Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2005-003
AIM-2005-001

January 18, 2005

Biologically-Inspired Robust Spatial Programming

Jacob Beal and Gerald Sussman

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Abstract

Inspired by the robustness and flexibility of biological systems, we are developing linguistic and programming tools to allow us to program spatial systems populated by vast numbers of unreliable components interconnected in unknown, irregular, and time-varying ways. We organize our computations around geometry, making the fact that our system is made up of discrete individuals implicit. Geometry allows us to specify requirements in terms of the behavior of the space occupied by the aggregate rather than the behavior of individuals, thereby decreasing complexity. So we describe the behavior of space explicitly, abstracting away the discrete nature of the components. As an example, we present the Amorphous Medium Language, which describes behavior in terms of homeostatic maintenance of constraints on nested regions of space.

Increasingly, we are faced with the prospect of programming spatially embedded mesh networks composed of huge numbers of unreliable parts. Projects in such diverse fields as sensor networks (e.g. NEST [12]), peer-to-peer wireless (e.g. RoofNet [2]), smart materials (e.g. smart dusts [16, 26]), and biological computation [18, 30, 31] all envision using networks of large size (ranging from thousands to trillions of nodes) where only nodes nearby in space can communicate directly, and the network as a whole approximates the physical space through which it is deployed.

Biological systems inspire us. Consider, for example, my hand. Huge numbers of unreliable processors (cells) collaborate to form a hand, and most hands are identical in all but the least important ways. Moreover, my hand reliably maintains its structure even in the face of large-scale damage, from cuts to broken bones. Biological systems exhibit great flexibility: with only a small change of parameters the same process that builds a human hand can also make a lion's paw or a bat's wing. We aspire to learn to build systems that have the same level of precision, flexibility, and robustness as the processes that build and maintain our hands.

Our goal is to formulate engineering principles for controlling vast systems of unreliable components connected in unknown, irregular, and time-varying ways. One way to capture engineering principles is to invent programming languages. When we invent a programming language we choose what is explicit and must be addressed by the programmer and what is implicitly represented in the the structure of the language. In order to make some ideas effectively implicit we must understand them thoroughly, and the correct execution of our programs is a powerful test of this understanding.

We consider systems that are densely distributed in ordinary two-dimensional or three- dimensional space, and we assume that computation and local communication is cheap. Thus we can organize our computations around geometry, making the fact that our system is made up of discrete individuals implicit. The geometric abstraction allows us to specify requirements in terms of the behavior of the space occupied by the aggregate rather than the behavior of individuals, thereby decreasing complexity. So we must invent languages that describe the behavior of space explicitly, abstracting away the discrete nature of the components.

Space

The concept of space is a powerful intellectual tool that helps us to separate systems into parts that can work somewhat independently. In particular, things that are "near" each other can interact easily, whereas things that are "far" apart are effectively isolated from each other. To make effective use of this idea, we need to be able to identify particular pieces of space, called regions, to name them, and to be able to attach processes that direct their behavior.

We may have a problem that is naturally embedded in the physical space that we inhabit. For example, we may have to administer and control a farm, distributing resources, such as water, to the areas that need it, and dispatching machinery, such as a harvester, to an appropriate area.

Or perhaps we need to simulate a mechanical structure, such as a bridge, to assess the forces in the members under various load conditions, before the structure is to be built. Unlike the farm example, the computational space is not coextensive with the physical space being simulated, but there is an embedding of the system being simulated in the computational space.

Organization

In Section A we introduce the problem domain in which we will be working. We describe relevant previous work on spatial languages in Section B.1, and robust primitives in Section B.2. Finally in Section C we sketch how robust primitives can be combined to form a robust spatial language, illustrated with simple programming examples.

A Amorphous Computing

The amorphous computing [1] engineering domain presents a set of challenging requirements to the system designer, forcing confrontation of issues of robustness, distribution, and scalability in a spatially embedded network. The constraints of amorphous computing derive much of their inspiration from biological systems engaged in morphogenesis and regeneration.

In an amorphous system the number of devices may be large, anywhere from thousands to millions or even billions. Practically, an algorithm is only reasonable if its per-device asymptotic complexity (e.g. space or bandwidth per device) is polynomial in $\log n$ (where n is the number of devices). Any bound significantly greater than $O(\log n)$ should be treated with considerable suspicion. Unlike ad-hoc networking and sensor networks scenarios, amorphous computing generally assumes cheap energy, local processing, and storage: as long as they do not have a high per-device asymptotic complexity, minimizing them is not of particular interest.

Because there are so many devices, the system must not be very sensitive to care and feeding of the individual devices. In general, it is assumed that every device is identically programmed, but that there can be a small amount of differentiation based on the initial conditions. Once the system is running, however, there is no per-device administration allowed.

The network graph is determined by the spatial distribution of the devices, which only communicate locally. Devices are generally assumed to be immobile unless the space in which they are embedded is moved (e.g. cutting and pasting "smart paper").[1] We model the communication with a simple unit disk model: a bidirectional link exists between two devices if and only if they are less than distance r apart. The network is expected to have a high diameter, and we assume that the time for information to propagate through the network is proportional to the number of hops it needs to travel. The communication complexity is best measured by *maximum communication density*—the maximum bandwidth required by a device—rather than by number of messages.

