Swarm Intelligence manuscript No.
(will be inserted by the editor)

Functional Blueprints: An Approach to Modularity in Grown Systems

Jacob Beal

Received: date / Accepted: date

Abstract The engineering of grown systems poses fundamentally different system integration challenges than ordinary engineering of static designs. On the one hand, a grown system must be capable of surviving not only in its final form, but at every intermediate stage, despite the fact that its subsystems may grow unevenly or be subject to different scaling laws. On the other hand, the ability to grow offers much greater potential for adaptation, either to changes in the environment or to internal stresses developed as the system grows. I observe that the ability of subsystems to tolerate stress can be used to transform incremental adaptation into the dynamic discovery of viable growth trajectories for the system as a whole. Using this observation, I propose an engineering approach based on *functional blueprints*, under which a system is specified in terms of desired performance and means of incrementally correcting deficiencies. I explore how manifold geometric programming can support such an approach by simplifying the construction of distortion-tolerant programs, then demonstrate the functional blueprints approach by applying it to integrate simplified models of tissue growth and vascularization, and further show how the composed system may itself be modulated for use as a component in a more complex design.

1 Introduction

One of the most remarkable facts about animals is that they are not generally injured by their own growth. An animal is composed of many tightly integrated systems, all interlocking in multiple ways. For example, bones fit together in joints that permit a useful range of motion, muscles attach to the bones in a pattern that allows them to work together effectively to move the body, the circulatory system delivers oxygen and nutrients to every portion of the bones and muscles via an intricate network of vessels, and their waste products are carried away for removal by the kidneys. As the animal grows, from an embryo to a mature adult, all of these systems are constantly adapting in order to remain integrated and fully functional.

This is not generally the case for our current engineered systems. Many artifacts, such as cars and airplanes, have no real capacity for growth at all. In engineered systems

BBN Technologies, Cambridge, MA, USA E-mail: jakebeal@bbn.com

that do grow, the growth is often accompanied by significant degradation of function as the existing balance of systems is disrupted and painstakingly reintegrated. Adding an extension to a house means months of dust, being unable to use existing rooms, and electrical and plumbing disruptions. Expanding the road networks of a growing city requires years of detours and traffic disruptions, not to mention economic disruption for businesses nearby the construction. Upgrading the software of a computer often requires a reboot and leaves a trail of incompatibilities and ongoing headaches (though autonomic computing has made some progress in this area). Beyond the obvious differences in mechanical and material properties, we simply do not know how to describe our designs in a way that allows for disruption-free growth. We may thus be led to consider languages for adaptable design, both to better understand animal development and also to improve engineered systems. This is particularly pressing given the rapid progress occurring in synthetic biology (e.g. engineered pattern formation (Basu et al, 2005) and standardized DNA assembly protocols (Shetty et al, 2008)), where the systematic engineering of DNA programs promises to soon allow us to create engineered objects that are literally grown from living cells.

One particularly elegant example of growth and adaptivity in biological systems is the vascular system (Carmeliet, 2003). Under normal conditions, sufficient oxygen diffuses through the walls of capillaries into the surrounding tissue. When cells are not receiving enough oxygen, however, they become stressed and emit a chemical signal that causes nearby capillaries to leak. The vascular system also has an elegant program for regulating its capacity. When a capillary leaks often, a new capillary begins to grow out of the leaky area, increasing the available blood supply to the oxygen-starved region. Blood vessels are elastic, and when they are frequently stretched, the cells divide, increasing the capacity of the vessel; likewise, when frequently contracted, cells die and shrink the vessel. Thus, the vascular system incrementally grows and shrinks to match the demand of the tissues it serves, branching into under-served regions and adjusting the size of vessels to match the flow through them.

In this paper, I propose an engineering approach of *functional blueprints* inspired by this and other similar adaptive biological systems. If each system is capable of operating under minor stress and of incrementally adjusting to decrease stress, then feedback between components should allow all the subsystems comprising a natural or engineered system to maintain a tight integration as the system grows, even if the relationship and relative sizes of subsystems are changing. Functional blueprints attempt to capture this by specifying a system in terms of desired performance and means of incrementally correcting deficiencies.

In the remainder of this paper, I first discuss how stress tolerance can enable integrated growth, then formalize this idea with a definition for a functional blueprint. I next show that functional blueprints can be readily constructed using a programming approach based on manifold geometry, such as that instantiated in the Proto (Beal and Bachrach, 2006) spatial computing language. Finally, I demonstrate the functional blueprint approach by applying it to integrate simplified models of cell density maintenance and vascularization to produce synchronized tissue growth, then show how the composed system may itself be modulated for use as a component in a more complex design.

1.1 Related Work

Morphogenesis in natural systems has been a subject of intensive study. In recent years, deciphering of genetic mechanisms controlling development, such as how the *hox* gene complex produces the overall body plan of animals, has led to a synthesis of evolution and development (EvoDevo) (c.f. Carroll (2005)), and theories of how the adaptivity of organisms to body plan variations may facilitate evolution (Kirschner and Norton, 2005).

Inspiration from natural systems has led to investigation of how growable patterns might be programmed, generally focusing on the establishment of shape, with less attention to integration of function. Doursat, for example, has developed a *hox*-gene based network model for artificial evolution of animal-like systems (Doursat, 2008). Similarly, the development of structure in growing plants has long been modeled at a high level by term-rewriting systems (Prusinkiewicz and Lindenmayer, 1990), which the MGS language (Spicher and Michel, 2006) extends into a general model of structure development through topological rewriting. Other notable approaches include Coore's Growing Point Language (Coore, 1999), which uses a botanical metaphor to create topological structure and Nagpal's Origami Shape Language (Nagpal, 2001), which creates geometric forms through folding. Most similar to the work presented in this paper is Werfel's work on distributed construction (Werfel, 2006), which has been extended to use functional constraints to generate adaptive structure in response to environmental stimuli (Werfel and Nagpal, 2007).

The problems of integration addressed in this paper are also related to control theory. Standard control theory, however, has difficulty addressing systems with large numbers of non-linearly interacting parts, which are typical of growing systems. A notable exception may be viability theory (Aubin, 1991), a branch of mathematical theory which is intended to address such concerns, thought it still focuses on systems well-described by differential equations.

Grown systems may also be viewed as a special case of swarm systems, in which the typical mobility of individual cells is highly limited. Swarm intelligence may be defined as self-organizing global behavior from local or stigmergic interactions of a large number of relatively homogeneous individuals (Dorigo and Birattari, 2007). While much swarm intelligence research has focused on individuals that move freely with respect to one another, as in the case of ant- and termite-inspired systems (e.g. Dorigo and Stutzle (2004), Werfel (2006)) or flocking models (e.g. Reynolds (1987), Couzin et al (2005)), this is not always the case. Many projects in swarm robotics and self-reconfigurable robotics have investigated the formation of shapes from collections of robots. For example, Stoy and Nagpal investigated shape formation by a swarm where individuals construct a lattice-like structure through gradient climbing (Stoy and Nagpal, 2004), and the Swarm-bots project has investigated small-scale morphogenesis with robots that clamp onto one another in order to collectively accomplish tasks that they cannot accomplish individually, such as crossing gaps and balancing on thin bridges (O'Grady et al, 2009, 2010). The sort of grown systems envisioned in this paper are simply another point on this spectrum, where mobility is typically lower and devices may be more elastic and capable of self-reproduction.

2 Stress Tolerance Enables Integrated Growth

The basic insight enabling this new approach is as follows: a stress-tolerant system can exploit its tolerance to navigate dynamically through the space of viable designs. This is rather foreign to the typical engineering approach to failure tolerance. Usually, an engineer designing a system treats its ability to tolerate failures like guard rails on a highway: important for safety, changing terrible outcomes into merely bad, but never touched under normal circumstances. Alternately, though, we can treat the system's robustness as a guide, the way that a blind person might use a guard rail to follow the twists and turns of the road.

Under this alternate view, stress within the system becomes the coordinating signal by which independently developing subsystems are integrated. When the system is far from the edge of its viability envelope, it can develop freely. When it comes near the edge, however, and its viability begins to be impaired, then the growth of the subsystems driving it to non-viability is slowed or stopped temporarily. Other subsystems, triggered to act by the increased stress, adjust to bring the system as a whole back within the viability envelope. The driving subsystems are then re-enabled, and the cycle of growth and correction begins again.

Critically, this is only possible if the system is able to determine the direction of stress, and if stress caused by one system can be relieved by adjustment of another. For example, if a beam has become the wrong length due to the change of structure around it, the beam will experience tensile stress if it is too short and compressive stress if it is too long. If only the magnitude of error is measured, then the beam cannot know whether it should grow or shrink to reduce stress, but if the direction of the stress is measured then the appropriate corrective measure becomes obvious.

For example, consider an abstract system with two attributes, whose combination is viable only in the complex envelope shown in Figure 1. The horizontal attribute drives system growth, increasing whenever the system is clearly viable (horizontal arrows). When the system's viability begins to be impaired (faint blue), the secondary attribute adjusts to correct it (vertical arrows). Given some hysteresis in the switch between driving and correction, the switch in modes need only occur a finite number of times. By repeatedly switching between driving growth and relieving stress, a system may navigate a complex viability envelope.

