Neural Comput & Applic (2010) 19:825–847
DOI 10.1007/s00521-010-0382-8

SWARM ROBOTICS

Composable continuous-space programs for robotic swarms

Jonathan Bachrach · Jacob Beal · James McLurkin

Received: 5 June 2009 / Accepted: 5 May 2010 / Published online: 28 May 2010

Abstract Programmability is an increasingly important barrier to the deployment of multi-robot systems, as no prior approach allows routine composition and reuse of general aggregate behaviors. The Proto spatial computing language, however, already provides this sort of aggregate behavior programming for non-mobile systems using an abstraction of the network as a continuous-space-filling device. We extend this abstraction to mobile systems and show that Proto can be applied to multi-robot systems with an actuator that turns a vector field into device motion. Proto programs operate on fields of values over an abstract device called the *amorphous medium* and can be joined together using functional composition. These programs are then automatically transformed for execution by individual devices, producing an approximation of the specified continuous-space behavior. We are thus able to build up a library of simple swarm behaviors, and to compose them together into highly succinct programs that predictably produce the desired complex swarm behaviors, as demonstrated in simulation and on a group of 40 iRobot SwarmBots.

J. Bachrach
Other Lab, 2300 3rd Street, San Francisco,
CA 94107, USA
e-mail: jrb@pobox.com

J. Beal (✉)
BBN Technologies, 10 Moulton Street, Cambridge,
MA 02138, USA
e-mail: jakebeal@bbn.com

J. McLurkin
Rice University, 6100 Main, Houston, TX 77005, USA
e-mail: jmclurkin@rice.edu

Keywords Spatial computing · Amorphous medium · Swarm robotics · Amorphous computing · Multi-robot

1 Introduction

As highly capable robots become increasingly available and less expensive, there is increasing interest in applications in which swarms of robots work together. For example, a swarm of lightweight scout robots might search a disaster area and coordinate with a team of more capable rescue robots that can aid victims, or a swarm of aerial vehicles might team with firefighters to survey and manage wildfires and toxic spills, or a group of autonomous underwater vehicles might survey their environment and autonomously task portions of the swarm to concentrate data gathering on particular interesting phenomena.

A major barrier to developing such applications, however, is the programming of robust aggregate behaviors for the swarm. The dream is that using a high-level language to program a multi-robot application, a programmer would be able to succinctly implement robust group behavior primitives and to quickly compose new programs out of existing primitives and simpler programs. In current practice, however, a programmer typically specifies the behavior of individual robots and attempts to show that their interactions will produce the desired aggregate behavior. The mapping from robot actions to group actions is often complex and difficult to invert, and so multi-robot applications tend to be extremely difficult to create, even harder to extend with new functionality, and often exhibit unexpected fragility and scaling problems.

We observe that when communication is local (e.g., short-range radio), the collection of robots as a whole may be viewed as a *spatial computer*—a collection of

computational devices distributed through a physical space in which the difficulty of moving information between any two devices is strongly dependent on the distance between them, and the "functional goals" of the system are generally defined in terms of the system's spatial structure. The Proto language [7] already addresses these aggregate programming challenges for spatial computers composed of non-mobile devices, using the *amorphous medium* abstraction, which views the network as a continuous-space-filling device [6] Proto programs operate on fields of values over this abstract device and can be joined together using functional composition. These programs are then automatically transformed for execution by individual devices, producing an approximation of the specified continuous-space behavior. As we shall see in Sect. 3.3, we can accomplish this by transforming global programs for local execution on an amorphous medium where each point contains a particular stack-based virtual machine and a discrete kernel approximates execution of these virtual machines on discrete devices.

Although originally designed for static systems, we show that Proto can be applied to multi-robot systems with an actuator that turns a vector field into device motion. We begin by reviewing the amorphous medium abstraction and describing its extension to moving devices. Following a brief review of Proto, we demonstrate how the language can be used to succinctly specify robust, scalable behaviors for a swarm of robots, that these behaviors can be composed together to form complex programs with predictable behaviors and that such complex programs can inherit robustness and scalability from their components. Finally, we verify the applicability of Proto to multi-robot systems by executing some of the behaviors we have developed on a group of 40 iRobot SwarmBots[1] and showing that the quality of the approximation remains within reasonable bounds.

1.1 Related work

There are many domain-specific programming models for spatial computers. Most of these, such as Swarm [30], TinyOS [20], Paintable Computing [12], and CAs [27], involve programming the behavior of the devices, rather than the behavior of the aggregate. An early exception is CMost, the operating system for the CM-5 [34], which allows operations on distributed fields of values, but assumes a fixed population of devices arranged in a grid.

Although in most multi-robot systems the programmers work at the level of individual robots, there have been a number of approaches to aggregate programming. Mataric [28] introduced the notion of basis behaviors and group computing, but the basis behaviors are challenging to combine. Works by Klavins [22] and Kloetzer and Belta [24] have promoted the idea of high-level descriptors for swarm flocking and the ability to compile out rules. The high level is more akin to what we will show, but these systems mainly produce motion control laws on more capable robots—with GPS, and global clocks—where interactions are less critical in determining robot behavior. Furthermore, the languages are focused on motion and do not provide very expressive means for distributed sensing and distributed state, which one might desire for a robot swarm application. More recently, Meld [1] and LDP [37] have taken a logic programming approach to shape formation in modular robotic ensembles, which LDP seeks to extend to more general distributed programming. These languages offer no means of abstraction, and the logical programming model means that composing together programs can produce hard to predict effects.

Much of Proto's previous application has been in the area of sensor networks. Other sensor network languages focus almost exclusively on data collection, often abstracting away the spatial nature of the network in favor of well-established tools or theoretical frameworks. For example, the Regiment [32] programming language operates on geometric regions of space, but is targeted toward sensor network data gathering and only distributes some operations across space, while its successor WaveScript [33] drops the spatial operations in favor of a tighter stream semantics. Others are farther still: TinyDB [26] allows the user to interact with the network as though it were a database, and Kairos [19] focuses on the manipulation of abstract graphs.

The structure of Proto as a dynamic network of streams is strongly influenced by previous work on Gooze [2], as are many of the compilation strategies used to compact Proto code for execution on embedded systems. More generally, there is a long tradition of stream processing in programming languages. The closest and most recent work is Functional Reactive Programming (FRP) [17] that is based on Haskell [21], which is a statically typed programming language with lazy evaluation semantics. In these systems, less attention is spent on run-time space and time efficiency, and the type system is firmly wedded to Haskell, with all of its strengths and weaknesses.

2 Duality of swarm and space

The *amorphous medium* abstraction [6] is derived from the observation that in many spatial computing applications, we are interested not in the particular devices that make up our network, but rather in the space through which they are distributed. The point of a sensor network, for example, is generally the environmental values that it senses. If more

[1] SwarmBots [29] are not to be confused with Swarm-bots [31], a similarly named robotic platform.

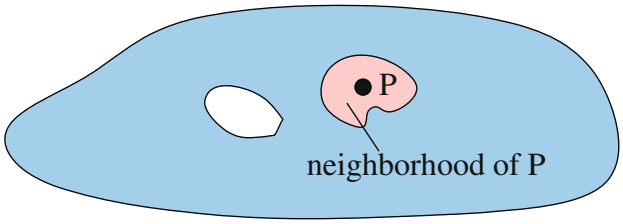


Fig. 1 An amorphous medium is a manifold where every point is a device that knows its neighbors' recent past state

sensors are available, the area of interest can be inspected at a higher resolution, but the essential task remains the same.

The amorphous medium abstraction takes this to its logical extreme: an amorphous medium is a manifold M with a computational device at every point (Fig. 1). Adding in the dimension of time, we may also consider the foliated manifold $M \times T$, where each point describes the state of a device $m \in M$ at time $t \in T$. Information propagates through this medium at a maximum velocity c. Each device is associated with a neighborhood of nearby devices and knows the state of every device in its neighborhood intersected with its past light cone (i.e., the most recent information that can have arrived from its neighbors). Given a program and the state across any space-like surface of $M \times T$ (i.e., a continuous surface in which every point in M appears precisely once and no two points have access to one another's past state), we may thus compute all future state (and past state as well if the computation is reversible).

2.1 Approximation by discrete devices

While an amorphous medium cannot, of course, be constructed, any actual spatial computer can be viewed as a discrete approximation of an amorphous medium for the space that it fills. If programs are formulated with continuous units of measure, such as meters and seconds, and an appropriate conversion is made between continuous and discrete units, then a continuous-space program can be executed approximately on the discrete network, and it is possible to predict the quality of the discrete approximation of the continuous program—see, for example [5] and [9].

There is a duality in this relationship between a space of interest and a network of devices. On the one hand, we may consider a space, and ask how well it is approximated by a given network of devices as shown in Fig. 2. On the other hand, we may consider a network of devices and ask what space is best approximated by this network.

It is an open question what is the best way of determining the quality of approximation, though it is certain that there will always be calculations that can occur on a manifold that cannot be well approximated on a finite set of devices. More than likely, the ultimate answer depends on the communication model and the particular application under consideration. For example, devices using infrared LEDs to communicate will be blocked from communicating by obstacles that would not block broadcast radio communication. Devices using infrared would thus need more devices or more carefully placed devices in order to approximate the connectivity of a building with a complex floor plan, but once connected would have edges that better approximate the geometry because they do not go through walls.

In order to have some basis to build from, we consider a unit disk model of connectivity, in which each device is connected to all others within r meters. While this model is a vast oversimplification, it is still a reasonable generic starting point for describing spatial computers, since the most important property of the communication model is strong locality.

Given this model, we may estimate whether a homogeneous distribution approximates space well by looking at the expected number of neighbors with which each device communicates. For example, Kleinrock and Silvester [23] show that at two or more expected neighbors, reasonable forward progress can be expected in each hop through the network; for distributed distance measures, a good threshold is 10 expected neighbors [5]. Assuming that the expected number of neighbors is sufficient, the responsibility of devices for approximating portions of the space can be determined using a Voronoi decomposition.

Going in the other direction, a conservative approach to determine a space that is well approximated by a discrete

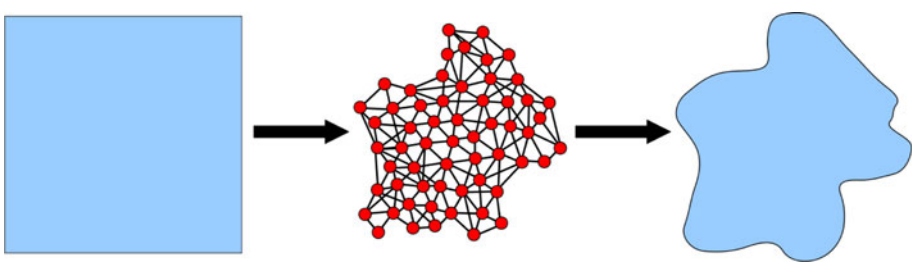
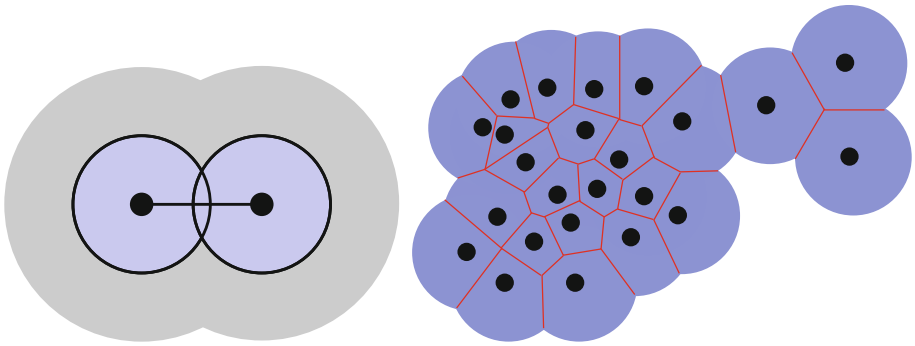


Fig. 2 Duality of space and network: we may consider either how well a space of interest is approximated by a given network, or what space the network best approximates. For example, the square space on the *left* is poorly approximated by the network at *center*, but the network approximates well the oddly shaped space on the *right*



(a) **(b)**

Fig. 3 A conservative approach to determining a space that is well approximated by a discrete network (assuming unit disk communication) is to surround each device with a half-communication radius disk (**a**). The union of all such disks is a space well approximated by the network (**b**), and a Voronoi decomposition of the space (*red lines*) can be used to determine which portions of the space are to be approximated by which discrete devices

network is to surround each device with a disk whose radius is $\frac{1}{2}r$ (Fig. 3). The union of all such disks is a space well approximated by the network: there is a straight-line path connecting any pair of devices that can communicate directly, and when two devices are in disconnected components of the network, there is no path connecting them through the space.