There are strict limitations on the assumed network infrastructure. The system executes with *partial synchrony*: each device may be assumed to have a clock which ticks regularly, but the clocks may show different times, run at (boundedly) different rates, and have different phases. The system may not assume complex services: no global naming, routing, or coordinate service may be assumed.

In a large, spatially distributed network of devices, failures are not isolated events. In a network with myriad parts, point failures are best measured by an expected rate of device failure. This suggests also that methods of analysis like half-life analysis [21] will be more useful than standard f-failure analysis. Failures may also be geometrically local. For example, a part of the network

[1] Note that mobile devices might be programmed as immobile virtual devices [13, 14].

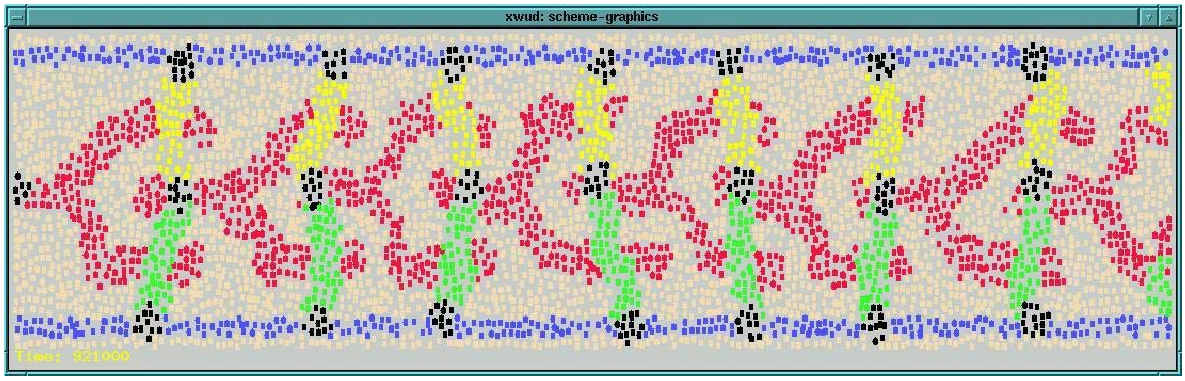


Figure 1: A chain of inverters, from [10], generated on an amorphous medium using GPL.

might be obliterated by a natural disaster or by spilling hot coffee on smart pants. It is reasonable to assume that the frequency of region failure decreases with increasing region size. Maintenance requires recovery or replacement of failed devices: in either case, new devices join the network, either individually or as a region. Finally, we assume that failing subsystems just stop working, not that they actively try to undermine the rest of the network.

B Previous work

There is a solid foundation for this work, both in the form of previous work on languages for programming space, which has exposed many of the challenges involved, and in algorithms which can be used to provide robust primitives for our language development.

B.1 Spatial Languages

Our approach is informed by previous work developing languages to program systems distributed through space.

B.1.1 Sensor Networks Languages

In sensor networks research, a number of high-level programming abstractions have been proposed to enable programming of large mesh networks. For example, GHT [28] provides a hash table abstraction for storing data in the network, and TinyDB [24] focuses on gathering information via query processing. These approaches, however, are data-centric rather than computation-centric, and do not provide guidance on how to do distributed manipulation of data, once gathered.

B.1.2 The Growing Point language

Daniel Coore's Ph.D. thesis [10] demonstrates that amorphous media can be configured by a program that is common to all the computing elements to generate highly complex pre-specified patterns such as the pattern representing the interconnection structure of an arbitrary electrical circuit (See Figure 1).

Coore's strategy is inspired by a botanical metaphor based on growing points and tropisms. To make this strategy explicit, he developed the Growing Point Language (GPL). A growing point is a locus of activity in an amorphous medium. A growing point propagates through the medium

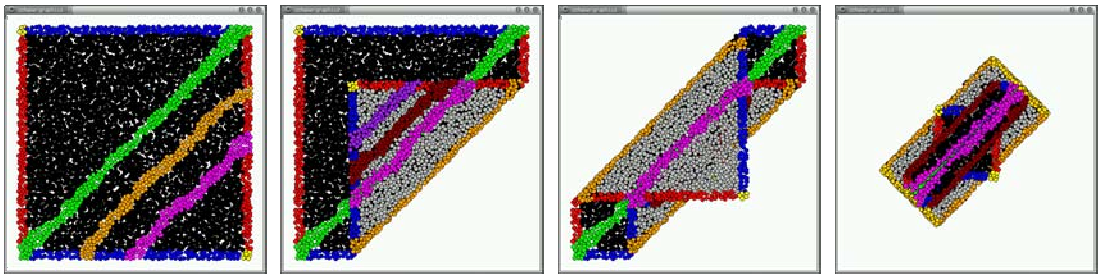


Figure 2: Folding an envelope structure, from [25]. A pattern of lines is constructed according to origami axioms. Elements then coordinate to fold the sheet using an actuation model based on epithelial cell morphogenesis.

by transferring its activity from one computing element to a neighbor. As a growing point passes through the medium it effects the differentiation of the behaviors of the computing elements it visits. The trajectory of the growing point is controlled by signals that are automatically carried through the medium from other differentiated elements. Such a response is called a tropism. In this way a GPL program can exploit locality to make crude geometric inferences.

