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—much as components such as capacitors, transistors, and resistors capture electromagnetism principles for electronic engineering.

To transform a science into an engineering discipline, we must

- decouple aspects of the problem from one another,
- identify an operating range,
- create standard interfaces for composition,
- identify primitive components that conform to the standards, and
- create abstraction rules that hide the complexity of systems of components.

We’ve begun this process in the domain of sensor/actuator network applications, observing that in many applications the deployed network approximates a physical space and that the space, rather than the network, is being programmed. This observation lets us use the amorphous medium abstraction to decouple self-management problems. So, global behavior descriptions in our Proto language can be compiled automatically into locally executed code that produces emergent phenomena matching the global description. We’ve experimentally verified our code both in simulation and (for small programs) on a network of sensor/actuator nodes called Mica2 motes.

Decoupling: Amorphous medium

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

The amorphous-medium abstraction captures the divide between specification and implementation; it’s a continuous computational material filling the space of interest. Every point in the medium is a computational device that independently executes the same code as every other device in the medium. (Executions diverge owing to differences in sensor values, randomness, and each device’s interactions with nearby devices.) Each device has a neighborhood of devices less than d units of distance away to which it exposes its internal state. Conversely, a device can read the internal state of devices in its neighborhood, obtaining values lagged proportionally to the distance separating them.

We can’t, of course, build a continuous medium containing uncountably many infinitely small computers. We can, however, approximate it by scattering a discrete set of devices throughout 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 that capture only a few electromagnetism phenomena. In both cases, restricting the behavior range supports engi-
neering abstractions that ignore much of the underlying system’s complexity.

The amorphous-medium abstraction separates the space being programmed from the devices carrying out the program, letting us decompose self-managing systems engineering into three layers of abstraction—global, local, and discrete (see figure 1). Each layer is supported by its own infrastructure component; this decouples 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 that approximately emulates 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 regions’ behavior.

Table 1 illustrates some design problems that this approach separates:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Infrastructure</th>
<th>Coordination</th>
<th>Control flow</th>
<th>Failures</th>
<th>Energy efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Proto libraries</td>
<td>Region</td>
<td>Homeostasis</td>
<td>User</td>
<td>Coordinations</td>
</tr>
<tr>
<td>Local</td>
<td>Proto compiler</td>
<td>Neighborhood</td>
<td>Stream network</td>
<td>Region</td>
<td>Reductions</td>
</tr>
<tr>
<td>Discrete</td>
<td>Kernel</td>
<td>Device</td>
<td>Rounds</td>
<td>Device</td>
<td>Packets</td>
</tr>
</tbody>
</table>

**Operating range: Amorphous computers**

We can consider a sensor/actuator network in which devices communicate only with nearby neighbors to be an amorphous computer. 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’ve chosen the amorphous-computer model for two reasons. First, its biologically inspired specifications imply the self-management issues

For information on amorphous-medium research and other related research, see the sidebar “Related Work on Engineering Self-Managing Systems.”

Figure 1. Our approach decouples self-management problems by decomposing self-managing systems 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 we build library code to allow description of the behavior of regions of the medium at the global layer. $P$ indicates a particular, arbitrary point.
Our research on self-managing systems engineering draws on previous work in many fields: our contribution lies in integrating the pieces that others have developed. Previous research on amorphous-medium languages proposed the amorphous-medium abstraction and general strategies for controlling an amorphous medium; this article describes a practical implementation. Other research on amorphous-computing languages has shared the same general goals but has been directed more toward problems of morphogenesis and pattern formation than general computation. Examples include Daniel Coore’s research on topological patterns and research on geometric-shape formation by Radhika Nagpal; Attila Kondacs; and Justin Werfel, Yaneer Bar-Yam, and Radhika Nagpal. A notable exception is William Butera’s work on paintable computing, which allows general computation but lacks an abstraction barrier separating an applications programmer from low-level details of network operation.

An alternate approach to engineering self-organizing systems is rooted in gossip communication, a technique we also use. However, gossip communication deploys less-powerful abstractions than our approach does, because it’s solving more-general networking problems. More distant are approaches based on alternate computational paradigms such as chemical computation and membrane computation. Sensor network researchers have proposed several other high-level programming abstractions to enable the programming of large networks. For example, GHT (Geographic Hash Table) provides a hash table abstraction for storing data in the network, and TinyDB focuses on gathering information through query processing. Both of these approaches, however, are data-centric rather than computation-centric and don’t provide guidance on how to do distributed manipulation of gathered data. TinyOS and the Hood abstraction provide useful general programming tools—indeed, our implementation of Proto on motes uses TinyOS—but the abstractions are less powerful and lead to bulkier, less reusable code. More similar is the Regiment language, which uses a stream-processing abstraction to distribute computation across the network. Regiment, however, is distributed only when the compiler finds optimization opportunities, and significant challenges remain in adapting its programming model to the sensor network environment.