2.2 Extending the amorphous medium to moving devices

At a surface level, extending the amorphous medium abstraction to moving devices is straightforward: merely add a function for moving devices to the language. Many swarm robotic applications are also focused not on the robots, but on the space through which they move. The focus of mapping is the environment being mapped. For search and rescue, it is the area being explored and the victims that may be discovered therein. Network coverage is about ensuring adequate signal-strength across the largest possible useful area. Just as in the static case, at any given instant in time, we may consider a space of interest (e.g., a desired formation of robots) and ask how well the swarm currently approximates it, or use a mapping like the half-disk mapping above to ask what space is currently well approximated by the swarm.

When the devices are capable of moving, we must pay more careful attention to the relationship between the network and the space it occupies. Moreover, because the goal of many swarm robotic programs is precisely to arrange the robots in some environment-dependent distribution, it is no longer reasonable to assume that devices are distributed homogeneously through some subspace. In fact, the whole point of a program may be to distribute devices inhomogeneously: for example, consider a network of mobile sensors, where we wish to have sensors distributed at low density throughout the entire environment, but have a portion of them cluster densely in regions where something interesting has been detected in order to provide high resolution.

We thus need to extend the amorphous medium abstraction to include some notion of the *density* with which devices are distributed in space. The continuous-space abstraction is obvious: the manifold of the amorphous medium shall be considered to consist of matter, and the density of this abstract matter is approximated by the proximity of robots to one another.

In a robotic swarm, robots can move closer together and farther apart while still being able to communicate reliably, so the matter of the amorphous medium is compressible. We shall thus consider it to be in a gaseous state. If the robots pack together closely enough that it is difficult for any robot to make significant progress without evading or being evaded by other robots, then the matter of the amorphous medium is no longer significantly compressible, and mass flow requires the development of vorticity (in the form of mutual evasion). We shall thus consider such an amorphous medium to be in a liquid state. In modular robotics, on the other hand, the robots are packed together at an approximately fixed density (though there may be voids in the structure). If they can move at all, it is often the case that the moving robots must be a relatively small percentage relative to a substrate of non-moving robots (see [36] for an elegant example of this). In this case, we shall consider the amorphous medium to be in a solid state.

In this paper, we consider only robots in the gaseous state. Using the unit disk model of communication (with communication radius r) and the half-disk mapping for deriving an amorphous medium from a distribution of robots, let the area of a robot's Voronoi cell in the manifold be V. The density of the amorphous medium within a cell

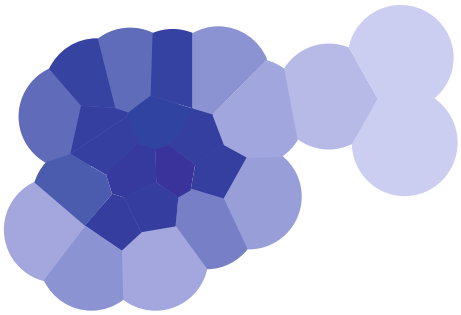


Fig. 4 For mobile devices, we extend the amorphous medium to include *density*. One reasonable approximation of the density at a point may be found by taking the area of a half-radius disk and dividing it by the area of a Voronoi cell. Here, the space from Fig. 3 is shown with higher density indicated by *darker colors*

may then be approximated by $\rho = \frac{1/4\pi r^2}{V}$ (Fig. 4). Mass flow in such an amorphous medium may then be calculated as a vector field over the manifold and approximated by moving each robot according to the vector at its location.

The normalization of the density by half-disk area makes this a scalable measure: for a given expected number of neighbors, the expected density remains constant. The density at which the amorphous medium transitions between gaseous and liquid behavior, however, is of course platform dependent, as it is set by the ratio between the size of a robot and the range of its communication.

Note, however, that the quality of approximation of mass flow under this mapping is an open question, and we make no assertions on that front. Note also that it is not immediately obvious whether adding a time dimension still produces a foliated manifold, given that the connectedness of the swarm may change over time, so the appropriate mathematical formalism for time evolution in this model is also an open question. Empirically, however, we shall see that using this model of approximation produces good results, which leads us to believe that appropriate formal answers to these questions can be found.

3 Review of Proto

The Proto language [7] uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled sub-problems, as shown in Fig. 5: global descriptions of programs as functional operations on fields of values, compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network. In this section, we briefly review the Proto language and the enabling infrastructure that allows it to approximate global continuous programs in terms of discrete local interactions. For full details, see [8] and [4]. Proto may be obtained online at http://www.stpg.csail.mit.edu/proto.html.

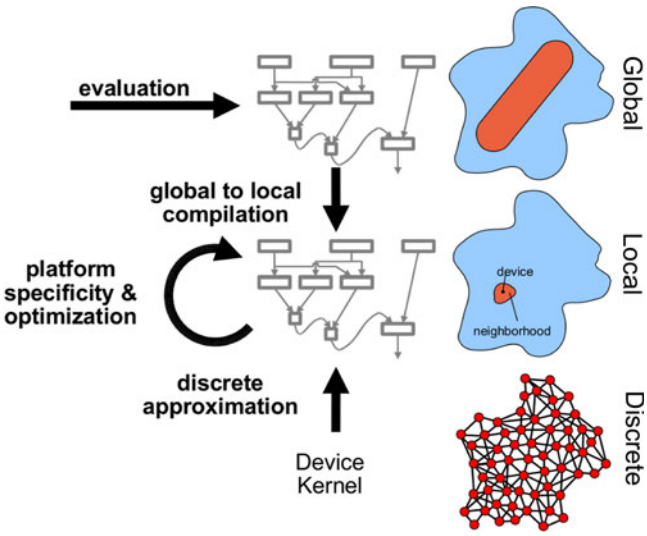
3.1 Fields and functional composition

Proto is a functional language. Its primitive elements are mathematical operations on fields, where a field is a function that maps every point in space to a value. These elements are composed using the rules of mathematical function composition. A Proto program may thus by interpreted as a dataflow graph of operations on fields, such as that shown in Fig. 6. A program is then evaluated against a manifold to produce a field whose values evolve over time.

For example, the expression `4` is an operator of no arguments that produces a field mapping every point in the amorphous medium to the scalar value 4. The operator `sqrt` takes a scalar field and produces a scalar field where every point that mapped to n in the input field maps to $\sqrt{n}$ in

Fig. 5 The Proto language uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled sub-problems: global descriptions of programs as functional operations on fields of values, compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network

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



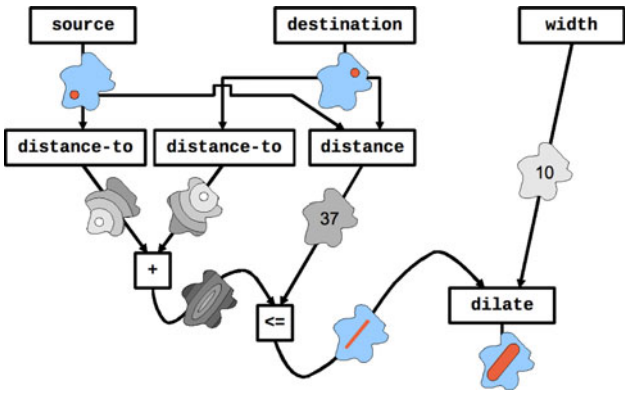


Fig. 6 A Proto program—here creating a channel between two regions—specifies a dataflow graph of operations on fields. The program is shown evaluated on an irregularly shaped space, with scalar fields *gray* (lighter is less) and *Boolean colored* (`true` is *red*). The inputs are Boolean source and destination regions and the output is a channel created by dilating a minimal length path created between these inputs. The path is computed using the triangle inequality to be a Boolean region true where the sum of minimal distances at a given point to the source and destination region, respectively, is equal to the minimal distance between the source and destination regions

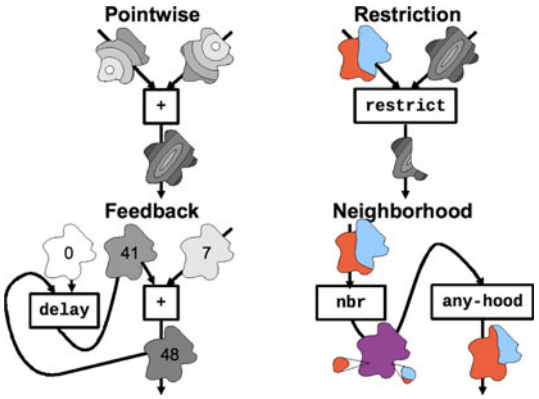


Fig. 7 Proto uses four families of operations: pointwise operations that involve neither space nor time, restriction operations that limit execution to a subspace, feedback operations that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state and space–time measures, then summarize the values computed over neighborhoods with set operations. Feedback operations are composed of a feedback graph producing outputs from inputs and previous outputs and a delay operator producing previous outputs taking an initial value and current output as inputs. The neighborhood operations are composed of a `nbr` operator which produces fields of fields of its input (i.e., a field for each neighbor) and a summary operator (e.g., `any-hood`) which reduces that field of fields into a single output field

the output field. Proto uses LISP-style parenthetical notation to show composition (though the reader should remain aware that Proto is not a LISP). For example, `(sqrt 4)` applies the `sqrt` operator to a field mapping every point to 4, producing a field that maps every point to 2.

The instantaneous evaluation of an expression—its evaluation at a given point in time—is simply its application to an amorphous medium, producing a set of fields. Figure 6 shows an example of a complex computation of this sort, from [3], finding a redundant path between source and destination regions. To find out what values have actually been computed, one evaluates the field at particular points.

To evaluate an expression across time, we perform a sequence of instantaneous evaluations. Each instantaneous evaluation depends only on the previous evaluation (via the `delay` operator—see Sect. 3.2). Typically, these occur at fixed intervals, but Proto uses a continuous time semantics (described in [4]) that allows devices to update asynchronously at a variable rate.

3.2 Four families of primitives

Proto uses four families of operations (Fig. 7): pointwise operations like + that involve neither space nor time, restriction operations that limit execution to a subspace, feedback operations that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state and space–time measures, then summarize the values computed over neighborhoods with set operations like integral or minimum.

3.2.1 *Pointwise*

Any purely functional operation on a normal computer can be a pointwise operation on an amorphous medium: the amorphous medium version simply applies the operation uniformly to every point in the amorphous medium. For example, constants like `4` and mathematical operators like + and `sqrt` are pointwise operations. Other examples include constructing tuples with the `tup` operation and accessing their elements, sensors and actuators, and `mux`—a conditional "multiplexer" operator that uses a truth value to select between the outputs of two branches running in parallel.

3.2.2 *Restriction*

Conditional code needs to be thought of differently when programming an aggregate rather than a single device, since in general different devices may need to take different branches. Proto supports this with the operation `restrict`, which limits the domain of fields. This functionality is made accessible to the programmer through the syntactic operator `if`, which executes each branch in a domain `restrict`ed by the Boolean test field. The outputs of the two branches are then combined back together with a `mux` operator that selects input piecewise using the same test value that split the branches. The result is conditional code that obeys the same intuitions as an `if` on a single device.

