Programmable Self-Assembly Using Biologically-Inspired Multiagent Control

Radhika Nagpal
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Boston, MA 02139, USA
radhi@ai.mit.edu

ABSTRACT

This paper presents a programming language that specifies a robust process for shape formation on a sheet of identically-programmed agents, by combining local organization primitives from epithelial cell morphogenesis and *Drosophila* cell differentiation with combination rules from geometry. This work represents a significantly different approach to the design of self-organizing systems: the desired global shape is specified using an abstract geometry-based language, and the agent program is *directly compiled* from the global specification. The resulting self-assembly process is extremely reliable in the face of random agent distributions, random agent death and varying agent numbers, without relying on global coordinates or centralized control.

Categories and Subject Descriptors

I.2.11 [**Computing Methodologies**]: Artificial Intelligence—*Distributed Artificial Intelligence, Multiagent systems, Languages and structures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Design, Languages, Algorithms, Reliability

Keywords

Morphogenesis, Amorphous Computing, Collective Behavior, Smart Matter, Pattern Formation, Paper-Folding

1. INTRODUCTION

This paper presents a programming language approach to self-assembling complex structures from locally-interacting agents, using techniques inspired by developmental biology. We present a programming language for instructing a sheet of locally-interacting, identically-programmed agents to assemble themselves into a predetermined global shape. The language specifies a global shape as a folding construction on a continuous sheet, using a set of axioms from paper-folding mathematics [6]. The global specification is subsequently compiled into local programs that are run on the individual agents. The agent programs use robust local organization primitives inspired by studies of cell differentiation and morphogenesis in multicellular organisms [10, 17]. With this language, a wide variety of folded shapes and 2D patterns can be specified at an abstract level, compiled into agent programs, and then synthesized using purely local interactions between identically-programmed agents.

This system has several unique features. In contrast to approaches based on cellular automata or evolution, the program executed by an agent is *automatically compiled* from the global shape description. We provide a small set of biologically-inspired primitives that form the basis of agent programs: gradients, neighborhood query, polarity inversion, cell-to-cell contact and flexible folding. The resulting process is extremely reliable in the face of random agent distributions, random agent death and varying agent numbers and does not rely on global coordinates or centralized control. We show that an average local neighborhood of 15 agents is sufficient to reliably self-assemble complex shapes and geometric patterns on randomly distributed agents. We also show that the derived agent program uses only a small amount of local state and the majority of the code is conserved across all global shapes.

This research is motivated by emerging technologies, such as MEMs[1] devices, that are making it possible to bulk-manufacture millions of tiny computing elements integrated with sensors and actuators and embed these into materials and structures. Already many novel applications that integrate computation into the environment are being envisioned and built: smart materials such as sensor covered beams that actively resist buckling[2], modular self reconfiguring robots[5], self-assembling nanostructures[7]. Meanwhile, new directions in biocomputing may even make it possible to harness the many sensors and actuators in cells and create programmable tissue substrates [16].

These novel computational environments pose many new challenges, that are not met by distributed and parallel computing. These applications will require coherent and robust behavior from the interactions of multitudes of agents and their interactions with the environment. Individual agents will have limited resources and reliability and the intercon-

[1]Micro-electronic Mechanical Devices. Integrates mechanical sensors/actuators with silicon based integrated circuits.

nects between agents will be local, irregular and possibly time-varying. These new environments fundamentally stress the limits of our current engineering and programming techniques, which rely heavily on precision parts and strongly regulated environments to achieve fault-tolerance.

Approaches within the applications community have been dominated by a centralized, hierarchical mind-set. Within the MEMs community, programming strategies have for the most part been centralized applications of traditional control theory; the few decentralized approaches assume access to global knowledge of the system and tend to focus on hierarchical control[2]. Centralized hierarchies are not scalable and can be quite brittle, catastrophically failing if a high-level node fails. In the reconfigurable robotics community, the focus has been on centralized and heuristic searches, which quickly become intractable for large numbers of modules[14]. There is a strong tendency to depend on centralized information, such as global clocks or external beacons for triangulating position, which puts severe limitations on the possible applications and environments and exposes easily attacked points of failure. These programming strategies put pressure on system designers to build complex, precise (and thus expensive) agents rather than cheap, mass-produced, unreliable computing agents that one can conceive of just throwing at a problem.

Currently, however, few alternatives exist. Approaches based on cellular automata and artificial life research have been difficult to generalize; local rules are constructed empirically without providing a framework for constructing local rules to obtain any desired goal. One notable exception is Mataric's work on a language for synthesizing complex behavior in colonies of ant-like robots, by combining a set of basis behaviors[11]. However, interactions between the basis behaviors can be quite complex and for the most part complex behavior is generated by using evolutionary or learning approaches[11]. Evolutionary and genetic approaches are more general but the local rules are evolved without any understanding of how or why they work. This makes the correctness and robustness of the evolved system difficult to verify and analyze [4].

