Morphogenesis as an Amorphous Computation

Arnab Bhattacharyya
Project on Mathematics and Computation
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139
abhatt@mit.edu

ABSTRACT

In this paper, we present a programming language viewpoint for morphogenesis, the process of shape formation during embryological development. Specifically, we model morphogenesis as a self-organizing, self-repairing amorphous computation and describe a framework through which we can program large-scale shape formation by giving local instructions to cell-like objects. Then, using this programmatic perspective, we specify some example developmental processes and discuss the characteristics that make them suitable candidates for evolutionary variation and selection. Consistent with the theory of facilitated variation from evolutionary biology, we find that variation in developmental processes can be introduced and conserved due to the hierarchical organization of growth specification.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: Simulation Languages, Model Development; I.6.8 [**Simulation and Modeling**]: Types of Simulation–*Discrete event*

General Terms

Experimentation, Languages

Keywords

Amorphous Computing, Morphogenesis, Emergent Order

1. INTRODUCTION

Embryological development is a magnificent demonstration of how complexity can arise from initial simplicity. A single egg cell contains most of the information needed to position the millions of cells in a human body; moreover, the construction process is remarkably robust in that it can recover from a large number of cell deaths and malfunctions. Even after the initial construction is completed, the living system remains in a dynamic equilibrium, constantly replacing dead cells, healing tiny ruptures, and so on. Moreover, in some organisms, even large-scale regeneration occurs. Urodeles, like newts, are famous for regrowing entire limbs in such a way that the regenerated parts merge seamlessly into the preexisting tissue.

From the perspective of a computer scientist, studying such processes could bring novel insights into solving other problems, such as programming "smart matter" or directing a swarm of robots. But, even leaving aside these benefits, studying morphogenesis from a computational perspective is important for its own sake: expressing ideas computationally through a programming language forces disambiguation of ideas and provides a powerful tool for examining implications of new hypotheses. A good language in which to express morphogenetic processes will be an invaluable tool to verify plausibility of formal models and to discard inconsistent conjectures. Thus, in the following, the goal is to identify an effective computational framework and a set of morphogenetic primitives through which we can simulate interesting developmental processes.

In section 2, we describe a framework that enables programming of global topological changes through local cell-level instructions. We enumerate a set of basic developmental patterns that performs the role of primitive operations in our language for morphogenesis. Then, in section 3 we present three example developmental mechanisms, reminiscent of ones occurring in nature, and show how they can be implemented in our framework. Finally, we discuss the fact that each of these mechanisms possesses a lot of potential for variation and adaptability and how we can unleash these variations by making small changes to our programs.

2. A COMPUTATIONAL FRAMEWORK FOR MORPHOGENESIS

2.1 Requirements

The central challenge in modeling biology is managing the huge amount of concurrent, largely decentralized, behavior exhibited by the cells. How do you program a massively distributed, noisy system of components? This problem is formalized in the study of *amorphous computing*. An amorphous computing medium, as introduced by Abelson et al. in [1], is a system of tiny, computationally-limited elements scattered irregularly across a surface or a volume. These elements, which we shall call *organizers*, have a limited range of communication[1], can retain some local state, and can repli-

[1]It might be interesting to apply the framework of [5] to

CF'06, May 3–5, 2006, Ischia, Italy.

cate and kill themselves. The goal of the amorphous system is to attain some desired global state. So, while the organizers are only locally interacting, their collective behavior results in complex global behavior. Moreover, we want the amorphous computation to be robust; that is, the computation should still proceed in the face of random organizer death/failure.

Here, we apply the above computational paradigm to the creation of shapes from amorphous elements. Our goal, as stated above in the Introduction, is to be able to model the essential elements of biological developmental processes with amorphous programs. The amorphous scheme should have the following properties:

- an initially small group of organizers should replicate repeatedly until a desired shape is attained
- organizers should be allowed to move during the course of development
- the shape development should be robust in the face of organizer death and malfunction

However, notice that a trivial development scheme that contains three gradient fields to provide a global coordinate system in a three-dimensional world and that deterministically instructs each organizer to grow another organizer in a particular direction based on the coordinate system can satisfy our criteria. Clearly, there is some other aspect of biological development that is not captured by the above criteria. This, we discuss next.

2.2 Variation and Exploratory Behavior in Morphogenesis

Multicellular organisms have a small number of genes (only about 25,000 in humans) and a large fraction of these genes is conserved across species. In this context, the sheer variety of organisms in terms of their shapes is extraordinary. How can the relatively small differences between the morphogenetic programs of organisms create so much diversity of anatomy and physiology? Kirschner and Gerhart, in their book, *The Plausibility of Life* [6], show that much of the diversity in form arises from the exploratory nature of the developmental processes themselves. The argument is that the morphogenetic program prescribe the means to develop, not the outcome of the development.

