are satisfied within an arbitrary tolerance level.

Impulse-based methods like the TPS are compelling in large part because they alleviate the need to choose regular time intervals. In the case of a biological system, the selection of an appropriate time interval would be complicated by the fact that interacting biological entities may differ in size and momentum by many orders of magnitude. A suitable interval for one entity may be too large or too small for another. The absence of regular time intervals proved beneficial in our work as it allowed us to select times for random impulses from exponential distributions. Otherwise, those times would have had to be discretized. With their ability to respond to information delivered at any simulated time, impulse-based methods are likely among the easiest to integrate with other simulation algorithms.

6.3 On DEVS-Based Simulations of Biological Systems

Arguably the most compelling reason to use DEVS is that the formalism facilitates hierarchical model design. The separation of model and simulator is also a significant advantage, but the separation of different aspects of the model is the key to addressing the complexity of a biological system and the methods used to simulate it.

The most intuitive way to structure a biological DEVS model is to define separate submodels for separate biological entities. This is the approach taken in [32], in which separate DEVS submodels are used represent a cell membrane, a cytoplasm, and a nucleus. The same paper argues that DEVS is in some regards less suitable for biological systems than other modeling formalisms such as stochastic Petri Nets and stochastic π-Calculus, as it is difficult to simulate interactions between three or more entities using links between pairs of DEVS submodels.

Adopting an alternative formalism is one way to address the shortcomings of DEVS identified in [32], but one can also design a DEVS model hierarchy in a different fashion. In previous work, numerous cellular biological models have been designed by partitioning space into discrete regions, each with an associated DEVS atomic model [30]. Described in this thesis is the first continuous-space biological DEVS model. The DEVS TPS model also partitions space using submodels, but only at a lower level in the hierarchy. At the uppermost lever, we define different submodels to address different aspects of a TPS algorithm: the detection of collisions, the response of particles to collisions, and the generation of random impulses. This is the approach we recommend for the design of biological models in general: to separate different algorithms or aspects of algorithms at upper levels in a DEVS model hierarchy, and to partition space at lower levels.

We argue that suitable hierarchies can help biologists and medical researchers simulate complex biological systems with code that can be easily understood and modified. One can interpret the code of our random impulse model on page 53, for example, without being distracted by the complications of particle collisions. If one were to implement the first modification of Section 6.1, the $responder$ submodel would need to be modified, but the $detector$ could be ignored. For the second modification, a modeler would have to re-design the $detector$, but would not have to worry about the $responder$.

We now discuss how our use of DEVS fits in with the pursuit of biological simulation algorithm integration described in Section 2.4. Ours is a single-formalism approach in which different algorithms are each given their own DEVS model, and the algorithms are integrated via the coupling of those models.

Recall from Section 2.4 that [35] is a good example of algorithm integration. This recent work combines the Next Subvolume Method of [12], which tracks the concentrations of chemicals in various subvolumes, with an algorithm that tracks the positions of relatively large individual particles. Looking at the diagram in Figure 32, 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 a particle. The algorithm of [35] approximates the effect of large individual particles on the reaction and diffusion of surrounding chemicals.

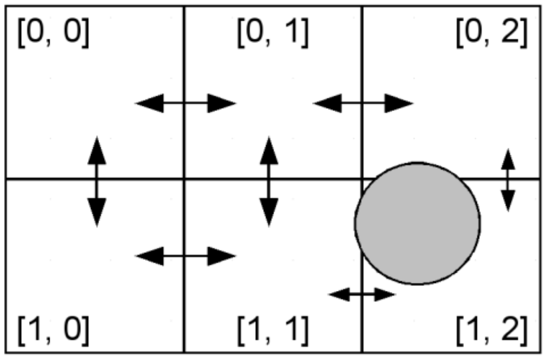


Figure 32: A model in which chemicals react within each subvolume, diffuse between subvolumes as shown by the arrows, and avoid the large particle.

Now suppose we designed a DEVS model, named NSM, that uses the enhanced version of the Next Subvolume Method described in [35] to output the concentration of each chemical in each subvolume. We could then combine it with the TPS, which would calculate the motion of the large particles in the system. Assuming that NSM 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 Next Subvolume Method, we would link all three DEVS models as shown in Figure 33. A change in a particle's motion is sent from the TPS to the $TPSS$, and a change in the particle's position is then sent to the NSM. Finally, a change in chemical concentrations may be sent back to the TPS as a $transition$ message to affect the formation and disruption of deformable structures.

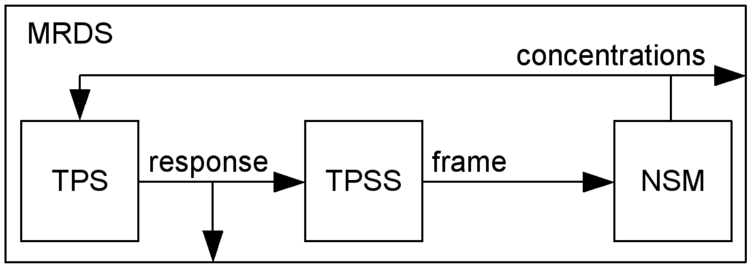


Figure 33: A hypothetical DEVS coupled model that combines two algorithms.

DEVS models could be defined to predict the potential fields that propagate along nerve cell membranes, or simulate hydrodynamic resistance using computational fluid dynamics. These models could then be linked with dynamic simulation DEVS models like the TPS, or chemical reaction and diffusion DEVS models like the hypothetical NSM, in pursuit of realistic yet well-designed simulations of biological systems.

7 Conclusion

Recall that, as Kitano explains in [1], a simulation may be used by a biologist or medical researcher to investigate a hypotheses formed from the observation of experimental data. Our simulated presynaptic nerve terminal, for example, may be used to test the idea that the role of synapsin is to hinder the depletion of vesicles docked at the active zone. Inevitably, a simulation will make new predictions, and those predictions can be tested using new *in vitro* and *in vivo* studies. Our simulations predict how spatial distributions of vesicles change over time, and new experiments would be needed to determine whether similar distributions occur in reality under similar conditions. New experimental data will lead to new hypotheses, which will in turn motivate improvements to a simulation. It may be found, hypothetically, that long filaments known as actin must be added to our presynaptic nerve terminal model in order to yield results that match observations. The hope is that an iterative research process, involving both experimentation and simulation, will lead to an improved understanding of a biological system and the ability to predict its behaviour.

The two main contributions of this thesis are not specific to the presynaptic nerve terminal model or any other particular biological system. They are, rather, a simulation method and a modeling technique that can be applied to the study of a wide range of biological systems.

The TPS method simulates the dynamics of deformable structures using impulses that occur in response to particle collisions. The detection and resolution of these collisions require only analytic computations, whereas many alternative methods involve systems of equations or inequalities that need to be solved simultaneously or iteratively. In the design of the TPS, we addressed the threat of simultaneous and nearly-simultaneous collisions by introducing a minimum duration $\Delta t_{restitute}$ between a collision's loading and restitution phases. In hindsight, we recommend that others use this parameter as a maximum duration, not a minimum. One advantage of the TPS over most of its alternatives is that it alleviates the need for regular time intervals. These intervals must otherwise be selected to accomodate interactions between biological entities that may differ in size by orders of magnitude. Results obtained to date suggest that the TPS requires further enhancement if it is to be effective for macroscopic deformable structures subject to sustained external forces. The method is already promising, and in use, for simulations of small-scale self-assembling deformable biological structures exhibiting random motion.

The DEVS formalism has previously been used to design both non-spatial and cellular biological DEVS models. We have applied the formalism to biological models with continuous space. The real question was not whether continuous-space biological DEVS models were possible to design, but whether suitable hierarchies could be chosen to fully benefit from the formalism's adoption. The presented DEVS TPS model featured a hierarchy that separated various aspects of the TPS method: the generation of random impulses, the tracking of particles from one subvolume to another, the detection of collisions within each subvolume, and the response of particles to collisions.

When opting for a hierarchical structure, a modeler accepts a certain amount of additional complexity in the form of messages that must be passed from one submodel to another. The benefit is that a complex simulation routine may be effectively replaced by a reusable simulator and a set of relatively simple transition functions. We have presented a minimalistic formulation of DEVS in which the parameters of any DEVS model are repeatedly transformed from one layer to the next, ultimately yielding a vector of the form $[\delta_{ext}, \delta_{int}, ta]$. The simplicity of this layering technique should appeal to programmers who wish to implement their own DEVS simulator and hierarchical models.

It seems likely that the pursuit of ever more realistic biological models will increasingly motivate the integration of different biological simulation algorithms. Our advice to others is that such algorithms be coupled as submodels at upper levels in a DEVS model hierarchy. We also recommend that space be partitioned at lower levels. It is intuitive to associate DEVS models with specific biological entities, but we warn against this approach. Properties of different biological entities can instead be organized by an appropriate selection of model parameters and state variables.

References

[1] Hiroaki Kitano. Computational systems biology. *Nature*, 420:206–210, 2002.

[2] David Fange and Johan Elf. Noise-Induced Min Phenotypes in E. coli. *PLoS Computational Biology*, 2(6):e80, 2006.

[3] Joel Brown, Stephen Sorkin, Cynthia Bruyns, Jean-Claude Latombe, Michael Stephanides, and Kevin Montgomery. Real-time simulation of deformable objects: Tools and application. In *Computer Animation*, pages 228–236, 2001.

[4] Trent M. Guess and Lorin P. Maletsky. Computational modelling of a total knee prosthetic loaded in a dynamic knee simulator. *Medical Engineering & Physics*, 27(5):357–367, 2005.

[5] Tong Wang, Tsorng-Whay Pan, Z. W. Xing, and Roland Glowinski. Numerical simulation of rheology of red blood cell rouleaux in microchannels. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 79(4):041916+, 2009.

[6] Michael L. Klein and Wataru Shinoda. Large-Scale Molecular Dynamics Simulations of Self-Assembling Systems. *Science*, 321(5890):798–800, 2008.

[7] Herbert M. Sauro, David Harel, Marta Kwiatkowska, Clifford A. Shaffer, Adelinde M. Uhrmacher, Michael Hucka, Pedro Mendes, Lena Strömback, and John J. Tyson. Challenges for Modeling and Simulation Methods in Systems Biology. In *Proceedings of the Winter Simulation Conference (WSC)*, Monterey, CA, USA, 2006.

[8] Carsten Maus, Mathias John, Mathias Röhl, and Adelinde M. Uhrmacher. Hierarchical Modeling for Computational Biology. *Formal Methods for Computational Systems Biology*, 5016:81–124, 2008.

[9] Kouichi Takahashi, Satya Nanda Vel Arjunan, and Masaru Tomita. Space in systems biology of signaling pathways towards intracellular molecular crowding in silico. *FEBS Letters*, 579(8):1783–1788, 2005.

[10] Daniel T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[11] Dieter Armbruster, John D. Nagy, E. A. F. van de Rijt, and J. E. Rooda. Dynamic Simulations of Single-Molecule Enzyme Networks. *Journal of Physical Chemistry B*, 113(16):5537–5544, 2009.

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

[13] Dominic P. Tolle and Nicolas Le Novère. Particle-Based Stochastic Simulation in Systems Biology. *Current Bioinformatics*, 1(3):315–320, 2006.

[14] Laurier Boulianne, Michel Dumontier, and Warren J. Gross. A Stochastic Particle-Based Biological System Simulator. In *Proceedings of the Summer Simulation Conference (SCSC)*, San Diego, CA, USA, 2007.

[15] Laurier Boulianne, Sevin Al Assaad, Michel Dumontier, and Warren Gross. Grid-Cell: a stochastic particle-based biological system simulator. *BMC Systems Biology*, 2(1):66–74, 2008.

[16] Joel R. Stiles and Thomas M. Bartol. Monte Carlo Methods for Simulating Realistic Synaptic Microphysiology Using MCell. *In Computational Neuroscience: Realistic Modeling for Experimentalists (Edited by Erik De Schutter; published by CRC Press)*, pages 87–127, 2001.

[17] Steven J. Plimpton and Alex Slepoy. Microbial Cell Modeling via Reacting Diffusive Particles. *Journal of Physics: Conference Series 16*, pages 305–309, 2005.

[18] Steven J. Plimpton and Alex Slepoy. ChemCell: A Particle-Based Model of Protein Chemistry and Diffusion in Microbial Cells. *Sandia National Laboratories*, 2003.

[19] Steven S. Andrews and Dennis Bray. Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Physical Biology*, 1:137–151, 2004.

[20] Jeroen S. van Zon and Pieter Rein ten Wolde. Greens Function Reaction Dynamics: a new approach to simulate biochemical networks at the particle level and in time and space. *Journal of Chemical Physics*, 123(23):234910.1–234910.16, 2005.

[21] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2000.

[22] Bernard P. Zeigler. Systems Movement: Autobiographical Retrospectives. *International Journal of General Systems*, 32(3):221–236, 2003.

[23] Bernard P. Zeigler and Sankait Vahie. DEVS formalism and methodology: unity of conception/diversity of application. In *Proceedings of the Winter Simulation Conference (WSC)*, New York, NY, USA, 1993.

[24] Gabriel A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, 2009.

[25] Jean-Baptiste Filippi and Paul Bisgambiglia. General methodology 2: enabling large scale and high definition simulation of natural systems with vector models and JDEVS. In *Proceedings of the Winter Simulation Conference (WSC)*, San Diego, CA, USA, 2002.

[26] Fernando J. Barros. Abstract Simulators for the DSDE Formalism. In *Proceedings of the Winter Simulation Conference (WSC)*, Washington, DC, USA, 1998.

[27] Gabriel A. Wainer. Creating Advanced Fire-Spreading models using the CD++ toolkit. In *Proceedings of the 3rd Biennial meeting of the International Environmental Modelling and Software Society (iEMSs)*, Burlington, VT, USA, 2006.

[28] Xiaolin Hu, Alexandre Muzy, and Lewis Ntaimo. A Hybrid Cellular-Agent Space Modeling Approach for Fire Spread and Suppression Simulation. In *Proceedings of the Winter Simulation Conference (WSC)*, Orlando, FL, USA, 2005.

[29] Adelinde M. Uhrmacher and Corrado Priami. Discrete Event Systems Specification in Systems Biology - a Discussion of Stochastic Pi Calculus and DEVS. In *Proceedings of the Winter Simulation Conference (WSC)*, Orlando, FL, USA, 2005.

[30] Gabriel Wainer, Shafagh Jafer, Banan Al-Aubidy, Alex Dias, Roderick Bain, Michel Dumontier, and James Cheetham. Advanced DEVS models with application to biomedicine. In *Proceedings of the Artificial Intelligence, Simulation and Planning Conference (AIS)*, Buenos Aires, Argentina, 2007.

[31] Rhys Goldstein and Gabriel Wainer. Modelling Tumor-Immune Systems with Cell-DEVS. In *Proceedings of the European Conference on Modelling and Simulation (ECMS)*, Nicosia, Cyprus, 2008.

[32] Roland Ewald, Carsten Maus, Arndt Rolfs, and Adelinde M. Uhrmacher. Discrete Event Modelling and Simulation in Systems Biology. *Journal of Simulation*, 1(2):81–96, 2007.

[33] Adelinde M. Uhrmacher, Jan Himmelspach, Mathias Röhl, and Roland Ewald. Introducing Variable Ports and Multi-Couplings for Cell Biological Modeling in DEVS. In *Proceedings of the Winter Simulation Conference (WSC)*, Monterey, CA, USA, 2006.

[34] Adelinde M. Uhrmacher, Roland Ewald, Mathias John, Carsten Maus, Matthias Jeschke, and Susanne Biermann. Combining Micro and Macro-Modeling in DEVS for Computational Biology. In *Proceedings of the Winter Simulation Conference (WSC)*, Washington D.C., USA, 2007.

[35] Matthias Jeschke and Adelinde M. Uhrmacher. Multi-Resolution Spatial Simulation for Molecular Crowding. In *Proceedings of the Winter Simulation Conference (WSC)*, Miami, FL, USA, 2008.

[36] Adelinde M. Uhrmacher, Jan Himmelspach, Matthias Jeschke, Mathias John, Stefan Leye, Carsten Maus, Mathias Röhl, and Roland Ewald. One Modelling Formalism & Simulator Is Not Enough! A Perspective for Computational Biology Based on James II. In *Proceedings of the 1st International Workshop on Formal Methods in Systems Biology (FMSB)*, Cambridge, UK, 2008.

[37] Antoine Defontaine, Alfredo Hernández, and Guy Carrault. Multi-formalism Modelling of Cardiac Tissue. *Acta Biotheoretica*, 52:273–290, 2004.

[38] Spencer Borland and Hans Vangheluwe. Transforming Statecharts to DEVS. In *Proceedings of the Summer Computer Simulation Conference. Student Workshop*, Montreal, QC, Canada, 2003.

[39] Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-modelling. *Fundamental Approaches to Software Engineering*, pages 174–188, 2002.

[40] Kouichi Takahashi. *Multi-algorithm and multi-timescale cell biology simulation*. PhD, Keio University, Fujisawa, Japan, 2004.

[41] Brian Vincent Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD, University of California at Berkeley, Berkeley, CA, USA, 1996.

[42] David Baraff. Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. *Computer Graphics*, 23(3):223–232, 1989.

[43] Matthew Moore and Jane Wilhelms. Collision Detection and Response for Computer Animation. *Computer Graphics*, 22(4):289–298, 1988.

[44] Patricia Moore and Derek Molloy. A Survey of Computer-Based Deformable Models. In *Proceedings of the International Machine Vision and Image Processing Conference (IMVIP)*, Maynooth, Ireland, 2007.

[45] François Conti, Oussama Khatib, and Charles Baur. Interactive rendering of deformable objects based on a filling sphere modeling approach. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Taipei, Taiwan, 2003.

[46] Johan Jansson and Joris S. M. Vergeest. A discrete mechanics model for deformable bodies. *Computer-Aided Design*, 34(12):913–928, 2002.

[47] Sarah F. F. Gibson and Brian Mirtich. A Survey of Deformable Modeling in Computer Graphics. *Mitsubishi Electric Reasearch Laboratories (www.merl.com)*, 1997.

[48] Kazuyoshi Tagawa, Koichi Hirota, and Michitaka Hirose. Impulse Response Deformation Model: an Approach to Haptic Interaction with Dynamically Deformable Object. In *Proceedings of the Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS)*, Alexandria, VA, USA, 2006.

[49] Richard Keiser, Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Contact Handling for Deformable Point-Based Objects. In *Proceedings of the Vision, Modeling, and Visualization Conference (VMV)*, Stanford, CA, USA, 2004.

[50] Jan Bender and Daniel Bayer. Impulse-based simulation of inextensible cloth. In *Proceedings of The International Conference on Computer Graphics and Visualization (IADIS)*, Amsterdam, Netherlands, 2008.

[51] Raphael Diziol, Jan Bender, and Daniel Bayer. Volume Conserving Simulation of Deformable Bodies. In *Proceedings of Eurographics*, Munich, Germany, 2009.

[52] Thomas C. Südhof and Klaus Starke. *Pharmacology of Neurotransmitter Release*. Springer, 2008.

[53] William S. Trimble, Michal Linial, and Richard H. Scheller. Cellular and Molecular Biology of the Presynaptic Nerve Terminal. *Annual Review of Neuroscience*, 14:93–122, 1991.

[54] Pietro De Camilli. Keeping synapses up to speed. *Nature*, 375:450–451, 1995.

[55] Jay S. Coggan, Thomas M. Bartol, Eduardo Esquenazi, Joel R. Stiles, Stephan Lamont, Maryann E. Martone, Darwin K. Berg, Mark H. Ellisman, and Terrence J. Sejnowski. Evidence for Ectopic Neurotransmission at a Neuronal Synapse. *Science*, 309(5733):446–451, 2005.

[56] Rhys Goldstein, Gabriel Wainer, James J. Cheetham, and Roderick S. Bain. Vesicle-Synapsin Interactions Modeled with Cell-DEVS. In *Proceedings of the Winter Simulation Conference (WSC)*, Miami, FL, USA, 2008.

[57] Eric C. R. Hehner. from Boolean Algebra to Unified Algebra. *the Mathematical Intelligencer*, 26(2):3–19, 2004.

A Notation

A.1 Expressions

In general, we are likely to adopt a standard mathematical notation unless we have an alternative that eliminates ambiguity or reduces the number of necessary operators. A simple example of ambiguity is shown below.

$$f\,(x+y)$$
$$a\,(b+c)$$

All would agree that the first expression above represents the result of a function f applied to the sum of x and y. The second expression might appear to be the product of a and the sum of b and c, but this interpretation would be inconsistent with that of the first expression. To avoid ambiguity, we refrain from using juxtaposition to indicate multiplication, adopting instead a small dot as shown below.

$$a{\cdot}(b+c)$$

We also use parentheses in all cases when applying a function. We write $sin\,(x)$, for example, and avoid expressions like $(\sin x)$. Note that function application is left-associative.

$$g\,(x)\,(y) = (g\,(x))\,(y)$$

Eric C. R. Hehner demonstrates in [57] how the use of $\top$ for both infinity and the “true” boolean value, and the use of $\bot$ for both negative infinity and the “false” boolean value, can dramatically reduce the number of operators one needs to adopt. Below are examples of Hehner’s “unified algebra”.

$$(3<5)=\top$$
$$(3\geq 5)=\bot$$
$$\top+1=\top$$
$$2^{\bot}=0$$

An immediate benefit of unified algebra is the elimination of an extra operator like $\neg$ to negate a boolean. The numerical negation operator $-$ can be used instead.

$$-(3 < 5) = -\top = \bot$$

Hehner uses the symbols $\vee$ and $\wedge$ as infix operators that yield, respectively, the maximum and minimum of the two operands.

$$(3 \vee 5) = 5$$
$$(3 \wedge 5) = 3$$

By design, the new interpretation of $\vee$/$\wedge$ as maximum/minimum operators is consistent with their traditional interpretation as boolean OR/AND operators.

$$(\top \vee \bot) = \top$$
$$(\top \wedge \bot) = \bot$$
$$(\top \wedge \top) = \top$$
$$(\bot \wedge \bot) = \bot$$

The traditional mathematical notation that selects between candidate expressions has an unmatched brace on the left, followed by the candidate expressions themselves, followed on the right by the corresponding conditions. Adopting the arrangement more familiar to programmers, we place the conditions on the left. We also discard the brace, and enclose an entire selection expression in parentheses as demonstrated below.

$$sgn\,(5) = \left(\begin{array}{ll} 5 < 0 & \rightarrow -1 \\ 5 = 0 & \rightarrow 0 \\ 5 > 0 & \rightarrow 1 \end{array}\right) = 1$$

If none of the conditions are true, the value of the selection expression is undefined.

We use the function composition operator $\circ$ in the usual way.

$$(sin \circ cos)\,(x) = sin\,(cos\,(x))$$

We also permit functions to be composed using various other operations, given that at least one of the operands is a function itself. Several examples are below, some of which reflect common practice and some of which are unusual.

$$\begin{aligned}
&(sin + cos)(x) = sin(x) + cos(x)\\
&(sin + 5)(x) = sin(x) + 5\\
&(1 - cos)(x) = 1 - cos(x)\\
&(2 \cdot cos)(x) = 2 \cdot cos(x)\\
&(sin \cdot cos)(x) = sin(x) \cdot cos(x)\\
&sin^2(x) = (sin(x))^2 = sin(x)^2\\
&\sqrt{sin}(x) = \sqrt{sin(x)}\\
&(sin < cos)(x) = (sin(x) < cos(x))\\
&(sin = cos)(x) = (sin(x) = cos(x))\\
&(sin \wedge cos)(x) = (sin(x) \wedge cos(x))\\
&(sin \vee 0)(x) = (sin(x) \vee 0)
\end{aligned}$$

We use braces $\{\ldots\}$ to enclose comments, and avoid the use of sets. In cases where one would normally use a set, we would use a predicate, a function that results in either $\top$ or $\bot$. The predicate $\mathbb{N}$ results in $\top$ if and only if its argument is a natural number. Similarly, $\mathbb{F}(x)$ is true if and only if x is a function.

$$\begin{array}{ll}
\mathbb{N}(3) & \{\text{this is true } (\top)\}\\
\mathbb{N}\left(\frac{3}{2}\right) & \{\text{this is false } (\bot)\}\\
\mathbb{F}(3) & \{\text{this is false } (\bot)\}\\
\mathbb{F}(sin) & \{\text{this is true } (\top)\}
\end{array}$$

For any set A, one can define a predicate p_A such that $x \in A$ is the same as $p_A(x)$. We exploit function composition to allow operations on predicates that are analogous to conventional set operations.

$$\begin{array}{ll}
(p_A \vee p_B) & \{\text{analogous to } A \cup B\}\\
(p_A \wedge p_B) & \{\text{analogous to } A \cap B\}
\end{array}$$

We use symbols enclosed in double quotes "..." as values. We also use the symbol $\varnothing$ as some value generally meaning “nil” or “not a number”.

$$\begin{pmatrix} \mathbb{N}(x) & \rightarrow \text{“}x\_is\_a\_positive\_integer\text{”} \\ x = \varnothing & \rightarrow \text{“}x\_is\_\varnothing\text{”} \end{pmatrix}$$

A.2 Definitions

Definitions associate values with variables explicitly using the assignment symbol $:=$. Consistent with mathematical convention, once a variable is assigned a value in a particular context, it may not be assigned a new value. Unless indicated by the surrounding text, by indentation, or by a scope identifier (explained further below), the scope of a defined variable like $\Delta Q_0{}'$ is generally this entire document.

$$\Delta Q_0{}' := 77$$

Functions are typically defined with arguments indicated in parentheses to the left of the assignment symbol.

$$sgn(x) := \begin{pmatrix} x < 0 & \rightarrow -1 \\ x = 0 & \rightarrow 0 \\ x > 0 & \rightarrow 1 \end{pmatrix}$$

It is important to remember that we treat functions as values, passing them as arguments or assigning them to variables directly. The following two definitions of *cot* are equivalent, for example. In the second case, *cot* is defined in the same manner as a scalar variable, but the right-hand side happens to represent a function.

$$cot(x) := \frac{1}{tan(x)}$$

$$cot := \frac{1}{tan}$$

Formally, we allow functions to take only a single argument. To pass multiple values into a function, we generally make the sole argument a vector. Informally, when a function takes as its argument a vector of fixed length, we refer to the elements of the vector as "arguments". An example is the mod function, which takes as its sole formal argument a vector of the form $[x, y]$. We refer to the numbers x and y as arguments. The result of $mod\left([x, y]\right)$ is the remainder of x when divided by y.

We use recursion to describe iterative processes. Although variables cannot be re-assigned values, the arguments of a recursive function may take on different values each time the function is invoked. Below is the recursive function GCD, which yields the greatest common divisor of the natural numbers m and n.

$$GCD\left([m, n]\right) := \begin{pmatrix} m = n & \rightarrow m \\ m < n & \rightarrow GCD\left([m, mod\left([n, m]\right)]\right) \\ m > n & \rightarrow GCD\left([mod\left([m, n]\right), n]\right) \end{pmatrix}$$

In order to control the scope of each variable, we arrange all definitions in a hierarchy. In the example below, the variables AAA, BBB, are CCC are all defined at the outermost level of the hierarchy. Each defined variable may have one or more "sub-variables" defined in an indented position underneath it. Sub-variables of AAA include α and β, for example, and AAA is can be described as the "parent variable" of α and β.

$$\begin{array}{ll}
AAA := \alpha + \beta & \{\text{in scope} : AAA, \alpha, \beta, BBB, CCC\} \\
\quad \alpha := 4 & \{\text{in scope} : AAA, \alpha, \beta, BBB, CCC\} \\
\quad \beta := CCC - 7 & \{\text{in scope} : AAA, \alpha, \beta, BBB, CCC\} \\
 & \\
BBB\left([x, y]\right) := z & \{\text{in scope} : AAA, BBB, x, y, f, w, z, CCC\} \\
\quad f\left(t\right) := AAA + CCC + t & \{\text{in scope} : AAA, BBB, x, y, f, t, w, z, CCC\} \\
\quad w := 2 \cdot \gamma + y & \{\text{in scope} : AAA, BBB, x, y, f, w, \gamma, z, CCC\} \\
\qquad \gamma := f\left(x\right) & \{\text{in scope} : AAA, BBB, x, y, f, w, \gamma, z, CCC\} \\
\quad z := w \cdot f\left(\epsilon\right) & \{\text{in scope} : AAA, BBB, x, y, f, w, \epsilon, \kappa, z, CCC\} \\
\qquad \epsilon := 0.1 \cdot \kappa & \{\text{in scope} : AAA, BBB, x, y, f, w, \epsilon, \kappa, z, CCC\} \\
\qquad \kappa := \sqrt{CCC} & \{\text{in scope} : AAA, BBB, x, y, f, w, \epsilon, \kappa, z, CCC\} \\
 & \\
CCC := 10 & \{\text{in scope} : AAA, BBB, CCC\}
\end{array}$$

The variables indicated as "in scope", in any of the comments above, can be used on the right-hand side of the corresponding assignment symbol. Note that all sub-variables with a common parent variable can be used to define that parent variable, one another, and sub-variables of one another. Similarly, function arguments are in scope on the right-hand side of the definition in which they are introduced, and on the right-hand side of all further indented definitions underneath.

Because definitions can be nested deeply within one another, indentation is not always practical. We therefore adopt "scope identifiers", variable names enclosed in angle brackets $\ll \ldots \gg$, to define these hierarchies. As an example, we give an alternative presentation of AAA, BBB, and CCC equivalent to the one above.

Because AAA has a global scope, we need not include a scope identifier.

$$AAA := \alpha + \beta$$

The scope identifier below precedes the definitions of α and β to indicate that they are sub-variables of AAA.

$\ll AAA \gg$

$$\alpha := 4$$
$$\beta := CCC - 7$$

The absence of a scope identifier indicates that BBB has a global scope.

$$BBB\left([x, y]\right) := z$$

We now define f, indicating that it is a sub-variable of BBB.

$\ll BBB \gg$

$$f\left(t\right) := AAA + CCC + t$$

An ellipsis (. . .) in a scope identifier represents the text inside the preceding scope identifier. In this case, that text happens to be BBB.

$\ll \ldots \gg$

$$w := 2{\cdot}\gamma + y$$

Semicolons delimit variables and sub-variables in a scope identifier. Below, γ is defined as a sub-variable of w, which itself is a sub-variable of BBB.

$\ll BBB; w \gg$

$$\gamma := f(x)$$

Below we define z, a subvariable of BBB.

$\ll BBB \gg$

$$z := w{\cdot}f(\epsilon)$$

To interpret the scope identifier below, we substitute the text of preceding scope identifier, which happens to be BBB, for the ellipsis.

$\ll \ldots; z \gg$

$$\epsilon := 0.1{\cdot}\kappa$$
$$\kappa := \sqrt{CCC}$$

Note that we can combine scope identifiers with indentation. We could, for example, have replaced the definitions of z, γ, and κ above with the following.

$\ll BBB \gg$

$$\begin{aligned}&z := w{\cdot}f(\epsilon)\\ &\qquad \epsilon := 0.1{\cdot}\kappa\\ &\qquad \kappa := \sqrt{CCC}\end{aligned}$$

We define CCC with global scope to complete the example.

$$CCC := 10$$

A.3 Selectors

We use the term “selector” to refer to a function with a domain that consists of a finite number of discrete points. It can be viewed as an abstract version of an “associate array” or “dictionary” data structure, a common programming language feature.