By contrast, biological systems achieve incredible robustness in the face of constantly dying and replacing parts. The precision and reliability of embryogenesis in the face of unreliable cells, variations in cell numbers, and changes in the environment, is enough to make any engineer green with envy. Currently, developmental biology remains untapped as a source for algorithms, in spite of its incredible robustness and complexity.

We propose to use morphogenesis and developmental biology as a source of mechanisms and general principles for organizing complex behavior. Our approach is to formalize these general principles as *programming languages* — with explicit primitives, means of combination, and means of abstraction — thus providing a framework for the design and analysis of self-organizing systems. This paper presents an example of applying this approach to self-assembly and is part of larger vision called Amorphous Computing to explore new programming models for collective behavior[1]. Recently, work on a modular self-reconfigurable robot has applied techniques similar to those developed in our research group for self-assembling branching structures[5, 3]. We believe that these new programming models will impact the design of and approach to reconfigurable robotics, self as-

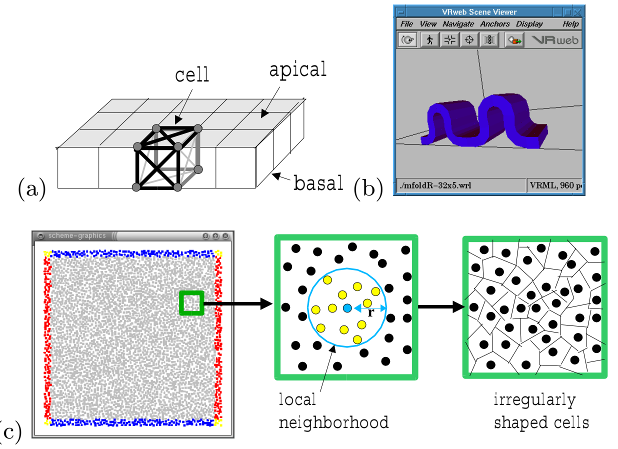


Figure 1: (a) A programmable material (b) Dynamic simulation of a sheet folding (c) Simulation model for a programmable sheet.

sembly, and smart-matter applications, and also influence our engineering principles for robust design.

The remainder of the paper is organized as follows: the first three sections present the programmable sheet model, the global shape language and the global to local compilation process. The next sections present several simulation examples along with an analysis of the resource consumption and robustness of the system.

2. A PROGRAMMABLE MATERIAL

Imagine a flexible substrate, consisting of millions of tiny interwoven programmable fibers, that can be programmed to assume a different global shapes. One could design complex static and dynamic structures from a single substrate. For example a programmable assembly line that moves objects by producing ripples; manufacturing by programming; reconfigurable structures for deploying in space, that fold compactly for storage but then unfold on site.

Morphogenesis (creation of form) in developmental biology can provide insights for creating programmable materials that can change shape. Epithelial cells in particular generate a wide variety of structures: skin, capillaries, and embryonic structures (gut, neural tube), through the coordinated effect of local shape changes in individual cells. Odell *et al.* have provided a mechanical model for epithelial cells[13]; the cell can actively change its shape by contracting fibers in its apical (top) and basal (bottom) membranes. The fibers are modeled by controlled damped springs. Odell *et al.* used a ring of such cells to model epithelial cell folding during neurulation and gastrulation in embryos.

Our model for a flexible programmable material is inspired by epithelial cell sheets — the programmable material is composed of a single layer sheet of flexible agents that create complex structures through the coordination of local shape changes (figure 1(a)). The actuation model is based on the Odell's epithelial cell model. A single agent has limited impact, but when many agents along a line coordinate the folding of their apical or basal fibers, the sheet is folded. Such a sheet can form a wide variety of structures by folding, as shown in figure 1(b). Numerical simulations of such structures however quickly become computationally intractable as the number of agents increases.

For the purpose of exploring and simulating shapes formed by folding, a we use a simpler and more general model of a programmable sheet. The sheet consists of a single layer of thousands of randomly and densely distributed irregularly-shaped agents. Figure 1(c) shows a simulation image of such a sheet; the simulation images always show the apical (top) surface of the sheet. The agents are represented by dots but are assumed to fill the space. The simulator computes the result of the actuation of many agents and performs the configuration change. This process is described in more detail in section 4.2.

2.0.1 *Agent Model*

The computational model for the agent is also influenced by biological cells. All agents have the *identical* program, but execute it autonomously based on local communication and internal state. Communication is strictly local: an agent can communicate only with a small local neighborhood of agents within a distance r, or through surface contact with other agents as a result of folding. The sheet starts out with a few simple initial conditions and apical/basal polarity, but apart from that the agents have no knowledge of global position or interconnect topology. There are no external beacons for triangulating position. Individual agents have limited resources and instead of unique identifiers they have random number generators to break symmetry. The motivation for these characteristics comes from the envisioned applications.

