Protoswarm: A Language for Programming Multi-Robot Systems Using the Amorphous Medium Abstraction

Jonathan Bachrach
MIT CSAIL
Cambridge, MA 02139 USA
Email: jrb@csail.mit.edu

James McLurkin
MIT CSAIL
Cambridge, MA 02139 USA
Email: jamesm@csail.mit.edu

Anthony Grue
Microsoft
Redmond, WA 98052 USA
Email: tonyg@alum.mit.edu

***Abstract*—Multi-robot systems are becoming increasingly prevalent, but programmability is a major barrier to their deployment. Present systems force programmers to think in terms of individual agents. Application code becomes entangled with details of coordination and robustness and often does not compose well or translate to other domains. We offer an alternate approach whereby the programmer controls a single virtual *spatial computer* which fills the environment space. The computations on this spatial computer are actually performed by a large number of locally-interacting individual agents. This abstracts the actual computational hardware behind the spatial computer interface, and allows the programmer to focus on a single model of global computation. We achieve this abstraction with two components: a language that embodies continuous space and time semantics and a runtime library that implements these semantics approximately. We demonstrate the efficacy of our approach with multi-agent algorithms in both simulation and on a group of 40 robots.**

I. Introduction

Multi-robot applications involve coordinating the movement of robots in space over time. However, programming multi-robot applications requires the user to write software for individual robots and then imagine how these robots will interact to produce the final application. The mapping from robot actions to group actions is often complex and difficult to invert, making programming these systems challenging.

The dream is that by using a high level language to program a multi-robot application, we would empower a programmer to succinctly implement low-level repair and group functionalities and to quickly compose new programs out of these components. Programs would be developed with a more intentional macro perspective, and modularity would be strongly promoted.

Recently there has been significant success in this direction in the domain of sensor networks in developing real high-level programming languages that utilize common programming models. For example, TinyDB [11] takes the database point of view, and Regiment [16] and Proto [4] [2] take the reactive/streams point of view.

However, one limitation of the sensor network languages so far is the focus on sensing and data gathering, and the lack of actuation and control as a fundamental part of the programming model. This makes them illsuited for application to mobile robots. Furthermore the underlying implementations often assume static networks of immobile nodes, and topologies that change infrequently due to node failures. In contrast, mobile robots are dynamic network and this poses significant challenges in how the abstractions/primitives of these languages can be supported.

In this paper, we present a new language, called Protoswarm, that is inspired by the continuous space-time model of Proto and extends this type of model to program swarms of robots. In particular, we present a virtual *spatial computer*, built out of a continuous *Amorphous Medium*, which fills the environment space. Programmers develop code for this medium without considering the details of the individual agents. The computations on this spatial computer are actually performed by a population of locally-interacting agents. The agents approximate the virtual computer presented to the user. We achieve this Amorphous Medium Abstraction [3] using two mechanisms: a language, called Protoswarm, which provides continuous space and time semantics, and a runtime library which approximates the semantics on the given hardware.

In this paper, we describe the use of the abstraction and the Protoswarm language in the domain of multi-robot applications. We show how

this language can allow the programmer to write programs that are insulated from the details of the hardware allowing the same program to run on a variety of hardware, to scale to bigger populations, and to be largely robust to robot failures. This paper is divided into three main sections. First, we introduce the framework, and the Protoswarm language. Next, we discuss primitive functions, spatial behaviors, and behavioral combinators. Finally, we present experiments in simulation and on a group of 40 robots and provide experimental results of the accuracy and robustness of our approach.

II. Related Work

There are many domain-specific programming models for spatial computers, Swarm [15], TinyOS [7], Paintable Computing [5], and CAs [12], but they all involve programming the behavior of the devices, rather than the behavior of the aggregate. A notable exception is CMost, the operating system for the CM-5 [17], which allows operations on fields of devices, but assumes a fixed population of devices arranged in a grid.

In related swarm languages, programmers are similarly forced to program and manage individual robots. Mataric [13] introduced the notion of basis behaviors and group computng, but the basis behaviors are more challenging to combine than in Protoswarm. More recently, works by Klavins [9] and Kloetzer [10] 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. However 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 focussed on motion, and do not provide very expressive means of distributed sensing and distributed state, that one might like to do wth a robot swarm application.

In contast, sensor networks have focussed almost exclusively on data collection. They have focussed on using well-known complete languages such as SQL or functional languages, which come with strong guarantees about what can be computed and many algorithmic tools to support the language implementation. For example, the Regiment [16] programming language operates on geometric regions of space, but is targeted towards sensor-network data-gathering and only distributes some operations across space.