Kirschner and Gerhart's book provides a wonderful example to illustrate the point: the role of microtubules in giving shape to individual cells. In a typical cell, long thin filaments called microtubules grow from the nucleation center to the cell membrane providing stability to the cell structure. However, these microtubules are not static; instead, each filament grows out for a short period and then shrinks back toward its point of origin, getting spontaneously replaced with a new one which grows in another random direction. This normal lifecycle of the filament is interrupted if a stabilizing signal is sensed at the cell membrane. If the stabilizing signal is sensed, the microtubule persists and does not depolymerize. Hence, by this adaptation process, the microtubules automatically cluster themselves toward the stabilizing region of the cell membrane, giving the cell a particular shape. The key insight is that for the cell's shape to change, the growing mechanism of the microtubule need not change; only the location of the stabilizing signal has to change. Many different cell shapes can be generated from exactly the same exploratory process of microtubule assembly under different stabilizing agents acting peripherally.

This observation is crucial to explaining the diversity of morphological forms in nature. A single genotypic change can create different morphological changes in many parts of the organism, not because the other parts also simultaneously and improbably experience the right genetic mutations, but because these parts are created by adaptive exploratory developmental processes that undergo selection. A characteristic of developmental mechanisms in Nature is that they contain the potential for rapid variation. We should require the same for the mechanisms that are expressed in our programs. More generally, Kirschner and Gerhart advocate a theory of facilitated variation to explain how variation is introduced and conserved in evolution. In the next section, we will show how our programming constructs correspond to the tenets of this theory.

2.3 Programmable Components in the Framework

2.3.1 The Universe

In order to be able to support shape development, the universe in which the organizers are embedded must have some geometric structure. We restrict our universe to be a bounded Euclidean space. Although all the simulations described here will be in a two-dimensional space, the same programs also work in a three-dimensional universe. Note that the universe is not discretized in any sense.

In addition to being the substrate in which the organizers are embedded, the universe also serves as the communication medium. There are two distinct types of signals that can be transmitted through this medium. The first type is morphogenic signals, short-range signals emitted by organizers. A *morphogen* is a signal secreted by a local source that decreases in concentration[2] as distance from the source grows ([8, 10]). In our context, the decrease in concentration of the morphogen is implemented very simply; when an organizer centered at position $\mathbf{x}$ emits such a signal at concentration v, then the concentration at the surrounding points is given by the following scalar field f:

$$f(\mathbf{z}) = \begin{cases} v & \text{if } |\mathbf{z}-\mathbf{x}| \leq 2d \\ 0 & \text{otherwise} \end{cases}$$

where d is the organizer diameter. As described in section 2.3.2, organizers can respond to a positive concentration of ambient signal. A morphogenic gradient can be established if organizers sensing a concentration v of a signal itself start secreting the same morphogen at a lower concentration. Such gradient fields enable differentiation and regional specification of embryonic cells and will do the same in our framework.

The second type of signaling the underlying medium supports is background fields. A background field is a static field that can be present at all points in the universe, providing global positional information to each organizer. The

formalize and reason about collective knowledge in such a setting.

[2] "Concentration" here simply means a measure of the signal strength. One can think of morphogens coming in packets; then the concentration of a morphogen is the size of its packet.

background fields represent the input from the environment in which the organizers grow. For example, cell polarization occurs during development in response to interactions with the extracellular matrix. Background fields are also similar to the Bicoid morphogenic field in *Drosophila* which establishes the fruit-fly embryo's anterior-posterior axis at the earliest stages.

2.3.2 Organizers

In our model, organizers are hard spheres with a fixed diameter d. For an organizer, all sense of directionality and position is based on the ambient background fields and the morphogenic signals received. For instance, an organizer can specify that it replicate only along the direction of increasing concentration of a background signal. In this case, when the organizer replicates (by the process described in 2.3.3), it samples a large number of random directions and places its descendant in the direction of the greatest concentration increase. Such a mechanism enables growth patterns such as the one shown in Figure 1. Here a background signal is secreted by a source near the middle of the rightmost edge of the universe. The organizers grow toward it, going around the gray obstacle because there is no background field inside the obstacle.

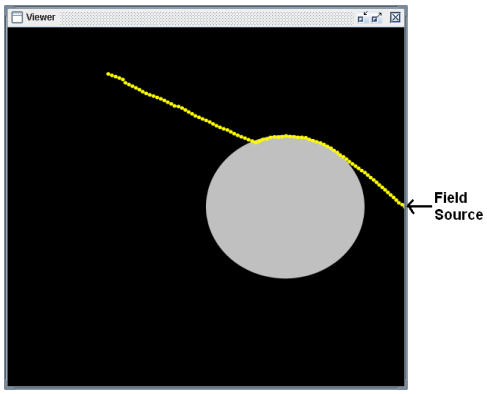


Figure 1: Organizers growing around obstacle towards the indicated source.

Each organizer has receptors to sense local morphogenic fields. The morphogen signals are permissive, that is, the complete response to the signals are already built into the organizers. The signal concentrations themselves do not encode instructions for the cell's response, which would be the case if the signals were instructive. Thus, from an evolutionary standpoint, it is easy for cell behavior to change because it is possible for some cells to repress their reaction to the signal without affecting the other cells' responses. If the signals were instructive, changing the cellular behavior would involve changing the instructive signals which could potentially affect all receiving cells. The use of permissive signaling to effect easily mutable regulatory connections is termed *weak regulatory linkage* in [6] and is one of the key tenets in the theory of facilitated variation.

