MIT-CSAIL-TR-2006-040 May 27, 2006

Amorphous Medium Language

Jacob Beal

Amorphous Medium Language

Jacob Beal[*]
Massachusetts Institute of Technology
77 Massachusetts Ave
Cambridge, MA USA
jakebeal@mit.edu

ABSTRACT

Programming reliable behavior on a large mesh network composed of unreliable parts is difficult. Amorphous Medium Language addresses this problem by abstracting robustness and networking issues away from the programmer via a language of geometric primitives and homeostasis maintenance.

AML is designed to operate on a high diameter network composed of thousands to billions of nodes, and does not assume coordinate, naming, or routing services. Computational processes are distributed through geometric regions of the space approximated by the network and specify behavior in terms of homeostasis conditions and actions to be taken when homeostasis is violated.

AML programs are compiled for local execution using previously developed amorphous computing primitives which provide robustness against ongoing failures and joins and localize the impact of changes in topology. I show some examples of how AML allows complex robust behavior to be expressed in simple programs and some preliminary results from simulation.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Algorithms, Languages, Reliability

Keywords

Amorphous Computing, Ultrascale Sensor Networks

1. INTRODUCTION

Increasingly, we are faced with the prospect of programming spatially embedded mesh networks composed of huge numbers of unreliable parts. Projects in the fields of sensor networks (e.g. NEST[16]), peer-to-peer wireless (e.g. RoofNet[2]), and smart materials (e.g. smart dusts[15, 1]) have the capacity to create multi-thousand node networks in the near future, and might easily scale to millions and beyond if the technology takes off.

Controlling such a network 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.

Observing that a mesh network can be viewed as an approximation of the space it occupies, I 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. An AML program can then be expressed in terms of previously developed amorphous computing primitives which approximate the prescribed behavior on the network.

The ultimate goal of AML is to transform problems of communication and coordination in much the same way that garbage collection has transformed problems of memory management. A programmer would still have to consider these issues, but as engineering parameters to be managed rather than problems to be solved in detail.

If successful, it should provide a good platform for implementing a large variety of applications. For example, it should be possible to efficiently program sensor-networks applications like monitoring a wildlife habitat[22]. At the same time, it should be possible to write infrastructure to support ad-hoc wireless networks and even far-out applications like nanotechnological manufacturing.[13]

In the remainder of the paper, I first formalize the network model and give a brief overview of the amorphous computing primitives which are currently used by AML. I then describe AML's key design elements — spatial processes, active process maintenance, and homeostasis — and illustrate the language by means of examples. Finally, I compare AML with related work.

[*]Student author. This work partially supported by NSF grant #6895292.

2. NETWORK MODEL

AML is designed with respect to the following assumptions about the network:

- The number of nodes n is large, from thousands to billions.
- Nodes are distributed through space and collaborate via communication with neighbors no more than r distance away.
- Nodes are immobile.[1]
- Memory, processing and energy capacity are not limiting resources.
- Communication complexity is measured as maximum communication density — maximum bits per second transmitted by any node in the network.
- Execution is partially synchronous — each node 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.
- Naming, routing, or coordinate services may not be assumed.
- Nodes randomly fail and replacement nodes join at equal expected rates of f per second.
- Arbitrary regional stopping failures and joins may occur, including changes in the connectedness of the network.
- Nodes communicate with their neighbors via reliable broadcast.

The main departures from more common sensor network models is the relaxation of energy as a constraint. This is largely due to the focus on robustness, since there is an inverse relationship between transmission frequency and time to detect a failure.

Note also that uniform distribution of nodes and connection with neighbors less than distance r are not required, but that system performance may degrade when distribution is very non-uniform or when too few nearby neighbors cannot communicate.

In addition, several design goals provided further guidance:

- Due to the large number of nodes, asymptotic communication complexities should be limited to $O(\lg n)$.
- Due to the potentially large diameter d of the network, asymptotic time complexities should be limited to $O(d \lg d)$.
- Nodes should be identically programmed and differentiated at the beginning of execution only by minor differences in initial conditions.

