2013 IEEE 9th International Conference on e-Science

# Biocharts: Unifying Biological Hypotheses with Models and Experiments

Hillel Kugler
Microsoft Research Cambridge
Email: hkugler@microsoft.com

***Abstract*—Understanding how biological systems develop and function remains one of the main open scientific challenges of our times. An improved quantitative understanding of biological systems, assisted by computational models is also important for future bioengineering and biomedical applications. We present a computational approach aimed towards unifying hypotheses with models and experiments, allowing to formally represent what a biological system does (specification) how it does it (mechanism) and systematically compare to data characterizing system behavior (experiments). We describe our Biocharts framework geared towards supporting this approach and illustrate its application in several biological domains including bacterial colony growth, developmental biology, and stem cell population dynamics.**

***Index Terms*—Computational Systems Biology; Visual Formalisms; Temporal Logic; Developmental Biology; Stem Cells.**

## I. Introduction

Computer technology and software have changed the way biological research is performed. A striking example is the genomic revolution, which would have been impossible without the practical support of software in storing, accessing and analyzing very large data sets [1], [2]. However there is a deeper connection between biology and software, based on the resemblance between a biological system and a software system, both being complex information processing "machines" composed of many small units functioning together to achieve various high-level requirements [3]. One of the unique features of biological systems that make them different than systems studied in other natural sciences is the inherent process of computation that living systems perform. A main challenge for the biological sciences in the next years will be developing a theoretical framework to study biological computation that will allow synthesizing pieces of the puzzle we know or aim to discover using reductionist methods into a coherent system level understanding of life. This could enable an understanding of what cells and organisms compute, how they do it, what happens if computation goes wrong, and how can such "errors" in biological computation modules be fixed. As a starting point to even begin and tackle these questions, a framework that allows a formal yet user friendly representation of biology is needed. Here we outline a proposal for such a framework, towards unifying hypotheses with models and experiments, allowing to formally represent what a biological system does (specification) how it does it (mechanism) and systematically compare models to data characterizing the system behavior (experiments). We introduce our method that unifies scenario-based and state-based modeling, provides a visual interface to temporal logic via scenarios, and allows scalable simulation, visualization and analysis.

## II. Scenarios

Scenarios have emerged as a natural way to describe and study system behavior (see e.g. [4], [5], [6], [7], [8]), and are especially useful in modeling reactive systems [9], whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Biological systems can be viewed as the ultimate reactive systems, and thus methods influenced by engineered reactive systems are now being adapted and developed for modeling biological systems. In a scenario-based approach, each scenario captures a "story" about some aspect of behavior, and all the scenarios together describe and define the overall system dynamics. Here we present our modeling framework based on the language of Generalized Live Sequence Charts (GLSCs), that extends Live Sequence Charts [10] and adapts it [11] towards biological modeling.

### A. Generalized Live Sequence Charts

Generalized Live Sequence Charts (GLSCs) is a visual language for modeling behavioral requirements, and forms a foundation for our framework for modeling biological systems, Biocharts. We describe the main language features, outline the semantic definitions and explain some of the design decisions and the motivation from the biological perspective.

We consider an object oriented setting, where a system is composed of a set of classes and objects. Each object is an instance of one of the classes. The dynamic behavior is defined by GLSCs, which capture the specification (what the system should do), mechanistic behavior (how the biological system works), design principles (including system invariants and temporal properties), and experimental results (what is known about the biological system from lab experiments). A powerful capability of GLSCs and the Biocharts framework is the ability to represent specifications, mechanisms and experiments within the same framework.

More formally, an *object system* $\mathcal{O}$ is a set of *classes* and a set of *objects*, with each object mapping to (by being an instance of) exactly one class. *GLSC requirements* are a set of GLSCs which collectively describe the behavior of $\mathcal{O}$. A *Generalized LSC* (GLSC) $L$ describing $\mathcal{O}$ is defined by $L = \langle \mathcal{I}, M, evnts, temp, side, \mathcal{V} \rangle$ where:

978-0-7695-5083-1/13 $25.00 © 2013 IEEE
DOI 10.1109/eScience.2013.41

$\mathcal{I}$ is a set of *instances*, each defined as a pair $\langle \texttt{locs}, O \rangle$ where `locs` is a finite totally ordered set of locations labeled $1, 2, \ldots max_O$ and $O \in \mathcal{O}$ is the object (or class) corresponding to the instance. For any $I \in \mathcal{I}$, we denote the set of $I$'s locations by $\texttt{locs}(I)$. We assume that the $\texttt{locs}(I)$'s are mutually distinct, and denote by $\texttt{locs}(L)$ the set of all locations of $L$.

$M$ is a set of *methods* and *messages*. A method is a 4-tuple of the form $\langle \texttt{name}, sender, receiver, \texttt{mode} \rangle$ where `name` is the name of the method, *sender* and *receiver* are the names of the instances from which the method is sent or by which it is received, and $\texttt{mode} \in \{a, s\}$ denotes whether the message is *asynchronous* or *synchronous*. A message is a method parameterized by a *variable* or *property*.

$evnts\colon \texttt{locs}(L) \rightarrow E$ maps locations to the events that occur in them. The set $E$ of events consists of the set $E_v$ of *visible events* (methods and messages), and the set $E_h$ of *hidden events* (assignments, conditions, and jumps), thus $E = E_v \cup E_h$. The set of $L$*'s visible events*, denoted by $E_v$, consists of the sending/receiving events of messages and methods, i.e., $E_v = \{snd(m), rcv(m) : m \in M\}$. Function $evnts^{-1}\colon E \times L \rightarrow \texttt{locs}(L)$ maps events to a set of locations within GLSC $L$.

$temp\colon \texttt{locs}(L) \rightarrow \{H, C, W\}$ maps locations to hot, cold, or warm temperatures.

$side\colon E \rightarrow \{sys, env\}$ maps events as originating from the *system* or the external *environment*. Hidden events are always system events.