To make weak regulatory linkage work, it is crucial that all the organizers are *not* identically programmed. In an adult vertebrate, there are about three hundred differentiated cell types, each with several subtypes. Moreover, during development, there exist many more types of cells that are transient but are required for proper morphogenesis. Each of these cell types has different responses to morphogens, different adhesivities, different lifetimes, different clock speeds, and so on. Such categorization of cells into different types provides evolutionary advantage. It allows each type of cell to evolve independently and parallelly. Thus, it avoids the pleiotropy problem where positive changes in cells in one region of the organism result in negative changes for cells in another different region. There exists strong biochemical evidence to support the existence of such a design. In the 1990s, biochemists discovered that during the phylotypic stage of development, the embryo divides itself into compartments, each of which responds in a particular way to signaling morphogens and has different conserved core processes. Remarkably, many features of the compartment map are common to a wide range of phyla; this conservation suggests that compartments provide a flexible scaffold on which to build the rest of the organism's anatomy.

To implement these concepts in our programmatic framework, we use a static type system. We impose a type environment in which each organizer has a type, and organizers of the same type share the same developmental program. So, the collection of organizers of the same type may be thought to form a compartment. Organizers are called so because it is their job to organize the growth of the compartment they belong to. Usually organizers of the same type form a physically contiguous region, and if some of the organizers in a compartment die, the rest can usually quickly regrow the missing region. Subtyping is used to implement compartment subtypes.

2.3.3 Growth and Regeneration

In our setting, growth occurs through a process functionally similar to directed asymmetric mitosis ([9]). A mother organizer checks if it is possible to grow in some direction (that is specified with respect to the background field gradients or the direction of incoming morphogenic signals) and if so, generates a daughter organizer in that direction. True to our goals of regeneration, an organizer usually never explicitly records the fact that it has already grown in a direction; instead, it continually keeps searching for a direction in which it can grow. Thus, in the event that the daughter organizer dies (and growth of the mother organizer is not repressed), the mother organizer can regenerate the missing part.

An important issue in this regard is the interaction between growth and organizer type. Can an organizer generate organizers only of a specific type or can it grow organizers of many types? Our answer is that all organizers are *pluripotent*, that is, capable of growing any type of organizer in the system. But each organizer is usually inhibited from growing a large fraction of the types. Change in inhibitions can be instigated from various causes such as sensing a different concentration of a morphogen or sensing of a different background field gradient[3] or aging.

More explicitly, our programmatic framework has the notion of a *grower*. A grower is a program embedded inside an organizer which, when invoked, generates a daughter organizer of a specific type. Pluripotence means that all organizers in the amorphous system have access to the same fixed set of growers. But an organizer typically does not ex-

[3]Although the background fields are unchanged in time, the organizers might move.

press all its growing capabilities. Each organizer can inhibit or activate some of its growers and can put a priority ordering on its activated growers. Also, among the activated growers, some may not lead to a new daughter because of environmental conditions such as if there is no space to grow in a particular direction; let us call growers that do lead to new growth *potent*. On each time-step, the organizer finds the most preferred activated potent grower and, if there exists one, invokes that grower to create a daughter organizer. Note that the decoupling between growers and organizers is a fundamental part of the design of our framework. Given the right environment, any organizer can produce any other type of organizer.

Such a scheme is consistent with our goal to facilitate the exploration described in section 2.2. A large amount of variation can be introduced without touching the main framework of growth described above at all. Changes of concentration of morphogen secretions can inhibit or activate growers in different organizers. More modularly, a particular type of organizer can change the criteria according to which growers are inhibited. For example, an organizer type can lower the age at which it begins to stop inhibiting a grower, or it can increase the morphogen concentration required for inhibiting a grower to such a level that the grower is never inhibited. Such variations are suitable targets for evolutionary pressure.

2.3.4 Death

An important part of development is cell death or apoptosis. For example, during the formation of the digits, a hand plate initially forms and, then, the interdigital mesenchyme cells die. As described by Bard in [2], cell death is especially responsible for sculpting the finer details of tissue organization. In our framework, apoptosis is implemented very simply. An organizer can kill itself at any point. The space in the universe previously occupied by the organizer is freed. Also, in the simulation, the computer frees all resources related to the dead organizer.

A common problem that arises when implementing apoptosis is that organizers surrounding the dead ones often try to grow to reclaim the vacated space. One way to prevent this is to have dying organizers release a signal that sterilizes the live organizers bordering the empty space.

2.3.5 Movement

Cellular mobility is a crucial ingredient in development, especially in vertebrate development where the migration of neural crest cells results in the formation of a large number of tissues. Also, from an evolutionary standpoint, the process of migration is particularly conducive to exploratory behavior, since changes in the cellular environment change the locations where migrating cells choose to settle. It is worth noting that all cells have the molecular apparatus to move, although few do during development because most are inhibited by the physical contact with other cells ([2]).