Finally, the structure of Protoswarm as a dynamic network of streams is strongly influenced by previous work on Gooze[1], as are many of the compilation strategies used to compact Protoswarm code for execution on robots. There is a long tradition of stream processing in programming languages. The closest and most recent work is Functional Reactive Programming (FRP) [6] that is based on Haskell [8], which is a statically typed programming language with lazy evaluation semantics. In these systems, less attention is spent on runtime space and time efficiency, and the type system is firmly wedded to Haskell, with all of its strengths and weaknesses.

Our goal is to combine these two points of view – both sensing and motion control are fundamental parts of programming robot swarms. We would like to take advantage of these types of languages in robot swarms, since many times mobile sensor networks are essentially robot swarms and vice versa.

The Proto language is described in [4], and its applicability to sensor networks in [2]. The amorphous medium abstraction was first proposed in [3].

III. Multi-Agent Programming in Protoswarm

In this section, we introduce the Protoswarm language, and build up facilities that support high-level modular multi-agent programming. In Protoswarm, the computational model is based on manifolds of space that execute code, called the Amorphous Medium. The medium has computational state and physical extent, both of which evolve over time. We assume that the medium is populated by an infinite number of agents, and each agent can only communicate with neighbors within a fixed distance. Programs for continuous regions are then run approximately on a discrete set of agents. Each agent runs identical code but their execution diverges due to differing local state and environment and interactions with neighbors.

Protoswarm programs are written as expressions over fields, where fields are mappings from manifolds to values. Expressions are executed repeatedly, producing streams of fields. Behaviors are produced from vector fields by points in space moving in the direction of the vectors.

Protoswarm is inspired by the programming model of Proto [4]. We treat the world as fields/streams and the computing constructs compute on these streams. Also we treat the system as a spatial computer, so all computing constructs work on neighborhoods without reference to exact neighbors. Unlike Proto however agents can move

in space, and this is described by adding movement actuation. Figure 1 outlines the basics of Protoswarm broken into ten categories. Consult [4] for more information on the Protoswarm programming language. Now we describe some constructs one can build in this language to program at the group level.

A. Basic Spatial and Temporal Functions

This section develops a number of basic spatial and temporal functions that are useful in programming the Amorphous Medium. We describe functions to measure distance, elect a leader, define a subregion, broadcast data, and provide a global clock.

To extend geometry beyond local neighborhoods, we need to be able to calculate distances between points in space that are further apart than a robot's communication range. The first function, called `distance-to`, measures the distance between any point and a designated source region. This function calculates the distance using a relaxation algorithm. The distance is set to zero inside the source region. Other points compute their distance to the source region by minimizing over the sum of the distances to each neighboring point and its current distance estimate. In Protoswarm code:

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

We can use this measure of distance to construct higher-level spatial constructs. We can calculate a dilated region around a source region by including all points that are within `rad` distance from any point in the source region:

```
(def dilate (is-source rad)
  (< (distance-to is-source) rad))
```

The predicate in the second line evaluates to true for any point within distance `rad` of the source region. This produces a new region that is a dilated version of the source region. For example, the dilation of a lit region:

```
(def is-light ()
  (> (light) 0.5))

(green (dilate (is-light) 0.5))
```

produces the results shown in Figure 2.

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

```
(def gradcast (src val)
  (rep res val
    (mux src val
      (2nd (min-hood
             (nbr (tup (distance-to src) res))
             1st)))))
```

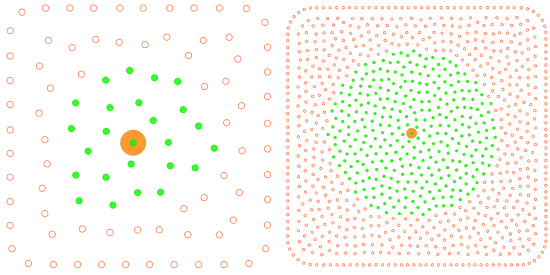

Fig. 2. Dilation simulation example with 100 and 1000 robots, where the orange disks represent lit regions.

Points receive values from nearest sources by minimizing over `distance-to`/`value` tuples using the `distance-to` value as the key.

We can elect a leader over a region according to the following algorithm:

```
(def elect-leader (id)
  (= (rep minid id (min-hood (nbr minid)))
     id))
```

which works by minimizing over each point's unique `id`'s.

Finally, we can define a time synchronization function as follows:

```
(def time ()
  (rep t 0 (max-hood (+ (nbr t) (nbr-lag)))))
```

Each agent has a free-running clock `t`. Using a function similar to leader election, the maximal clock value `t` over a region is computed. This clock becomes the clock for the entire region. Ultimately, the highest value will come from the fastest clock, and the region will remain in sync from that point on.