[1]Note that mobile nodes might be programmed as immobile virtual nodes[11, 12].

3. AMORPHOUS COMPUTING MECHANISMS

Several previously developed algorithms from amorphous computing (which uses biological metaphors to program ultra-scaling mesh networks) are used to implement AML primitives. Each mechanism summarized here has been implemented as a code module and demonstrated in simulation.

3.1 Shared Neighborhood Data

This simple module allows neighboring nodes to communicate by means of a shared-memory region, similar to the systems described in [6, 27]. Each node maintains a table of key-value pairs which it wishes to share. Periodically each node 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.

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

3.2 Regions

The region module maintains labels for contiguous sets of nodes, using a mechanism similar to that in [26]. A Region is defined by a name and a membership test. When seeded in one or more nodes, a Region spreads via shared neighborhood data to all adjoining nodes that satisfy the membership test. When a Region is deallocated, a garbage collection mechanism spreads the deallocation throughout the participating nodes, attempting to ensure that the defunct Region is removed totally.

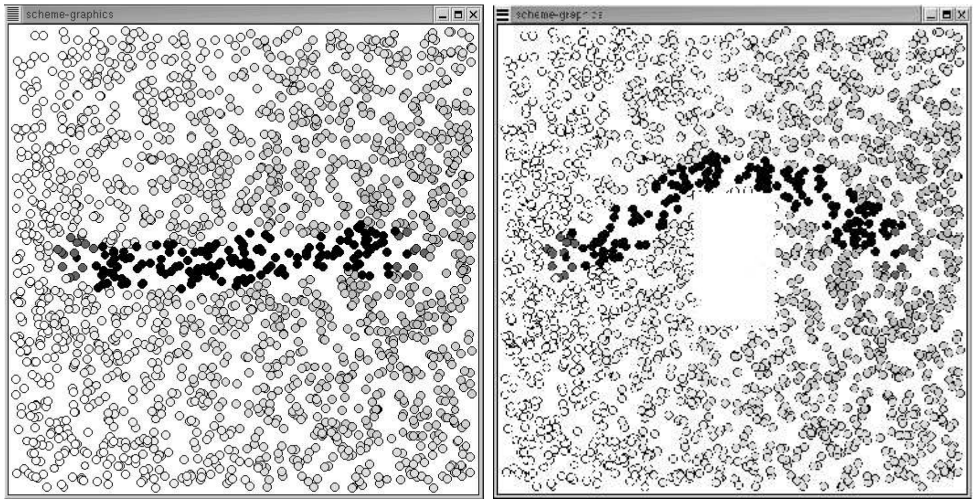
Note that failures or evolving system state may separate a Region into disconnected components. While these are still logically the same Region, and may rejoin into a single connected component in the future, information might not pass between disconnected components. As a result, the state of disconnected 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 node belonging to the root region. In order for a node 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[19, 5] propagates information throughout a Region via shared neighborhood data. 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



(a) Before Failure (b) After Repair

Figure 1: A line being maintained by active gradients, from [7]. 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.

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 node 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 nodes participating in a 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).

The Paxos consensus algorithm[18] has been demonstrated in an amorphous computing context[5], 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

Using consensus and reduction, quorum-based atomic memory can be implemented on a reconfigurable set of nodes[20, 14]. This has been demonstrated in simulation for amorphous computing[5], and scales as the underlying consensus and reduction algorithms do.

3.6 Active Gradient

An active gradient[8, 10, 7] maintains a hop-count upwards from its source or sources — a set of nodes which declare themselves to have count value zero — giving an approximate spherical distance measure useful for establishing regions. The gradient converges to the minimum hop-count and repairs its 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. 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 every node 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 node based on a versioned gradient. The gradient flows outward from the center, identifying all nodes within r hops (the Persistent Node's **core**) as members of the Persistent Node, while a heuristic calculation flows inward from all nodes 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 node in this region is aware of the existence and location of the Persistent Node. If any node 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.