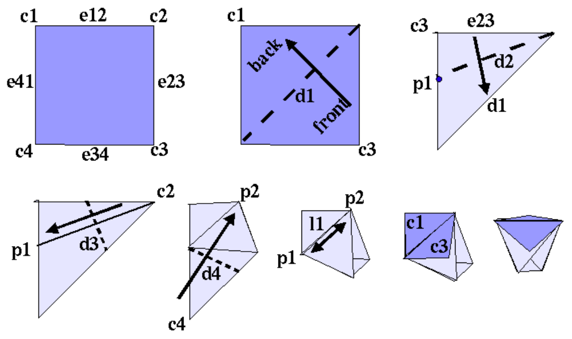
3. GLOBAL SHAPE SPECIFICATION

Given a flexible sheet which can fold, we need a language for specifying the desired global shape. This can be viewed as similar to folding a sheet of paper. Paper-folding (origami) provides a natural, although somewhat unusual, way for describing shapes that can formed by folding a sheet. In the past decade there has been renewed interest in the mathematics of paper-folding. Huzita has presented a set of six axioms for constructing shapes using straight line folds that describe a large class of folded shapes [6]. The first four are the most commonly used:

1. Given two points $p1$ and $p2$, fold a line between them.
2. Given two points $p1$ and $p2$, fold $p1$ onto $p2$ (the perpendicular bisector of the the line $p1p2$).
3. Given two lines $L1$ and $L2$, fold $L1$ onto $L2$ (the bisector of the angle between $L1$ and $L2$).
4. Given $p1$ and $L1$, fold $L1$ onto itself through $p1$ (the line perpendicular to $L1$ through $p1$).

These paper-folding axioms have considerable descriptive power. Huzita has proven that four axioms can construct all plane Euclidean constructions[6]. Lang[9] has shown that tree-based folded shapes can be automatically generated by computer and methods exist for constructing scaled polygonal shapes. There is a large practical literature of shapes that can be constructed using these techniques. As the relationship between paper-folding constructions and geometry is explored further, the results will directly impact this work.

We have developed a global shape specification language based on Huzita's axioms and paper-folding practice, called



```
;; OSL Cup program
;;---------------------
(define d1 (axiom2 c3 c1))
(define front (create-region c3 d1))
(define back  (create-region c1 d1))
(execute-fold d1 apical landmark=c3)

(define d2 (axiom3 e23 d1))
(define p1 (intersect d2 e34))
(define d3 (axiom2 c2 p1))
(execute-fold d3 apical landmark=c2)

(define p2 (intersect d3 e23))
(define d4 (axiom2 c4 p2))
(execute-fold d4 apical landmark=c4)

(define l1 (axiom1 p1 p2))
(within-region front
  (execute-fold l1 apical landmark=c3))
(within-region back
  (execute-fold l1 basal landmark=c1))
```

Figure 2: Folding diagram for a cup, and the corresponding OSL Program

the Origami Shape Language (OSL). The language is described in detail in [12]; here we illustrate most aspects of it through a simple but functional example, a cup.

The folding diagram for the cup and the corresponding OSL program are shown in figure 2. The basic elements of the language are points, lines and regions. Initially, the sheet starts out with four corner points (`c1-c4`) and four edge lines (`e12-e41`) and an apical-basal polarity. The axioms generate new lines from existing points and lines, while new points are created by intersecting lines. For example, the first cup operation constructs the diagonal `d1` from the points `c1` and `c2` by using axiom 2. The diagonal divides the sheet into two regions, `front` and `back`; a region is defined by a line that divides the sheet into two and a point on one side of the line. We can restrict later operations to specific regions. Not all lines are folded, for example the line `d2` is an intermediate step used only to create point `p1`. Therefore the fold execution is separated from the axioms. In origami diagrams there is an implicit top surface facing the viewer. The top surface is made explicit in OSL by having the sheet maintain an apical (top) and basal (bottom) surface. There are two types of folds, *apical* and *basal*. Given a crease, a fold that puts the apical surface on the inside is an apical fold and vice versa. After a fold is executed, the apical surface must be re-determined. The landmark in execute-fold specifies the side of the crease that moves in the diagram and hence the side that will reverse its apical/basal polarity. The choice of landmarks is important. In the case of the cup, the choice of landmarks ensures that both lateral flaps of the cup end up on the same side. The fold is always a flat fold, and hence the structure created by OSL are flat but layered structure. Folds are assumed to go through all layers of the paper, unless they are restricted to a region. For example, the line `l1`, created using axiom 1, goes through both layers of the sheet. However we want to fold the front layer towards us and back layer away from us. We use the regions to accomplish this; an apical fold is executed in the `front` region and a basal fold in the `back` region. The same idea can be used for folding a subset of the layers. In the end if we open the sheet we get a crease pattern that tells us the location of all the lines which can be used for verification.

