Infrastructure for Engineered Emergence on Sensor/Actuator Networks

Jacob Beal and Jonathan Bachrach
MIT CSAIL

April 28, 2006

Abstract

Our ability to control emergent phenomena depends on decomposing them into aspects susceptible to independent engineering. The *amorphous medium* abstraction separates *what* behavior is desired on a continuous space and *how* the behavior is implemented on a sensor/actuator network approximating the space, while the Proto language provides a means of composing self-organizing primitives on an amorphous medium. We thus separate the engineering problem into three components: a discrete kernel to emulate an amorphous medium and distribute code, a compiler for Proto, and implementations of high-level coordination and homeostasis primitives, allowing simple and concise expression of programs controlling spatial behaviors. Programs written using our implementation of this framework have been verified in simulation on over ten thousand nodes, as well as on a network of Berkeley Motes.

1 Self-Managing Systems Engineering

The study of self-organizing systems has now reached the tool-building phase, in which a new discipline of *self-managing systems engineering* can begin to emerge.

The next step is to refine the principles of self-organization into a system of composable parts suitable for engineering—as the principles of electromagnetism are captured for electronic engineering in components like capacitors, transistors and resistors.

To transform a science into an engineering discipline, we must identify an operating range, decouple aspects of the problem from one another, create standard interfaces for composition, identify primitive components which conform to the standards, and create rules of abstraction which hide the complexity of systems of components.

We have begun this process in the domain of sensor/actuator network applications, observing that in many applications, the network is deployed to approximate a physical space, and that what is being programmed is the space rather than the network. This observation allows us to decouple self-management problems using the *amorphous medium* abstraction, so that global behavior descriptions in our Proto language can be compiled automatically into locally executed code which produces emergent phenomena matching the global description. We have experimentally verified our code both in simulation and (for small programs) on a network of sensor/actuator nodes.

2 Decoupling: Amorphous Medium

Consider deploying a network of devices to manage a large farm. The tasks to be carried out by the devices—irrigation, pest management, and fertilization, for example—are naturally specified in terms of regions of the farm: "a potato field is watered every so-many hours during hot weather" or "minor alfalfa weevil infestations are treated with an early harvest, but major infestations are treated with pesticides." An applications programmer for farms should be able to write code at this layer of abstraction, rather than having to specify how the devices in the fields will coordinate to carry out the programs.

The divide between specification and implementation is captured by the *amorphous medium* abstraction: an amorphous medium is a continuous computational material filling the space of interest. Every point in the medium is a computational device which independently executes the same code as every other device in the medium.[1] Each device has a *neighborhood* of devices less than d units of distance away, and exposes its internal state to its neighborhood. Conversely, a device can read the internal state of devices in its neighborhood, obtaining values lagged proportional to the distance separating them.

We cannot, of course, build a continuous medium containing uncountably many infinitely small computers. We can, however, approximate it by scattering a discrete set of devices through the medium. We then compute using as our basis the relatively few systems whose discrete behavior is a good approximation of their continuous behavior, just as electronic engineering uses components which capture only a few of the phenomena of electromagnetism. In both cases, restricting the range of behavior supports engineering abstractions that ignore much of the complexity of the underlying system.

The amorphous medium abstraction separates the space being programmed from the devices carrying out the program, allowing us to decompose self-managing systems engineering into three layers of abstraction—global, local, and discrete—each supported by its own infrastructure component, which decouple aspects of self-managing systems design into largely independent subproblems:

- The discrete layer consists of devices embedded in space exchanging messages with nearby neighbors. Infrastructure for this level is a *discrete kernel* which provides approximate emulation of an amorphous medium.
- The local layer executes on the amorphous medium, using our *Proto* language to specify a uniform behavior for each point.
- The global layer executes on the amorphous medium, using a library of amorphous computing algorithms translated into Proto to control the behavior of regions.

Figure 1 illustrates some of the design problems separated by use of these abstraction layers.

- The implementatin has three infrastructure components: a kernel providing the neighborhood abstraction, a compiler for the Proto language, and libraries of long-range coordination and control primitives coded in Proto.
- Global layer coordination primitives operate on regions and are implemented with local layer interactions between points and their neighborhoods. The neighborhood is, in turn, implemented by messages passed between discrete devices.
- Global control is described in terms of homeostatic processes continually moving regions towards a desired behavior. These are implemented as networks of streams in the local layer, which compile to update code executed periodically in the discrete layer.
- Diifferent modes of failure are handled by different layers: individual device crashes are masked by the neighborhood abstraction, outside events which destroy regions of the network need are handled by homeostatic primitives, and bugs in the user's code are minimized by a clean global layer interface.

[1] Executions diverge due to differences in sensor values, randomness, and interaction with their neighborhoods.

Layer	Infrastructure	Coordination	Control Flow	Failures	Energy Efficiency
Global	Proto Libraries	Region	Homeostasis	User	Coordinations
Local	Proto Compiler	Neighborhood	Stream Network	Region	Reductions
Discrete	Kernel	Device	Rounds	Device	Packets

Figure 1: Decomposing self-managing systems engineering into global, local, and discrete abstraction layers separates many design problems into largely independent subproblems.