Finally, Proto’s structure as a dynamic network of streams is strongly influenced by Jonathan Bachrach’s previous work on Gooze, as are many compilation strategies used to compact Proto code for execution on motes. There’s a long tradition of stream processing in programming languages. The closest and most recent work is Functional Reactive Programming, which is based on Haskell, a statically typed programming language with lazy evaluation semantics. FRP has been demonstrated on robotics, graphics, and user interface design. These systems focus less on runtime space and time efficiency than our approach does, and the type system is firmly wedded to Haskell, with all its strengths and weaknesses.

References
10. A. Demers et al., “Epidemic Algorithms for Replicated Database of robustness, distribution, and scalability. Second, real-world sensor/actuator networks are growing rapidly in scale and capability, bringing them into closer alignment with the amorphous-computer model.

In particular, we designed Proto and its infrastructure to operate on sensor/actuator networks with these properties:

- The number of devices \( n \) might range from dozens to billions.
- Devices are distributed arbitrarily through space and collaborate via unreliable broadcast to neighbors no farther than \( d \) distance away.
- Devices move much more slowly than communication, i.e., at all.
- Memory and processing aren’t limiting resources. (Profligate expenditure of either is still bad, and memory is an important constraint for our mote implementation.)
- Execution is partially synchronous—each device has a clock that ticks regularly, but frequency might vary by up to \( e \), and clocks have an arbitrary initial time and phase.
- The networks don’t provide naming, routing, and coordinate services. (These services might be made available as sensor values, with appropriate characterization of reliability and error.)
- Arbitrary point and region stopping failures and joins might occur, including changes in the network’s connectedness.

Our operating-range specification doesn’t directly address energy consumption, although it has been a concern in implementation. Each abstraction layer can independently address energy issues, however, and we expect that optimizing the discrete-kernel implementation...
will extract a large savings. Although some excess energy expenditure will likely remain, we consider the gain in engineering capability to be worth moderately inefficient energy expenditure.

**Abstraction and composition: Proto**

Proto’s semantics capture interface standards, primitive components, and abstraction rules. It combines the dynamic stream networks of Gooze\(^2\) with previous work on amorphous-medium languages.\(^3\)\(^4\)

**Primitives and composition**

Proto programs 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. To compose programs, we use functional operators. So, the expression \((+ 2 5)\) yields a program that emits a stream of sevens. The compiler evaluates the operator and operand expressions using the same rules as in Scheme. The operands are streams, and the operator constructs a stream of output values from sets of values that are 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 that consume them (see figure 2). The root’s output stream serves as a reference to the program ascending from it.

**Types**

Proto is strongly typed like ML\(^5\) (Meta-Language) and Haskell,\(^6\) and, unlike statically typed languages such as C, types are inferred automatically from literals and function calls. So, users rarely need to deal with types, but they’re useful for describing the various kinds of available data and the built-in operators’ signatures.

Proto permits Boolean, character, number, and symbol data types. We can combine these base types 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.

**I/O**

The `sense` operator accesses input from the outside world or other programs running on a device. For example,

\[
\text{(sense :light)}
\]

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

\[
\text{(actuate :sound (sense :light))}
\]

sends the light value to the sound actuator. (The mechanism for binding sensors to names is implementation dependent, as are the value when `sense` is applied to an unbound name and the result of multiple streams being sent to the same actuator.)

**State**

To establish persistent state, Proto uses delay loops specifying an initial value and an expression for calculating the next value from

\[
\text{Figure 2. A Proto program is a network of operator instances ascending from a single root. The root’s output stream serves as a reference to the program.}
\]
current values. For example, the expression

\[
(\text{letfed} ((i 0 (1 + i)))
    i)
\]

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

**Communication**

Unlike discrete networks, each point in an amorphous medium has an infinite number of neighbors. So, 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

\[
(\text{reduce-nbrs} (\text{sense} :\text{light}) \text{or} \text{nil})
\]

The first argument is the value to reduce, the second is the reduction function, and the third is the reduction’s initial value. 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, we can perhaps better understand `reduce-nbrs` as a transform that operates on nearby space rather than as communication.

