Published in final edited form as:

Proc Winter Simul Conf. 2016 December ; 2016: 1230–1240. doi:10.1109/WSC.2016.7822179.

TOWARDS A MULTI-SCALE AGENT-BASED PROGRAMMING LANGUAGE METHODOLOGY

Endre Somogyi,
Dept. of Computer Science, Biocomplexity Institute, Indiana University, Bloomington, IN 47405, USA

Amit Hagar, and
Dept. of History & Philosophy of Science & Medicine, Biocomplexity Institute, Indiana University, Bloomington, IN 47405, USA

James A. Glazier
Dept. of Intelligent Systems Engineering, Biocomplexity Institute, Indiana University, Bloomington, IN 47405, USA

Abstract

Living tissues are dynamic, heterogeneous compositions of *objects*, including molecules, cells and extra-cellular materials, which interact via chemical, mechanical and electrical *process* and reorganize via transformation, birth, death and migration *processes*.

Current programming language have difficulty describing the dynamics of tissues because: 1: Dynamic sets of objects participate simultaneously in multiple processes, 2: Processes may be either continuous or discrete, and their activity may be conditional, 3: Objects and processes form complex, heterogeneous relationships and structures, 4: Objects and processes may be hierarchically composed, 5: Processes may create, destroy and transform objects and processes. Some modeling languages support these concepts, but most cannot translate models into executable simulations.

We present a new hybrid *executable modeling language* paradigm, the Continuous Concurrent Object Process Methodology (*CCOPM*) which naturally expresses tissue models, enabling users to visually create agent-based models of tissues, and also allows computer simulation of these models.

1 INTRODUCTION

Living tissues result from the complex interplay among a wide variety of *objects*, including molecules, macromolecules, organelles, cells, extra-cellular matrix, fluids and tissues participating in spatial, mechanical and chemical *processes* at multiple scales (Brodland 2015, Gilbert 2014, Alberts, Bray, Hopkin, Johnson, Lewis, Raff, Roberts, and Walter 2013). The disruption of *any* of these objects or processes can severely impair tissue development or function.

In nature, changing sets of objects can participate concurrently in multiple processes. Process activity and object participation may both be conditional. Processes may be active

continuously, trigger explicitly under specific conditions, or may invoke other processes. Processes may also combine or aggregate to form composite processes out of child processes, and child-process ordering may be either concurrent or sequential. For example, in a simplified description of the respiration process in a cell object, glucose transporter objects (GLUTx) transport glucose (GLU) across the cell membrane. The glycolysis pathway consumes GLU, producing a pair of high energy ATP molecules. The GLUTx also consumes ATP to function. The transport process moves GLU into the cell at the same time that the glycolysis process consumes it. As another example, multiple sub-processes compose the complex continuous process of cell growth, whose net effect is to consume extra-cellular chemical objects, increase the volume of the cell object and produce waste product objects. Mitosis is a complex discrete process in which the cell object prepares for and completes cell division. Specific conditions trigger mitosis and the mitosis process transforms the parent cell and creates a new daughter cell. Cell adhesion is a continuous process which causes cells to adhere to cells of specific cell types under specific circumstances.

The ensemble of biological objects and processes in a tissue form a biological description, or *biological model*. Our terms – “consumes”, “produces”, “rate” – suggest that we can naturally describe many biological processes using equations which specify variables (*properties*) representing the *states* of biological objects, the relationships between variables and how variables change over time. For example, a rate equation can describe the flux of a chemical compound through a cell membrane. Reaction-kinetic equations can describe reacting chemical compounds. Force equations can describe adhesion or movement processes. Algorithmic descriptions are more natural for other kinds of processes. For example, condition rules can trigger a mitosis process. The sequential invocation of a series of sub-processes to replicate DNA, separate DNA, and divide the parent cell into two cells, can model biological mitosis. Both mathematical and algorithmic descriptions may be appropriate for different kinds of processes. A biological model plus mathematical or algorithmic descriptions of all processes in the model forms a *quantitative model*.

Established compiler design techniques can readily convert algorithmic processes into executable code. However, in order to translate a set of a mathematical process descriptions into an executable simulation, a compiler must analyze the equations comprising the mathematical processes and determine how to solve for their state variables. Mathematical descriptions do not directly determine their solution algorithms, so many different solution algorithms may correspond to the same mathematical description (Fisher and Henzinger 2007).