The Persistent Node is also 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

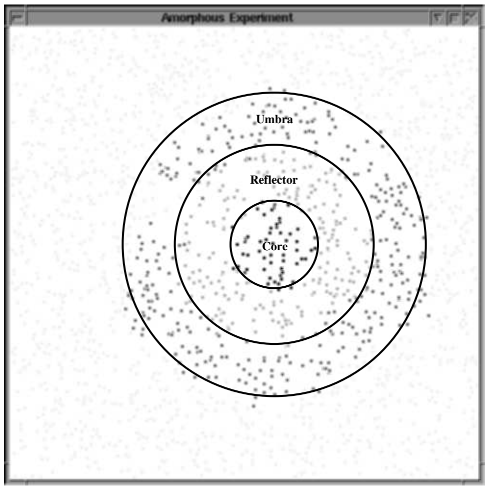


Figure 2: Anatomy of a Persistent Node. The innermost circle is the core (black), which arts as a virtual node. Every node within middle circle is in the reflector (light grey), which calculates which direction the Persistent Node will move. The outermost circle (dark grey) is the umbra (k=3 in this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

it. A Persistent Node moves and repairs itself in time linear in its diameter.

4. AMORPHOUS MEDIUM LANGUAGE

I want to be able to program a mesh network 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 nodes or communication between them. Instead, the language should describe behavior in terms of spatial regions of what I will term the *amorphous medium* - the manifold induced by the topology of the 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 nodes, 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.

Three key components of AML's design help to fulfill these goals: spatial processes, active process maintenance, and homeostasis.

4.1 Spatial Processes

Each running process is distributed throughout a region of space, executing behavior defined for a generic point in the process. Every node in that region is running the process, and sharing process variables with those neighbors that are running the same process.

The collection of processes forms a tree, whose root covers the entire network, and where subprocesses are confined within their parent processes. A subprocess may be further limited in scope by confinement to an active gradient, Persistent Node, or other arbitrary test. This structure is implemented simply, by means of the Region mechanism (Section 3.2).

The set of nodes participating in a processes may migrate, expand, and contract in response to changing conditions of the network (or its own movement heuristic, in the case of a Persistent Node). Note also that there may be parts of a process which cannot communicate directly with one another: for all intents and purposes these may act as two independent processes, evolving separately, but if they are brought together later, the two parts will fuse.

Many copies of the same process may run at the same node as well, 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.

Process execution is coordinated by means of its variables, whose values are produced via aggregation across the set of participating nodes. Depending on the requirements of the computation, the variable may be implemented via gossip, reduction, or distributed atomic object. Gossip is cheapest, but 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). Finally, an atomic object gives guaranteed consistency, but is most costly and may not be able to progress in the face of partition or high failure rates.

Processes are invoked in AML with a call to (SUBPROCESS (*name parameters*) *extent*).

4.2 Active Process Maintenance

The fluid nature of processes means that a process might find itself orphaned — for example, on the other side of a network partition from the sibling process that needed its output. In that case, it would be nice if the process just went away rather than cluttering up memory and swiping cycles for its now irrelevant task.[2]

AML addresses this issue by requiring that processes be actively supported by their parents (except for the root process). Support is maintained by an active gradient, and any process for which there is no support at a node is garbage collected from that node. Note that this also implies that processes do not return at a discrete point — a running process simply continues to update its output until its parent no longer needs it.

Thus, the orphaned process in the example above will no longer have support from its parent, because the sibling demanding its output is no longer connected to its portion of the parent, and will eventually be garbage collected.

Support for a subprocess is expressed by repeated invocation of the SUBPROCESS call.