Thus we see that it is possible to use systemic stress as a signal to coordinate the growth of independently developing subsystems. This will not, of course, work for all possible such viability spaces: if the viability space includes a "dead end" that the driving attribute can push into, then it cannot be successfully navigated without additional guidance. For many systems, however, such as those where the coordination problem is rooted in the difference of scaling laws, (e.g. bone length (linear) vs. muscle cross-section (square) vs. lung capacity (cubic)), the viability space is guaranteed to be navigable. Note also that when stress is localized, the process of correction can be localized as well, allowing navigation to be parallelized when systems are not directly affecting one another. For example, different sets of developing muscles can be sore at the same time.

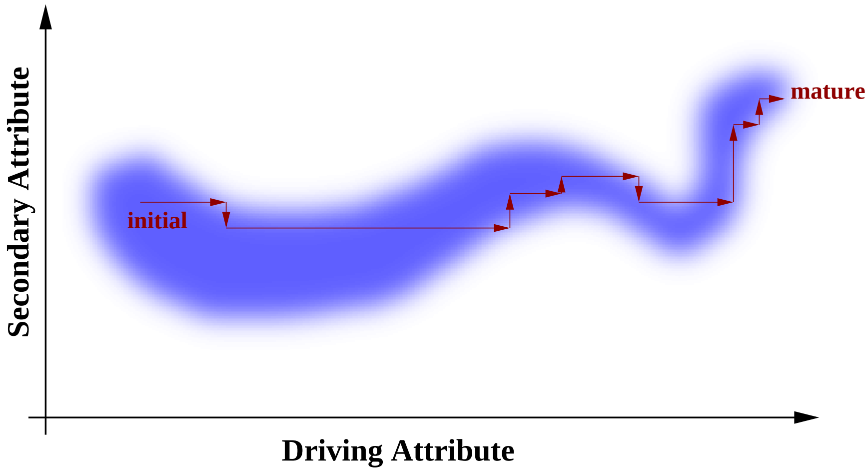


Fig. 1 In this abstract example, a growing system with two attributes uses stress tolerance to navigate through a complex viability envelope (blue). When unconstrained, the system grows its driving attribute (horizontal arrows). When the system's viability begins to be impaired (faint blue), it relieves that stress by adjusting its secondary attribute (vertical arrows). By repeatedly switching between driving growth and relieving stress, the system is able to navigate a complex viability envelope.

3 Functional Blueprints

Having made the observation that stress tolerance can allow a system to dynamically discover trajectories through its viability space, we can now take the next step and propose an engineering framework for predictably constructing such systems. Let us thus define a *functional blueprint* for some system X to consist of four elements:

1. A *system behavior* that degrades gracefully across some range of viability. Formally, if C_X is a manifold of possible configurations of system X, then it must be possible to establish a concave viability function v_X mapping $C_X \to [0, \infty)$ such that for any configuration $c_X \in C_X$, only viable configurations have $v_X(c_X) > 0$,[1] and for any such configuration there exists a ball $B \subseteq C_X$ centered on c_X such that $v_X(B) > 0$.
2. A *stress metric* quantifying the degree and direction of stress on the system. Formally, let the stress metric s_X be a vector field on C_X such that s_X is the gradient of some legal viability function for C_X.
3. An *incremental program* that relieves stress through growth (or possibly shrinking). Formally, let this be a parametrized map $i_{X,\epsilon,d} : C_X \to C_X$ that shifts a configuration by ϵ distance in the direction d.
4. A program to construct an initial *minimal system*. This initial minimal system, which we label X_0, must be viable, i.e. $v_X(X_0) > 0$.

[1] Note that not all viable configurations need have $v_X(c_X) > 0$: the point is for the viability function to serve as a conservative guide for system growth, not to capture the precise boundary at which the system fails.

Graceful degradation of system behavior asserts that the core functionality of the system must not have a sharp transition between viable and non-viable.[2] The stress metric and incremental program combine to shift a degraded system's configuration back toward viability. Finally, the minimal system makes sure there is some viable place to start.

To transfer these properties to a composite system, it is necessary only to ensure that the subsystems are coupled such that the side effects of subsystems on one another are incremental. Formally, the action of each subsystem X's incremental program on each other subsystem Y forms a continuous map, $\pi_{X,Y}$. Given such a coupling of functional blueprints, it is always the case that it is possible to adjust any given subsystem by some small increment without knocking any other subsystem out of its range of viability. This can be proved by construction:

Theorem 1 *Consider a system S, for which every subsystem has a functional blueprint, and let X and Y be subsystems of S. For any given configuration c_S, if $v_X(c_X) > 0$, then there exists a $\delta > 0$ such that $c'_S = i_{Y,\epsilon,d}(c_S)$ has $v_X(c'_X) > 0$ for every d and $\epsilon \leq \delta$.*

Proof By graceful degradation, we know that there exists a ball B centered on c_X such that every point $b \in B$ also has $v_X(b) > 0$. By the continuity of the coupling map $\pi_{X,Y}$, we know that the preimage of $\pi^{-1}_{X,Y}(B)$ is an open set. Being an open set, the preimage must contain some ball B' of radius δ around the configuration c_Y. By the definition of an incremental program, any configuration c'_S accessible via subsystem Y's incremental program $i_{Y,\epsilon,d}(c_S)$ is within the ball B' for $\epsilon \leq \delta$. Since B' is a subset of $\pi^{-1}_{X,Y}(B)$, c'_X must be within B and must therefore have $v_X(c'_X) > 0$. □

A simple growing composite system, such as the one illustrated in Figure 1, can thus be constructed simply by taking the composite stress to be the maximum stress of any subsystem and executing the incremental program of the maximally stressed subsystem. The system can then be navigated toward a desired mature form by driving any subsystem or collection of subsystems whenever the composite stress is low enough.

4 Distortion-Tolerant Programming

We have now seen that, in theory at least, the functional blueprint framework can be used to create components that use stress signals to couple their development. Taking advantage of this might be very difficult, however, since the program actually implementing each component needs to be constructed in such a way that it tolerates the distortions caused by its ongoing change in structure, or the structural changes imposed on it by other components with which it interacts. The practicality of the functional blueprint approach will thus be greatly enhanced if we are able to construct our programs using some representation that is inherently tolerant to distortions.

One possible such approach is to define functional blueprints in terms of manifold geometry computations. We shall demonstrate the potential of this approach by working through several examples of scalable and tiled pattern construction using the Proto spatial computing language (Beal and Bachrach, 2006). In Proto, computations

[2] The formal definition is slightly weaker than continuity, but also asserts concavity to support the stress function being used for guidance.

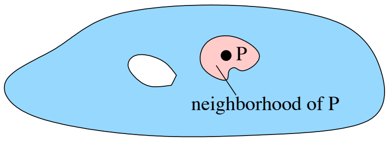


Fig. 2 An amorphous medium is a manifold where every point is a device that knows its neighbors' recent past state.

are specified in terms of information flow and geometric operations across the volume of space where the computation is being executed. Using Proto's manifold model of space, we shall see that it is possible to construct geometric descriptions of structure that automatically adapt to match distortions in the shape of the component on which they are being executed.

4.1 Proto

We now give a brief overview of Proto: for a detailed explanation, see Beal and Bachrach (2006) or the tutorials included with the MIT Proto distribution (MIT Proto, Retrieved November 22, 2010). To aid comprehension, however, each program we present will have its structure explained through embedded comments. The reader is further advised that, although Proto uses LISP-like syntax and keywords, its semantics is not the same.

Proto is a functional dataflow language in which the aggregate of devices to be programmed is viewed as a discrete approximation of the continuous space through which they are distributed, using the *amorphous medium* abstraction (Beal, 2004). An amorphous medium is a manifold with a computational device at every point, where every device knows the recent past state of all other devices in its neighborhood (Figure 2). While an amorphous medium cannot, of course, be constructed, it can be approximated on the discrete network of locally communicating devices, such as we might expect to find in a grown system comprised of many small cell-like components.

Proto uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled sub-problems: global descriptions of programs, compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network. Aggregate behavior descriptions are thus transformed into a program to execute on each device (every device is given the same program) and an interaction protocol by which devices cooperate to approximate the desired aggregate behavior.

When interpreted, a Proto program produces a dataflow graph of operations on fields, where some of those operations may be functions implemented with their own dataflow graph on fields. This program is then evaluated against a manifold to produce a field with values that evolve over time. Proto uses four families of operations: pointwise operations like `+` that involve neither space nor time, restriction operations like `if` that limit execution to a subspace, feedback operations like `letfed` that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state (gathered with `nbr`) and space-time measures like `nbr-range`, then summarize the computed values in the neighborhood with a set operation like integral (`int-hood`) or minimum (`min-hood`).

With Proto's carefully chosen set of operators, compilation and discrete approximation are straightforward, because each primitive is chosen to have a continuous-space specification that can be coherently approximated with local device actions, and their composition rules preserve this property (except in extreme circumstances). Thus, Proto allows a programmer to specify a distributed program in terms of information flow and geometric operations over the space occupied by the aggregate of devices.