Agent-based physical simulations of tissues enable researchers to better understand how tissue organization and function arise out of the constituent objects and processes, which occur across a range of length and time scales. Building such models requires creating and assembling many object and process components [REF]. For decades, engineers have used Computer Aided Engineering (*CAE*) simulations to facilitate design, development and testing of complex products. CAE programs allow engineers to construct virtual representations (*models*) of proposed component objects, assemble them into composite device objects, and simulate their processes and resulting behaviors in virtual experiments to

evaluate their performance, reliability and failure modes. Virtual experiments allow engineers to test a much wider range of possible component and device object designs under a much wider range of conditions than would be practical if they realized each component object, device object and test condition.

Certain areas in the life sciences have successfully used CAE techniques, such as molecular dynamics and reaction kinetics models in computational drug design, finite element models in prosthetics design and virtual-surgery training. Life science practitioners in these fields use specialized CAE tools in much the same way engineers use packages like SolidWorks. First, they build a model out of mainly pre-existing component objects using a visual model development environment. They then compile and link the model with a simulation engine, define its parameters and initial and boundary conditions, and finally execute it. (Fisher and Henzinger 2007).

Common engineered objects such as hydraulics, electronics, suspensions, airframes, and control systems have simpler model definitions than tissues. The number of objects, their identity, and their connectivity –which objects participate in a specific application of a given process – typically remain fixed for the duration of the simulation. For example, in simulations of automotive suspensions or electronic circuits, the set of model objects remains fixed. Models do not dynamically add or remove wheels in a vehicle's suspension nor do they add or remove circuit elements in an electronic circuit. While engineering simulations of vehicle suspensions, artificial heart valves, and bone replacements include processes which deform component objects, they tend not to involve structural reorganization of objects. Tissue processes, on the other hand, continuously create, destroy, move and re-arrange, reshape and change the identity of objects.

2 REQUIREMENTS

Researchers frequently need to write computer *programs* to *model* or simulate a real-world system. A programmer creates a mental model of the system and writes *source code* in a *programming language* to implement that model. Both humans and computers understand computer programming languages, enabling programmers to direct the computer to carry out a calculation or task. *Modeling*, on the other hand, is the *formalization* of hypotheses concerning a real-world system. A model embodies understanding of that system. A *modeling language* specifically helps formalize a modeler's conceptualization and understanding of a real-world system. Model definition has significant value, because it provides a framework to capture and formalize domain knowledge. A model's components and sub-models form a repository, which supports capture, storage, searching and extraction of domain knowledge.

Computer programming often requires modeling, but programmers rarely formalize the modeling component of program development and most programming languages do not support modeling intrinsically, because they lack tools to capture and represent meaning. On the other hand, many modeling languages do not produce executable programs, because they do not contain sufficient procedural detail to specify an unambiguous sequence of instructions.

An *executable modeling language* both allows representation of the understanding of a system as a model *and* contains sufficient information to allow conversion of the model into executable instructions.

An executable modeling language must be able to: a) define model *components* (objects and processes), b) assemble components to form composite components and models, and c) compile models into *executable models* which a computer can simulate.

To enable general-purpose object-oriented programming languages to support executable modeling of the complex and varied objects and processes in living tissues, the Continuous Concurrent Object Process Methodology (*CCOPM*) adds support for the knowledge representation and capture aspects of modeling. In addition, while most programming languages can define data structures (corresponding to simple model objects) and functions that manipulate these data structures (corresponding to simple model processes), the CCOPM adds intrinsic support for the many object and process types not present as primitives in general-purpose languages.

2.1 Expressive Capacity

A domain-specific modeling language should express concepts in the problem domain in a form natural and intuitive to domain experts. Tissue physiology includes a number of complex concepts not common in engineering, which present challenges for programming language design.

A modeling language must determine how to categorize modeled components. Unfortunately, terms frequently have different meanings in different disciplines. Biology uses the term *type* fluidly. The term *genotype* refers to inheritable information, specifically an organism's set of genes. Genotype can refer to the entire genome, or to specific variants of specific genes (*alleles*). A *phenotype* is a category for which a set of actual observed properties or characteristics, such as morphology, development, or behavior, define membership. Phenotypes definitions are non-exclusive and ambiguous, and objects often interpolate between or simultaneously participate in multiple phenotypes. We often ask the degree to which a cell corresponds to a particular phenotype, for example, cells frequently and gradually transition between mesenchymal and epithelial phenotypes. In our discussion, we abbreviate *phenotype* to *type* except when we need to distinguish genotype from phenotype.

