

A modeling and simulation language for biological cells with coupled mechanical and chemical processes

Endre Somogyi¹ and James A. Glazier²

¹Department of Computer Science, Indiana University

²Department of Intelligent Systems Engineering, Indiana University

January 3, 2017

Abstract

Biological cells are the preeminent active matter. Cells sense and respond to mechanical, chemical or electrical environmental stimuli with a range of behaviors, including dynamic changes in morphology and mechanical properties, changes in chemical uptake or secretion, cell differentiation, proliferation, death, or migration.

Modeling and simulation of such dynamic phenomena poses a number of computational challenges. A modeling language must first be able to naturally represent both the complex intra and extra-cellular spatial structures, and their coupled dynamics of mechanical, chemical and electrical objects and processes. In order to be useful to domain experts, a modeling language should be based on mechanical and chemical constructs native to the problem domain. A compiler must then be able to generate an executable model from this physically motivated description. Finally, an executable model must efficiently calculate the time evolution of such dynamic and inhomogeneous phenomena.

We present a spatial hybrid systems modeling language, compiler and mesh-free Lagrangian based simulation engine which will enable domain experts to define models using natural, biologically motivated constructs and to simulate time evolution of coupled cellular, mechanical and chemical processes acting on a time varying number of cells and their environment.

Keywords: Biological Systems Modeling, Simulation, Spatial Hybrid Systems

1 Introduction

All living things today are the result of a complex interplay between chemical, mechanical, electrical and other physical processes [4]. Biological cells exist in a dynamic, spatial fluid environment where they sense a range of stimuli such as physical forces, chemical and electrical signals, fluid pressure and flow, or other physical properties of their environment. Cells can react with a range of

responses including: changes in morphology or mechanical properties, proliferation, death, differentiation, motion or production of signals.

A mechanistic model of natural phenomena seeks to represent a complex system by proposing a set of mechanisms that replicate observed behavior of the system in response to stimuli. Models are not intended to replicate physical reality in exacting detail, but rather to present a simplified mechanism in order to better understand the observed behavior. Mechanistic models help us understand how something works and how to predict its behavior. Simulations of mechanistic models enable researchers to test and validate hypotheses. Simulations are often faster than real time, especially when compared to the time needed for wet-lab testing. Simulations are generally cheaper, safer and sometimes more ethical than conducting real-world experiments.

Empirical scientists can have difficulty developing executable mechanistic models of observed biological phenomena because their models are typically written in programming languages that do not easily describe physical concepts. Most programming languages are either designed as abstractions of the underlying Von Neumann hardware architecture, or are meant to represent models of computation (e.g., λ calculus). These languages are not designed to represent or model natural, physical phenomena.

In order to enable physical scientists to easily create mechanistic models, we are developing a new programming language and simulation environment, *Mechanica*. Our motivating problem domain is molecular and cellular biology; however the environment we are developing can, we expect, be used to model many different kinds of natural phenomena which couple chemical and mechanical processes.

Our problem domain can be described in terms of objects and processes. Objects are the physical things being studied. Here, we use the term ‘objects’ to refer to empirically observed things in our problem domain. Process refers to a ‘mover’ in the Aristotelian sense. Objects are state-full things that exist as snapshots in time, objects themselves are time-invariant – they do not have any intrinsic dynamics. Processes act on objects to cause them to change over time, they define the dynamics and interactions of and between objects. These objects and processes work together to define how a system evolves over time, i.e., they work together to produce a dynamical model. Some examples of this object/process formalism applied to biological phenomena are 1) chemical kinetics: molecules are defined as objects, reactions are defined as processes. A process may consume multiple substrates and produce multiple products; 2) cellular mitosis: the mitosis process consumes a cell and produces two cells; 3) cellular chemical uptake: the process acts on a cell type, consumes external chemical quantities and produces internal chemical quantities.

Natural phenomena, particularly those found in molecular and cellular biology have always posed challenges for formal description and modeling. The following list illustrates the complexities and dynamism of our problem domain which are not adequately addressed via current modeling and programming languages.

1. Our problem domain, molecular and cellular biology, defines a spatial fluid environment. Objects in this universe have a definite location and orientation, and occupy a specific region of space. We can partition this universe into multiple, possibly overlapping, spatial regions, and each region in turn can be partitioned into multiple child regions. Physical objects in this universe may also contain multiple child objects.