4.2 Pattern Formation on Distorting Manifolds

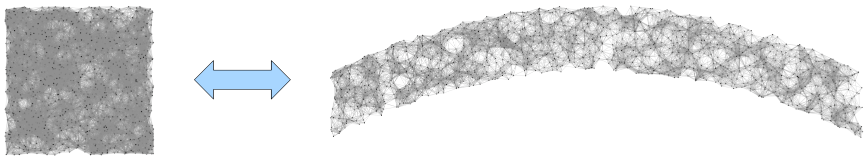
Because Proto is based on a manifold model of space, geometric properties such a distance are measured *through* a component, rather than external to it. This may prove useful in the development of functional blueprints, because the definition of many basic geometric operations changes to reflect the manifold. If a functional blueprint is constructed in terms of such geometric operations, then a developing component that is being warped, whether by internal or external forces, need not adjust its program when its shape is changed. Indeed, if the program is formulated as a feed-forward composition of self-stabilizing geometric operators, then its adaptability is entirely implicit.

In his dissertation, Yamins (2007) demonstrated that there are precisely two classes of patterns that can in general be formed on the basis of local interactions with a constant number of bits of state: scalable “proportional” patterns and tiled “repeat” patterns. Since the genetic regulatory networks of biological organisms appear to have only a few distinguishable bits of state per regulatory interaction, it is reasonable to suppose that a significant fraction of multicellular organism development may be specified in terms of combinations of these two types of pattern.

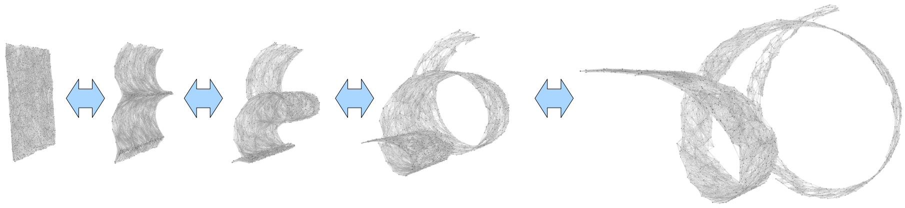
Accordingly, we may experimentally validate the hypothesis that a manifold geometric programming approach, such as that instantiated in Proto, has the potential to simplify the construction of distortion-tolerant programs for functional blueprints, by constructing several instances of these two types of pattern. The distortion tolerance of each pattern will be tested by executing the program in the MIT Proto simulator on four different instances of a distorting space:

- Square space, distorting into a parabolic space, then returning to the original configuration (Figure 3(a)).
- Square space, distorting into a three-dimensional spiral, then returning to the original configuration (Figure 3(b)).
- Irregular space, distorting along a parabolic form, then returning to the original configuration (Figure 3(c)).
- Irregular space, distorting to split apart the top 4/5 into two halves, then returning to the original configuration (Figure 3(d)).

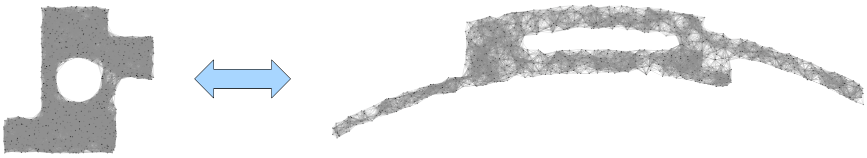
In the first two cases, simulations are run on 2000 devices distributed evenly through a 100×100 unit the space; in the latter two, the same density of devices is used: the simulator discards all those distributed to non-covered portions of the initial square distribution. Devices communicate with one another using local unit disc broadcasts with a range of r units. Each simulation begins by executing the program for a short period of time to allow it to stabilize an initial pattern, then executes the distortion. The distortion is executed simply by moving devices incrementally along a straight line from their original position to their final position and back (distortion functions are listed in the Supplementary Information).



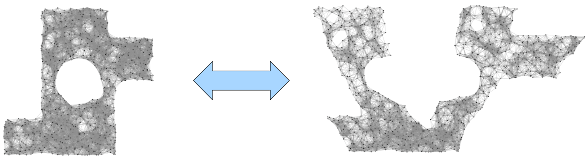
(a) Parabolic distortion of square



(b) 3D curl distortion of square



(c) Parabolic distortion of complex space



(d) Splitting distortion of complex space

Fig. 3 Four distortions used to validate the distortion tolerance of manifold geometric programs expressed in Proto.

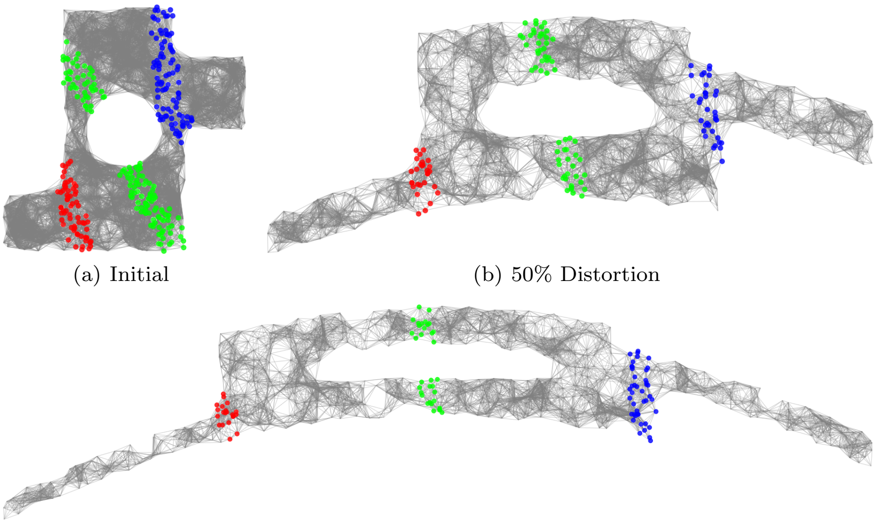
The patterns we test are selected for their susceptibility to verification via visual inspection. Only representative samples of the images will be shown here; the rest may be found in the Supplementary Information.

4.3 Scalable Patterns

For a scalable pattern, we seek to maintain some long-range relationship across the aggregate, such as a size ratio between regions. Our primary tool here will be measurements of distance through the manifold to the closest point in a designated region. In developing animals, such metrics are often established via chemical gradients. In Proto, this is computed by the `distance-to` function, which implements the fast self-healing `CRF-Gradient` algorithm from Beal et al (2008).

One basic and frequently used pattern formation building block that can built immediately from distance measurements is a bisector:

```
(def bisector (a b epsilon)
  ;; measure distance to source regions
  (let ((d1 (distance-to a)) (d2 (distance-to b)))
```



(a) Initial

(b) 50% Distortion

(c) Full Distortion

Fig. 4 Bisector program executing on a complex shape under parabolic distortion

```
    ;; bisector is points where distances are close to equal
    (< (abs (- d1 d2)) epsilon)))
```

This is a generalized version of bisection, where the inputs `a` and `b` may be regions as well as points. We measure the distance to each source region and compare it. If the two distances are approximately equal (within some specified error `epsilon`), then that device must be part of the bisector.

Figure 4 shows a three-bisector program running on a complex space under distortion:

```
(let ((bb (green (bisector s1 s2 (hood-radius)))))
  (red (bisector bb s1 (hood-radius)))
  (blue (bisector bb s2 (hood-radius))))
```

where `s1` and `s2` are the set of all devices within 5 units of the left and right edges of the space and `hood-radius` is the distance over which devices can communicate with one another. This program creates a green line halfway between the edges, a red line halfway between the green and left edge, and a blue line halfway between the green and right edge. Notice how all three bisector lines shift their position as the space distorts, with the green center line making the most radical shift.

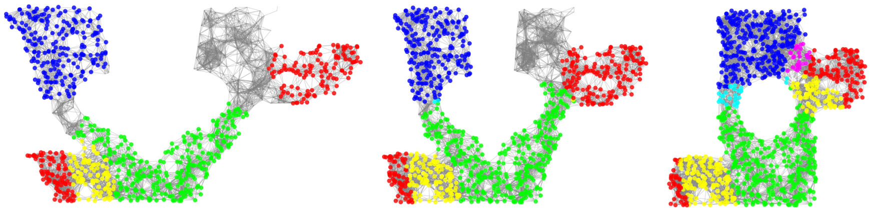
Another simple example is dilation:

```
(def dilate (source radius)
  (< (distance-to source) radius))
```

This program simply selects the region of all devices within `radius` units of a source point by measuring distance to the source and comparing against the radius input.

Figure 5 shows a pattern formed by four applications of dilation:

```
(all
  (red (dilate (or s1 s2) 30))
  (green (dilate s4 50))
  (blue (dilate s3 40)))
```



(a) Full Distortion (b) 50% Reverse Distortion (c) Final

Fig. 5 Dilation program executing on a complex shape under splitting distortion

where `s1`, `s2`, `s3`, and `s4` are four designated devices. This creates four circles of color, which are shown blended together (e.g. the green/red overlap is yellow). Notice that when the space reconnects, the blue circle spreads into the portion that has become adjacent.