The *type* of an object in a programming language must be unambiguous, since the type assigns *meaning* to a block of memory. A type formally defines the layout of a memory block and the types of data it stores. *Typed* programming languages associate a type definition with every named language construct (object, processes, functions, variables, expressions, etc.). A *type system* is a collection of rules that define the set of object types which can participate in a given process.

Most programming languages lack a concept corresponding to phenotype. A programming language cannot, in general, look at a memory block and determine the type of object it contains by analyzing its contents. A phenotype, on the other hand, is a list of conditions that

determine how to categorize an object from the object's properties. However *both* programming languages and biology have type systems. Indeed, biological modeling is the creation of a type system for a specific situation –collecting and defining a set of rules which specify which types of objects participate in which types of processes.

CCOPM extends the programming language concept of type with a rule-based definition of type. The type definition needs to be fuzzy so that we can ask how much an object instance participates in a type. Current programming languages support only Boolean type inquiries, in the CCOPM inquiring if an object is of a specified type returns a real value between 0 and 1.

In tissues, dynamic sets of objects participate simultaneously in multiple processes. The CCOPM must therefore be able to execute *efficiently* many processes concurrently. The number of active processes may range from hundreds for the simple simulations, to many millions, more than the number of processors in most computers. The CCOPM compiler and run time must handle concurrency in continuous and discrete processes differently. Mathematical equations define continuous processes. The CCOPM compiler must convert these relations into numerically-solvable differential equations and invoke numerical solvers to evaluate the time evolution of their continuous state variables. Discrete processes resemble messages in actor-oriented programming languages 3, with the CCOPM runtime creating a *thread pool*, storing pending discrete processes invocations in a queue, and dispatching them to available threads.

In biology, complex conditions determine the activity of continuous processes and the triggering of discrete processes. The CCOPM must be able to: a) express these conditions, and b) continuously monitor these conditions and appropriately activate, terminate or trigger processes. When a discrete process triggers, the CCOPM runtime must create a closure for the process (the set of variables available to the triggered process at the moment it triggers), then place the process and its associated closure into a priority queue.

The CCOPM must express complex, heterogeneous spatial relationships and structures. Many types of objects and processes exist in an explicit or implied containing space. Some types of objects may have spatial extent or shape, be volume excluding or contain other objects spatially. Volume exclusion and containment may be complex. CCOPM objects and processes must be composable – objects may contain other objects and process may contain other processes. Objects with spatial extent and spatial objects contained in other spatial objects require geometric transform to map their local coordinate systems to the containing spatial object's coordinates. All spatial objects reside within a top-level global coordinate system.

Biological tissues continually create, destroy and rearrange both objects and processes. The CCOPM must support creating and destroying both discrete objects such as cells and continuum objects such as chemical concentrations, fields and fluids. Objects frequently change type, e.g., through cell differentiation or chemical reactions. The CCOPM requires a type system for expressing and querying type information.

Biological objects always exist in an environment. The phenotype of a biological object changes with its environment. Since biological objects move through numerous local environments, the CCOPM must automatically provide the local environmental state to any processes in which an object participates. Equivalently, we could say that the local environment participates in all processes. Many programs organize objects in containment hierarchies, since the processes in which an object participates often need to obtain information from the objects containing that object. Gil and Lorenz (Gil and Lorenz 1996) have denoted the automatic provision of this information, *environmental acquisition*.

2.2 Usability

The CCOPM should also be convenient to use, with a minimal learning curve, and present complex information in a comprehensive and readily understandable form. The CCOPM should also enable domain experts to create, view and edit models using familiar representations.

Diagrams are nearly always a first step in capturing knowledge. Textbooks and journal articles nearly always include diagrams and explanatory text. Diagrams both summarize mechanistic understanding of sets of observations and qualitatively and/or quantitaively present information. Unfortunately, while diagrams can be intuitive to humans, they lack semantic information intelligible to computers. For example diagrams extensively and ambiguously use the symbol →. Depending on context, → can mean influencing, moving towards, becoming, consuming or several of these meanings at once. Even humans require tacit, domain-specific knowledge to interpret (and misinterpret) diagrams. Diagrams in many domains are essentially free-form, with little standardization. While systems biology and object-process modeling have developed standard graphical notations such as the systems biology graphical notion (http://www.sbgn.org) and object-process methodology (Dori 2002), these standards are far from universal. Where practical, the CCOPM should follow existing standards of graphical notation.