2. Material objects may be homogeneous or heterogeneous. Spatial objects are comprised of an interior, exterior, and boundary. Spatial objects also define a local environment, and contained objects can interact with other objects in this environment. For example, the interior of biological cells is incredibly heterogeneous. The cell membrane separates regions internal and external to the cell, and encloses a spatially varying gel-like substance called the cytosol, where many of the cell's internal chemical processes occur. A cell contains numerous objects such as organelles, nuclei, etc., and the cytosol is the local environment for these objects. The cell has a highly dynamic fiber network called the cytoskeleton which gives rise to the mechanical properties of the cell.
3. Spatial objects may move in space, expand, contract, change shape. Material objects have a set of constitutive relations which determine how the object and any contained material objects respond when an external force is applied to the object. Material objects can undergo active or passive morphological changes. Material objects may be solid, rigid, flexible, liquid, or even gas. Objects can gradually transition between states, e.g., solid objects can disassociate into their constituent elements. (For example, a melt transitions an object from solid to liquid.)
4. Spatial objects have a wide range of length scales. A single model frequently contains objects with several orders of magnitude difference in length scale. Objects from nanometer-sized atoms in solution interact with 10 micron-sized cell membranes. For example, 10 nm sized actin monomers assemble to form dimers, trimers, oligomers, eventually to form fibers spanning the entire cell length. Solvents (usually liquids or gels) can transport small dissolved molecules (solutes). Other kinds of objects such as vesicles also frequently transport these solutes.
5. Objects may be created, destroyed, or transformed from one state or type to another. They may combine to form complexes with very different properties (e.g. hydrogen and oxygen are both gases, whilst water is a liquid).
6. Objects can have multiple attributes or states. Macromolecules frequently have multiple phosphorylation sites, binding sites, and protonation states. Objects such as enzymes or channels may be active or inactive. These objects behave very differently based on their state. Interactions with other objects can cause the state of objects to transition from one state to another. For example, an enzyme may phosphorylate a site on a macromolecule, or a signaling molecule may bind with a channel, activating or deactivating it.
7. Biological processes behave very differently depending on their local environment, totally unlike a well-engineered solution where one can unplug one component and replace it with another.
8. Multiple processes and multiple forces can act concurrently on objects. Multiple reactions within an object may concurrently produce and consume a chemical.

2 Related Work

Nature operates in continuous time, whereas most programming languages do not have a true concept of time. Most mainstream programming languages are well suited for describing computation, but less so for describing a physical model. A language for naturally describing physically motivated mechanisms needs at least three basic capabilities: a way to describe continuous time transformations via equations, an intrinsic concept of space and a way to represent material forces and deformations.

The vast majority of mainstream programming languages do not have a concept of continuous time transformations, rather they specify either computation or state transition via a sequence of discrete instructions. Users can of course develop models using mainstream languages as they are all Turing Complete. However, this forces users to think about low-level computational implementations of mechanisms rather than the mechanisms themselves.

A number of equation oriented modeling languages such as Modelica [12], The Math Works' Simulink or National Instrument's LabVIEW enable model design via continuous time mathematical relationships. These languages are frequently used in application domains such as electrical circuits, multi-body systems, drive trains, hydraulics, and chemical processes – any domain that can be modeled by a set of differential-algebraic equations. These lack an intrinsic notion of space, and have difficulty describing dynamic connectivity or a dynamic number of objects.

With certain exceptions, most programming languages do not have an intrinsic concept of spatiality [1]. However the 3π [5] language does treat dynamic spatial arrangements and also has a concept of force. Molecular dynamics (MD) simulations are intrinsically spatial and based on the core idea of forces. They typically do not support structural rearrangement and are restricted to a finite set of predefined forces. With different force fields, MD can simulate coarse-grained models of polymers [11].

Process Calculi (“ π calculi”) are models of computation for concurrent systems. Process calculus based languages have simple, well defined formal semantics and are used to model individual molecules and individual reactions [14]. 3π -calculus [5] also includes an intrinsic notion of a containing space. All processes in 3π -calculus exist in a three-dimensional spatial domain, with 3D transforms defining their locations relative to their parent processes. π -calculus naturally describes discrete biological processes, but not continuous or spatially extended objects and processes.

Interchange formats such as the Systems Biology Markup Language (SBML) <http://www.sbml.org/> can describe a restricted phenomenon such as chemical reactions in a well-stirred compartment, have no concept of forces or dynamic geometry or structural rearrangement, and are too restrictive for use as general modeling languages. SBML models can be connected together as part of a larger simulation environment such as discrete event simulation [2].

A number of biologically oriented cell and tissue simulators such as MCell [16] can simulate spatial chemical processes such as reaction-diffusion. MCell models are specified with a natural and elegant rule-based approach similar to BioNetGen [7]. However, most cell simulators can only model chemical processes with fixed geometries, most cell simulators do not appear to model mechanical or electrical processes or dynamic morphology. MCell for example is a particle based simulator, but it is restricted to point particles with no volume exclusion.