In general, we don’t want to tie our program’s behavior to neighborhood sizes, so Proto provides special operators for measuring a neighbor’s space distance, time distance, and volume: `nbr-dist`, `nbr-log`, and `infinite`, respectively. (These might 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.)

Thus, for example, we can measure the distance to a light with a gradient flowing from the source:

\[
(\text{letfed} ((n \text{infinity} (+ 1 (\text{if} \text{src} 0 (\text{reduce-nbrs} (+ n \text{nbr-dist}) \text{min} \text{infinity})))))
    (\text{if} \text{src} 0
        (\text{reduce-nbrs} (* 1 \text{infinite}) + 0)
        (\text{reduce-nbrs} (* \text{infinite} 0))))
\]

(Biological systems often use chemical diffusion from a source as a distance measure, and various distributed-computing fields have co-opted “gradient” by analogy to mean a distance-to-source measurement created by gossip.)

The addition of the 1 drives the distance upward at those points that aren’t connected to the source, allowing the gradient to adapt to changing sources in the same way as Lauren Clement and Radhika Nagpal’s active gradients. 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. So, \(n\) is pegged to zero at light sources and floats up by one for each distance unit. Each point converges to the estimated distance to the nearest light source.

When compiling Proto expressions into executable code, the compiler identifies the values that `reduce-nbrs` expressions need so that the discrete kernel can export them to its neighbors whenever they change. Any reduction that 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 `infinite` operators to allow integer. For example,

\[
(\text{reduce-nbrs} (* (\text{sense} :\text{light}) \text{infinite}) + 0)
\]

finds the average light value in each point’s neighborhood (the second `reduce-nbrs` expression normalizes the integral). However, we must subtly redefine some operators such as `random` to have a compatible amorphous-medium semantics and discrete-kernel implementation.

**Abstraction**

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

\[
(\text{def} \text{gradient} (\text{src})
    (\text{letfed} ((n \text{infinity} (+ 1 (\text{if} \text{src} 0 (\text{reduce-nbrs} (+ n \text{nbr-dist}) \text{min} \text{infinity})))))
    (\text{reduce-nbrs} (* 1 \text{infinite}) + 0))
\]

and a generic averaging operator

\[
(\text{def} \text{local-average} (x)
    (/ (\text{reduce-nbrs} (* x \text{infinite}) + 0)
        (\text{reduce-nbrs} (* 1 \text{infinite} + 0)))
\]

We can then use these operators in expressions, including definitions of operators at a higher abstraction level. So, for example, we can write the expression

\[
(\text{let} ((\text{n} \text{infinity} (+ 1 (\text{if} \text{src} 0 (\text{reduce-nbrs} (+ n \text{nbr-dist}) \text{min} \text{infinity})))))
    (\text{reduce-nbrs} (* 1 \text{infinite} + 0)))
\]

that outputs `true` anywhere within two units’ distance of a light.

**Execution**

Pulling a value from a program’s output stream initiates a round of execution. (The discrete kernel generally discards these values, so a program’s ultimate goal must be achieved through actuation.) The discrete kernel then distributes execution up the network as operators pull values from their inputs. If an operator doesn’t pull a value from one of its inputs, the upstream operator doesn’t execute and hibernates, discarding any internal state until it begins executing again and reboots. For example, assuming a Boolean sound sensor, the program

\[
(\text{when} (\text{sense} :\text{sound})
    (\text{let} ((\text{src} \text{true})
        (\text{reduce-nbrs} (* 1 \text{infinite} + 0))))
\]

runs the gradient only where there’s sound. Consequently, points within two units’ distance but separated by a quiet area will output `false`, because the gradient isn’t running in the intervening area (see figure 3).

The discrete kernel instantiates the expression associated with an interpreted operator into an encapsulated network once for each active instance of the operator. When an instance hibernates, the discrete kernel discards this network, along with any state in its loops, and restarts the network from scratch when it next becomes active. Among other things, this allows recursion because the discrete kernel constructs a potentially infinite network structure only for the levels in use.

A process module’s output might serve as input to more than one other module. For example,
always runs the gradient but squares the output when there’s sound (see figure 4). 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 execution round. Conversely, as long as at least one process module pulls a value, the subprogram will execute.

**Miscellany**

Proto lets a programmer define new primitive operators. Although primitive operators aren’t strictly necessary, they’re generally faster and more memory efficient because Proto can perform the calculations 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, Adam Eames’s algorithm for distributed discovery of minimum threat paths requires 2,000 lines of nesC code, while an equivalent Proto implementation is a mere 25 lines long.