$\mathcal{V}$ is a set of typed *variables*. The types directly supported by GLSCs are bounded non-negative integers, a numeric integer range, enumeration, and Boolean.

An example GLSC with labeled locations appears in Fig. 1

### B. Executions

A *GLSC execution* tracks the dynamic progress along a scenario. For this purpose, a GLSC execution is coupled with state information—a mapping from the GLSC variables $\mathcal{V}$ to values, and also a *cut*, defined as a set of locations with precisely one location from each instance line. Given a GLSC $L$ and execution $L_i$ (there can be several executions per GLSC at runtime being tracked simultaneously) the lifetime of the variables appearing in execution $L_i$ corresponds to the lifetime of the execution itself. We avoid undefined values by requiring that any variable in a GLSC has a predefined initial value (can be a nondeterministic or random assignment), which is used to initialize the variable each time an execution is created. The *initial cut* is a cut whose instances are all at their topmost locations. An event $e \in E$ is *enabled* in execution $L_i$ if the cut is located one position above each location in $evnts^{-1}(e, L_i)$. An event is *minimal* if it is enabled from the initial cut.

Liveness of an execution (the requirements on progress for a given cut) is defined by cut *temperature*, which in turn is determined by the temperature of the cut locations. A *hot cut* contains at least one hot location. A *cold cut* contains all cold locations. Progress is required from a hot cut, whereas no progress is required from cold cuts. According

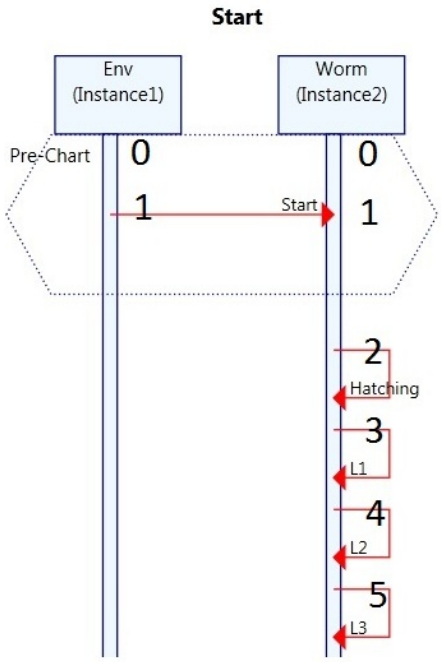


Fig. 1. An example of a scenario describing larval stages in *C. elegans* development, with locations labeled for illustration purposes.

to our semantics, if an execution reaches a cold cut it closes successfully. A *warm cut* contains at least one warm location, but no hot locations. Warm cuts do not require progress and thus an execution can remain infinitely often in a warm cut without being considered as a violation. When an enabled event occurs while being in a warm cut this will cause progress [12].

These definitions are the basis for defining whether a system satisfies a given GLSC requirements. An execution is *satisfied* when it reaches a cold cut. Special cases are progressing to the end of the chart, as the final locations of all instances are cold, or a cold condition evaluating to false. An execution *violates* when the partial order is violated (e.g. by the occurrence of a message appearing in the execution which is not yet enabled) or when a hot condition evaluates to false. To define whether a system satisfies its GLSC requirements we consider all traces composed of visible events that the system can generate, and require that for all traces non of the GLSC executions are violated. This relies on defining a step semantics, and determining a sequence of hidden events that is taken in response to a visible event.

### C. Language Extensions and Semantics

At the core of executing, monitoring, verifying and applying synthesis methods is the notion of a step. In a similar way to the definition of statechart semantics [13], [14], we outline an operational semantics for GLSCs. A full definition including all language features is beyond the scope of this paper, additional information is available online [15]. Intuitively, a step is composed of a visible event, and the response to it,

a set of hidden events that can update local variables and progress locations in GLSCs. We first provide more details on the definition of visible and hidden events, before outlining the step semantics.

An *instance element* is a graphical representation of an event (see Fig. 2). Each instance element is anchored to one or more instance lines, the instances of which are called *participating instances*. There are two kinds of events:

Visible events include messages and methods, which are unidirectional communications from a sender object to a receiver object. Sending and receiving are considered two distinct events for asynchronous communications. For synchronous communications, sending and receiving are treated as a single event. Methods are depicted on the GLSC as an arrow pointing from the sender to the receiver, with the method name above the arrow. Synchronous methods are drawn with a solid arrow head, whereas asynchronous methods are drawn with an open arrow head.

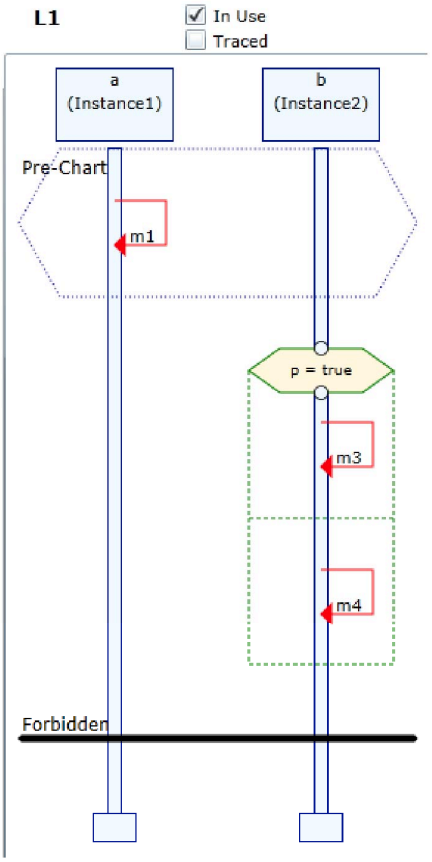


Fig. 2. An example of a scenario describing the behavior of two objects `a` and `b`. If method `m1` of object `a` occurs, as specified in the prechart, then object `b` will execute method `m3` if property `p` is `TRUE`, otherwise if property `p` is `FALSE` it will execute method `m4`.