3 Approach

The *Mechanica* model declaration language enables empirical scientists to build mechanistic models of natural phenomena. Its syntax is based on a simplified JavaScript, one of the most widely used programming languages today. *Mechanica* adopts a number of syntactic concepts from POV-Ray, a widely used spatial scene description language. *Mechanica*'s pattern matching rules incorporate ideas from BioNetGen, OCaml and Mathematica.

Mechanica is based on two basic constructs: objects and processes. The *Mechanica* concepts of objects and processes are intended to reflect the objects and processes found in naturally occurring phenomena. The description of a naturally occurring object maps relatively easily to the traditional computational notion of an object in that both represent a particular state. *Mechanica* objects can represent things like molecules, proteins, cells, or fluids. Processes in *Mechanica* are, however, quite different from processes in the traditional computing sense. A computing process is typically a sequence of instructions that defines a set of state changes in objects. Whilst it is possible to define a *Mechanica* process in such a way, *Mechanica* processes typically define a *continuous* transformation and the rate at which the transformation occurs. A process may, for example, represent a transformation in which substrates are consumed and products are produced, or it may alter the state of an object, either continuously or discretely.

Processes may create and destroy objects, alter the state of objects, transform objects, or consume and produce sets of objects. As in nature, multiple *Mechanica* processes can act concurrently on objects and may act at different rates, but may only be active under certain circumstances. Processes may be continuously active, or may be explicitly triggered by specific conditions or invoked directly by some other process. *Mechanica* supports two basic types of processes: continuous and discrete. Continuous processes, as their name implies, operate in continuous time. A continuous process must specify a *rate* at which the process occurs. Discrete processes are closer to the traditional computer science notion of a process, in that they specify a discrete state change. Discrete processes must prescribe a set of conditions that in turn specify the circumstances under which the processes should be triggered.

Mechanica objects start with a fairly conventional definition of objects and build on this to support the kind of physical objects found in our problem domain. Objects in *Mechanica* are similar to objects in mainstream languages like Java in that objects are effectively a set of named fields and each field can store values. An object is an instance of a *type*. A type defines the object's structure and layout, and a *state space*, or the set of permissible values that can be stored in the object. Objects have an identity, a *symbol* which names and refers to the object. Objects can be primitive data types (e.g., scalars, integers, booleans, enums) or they can be composite structures.

Mechanica represents materials as an interacting collection of point particles. Points typically represent 'parcels' or small pieces of real materials, rather than actual atoms. In this sense, we have a coarse grained representation. This particle-based approach [13] enables *Mechanica* to create a completely unified treatment of all materials. Solids are simply strongly bonded particles, fluids are weakly bonded particles, and *Mechanica* can simulate any range in between. Particle based approaches for material simulation are used every day for models ranging from nano-scale molecular dynamics up to galactic simulations [9], where essentially the only difference between these two extremes is the functional form of the particle interactions. Particle based approaches are like-

wise frequently used simulate melting, freezing and other highly dynamic spatial processes found in cell biology [10]. With Mechanica, users have complete freedom to define whatever particle relationships and interactions they see fit.

Points are fundamental building block of materials, they have a definite position, (possibly zero) spatial extent as well as (optional) orientation, mass and other properties. Complex materials such as fibers or surfaces are collections of points joined by *connectors*. A connector defines an interaction between objects. A connected set of objects forms a *complex*. Connectors are very general, but one of the things that connectors can do is specify the constitutive relations for a set of material points. Connectors can specify that a particular arrangement of points remains fixed such that the points form a rigid bond. Connectors are commonly used to define a force relationship between points, such as a bond, angle or dihedral force. A dimer, for example, can be composed of two monomers as

```
1 a:Point(0,0,0,mass=1); b:Point(1,0,0,mass=1); force(a,b){-k*(1-
    dist(a,b)**2}
```

where a and b are both point particles, and are explicitly connected with a Hookean spring force. The body of the force definition can contain arbitrary user specified code. This yields a very flexible model in that users may specify the interaction force as complex as they like. For example, a two body force could be as simple as a Hookean interaction, or as complex as a dipole, quadrupole or octopole interactions.