**Raising the abstraction level**

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

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

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

which adapts to changing sources equivalently to Clement and Nagpal’s active gradients.

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

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

which returns true only within the boundaries containing the source. We can use `bound` to reexpress the program in figure 4 as

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

**Coordinates**

Another useful example is the coordinate system mechanism from William Butera’s paintable computing.

We derive the coordinate system from a provided source and destination. We need to measure the distance between these places, which we do with a `distance` operator that uses our previously defined `gradient` operator:

```lisp
(def distance (p1 p2)
  (letf ((d 0 (reduce-nbrs d max (* (gradient p1) (if p2 1 0))))
        (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 (< dp (y dp) axis))))
        (tuple (if neg (next x x) x)))))
```

Figure 3. All communication proceeds through neighborhoods, so a gradient (gray) spreading from regions with light (black) that runs only when there’s sound (white boxes) can’t cross a gap where there is no sound.

Figure 4. Subprograms might feed multiple inputs. The subprogram caches its output so that it executes only the first time its output stream is pulled in a given round.
Homeostasis

To accomplish long-range coordination, we use homeostatic operators that are always relaxing toward a correct solution.

For example, we can combine a heartbeat and an estimate of lag to define a simple time synchronization operator that converges toward a shared time. If the heartbeat arrives from a shorter route and advances time too quickly, the lag drops as the gradient records the shorter distance. If communication disruptions interefere with the heartbeat, the lag gradient floats upward, driving the time locally:

\[
\text{time-gradient} = \frac{\text{heart} + \text{lag}}{\text{period}}
\]

We could do this with an externally supplied phase coordinate (established, for example, using Butera’s algorithm) and a heartbeat for synchronization,

\[
\text{oscillate} = \sin \left( \frac{\text{sync-time} + \text{local-average phase}}{\text{period}} \right)
\]

or by calculating the oscillation vector internally. We can specify a vector in terms of a source and destination and find a wave front perpendicular to that by calculating its bisector,

\[
\text{bisector} = \left( \text{abs} \left( \frac{\text{gradient} \cdot \text{dst axis}}{\text{vector}} \right) \right)
\]

which might need to be swollen to make it a boundary impermeable to communication:

\[
\text{impermeable} = \text{reduce-nbrs} \text{nbr set or nil)}
\]

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

\[
\begin{align*}
\text{abs}->\text{signed} & = \begin{cases} 
\text{val} & \text{if } \text{is-plus} \\
-\text{val} & \text{otherwise}
\end{cases} \\
\text{plane-wave} & = \begin{cases} 
\text{src} & \text{if } \text{period} \\
\text{dst} & \text{else}
\end{cases} \\
\end{align*}
\]

and use it to negate the phase on a plane wave’s \text{src} side:

\[
\begin{align*}
\text{oscillate} & = \sin \left( \frac{\text{sync-time} + \text{local-average phase}}{\text{period}} \right) \\
\text{plane-wave} & = \text{distance} \text{src} \text{dst})
\end{align*}
\]

Implementation and verification

We conducted experimental verification using a simulator and an implementation of Proto for Mica2 motes.

The discrete-kernel implementation

Motes present significant challenges for any language implementation, but especially for high-level languages such as Proto. Mica2 motes are 8-bit microcontrollers running at 16 MHz, have only a scant 4 Kbytes 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 fitting the operator trees in the 4 Kbytes of RAM on the ATmega128 memory card. 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 then reused across rounds.

Each mote has a C machine structure that 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 the 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 both a time-out counter tracking the time elapsed 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 maximum table size of eight neighbors and a single-packet export mechanism. Each export packet can support up to six number values in our current implementation. Supporting multipacket exports is straightforward, 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 constructing, opening, and closing operator instances and for executing operator code. Additionally, the operator
class contains the number of exported values, operator children, local-
state data, and construction arguments and the corresponding byte-
code. Operator instances contain a pointer to the operator class, its
time stamp, output data, and operator-specific data (for example, the
reduce-nbrs operator instances hold an offset into the export/neighbor-
hood tuple), and pointers to operator children.