The variable A is assigned a selector below. Domain values appear on the left of the arrows, whereas the corresponding elements appear on the right. Note that all definitions in this section pertain only to this section.

$$A := \begin{bmatrix} \text{“}abc\text{”} & \rightarrow 0.5 \\ -99 & \rightarrow \varnothing \\ \text{“}def\text{”} & \rightarrow 0 \end{bmatrix}$$

The selector A is a function ($\mathbb{F}(A) = \top$) that maps the values “abs”, -99, and “def” to the values 0.5, $\varnothing$, and 0.

$$A(\text{“}abc\text{”}) = 0.5$$
$$A(-99) = \varnothing$$
$$A(\text{“}def\text{”}) = 0$$

The same composition rules that apply to functions apply also to selectors. The expression $A = 0$ is itself a selector, as the equals sign operates on each element. To compare entire selectors, we use the equivalence symbol $\equiv$. (We also permit the comparison of scalar values with $\equiv$.)

$$(A = 0) \equiv \begin{bmatrix} \text{“}abc\text{”} & \rightarrow 0.5 = 0 \\ -99 & \rightarrow \varnothing = 0 \\ \text{“}def\text{”} & \rightarrow 0 = 0 \end{bmatrix} \equiv \begin{bmatrix} \text{“}abc\text{”} & \rightarrow \bot \\ -99 & \rightarrow \bot \\ \text{“}def\text{”} & \rightarrow \top \end{bmatrix}$$

A “vector” is a selector with a domain that consists of all of natural numbers less than some natural number n, where n is the “length” of the vector. When expressing a vector, we may omit the domain and the arrows, including just the elements surrounded by brackets. We arrange elements either horizontally, in which case they are delimited by commas, or vertically, in which case no delimiter is necessary.

$$[6.2, 0.3, -1.9] \equiv \begin{bmatrix} 6.2 \\ 0.3 \\ -1.9 \end{bmatrix} \equiv \begin{bmatrix} 0 & \rightarrow 6.2 \\ 1 & \rightarrow 0.3 \\ 2 & \rightarrow -1.9 \end{bmatrix}$$

The prefix operator $\Diamond$, when applied to a selector, results in a vector that lists each value in the selector’s domain.

$$\Diamond A \equiv [\text{“}abs\text{”}, -99, \text{“}def\text{”}]$$

Using the composition operator $\circ$ between a selector and a vector of values in its domain, we can acquire a vector containing the corresponding selector elements. This is not extension of $\circ$, but a logical consequence of its usual interpretation.

$$A \circ [\text{“}def\text{”}, -99] \;\equiv\; [0, \varnothing]$$

We use the operator $\lhd$ to combine two selectors. If both selectors have different elements for the same domain value, then the element of the selector to the right of $\lhd$ is chosen. The expression $A \lhd B$ means “take the selector A, and add or replace its mappings with those of B”. Of course A is not modified, since we do not allow changes to variables, but the combined selector is the result of the expression.

$$A \lhd \begin{bmatrix} \text{“}abc\text{”} & \rightarrow \text{“}Q\text{”} \\ -99 & \rightarrow \bot \\ 1000 & \rightarrow \text{“}R\text{”} \end{bmatrix} \;\equiv\; \begin{bmatrix} \text{“}abc\text{”} & \rightarrow \text{“}Q\text{”} \\ -99 & \rightarrow \bot \\ \text{“}def\text{”} & \rightarrow 0 \\ 1000 & \rightarrow \text{“}R\text{”} \end{bmatrix}$$

The following demonstrates the removal of selector elements. The domain values indicating the mappings to exclude are listed in a vector.

$$A \ntriangleleft [-99, \text{“}def\text{”}] \;\equiv\; \begin{bmatrix} \text{“}abc\text{”} & \rightarrow 0.5 \end{bmatrix}$$

The expression $..n$ represents a length n vector with values increasing from 0 to $n-1$. As indicated below, this happens to be an identify function with a restricted domain.

$$..5 \;\equiv\; [0,1,2,3,4] \;\equiv\; \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \;\equiv\; \begin{bmatrix} 0 & \rightarrow 0 \\ 1 & \rightarrow 1 \\ 2 & \rightarrow 2 \\ 3 & \rightarrow 3 \\ 4 & \rightarrow 4 \end{bmatrix}$$

Being functions, vectors can be composed using operators. We exploit this below to express an arithmetic sequence.

$$3 + 2\cdot..5 \;\equiv\; [3, 5, 7, 9, 12]$$

The $\|$ operator concatenates vectors.

$$\begin{aligned}
&[3,5,7] \,\|\, [9,12] \;\equiv\; [3,5,7,9,12] \\
&[3,5,7] \,\|\, [9,12] \,\|\, [4,2] \;\equiv\; [3,5,7,9,12,4,2]
\end{aligned}$$

The symbol $\#$, when applied to a vector as a prefix operation, results in the vector's length.

$$\begin{aligned}
&\#\,[3,5,7,9,12] = 5 \\
&\#\,[\,] = 0
\end{aligned}$$

The function del removes an element from a vector. The expression $del\,([X,i])$ removes the element at index i.

$$del\,([[3,5,7,9,12,4,2]\,,3]) \;\equiv\; [3,5,7,12,4,2]$$

Suppose X and Y are vectors that, individually, have no duplicate values. Then the expression $union\,([X,Y])$ results in a vector similar to $X \,\|\, Y$, except with all values common to both X and Y included only once.

$$union\,([[3,5,7,9]\,,[4,5,8,9]]) \;\equiv\; [3,5,7,9,4,8]$$

The prefix operators $\sum$, $\bigwedge$, and $\bigvee$ can be applied to selectors or vectors to yield the sum, minimum, or maximum of the elements.

$$\begin{aligned}
&\textstyle\sum\,[3,7,5] = 15 \\
&\textstyle\bigwedge\,[3,7,5] = 3 \\
&\textstyle\bigvee\,[3,7,5] = 7 \\
&\textstyle\bigvee\,([3,7,5] = 5) = \bigvee\,[\bot,\bot,\top] = \top
\end{aligned}$$

The $\|$ symbol can be applied as a prefix operator to concatenate all vectors in a vector of vectors.

$$\|\;[[3,7,5]\,,[0]\,,[2,4]] \;\equiv\; [3,7,5,0,2,4]$$

An expression of the form $x : X$ yields $\top$ if any element of X is x, and $\bot$ otherwise. The expression below, for example, is true.

$$7 : [3, 7, 5]$$

If a function, like B below, is defined using a selection expression with a single condition of the form $x : X$, then the function is a selector and X is the domain.

$$B\,(x) := \left(\begin{array}{ll} x : [3, 7, 5] & \to x^2 \end{array}\right)$$

$$B \;\equiv\; \left[\begin{array}{ll} 3 & \to 9 \\ 7 & \to 49 \\ 5 & \to 25 \end{array}\right]$$

A selector can also be formed using multiple conditions of the form $x : X$.

$$C\,(x) := \left(\begin{array}{ll} x : [3, 7] & \to x^2 \\ x : [5, 2] & \to x^3 \end{array}\right)$$

$$C \;\equiv\; \left[\begin{array}{ll} 3 & \to 9 \\ 7 & \to 49 \\ 5 & \to 125 \\ 2 & \to 8 \end{array}\right]$$

The function $sort$ results in a non-decreasing permutation of its vector argument.

$$sort\,([6, 3, 7, 5]) \;\equiv\; [3, 5, 6, 7]$$

The function cmp is defined as follows, yielding -1, 0, or 1 depending on whether the first argument is smaller, equal to, or larger than the second.

$$cmp\,([x, y]) := sgn\,(x - y)$$

The function $sort_{cmp}$ is similar to $sort$, but uses its second argument to compare pairs of elements of the first argument.

$$\begin{aligned} sort_{cmp}\left(\left[[6,3,7,5], cmp\right]\right) &\equiv [3,5,6,7] \\ sort_{cmp}\left(\left[[6,3,7,5], -cmp\right]\right) &\equiv [7,6,5,3] \end{aligned}$$

In addition to vectors, we observe a special type of selector called a "multi-dimensional array". Recall that if n is a natural number, then $..n$ is an identity function with a domain consisting of natural numbers. If N is a vector of natural numbers, then $..N$ is an identity function with a domain consisting of vectors of natural numbers. The elements of the domain vectors of $..N$ are all less than the corresponding elements of N.

Below, as an example, we define the multi-dimensional array D.

$$D := [10, -5] + 5 \cdot .. [2, 3]$$

The domain of D includes 6 vectors (2·3).

$$\Diamond D \equiv \begin{bmatrix} [0,0] \\ [0,1] \\ [0,2] \\ [1,0] \\ [1,1] \\ [1,2] \end{bmatrix}$$

An element corresponding to a domain vector X has a value of $[10, -5] + 5 \cdot X$, consistent with the definition of D.

$$\begin{aligned} D([0,0]) &\equiv [10,-5] \\ D([0,1]) &\equiv [10,0] \\ D([0,2]) &\equiv [10,5] \\ D([1,0]) &\equiv [15,-5] \\ D([1,1]) &\equiv [15,0] \\ D([1,2]) &\equiv [15,5] \end{aligned}$$

A.4 Probability

Instead of using random variables, we use "random functions". A random function is invoked with no argument to yield a value randomly selected from the distribution associated with the function.

The functions $uniform$, $uniform_{\mathbb{N}}$, and $exponential$ are not random functions. Rather, they are invoked with arguments in order to obtain random functions. Each of the three entire expressions below represents a random function.

$$\begin{array}{l} uniform\left([a,b]\right) \\ uniform_{\mathbb{N}}\left(n\right) \\ exponential\left(\tau\right) \end{array}$$

The following three expressions, which include the suffix $()$, represent values selected randomly from a uniform distribution bounded by a and b, a uniformly-distributed set of natural numbers bounded by 0 and $n-1$, and an exponential distribution with a mean value of τ.

$$\begin{array}{ll} uniform\left([a,b]\right)() & \{\text{random real number between } a \text{ and } b\} \\ uniform_{\mathbb{N}}\left(n\right)() & \{\text{random natural number less than } n\} \\ exponential\left(\tau\right)() & \{\text{random real number with mean value } \tau\} \end{array}$$

Random functions can be composed from operations on other random functions, as demonstrated by the third and fourth expressions below. Note that a "random predicate" is a random function that yields either $\top$ or $\bot$.

$$\begin{array}{ll} uniform_{\mathbb{N}}\left(2\right) & \{\text{random function}\} \\ uniform_{\mathbb{N}}\left(2\right)() & \{\text{random natural number } (0 \text{ or } 1)\} \\ uniform_{\mathbb{N}}\left(2\right)=1 & \{\text{random predicate}\} \\ \left(uniform_{\mathbb{N}}\left(2\right)=1\right)() & \{\text{random boolean } (\top \text{ or } \bot)\} \end{array}$$

The function P, when applied to a random predicate, yields the probability that the random predicate will yield $\top$ as opposed to $\bot$. With this definition, our use of P with random functions is similar to the traditional use of P with random variables.

$$P\left(uniform_{\mathbb{N}}\left(2\right)=1\right)=0.5$$

Below, as an example, we calculate the probability that one of the chemical reactions described in Section 5.3 will occur during an action potential of duration Δt_{AP}. Recall that these reactions include the separation of vesicles from synapsin or docking sites. The time when one of these reactions occurs for a given particle is sampled from an exponential distribution with mean value τ.

$$\begin{aligned}
&P\left(exponential\left(\tau\right)\leq\Delta t_{AP}\right)\\
&\quad=\int_{0}^{\Delta t_{AP}}\left(\frac{1}{\tau}\cdot e^{-\frac{t}{\tau}}\right)dt\\
&\quad=\left(-e^{-\frac{t}{\tau}}\right)\Big|_{t=0}^{t=\Delta t_{AP}}\\
&\quad=-e^{-\frac{\Delta t_{AP}}{\tau}}-\left(-e^{-\frac{0}{\tau}}\right)\\
&\quad=1-e^{-\frac{\Delta t_{AP}}{\tau}}
\end{aligned}$$

B Tethered Particle System Formulas

B.1 TPS Functions

The selector $tethered\_particle\_system$ provides access to several key functions describing particle interactions in a TPS model. The comments in its definition below indicate the primary result of each function, as well as the section in which each function is defined.

$$tethered\_particle\_system := \ldots$$
$$\begin{bmatrix} \text{``}position\text{''} & \rightarrow position \\ & \quad \{\text{particle position (Section B.1)}\} \\ \text{``}detect\text{''} & \rightarrow detect \\ & \quad \{\text{collision time (Section B.2)}\} \\ \text{``}impulse\text{''} & \rightarrow impulse \\ & \quad \{\text{collision impulse (Section B.3)}\} \\ \text{``}load\text{''} & \rightarrow load \\ & \quad \{\text{state after loading (Section B.4)}\} \\ \text{``}restitute\text{''} & \rightarrow restitute \\ & \quad \{\text{state after restitution (Section B.4)}\} \\ \text{``}detect_{RI}\text{''} & \rightarrow detect_{RI} \\ & \quad \{\text{random impulse time (Section B.5)}\} \\ \text{``}impulse_{RI}\text{''} & \rightarrow impulse_{RI} \\ & \quad \{\text{random impulse (Section B.5)}\} \\ \text{``}impact\text{''} & \rightarrow impact \\ & \quad \{\text{state after an impulse (Section B.5)}\} \end{bmatrix}$$

Although we are not concerned with DEVS in this appendix, we will allow ourselves to be somewhat influenced by the formalism's convention for representing system states. The system state of a TPS includes, among other things, the current position of each particle. We represent the system state with two variables: Ψ, the constant component of the state, and t, the current time. As previously explained, the system state of a DEVS includes the time elapsed since the last event instead of the current time, but we will store enough information in Ψ to calculate one from the other. The following expressions list the attributes recorded in Ψ for a particle A identified by id_A.

$$\begin{array}{ll}
\Psi\left(id_A\right)\left(\text{“}spc\text{”}\right) & \{\text{species of particle A}\} \\
\Psi\left(id_A\right)\left(\text{“}t\text{”}\right) & \{\text{time when particle A obtained its velocity}\} \\
\Psi\left(id_A\right)\left(\text{“}\vec{u}\text{”}\right) & \{\text{position where particle A obtained its velocity}\} \\
\Psi\left(id_A\right)\left(\text{“}\vec{v}\text{”}\right) & \{\text{velocity of particle A}\} \\
\Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right) & \{\text{IDs of particles tethered to particle A}\} \\
\Psi\left(id_A\right)\left(\text{“}loaded\text{”}\right) & \{\text{restitution impulses associated with particle A}\}
\end{array}$$

The attribute associated with “spc” is a symbol used to reference other particle properties. The “t” attribute is a real number, and those associated with “$\vec{u}$” and “$\vec{v}$” are vectors of real numbers. The “$tethered$” attribute is a vector of particle IDs. The equation below indicates that the particle with ID 7 is tethered to those with IDs 2, 9, 8, and 5.

$$\Psi\left(7\right)\left(\text{“}tethered\text{”}\right) \equiv [2, 9, 8, 5]$$

The “$loaded$” attribute is a selector that maps particle IDs to momentum vectors. From the following equation we would infer that the particles identified by 7 and 2 are loaded with an associated restitution impulse of $\vec{p}_{[7,2]}$, and that the particles with IDs 7 and 6 are loaded with $\vec{p}_{[7,6]}$. Restitution impulses are explained in Section B.4.

$$\Psi\left(7\right)\left(\text{“}loaded\text{”}\right) \equiv \left[2 \to \vec{p}_{[7,2]}, 6 \to \vec{p}_{[7,6]}\right]$$

We now define the first of the $tethered\_particle\_system$ functions. Given Ψ, which records the position $\vec{u}_A$ of particle identified by id_A at some time t_A, $position$ yields the position $\vec{u}_A{}'$ of A at the current time t. This is accomplished by simply adding to $\vec{u}_A$ the product of the particle's velocity $\vec{v}_A$ and the elapsed time $t - t_A$.

$\ll$ *tethered_particle_system* $\gg$

$$\begin{aligned}
&position\left([\Psi, t, id_A]\right) := \vec{u}_A{}' \\
&\quad t_A := \Psi\left(id_A\right)(\text{“}t\text{”}) \\
&\quad \vec{u}_A := \Psi\left(id_A\right)(\text{“}\vec{u}_A\text{”}) \\
&\quad \vec{v}_A := \Psi\left(id_A\right)(\text{“}\vec{v}_A\text{”}) \\
&\quad \vec{u}_A{}' := \vec{u}_A + \vec{v}_A \cdot (t - t_A)
\end{aligned}$$

Many of the other *tethered_particle_system* functions require the properties associated with the species of each particle. These properties, listed below for a species identified by spc_A, are included in the selector Ω_ψ.

$\Omega_\psi\left(spc_A\right)(\text{“}m\text{”})$	{mass of a particle of the species identified by $\mathrm{src_A}$}
$\Omega_\psi\left(spc_A\right)(\text{“}\tau_{RI}\text{”})$	{mean time between random impulses}
$\Omega_\psi\left(spc_A\right)(\text{“}k_{RI}\text{”})$	{shape of random impulse probability distribution}
$\Omega_\psi\left(spc_A\right)(\text{“}\mu_{RI}\text{”})$	{mean magnitude of random impulse }

Other properies, stored in the selector $\Omega_{\psi\psi}$, are associated with pairs of species. One example, included in the list below, is the blocking distance $\Delta u_{blocking}$ between one particle of species A and another of species B.

$\Omega_{\psi\psi}\left([spc_A, spc_B]\right)(\text{“}c_{rebound}\text{”})$	{rebounding coefficient}
$\Omega_{\psi\psi}\left([spc_A, spc_B]\right)(\text{“}c_{retract}\text{”})$	{retraction coefficient}
$\Omega_{\psi\psi}\left([spc_A, spc_B]\right)(\text{“}c_{revolve}\text{”})$	{revolution coefficient}
$\Omega_{\psi\psi}\left([spc_A, spc_B]\right)(\text{“}\Delta u_{blocking}\text{”})$	{blocking distance}
$\Omega_{\psi\psi}\left([spc_A, spc_B]\right)(\text{“}\Delta u_{tethering}\text{”})$	{tethering distance}

When accessing the properties of $\Omega_{\psi\psi}$, the order of the species IDs spc_A and spc_B is irrelevant. We require the following to hold for all $[spc_A, spc_B]$ in the domain of $\Omega_{\psi\psi}$.

$$\Omega_{\psi\psi}\left([spc_A, spc_B]\right) \equiv \Omega_{\psi\psi}\left([spc_B, spc_A]\right)$$

B.2 Collision Detection

Collision detection is the task of identifying the next collision, the time when that collision will occur, and the two particles involves. A simple way to accomplish this is to calculate the collision time for each pair of particles in the model. One could then select the earliest time from the list.

Here we derive the $tethered\_particle\_system$ function $detect$, which yields the remaining time Δt before particles A and B collide with one another. If the particles are not going to collide, given their current positions and velocities, then $\Delta t = \top$. For blocking collisions, calculations similar to those below have undoubtedly been formulated many times over. For tethering collisions, the presented formulas are likely novel.

Given the current position $\vec{u}_A{}'$ and the velocity $\vec{v}_A$ of particle A, its extrapolated position after a time of Δt is $\vec{u}_A{}' + \vec{v}_A \cdot \Delta t$. Subtracting this from the extrapolated position of particle B, then taking the magnitude of the result, we obtain the distance Δu between the particles after the time Δt elapses. The following shows expressions for Δu^2 manipulated.

$$\begin{aligned}
&\Delta u^2 \ldots \\
&\quad = \sum\left(((\vec{u}_B{}' + \vec{v}_B \cdot \Delta t) - (\vec{u}_A{}' + \vec{v}_A \cdot \Delta t))^2\right) \ldots \\
&\quad = \sum\left(((\vec{v}_B - \vec{v}_A) \cdot \Delta t + (\vec{u}_B{}' - \vec{u}_A{}'))^2\right) \ldots \\
&\quad = \sum\left((\vec{v}_B - \vec{v}_A)^2 \cdot \Delta t^2 + 2 \cdot (\vec{u}_B{}' - \vec{u}_A{}') \cdot (\vec{v}_B - \vec{v}_A) \cdot \Delta t + (\vec{u}_B{}' - \vec{u}_A{}')^2\right) \ldots \\
&\quad = \sum\left((\vec{v}_B - \vec{v}_A)^2\right) \cdot \Delta t^2 + 2 \cdot \sum\left((\vec{u}_B{}' - \vec{u}_A{}') \cdot (\vec{v}_B - \vec{v}_A)\right) \cdot \Delta t + \sum\left((\vec{u}_B{}' - \vec{u}_A{}')^2\right)
\end{aligned}$$

We are interested in the time before a collision. Because collisions occur when the distance Δu between two particles reaches a known $\Delta u_{blocking}$ or $\Delta u_{tethering}$ value, the only unknown is Δt. Rearranging the above yields (6), a quadratic polynomial.

$$\begin{aligned}
0 = \ldots & \\
&\sum\left((\vec{v}_B - \vec{v}_A)^2\right) \cdot \Delta t^2 + \ldots \\
&2 \cdot \sum\left((\vec{u}_B{}' - \vec{u}_A{}') \cdot (\vec{v}_B - \vec{v}_A)\right) \cdot \Delta t + \ldots \\
&\sum\left((\vec{u}_B{}' - \vec{u}_A{}')^2\right) - \Delta u^2
\end{aligned} \tag{6}$$

Introducing the coefficients a, b, and c, (6) can be written as follows.

$$a{\cdot}\Delta t^2 + b{\cdot}\Delta t + c = 0$$

We isolate Δt with the quadratic equation.

$$\Delta t = \frac{-b \pm \sqrt{b^2 - 4{\cdot}a{\cdot}c}}{2{\cdot}a} = \frac{-b \pm \sqrt{d}}{2{\cdot}a}$$

The function coeffs_{detect} gives the detection coefficients a, b, and c, as well as the discriminant d.

$\ll$ *tethered_particle_system* $\gg$

```
$\mathit{coeffs}_{detect}\left([\Psi, t, id_A, id_B, \Delta u]\right) := [a, b, c, d]$
    $\vec{u}_A{}' := position\left([\Psi, t, id_A]\right)$
    $\vec{u}_B{}' := position\left([\Psi, t, id_B]\right)$
    $\vec{v}_A := \Psi\left(id_A\right)\left(\text{“}\vec{v}\text{”}\right)$
    $\vec{v}_B := \Psi\left(id_B\right)\left(\text{“}\vec{v}\text{”}\right)$
    $a := \sum\left(\left(\vec{v}_B - \vec{v}_A\right)^2\right)$
    $b := 2{\cdot}\sum\left(\left(\vec{u}_B{}' - \vec{u}_A{}'\right)\cdot\left(\vec{v}_B - \vec{v}_A\right)\right)$
    $c := \sum\left(\left(\vec{u}_B{}' - \vec{u}_A{}'\right)^2\right) - \Delta u^2$
    $d := b^2 - 4{\cdot}a{\cdot}c$
```

Although we now have an expression for Δt, there are many cases that need to considered in pursuit of its value. We begin by defining $detect_{blocking}$, which results in the time remaining before a blocking collision occurs between two particles. Arguments include the system state variables Ψ and t, the particle IDs id_A and id_B, $\Omega_{\psi\psi}$, and Δt_{max}. This last parameter is some time value that should dramatically exceed the simulated duration of any reasonable simulation run. We introduce Δt_{max} only to avoid computation errors associated with small denominators.

$\ll$ *tethered_particle_system* $\gg$

$$\begin{aligned} detect_{blocking}\left([\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}]\right) &:= \Delta t \\ \Delta u &:= \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{“}\Delta u_{blocking}\text{”}\right) \\ spc_A &:= \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right) \\ spc_B &:= \Psi\left(id_B\right)\left(\text{“}spc\text{”}\right) \end{aligned}$$

Observe that Δu is obtained from the blocking distance associated with the species of the two particles. If this distance is 0, blocking collisions never occur and hence $\Delta t = \top$.

$\ll \ldots; detect_{blocking} \gg$

$$\Delta t := \left(\begin{array}{ll} \Delta u = 0 & \rightarrow \top \\ \Delta u > 0 & \rightarrow \Delta t_{coeffs} \end{array}\right)$$

Proceeding with a non-zero blocking distance, we now need a, b, c, and d. Inspecting the definition of b, we can see that $b \geq 0$ indicates the particles are not approaching one another; consequently, a blocking collision will never occur. If $d < 0$, the absence of real solutions to the quadratic equation indicates that the particles never cross paths.

$\ll \ldots; \Delta t \gg$

$$\begin{aligned} \Delta t_{coeffs} &:= \left(\begin{array}{ll} (b \geq 0) \vee (d < 0) & \rightarrow \top \\ (b < 0) \wedge (d \geq 0) & \rightarrow \Delta t_{quad} \end{array}\right) \\ [a, b, c, d] &:= coeffs_{detect}\left([\Psi, t, id_A, id_B, \Delta u]\right) \end{aligned}$$

Proceeding with approaching particles and a non-negative d, the quadratic equation may be applied. We must select the negative sign for the $\sqrt{d}$ term in the numerator, as we desire the first of the two future collision times. Negative numerators, which may occur due to round-off errors, are raised to 0. Also, Δt_{max} is used to avoid divide-by-zero errors.

$\ll \ldots; \Delta t_{coeffs} \gg$

$$\begin{aligned} \Delta t_{quad} &:= \left(\begin{array}{ll} numer \geq denom \cdot \Delta t_{max} & \rightarrow \top \\ numer < denom \cdot \Delta t_{max} & \rightarrow \frac{numer}{denom} \end{array}\right) \\ numer &:= \left(-b - \sqrt{d}\right) \vee 0 \\ denom &:= 2 \cdot a \end{aligned}$$

We now turn out attention to tethering collisions, and define $detect_{tethering}$. The variable Δu is now assigned the tethering distance associated with the species of the two particles.

$\ll$ *tethered_particle_system* $\gg$

$$\begin{aligned}
detect_{tethering}\left([\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}]\right) &:= \Delta t \\
\Delta u &:= \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{“}\Delta u_{tethering}\text{”}\right) \\
spc_A &:= \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right) \\
spc_B &:= \Psi\left(id_B\right)\left(\text{“}spc\text{”}\right)
\end{aligned}$$

An infinite tethering distance indicates that tethering collisions never occur.

$\ll \ldots; detect_{tethering} \gg$

$$\Delta t := \begin{pmatrix} \Delta u = \top & \rightarrow \top \\ \Delta u < \top & \rightarrow \Delta t_{coeffs} \end{pmatrix}$$

Proceeding with a finite tethering distance, we obtain a, b, c, and d and consider two cases. If the particles are not approaching ($b \geq 0$) and the distance between particles is at least the tethering distance already ($c \geq 0$), then we evaluate Δt_{taut}. Otherwise we evaluate Δt_{quad}.

$\ll \ldots; \Delta t \gg$

$$\begin{aligned}
\Delta t_{coeffs} &:= \begin{pmatrix} (b \geq 0) \wedge (c \geq 0) & \rightarrow \Delta t_{taut} \\ (b < 0) \vee (c < 0) & \rightarrow \Delta t_{quad} \end{pmatrix} \\
[a, b, c, d] &:= coeffs_{detect}\left([\Psi, t, id_A, id_B, \Delta u]\right)
\end{aligned}$$

In the case associated with Δt_{taut}, a tethering collision ought to occur immediately. However, the relative speed of the particles, indicated by a, might be so small that the collision ought to be avoided completely.

$\ll \ldots; \Delta t_{coeffs} \gg$

$$\Delta t_{taut} := \begin{pmatrix} b \geq a \cdot \Delta t_{max} & \rightarrow \top \\ b < a \cdot \Delta t_{max} & \rightarrow 0 \end{pmatrix}$$

In the case associated with Δt_{quad}, the quadratic equation may be applied. As was the case for blocking collisions, a selection expression addresses the possibility of a divide-by-zero error. We now take the positive sign for the $\sqrt{d}$ term in the numerator, as the negative sign would yield a past collision time. Negative numerators are raised to zero as before, but now we also raise a negative d to zero for reasons explained below.

$\ll \ldots \gg$

$$\begin{aligned} \Delta t_{quad} &:= \left(\begin{array}{ll} numer \geq denom{\cdot}\Delta t_{max} & \rightarrow \top \\ numer < denom{\cdot}\Delta t_{max} & \rightarrow \dfrac{numer}{denom} \end{array} \right) \\ numer &:= \left(-b + \sqrt{d \vee 0}\right) \vee 0 \\ denom &:= 2{\cdot}a \end{aligned}$$

A negative value of d may occur if the distance between the particles exceeds the tethering distance ($c > 0$), the particles are approaching ($b < 0$), but the particles will end up separating before the distance between them reaches $\Delta u_{tethering}$. By raising a negative d to zero, we opt for a collision at the time when the distance between the particles is minimized. For the sake of brevity we omit a formal treatment of this case. Consider, however, that with the help of a positive $\theta_{revolve}$, a sequence of tethering collisions with $d < 0$ will cause two particles to spiral inwards. Eventually, the distance between them will decrease to $\Delta u_{tethering}$.

Finally we can define $detect$. It has all the parameters of $detect_{blocking}$ and $detect_{tethering}$, as well as an additional argument $type_{collision}$ that indicates the type of collision to detect. The value of $type_{collision}$ is either "$blocking$" or "$tethering$".

$\ll tethered\_particle\_system \gg$

$$\begin{aligned} detect\left([\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}, type_{collision}]\right) &:= \Delta t \\ \Delta t &:= \left(\begin{array}{lll} type_{collision} & \equiv \text{“}blocking\text{”} & \rightarrow detect_{blocking}\,(args) \\ type_{collision} & \equiv \text{“}tethering\text{”} & \rightarrow detect_{tethering}\,(args) \end{array} \right) \\ args &:= [\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}] \end{aligned}$$

B.3 Collision Impulses

This section and the next address collision response, the counterpart to collision detection. Given particle A with velocity $\vec{v}_A$ and particle B with velocity $\vec{v}_B$, we assume that a collision between A and B will occur before the time t advances. In other words, if we were to evaluate $detect\left([\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}, type_{collision}]\right)$, the result would be zero.

What we need to calculate are the particle velocities $\vec{v}_A{}'$ and $\vec{v}_B{}'$ immediately after the collision. Whereas calculations that resolve rebounding particles have been formulated many times over, the presented treatment of retracting and revolving particles is probably new. Regardless of the type of collision that occurs, the quantity that determines $\vec{v}_A{}'$ and $\vec{v}_B{}'$ is the impulse $\Delta\vec{p}$. This is the result of the $tethered\_particle\_system$ function $impulse$, which we define in this section.

The following applies the law of conservation of momentum to particles A and B, equating the combined momentum before and after the collision. Particle A has mass m_A, and B has mass m_B.

$$m_A \cdot \vec{v}_A + m_B \cdot \vec{v}_B \equiv m_A \cdot \vec{v}_A{}' + m_B \cdot \vec{v}_B{}'$$

We rearrange terms to obtain two expressions for the impulse $\Delta\vec{p}$, the momentum that A gains as a result of the collision.

$$\Delta\vec{p} \equiv m_A \cdot (\vec{v}_A{}' - \vec{v}_A) \equiv -m_B \cdot (\vec{v}_B{}' - \vec{v}_B)$$

Found by manipulating the above, (7) yields the new velocities given $\Delta\vec{p}$.

$$\begin{array}{lll} \vec{v}_A{}' \equiv \vec{v}_A + \dfrac{\Delta\vec{p}}{m_A} & \quad & \text{(a)} \\ \vec{v}_B{}' \equiv \vec{v}_B - \dfrac{\Delta\vec{p}}{m_B} & \quad & \text{(b)} \end{array} \tag{7}$$

In pursuit of $\Delta\vec{p}$, we subtract (7a) from (7b) to obtain the following.

$$\vec{v}_B{}' - \vec{v}_A{}' \equiv \vec{v}_B - \vec{v}_A - \left(\frac{1}{m_A} + \frac{1}{m_B}\right) \cdot \Delta\vec{p}$$

Isolating $\Delta\vec{p}$, we obtain an expression for the impulse that depends on relative particle velocities.

$$\Delta\vec{p} \equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot ((\vec{v}_B - \vec{v}_A) - (\vec{v}_B{}' - \vec{v}_A{}')) \tag{8}$$

For the sake of convenience we introduce $\vec{v}_{AB}$ and $\vec{v}_{AB}{}'$ to represent the initial and final relative velocities. Suppose $\hat{u}$ is a unit vector aligned with the axis that runs through the centers of both particles. It points from particle A to B. The relative velocity $\vec{v}_{AB}$ can then be expressed as the sum of two components: $\vec{v}_{\hat{u}}$, which is parallel to $\hat{u}$, and $\vec{v}_{\hat{w}}$, which is perpendicular to $\hat{u}$. Similarly, $\vec{v}_{AB}{}'$ can be divided into the components $\vec{v}_{\hat{u}}{}'$ and $\vec{v}_{\hat{w}}{}'$.

$$\begin{aligned} \vec{v}_B - \vec{v}_A \;&\equiv\; \vec{v}_{AB} \;\equiv\; \vec{v}_{\hat{u}} + \vec{v}_{\hat{w}} \\ \vec{v}_B{}' - \vec{v}_A{}' \;&\equiv\; \vec{v}_{AB}{}' \;\equiv\; \vec{v}_{\hat{u}}{}' + \vec{v}_{\hat{w}}{}' \end{aligned} \tag{9}$$

Here we begin introducing approximations, and adopt the well-known frictionless collision model expressed in (10). In this model, the relative velocity along the $\hat{u}$ axis is reversed in direction and reduced to a fraction of its former magnitude. The factor $c_{restititute}$ is the coefficient of restitution, and we require $0 \leq c_{restitute} \leq 1$. Perpendicular to $\hat{u}$, the velocity does not change.

$$\begin{aligned} \vec{v}_{\hat{u}}{}' \;&\equiv\; -c_{restitute} \cdot \vec{v}_{\hat{u}} &&\text{(a)} \\ \vec{v}_{\hat{w}}{}' \;&\equiv\; \vec{v}_{\hat{w}} &&\text{(b)} \end{aligned} \tag{10}$$

Applying the approximation of (10), we derive an expression for the impulse.

$$\begin{aligned} &\Delta\vec{p} \ldots \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot ((\vec{v}_B - \vec{v}_A) - (\vec{v}_B{}' - \vec{v}_A{}')) \ldots && \{\text{from } (8)\} \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot ((\vec{v}_{\hat{u}} + \vec{v}_{\hat{w}}) - (\vec{v}_{\hat{u}}{}' + \vec{v}_{\hat{w}}{}')) \ldots && \{\text{substitution using } (9)\} \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot ((\vec{v}_{\hat{u}} - \vec{v}_{\hat{u}}{}') + (\vec{v}_{\hat{w}} - \vec{v}_{\hat{w}}{}')) \ldots && \{\text{rearranging of terms}\} \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot ((\vec{v}_{\hat{u}} - \vec{v}_{\hat{u}}{}') + (\vec{v}_{\hat{w}} - \vec{v}_{\hat{w}})) \ldots && \{\text{substitution using } (10b)\} \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot (\vec{v}_{\hat{u}} - \vec{v}_{\hat{u}}{}') \ldots \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot (\vec{v}_{\hat{u}} - (-c_{restitute} \cdot \vec{v}_{\hat{u}})) \ldots && \{\text{substitution using } (10a)\} \\ &\quad\equiv \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot (1 + c_{restitute}) \cdot \vec{v}_{\hat{u}} \end{aligned}$$

We will use impulse derived above not only for rebounding particles, its usual application, but for retracting particles as well. The expression is repeated in (11a) below. In the case of revolving particles, (11a) could result in an excessive number of extremely small impulses. In such cases we abandon the approximation of (10), and use (11b) derived from (8) and (9).

$$\begin{aligned} \Delta\vec{p} \;&\equiv\; \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot (1 + c_{restitute}) \cdot \vec{v}_{\hat{u}} &&\text{(a)} \\ \Delta\vec{p} \;&\equiv\; \left(\frac{1}{m_A} + \frac{1}{m_B}\right)^{-1} \cdot (\vec{v}_{AB} - \vec{v}_{AB}{}') &&\text{(b)} \end{aligned} \tag{11}$$

Arguments of the function $impulse$ include the system state variables Ψ and t, and the IDs of the colliding particles A and B. The masses of both particles m_A and m_B could be obtained from Ω_ψ, but we omit this variable. Included instead are the parameters M_A and M_B, which allow loaded group masses to be used in place of individual particle masses. The other arguments are $\Omega_{\psi\psi}$, which stores the coefficients of restitution; $\theta_{revolve}$, which distinguishes retraction from revolution; and $type_{collision}$, which is either "$blocking$" or "$tethering$". The result of $impulse$ is $\Delta\vec{p}$. The species, current position, and pre-collision velocity of each particle is obtained, as shown below, as well as the combined mass M_{AB}.