Fibers may be represented as a collection of points connected with spring like connectors, where each connector attaches two points. Surfaces are a collection of points connected with Face connectors and each face connects three points. Spatial regions have a definite inside, boundary and outside, and can contain multiple child regions. Materials such as fluids, gels, solids are represented as a set of points that have some form of non-bonded interaction. Fluids are freely moving solvent particles; each solvent particle represents a ‘parcel’ of real fluid. Fluids would typically have a force term which accounts for convection, dissipation and random motion. We provide a basic fluid force, `FluidForce`, which implements the force term from [8]. A modeler might want to fill a region with two different kind of fluids, and have both of these fluids bounded by this region. The modeler first creates a region, defines two fluids in its spatial environment, then defines the interaction between the two fluids as well as between the two fluids and the surface.

```
1 SpatialRegion{
2   surface:Sphere { radius:5, resolution:1 };
3   a:fill(type=Point{diameter=5});
4   b:fill(type=Point{diameter=1});
5   FluidForce(a,a);          FluidForce(b,b);          FluidForce(a,b)
6   ;
7   FluidForce(a,surface); FluidForce(b,surface); Rigid(surface,
8   surface);
9 }
```

Here, the `fill` function generates a set of objects which fill all the available space in a region. The first particle type is larger then the second, so the second `fill` will then fill all of the remaining

space with the smaller particles. The user must specify an interaction between each of the particle types – if no connection is specified, the materials will not interact, and will pass through each other. The local symbols `surface`, `a`, and `b` all refer to collections of points. A two-body term applied to a collection is treated as an all-all force, but the cutoff distance of the function prevents an $\mathcal{O}(N^2)$ problem. The `Rigid` connection specifies that the surface particles are rigidly connected.

A biological cell has on the order of 10^{14} atoms, so it is simply not computationally feasible to model a cell at all-atom resolution. Many smaller ($< 100\text{nm}$) molecules are soluble in and readily diffuse through solution. Since we are interested in micro- rather than nano-scale phenomena, these diffuse chemicals are approximated as continuous, spatially varying concentrations, that is, as chemical concentration fields. `Mechanica` has concentration and amount data types which can be attached to any material point. As each material point moves, the attached values move along with it. Chemical concentrations can be attached to any kind of material type such as fluids (solvents), surfaces (cell membranes) or fibers, and they can also be attached to spatial regions. If a concentration is attached to a region, the value is constant for the entire region. Concentration or amount types can be added to any region or material simply by defining this in the definition. For example, we could add `A` as a concentration, `B` as a constant concentration (boundary value), and `C` as an amount to material as:

```
1 MyMaterial : Point { A:conc; B:const conc(5.0); C:amount(1.23);
  }
```

Chemical reactions are fundamentally important in biology. Reactions define the transformation of a set of reactants into a set of products. `Mechanica` generalizes this concept of reactions into processes which describe the transformation of different types of objects, as well as amount and concentration types. Processes between a set reactants and products, such as set of concentration types in a region, define a reaction network as in Fig. 1. The `Mechanica` compiler reads the process definitions and generates a system of ordinary differential equations (ODEs) which define the time evolution of reactants and products. Process definition syntax is similar to the conventional chemical reaction notation, and is based on Microsoft Typescript type specification. For example, to add a chemical reaction network to a spatial region, one would simply write a set of transformation processes in the spatial region’s code block. Processes’ definitions begin with the `proc` keyword, have an optional name, a set of reactants, a set of products and the reaction rate expression as in Fig. 1. If the spatial region does not already have local definitions corresponding to product names from a transformation process definition, then the process definition implicitly defines concentration types in this region for these names. We do this as a user convenience since chemical reaction networks are among the most common modeling tasks.

Processes can be used to define transformations between values located in different spatial regions, such as a flux between two nearby material points. Thus, transformation processes in `Mechanica` can be used to define general reaction-transport problems. Spatial process definitions are written and behave identically to processes confined to one region. The spatial information is simply added in the argument definitions. For example, to define a Fickian (passive diffusion) flux of a concentration type between material points we would write:

```
1 proc (a:MyMaterial.A) -> (b:MyMaterial.A) when (dist(a,b) < 5) {
  k * (a - b)};
```

```

type MyCell : SpatialRegion {
  surface : Sphere {
    radius:5, resolution:1
  };
  proc (A) -> (X) {k1 * A};
  proc (X, 2 Y) -> (3 X) {
    k2 * X * Y**2
  };
  proc (B, X) -> (Y, D) {
    k3 * B * X
  };
  proc (X) -> (E) {k4 * X};
}

```

$$\begin{bmatrix} dA/dt \\ dB/dt \\ dD/dt \\ dE/dt \\ dX/dt \\ dY/dt \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & -1 & -1 \\ 0 & -2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

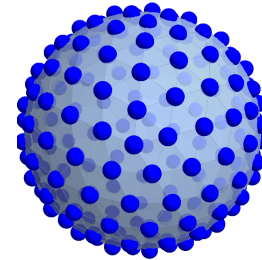


Figure 1: Mechanics source code for creating a SpatialRegion derived type which is bounded by a spherical surface composed of a set of points and triangular faces. This type defines four local transformation processes. A set of continuous processes define a transformation network.