2.3 Model Composition

Complex objects in biology are almost always composites of smaller objects (whether organs, tissues, cells, organelles or molecules). Tissues modeling requires composition of sub-models into larger models, for example models of synapses may couple models of sub-cellular chemical reaction to models of extracellular diffusion (Greget, Pernot, Bouteiller, Ghaderi, Allam, Keller, Ambert, Legendre, Sarmis, Haeberle, Faupel, Bischoff, Berger, and Baudry 2011). Multi-cell models may couple models of models of intra- and extracellular dynamics to cell motion (Swat, Thomas, Belmonte, Shirinifard, Hmeljak, and Glazier 2012). Whole-cell models may couple Boolean network models of regulation to flux balance and chemical reaction-kinetics models (Karr, Sanghvi, Macklin, Gutschow, Jacobs, Bolival Jr., Assad-Garcia, Glass, and Covert 2012).

Most developers of tissue simulations employ general-purpose programming languages to couple sub-models, which requires developer to learn numerous domain-specific languages *and* general purpose programming languages *and* APIs for integrating appropriate numerical solvers. The resulting conglomerate models cannot readily be shared or exchanged, because

general-purpose programming languages do not retain the original biological and mathematical knowledge used to create the model. Sharing general-purpose language composite models is like taking a program written in a high-level programming language, compiling it for a specific processor architecture, distributing the binary, and expecting the user to decompile the binary and reverse-engineer the original source code. Interchange standards, like SBML for chemical networks, support sharing of select types of biological model sub-components, but no current language supports sharing tissue models or retains the biological knowledge embodied in these models. The CCOPM must provide natural tools that allow the simple composition of components and also the extraction of components from composite models.

3 RELATED WORK

We explored the suitability of many programming language paradigms, including functional-, procedural-, object-, actor-, logic- and equation-oriented approaches for agent-based tissue modeling. No single approach meets the requirements discussed in § 2; though, each offers concepts which we incorporated into our methodology.

Alan Kay, a pioneer of object-oriented (*OO*) programing invoked cell-to-cell signaling to motivate the OO paradigm, “I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages.” (Kay 2003). OO languages define two categories: objects and messages, and objects communicate with one another via discrete messages. However, cell communication and biological processes like chemical reactions, diffusion and cell motion are continuous, and easier to describe using mathematical relations than messages.

Actor oriented (*AO*) languages such as Erlang and Scala (Karmani and Agha 2011) are similar to OO languages, except that each object (*actor*) runs concurrently and communicates with other actors via messages. Actors may send messages, create or destroy actors, and modify their own states in response to messages. Unlike most OO languages, AO languages automatically handle concurrency issues (threading, locking, etc.) saving developers a significant burden. The AO approach is closer to Alan Kay’s concept of cell communication, with all communication via messages, and each actor (cell) only able to modify its own internal state. As in nature, AO objects and messages act asynchronously. AO are a natural implementation of biological discrete processes, but, as with OO languages, AO language messages do not naturally describe continuous processes.

Robin Milner (Milner 1982) devised process calculus (“π-calculus”) languages to help formalize models of communicating phenomena. π-calculi can visually represent and simulate biochemical reactions of molecules (Phillips and Cardelli 2007). π-calculus languages define computations as sequences of causally dependent events. E.g. π-calculus models of chemical reaction networks create “processes” to define the *existence* of molecules. Multiple copies of these processes run in parallel to simulate multiple molecules. Communication between processes models chemical reactions between molecules. 3π-calculus (Cardelli and Gardner 2010) 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 their parent processes. π-calculus naturally describes discrete biological processes, but not continuous or spatially extended objects and processes.

Engineers employ equation-oriented (*EO*) (Morton 2003) languages, such as Modelica, LabVIEW or SBML, to model electrical circuits, drive trains, hydraulics, chemical reactions, etc. . . . EO languages specify relationships and dynamics of variables with mathematical equations. EO language compilers automatically determine independent/dependent variables relationships and solve equations simultaneously using suitable numerical algorithms. EO languages support interconnecting sub-models to create composite models. Many EO languages also support visual model definition and composition. EO languages are ideal for specifying models with a fixed number of objects and connectivity, but lack dynamic object creation, deletion, and dynamic rearrangement. Most EO languages lack an intrinsic notion of space.

