Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-039 May 27, 2006

Programming an Amorphous Computational Medium

Jacob Beal

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

Programming an Amorphous Computational Medium

Jacob Beal

Massachusetts Institute of Technology, Cambridge MA 02139, USA

Abstract. Amorphous computing considers the problem of controlling millions of spatially distributed unreliable devices which communicate only with nearby neighbors. To program such a system, we need a high-level description language for desired global behaviors, and a system to compile such descriptions into locally executing code which robustly creates and maintains the desired global behavior. I survey existing amorphous computing primitives and give desiderata for a language describing computation on an amorphous computer. I then bring these together in Amorphous Medium Language, which computes on an amorphous computer as though it were a space-filling computational medium.

1 Introduction

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 [17, 26]), and biological computation [18, 29, 30] all envision deployed 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.

Any such large spatially embedded mesh network may be considered an amorphous computer. Amorphous computing studies the problem of controlling these networks from a perspective of group behavior, taking inspiration from biological processes such as morphogenesis and regeneration, in which unreliable simple processing units (cells) communicating locally (e.g. with chemical gradients) execute a shared program (DNA) and cooperate to reliably produce an organism's anatomy.

Controlling an amorphous computer presents serious challenges. Spatially local communication means networks may have a high diameter, and large numbers of nodes place tight constraints on sustainable communication complexity. Moreover, large numbers also mean that node failures and replacements are a continuous process rather than isolated events, threatening the stability of the network. If we are to program in this environment, we need high-level programming abstractions to separate the behaviors being programmed from the networking and robustness issues involved in executing that program on a spatially embedded mesh network.

J.-P. Banâtre et al. (Eds.): UPP 2004, LNCS 3566, pp. 121–136, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Let us define an **amorphous medium** to be a hypothetical continuous computational medium filling the space approximated by an amorphous computer. I propose that this is an appropriate basis for abstraction: high-level behavior can then be described in terms of geometric regions of the amorphous medium, then executed approximately by the amorphous computer.

In support of this hypothesis, I first detail the amorphous computing scenario, then survey existing amorphous computing primitives and describe how they can be viewed as approximating the behavior of an amorphous medium. I then present the Amorphous Medium Language, which describes program behavior in terms of nested processes occupying migrating regions of the space in which the network is embedded, describe AML's key design elements — spatial processes, active process maintenance, and homeostasis — and illustrate the language by means of examples.

2 The Amorphous Computing Scenario

The amorphous computing engineering domain presents a set of challenging requirements and prohibitions to the system designer, forcing confrontation of issues of robustness, distribution, and scalability. These constraints derive much of their inspiration from biological systems engaged in morphogenesis and regeneration, which must be accomplished by coordinating extremely large numbers of unreliable devices (cells).

The first and foremost requirement is **scalability**: the number of devices may be large, anywhere from thousands to millions or even billions. Biological systems, in fact, may comprise trillions of cells. Practically, this means that an algorithm is reasonable only if its per-device asymptotic complexity (e.g. space or bandwidth per device) are polynomial in $\log n$ (where n is the number of devices) — and any bound significantly greater than $O(\lg n)$ should be treated with considerable suspicion. Further, unlike ad-hoc networking and sensor networks scenarios, amorphous computing generally assumes **cheap energy, local processing, and storage —** in other words, as long as they do not have a high per-device asymptotic complexity, minimizing them is not of particular interest.

The network graph is determined by the **spatial distribution** of the devices in some Euclidean space, which collaborate via **local communication.** Devices are generally **immobile** unless the space in which they are embedded is moved (e.g. cutting and pasting "smart paper")[1]. A unit disc model is often used to create the network, in which a bidirectional link exists between two devices if and only if they are less than distance r apart. Local communication implies that the network is expected to have a high diameter, and, assuming a packet-based communication model, this means that the time for information to propagate through the network depends primarily on the number of hops it needs to travel. Local communication also means that communication complexity is best mea-

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

sured by **maximum communication density** — the maximum bandwidth required by a device — rather than by number of messages.

Due to the number of devices potentially involved, the system must not depend on much 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 via initial conditions.** Once the system is running, however, there is **no per-device administration** allowed.

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. In addition, the system may not assume complex services not presently extended to the amorphous domain — this is applied particularly to mean **no global naming, routing, or coordinate service** may be assumed.

Finally, in a large, spatially distributed network of devices, failures are not isolated events. Due to the sheer number of devices, 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. In addition, because the network is spatially embedded, outside events may cause **failure of arbitrary spatial regions** — larger region failures are assumed to occur less frequently. Generally stopping failures have been used, in which the failing device or link simply ceases operating. Finally, maintenance requires recovery or replacement of failed devices: in either case, the effect is that **new devices join** the network either individually or as a region.