There are many patterns that are expressible in GPL. Examples include: Euclidean constructions, branching structures and simple text. Coore proves that amorphous media can be programmed to draw any prespecified planar graph, and obtains upper bounds on the amount of storage required by the individual processors to realize such a graph. He also analyzes how the effectiveness of GPL programs depends upon the distribution of the computing elements.

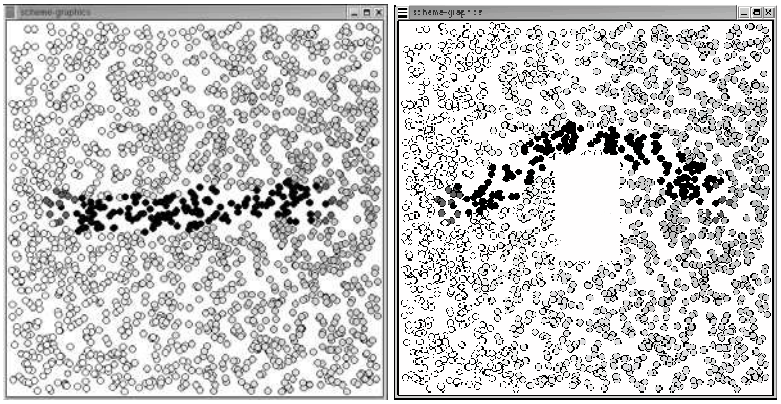
Once a pattern has been constructed, however, there is no clear mechanism to maintain it in the face of changes to the material, and GPL programs are not composable.

B.1.3 Marker propagation for amorphous particles

GPL is formulated in terms of abstractions that must ultimately be implemented by processes in the individual computational particles, which are all programmed identically. Ron Weiss developed a marker propagation language, the Microbial Colony Language [29], for programming the particles. In his model, the program to be executed by each particle is constructed as a set of rules. The state of each particle includes a set of binary markers, and rules are enabled by boolean combinations of the markers. The rules that are enabled are triggered by the receipt of labelled messages from neighboring particles. A rule may set or clear various markers, and it may send further messages. A message carries a count that determines how far it will diffuse, and a marker has a lifetime that determines how long its value lasts. Underlying this model is a runtime system that automatically propagates messages and manages the lifetimes of markers, so that the programmer need not deal with these operations explicitly. Weiss's system is almost powerful enough to represent the processes described by Coore's growing points, yet it is simple enough that it can be implemented in an elementary way, and does not depend on any arithmetic or data structures. The level of abstraction, however, is very low, and is more useful as a tool-set in which to implement higher level languages.

B.1.4 Origami-based Self-Assembly

Radhika Nagpal [25] developed a prototype model for controlling programmable materials. She showed how to organize a program to direct an amorphous sheet of deformable agents to cooperate



(a) Before Failure (b) After Repair

Figure 3: A line being maintained by active gradients, from [8]. A line (black) is constructed between two anchor regions (dark grey) based on the active gradient emitted by the right anchor region (light grays). The line is able to rapidly repair itself following failures because the gradient actively maintains itself.

to construct a large family of globally-specified predetermined shapes. She demonstrated this by presenting a language that allows a programmer to specify a sequence of folds, in a way inspired by Huzita's axioms for origami [17], that achieve the desired global arrangement (See Figure 2). She showed how this language can be compiled into a program that can be distributed to all of the agents. With a few differences of initial state (for example, agents on the edges of the sheet know that they are edge agents) the agents execute their copies of the program, interact with their neighbors, and fold up to make the predetermined shape.

Nagpal's techniques are quite robust. She has investigated and reported on the range of shapes that can be constructed using her method, and on their sensitivity to errors of communication, random cell death, and density of the cells. Once a shape has been constructed, however, there is no clear mechanism to maintain it in the face of changes to the material, and while the origami methods allow composition of shapes, the programs developed from them are only weakly composable.

B.1.5 Self-Repairing Topologies

Lauren Clement and Radhika Nagpal [8] developed an algorithm by which a line can maintain itself and repair damage (See Figure 3). They introduce a notion of active gradients, which are constantly working to maintain an estimate of shortest path from nearby points to the sources of the gradient. A line is formed by selecting two anchors, one of which sources a gradient, while the line climbs up the gradient from its intersection with the other anchor. Maintenance and repair are performed simply by not terminating the construction process. Although Clement and Nagpal propose a language for constructing robust graphs using this primitive, this has not yet been demonstrated.

B.1.6 Regiment

The Regiment language [27] uses a stream-processing abstraction to distribute computation across predicate-defined regions of a space. Regiment is closely related to Haskell, and its top-down design allows it to use the well-established formal semantics of stream-processing. Regiment's robustness

against failure, however, has not yet clearly been established, and there are significant challenges remaining in adapting its programming model to a mesh network environment.

B.2 Robust Primitives

Several existing amorphous computing algorithms will serve as useful primitives for constructing a language. Each mechanism summarized here has been implemented as a code module and demonstrated in simulation. Together they are a powerful toolkit from which primitives for high-level languages can be constructed.