$\ll$ *tethered_particle_system* $\gg$

$$\begin{aligned}
&impulse\left([\Psi, t, id_A, id_B, M_A, M_B, \Omega_{\psi\psi}, \theta_{revolve}, type_{collision}]\right) := \Delta\vec{p}\\
&\quad spc_A := \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right)\\
&\quad spc_B := \Psi\left(id_B\right)\left(\text{“}spc\text{”}\right)\\
&\quad \vec{u}_A{}' := position\left([\Psi, t, id_A]\right)\\
&\quad \vec{u}_B{}' := position\left([\Psi, t, id_B]\right)\\
&\quad \vec{v}_A := \Psi\left(id_A\right)\left(\text{“}\vec{v}\text{”}\right)\\
&\quad \vec{v}_B := \Psi\left(id_B\right)\left(\text{“}\vec{v}\text{”}\right)\\
&\quad M_{AB} := \left(\frac{1}{M_A} + \frac{1}{M_B}\right)^{-1}
\end{aligned}$$

The unit vector $\hat{u}$ points towards particle B from A.

$\ll \ldots; impulse \gg$

$$\hat{u} := \frac{(\vec{u}_B{}' - \vec{u}_A{}')}{\sqrt{\sum\left((\vec{u}_B{}' - \vec{u}_A{}')^2\right)}}$$

Below we define the relative velocity $\vec{v}_{AB}$, the relative velocity projected onto $\hat{u}$, and the relative velocity projected onto the plane perpendicular to $\hat{u}$.

$\ll \ldots \gg$

$$\begin{aligned}
\vec{v}_{AB} &:= \vec{v}_B - \vec{v}_A\\
\vec{v}_{\hat{u}} &:= \sum\left(\vec{v}_{AB}\cdot\hat{u}\right)\cdot\hat{u}\\
\vec{v}_{\hat{w}} &:= \vec{v}_{AB} - \vec{v}_{\hat{u}}
\end{aligned}$$

If we are dealing with a blocking collision, the impulse is simply that of (11a) but with $c_{restitute} = c_{rebound}$.

$\ll \ldots \gg$

$$\Delta\vec{p} := \left(\begin{array}{lll} type_{collision} & \equiv \text{“}blocking\text{”} & \to M_{AB} \cdot (1 + c_{rebound}) \cdot \vec{v}_{\hat{u}} \\ type_{collision} & \equiv \text{“}tethering\text{”} & \to \Delta\vec{p}_{tethering} \end{array} \right)$$
$$c_{rebound} := \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{“}c_{rebound}\text{”}\right)$$

Proceeding given a tethering collision, we must determine whether a retraction impulse is sufficient to draw the two particles together at an angle of at least $\theta_{revolve}$. If it is sufficient, we use (11a) again but with $c_{restitute} = c_{retract}$.

$\ll \ldots ; \Delta\vec{p} \gg$

$$\Delta\vec{p}_{tethering} := \left(\begin{array}{ll} sufficient & \to M_{AB} \cdot (1 + c_{retract}) \cdot \vec{v}_{\hat{u}} \\ \neg sufficient & \to \Delta\vec{p}_{revolve} \end{array} \right)$$
$$c_{retract} := \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{“}c_{retract}\text{”}\right)$$

To determine whether the retraction impulse is sufficient, we estimate the magnitude of the minimum post-collision relative velocity projected onto $\hat{u}$. This quantity is $v_{\hat{u}_{min}}$. The ratio of $v_{\hat{u}_{min}}$ to the magnitude of $\vec{v}_{\hat{w}}$ is the tangent of $\theta_{revolve}$, theoretically, but we must choose an angle slightly greater than $\theta_{revolve}$ to account for round-off errors. If the magnitude of $c_{retract} \cdot \vec{v}_{\hat{u}}$ exceeds $v_{\hat{u}_{min}}$, the retraction impulse is sufficient.

$\ll \ldots ; \Delta\vec{p}_{tethering} \gg$

$$sufficient := \sqrt{\sum\left(\left(c_{retract} \cdot \vec{v}_{\hat{u}}\right)^2\right)} > v_{\hat{u}_{min}}$$
$$v_{\hat{u}_{min}} := tan\left(\theta_{revolve} \cdot (1 + \epsilon_{revolve})\right) \cdot \sqrt{\sum\left(\vec{v}_{\hat{w}}^{\,2}\right)}$$
$$\epsilon_{revolve} := 2^{-24}$$

If the retraction impulse is insufficient, we adopt (11b) to model revolution. This requires the calculation of the final relative velocity $\vec{v}_{AB}{}'$, and the magnitude of this velocity is that of $\vec{v}_{AB}$ multiplied by a fraction. The fraction is determined such that, on average, the magnitude of the relative velocity scales down by a factor of $c_{revolve}$ each complete revolution.

$\ll \ldots \gg$

$$\Delta\vec{p}_{revolve} := M_{AB}\cdot(\vec{v}_{AB} - \vec{v}_{AB}{}')$$
$$\vec{v}_{AB}{}' := c_{revolve}{}^{\frac{\theta_{revolve}}{\pi}} \cdot \sqrt{\sum\left(\vec{v}_{AB}{}^2\right)} \cdot \hat{u}_{revolve}$$
$$c_{revolve} := \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{“}c_{revolve}\text{”}\right)$$

The direction indicated by $\hat{u}_{revolve}$ is found by taking $\vec{v}_{\hat{w}}$, and adding a component in the direction of $\hat{u}$ such that the particles are drawn together at an angle of $\theta_{revolve}$.

$\ll \ldots; \Delta\vec{p}_{revolve}; \vec{v}_{AB}{}' \gg$

$$\hat{u}_{revolve} := \frac{\vec{v}_{revolve}}{\sqrt{\sum\left(\vec{v}_{revolve}{}^2\right)}}$$
$$\vec{v}_{revolve} := \vec{v}_{\hat{w}} - tan\left(\theta_{revolve}\right) \cdot \sqrt{\sum\left(\vec{v}_{\hat{w}}{}^2\right)} \cdot \hat{u}$$

B.4 Loading and Restitution

Recall from Section 4.1 that the threat of large numbers of nearly-simultaneous collisions motivated the introduction of loaded groups of particles. The mass of a loaded group is the sum of the masses of each particle in the group, and the velocity of each particle in the group is the same. The formation of a loaded group after a collision is called loading, and the eventual separation of a loaded group is called restitution. Accordingly, we pursue the definitions of the $tethered\_particle\_system$ functions $load$ and $restitute$.

Figure 34 depicts a loaded group with particles identified by the natural numbers 0 through 10. Assume that particles shown in contact with one another, like particles 2 and 4, are directly loaded. Because 2 and 4 are directly loaded, 4 is in the domain of $\Psi\left(2\right)\left(\text{“}loaded\text{”}\right)$ and 2 is in the domain of $\Psi\left(4\right)\left(\text{“}loaded\text{”}\right)$. If we choose any pair of directly-loaded particles, then the group can be partitioned into two branches on either side. If we choose the pair $[2, 4]$, for example, then one branch includes the IDs in $[0, 1, 2, 3, 9, 10]$, and the other includes $[4, 5, 6, 7, 8]$.

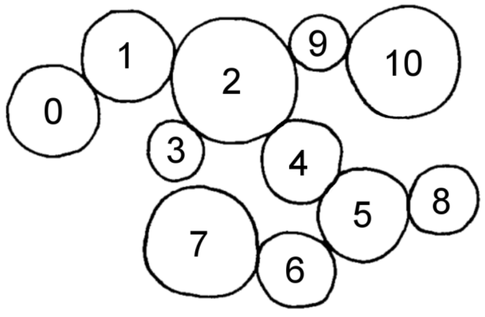


Figure 34: A loaded group of particles.

It is important to note that loaded groups never form loops. A loop would be present in Figure 34 if the particles 3 and 7 were directly loaded, for instance. We can be sure that a loop in a loaded group will never emerge, as this would require two particles in the same loaded group to collide with one another. Because all particles in a loaded group have the same velocity, no pair will collide before the group breaks up.

Before addressing $load$ and $restitute$, we define the functions $scan_{branch}$ and $adjust_{branch}$. The $scan_{branch}$ function traverses the branch that includes particle A, identified by the argument id_A, but not particle src, identified by id_{src}. The result includes the total mass M_A of the branch, a vector ID_{ψ_A} of IDs of particles in the branch, and a vector $ID_{\psi\psi_A}$ of pairs of directly-loaded particles. If id_A were 4 and id_{src} were 2, looking at Figure 34, then $ID_{\psi\psi_A}$ would include the pairs $[4,5]$, $[5,6]$, $[6,7]$, and $[5,8]$, though not necessarily in that order.

$\ll tethered\_particle\_system \gg$

$$
\begin{aligned}
&scan_{branch}\left([\Psi, t, id_A, id_{src}, \Omega_\psi]\right) := \left[M_A, ID_{\psi_A}, ID_{\psi\psi_A}\right]\\
&\qquad spc_A := \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right)\\
&\qquad m_A := \Omega_\psi\left(spc_A\right)\left(\text{“}m\text{”}\right)\\
&\qquad ID_{loaded} := \Diamond\left(\Psi\left(id_A\right)\left(\text{“}loaded\text{”}\right) \triangleleft\!\!| \; [id_{src}]\right)
\end{aligned}
$$

Note that ID_{loaded}, defined above, is a vector that includes the IDs of all particles directly loaded with particle A, except for that of particle src. The recursive function $loop$ traverses the branch by invoking $scan_{branch}$ on each particle listed in ID_{loaded}. The results are accumulated.

$\ll \ldots ; scan_{branch} \gg$

$$\begin{aligned}
&\left[M_A, ID_{\psi_A}, ID_{\psi\psi_A}\right] := \mathit{loop}\left([0, m_A, [id_A], [\,]]\right)\\
&\quad \mathit{loop}\left(\left[i, M_{A_i}, ID_{\psi_{A_i}}, ID_{\psi\psi_{A_i}}\right]\right) := \ldots\\
&\qquad\qquad \begin{pmatrix} i = \#ID_{\mathit{loaded}} & \rightarrow \left[M_{A_i}, ID_{\psi_{A_i}}, ID_{\psi\psi_{A_i}}\right] \\ i < \#ID_{\mathit{loaded}} & \rightarrow \mathit{loop}\left(\left[i+1, {M_{A_i}}', {ID_{\psi_{A_i}}}', {ID_{\psi\psi_{A_i}}}'\right]\right) \end{pmatrix}\\
&\qquad id_B := ID_{\mathit{loaded}}(i)\\
&\qquad \left[M_B, ID_{\psi_B}, ID_{\psi\psi_B}\right] := scan_{\mathit{branch}}\left([\Psi, t, id_B, id_A, \Omega_\psi]\right)\\
&\qquad {M_{A_i}}' := M_{A_i} + M_B\\
&\qquad {ID_{\psi_{A_i}}}' := ID_{\psi_{A_i}} \parallel ID_{\psi_B}\\
&\qquad {ID_{\psi\psi_{A_i}}}' := ID_{\psi\psi_{A_i}} \parallel ID_{\psi\psi_B} \parallel [[id_A, id_B]]
\end{aligned}$$

The function $adjust_{\mathit{branch}}$ is similar to $scan_{\mathit{branch}}$ in that it also traverses a branch. In the case of $adjust_{\mathit{branch}}$, the purpose is to change the velocity of each particle in the branch to the value of the argument $\vec{v}_A{}'$. The result is the updated state Ψ'.

$\ll tethered\_particle\_system \gg$

$$\begin{aligned}
&adjust_{\mathit{branch}}\left([\Psi, t, id_A, id_{src}, \vec{v}_A{}']\right) := \Psi'\\
&\quad \psi_A := \Psi(id_A)\\
&\quad \vec{u}_A{}' := \mathit{position}\left([\Psi, t, id_A]\right)\\
&\quad ID_{\mathit{loaded}} := \Diamond\left(\Psi(id_A)(\text{“}\mathit{loaded}\text{”}) \ntriangleleft [id_{src}]\right)
\end{aligned}$$

The recursive function *loop* yields Ψ_Δ, the state with all particles in the branch adjusted except the id_A particle.

$\ll \ldots ; adjust_{branch} \gg$

$$\begin{aligned}
&\Psi_\Delta := \mathit{loop}\left([0, \Psi]\right)\\
&\quad \mathit{loop}\left([i, \Psi_i]\right) := \begin{pmatrix} i = \#ID_{\mathit{loaded}} & \rightarrow \Psi_i \\ i < \#ID_{\mathit{loaded}} & \rightarrow \mathit{loop}\left([i+1, \Psi_i{}']\right) \end{pmatrix}\\
&\qquad id_B := ID_{\mathit{loaded}}(i)\\
&\qquad \Psi_i{}' := adjust_{\mathit{branch}}\left(\Psi_i, t, id_B, id_A, \vec{v}_A{}'\right)
\end{aligned}$$

The current time and a particle's current position must be recorded along with its new velocity.

$\ll \ldots \gg$

$$\psi_A{}' := \psi_A \lhd \begin{bmatrix} \text{“}t\text{”} & \to t \\ \text{“}\vec{u}\text{”} & \to \vec{u}_A{}' \\ \text{“}\vec{v}\text{”} & \to \vec{v}_A{}' \end{bmatrix}$$
$$\Psi' := \Psi_\Delta \lhd \begin{bmatrix} id_A \to \psi_A{}' \end{bmatrix}$$

Both $scan_{branch}$ and $adjust_{branch}$ can be applied to an entire loaded group instead of a single branch. This is done by invoking either function with an id_{src} value of $\varnothing$, assuming that $\varnothing$ is never used as a particle ID.

We now define $load$, which addresses situations in which a particle A collides with a particle B. Note that A and B can always be considered to be in loaded groups of at least one particle prior to the collision, so $load$ essentially joins two loaded groups together to form a larger group. The function has the same arguments as $impulse$, with one exception: the masses of each loaded group, M_A and M_B, are not supplied as arguments but instead obtained from $scan_{branch}$ using the argument Ω_ψ. The result of $load$ is the updated state Ψ', a vector ID_ψ listing all particles in the new loaded group, and a vector $ID_{\psi\psi}$ listing pairs of directly-loaded particles in the group.

$\ll tethered\_particle\_system \gg$

$$load\left([\Psi, t, id_A, id_B, \Omega_\psi, \Omega_{\psi\psi}, \theta_{revolve}, type_{collision}]\right) := [\Psi', ID_\psi, ID_{\psi\psi}]$$
$$\psi_A := \Psi\left(id_A\right)$$
$$\psi_B := \Psi\left(id_B\right)$$
$$\vec{v}_A := \psi_A\left(\text{“}\vec{v}\text{”}\right)$$
$$\vec{v}_B := \psi_B\left(\text{“}\vec{v}\text{”}\right)$$
$$\left[M_A, ID_{\psi_A}, ID_{\psi\psi_A}\right] := scan_{branch}\left([\Psi, t, id_A, \varnothing, \Omega_\psi]\right)$$
$$\left[M_B, ID_{\psi_B}, ID_{\psi\psi_B}\right] := scan_{branch}\left([\Psi, t, id_B, \varnothing, \Omega_\psi]\right)$$
$$M_{AB} := \left(\frac{1}{M_A} + \frac{1}{M_B}\right)^{-1}$$
$$ID_\psi := ID_{\psi_A} \parallel ID_{\psi_B}$$
$$ID_{\psi\psi} := ID_{\psi\psi_A} \parallel ID_{\psi\psi_B} \parallel [[id_A, id_B]]$$

We seek Ψ' to complete $load$. Before the collision, particle A belongs to a group of mass M_A and velocity $\vec{v}_A$. Similarly, B belongs to a group of mass M_B and velocity $\vec{v}_B$. Once the two groups are combined, they share the velocity $\vec{v}_{load}$. We will derive $\vec{v}_{load}$ from conservation of momentum.

$$M_A \cdot \vec{v}_A + M_B \cdot \vec{v}_B \equiv (M_A + M_B) \cdot \vec{v}_{load}$$

Isolating $\vec{v}_{load}$ yields the following.

$$\vec{v}_{load} \equiv \frac{M_A}{M_A + M_B} \cdot \vec{v}_A + \frac{M_B}{M_A + M_B} \cdot \vec{v}_B$$

We will define $\vec{v}_{load}$ with the following expression, however. This allows either M_A or M_B to be infinite, but not both.

$\ll \ldots ; load \gg$

$$\vec{v}_{load} := \left(1 + \frac{M_B}{M_A}\right)^{-1} \cdot \vec{v}_A + \left(1 + \frac{M_A}{M_B}\right)^{-1} \cdot \vec{v}_B$$

After loading and restitution, the total change in momentum must still satisfy the calculations of Section B.3. We therefore obtain the collision impulse $\Delta\vec{p}$.

$\ll \ldots \gg$

$$\Delta\vec{p} := impulse\left([\Psi, t, id_A, id_B, M_A, M_B, \Omega_{\psi\psi}, \theta_{revolve}, type_{collision}]\right)$$

We also define the loading impulse $\Delta\vec{p}_{load}$, the change in momentum that would have been necessary to give both groups the same velocity. The expression below is obtained from (11b), noting that the final relative velocity is zero.

$\ll \ldots \gg$

$$\Delta\vec{p}_{load} := M_{AB} \cdot (\vec{v}_B - \vec{v}_A)$$

Again, we desire an overall momentum change of $\Delta\vec{p}$. Effectively, a momentum change of $\Delta\vec{p}_{load}$ will be applied in the loading phase by changing particles velocities to $\vec{v}_{load}$. Thus, the difference between these momentum changes must be applied during the restitution phase. This difference, the restitution impulse, is assigned to $\Delta\vec{p}_{AB}$.

$\ll \ldots \gg$

$$\Delta\vec{p}_{AB} := \Delta\vec{p} - \Delta\vec{p}_{load}$$

The following definitions apply the change in velocity to $\vec{v}_{load}$, and record the restitution impulse associated with the loading of particles A and B.

$\ll \ldots \gg$

$$\begin{aligned}
&\Psi_{\Delta} := adjust_{branch}\left([\Psi, t, id_A, \varnothing, \vec{v}_{load}]\right)\\
&\Psi_{\Delta\Delta} := adjust_{branch}\left([\Psi_{\Delta}, t, id_B, \varnothing, \vec{v}_{load}]\right)\\
&{\psi_A}' := \psi_A \lhd \left[\ \text{“}loaded\text{”} \ \ \rightarrow \left[\ id_B \ \ \rightarrow \Delta\vec{p}_{AB}\ \right]\ \right]\\
&{\psi_B}' := \psi_B \lhd \left[\ \text{“}loaded\text{”} \ \ \rightarrow \left[\ id_A \ \ \rightarrow -\Delta\vec{p}_{AB}\ \right]\ \right]
\end{aligned}$$

Combining these adjustments yields Ψ', and completes the definition of $load$.

$\ll \ldots \gg$

$$\Psi' := \Psi_{\Delta\Delta} \lhd \begin{bmatrix} id_A & \rightarrow {\psi_A}' \\ id_B & \rightarrow {\psi_B}' \end{bmatrix}$$

At a simulated time of at least $\Delta t_{restitute}$ after particles A and B load, the function $restitute$ is invoked to apply the restitution impulse and separate the particles. The result of $restitute$ is the new state Ψ' and the vector ID_ψ of IDs of particles with new velocities.

$\ll tethered\_particle\_system \gg$

$$\begin{aligned}
&restitute\left([\Psi, t, id_A, id_B, \Omega_\psi]\right) := [\Psi', ID_\psi]\\
&\quad \psi_A := \Psi\left(id_A\right)\\
&\quad \psi_B := \Psi\left(id_B\right)\\
&\quad \vec{v}_{load} := \psi_A\left(\text{“}\vec{v}\text{”}\right)\\
&\quad \left[M_A, ID_{\psi_A}, ID_{\psi\psi_A}\right] := scan_{branch}\left([\Psi, t, id_A, id_B, \Omega_\psi]\right)\\
&\quad \left[M_B, ID_{\psi_B}, ID_{\psi\psi_B}\right] := scan_{branch}\left([\Psi, t, id_B, id_A, \Omega_\psi]\right)\\
&\quad ID_\psi := ID_{\psi_A} \parallel ID_{\psi_B}\\
&\quad \Delta\vec{p}_{AB} := \psi_A\left(\text{“}loaded\text{”}\right)\left(id_B\right)
\end{aligned}$$

After restitution, particle A ends up in one group and particle B ends up in another. The velocities ${\vec{v}_A}'$ and ${\vec{v}_B}'$ of these groups are calculated from (7) using the restitution impulse $\Delta\vec{p}_{AB}$.

$\ll \ldots ; restitute \gg$

$$\begin{aligned}
{\vec{v}_A}' &:= \vec{v}_{load} + \frac{\Delta\vec{p}_{AB}}{M_A}\\
{\vec{v}_B}' &:= \vec{v}_{load} - \frac{\Delta\vec{p}_{AB}}{M_B}
\end{aligned}$$

Individual state changes include the adjustment of the particle velocities and the removal of the recorded restitution impulses.

$\ll \ldots \gg$

$$\begin{aligned}
&\Psi_{\Delta} := adjust_{branch}\left(\left[\Psi, t, id_A, id_B, \vec{v}_A{}'\right]\right)\\
&\Psi_{\Delta\Delta} := adjust_{branch}\left(\left[\Psi_{\Delta}, t, id_B, id_A, \vec{v}_B{}'\right]\right)\\
&\psi_A{}' := \psi_A \lhd \left[\ \text{“}loaded\text{”} \ \ \rightarrow \psi_A\left(\text{“}loaded\text{”}\right) \mathrel{\ominus\!\!\!\lhd} \left[id_B\right]\ \right]\\
&\psi_B{}' := \psi_B \lhd \left[\ \text{“}loaded\text{”} \ \ \rightarrow \psi_B\left(\text{“}loaded\text{”}\right) \mathrel{\ominus\!\!\!\lhd} \left[id_A\right]\ \right]
\end{aligned}$$

The changes are combined to complete the definition of $restitute$.

$\ll \ldots \gg$

$$\Psi' := \Psi_{\Delta\Delta} \lhd \begin{bmatrix} id_A & \rightarrow \psi_A{}' \\ id_B & \rightarrow \psi_B{}' \end{bmatrix}$$

B.5 Random Impulses

Recall from Section 4.1 that random impulses are introduced to prevent the overall kinetic energy in a TPS model from converging to zero. Here we define the remaining three $tethered\_particle\_system$ functions: $detect_{RI}$, which yields the time before a random impulse; $impulse_{RI}$, which gives the momentum change associated with a random impulse; and $impact$, which applies an impulse to a loaded group.

Each random impulse is associated with a single particle. A particle of species spc_A experiences a random impulse after a time Δt, chosen from an exponential distribution. The mean time value, τ_{RI}, is a property associated with spc_A and accessed via Ω_ψ. The calculation of Δt is performed in the following definition of $detect_{RI}$.

$\ll tethered\_particle\_system \gg$

$$\begin{aligned}
&detect_{RI}\left(\left[spc_A, \Omega_\psi\right]\right) := \Delta t\\
&\qquad \tau_{RI} := \Omega_\psi\left(spc_A\right)\left(\text{“}\tau_{RI}\text{”}\right)\\
&\qquad \Delta t := exponential\left(\tau_{RI}\right)()
\end{aligned}$$

The function $impulse_{RI}$ requires spc_A and Ω_ψ, as well as the number of spatial dimensions n_{dim}. It results in $\Delta\vec{p}$, the change in a particle's momentum. The impulse $\Delta\vec{p}$ is a product of a momentum magnitude p_{RI}, a factor u_{RI}, and a direction $\hat{u}$.

$\ll$ *tethered_particle_system* $\gg$

$$\begin{aligned} impulse_{RI}\left([n_{dim}, spc_A, \Omega_\psi]\right) &:= \Delta\vec{p} \\ \Delta\vec{p} &:= p_{RI}\cdot u_{RI}\cdot\hat{u} \end{aligned}$$

The random impulse is independent from the velocity of the particle. From a practical perspective, this is necessary to ensure that the TPS does not treat one frame of reference differently from another; adding some velocity $\Delta\vec{v}$ to every TPS particle should ideally have no effect on how the particles interact. From a physical perspective, a velocity-independent random impulse could represent an impact between a particle and some other small yet fast object. The speed of this object would have to modeled as infinity, and hence in order for its momentum to be finite, its mass would have to modeled as zero.

Because $\Delta\vec{p}$ above does not depend on a particle's velocity, it is inappropriate to use random impulses to model drag force. Drag, which acts against a particle's motion relative to that of a surrounding fluid, can be represented by adding numerous small particles to a TPS model. Another possible option it to combine the TPS with algorithms for fluid dynamics.

The variable p_{RI} represents the maximum magnitude of the random impulse delivered by something we will call an "imaginary object". The maximum magnitude is delivered only if the imaginary object is heading towards the center of the particle immediately before the impact. The magnitude is selected randomly from a gamma distribution with shape parameter k_{RI} and mean value μ_{RI}. Like τ_{RI}, these parameters are recorded in Ω_ψ.

$\ll \ldots; impulse_{RI}; \Delta\vec{p} \gg$

$$\begin{aligned} p_{RI} &:= gamma\left(\left[k_{RI}, \frac{\mu_{RI}}{k_{RI}}\right]\right)() \\ k_{RI} &:= \Omega_\psi\left(spc_A\right)\left(\text{“}k_{RI}\text{”}\right) \\ \mu_{RI} &:= \Omega_\psi\left(spc_A\right)\left(\text{“}\mu_{RI}\text{”}\right) \end{aligned}$$

In pursuit of the factor u_{RI}, we will regard a particle as a sphere and neglect the size of the imaginary object. If the object were to strike the spherical particle heading directly towards the sphere's center, we would use $u_{RI} = 1$ and the magnitude of the random impulse $\Delta\vec{p}$ would be p_{RI}. But Figure 35 shows the more general case, in which the object strikes the spherical surface at an angle.

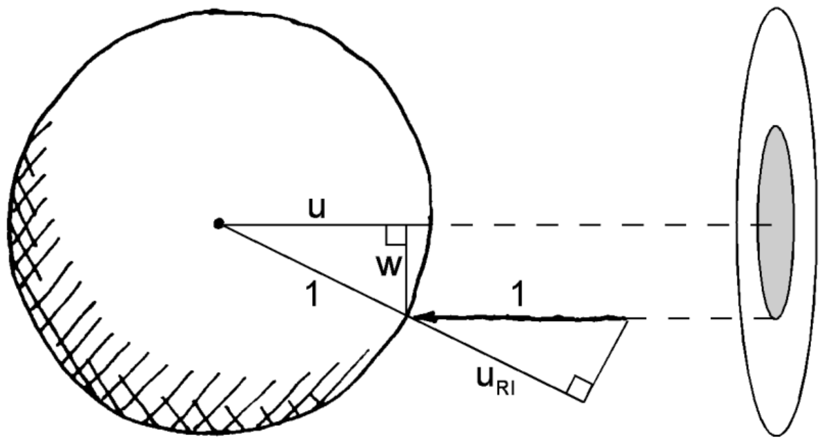


Figure 35: The geometry of a situation in which a small imaginary object, travelling along the arrow, strikes a spherical particle to deliver a random impulse.

Because we do not bother choosing a specific position and direction for the imaginary object, we will choose u_{RI} from a random distribution. We need not bother assigning a radius to the particle, as the distribution of u_{RI} does not depend on the particle's size, but we choose a radius of 1 regardless to simplify the mathematics. Consistent with Section B.3, the impulse delivered to the particle will depend on the component of the relative velocity aligned with the axis between the particle and the object at the time of impact. The ratio of this component's magnitude to p_{RI} is u_{RI}, as indicated in Figure 35.

Note that the triangle in Figure 35 with u_{RI} is congruent to the triangle with u and w, and hence $u = u_{RI}$. We will let U be a random function associated with u in the figure, and W will be a random function associated with w. The cumulative distribution function of U can be expressed in terms of W as follows.

$$\begin{aligned}
&\mathit{cdf}\,(U)\,(u)\ldots \\
&\quad = P\,(U \le u)\ldots && \{\text{definition of } \mathit{cdf}\} \\
&\quad = P\left(\sqrt{1-W^2} \le u\right)\ldots && \{\text{geometry (see Figure 35)}\} \\
&\quad = P\,(1-W^2 \le u^2)\ldots \\
&\quad = P\,(1-u^2 \le W^2)\ldots \\
&\quad = P\left(W \ge \sqrt{1-u^2}\right)\ldots \\
&\quad = 1 - P\left(W < \sqrt{1-u^2}\right) && \{\text{probability theory}\}
\end{aligned}$$

Note that (12), which summarizes the calculations above, holds for any number of dimensions.

$$cdf\,(U)\,(u) = 1 - P\left(W < \sqrt{1-u^2}\right) \tag{12}$$

Now, for the first time in all of Appendix B, we must introduce separate cases for one-, two-, and three-dimensional TPS models. We must do this because the probability distribution of W, and hence that of u_{RI}, depends on the number of dimensions n_{dim}. The direction $\hat{u}$ also depends on n_{dim}.

$\ll \ldots \gg$

$$[u_{RI}, \hat{u}] := \left(\begin{array}{ll} n_{dim} = 3 & \rightarrow [u_{RI_3}, \hat{u}_3] \\ n_{dim} = 2 & \rightarrow [u_{RI_2}, \hat{u}_2] \\ n_{dim} = 1 & \rightarrow [u_{RI_1}, \hat{u}_1] \end{array} \right)$$

In three dimension, the probability of $W < w$ for some w is the probably of the imaginary object passing through the shaded area in Figure 35. Note that the probability of the object passing through any 2D region in the outer circle, which has radius 1, is proportional to the area of that region.

$$P\,(W < w) = \frac{\pi \cdot w^2}{\pi \cdot 1^2} = w^2 \tag{13}$$

Thus for 3D models, we have the following CDF for U.

$$\begin{aligned} & cdf\,(U)\,(u) \ldots \\ & \quad = 1 - P\left(W < \sqrt{1-u^2}\right) \ldots && \{\text{from (12)}\} \\ & \quad = 1 - \sqrt{1-u^2}^{\,2} \ldots && \{\text{using (13)}\} \\ & \quad = 1 - (1-u^2) \ldots \\ & \quad = u^2 \end{aligned}$$

A CDF applied to its own random function has a uniform distribution, so we replace $cdf\,(U)\,(u)$ with $uniform\,([0,1])\,()$ and solve for u above. This gives us an expression to assign to u_{RI} in the 3D case.