The Object Process Methodology (OPM) (Dori 2002) is a process modeling language for conceptual modeling and analysis. OPM processes can create, delete or transform objects. OPM defines a visual layout standard, which we adopt. Like an ontology, OPM does not define computations, only object-process relationships and annotations.

4 THE CONTINUOUS CONDITIONAL OBJECT PROCESS METHODOLOGY

The Continuous Conditional Object Process Methodology (*CCOPM*) is a programming language paradigm that augments existing strongly typed object-oriented programming languages, such as Reticulated Python (Kent, Baker, Vitousek, and Siek 2014), or TypeScript (http://typescriptlang.org) by adding a set of constructs to describe, in a unified way, features of agent-based models. Our approach combines concepts pioneered in equation- and actor-oriented languages with the object-process dualism of the Cell Behavior Ontology (*CBO*) (Sluka, Shirinifard, Swat, Cosmanescu, Heiland, and Glazier 2014) and Dori's Object Process Methodology (*OPM*).

Objects are nouns, the *things* being described. Objects have quantitative and qualitative properties such as amount, concentration, mass and volume, which define the object's *state*. Objects extend standard programming language data structures. Objects may inherit from, extend, or contain other objects. Spatial objects have a coordinate transform to map their local coordinate system to their containing spatial object's coordinate system.

Processes are verbs, the actions of objects. Processes can change the state of, create, destroy or transform objects. Processes form two basic types: *Existential* processes create or destroy objects, and *transformational* processes change the states of objects. Each basic process type may be either *continuous* or *discrete*. Continuous processes operate continuously in simulation time. A continuous process has an associated rate expression, which defines the rate at which the process creates, destroys or transforms objects. Discrete process are instantaneously create, destroy or transform objects. External processes and internal conditions can both trigger discrete processes.

Conditions can define when a continuous process is active, or a discrete process triggers. Conditions are Boolean expressions which can query the properties of the process

participants. The CCOPM runtime evaluates conditions at every time step, and when conditions change value, activates, terminates or triggers the appropriate processes. When a discrete process triggers, the CCOPM runtime creates a closure and inserts the process and closure into a priority queue so the next available thread will read and invoke it.

Rate processes are a subtype of transformation process which generalizes the concept of chemical reactions to *any* object type. They define a rate function which determines the rate of change of one or more properties of the process' participants. E.g., force rate processes can move or deform spatial objects by altering their spatial properties. An object instance can participate in an unlimited number of rate processes simultaneously.

In the interests of numerical efficiency, the CCOPM runtime time evolves objects and processes by time slicing.

4.1 Visual Representation and Editing of Models

To support graphical display and editing of models, we creating an HTML5 widget to display CCOPM. This widget extends the algorithms Dr. Somogyi previously used to graphically render and edit MathML (Padovani 2003, Somogyi 2005). The widget parses the model source code into an abstract syntax tree (*AST*), which it treats as a *live document*. The widget allows editing and manipulation of the AST. The widget then translates the AST into an HTML DOM a web browser displays. The HTML contains SVG elements which correspond to model objects and process. The widget uses a standard Model-View-Controller (*MVC*) approach, where the AST is the 'model', the SVG canvas is the 'view', and the controller responds to any user events that the SVG elements signal.

Figures (1, 2) show sample visual model representations. Graphical elements use standard Systems Biology Graphical Notation (*SBGN*) glyphs, where possible. Since CellDesigner (Funahashi, Morohashi, Kitano, and Tanimura 2003), a diagram editor for drawing gene regulatory and biochemical networks uses SBGN, users who are accustomed to creating biochemical networks can apply their experience to building tissue models. We use OPM graphical conventions for hierarchical containment and other concepts not present in SBGN.

5 RESULTS AND MOTIVATING EXAMPLES

We have implemented a prototype CCOPM as a new programming language derived from Microsoft TypeScript named *Cayman*. TypeScript is a strongly typed language that: 1) is open source, 2) has a simple and readily extensible grammar, 3) has a parser and source-to-source translator written in TypeScript, which compiles TypeScript into other languages such as JavaScript. We have implemented a prototype source-to-source translator which compiles Cayman into a combination of conventional Python, SBML and XML executable in the CompuCell3D modeling environment. Future versions of Cayman will compile directly into machine executable code. CCOPM augments Cayman with a few new reserved keywords.