- Assuming energy is dominated by the cost of communication, the amount of communication depends on how many long-range coordination operations are used in the global layer, how many reductions over neighbor state are used to implement coordination at the local layer, and the packets transmitted and received to implement shared neighbor state at the discrete layer.

3 Operating Range: Amorphous Computers

A sensor/actuator network in which devices communicate only with nearby neighbors can be considered as an *amorphous computer*[1]. Amorphous computing takes inspiration from biological systems engaged in morphogenesis and regeneration, in which extremely large numbers of unreliable devices (cells) coordinate to achieve predictable results with high precision.

We have chosen the amorphous computer model for two reasons. First, its biologically inspired specifications imply the self-management issues of robustness, distribution, and scalability. Second, real-world sensor/actuator networks are growing rapidly in scale and capability, bringing them closer into alignment with the amorphous computer model.

In particular, Proto and its supporting infrastructure are designed to operate on sensor/actuator networks with the following properties:

- The number of devices n may range from dozens to billions.
- Devices are distributed arbitrarily through space and collaborate via unreliable broadcast to neighbors no more than r distance away.
- Devices are mostly immobile.[2]
- Memory and processing are not limiting resources.[3]
- Execution is partially synchronous—each device has a clock which ticks regularly, but frequency may vary by up to ϵ and clocks have an arbitrary initial time and phase.
- Naming, routing, and coordinate services are not provided.[4]
- Arbitrary point and region stopping failures and joins may occur, including changes in the connectedness of the network.

Our operating range specification does not directly address energy consumption, although it has been a concern in implementation. Energy issues can be addressed independently in each abstraction layer, however, and we expect that most possible savings can be extracted by optimizing the discrete kernel implementation. Although some excess energy expenditure will likely remain, we consider the gain in engineering capability to be worth moderate inefficiency in energy expenditure.

[2]Note that mobile devices might be programmed as immobile virtual devices[7, 8].
[3]Profligate expenditure of either is still bad, and memory is an important constraint for the Mote implementation.
[4]They may be made available as sensor values, with appropriate characterization of reliability and error.

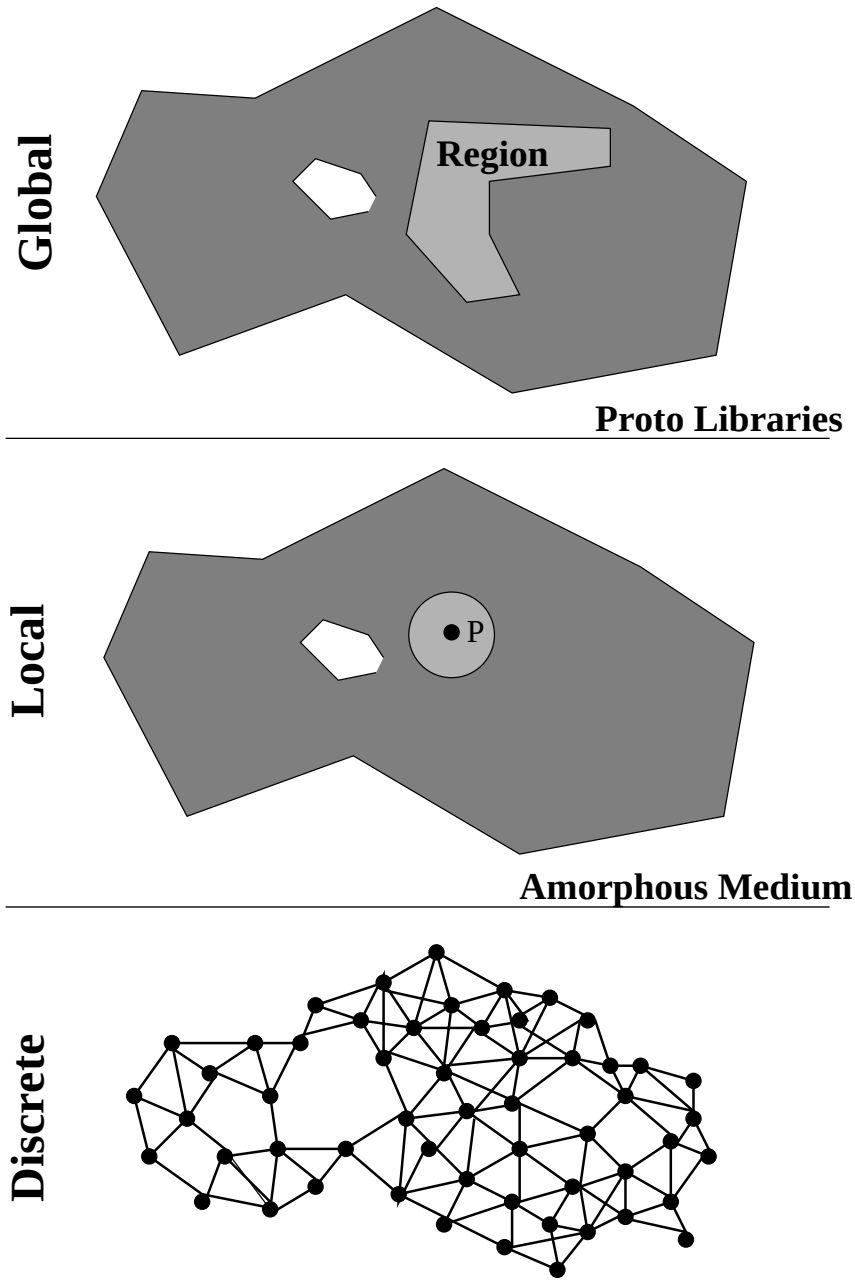

Figure 2: Self-management problems are decoupled by factoring into three abstraction layers: global, local, and discrete. Interactions between individual devices in the discrete layer emulate an amorphous medium. The local layer describes the behavior of points in the medium, from which library code is built to allow the description of the behavior of regions of medium at the global layer.