B.2.1 Shared Neighborhood Data

This simple mechanism allows neighboring devices to communicate by means of a shared-memory region, similar to the systems described in [7, 33]. Each device maintains a table of key-value pairs which it wishes to share. Periodically each device transmits its table to its neighbors, informing them that it is still a neighbor and refreshing their view of its shared memory. Conversely, a neighbor is removed from the table if more than a certain time has elapsed since its last refresh. The module can then be queried for the set of neighbors, and the values its neighbors most recently held for any key in its table.

Shared neighborhood data can also be viewed as a sample of the amorphous medium within a unit neighborhood. Aggregate functions of the neighborhood data (e.g. average or maximum) are then approximations of the same functions on a neighborhood of the amorphous medium.

Maintaining shared neighborhood data requires storage and communication density proportional to the amount of data being shared.

B.2.2 Regions

The region module maintains labels for contiguous sets of devices, approximating connected regions of the amorphous medium, using a mechanism similar to that in [32]. A Region is defined by a name and a boolean membership function, describing it in terms of particular places (e.g. “Boston”) which may evolve over time (e.g. “within 10 feet of me”) or in terms of arbitrary predicates (e.g. “where temperature exceeds 50F”) and evolving state (e.g. “where the computation is done”).

When seeded in one or more devices, a Region spreads via shared neighborhood data to all adjoining devices that satisfy the membership test. When a Region is deallocated, a garbage collection mechanism spreads the deallocation throughout the participating devices, attempting to ensure that the defunct Region is removed totally.

Failures or evolving system state may separate a Region into disconnected components. While these are still logically the same Region, and may rejoin into a single connected component in the future, information will not generally pass between disconnected components. As a result, the state of disconnected components of a Region may evolve separately, and in particular garbage collection is only guaranteed to be effective in a connected component of a Region.

Regions are organized into a tree, with every device belonging to the root Region. In order for a device to be a member of a Region, it must also be a member of that Region’s parent in the tree. This implicit compounding of membership tests allows Regions to exhibit stack-like behavior which will be useful for establishing execution scope in a high-level language.

Maintaining Regions requires storage and communication density proportional to the number of Regions being maintained, due to the maintenance of shared neighborhood data. Garbage collecting a Region requires time proportional to the diameter of the Region.

B.2.3 Gossip

The gossip communication module [6] propagates information throughout a Region via shared neighborhood data, after the fashion of flooding algorithms [22]. Gossip is all-to-all communication: each item of gossip has a merge function that combines local state with neighbor information to produce a merged whole. When an item of gossip is garbage-collected, the deallocation propagates slowly to prevent regrowth into areas which have already been garbage-collected.

Gossip requires storage and communication density proportional to the number and size of gossip items being maintained in each Region of which a device is a member, due to the maintenance of shared neighborhood data. Garbage collecting an item of gossip takes time proportional to the diameter of the region.

B.2.4 Consensus and Reduction

In a strong consensus process, every non-failing participating device must choose the same value if any one of them chooses a value, and that value must have been contributed by some device. Reduction is a generalization of consensus in which the chosen value is an aggregate function of values held by the participants (e.g. sum or average).

The Paxos consensus algorithm [20] has been demonstrated in an amorphous computing context [6], but it scales badly. A gossip-based algorithm currently under development promises much better results: it appears that running a robust reduction process on a Region may require only storage and communication density logarithmic in the diameter of the Region and time linear in the diameter of the Region.

B.2.5 Read/Write Atomic Objects

Atomic consistency means that all transactions with an object can be viewed as happening in some order, even if they overlap in time. If we designate a Region of an amorphous medium as a read/write atomic object, then reading or writing values at any point in the Region produces a consistent view of the value held by the Region over time.

Using consensus and reduction, quorum-based atomic transactions can be supported by a reconfigurable set of devices [23, 15]. This has been demonstrated in simulation for amorphous computing [6], and scales as the underlying consensus and reduction algorithms do.

B.2.6 Active Gradient

An active gradient [9, 11, 8] maintains a hop-count upwards from its source or sources—a set of devices which declare themselves to have count value zero—giving an approximation of radial distance in the amorphous medium. This is useful for establishing Regions. Points in the gradient converge to the minimum hop-count and repair their values when they become invalid. The gradient runs within a Region, and may be further bounded (e.g. with a maximum number of hops). When the supporting sources disappear, the gradient is garbage-collected; as in the case of gossip items, the garbage collection propagates slowly to prevent unwanted regrowth into previously garbage collected areas. A gradient may also carry version information, allowing its source to change more smoothly.

Maintaining a gradient requires a constant amount of storage and communication density for devices in range of the gradient, and garbage collecting a gradient takes time linear in the diameter of its extent.