$\ll \ldots; [u_{RI}, \hat{u}] \gg$

$$u_{RI_3} := \sqrt{uniform\,([0,1])\,()}$$

In two dimensions, the circle in Figure 35 becomes a line, and W is distributed as follows.

$$P(W < w) = \frac{w}{1} = w \tag{14}$$

Thus for the CDF of U we have the following.

$$\begin{aligned}
&\mathit{cdf}(U)(u)\ldots \\
&\quad = 1 - P\left(W < \sqrt{1-u^2}\right)\ldots && \{\text{from } (12)\} \\
&\quad = 1 - \sqrt{1-u^2} && \{\text{using } (14)\}
\end{aligned}$$

Replacing the CDF with the uniform distribution and solving for u gives us a definition of u_{RI} for the 2D case. When solving for u, we exploit the fact that $1 - uniform([0,1])$ has the same distribution as $uniform([0,1])$.

$\ll \ldots \gg$

$$u_{RI_2} := \sqrt{1 - uniform([0,1])()^2}$$

In one dimension, the imaginary object always heads towards the particle's center.

$\ll \ldots \gg$

$$u_{RI_1} := 1$$

Depending on the number of dimensions, the direction $\hat{u}$ of the random impulse is selected at random from the surface of a sphere, the surface of a circle, or the ends points of a line. Such calculations are well known, so we state them below without derivation.

$\ll \ldots \gg$

$$\begin{aligned}
&\hat{u}_3 := [u_x, u_y, u_z] \\
&\quad u_z := uniform([-1,1])() \\
&\quad [u_x, u_y] := \sqrt{1-{u_z}^2}\cdot[cos(\phi), sin(\phi)] \\
&\quad\quad \phi := uniform([0, 2\cdot\pi])()
\end{aligned}$$

$$\begin{aligned}
&\hat{u}_2 := [cos(\phi), sin(\phi)] \\
&\quad \phi := uniform([0, 2\cdot\pi])()
\end{aligned}$$

$$\hat{u}_1 := [2\cdot uniform_{\mathbb{N}}(2)() - 1]$$

Suppose we have obtained the random impulse $\Delta\vec{p}$ and wish to apply it to some particle A.

If A is in a loaded group, then its velocity must remain the same as all other particles in the group. We therefore apply $\Delta\vec{p}$ to the entire group. This adjustment is performed by the function $impact$ defined below, which results in the new state Ψ' and the IDs of all particles in the group. Note the use of the function $scan_{branch}$ to obtain the accumulated mass M_A of the group, which in turn is used to calculate the new velocity $\vec{v}_A{}'$, which in turn is passed as an argument to the function $adjust_{branch}$.

$\ll tethered\_particle\_system \gg$

$$\begin{aligned}
&impact\left([\Psi, t, id_A, \Omega_\psi, \Delta\vec{p}]\right) := [\Psi', ID_{\psi_A}] \\
&\qquad \vec{v}_A := \Psi\left(id_A\right)\left(\text{“}\vec{v}\text{”}\right) \\
&\qquad \left[M_A, ID_{\psi_A}, ID_{\psi\psi_A}\right] := scan_{branch}\left([\Psi, t, id_A, \varnothing, \Omega_\psi]\right) \\
&\qquad \vec{v}_A{}' := \vec{v}_A + \frac{\Delta\vec{p}}{M_A} \\
&\qquad \Psi' := adjust_{branch}\left([\Psi, t, id_A, \varnothing, \vec{v}_A{}']\right)
\end{aligned}$$

C DEVS Formulas

C.1 DEVS Functions

The selector $DEVS$, defined below, provides access to the DEVS simulator and various other functions that aid in the design of DEVS models. For each function, its primary result and the section describing it are indicated in the comments.

$$DEVS := \ldots \\ \begin{bmatrix} \text{“}simulate\text{”} & \rightarrow simulate \\ & \{\text{simulation results (Section C.2)}\} \\ \text{“}coupled_{DEVS}\text{”} & \rightarrow coupled_{DEVS} \\ & \{\text{DEVS coupled model (Section C.3)}\} \\ \text{“}pr_{\varnothing}\text{”} & \rightarrow pr_{\varnothing} \\ & \{\varnothing \text{ (Section C.4)}\} \\ \text{“}pr_{Z}\text{”} & \rightarrow pr_{Z} \\ & \{\text{ID of submodel with priority (Section C.4)}\} \\ \text{“}pr_{order}\text{”} & \rightarrow pr_{order} \\ & \{\text{ID of submodel with priority (Section C.4)}\} \\ \text{“}HL_{DEVS}\text{”} & \rightarrow HL_{DEVS} \\ & \{\text{DEVS hypercubic lattice model (Section C.5)}\} \end{bmatrix}$$

The following variables are used in various places to represent the indicated quantities.

t	{simulated time}
s	{state of a DEVS model}
x	{input value}
y	{output value}
n	{number of past events}
n_x	{number of past inputs}
n_y	{number of past outputs}

Because the simulated time is represented by a real number, and because multiple events can occur at the same simulated time, t is not an appropriate way to identify events. The natural number n is used instead to express the number of events that have already occurred. When we refer to the $[n_x, n_y, t, s]$ of event n, the following is true: exactly n_x inputs have been processed; the external transition function δ_{ext} has been evaluated n_x times; exactly n_y outputs have been processed; the internal transition function δ_{int} has been evaluated $n - n_x$ times; t is the simulated time at which the n^{th} event occurred; s is the state immediately after the n^{th} event. Although each input is processed by a separate external transition event, a single internal transition event can yield any number of outputs.

Each input and output has an associated simulated time and value. When we refer to the $[t, x]$ of input n_x, the following is true: exactly n_x inputs have previously been processed; t and x are the time and value associated with the $(n_x + 1)^{\text{th}}$ input. After n_y outputs have been processed, the $[t, y]$ of output n_y refers to the time and value associated with the $(n_y + 1)^{\text{th}}$ output. Note that n, n_x, and n_y all start at zero.

There are four main functions associated with the recording and retrieval of simulation information. We group all of these functions in a selector called IO, which is supplied to the simulator. As indicated below, the argument of IO identifies one of the four functions, and the identified function is applied to the argument that follows.

$$\begin{array}{ll}
IO\,(\text{“}input_s\text{”})\,(n) & \{\text{retrieves } [n_x, n_y, t, s] \text{ of event } n\} \\
IO\,(\text{“}output_s\text{”})\,([n, n_x, n_y, t, s]) & \{\text{records } [n_x, n_y, t, s] \text{ of event } n, \text{ results in } IO'\} \\
IO\,(\text{“}input_x\text{”})\,(n_x) & \{\text{retrieves } [t, x] \text{ of input } n_x\} \\
IO\,(\text{“}output_y\text{”})\,([n_y, t, y]) & \{\text{records } [t, y] \text{ of output } n_y, \text{ results in } IO'\}
\end{array}$$

By choosing the appropriate IO selector, each with its own set of recording/retrieval functions, a user can direct simulation information to either variables in RAM, files on a hard drive, or a peripheral device.

Discrete event simulations typically require a data structure called a priority queue or, as we will call it, a “future events list” (FEL). An FEL might be used as part of a DEVS model's state, as is the case for all of our DEVS coupled models. The selector $future\_events\_list$

provides access to an empty FEL and two FEL-related functions.

$$
\begin{array}{l}
future\_events\_list := \ldots \\
\qquad\qquad \left[\begin{array}{lll}
\text{``}FEL_{empty}\text{''} & \rightarrow FEL_{empty} & \{\text{an empty FEL}\} \\
\text{``}\delta_{FEL}\text{''} & \rightarrow \delta_{FEL} & \{\text{modifies an FEL}\} \\
\text{``}event_{FEL}\text{''} & \rightarrow event_{FEL} & \{\text{identifies the next event}\}
\end{array} \right]
\end{array}
$$

Once an empty FEL is obtained by the expression $future\_events\_list\,(\text{``}FEL_{empty}\text{''})$, the only way to modify it is with δ_{FEL}. Given a future events list FEL, the identity id_{event} of some event, the time t_{event} of that event, and a priority function pr, δ_{FEL} results in the new future events list FEL' with the event added. If an event identified by id_{event} is already in the FEL, then the timing and/or priority of the event is replaced. To remove an existing event, t_{event} is given the infinite value $\top$.

$\ll future\_events\_list \gg$

$$\delta_{FEL}\left([FEL, id_{event}, t_{event}, pr]\right) := FEL'$$

The function $event_{FEL}$ reports the identity and time of the event in FEL that is scheduled to occur first.

$\ll \ldots \gg$

$$event_{FEL}\,(FEL) := [id_{event}, t_{event}]$$

Complete definitions of an IO selector and a future events list are not presented.

C.2 DEVS Simulator

Here we present our DEVS simulator as a function, named $simulate$, defined as follows.

$\ll DEVS \gg$

$$
\begin{array}{l}
simulate\left([model, IO_{start}, \Delta n, start, stop, t_{suspend}, starve]\right) := [IO_{last}, last, status_{last}] \\
\qquad [\delta_{ext}, \delta_{int}, ta] := model
\end{array}
$$

The first argument is the DEVS model from which the functions $[\delta_{ext}, \delta_{int}, ta]$ are obtained. The argument IO_{start} is the IO selector described in Section C.1, as it initially exists. The function $simulate$ may be invoked not only to start a new simulation, but also to continue a simulation that was previously started and stopped. We therefore include the argument Δn, the number events between checkpoints when the state of the model is recorded. The argument $start$ is the number of events that have already been processed. If starting a new simulation, $start$ is set to zero. If restarting a simulation from a checkpoint, $start$ should be the appropriate multiple of Δn.

A simulation is complete when the simulated time becomes infinite. There are several ways to stop a simulation early, however. One way uses the argument $stop$, the maximum allowed number of events. Note that $stop = \top$ renders this parameter irrelevant. Another way to end a simulation is to specify a simulated time $t_{suspend}$ at which the simulation is suspended. The boolean $starve$, if truthful, indicates that the simulation is to end when the inputs are exhausted. A programmer should implement the functions of the IO selector such that, if the computer process running the simulation is terminated, the results can be salvaged and the simulation can be restarted from the most recent checkpoint.

The results of $simulate$ include IO_{last} which, as the final version of the IO selector, provides access to the simulation results. The final number of events is given by $last$, and $status_{last}$ reports the condition that ended the simulation.

To start the simulation procedure, the IO selector is first used to get the initial values of $[n_x, n_y, t, s]$. Along with IO_{start} and $start$, these become arguments of the function $loop$.

$\ll \ldots; simulate \gg$

$$[n_{x_{start}}, n_{y_{start}}, t_{start}, s_{start}] := IO_{start}\,(\text{``}input_s\text{''})\,(start)$$
$$[IO_{last}, n_{last}, status_{last}] := loop\,([IO_{start}, start, n_{x_{start}}, n_{y_{start}}, t_{start}, s_{start}])$$

The main simulation loop repeats as long as $status \equiv \varnothing$, updating $[IO, n, n_x, n_y, t, s]$ with each processed event.

$\ll \ldots ; [IO_{last}, n_{last}, status_{last}] \gg$

$$loop\left([IO, n, n_x, n_y, t, s]\right) := \begin{pmatrix} status \not\equiv \varnothing & \rightarrow [IO, n, status] \\ status \equiv \varnothing & \rightarrow loop\left([IO', n', n_x{}', n_y{}', t', s']\right) \end{pmatrix}$$

Inside the loop, it is necessary to obtain the time t_{ext} of the next external transition, as well as the time t_{int} of the next internal transition. The time t_{event} of the next event is the minimum of these. Note that the input value associated with t_{ext} is also acquired.

$\ll \ldots ; loop \gg$

$$\begin{aligned} &[t_{ext}, x] := IO\left(\text{“}input_x\text{”}\right)(n_x) \\ &t_{int} := t + ta(s) \\ &t_{event} = t_{ext} \wedge t_{int} \end{aligned}$$

Various conditions are evaluated to determine whether the simulation should stop.

$\ll \ldots \gg$

$$\begin{aligned} &\alpha_{complete} := (t_{event} = \top) \\ &\alpha_{suspend} := (t_{event} \geq t_{suspend}) \\ &\alpha_{starve} := (starve \wedge (t_{ext} = \top)) \\ &\alpha_{stop} := (n \geq stop) \end{aligned}$$

If any of the conditions are satisfied, $status$ is assigned a corresponding label and the loop terminates. Otherwise, $status$ is assigned $\varnothing$ and the loop continues.

$\ll \ldots \gg$

$$status := \begin{pmatrix} \alpha_{complete} & \rightarrow \text{“}completed\text{”} \\ -\alpha_{complete} \wedge \alpha_{suspend} & \rightarrow \text{“}suspended\text{”} \\ -\alpha_{complete} \wedge -\alpha_{suspend} \wedge \alpha_{starve} & \rightarrow \text{“}starved\text{”} \\ -\alpha_{complete} \wedge -\alpha_{suspend} \wedge -\alpha_{starve} \wedge \alpha_{stop} & \rightarrow \text{“}stopped\text{”} \\ -\alpha_{complete} \wedge -\alpha_{suspend} \wedge -\alpha_{starve} \wedge -\alpha_{stop} & \rightarrow \varnothing \end{pmatrix}$$

The variable $checkpoint$ indicates whether the number of processed events is a multiple of Δn.

$\ll \ldots \gg$

$$checkpoint := \begin{pmatrix} \Delta n = \top & \rightarrow \bot \\ \Delta n < \top & \rightarrow (mod([n, \Delta n]) = 0) \end{pmatrix}$$

If the number of processed events is a multiple of Δn, or if the simulation is at an end, then the state of the simulation is recorded. There is one exception: the state is not recorded if no events whatsoever have been processed since $simulate$ was invoked.

$\ll \ldots \gg$

$$IO_{\Delta} := \begin{pmatrix} record_s & \rightarrow IO\,(\text{“}output_s\text{”})\,([n, n_x, n_y, t, s]) \\ -record_s & \rightarrow IO \end{pmatrix}$$
$$record_s := (n > start) \wedge (checkpoint \vee (status \not\equiv \varnothing))$$

A prominent feature of the simulation loop is the selection of whether the next event is an external transition or an internal transition. This depends on which of the two times is lower, t_{ext} or t_{int}. We adopt the convention that, should the external transition time t_{ext} and internal transition time t_{int} happen to be the same value, the external transition is selected and the internal transition is deferred. This convention seems logical because it ensures that the input value x is available at the earliest possible stage.

$\ll \ldots \gg$

$$[IO', n', {n_x}', {n_y}', t', s'] := \begin{pmatrix} t_{ext} \leq t_{int} & \rightarrow [IO_{\Delta}, n+1, n_x+1, n_y, t_{ext}, s_{ext}] \\ t_{int} < t_{ext} & \rightarrow [IO_{int}, n+1, n_x, n_y + \#Y, t_{int}, s_{int}] \end{pmatrix}$$

If the external transition is selected as the next event, the number of events n and the number of inputs n_x are incremented. Also, the simulated time is advanced to t_{ext}, and the new state of the model s_{ext} is obtained from the external transition function δ_{ext}. Observe below the calculation of the elapsed time.

$\ll \ldots ; [IO', n', {n_x}', {n_y}', t', s'] \gg$

$$s_{ext} := \delta_{ext}\,([s, t_{ext} - t, x])$$

If the next event is an internal transition, the number of events n is incremented and the number of outputs n_y is increased by $\#Y$. Recall that a single internal transition can yield any number of outputs. The variable Y is a vector of output values; hence $\#Y$ is the number of outputs associated with the current event. The new simulated time becomes t_{int}.

As indicated below, the new state s_{int} is obtained along with Y from the internal transition function δ_{int}. Each output value in Y is individually recorded using the IO selector, a procedure defined formally by the function $loop_Y$.

$\ll \ldots \gg$

$$
\begin{aligned}
&[s_{int}, Y] := \delta_{int}(s) \\
&IO_{int} := loop_Y([0, IO_\Delta]) \\
&\quad loop_Y([i, IO_i]) := \left(\begin{array}{ll} i = \#Y & \rightarrow IO_i \\ i < \#Y & \rightarrow loop_Y([i+1, IO_i{}']) \end{array} \right) \\
&\qquad y := Y(i) \\
&\qquad IO_i{}' := IO_i(\text{“}output_y\text{”})([n_y + i, t_{int}, y])
\end{aligned}
$$

The definition of $simulate$ is now complete. In the remainder of this section, we outline an example of how the function is used. Suppose that the hypothetical DEVS model function foo_{DEVS}, described in Section 4.2, has been fully defined. We invoke the function using 75.2 and 4.4 as the two model parameters, which yields the initialization function $init_{foo}$, and the model foo.

$$[init_{foo}, foo] \;\equiv\; foo_{DEVS}([75.2, 4.4])$$

Suppose also that the initialization function requires two initialization parameters. For these we choose 6 and 19.9, obtaining the initial state s_{start}.

$$s_{start} \;\equiv\; init_{foo}([6, 19.9])$$

Finally, suppose that we have three inputs: one occurring at time 2.7 with value “A”, one occurring at time 10.1 with value “B”, and one occurring at time 408.0 with value “C”.

Somehow, an IO selector must be prepared with the properties specified below. The first indicates that after zero events have occurred, there have been zero inputs, zero outputs, the simulated time is zero, and the state of the model at this time is s_{start}. The next three properties indicate that the function $IO_{start}(\text{“}input_x\text{”})$ will deliver the input times and values listed above for the corresponding n_x values of 0, 1, and 2. The last property indicates that, for an n_x value of 3, the time of the corresponding input time is infinite.

$$\begin{aligned}
IO_{start}\left(\text{“}input_s\text{”}\right)(last) &\equiv [0, 0, 0, s_{start}] \\
IO_{start}\left(\text{“}input_x\text{”}\right)(0) &\equiv [2.7, \text{“}A\text{”}] \\
IO_{start}\left(\text{“}input_x\text{”}\right)(1) &\equiv [10.1, \text{“}B\text{”}] \\
IO_{start}\left(\text{“}input_x\text{”}\right)(2) &\equiv [408.0, \text{“}C\text{”}] \\
IO_{start}\left(\text{“}input_x\text{”}\right)(3) &\equiv [\top, \varnothing]
\end{aligned}$$

The following invocation of $simulate$ runs the simulation from the beginning, with checkpoints occurring every 100000 events. There is no limit on the number of events, but the simulation will be suspended if the simulated time reaches 1500. The last argument indicates that the simulation will continue after the three inputs are processed.

$$[IO_{last}, last, status] \equiv simulator\left([foo, IO_{start}, 100000, 0, \top, 1500, \bot]\right)$$

Once the simulation has ended, the final number of inputs processed, the final number outputs delivered, the final simulated time, and the final state can be obtained from the final IO selector.

$$[n_{x_{last}}, n_{y_{last}}, t_{last}, s_{last}] \equiv IO_{last}\left(\text{“}input_s\text{”}\right)(last)$$

There must be some way to access the output values. The output times and values may be added to a text file every time IO_i' is evaluated in the simulator, for example. Ideally, it would be convenient to use the output values of one simulation as input values in a subsequent simulation.

C.3 DEVS Coupled Models

Recall that in our formulation of DEVS, a coupled model has the same form as any other DEVS model, $[\delta_{ext}, \delta_{int}, ta]$, and can therefore be passed to $simulate$ as an argument. The function $coupled_{DEVS}$ accepts the model parameters $[M, C, pr]$. It results in an initialization function, $init_{coupled}$, and the DEVS model itself, $coupled$.

$\ll DEVS \gg$

$$coupled_{DEVS}\left([M, C, pr]\right) := [init_{coupled}, coupled]$$

The diagram in Figure 36 serves as an example of how a coupled model's structure is represented by the parameters M and C. In the diagram, "A", "B", and "C" are the names of submodels and "r", "p", "q", "u", "v", and "w" are names of ports. Arrows representing links connect a port of one model, either a submodel or the coupled model itself, to the port of another model.

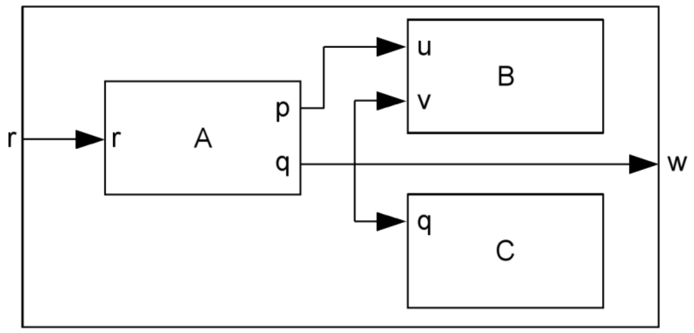


Figure 36: An example of a coupled DEVS model.

The selector M takes the name of a submodel as its argument, and results in the associated DEVS model. For the example of Figure 36, M would take the form below.

$$M \equiv \begin{bmatrix} \text{“}A\text{”} & \rightarrow [\delta_{ext_A}, \delta_{int_A}, ta_A] \\ \text{“}B\text{”} & \rightarrow [\delta_{ext_B}, \delta_{int_B}, ta_B] \\ \text{“}C\text{”} & \rightarrow [\delta_{ext_C}, \delta_{int_C}, ta_C] \end{bmatrix}$$

The parameter C can be either a regular function or a selector that represents the links in a coupled model. It does this by mapping a source model ID and port to a vector of destination models and ports. The first mapping below represents the arrow on the far left in Figure 36. The source $[\varnothing, \text{“}r\text{”}]$ represents the "r" port of the coupled model itself, and the destination $[\text{“}A\text{”}, \text{“}r\text{”}]$ is the "r" port of submodel "A". Note that $C\left([\text{“}A\text{”}, \text{“}q\text{”}]\right)$ is a vector of three destinations: the "v" port of submodel "B", the "w" port of the coupled model, and the "q" port of submodel "C".

$$C \equiv \begin{bmatrix} [\varnothing, \text{“}r\text{”}] & \rightarrow [[\text{“}A\text{”}, \text{“}r\text{”}]] \\ [\text{“}A\text{”}, \text{“}p\text{”}] & \rightarrow [[\text{“}B\text{”}, \text{“}u\text{”}]] \\ [\text{“}A\text{”}, \text{“}q\text{”}] & \rightarrow \begin{bmatrix} [\text{“}B\text{”}, \text{“}v\text{”}] \\ [\varnothing, \text{“}w\text{”}] \\ [\text{“}C\text{”}, \text{“}q\text{”}] \end{bmatrix} \end{bmatrix}$$

The priority function pr will be explained in Section C.4. The remainder of this section is devoted to the definition of $init_{coupled}$ and $coupled$.

A coupled model requires a future events list to track the timing of the future internal transitions of its submodels. We therefore ensure the $future\_events\_list$ functions are in scope.

$\ll \ldots; coupled_{DEVS} \gg$

$$\begin{bmatrix} FEL_{empty} \\ \delta_{FEL} \\ event_{FEL} \end{bmatrix} := future\_events\_list \circ \begin{bmatrix} \text{“}FEL_{empty}\text{”} \\ \text{“}\delta_{FEL}\text{”} \\ \text{“}event_{FEL}\text{”} \end{bmatrix}$$

The initial state s of a coupled model includes the initial states of each of its submodels. These initial states are provided to the initialization function $init_{coupled}$ by the initialization parameter S. A selector with the same domain as M, S maps a submodel ID to the initial state of the submodel.

$\ll \ldots \gg$

$$init_{coupled}(S) := s$$

Coupled models have four state variables, as indicated below.

$\ll \ldots; init_{coupled} \gg$

$$s := [t, T, S, FEL]$$

The state variable t is the simulated time of the previous event. It is initially zero. It is also necessary to record the simulated time of the previous event of each submodel. These times are recorded by the selector T. Initially, T yields zero for all submodel IDs.

$\ll \ldots; s \gg$

$$\begin{aligned} &t := 0 \\ &T\left(id_i\right) := \left(\begin{array}{ll} id_i : \Diamond M & \rightarrow 0 \end{array}\right) \end{aligned}$$

The state variable S records the current state of each submodel. Its initial value is, of course, provided by the initialization parameter. The remaining state variable is the future events list FEL, which is populated with the future internal transition times. The loop below iterates through each submodel, obtaining the transition time t_{int_i} using the initial state and the time advance function. Because each submodel has only one associated internal transition, the submodel ID id_i is an appropriate event ID for the FEL.

$\ll \ldots \gg$

$$\begin{aligned} &FEL := loop\left(\left[0, FEL_{empty}\right]\right) \\ &\quad ID_M := \Diamond M \\ &\quad loop\left(\left[i, FEL_i\right]\right) := \left(\begin{array}{ll} i = \#ID_M & \rightarrow FEL_i \\ i < \#ID_M & \rightarrow loop\left(\left[i+1, FEL_i{}'\right]\right) \end{array}\right) \\ &\qquad id_i := ID_M\left(i\right) \\ &\qquad s_i := S\left(id_i\right) \\ &\qquad \left[\delta_{ext_i}, \delta_{int_i}, ta_i\right] := M\left(id_i\right) \\ &\qquad t_{int_i} := ta_i\left(s_i\right) \\ &\qquad FEL_i{}' := \delta_{FEL}\left(\left[FEL_i, id_i, t_{int_i}, pr\right]\right) \end{aligned}$$

The initialization function $init_{coupled}$ now defined, we turn our attention to the DEVS model $coupled$.

$\ll DEVS; coupled_{DEVS} \gg$

$$coupled := \left[\delta_{ext}, \delta_{int}, ta\right]$$

Because messages are passed among the submodels in both δ_{ext} and δ_{int}, we start by defining reusable functions that address message-passing. The function map_{src_dst} maps source messages to the corresponding destination messages.

$\ll \ldots; coupled \gg$

$$map_{src\_dst}\left(\left[id_{src}, Y_{src}\right]\right) := ID\_X_{dst}$$

Looking at the example in Figure 36, suppose that submodel "A" has just undergone an internal transition which yielded three outputs. The first output is from the "p" port with a value of 72.9, the second is from the "q" port with a value of -4, and the third is also from the "q" port but carries a value of 15. As shown below, map_{src_dst} gives the destinations of each output.

$$map_{src\_dst}\left(\left[\begin{array}{cc} \text{“}A\text{”}, & \left[\begin{array}{l} [\text{“}p\text{”}, 72.9] \\ [\text{“}q\text{”}, -4] \\ [\text{“}q\text{”}, 15] \end{array}\right] \end{array}\right]\right) := \left[\begin{array}{l} [\text{“}B\text{”}, [\text{“}u\text{”}, 72.9]] \\ [\text{“}B\text{”}, [\text{“}v\text{”}, -4]] \\ [\varnothing, [\text{“}w\text{”}, -4]] \\ [\text{“}C\text{”}, [\text{“}q\text{”}, -4]] \\ [\text{“}B\text{”}, [\text{“}v\text{”}, 15]] \\ [\varnothing, [\text{“}w\text{”}, 15]] \\ [\text{“}C\text{”}, [\text{“}q\text{”}, 15]] \end{array}\right]$$

The first output of "A", represented by $[\text{“}p\text{”}, 72.9]$, becomes an input for the "u" port of submodel "B". Because the output of "A"'s "q" port forks three ways in Figure 36, the second and third outputs each have three destinations. Two of these destinations are input ports of other submodels, but one is the output port "w" of the overall coupled model. Note that $\varnothing$ identifies the coupled model itself as a message destination.

The result of map_{src_dst}, the vector of destination messages, is named ID_X_{dst} and defined with two nested loops. The outer loop, $loop_{src}$, iterates through each output of the message source. For each output, the model parameter C is used to obtain the destinations.

$\ll \ldots; map_{src\_dst} \gg$

$$ID\_X_{dst} := loop_{src}\left([0, [\,]]\right)$$

$$loop_{src}\left([i, ID\_X_{dst_i}]\right) := \left(\begin{array}{ll} i = \#Y_{src} & \rightarrow ID\_X_{dst_i} \\ i < \#Y_{src} & \rightarrow loop_{src}\left([i+1, ID\_X_{dst_i}{}']\right) \end{array}\right)$$

$$y_{src} := Y_{src}\left(i\right)$$

$$[port_{src}, msg] := y_{src}$$

$$ID\_PORT_{dst} := C\left([id_{src}, port_{src}]\right)$$

$$ID\_X_{dst_i}{}' := loop_{dst}\left([0, ID\_X_{dst_i}]\right)$$

The inner loop, $loop_{dst}$, iterates through each destination of a given source message. For each destination, a message is appended onto an intermediate version of ID_X_{dst}.

$\ll \ldots; ID\_X_{dst}; loop_{src}; ID\_X_{dst_i}{}' \gg$

$$loop_{dst}\left(\left[j, ID\_X_{dst_{ij}}\right]\right) := \left(\begin{array}{ll} j = \#ID\_PORT_{dst} & \rightarrow ID\_X_{dst_{ij}} \\ j < \#ID\_PORT_{dst} & \rightarrow loop_{dst}\left(\left[j+1, ID\_X_{dst_{ij}}{}'\right]\right) \end{array}\right)$$
$$[id_{dst}, port_{dst}] := ID\_PORT_{dst}(j)$$
$$ID\_X_{dst_{ij}}{}' := ID\_X_{dst_{ij}} \parallel [[id_{dst}, [port_{dst}, msg]]]$$

Now that we have completed map_{src_dst}, which maps source messages to destination messages, we define the function $update_{dst}$, which triggers the external transitions for each submodel receiving a message. Its arguments include the initial state s of the coupled model, the ID id_{src} of the source of a set of messages, and the vector Y_{src} containing those messages. The result is the coupled model's new state s', and the vector Y of messages leaving the coupled model.

$\ll DEVS; coupled_{DEVS}; coupled \gg$

$$update_{dst}([s, id_{src}, Y_{src}]) := [s', Y]$$

After the state variables are extracted from s, the destination messages are extracted from id_{src} and Y_{src} using map_{src_dst}. A loop is then used to obtain $[s', Y]$.

$\ll \ldots; update_{dst} \gg$

$$[t, T, S, FEL] := s$$
$$ID\_X_{dst} := map_{src\_dst}([id_{src}, Y_{src}])$$
$$[s', Y] := loop([0, T, S, FEL, [\,]])$$

The loop iterates over each destination message in ID_X_{dst}. For each message, the variable id_{dst} identifies either the receiving submodel or, if $\varnothing$, the coupled model itself. The port and value of the message are contained in x_{dst}.

$\ll \ldots; [s', Y] \gg$

$$loop([i, T_i, S_i, FEL_i, Y_i]) := \left(\begin{array}{ll} i = \#ID\_X_{dst} & \rightarrow [[t, T_i, S_i, FEL_i], Y_i] \\ i < \#ID\_X_{dst} & \rightarrow loop([i+1, T_i', S_i', FEL_i', Y_i']) \end{array}\right)$$
$$[id_{dst}, x_{dst}] := \#ID\_X_{dst}(i)$$

If the message is destined to be an output of the coupled model, x_{dst} is appended to an intermediate version of Y. If the message is destined for a submodel, then three of the state variables are updated.

$\ll \ldots ; \mathit{loop} \gg$

$$[T_i', S_i', FEL_i', Y_i'] := \left(\begin{array}{lll} id_{dst} & \equiv \varnothing & \rightarrow [T_i, S_i, FEL_i, Y_i \parallel [x_{dst}]] \\ id_{dst} & \not\equiv \varnothing & \rightarrow [T_\Delta, S_\Delta, FEL_\Delta, Y_i] \end{array} \right)$$

In the latter of the two cases above, we obtain the transition functions and time advance function associated with the destination submodel, the time t_{dst} of that submodel's previous transition, and the submodel's current state s_{dst}.

$\ll \ldots ; [T_i', S_i', FEL_i', Y_i'] \gg$

$$\begin{aligned} &[\delta_{ext_{dst}}, \delta_{int_{dst}}, ta_{dst}] := M\,(id_{dst}) \\ &t_{dst} := T_i\,(id_{dst}) \\ &s_{dst} := S_i\,(id_{dst}) \end{aligned}$$

The new state of the submodel, s_{dst}', is obtained from its external transition function. Note the use of t_{dst} to compute the elapsed time.

$\ll \ldots \gg$

$$s_{dst}' := \delta_{ext_{dst}}\,([s_{dst}, t - t_{dst}, x_{dst}])$$

The state variable T is updated to reflect the new submodel transition time, and S is modified with the new submodel state.