The text-based examples below demonstrate a few semantic aspects of CCOPM. Every element of CCOPM also has a visual representation which we do not discuss here.

Conditional Processes have operational semantics and become active whenever process conditions are met. The CCOPM runtime continuously monitors changes in these conditions and activates, terminates or triggers processes accordingly. In our prototype CCOPM, the triggering clause compiles to an SBML expression, solver's SBML engine monitors the status of this expression, and, when that status becomes true, calls and executes the body of the process, which CCOPM compiles into standard Python. The syntax for a sample conditional process is:

```
proc my_apoptosis(x:SomeType) when (x.health < 10) {x.die();}
```

Chemical Reactions are one of the fundamental phenomena in biology. A chemical reaction is a process which consumes a set of reactants and produces a set of products at a specified rate. Here, the reaction $2\mathrm{E}+\mathrm{S} \underset{k_2 ES}{\overset{k_1 E^2 \cdot S}{\rightleftarrows}} \mathrm{ES} \xrightarrow{k_3 ES} \mathrm{E}+\mathrm{P}$ can be specified as two processes:

```
proc p1(2 E, S) <-> (ES) {k1 * E**2 * S - k2 * ES;}
proc p2(ES) -> (E, P) {k3 * ES;}
```

Whenever two chemical objects of the proper type exist at the same location (or within the same containing object if they are non-spatial objects), they automatically participate in any appropriate reactions. If the chemical objects exit within a cell object, the compiler converts these process definitions into an appropriate SBML model, which it then attaches to the cell object.

Membrane Transport is critical for cell function. In the example, an amount of a spatial chemical object outside a cell diffuses across the cell object spatial boundary to become an amount of a nonspatial, internal chemical object.

```
proc my_trans(src:MyCellType.surface.glucose) -> (dst:MyCellType.glucose) {
       return 0.01 * (sum(src)/area(src) - dst/volume(dst));
}
```

The CCOPM compiler generates code which instructs the runtime to consume an amount of glucose at every location at which the spatially-extended cell object contacts the glucose spatial chemical field object, and add the consumed glucose to the glucose scalar chemical object which the cell object contains.

Cell Adhesion, Growth and Division are fundamental process to multicellular organization. Cells may adhere to other cells or to extracellular materials. Our example includes spatially extended spatial cell objects of two types, GreenCellType and BlueCellType, a chemical field spatial object of type Nutrient. Each cell object also contains a non-spatial scalar chemical object of type Protein. Both cell types inherit from the built-in Cell type, hence they inherit all of the properties and processes associated with the Cell type.

The Cell type in turn is a sub-type of the SpatialObject type. The SpatialObject type defines a number of functions, such as the distance between two object instances. The Nutrient object type inherits from the built-in ChemicalField type. ChemicalFields are a continuous object type that spans a SpatialRegion.

The model includes three processes: a transformative, continuous cell-adhesion process, a transformative, continuous transport process and an existential, discrete mitosis process. Figure (2) presents a graphical representation of these processes.

Two instances of a cell object can participate in an application of a cell-adhesion process, which is conditional on the types of the participating cells. The adhesion process is only active when the participating cell objects are within a certain distance of each other. As the cell objects are subtypes of the SpatialObject type, we use the ‘distance’ function to calculate the distance between their instances:

```
@adhesion(distance(a, b) < 1)
function green_adhesion(a:GreenCellType.surface, b:GreenCellType.surface) {
        return 1.5;
}
```

Cell objects, the nutrient chemical field object and the scalar intracellular protein object participate in an application of a transport process, which transfers an amount of extracellular nutrient in the nutrient object into an amount of intracellular protein in the protein object:

```
proc green_consumption(src:GreenCellType.surface.nutrient) ->
        (dst:GreenCellType.protein) {
        return 0.01 * (sum(src)/area(src) - dst/volume(dst))
}
```

The amount of protein in an intracellular protein object in an instance of a cell object reaching a threshold triggers a mitosis process, which destroys the participating cell instance and creates two new cell instances of the same type as the destroyed cell object instance:

```
trans mitosis(parent:GreenCellType) -> (d1:GreenCellType, d2:GreenCellType)
        when (parent.nutrient > 5) {
        d1 = parent;
        d2 = GreenCellType(parent);
        d1.protein = 0.5 * parent.protein;
        d2.protein = 0.5 * parent.protein;
        return (d1, d2);
}
```

6 CONCLUSIONS

No current programming language can both model and simulate the complex multi-scale structure and interactions of biological tissues in a unified way. The CCOPM extends general purpose programming languages to support such modeling and simulation.

