Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-042 May 27, 2006

Infrastructure for Engineered Emergence on Sensor/Actuator Networks

Jacob Beal and Jonathan Bachrach

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

Infrastructure for Engineered Emergence on Sensor/Actuator Networks

Jacob Beal and Jonathan Bachrach, *Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory*

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.

The ability to control emergent phenomena depends on decomposing them into aspects susceptible to independent engineering. For spatial self-managing systems, the amorphous-medium abstraction lets you separate the system's specification from its implementation.

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-

1541-1672/06/$20.00 © 2006 IEEE
Published by the IEEE Computer Society

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:

- Our approach's implementation 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.
- This approach describes global control as homeostatic processes continually moving regions toward a desired behavior. It implements these processes as networks of streams in the local layer, which compile to update code executed periodically in the discrete layer.
- Different layers handle different failure modes: the neighborhood abstraction masks individual device crashes, the homeostatic primitives handle outside events that destroy regions of the network, and a clean global-layer interface minimizes bugs in the user's code.
- Assuming that the cost of communication dominates energy consumption, the amount of communication depends on how many long-range coordination operations occur in the global layer, how many reductions over neighbor state are used to implement coordination at the local layer, and how many packets are transmitted and received to implement shared neighbor state at the discrete layer.

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

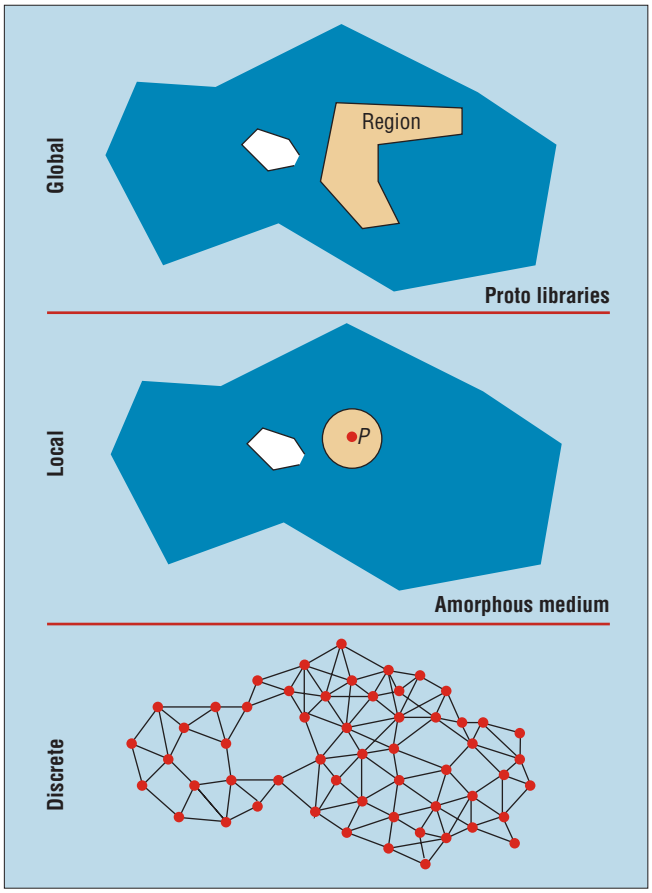

Figure 1. Our approach decouples self-management problems by decomposing self-managing systems into three abstraction layers: global, local, and discrete. Interactions between individual devices in the discrete layer emulate an amorphous medium. The local layer describes the behavior of points in the 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.

Operating range: Amorphous computers

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

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

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

Related Work on Engineering Self-Managing Systems

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;[1,2] 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[3] and research on geometric-shape formation by Radhika Nagpal;[4] Attila Kondacs;[5] and Justin Werfel, Yaneer Bar-Yam, and Radhika Hagpal.[6] 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.[7]

An alternate approach to engineering self-organizing systems is rooted in gossip communication,[8–10] 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[11,12] and membrane computation.[13]

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,[14] and TinyDB focuses on gathering information through query processing.[15] 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[16] and the Hood abstraction[17] 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.[18] 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,[19] 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,[20] which is based on Haskell,[21] a statically typed programming language with lazy evaluation semantics. FRP has been demonstrated on robotics,[22] graphics,[20] and user interface design.[23] 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