A somewhat more complicated example is maintaining the shortest path between two points:

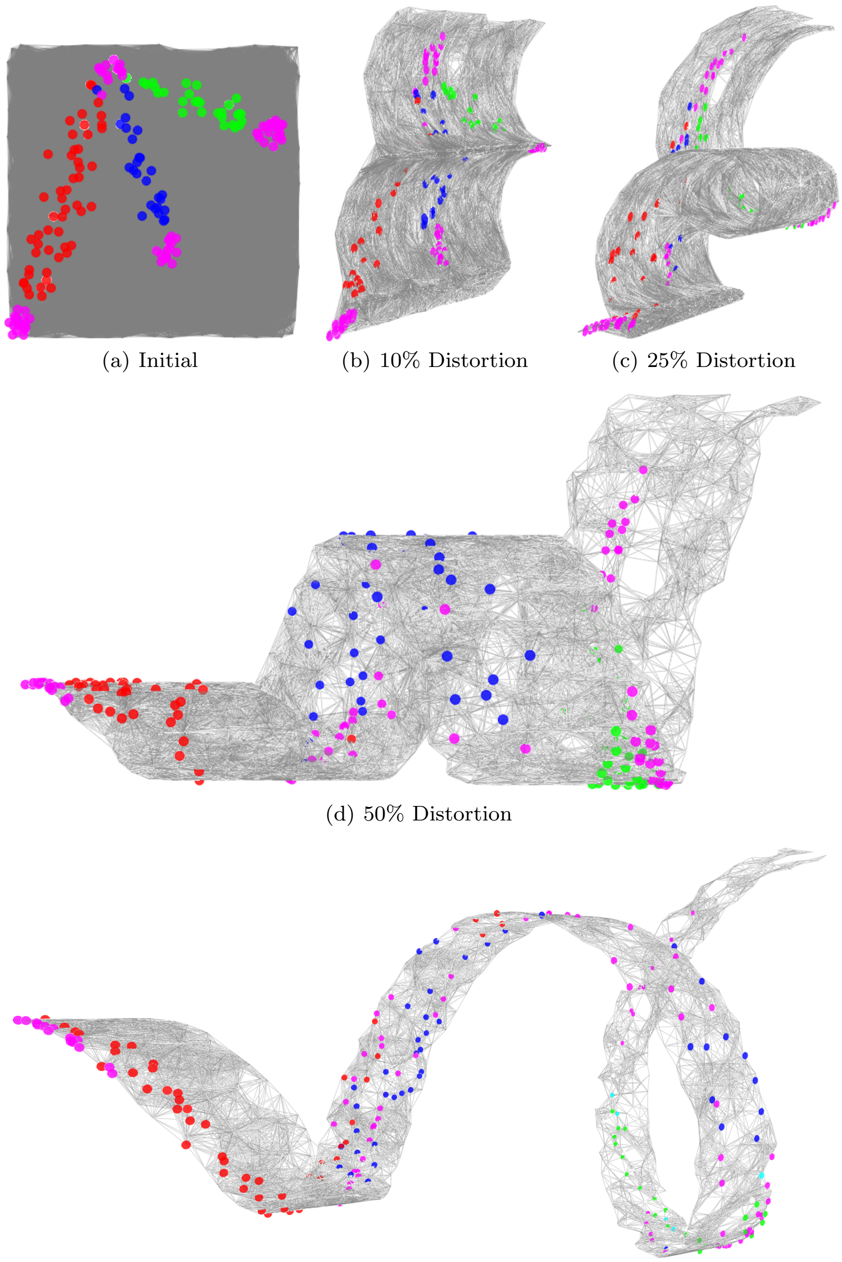
```
(def shortestpath (a b)
  ;; compute distance to b, with moderate permissiveness
  (let ((d (flex-gradient b 0.3 0.2 0.5)))
    (let ((path    ;; is this device on the shortest path?
           (> (rep next    ;; each device on the path points to the next
                   -1      ;; -1 means not on the path
                   ;; if you are a, or are pointed to by a neighbor
                   (mux (muxor a (any-hood (= (nbr next) (mid))))
                        ;; then choose the next device on the path
                        (2nd (min-hood (tup (if (= (nbr-range) 0)
                                                (inf)
                                                (nbr d))
                                            (nbr (mid)))))
                        -1)) ;; otherwise you aren't on the path
              -1)))
      path)))
```

For this program, we use the `Flex-Gradient` algorithm (Beal, 2009), which is designed to adjust its estimates more smoothly than `CRF-Gradient`. Devices then create a sequence of pointers descending the distance slow as fast as possible in order to induce the the shortest path.

Figure 6 shows a pattern created from shortest-path operations, evaluated on a square distorting into a three-dimensional curl:

```
(all
  (red (shortestpath s1 s3))
  (green (shortestpath s2 s3))
  (blue (shortestpath s4 s3))
  (if (or (or s1 s2) (or s3 s4))
    (rgb (tup 1 0 1)) ;; paint end-points magenta
    (tup 0 0 0))))
```

where `s1`, `s2`, `s3`, and `s4` are four designated regions (shown as magenta). This creates three paths, one red, one blue, and one green, all converging on the `s3` region, originally at the top of the square; since this is region-to-region shortest path computation, there are multiple paths formed. Notice that early in the distortion when the surface interpenetrates, the red and blue shortest paths pass beneath the forming loop. Once the curl has unrolled, however, they pass along the surface, frequently overlapping to form magenta points.

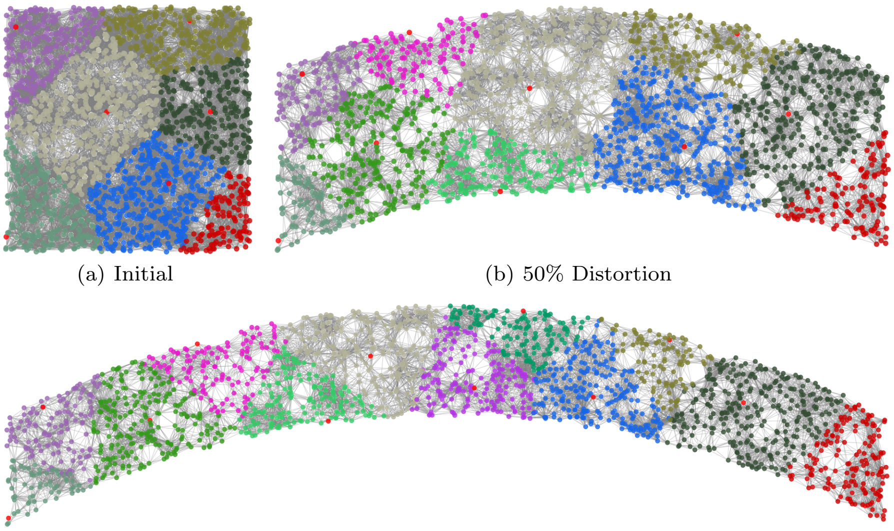


(a) Initial (b) 10% Distortion (c) 25% Distortion

(d) 50% Distortion

(e) Full Distortion

Fig. 6 Shortest path program executing on a square under 3D curl distortion, rotated to best show paths.



(a) Initial

(b) 50% Distortion

(c) Full Distortion

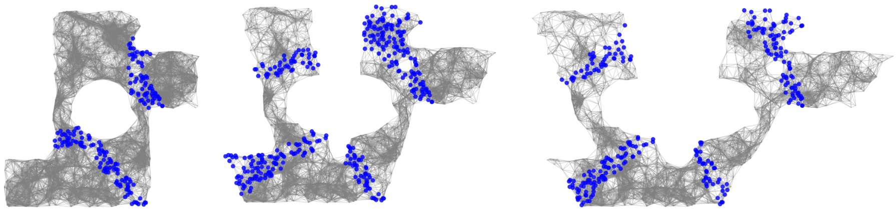
Fig. 7 Symmetry break program executing on a square under parabolic distortion

4.4 Tiled Patterns

For tiled patterns, we seek to maintain a uniform packing of patterns elements of approximately the same size. Our basic tool here will be a symmetry-breaking operation that breaks the space into regions of a controlled diameter. When a region grows too large, it will split into sub-regions; when a region grows too small, it will be absorbed into the regions surrounding it. We can implement a simple version of such a symmetry-break by combining sparse randomness and distance measurement:

```
(def symmetry-break (radius)
  ;; Track two variables: region ID and region leaders
  (rep (tup center id)
   (tup 0 -1) ;; initially, nothing is a center or a region
   (let* ((d (distance-to center))
          ;; a region is too small if its edge is less then 1/2 radius
          (too-close (dilate (any-hood (not (= id (nbr id))))
                             (- (* radius 0.5) (hood-radius)))))
     (tup
      (mux (and too-close center) ;; when too close, regions randomly die
        (< (rnd 0 1) (probe (p_keep radius) 0))
        (mux (> d (* radius 1.5)) ;; when too far, randomly seed new regions
          (< (rnd 0 1) (probe (p_flip radius) 1))
          center))
      (broadcast center (mid)))))) ;; designate region w. UID
```

This program basically works by combining distance measurement with sparse randomness to generate a random Voronoi partition. When a region is too big, devices far from the center flip a coin to decide whether to become the center of a new region, weighted so that collisions between newly generated regions will be rare: $p_{flip} = \frac{2ca}{\pi r^3}$, where c is the expected speed of information flow and r is `radius` and a is the area covered by this device. Two regions are too tightly packed together, as measured by the distance to its edges, the center flips a coin weighted so that it is unlikely that both "too-close" regions will die at once: $p_{keep} = 1 - \frac{c}{10r}$. Figure 7 shows regions generated by this process, with RGB color value set from the region id; note how their shapes change and they increase in number from 7 to 12 as the space expands under distortion.