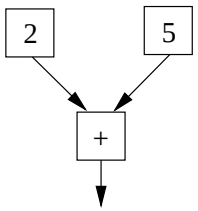

Figure 3: A Proto program is a network of operator instances ascending from a single root. The output stream of the root serves as a reference to the program.

4 Abstraction and Composition: Proto

Interface standards, primitive components, and rules of abstraction are captured in the semantics of *Proto*, a language we have constructed for specifying the behavior of points in the amorphous medium. Proto combines the dynamic stream networks of Gooze[2] with previous work on amorphous medium languages[4, 3].

4.1 Primitives and Composition

Programs in Proto produce a stream of output values. Proto uses Scheme syntax, but has its own set of types and primitive functions. For example, the expression

```
2
```

evaluates to a stream of twos. Programs are composed using functional operators, so the expression

```
(+ 2 5)
```

yields a program that emits a stream of sevens. Operator and operand expressions are evaluated with the same rules, as in Scheme. The operands are streams and the operator constructs a stream of output values from sets of values input from its operands. A program is a directed acyclic graph with a single root, with nodes that are instantiated operators and edges that connect from streams to the operator inputs which consume them (Figure 3). The output stream of the root serves as a reference to the program ascending from it.

4.2 Types

Proto is strongly typed like ML [14] and Haskell [12] and unlike statically typed languages such as C, types are inferred automatically from literals and function calls. Therefore, users rarely need to deal with types, but they are useful for describing the various kinds of data that are available and the signatures of built in operators.

Proto permits boolean, character, number, and symbol data types. These base types can be combined to form richer types using parameterized types, such as vectors, tuples, and functions. Vector and tuple types can be nested to create a rich set of derived types.

Proto supports overloaded operators and chooses the most applicable operator at compile time during type inference. This encourages reuse without sacrificing runtime efficiency.

4.3 I/O

Input from the outside world or other programs running on a device is accessed using the **sense** operator. For example,

```
(sense :light)
```

returns the value of the sensor named **light**. Similarly, a Proto program affects the outside world through the **actuate** operator. Thus, for example,

```
(actuate :sound (sense :light))
```

sends the light value to the sound actuator.[5]

4.4 State

Persistent state is established using delay loops, specifying an initial value and an expression for calculating the next value from current values. For example, the expression

```
(letfed ((i 0 (1+ i)))
  i)
```

creates one state variable, **i**, which starts at zero and increases by one each round.

4.5 Communication

Unlike discrete networks, each point in an amorphous medium has an infinite number of neighbors. As such, communication by message passing is impractical. Proto instead provides communication in the form of summaries of all the values in the neighborhood, using the **reduce-nbrs** operator to fold an expression across each point's neighborhood.

For example, assuming a boolean light sensor, we can dilate the lit region by one neighborhood radius with the expression

```
(reduce-nbrs (sense light) or nil)
```

The first argument is the value to be reduced, the second is the reduction function, and the third is the initial value of the reduction. When evaluated, **reduce-nbrs** begins with the initial value, then incorporates the values from its neighbors one at a time, using the reduction function, to produce the final result. Indeed, it is perhaps better to understand **reduce-nbrs** as a transform that operates on nearby space rather than as communication.

In general we do not want to tie the behavior of our program to neighborhood sizes, so Proto provides special operators for measuring the space distance, time distance, and volume of a neighbor: **nbr-dist**, **nbr-lag**, and **infinitesimal** respectively.[6]

Thus, for example, we can measure the distance to a light with a gradient flowing out from the source[7]

[5] The mechanism for binding sensors to names is implementation dependent, as is value when **sense** is applied to an unbound name, and the result of multiple streams being sent to the same actuator.

[6] These may be implemented coarsely or finely, depending on the hardware available: for example, our mote implementation estimates the distance to all neighbors as its radio range, and the time lag as one round.

[7] Biological systems often use chemical diffusion from a source as a distance measure, and various distributed computing fields have coopted "gradient" by analogy to mean a distance-to-source measurement created by gossip.

```
(letfed ((n infinity (if (sense light) 0
                         (+ (reduce-nbrs n min infinity)
                            (reduce-nbrs nbr-dist max 0)))))
  n)
```

Here, the **reduce-nbrs** expression starts with a value of **infinity** and combines it with each neighbor's value for **n** to find the minimum. As a result, **n** is pegged to zero at light sources and floats up by one for each unit of distance. Each point converges to the estimated distance to the nearest light source.