1. J. Beal, "Programming an Amorphous Computational Medium," *Unconventional Programming Paradigms*, LNCS 3566, Springer, 2005.
2. J. Beal and G. Sussman, *Biologically Inspired Robust Spatial Programming*, AI Memo 2005-001, Computer Science and Artificial Intelligence Laboratory, Massachusetts Inst. of Technology, 2005.
3. D. Coore, "Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer," PhD thesis, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 1999.
4. R. Nagpal, "Programmable Self-Assembly: Constructing Global Shape Using Biologically Inspired Local Interactions and Origami Mathematics," PhD thesis, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 2001.
5. A. Kondacs, "Biologically Inspired Self-Assembly of Two-Dimensional Shapes, Using Global-to-Local Compilation," *Proc. 2003 Int'l Joint Conf. Artificial Intelligence* (IJCAI 03), Morgan Kaufmann, 2003, pp. 633–638.
6. J. Werfel, Y. Bar-Yam, and R. Nagpal, "Building Patterned Structures with Robot Swarms," *Proc. 2005 Int'l Joint Conf. Artificial Intelligence* (IJCAI 05), Int'l Joint Conf. Artificial Intelligence, 2005, pp. 1495–1502.
7. W. Butera, "Programming a Paintable Computer," PhD thesis, Media Laboratory, Massachusetts Inst. of Technology, 2002.
8. O. Babaoglu, M. Jelasity, and A. Montressor, "Grassroots Approach to Self-Management in Large-Scale Distributed Systems," *Unconventional Programming Paradigms*, LNCS 3566, Springer, 2005, pp. 286–296.
9. K. Birman, S. Guha, and R. Murty, "Scalable, Self-Organizing Technology for Sensor Networks," *Advances in Pervasive Computing and Networking*, 2004, Springer, pp. 1–15.
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 r distance away.
- Devices move much more slowly than communication, if 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 ϵ, 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

Management," *Proc. 6th Ann. ACM Symp. Principles of Distributed Computing*, ACM Press, 1987, pp. 1–12.

11. J.P. Banatre and D. Le Metayer, "The Gamma Model and Its Discipline of Programming," *Science of Computer Programming*, vol. 15, no. 1, 1990, pp. 55–77.
12. G. Berry and G. Boudol, "The Chemical Abstract Machine," *Theoretical Computer Science*, vol. 96, no. 1, 1992, pp. 217–248.
13. G. Paun, "Computing with Membranes," *J. Computer and System Sciences*, vol. 61, no. 1, 2000, pp. 108–143.
14. S. Ratnasamy et al., "GHT: A Geographic Hash Table for Data-Centric Storage," *Proc. 1st ACM Int'l Workshop Wireless Sensor Networks and Applications*, ACM Press, 2002, pp. 78–87.
15. S.R. Madden et al., "Supporting Aggregate Queries over Ad Hoc Wireless Sensor Networks," *Proc. Workshop Mobile Computing and Systems Applications*, IEEE CS Press, 2002, pp. 49–58.
16. D. Gay et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 1–11.
17. K. Whitehouse et al., "Hood: A Neighborhood Abstraction for Sensor Networks," *Proc. 2nd Int'l Conf. Mobile Systems, Applications, and Services*, ACM Press, 2004, pp. 99–110.
18. R. Newton and M. Welsh, "Region Streams: Functional Macroprogramming for Sensor Networks," *Proc. 1st Int'l Workshop Data Management for Sensor Networks* (DMSN), Morgan Kaufmann, 2004.
19. J. Bachrach, "Gooze: A Stream Processing Language," *Proc. Lightweight Languages 2004*, 2004; recorded presentation at http://web.mit.edu/webcast/csail/mit-csail-lwl-04dec2004-afternoon2-80k.ram.
20. C. Elliott and P. Hudak, "Functional Reactive Animation," *SIGPLAN Notices*, vol. 32, no. 8, 1997, pp. 263–273.
21. S.P. Jones and J. Hughes, *Report on the Programming Language Haskell 98*, 1999; http://haskell.org/haskellwiki/Definition.
22. J. Peterson, P. Hudak, and C. Elliott, "Lambda in Motion: Controlling Robots with Haskell," *Proc. 1st Int'l Workshop Practical Aspects of Declarative Languages* (PADL), Springer, 1999, pp. 91–105.
23. A. Courtney, H. Nilsson, and J. Peterson, "The Yampa Arcade," *Proc. 2003 ACM Sigplan Haskell Workshop* (Haskell 03), ACM Press, 2003, pp. 7–18.

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.