Diffusion represented as Fickian flux of continuous valued concentrations between material points is the same approach that [8] used to model advection-diffusion-reaction, $dC_i/dt = \nabla(D_i \nabla C_i) + Q_i^s$, $i = 1, \dots, N$. They validated their approach against a variety of analytical solutions and demonstrated augmenting a particle fluid model with scalar values at each material site incurs negligible computational overhead.

Processes in Mechanics are very general, being used to define not only transformations of continuous valued objects like amounts and concentrations, but also discrete objects such as particles, materials, and other composite objects. Processes acting on discrete objects have the identical syntax as processes acting on continuous objects, the only difference is the semantic interpretation of the process body. A discrete process body is expected to return a probability value, a real value between zero and one, as opposed to a reaction rate for continuous processes. For example, say we want to consume two discrete particles of type A, and produce one particle of type B. We may also write a random decay process, where a single object of type B randomly splits into two objects or type A.

```

1 type A:Point{radius:1}; type B:Point{radius:2};
2 proc (a:A, b:A) -> (B) when (dist(a,b) < 5) {exp(-dist(a,b)**2)
  };
3 proc (a:B) -> (A, A) {rand()};

```

Processes can also be used to change the state of objects, i.e., to change the value stored in one or more fields. Process can be defined so that the reactants have to be in a certain state, so that they match a particular *pattern*. We borrow the record pattern matching syntax from OCaml/F# to specify the state of an object where a process can be applied. For example, say an object of type A has a field named `state` which is an enum, and we want to toggle this from `Inactive` to `Active` when this molecule is near a particle of type `Activator`:

```

1 proc (a:A{state=Inactive}, b:Activator) -> (a{with state=Active
  }, b)

```



```
2   when (dist(a,b) < 5) {1};
```

The `with` keyword specifies that the fields in the `with` expression should take on new values, but all other fields should remain unaltered. The second argument, `b` of type `Activator`, is unaffected and passes straight through. The body of this process again specifies application probability, and a `1` here tells the compiler that this process should always be applied whenever these two particles are within a distance of `5`.

Processes can also be used to dynamically form and dissolve bonded relationships. *Mechanica* uses the `site` data type to represent binding sites; any spatial object can have any number of binding sites. We may for example add two binding sites to a point type, and use a process to bind two objects together dynamically when certain circumstances are met.

```
1  type A:Point{s1:site(empty); s2:site(empty)};
2  proc (a:A{s1=empty}, b:A{s2=empty}) -> (force(a.s1,b.s2){-k*(1-
    dist(a-b)**2)})
3  when (dist(a,b) < 5) {1};
```

This process binds the `s1` binding site of `a` to the `s2` binding site of `b`. When `force` or some other connector attaches to a binding site, the *Mechanica* semantics state that that binding site should become non-empty.

Most traditional languages employ symbol scoping blocks. These are regions of code that define where a symbol (variable definition) is visible. Scoping blocks can contain other scoping blocks, and depending on the language, code in the child blocks can usually access variables defined in parent blocks. Lexically scoped languages resolve non-local symbols to the scoping block in which the function was defined. Dynamically scoped languages such as `PERL` resolve non-local symbols to the scoping block in which the function is executing.

In order to account for the spatial nature of our problem domain, we introduce a new kind of scope resolution we call *spatial scoping*. Spatial scoping is based on the idea that processes represent a physical process, so processes always have to execute somewhere in space. The present location of a processes is implicit, much like the `this` pointer is implicit in `Java`. Non-local symbols in process bodies resolve to the present spatial environment in which the process currently runs. Spatial scoping is similar to dynamic scoping in that symbol resolution depends on the calling context of a function, but in spatial scoping, that calling context is a containing spatial region.

Users typically never have to explicitly keep track of coordinate systems; instead the runtime manages all this. All users have to know is the name of the value that they want to read. Spatial scoping treats different kinds of spatial objects uniformly. For example, consider a transformation process which converts one substance to another, where the rate (the body of the transformation process definition) depends on the concentration of some other substance,

```
1  proc (A) -> (B) {k1 * A * C};
```

Here, `C` is a non-local symbol, so the runtime will search in the containing spatial environment. This process for example could be attached to a fluid particle, like a solvent or cytoplasm particle. This particle type may have a `C` attribute attached to it, in which case `C` resolves to that value. If the particle does not have such an attribute, then the compiler will check if the containing spatial

environment defines the C symbol. The containing spatial region may define the C as a scalar, in which case the value of C is uniform for the entire region, and this symbol resolves to that value. Or C may be defined as a spatially varying field, in which case the compiler will generate code that reads from that field at the present location of the particle. If the containing environment does not contain a C symbol, then the compiler will check if a scalar field 3 defines this symbol. If so, then the symbol resolves to the field value at the location of the current particle. If this symbol cannot be resolved in any containing environment, and there is no field definition, then the compiler will report an error.