When Proto expressions are compiled into executable code, the values needed by **reduce-nbrs** expressions are identified by the compiler so that the discrete kernel can export them to its neighbors whenever they change. Any reduction which can be approximated using a sampling of neighbor state can be implemented on a real network by the discrete kernel. This covers a wide range of functions, particularly with the inclusion of the distance and **infinitesimal** operators to allow integration. For example,

```
(/ (reduce-nbrs (* (sense :light) infinitesimal) + 0)
   (reduce-nbrs (* 1 infinitesimal) + 0))
```

finds the average light value in each point's neighborhood (the second **reduce-nbrs** expression normalizes the integral). Some operators such as **random**, however, must be subtly redefined to have a compatible amorphous medium semantics and discrete kernel implementation.

4.6 Abstraction

Proto expressions can be abstracted to create new operators, just as ordinary Scheme expressions can be abstracted to create new functions. For example, we can make a generic gradient operator

```
(def gradient (src)
  (letfed ((n infinity (if src 0 (+ (reduce-nbrs n min infinity)
                                    (reduce-nbrs nbr-dist max 0)))))
      n))
```

and a generic averaging operator

```
(def local-average (x)
  (/ (reduce-nbrs (* x infinitesimal) + 0)
     (reduce-nbrs (* 1 infinitesimal) + 0)))
```

These operators then can be used in expressions, including definitions of operators at higher level of abstraction. Thus, for example, we can write the expression

```
(<= (gradient (sense :light)) 2)
```

that outputs **true** anywhere within 2 units distance of a light.

4.7 Execution

Pulling a value from a program's output stream initiates a round of execution.[8] Execution is then distributed up the network as operators pull values from their inputs. If an operator does not pull a value from one of its inputs, the upstream operator is not executed and goes into hibernation, discarding any internal state until such time as it begins being executed again and reboots.

For example, assuming a boolean sound sensor, the program

[8] These values are generally discarded, so the ultimate goal of a program must be achieved via actuation.

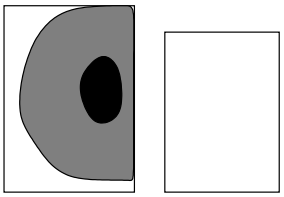

Figure 4: All communication proceeds through neighborhoods, so a gradient (grey) spreading from regions with light (black) that runs only when there is sound (white boxes) cannot cross a gap where it does not run.

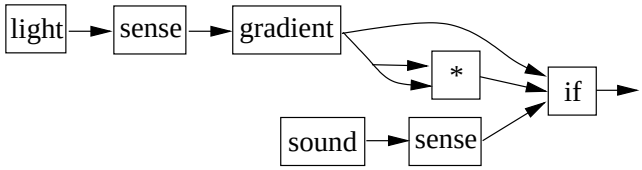

Figure 5: Subprograms may be used to feed multiple inputs. The subprogram caches its output so that it only executes only the first time its output stream is pulled in a given round.

```
(when (sense :sound)
  (<= (gradient (sense :light)) 2))
```

only runs the gradient where there is sound. As a consequence, points within 2 units distance but separated by a quiet area will output **false**, since the gradient is not running in the intervening area (Figure 4).

The expression associated with an interpreted operator is instantiated into an encapsulated network once for each active instance of the operator. When an instance hibernates, this network is discarded, along with any state in its loops, to be restarted from scratch when it next becomes active. Among other things, this allows recursion since the potentially infinite network structure is only constructed for the levels which are actually currently in use.

The output of a process module may be used as input by more than one other module. For example,

```
(let ((d (gradient (sense :light))))
  (if (sense :sound) (* d d) d))
```

always runs the gradient, but squares the output when there is sound (Figure 5). Execution carries a time-stamp identifying the round, so that the subprogram can return the same result every time a downstream process module pulls a value during a single round of execution. Conversely, as long as at least one process module pulls a value, the subprogram will execute.

4.8 Miscellany

Proto allows a programmer to define new primitive operators. Although not strictly necessary, primitive operators are generally faster and more memory-efficient because calculations can be performed without instantiating and walking a network of streams, as happens in an interpreted operator.

Proto code is quite compact, which is unsurprising, given its LISP roots. For example, Eames's algorithm for distributed discovery of minimum threat paths[9] requires 2000 lines of nesC[10] code, while an equivalent implementation in Proto is a mere 25 lines long.

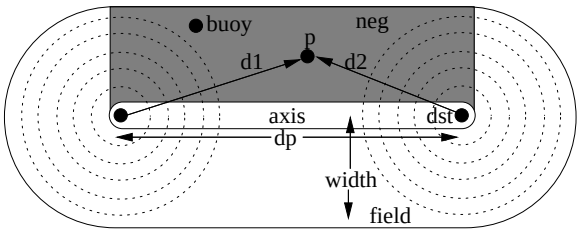

Figure 6: Finding coordinates with a mechanism adapted from paintable computing[5]: the two anchor points of the coordinate system send out gradients, producing **d1**, **d2** and **dp** which determine the location of **p** except for the sign of its vertical coordinate. The sign is found by using leader election to break symmetry.

5 Raising the Abstraction Level

Using Proto, we can implement composable abstractions for controlling a sensor/actuator network.

Gradients, for example, are a commonly used amorphous computing primitive. Clipping a gradient against a maximum distance produces a dilation operator

```
(def dilate (n source)
  (<= (gradient source) n))
```

which adapts to changing sources equivalently to Clement and Nagpal's active gradients[6].