Rather than specify a global shape directly, this language specifies a process for folding the shape. However this specification is abstract — the process is on a continuous sheet with no notion of agents or self-assembly.

4. GLOBAL TO LOCAL COMPILATION

The agent program is directly compiled from the global shape specification. We provide a small set of primitives for organization at the local level and a means for combining these primitives in robust and predictable ways to create the agent program for a given shape. Hence the relationship between local behavior and global behavior is well understood. However, like any other emergent system, the eventual shape emerges as a result of the local interactions between the agents. The compilation process confers many advantages: we can use theoretical results from paper-folding to reason about the kinds of shapes can and cannot be self-assembled and we can use the decomposition into primitives and means of combination to analyze the robustness of the system.

This section describes the compilation process in three steps: First, we present a small set of biologically-inspired primitives that form the basis of the multiagent control. Then we show how each of the operations in OSL, such as the axioms, can be implemented by simple agent programs. Finally we show how a shape specification is compiled into a agent program.

4.1 Biologically-inspired Primitives

The agent programs are based on five primitives: gradients, neighborhood query, polarity inversion, cell-to-cell contact and flexible folding. The primitives are simple ways in which a agent interacts with its local neighborhood and uses its sensing/actuation capabilities. These primitives are inspired by biologists' understanding of how pattern and morphology appear in the development of embryos such as the *Drosophila* and sea urchin [10, 17].

Gradients: Gradients are analogous to chemical gradients secreted by biological cells; the concentration provides an estimate of distance from the source of the chemical. Gradients are believed to play an important role in providing position information in morphogenesis[17]. For instance, in the *Drosophila*, two different proteins are emitted from opposite ends of the embryo and are used by cells to determine whether they lie in the head, thorax or abdominal regions[10].

An agent creates a gradient by sending a message to its local neighborhood with the gradient name and a value of zero. The neighboring agents forward the message to their neighbors with the value incremented by one and so on, until the gradient has propagated over the entire sheet. Each agent stores the minimum value it has heard for a particular gradient name, thus the gradient value increases away from the source. Because agents communicate with only neighboring agents within a small radius, the gradient provides an estimate of distance from the source. Gradients are quite general and similar primitives have been used in other contexts [5, 15, 3][2]. The source of a gradient could be a group of agents, in which case the gradient value reflects the shortest distance to any of the sources. Thus, the shape and positions of the sources affects the spatial pattern of gradient values. For example if a single cell emits a gradient then the value increases as one moves radially away from the cell but if a line of cells emit a gradient then the gradient value increases as one moves perpendicularly away from the line.

Neighborhood Query: This primitive allows an agent to query its local neighborhood and collect information about their state. For example an agent may collect neighboring values of a gradient for comparison. This primitive is from cellular automata[15]. An agent can also broadcast a message to its entire local neighborhood.

Polarity Inversion: As part of the initial conditions of the sheet, each agent has an internal apical-basal polarity. Originally all agents have the same polarity, but an agent can choose to invert its internal apical-basal polarity.

Cell-to-cell Contact: This primitive allows communication through physical contact. When agents come into direct physical contact with each others apical or basal surface, as a result of changes in the shape of the sheet, they become part of each others local communication neighborhood. When agents cease to be in contact, then that communication bond is broken. An agent does not distinguish between the original neighborhood and the contact neighborhood.

Cell-to-cell contact is the main way in which changes to the environment (sheet) affect the behavior of the agents. It allows multiple layers of the sheet to act as a single fused layer. For example, cell-to-cell contact indirectly affects the way gradients propagate by changing the local neighborhoods. Gradients seep through regions of the sheet that are in contact with each other as if the sheet were a single layer.

Flexible Folding: This primitive comes from the actuation model of the agent. An agent can contract its apical or basal fibers in order to affect the shape of the sheet. In the simulation environment, the requests from the agents to contract apical/basal surfaces are collected and then the simulator globally computes the result of those actuations. The simulator also recalculates the cell-to-cell contact neighborhoods as a result of the change in sheet configuration. We make this computation feasible by restricting the allowed folds to straight flat folds.

[2]The gradient primitive is different from reaction-diffusion and ant pheromone-like primitives that depend on more strictly modeling diffusion, chemical reactions and/or evaporation.