(a) Initial (b) 50% Distortion (c) Full Distortion

Fig. 8 Tiled bisector program executing on a complex shape under splitting distortion

Once we have broken our space into regions, we can tile a pattern across it by constructing a scalable pattern in each tile. The pattern must be scalable because the regions that we have defined are irregular. For example, this program runs a bisector between centers and edges on tiles with a characteristic radius of 50 units:

```
(let* ((region (symmetry-break 50))
       ;; an edge is any device with a neighbor in another region
       (edges (any-hood (not (= (2nd region) (nbr (2nd region))))))
       ;; expand center to match edge thickness
       (fat-center (any-hood (nbr (1st region)))))
  (blue (bisector fat-center edges)))
```

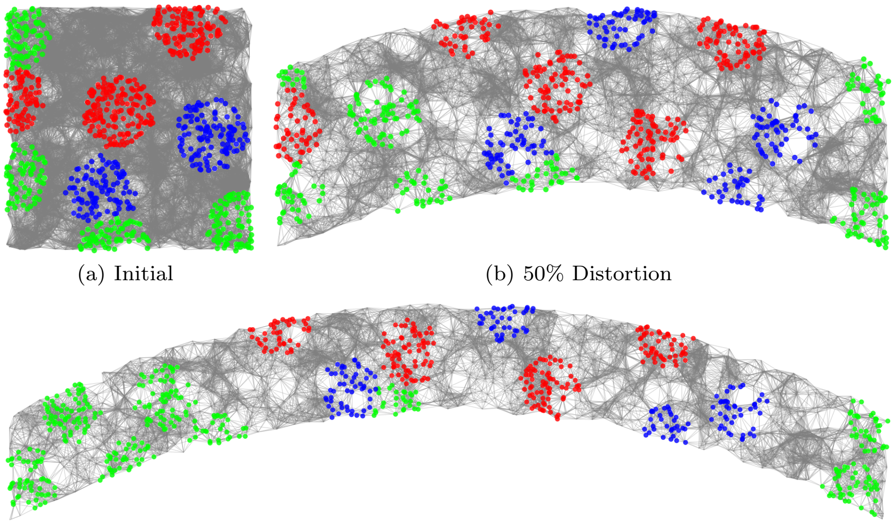
Figure 8 shows this pattern running on an irregular space under a splitting distortion. Notice how after the top space splits, another region has formed and the bisectors have shifted to reflect the changed space on which they are executing.

One final example: using a symmetry break to create random red, green, or blue polka dots:

```
(let ((center (1st (symmetry-break 30))))
  ;; form a polka dot around each region center
  (if (dilate center 15)
      ;; have the center pick a color and sent it to the whole dot
      (let ((color (broadcast center (once (floor (rnd 0 3))))))
        (rgb (tup (= color 0) (= color 1) (= color 2))))
      (tup 0 0)))
```

Figure 9 shows this pattern running on a square space under parabolic distortion. Notice how the dots move with the space and multiply as it grows, but the left-most red dot disappears as the space shrinks vertically and it loses out to the neighboring green dots.

These are only a few of the wide variety of structures that can be specified as a combination of scalable and tiled patterns. They are not, of course, sufficient on their own to specify everything we are likely to wish to construct using functional blueprints. What these examples demonstrate, however, is that complex geometry-based programs can derive their adaptivity from the adaptivity of fundamental geometric operations like distance measurement and symmetry breaking, thereby simplifying the challenge of encoding functional blueprints.



(a) Initial (b) 50% Distortion

(c) Full Distortion

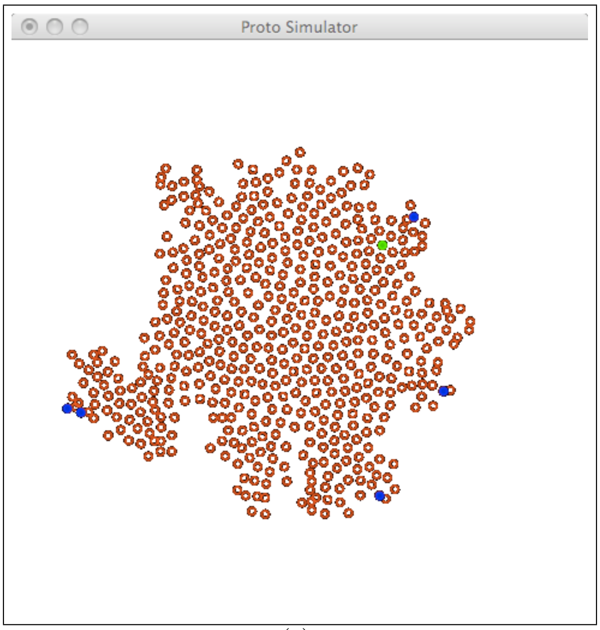
Fig. 9 Polka dot program executing on a square under parabolic distortion

5 Example Application: Tissue Growth

Having proposed a framework for the design of grown systems, and selected a manifold geometric programming model to simplify pattern specification, let us now demonstrate the feasibility of functional blueprints by developing a simplified model of tissue growth, in which the growth of a sheet of cells is synchronized with the growth of the blood vessels that supply them with oxygen. This example system should not be regarded as a serious model of tissue growth, but as a cartoon to demonstrate the feasibility of the engineering approach under discussion.

This simplified model consists of two subsystems, each specified with a functional blueprint, and implemented in Proto. The cell density subsystem attempts to keep cells packed at a moderate density via motion, reproduction, and apoptosis. A consequence of this density maintenance is tissue growth at an approximately constant rate of expansion: cells at the surface of the tissue generally have a low average density, since there are no cells to one side of them, so unless they are regulated otherwise, they will tend to reproduce. The vascularization subsystem, on the other hand, attempts to ensure that no cell is too distant from a network conveying oxygenated blood outward from a source.[3] These two systems are linked together by adding a regulatory input to the cell density subsystem, such that cells will not attempt to reproduce if they are oxygen-starved. The resulting composite system produces smoothly synchronized tissue growth, and can be modulated to produce shaped tissue by external regulation of either subsystem.

[3] In this simplified system, venous return is not modeled, but could be implemented using a complementary mechanism.



(a)

Fig. 10 Density maintenance and growth: simulation of an expanding sheet of cells, where blue cells are reproducing and green are dying.

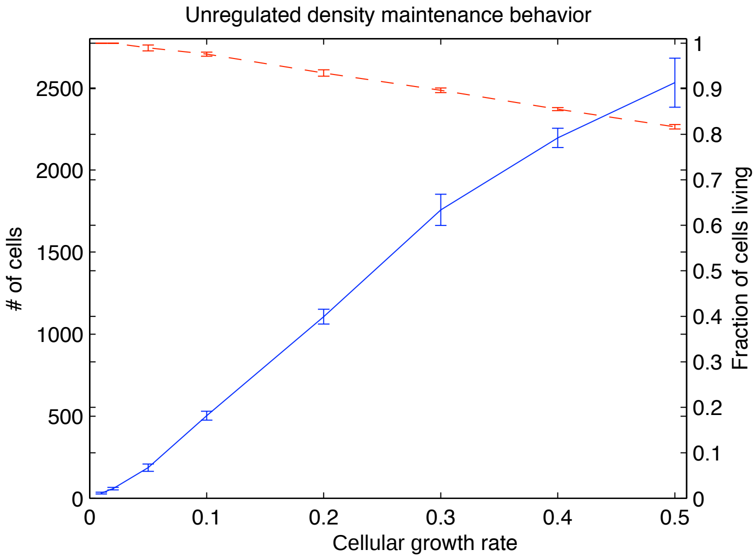
5.1 Cell Density

In this simplified model, the base structure and expansion of a sheet of cells is produced by a system that attempts to keep cells packed at a moderate density. We can implement such a system as follows:

```
(def cell-density (grow shrink p_d)
  (let ((packing (num-nbrs)))
    (clone (and (and (< packing 8)  ; If too few neigbbors
                     grow)          ; and allowed to grow
                (< (rnd 0 1) p_d))) ; randomly reproduce
    (die (and (or (> packing 15)   ; If too many neighbors,
                  shrink)          ; or commanded to shrink
              (< (rnd 0 1) p_d)))) ; randomly suicide
  (disperse 9))  ; Move away from other nearby cells
```

Here the desired system behavior is to maintain a moderate spacing between cells, which exhibits graceful degradation if the cells have some tolerance for overcrowding or underpopulation. The system is thus stressed when there are too many neighbors (here defined as more than 15), too few neighbors (here defined as less than 8), or if the neighbors aren't at a desired separation (here defined as 0.6 communication radii).