Hidden events are internal to a GLSC. All instance elements denoting hidden events cause synchronization among the participating instances. They are depicted graphically as shown in Fig. 2, 3. Assignments are shown as a rectangle with an assignment expression inside. This causes an assignment of a GLSC variable to the value of an expression. Conditions are shown as a hexagon containing a conditional expression. False evaluation results in either satisfaction (closure) of the GLSC if cold, or violation of the requirements if hot. True evaluation results in the cut being able to pass beyond the condition. Jumps are not shown on the chart directly, but are used in conjunction with conditions to implement higher level control structures like loops and if-then-else. Synchronization is represented using a hexagon with no conditional expression inside and is syntactic sugar for a true condition, specifying a synchronization with no other side-effect.

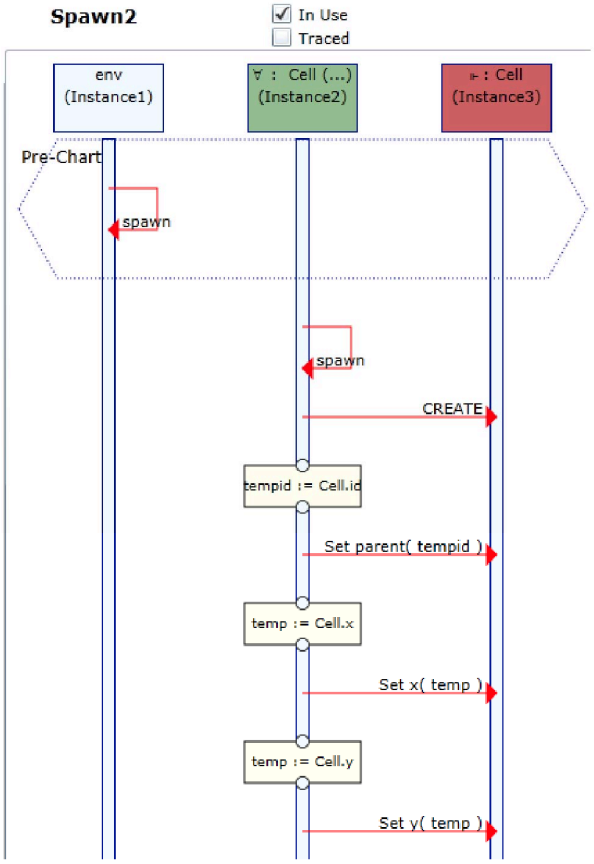


Fig. 3. An example of a scenario describing class behavior, showing universal quantification and dynamic object creation.

Visible events occurrences are global and not necessarily associated with a specific GLSC. When an event occurs, each GLSC execution is responsible for *binding* the event to an instance element within the chart. If the event carries additional information, such as parameters, additional actions may be taken as part of the binding, such as unification. For example, if a message event contains a constant actual parameter $k$ and some GLSC has a message instance element

with formal parameter $X$, the system will unify $k$ and $X$.

**Algorithm 1** Step algorithm

```
1: function STEP(v,mode)
2:     Create new executions for minimal events
3:     if v is a message then
4:         for each GLSC ℓ in the requirements do
5:             for each execution e of ℓ do
6:                 Bind visible event and perform unification
7:                 if unification failed then
8:                     Return failure
9:                 end if
10:            end for
11:        end for
12:    end if
13:    for each GLSC ℓ in the requirements do
14:        for each execution e of ℓ do
15:            Bind visible event v to an instance element i
16:            if e.SingleStep(i) == success then
17:                ProcessHiddenEvents(e,mode)
18:            else
19:                Partial order violation of ℓ
20:            end if
21:        end for
22:    end for
23:    Compute the set of minimal events
24:    Compute the set of enabled events
25: end function
```

We provide a schematic description of the step semantics in Algorithms 1,2. The routine `Step` gets a visible event and the execution mode, performs bindings and unification and invokes the routine `SingleStep` that moves a cut onto an instance element (which is bound to some enabled event). It performs a series of final checks before declaring the step successful and updating the cut. After updating the cut, bookkeeping is performed. If the GLSC contains a subchart, the subchart receives its own execution. Subcharts report the result of their execution (i.e., satisfied or violated) to the parent chart.

## III. STATECHARTS

We adapt classical statecharts [16] following the directions outlined in [17], [18]. The statechart itself is similar to the description in [16], [13], [14], in that there are three types of states: *OR-states*, *AND-states* and *basic states*. The OR-states have *substates* related to each other by "exclusive or", AND-states have *orthogonal components* that are related by "and", while basic states have no substates, and are the lowest in the state hierarchy. When building a statechart there is an implicit additional state, the root state, which is the highest in the hierarchy. The *active configuration* is a maximal set of states that the statechart can be in simultaneously, including the root state, exactly one substate for each OR-state in the set, all substates for each AND-state in it and no additional states. The general syntax of an expression labeling a transition in a statechart is "$m[c]/a$" where $m$ is the method that triggers the transition, $c$ is a condition that guards the transition from being taken unless it is true when $m$ occurs, and $a$ is an action that is carried out if and when the transition is taken. All of these parts are optional.

**Algorithm 2** Advancing the Cut by a Single Step

```
1: function SINGLESTEP(instance element i)
2:     if i does not appear in the current GLSC, isn't minimal, or isn't enabled then
3:         Return success
4:     end if
5:     Advance the cut
6:     if i is an asynchronous message receive and message wasn't sent then
7:         Return failure
8:     end if
9:     if i is a hidden event then
10:        if the element cannot be entered then
11:            Return failure
12:        else
13:            Perform actions associated with hidden event (if any)
14:        end if
15:    end if
16:    if the LSC is in a cold cut or cold condition is false then
17:        Return success
18:    end if
19:    if i is a subchart then
20:        Spawn a new execution
21:    end if
22:    Compute enabled events
23: end function
```

### A. Classes and Objects

For Biocharts we adapt some of the principles of [19], [14], especially the way statecharts are incorporated into an object-oriented framework. The motivation for this decision is that typical biological models require specifying many entities (e.g., cells) with the same specification but each one in a different active configuration. These entities can be created and destructed dynamically during execution (representing, e.g., cells being born or dying), so the object oriented framework is a natural one for representing such models.