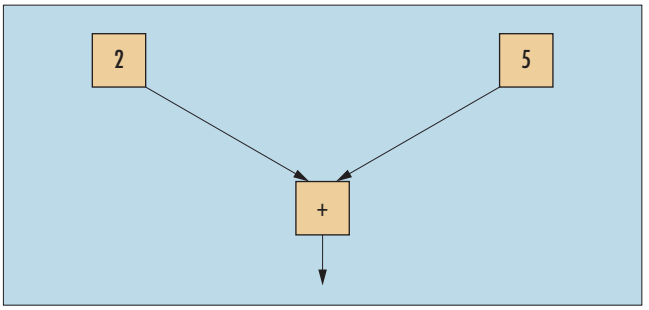

Figure 2. A Proto program is a network of operator instances ascending from a single root. The root's output stream serves as a reference to the program.

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,

```
(sense :light)
```

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

```
(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

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.

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

```
(reduce-nbrs (sense :light) or 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-lag, and infinitesimal, 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:

```
(letfed ((n infinity (+ 1 (if (sense :light) 0
                              (reduce-nbrs (+ n nbr-dist) min infinity)))))
  (– n 1))
```

(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.[7] 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 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). 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

```
(def gradient (src)
  (letfed ((n infinity (+ 1 (if src 0 (reduce-nbrs (+ n nbr-dist) min infinity)))))
    (– n 1)))
```

and a generic averaging operator

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

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

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

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

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,

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

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[8] requires 2,000 lines of nesC[9] 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

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

which adapts to changing sources equivalently to Clement and Nagpal's active gradients.[7]

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

```
(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

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

Coordinates

Another useful example is the coordinate system mechanism from William Butera's paintable computing.[10]

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:

```
(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 to the source. This is fairly fragile, so we instead find the trail geometrically by triangulation against **distance**, then widen it using **dilate**:

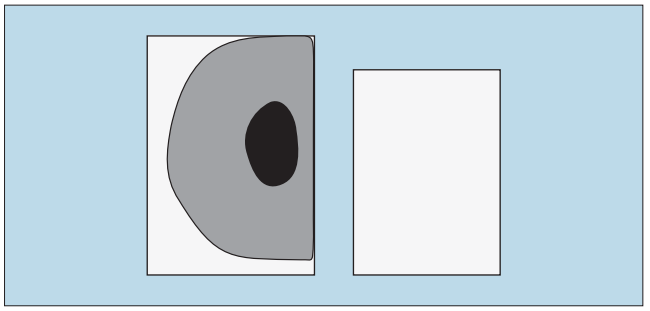

Figure 3. All communication proceeds through neighborhoods, so a gradient (gray) spreading from regions with light (black) that runs only when there's sound (white boxes) can't cross a gap where there is no sound.

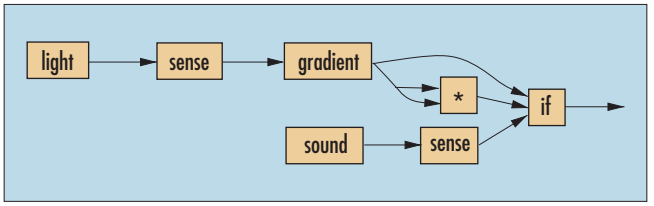

Figure 4. Subprograms might feed multiple inputs. The subprogram caches its output so that it executes only the first time its output stream is pulled in a given round.

```
(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: we use **choose-leader** 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)))
```

We can then define the complicated **coordinates** mechanism (see figure 5) as an operator that, 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)))
```

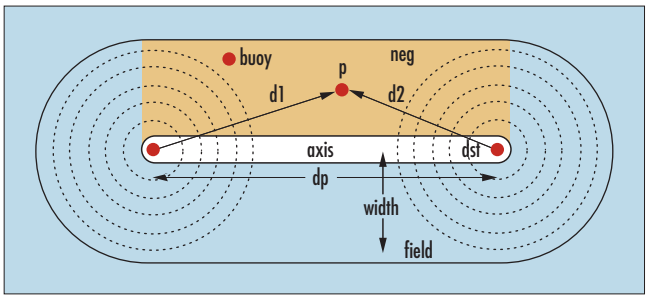

Figure 5. Finding coordinates with a mechanism adapted from paintable computing:[10] The coordinate system's two anchor points send out gradients, producing d1, d2, and dp, which determine the location of p except for the sign of its vertical coordinate. We find the sign by using leader election to break symmetry. Note that *x* values range vertically and *y* values range horizontally.

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 interfere with the heartbeat, the lag gradient floats upward, driving the time locally:

```
(def time-gradient (src)
  (letfed ((n infinity (+ 1 (if src 0 (reduce-nbrs (+ n nbr-lag) min infinity)))))
    (– n 1)))
(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 such as sinusoidal oscillations—useful for locomotion in distributed robotics or moving objects around an active surface.