3 Amorphous Computing Mechanisms

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.

3.1 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, 32]. Each device maintains a table of key-value pairs which it wishes to share (for example, the keys might be variable names and the values their bindings). 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 number of values being stored, size of the values, and number of neighbors.

3.2 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 [31]. A Region is defined by a name and a boolean membership function. 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.

Note that 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.

3.3 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.

3.4 Consensus and Reduction

Non-failing devices participating in a strong consensus process must all choose the same value if any of them choose a value, and the chosen value must be held by at least one of the participants. 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) — as before, all non-failing devices complete the operation holding the same value.

The Paxos consensus algorithm [20] has been demonstrated in an amorphous computing context [6], but 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.

3.5 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.

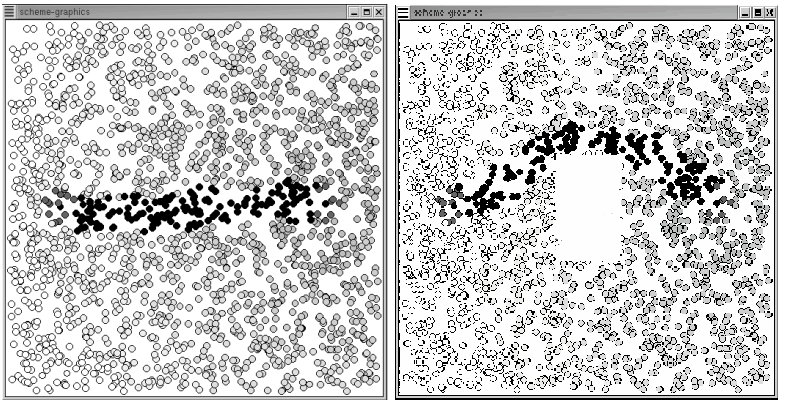
3.6 Active Gradient

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, useful for establishing Regions. Points in the gradient converge to the minimum hop-count and repairs 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. Figure 1 shows an active gradient being used to maintain a line.

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

3.7 Persistent Node

A Persistent Node [4] is a robust mobile virtual object that occupies a ball in the amorphous medium (See Figure 2). A Persistent Node is a read/write



(a) Before Failure (b) After Repair

Fig. 1. 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

object supporting conditionally atomic transactions. A variant guaranteeing atomic transactions has been developed [6], which trades off liveness in favor of consistency and costs more due to underlying consensus and reduction algorithms.

A Persistent Node is implemented around an active gradient flowing outward from its center. All devices within r hops (the Persistent Node's *core*) are identified 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.

Persistent Nodes are useful as primitives for establishing regions, much like active gradients. A Persistent Node, however, can change its position within a larger region in response to changing conditions.

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.