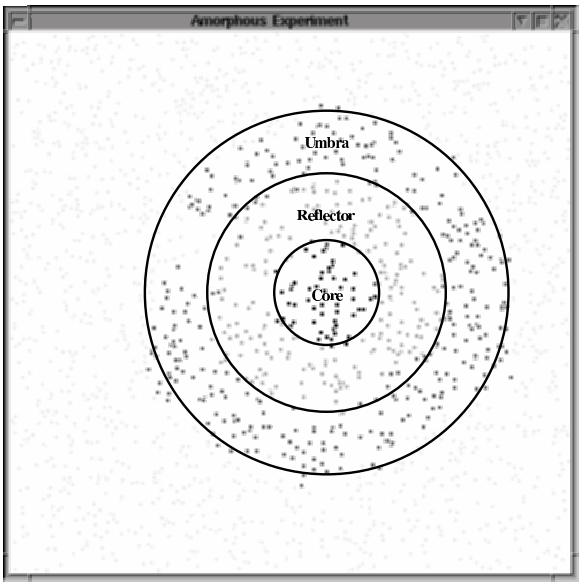


Figure 4: Anatomy of a Persistent Node. The innermost circle is the core (black), which arts as a virtual node. Every device within the middle circle is in the reflector (light grey), which calculates which direction the Persistent Node will move. The outermost circle (dark grey) is the umbra (k=3 in this example), which knows the identity of the Persistent Node, but nothing else.

B.2.7 Persistent Node

A Persistent Node [4] is a robust mobile virtual node based on a versioned gradient that occupies a spherical region of the amorphous medium (See Figure 4). The gradient flows outward from the center, identifying all devices within r hops (the Persistent Node's **core**) as members of the Persistent Node, while a heuristic calculation flows inward from all devices within $2r$ hops (the **reflector**) to determine which direction the center should be moving. The gradient is bounded to kr hops (the **umbra**), so every device in this region is aware of the existence and location of the Persistent Node. If any device in the core survives a failure, the Persistent Node will rebuild itself, although if the failure separates the core into disconnected components, the Persistent Node may be cloned. If the umbras of two clones come in contact, however, they will resolve back into a single Persistent Node via the destruction of one clone.

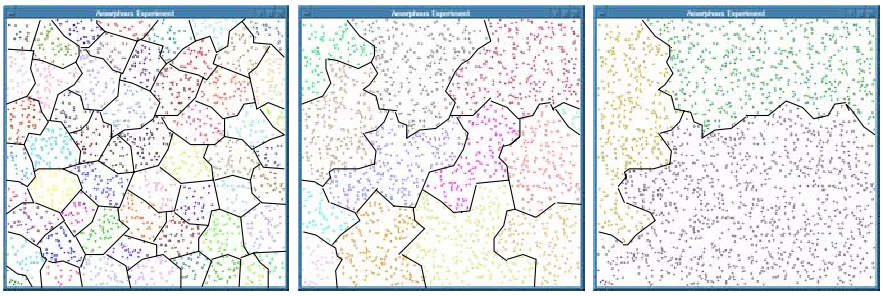
A Persistent Node is a Region whose parent is the Region in which its gradient runs. In addition, a Persistent Node is a read/write object supporting conditionally atomic transactions.

Maintaining a Persistent Node requires storage and communication density linear in the size of the data stored by it. A Persistent Node moves and repairs itself in time linear in its diameter.

B.2.8 Hierarchical Partitioning

A Region can be partitioned through the use of Persistent Nodes: [5] nodes of a characteristic radius are generated and allowed to drift, repelling one another, until every device in the Region is near some Persistent Node, and devices choose a node with which to associate, with some hysteresis. A set of partitions with exponentially increasing diameter can be wired together to form a hierarchical partition that is highly resilient against failures (see Figures 5 and 6).

Maintaining a hierarchical partition takes storage and communication logarithmic in the diameter of the Region being partitioned, and creating the partition takes time log-linear in the diameter.



(a) Level 1 (b) Level 2 (c) Level 3

Figure 5: Hierarchical partitioning via Persistent Nodes for a simulation with 2000 devices. There are five levels in the resulting hierarchy. The top and bottom levels of the hierarchy are uninteresting, as the top has every particle in the same group and the bottom has every particle in a different group; these images show the middle three levels of the hierarchy, with each ith level node a different color, and thick black lines showing the approximate boundaries. The logical tree is shown in Figure 6.

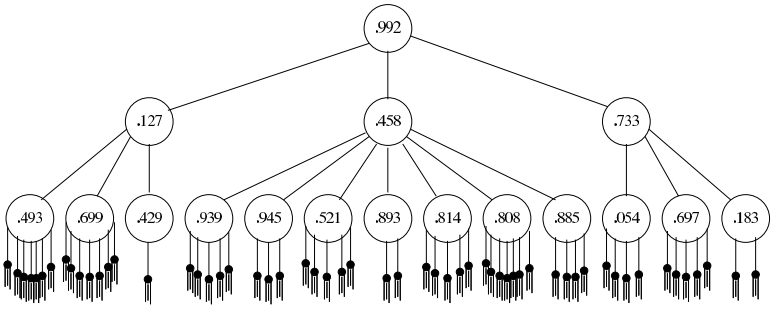


Figure 6: A hierarchy tree produced by the hierarchical partitioning shown in Figure 5. The numbers are the names of Persistent Nodes in the top three levels; the level 1 nodes are shown as black dots and the 2000 leaf nodes are not explicitly enumerated.