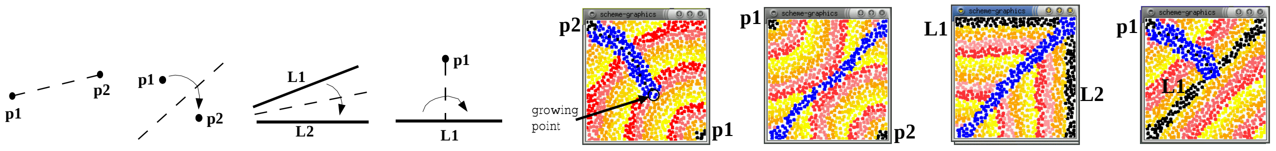


Figure 3: Huzita's axioms 1-4 and their implementation by the agents

4.2 Composition into Local Rules

This section describes how each of the global OSL operations can be implemented as a simple agent program (also called a local rule) using the set of primitives. An OSL point is represented by a group of agents approximately the size of a local neighborhood. A OSL line is represented by a line of agents of width approximately the diameter of the local neighborhood. All agents in a line or point group are equal i.e. no one agent is in charge of the group. Each agent has a boolean variable in its internal state for each distinct point/line; the variable is true if the agent is part of the point/line. Initially the sheet starts out with four distinct lines and points (edges and corners), so all agents have boolean state variables `e12, e23, e34, e41, c1, c2, c3 c4`. If an agent is part of the edge `e12` then the corresponding state variable is true. The initial conditions are very simple; agents do not know where they are within an edge and the remainder of the sheet is homogeneous, just like a blank sheet of paper.

The axioms use gradients to determine which agents belong to the crease. The axioms make use of the fact that gradients provide a distance estimate as well as reflect the shape of the source. Figure 3 shows agents forming lines that implement Huzita's axioms 1–4.

Axiom 1 uses tropism to create a crease from point $p1$ to $p2$. Tropism implies the ability to sense the direction of a gradient by comparing neighboring values. The agents in $p2$ create a gradient and the agents in $p1$ grow a crease towards $p2$ by following decreasing gradient values. This technique was introduced by Coore for creating line patterns based on tropism[3]. Axiom 2 creates a crease line such that any point on the crease is equidistant from points $p1$ and $p2$. Therefore if $p1$ and $p2$ generate two different gradients, each agent can compare if the gradient levels are approximately equal to determine if it is in the crease line. Axiom 3 uses the same local rules as axiom 2. The difference is that the gradients are produced by lines rather than points, and thus the gradient values increase as one moves perpendicular to the input line. Axiom 4 reuses the axiom 1 local rules to grow a crease from point $p1$ to the crease $l1$.

The local rules for an axiom can be expressed as a simple agent program. In Figure 4 the local rule for axiom 2 is expressed as a procedure. The local rule returns a true or false value depending on whether the agent belongs to the new point or crease and this true/false value can be stored in some new local state variable. Each axiom attempts to produce creases that are approximately twice the width of the local communication distance r.

A key piece to making the axioms work is the effect of cell-to-cell on the behavior of gradients on a folded sheet. Because multiple layers of the sheet can act as a single layer, the gradient values reflect the Euclidean distance from the source rather than the distance along the plane of the sheet.

```
(define (axiom2-rule p1 p2 g1 g2)
  ; arguments are boolean state p1 p2
  ; and gradient names g1 g2
  (if p1 (create-gradient g1))
  (if p2 (create-gradient g2))
  (wait-for-gradients g1 g2)
  ; generates a crease of width apprx 2r
  (if (< (abs (- g1 g2)) 1)
      #t
      #f))
```

Figure 4: Local Rule for Axiom 2

This allows all of the previously defined axiom local rules to work without modification on folded sheets, and the crease lines go through all layers of the sheet as expected.

Regions allow the user to *restrict the context* in which a local rule applies. A region is created by using bounded gradients, which implies that certain types of cells will not forward the gradient message; the intuition being that certain cells can act as barriers to particular gradients. The agents in the point `p1` create a bounded gradient that can not pass through the agents in the dividing line `l1`. The agents that receive the bounded gradient become part of the region. Hence we can use gradients to mark regions.

The last important global operation is `execute-fold`. In order to execute a fold along a line $l1$, all the agents in $l1$ make actuation requests to the simulator. The simulator approximates the configuration by computing the best-fit line of the set of actuating agents. The simulator then determines whether the best-fit line is a justified approximation for a given set of agents — if the line is discontinuous or significantly different from the expected width, then the simulator returns a failure notice. Thus we restrict the simulator to perform only straight line folds (rather than arbitrary deformations of the sheet) which makes the computation feasible. The simulator recalculates the contact neighbors are recalculated every time the sheet is folded. A second piece to execute-fold is re-determining the apical surface. The landmark agents create a bounded gradient thus marking the region on one side of the crease. After the fold, these agents reverse their polarity. Thus the sheet constantly maintains an apical and basal surface. This idea is inspired by tissue induction, where one region of cells can induce its polarity on another region.