We could do this with an externally supplied phase coordinate (established, for example, using Butera's algorithm) 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 wave front 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 might 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 a plane wave's src side:

```
(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 wave's period to the vector's length:

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

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 bytecode. 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/neighborhood 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 sensor network through a base station connected by serial cable to the PC. The implementation virally forwards received scripts to neighboring motes, using a mechanism similar to those that James McLurkin,[11] Jonathan Hui and David Culler,[12] and Andrew Sutherland[13] 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 supports 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 operators 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 neighborhood 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 platform-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 actuators API for each mote input and output. The compiled code, including 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 friendlier 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 development, 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.

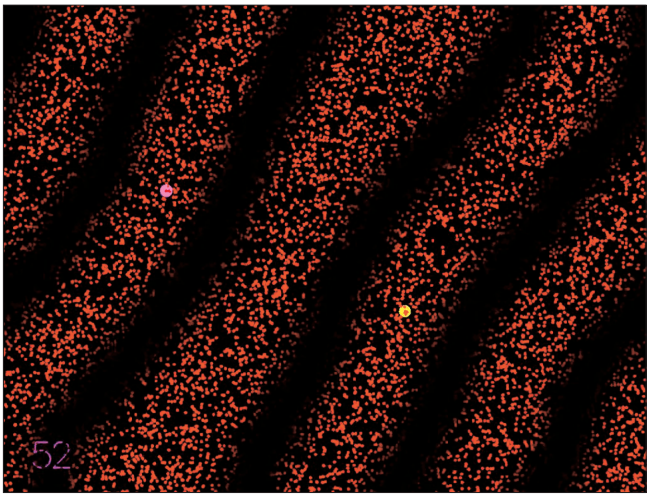

Figure 6. A plane-wave oscillator running on 10,000 simulated devices. The placement of source (yellow) and destination (magenta) markers in the devices' sensor field determines the wave's period and direction.

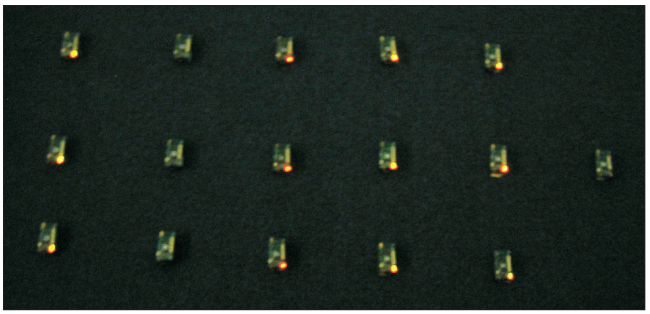

Figure 7. A group of motes running the oscillator program, displaying the output on their LEDs. The motes are given a synthetic coordinate for their phase.

Once a program runs in simulation, we can transfer it to the motes; this provides ground truth as to whether our building blocks compose 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:

```
(def gradient (src)
  (letfed ((n (infinity) (if src 0 (+ 1 (fold-hood min (infinity) n)))))
    n))
(def sync-time (src)
  (let ((lag (gradient src)))
    (letfed ((t 0 (if src (+ 1 t) (fold-hood max 0 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.