We can then gradually raise the level of abstraction by building on our growing library of primitives, as in this bounding program

```
(def bound (source max boundary)
  (when (not boundary) (dilate max source)))
```

which returns true only within the boundaries containing the source. **Bound** can be used to re-express the program illustrated in Figure 4 as

```
(bound (sense :light) 2 (not (sense :sound)))
```

5.1 Coordinates

Another useful example is the coordinate system mechanism from Butera's paintable computing[5].

The coordinate system is derived from a provided source and destination. We will need to measure the distance between these places, which we can do with a **distance** operator that uses our previously defined **gradient** operator

```
(def distance (p1 p2)
  (letfed ((d 0 (reduce-nbrs d max (* (gradient p1) (if p2 1 0)))))
    d))
```

The paintable computing **channel** mechanism, which finds a wide path connecting two points, uses a trail-following operator to trace a gradient back up to the source. This is fairly fragile, so we instead find the trail geometrically by triangulation against **distance**, then widen it using **dilate**

```
(def channel (src dst width)
  (let* ((d (distance src dst))
         (trail (<= (+ (gradient src) (gradient dst)) d)))
    (dilate width trail)))
```

Implementing the **coordinates** mechanism requires one more operator: **choose-leader** is used to break symmetry by selecting a single location in the channel

```
(def choose-leader (selector)
  (letfed ((v (if selector (random 1.0) infinity))
           (minv v (reduce-nbrs minv min minv)))
    (and (< v infinity) (= v minv) v)))
```

The complicated coordinates mechanism (Figure 6) can then be defined as an operator which, despite comprising many complex operators is relatively straightforward for a programmer to create and understand.

```
(def coordinates (src dst width)
  (let* ((field (channel src dst width))
         (axis (channel src dst 1))
         (d1 (gradient src))
         (d2 (gradient dst))
         (dp (distance src dst))
         (buoy (choose-leader (and field (< d1 dp) (< d2 dp))))
         (y (/ (+ (* d2 d2) (- (* d1 d1)) (* dp dp)) (* 2 dp)))
         (x (sqrt (- (* d2 d2) (* y y))))
         (neg (bound buoy (+ width dp) (or (< y 0) (> y dp) axis))))
    (tuple (if neg (- x) x) y)))
```

5.2 Homeostasis

Long-range coordination can be accomplished by means of homeostatic operators which are always relaxing towards a correct solution.

For example, a simple time synchronization operator can be defined to converge toward a shared time by using a paired heartbeat and estimated lag. If the heartbeat arrives from a shorter route and advances time too fast, the lag drops as the gradient records the shorter distance; if communication disruptions interfere with the heartbeat, the lag gradient floats upwards, driving the time locally.

```
(def time-gradient (src)
  (letfed ((n infinity (if src 0 (+ (reduce-nbrs n min infinity)
                                    (reduce-nbrs nbr-lag max 0)))))
      n))

(def sync-time (src)
  (let ((lag (time-gradient src)))
    (letfed ((time 0 (if src (1+ time) (reduce-nbrs time max 0))))
      (+ time lag))))
```

Using this abstraction, we can establish long-range coordinated behavior like sinusoidal oscillations—useful for locomotion in distributed robotics or moving objects around an active surface.

This could be done with an externally supplied phase coordinate (established, for example, using Butera's algorithm above) and a heartbeat for synchronization

```
(def oscillate (heart phase period)
  (sin (/ (+ (sync-time heart) phase) period)))
```

or by calculating the oscillation vector internally. We can specify a vector in terms of a source and destination and find a wavefront perpendicular to that by calculating their bisector

```
(def bisector (a b)
```

```
  (let ((dif (abs (- (gradient a) (gradient b)))))
    (<= dif (reduce-nbrs nbr-dist max 0))))
```

which may need to be swollen to make it a boundary impermeable to communication

```
(def impermeable (set)
  (reduce-nbrs set or nil))
```

To break symmetry and allow the oscillation to propagate in one direction rather than flowing outward from the bisector, we define

```
(def abs->signed (val is-plus)
  (if (bound is-plus (maxdist) (impermeable (= val 0)) val (- val)))
```

and use it to negate the phase on the **src** side of a plane wave

```
(def plane-wave (src dst period)
  (let ((phase (abs->signed (gradient (bisector src dst)) dst)))
    (sin (/ (+ (sync-time src) (local-average phase)) period))))
```

All that remains, then, is to set the period of the wave to the length of the vector:

```
(def oscillate (src dst)
  (plane-wave src dst (distance src dst)))
```

6 Implementation and Verification

Experimental verification is conducted using a simulator and an implementation of Proto for Mica2 motes.

Motes present significant challenges for any language implementation, but especially for high level languages like Proto. The Mica2 motes are 8 bit microcontrollers running at 16MHz, have only a scant 4KB of RAM, run on two AA batteries, and contain a relatively slow radio that can send a maximum of approximately thirty 32 byte packets per second.

The biggest challenge of getting Proto to run on the motes is to get the operator trees to fit in the 4K of RAM on the Atmel Mega128. This tiny memory forces a very simple memory management scheme. Fortunately, stream processing permits data structures to be mostly preallocated when trees are opened and reused across rounds.

Each mote has a C machine structure which provides the Proto discrete level operating system data structures for the running scripts. Specifically, it holds the machine id, script, version, operator tree, time-stamp, export tuple, neighborhood table, and sensor and actuator data.