C Language for Robust Spatial Programming

We are formulating linguistic and programming tools for robust spatial programming. We want to be able to program an amorphous computer as though it were a space filled with a continuous medium of computational material: the actual executing program should produce an approximation of this behavior on a set of discrete points. This means that there should be no explicit reference to individual devices or to the communication among them. Instead, the language should describe behavior in terms of spatial regions of the amorphous medium: the manifold induced by neighborhoods in the amorphous computer's mesh network.

The program should be able to be specified without knowledge of the particular amorphous medium on which it will be run. Moreover, the shape of the medium should be able to change as the program is executing, through failure and addition of devices, and the running program must be able to adjust to its new environment gracefully.

Finally, since failures and additions may disconnect and reconnect the medium, a program which is separated into two different executions must be able to reintegrate when the components of the medium rejoin.

C.1 AML

We are currently developing an Amorphous Medium Language (AML), as a preliminary experiment. It provides one way to combine the robust primitives identified above in an attempt to achieve our goals. AML elaborates these ideas in terms of principles of design: spatial processes, aggregate state, active process maintenance, and homeostasis.

C.1.1 Spatial Processes

In AML each active process is associated with a region of space. The programs that direct a process are written to be executed at a generic place in their region. Every device in the region is running the process, continually sharing process variables with those neighbors (in the region) that are running the same process.

A process may designate (perhaps overlapping) subregions of its region to execute subprocesses. These subregions may be delineated by an active gradient, a Persistent Node, or some characteristic function. Thus, the collection of processes forms a tree, whose root operates on a region that covers the entire network.

The set of devices constituting a region that is participating in a processes may migrate, expand, or contract in response to changing conditions in the network or in its state, subject to the constraint that it remains in the region owned by its parent process. Since regions assigned to processes may overlap, the devices in the overlap will be running all of the processes associated with that part of space, and all of the parent processes that cover that region.

C.1.2 Aggregate State

Processes have state, described as the values of program variables. The values of these variables are determined by aggregation across the set of participating devices. Depending on the requirements of the computation, a variable may be implemented by gossip, by reduction, or as a distributed atomic object. Gossip is the cheapest mechanism, but it provides no consistency guarantees and can be used only for certain aggregation functions (e.g. maximum or union). Reduction is similar, but somewhat more costly and allows non-idempotent aggregation functions (e.g. average or sum). An

```
(defprocess root ()
  (defvariable x #'max :base 0)
  (maintain (= (local x) (density))
            (set! x (density)))
  (always (actuate 'color (regional x))))
```

Figure 7: Code to calculate maximum density (number of neighbors). The density at each point is written to variable X, which aggregates them using the function MAX. Each point then colors itself using the aggregate value for X.

atomic object gives guaranteed consistency, but it is expensive and may not be able to progress in the face of partition or high failure rates.

If a process is partitioned by a failure into two parts that cannot communicate then each part is an independent process, which evolves separately. Differences between the parts must be resolved if the parts later merge.

C.1.3 Active Process Maintenance

To remain active a process must be actively supported by its parent process. If a process is not supported it will die. The root process is eternally supported. Support is implemented by an active gradient mechanism. When a process loses support and dies the resources held by that process in each device are recycled by a garbage collector.[2]

C.1.4 Homeostasis and Repair

In AML processes are specified by procedures that describe conditions to be maintained and actions to be taken if those conditions are not satisfied. If a failure is small enough the actions are designed by the programmer to repair the damage and restore the desired conditions.

As a result, incomplete computation and disruption caused by failures can be handled uniformly. Failures produce deviations in the path towards homeostasis, and if the system state converges toward homeostasis faster than failures push it away, then eventually the process will complete its computation.

C.2 Examples

We currently have a partial implementation of AML, embedded in Common Lisp. The implementation allows us to experiment with the kinds of processes that can be expressed. We show two simple examples to give the flavor of the ideas.

C.2.1 Maximum Density

Calculating maximum density is the AML equivalent of a "hello world" program. To express this, we will need only one simple process. See Figure 7.

The AML process ROOT is the entry point for a program, similar to a MAIN function in C or Java. When an AML program runs, the ROOT process runs across the entire space, and is automatically supported everywhere so it will never be garbage collected.

[2] This is much the same problem as addressed in [3].

Processes are defined with the command (DEFPROCESS *name* (*arguments*) *statement* ...). In this case, the name is ROOT and there are no arguments, since there are no initial conditions. The statements of a process are variable definitions and homeostasis conditions.

The first statement creates a variable, X, which we will use to aggregate the density. Variables are defined with the command (DEFVARIABLE *name aggregation-function arguments*). In this case, since we want to calculate maximum density, the aggregation function will be MAX. Aggregation in AML is executed by taking a base aggregate value and updating it by merging it with other values or aggregates. If there are no values to aggregate, the variable has the given base value.

Now X will collect the maximum value over the process region, but haven't specified how it gets any values to start with. Every variable has three values: a local value, a neighborhood aggregate value, and a regional aggregate value. Setting a variable sets only its local value, though the aggregates may change as a result. Reading any value from a variable is instantaneous, but the neighborhood aggregate may be slightly stale, and the regional aggregate may be quite out of date.

The second statement is a homeostasis condition that deals only with the local value of X. The function (DENSITY) is a built-in function that returns the estimated density of the device's neighborhood.[3] The MAINTAIN condition may be read as: if the local value for X isn't equal to the density, set it equal to the density.