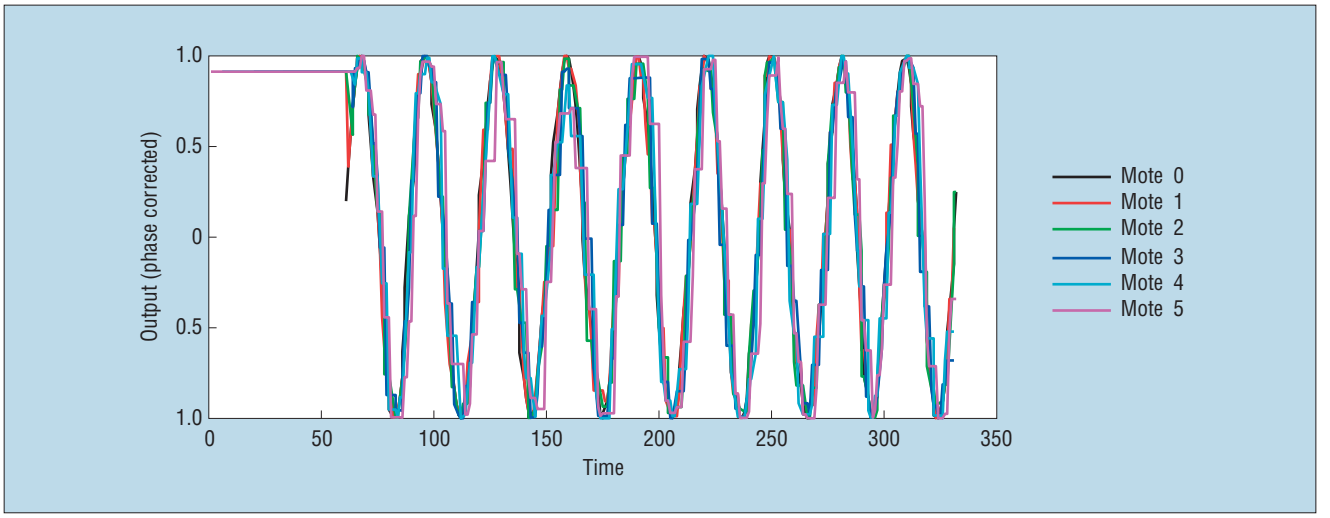

Figure 8. Output of a group of six motes in a line running the oscillator program, subtracting phase. The motes synchronize and begin oscillating shortly after the leader, Mote 0, is turned on. Dropped packets and variable execution rates cause the executions on the various motes to diverge rapidly, while the time synchronization operator continually draws them back toward synchrony.

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-communication 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

1. H. Abelson et al., *Amorphous Computing*, tech. report AIM-1665, Artificial Intelligence Laboratory, Massachusetts Inst. of Technology, 1999.
2. J. Bachrach, "Gooze: A Stream Processing Language," *Proc. Lightweight Languages 2004*, 2004; recorded presentation at http://web.mit.edu/webcast/csail/mit-csail-lwl-04dec2004-afternoon2-80k.ram.
3. J. Beal, "Programming an Amorphous Computational Medium," *Unconventional Programming Paradigms*, LNCS 3566, Springer, 2005, pp. 121–136.
4. J. Beal and G. Sussman, *Biologically Inspired Robust Spatial Programming*, AI Memo 2005-001, Computer Science and Artificial Intelligence Laboratory, Massachusetts Inst. of Technology, 2005.
5. R. Milner et al., *The Definition of Standard ML—Revised*, MIT Press, 1997.

6. S.P. Jones and J. Hughes, *Report on the Programming Language Haskell 98*, 1999; http://haskell.org/haskellwiki/Definition.

7. L. Clement and R. Nagpal, "Self-Assembly and Self-Repairing Topologies," *Proc. Workshop Adaptability in Multi-Agent Systems, RoboCup Australian Open*, 2003; www.ict.csiro.au/staff/Mikhail.Prokopenko/aorc2003_program.htm.

8. A. Eames, "Enabling Path Planning and Threat Avoidance with Wireless Sensor Networks," master's thesis, Dept. Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 2005.

9. D. Gay et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 1–11.

10. W. Butera, "Programming a Paintable Computer," PhD thesis, Media Laboratory, Massachusetts Inst. of Technology, 2002.

11. J. McLurkin, "Stupid Robot Tricks: A Behavior-Based Distributed Algorithm Library for Programming Swarms of Robots," master's thesis, Dept. Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 2004.

12. J.W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. 2nd Int'l Conf. Embedded Networked Sensor Systems*, ACM Press, 2004, pp. 81–94.

13. A. Sutherland, "Towards RSEAM: Resilient Serial Execution on Amorphous Machines," master's thesis, Dept. Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, 2003.

The Authors

Jacob Beal is a PhD student at the Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory. His research interests include amorphous computing and human-like AI. He received his master's in computer science from MIT. Contact him at the MIT Computer Science and Artificial Intelligence Laboratory, Rm. 32-392, 32 Vassar St., Cambridge, MA 02139; jakebeal@mit.edu.

Jonathan Bachrach is a research scientist at the MIT Computer Science and Artificial Intelligence Laboratory. His research interests include robotics, sensor networks, and programming languages. He received his PhD in computer science from the University of Massachusetts at Amherst. Contact him at the MIT Computer Science and Artificial Intelligence Laboratory, Rm. 32-392, 32 Vassar St., Cambridge, MA 02139; jrb@csail.mit.edu.