What prevents most cells from moving is the adhesive environment of their neighbors. At the same time, what causes movement of most cells is adhesion to other moving cells. Thus, cell mobility is a competitive balance between the motile forces exerted by the cell's cytoskeleton and the adhesive forces exerted by the cell's neighbors. A more formal analysis is presented in Chapter 5 of [2]. In our framework, we simplify the situation by requiring that any motile force stronger than usual overrides the adhesive forces. So, an organizer can, at will, move in any direction, provided that the destination is not occupied. Moreover, any organizer adhering to the moving organizer will also feel additional motile force, causing it to move in the same direction in synchrony with the original moving organizer or until it is blocked. Thus, mass movement of organizers can be initiated and controlled through a process akin to chemotaxis, the biological mechanism by which cells move relative to a background field gradient.

The adhesivity of an organizer is dynamically maintained and is not fixed across a compartment. Also, properly speaking, each organizer does not have a single adhesivity property; instead, an organizer adheres differently to different types of organizers. When an organizer is attracted to two moving organizers of different types, its movement will be biased in the direction of the organizer it more strongly adheres to.

2.4 The Simulator

The execution model for each organizer is simple. At each time-step, the organizer's transfer function is called. The transfer function is responsible for carrying out the following actions:

- Increase age
- Refresh background field value measurements
- Receive morphogens
- Based on the results of the above actions, do if applicable:
  - Emit morphogens
  - Inhibit or activate growers
  - Invoke the most preferred active potent grower
  - Die

The organizer's age is a local clock count maintained at each organizer[4]. The computer simulator asynchronously executes the organizers' transfer functions. The simulator has six threads and each one cycles through 70% of the organizers in random order. Priority is given to organizers in whose neighborhood growth or death has most recently taken place. Simulation of development is accelerated with this "most-recently-grown" caching strategy. The primary benefit of using such a nondeterministic execution strategy is that it forces the morphogenetic scheme to be independent of the exact order in which the transfer functions of the organizers are executed.

2.5 An Implementation of the Framework

We implemented the above framework in the JAVA programming language. The language to express morphogenetic schemes is described briefly here, since we will use it to specify some developmental processes in the subsequent section.

[4]The rate at which the organizer ages might vary by compartment. Intracellular clock mechanisms are well supported by experiments; in several cases, cells removed from their normal environment before the initiation of morphogenesis undergo *in vitro* and roughly on schedule some of the changes that accompany the initiation of morphogenesis *in vivo* ([2]).

In the JAVA implementation, organizer types are implemented as classes, and particular organizers as instantiations of these classes. To create a new organizer type, the programmer has to subclass the `Organizer` class and provide an implementation of the transfer function shared by all organizers of that type. For example, Figure 2 shows the description of a simple organizer. Organizers of type *StdCell* have receptors for two morphogens, *sterile* and *poison*; this is specified by the call to `registerReceptor` [5]. When a *StdCell* organizer detects a concentration level[6] of *poison* that is between 1 and ∞, it kills itself. Notice how this is implemented: the morphogen does not carry any special instruction causing the organizer to die; instead, the organizer has a thunk that is a built-in response to *poison* that is activated upon the morphogen's receipt. Similarly, receipt of the *sterile* morphogen causes all growers to be inhibited. Thus, the morphogens form a weak regulatory system, as discussed earlier.

A programmer can implement a grower in this framework by subclassing the abstract class `Grower`. All concrete subclasses of `Grower` must implement the `grow` method which returns a new daughter organizer and its orientation relative to a parent organizer. Figure 3 shows the implementation of a simple grower, named *StdGrower*. When an organizer invokes a *StdGrower*, the grower chooses a random direction and if there is not already another organizer adjacent in that direction, it grows a new daughter *StdCell* organizer in that direction. The orientation is specified as a `Vector` object. (The global coordinates used inside a `Vector` object are invisible to programmer-created growers and organizers. So, access to a `Vector` does not imply access to a global coordinate system.)

By default, an organizer inhibits all its growers. So, in order to grow, an organizer must explicitly activate its growers, either in the constructor or inside thunks that are executed upon receipt of certain morphogens or detection of some level of background fields or attainment of some age or some combination of these events. Also, as described in section 2.3.3, an organizer must put a priority ordering on the growers. Figure 4 shows how these are done in the context of a specific example, a *FillerCell*. During construction of a *FillerCell*, the *StdGrower* is registered, meaning that *StdGrower* is higher than any other grower in the priority ordering for this organizer. If a second grower were to be registered after *StdGrower*, it would have lower priority than the *StdGrower* but higher priority than any other grower. In other words, the lexical order of the `registerGrower` statements determines the priority ordering; thus, evolution can select for the best ordering by permuting these statements in the program text. The *FillerCell* organizer activates its grower when it reaches the age of 10. So, if we start with a single *FillerCell* organizer, eventually its descendants will fill the entire universe.

The examples above have not illustrated some other important features of the general framework, such as morphogen broadcasting, background field measurements, cell movement, and adhesivity. But the grammar for expressing these concepts in our implementation is easy to understand, given the previous discussion and examples.

[5]The second argument to `registerReceptor` specifies whether the organizer should keep track of the maximum concentration of the morphogen received by any of its receptors or the minimum.

[6]In this implementation, concentration levels are always nonnegative integers.