The neighborhood data is a limited size table of associations between machine id and import tuples. The neighborhood table is populated dynamically and stale entries are replaced. In addition to an ID and import values, each entry contains a timeout counter tracking how long it has been since the last update, and an area estimate used for integration. At the end of each evaluation round, exposed state is calculated and added to an export buffer for later transmission.

On the motes, we use a max table size of 8 neighbors and a single packet export mechanism. Each export packet can support up to six number values in our current implementation. It is straightforward to support multi-packet exports and we plan to do so in the near future.

Each primitive operator has a class structure representing static properties and a corresponding C structure representing its runtime values. An operator class contains the operator protocol in the form of function pointers for the construction, opening, and closing of operator instances, and the execution of operator code. Additionally, the operator class contains the number of exported values, operator children, local state data,

and construction arguments, and contains the corresponding bytecode. Operator instances contain a pointer to the operator class, its time-stamp, output data, operator specific data (e.g. the **reduce-nbrs** operator instances hold an offset into the export/neighborhood tuple), and pointers to operator children.

Proto scripts are written on a PC, translated by the Proto compiler to bytecodes and then injected packet by packet over the air into the sensor network through a base station connected by serial cable to the PC. Received scripts are virally forwarded to neighboring motes using a mechanism similar to those described in [13], [11], and [15]. The programmer then needs only to program a single device and the code will spread through the network to upgrade the rest. To prevent conflicts during an upgrade process, each state broadcast also contains a version number, allowing devices to ignore state from different versions. Currently, only single packet scripts are supported but it is straightforward to implement the multi-packet case. Once the complete set of script packets are loaded onto a mote, the script is interpreted on a virtual stack machine producing a new operator tree. Once constructed and installed, the operator tree is executed top down, each operator executing and producing a value once for each round.

The Proto compiler performs type inference and method selection while translating scripts to byte codes. Type inferencing allows the resolution of overloaded operators into efficient type specific operators and eventual bytecodes. In order to support full type inference and the generation of type specific bytecodes, all script operators are inlined and specialized.

To ease porting, Proto is implemented in such a way as to minimize platform specific code. The platform independent code consists of the neighborhood management, script dissemination and interpretation, and primitive operators. Primitive operators are written in stylized C that permits maximal code sharing. Currently, the total amount of platform independent code is 1505 lines.

The platform specific code consists of low level timing, low level network code, and sensor/actuator code, and currently amounts to 270 lines on the Mica2 motes. The timing code phases the execution and export stages. On the mote, it is implemented using a TinyOS timer event firing every 128 milliseconds. This can easily be sped up in the future. The low level networking code sends and receives script and neighborhood packets. Packets arrive as events but their processing is handled in tasks to ensure synchronization of global data. Finally, the sensors and actuators API is implemented for each of the mote inputs and outputs. The total size of the compiled code including Proto and TinyOS is 31252 bytes.

The simulator permits the running of much larger networks (over 10,000 nodes), larger applications, flexible visualization, and friendlier code development and debugging. As in the mote port, only a small amount of platform specific code is necessary to implement. The bulk of the simulator code facilitates visualization, code development, and debugging.

6.1 Verification Example

Verification begins in the simulator. For example, Figure 7 shows the plane-wave based oscillator running in simulation on 10,000 nodes, using hopcount for distance and lag. Once a program runs in simulation, it can be transferred to Motes which provide ground truth as to whether our building blocks compose correctly, respecting their prescribed interface. For example, Figures 8 shows a small group of Motes running an oscillator with phase and leadership supplied. This is specified completely by the implementation Proto code:

```
(def id (x) x)
(def max (x y) (if (< x y) y x))
(def min (x y) (if (< y x) y x))
(def maxhops () 99)
(def gradient (src)
  (letfed ((n (maxhops) (if src 0 (+ 1 (fold-hood min (maxhops) id n))))) n))
(def sync-time (src)
```