3.2.3 State

In Proto, program state is handled using feedback loops built around a `delay` operation for time-shifting values and a `dt` operation that tracks the elapsed time since the last update. The `delay` operation takes two fields, one whose values are to be delayed and one with initial values. The value of the field it produces depends on the domain of the delayed field at the previous instantaneous evaluation: any point in the domain at both the current and previous evaluations receives the value of the delayed field at the previous evaluation; all others receive the initial values. Note that this implies that in the first instantaneous evaluation of a program, every point receives the initial values, and also that this interacts with the domain restriction of `if` to reset state in the branch not taken. Expressing state using delay rather than mutation semantics is somewhat more restricted, but makes both composition and the compiler's transformation from global to local simpler.

A programmer does not typically use `delay` directly, but instead the feedback loop constructs `letfed` and `rep`. The `letfed` construct is a syntactic operator with the signature `(letfed vars . body)`, where `vars` is a list of feedback variable declarations `(var init evolve)`. The `evolve` expression calculates the current value of `var`, using a delayed value (initialized with the corresponding `init`) for any feedback variable in the expression. The syntactic form `(rep var init evolve)` captures a common `letfed` pattern: it is equivalent to `(letfed ((var init evolve)) var)`.

Thus, for example, a timer can be implemented with:

```
(rep t 0(+ t (dt)))
```

which computes an elapsed time `t` that starts at zero and rises by `dt` at each instantaneous evaluation. In a more complicated example, a stoplight finite state machine can be implemented with:

```
(def stoplight ()
  (letfed (((tup l t)
    (tup 'stop 30)
    (if ( > t 0) (tup l (- t (dt)))
      (case l
        (go      (tup 'stopping (+ 3 t)))
        (stop    (tup 'go (+ 27 t)))
        (stopping (tup 'stop (+ 30 t)))))))
    (green (= l 'go))
    (red (= l 'stop))
    (yellow (= l 'stopping))))
```

The system state is a tuple of the current light and a timer, which is translated into an actuation of lights in the body of the `letfed`. As time advances, the timer counts down toward zero. Once it hits zero, the state advances (changing the light) and the timer resets. The only difference between this and a more conventional finite state machine implementation is that the length of each step of the machine is not assumed.

3.2.4 Neighborhood

Neighborhood operations express the flow of information through a spatial computer. As such, the set of neighborhood operations in a program completely specify the communication that will take place between devices when that program is run on a spatial computer: if an operation uses a neighborhood value, then in the local version each device proactively transmits that value to its neighbors (or a null value when excluded from the domain by a `restrict` operation).

Neighborhood operations follow a fixed pattern. First, values are gathered from the neighborhood to produce a field where the value of each point is a field mapping neighbors to values. These are gathered using the `nbr` operator `(nbr v)`, which selects neighborhood values to produce a field where the value at each point is a field mapping neighbors to their values of `v`, and also with a set of space–time measurements such as `nbr-range` and `nbr-lag`, which collect local information about the structure of the spatial computer. These operations imply communication, and for simplicity are not allowed to be nested.

Computations may then be performed on fields of neighborhood values with polymorphic pointwise operations that have been "pushed down" to operate on the elements of a neighborhood. For example, when + is applied to fields of neighborhood values, it produces a field where the value of each point is a field that maps points in its neighborhood to the sum of the values they mapped to in the input fields.

Finally, the values of each neighborhood are summarized back to a single value. These summaries are typically set operations, such as `min-hood`, which produces a field whose value at each point is the infimum of the neighborhood values for that point in the input field, and `int-hood`, where each point maps to the integral of the neighborhood values.

Long-distance communication is then produced by combining feedback and neighborhood operations, with the neighborhood operation moving information a short distance and feedback chaining across neighborhoods. For example, we can estimate the distance from every point to the nearest source point by chaining the triangle inequality over neighborhoods:

```
(def distance-to (source)
  (rep d (inf)
    (mux source 0
      (min-hood (+ (nbr d) (nbr-range))))))
```

In this program (sometimes termed a *gradient* for historical reasons), `d` is a feedback variable containing the estimated distance to the source and `source` is a Boolean field where source points map to `true`. The distance estimate is set to zero at the source and elsewhere is computed with the triangle inequality, using `min-hood` to minimize over the sum of the distance from each neighbor to the source and the distance to that neighbor.

With these four families of operators, the amorphous medium abstraction provides a natural way to express spatial programs. For example, `distance-to` is defined over continuous spaces, in units of meters rather than hops, and can thus be used directly to build up higher-level geometric constructs such as bisectors and region dilation. Furthermore, because these are defined on with respect to a manifold, rather than a Euclidean subspace, a change in the structure of the network automatically results in a matching change in the geometric construct being produced. For example, because the `distance-to` measures meters through the space occupied by devices, changing the device distribution may change which devices form a bisector in that space, but the bisector will still be found correctly, no matter how warped the space becomes.

3.3 From global to local

With this choice of appropriate space–time operators, compilation and discrete approximation are straightforward. Our implementation of Proto accomplishes this with a rendezvous at the amorphous medium abstraction: the compiler transforms global programs for local execution on an amorphous medium where each point contains a particular stack-based virtual machine, and a *discrete kernel* approximates execution of these virtual machines on discrete devices.

3.3.1 Compilation and execution

The design of the virtual machine was strongly shaped by a major early challenge in the development of Proto: fitting an executing implementation into the approximately two kilobytes of RAM available on a Mica2 Mote. The virtual machine is thus a simple stack machine that executes scripts of 1-byte instructions, manipulating data tagged as one of several types—scalars, tuples, functions, and dead values. Execution happens in regular rounds, with one invocation of the script per round.

The compiler translates Proto expressions into dataflow graphs. As it transforms code to graph, the compiler infers data types, then simplifies the graph by inlining, folding constant sub-expressions, and removing empty structures. The compiler then walks the graph, translating it into a script that computes a single round of execution. In the process, `restrict` instructions become branch code, `delay` instructions have state storage allocated, and neighborhood operators export values.

Besides normal operating system responsibilities, such as interfacing with sensors and actuators and scheduling regular execution of the script, the discrete kernel provides two special functions: approximation of neighborhood operations and distribution of programs. Neighborhood operations are supported by a neighborhood communication module that broadcasts exported values halfway between executions and maintains a best-effort table (similar to that in [12] and others) containing a device's neighbors, information they have transmitted to be consumed by `nbr` operations, and space–time measurements. Meanwhile, receipt of packets proceeds in the background. Neighborhood operations are then approximated by walking through the table, calculating from the information it contains and combining it incrementally into an approximate summary value.

Distribution of programs is a service provided so that a user can load a program into the whole computer by touching only a single device. We implement this capability with a viral programming mechanism similar to those in [29] and [25]. Compiled programs are versioned and broken up into packets for transmission. Each device then broadcasts script digests stating which script packets it has and sends script packets when a neighbor needs a packet it has. The compiled program also carries with it all of the necessary information for allocating memory and starting the computation running. This process is expensive in communication and power consumption, but occurs infrequently.

When a script is loaded onto a device, the link/load code is executed to allocate memory, initialize the program, and return a pointer to the start of the script; when a script is unloaded, exit code releases the memory. Memory is allocated into many different sections (See Fig. 8) with the majority generally dedicated to the neighborhood. At present, all of these are kept in RAM but the script sections could be moved to slower storage. The size of each section of memory is determined statically by the compiler, meaning that the compiler can also determine whether a given platform will be able to execute a particular program.

Within the neighborhood memory section, structures encode neighbors' ID, timeout, position, timestamp, and data values. Because of memory considerations, a limited number of neighbor structures are allowed. This number is

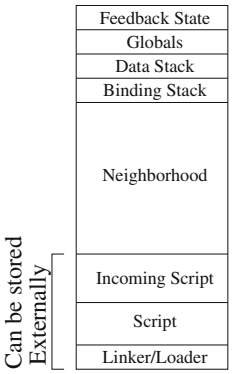


Fig. 8 Memory is allocated into many different sections, with most going to the neighborhood. The script sections could be moved to external storage to save memory

fixed for each platform, so that the behavior of code is not affected by the context in which it is called. Optimal handling of excess neighbors is a topic for future research. At present, neighbors are taken on a first come first serve basis and discarded if they have not been heard from within a timeout period.

The neighborhood mechanism also regulates power by an exponential backoff in frequency of transmission when data is not changing. The gradual backoff makes it less likely that dropped packets will cause a neighbor to mistakenly drop a slowly transmitting device. Since broadcast is a significant fraction of power consumption, this backoff can save large amounts of power during periods of stability.

3.3.2 *Cross-platform portability*

Portability between platforms comes primarily from the approximation of continuous-space operations on the virtual machine. The rest comes from a thin hardware abstraction layer including a device interface provides sensing, actuation, communication, and geometry. This allows the same code to run on different platforms and makes it easier to test code in simulation.

Sensors and actuators are implemented as instructions that call a platform-specific handler for the device, with default results provided for unavailable devices. The handler for actuators must resolve conflicts for multiple actuations during a single round; at present, we have only used lowest-and-rightmost precedence, though any method is acceptable as long as it is consistent across platforms.

The device interface also allows debugging support through virtual sensors and actuators: at present, there is a probe device that exposes values, a peek/poke interface that manipulates sensor and geometry information, and an interface for breakpoints and communication tracing.

Neighborhood support depends on communication and geometry. Communication simply provides the local broadcast interface needed for neighborhood support. Geometric information, in the form of relative coordinates, area, time lag, etc. is derived from whatever localization hardware the platform provides. This can be as crude as an estimate of approximate communication range or as sophisticated as precise range-finding and global coordinates.

At present, Proto's discrete kernel has been implemented in simulation and on four hardware platforms (Fig. 9):

3.3.2.1 Simulation The Proto simulator provides interactive development and rich visualization and debugging support. This platform simulates swarm robots and modular robots using the Open Dynamics Engine[2] (ODE) implementing robots with 3D geometric extent, collision detection, and motor dynamics or optionally a home-spun particle physics engine with point robots, wall collision detection, and motor dynamics. This platform has the most resources: a general purpose CPU that time shares Proto virtual machines on top of Linux, Mac OS X, or Windows. The low-level machine module interfaces to the robot simulator to drive the motors and sense the environment. The blackboard system is implemented as call outs and call backs on top of the simulator efficiently maintaining actual neighborhood and geometric relationships. Neighborhoods are defined using a communication radius and communication itself is defined to always succeed. The ODE simulator has proved to be a good approximation to real robotic system and allows a large number of issues to be worked out before deployment.

3.3.2.2 Mica2 Motes Proto runs on Mica2 Motes [20] on top of TinyOS. The Mica2 has an 16 MHz 8-bit processor, 4 K of RAM, and 128 K of flash memory. The hardware abstraction layer uses TinyOS's radio communication stack for transceiving data and script in 32-byte packets at 19.2 kbps, and its timer, sensor, and actuation modules for implementing the rest of the hardware layer. In this implementation, we assumed the presence of a global positioning service and send global coordinates to neighbors who then calculate relative coordinates.

3.3.2.3 Topobo Topobo [35] is a modular robotics platform where devices are physically connected using passive linkage components connected from one motor to a join point on another device. This platform has the least resources: a 16 MHz 8-bit processor, 2-K RAM, 32K of flash memory, four 9,600-kbps wired connections to its neighbors, and a user interface of a button and two LEDs on each device. There is no native Topobo operating system, so the Proto implementation includes low-level modules to drive the motor, run the user interface, and

[2] http://www.ode.org.