The incremental program relieves stress in a straightforward manner: when there are too many neighbors, the cell apoptoses (dies) with probability p_d, and when there are too few neighbors, the cell reproduces with probability p_d (the *grow* enabling input and *shrink* forcing input allows these actions to be modulated by an enclosing system).



(a)

Fig. 11 Behavior of density maintenance and growth system with respect to the growth rate parameter p_d. The number of cells after 200s of growth (blue) is small at low p_d, but the fraction of cells surviving (red) begins to drop at high p_d. Thus the best system behavior is at moderate p_d, but viable behavior spans a large range of values.

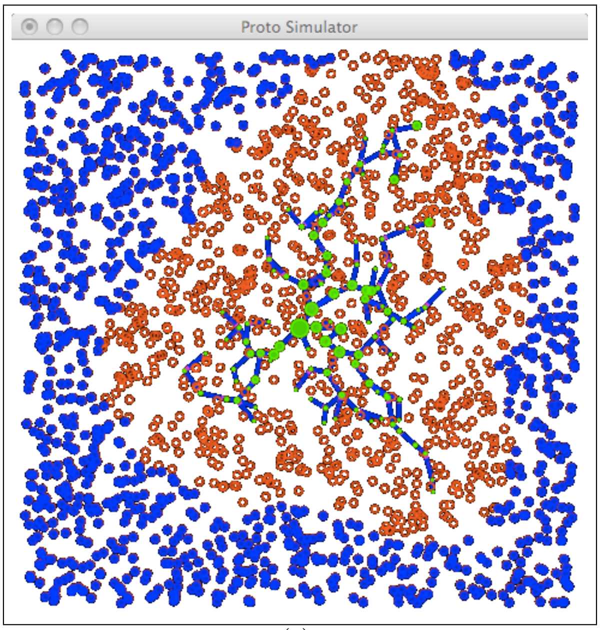
When the neighbors are not at a desired separation, they move towards it using spring forces:

```
(def disperse (preferred-distance)
  (* (/ 1 (int-hood 1)) ; normalize over neighbors...
     (int-hood
      ;; the difference from preferred distance
      (let ((dr (- (nbr-range) preferred-distance)))
        ;; (multiplied by 1/10 when attracting)
        (* (mux (< dr 0) dr (* 0.1 dr))
           ;; turning distance into a vector to the neighbor
           (normalize (nbr-vec)))))))
```

in which attractive forces are weaker than repulsive forces such that the farther neighbors do not exert too much influence and collapse the diameter of communicating clusters.

Note that since cells at the surface of the sheet have an expected density half that of cells in the interior of the sheet, their density will be considered too low (except in temporary high-density pockets) and they will reproduce. This has the desirable consequence of continually expanding the sheet of cells such that the edge moves outward at an expected constant rate (Figure 10).

Figure 11 shows this cell density system in experiments where the growth parameter p_d ranges from 0.01 to 0.5. For each parameter value, 10 trials were run, beginning with 10 cells distributed in a volume 10 units square and continuing for 200 simulated seconds. As the growth rate p_d rises, the final number of cells in the system (blue solid line) rises sharply, but the fraction of cells that die rises as well (red dashed line shows



(a)

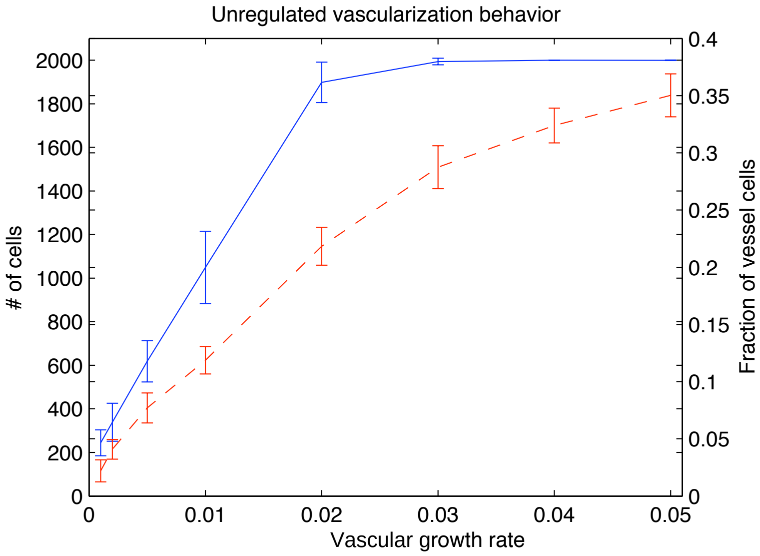
Fig. 12 Vascularization: simulation of an expanding vascular network (green dots, blue lines) expanding the area of oxygenation (red) into a sheet of underoxygenated cells (blue). The area of a dot is proportional to the size of the network descending from it.

surviving cells). An intermediate level in the range 0.1 to 0.3 appears to offer the best trade-off, with graceful degradation as the parameter moves away from that level.

5.2 Vascularization

Oxygen delivery by a vascular system requires that there be a capillary vessel relatively close to every cell. When this is not the case, the cell becomes stressed by lack of sufficient oxygen—a graceful degradation situation since the cell does not die. The simple vascularization system here measures stress by distance to the nearest vessel:

```
(def vascularize (source service-range p_v)
  (rep (tup vessel served parent) ; three state variables
       ;; the source starts as a vessel; everthing else doesn't
       (tup source source (if source (mid) -1))
       (mux source
         (tup 1 1 -1) ; source is always a parentless vessel
         (let ((service (< (distance-to vessel) service-range))
               (server (gradcast vessel (mid)))
               (children (sum-hood (= (mid) (nbr parent))))
               (total-children (tree-children parent)))
           ;; adjust radius, for visualization
           (radius-set
            (mux vessel (* 0.5 (sqrt (+ 1 total-children))) 2))
           ;; grow/shrink vessel network
           (mux vessel
```



(a)

Fig. 13 Behavior of vascularization system with respect to the growth rate parameter p_v. The oxygenated area (blue solid line) expands slowly at low p_v, but the fraction of cells incorporated into vessels (red dashed line) is large at high p_v. Thus the best system behavior is at moderate p_v, but viable behavior spans a large range of values.

```
;; if a cell is already a vessel, then:
(mux (or (muxand
          ;; if there is too much branching...
          (any-hood
           (and (= (nbr (mid)) parent)
                (> (nbr children)
                   (mux (nbr source) 6 3))))
          ;; ... and there is no smaller branch
          (not (any-hood
                (< (nbr total-children)
                   total-children))))
         ;; or if the path to the source is broken
         (not (any-hood
               (and (nbr vessel)
                    (= (nbr (mid)) parent)))))
  (tup 0 1 -1)    ; ... then vessel is discarded
  (tup 1 1 parent)) ; else vessels stay fixed
;; if a cell is not a vessel, then:
(mux (muxand
      (muxand
       ;; if some neighbor is a vessel
       (any-hood (nbr vessel))
       ;; and there is need of more vessels
       (dilate (not served) service-range))
      ;; and a coin-flip comes up heads
      (< (rnd 0 1) p_v)))
  (tup 1 1 server) ; then become a vessel
  (tup 0 service -1))))))) ; else stay non-vessel
```

where `tree-children` is a simple program counting each cell's number of descendants in the vascular tree:

```
(def tree-children (parent) ; given each cell's parents:
  (rep nc          ; 'nc' is the estimated number of children,
       0           ; which starts each cell as a leaf
       (sum-hood                 ; summing over neighbors,
         (if (= (nbr parent) (mid)) ; if a neighbor is a child,
             (+ 1 (nbr nc))         ; add it and its chilren;
             0))))                  ; otherwise, nothing
```

Every cell tracks whether it is part of a vessel and, if not, whether it has *service* from a vessel within the *service-range*, In the beginning, only the *source* cell(s) are part of a vessel. Later, the incremental program adds or removes cells from vessels to incrementally adjust the network. Cells join a vessel at a growth rate p_v when adjacent to a vessel and in range of an unserved cell. Vessel cells undifferentiate when they lose their connection to the source or when too many other vessel cells share the same junction.

Figure 12 shows a snapshot of typical simulated behavior, visualized using:

```
(def drawvasc (v)
  (let ((vessel (1st v)) (served (2nd v)) (parent (3rd v)))
    (green vessel)           ; green for vessel cells
    (blue (not served))      ; blue for oxygen-starved cells
    ;; line to vessel cell's parent
    (sum-hood (* (= (nbr (mid)) (3rd v)) (nbr-vec)))))
```

Figure 13 shows the vascularization system in experiments where the growth parameter p_v ranges from 0.001 to 0.05. For each parameter value, 10 trials were run, where the network is grown for 200 simulated seconds from a seed point in the middle of a network of 2000 devices and devices are distributed on a square 300 by 300 units with a vascularization service range of 50 units. The higher the growth rate p_v, the faster that vascularization proceeds and therefore the larger an area that is served (blue solid line). The faster that vascularization proceeds, however, the more redundancy in the system, as reflected by the fraction of cells designated as vessels (red dashed line).