A system is composed of classes, and a statechart can describe the modal behavior of the class; that is, how it reacts to messages it receives by defining the actions taken and the new mode entered. A class that has an associated statechart describing its behavior is called a *reactive class*. During runtime there can exist many objects of the same class, called *instances*, and each can be in a different active configuration – a set of states in which the instance resides. Thus, a new statechart is "born" for each new instance of the class, and it runs independently of the others. When a new

instance is created, the statechart enters its initial states by taking default transitions recursively until it is in an active configuration.

A new feature that we implement for Biocharts, building on our experience from biological projects, is to enable an object to dynamically create a new object of the same class in exactly the same active state as the original object is in at the time of creation. This is useful in various biological contexts; for example, during cell division, where daughter cells typically inherit the state of the mother cell. In this case, the statechart of the daughter cell is "born" in an active configuration identical to its mother cell, and no default transitions are taken as part of this initialization.

As mentioned above, the general syntax of an expression labeling a transition in a statechart is "$m[c]/a$", for message $m$, condition $c$ and action $a$. The message $m$ is either a method or a message as described in the GLSC definitions.

### B. Steps

At the heart of statechart semantics is the precise definition of the effect of a step, which takes the system from one stable configuration to the next one. In general, we adapt the definitions of [14]. Two important semantic decisions are that changes made in a given step take effect in the current step, and a step take zero time.

Biocharts also supports running lower level modules as programs from GLSCs or statecharts. We now explain how the low level part, which will typically describe the dynamics of the biological pathways and networks, is integrated with the high-level part. A state in a Biochart's statechart can include a 'low-level' module, which is a program $P$. This $P$ is activated on entering the state, by calling $P.Start$, and is stopped when the state is exited, by calling $P.Stop$. The program $P$ has input variables $x_1, x_2, \cdots x_l$, output variables $y_1, y_2, \cdots y_m$ and local variables $z_1, z_2, \cdots z_n$. The input and output variables are part of the object's variables, so that, for example, another program $P'$ activated in an orthogonal state can use an output variable of $P$ as one of its input variables. Input variables are accessed by the program $P$ by calling $x_i.get()$ initially and at any stage of its computation. Similarly, $P$ can set the value of the output variables by calling $y_j.set(val)$.

## IV. Biological Systems

We illustrate the use of the Biocharts framework in several biological domains, emphasizing the general concepts and modeling principles, more details are available online [15] including a tutorial, user guide and access to the models described.

### A. Chemotaxis and Bacterial Colony Growth

Understanding the biological processes and population dynamics of bacteria is a basic research area in biology. In recent years there is also a growing interest in designing bacteria to perform useful tasks in the field of synthetic biology, e.g., for the production of biofuels [20]. We developed a model that describes bacterial population dynamics, allowing to specify individual bacteria and how they utilize chemotaxis while searching for food, specifying bacterial metabolism and cell division and enabling to study emerging system level properties of the overall colony growth [17]. One the motivations for building such a model, is that although bacterial chemotaxis [21] is well studied, there are still significant unknown questions in understanding how chemotaxis works and the effect it has on the overall population dynamics, which is key for bacteria colony survival and important for many of the potential synthetic biology applications.

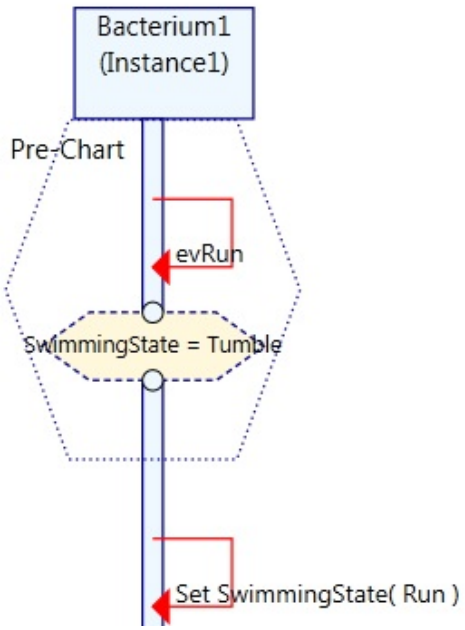


Fig. 4. Bacterial chemotaxis, moving from a $Tumble$ to a $Run$ state.

Bacteria movement is an advantage in heterogeneous environments where nutrients and toxins are not spread out evenly. To guide the movement in a beneficial direction, the bacteria must be able to sense gradients in its environment. Most bacteria are generally considered to be too small to sense gradients across their diameter. Thus, they are forced to sense temporal changes in concentrations during their motion. Movement in bacteria is usually composed of a repeated sequence, consisting of a relatively straight motion followed by a reorientation to another direction. By making the reorientations more frequent when the gradient is in an undesirable direction and less frequent when it is in a beneficial direction, the bacterium can relocate itself to a more suitable place.