$\ll \ldots \gg$

$$\begin{aligned} T_\Delta &:= T_i \lhd \left[\ id_{dst} \ \rightarrow t\ \right] \\ S_\Delta &:= S_i \lhd \left[\ id_{dst} \ \rightarrow s_{dst}'\ \right] \end{aligned}$$

The time advance function of the submodel is invoked with its new state, and the resulting time is used to schedule the submodel's next internal transition in the FEL.

$\ll \ldots \gg$

$$\begin{aligned} FEL_\Delta &:= \delta_{FEL}\,([FEL_i, id_{dst}, t_{int_{dst}}, pr]) \\ t_{int_{dst}} &:= t + ta_{dst}\,(s_{dst}') \end{aligned}$$

With update_{dst} at our disposal, the transition functions of $\mathit{coupled}$ are relatively easy to define. We start with the external transition function.

$\ll DEVS; coupled_{DEVS}; coupled \gg$

$$\delta_{ext}\left([s, \Delta t_{el}, x]\right) := s'$$

First, the state variables are extracted from s. Second, the current time is advanced by the elapsed time to yield an intermediate state s_Δ.

$\ll \ldots; \delta_{ext} \gg$

$$\begin{aligned}
[t, T, S, FEL] &:= s \\
s_\Delta &:= [t_\Delta, T, S, FEL] \\
t_\Delta &:= t + \Delta t_{el}
\end{aligned}$$

The $update_{dst}$ function takes care of the rest, invoking external transitions on each submodel receiving a message from the input of the coupled model.

$\ll \ldots \gg$

$$[s', Y] := update_{dst}\left([s_\Delta, \varnothing, [x]]\right)$$

Next we define the internal transition function.

$\ll DEVS; coupled_{DEVS}; coupled \gg$

$$\delta_{int}\left(s\right) := [s', Y]$$

The state variables are obtained, along with the ID id_{src} of the next submodel to undergo an internal transition, and the time t_Δ of that transition. This submodel is the source model. Its transition functions, time advance function, and current state are obtained.

$\ll \ldots; \delta_{int} \gg$

$$\begin{aligned}
&[t, T, S, FEL] := s \\
&[id_{src}, t_\Delta] := event_{FEL}\left(FEL\right) \\
&[\delta_{ext_{src}}, \delta_{int_{src}}, ta_{src}] := M\left(id_{src}\right) \\
&s_{src} := S\left(id_{src}\right)
\end{aligned}$$

The internal transition function of the source submodel is invoked to yield its new state ${s_{src}}'$ and its outputs Y_{src}.

$\ll \ldots \gg$

$$[{s_{src}}', Y_{src}] := \delta_{int_{src}}\left(s_{src}\right)$$

An intermediate state s_Δ is prepared that accounts for the internal transition of the source submodel.

$\ll \ldots \gg$

$$\begin{aligned} s_\Delta &:= [t_\Delta, T_\Delta, S_\Delta, FEL] \\ &\quad T_\Delta := T \lhd \left[\ id_{src} \ \ \rightarrow t_\Delta \ \right] \\ &\quad S_\Delta := S \lhd \left[\ id_{src} \ \ \rightarrow s_{src}{}' \ \right] \\ &\quad FEL_\Delta := \delta_{FEL}\left([FEL, id_{src}, t_{int_{src}}, pr]\right) \\ &\qquad t_{int_{src}} := t_\Delta + ta_{src}\left(s_{src}{}'\right) \end{aligned}$$

The $update_{dst}$ function updates the state again, this time accounting for the external transitions of the submodels receiving messages from the source.

$\ll \ldots \gg$

$$[s', Y] := update_{dst}\left([s_\Delta, id_{src}, Y_{src}]\right)$$

The time advance function of a coupled model queries the FEL, reporting the time remaining until the next submodel undergoes an internal transition.

$\ll DEVS; coupled_{DEVS}; coupled \gg$

$$\begin{aligned} ta\left(s\right) &:= t_{event} - t \\ &\quad [t, T, S, FEL] := s \\ &\quad [id_{event}, t_{event}] := event_{FEL}\left(FEL\right) \end{aligned}$$

C.4 Priority Functions

A priority function of either an FEL or a coupled model takes two IDs as arguments, and results in one of these IDs or $\varnothing$. Suppose we are given, for some priority function pr, that $pr\left([\text{“}A\text{”}, \text{“}B\text{”}]\right) \equiv \text{“}B\text{”}$. If “A” and “B” are the IDs of two events occurring at the same time, and pr was used to populate an FEL with both events, then the event labelled “B” occurs first. If “A” and “B” are the IDs of two submodels with internal transitions occurring at the same time, then the internal transition of the submodel labelled “B” occurs first. If $pr\left([\text{“}A\text{”}, \text{“}B\text{”}]\right) \equiv \varnothing$, then the first event/transition is chosen randomly.

In some cases, we want all simultaneous events to occur in a random order. We therefore define a priority function named $pr_\varnothing$ that always results in $\varnothing$.

$\ll DEVS \gg$

$$pr_{\varnothing}\left([id_A, id_B]\right) := \varnothing$$

In the case of a non-spatial coupled DEVS model in which the submodels have distinct roles, one might hesitate to order simultaneous transitions randomly. But in a cellular DEVS model, one in which each lattice cell has its own DEVS model, the use of $pr_{\varnothing}$ might prevent a spatial bias from emerging in simulation results. Because the order of lattice cells would be chosen randomly, no spatial region would be favoured over any other in the resolution of simultaneous events.

Note that the $future\_events\_list$ functions must address the possibility that the order of three or more simultaneous events must be randomized. When comparing two such events, it would likely be an error to simply select one of the two to occur first. Instead, a uniform random number should be associated with each simultaneous event, and the order of those events should be based on those numbers. For the sake of efficiently, a random number should only be generated for an event once it is confirmed that there is at least one other event occurring at the same time.

Sorting events using numbers is convenient in many situations, regardless of whether those numbers are random. We therefore define a function pr_Z to assist this approach. Although pr_Z has a third argument, and cannot be used as an argument for δ_{FEL} or $coupled_{DEVS}$, the function can be used to construct a two-argument priority function. The third argument of pr_Z is a function named Z that maps IDs to numbers.

$\ll DEVS \gg$

$$\begin{aligned} pr_Z\left([id_A, id_B, Z]\right) &:= id_{pr} \\ z_A &:= Z\left(id_A\right) \\ z_B &:= Z\left(id_B\right) \end{aligned}$$

The ID with the smaller associated number is the result of pr_Z, the ID of the event/submodel with priority. If the numbers are equal, then the result is $\varnothing$.

$\ll \ldots; pr_Z \gg$

$$id_{pr} := \begin{pmatrix} z_A < z_B & \rightarrow id_A \\ z_A > z_B & \rightarrow id_B \\ z_A = z_B & \rightarrow \varnothing \end{pmatrix}$$

Another way to construct a priority function is to exploit the function pr_{order}. In this case the third argument is a vector, named $order$, that contains IDs. Priority is given to the event/submodel that appears first in the vector.

$\ll DEVS \gg$

$$pr_{order}\left([id_A, id_B, order]\right) := id_{pr}$$

The result is obtained by iterating through each ID in $order$ until either id_A or id_B is found. If neither ID is found, the result is $\varnothing$.

$\ll \ldots; pr_{order} \gg$

$$\begin{aligned}
&id_{pr} := loop\left(0\right)\\
&\quad loop\left(i\right) := \begin{pmatrix} i = \#order & \rightarrow \varnothing \\ i < \#order & \rightarrow check \end{pmatrix}\\
&\quad\quad check := \begin{pmatrix} order\left(i\right) \ \equiv \ id_A & \rightarrow id_A \\ order\left(i\right) \ \equiv \ id_B & \rightarrow id_B \\ \left(order\left(i\right) \ \not\equiv \ id_A\right) \wedge \left(order\left(i\right) \ \not\equiv \ id_B\right) & \rightarrow loop\left(i+1\right) \end{pmatrix}
\end{aligned}$$

C.5 DEVS Hypercubic Lattice Models

A DEVS hypercubic lattice model is a coupled model in which submodels are arranged conceptually in a hypercubic lattice of n_{dim} dimensions. Each submodel interacts with its, at most, $2 \cdot n_{dim}$ adjacent neighbors.

The DEVS model function HL_{DEVS} is similar to the hypothetical bar_{DEVS} function of Section 4.2; it results in a DEVS model of the form $[\delta_{ext}, \delta_{int}, ta]$, but defines none of these functions directly. Instead, it transforms one set of model parameters into another set of model parameters, then uses another DEVS model function to transform the second set into the DEVS model. In this case the parameters $[N, HLm_{DEVS}, pr]$ are transformed into $[M, C, pr]$, and $coupled_{DEVS}$ is used to obtain $[\delta_{ext}, \delta_{int}, ta]$.

$\ll DEVS \gg$

$$HL_{DEVS}\left([N, HLm_{DEVS}, pr]\right) := [init_{HL}, HL]$$

The parameter N is a vector listing the dimensions of the hypercubic lattice. From N one can obtain the number of submodels, as well as the coordinates of each submodel. A submodel's coordinates serve both as its ID and as the parameters of the DEVS model function HLm_{DEVS}. By applying HLm_{DEVS} to a set of coordinates, one obtains the DEVS model of the submodel at those coordinates. This is done below to define M.

$\ll \ldots \gg$

$$M\left(coords\right) := \left(\ coords : ..N \ \ \rightarrow HLm\ \right)$$
$$\left[init_{HLm}, HLm\right] := HLm_{DEVS}\left(coords\right)$$

The main reason why we bother introducing HL_{DEVS} is that it introduces a layer which handles links between submodels in a hypercubic lattice. One can use HL_{DEVS} to define a range of different cellular models, effectively reusing the coupled model parameter C.

$\ll \ldots \gg$

$$C\left(\left[id_{src}, port_{src}\right]\right) := ID\_PORT_{dst}$$

For DEVS hypercubic lattice models, we divide port values into two components: a general component and a specific component.

$\ll \ldots ; C \gg$

$$\left[general_{src}, specific_{src}\right] := port_{src}$$

A message can be classified into one of five categories according to its source and the general component of its port. Messages sent to the hypercubic lattice model ($id_{src} \equiv \varnothing$) can be directed to a single submodel ($general_{src} \equiv$ "*one*"), a submodel on the edge ($general_{src} \equiv$ "*adj*"), or all submodels ($general_{src} \equiv$ "*all*"). Messages originating from a submodel can be directed out of the hypercubic lattice model ($general_{src} \equiv$ "*out*"), or to an adjacent submodel ($general_{src} \equiv$ "*adj*").

$\ll \ldots \gg$

$$ID\_PORT_{dst} := \left(\begin{array}{lll} id_{src} \equiv \varnothing & \rightarrow & \left(\begin{array}{lll} general_{src} \equiv \text{"one"} & \rightarrow C_{one} \\ general_{src} \equiv \text{"adj"} & \rightarrow C_{edge} \\ general_{src} \equiv \text{"all"} & \rightarrow C_{all} \end{array}\right) \\ id_{src} \not\equiv \varnothing & \rightarrow & \left(\begin{array}{lll} general_{src} \equiv \text{"out"} & \rightarrow C_{out} \\ general_{src} \equiv \text{"adj"} & \rightarrow C_{adj} \end{array}\right) \end{array}\right)$$

If a message entering a hypercubic lattice model is sent to a single submodel ($general_{src} \equiv$

"*one*"), the specific component of the port must contain the coordinates of that submodel. The submodel receives the message with a two-component port, the first component being "*in*" and the second component depending on the source port.

$\ll \ldots; ID\_PORT_{dst} \gg$

$$\begin{aligned}
&C_{one} := [[coords_{dst}, port_{dst}]] \\
&\quad [coords_{dst}, specific_{dst}] := specific_{src} \\
&\quad port_{dst} := [\text{“}in\text{”}, specific_{dst}]
\end{aligned}$$

The "*adj*" inputs allow multiple DEVS hypercubic lattice models to be linked together. Messages sent off the edge of one hypercubic lattice model may end up arriving at the edge of the adjacent hypercubic lattice model. In this case the port includes a dimension index i_{dim}, a direction dr, and a projection $proj$. Suppose we have a DEVS hypercubic lattice model with $N \equiv [4, 4, 4]$. An "*adj*" message is received with $i_{dim} = 1$, indicating the second dimension, and $proj \equiv [2, \varnothing, 1]$. If $dr = 1$, the "*adj*" message is received by the submodel with coordinates $[2, 0, 1]$. If $dr = -1$, the message is received by submodel $[2, 3, 1]$. From the perspective of the receiving submodel, an "*adj*" message redirected from the input of a hypercubic lattice model has the same form as a message arriving from an adjacent submodel.

$\ll \ldots \gg$

$$\begin{aligned}
&C_{edge} := [[coords_{dst}, port_{dst}]] \\
&\quad [i_{dim}, dr, proj] := specific_{src} \\
&\quad specific_{dst} := [i_{dim}, dr] \\
&\quad coords_{dst} := proj \lhd \left[\; i_{dim} \quad \to \frac{1-dr}{2} \cdot (N(i_{dim}) - 1) \;\right] \\
&\quad port_{dst} := [\text{“}adj\text{”}, specific_{dst}]
\end{aligned}$$

A loop is used to distribute "*all*" inputs to each hypercubic lattice submodel. From a submodel's point of view, receiving a message in this manner is identical to receiving a "*one*" message. Note that we need not worry about the order in which the submodels receive an "*all*" message, as only external transitions occur. It is only the order of internal transitions that must be controlled.

$\ll \ldots \gg$

$$\begin{aligned} & C_{all} := loop\left([0, [\,]]\right) \\ & \quad ID_M := \Diamond..N \\ & \quad loop\left([i, ID\_PORT_{dst_i}]\right) := \left(\begin{array}{ll} i = \#ID_M & \rightarrow ID\_PORT_{dst_i} \\ i < \#ID_M & \rightarrow loop\left([i+1, ID\_PORT_{dst_i}{}']\right) \end{array} \right) \\ & \qquad coords_{dst} := ID_M\left(i\right) \\ & \qquad port_{dst} := [\text{“}in\text{”}, specific_{src}] \\ & \qquad ID\_PORT_{dst_i}{}' := ID\_PORT_{dst_i} \parallel [[coords_{dst}, port_{dst}]] \end{aligned}$$

If an “*out*” message is sent from a submodel to an output of the hypercubic lattice model that contains it, the port of the message leaving the lattice is the same as the port leaving the submodel.

$\ll \ldots \gg$

$$C_{out} := [[\varnothing, port_{src}]]$$

If an “*adj*” message is sent from a submodel to an adjacent submodel, we first check if that adjacent submodel exists. If it does, we evaluate C_{inside}. If message is being sent off the edge of the hypercubic lattice, then the receiving submodel does not exist and we evaluate C_{edge}.

$\ll \ldots \gg$

$$\begin{aligned} & C_{adj} := \left(\begin{array}{ll} 0 \leq depth < N\left(i_{dim}\right) & \rightarrow C_{inside} \\ -\left(0 \leq depth < N\left(i_{dim}\right)\right) & \rightarrow C_{edge} \end{array} \right) \\ & \quad [i_{dim}, dr] := specific_{src} \\ & \quad depth := id_{src}\left(i_{dim}\right) + dr \end{aligned}$$

For the C_{inside} case, the “*adj*” message received by the adjacent submodel has the dimension index i_{dim} and direction dr in its port.

$\ll \ldots ; C_{adj} \gg$

$$\begin{aligned} & C_{inside} := [coords_{dst}, port_{dst}] \\ & \quad coords_{dst} := id_{src} \lhd \left[\begin{array}{ll} i_{dim} & \rightarrow depth \end{array} \right] \\ & \quad port_{dst} := [\text{“}adj\text{”}, specific_{src}] \end{aligned}$$

In the case of C_{edge}, the “*adj*” message is output from the hypercubic lattice model. The port contains the projection $proj$ as well as the dimension and direction.

$\ll \ldots \gg$

$$\begin{aligned}
C_{edge} &:= [\varnothing, port_{dst}] \\
port_{dst} &:= [\text{“}adj\text{”}, specific_{dst}] \\
specific_{dst} &:= [i_{dim}, dr, proj] \\
proj &:= id_{src} \lhd \left[\; i_{dim} \quad \to \varnothing \;\right]
\end{aligned}$$

At this point M and C have been defined, and pr was passed in at the start as an argument. The $coupled_{DEVS}$ function can therefore be invoked to yield the DEVS model and an initialization function.

$\ll DEVS; HL_{DEVS} \gg$

$$[init_{coupled}, HL] := coupled_{DEVS}\left([M, C, pr]\right)$$

The DEVS model HL is now defined, but we are still in need of the initialization function $init_{HL}$. It takes as its initialization parameter a function named ARG_{HLm}. The function ARG_{HLm} takes a set of coordinates $coords_{HLm}$ as its argument, and results in a new set of initialization parameters. This new set of parameters, $ARG_{HLm}\left(coords_{HLm}\right)$, is used to initialize the state of the submodel at $coords_{HLm}$.

To define $init_{HL}$, we map ARG_{HLm} into S, then use $init_{coupled}$.

$\ll \ldots \gg$

$$\begin{aligned}
init_{HL}\left(ARG_{HLm}\right) &:= s \\
s &:= init_{coupled}\left(S\right)
\end{aligned}$$

The selector S is constructed by iterating over each submodel. The initial state s_{HLm} of each submodel is obtained by evaluating $init_{HLm}\left(args\right)$, where $init_{HLm}$ comes from HLm_{DEVS} and $args$ is from $ARGS_{HLm}$.

$\ll \ldots; init_{HL}; s \gg$

```
$S := loop\,([0, [\,]])$
    $ID_M := \Diamond..N$
    $loop\,([i, S_i]) := \begin{pmatrix} i = \#ID_M & \rightarrow S_i \\ i < \#ID_M & \rightarrow loop\,([i+1, S_i{}']) \end{pmatrix}$
        $coords_{HLm} := ID_M\,(i)$
        $args := ARGS_{HLm}\,(coords_{HLm})$
        $[init_{HLm}, HLm] := HLm_{DEVS}\,(coords_{HLm})$
        $s_{HLm} := init_{HLm}\,(args)$
        $S_i{}' := S_i \lhd \left[\; coords_{HLm} \;\; \rightarrow s_{HLm} \;\right]$
```

D DEVS TPS Model Formulas

D.1 DEVS TPS Functions

Defined below, the selector $DEVS\_tethered\_particle\_system$ provides access to the DEVS model functions yielding the TPS model and its various submodels.

$$DEVS\_tethered\_particle\_system := \ldots$$
$$\begin{bmatrix} \text{``}TPS_{DEVS}\text{''} & \rightarrow TPS_{DEVS} \\ & \{\text{DEVS TPS model (Section D.2)}\} \\ \text{``}detector_{DEVS}\text{''} & \rightarrow detector_{DEVS} \\ & \{\text{DEVS detector model (Section D.2)}\} \\ \text{``}RI_{DEVS}\text{''} & \rightarrow RI_{DEVS} \\ & \{\text{DEVS random impulse model (Section D.3)}\} \\ \text{``}responder_{DEVS}\text{''} & \rightarrow responder_{DEVS} \\ & \{\text{DEVS responder model (Section D.4)}\} \\ \text{``}tracker_{DEVS}\text{''} & \rightarrow tracker_{DEVS} \\ & \{\text{DEVS tracker model (Section D.5)}\} \\ \text{``}lattice_{DEVS}\text{''} & \rightarrow lattice_{DEVS} \\ & \{\text{DEVS lattice model (Section D.6)}\} \end{bmatrix}$$

The variables and parameters of the TPS are similar to those in Appendix B. Used frequently, the variables id_A and id_B are integers that identify particles, and Ψ contains information on each particle including its velocity and former position. The model parameters $\theta_{resolve}$, Δt_{max}, and $\Delta t_{restitute}$ are the same here as in Appendix B, but two notable functions are now introduced.

$$\begin{aligned} &attach\left([id_A, id_B, \Psi, \Phi, \Omega_\psi, \Omega_{\psi\psi}]\right) && \{\text{if true, particles become tethered}\} \\ &detach\left([id_A, id_B, \Psi, \Phi, \Omega_\psi, \Omega_{\psi\psi}]\right) && \{\text{if true, particles separate}\} \end{aligned}$$

The function $attach$ is a predicate, resulting in $\top$ if two particles are to become tethered, and $\bot$ otherwise. It is invoked at the simulated time when two approaching particles undergo a blocking collision. Its arguments include the IDs of both particles, Ψ, a value named Φ that will be discussed later, and the particle property selectors Ω_{ψ} and $\Omega_{\psi\psi}$. The $detach$ function takes the same arguments, but is invoked when two separating tethered particles reach their tethering distance. If $\top$, the particles cease to be tethered and continue separating with no applied impulses. If $detach$ results in $\perp$, the particles remain tethered and therefore retract or revolve.

In Appendix B we defined a function to detect collisions between two specific particles, but did not worry about applying the function to various particle pairs in a system of many particles. As explained in Section 4.3, collision detection is to be performed using a lattice of subvolumes. The dimensions of the lattice are given by N, the first parameter of the HL_{DEVS} function of Appendix C. The length of each side of each subvolume is a, so $N \cdot a$ gives the lengths of the sides of the overall region. The center of the overall region is $\vec{u}_{center}$. The function $corner$, defined below, takes these three parameters as arguments and yields the position $\vec{u}_{corner}$ of the lower corner of the overall region.

$\ll DEVS\_tethered\_particle\_system \gg$

$$corner\left([N, a, \vec{u}_{center}]\right) := \vec{u}_{corner}$$
$$\vec{u}_{corner} := \vec{u}_{center} - \frac{N}{2} \cdot a$$

Recall from Figures 17 and 18 of Section 4.3 that, conceptually, there are two circles (or spheres) around each square (or cubic) subvolume. We will refer to the inner circle or sphere as the “arrival orb”. The outer circle/sphere is the “departure orb”. The size of the arrival and departure orbs is determined by two values: $\epsilon_{arrival}$ and $\epsilon_{departure}$. The values are arbitrary; as long as they are relatively small, and $\epsilon_{arrival} < \epsilon_{departure}$, they should have little influence on simulation results.

$\ll \ldots \gg$

$$\epsilon_{arrival} := \frac{1}{64}$$
$$\epsilon_{departure} := \frac{1}{8}$$

The radius of the arrival orb is $R_{arrival}$, and $R_{departure}$ is the radius of the departure orb. The function $radius$ calculates these values using $\epsilon_{arrival}$ and $\epsilon_{departure}$. Note that R is the radius of a circle/sphere circumscribed around a subvolume.

$\ll \ldots \gg$

$$\begin{aligned} & radii\left([n_{dim}, a]\right) := [R_{arrival}, R_{departure}] \\ & \quad R := \frac{\sqrt{n_{dim}}}{2} \cdot a \\ & \quad R_{arrival} := (1 + \epsilon_{arrival}) \cdot R \\ & \quad R_{departure} := (1 + \epsilon_{departure}) \cdot R \end{aligned}$$

Recall from Appendix B that Ω_ψ contains sets of properties associated with each particle species, and $\Omega_{\psi\psi}$ contains sets of properties associates with pairs of particle species. The TPS model parameters include the two very similar selectors ω_ψ and $\omega_{\psi\psi}$.

$$\begin{array}{ll} \omega_\psi & \{\text{properties of individual particle species}\} \\ \omega_{\psi\psi} & \{\text{properties of pairs of particle species}\} \end{array}$$

For each particle species, we require ω_ψ to contain the mass of each particle, but do not require it to contain the random impulse parameters τ_{RI}, k_{RI}, μ_{RI}. If it contains any of these random impulse parameters, however, then it should contain all three.

$$\begin{array}{ll} \omega_\psi\left(spc_A\right)\left(\text{“}m\text{”}\right) & \{\text{required}\} \\ \omega_\psi\left(spc_A\right)\left(\text{“}\tau_{RI}\text{”}\right) & \{\text{optional}\} \\ \omega_\psi\left(spc_A\right)\left(\text{“}k_{RI}\text{”}\right) & \{\text{optional}\} \\ \omega_\psi\left(spc_A\right)\left(\text{“}\mu_{RI}\text{”}\right) & \{\text{optional}\} \end{array}$$

Given ω_ψ and $\omega_{\psi\psi}$, Ω_ψ and $\Omega_{\psi\psi}$ are calculated automatically. The selector Ω_ψ differs from ω_ψ in that it must contain all for of the parameters listed above. Also, for each particle species, Ω_ψ contains the "detection radius". Detection radii are illustrated in Section 4.3's Figure 17 and Figure 18. They help determine the distance at which subvolumes lose or gain awareness of particles. For a species identified by spc_A, the detection radius is $\Omega_\psi\left(spc_A\right)\left(\text{“}r\text{”}\right)$.

$$\begin{array}{l} \Omega_\psi\left(spc_A\right)\left(\text{“}m\text{”}\right) \\ \Omega_\psi\left(spc_A\right)\left(\text{“}\tau_{RI}\text{”}\right) \\ \Omega_\psi\left(spc_A\right)\left(\text{“}k_{RI}\text{”}\right) \\ \Omega_\psi\left(spc_A\right)\left(\text{“}\mu_{RI}\text{”}\right) \\ \Omega_\psi\left(spc_A\right)\left(\text{“}r\text{”}\right) \end{array}$$

Both $\omega_{\psi\psi}$ and $\Omega_{\psi\psi}$ contain, for each pair of particle species, all three restitution coefficients and a blocking distance and a tethering distance.

$$\begin{array}{ll}
\omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{rebound}\text{''}\right) & \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{rebound}\text{''}\right) \\
\omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{retract}\text{''}\right) & \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{retract}\text{''}\right) \\
\omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{revolve}\text{''}\right) & \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}c_{revolve}\text{''}\right) \\
\omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}\Delta u_{blocking}\text{''}\right) & \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}\Delta u_{blocking}\text{''}\right) \\
\omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}\Delta u_{tethering}\text{''}\right) & \Omega_{\psi\psi}\left([spc_A, spc_B]\right)\left(\text{``}\Delta u_{tethering}\text{''}\right)
\end{array}$$

Recall that if $[spc_A, spc_B]$ is in the domain of $\Omega_{\psi\psi}$, then so is $[spc_B, spc_A]$, and both pairs give the same parameters.

$$\Omega_{\psi\psi}\left([spc_A, spc_B]\right) \equiv \Omega_{\psi\psi}\left([spc_B, spc_A]\right)$$

The parameter $\omega_{\psi\psi}$ of the DEVS model TPS differs in this regard. For the sake of convenience, if $[spc_A, spc_B]$ is in the domain of $\omega_{\psi\psi}$, we assume that $[spc_B, spc_A]$ is not.

Another difference between $\omega_{\psi\psi}$ and $\Omega_{\psi\psi}$ is that the latter contains blocking and tethering distances between each species and each subvolume. We use the value $\varnothing$ to identify a subvolume, as opposed to a particle species. The blocking distance associated with $[spc_A, \varnothing]$ or $[\varnothing, spc_A]$ determines when a subvolume gains awareness of a particle of the species with ID spc_A. The tethering distance determines when the subvolume loses awareness of the particle.

$$\begin{array}{l}
\Omega_{\psi\psi}\left([spc_A, \varnothing]\right)\left(\text{``}\Delta u_{blocking}\text{''}\right) \\
\Omega_{\psi\psi}\left([\varnothing, spc_A]\right)\left(\text{``}\Delta u_{blocking}\text{''}\right) \\
\Omega_{\psi\psi}\left([spc_A, \varnothing]\right)\left(\text{``}\Delta u_{tethering}\text{''}\right) \\
\Omega_{\psi\psi}\left([\varnothing, spc_A]\right)\left(\text{``}\Delta u_{tethering}\text{''}\right)
\end{array}$$

Suppose that spc_A and spc_B identify, respectively, the species of particles A and B. Note that the two particles may have the same species ($spc_A \equiv spc_B$ is possible). If the tethering distance between A and B is infinite, then effectively the two particles never become tethered. In that case, the sum of the detection radius of A and that of B must be at least the blocking distance. If the tethering distance is finite, then the sum of the detection radii must be at least the tethering distace.

$$\Omega_{\psi}(spc_A)(\text{“}r\text{”}) + \Omega_{\psi}(spc_B)(\text{“}r\text{”}) \geq \begin{pmatrix} \Delta u_{tethering} = \top & \rightarrow \Delta u_{blocking} \\ \Delta u_{tethering} < \top & \rightarrow \Delta u_{tethering} \end{pmatrix}$$
$$\Delta u_{blocking} = \Omega_{\psi\psi}([spc_A, spc_B])(\text{“}\Delta u_{blocking}\text{”})$$
$$\Delta u_{tethering} = \Omega_{\psi\psi}([spc_A, spc_B])(\text{“}\Delta u_{tethering}\text{”})$$

Looking now at a single particle, its blocking distance with a subvolume is equal to its detection radius plus the radius of the arrival orb. The tethering distance is the sum of the detection radius and the radius of the departure orb.

$$\Omega_{\psi\psi}([spc_A, \varnothing])(\text{“}\Delta u_{blocking}\text{”}) = \Omega_{\psi}(spc_A)(\text{“}r\text{”}) + R_{arrival}$$
$$\Omega_{\psi\psi}([spc_A, \varnothing])(\text{“}\Delta u_{tethering}\text{”}) = \Omega_{\psi}(spc_A)(\text{“}r\text{”}) + R_{departure}$$

Note that we would like to minimize all of the blocking distances between subvolumes and particles. Smaller distances correspond to smaller numbers of particles tracked by each subvolume, and hence faster simulations. The danger is that, if we make the blocking distances too small, a collision may go undetected.

In Figure 37, a collision between particles A and B takes place just barely within the boundary of a subvolume. We have yet to formally define the position where a collision “takes place”, but the distance between this “collision point” and the center of either particle is at most the detection radius of that particle. In Figure 37, the distance between the collision point and either particle is exactly the detection radius.

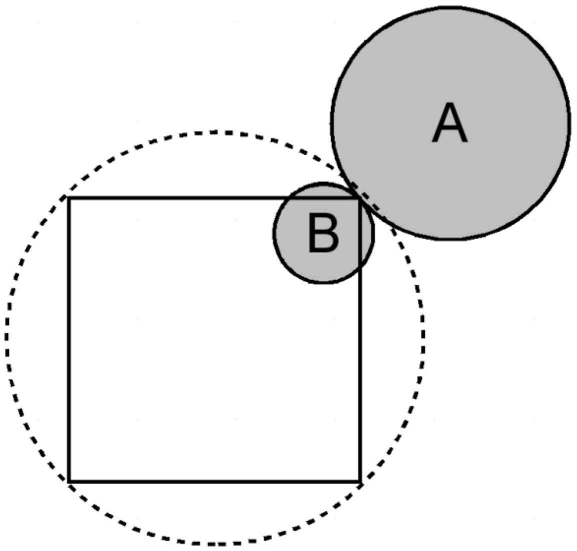


Figure 37: A "worst-case scenario" for collision detection; the collision point is within the square subvolume, but particle A is at a distance.

We are confident, looking at Figure 37, that the subvolume model is aware of particle B. Particle A, however, is almost entirely on the outside. The distance between the center of particle A and the center of the subvolume is at most $\Omega_\psi\left(spc_A\right)\left(\text{“}r\text{”}\right) + R$, where R is the radius of a circle circumscribed around the subvolume. The blocking distance between the subvolume and the particle is $\Omega_\psi\left(spc_A\right)\left(\text{“}r\text{”}\right) + R_{arrival}$, as specified by the equations above. Because $R_{arrival} > R$, we can be sure that the subvolume is aware of particle A.

At this point we have specified constraints on the detection radii that determine the particle-subvolume blocking/tethering distances of Ω_ψ and $\Omega_{\psi\psi}$. Given ω_ψ and $\omega_{\psi\psi}$, the set of detection radii that satisfy these constraints is by no means unique. In the function $properties_{TPS}$, we select distances that are not necessarily optimal, but reasonable. The function takes ω_ψ and $\omega_{\psi\psi}$ as arguments, along with $R_{arrival}$ and $R_{departure}$, and results in Ω_ψ and $\Omega_{\psi\psi}$.
$\ll \ldots \gg$

$$properties_{TPS}\left(\left[\omega_\psi, \omega_{\psi\psi}, R_{arrival}, R_{departure}\right]\right) := \left[\Omega_\psi, \Omega_{\psi\psi}\right]$$

Given particle species IDs spc_A and spc_B, the TPS model parameter $\omega_{\psi\psi}$ has either $[spc_A, spc_B]$ or $[spc_B, spc_A]$ in its domain, but not both. We define $\omega_{\psi\psi}{}'$ so that we can obtain particle-pair information without worrying about the order in which species identifiers are listed.