3. FACILITATED VARIATION

In this section, we describe what it means for morphogenetic schemes to be "evolutionarily feasible." An evolutionarily feasible developmental scheme has two characteristics: robustness and variation. Many mechanisms in nature are intrinsically very robust to external change. The microtubule example in section 2.2 illustrates this. Another example is the growth of capillaries in mammals. Capillaries seem to branch out randomly to explore their surroundings, and their rate of branching increases when they reach oxygen-starved tissue and decreases when they reach tissue with enough available oxygen. Thus, the capillaries automatically adapt to any change in the external tissue morphology. If the capillary branching system was not adaptive, then a change to the tissue structure would need to be accompanied by changes to the capillary branching mechanism for the organism to remain viable. But instead, evolution is a more conservative process: the system remains functional even while the individual components change. This is why genotypic changes can be conserved through evolution. But if changes to the genotype are restricted to those which keep the system functional, how does all the phenotypic diversity in life arise? For example, it is conjectured that the protostomes had a deuterostomic ancestor. But protostomic and deuterostomic embryogenesis are so very different; how did the evolution from deuterostomes to protostomes occur in a "homotopic" fashion? The answer is that developmental mechanisms are designed so that a small genotypic change leads to a large phenotypic change. And this genotypic change can often be conserved because the unchanged mechanisms are robust enough to retain the organism's viability. Thus, developmental processes in nature intrinsically facilitate evolution by making individual components very susceptible to variation while requiring that they adapt to variations in other components.

We would like the developmental processes we create to similarly facilitate evolution. Below we sample a few types of morphogenetic processes, expressed using our framework from section 2, that exhibit robustness and variation.

3.1 From a circle to an ellipse

Here is a very simple scheme for developing a solid sphere. The morphogenesis starts off with a single *starter* organizer which emits a morphogen called a at a fixed concentration level. The *starter* organizer continually produces organizers of type *ball* uniformly in all directions through the *StdGrower* grower from Figure 3. Each *ball* organizer records max, the maximum concentration of a that it receives, and itself emits a at concentration $max - 1$. Also, each *ball* organizer uses the *StdGrower* grower to produce more of its own type in random directions, unless it detects that the concentration of a is below some threshold in which case, the grower is inhibited. Such a mechanism creates a solid sphere, because the growing process is completely unbiased in any direction. For convenience, suppose the single *starter* organizer is generated from another organizer of type *precursor*. The *precursor* organizer generates the single *starter* organizer, which moves randomly for some time and then starts creating the ball. Development starting from such a setup is shown in Figure 5.

```
import framework.Organizer;

public class StdCell extends Organizer {

   public StdCell()     {
       registerReceptor("sterile",true);  // track maximum concentration of sterile
                                          // among all of the receptors
       registerReceptor("poison",true);

       addMorphogenAction("sterile",1,Double.MAX_VALUE,
                          new Runnable() {
                              public void run() {
                                  inhibitAllGrowers();  // do this when sterile conc. ≥ 1
                              }
                          });

       addMorphogenAction("poison",1,Double.MAX_VALUE,
                          new Runnable() {
                              public void run() {
                                  die();
                              }
                          });
   }
}
```

Figure 2: Description of a very simple organizer

```
import framework.Grower;

public class StdGrower extends Grower {

    public OrganizerVectorPair grow(Organizer org, Universe uni) {
        Vector gdir = uni.getRandomDirection(org);

        // grow StdCell if not blocked
        if(!uni.isOccupied(org,gdir,Organizer.class)) {
            return new OrganizerVectorPair(new StdCell(), gdir);
        }

        return null;
    }
}
```

Figure 3: Description of a simple grower that grows a new *StdCell* in a random direction

```
public class FillerCell extends StdCell {

   public FillerCell()     {
     super();

     registerGrower(new StdGrower(),true);
     addAgeAction(10, Integer.MAX_VALUE,
                          new Runnable() {
                              public void run() {  // do this when age ≥ 10
                                  activateGrower(StdGrower.class);
                              }
                          });
   }
}
```

Figure 4: Description of a growing organizer

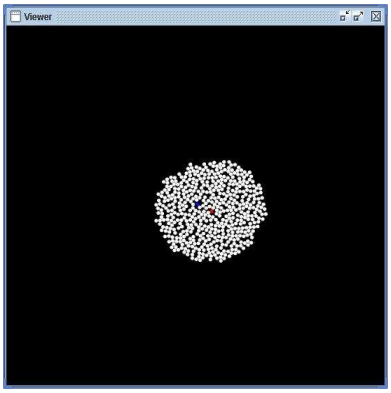


Figure 5: A solid sphere. The *starter* organizer at the center of the ball is colored red.

Next, consider what happens when the *precursor* organizer generates two of the *starter* organizers. Such a setup is shown in Figure 6(a). Even when the balls generated from the two *starter* cells overlap, the balls are maintained independently. Each *ball* organizer effectively measures the distance to the closest of the two *starter* organizers (by observing the local concentration of a) and decides whether to grow or not based upon this information. Although a little trivial in this case, one should note that the entire structure of the final shape has been changed without changing the code of the *starter* or *ball* organizers. Also, the change in the description of the *precursor* organizer is syntactically very small: the grower generating the *starter* cell is invoked twice instead of once.