PURPOSE The IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.

MEMBERSHIP Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

COMPUTER SOCIETY WEB SITE The IEEE Computer Society's Web site, at **www.computer.org**, offers information and samples from the society's publications and conferences, as well as a broad range of information about technical committees, standards, student activities, and more.

BOARD OF GOVERNORS

Term Expiring 2006: *Mark Christensen, Alan Clements, Robert Colwell, Annie Combelles, Ann Q. Gates, Rohit Kapur, Bill N. Schilit*

Term Expiring 2007: *Jean M. Bacon, George V. Cybenko, Antonio Doria, Richard A. Kemmerer, Itaru Mimura, Brian M. O'Connell, Christina M. Schober*

Term Expiring 2008: *Richard H. Eckhouse, James D. Isaak, James W. Moore, Gary McGraw, Robert H. Sloan, Makoto Takizawa, Stephanie M. White*

Next Board Meeting: *16 June 06, San Juan, PR*

IEEE OFFICERS

President : MICHAEL R. LIGHTNER
President-Elect: LEAH H. JAMIESON
Past President: W. CLEON ANDERSON
Executive Director: JEFFRY W. RAYNES
Secretary: J. ROBERTO DE MARCA
Treasurer: JOSEPH V. LILLIE
VP, Educational Activities: MOSHE KAM
VP, Pub. Services & Products: SAIFUR RAHMAN
VP, Regional Activities: PEDRO RAY
President, Standards Assoc: DONALD N. HEIRMAN
VP, Technical Activities: CELIA DESMOND
IEEE Division V Director: OSCAR N. GARCIA
IEEE Division VIII Director: STEPHEN L. DIAMOND
President, IEEE-USA: RALPH W. WYNDRUM, JR.

IEEE computer society
60TH anniversary

COMPUTER SOCIETY OFFICES

Washington Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1992
Phone: +1 202 371 0101
Fax: +1 202 728 9614
E-mail: hq.ofc@computer.org

Los Alamitos Office
10662 Los Vaqueros Cir., PO Box 3014
Los Alamitos, CA 90720-1314
Phone: +1 714 821 8380
E-mail: help@computer.org
Membership and Publication Orders:
Phone: +1 800 272 6657
Fax: +1 714 821 4641
E-mail: help@computer.org

Asia/Pacific Office
Watanabe Building
1-4-2 Minami-Aoyama, Minato-ku
Tokyo 107-0062, Japan
Phone: +81 3 3408 3118
Fax: +81 3 3408 3553
E-mail: tokyo.ofc@computer.org

IEEE

EXECUTIVE COMMITTEE

President:
DEBORAH M. COOPER*
PO Box 8822
Reston, VA 20195
Phone: +1 703 716 1164
Fax: +1 703 716 1159
d.cooper@computer.org
President-Elect: MICHAEL R. WILLIAMS*
Past President: GERALD L. ENGEL*
VP, Conferences and Tutorials:
RANGACHAR KASTURI (1ST VP)*
VP, Standards Activities: SUSAN K. (KATHY) LAND (2ND VP)*
VP, Chapters Activities:
CHRISTINA M. SCHOBER*
VP, Educational Activities: MURALI VARANASI†
VP, Electronic Products and Services:
SOREL REISMAN†
VP, Publications: JON G. ROKNE†
VP, Technical Activities: STEPHANIE M. WHITE*
Secretary: ANN Q. GATES*
Treasurer: STEPHEN B. SEIDMAN†
2006–2007 IEEE Division V Director:
OSCAR N. GARCIA†
2005–2006 IEEE Division VIII Director:
STEPHEN L. DIAMOND†
2006 IEEE Division VIII Director-Elect:
THOMAS W. WILLIAMS†
Computer Editor in Chief: DORIS L. CARVER†
Executive Director: DAVID W. HENNAGE†
* voting member of the Board of Governors
† nonvoting member of the Board of Governors

EXECUTIVE STAFF

Executive Director: DAVID W. HENNAGE
Assoc. Executive Director: ANNE MARIE KELLY
Publisher: ANGELA BURGESS
Associate Publisher: DICK PRICE
Director, Administration: VIOLET S. DOAN
Director, Information Technology & Services:
ROBERT CARE
Director, Business & Product Development:
PETER TURNER

rev. 6 March 06