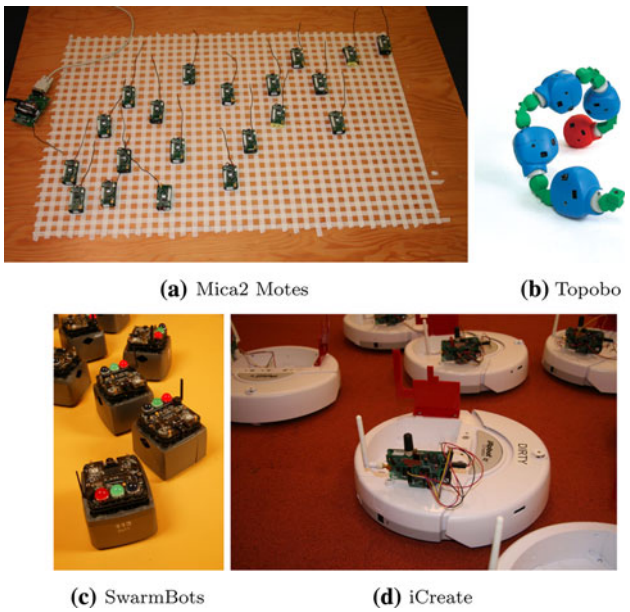


Fig. 9 Proto's discrete kernel has been implemented on hardware for sensor networks (**a**), modular robotics (**b**), and swarm robotics (**c**) and (**d**). (Photo credit: **b** Hayes Raffle and Amanda Parkes and **c** James McLurkin and Swaine Photography)

communicate packets. Geometric information is available only through connectivity.

3.3.2.4 SwarmBots Proto runs on McLurkin and iRobot's swarm robots [29] on top of their embedded threaded operating system. Resources are plentiful: 40 MHz 32-bit processor, 648 K of RAM, 3 MB of flash memory, and 125-Kbps infrared communication that also provides ranging and direction information. The operating system provides its own incompatible neighborhood system, which we use as a blackboard system to simulate local broadcast.

3.3.2.5 iCreate Proto runs on an Atheros radio-on-a-chip MIPS platform running OpenWRT[3] Linux providing the brains and networking for an iRobot iCreate robot [13]. Proto itself runs on top of the OpenWRT Linux operating system using a single thread. Resources are relatively plentiful: 180 MHz 32-bit MIPS processor, 32 MB of RAM, 8MB of flash memory, and WIFI communication that also provides crude ranging information. We implement the Proto blackboard system on top of low-level WIFI using promiscuous broadcast mode which permits efficient communication and neighborhood detection. The iCreate differential wheel robotic platform is controlled and sensed through a serial interface on the Atheros board at intervals of 50 ms. The iCreate provides coarse odometry, bumpers, and cliff sensors.

[3] http://www.openwrt.org.

4 Building up complex swarm behaviors

With Proto's four families of space–time primitives and automatic global-to-local translation, all based on the amorphous medium abstraction, it is straightforward to build complex swarm behaviors. In this section, we illustrate how the synthesis of these principles in Proto enables scalable and robust behavior, differentiation of swarm behavior with respect to space and time, and the incremental construction of complex swarm behaviors by composing together simpler components. As we do so, we shall build up a search-and-rescue demonstration in simulation as a composition of 8 simpler programs, using a total of 56 lines of Proto code.

4.1 Motion from vector fields

The basic idea of how Proto can be used to control robotic swarms has already been hinted at in Sect. 2.2: we compute a vector field over the amorphous medium approximated by the current distribution of robots, then pass that vector field to a new actuator, `mov`, which interprets its input as a specification for desired mass flow in the amorphous medium. This is approximated discretely simply by having each robot attempt to move in the direction and speed specified by the vector at its location.

For example, a random vector field may be created by the Proto expression:

```
(def rand-vec
  (tup (rnd -1 1) (rnd -1 1) (rnd -1 1)))
```

where the `rnd` operators create fields of random scalar values[4] within the specified bounds, and the `tup` operator combines them to produce a field of random vectors (Fig. 10a). Thus far, this is only a field, and we have not yet produced any motion. We can program a swarm of robots to move according to this vector field by using `mov` as follows:

```
(mov (rand-vec))
```

This produces random movement of robots not unlike Brownian motion, though actual Brownian motion requires somewhat more code to ensure isotropy and proper scaling.

Many existing algorithms for the control of swarms can be translated into simple Proto programs. For example, we can implement a flocking algorithm, adapted from [15] and [14], as follows:

[4] See [8] for more on the subtleties of random values on an amorphous medium.

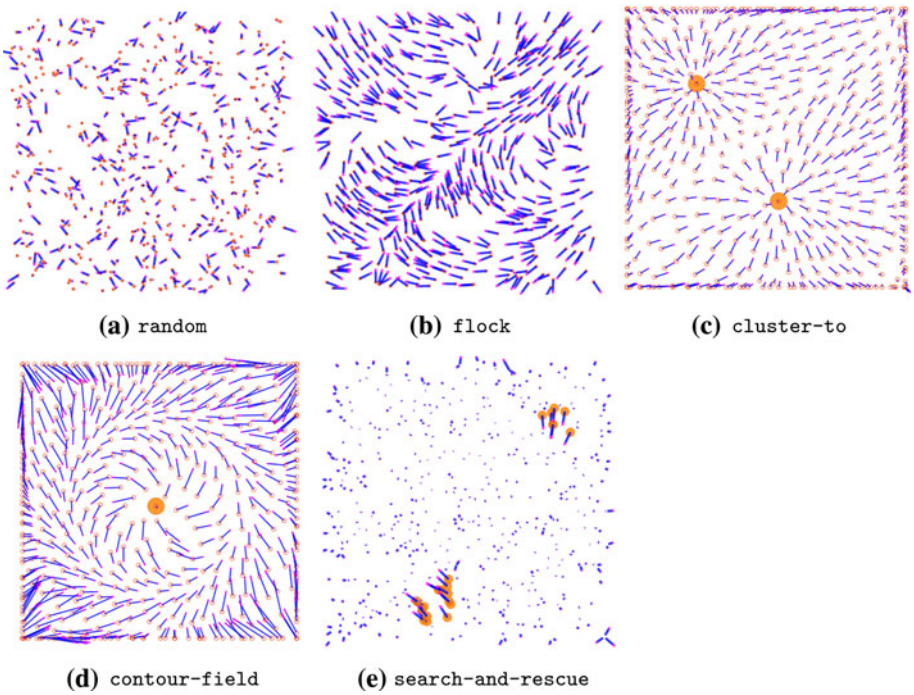


Fig. 10 Swarm robotic programs in Proto calculate a vector field over the space occupied by the robots, specifying what motion a robot at any point should take. Shown above are programs calculating vector fields on a swarm of 500 robots for behaviors of random motion (**a**), flocking (**b**), clustering to a source (**c**), orbiting around a source at a fixed distance (**d**), and search and rescue (**e**)

```
(def flock (dir)
  (rep v
  (tup 0 0 0)
  (let ((d (normalize
    (int − hood
      (if (and ( > (nbr − range) 0)( < (nbr − range) (∗ (comm − range)0.333)))
        (∗ − 1(normalize (nbr − vec))); repel nearby mass
        (if ( > (nbr − range) (∗ (radio − range)0.666))
          (∗ 0.2 (normalize (nbr − vec))) ; attract distant mass
          (normalize (nbr v))))))))      ; between, align vectors
  (normalize (+ dir d)))))           ; mix with preferred dir
```

In this program, nearby matter repels, distant matter attracts, and matter at an intermediate range attempts to align vectors, creating a vector field that tends to produce an even distribution of mass moving all in the same direction (Fig. 10b). This is mixed together with another field, `dir`, that specifies preference about the direction of motion, allowing a small percentage of informed individuals to guide the whole flock, as per [15].

More complicated vector fields can be formed by composing Proto programs together in the calculation, such as motion toward a set of targets (Fig. 10c) or orbiting a region at a fixed distance (Fig. 10d). We can also combine different vector fields in various ways, such as differentiating by region to specialize a group of robots into searchers and teams of rescuers (Fig. 10e).

As a swarm computes desired mass flow in an amorphous medium, the robots move to approximate that flow, reshaping the space. For example, Fig. 11 shows 500 point robots moving under the guidance of `flock`. As noted in Sect. 2.2, there are open questions about quality of

approximation and how best to model the relationship between space and robots. In practice, however, we shall see that adopting this vector-field approach to computing swarm behavior makes it fairly straightforward to predictably produce robust and scalable complex swarm behaviors by combining simpler behaviors.

```
(def distance-to (source)
  (1st
    (rep (tup d v)          ; d = distance estimate, v = value rising
      (tup (inf) 0)          ; initial value
    (muxsource
      (tup 0 0)             ; source is always distance zero
      (mux (max-hood+       ; test whether to apply constraint
        (<= (+ (nbr d) (nbr-range)(* v(+ (nbr-lag) (dt)))) d))
       (tup (min-hood+ (+ (nbr d) (nbr-range))) 0) ; apply triangle inequality
       (let ((v0 (/(comm-range) (* (dt) 12))))      ; rise rate based on info speed
        (tup (+ d (* v0 (dt))) v0)))))))
```

4.2 Scalability and robustness

Using a continuous-space abstraction aids in the construction of scalable and robust programs using an appropriate choice of measurement units to separate the specification of a program from its instantiation on a particular set of devices. Programs written in Proto are, abstractly, executing on an infinite number of devices, so different numbers and arrangements of devices may simply be viewed as different discrete approximations. Thus, we may expect that program that respects the abstraction and runs correctly on one hundred devices should run just as easily on one thousand devices, one million devices, or even greater numbers. This can, of course, break down if the scaling of the swarm means that devices are unable to maintain the amorphous medium abstraction—for example, if the number of neighbors of a device is so high that its communication facilities become saturated.

Typically, a program is expressed in physical units such as meters and seconds. There are also parameters whose values are expressed in these units, but which vary depending on the nature of the spatial computer on which the program is being executed. For example, the `flock` program expresses interaction with neighbors in terms of fractions of the expected communication range.

A prime example of a program that takes advantage of the geometric abstraction to achieve scalability and robustness is a self-healing version of the distributed distance estimation algorithm from Sect. 3.2:

This program, implementing the `CRF-Gradient` algorithm from [10], uses the triangle inequality to constrain distance estimates downward to the minimum path from each point to the nearest location in the source region and relaxes that constraint based on the speed that information propagates through the space when a distance estimate needs to rise. For a thorough description, see [10].

Formulating the program in terms of these geometric relationships provides robustness as well, allowing it to self-stabilize (adjust from arbitrary values to correct values) in $O(diameter/c)$ time, where *diameter* is the maximum physical distance between devices and c is the maximum rate at which information propagates through the network [11]. Figure 12 shows an example of self-healing in a network of 1,000 simulated devices. Note that the performance of the algorithm is independent of the number of devices or their particular locations in space, and depends only on the bulk geometric properties of the aggregate.