In Fig. 4 we show an example of a scenario describing the behavior of a bacterium after reorientation. If method $evRun$ occurs, and the property $SwimmingState$ of $Bacterium1$ is equal to $Tumble$ as specified in the prechart (the dashed hexagon in the top of the chart), then the bacterium will execute message $SetSwimmingState(Run)$ resulting in set-

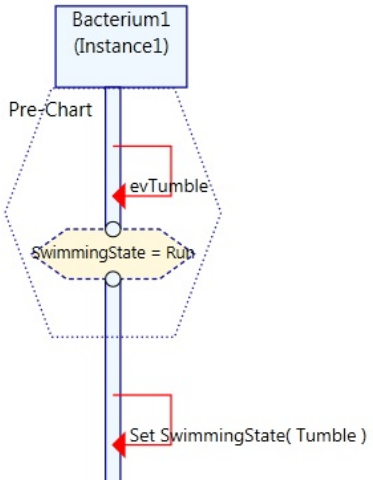


Fig. 5. Bacterial chemotaxis, moving from a $Run$ to a $Tumble$ state.

ting the property $SwimmingState$ to $Run$. In a similar way, another scenario shown in Fig. 5 specifies moving from state $Run$ to state $Tumble$.

The Biocharts framework allows using combined representations, via GLSCs or statecharts, a statechart representation of the $Bacterium$ class appears in Fig. 6. This high-level statechart model invokes lower-level modules capturing the molecular simulation of the pathway involved in chemotaxis [22], as well as solvers tackling the metabolic modelling using flux balance analysis (FBA) [23]. The invocations are carried out based on the statecharts active state configuration. Internally, the statechart execution is achieved via an automatic translation to GLSCs that preserves the statecharts semantics and utilizes the GLSC execution algorithm, a full description of the technical details is beyond the scope of this paper.

### B. Stem cell population dynamics

Stem cells are defined by their capacity to proliferate and to differentiate into specific cell types. Understanding stem cells is both a fascinating scientific question, but also of practical importance as improper control of stem cell populations may underlie tissue degeneration and cancer. The *C. elegans* germline [24] is a tractable system to study stem cell population dynamics.

In the germline, which has the form of a U-shaped tube composed of two "arms", see Fig. 7, germ cells are typically arranged in different regions which contain cells at various differentiation stages, see also Fig. 8. The stem cells are located at the more distal region (called the proliferative zone), and they proliferate to initially grow the germline population and later to maintain it by replacing cells that have matured as oocytes or sperm, or have died. A well orchestrated computation is needed to ensure proper functioning of the system, making sure enough cells are created and at the right times and positions. It is crucial that the underlying computation is accurate and robust, as fertilized eggs lead to

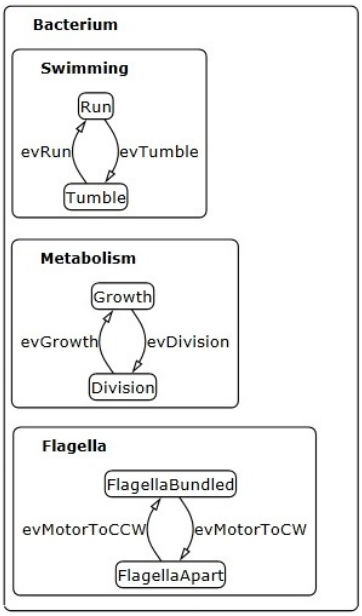


Fig. 6. Part of a Statechart describing Bacterial Chemotaxis

the new generation of worms and are thus key for the species survival.

A cell called the DTC (Distal Tip Cell) is positioned at the distal end of each of the gonad arms, and produces a signal that instructs the cells in the proliferative zone to remain stem cells. A simplified view of the genetic pathway (can be viewed as a high-level intuitive yet informal description of the cellular biological circuit) appears in Fig. 9. A formal and more detailed representation of the underlying biological model appears in the statechart of Fig. 10.

Running model simulations allow a dynamic view into germline development and maintenance, and performing various in-silico experiments. An interesting question related to the overall computation is how fast should stem cells divide to properly maintain the system. Cells have an "internal clock" called the cell cycle [25] that controls the timing of cellular division. The speed of the cell cycle in *C. elegans* is known to be faster in early development stages than in the adult. Model simulations allowed making predictions of the effects of changes to the speed of the cell cycle on the stem cell population. The model predicted that slowing of cell cycle in early development can have significant effects on the proliferative zone in the adult. By utilizing a strain with a mutation that has a slower cell cycle, the in-silico experiments were carried out in the lab and these predictions were conformed experimentally [26]. These results suggest that developmentally regulated cell cycle changes might be an important design principle that impacts the maintenance of stem cell systems.

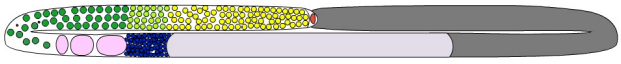

Fig. 7. Schematic picture of the *C. elegans* germline, cells in only one of the gonad arms (left side in figure) are shown. The Distal Tip Cell (DTC) is shown in red at the edge of the gonad arm, The stem cells in the proliferative zone are in yellow, cells in the transition zone are in light green, differentiating cells in green, followed by oocytes in pink and sperm in purple.

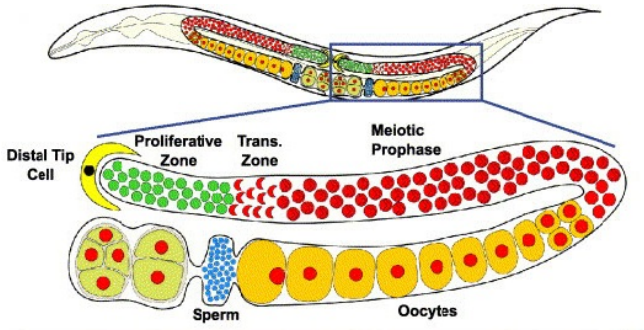


Fig. 8. A more detailed schematic picture of the germline and gonad, and its position within *C. elegans* [27]. The regions shown are from distal to proximal : Distal Tip Cell, Proliferative Zone (stem cells), Transition Zone, Meiotic Prophase (Differentiating cells), Oocytes, Sperm, Fertilized eggs.