Next, consider what happens when the two *starter* organizers are not exactly identical. In particular, suppose that one emits morphogen a and the other emits a different morphogen b. Also, instead of having the *ball* cells inhibit their growers if the local concentration of a is below some threshold, suppose that it is the case that the *ball* cells inhibit their growers only if the sum of the local concentration of the a morphogen and the local concentration of the b morphogen is below some threshold[7]. (In fact, this could have been the case in the earlier examples also. In those cases, the concentration of b was always zero.) This developmental scheme now specifies an ellipse, as shown in Figure 6(b). Appendix A shows the code (in terms of the JAVA implementation of section 2.5) for these examples. As annotated there, figures 5, 6(a), and 6(b) all come about through changes in the last line of the description of the *precursor* organizer. Duplication followed by speciation of the grower activated by the *precursor* organizer leads to a change from a circle to an ellipse in the final shape.

3.2 Imitating microtubule growth

The example in this section is reminiscent of the cytoskeleton assembly process described in section 2.2. Because there are no subcellular structures in our framework, we will amplify the entire process to the tissue level: our desired structure consists of a hollow ball of organizers that is supported by filaments growing from a structure at the center of the hollow ball to the surface. Each filament consists of a chain of organizers that decays away unless a stabilizing signal is received. This example will show some of the expressiveness and robustness inherent in the hierarchical compartmentalization of organizers.

Here is a possible developmental mechanism for growing such a structure. The morphogenesis starts off with a single *initiator* organizer. The *initiator* replicates repeatedly to form a solid ball (via the mechanism described above in section 3.1), which we will call the nucleation center. Next, these organizers in the nucleation center replicate (via another invocation of *StdGrower*) to form a solid ball concentric with the nucleation center, as shown in Figure 7(a). Some of the organizers in the larger sphere die, such that we get a shell with the nucleation center in the middle, as in Figure 7(b). Meanwhile, the organizers in the nucleation center have begun generating filaments that extend from the center in random directions. The way a straight line of organizers can be grown is by invoking a special grower, described in Figure 8, that grows a daughter organizer in the direction opposite to the direction of any adjacent organizer. The filament organizers are intrinsically unstable; they die after a short time (as measured by their internal clocks). However, some special organizers on the shell can emit a stabilizing signal that prevents the filament organizers from dying. In this way, the filaments dynamically cluster toward the side of the shell that is stabilizing, as shown in Figure 7(c).

Although in an artificial context, this example illustrates the robustness of morphogenesis. The organizers emitting the stabilizing signal could be involved in some other complicated morphogenetic process and their location could vary randomly but the filaments would still cluster towards them. No change in the code for describing the filament organizers is needed. A second important point is that the organizers in the nucleation center essentially use randomness in order to locate the stabilizing region of the shell. This is an example of the "exploratory growth" notion in [6].

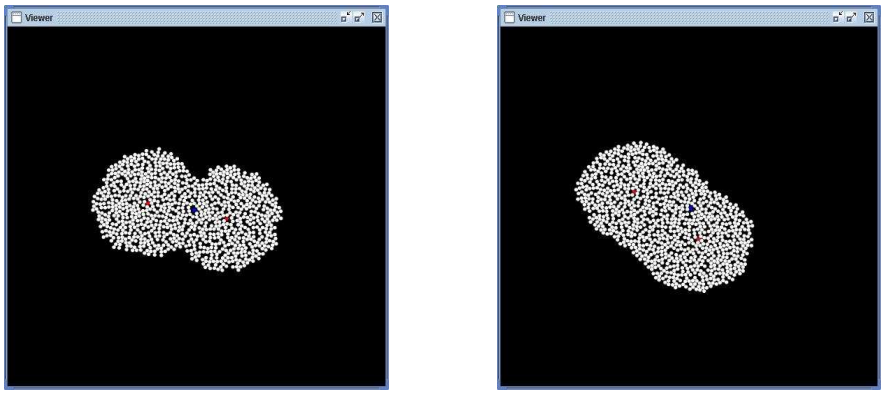
3.3 Imitating neural crest cell migration

Until now, none of the examples has significantly involved cell movement[8]. But, as mentioned before, cell migration is an important part of development, especially in vertebrates where the neural crest cell (NCC) migration is responsible for structures ranging from horns to hoofs. It is especially interesting that variation in the outcome of NCC migration is a major source of phenotypic variation among the vertebrates. The result of NCC migration is specified by the adhesivities and type of the surrounding cells. We will model a formal analog of NCC migration to understand the evolutionary feasibility of such a scheme.