B. Building Basic Behaviors

In this section, we build upon our basic spatial and temporal functions to produce simple motion primitives for wandering, clustering, and dispersion. We move regions by defining a vector field over a region and using this vector field to move the points in the region. For example, a random vector field is produced by:

```
(def brownian (s)
  (tup (rnd (- s) s) (rnd (- s) s)))
```

This produces a tuple at each point in the region that represents a random change in that point's current goal position. Note that this is simply a field; we have not produced any motion yet. We can move each point to its goal position with:

literals – *are self evaluating:*

```
#t → #t
#f → #f
1 → 1
```

tuples – *produce tuples of evaluated expressions and can be nested arbitrarily:*

```
(tup 1 2) → (tup 1 2)
(tup (tup 1 2) 3) → (tup (tup 1 2) 3)
(elt (tup 1 2) 0) → 1
(1st (tup 1 2)) → (elt (tup 1 2) 0)
(2nd (tup 1 2)) → (elt (tup 1 2) 1)
```

introspection – *provides point properties, where* `comm-range` *produces the communication range,* `(id)` *produces the point's unique number and* `(dt)` *produces the time since last execution:*

```
(comm-range) → num
(id) → num
(dt) → num
```

sensors – *provide sensory data:*

```
(button) → num
(light) → num
```

actuators – *drive the agent towards target states:*

```
(rgb (tup (button) 0 0) → tup
(mov (tup 1 0)) → (tup 1 0)
```

conditionals – *produce selected expression, but where* `mux` *evaluates both branches and* `if` *evaluates only the taken branch:*

```
(mux #t (tup 1 2) (tup 2 3)) → (tup 1 2)
(if #f (red 1) (blue 1)) → 1
```

pointwise operators – *provide random, trigonometric, and arithmetic functions:*

```
(pi) → π
(inf) → ∞
(rnd -1 1) → num
(neg (pi)) → −π
(sin (rnd (neg (pi)) (pi))) → num
(+ 1 2) → 3
(+ (tup 1 2) (tup 2 3)) → (tup 3 5)
```

functions – *create functional abstractions, where* `fun` *produces anonymous and* `def` *produces named functions:*

```
((fun (x) (* x x)) 2) → 4
(def sqr (x) (* x x)) → fun
(sqr 2) → 4
```

temporal integration – *maintains and combines values over time using feedback loops with initial and update expressions, which can optionally be tuple expressions:*

```
(rep t 0 (+ t (dt))) → num
(rep (tup x y) (tup 0 0) (tup y x)) → tup
(once x) → (rep y x y)
```

spatial integration – *summarizes neighborhood values using* `nbr` *expressions, with* `min-hood` *for minimizing,* `max-hood` *for maximizing,* `int-hood` *for integrating over neighborhood expressions, and where* `(nbr-lag)` *is the time since receipt of last data from neighbor, and* `(nbr-vec)` *is the relative vector to the neighbor.*

```
(min-hood (nbr d)) → num
(max-hood (+ (nbr t) (nbr-lag))) → num
(int-hood (nbr-vec)) → num
```

Fig. 1. Overview of Protoswarm Basics. All expressions evaluate to and operate on streaming fields.

```
(mov (brownian))
```

which produces the desired behavior.

Regions can be clustered into a set of smaller regions by moving each point towards the average of the positions of all neighboring points:

```
(def cluster ()
  (int-hood (nbr-vec)))
```

As this code executes, points near the boundary of the region move towards the center of the region. No attempt is made to keep the region coherent as it clusters. Eventually, the region will contract to a set of points.

Conversely, regions can be dispersed by creating virtual springs between points with a resting length of d. The following fragment:

```
(def disperse (d)
  (int-hood
    (* (- 1 (/ d (nbr-range))) (nbr-vec)))))
```

moves points to minimize the spring energy between neighbors. This eventually results a uniform dispersion [18].

In order to perform clustering and dispersion directed towards (or away from) a region, we need a way to determine the direction to a region. We can interpret a field of scalars in a region as the z-values of a topographic terrain. We can then compute the gradient at any point in the region by finding directions of maximal increase in height:

```
(def grad (field)
  (int-hood
    (* (/ (- (nbr field) field) (nbr-range))
       (nbr-vec))))
```

The third line computes a vector towards each neighbor, with magnitude equal to the gradient of the `field` towards each neighbor. The `int-hood` operator integrates all the gradient vectors within a local region around each point, called a *neighborhood*. The neighborhood of a point is the circle of radius `comm-range` centered at that point. Essentially, this function computes the average gradient vector towards the source from all the points in the neighborhood.