4.3 Compilation of the Agent Program

Compiling an OSL program involves creating local boolean state variables for each distinct point and line and then translating each OSL operation into a call to the corresponding agent procedure with the appropriate arguments. The compiler assigns different gradient names for each call.

For example, the OSL code

```
(define d2 (axiom3 e23 d1))
(define d3 (axiom2 c2 p1))
```

becomes the agent program

```
(define d2 #f) ; corresponding local state
(define d3 #f)
(set! d2 (axiom3-rule e23 d1 g1 g2))
(set! d3 (axiom2-rule c2 p1 g3 g4))
```

The agent program mirrors the original OSL cup program. However note that at the OSL level there is no notion of gradients, or even agents. The compilation process from global to local is easy to understand. However the agent programs generated are no different from any other emergent systems — the eventual shape "emerges" as a result of local interactions between the agents and the initial conditions.

4.4 Resource Consumption

Although the process is communication intensive, the resource requirements per agent are surprisingly small — a small local state and mostly fixed code space. Each distinct point, line or region contributes a boolean to the local state. Many gradients are created, however their use is short-lived. Each operation uses no more than 3 distinct gradients, therefore an agent does not need to store more than 6 distinct gradients at any time (the previous set of gradients is kept around just in case neighbors are still finishing previous operation). The storage per gradient is proportional to the diameter of the sheet. Another interesting property is that the majority of the agent code is conserved across all shapes. The fixed code implements the primitives (like gradients) and the OSL operations. This is fixed across all shapes. Only a small part of the code corresponds to the actual shape sequence. This has an interesting analogy to biology: DNA is highly conserved across all living things.

5. EXAMPLES

The examples presented were generated by specifying the folding construction using OSL, compiling the OSL program to generate the agent program and then executing the agent program on the simulated programmable sheet. The initial conditions are always the same: boundary conditions and apical/basal polarity. Figure 5 shows a simulation of the cup folding on a sheet with 4000 agents with an average neighborhood size of 15 agents, using the program in figure 2. Figure 6 shows the sheet folding itself into an envelope.

The OSL language can also be used to create patterns. Based on Huzita's result,s we can theoretically *self-assemble any plane Euclidean construction*, i.e. any pattern that can be described using straight edge and compass construction. Practically there are resolution limitations based on how many agents there are. However the important point is that this is gives us a systematic way of self-assembling a very large class of patterns. An example of such a class is CMOS logic patterns. The figure 7 shows a caricature of an inverter chain pattern, generated by subdividing the sheet into regions and then laying down the inverter pattern within each segment. The OSL program allows procedures that capture repeated sequences, therefore the inverter chain program can be written in a modular fashion.

6. ROBUSTNESS

The shape formations could fail in many possible ways: agents forming an incorrect or crooked crease line, no agents at the intersection of two lines, errors in gradients and cell-to-cell contact, leaking regions etc. However such failures are extremely rare and the simulations presented are very reliable. The multiagent control does not rely on regular agent placement, global coordinates, or centralized control. Instead, robustness is achieved by depending on large and dense agent populations, using average behavior rather than individual behavior and trading off precision for reliability. In [12] we provide an extensive analysis of the behavior of gradients and axioms and their tolerance to random distributions and cell death. Here we present a few examples of the results.

The reliability depends critically on density. The probability that an agent has no neighbors or that a large region has no agents, should be extremely low. In addition, the accuracy of the gradient distance estimate also depends on the density. We can leverage results in the field of packet radio networks to determine the desired density. Kleinrock and Silvester presented theoretical results showing that the expected error in the distance estimate provided by a gradient rapidly decreases as the density is increased, until the expected neighborhood size is around 15-20 neighbors where it levels off[8]. We can further increase the accuracy of the distance estimate if each agent computes an average of its neighbors (smoothed value). Our experimental results closely match the theoretical results as shown in graph 8(a) and show that good accuracy can be obtained with a reasonable neighborhood size.

We can also analyze the behavior of the composition of gradients. In axiom 2 when two gradients are combined, their errors create an interference pattern. As a result the combined error varies depending on the position with respect to the sources. Axiom 2 in fact creates a crease in the region with the least error; the result is that patterns such as the inverter chain (figure 7) are geomtrically accurate even though the agents are distributed randomly. We provide a formula for the uncertainty in the position of a agent and use this to show that the accuracy of axiom 2 decreases as the ratio between the crease length and the distance between the input points increases. Graph 8(b) shows experimental results that support this conclusion; in fact the width of the line widens as the crease:input ratio is increased. Similarly one can analyze the behavior of the remaining axioms. These analyses also impact other systems that are built on top of gradient like primitives.