Proto scripts are written on a PC, translated by the Proto compiler
to bytecodes, and injected packet by packet over the air into the sen-
sor network through a base station connected by serial cable to the
PC. The implementation virally forwards received scripts to neigh-
boring motes, using a mechanism similar to those that James
McLurkin, Jonathan Hui and David Culler, and Andrew Suther-
land have described. The programmer then needs only to program
a single device, and the code will spread through the network to
upgrade the other devices. To prevent conflicts during an upgrade,
each state broadcast also contains a version number, letting devices
ignore state from different versions. The current implementation sup-
ports only single-packet scripts, but implementing multipacket scripts
would be straightforward.

Once the complete set of script packets are loaded onto a mote, a
virtual stack machine interprets the script, producing a new operator
tree. Once the kernel has constructed and installed the operator tree,
the tree executes 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 bytecodes. Type inferencing allows the
resolution of overloaded operators into efficient type-specific oper-
ators and eventual bytecodes. To support full type inference and the
generation of type-specific bytecodes, all script operators are inlined
and specialized.

To ease porting, we’ve implemented Proto to minimize platform-
specific code. The platform-independent code consists of the neigh-
brhood management, script dissemination and interpretation, and
primitive operators. Primitive operators are written in stylized C that
permits maximal code sharing. Currently, we have 1,505 lines of plat-
form-independent code.

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, we’ve implemented it using a TinyOS
timer event firing every 128 milliseconds. We can easily speed this
up in the future. The low-level networking code sends and receives
script and neighborhood packets. Packets arrive as events, but the
implementation processes them in tasks to ensure synchronization
of global data. Finally, the kernel implements the sensors and actu-
ators API for each mote input and output. The compiled code, includ-
ing Proto and TinyOS, constitutes 31,252 bytes.

The simulator permits the running of much larger networks (over
10,000 nodes), larger applications, flexible visualization, and friend-
lier code development and debugging. As in the mote port, we need
to implement only a small amount of platform-specific code. The
bulk of the simulator code facilitates visualization, code develop-
ment, and debugging.

Verification example

Verification begins in the simulator. For example, figure 6 shows
the plane-wave-based oscillator running in simulation on 10,000
nodes, using hop count for distance and lag.

Once a program runs in simulation, we can transfer it to the motes;
this provides ground truth as to whether our building blocks com-
pose correctly, respecting their prescribed interface. For example,
figure 7 shows a small group of motes running an oscillator with
phase and leadership supplied. This is specified completely by the
implementation Proto code:

\[
(\text{def} \ \text{gradient} \ (\text{src}) \n= \begin{cases} \text{letfed} \ ((n (\infty) (\text{if} \ \text{src} 0 (+ 1 (\text{fold-hood min} (\infty) n)))) \\ n)) \\ \text{def} \ \text{sync-time} \ (\text{src}) \n= \begin{cases} (\text{let} \ ((\text{lag} (\text{gradient src}))) \\ (\text{lefted} ((t 0 (\text{if} \ (\text{src} (+ 1 t) (\text{fold-hood max 0 t})))) \\ (+ 1 \text{lag})))) \\ \text{def} \ \text{osc} \ (\text{src pos period}) \\ (\text{sin} (/ (+ (\text{sync-time src} \text{pos period}))) \\ (\text{leds} (/ (+ (\text{osc} (\text{sense 1} (\text{left (coord) 0} 5) 1) 2)) 
\end{cases}
\]

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 oscillation’s output on their LEDs. Plotting the values and subtracting the supplied phase difference (see figure 8), we find that the composition works. The oscillator, as we expected, diverges owing to communication difficulties and the variable rate of execution on individual motes, but the time synchronization operator continually draws it back toward synchrony.

Our work on Proto and the amorphous-medium abstraction has laid the groundwork on which the discipline of self-managing systems engineering can continue to develop. As you would expect in a young field, many problems of varying difficulty remain. Much research is necessary 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. Here are 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 (time-division multiple access), CSMA/CD (carrier sense multiple access with collision detection), or other wireless-communications algorithms.
• More closely aligning the Proto implementation with Proto’s semantics.
• Optimizing code by the Proto compiler for time and space. Much more space- and time-efficient representations of operator trees are possible. In particular, we’ll investigate both placing static operator data in program memory and on-the-fly code generation in the near future.
• Verification of larger programs, by either adding multipacket support for motes or moving to less-constrained hardware.

Moving beyond implementation, it’s 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’ve 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 might effectively pipeline when composed.

Finally, as the discipline of self-managing systems engineering develops, it could 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 these techniques might be able to solve problems in nonspatial domains such as semantic networks. Our preliminary investigations suggest that Proto should be usable in any domain approximated by a network with a high diameter and small neighborhood.

References