Using the geometric primitives supplied by Proto tends to produce scalable and robust behaviors even when the relationship of the program to the continuous-space abstraction is less clear, as in this program for dispersing robots evenly through space via spring forces:

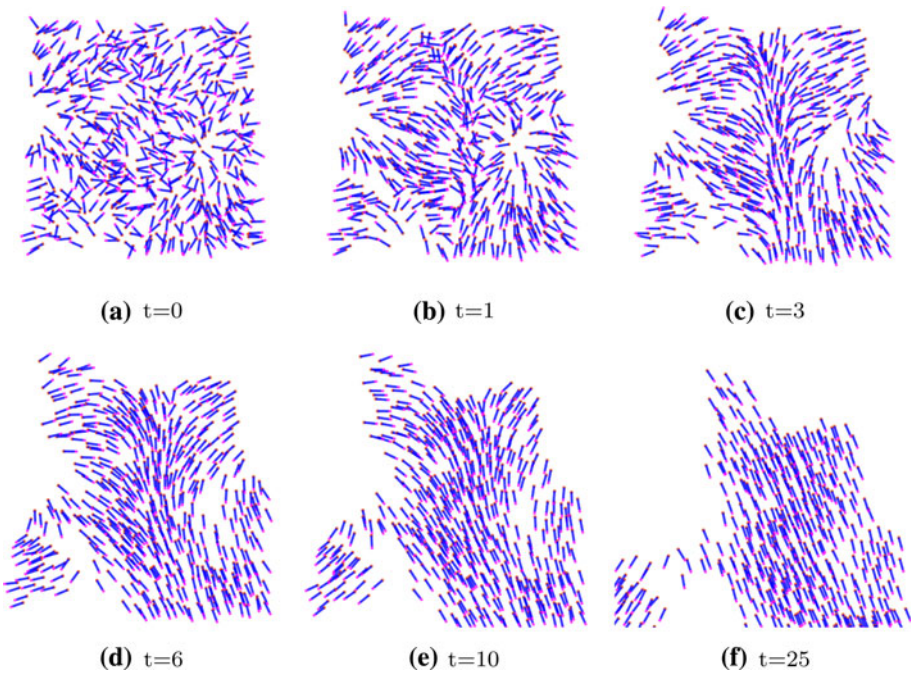


Fig. 11 As a swarm computes desired mass flow in an amorphous medium, the robots move to approximate that flow, reshaping the space, as in these snapshots from an evolving `flock` program executing on a swarm of 500 robots starting from a random distribution

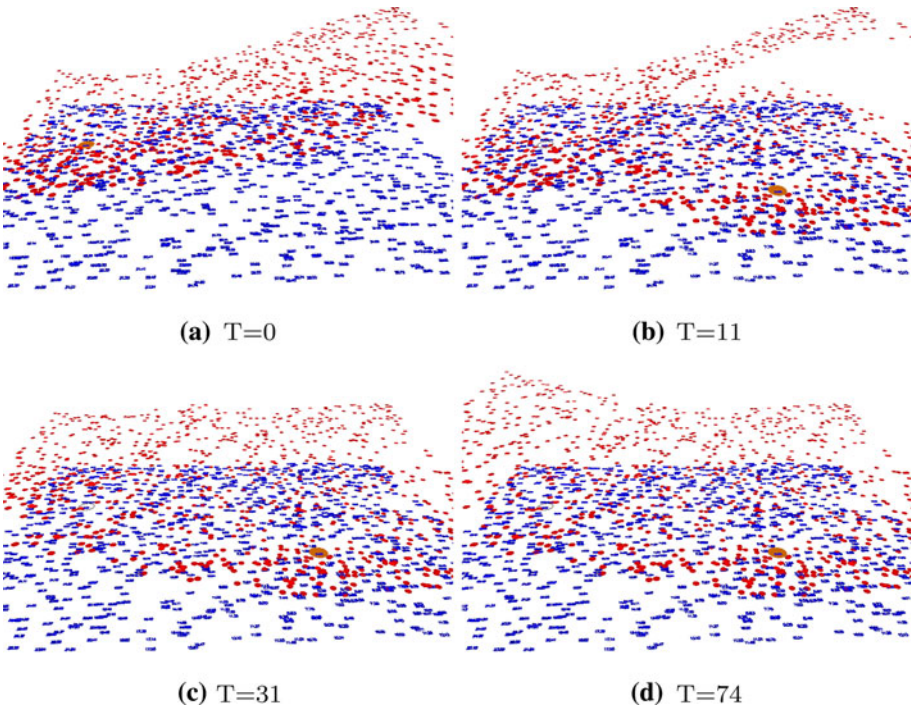


Fig. 12 A self-healing distance estimate, implemented with the `CRF-Gradient` algorithm, reconfigures in response to a change of source location (*orange* a minimum of *red dots*), running in simulation on a network of 1,000 devices, 19 hops across. The network is viewed at an angle, with the value shown as the height of the *red dot* above the device (*blue*). The source location starts at the *upper left* in the picture labeled T = 0 and then is immediately moved to the *lower right* in the remaining pictures and time steps. Reconfiguration spreads quickly through areas where the new value is lower than the old (**b**), then slows in areas where the new value is significantly higher (**c**), completing 74 rounds after the source moves

```
(def disperse ()
  (* (/ 1 (int - hood 1))                          ; normalize for neighborhood size
    (int - hood(* (- (nbr - range)(*0.9(comm - range))) ; integrate distance from fixed - point...
      (normalize (nbr - vec))))))                  ; ... times direction to neighbor
```

In this program, virtual springs connect a robot to its neighbors, attempting to keep a separation of 90% of the communication range. Using an integral to summarize the spring forces, however, ensures that robots are subject to a force scaled by the density of their packing (Fig. 13), and normalizing by neighborhood size means that the scaling of force due to neighborhood size is only due to the location of the equilibrium point.

The choice of spring forces as a physical model to emulate also allows this program to be robust and self-repairing when robots are confined within a bounded region. As shown in Fig. 14, even a large perturbation is rapidly recovered from as the spring forces push robots toward equilibrium.

Such geometric programs also often scale in the number of spatial dimensions, meaning that the same program should be usable for controlling swarms of both surface-bound robots and also underwater, aerial, or space-faring robots. The `flock` and `disperse` programs, for example, are entirely based on vector mathematics, and execute in three-dimensions as easily as in two, as shown on a swarm of 500 robots in Fig. 15. Likewise, the triangle inequality and velocity-based relaxation used in self-healing `distance-to` owe nothing to two dimensions and execute equivalently in three dimensions. In principle, the programs should also execute appropriately in one dimensional or more than three dimensional space, but two and three dimensions are most relevant for robotic swarms.

4.3 Composition

We have shown how a continuous-space abstraction facilitates the construction of robust and scalable behaviors. We now show how functional composition allows succinct and modular combination of such basic behaviors together to create larger swarm applications, and that these

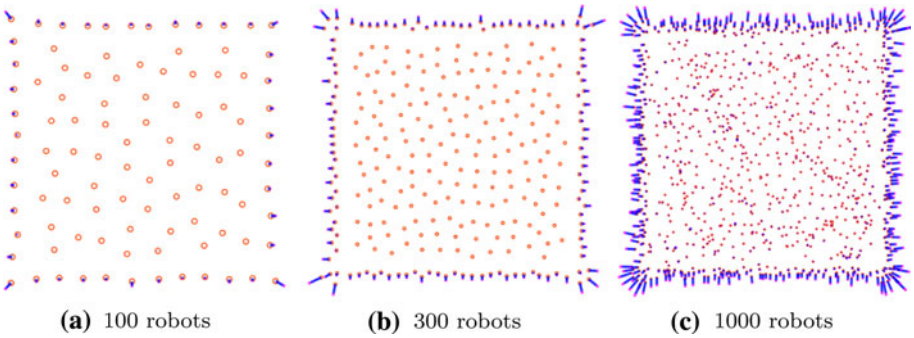


Fig. 13 The `disperse` algorithm distributes robots evenly through space by integrating spring forces over their neighborhoods. Note that force expressed with an integral automatically scales with the density of packing, so that the magnitude of the vectors changes little across an order of magnitude of different numbers of robots

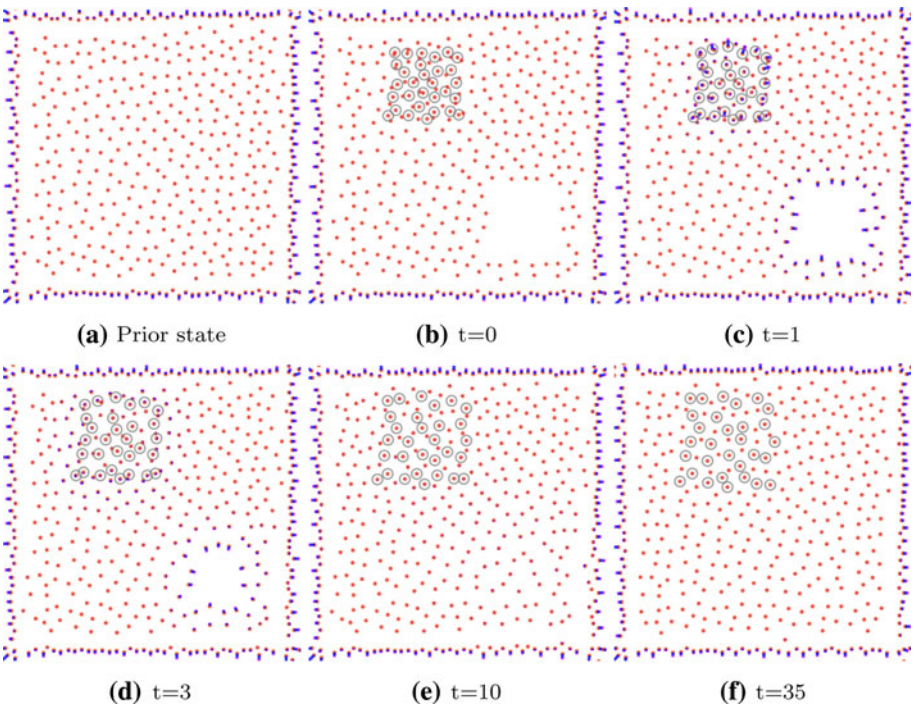


Fig. 14 In a swarm of 500 robots running `disperse` in a bounded space (**a**), even a large perturbation (**b**) such as a motion of a cluster of 30 robots (circled devices) is rapidly recovered from by the algorithm's continual adjustment (**c**, **d**, **e**) toward equilibrium (**f**)

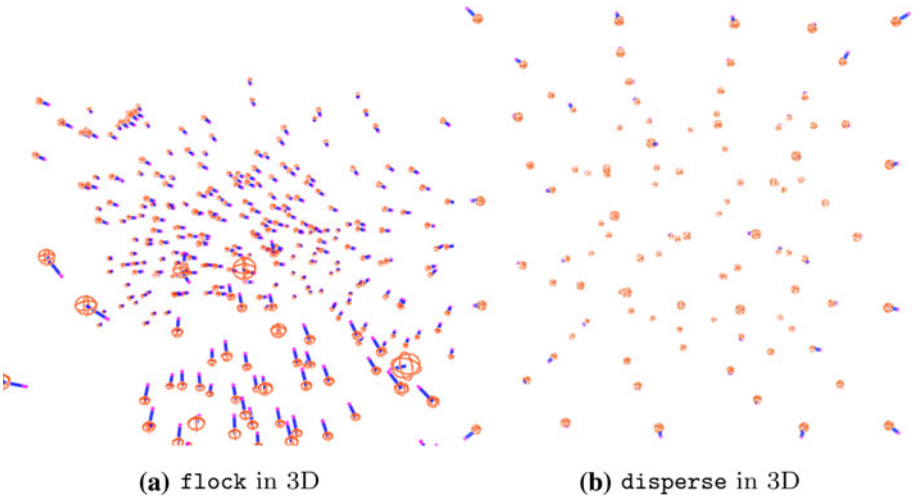


Fig. 15 Since the `flock` and `disperse` programs are expressed in terms of vector mathematics, they execute in three dimensions as easily as in two

behaviors can inherit the robustness and scalability properties of the behaviors they are built out of.

4.3.1 *Functional composition*

Because Proto uses functional composition and captures all state in feedback operations, once a useful spatial construct like self-healing `distance-to` has been created, we may think of it as though it were a primitive operation and use many instances without fear of interference between them.

For example, we can use the `distance-to` function to create higher-level spatial constructs, such as a dilation operator:

```
(def dilate (source rad)
  (< (distance − to source) rad))
```

that selects every point within distance `rad` of the source region. Unsurprisingly, the scalability of `distance-to` carries through the compositions: Figure 16 shows swarms of different density executing the same program, `(green (dilate (is-light) 0.5))`, to turn on a green light at every robot within 0.5 m of an active light sensor.