The self-assembly process can tolerate a small amount of random agent death. All of the local primitives are designed with a certain amount of redundancy; the behavior of a line or point is the average of many equal agents. The random distribution also plays a role: even if a small fraction of agents randomly die, the result is still a random distribution. Therefore the algorithms designed for random distributions have some amount of inherent tolerance to random agent death. Gradients are extremely robust to random death and there is no fixed hierarchy or centralized control that can be easily disrupted. Points and lines are the only sections that have long term roles and the sheet formation is vulnerable to large regional failures that kill entire points or lines. The reliability will be affected if enough agents die such that the expected local neighborhood goes significantly below 15.

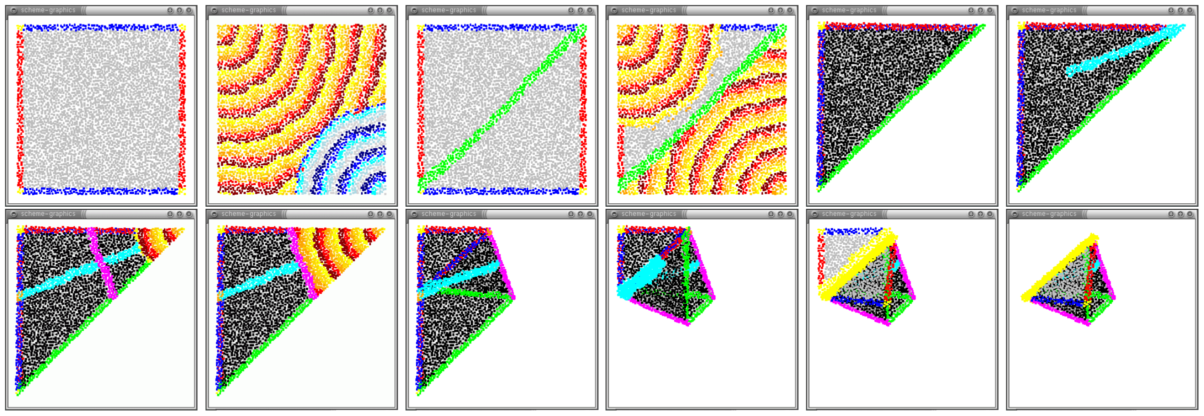


Figure 5: Simulation images from folding a cup

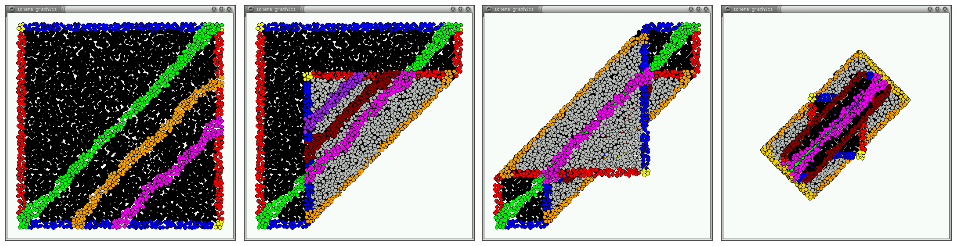


Figure 6: Folding an envelope structure

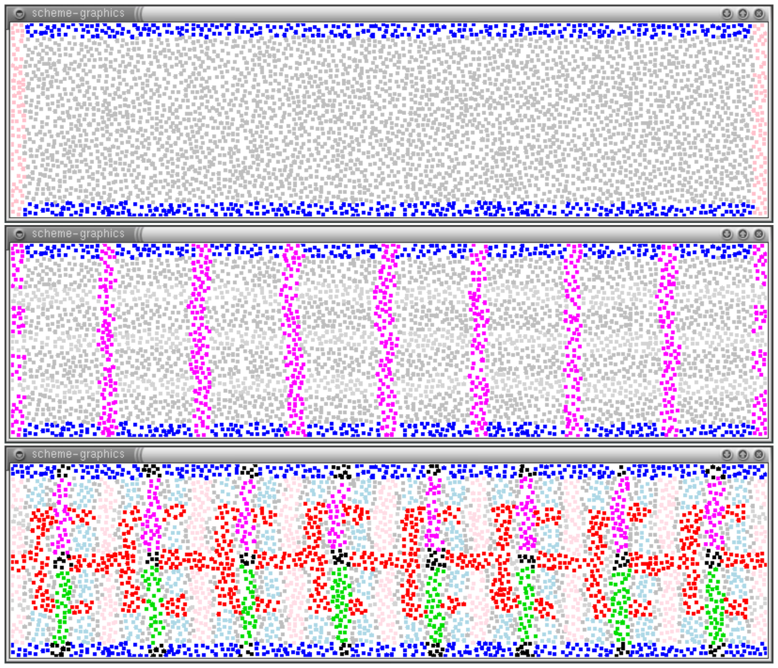
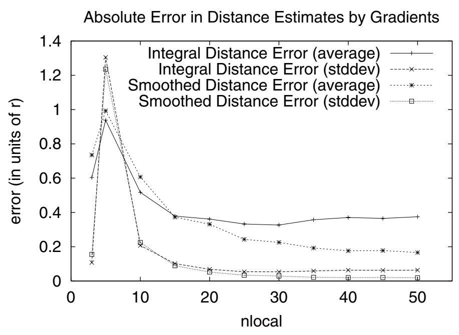