5.3 Composite Behavior

These two subsystems can be linked together into a simplified model of tissue growth by the simple expedient of enabling growth in the cell density system only for those cells served by vascularization:

```
(def tissue (src pd pv)
  ;; run vascularization and density maintenance
  (let* ((v (vascularize src 50 pv))
         (dir (cell-density (2nd v) 0 pd)))
    ;; allow all but vascularization source to move
    (if (not src) (mov dir) (tup 0 0 0))
    ;;  visualize vascularization
    (drawvasc v)))
```

Having implementing these two subsystems using functional blueprints, this simple coupling suffices for them to regulate one another into synchronized growth.

Figure 15 compares regulated behavior (blue line) with unregulated subsystems (red dashes), showing a smooth shift in regulatory dominance of the coupled system

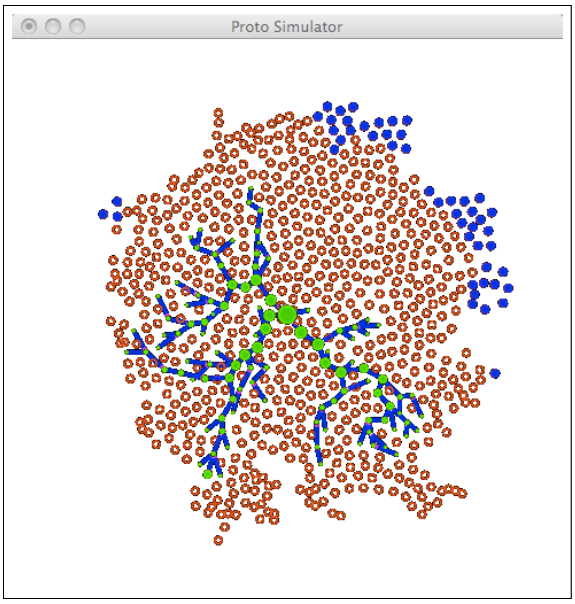


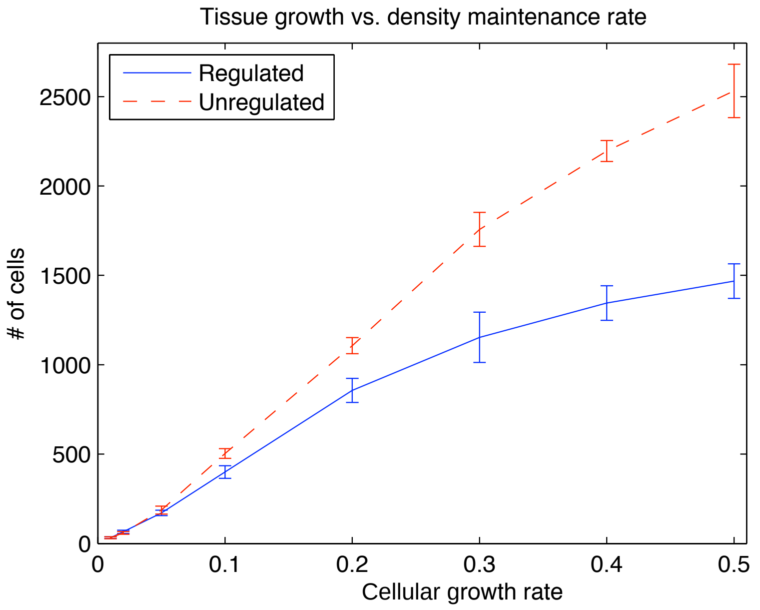
Fig. 14 Synchronized growth of a tissue: growth from cell density maintenance is enabled only for cells served by vascularization.

as p_d and p_v are varied. For each set of parameter values, 10 trials were run, beginning with 10 cells distributed in a volume 10 units square and continuing for 200 simulated seconds. In Figure 15(a), p_d is varied from 0.01 to 0.5 as above, while p_v is held constant at 0.02. At low values of p_d, growth from density maintenance dominates, but as p_d rises, cells spread outward faster and their growth begins to be checked by the rate of vascularization instead. In Figure 15(b), p_v is varied from 0.001 to 0.05 as above, while p_d is held constant at 0.1. At low values of p_v, vascularization is the limiting factor, but by $p_v = 0.02$ the limiting factory has shifted to the rate of growth from density maintenance.

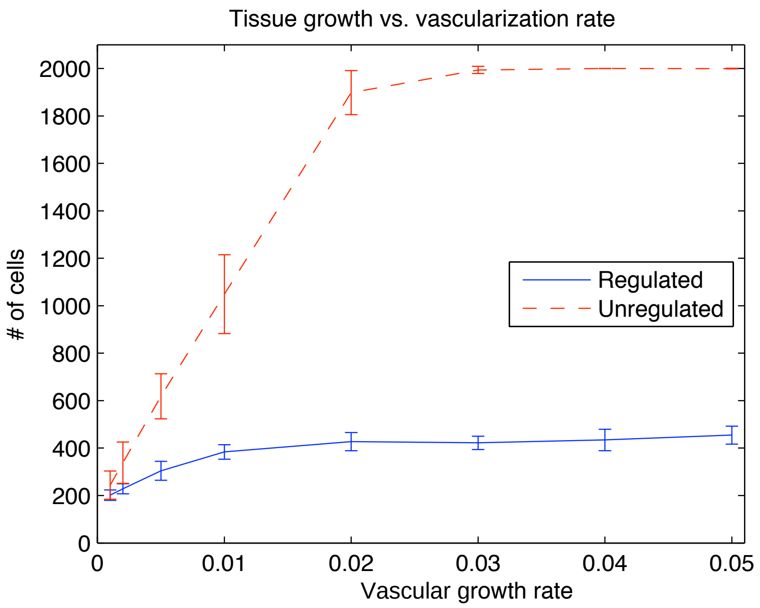
This composite system may itself be viewed in terms of a functional blueprint, as these results illustrate, where both density and vascularization are being maintained in the face of stress, and the failure of either checks the other's progress. Moreover, just as the cell density subsystem was modulated to form a growing tissue, so may the tissue be modulated to grow complex shapes. This can be done by modulating either the cell density subsystem or the vascularization subsystem. To modulate shape using the cell density subsystem, growth is enabled for cells within the constraints of the shape and apoptosis for cells outside of the shape; vascularization modulation is done similarly, but shrinks the target shape by the service range of vascularization.

For example, Figure 16 shows the result of constructing a letter "F" through regulating cell density (Figure 16(a)) and through regulating vascularization (Figure 16(b)).[4] Moreover, the functional blueprint separates the result of modulated tissue growth from the details of its execution, as illustrated by the equivalent constructions in Figure 16(a) and Figure 17.

[4] For simplicity in this demonstration, the "F" bounds are set by external localization, though it could be self-organized with variety methods (see Doursat (2008), Kondacs (2003)).

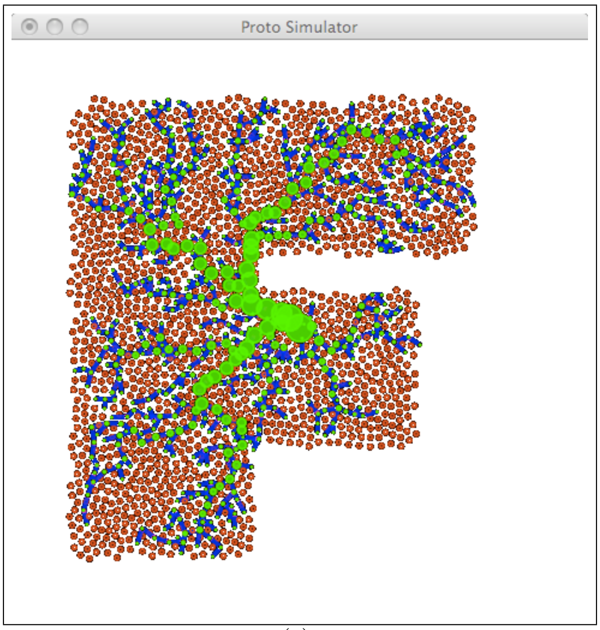


(a)

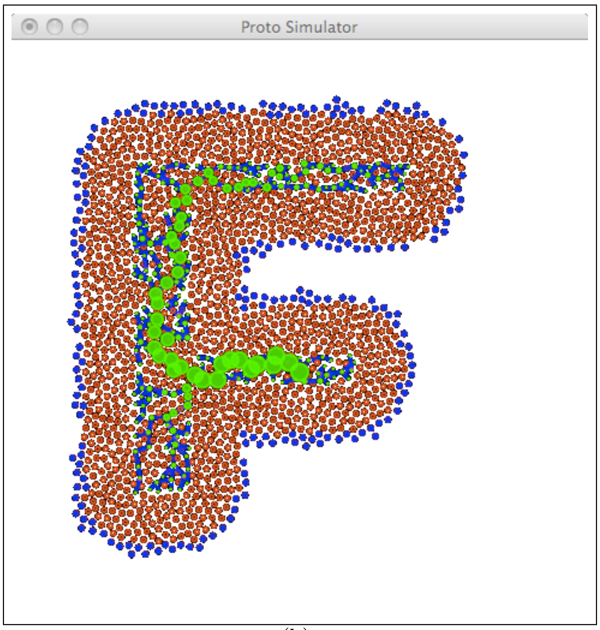


(b)

Fig. 15 Linking density maintenance and vascularization results in synchronized tissue growth, with either subsystem able the regulate the behavior of the composite system.