The `distance-to` function can also be used to broadcast values from a source region to the rest of the environment:

```
(def broadcast (source sent)
  (rep received sent
    (mux source sent
      (2nd (min − hood (nbr (tup
        (distance − to source) received)))))))
```

Here, the `distance-to` function creates an implicit spanning tree: tuples are compared lexicographically, so minimizing over a tuple of distance and `received` value has the effect of selecting the neighboring value closest to the source. Thus, new copies of the `sent` value move outwards along this implicit spanning tree away from the source. Because the `distance-to` function is self-healing, the broadcast function is as well—as the network changes, the distance estimates shift, and the path taken by information adapts as well.

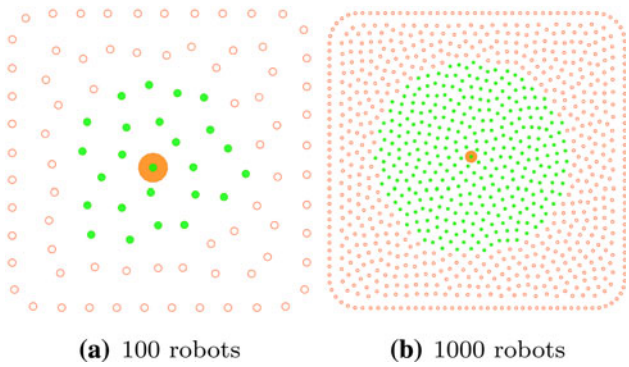


Fig. 16 Scalability carries through composition: the `dilate` operator behaves equivalently in swarms of 100 and 1,000 robots in this example, turning on a *green light* on every robot within 0.5 m of an active light sensor (*orange* disk)

Let us illustrate this with yet another function based on `distance-to`, which calculates the minimum distance between two regions and broadcasts it to every robot in the network:

```
(def distance (region1 region2)
  (broadcast region2 (distance − to region1)))
```

This uses one instance of `distance-to` to measure the distance between regions, and another instance to propagate that information throughout the network. As seen in Fig. 17, basing this function on self-healing primitives causes it to automatically detect a change in the distance between the designated regions and propagate the new information throughout the network of robots.

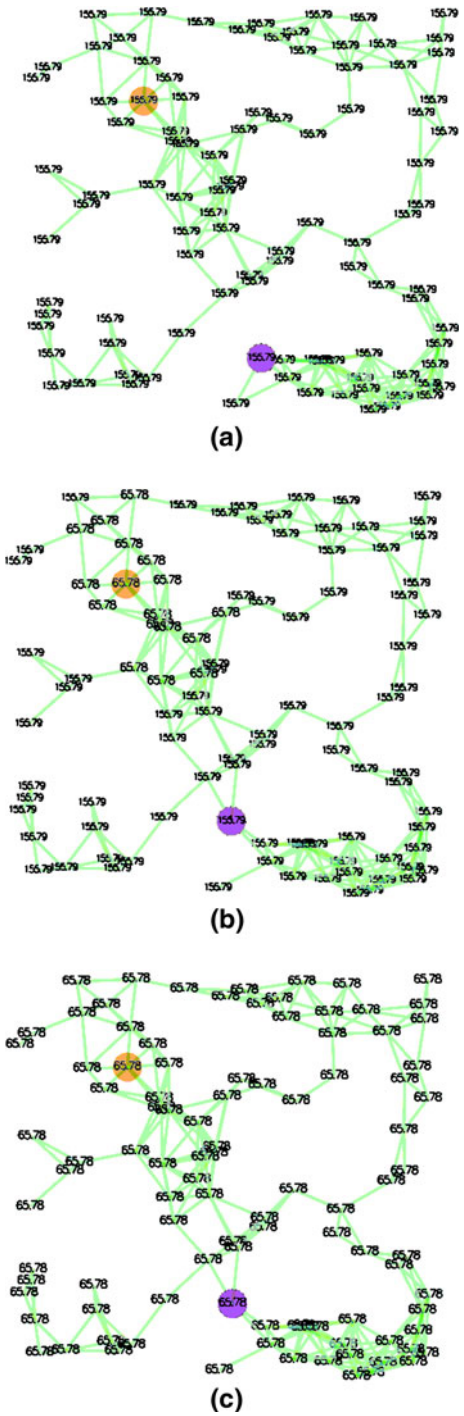


Fig. 17 Self-healing can carry through composition: here, a program based on two instances of self-healing `distance-to` computes the distance between two regions (*orange* and *purple*) to be 156.79 m (**a**). When one device moves, changing the connectivity of the network and becoming much closer, the self-healing of `distance-to` quickly causes the distance estimate to adjust to the new value, 65.78 m, which begins propagating through the network (**b**), converging again after 24 rounds (**c**)

We can transform such scalar field calculations into vectors to direct the motion of robots by computing a potential function and calculating its gradient to find the direction and magnitude of steepest descent:

```
(def grad (v)
  (* (/ 1 (int-hood 1)) ; normalize over neighborhood
    (int-hood (if (= (nbr-range) 0)
      (tup 0 0 0) ; ignore singularity
      (* (/ (- v (nbr v)) (nbr-range))
        (normalize (nbr-vec)))))))
```

Since an analytic description of the input field `v` is not generally available, this `grad` function estimates the gradient empirically from the change in values within its neighborhood.

When robots are instructed to follow such a gradient, they move toward regions of minimum potential. For example, they can be instructed to cluster toward a set of sources:

```
(def cluster-to (source)
  (grad (distance-to source)))
```

as shown in Fig. 10c. A more complex use is following a contour line of equal value, which can done by summing vectors toward and tangential to the desired contour line:

```
(def contour-field (field level)
  (let* ((vec (grad field)))
    (+ (* c (- level field) vec)
      (rotate pi/2 vec))))
```

where `c` is a feedback constant less than one. This creates a field with a stable limit cycle along the desired contour, as shown in Fig. 10d.

4.3.2 *Heterogeneous behavior*

Thus far, we have mostly seen only examples where the entire swarm of robots are all always performing the same behavior. In most applications, however, we would like the swarm to be able to differentiate its behavior in space and time. We have already seen an example of temporal differentiation of behavior in the `stoplight` finite state machine in Sect. 3.2. We will now see how domain restriction allows spatial differentiation of behavior and how this can be combined with temporal differentiation to assign groups of robots to tasks that change over time.

For example, let us consider a more sophisticated version of the `cluster-to` function discussed in the

previous section, in which robots assume one of two roles: movers and guides. The guides remain stationary, computing directions for how to get to the source, and the movers follow their directions and actually move toward the source. First, we use `if` to restrict a `distance-to` calculation to run only on a subset of the robots, but share the fruit of the distance calculation with any nearby non-calculating robots:

```
(def share-distance-to (is-calculating source)
  (let ((base (if is-calculating
    (distance-to source) (inf))))
    (mux is-calculating base
      (min-hood (+ (nbr-range) (nbr base))))))
```

We can then build our desired function, which we call `nav-grad`, to navigate down a `share-distance-to` potential field as follows:

```
(def nav-grad (is-mover source)
  (let ((g (grad (share-distance-to
    (not is-mover) source))))
    (mux (and is-mover (> (len g) 0))
      (normalize g) (tup 0 0)))))
```

where the guides calculate the `distance-to` scalar field upon which the gradient is based and the movers follow the gradient, as shown in Fig. 18a.

Since the components are modular and self-healing, it is relatively easy to improve this behavior. For example, we can have each subset of the swarm work together better as a team, changing the guides to be scouts, which use `disperse` to actively explore the space, and making the movers group together as a flock by interpreting the output of `nav-grad` as the preferred direction for flocking, rather than a direct motion command:

```
(def flock-nav-grad (is-mover source)
  (mux is-mover
    (flock (* 0.5 (nav-grad is-mover source)))
    (* 0.1 (if is-mover (tup 0 0) (disperse)))))
```

Since the scouts are moving, the gradient values are no longer fixed, but as long as the motion is relatively slow compared to the rate of self-healing, the program will perform as desired (Fig. 18b).

Spatial and temporal differentiation combine cleanly, allowing complex swarm behavior to be specified simply. For example, with the addition of a small utility to detect transitions from true to false in a Boolean field:

```
(def falling-edge (v)
  (mux and (not v) (delay v)))
```

we can combine the `nav-grad`, `broadcast`, `disperse` and `flock` behaviors to produce a simple search and rescue program:

```
(def search-and-rescue (rescuer victim base)
  (let* ((rescued (falling-edge victim)))
    (mux rescuers
      (letfed ((searching #t
        (if (any-hood
          (if searching
            (nbr rescued)
            (and (< (nbr-range) 3) (nbr base))))
          (not searching)
          searching)))
      (flock (nav-grad rescuers
        (if (broadcast rescuers searching)
          victim base))))
      (* 0.1 (if rescuers (tup 0 0 0) (disperse))))))
```

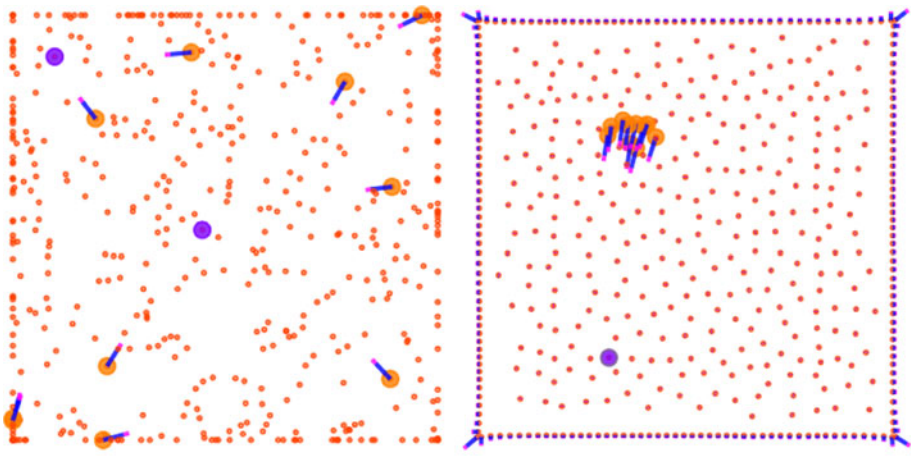


Fig. 18 Restriction enables heterogeneous behavior in a group of robots, such as "movers" (*orange*) following directions, individually (**a**) or as a flock (**b**), to a desired destination (*purple*) as calculated by "guides" (*red*)

The `rescuer` parameter is a Boolean field dividing robots into scouts, which disperse through the space to find victims (whose detection by some sensor suite is indicated by the `victim` Boolean field), and rescuers, which move as a flock guided by the scout robots to collect victims and return them near to a `base` location. Note that it is out of our scope to address how exactly a particular robotic platform might collect victims or transport them to the base: given our interests are in coordination of heterogeneous behaviors, we merely assume it is sufficient to get a coherent group of rescuers close by the victim or the base, respectively.

Six different interacting behaviors for robots are managed in the `search-and-rescue` program, involving composite behavior and heterogeneity in both space and time. Rescuers maintain cohesion as a flock at the same time as they are switching between searching for victims to collect (cued by the scouts) and returning collected victims to the base. Scouts maintain an even dispersal through space while switching between guiding the rescuers to victims and guiding the rescuers to the base.

Moreover, because the program is a composition of self-healing and scalable components that do not interfere with one another, the composite is itself self-healing and scalable. Figure 19 shows `search-and-rescue` running on 100, 300, and 1,000 simulated robots, and Fig. 20 shows an example of self-healing as the program adjusts to a simultaneous appearance of new victims, displacement of scouts, and addition of new rescuers.