$\ll \ldots; properties_{TPS} \gg$

$$\omega_{\psi\psi}{}'\left([spc_A, spc_B]\right) := \begin{pmatrix} [spc_A, spc_B] : \Diamond\omega_{\psi\psi} & \rightarrow \omega_{\psi\psi}\left([spc_A, spc_B]\right) \\ [spc_B, spc_A] : \Diamond\omega_{\psi\psi} & \rightarrow \omega_{\psi\psi}\left([spc_B, spc_A]\right) \end{pmatrix}$$

In pursuit of a reasonable set of detection radii, we make the very debatable assumption that the majority of TPS models will have a relatively large number of small particles, and a relatively small number of large particles. We also assume that the size of a particle is roughly proportional to its blocking distance with another particle of the same species. Thus we seek to minimize the detection radii of particles with small "self-blocking" distances. The first step in the process is to sort all species IDs by associated self-blocking distance. Note that, if $i > j$, then the self-blocking distance of the species with ID $SPC_{sorted}(i)$ at least that of the species with ID $SPC_{sorted}(j)$.

$\ll \ldots \gg$

$$\begin{aligned} SPC_{sorted} &:= sort_{cmp}\left([\Diamond\omega_{\psi}, cmp_{spc}]\right) \\ cmp_{spc}\left([spc_A, spc_B]\right) &:= cmp\left([\Delta u_{AA}, \Delta u_{BB}]\right) \\ \Delta u_{AA} &:= \omega_{\psi\psi}{}'\left([spc_A, spc_A]\right)\left(\text{“}\Delta u_{blocking}\text{”}\right) \\ \Delta u_{BB} &:= \omega_{\psi\psi}{}'\left([spc_B, spc_B]\right)\left(\text{“}\Delta u_{blocking}\text{”}\right) \end{aligned}$$

We define R_{ψ} such that the detection radii r_i of the species with ID spc_i is $R_{\psi}(spc_i)$. This radius is at least $\Delta u_{ii}/2$, which is determined from the self-blocking and self-tethering distance associated with spc_i.

$\ll \ldots \gg$

$$\begin{aligned} R_{\psi} &:= loop_i\left([0, [\,]]\right) \\ loop_i\left([i, R_{\psi_i}]\right) &:= \begin{pmatrix} i = \#SPC_{sorted} & \rightarrow R_{\psi_i} \\ i < \#SPC_{sorted} & \rightarrow loop_i\left([i+1, R_{\psi_i}{}']\right) \end{pmatrix} \\ spc_i &:= SPC_{sorted}(i) \\ \Delta u_{ii} &:= \begin{pmatrix} \Delta u_{tethering} = \top & \rightarrow \Delta u_{blocking} \\ \Delta u_{tethering} < \top & \rightarrow \Delta u_{tethering} \end{pmatrix} \\ \Delta u_{blocking} &= \omega_{\psi\psi}{}'\left([spc_i, spc_i]\right)\left(\text{“}\Delta u_{blocking}\text{”}\right) \\ \Delta u_{tethering} &= \omega_{\psi\psi}{}'\left([spc_i, spc_i]\right)\left(\text{“}\Delta u_{tethering}\text{”}\right) \\ R_{\psi_i}{}' &:= R_{\psi_i} \lhd \left[\; spc_i \;\; \rightarrow r_i \;\right] \end{aligned}$$

When it comes time to calculate the detection radius associated with index i, the detection radius associated with j, where $j < i$, has already been calculated. By iterating through each of the known detection radii, the estimate of the detection radius r_i may be increased from its initial value of $\Delta u_{ii}/2$.

$\ll \ldots; R_\psi; loop_i; {R_{\psi_i}}' \gg$

$$
\begin{aligned}
r_i &:= loop_j\left(\left[0, \frac{\Delta u_{ii}}{2}\right]\right)\\
&\quad loop_j\left([j, r_j]\right) := \begin{pmatrix} j = i & \rightarrow r_j \\ j < i & \rightarrow loop_j\left([j+1, {r_j}']\right) \end{pmatrix}\\
&\qquad spc_j := SPC_{sorted}(j)\\
&\qquad \Delta u_{ij} := \begin{pmatrix} \Delta u_{tethering} = \top & \rightarrow \Delta u_{blocking} \\ \Delta u_{tethering} < \top & \rightarrow \Delta u_{tethering} \end{pmatrix}\\
&\qquad\quad \Delta u_{blocking} = {\omega_{\psi\psi}}'\left([spc_i, spc_j]\right)\left(\text{“}\Delta u_{blocking}\text{”}\right)\\
&\qquad\quad \Delta u_{tethering} = {\omega_{\psi\psi}}'\left([spc_i, spc_j]\right)\left(\text{“}\Delta u_{tethering}\text{”}\right)\\
&\qquad {r_j}' := r_j \vee \left(\Delta u_{ij} - R_{\psi i}\left(spc_j\right)\right)
\end{aligned}
$$

With the detection radii determined, the selector Ω_ψ can be defined. By default, for any given particle, random impulses occur at an average time interval of infinity. In other words, there are no random impulses. The three default random impulse parameters associated with spc_A are replaced by those in $\omega_\psi(spc_A)$, if they exist. The associated detection radius, which is not to be found in $\omega_\psi(spc_A)$, is obtained from R_ψ.

$\ll DEVS\_tethered\_particle\_system; properties_{TPS} \gg$

$$
\Omega_\psi(spc_A) := \begin{pmatrix} spc_A : \Diamond\omega_\psi & \rightarrow \begin{bmatrix} \text{“}\tau_{RI}\text{”} & \rightarrow \top \\ \text{“}k_{RI}\text{”} & \rightarrow \varnothing \\ \text{“}\mu_{RI}\text{”} & \rightarrow \varnothing \\ \text{“}r\text{”} & \rightarrow R_\psi(spc_A) \end{bmatrix} \lhd \omega_\psi(spc_A) \end{pmatrix}
$$

The selector $\Omega_{\psi\psi}$ includes ${\omega_{\psi\psi}}'$, which contains the five collision parameters for each particle pair, and $\Omega_{\varnothing\varnothing}$, which contains the particle-subvolume blocking and tethering distances.

$\ll \ldots \gg$

$$
\Omega_{\psi\psi} := {\omega_{\psi\psi}}' \lhd \Omega_{\varnothing\varnothing}
$$

The particle-subvolume blocking and tethering distances depend on the detection radii and $R_{arrival}$ and $R_{departure}$, as previously explained. Note that we define $\Omega_{\varnothing\varnothing}$, and hence $\Omega_{\psi\psi}$, such that we need not worry about whether the particle species ID is listed before or after the value $\varnothing$ that indicates the subvolume.

$\ll \ldots;\Omega_{\psi\psi} \gg$

$$
\begin{aligned}
\Omega_{\varnothing\varnothing}\left([spc_A, spc_B]\right) &:= \begin{pmatrix} [spc_A, spc_B] : \lozenge SPC_{\psi\varnothing} & \rightarrow \Omega_{A\varnothing} \\ [spc_A, spc_B] : \lozenge SPC_{\varnothing\psi} & \rightarrow \Omega_{\varnothing B} \end{pmatrix} \\
SPC_{\psi\varnothing}\left(i\right) &:= \begin{pmatrix} i : ..\#SPC_{sorted} & \rightarrow \left[SPC_{sorted}\left(i\right), \varnothing\right] \end{pmatrix} \\
SPC_{\varnothing\psi}\left(i\right) &:= \begin{pmatrix} i : ..\#SPC_{sorted} & \rightarrow \left[\varnothing, SPC_{sorted}\left(i\right)\right] \end{pmatrix} \\
\Omega_{A\varnothing} &:= \begin{bmatrix} \text{“}\Delta u_{blocking}\text{”} & \rightarrow \Omega_{\psi}\left(spc_A\right)\left(\text{“}r\text{”}\right) + R_{arrival} \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \Omega_{\psi}\left(spc_A\right)\left(\text{“}r\text{”}\right) + R_{departure} \end{bmatrix} \\
\Omega_{\varnothing B} &:= \begin{bmatrix} \text{“}\Delta u_{blocking}\text{”} & \rightarrow \Omega_{\psi}\left(spc_B\right)\left(\text{“}r\text{”}\right) + R_{arrival} \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \Omega_{\psi}\left(spc_B\right)\left(\text{“}r\text{”}\right) + R_{departure} \end{bmatrix}
\end{aligned}
$$

D.2 DEVS TPS Coupled Models

Here we formalize the upper levels of the DEVS TPS model hierarchy, defining the coupled models TPS and $detector$. The $lattice$, which is both a hypercubic lattice model and a coupled model, is defined in Section D.6.

Like all DEVS model functions, TPS_{DEVS} results in an initialization function and a DEVS model.

$\ll DEVS\_tethered\_particle\_system \gg$

$$
\begin{aligned}
&TPS_{DEVS}\left([N, a, \vec{u}_{center}, \omega_{\psi}, \omega_{\psi\psi}, attach, detach, \theta_{revolve}, \Delta t_{max}, \Delta t_{restitute}]\right) := \ldots \\
&\qquad [init_{TPS}, TPS]
\end{aligned}
$$

The definition of TPS exploits two of the functions from Appendix C.

$\ll \ldots; TPS_{DEVS} \gg$

$$\begin{bmatrix} coupled_{DEVS} \\ pr_{order} \end{bmatrix} := DEVS \circ \begin{bmatrix} \text{``}coupled_{DEVS}\text{''} \\ \text{``}pr_{order}\text{''} \end{bmatrix}$$

The number of spatial dimensions is assigned to n_{dim}.

$\ll \ldots \gg$

$$n_{dim} := \#N$$

The model parameters ω_ψ and $\omega_{\psi\psi}$ are converted into Ω_ψ and $\Omega_{\psi\psi}$ as explained in Section D.1.

$\ll \ldots \gg$

$$[\Omega_\psi, \Omega_{\psi\psi}] := properties_{TPS}\left([\omega_\psi, \omega_{\psi\psi}, R_{arrival}, R_{departure}]\right)$$
$$[R_{arrival}, R_{departure}] := radii\left([n_{dim}, a]\right)$$

Initialization functions and DEVS models are obtained for each of the three submodels of the TPS.

$\ll \ldots \gg$

$$\begin{aligned}
&[init_{RI}, RI] := RI_{DEVS}\left([n_{dim}, \Omega_\psi]\right) \\
&[init_{responder}, responder] := \ldots \\
&\qquad responder_{DEVS}\left([\Omega_\psi, \Omega_{\psi\psi}, attach, detach, \theta_{revolve}, \Delta t_{restitute}]\right) \\
&[init_{detector}, detector] := detector_{DEVS}\left([N, a, \vec{u}_{center}, \Omega_\psi, \Omega_{\psi\psi}, \Delta t_{max}]\right)
\end{aligned}$$

From the DEVS models obtained above, we can define the coupled model parameter M.

$\ll \ldots \gg$

$$M := \begin{bmatrix} \text{``}RI\text{''} & \to RI \\ \text{``}responder\text{''} & \to responder \\ \text{``}detector\text{''} & \to detector \end{bmatrix}$$

The coupled model parameter C formally describes the links between the submodels of the TPS, which are shown in Figure 14 of Section 4.3.

$\ll \ldots \gg$

$$C\left([id_{src}, port_{src}]\right) := ID\_PORT_{dst}$$

Each message in a TPS model has four possible sources: the input of the overall model, the output of the random impulse model RI, the output of the $responder$, and the output of the $detector$. We partition the definition of C accordingly.

$\ll \ldots ; C \gg$

$$ID\_PORT_{dst} := \begin{pmatrix} id_{src} & \equiv & \varnothing & \to C_{\varnothing} \\ id_{src} & \equiv & \text{``}RI\text{''} & \to C_{RI} \\ id_{src} & \equiv & \text{``}responder\text{''} & \to C_{responder} \\ id_{src} & \equiv & \text{``}detector\text{''} & \to C_{detector} \end{pmatrix}$$

There are two types of messages accepted by a TPS model: $transition$ messages and $impulse$ messages. Both of these types are directed to the $responder$ submodel.

$\ll \ldots ; ID\_PORT_{dst} \gg$

$$C_{\varnothing} := \begin{pmatrix} port_{src} \equiv \text{``}transition\text{''} & \to [[\text{``}responder\text{''}, \text{``}transition\text{''}]] \\ port_{src} \equiv \text{``}impulse\text{''} & \to [[\text{``}responder\text{''}, \text{``}impulse\text{''}]] \end{pmatrix}$$

The RI submodel outputs $impulse$ messages, also bound for the $responder$.

$\ll \ldots \gg$

$$C_{RI} := \begin{pmatrix} port_{src} \equiv \text{``}impulse\text{''} & \to [[\text{``}responder\text{''}, \text{``}impulse\text{''}]] \end{pmatrix}$$

The responder outputs messages that indicate the attachment and detachment of particles, the occurance of impulses and loading events and restitution events, and responses to impulses or loading or restitution. All of these are directed to the output of the overall TPS model. The $response$ messages are also directed to the input of the $detector$.

$\ll \ldots \gg$

$$C_{responder} := \begin{pmatrix} port_{src} \equiv \text{``}attachment\text{''} & \to [[\varnothing, \text{``}attachment\text{''}]] \\ port_{src} \equiv \text{``}detachment\text{''} & \to [[\varnothing, \text{``}detachment\text{''}]] \\ port_{src} \equiv \text{``}impulse\text{''} & \to [[\varnothing, \text{``}impulse\text{''}]] \\ port_{src} \equiv \text{``}loading\text{''} & \to [[\varnothing, \text{``}loading\text{''}]] \\ port_{src} \equiv \text{``}restitution\text{''} & \to [[\varnothing, \text{``}restitution\text{''}]] \\ port_{src} \equiv \text{``}response\text{''} & \to \begin{bmatrix} [\text{``}detector\text{''}, \text{``}response\text{''}] \\ [\varnothing, \text{``}response\text{''}] \end{bmatrix} \end{pmatrix}$$

The $detector$ has two types of output messages. An $escape$ message is directed to the output of the TPS to indicate that a particle has escaped the region in which collisions are detected. A $collision$ message goes to the $responder$.

$\ll \ldots \gg$

$$C_{detector} := \begin{pmatrix} port_{src} \equiv \text{“}escape\text{”} & \rightarrow [[\varnothing, \text{“}escape\text{”}]] \\ port_{src} \equiv \text{“}collision\text{”} & \rightarrow [[\text{“}responder\text{”}, \text{“}collision\text{”}]] \end{pmatrix}$$

The priority function indicates that the $responder$ has priority over the $detector$, which is important to ensure that one collision is resolved before another is detected. The $detector$ has priority over the RI submodel.

$\ll DEVS\_tethered\_particle\_system; TPS_{DEVS} \gg$

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

With M, C, and pr defined, $coupled_{DEVS}$ is invoked to yield a useful initialization function and the final DEVS model.

$\ll \ldots \gg$

$$[init_{coupled}, TPS] := coupled_{DEVS}\left([M, C, pr]\right)$$

The final initialization function $init_{TPS}$ makes use of the $init_{coupled}$ function obtained above. Initialization parameters include Ψ, which contains the initial information about each particle, and Φ, which affects the attachment and detachment of colliding particles.

$\ll \ldots \gg$

$$\begin{aligned} init_{TPS}\left([\Psi, \Phi]\right) &:= s \\ s &:= init_{coupled}\left(S\right) \\ S &:= \begin{bmatrix} \text{“}RI\text{”} & \rightarrow init_{RI}\left(\Psi\right) \\ \text{“}responder\text{”} & \rightarrow init_{responder}\left([\Psi, \Phi]\right) \\ \text{“}detector\text{”} & \rightarrow init_{detector}\left(\Psi\right) \end{bmatrix} \end{aligned}$$

Reflecting Section 4.3’s Figure 15, the definition of the $detector$ submodel is similar to that of the TPS. It makes use of two DEVS functions from Appendix C, as well as the initialization functions and DEVS models of the submodels $tracker$ and $lattice$.

$\ll DEVS\_tethered\_particle\_system \gg$

$$\begin{aligned}
&detector_{DEVS}\left([N, a, \vec{u}_{center}, \Omega_{\psi}, \Omega_{\psi\psi}, \Delta t_{max}]\right) := [init_{TPS}, TPS] \\
&\quad \begin{bmatrix} coupled_{DEVS} \\ pr_{order} \end{bmatrix} := DEVS \circ \begin{bmatrix} \text{``}coupled_{DEVS}\text{''} \\ \text{``}pr_{order}\text{''} \end{bmatrix} \\
&\quad [init_{tracker}, tracker] := tracker_{DEVS}\left([N, a, \vec{u}_{center}, \Omega_{\psi}]\right) \\
&\quad [init_{lattice}, lattice] := lattice_{DEVS}\left([N, a, \vec{u}_{center}, \Omega_{\psi}, \Omega_{\psi\psi}, \Delta t_{max}]\right)
\end{aligned}$$

The $tracker$ and $lattice$ DEVS models are used to define M.

$\ll \ldots; detector_{DEVS} \gg$

$$M := \begin{bmatrix} \text{``}tracker\text{''} & \rightarrow tracker \\ \text{``}lattice\text{''} & \rightarrow lattice \end{bmatrix}$$

Messages described by C originate at the input of the $detector$, the output of the $tracker$, and the output of the $lattice$.

$\ll \ldots \gg$

$$\begin{aligned}
&C\left([id_{src}, port_{src}]\right) := ID\_PORT_{dst} \\
&\quad ID\_PORT_{dst} := \begin{pmatrix} id_{src} & \equiv & \varnothing & \rightarrow C_{\varnothing} \\ id_{src} & \equiv & \text{``}tracker\text{''} & \rightarrow C_{tracker} \\ id_{src} & \equiv & \text{``}lattice\text{''} & \rightarrow C_{lattice} \end{pmatrix}
\end{aligned}$$

Only $response$ messages are received at the $detector$'s input. They are directed to the $tracker$.

$\ll \ldots; C; ID\_PORT_{dst} \gg$

$$C_{\varnothing} := \left(\; port_{src} \equiv \text{``}response\text{''} \quad \rightarrow [[\text{``}tracker\text{''}, \text{``}response\text{''}]] \; \right)$$

The tracker outputs $response$ messages like the ones it receives, except that the symbol "$response$" occurs only as part of the port. The port also includes the coordinates $coords_{subV}$ of a single subvolume, the identity of the "one" subvolume model in the $lattice$ that is to receive the message.

$\ll \ldots \gg$

$$
\begin{aligned}
&C_{tracker} := \Big(\; general_{src} \equiv \text{“}response\text{”} \quad \rightarrow [[\text{“}lattice\text{”}, port_{dst}]] \; \Big) \\
&\qquad [general_{src}, specific_{src}] := port_{src} \\
&\qquad coords_{subV} := specific_{src} \\
&\qquad port_{dst} := [\text{“}one\text{”}, specific_{dst}] \\
&\qquad\qquad specific_{dst} := [coords_{subV}, \text{“}response\text{”}]
\end{aligned}
$$

When a particle in one subvolume approaches an adjacent subvolume, an adj message is sent from the first subvolume model to the second. If an adj message is sent beyond a boundary of the overall lattice, the $lattice$ model itself outputs the message. The $detector$ then outputs this message as an $escape$ message. There are three other types of $lattice$ output messages: $collision$ messages, which the $detector$ outputs; $arrival$ messages, which are directed to the $tracker$; and $departure$ messages, which are also sent to the $tracker$.

$\ll \ldots \gg$

$$
C_{lattice} := \begin{pmatrix} general_{src} \equiv \text{“}adj\text{”} & \rightarrow [[\varnothing, \text{“}escape\text{”}]] \\ general_{src} \equiv \text{“}out\text{”} & \rightarrow C_{out} \end{pmatrix}
$$

$$
C_{out} := \begin{pmatrix} specific_{src} \equiv \text{“}collision\text{”} & \rightarrow [[\varnothing, \text{“}collision\text{”}]] \\ specific_{src} \equiv \text{“}arrival\text{”} & \rightarrow [[\text{“}tracker\text{”}, \text{“}arrival\text{”}]] \\ specific_{src} \equiv \text{“}departure\text{”} & \rightarrow [[\text{“}tracker\text{”}, \text{“}departure\text{”}]] \end{pmatrix}
$$

Every $response$ message from the $tracker$ must be delivered to the $lattice$ before the $lattice$ outputs a $collision$ message. This is important because a new particle trajectory in a $response$ message may influence future collisions.

$\ll DEVS\_tethered\_particle\_system; detector_{DEVS} \gg$

$$
\begin{aligned}
pr\left([id_A, id_B]\right) &:= pr_{order}\left([id_A, id_B, order]\right) \\
order &:= [\text{“}tracker\text{”}, \text{“}lattice\text{”}]
\end{aligned}
$$

The $coupled_{DEVS}$ produces the DEVS model $detector$ and a function named $init_{coupled}$, and $init_{coupled}$ is used in the initialization function $init_{detector}$.

$\ll \ldots \gg$

$$\begin{aligned}
&[init_{coupled}, detector] := coupled_{DEVS}\left([M, C, pr]\right)\\
&init_{detector}\left(\Psi\right) := s\\
&\quad s := init_{coupled}\left(S\right)\\
&\qquad S := \begin{bmatrix} \text{“}tracker\text{”} & \rightarrow init_{tracker}\left(\Psi\right) \\ \text{“}lattice\text{”} & \rightarrow init_{lattice}\left(\varnothing\right) \end{bmatrix}
\end{aligned}$$

D.3 DEVS Random Impulse Model

The random impulse model generates impulses of randomized timing, magnitude, and direction to be applied to individual particles in a tethered particle system. Its parameters include the number of spatial dimensions n_{dim}, and Ω_ψ.

$\ll DEVS\_tethered\_particle\_system \gg$

$$RI_{DEVS}\left([n_{dim}, \Omega_\psi]\right) := [init_{RI}, RI]$$

We make use of the random impulse functions of Appendix B, as well as several future events list and DEVS-related functions of Appendix C.

$\ll \ldots; RI_{DEVS} \gg$

$$\begin{aligned}
&\begin{bmatrix} detect_{RI} \\ impulse_{RI} \end{bmatrix} := tethered\_particle\_system \circ \begin{bmatrix} \text{“}detect_{RI}\text{”} \\ \text{“}impulse_{RI}\text{”} \end{bmatrix}\\
&\begin{bmatrix} FEL_{empty} \\ \delta_{FEL} \\ event_{FEL} \end{bmatrix} := future\_events\_list \circ \begin{bmatrix} \text{“}FEL_{empty}\text{”} \\ \text{“}\delta_{FEL}\text{”} \\ \text{“}event_{FEL}\text{”} \end{bmatrix}\\
&pr_{\varnothing} := DEVS\left(\text{“}pr_{\varnothing}\text{”}\right)
\end{aligned}$$

The initialization function requires Ψ, a selector containing information about the initial state of the particles.

$\ll \ldots \gg$

$$init_{RI}\left(\Psi\right) := s$$

There are three state variables.

$\ll \ldots; init_{RI} \gg$

$$s := [t, SPC, FEL]$$

The state variable t, the current time, is initially zero. The selector SPC maps a particle ID to its associated species ID.

$\ll \ldots; s \gg$

$$\begin{aligned}
&t := 0\\
&SPC\,(id_A) := \Big(\; id_A : \Diamond\Psi \quad \rightarrow \Psi\,(id_A)\,(\text{“}spc\text{”}) \;\Big)
\end{aligned}$$

The state variable FEL is a future events list that records, for each particle, the time when its next random impulse is scheduled to occur. To obtain the initial FEL we invoke, for each particle, the $detect_{RI}$ function. Note that the use of $pr_{\varnothing}$ indicates that, if two random impulses are to occur at the same simulated time, the order in which they occur is randomized.

$\ll \ldots \gg$

$$\begin{aligned}
&FEL := loop\,(0, FEL_{empty})\\
&\quad ID_{\psi} := \Diamond\Psi\\
&\quad loop\,(i, FEL_i) := \begin{pmatrix} i = \#ID_{\psi} & \rightarrow FEL_i \\ i < \#ID_{\psi} & \rightarrow loop\,(i+1, FEL_i{}') \end{pmatrix}\\
&\qquad id_A := ID_{\psi}\,(i)\\
&\qquad spc_A := SPC\,(id_A)\\
&\qquad t_A := detect_{RI}\,(spc_A, \Omega_{\psi})\\
&\qquad FEL_i{}' := \delta_{FEL}\,([FEL_i, id_A, t_A, pr_{\varnothing}])
\end{aligned}$$

With the initialization function complete, we turn our attention to the DEVS model itself.

$\ll DEVS\_tethered\_particle\_system; RI_{DEVS} \gg$

$$RI := [\delta_{ext}, \delta_{int}, ta]$$

The random impulse model should receive no inputs; hence the external transition function should never be invoked.

$\ll \ldots; RI \gg$

$$\delta_{ext}\left([s, \Delta t_{el}, x]\right) := (\ )$$

If the internal transition function is invoked, we know that the simulation has reached the simulated time when the next impulse in the FEL must be applied. We obtain the identity id_A of the particle that is to receive the impulse, the time t' of the impulse, and the particle's species ID spc_A.

$\ll \ldots \gg$

$$\begin{aligned}
&\delta_{int}(s) := [s', Y] \\
&\qquad [t, SPC, FEL] := s \\
&\qquad [id_A, t'] := event_{FEL}(FEL) \\
&\qquad spc_A := SPC(id_A)
\end{aligned}$$

The state variable t is replaced with t', and the FEL is updated with the time of the next impulse to be applied to the particle.

$\ll \ldots; \delta_{int} \gg$

$$\begin{aligned}
&s' := [t', SPC, FEL'] \\
&\qquad t_A := t' + detect_{RI}\left([spc_A, \Omega_\psi]\right) \\
&\qquad FEL' := \delta_{FEL}\left([FEL, id_A, t_A, pr_\varnothing]\right)
\end{aligned}$$

One message is output, an $impulse$ message identifying the particle and indicating the randomized momentum change Δp it is to receive.

$\ll \ldots \gg$

$$\begin{aligned}
&Y := [[\text{“}impulse\text{”}, y]] \\
&\qquad \Delta p := impulse_{RI}\left([n_{dim}, spc_A, \Omega_\psi]\right) \\
&\qquad y := [id_A, \Delta p]
\end{aligned}$$

The time advance function results in the time before the next impulse, as indicated by the FEL. Because computer round-off errors can lead to a discrepancy between times recorded in state variables, we make a habit of ensuring that the result of ta is never negative.

$\ll DEVS\_tethered\_particle\_system; RI_{DEVS}; RI \gg$

$$\begin{aligned}
&ta(s) := \Delta t_{int} \\
&\qquad [t, SPC, FEL] := s \\
&\qquad [id_A, t_A] := event_{FEL}(FEL) \\
&\qquad \Delta t_{int} := (t_A - t) \vee 0
\end{aligned}$$

D.4 DEVS Responder Model

Here we define the $responder$ model.

$\ll DEVS\_tethered\_particle\_system \gg$

$$responder_{DEVS}\left([\Omega_{\psi}, \Omega_{\psi\psi}, attach, detach, \theta_{revolve}, \Delta t_{restitute}]\right) := \ldots \\ [init_{responder}, responder]$$

First we obtain several functions from Appendix B and Appendix C.

$\ll \ldots; responder_{DEVS} \gg$

$$\begin{aligned} &\begin{bmatrix} impact \\ load \\ restitute \end{bmatrix} := tethered\_particle\_system \circ \begin{bmatrix} \text{“}impact\text{”} \\ \text{“}load\text{”} \\ \text{“}restitute\text{”} \end{bmatrix} \\ &\begin{bmatrix} FEL_{empty} \\ \delta_{FEL} \\ event_{FEL} \end{bmatrix} := future\_events\_list \circ \begin{bmatrix} \text{“}FEL_{empty}\text{”} \\ \text{“}\delta_{FEL}\text{”} \\ \text{“}event_{FEL}\text{”} \end{bmatrix} \\ &pr_Z := DEVS\left(\text{“}pr_Z\text{”}\right) \end{aligned}$$

For reasons explained later, $responder$ requires a priority function dependent on its state variable $N_{loading}$.

$\ll \ldots \gg$

$$\begin{aligned} get_{pr_{restitute}}\left(N_{loading}\right) &:= pr_{restitute} \\ pr_{restitute}\left([id_{event_A}, id_{event_B}]\right) &:= pr_Z\left([id_{event_A}, id_{event_B}, Z]\right) \\ Z\left(id_{event}\right) &:= -N_{loading}\left(id_{event}\right) \end{aligned}$$

The initialization function has two arguments: Ψ, containing particle information, and Φ, which affects the attachment and detachment of particles.

$\ll \ldots \gg$

$$init_{responder}\left([\Psi, \Phi]\right) := s$$

The state variables are described below by the comments in the vector assigned to s.

$\ll \ldots; init_{responder} \gg$

$$s := \begin{bmatrix} t & \{\text{current time}\} \\ \Psi & \{\text{particle system state}\} \\ \Phi & \{\text{previous transition state}\} \\ FEL & \{\text{future events list}\} \\ N_{loading} & \{\text{order of past loading events}\} \\ n_{loading} & \{\text{number of past loading events}\} \\ ID_{attachment} & \{\text{IDs of current particle attachments}\} \\ ID_{detachment} & \{\text{IDs of current particle detachments}\} \\ ID_{impulse} & \{\text{IDs of current particle impulses}\} \\ ID_{loading} & \{\text{IDs of current loading events}\} \\ ID_{response} & \{\text{IDs of current particle responses}\} \end{bmatrix}$$

State variables include the time, which is initially zero, and an FEL of future restitution events, which is initially empty. The initially-empty selector $N_{loading}$ and initially-zero integer $n_{loading}$ reflect the fact that there have been no past loading events. Most of the ID vectors are initially empty, meaning that there are no messages to output. The exception is the $ID_{response}$ vector, which initially lists the IDs of all particles. As a consequence, the responder will output the position and trajectory of each particle at the beginning of a simulation.

$\ll \ldots; s \gg$

$$\begin{aligned} & t = 0 \\ & FEL := FEL_{empty} \\ & N_{loading} := [\,] \\ & n_{loading} := 0 \\ & ID_{attachment} := [\,] \\ & ID_{detachment} := [\,] \\ & ID_{impulse} := [\,] \\ & ID_{loading} := [\,] \\ & ID_{response} := \Diamond\Psi \end{aligned}$$

In our implementation, we listed all pairs of tethered particles in the initial $ID_{attachment}$, as we wanted our output file to provide this information.

The initialization function complete, we now define the model itself.

$\ll DEVS\_tethered\_particle\_system; responder_{DEVS} \gg$

$$responder := [\delta_{ext}, \delta_{int}, ta]$$

We start with the external transition function.

$\ll \ldots; responder \gg$

$$\delta_{ext}\left([s, \Delta t_{el}, x]\right) := s'$$

After obtaining, the state variables, the new time t' is calculated from the old time t and the elapsed time Δt_{el}. The input is separated into $port$ and msg components.

$\ll \ldots; \delta_{ext} \gg$

$$\begin{aligned} &\left[\begin{array}{c} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response} \end{array}\right] := s \\ &t' := t + \Delta t_{el} \\ &[port, msg] := x \end{aligned}$$

There are three types of inputs: $transition$ messages, $impulse$ messages, and $collision$ messages.

$\ll \ldots \gg$

$$s' := \left(\begin{array}{lll} port \equiv \text{“}transition\text{”} & \rightarrow s_{transition} \\ port \equiv \text{“}impulse\text{”} & \rightarrow s_{impulse} \\ port \equiv \text{“}collision\text{”} & \rightarrow s_{collision} \end{array}\right)$$

A $transition$ message simply replaces the Φ value, which alters the conditions under which particles become tethered to one another or separate from one another.

$\ll \ldots; s' \gg$

$$\begin{aligned} s_{transition} &:= \left[\begin{array}{c} t', \Psi, \Phi', FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response} \end{array}\right] \\ \Phi' &:= msg \end{aligned}$$

An $impulse$ message alters the trajectory of a particle, and all other particles loaded to that particle. This change requires a modification of Ψ, which results from the $impact$ function of Appendix B. Before time advances, the $responder$ is to output both an $impulse$ message, and a separate $response$ message for each affected particle. This requires IDs to be added to $ID_{impulse}$ and $ID_{response}$.

$\ll \ldots \gg$

$$s_{impulse} := \left[\begin{array}{c} t', \Psi', \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}{}', ID_{loading}, ID_{response}{}' \end{array} \right]$$

$$\begin{aligned} &[id_A, \Delta p] := msg \\ &[\Psi', ID_{response_A}] := impact\left([\Psi, t', \Omega_\psi, \Delta p]\right) \\ &ID_{impulse}{}' := ID_{impulse} \parallel [id_A] \\ &ID_{response}{}' := ID_{response} \parallel ID_{response_A} \end{aligned}$$

The remainder of the external transition function deals with the handling of $collision$ messages. The message itself indicates the type of collision, either “blocking” or “tethering”, and the IDs of the two particles involved. We must first determine whether the particles are to end up travelling away from one another ($parting = \top$), or loading ($particle = \bot$).