4.3 Homeostasis and Repair

The ongoing nature of failures in a large network means that running an exception handler when a failure occurs is

[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))))
```

Figure 3: 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.

not reasonable — small failures are the rule, not the exception! Moreover, if two successive large failures were to occur, handling an exception caused by the second in the exception handler triggered by the first could be messy.

To sidestep the entire issue, AML formulates procedures in terms of homeostasis conditions and repair actions to take when they are violated. As a result, incomplete computation and disruption caused by failures can be handled uniformly. Failures are simply an additional perturbation in the path towards homeostasis, and if the system state converges toward homeostasis faster than failures push it away, then eventually the procedure will complete its computation.

There are two ways of expressing homeostasis conditions in AML: (AVOID *condition repair*) and (MAINTAIN *condition repair*). The AVOID expression evaluates *repair* whenever *condition* evaluates to true; MAINTAIN evaluates *repair* whenever *condition* is false. Note that there is no guarantee that the repair will eventually succeed, since its content is completely unconstrained.

5. EXAMPLES

I will illustrate how AML programs operate by means of two examples. These examples use the CommonLISP formatting processed by my AML compiler. The primitives expressed here map fairly directly onto existing amorphous computing algorithms, thereby simplifying the task faced by the compiler.

5.1 Maximum Density

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

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 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 node'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 node'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 nodes in the network start with the color for zero, then turn all different 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.

5.2 Blob Detection

With only slightly more complexity, we can write a program to detect blobs in a binary image. In this scenario, an image is mapped onto a space and nodes distributed to cover the image. The image is input to the network via a sensor named IMAGE, which reads BLACK for nodes located at black points of the image and WHITE for nodes 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

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

```
(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))))
```

Figure 4: 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.

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 takes no arguments, and its extent is defined by an active gradient going out FUZZINESS hops from each node where the image sensor reads BLACK.

This elegantly segments the image into blobs: from each black node a gradient spreads the process out for FUZZINESS hops in all directions, so any two black nodes separated by at most twice-FUZZINESS hops of white nodes will be in a connected component of the MEASURE-BLOB process. Where there are more than twice-FUZZINESS hops of white nodes 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 node, 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.

6. RELATED WORK

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[25] provides a hash table abstraction for storing data in the network, and TinyDB[21] 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[24] 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[9], and the work by Nagpal[23] and Kondacs[17] on geometric shape formation. A notable exception is Butera's work on paintable computing[6], which allows general computation, but operates at a lower level of abstraction than AML.

7. CONCLUSION

AML provides a language of abstractions for describing programs for large spatially embedded mesh networks. 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.

At present, the compiler correctly compiles and executes simple AML programs in simulation. Most immediate of future work is the continued refinement of the AML compiler on the basis of existing primitives. Further development of amorphous computing primitives is likely to enhance AML's performance, particularly with respect to variable aggregation. Finally, writing AML programs for real sensor-network problems will continue to refine the semantics, and testing AML code on deployed networks will provide more tangible evidence of its utility.

8. REFERENCES

[1] NMRC scientific report 2003. Technical report, National Microelectronics Research Centre, 2003.

[2] D. Aguayo, J. Bicket, S. Biswas, D. S. J. D. Couto, and R. 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*, pages 55–59, 1977.

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

[5] J. Beal and S. 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.

[6] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

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

[8] D. Coore. Establishing a coordinate system on an amorphous computer. In *MIT Student Workshop on High Performance Computing*, 1998.

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

[10] D. Coore, R. Nagpal, and R. Weiss. Paradigms for structure in an amorphous computer. Technical Report AI Memo 1614, MIT, 1997.

[11] 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.

[12] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Virtual mobile nodes for mobile ad hoc networks. In *DISC04*, Oct. 2004.

[13] K. E. Drexler, C. Peterson, and G. Pergamit. *Unbounding the future: the nanotechnology revolution*. Morrow, 1991.

[14] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II:: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*, pages 259–269, June 2003.

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

[16] D. IXO. Networked embedded systems technology program overview.

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

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[19] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[20] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC*, pages 173–190, 2002.

[21] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[22] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications*. ACM Press, September 2002.

[23] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001.

[24] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.

[25] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a geographic hash table for data-centric storage. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM Press, 2002.

[26] M. Welsh and G. 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.

[27] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.