Mathematical constructs such as fields and vectors exist in the language of mathematics and are not tied to any physical concept, but are a convenient way to represent many physical concepts. Many physically motivated processes frequently use *fields* in their descriptions. A field is a mathematical function which associates a value with a spatial location.

A field can also be defined as a mapping between a value at a spatial location and a set of values associated with material points. In general, any continuously valued field, $A(\mathbf{r})$ can be approximated as a sum over a set of appropriate *kernel* functions, $A(\mathbf{r}) = \sum_i f(A_i, \mathbf{r}, \mathbf{r}_i)$, where A_i is some quantity located at the source location \mathbf{r}_i , and \mathbf{r} is the field location. This definition enables us to define any continuous field as a function of the material. For example the kernel function for an electric scalar potential would be $q_i/4\pi\epsilon_0|\mathbf{r}-\mathbf{r}_i|$. In general, if the value of some scalar quantity A is known at finite set of material points, then value may be approximated at any location in space $A(\mathbf{r})$ with a suitable an interpolation function [9]. This allows us to attach a chemical concentration to a material point and evaluate that concentration at any point in space. For example, one might define a chemical compound attached to solvent points and read the chemical concentration value at some non-solvent material point. We provide default kernels for chemical concentrations and charge, but users are free to define their own kernels.

3.1 Mechanics model of chemotaxis

Chemotaxis is a biological processes which describes the motion of a cell towards or away from a chemical signal. Chemotaxis is only one biological example which illustrates the coupling of chemical and mechanical processes. We use chemotaxis as an example to demonstrate how a few mechanisms of the chemotaxis processes can be represented in *Mechanica*. There are many different chemotaxis models, for an excellent review of our current knowledge of chemotaxis, we suggest [6] and [3]. The model we develop here is simplistic and incomplete, and is not intended to be a treatise on chemotaxis modeling but rather to demonstrate mechanistic model description capabilities of the *Mechanica* language.

Cells undergoing chemotaxis can sense spatial gradients in chemoattractant concentrations. Chemoattractants are frequently secreted from some source, such as a bacterium at a specific location. Here, they are secreted from a fixed point source in space. To start our model, we first need to define the environment. We use an explicit solvent model, so that the solvent will automatically be pushed out of the way as other material objects pass through it. We write a rule in the top level spatial region which fills it with an explicit solvent. This rule basically states that any space that is not filled with sub-objects gets filled with solvent. This rule creates a symbol called ‘solvent’ which is of type water. CXC is a common chemoattractant, our model will secrete CXC from a point source

located at (20,20,0). Because the CXC symbol is listed as a product, the rule implicitly adds the symbol CXC as a chemical concentration attached to the solvent.

At this point, the solvent will transport the CXC along with it, but we allow want to allow the CXC to diffuse through the solvent, we can use the built-in diffusion rule, or we can write our own. Say we want simple Fickian diffusion between solvent particles.

```
1 solvent:fill(type=Water);
2 proc (empty(20,20,0)) -> (solvent.CXC) {aRateConstant};
3 proc (a:solvent.CXC) -> (b:solvent.CXC) when (dist(a,b) < 5) {k
  * (a - b)};
```

Now that the environment is defined, we are ready to define a basic cell model. We only need a single basic cell object that sticks to a horizontal surface. We can define a basic spatial region derived type as in Fig. 1, and we can place a single instance of it at the origin via

```
1 mycell:MyCell{origin:(0,0,2.5)};
```

We know from observation that a cell likes to stick to the surface that it's on. We can represent that with a force that causes the surface of the cell to stick to the base surface, say the Y-Z plane. We know that stickiness tends to be asymmetric, so we can write the force rule to activate at a shorter distance than when it deactivates. We can think of it as similar to a Velcro adhesion, where the two components must be close to activate, but the force can hold on a bit longer when its stretched. In addition to the when clause, forces can also have a while clause which lets users write a deactivation condition that differs from the activation condition.

```
1 force(a:mycell.surface,b:floor) when (dist(a,b) < .5) while (
  dist(a,b) < 2)
2 {-k*(dist(a,b))**2}
```