3.8 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,

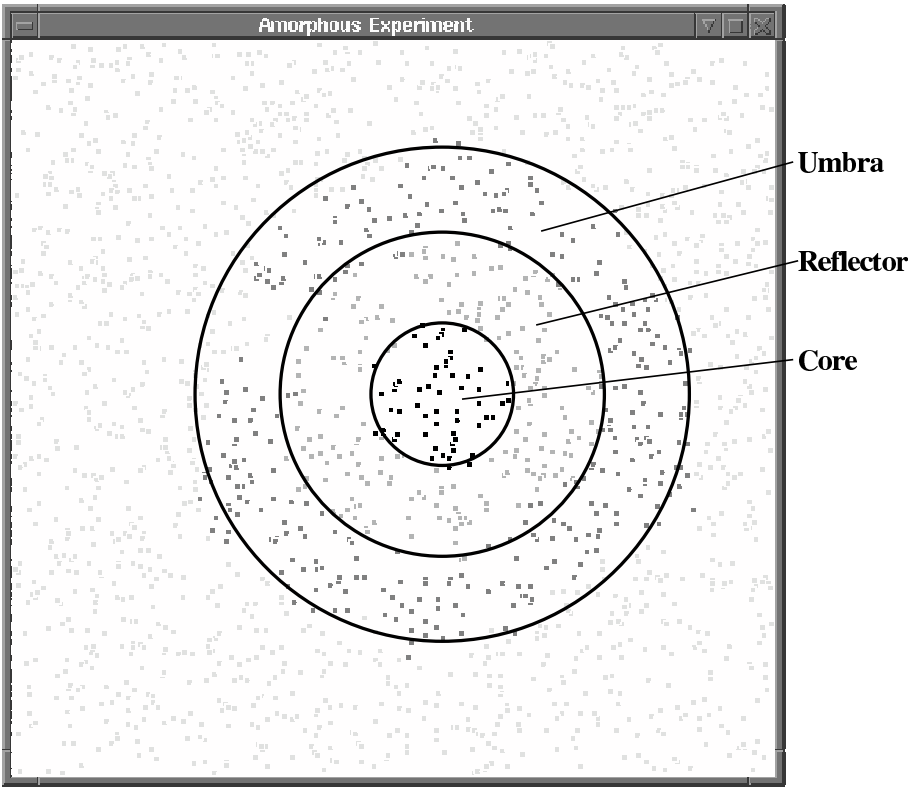
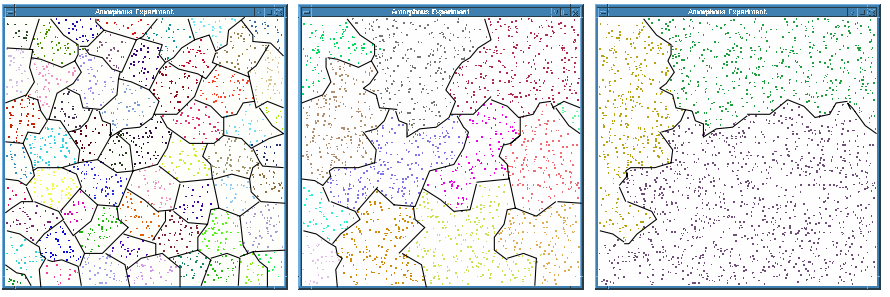


Fig. 2. Anatomy of a Persistent Node. The innermost circle is the core (black), which arts as a virtual node. Every device within middle circle is in the reflector (light grey), which calculates which direction the Persistent Node will move. The outermost circle (dark grey) is the umbra (k=3 in this example), which knows the identity of the Persistent Node, but nothing else

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 generate clustering relations which form a hierarchical partition of the Region. Although the partitions are not nested, the clusters which they generate are, and the resulting structure is highly resilient against failures — unlike other clustering algorithms, partitioning on the basis of persistent nodes bounds the extent of address changes caused by a failure linearly in the size of a circumscription of the failure. Figures 3 and 4 show an example of partitioning.

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

Fig. 3. 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 4

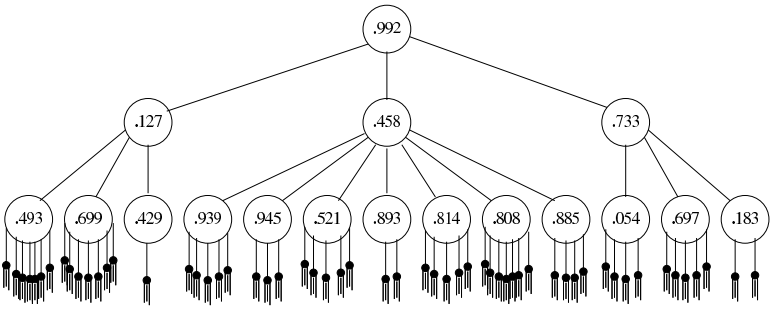


Fig. 4. A hierarchy tree produced by the hierarchical partitioning shown in Figure 3. The numbers are the names of Persistent Nodes in the top three levels; the level 1 nodes are shown as black dots and the 2000 leaf nodes are not explicitly enumerated

4 Amorphous Medium Language

I want to be able to program an amorphous computer as though it is 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 statements about individual devices or communication between them. Instead, the language should describe behavior in terms of spatial regions of the amorphous medium, which I will take to be 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 geometry and

topology of the medium should be able to change as the program is executing, through failure and addition of devices, and the running program 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.

Amorphous Medium Language combines mechanisms from the previous section to fulfill these goals via three key design components: spatial processes, aggregate state, active process maintenance, and homeostasis.

4.1 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 (This constraint yields subroutine compositional semantics). 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.

Many copies of the same process may run at the same device, as long as they have different parameters or different parent processes. So if process Foo calls (Fibonacci 5) and (Fibonacci 10), it creates two different processes, and if Foo and Bar both call (Fibonacci 5) it creates two different processes, but if Foo calls (Fibonacci 5) twice it creates only a single process.

4.2 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 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.

4.3 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]

4.4 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.

4.5 Examples

I will illustrate how AML programs operate by means of examples. These examples use the CommonLISP formatting processed by my AML compiler. These examples are meant to illustrate how ideas are expressed in AML.