The third statement is just a way of displaying the results. It is formulated as a homeostasis condition, as (ALWAYS *action*) can be expanded to (MAINTAIN (FALSE) *action*). Here the displayed color is set to the regional value of the variable (REGIONAL X), which gives the current best estimate of the density.

The procedure (ACTUATE *actuator value*) is a built-in function to allow AML programs to write to an external interface (its converse (READ-SENSOR *sensor*) reads from the external interface). The code for color actuation contains a map from numbers to colors. When this program is run, all the devices in the network start with the color for zero, then turn various colors as each writes its density to X locally. The highest value colors then spread outward through their neighbors until each connected component on the network is colored uniformly according to the highest density it contains.

C.2.2 Blob Detection

With only slightly more complexity, we can write a program to detect blobs in a binary image. See Figure 8. Here we posit that an image is projected onto our space. The image is input to the network via a sensor named IMAGE, which reads BLACK for devices located at black points of the image and WHITE for devices located at white points of the image. The goal of the program is to find all of the contiguous regions of black and measure their areas.

Unlike the maximum density program, the ROOT process for blob detection takes a FUZZINESS argument, which specifies how far apart two black regions can be and still be considered contiguous.

The first statement in the ROOT process declares a variable, BLOBS, which uses UNION to aggregate the the blobs detected throughout the network into a global list.

The second statement is an ALWAYS condition which runs a blob measuring process anywhere that there is black. The MEASURE-BLOB process (see Figure 9) takes no arguments, and its extent is defined by an active gradient going out FUZZINESS hops from each device where the image sensor reads BLACK.

This elegantly segments the image into blobs: from each black device a gradient spreads the process out for FUZZINESS hops in all directions, so any two black devices separated by at most

[3] Density is most simply calculated as number of neighbors, but might be smoothed for more consistent estimates.

```
(defprocess root (fuzziness)
  (defvariable blobs #'union :base '())
  (always
    (when (eq (read-sensor 'image) 'black)
      (subprocess (measure-blob) :gradient fuzziness)
      (setf blobs
       (list (get-from-sub (measure-blob) blob)))))
  (avoid
    (read-sensor 'query)
    (let ((q (first (read-sensor 'query))))
      (cond ((eq q 'blobs)
             (actuate 'response (regional blobs)))
            ((eq q 'area)
             (actuate 'response
              (fold #'+ (mapcar #'second
                                (regional blobs))))))))))
```

Figure 8: Code to find a set of fuzzy blobs and their areas in a binary image. Each contiguous black area of the image runs a connected MEASURE-BLOB process that names it and calculates its area. The set of blobs is collected by the ROOT process and made accessible to the user on the RESPONSE actuator in response to requests on the QUERY sensor.

twice-FUZZINESS hops of white devices will be in a connected component of the MEASURE-BLOB process. Where there are more than twice-FUZZINESS hops of white devices separating two black points, however, the MEASURE-BLOB process is not connected and each component calculates independently, effectively as a separate blob!

The final statement of the ROOT process sets up a user interface in terms of an AVOID homeostasis condition. When there is a request queued up on the QUERY sensor, it upsets homeostasis, which the repair action attempts to rectify by placing an answer, calculated from the regional aggregate value of BLOBS, on the RESPONSE actuator. The user may then remove the serviced request from the queue, restoring homeostasis.

The MEASURE-BLOB process (see Figure 9) has two responsibilities: give itself a unique name, and calculate its area. The UID variable, whose aggregate will be the name of the blob, uses two arguments which we haven't seen before. The ATOMIC argument means that the regional aggregate value of UID will be consistent across the process and as a side effect will be more stable in its value. We use the INIT argument, on the other hand, to start UID with a random value at each point. As a result, UID will eventually have a random number as its regional aggregate value which is unlikely to be the same as that of another blob.

The AREA variable also uses an INIT argument, which sets everything to be 1. This serves as the point mass of a device, which we integrate across the process to find its area using SUM as an aggregator. We must ensure that no point is counted more than once, however, so we use the REDUCTION argument to specify that the aggregation must be done that way rather than defaulting to gossip.

Finally, we declare the BLOB variable and add an ALWAYS statement to make it a list of UID and AREA, packaging a result for the MEASURE-BLOB process to be read by the ROOT process. The ROOT process can read variables in with child processes with the command (GET-FROM-SUB (*name*

```
(defprocess measure-blob ()
  (defvariable uid #'max :atomic :base 0 :init (random 1))
  (defvariable area #'sum :reduction :base 0 :init 1)
  (defvariable blob :local)
  (always (setf blob (list uid area))))
```

Figure 9: The blob-measurement process needs three variables. The UID is a name for the connected blob component. The area accumulates the area and blob is the result to be used by the parent process.

parameters) *variable*), and uses this to set the local value of BLOBS.

Thus, given a binary image, each contiguous region of black will run a MEASURE-BLOB process which names it and calculates its area. The ROOT process then records this information, which propagates throughout the network until there is a consistent list of blobs everywhere.

C.3 Problems

AML does not solve all of the problems we wish to address, but it has exposed a clear set of problems, some of which appear to be challenges inherent to programming space.