All of this complicated heterogeneous scalable and self-healing behavior is expressed in only 56 total lines of Proto code, counting all of the subprograms that are composed to produce `search-and-rescue`. We argue that the simplicity of this program and the modularity of its components, most of which we have shown being used in other examples as well, are evidence that the continuous-space

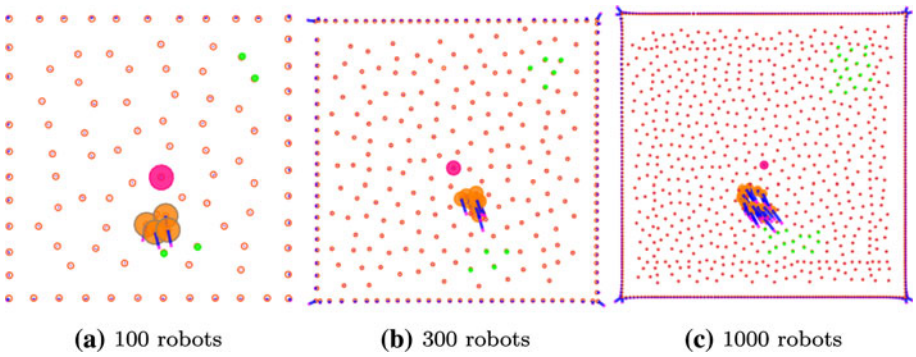


Fig. 19 A composite program for complex, heterogeneous behavior like `search-and-rescue` can inherit scalability from its components, as shown by these equivalent executions on 100, 300, and 1,000 simulated robots, where rescuers (*orange*) flock to collect victims (*green*) and return them to a base (*pink*)

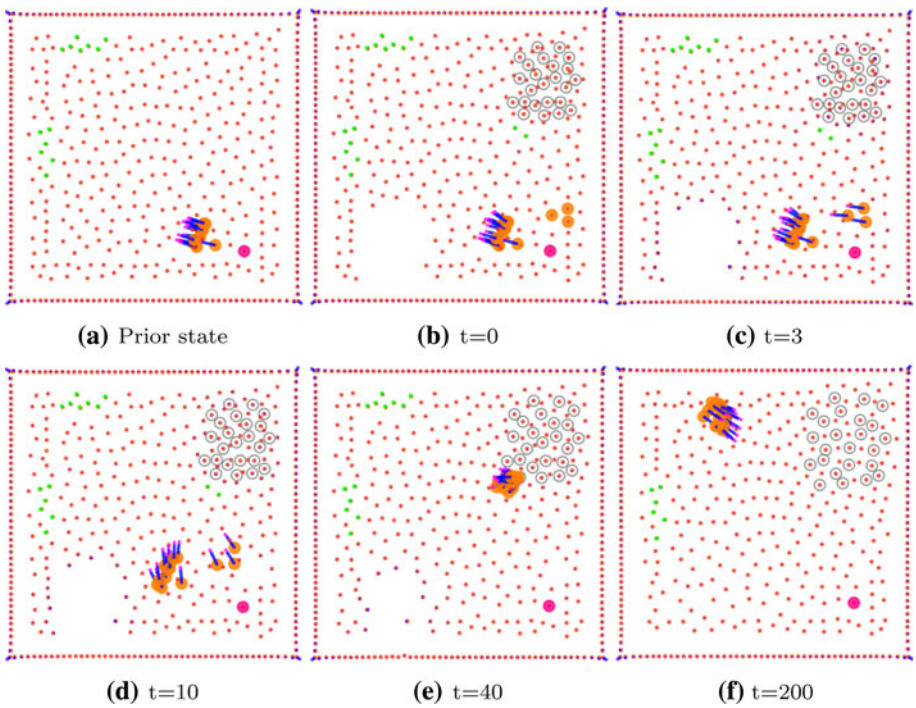


Fig. 20 A composite program for complex, heterogeneous behavior like `search-and-rescue` can inherit self-healing from its components, as shown in this execution on 500 simulated robots, where rescuers (*orange*) flock to collect victims (*green*) and return them to a base (*pink*). From an initial stable behavior (**a**), new victims and rescuers appear and a group of scouts is displaced (**b**). Within a few seconds the robots begin to adapt (**c**), quickly converging to handle the new victims (**d**, **e**), and eventually showing no evidence of the perturbation (**f**)

approach embodied in Proto is a powerful approach to programming robotic swarm behavior.

5 Verification on SwarmBots

We verified the applicability of Proto to multi-robot systems by executing three programs with easily quantifiable performance on a group of 40 iRobot SwarmBots. In every case, the quality of the approximation remains within reasonable bounds given the nature of the swarm. We may thus expect that composition of such primitives into complex Proto programs, as in Sect. 4, can produce predictable results for mobile devices just as it does for stationary devices [5].

The iRobot SwarmBot (Fig. 21) is a platform designed for distributed algorithm development [29]. Each SwarmBot is autonomous and has a 40 MHz 32-bit ARM Thumb microprocessor, which provides ample processing power for our algorithms. Robots periodically transmit to nearby neighbors within line of sight using an infrared communication system—for our experiments we set the power to produce a range of about 1.0 m. The infrared communication also allows each robot to determine the position and orientation of its neighbors within its own relative coordinate system, providing data for the `nbr-range`, `nbr-angle`, and `nbr-vec` functions: at a range of 500 mm between robots, this system has an accuracy of 2 degrees angle and 20 mm range under low-interference conditions. In addition, the robots use an omnidirectional bump skirt to avoid contact with obstacles.

We used a group of 40 SwarmBots distributed amorphously within a 2.43 m × 2.43 m (8′ × 8′) square pen. Position data was collected from the robots using a ceiling-mounted vision tracking system that recorded the positions of each robot at 1Hz and a calibrated accuracy of approximately 15mm, and radio telemetry was used to record each robot's internal state. Due to limitations of the telemetry system, we limit the maximum speed of the robots to 80 mm/s.

We begin by estimating distance on unmoving robots, to confirm that the implementation of Proto on SwarmBots is operating correctly:

$$(\texttt{distance} - \texttt{to}\,(\texttt{elect} - \texttt{leader}\,(\texttt{id})))$$

where `elect-leader` is a simple program of dubious scalability and no self-healing:

$$(\texttt{def elect} - \texttt{leader}\,(\texttt{id})\;(=\;(\texttt{rep minid id}\,(\texttt{min} - \texttt{hood}\,(\texttt{nbr minid})))\;\texttt{id}))$$

that gossips to find the minimum of the unique identifiers assigned to each robot, thereby selecting one robot to designate as leader (note that scalable and robust distributed leader election requires a much more sophisticated algorithm, of which there are many in the literature; here, we are just symmetry-breaking for the purpose of experiments). This leader is then used as the source for `distance-to`. Figure 22 shows the resulting estimates of distance to the nearest source, plotted against true straight-line distance. As expected, a single leader is elected—as demonstrated by the single robot with a distance estimate of zero—and all other robots estimate distances close to their true distance. Across the four hops from the leader to the furthest robots, these gradually become overestimates, which [23] and [5] predict should occur as accumulated error from the difference between straight-line and communication-graph distance begins to dominate over error from imprecise sensing.

We next verify that a running Proto program can use self-healing to adapt to mobility of a robot, running

$$(\texttt{dilate}\,(\texttt{elect} - \texttt{leader}\,(\texttt{id}))\,0.8)$$

and using radio control to drive the robot elected as leader in an arbitrary path through the swarm (Fig. 23a). The `dilate` function compares a distance estimate from

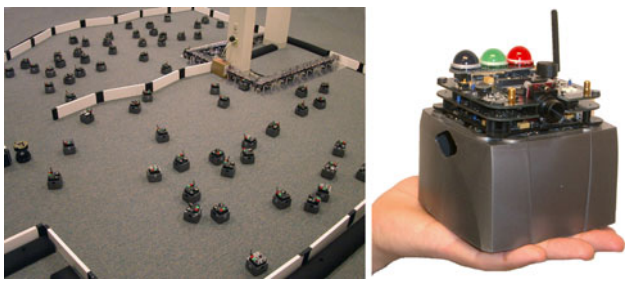


Fig. 21 Experiments were carried out on a group of 40 iRobot SwarmBots, a platform designed for distributed algorithm development. Each SwarmBot is autonomous and is equipped with bump sensors, light sensors, and an infrared inter-robot communication and localization system

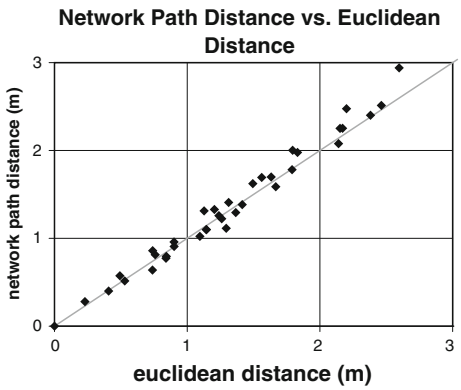
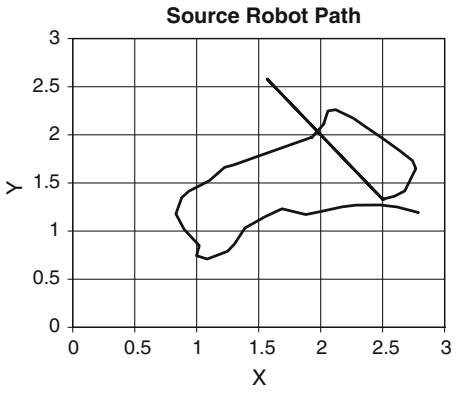
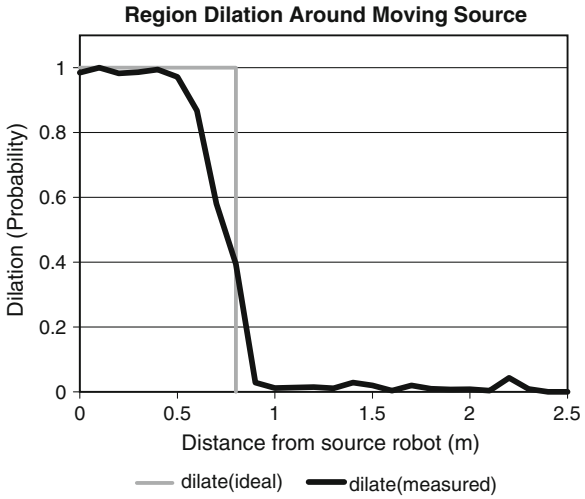


Fig. 22 The distance estimates produced by `distance-to` executing on real robots closely approximate true distances



(a) Source Robot Path

(b) Robots Running `dilate`



(c) Accuracy of Dilation

Fig. 23 When the leader (robot with *white light* in **b**) is moved arbitrarily through the swarm (**a**), a region selected by the `dilate` function (robots with *blue lights* in **b**) adjusts to follow it. Despite the constant motion of the source, the likelihood of a robot correctly deciding whether it is within the dilated region is high (**c**)

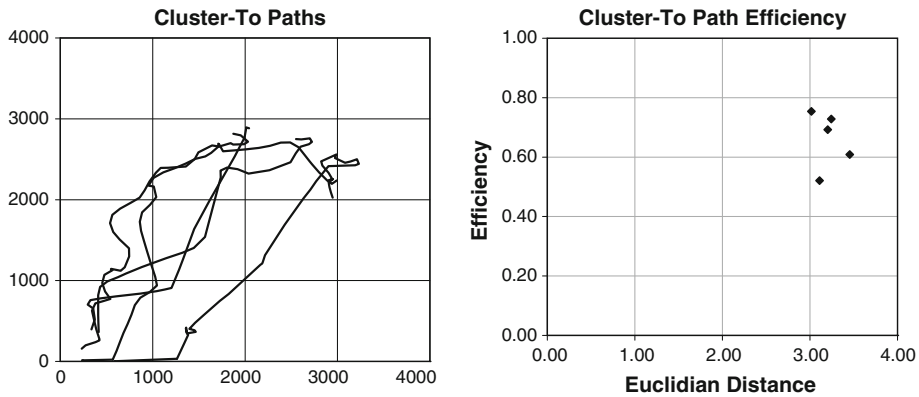
`distance-to` to its threshold distance to select the set of points within k meters of the source—in this case 0.8 m (Fig. 23b). We measured the positions of the robots and their current decision of whether they were in the dilated region, then grouped the measurements into 0.1-m bands of true distance to the leader and calculated the likelihood that a robot held the correct estimate at any given point in time as a function of distance from the source (Fig. 23c). Despite the constant motion of the source, the likelihood of a robot correctly deciding whether it is within the dilated region is high, although there is, unsurprisingly, significant error near the transition point.