Figure 7: Differentiation into an inverter chain pattern

(a) 

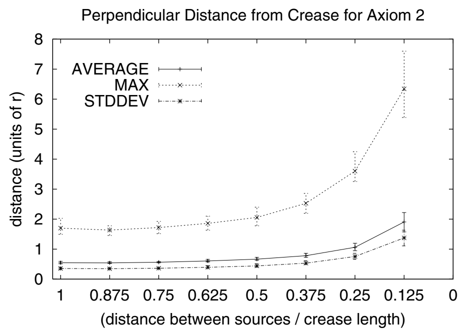
(b) 

Figure 8: Affect of the local neighborhood size on gradient and axiom 2 accuracy

The agent program is also tolerant to varying agent numbers. Using our results on the accuracy of the axioms, one can determine lower bounds on the number of agents needed to form a given structure. However, the agent program works without modification for larger numbers of agents. Not only is the local program independent of the total number of agents, the shape/pattern will automatically scale as the size of the sheet increases.

7. CONCLUSIONS

This work represents a different approach to engineering self-organizing systems. Rather than trying to map a desired goal directly to the behavior of individual agents, the problem is broken up into two pieces: a) how to achieve the goal globally b) how to map the construction steps to local rules. We take advantage of current understanding in other disciplines of how to decompose a problem. This approach suggests that exploring new global paradigms is at least as important as experimenting with local rules. Future work will focus on extending this approach to volumetric 3D shapes and investigating the formation of shape by replication (growth), mobility, and deletion (agent death). In the long run we hope to apply these frameworks to achieving coherent behavior from aggregates of genetically-modified biological cells [16].

8. 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(5), May 2000.

[2] A. Berlin. *Towards Intelligent Structures: Active Control of Buckling*. PhD thesis, MIT, Dept of Electrical Eng. and Computer Science, May 1994.

[3] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, Dept of Electrical Eng. and Computer Science, Feb. 1999.

[4] S. Forrest and M. Mitchell. What makes a problem hard for a genetic algorithm? *Machine Learning*, 13:285–319, 1993.

[5] Hogg, Bojinov, and Casal. Multiagent control of self-reconfigurable robots. In *4th International Conference on Multi-Agent Systems*, July 2000.

[6] H. Huzita and B. Scimemi. The algebra of paper-folding. In *First International Meeting of Origami Science and Technology*, Ferrara, Italy, 1989.

[7] R. Jackman, S. Brittain, A. Adams, M. Prentiss, and G. Whitesides. Design and fabrication of topologically complex, three-dimensional microstructures. *Science*, 280:2089–2091, 1998.

[8] L. Kleinrock and J. Silvester. Optimum tranmission radii for packet radio networks or why six is a magic number. In *Proc. Natnl. Telecomm. Conf.*, pages 4.3.1–4.3.5, 1978.

[9] R. J. Lang. A computational algorithm for origami design. In *Annual Symposium on Computational Geometry*, Philadelphia, PA, 1996.

[10] P. A. Lawrence. *The Making of a Fly: the Genetics of Animal Design*. Blackwell Science, Oxford, U.K., 1992.

[11] M. Mataric. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16((2-4)):321–331, Dec. 1995.

[12] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Dept of Electrical Engineering and Computer Science, June 2001.

[13] G. Odell, G. Oster, P. Alberch, and B. Burnside. The mechanical basis of morphogenesis: 1. epithelial folding and invagination. *Developmental Biology*, 85:446–462, 1981.

[14] Pamecha, Ebert-Uphoff, and Chirikjian. Useful metrics for modular robot planning. *IEEE Trans. on Robotics and Automation*, 13(4), Aug. 1997.

[15] M. Resnick. *Turtles, Termites and Traffic Jams*. MIT Press, Cambridge, MA, 1994.

[16] R. Weiss, G. Homsy, and T. Knight. Toward in vivo digital circuits. In *Dimacs Workshop on Evolution as Computation*, Jan. 1999.

[17] L. Wolpert. Positional information and the spatial pattern of cellular differentiation. *Journal of Theoretical Biology*, 25:1–47, 1969.