Acknowledgments

We acknowledge generous financial support from the Falk Medical Research Trust Catalyst Program, 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 and an Indiana University Cooperative Research grant. We thank Dr. Marie Gingras, Dr. Jeremy Siek and Dr. James Sluka for their discussion and insights.

References

Alberts, B., Bray, D., Hopkin, K., Johnson, A., Lewis, J., Raff, M., Roberts, K., Walter, P. Essential Cell Biology, Fourth Edition. New York, NY: Garland Science; 2013.

Brodland GW. How computational models can help unlock biological systems. Seminars in Cell & Developmental Biology. 2015; 47–48:62–73.

Cardelli, L., Gardner, P. Conference on Computability in Europe. Springer; 2010. Processes in space; p. 78-87.

Dori, D. Object-Process Methodology: A Holistic Systems Paradigm. Berlin: Springer; 2002.

Fisher J, Henzinger TA. Executable cell biology. Nature Biotechnology. 2007; 25:1239–1249.

Funahashi A, Morohashi M, Kitano H, Tanimura N. CellDesigner: a process diagram editor for gene-regulatory and biochemical networks. Biosilico. 2003; 1:159–162.

Gil J, Lorenz DH. Environmental acquisition: a new inheritance-like abstraction mechanism. ACM SIGPLAN Notices. 1996; 31:214–231.

Gilbert, SF. Developmental Biology. Sunderland, MA: Sinauer Associates; 2014.

Greget R, Pernot F, Bouteiller JMC, Ghaderi V, Allam S, Keller AF, Ambert N, Legendre A, Sarmis M, Haeberle O, Faupel M, Bischoff S, Berger TW, Baudry M. Simulation of Postsynaptic Glutamate Receptors Reveals Critical Features of Glutamatergic Transmission. PLoS ONE. 2011 Dec.6:e28380. [PubMed: 22194830]

Karmani, RK., Agha, G. Encyclopedia of Parallel Computing. Boston, MA: Springer; 2011. Actors; p. 1-11.

Karr JR, Sanghvi JC, Macklin DN, Gutschow MV, Jacobs JM, Bolival B Jr, Assad-Garcia N, Glass JI, Covert MW. A Whole-Cell Computational Model Predicts Phenotype from Genotype. Cell. 2012 Jul.150:389–401. [PubMed: 22817898]

Kay, Alan C. [Accessed Mar. 30, 2016] Alan Kay on the Meaning of ”Object-Oriented Programming”. 2003. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

Kent, AM., Baker, J., Vitousek, MM., Siek, JG. Design and evaluation of gradual typing for Python. Proceedings of the 10th ACM Symposium on Dynamic Languages; Association of Computing Machinery; 2014 May. p. 45-56.

Milner, R. A Calculus of Communicating Systems. Secaucus, NJ: Springer; 1982.

Morton W. Equation-oriented simulation and optimization. South African Computer Journal. 2003; 69:317–358.

Padovani, L. Ph D thesis. Bologna: 2003 Jan. MathML Formatting.

Phillips, A., Cardelli, L. Efficient, correct simulation of biological processes in the stochastic π-calculus. International Conference on Computational Methods in Systems Biology; Springer; 2007. p. 184-199.

Sluka JP, Shirinifard A, Swat M, Cosmanescu A, Heiland RW, Glazier JA. The cell behavior ontology: describing the intrinsic biological behaviors of real and model cells seen as active agents. Bioinformatics. 2014; 30:2367–2374. [PubMed: 24755304]

Somogyi, Endre. [Accessed Mar. 30, 2016] gNumerator MathML rendering library. 2005. http://numerator.sourceforge.net

Swat MH, Thomas GL, Belmonte JM, Shirinifard A, Hmeljak D, Glazier JA. Multi-Scale Modeling of Tissues Using CompuCell3D. Methods in Cell Biology. 2012; 110:325–366. [PubMed: 22482955]

Biographies

ENDRE SOMOGYI is a Lecturer of Computer Science at Indiana University. His interests are investigating how natural phenomena may be understood via programming languages and simulation, and the design of domain-specific programming languages and compilers for physics based modeling and simulation. somogyie@indiana.edu.