(a)



(b)

Fig. 16 The tissue growth system can be modulated to produce complex patterns, such as the letter "F", by modulating growth of either the cell density (a) or vascularization subsystems (b). In both cases shown, the letter grows from a seed near its center.

6 Contributions and Discussion

We have demonstrated that a *functional blueprint* approach can be used to create grown system that are dynamically integrated and that smoothly transfer regulatory control across regimes. These systems can be interconnected to form composite systems with the same properties of dynamic integration and smooth transfer of regulatory control.

While this is early work, the simplicity of creating and integrating the models discussed in this paper indicates good potential for further development. Indeed, it is hoped that the reader will find most value not in the results that have been presented, but in the problems they open up and the hints that they offer toward how these problems might be addressed.

Take, for example, the formal definition of functional blueprints. The definition set forth in this paper is conservative, constraining the set of possible blueprint designs as little as possible. Intuitive definitions, such as "graceful degradation," are much stronger than the weak formal definition given: "graceful degradation" implies notions like wide margins of error, gentle slopes, and safe misbehavior that are simply not implied by the definition given. Important questions for the implementation and practicality of functional blueprints are thus left open. For example, we have seen that it is always possible to navigate a viability space by means of incremental changes to individual subsystems, but there is no bound established on how difficult it is to find such a path or the complexity of navigating it. Moreover, it will be important to ensure acceptable complexity for composed systems as well. The tissue growth example gives us one hint: its set of possible viable configurations is both large and well connected, and may be key to allowing the system to be driven so quickly by independent choices of individual elements.

Likewise, when there are many subsystems interacting, or when a subsystem can attempt to relieve stress in multiple ways, what ways of mixing those options together are likely to produce good navigation of viability space? Once again, we may take a hint from the tissue example: density maintenance uses two mechanisms simulataneously: movement and reproduction/apoptosis. If either is run too strongly, behavior becomes incoherent. Perhaps then, an appropriate means of combining incremental adjustments together is some form of vector normalization, such that closely related adjustments reduce one anothers' actions strongly while distantly related adjustments mute more weakly.

At present, however, these are but early hints. More work is required, both theoretical and practical, to determine where the functional blueprint concept can be applied well, and what are good practices for doing so. Ultimately, we hope that the decoupling of structure from the program guiding development will lead to more adaptivity in engineered systems as well as stronger biological models for evolvability and phenotypic adaptation.

References

Aubin JP (1991) Viability theory. Birkhauser, Basel, Switzerland

Basu S, Gerchman Y, Collins CH, Arnold FH, Weiss R (2005) A synthetic multicellular systems for programmed pattern formation. Nature 434:1130–1134

Beal J (2004) Programming an amorphous computational medium. In: Unconventional Programming Paradigms International Workshop, Springer Berlin, Lecture Notes in Computer Science, vol 3566, pp 121–136

Beal J (2009) Flexible self-healing gradients. In: ACM Symposium on Applied Computing, ACM, New York, NY, USA, pp 1197–1201

Beal J, Bachrach J (2006) Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intelligent Systems 21:10–19

Beal J, Bachrach J, Vickery D, Tobenkin M (2008) Fast self-healing gradients. In: ACM Symposium on Applied Computing, ACM, New York, NY, USA

Carmeliet P (2003) Angiogenesis in health and disease. Nature Medicine 9(6):653–660

Carroll SB (2005) Endless Forms Most Beautiful: The New Science of Evo Devo and the Making of the Animal Kingdom. W. W. Norton & Company, New York, NY, USA

Coore D (1999) Botanical computing: A developmental approach to generating inter connect topologies on an amorphous computer. PhD thesis, MIT, Cambridge, MA, USA

Couzin I, Krause J, Franks N, Levin S (2005) Effective leadership and decision making in animal groups on the move. Nature 433:513–516

Dorigo M, Birattari M (2007) Swarm intelligence. Scholarpedia 2(9):1462

Dorigo M, Stutzle T (2004) Ant Colony Optimization. MIT Press, Cambridge, MA

Doursat R (2008) The growing canvas of biological development: Multiscale pattern generation on an expanding lattice of gene regulatory networks. In: Unifying Themes in Complex Systems Vol VI: Proceedings of the 6th International Conference on Complex Systems, Springer-Verlag, New York, NY, USA

Kirschner MW, Norton JC (2005) The Plausibility of Life: Resolving Darwin's Dilemma. Yale University Press, New Haven, CT, USA

Kondacs A (2003) Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In: International Joint Conference on Artificial Intelligence (IJCAI), AAAI, Menlo Park, CA, USA

MIT Proto (Retrieved November 22, 2010) MIT Proto. software available at `http://proto.bbn.com/`

Nagpal R (2001) Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics. PhD thesis, MIT, Cambridge, MA, USA

O'Grady R, Christensen A, Dorigo M (2009) Swarmorph: Multi-robot morphogenesis using directional self-assembly. IEEE Transactions on Robotics 25(3):738–743

O'Grady R, Christensen AL, Pinciroli C, Dorigo M (2010) Robots autonomously self-assemble into dedicated morphologies to solve different tasks. In: Proceedings of 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), IFAAMAS, Toronto, Canada, pp 1517–1518

Prusinkiewicz P, Lindenmayer A (1990) The Algorithmic Beauty of Plants. Springer-Verlag, New York, NY, USA

Reynolds C (1987) Flocks, herds, and schools: A distributed behavioral model. Computer Craphics (SIGGRAPH '87 Conference Proceedings) 21(4):25–34

Shetty RP, Endy D, Thomas F Knight J (2008) Engineering biobrick vectors from biobrick parts. Journal of Biological Engineering 2(5):no page numbers

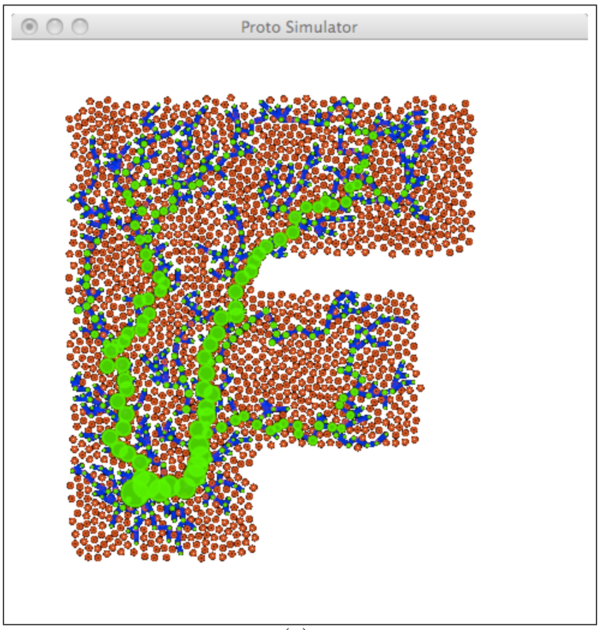
Spicher A, Michel O (2006) Declarative modeling of a neurulation-like process. BioSystems 87:281–288

Stoy K, Nagpal R (2004) Self-reconfiguration using directed growth. In: Intl. Symposium on Distributed Autonomous Robotic Systems (DARS), Springer-Verlag, New York, NY, USA

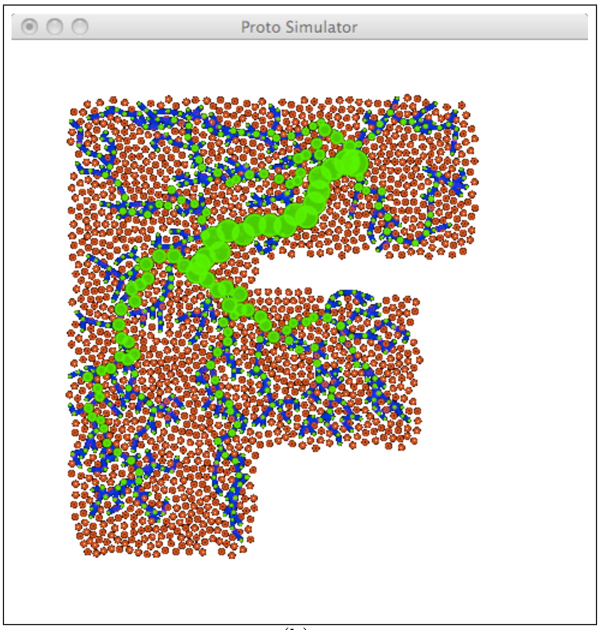
Werfel J (2006) Anthills built to order: Automating construction with artificial swarms. PhD thesis, MIT, Cambridge, MA, USA

Werfel J, Nagpal R (2007) Collective construction of environmentally-adaptive structures. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007), IEEE, Piscataway, NJ, USA

Yamins D (2007) A theory of local-to-global algorithms for one-dimensional spatial mu lti-agent systems. PhD thesis, Harvard, Cambridge, MA, USA



(a)



(b)

Fig. 17 A functional blueprint separates the result of modulated tissue growth from the details of its execution, as shown by equivalent constructions of the letter “F” when grown from a seed in the lower right (a), upper left (b) or center (Figure 16(a)).