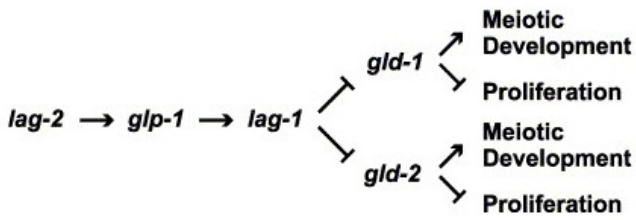


Fig. 9. A Diagram showing a simplified version of the pathway controlling the proliferation/differentiation decision.

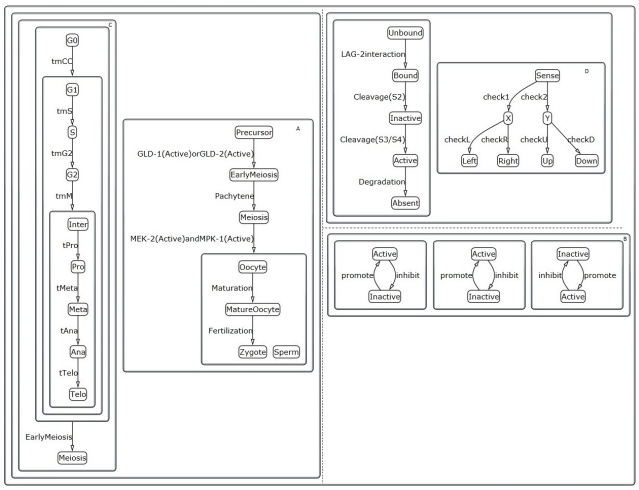


Fig. 10. Top : A statechart of a germline cell.

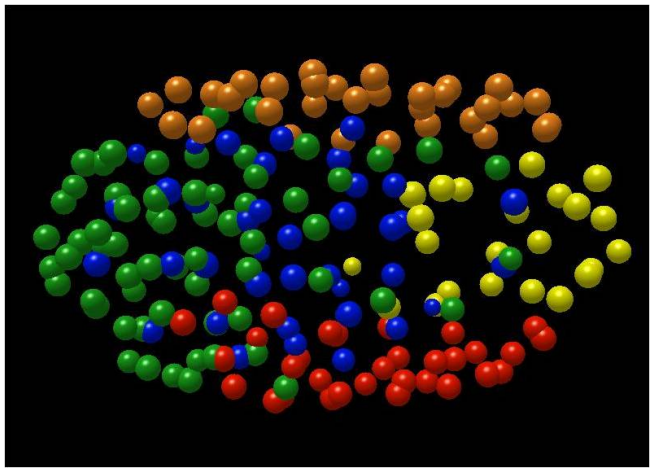

Fig. 11. Embryonic development in *C. elegans*, each cell is shown as an animated sphere, color coding is used to show cells that are progeny of the same mother cell, this allows visualizing that the "orange" cells organize in the top of the figure whereas the "red" cells organize in the bottom. Information is displayed using the Network3D visualization tool [28], data captured from live videos and image analysis algorithms in [29].

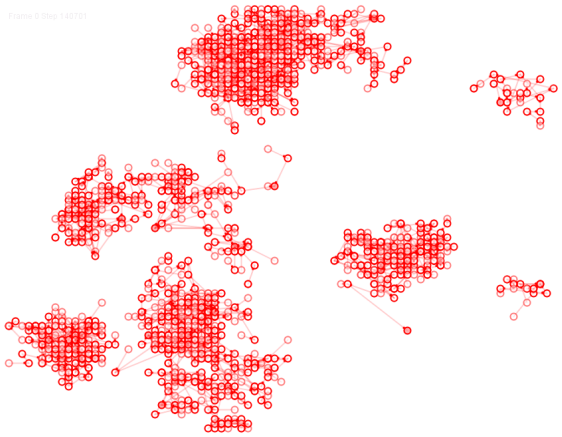

Fig. 12. Simulation of spatial behavior of cell populations : The execution engine can be connected to custom built visualizations, providing information on cell positions and lineage relationship, in this image an arrow between a daughter and mother cell, selecting a cell in the visualization displays the current properties of this cell.

### *C. Cell lineage*

Understanding the cell lineage, the tree of divisions in a multicellular organism, from the first cell division to differentiated cells is a key research area in developmental biology. Biological systems have evolved robust ways to build an organism from an egg to the developed embryo and for maintaining required cells during later stages in the life of the organism, e.g., for reproduction or replacing cells that die. An ideal system to study the lineage is *C. elegans* [30]. The entire somatic lineage of *C. elegans* is known [31], it contains 959 cells and during normal development (termed wild-type) is generated in a reproducible and almost deterministic way. We

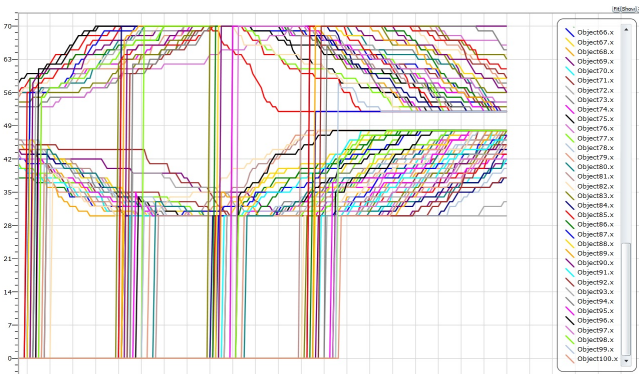

Fig. 13. Visualization plots of complex multi-cellular systems : The user interface allows to specify which cellular properties in the model to plot during model simulation. In this image we are plotting the spatial positions (x,y coordinates) for a stem cell model. Despite the abundance of data, the general picture allows a quick way to compare for example between wildtype (normal) and mutant (perturbed) behavior. Vertical lines initiating at regular time points represent new cells being born.

have constructed a scenario-based representation in Biocharts of the entire lineage. As shown in Fig. 14, the occurrence of method $Start$ in object $Worm$ will trigger the first cell, $P0$ to be born and then divide, later triggering cell $P1$ to divide. The first cell division is specified in Fig. 15, where $P0$ divides to create daughter cells $P1$ and $AB$. The division of cell $P1$ is specified in Fig. 16. The model specifies timing constraints on divisions and allows ablating cells. This preliminary model can serve as a "skeleton" to study an entire organism development, and for connecting computational models of specific parts of the lineage, e.g., vulval development [32] and germline development [26] towards studying the interactions between subsystems.