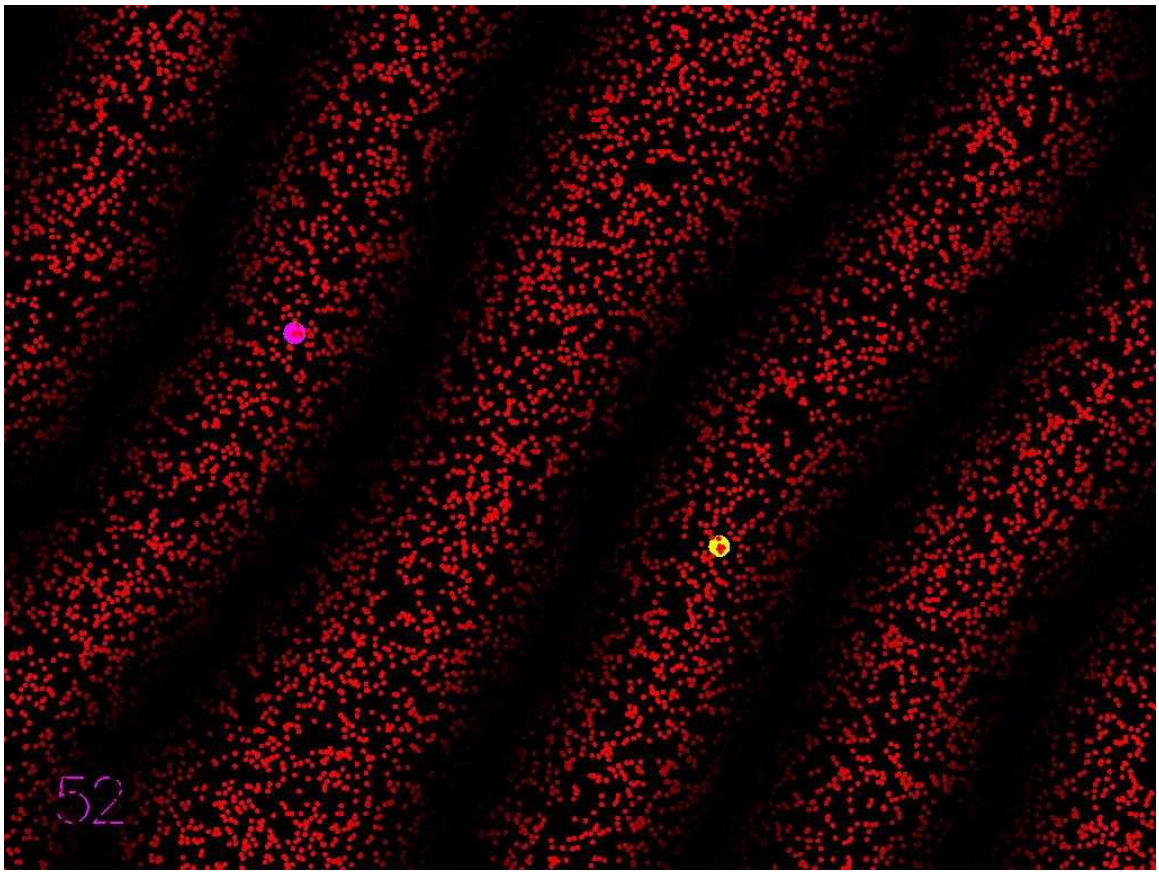

Figure 7: Plane wave oscillator running on 10,000 simulated devices. The period and direction of the wave is determined by the placement of source (yellow) and destination (magenta) markers in the devices' sensor field.

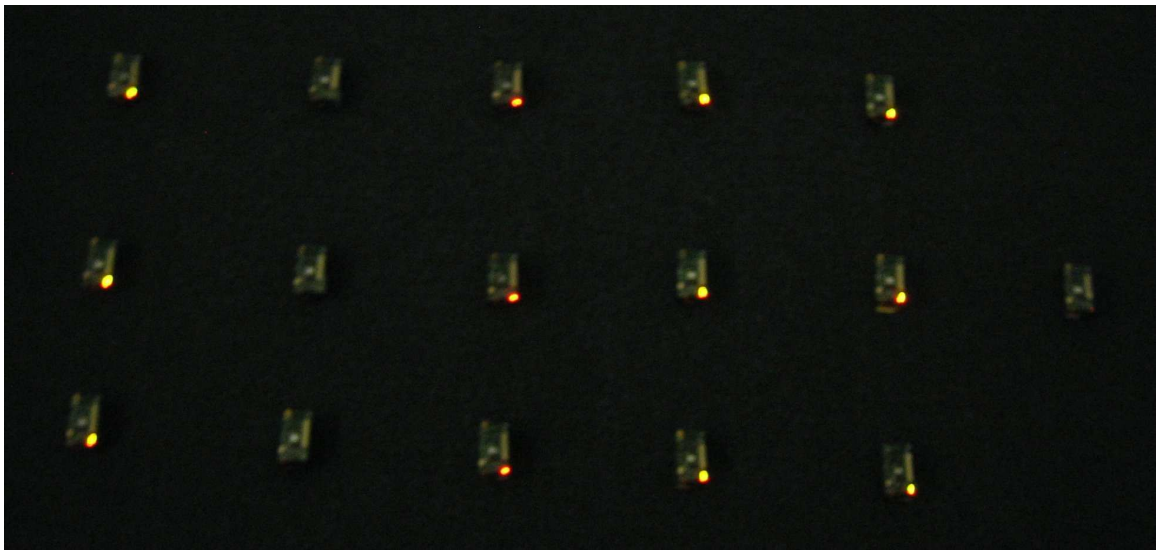

Figure 8: A group of Motes running the oscillator program, displaying the output on their LEDs. The motes are given a synthetic coordinate for their phase.