Consider the following developmental mechanism. The morphogenesis starts (as usual) with a single organizer of type *initiator*. The *initiator* organizer invokes the *StdGrower* to generate a ball of organizers of type *ball*. The *ball* organizers in the center undergo apostosis, so that a shell of *ball* organizers is left. There exists an organizer type *ncc*, a subtype of the *ball* type, to which belongs some of the organizers lying on the inner surface of the shell. These *ncc*

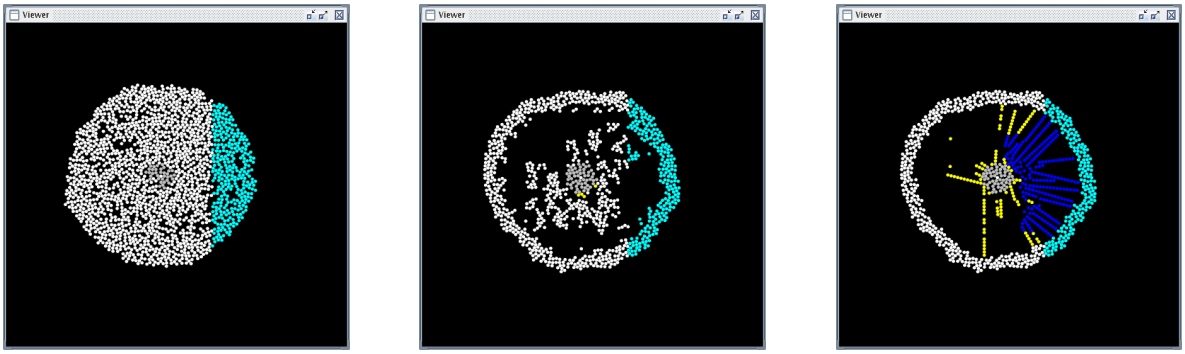
[7]A physical motivation for considering the sum might be that a and b are indistinguishable inside the organizer although they are distinguishable by the receptors

[8]Note that without movement, it is mostly easy to regenerate missing regions in our framework. But when cells can move, regeneration becomes a very difficult goal to ensure in general. We have not tried to tackle this problem here.



(a) (b)

Figure 6: (a) shows growth of two solid spheres. In this case, both *starter* organizers are emitting the same morphogen. (b) shows growth of an ellipse. Here, the *starter* organizers, present at the foci of the ellipse, are emitting different morphogens. In both parts, the *starter* organizers are colored red while the *precursor* is colored blue. Note that all these structures are regenerative; the shape automatically reforms if some of the organizers die.



(a) (b) (c)

Figure 7: Simulation of the microtubule development mechanism. There is a background field which increases uniformly from left to right. Organizers on the shell which sense the background field value to be greater than some threshold emit the stabilizing signal; these are colored cyan above. The stable filament organizers are colored blue and the unstable ones yellow. The grey region is the nucleation center.

```
public class MicrotubuleGrower extends Grower {

    public OrganizerVectorPair grow(Organizer parent, Universe uni) {

        // get direction of the parent relative to sender of
        // the m morphogen
        Vector gdir = parent.currentReception("m").dir;

        if(!uni.isOccupied(parent,gdir,Organizer.class))   {

            // grow a new filament organizer in the direction
            // opposite to the receiving direction
            return new OrganizerVectorPair(
                                        new FilamentCell(),
                                        gdir);
        }

        return null;
    }
}
```

Figure 8: Grower for growing a line of filament organizers.

organizers now start taking random walks biased towards the radially inwards direction. An *ncc* organizer can stop moving due to one of two reasons: (i) It come into contact with an organizer that is very adhesive to it, (ii) It comes into contact with a physical boundary. (Both these cases are well-motivated biologically. See Chapter 5 in [2].) As the *ncc* organizer stops, it releases a morphogen that depends upon the local concentration of other morphogens and background fields. Thus, effectively, the *ncc* differentiates differently based upon where it lands[9]. Finally, the specialized morphogen interacts differently with the surrounding organizers to activate or inhibit specialized growers.

It is evident that such a scheme is a rich source of variation. The differentiation of the *ncc* organizers can change, the adhesivity of the environment can change, the activation/inhibition of growers due to the morphogens released by the *ncc* organizers can change, and so on. The more intriguing question is robustness. Because the *ncc* organizers are taking random walks, the order in which the *ncc* cells differentiate is not unique and could lead to different growth patterns from the same genotype. Here, evolution must have selected the most robust, viable set of growers, complementary to the way robustness helps conserve variation.

4. CONCLUSION

4.1 Previous Works and Contribution

There has been some important work in the past to understand development from the perspective of amorphous computation. Two of the most important ones are [4], Coore's Ph.D. thesis on specifying spatial patterns using a developmental language, and [7], Kondac's work on biologically-inspired self-assembly of two-dimensional shapes. Also relevant is [3], the work by Nagpal and Clement on extending Coore's work to regenerative spatial patterns. The endeavor here is to be able to convert a globally-specified output specification into a local program that can be executed in a distributed manner by cells in an amorphous medium. [4] and [3] develop this paradigm for spatial patterns, while [7] develops it for replicating cells.

The main contribution in this paper is a hierarchical organization of growth specification. With a hierarchy, code reuse becomes possible and it becomes possible to understand that growth leading to dramatically different structures might arise from the same grower under different environmental conditions. In general, the growing programs become more modular and robust, and regeneration almost comes automatically for many morphogenetic schemes expressed in our framework. Another fundamental difference between the previous works cited and the present paper is that we are trying to understand biology from a computational perspective instead of applying biological ideas to create algorithms. So, it becomes more important to study actual biological experiments and think about how they fit into the given framework.