Finally, we verify that Proto computations can be used to guide robots by computing a vector field for the robot to follow:

```
(mov (mux (elect − leader (− (id)))
  (cluster − to (elect − leader (id)))
  (tup 0 0)))
```

This program selects two distinct leaders, the robots with minimum and maximum unique identifiers, and drives the maximum robot toward the minimum robot by computing a gradient on the estimates of distance from the minimum robot. The maximum robot is then commanded to follow that vector field while all other robots remain stationary. In twenty trials of this program, the maximum robot navigated successfully to the minimum robot every time, although error and the discrete approximation cause paths to wander somewhat, as shown by a plot of five sample paths in Fig. 24a. The overall efficiency of the navigation is reasonable and consistent, as shown by the plot of ratio of actual to shortest path distance against shortest path distance, shown for the sample paths in Fig. 24b.

Although these three experiments are simple and do not exercise the full mobility of the swarm, they verify that Proto programs can be accurately approximated on robots as well as on static devices, and give us reason to believe that the much more complex programs described and



(a) Paths followed by mobile robot **(b)** Path efficiency

Fig. 24 Computing a gradient on the distance to a source robot allows a mobile robot to navigate efficiently toward the source. Although error and the discrete approximation cause paths to wander somewhat, as seen in the five paths shown in **a**, the overall efficiency of the navigation is reasonable and consistent, as shown by the plot of ratio of actual to shortest path distance against shortest path distance in **b**

simulated in Sect. 4 should be able to execute predictably on real robots.

6 Contributions and challenges

We have shown that the Proto language can be used to construct robust, scalable behaviors for robotic swarms. This is enabled by viewing the robotic swarm as a spatial computer, using a generalization of the amorphous medium abstraction to mobile devices. When properly designed, swarm behaviors written in Proto may be simply composed to produce complex behaviors that inherit robustness and scalability from their components. We have further verified the applicability of Proto to multi-robot systems by executing some of the behaviors we have developed on a group of 40 iRobot SwarmBots and showing that the quality of the approximation remains within reasonable bounds.

At a pragmatic level, we expect that Proto will continue to become more useful as a robotic programming platform as more robotic applications are developed in it and as the library of available components is extended. Translating existing approaches into Proto challenges the language, ensuring that it is sufficiently general, as well as the scalability and robustness of the approaches, since it has been our experience that transformation into a continuous-space formulation often exposes hidden assumptions and scalability problems.

Many open questions remain, however, both in the fundamentals of Proto and its application to mobile devices. One critical question is how best to characterize the duality between network and space and, in the domain of robotics, to relate the specification of mass flow in the continuous abstraction to its approximation by robotic motion. We have also treated only the case of robots in the gaseous state—while we expect that Proto will also be applicable to tightly packed robots and modular robots, this has not yet been demonstrated and the details of the application are likely to differ.

Another important challenge is how to handle swarms that split into sub-swarms. While the basic continuous-space approach is straightforward (a split swarm maps to a disconnected manifold), it is subject to the same difficulties that other swarm algorithms face. McLurkin's thesis [29] provides much information on this problem. One way to address this is to employ a cohesion mechanism to keep swarms together using any information available such as GPS and landmarks. Another approach is to use search mechanisms to have sub-swarms find and rejoin their swarms. While there are many approaches, the continuous-space approach does not dictate a particular one, but should be able to support a generalized version of any discrete algorithm.

We also do not address sensor noise in this paper, but note that its effects can be mitigated either within the virtual machine, by combining sensor information from neighboring robots, or in the continuous-space abstraction by bounding the impact of discretization error, as shown for the case of geometric computations based on distance measurements in [5]. Although this result is a promising beginning, the three-way relation between sensor error, discretization error, and program behavior still needs to be extended to cover more general geometry-based programs.

Heterogeneity of robots in a swarm can be addressed in a straightforward manner as long as the robots adhere to the virtual sensor/actuator abstraction layer. For example, a continuous-space program does not care what sensors a robot uses to discover its relative position. The availability of a particular sensor or actuator can also be indicated as a Boolean-valued virtual sensor or actuator, and a program can case its execution based on which resources are available where.

Finally, although we hope to have made the advantages of a continuous-space approach clear in this paper, the advantage of Proto over other swarm programming

approaches has not been measured through quantitative user studies. One result suggestive of the large advantage that we expect such studies would discover is presented in [3], where we showed an implementation of Eames's algorithm for distributed discovery of minimum threat paths [16] in 25 lines of Proto code, while Eames's implementation in nesC [18] code was approximately 2000 lines long.

Although these challenges are significant, they are largely ones that any approach to programming swarm behaviors must face. Taking a spatial computing approach to swarm robotic systems allows us to tackle each relatively independently, however, and we believe that the factoring of the problem provided by Proto is a route toward rapid advancement of swarm robotic capabilities.

Acknowledgments This work was partially funded by the National Science Foundation via NSF grant CCF-0621897.

References

1. Ashley-Rollman MP, Goldstein SC, Lee P, Mowry TC, Pillai P (2007) Meld: a declarative approach to programming ensembles. In: IEEE international conference on intelligent robots and systems (IROS '07), IEEE Press, pp 2794–2800
2. Bachrach J (2004) Gooze: a stream processing language. In: Lightweight languages 2004, http://www.ll4.csail.mit.edu/
3. Bachrach J, Beal J (2006) Programming a sensor network as an amorphous medium. In: Distributed computing in sensor systems (DCOSS) 2006 Poster
4. Bachrach J, Beal J, Fujiwara T (2007) continuous-space–time semantics allow adaptive program execution. In: IEEE SASO 2007, IEEE Press, pp 315–319
5. Bachrach J, Beal J, Horowitz J, Qumsiyeh D (2008) Empirical characterization of discretization error in gradient-based algorithms. In: IEEE international conference on self-adaptive and self-organizing systems (SASO) 2008, IEEE Press, pp 203–212
6. Beal J (2004) Programming an amorphous computational medium. In: Unconventional programming paradigms international workshop, Springer, Berlin, vol 3566, pp 121–136
7. Beal J, Bachrach J (2006) Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intell Syst 21(2):10–19
8. Beal J, Bachrach J (2007) Programming manifolds. In: DeHon A, Giavitto JL, Gruau F (eds) Computing Media and Languages for Space-Oriented Computation, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, no. 06361 in Dagstuhl Seminar Proceedings
9. Beal J, Bachrach J, Tobenkin M (2007) Constraint and restoring force. Technical Report MIT-CSAIL-TR-2007-044, MIT CSAIL
10. Beal J, Bachrach J, Vickery D, Tobenkin M (2008) Fast self-healing gradients. In: ACM symposium on applied computing, ACM, pp 1969–1975
11. Beal J, Bachrach J, Vickery D, Tobenkin M (2009) Fast self-stabilization for gradients. In: Distributed computing in sensor systems (DCOSS) 2009, Springer, Berlin, pp 15–27
12. Butera W (2002) Programming a paintable computer. Ph.D. thesis, MIT
13. Correll N, Bachrach J, Vickery D, Rus D (2009) Ad-hoc wireless network coverage with networked robots that cannot localize. In: IEEE international conference on robotics and automation, Kobe, IEEE Press
14. Couzin I, Krause J, James R, Ruxton G, Franks N (2002) Collective memory and spatial sorting in animal groups. J Theor Biol 218:1–11
15. Couzin I, Krause J, Franks N, Levin S (2005) Effective leadership and decision making in animal groups on the move. Nature 433:513–516
16. Eames A (2005) Enabling path planning and threat avoidance with wireless sensor networks. Master's thesis, MIT
17. Elliott C, Hudak P (1997) Functional reactive animation. In: Proceedings of the ACM SIGPLAN international conference on functional programming (ICFP '97), ACM, vol 32, no, 8, pp 263–273, http://citeseer.ist.psu.edu/article/elliott97functional.html
18. Gay D, Levis P, von Behren R, Welsh M, Brewer E, Culler D (2003) The nesc language: a holistic approach to networked embedded systems. In: Proceedings of programming language design and implementation (PLDI) 2003, ACM, pp 1 – 11
19. Gummadi R, Gnawali O, Govindan R (2005) Macro-programming wireless sensor networks using kairos. In: DCOSS, Springer, Berlin, pp 126–140
20. Hill J, Szewcyk R, Woo A, Culler D, Hollar S, Pister K (2000) System architecture directions for networked sensors. In: In architectural support for programming languages and operating systems (ASPLOS) 2000, pp 93–104
21. Jones SP, Hughes J (2002) The revised report on the programming language haskell 98. http://www.haskell.org/onlinereport/
22. Klavins E (2004) A language for modeling and programming cooperative control systems. In: Proceedings of the international conference on robotics and automation, IEEE Press, pp 3403–3410
23. Kleinrock L, Silvester J (1978) Optimum transmission radii for packet radio networks or why six is a magic number. In: National Telecommunications Conference, pp 4.3.1–4.3.5
24. Kloetzer M, Belta C (2006) Hierarchical abstractions for robotic swarms. In: IEEE international conference on robotics and automation, IEEE Press, pp 952–957
25. Levis P, Patel N, Culler D, Shenker S (2004) Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: Proceedings of the 1st conference on symposium on networked systems design and implementation, ACM, vol 1, pp 15–28
26. Madden S, Franklin M, Hellerstein J, Hong W (2005) Tinydb: an acquisitional query processing system for sensor networks. In: ACM Transactions on Database Systems (TODS), ACM, vol 30, pp 122–173
27. Margolus N (1993) CAM-8: A computer architecture based on cellular automata. In: Pattern formation and lattice-Gas Automata
28. Mataric M, Marjanovic M (1993) Synthesizing complex behaviors by composing simple primitives. In: Proceedings, self organization and life, from simple rules to global complexity, European Conference on Artificial Life (ECAL-93), pp 698–707
29. McLurkin J (2004) Stupid robot tricks: a behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT
30. Minar N, Burkhart R, Langton C, Askenazi M (1996) The swarm simulation system, a toolkit for building multi-agent simulations. Technical Report Working Paper 96-06-042, Santa Fe Institute
31. Mondada F, Gambardella LM, Floreano D, Nolfi S, Deneubourg JL, Dorigo M (2005) The cooperation of swarm-bots: physical interactions in collective robotics. IEEE Robotics Autom Mag 12(2):21–28
32. Newton R, Welsh M (2004) Region streams: functional macroprogramming for sensor networks. In: First international workshop on Data Management for Sensor Networks (DMSN), ACM, pp 78–87

33. Newton R, Girod L, Craig M, Madden S, Morrisett G (2008) Wavescript: a case-study in applying a distributed stream-processing language. Technical Report MIT-CSAIL-TR-2008-005, MIT CSAIL
34. Palmer J, GL Steele J (1992) Connection machine model cm-5 system overview. In: Fourth symposium on the frontiers of massively parallel computation, IEEE Press, pp 474–483
35. Raffle H, Parkes A, Ishii H (2004) Topobo: a constructive assembly system with kinetic memory. In: Proceedings of the SIGCHI conference on human factors in computing systems, ACM, pp 647–654
36. Rosa MD, Goldstein SC, Lee P, Campbell JD, Pillai P (2006) Scalable shape sculpting via hole motion: motion planning in lattice-constrained module robots. In: IEEE International Conference on Robotics and Automation (ICRA '06), IEEE Press
37. Rosa MD, Goldstein SC, Lee P, Campbell JD, Pillai P (2008) Programming modular robots with locally distributed predicates. In: IEEE International Conference on Robotics and Automation (ICRA '08)