Maximum Density. Calculating maximum density is the AML equivalent of a "hello world" program. To run this, we will need only one simple process (See Figure 5).

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.

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; while the process is active, it runs cyclically, clocked by the shared neighbor data refreshes. Each cycle the process first updates variable aggregate values, then maintains its homeostasis conditions in the order they are defined.

The first statement creates a variable, X, which we will use to aggregate the density. Variables are defined with the command (DEFVARIABLE *name*

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

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

Fig. 5. 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

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 — this also means that if there are no values to aggregate, the variable equals the base value. Since the default base value, NIL, is not a reasonable value for density, we use the optional BASE argument to set it to zero instead.

Now we have a variable which will calculate its maximum value over the process region, but haven't specified how it gets any values to start with. First, though, a word about the different ways in which we can talk about the value of variable X. 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, based on the current best estimate, but the neighborhood aggregate may be one cycle stale, and the regional aggregate may be indefinitely 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] so 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.

These local values for density are then aggregated by the variable, and the current best estimate can be read using (REGIONAL X), as is done in the third statement. The third statement is an ALWAYS homeostasis condition, which is syntactic sugar for (MAINTAIN NIL ...), so that it is never satisfied and runs its action every cycle. The action, in this case, reads the regional value of X and sends it to the device's color in a display — (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).

When this program is run, all the devices in the network start with the color for zero, then turn all different colors as each writes its density to X locally. The

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

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.

Blob Detection. With only slightly more complexity, we can write a program to detect blobs in a binary image (See Figure 6). In this scenario, an image is mapped onto a space and devices distributed to cover the image. 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 an argument — FUZZINESS — which specifies how far apart two black regions can be and still be considered contiguous.

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

```
(defprocess root (fuzziness)
  (defvariable blobs #'union)
  (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)))))))))

(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))))
```

Fig. 6. 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

The second statement is an ALWAYS condition which runs a blob measuring process anywhere that there is black. The MEASURE-BLOB process 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 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 MEASURE-BLOB process 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 parameters*) *variable*), and uses this to set the local value of BLOBS.

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 would then remove the serviced request from the queue, restoring homeostasis.

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.

4.6 Related Languages

In sensor networks research, a number of other 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.

Both of 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.

More similar is the Regiment [27] language, which uses a stream-processing abstraction to distribute computation across the network. Regiment is, in fact, complementary to AML: its top-down design allows it to use the well-established formal semantics of stream-processing, while AML's programming model is still evolving. Regiment's robustness against failure, however, has not yet clearly been established, and there are significant challenges remaining in adapting its programming model to the sensor-network environment.

Previous work on languages in amorphous computing, on the other hand, has worked with much the same failure model, but has been directed more towards problems of morphogenesis and pattern formation than general computation. For example, Coore's work on topological patterns [10], and the work by Nagpal [25] and Kondacs [19] on geometric shape formation. A notable exception is Butera's work on paintable computing [7], which allows general computation, but operates at a lower level of abstraction than AML.

5 Conclusion

Considering the space occupied by an amorphous computer as an amorphous medium provides a useful programming abstraction. The amorphous medium abstraction can be programmed in terms of the behavior of spaces defined using geometric primitives, and the amorphous computer can approximate execution on the actual network.

Amorphous Medium Language is an example of how one can program using the amorphous medium abstraction. Processes are distributed through space, and run while there is demand from their parent processes. Within a connected process region, data is shared via variables aggregated over the region, and computation executes in response to violated homeostasis conditions.

An AML compiler in progress is being used as a workbench for further language development, concurrent with investigation of more lightweight implementations of robust primitives and of analysis techniques capable to dealing with the challenges of an amorphous environment. All of the mechanisms described above have been implemented, but only gossip, regions, and shared data are fully integrated with the compiler, which is run on a few test programs whose behavior is demonstrated in simulation on 500 nodes.

AML does not solve all of the problems in its domain, but it has exposed a clear set of problems, some of which appear to be challenges inherent to programming space. These outstanding challenges are the basis of current investigation.

- 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. Frederic Gruau, Philippe Malbos. "The Blob: A Basic Topological Concept for "Hardware-Free" Distributed Computation." in *Unconventional Models of Computation* (UMC 2002). Springer, 2002.
17. 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.
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 and Tom Knight "Engineered Communications for Microbial Robotics" in *Proceedings of the Sixth International Meeting on DNA Based Computers (DNA6)*, June 2000
30. 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.
31. 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.
32. 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.