4.2 Future Directions

The present study can be extended in many different ways. Here are a few:

- Present evolution as an explicit source-to-source transform on the organizer type descriptions. Then, it might be possible to understand some intriguing evolutionary selections and test taxonomical hypotheses.
- Test conjectures of morphogenetic mechanisms. For example, we are considering a possible morphogenetic mechanism that might explain why protostomes and deuterostomes are closely related evolutionarily, while their embryogenesis schemes are so widely different.
- Formalize the notion of facilitated variation and self-assembly in more complexity-theoretic terms.

4.3 Acknowledgements

I would like to thank Gerry Sussman greatly for introducing me to these problems, teaching me a lot of developmental biology, and giving me fruitful ideas and directions to pursue. Also, many thanks to Hal Abelson for critically reviewing an earlier version of this paper and for his many insightful comments and suggestions. Finally, thanks to my brother, Shamik, for helping me understand some of the biology.

5. REFERENCES

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43, May 2000.

[2] J. Bard. *Morphogenesis*. Cambridge University Press, Cambridge, UK, 1992.

[3] L. Clement and R. Nagpal. Self-assembly and self-repairing topologies. *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, 2003.

[4] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, February 1999.

[5] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[6] M. W. Kirschner and J. C. Gerhart. *The Plausibility of Life*. Yale University Press, 2005.

[7] A. Kondacs. Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation. *International Joint Conference on Artificial Intelligence*, 2003.

[8] C. Nüsslein-Volhard. Gradients that organize embryo development. *Scientific American*, August 1996.

[9] I. Salazar-Ciudad, J. Jernvall, and S. A. Newman. Mechanisms of pattern formation in development and evolution. *Development*, 130:2027–2037, 2003.

[10] J. Slack. *From egg to embryo*. Cambridge University Press, Cambridge, UK, 1991.

[9]This simulates the biological conjecture that NCC differentiation is determined by the extracellular matrix.

APPENDIX

A. GROWING AN ELLIPSE

Listing 1: A standard grower from the library

```
package growers;
import framework.Grower;

public class StdLumpGrower extends Grower {
        private Class<? extends Organizer> cls;

        public StdLumpGrower(Class<? extends Organizer> c)      {
                cls = c;
        }

        public OrganizerVectorPair grow(Organizer org, Universe uni) {
                Vector gdir = uni.getDirection(org);
                if(!uni.isOccupied(org,gdir,Organizer.class))
                        try {
                                return new OrganizerVectorPair(
                                                        cls.newInstance(),
                                                        gdir);
                        } catch (Exception e) {}
                return null;
        }
}
```

Listing 2: Description of the *starter* organizer type with parametrized morphogen name

```
package examples.ellipse;

import java.awt.Color;
import organizers.StdCell;

public class EllipseStarter extends StdCell{
        public EllipseStarter(final String chem)        {
                super();
                color = Color.red;
                registerGrower(new EllipseCellGrower(),false);
                addAgeAction(0,40,new Runnable() {
                        public void run() {
                                moveRandomly();
                        }
                });
                addAgeAction(60,60,
                                new Runnable() {
                                        public void run() {
                                                activateGrower(EllipseCellGrower.class);
                                        }
                                });
                addGeneralAction(new Runnable() {
                        public void run() {
                                broadcast(chem,8);
                        }
                });
        }
}
```

Listing 3: A *starter* organizer releasing morphogen a

```
package examples.ellipse;

public class EllipseStarter_a extends EllipseStarter {
        public EllipseStarter_a()       {
                super("a");
        }
}
```

Listing 4: A *starter* organizer releasing morphogen b

```
package examples.ellipse;

public class EllipseStarter_b extends EllipseStarter {
        public EllipseStarter_b()       {
                super("b");
        }
}
```

Listing 5: Description of the *ball* organizer type

```
package examples.ellipse;
import organizers.StdCell;

public class EllipseCell extends StdCell {
        public EllipseCell()    {
                super();
                registerGrower(new EllipseCellGrower(),true);
                registerReceptor("b",true);
                registerReceptor("c",true);
                addGeneralAction(new Runnable() {
                        public void run() {
                                if((currentReception("b").level +
                                    currentReception("c").level) < 5)
                                        inhibitGrower(EllipseCellGrower.class);
                        }
                });
                addGeneralAction(new Runnable() {
                        public void run() {
                                if((currentReception("b").level +
                                    currentReception("c").level) >= 5)
                                        activateGrower(EllipseCellGrower.class);
                        }
                });
                addGeneralAction(new Runnable() {
                        public void run() {
                                broadcast("b",currentReception("b").level-1);
                        }
                });
                addGeneralAction(new Runnable() {
                        public void run() {
                                broadcast("c",currentReception("c").level-1);
                        }
                });
        }
}
```

Listing 6: The crucial *Precursor* organizer type

```
package examples.ellipse;
import java.awt.Color;
import growers.SingleTimeGrower;
import organizers.StdCell;

public class Precursor extends StdCell {
        public Precursor()      {
                super();
                color = Color.blue;
                registerGrower(new SingleTimeGrower(EllipseStarter_a.class),true);

                // For figure 5, comment out next line of code
                // For figure 6a, change next line of code to class EllipseStarter_a
                // For figure 6b, leave next line of code as is
                registerGrower(new SingleTimeGrower(EllipseStarter_b.class),true);
        }
}
```