The plasma membrane of biological cells is very flexible, with little structural support. However, it is connected to an underlying cortical actin network which gives the membrane a certain stiffness and elasticity. Mechanics represents surfaces as a collection of points connected via triangular faces. Much like molecular dynamics packages provide a repertoire of force types to define bonded relationships such as springs, angles, dihedrals and torsions, Mechanics also provides a similar set of force types. Stiffness and elasticity can be represented with the appropriately named `Elasticity` and `Stiffness` forces. Stiffness is similar to an angle force, except that each point in surface connects with three other points, and elasticity is a measure of how much each face can stretch.

```
1 Elasticity(mycell.surface, 0.1);
2 Stiffness(mycell.surface, 0.1, 0.1);
```

The cytoplasm of a real cell is highly heterogeneous. However, for this simple model we will use a very basic cytoplasm model where we approximate the cytoplasm as water. We can fill the body of the cell with water by adding the following line to the cell definition.

```
1 mycell.body:fill(type=Water);
```

A biological cell's surface often contains many chemoattractant receptors. Essentially these bind with a chemoattractant molecule, often degrade the chemoattractant, and initiate a spatially varying signaling cascade internal the cell. The receptor is a trans-membrane protein. We observe that the receptor can be either active or inactive. Inactive receptors can bind with a chemoattractant, doing so, they become active and degrade the chemoattractant. Only active receptors initiate signaling cascade. We can model these behaviors with the following rules

```

1 mycell.surface.Recpt:conc({state=Active|Inactive}, 12);
2 proc (a:solvent.CXC) -> (b:mycell.surface.CXC) when (dist(a,b) <
   5) {k * (a - b)};
3 proc (a:CXC, b:Recept{state=Inactive}) -> (b{with state=Active})
   {k2 * a * b};

```

The first rule here states that `mycell.body.Recpt` should be a concentration type, but this type may have a state that is either Active or Inactive. We again borrow the Microsoft TypeScript type specification syntax to define enumerated types. Because this is a concentration, the Mechanica compiler actually allocates space for two separate chemical species, one for the active, another for the inactive states. However, users simply refer to these using the usual Mechanica pattern matching syntax. The second rule is a spatial flux CXC from the solvent to the cell surface, and the third rule applies to each cell surface point and consumes CXC and produces activated surface receptors.

Signaling cascades inside the cell usually involve many stages, but here we write an very crude approximation that an internal compound, say Rho (known to induce actin polymerization) can be activated by an active surface receptor, Recept. Many models of chemotaxis propose that actin filament polymerization in the leading edge of a migrating cell is principally responsible for lamellipodia extension. We can approximate this mechanism with a set of spring like forces that push between the cell nucleus and the cell surface membrane. In the presence of an active Rho, the springs will extend, and otherwise have no effect. We model Rho as a cytosol diffusive compound which decays from the active to inactive states.

```

1 mycell.body.Rho:conc({state=Active|Inactive}, 1.5);
2 proc (a:Rho({state=Inactive}, b:Recept{state=Active}) ->
3   (a:Rho({with state=Active}, b:Recept{with state=Inactive}) {k
   * a * b};
4 proc (a:Rho({state=Active}) -> (a:Rho({with state=Inactive}) {k
   * a};
5 force(a:mycell.nucleus, b:mycell.surface) {
6   -k * (dist(a,b)**2 - restLength + k2 * Rho{state=Active});
7 }

```

Here, symbol, `Rho{state=Active}` is read at the location of the spring force center. This symbol is defined as chemical concentration attached to the cytosol, the compiler generates code which interpolates the value of Rho at the spring location. A high concentration of Rho effectively causes the spring to extend, thus pushing on the cell surface.

3.2 Compiler Details

The front end of the Mechanics compiler is a relatively simple recursive descent parser, written in OCaml, and based on a simplified Microsoft Typescript syntax. The semantic analysis phase of the compiler, however, has to do significantly more work than a conventional compiler. Unlike conventional procedural languages where there is a reasonably close mapping between each syntax statement and the resulting machine instructions, Mechanics statements specify rules and relationships rather than explicit imperative instructions. The Mechanics compiler however must analyze all of the model processes and rules and use this information to essentially generate two sets of equations:

$$\frac{d^2\mathbf{r}_i}{dt^2} = \frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i}\mathbf{F}_i = \frac{1}{m_i} \left(\sum_{i \neq j} (\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R) + \mathbf{F}_i^{ext} \right), \quad (1)$$

$$\frac{dA_i}{dt} = Q_i = \sum_{i \neq j} (Q_{ij}^D + Q_{ij}^R) + Q_i^S. \quad (2)$$

One set of equations, (1) defines the spatial time evolution of the material points themselves, and the other set, (2), defines the time evolution of amounts located at each material point. The C , D , and R superscripts represent conservative, dissipative and random terms. This approach is similar to [9].