- It is not clear at this time how large a range of behaviors can be specified as homeostatic conditions. It does seem adequate to allow the construction and maintenance of arbitrary graphs and some geometric constructions. The code that is executed when a condition is violated is intended to remedy the situation, but there is no guarantee that the repair behavior is actually making progress.
- AML programs can be composed with subroutine semantics, but it is less clear how to implement functional composition, particularly when the two programs may occupy different regions of space.
- Moving information between processes logically separated in the process tree is awkward, since information must be routed through a common parent.
- Integration of other programming ideas, such as streaming, may help to address some of these problems. We may also need to develop robust primitives better suited for our environment.
- At a more fundamental level, the tradeoff between consistency and liveness needs continued exploration, as does the problem of how to merge state when separated regions of a process rejoin.
- Failures which involve unpredictable behavior of a device are not currently dealt with, and may cause widespread disruption in the network.

Indeed, even analyzing the behavior of an aggregate in which devices are continually failing and being replaced is not a well understood problem.

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. AI Memo 1665, MIT, 1999.

[2] Daniel Aguayo, John Bicket, Sanjit Biswas, Douglas S. J. De Couto, and Robert Morris, "MIT roofnet implementation," 2003.

[3] H. Baker and C. Hewitt, "The incremental garbage collection of processes.," in *ACM Conference on AI and Programming Languages*, 1977, pp. 55–59.

[4] J. Beal. "Persistent nodes for reliable memory in geographically local networks." Tech Report AIM-2003-11, MIT, 2003.

[5] J. Beal. A robust amorphous hierarchy from persistent nodes. In *CSN*, 2003.

[6] Jacob Beal and Seth Gilbert, "RamboNodes for the metropolitan ad hoc network," in *Workshop on Dependability in Wireless Ad Hoc Networks and Sensor Networks, part of the International Conference on Dependable Systems and Networks*, June 2003.

[7] William Butera, *Programming a Paintable Computer*, Ph.D. thesis, MIT, 2002.

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

[9] Daniel Coore. "Establishing a Coordinate System on an Amorphous Computer." MIT Student Workshop on High Performance Computing, 1998.

[10] Daniel Coore, "Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer." Ph.D. thesis, MIT, 1999.

[11] Daniel Coore, Radhika Nagpal and Ron Weiss. "Paradigms for structure in an amorphous computer." MIT AI Memo 1614.

[12] DARPA IXO, "Networked embedded systems technology program overview," .

[13] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch., "Geoquorums: Implementing atomic memory in mobile ad hoc networks," in *Proceedings of the 17th International Symposium on Distributed Computing (DISC 2003)*, 2003.

[14] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alex A. Shvartsman, and Jennifer L. Welch, "Virtual mobile nodes for mobile ad hoc networks," in *DISC04*, Oct. 2004.

[15] Seth Gilbert, Nancy Lynch, and Alex Shvartsman, "RAMBO II:: Rapidly reconfigurable atomic memory for dynamic networks," in *DSN*, June 2003, pp. 259–269.

[16] V. Hsu, J. M. Kahn, and K. S. J. Pister, "Wireless communications for smart dust," Tech. Rep. Electronics Research Laboratory Technical Memorandum Number M98/2, Feb. 1998.

[17] H. Huzita and B Scimemi "The Algebra of Paper-folding," in *First International Meeting of Origami Science and Technology,* Ferrara, Italy, 1989.

[18] Thomas F. Knight Jr. and Gerald Jay Sussman. "Cellular gate technology." In *Unconventional Models of Computation*, pages 257-272, 1997.

[19] Attila Kondacs, "Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[20] Leslie Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[21] D. Liben-Nowell, H. Balakrishnan, D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, 2002.

[22] Nancy Lynch, *Distributed Algorithms*, Morgan Kaufman, 1996.

[23] Nancy Lynch and Alex Shvartsman., "RAMBO: A reconfigurable atomic memory service for dynamic networks," in *DISC*, 2002, pp. 173–190.

[24] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Workshop on Mobile Computing and Systems Applications*, 2002.

[25] Radhika Nagpal, *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*, Ph.D. Thesis, MIT, 2001.

[26] "NMRC scientific report 2003," Tech. Rep., National Microelectronics Research Centre, 2003.

[27] Ryan Newton and Matt Welsh, "Region streams: Functional macroprogramming for sensor networks," in *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.

[28] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker, "GHT: a geographic hash table for data-centric storage," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. 2002, pp. 78–87, ACM Press.

[29] Ron Weiss, *Cellular Computation and Communications using Engineered Genetic Regulatory Networks*, Ph.D. Thesis, MIT, 2001

[30] Ron Weiss and Tom Knight "Engineered Communications for Microbial Robotics" in *Proceedings of the Sixth International Meeting on DNA Based Co mputers (DNA6)*, June 2000

[31] Ron Weiss and Subhyu Basu. "The device physics of cellular logic gates." *In NSC-1: The First Workshop on NonSilicon Computing*, pages 54– 61, 2002.

[32] Matt Welsh and Geoff Mainland, "Programming sensor networks using abstract regions," in *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, Mar. 2004.

[33] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. 2004, ACM Press.