## V. Conclusion

We present a computational approach aimed at unifying hypotheses with models and experiments, allowing to formally represent what a biological system does (specification) how it does it (mechanism) and systematically compare to data characterizing system behavior (experiments). The approach is supported by visual languages that allow an easier entry point for non programmers, both for specifying the models, for examining existing rules and hypotheses and for tracking simulation and analysis results. The framework is available online and can be run as a desktop application or in a web browser. As our knowledge of biological systems becomes more detailed, the need to synthesize the information into coherent and predictive system models becomes more important, thus we provide a semantic foundation and prototype tool supporting the development of realistic mechanistic models of multicellular systems that enables interpreting and re-examining existing data in a systematic way and investigating new hypotheses in-silico towards experimental testing of predictions in the lab.

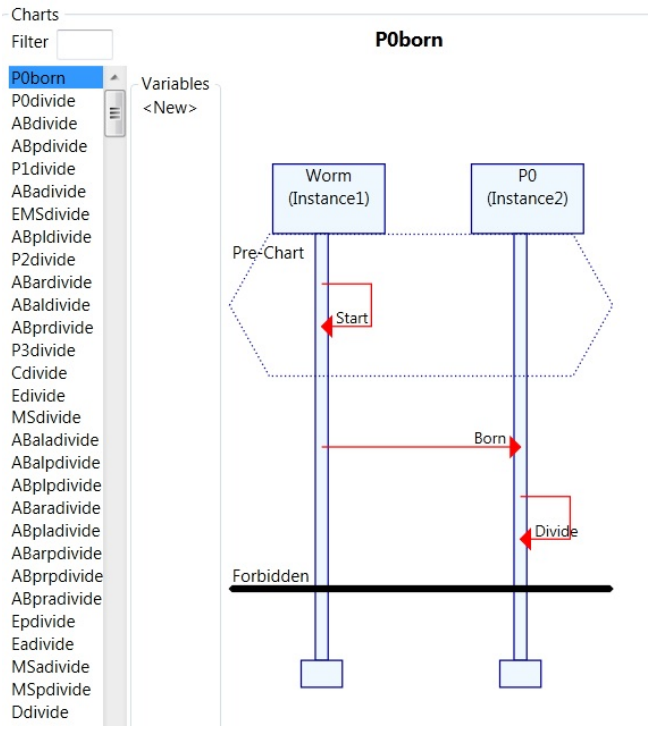


Fig. 14. The lineage in *C. elegans*

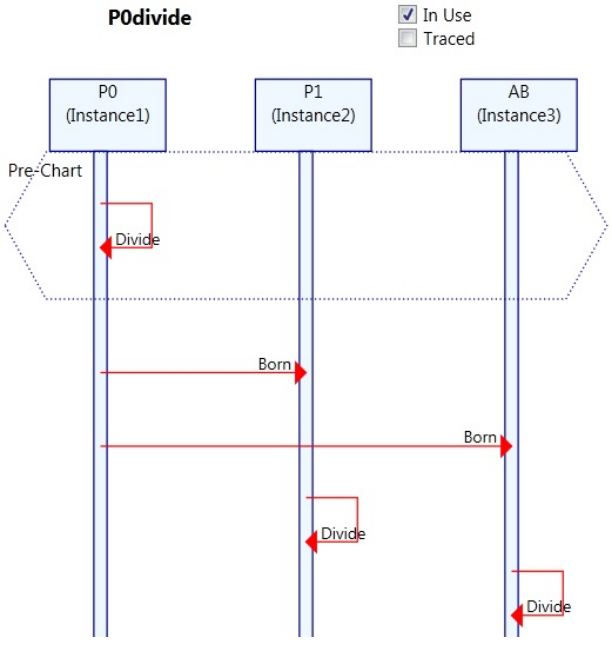


Fig. 15. The first division in *C. elegans*

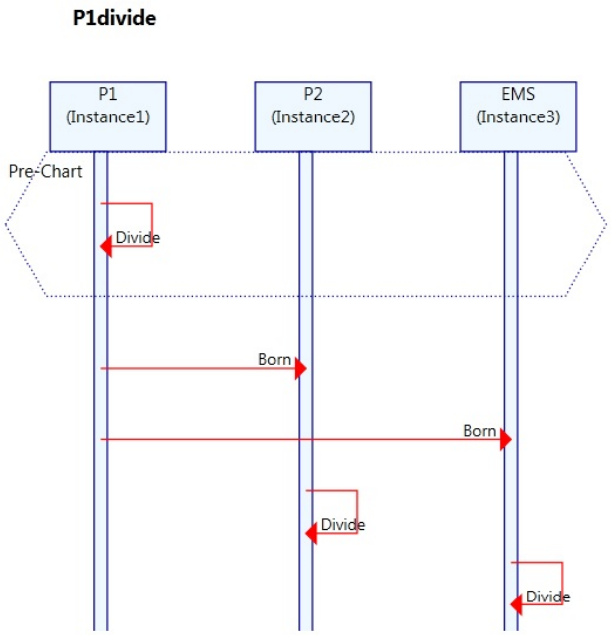


Fig. 16. Cell P1 division in *C. elegans*

## ACKNOWLEDGMENT

The author would like to thank Cory Plock for a long term fruitful collaboration on scenario-based modeling and for his dedicated work in developing the Biocharts framework.

## REFERENCES

[1] E.S. Lander et al., "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[2] J.C. Venter et al., "The sequence of the human genome," *Science Signaling*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[3] P. Nurse, "Life, logic and information," *Nature*, vol. 454, no. 7203, pp. 424–426, 2008.