The Mechanics compiler performs semantic analysis of the continuous process definitions using techniques we established in [15]. Semantic analysis of the force functions is an extension of these techniques. Both systems of equations are compiled to C code which is then linked with a modified version of Pedro Gonnet's mdcore library <http://mdcore.sourceforge.net/>. The Mechanics runtime monitors the values of the when clauses (if present) and activates or deactivates the corresponding processes and forces accordingly. If a when clause contains a cutoff distance, the runtime uses this information to create the Verlet and cell lists to ensure optimal performance.

Unlike traditional molecular dynamics or particle systems, the particle count in Mechanics models changes over time. The runtime initially allocates a large block, more than the initial particle count, and as particle count increases, it performs a realloc on that block. We are investigating more efficient approaches for dealing with this challenge.

4 Conclusion

We are developing the first known modeling language which can represent generalized continuous transformation processes, forces, fluid flow, and material state transition in the same language. The Mechanics language enables users to create physically motivated models of complex natural phenomena using mechanistic instead of computational building blocks.

Acknowledgements

We acknowledge generous financial support the National Institutes of Health, National Institute of General Medical Sciences, National Institute of Environmental Health Sciences and National Institute of Biomedical Imaging and Bioengineering, grants U01 GM111243, R01 GM076692 and R01 GM077138. We thank Dr. Marie Gingras, Dr. Amr Sabry and Dr. James Sluka for their discussion and insights.

References

- [1] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the Aggregate: Languages for Spatial Computing. In *Formal and Practical Aspects of Domain-specific Languages Recent Developments*, pages 436–501. Information Science Reference, Hershey, PA, 2013.
- [2] L Belloli, G Wainer, and R Najmanovich. Parsing and model generation for biological processes. In *Proceedings of the Symposium on Theory of Modeling Simulation*. ACM, 2016.
- [3] Laurent Blanchoin, Rajaa Boujemaa-Paterski, Cécile Sykes, and Julie Plastino. Actin Dynamics, Architecture, and Mechanics in Cell Motility. *Physiological reviews*, 94(1):235–263, January 2014.
- [4] G W Brodland. How computational models can help unlock biological systems. *Seminars in Cell & Developmental Biology*, 47-48:62–73, 2015.
- [5] L Cardelli and P Gardner. Processes in space. In *Conference on Computability in Europe*, pages 78–87, Berlin Heidelberg, 2010. Springer, Programs.
- [6] Gaudenz Danuser, Jun Allard, and Alex Mogilner. Mathematical Modeling of Eukaryotic Cell Migration: Insights Beyond Experiments. *Annual Review of Cell and Developmental Biology*, 29(1):501–528, 2013.
- [7] James R Faeder, Michael L Blinov, and William S Hlavacek. Rule-Based Modeling of Biochemical Systems with BioNetGen. In *Systems Biology*, pages 113–167. Humana Press, Totowa, NJ, January 2009.
- [8] Zhen Li, Alireza Yazdani, Alexandre Tartakovsky, and George Em Karniadakis. Transport dissipative particle dynamics model for mesoscopic advection-diffusion-reaction problems. *Journal of Chemical Physics*, 143(1):014101, July 2015.
- [9] M B Liu and G R Liu. Smoothed Particle Hydrodynamics (SPH): an Overview and Recent Developments. *Archives of computational methods in engineering*, 17(1):25–76, 2010.
- [10] M B Liu, G R Liu, L W Zhou, and J Z Chang. Dissipative particle dynamics (DPD): an overview and recent developments. *Archives of Computational Methods ...*, 22:529–556, 2015.

- [11] P Malfreyt and D J Tildesley. Dissipative Particle Dynamics Simulations of Grafted Polymer Chains between Two Walls. *Langmuir*, 16(10):4732–4740, May 2000.
- [12] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6:501–510, April 1998.
- [13] Matthias Müller, David Charypar, and Markus H Gross. Particle-based fluid simulation for interactive applications. In *2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003.
- [14] A Phillips and L Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *International Conference on Computational Methods in Systems Biology*, pages 184–199. Springer, 2007.
- [15] Endre T Somogyi. *Simulation of electrochemical and stochastic systems using just in time compiled declarative languages*. PhD thesis, Indiana University, Bloomington, December 2014.
- [16] Joel R Stiles, Thomas M Bartol, et al. *Monte Carlo methods for simulating realistic synaptic microphysiology using MCell*. CRC Press, Boca Raton, FL, 2001.