AMIT HAGAR is Professor and Chair of the Department of History and Philosophy of Science and Medicine. His interests are the the foundations of modern physics, especially in the notion of objective chance and the biophysical modeling of cancer. hagara@indiana.edu.

JAMES A. GLAZIER is a fellow of the American Physical Society and of the Institute of Physics (London), is Professor of Intelligent Systems Engineering, Adjunct Professor of Physics and Biology and Director of the Biocomplexity Institute at Indiana University, Bloomington. His research interests include the development of open-source software tools for simulating virtual tissues, development of standards for model representation and sharing, and the development of personalized-medicine models of developmental diseases. jaglazier@gmail.com.

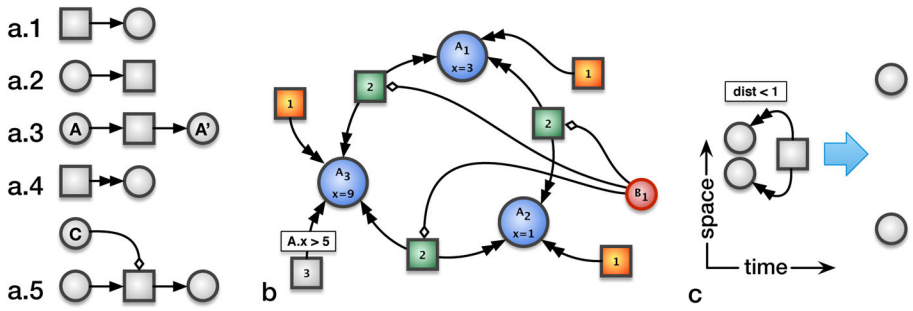

Figure 1.

Objects and Processes: following the SBGN standard, objects are circles and processes are squares. a) Types of process relationship: a.1) a process creates an object, a.2) a process destroys an object, a.3) a process destroys an object of type A and creates an object of type A', a.4) application of a continuous transformation process with a participating object, the process modifies a property of the object, a.5) information flow from a participating object to a process, an object of type C modifies a process which destroys an object and creates an object. b) Multiple applications of processes 1 and 2 with three participating instances of objects of type A. Process 2 uses information from participating object B_1 to influence how it changes the participating A object instances. Process 3 is a conditional process, only active when $A.x > 5$. c) A conditional adhesion process with two participating objects initially in close proximity. The adhesion process is only active when the distance between its two participating objects is less than 1. At a later time, the objects have moved apart, and the adhesion process is no longer active.

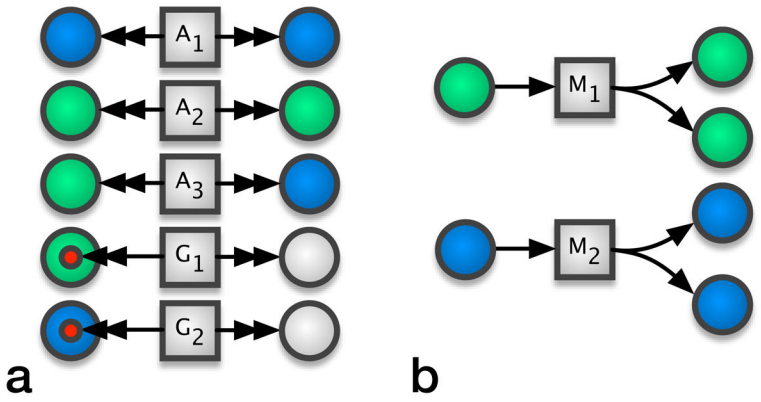

Figure 2.

Processes related to cell adhesion, growth and division. a) Cell objects participate in adhesion processes, A_1, A_2, A_3, conditional on the cell object type. The nutrient chemical field spatial object and a protein non-spatial object contained in a cell object participate in a transfer process G_1 and G_2, conditional on the cell object type. The transformation process reduces the amounts of Nutrient in the chemical field object and increases the amount of Protein in the protein object (red object contained within the cell objects). b) Cell objects participate in mitosis processes M_1, M_2, conditional on cell type. The mitosis process destroys a cell object of a given type and creates two cell objects of the same type.