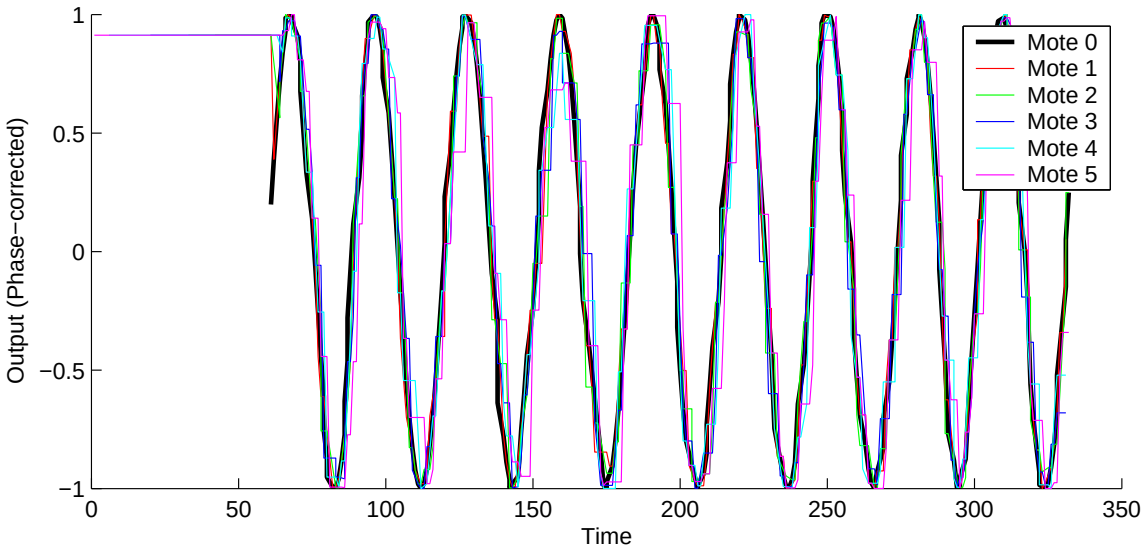

Figure 9: Output of a group of six Motes in a line running the oscillator program, subtracting phase. The Motes synchronize and begin oscillating shortly after the leader, Mote 0, is turned on. Dropped packets and variable execution rates cause the executions on the various Motes to diverge rapidly, while the **time-sync** operator continually draws them back towards synchrony.

```
  (let ((lag (gradient src)))
    (letfed ((t 0 (if src (+ 1 t) (fold-hood max 0 id t))))
      (+ t lag))))
(def osc (src pos period)
  (sin (/ (+ (sync-time src) pos) period)))
(leds (/ (+ (osc (sense 1) (elt (coord) 0) 5) 1) 2))
```

where **fold-hood** is equivalent to **reduce-nbrs**, **leds** is an actuator for the Mote LEDs, **coord** senses the supplied phase and **(sense 1)** senses leadership. This evaluates to a script of 98 bytecodes and an operator tree of 658 bytes.

The Motes synchronize and begin oscillating shortly after the leader, Mote 0, is turned on, displaying the output of the oscillation on their LEDs. Plotting the values and subtracting the supplied phase difference (Figure 9), we find that the composition works and the oscillator is, as expected, diverging due to communication difficulties and the variable rate of execution on individual Motes, but is continually drawn back toward synchrony.

7 Future Directions

Our work on Proto and the amorphous medium abstraction has laid a groundwork on which the discipline of self-managing systems engineering can continue to develop. As one would expect in a young field, there are many open problems of varying difficulty.

Much work remains to be done on the practical matters of implementation. Although these problems are less novel, solving them and integrating the solutions into the overall infrastructure is necessary to provide a solid foundation for ongoing research. A few particularly noteworthy implementation needs:

- Energy management in the discrete kernel, such as adjusting transmission frequency and contents to lower expenditure when data is changing slowly.
- Improved bandwidth utilization in the discrete kernel via TDMA, CSMA/CD, or other wireless communication algorithms.
- Bringing the Proto implementation into closer alignment with Proto's semantics.
- Optimization of code by the Proto compiler for both time and space. Much more space and time efficient representations of operator trees are possible. In particular, we will be investigating the placement of some of the static operator data in program memory and on the fly code generation techniques in the near future.
- Verification of larger programs, either by adding multi-packet support for Motes or moving to less constrained hardware.

Moving beyond implementation, it is an open question what types of abstractions are most intuitive for global control of spaces. Candidates in the form of distributed algorithms from amorphous computing and elsewhere need to be imported to Proto and analyzed within its context.

Although we have presented a means of composition, a tighter characterization of composed systems is likely possible. In particular, some amorphous computing algorithms generally run faster and more resiliently than the loose bounds established for them, and may effectively pipeline when composed.

Finally, as the discipline is developed, it may be extended into domains beyond sensor/actuator networks. In particular, the amorphous medium abstraction should hold for any problem in which the network of computational devices approximates the topology of the problem being solved. This suggests that problems in non-spatial domains such as semantic networks may be solvable with the same techniques: our preliminary investigations suggest that Proto should be usable in any domain approximated by a network with high diameter and small neighborhood size.

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. Technical Report AIM-1665, MIT, 1999.

[2] Jonathan Bachrach. Gooze: a stream processing language. In *Lightweight Languages 2004*, November 2004.

[3] Jacob Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, September 2004.

[4] Jacob Beal and Gerald Sussman. Biologically-inspired robust spatial programming. Technical Report AI Memo 2005-001, MIT, January 2005.

[5] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

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

[7] 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.

[8] 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*, October 2004.

[9] Adam Eames. Enabling path planning and threat avoidance with wireless sensor networks. Master's thesis, MIT, June 2005.

[10] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[11] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[12] S. P. Jones and J. Hughes. Report on the programming language haskell 98., 1999.

[13] James McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT, 2004.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised*. MIT Press, 1997.

[15] Andrew Sutherland. Towards rseam: Resilient serial execution on amorphous machines. Master's thesis, MIT, 2003.