[4] P. Heymans and E. Dubois, "Scenario-based techniques for supporting the elaboration and the validation of formal requirements," *Requirements Engineering Journal*, vol. 3, pp. 202–218, 1998, springer-Verlag.

[5] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *22nd International Conference on Software Engineering (ICSE 2000)*. ACM Press, 2000, pp. 314–323.

[6] M. Lettrari and J. Klose, "Scenario-based monitoring and testing of real-time UML models," in *4th Int. Conf. on the Unified Modeling Language, Toronto*, October 2001.

[7] S. Leue and T. Systä, "Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers," in *Scenarios: Models, Transformations and Tools*, ser. LNCS, vol. 3466. Springer-Verlag, 2005.

[8] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios," *ACM Trans. Software Engin. Methods*, vol. 13, no. 1, pp. 37–85, 2004.

[9] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, ser. NATO ASI Series, K. R. Apt, Ed., vol. F-13. New York: Springer-Verlag, 1985, pp. 477–498.

[10] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[11] H. Kugler, C. Plock, and A. Roberts, "Synthesizing Biological Theories," in *Proc. 23$^{rd}$ Int. Conf. on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer-Verlag, 2011, pp. 579–584.

[12] C. Plock, "Synthesizing executable programs from requirements," Ph.D. dissertation, New York Univ., 2008.

[13] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Trans. Software Engin. Methods*, vol. 5, no. 4, pp. 293–333, 1996.

[14] D. Harel and H. Kugler, "The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML)," in *Integration of Software Specification Techniques for Application in Engineering*, ser. LNCS, vol. 3147. Springer-Verlag, 2004, pp. 325–354.

[15] H. Kugler, "Biocharts Project Website," 2013, `http://research.microsoft.com/Biocharts/`.

[16] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987, (Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.).

[17] H. Kugler, A. Larjo, and D. Harel, "Biocharts: A Visual Formalism for Complex Biological Systems," *J. R. Soc. Interface*, vol. 7, no. 48, pp. 1015–1024, 2010.

[18] D. Harel and H. Kugler, "Some Thoughts on the Semantics of Biocharts," in *Time for Verification*, ser. LNCS, Z. Manna and D. Peled, Eds., vol. 6200. Springer-Verlag, 2010, pp. 185–194.

[19] D. Harel and E. Gery, "Executable object modeling with statecharts," *Computer*, vol. 30, no. 7, pp. 31–42, July 1997, also in *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, March, 1996, pp. 246–257.

[20] T. Howard, S. Middelhaufe, K. Moore, C. Edner, D. Kolak, G. Taylor, D. Parker, R. Lee, N. Smirnoff, S. Aves, and J. Love, "Synthesis of customized petroleum-replica fuel molecules by targeted modification of free fatty acid pools in Escherichia coli," *Proceedings of the National Academy of Sciences*, vol. 110, no. 19, pp. 7636–7641, 2013.

[21] G. Wadhams and J. Armitage, "Making sense of it all: bacterial chemotaxis," *Nature Reviews Molecular Cell Biology*, vol. 12, no. 5, pp. 1024–1037, 2004.

[22] C. Morton-Firth, T. Shimizu, and D. Bray, "A Free-energy-based Stochastic Simulation of the Tar Receptor Complex," *J Mol Biol.*, vol. 286, pp. 1059–1074, 1999.

[23] A. Feist, C. Henry, J. Reed, M. Krummenacker, A. Joyce, P. Karp, L. Broadbelt, V. Hatzimanikatis, and B. Palsson, "A genome-scale metabolic reconstruction for Escherichia coli K-12 MG1655 that accounts for 1260 ORFs and thermodynamic information," *Molecular Systems Biology*, vol. 3, no. 121, 2007.

[24] E.J.A. Hubbard, "*Caenorhabditis elegans* germ line: a model for stem cell biology," *Dev. Dyn.*, vol. 236, no. 5, pp. 3343–3357, 2007.

[25] P. Nurse, P. Thuriaux, and K. Nasmyth, "Genetic control of the cell division cycle in the fission yeast Schizosaccharomyces pombe," *Molecular and General Genetics*, vol. 146, no. 2, pp. 167–178, 1976.

[26] Y. Setty, D. Dalfo, D. Korta, E.J.A. Hubbard, and H. Kugler, "A model of stem cell population dynamics: in-silico analysis and in-vivo validation," *Development*, vol. 139, no. 1, pp. 47–56, 2012.

[27] D. Hansen, L. Wilson-Berry, T. Dang, and T. Schedl, "Control of the proliferation versus meiotic development decision in the c. elegans germline through regulation of gld-1 protein accumulation," *Development*, vol. 131, no. 1, pp. 93–104, 2004.

[28] R. Williams, "Network 3d," 2012, http://research.microsoft.com/en-us/um/cambridge/groups/science/tools/network3d/network3d.htm.

[29] Z. Bao, J. Murray, T. Boyle, S. Ooi, M. Sandel, and R. Waterston, "Automated cell lineage tracing in Caenorhabditis elegans," *Proceedings of the National Academy of Sciences*, vol. 103, no. 8, pp. 2707–2712, 2006.

[30] S. Brenner, "The genetics of Caenorhabditis elegans," *Genetics*, vol. 77, pp. 71–94, 1974.

[31] J. Sulston and H. Horvitz, "Post-embryonic cell lineages of the nematode, Caenorhabditis elegans," *Developmental Biology*, vol. 56, no. 1, pp. 110–156, 1977.

[32] N. Kam, H. Kugler, R. Marelly, L. Appleby, J. Fisher, A. Pnueli, D. Harel, M. Stern, and E.J.A. Hubbard, "A scenario-based approach to modeling development: A prototype model of C. elegans vulval fate specification," *Developmental Biology*, vol. 323, no. 1, pp. 1–5, 2008.