$\ll \ldots \gg$

$$s_{collision} := \left[\begin{array}{ll} parting & \rightarrow s_{parting} \\ \neg parting & \rightarrow s_{loading} \end{array} \right]$$

$$\begin{aligned} &[ID_{AB}, type_{collision}] := msg \\ &[id_A, id_B] := sort\left(ID_{AB}\right) \end{aligned}$$

The evaluation of $parting$ involves a lengthy set of nested cases, as we must consider the type of collision, whether the particles are already tethered, and whether they attach or detach. Along with $parting$, we calculate an intermediate version of Ψ, and final versions of $ID_{attachment}$ and $ID_{detachment}$.

$\ll \ldots; s_{collision} \gg$

$$\begin{bmatrix} parting \\ \Psi_{\Delta} \\ ID_{attachment}' \\ ID_{detachment}' \end{bmatrix} := \left(\begin{array}{lcll} type_{\text{collision}} & \equiv & \text{“}blocking\text{”} & \rightarrow \begin{bmatrix} \bot \\ \Psi_{blocking} \\ ID_{attachment_{blocking}} \\ ID_{detachment} \end{bmatrix} \\ type_{\text{collision}} & \equiv & \text{“}tethering\text{”} & \rightarrow \begin{bmatrix} parting_{tethering} \\ \Psi_{tethering} \\ ID_{attachment} \\ ID_{detachment_{tethering}} \end{bmatrix} \end{array} \right)$$

In the case of a blocking collision, the particles may not part and may not detach, though Ψ and $ID_{attachment}$ may change.

$\ll \ldots; [parting, \Psi_{\Delta}, ID_{attachment}', ID_{detachment}'] \gg$

$$\begin{bmatrix} \Psi_{blocking} \\ ID_{attachment_{blocking}} \end{bmatrix} := \left(\begin{array}{ll} tethered & \rightarrow \begin{bmatrix} \Psi \\ ID_{attachment} \end{bmatrix} \\ -tethered & \rightarrow \begin{bmatrix} \Psi_{separate} \\ ID_{attachment_{separate}} \end{bmatrix} \end{array} \right)$$

$$tethered := \left(id_B : \Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right)\right)$$

In the case of a blocking collision in which the particles are not already tethered, the $attach$ function is evaluated to determine whether the particles become tethered. If they do, then Ψ is updated and the need for an $attachment$ message is recorded.

$\ll \ldots; \left[\Psi_{blocking}, ID_{attachment_{blocking}}\right] \gg$

$$\begin{bmatrix} \Psi_{separate} \\ ID_{attachment_{separate}} \end{bmatrix} := \left(\begin{array}{ll} -\alpha & \rightarrow \begin{bmatrix} \Psi \\ ID_{attachment} \end{bmatrix} \\ \alpha & \rightarrow \begin{bmatrix} \Psi_{attach} \\ ID_{attachment} \parallel [[id_A, id_B]] \end{bmatrix} \end{array} \right)$$

$$\alpha := attach\left([id_A, id_B, \Psi, \Phi, \Omega_{\psi}, \Omega_{\psi\psi}]\right)$$

If the two particles become tethered, the list of tethered particles in $\Psi\left(id_A\right)$ must include id_B, and the list in $\Psi\left(id_B\right)$ must include id_A.

$\ll \ldots; \left[\Psi_{separate}, ID_{attachment_{separate}}\right] \gg$

$$\Psi_{attach} := \Psi \lhd \begin{bmatrix} id_A & \to \Psi\left(id_A\right) \lhd \begin{bmatrix} \text{“}tethered\text{”} & \to TL_A \end{bmatrix} \\ id_B & \to \Psi\left(id_B\right) \lhd \begin{bmatrix} \text{“}tethered\text{”} & \to TL_B \end{bmatrix} \end{bmatrix}$$
$$TL_A := \Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right) \parallel \left[id_B\right]$$
$$TL_B := \Psi\left(id_B\right)\left(\text{“}tethered\text{”}\right) \parallel \left[id_A\right]$$

In the case of a tethering collision, the attachment of the particles is not a possibility, though we must still consider changing $parting$, Ψ, and $ID_{detachment}$.

$\ll DEVS\_tethered\_particle\_system; responder_{DEVS}; responder;$
$\delta_{ext}; s'; s_{collision}; [parting, \Psi_{\Delta}, ID_{attachment}', ID_{detachment}'] \gg$

$$\begin{bmatrix} parting_{tethering} \\ \Psi_{tethering} \\ ID_{detachment_{tethering}} \end{bmatrix} := \left(\begin{array}{ll} \neg tethered & \to \begin{bmatrix} \top \\ \Psi \\ ID_{detachment} \end{bmatrix} \\ tethered & \to \begin{bmatrix} parting_{tethered} \\ \Psi_{tethered} \\ ID_{detachment_{tethered}} \end{bmatrix} \end{array} \right)$$
$$tethered := \left(id_B : \Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right)\right)$$

If two particles undergo a tethering collision, and if those particles are already tethered, then $detach$ is used to determine whether the particles cease to be tethered. If they do, then Ψ is updated and the $ID_{detachment}$ includes the particle IDs.

$\ll \ldots; \left[parting_{tethering}, \Psi_{tethering}, ID_{detachment_{tethering}}\right] \gg$

$$\begin{bmatrix} parting_{tethered} \\ \Psi_{tethered} \\ ID_{detachment_{tethered}} \end{bmatrix} := \left(\begin{array}{ll} \neg\alpha & \to \begin{bmatrix} \bot \\ \Psi \\ ID_{detachment} \end{bmatrix} \\ \alpha & \to \begin{bmatrix} \top \\ \Psi_{detach} \\ ID_{detachment} \parallel \left[\left[id_A, id_B\right]\right] \end{bmatrix} \end{array} \right)$$
$$\alpha := detach\left(\left[id_A, id_B, \Psi, \Phi, \Omega_{\psi}, \Omega_{\psi\psi}\right]\right)$$

If the two particles detach, the list of tethered particles in $\Psi\left(id_A\right)$ must exclude id_B, and the list in $\Psi\left(id_B\right)$ must exclude id_A.

$\ll \ldots; \left[parting_{tethered}, \Psi_{tethered}, ID_{detachment_{tethered}}\right] \gg$

$$\begin{aligned}\Psi_{detach} &:= \Psi \lhd \left[\begin{array}{ll} id_A & \rightarrow \Psi\left(id_A\right) \lhd \left[\begin{array}{ll} \text{“}tethered\text{”} & \rightarrow TL_A \end{array}\right] \\ id_B & \rightarrow \Psi\left(id_B\right) \lhd \left[\begin{array}{ll} \text{“}tethered\text{”} & \rightarrow TL_B \end{array}\right] \end{array}\right] \\ TL_A &:= del\left(\left[\Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right), id_B\right]\right) \\ TL_B &:= del\left(\left[\Psi\left(id_B\right)\left(\text{“}tethered\text{”}\right), id_A\right]\right)\end{aligned}$$

The boolean $parting$ is now fully specified. If the particles do part, no further changes to the state variables are needed.

$\ll DEVS\_tethered\_particle\_system; responder_{DEVS}; responder; \delta_{ext}; s'; s_{collision} \gg$

$$s_{parting} := \left[\begin{array}{c} t', \Psi_\Delta, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}', \ldots \\ ID_{detachment}', ID_{impulse}, ID_{loading}, ID_{response} \end{array}\right]$$

The remainder of the definition of δ_{ext} focuses on $collision$ messages that do result in particle loading.

$\ll \ldots \gg$

$$s_{loading} := \left[\begin{array}{c} t', \Psi', \Phi, FEL', N_{loading}', n_{loading}', ID_{attachment}', \ldots \\ ID_{detachment}', ID_{impulse}, ID_{loading}', ID_{response}' \end{array}\right]$$

The final value of Ψ is determined from the $load$ function of Appendix B, along with IDs of all loaded particles in the group and all pairs of directly loaded particles in the group.

$\ll \ldots; s_{loading} \gg$

$$\left[\Psi', \Delta ID_{response}, ID_{\psi\psi}\right] := load\left(\left[\Psi_\Delta, t', id_A, id_B, \Omega_\psi, \Omega_{\psi\psi}, \theta_{revolve}, type_{collision}\right]\right)$$

The pair of colliding particles is added to the domain of the state variable $N_{loading}$, the associated value being the number $n_{loading}$ of past loading events. This number is itself incremented, so that each loading event is assigned a greater number than those that preceded it.

$\ll \ldots ; s_{loading} \gg$

$$\begin{aligned} {N_{loading}}' &:= N_{loading} \lhd \left[\; [id_A, id_B] \quad \to n_{loading} \; \right] \\ {n_{loading}}' &:= n_{loading} + 1 \end{aligned}$$

The state variable $ID_{loading}$ is updated to produce a single $loading$ message, and $ID_{response}$ is expanded to produce a separate $response$ message for each particle in the loaded group.

$\ll \ldots \gg$

$$\begin{aligned} {ID_{loading}}' &:= ID_{loading} \parallel [[id_A, id_B]] \\ {ID_{response}}' &:= ID_{response} \parallel \Delta ID_{response} \end{aligned}$$

For each pair of directly loaded particles in the group, the FEL is updated such that the time at which restitution occurs exceeds the current time t' by $\Delta t_{restitute}$. The result is that many pairs of particles are scheduled for restitution at the same future simulated time. This is where $N_{loading}$ comes into play. It is passed to $get_{pr_{restitute}}$, defined earlier, which yields a priority function $pr_{restitute}$. This priority function ensures that the most recently-loaded particle pairs are the first to undergo restitution.

$\ll \ldots \gg$

$$\begin{aligned} FEL' &:= loop\left([0, FEL]\right) \\ loop\left([i, FEL_i]\right) &:= \begin{pmatrix} i = \#ID_{\psi\psi} & \to FEL_i \\ i < \#ID_{\psi\psi} & \to loop\left([i+1, {FEL_i}']\right) \end{pmatrix} \\ [id_C, id_D] &:= sort\left(ID_{\psi\psi}\left(i\right)\right) \\ pr_{restitute} &:= get_{pr_{restitute}}\left({N_{loading}}'\right) \\ {FEL_i}' &:= \delta_{FEL}\left(\left[FEL_i, [id_C, id_D], t' + \Delta t_{restitute}, pr_{restitute}\right]\right) \end{aligned}$$

Having completed the definition of δ_{ext}, we turn our attention to the internal transition function.

$\ll DEVS\_tethered\_particle\_system; responder_{DEVS}; responder \gg$

$$\delta_{int}\left(s\right) := [s', Y]$$

First we separate the state variables.

$\ll \delta_{int} \gg$

$$\begin{bmatrix} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response} \end{bmatrix} := s$$

To obtain the new state and vector of outputs, we consider five different cases.

$\ll \ldots \gg$

$$[s', Y] := \begin{pmatrix} \alpha_{attachment} & \rightarrow [s_{attachment}, Y_{attachment}] \\ \alpha_{detachment} & \rightarrow [s_{detachment}, Y_{detachment}] \\ \alpha_{impulse} & \rightarrow [s_{impulse}, Y_{impulse}] \\ \alpha_{loading} & \rightarrow [s_{loading}, Y_{loading}] \\ \alpha_{other} & \rightarrow [s_{other}, Y_{other}] \end{pmatrix}$$

The conditions depend on which ID vectors contain elements, indicating output messages, and which ID vectors are empty.

$\ll \ldots ; [s, Y'] \gg$

$$\begin{aligned} &\alpha_{attachment} := (\#ID_{attachment} > 0) \\ &\alpha_{detachment} := (\#ID_{attachment} = 0) \wedge (\#ID_{detachment} > 0) \\ &\alpha_{impulse} := (\#ID_{attachment} = \#ID_{detachment} = 0) \wedge (\#ID_{impulse} > 0) \\ &\alpha_{loading} := (\#ID_{attachment} = \#ID_{detachment} = \#ID_{impulse} = 0) \wedge (\#ID_{loading} > 0) \\ &\alpha_{other} := (\#ID_{attachment} = \#ID_{detachment} = \#ID_{impulse} = \#ID_{loading} = 0) \end{aligned}$$

If one or more attachments have occurred, a separate *attachment* message is output for each element in the vector $ID_{attachment}$. The vector is then emptied.

$\ll \ldots \gg$

$$\begin{aligned} s_{attachment} &:= \begin{bmatrix} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, [\,], \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response} \end{bmatrix} \\ Y_{attachment}(i) &:= \left(\; i : \Diamond ID_{attachment} \quad \rightarrow [\text{“}attachment\text{”}, ID_{attachment}(i)] \; \right) \end{aligned}$$

The same is done for detachments, impulses, and loading events.

$\ll \ldots \gg$

$$s_{detachment} := \left[\begin{array}{c} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ [\,], ID_{impulse}, ID_{loading}, ID_{response} \end{array} \right]$$
$$Y_{detachment}(i) := \left(\; i : \Diamond ID_{detachment} \quad \rightarrow [\text{“}detachment\text{”}, ID_{detachment}(i)] \; \right)$$
$$s_{impulse} := \left[\begin{array}{c} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, [\,], ID_{loading}, ID_{response} \end{array} \right]$$
$$Y_{impulse}(i) := \left(\; i : \Diamond ID_{impulse} \quad \rightarrow [\text{“}impulse\text{”}, ID_{impulse}(i)] \; \right)$$
$$s_{loading} := \left[\begin{array}{c} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, [\,], ID_{response} \end{array} \right]$$
$$Y_{loading}(i) := \left(\; i : \Diamond ID_{loading} \quad \rightarrow [\text{“}loading\text{”}, ID_{loading}(i)] \; \right)$$

The remaining types of internal transitions involve either the output of $response$ messages, or the processing of restitution events. The timing of a restitution event is determined by the next event in the FEL. If the next restitution occurs in the future, then any queued $response$ messages are output. If there are no queued $response$ messages, or if the next restitution event occurs at the present time, then the restitution is processed.

$\ll \ldots \gg$

$$[s_{other}, Y_{other}] := \left(\begin{array}{ll} \alpha_{response} & \rightarrow [s_{response}, Y_{response}] \\ -\alpha_{response} & \rightarrow [s_{restitute}, Y_{restitute}] \end{array} \right)$$
$$[ID_{AB}, t_{restitute}] := event_{FEL}(FEL)$$
$$\alpha_{response} := (\#ID_{response} > 0) \wedge (t_{restitute} > t)$$

In the case that a $response$ message from the $ID_{response}$ vector is to be output, the vector itself is replaced with an empty vector.

$\ll \ldots ; [s_{other}, Y_{other}] \gg$

$$s_{response} := \left[\begin{array}{c} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, [\,] \end{array} \right]$$

Each $response$ message contains a particle ID, the species of that particle, the position $\vec{u}_A$ of the particle at time t_A, and the velocity of the particle.

$\ll\ldots\gg$

$$Y_{response}(i) := \left(\; i : \Diamond ID_{response} \quad \rightarrow [\text{“}response\text{”}, response] \; \right)$$
$$id_A := ID_{response}(i)$$
$$spc_A := \Psi(id_A)(\text{“}spc\text{”})$$
$$t_A := \Psi(id_A)(\text{“}t\text{”})$$
$$\vec{u}_A := \Psi(id_A)(\text{“}\vec{u}\text{”})$$
$$\vec{v}_A := \Psi(id_A)(\text{“}\vec{v}\text{”})$$
$$response := [id_A, spc_A, t_A, \vec{u}_A, \vec{v}_A]$$

When processing a restitution, the state time variable is updated with the time of the event. Appendix B's $restitute$ function provides the new Ψ, as well as a list of affected particles. The restitution event is removed from the FEL, and the past loading event is removed from $N_{loading}$. The $union$ function is used to append the new vector of responses unto the old vector, removing duplicates.

$\ll\ldots\gg$

$$s_{restitute} := \begin{bmatrix} t', \Psi', \Phi, FEL', N_{loading}', n_{loading}, ID_{attachment}, \ldots \\ ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response}' \end{bmatrix}$$
$$t' := t_{restitute}$$
$$[\Psi', ID_{restitute}] := restitute([\Psi, t', id_A, id_B, \Omega_\psi])$$
$$[id_A, id_B] := ID_{AB}$$
$$FEL' := \delta_{FEL}([FEL, ID_{AB}, \top, pr_{restitute}])$$
$$pr_{restitute} := get_{pr_{restitute}}(N_{loading})$$
$$N_{loading}' := N_{loading} \ntriangleleft [ID_{AB}]$$
$$ID_{response}' := union([ID_{response}, ID_{restitute}])$$

Finally, completing δ_{int}, the $restitution$ message is output.

$\ll\ldots\gg$

$$Y_{restitute} := [[\text{“}restitution\text{”}, ID_{AB}]]$$

All that remains in the definition of the $responder$ model is the time advance function.

$\ll DEVS\_tethered\_particle\_system; responder_{DEVS}; responder \gg$

$$ta\left(s\right) := \Delta t_{int}$$
$$\left[\begin{array}{l} t, \Psi, \Phi, FEL, N_{loading}, n_{loading}, ID_{attachment}, \ldots \\ \quad ID_{detachment}, ID_{impulse}, ID_{loading}, ID_{response} \end{array}\right] := s$$

The time before the next internal transition is zero if any of the ID vectors contain elements. Otherwise, it is the time of the next restitution event in the FEL.

$\ll \ldots; ta \gg$

$$\Delta t_{int} := \left(\begin{array}{ll} \alpha_{ID} & \rightarrow 0 \\ -\alpha_{ID} & \rightarrow \left(t_{restitute} - t\right) \vee 0 \end{array}\right)$$
$$\alpha_{ID} := \ldots$$
$$\left(\#ID_{attachment} > 0\right) \vee \ldots$$
$$\left(\#ID_{detachment} > 0\right) \vee \ldots$$
$$\left(\#ID_{impulse} > 0\right) \vee \ldots$$
$$\left(\#ID_{loading} > 0\right) \vee \ldots$$
$$\left(\#ID_{response} > 0\right)$$
$$\left[ID_{AB}, t_{restitute}\right] := event_{FEL}\left(FEL\right)$$

D.5 DEVS Tracker Model

As a submodel of the $detector$, the role of the $tracker$ is to redirect each incoming $response$ message to each subvolume that is aware of the particle identified by that message. Model parameters include the dimensions N of the lattice of subvolumes, the length a of each subvolume, the position $\vec{u}_{center}$ of the lattice center, and Ω_{ψ}.

$\ll DEVS\_tethered\_particle\_system \gg$

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

Also of use are the $position$ function of Appendix B, the position $\vec{u}_{corner}$ of the corner of the lattice, the number of dimensions n_{dim}, and the arrival and departure radii.

$\ll \ldots; tracker_{DEVS} \gg$

$$\begin{aligned}
&position := tethered\_particle\_system\left(\text{“}position\text{”}\right)\\
&n_{dim} := \#N\\
&\vec{u}_{corner} := corner\left(\left[N, a, \vec{u}_{center}\right]\right)\\
&\left[R_{arrival}, R_{departure}\right] := radii\left(\left[n_{dim}, a\right]\right)
\end{aligned}$$

State variables include a coordinates list CL and a response list RL. The response list is initially empty.

$\ll \ldots \gg$

$$\begin{aligned}
init_{tracker}\left(\Psi\right) &:= s\\
s &:= \left[CL, RL\right]\\
RL &:= [\,]
\end{aligned}$$

The coordinates list CL lists for each particle, the coordinates of all subvolumes that are aware of that particle. If the vector $CL\left(id_A\right)$ is missing the coordinates of a subvolume that should be aware of the particle with ID id_A, then that subvolume will not receive relevant $response$ messages; a situation that must be avoided. It is not a problem, however, if $CL\left(id_A\right)$ contains the coordinates of a subvolume that need not be aware of the particle. Based on this reasoning, we overestimate the number of subvolumes initially listed in CL. For each particle, we initially select a set of subvolumes that together represent a rectangular region. The boundaries of this set of subvolumes is identified by the coordinates of two subvolumes at opposite corners: $coords_{lower}$ and $coords_{upper}$.

$\ll \ldots; init_{tracker}; s \gg$

$$CL\left(id_A\right) := \Big(\ id_A : \Diamond\Psi \quad \rightarrow coords_{lower} + ..\left(coords_{upper} - coords_{lower} + 1\right)\ \Big)$$

To obtain the pair of bounding coordinates, we first calculate two sets of distances $\Delta\vec{u}_{lower}$ and $\Delta\vec{u}_{upper}$. When measured from the lower corner of the lattice, these distances yield positions on opposite corners of a rectangular volume that encompasses a particle's "region of influence". The region of influence, in this case, is a circle or sphere around a particle's center. The radius of the circle/sphere is the detection radius of the particle plus the maximum distance that a detection orb can extend beyond the boundary of a subvolume.

$\ll \ldots ; CL \gg$

$$\begin{bmatrix} \Delta\vec{u}_{lower} \\ \Delta\vec{u}_{upper} \end{bmatrix} := \begin{bmatrix} \vec{u}_A - \vec{u}_{corner} - r_A - \left(R_{departure} - \frac{a}{2}\right) \\ \vec{u}_A - \vec{u}_{corner} + r_A + \left(R_{departure} - \frac{a}{2}\right) \end{bmatrix}.$$
$$spc_A := \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right)$$
$$r_A := \Omega_\psi\left(spc_A\right)\left(\text{“}r\text{”}\right)$$
$$\vec{u}_A := position\left(\left[\Psi, 0, id_A\right]\right)$$

The coordinates used to initialize CL are derived as follows from the distances calculated above. The expression $\lfloor X \rfloor$ represents a vector with each element of X rounded down to an integer.

$\ll \ldots \gg$

$$coords_{lower} := \left\lfloor \frac{\Delta\vec{u}_{lower}}{a} \right\rfloor$$
$$coords_{upper} := \left\lfloor \frac{\Delta\vec{u}_{upper}}{a} \right\rfloor$$

We now shift our attention from the initialization function to the $tracker$ model itself.

$\ll DEVS\_tethered\_particle\_system; tracker_{DEVS} \gg$

$$tracker := \left[\delta_{ext}, \delta_{int}, ta\right]$$

We begin defining the external transition function by obtaining the state variables of s, as well as the port and message components of the input x.

$\ll \ldots ; tracker \gg$

$$\delta_{ext}\left(\left[s, \Delta t_e l, x\right]\right) := s'$$
$$\left[CL, RL\right] := s$$
$$\left[port, msg\right] := x$$

There are three types of input messages.

$\ll \ldots ; \delta_{ext} \gg$

$$s' := \begin{pmatrix} port \equiv \text{“}response\text{”} & \rightarrow s_{response} \\ port \equiv \text{“}arrival\text{”} & \rightarrow s_{arrival} \\ port \equiv \text{“}departure\text{”} & \rightarrow s_{departure} \end{pmatrix}$$

A $response$ message is simply added to the response list state variable.

$\ll \ldots ; s' \gg$

$$\begin{aligned} s_{response} &:= [CL, RL'] \\ RL' &:= RL \parallel [msg] \end{aligned}$$

An $arrival$ message indicates that a particular subvolume with ID $coords$ has gained awareness of a certain particle with ID id_A. The coordinates list is updated accordingly.

$\ll \ldots \gg$

$$\begin{aligned} s_{arrival} &:= [CL', RL] \\ [id_A, coords] &:= msg \\ CL' &:= \left(\begin{array}{ll} coords : CL(id_A) & \to CL \\ -(coords : CL(id_A)) & \to CL_\Delta \end{array} \right) \\ CL_\Delta &:= CL \lhd \left[\begin{array}{ll} id_A & \to CL(id_A) \parallel [coords] \end{array} \right] \end{aligned}$$

A $departure$ message indicates that a particular subvolume with ID $coords$ has lost awareness of a certain particle with ID id_A.

$\ll \ldots \gg$

$$\begin{aligned} s_{departure} &:= [CL', RL] \\ [id_A, coords] &:= msg \\ CL' &:= \left(\begin{array}{ll} -(coords : CL(id_A)) & \to CL \\ coords : CL(id_A) & \to CL_\Delta \end{array} \right) \\ CL_\Delta &:= CL \lhd \left[\begin{array}{ll} id_A & \to del([CL(id_A), coords]) \end{array} \right] \end{aligned}$$

With δ_{ext} complete, we look at the internal transition function. Internal transitions occur only when there are $response$ messages to output. The response list is then emptied.

$\ll DEVS\_tethered\_particle\_system; tracker_{DEVS}; tracker \gg$

$$\begin{aligned} \delta_{int}(s) &:= [s', Y] \\ [CL, RL] &:= s \\ s' &:= [CL, [\,]] \end{aligned}$$

For each received $response$ in RL, a separate $response$ message is output for each set of coordinates listed in CL for the associated particle. Note that Y_{RL} is a vector of vectors of $response$ messages, whereas Y, defined as $\parallel Y_{RL}$, is simply a vector of $response$ messages.

$\ll \ldots; \delta_{int} \gg$

$$
\begin{aligned}
&Y := \parallel Y_{RL} \\
&\quad Y_{RL}(i) := \left(\; i : \Diamond RL \quad \rightarrow Y_{RL_{CL}} \; \right) \\
&\qquad response := RL(i) \\
&\qquad [id_A, spc_A, t_A, \vec{u}_A, \vec{v}_A] := response \\
&\qquad Y_{RL_{CL}}(j) := \left(\; j : \Diamond (CL(id_A)) \quad \rightarrow [port, response] \; \right) \\
&\qquad\quad coords := CL(id_A)(j) \\
&\qquad\quad port := [\text{“}response\text{”}, coords]
\end{aligned}
$$

The time advance function yields zero if there are $response$ messages to output, and infinity otherwise.

$\ll DEVS\_tethered\_particle\_system; tracker_{DEVS}; tracker \gg$

$$
\begin{aligned}
&ta(s) := \Delta t_{int} \\
&\quad [CL, RL] := s \\
&\quad \Delta t_{int} := \begin{pmatrix} \#RL > 0 & \rightarrow 0 \\ \#RL = 0 & \rightarrow \top \end{pmatrix}
\end{aligned}
$$

D.6 DEVS Lattice Model

Here we define the $lattice$, the other submodel of the $detector$. First we obtain several functions from Appendix B and Appendix C.

$\ll DEVS\_tethered\_particle\_system \gg$

$$
\begin{aligned}
&lattice_{DEVS}([N, a, \vec{u}_{center}, \Omega_{\psi}, \Omega_{\psi\psi}, \Delta t_{max}]) := [init_{lattice}, lattice] \\
&\quad \begin{bmatrix} position \\ detect \end{bmatrix} := tethered\_particle\_system \circ \begin{bmatrix} \text{“}position\text{”} \\ \text{“}detect\text{”} \end{bmatrix} \\
&\quad \begin{bmatrix} FEL_{empty} \\ \delta_{FEL} \\ event_{FEL} \end{bmatrix} := future\_events\_list \circ \begin{bmatrix} \text{“}FEL_{empty}\text{”} \\ \text{“}\delta_{FEL}\text{”} \\ \text{“}event_{FEL}\text{”} \end{bmatrix} \\
&\quad pr_{\varnothing} := DEVS(\text{“}pr_{\varnothing}\text{”})
\end{aligned}
$$

Useful variables include the number of dimensions and the position of the corner of the lattice.

$\ll \ldots; lattice_{DEVS} \gg$

$$n_{dim} := \#N$$
$$\vec{u}_{corner} := corner\left([N, a, \vec{u}_{center}]\right)$$

The bulk of our efforts will go into definition of the subvolume model $subV$. Its sole model parameter is its coordinates.

$\ll \ldots \gg$

$$subV_{DEVS}\left(coords\right) := [init_{subV}, subV]$$

The function $detect_{event}$ results in the time remaining until the event identified by id_{event} occurs. The parameters Ψ and t are state variables that together capture the present positions and velocities of each particle.

$\ll \ldots; subV_{DEVS} \gg$

$$detect_{event}\left([\Psi, t, id_{event}]\right) := \Delta t$$

Events all consist of an event type and two IDs.

$\ll \ldots; detect_{event} \gg$

$$[type_{event}, ID_{AB}] := id_{event}$$
$$[id_A, id_B] := ID_{AB}$$

There are four types of events, including "blocking" and "tethering" events associated with particle collisions, "approach" events that occur when a particle reaches the arrival orb of a neighboring subvolume, and "departure" events that occur when a particle leaves the departure orb of the subvolume identified by $coords$. The mathematics of detecting blocking and approach events is the same, as is the mathematics of tethering and departure events. The calculation is performed by Appendix B's $detect$ function.

$\ll \ldots; detect_{event} \gg$

$$\Delta t_{event} := detect\left([\Psi, t, id_A, id_B, \Omega_{\psi\psi}, \Delta t_{max}, type_{collision}]\right)$$
$$type_{collision} := \begin{pmatrix} type_{event} : [\text{“}blocking\text{”}, \text{“}approach\text{”}] & \rightarrow \text{“}blocking\text{”} \\ type_{event} : [\text{“}tethering\text{”}, \text{“}departure\text{”}] & \rightarrow \text{“}tethering\text{”} \end{pmatrix}$$

The time Δt_{event} produced by $detect$ is the final result, unless it is less than infinity and we are dealing with a collision between two particles. We know that we are dealing with a particle-particle collision when both id_A and id_B are natural numbers.

$\ll \ldots \gg$

$$\Delta t := \left(\begin{array}{ll} (\Delta t_{event} = \top) \vee (-\mathbb{N}(id_A) \vee -\mathbb{N}(id_B)) & \rightarrow \Delta t_{event} \\ (\Delta t_{event} < \top) \wedge (\mathbb{N}(id_A) \wedge \mathbb{N}(id_B)) & \rightarrow \Delta t_{collision} \end{array} \right)$$

For a particle-particle collision, we wish to schedule the event only if the collision point $\vec{u}_{collision}$ is located within the subvolume identified by $coords$. If it is not in this subvolume, then we give a result of infinity.

$\ll \ldots ; \Delta t \gg$

$$\Delta t_{collision} := \left(\begin{array}{lll} coords_{collision} & \equiv coords & \rightarrow \Delta t_{event} \\ coords_{collision} & \not\equiv coords & \rightarrow \top \end{array} \right)$$

The collision point $\vec{u}_{collision}$ is calculated below, along with the coordinates $coords_{collision}$ of the subvolume that contains it.

$\ll \ldots ; \Delta t_{collision} \gg$

$$\begin{aligned} &\vec{u}_A{}' := position\left([\Psi, t + \Delta t_{event}, id_A]\right) \\ &\vec{u}_B{}' := position\left([\Psi, t + \Delta t_{event}, id_B]\right) \\ &spc_A := \Psi(id_A)(\text{“}spc\text{”}) \\ &spc_B := \Psi(id_B)(\text{“}spc\text{”}) \\ &r_A := \Omega_\psi(spc_A)(\text{“}r\text{”}) \\ &r_B := \Omega_\psi(spc_B)(\text{“}r\text{”}) \\ &\vec{u}_{collision} := \frac{r_B \cdot \vec{u}_A{}' + r_A \cdot \vec{u}_B{}'}{r_A + r_B} \\ &coords_{collision} := \left\lfloor \frac{\vec{u}_{collision} - \vec{u}_{corner}}{a} \right\rfloor \end{aligned}$$

The initial state of a $subV$ model does not depend on any parameters, so we use a dummy variable as the argument of the initialization function.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS} \gg$

$$init_{subV}(dummy) := s$$

Along with t and Ψ, a subvolume's state variables include a list of future events and a list of arrival messages to be output.

$\ll \ldots ; init_{subV} \gg$

$$s := [t, \Psi, FEL, ID_{arrival}]$$

The initial time is zero.

$\ll \ldots ; s \gg$

$$t := 0$$

Note that Ψ was not passed in as an argument. Particle information is to be recorded only when $response$ messages are received. Because we designed the $responder$ to output response messages immediately, subvolumes will become aware of particles after the simulation starts but before time advances.

Despite the absence of particles, the initial Ψ is not empty. We add $(2{\cdot}n_{dim} + 1)$ "orbs" to the state variable, one for each of the neighboring subvolumes and one for the subvolume associated with $coords$. It is the presence of these orbs in Ψ that allow us to use the $detect$ function to identify approach and departure events as well as particle-particle collisions. The identity of the orb associated with a neighboring subvolume takes the form $[i_{dim}, dr]$, where i_{dim} identifies a dimension and dr a direction. The identity of the orb associated with $coords$ is $\varnothing$.

$\ll \ldots \gg$

$$\Psi\,(heading) := \Big(\; heading : H \;\; \rightarrow \psi_A \;\Big)$$
$$H := [\varnothing] \parallel ID_{adj}$$
$$ID_{adj}\,(i) := \Big(\; i : ..\,(2{\cdot}n_{dim}) \;\; \rightarrow [i_{dim}, dr] \;\Big)$$
$$i_{dim} := \left\lfloor \frac{i}{2} \right\rfloor$$
$$dr := 2{\cdot}mod\,([i, 2]) - 1$$

Each orb has a species of $\varnothing$.

$\ll \ldots ; \Psi \gg$

$$\psi_A := \begin{bmatrix} \text{"}spc\text{"} & \rightarrow \varnothing \\ \text{"}t\text{"} & \rightarrow 0 \\ \text{"}\vec{u}\text{"} & \rightarrow \vec{u}_A \\ \text{"}\vec{v}\text{"} & \rightarrow \vec{v}_A \end{bmatrix}$$

The position of each orb is the center of its subvolume, and the orb's velocity is represented by a vector of zeros.

$\ll \ldots; \psi_A \gg$

$$
\begin{aligned}
&[i_{dim}, dr] := heading \\
&coords_A := \begin{pmatrix} heading \;\equiv\; \varnothing & \rightarrow coords \\ heading \;\not\equiv\; \varnothing & \rightarrow coords_{adj} \end{pmatrix} \\
&\qquad coords_{adj} := coords \lhd \left[\; i_{dim} \quad \rightarrow coords\,(i_{dim}) + dr \;\right] \\
&\vec{u}_A := \vec{u}_{corner} + a\cdot\Big(coords_A + \frac{1}{2}\Big) \\
&\vec{v}_A := 0\cdot..n_{dim}
\end{aligned}
$$

With no particles, initially, the FEL and list of $arrival$ messages are both empty.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS}; init_{subV}; s \gg$

$$
\begin{aligned}
&FEL := FEL_{empty} \\
&ID_{arrival} := [\,]
\end{aligned}
$$

The DEVS model itself is now defined.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS} \gg$

$$subV := [\delta_{ext}, \delta_{int}, ta]$$

We start, as usual, with the external transition function.

$\ll \ldots; subV \gg$

$$\delta_{ext}\,([s, \Delta t_{el}, x]) := s'$$

First we obtain state variables, the new time t', and the port and message components of x. The port has general and specific components. The message, regardless of its type, has the five listed particle attributes.

$\ll \ldots; \delta_{ext} \gg$

$$
\begin{aligned}
&[t, \Psi, FEL, ID_{arrival}] := s \\
&t' := t + \Delta t_{el} \\
&[port, msg] := x \\
&[general, specific] := port \\
&[id_A, spc_A, t_A, \vec{u}_A, \vec{v}_A] := msg
\end{aligned}
$$

Consider first the case of an adj message, where a particle has approached from an adjacent subvolume. If the present subvolume is already aware of the particle, there is nothing to do except update the time. Consider now the case of a $response$ message, or an adj message describing a new particle. In this situation the message contains new particle information that must be incorporated into the state.

$\ll \ldots \gg$

$$s' := \left(\begin{array}{lll} (general \ \equiv \ \text{“}adj\text{”}) \wedge (id_A : \Diamond\Psi) & \rightarrow [t', \Psi, FEL, ID_{arrival}] \\ (general \ \equiv \ \text{“}response\text{”}) \vee -(id_A : \Diamond\Psi) & \rightarrow [t', \Psi', FEL', ID_{arrival}'] \end{array} \right)$$

New particle information is used to update Ψ.

$\ll \ldots ; s' \gg$

$$\Psi' := \Psi \lhd \left(id_A \ \rightarrow \left[\begin{array}{ll} \text{“}spc\text{”} & \rightarrow spc_A \\ \text{“}t\text{”} & \rightarrow t_A \\ \text{“}\vec{u}\text{”} & \rightarrow \vec{u}_A \\ \text{“}\vec{v}\text{”} & \rightarrow \vec{v}_A \end{array} \right] \right)$$

The particle A of the $response$ message must be compared with each orb or particle B listed in Ψ, except for itself. If id_B is a natural number, then B is a particle, and blocking and tethering events must be considered. If id_B is a function, then it is a vector of the form $[i_{dim}, dr]$, meaning that B is the orb of an adjacent subvolume, meaning that approach events must be considered. If id_B is $\varnothing$, then B is the orb of the present subvolume, meaning that departure events must be considered.

$\ll \ldots \gg$

$$FEL' := loop_\psi([0, FEL])$$

$$ID_\psi := \Diamond(\Psi' \ntriangleleft [id_A])$$

$$loop_\psi([i, FEL_i]) := \left(\begin{array}{ll} i = \#ID_\psi & \rightarrow FEL_i \\ i < \#ID_\psi & \rightarrow loop_\psi([i+1, FEL_i']) \end{array} \right)$$

$$id_B := ID_\psi(i)$$

$$ID_{AB} := \left(\begin{array}{ll} \mathbb{N}(id_B) & \rightarrow sort([id_A, id_B]) \\ -\mathbb{N}(id_B) & \rightarrow [id_A, id_B] \end{array} \right)$$

$$events := \left(\begin{array}{ll} \mathbb{N}(id_B) & \rightarrow [\text{“}blocking\text{”}, \text{“}tethering\text{”}] \\ \mathbb{F}(id_B) & \rightarrow [\text{“}approach\text{”}] \\ id_B \ \equiv \ \varnothing & \rightarrow [\text{“}departure\text{”}] \end{array} \right)$$

For each particle B indexed by i, and for each type of event indexed by j, the future event is added to the FEL.

$\ll \ldots; FEL'; loop_{\psi} \gg$

$$\begin{aligned}
&FEL_i{}' := loop_{event}\left([0, FEL_i]\right)\\
&\qquad loop\left([j, FEL_j]\right) := \begin{pmatrix} j = \#events & \rightarrow FEL_j \\ j < \#events & \rightarrow loop\left([j+1, FEL_j{}']\right) \end{pmatrix}\\
&\qquad\qquad type_{event} := events\left(j\right)\\
&\qquad\qquad id_{event} := [type_{event}, ID_{AB}]\\
&\qquad\qquad t_{event} := t' + detect_{event}\left([\Psi', t', id_{event}]\right)\\
&\qquad\qquad FEL_j{}' := \delta_{FEL}\left([FEL_j, id_{event}, t_{event}, pr_{\varnothing}]\right)
\end{aligned}$$

If the message was an adj message, and if the particle it described is new to the present subvolume, then the particle ID is added to $ID_{arrival}$. This completes the external transition function.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS}; subV; \delta_{ext}; s' \gg$

$$ID_{arrival}{}' := \begin{pmatrix} general \equiv \text{“}adj\text{”} & \rightarrow ID_{arrival} \parallel [id_A] \\ general \equiv \text{“}response\text{”} & \rightarrow ID_{arrival} \end{pmatrix}$$

In the internal transition function, there are two cases to consider. If $ID_{arrival}$ contains IDs, they must be sent as $arrival$ messages to the tracker. Otherwise, we must process an event from the FEL.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS}; subV \gg$

$$\begin{aligned}
&\delta_{int}\left(s\right) := [s', Y]\\
&\qquad [t, \Psi, FEL, ID_{arrival}] := s\\
&\qquad [s', Y] := \begin{pmatrix} \#ID_{arrival} > 0 & \rightarrow [s_{arrival}, Y_{arrival}] \\ \#ID_{arrival} = 0 & \rightarrow [s_{other}, Y_{other}] \end{pmatrix}
\end{aligned}$$

In first case, $ID_{arrival}$ is emptied and its contents are used to form individual output messages.

$\ll \ldots ; \delta_{int} ; [s', Y] \gg$

$$
\begin{aligned}
& s_{arrival} := [t, \Psi, FEL, [\,]] \\
& Y_{arrival}(i) := \left(\; i : \Diamond ID_{arrival} \quad \rightarrow [port, msg] \; \right) \\
& \qquad id_A := ID_{arrival}(i) \\
& \qquad port := [\text{“}out\text{”}, \text{“}arrival\text{”}] \\
& \qquad msg := [id_A, coords]
\end{aligned}
$$

In the second case, we obtain the next event from the FEL and remove it. We also check to see whether we are dealing with an approach event, a particle-particle collision, or a departure event.

$\ll \ldots \gg$

$$
[s_{other}, Y_{other}] := \begin{pmatrix} type_{event} \;\equiv\; \text{“}approach\text{”} & \rightarrow [s_{approach}, Y_{approach}] \\ type_{event} : [\text{“}blocking\text{”}, \text{“}tethering\text{”}] & \rightarrow [s_{collision}, Y_{collision}] \\ type_{event} \;\equiv\; \text{“}departure\text{”} & \rightarrow [s_{departure}, Y_{departure}] \end{pmatrix}
$$

$$
\begin{aligned}
& [id_{event}, t'] := event_{FEL}(FEL) \\
& FEL_{\Delta} := \delta_{FEL}([FEL, id_{event}, \top, pr_{\varnothing}]) \\
& [type_{event}, ID_{AB}] := id_{event} \\
& [id_A, id_B] := ID_{AB}
\end{aligned}
$$

If the event is an approach, an adj message is prepared to inform the adjacent subvolume.

$\ll \ldots ; [s_{other}, Y_{other}] \gg$

$$
\begin{aligned}
& s_{approach} := [t', \Psi, FEL_{\Delta}, [\,]] \\
& Y_{approach} := [[port, msg]] \\
& \qquad port := [\text{“}adj\text{”}, [i_{dim}, dr]] \\
& \qquad\qquad [i_{dim}, dr] := id_B \\
& \qquad msg := [id_A, spc_A, t_A, \vec{u}_A, \vec{v}_B] \\
& \qquad\qquad spc_A := \Psi(id_A)(\text{“}spc\text{”}) \\
& \qquad\qquad t_A := \Psi(id_A)(\text{“}t\text{”}) \\
& \qquad\qquad \vec{u}_A := \Psi(id_A)(\text{“}\vec{u}\text{”}) \\
& \qquad\qquad \vec{v}_A := \Psi(id_A)(\text{“}\vec{v}\text{”})
\end{aligned}
$$

If the event is a particle-particle collision, a $collision$ message is prepared that ultimately goes to the $responder$.

$\ll \ldots \gg$

$$\begin{aligned} s_{collision} &:= [t', \Psi, FEL_{\Delta}, [\,]] \\ Y_{collision} &:= [[port, msg]] \\ port &:= [\text{“}out\text{”}, \text{“}collision\text{”}] \\ msg &:= [ID_{AB}, type_{event}] \end{aligned}$$

If the event is a departure, then first the departing particle is removed from Ψ.

$\ll \ldots \gg$

$$\begin{aligned} s_{departure} &:= [t', \Psi', FEL', [\,]] \\ \Psi' &:= \Psi \ntriangleleft [id_A] \end{aligned}$$

Next, in the case of a departure, the departing particle is compared in a loop to every particle and orb remaining in Ψ.

$\ll \ldots; s_{departure} \gg$

$$\begin{aligned} FEL' &:= loop_{\psi}\left([0, FEL_{\Delta}]\right) \\ ID_{\psi} &:= \Diamond \Psi' \\ loop_{\psi}\left([i, FEL_i]\right) &:= \begin{pmatrix} i = \#ID_{\psi} & \rightarrow FEL_i \\ i < \#ID_{\psi} & \rightarrow loop_{\psi}\left([i+1, FEL_i{}']\right) \end{pmatrix} \\ id_C &:= ID_{\psi}\left(i\right) \\ ID_{AC} &:= \begin{pmatrix} \mathbb{N}\left(id_C\right) & \rightarrow sort\left([id_A, id_C]\right) \\ -\mathbb{N}\left(id_C\right) & \rightarrow [id_A, id_C] \end{pmatrix} \\ events &:= \begin{pmatrix} \mathbb{N}\left(id_C\right) & \rightarrow [\text{“}blocking\text{”}, \text{“}tethering\text{”}] \\ \mathbb{F}\left(id_C\right) & \rightarrow [\text{“}approach\text{”}] \\ id_C \equiv \varnothing & \rightarrow [\text{“}departure\text{”}] \end{pmatrix} \end{aligned}$$

Instead of adding events to the FEL, as done by the analogous set of nested loops in δ_{ext}, here we remove events. Note that the future event times below are all $\top$.

$\ll \ldots; FEL'; loop_\psi \gg$

$$\begin{aligned}
&FEL_i{}' := loop_{event}\left([0, FEL_i]\right)\\
&\qquad loop\left([j, FEL_j]\right) := \left(\begin{array}{ll} j = \#events & \rightarrow FEL_j \\ j < \#events & \rightarrow loop\left([j+1, FEL_j{}']\right)\end{array}\right)\\
&\qquad\quad type_{event} := events\left(j\right)\\
&\qquad\quad id_{event} := [type_{event}, ID_{AC}]\\
&\qquad\quad FEL_j{}' := \delta_{FEL}\left([FEL_j, id_{event}, \top, pr_{\varnothing}]\right)
\end{aligned}$$

The identity of the departing particle is output to inform the $tracker$.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS}; subV; \delta_{int}; [s', Y]; [s_{other}, Y_{other}] \gg$

$$\begin{aligned}
&Y_{departure} := [[port, msg]]\\
&\qquad port := [\text{“}out\text{”}, \text{“}departure\text{”}]\\
&\qquad msg := [id_A, coords]
\end{aligned}$$

The $subV$ time advance function yields zero if there are $arrival$ messages to output. Otherwise, the time of the next event is obtained from the FEL.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS}; subV_{DEVS}; subV \gg$

$$\begin{aligned}
&ta\left(s\right) := \Delta t_{int}\\
&\qquad [t, \Psi, FEL, ID_{arrival}] := s\\
&\qquad \Delta t_{int} := \left(\begin{array}{ll} \#ID_{arrival} > 0 & \rightarrow 0 \\ \#ID_{arrival} = 0 & \rightarrow (t_{event} - t) \vee 0\end{array}\right)\\
&\qquad\quad [id_{event}, t_{event}] := event_{FEL}\left(FEL\right)
\end{aligned}$$

With individual subvolume models now fully defined, we must complete the definition of the lattice model that contains them. The $lattice$ is a DEVS hypercubic lattice model, and so we use HL_{DEVS}. The second parameter of HL_{DEVS} is the DEVS model function of each submodel of the lattice; that is, $subV_{DEVS}$. Note that the priority function $pr_{\varnothing}$ is used, which randomizes the order of simultaneous internal transitions. This is a logical convention for hypercubic lattice models. In the TPS and $detector$ coupled models, the order of submodels was deliberately fixed for good reason; but in those cases, the submodels had very distinct roles. In the case of lattice models representing space, each submodel generally performs a similar role to that of other submodels.

$\ll DEVS\_tethered\_particle\_system; lattice_{DEVS} \gg$

$$[init_{HL}, lattice] := HL_{DEVS}\left([N, subV_{DEVS}, pr_{\varnothing}]\right)$$

All that remains is the initialization function of the $lattice$. Recall that the initialization function of $subV$ required only a dummy argument. Using $init_{HL}$ as we are, this $init_{subV}$ argument is the result of $ARG_{lattice}$ when applied to the coordinates of the subvolume. Because our $ARG_{lattice}$ always results in $\varnothing$, $init_{HL}$ will evaluate $init_{subV}(\varnothing)$ to obtain the initial state of each $subV$.

$\ll \ldots \gg$

$$init_{lattice}(dummy) := s$$
$$s := init_{HL}(ARG_{lattice})$$
$$ARG_{lattice}(coords) := \varnothing$$

E Presynaptic Nerve Terminal Model Parameters

Here we present the TPS model parameters used for the simulations described in Section 5.1, which capture the formation of vesicle clusters in presynaptic nerve terminals. All definitions below pertain only to this appendix.

Recall that TPS_{DEVS} takes the following arguments.

$$[N, a, \vec{u}_{center}, \omega_{\psi}, \omega_{\psi\psi}, attach, detach, \theta_{revolve}, \Delta t_{max}, \Delta t_{restitute}]$$

The lattice dimensions of N determine the number of subvolumes used to detect particle collisions. We use 14 subvolumes along both horizontal axes, and 20 along the vertical axis which must accomodate the Z particle. Each subvolume has a length of 50 nm along each axis, and the entire region is centered at $[0, 0, 0]$.

$$N \equiv [14, 20, 14]$$
$$a = 50$$
$$\vec{u}_{center} \equiv [0, 0, 0]$$

Presynaptic compartments vary in size and shape, but we use a sphere of radius 250 nm. We represent the active zone as a region of the surface of that sphere. The active zone has a circular boundary of radius 150 nm. Although they are not necessarily spherical in reality, we approximate the sizes of docking sizes, vesicles, and halves of synapsins with radius values of 5 nm, 20 nm, and 2.5 nm respectively.

$$r_M = 250$$
$$r_Z = 150$$
$$r_D = 5$$
$$r_V = 20$$
$$r_S = 2.5$$

We assign the relative masses of vesicles and synapsins to be proportional to the volume of space they occupy as estimated from their radii. Although docking site particles are considerably smaller than vesicle particles, we give both types equal mass in order to reduce the motion of docking sites.

$$\begin{aligned} m_V &= \frac{4}{3}\cdot\pi\cdot{r_V}^2 \\ m_S &= \frac{4}{3}\cdot\pi\cdot{r_S}^2 \\ m_D &= m_V \end{aligned}$$

The parameters of a TPS model include average momentum values associated with random impulses. We find it convenient to select speed values instead, each of which is later multiplied by a particle mass. For example, the vesicle speed v_V may be multiplied with the vesicle mass m_V to yield the average momentum of a random vesicle impulse.

$$\begin{aligned} v_D &= 0.125 \\ v_V &= 2 \\ v_S &= 0.5 \end{aligned}$$

The masses and speeds above are used to define the model parameter ω_ψ, which contains information pertaining to individual particle species.

$$\omega_\psi \equiv \begin{bmatrix} \text{“}M\text{”} & \rightarrow \begin{bmatrix} \text{“}m\text{”} & \rightarrow \top \end{bmatrix} \\ \text{“}Z\text{”} & \rightarrow \begin{bmatrix} \text{“}m\text{”} & \rightarrow \top \end{bmatrix} \\ \text{“}D\text{”} & \rightarrow \begin{bmatrix} \text{“}m\text{”} & \rightarrow m_D \\ \text{“}\tau_{RI}\text{”} & \rightarrow 1 \\ \text{“}k_{RI}\text{”} & \rightarrow 64 \\ \text{“}\mu_{RI}\text{”} & \rightarrow m_D\cdot v_{RI_D} \end{bmatrix} \\ \text{“}V\text{”} & \rightarrow \begin{bmatrix} \text{“}m\text{”} & \rightarrow m_V \\ \text{“}\tau_{RI}\text{”} & \rightarrow 2 \\ \text{“}k_{RI}\text{”} & \rightarrow 16 \\ \text{“}\mu_{RI}\text{”} & \rightarrow m_D\cdot v_{RI_V} \end{bmatrix} \\ \text{“}S\text{”} & \rightarrow \begin{bmatrix} \text{“}m\text{”} & \rightarrow m_S \\ \text{“}\tau_{RI}\text{”} & \rightarrow 4 \\ \text{“}k_{RI}\text{”} & \rightarrow 4 \\ \text{“}\mu_{RI}\text{”} & \rightarrow m_D\cdot v_{RI_S} \end{bmatrix} \end{bmatrix}$$

We select a "default" coefficient of restitution to apply to collisions of various types of particles.

$$c_{restitute} = 0.1$$

Recall that the parameter $\omega_{\psi\psi}$ records blocking and tethering distances and restitution coefficients for each pair of particle species. As our presynaptic nerve terminal model is designed to have exactly one M particle, M-M collisions cannot occur. It is still important to assign a blocking distance that reflects the approximate size of the particle, for this distance is used to compute the M species detection radii (see Appendix D). We make a habit of assigning a rebounding coefficient whenever the blocking distance is positive, which is why the $c_{restitute}$ value appears below. Because M-M tethering collisions will never occur, we may safely use $\varnothing$ for the retraction and revolution coefficients, and infinity ($\top$) for the tethering distance.

$$\omega_{\psi\psi}\left(\left[\text{“}M\text{”}, \text{“}M\text{”}\right]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 2{\cdot}r_M \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Because we also have only one immobile Z particle to worry about, we define $\omega_{\psi\psi}\left(\left[\text{“}Z\text{”}, \text{“}Z\text{”}\right]\right)$ in a similar manner to $\omega_{\psi\psi}\left(\left[\text{“}M\text{”}, \text{“}M\text{”}\right]\right)$.

$$\omega_{\psi\psi}\left(\left[\text{“}Z\text{”}, \text{“}Z\text{”}\right]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 2{\cdot}r_Z \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Because the M and Z particles are immobile, they never collide. Below we assign a positive blocking distance and a rebounding coefficient, though it is not necessary.

$$\omega_{\psi\psi}\left([\text{“}Z\text{”}, \text{“}M\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow r_Z + r_M \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Generally speaking, we add the radii of two particles to obtain the blocking distance, as done below for the case of two docking sites. Note that docking sites never become tethered together.

$$\omega_{\psi\psi}\left([\text{“}D\text{”}, \text{“}D\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 2 \cdot r_D \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Docking sites undergo both blocking collisions and tethering collisions with the M particle, which keeps them near the membrane of the presynaptic compartment. They are free to move along the membrane, but perpendicular to the membrane they are given only a distance of r_D to maneuver. Because we have assigned a finite tethering distance, we use $c_{restitute}$ for the retraction and revolution coefficients.

$$\omega_{\psi\psi}\left([\text{“}D\text{”}, \text{“}M\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow r_M - r_D \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow r_M \end{bmatrix}$$

Because docking sites are tethered to the Z particle, they are constrained to the active zone of the membrane. Note that if a docking site is displayed in a visualization program as a sphere of radius r_D, the tethering distance of $r_Z - r_D$ keeps the sphere completely within the boundary of the Z sphere of radius r_Z. Because we do not want to prevent docking sites from moving towards the center of the Z particle, we choose a blocking distance of zero. Guaranteed, as we are, that D-Z blocking collisions do not occur, we may use $\varnothing$ for the rebounding coefficient.

$$\omega_{\psi\psi}\left([\text{“}D\text{”}, \text{“}Z\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow r_Z - r_D \end{bmatrix}$$

Two vesicles may undergo a blocking collision at a distance of twice the vesicle radius.

$$\omega_{\psi\psi}\left([\text{“}V\text{”}, \text{“}V\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 2{\cdot}r_V \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

The tethering distance of $r_M - r_V$ keeps vesicles inside the compartment.

$$\omega_{\psi\psi}\left([\text{“}V\text{”}, \text{“}M\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow r_M - r_V \end{bmatrix}$$

Vesicles do not interact with the Z particle.

$$\omega_{\psi\psi}\left([\text{“}V\text{”}, \text{“}Z\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Vesicles may undergo both blocking and tethering collisions with docking sites.

$$\omega_{\psi\psi}\left([\text{“}V\text{”}, \text{“}D\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow r_V + r_D \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow r_V + 2{\cdot}r_D \end{bmatrix}$$

Synapsin particles are tethered together in pairs. In the images of Section 5, the synapsin particles are shown with radii of $2{\cdot}r_S$ instead of r_S. This is done in order to make tethered pairs of S particles appear to remain connected.

$$\omega_{\psi\psi}\left([\text{“}S\text{”}, \text{“}S\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow r_S \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow 3{\cdot}r_S \end{bmatrix}$$

The tethering distance of $r_M - r_S$ keeps synapsin particles inside the compartment.

$$\omega_{\psi\psi}\left([\text{“}S\text{”}, \text{“}M\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow r_M - r_S \end{bmatrix}$$

Synapsins do not interact with the Z particle.

$$\omega_{\psi\psi}\left([\text{“}S\text{”}, \text{“}Z\text{”}]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Synapsins do not interact with docking sites.

$$\omega_{\psi\psi}\left(\left[\text{“}S\text{”}, \text{“}D\text{”}\right]\right) \equiv \begin{bmatrix} s\text{“}c_{rebound}\text{”} & \rightarrow \varnothing \\ \text{“}c_{retract}\text{”} & \rightarrow \varnothing \\ \text{“}c_{revolve}\text{”} & \rightarrow \varnothing \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow 0 \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow \top \end{bmatrix}$$

Synapsins may undergo both blocking and tethering collisions with vesicles. This completes the specification of $\omega_{\psi\psi}$.

$$\omega_{\psi\psi}\left(\left[\text{“}S\text{”}, \text{“}V\text{”}\right]\right) \equiv \begin{bmatrix} \text{“}c_{rebound}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{retract}\text{”} & \rightarrow c_{restitute} \\ \text{“}c_{revolve}\text{”} & \rightarrow c_{restitute} \\ \text{“}\Delta u_{blocking}\text{”} & \rightarrow r_S + r_V \\ \text{“}\Delta u_{tethering}\text{”} & \rightarrow 2{\cdot}r_S + r_V \end{bmatrix}$$

We now pursue a definition of the model parameter $attach$, the function invoked at each blocking collision between untethered particles. We start by defining a few other functions, starting with free_{ψ}. This function takes the particle information selector Ψ, a particle ID id_A, and a species spc_B. It results in $\top$ if the particle identified by id_A is not tethered to any particle of species spc_B. Otherwise the result is $\bot$.

```
$\mathit{free}_{\psi}\left(\left[\Psi, id_A, spc_B\right]\right) := \alpha$
    $ID_{tethered} := \Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right)$
    $\alpha := loop\left(0\right)$
        $loop\left(i\right) := \begin{pmatrix} i = \#ID_{tethered} & \rightarrow \top \\ i < \#ID_{tethered} & \rightarrow check \end{pmatrix}$
            $id_{tethered} := ID_{tethered}\left(i\right)$
            $spc_{tethered} := \Psi\left(id_{tethered}\right)\left(\text{“}spc\text{”}\right)$
            $check := \begin{pmatrix} spc_{tethered} \equiv spc_B & \rightarrow \bot \\ spc_{tethered} \not\equiv spc_B & \rightarrow loop\left(i+1\right) \end{pmatrix}$
```

Given Ψ and the ID id_S of a synapsin particle, $\mathit{identify}_{counterpart}$ results in the ID of the unique synapsin particle tethered to it.

```
$\textit{identify}_{\textit{counterpart}}\left([\Psi, id_S]\right) := id_{counterpart}$
    $ID_{tethered} := \Psi\left(id_A\right)\left(\text{“}tethered\text{”}\right)$
    $id_{counterpart} := \textit{loop}\left(0\right)$
        $\textit{loop}\left(i\right) := \left(\begin{array}{ll} i < \#ID_{tethered} & \rightarrow \textit{check} \end{array}\right)$
            $id_{tethered} := ID_{tethered}\left(i\right)$
            $spc_{tethered} := \Psi\left(id_{tethered}\right)\left(\text{“}spc\text{”}\right)$
            $\textit{check} := \left(\begin{array}{lll} spc_{tethered} \equiv \text{“}S\text{”} & \rightarrow id_{tethered} \\ spc_{tethered} \not\equiv \text{“}S\text{”} & \rightarrow \textit{loop}\left(i+1\right) \end{array}\right)$
```

Given Ψ, the ID id_S of a synapsin particle, and the ID id_V of a vesicle, $\textit{free}_{\textit{counterpart}}$ results in $\top$ if the synapsin particle's counterpart is not tethered to that vesicle. Otherwise the result is $\perp$.

```
$\textit{free}_{\textit{counterpart}}\left([\Psi, id_S, id_V]\right) := id_{counterpart}$
    $id_{counterpart} := \textit{identify}_{\textit{counterpart}}\left([\Psi, id_S]\right)$
    $ID_{tethered} := \Psi\left(id_{counterpart}\right)\left(\text{“}tethered\text{”}\right)$
    $\alpha := \textit{loop}\left(0\right)$
        $\textit{loop}\left(i\right) := \left(\begin{array}{ll} i = \#ID_{tethered} & \rightarrow \top \\ i < \#ID_{tethered} & \rightarrow \textit{check} \end{array}\right)$
            $id_{tethered} := ID_{tethered}\left(i\right)$
            $\textit{check} := \left(\begin{array}{ll} id_{tethered} \equiv id_V & \rightarrow \perp \\ id_{tethered} \not\equiv id_V & \rightarrow \textit{loop}\left(i+1\right) \end{array}\right)$
```

We now define $attach$, which results in the boolean α. If $\alpha = \top$, the two untethered colliding particles become tethered. Otherwise, $\alpha = \perp$, and the particles remain untethered. Note there are only two pairs of particles species that can lead to a particle attachment, V and S particles, and V and D particles. In the former case, α_{VS} is evaluated. A truthful result requires that the synapsin particle is free of vesicles ($\textit{free}_{\psi}\left(\Psi, id_S, \text{“}V\text{”}\right)$), and that the synapsin particle's counterpart is free of the particular vesicle involved in the collision ($\textit{free}_{\textit{counterpart}}\left(\Psi, id_S, id_V\right)$). In the case of V and D particles, α_{VD} is evaluated. The particles become tethered only if the docking site is free of vesicles, ($\textit{free}_{\psi}\left(\Psi, id_D, \text{“}V\text{”}\right)$) and the vesicle is free of docking sites (*free*$_\psi(\Psi, id_V, \text{“}D\text{”})$).

$$
\begin{aligned}
&\mathit{attach}\left([id_A, id_B, \Psi, \Phi, \Omega_\psi, \Omega_{\psi\psi}]\right) := \alpha\\
&\quad spc_A := \Psi\left(id_A\right)\left(\text{“}spc\text{”}\right)\\
&\quad spc_B := \Psi\left(id_B\right)\left(\text{“}spc\text{”}\right)\\
&\quad \mathbf{\mathit{VS}} := \left(\left(spc_A \equiv \text{“}V\text{”}\right) \wedge \left(spc_B \equiv \text{“}S\text{”}\right)\right) \vee \ldots\\
&\qquad\qquad \left(\left(spc_A \equiv \text{“}S\text{”}\right) \wedge \left(spc_B \equiv \text{“}V\text{”}\right)\right)\\
&\quad \mathbf{\mathit{VD}} := \left(\left(spc_A \equiv \text{“}V\text{”}\right) \wedge \left(spc_B \equiv \text{“}D\text{”}\right)\right) \vee \ldots\\
&\qquad\qquad \left(\left(spc_A \equiv \text{“}D\text{”}\right) \wedge \left(spc_B \equiv \text{“}V\text{”}\right)\right)\\
&\quad \alpha := \begin{pmatrix} \mathbf{\mathit{VS}} & \rightarrow \alpha_{\mathbf{\mathit{VS}}}\\ \mathbf{\mathit{VD}} & \rightarrow \alpha_{\mathbf{\mathit{VD}}}\\ -\mathbf{\mathit{VS}} \wedge -\mathbf{\mathit{VD}} & \rightarrow \bot \end{pmatrix}\\
&\qquad \alpha_{\mathbf{\mathit{VS}}} := \mathbf{\mathit{free}}_\psi\left(\Psi, id_S, \text{“}V\text{”}\right) \wedge \mathbf{\mathit{free}}_{counterpart}\left(\Psi, id_S, id_V\right)\\
&\qquad\quad [id_V, id_S] := \begin{pmatrix} spc_A \equiv \text{“}V\text{”} & \rightarrow [spc_A, spc_B]\\ spc_B \equiv \text{“}V\text{”} & \rightarrow [spc_B, spc_A] \end{pmatrix}\\
&\qquad \alpha_{\mathbf{\mathit{VD}}} := \mathbf{\mathit{free}}_\psi\left(\Psi, id_D, \text{“}V\text{”}\right) \wedge \mathbf{\mathit{free}}_\psi\left(\Psi, id_V, \text{“}D\text{”}\right)\\
&\qquad\quad [id_V, id_D] := \begin{pmatrix} spc_A \equiv \text{“}V\text{”} & \rightarrow [spc_A, spc_B]\\ spc_B \equiv \text{“}V\text{”} & \rightarrow [spc_B, spc_A] \end{pmatrix}
\end{aligned}
$$

For the results of Section 5.1, tethered particles need never detach. The detach function therefore yields $\bot$.

$$\mathit{detach}\left([id_A, id_B, \Psi, \Phi, \Omega_\psi, \Omega_{\psi\psi}]\right) := \bot$$

Both the attach and detach functions become more complicated if we are to model action potentials and exocytosis, as is done in the simulations of Section 5.3.

We choose $\theta_{revolve}$ such that a complete revolution of two tethered particles is resolved in 32 distinct collisions. We choose some very long duration for Δt_{max}, and some relatively short duration for $\Delta t_{restitute}$.

$$
\begin{aligned}
\theta_{revolve} &= \frac{\pi}{16}\\
\Delta t_{max} &= 2^{30}\\
\Delta t_{restitute} &= \frac{1}{256}
\end{aligned}
$$