Distance-based dispersion and clustering can be defined by moving towards or away from this gradient vector:

```
(def disperse-from (src)
  (grad (distance-to src)))

(def cluster-to (src)
  (* -1 (grad (distance-to src))))
```

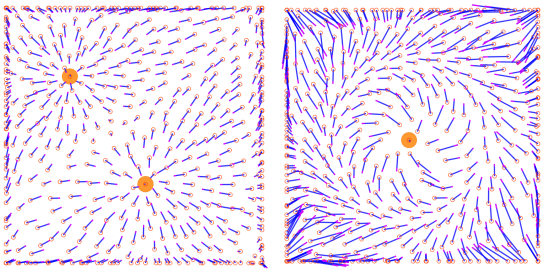

Fig. 3. Cluster-to and contour-field vector field examples run on 500 simulated robots.

The left picture of Figure 3 shows the vector field produced by:

```
(cluster-to (is-light))
```

where the vectors points towards lit regions.

Finally, we can follow a contour line in the topography of the field. Our approach is to create another field that has a stable limit cycle along the contour at a given level. We can generate this field by summing vectors pointed both towards and tangential to the desired topographic 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. The following example, produces a vector field causing points to orbit at 0.5 meter around the lit region:

```
(contour-field (distance-to (is-light)) 0.5)
```

as shown in the right picture of Figure 3.

C. Behavioral Combinators

In order to construct more complicated behaviors, we need a method for behavioral composition. The first mechanism spatially composes behaviors. For example, we can create a behavior that makes agents disperse and remain somewhat stationary over certain areas, while wandering everywhere else. The following example creates a dispersal field in lit areas, and a brownian field in other regions:

```
(def cover-light ()
  (if (is-light) (disperse) (brownian)))

(mov (cover-light))
```

The second mechanism composes behaviors over time, sequencing behaviors according to events. For example, we can sequence dispersion for 2 seconds followed by wandering for 3 seconds:

```
(loop (while (wait 2) (disperse))
      (while (wait 3) (brownian)))
```

or wander until coming in contact with an object, then pushing it for 5 seconds:

```
(loop (while (not (is-near-object)) (brownian))
      (while (wait 5) (push-object)))
```

In general, we can sequence arbitrary behaviors by introducing the notion of finite streams, which are truncated by some event. The `while` function creates a stream of fields while the predicate is true. Finite streams are represented as a tuple of value fields and a boolean *active field* that is true when the stream is active. The `loop` function transitions from finite stream to stream based on the active field.

IV. Implementation

The Protoswarm implementation addresses three separate challenges: (a) how to implement the primitives in a fault-tolerant manner in the face of agent movement, (b) how to translate swarm programs onto actual robots in an efficient and portable manner, and (c) how to support program development.

In order to facilitate portability, Protoswarm is implemented on top of a virtual machine and a hardware abstraction layer whose device interface provides sensing, actuation, communication, and neighbor positions. This allows the same code to run on different hardware platforms and in simulation.

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 communications and localization are supported by a best-effort communication scheme. The most recent information on neighbor's relative positions and shared variables are stored in a table [5]. Neighborhood operations then access the table, combining the most recent values into an approximate summary value. The virtual machine maintains the table by gathering shared values during each round of execution. These are then transmitted each round while receipt of packets proceeds in the background.

A. Simulator

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

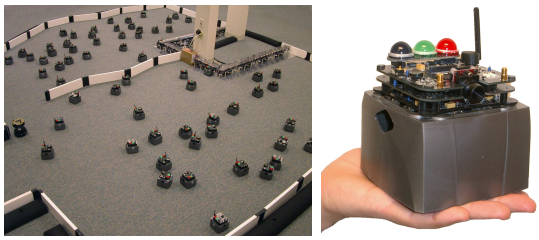

Fig. 4. The iRobot SwarmBot is designed for distributed algorithm development. Each SwarmBot has an infra-red localization and communication system which enables nearby robots to communicate and determine the position and orientation of their neighbors. An omnidirectional bump skirt provides low-level obstacle avoidance. A 40 MHz 32-bit ARM Thumb microprocessor provides enough processing power for our algorithms.

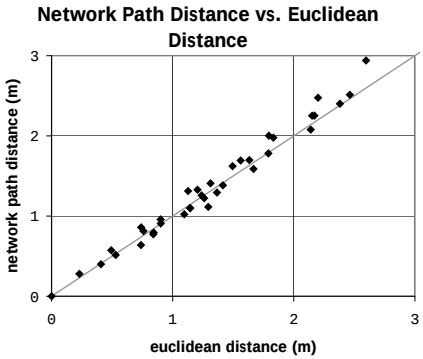

Fig. 5. Distance-to data plotting actual versus computed distances on 28 robots.

B. Mobile Robot Implementation

We implemented Protoswarm on a group of 40 autonomous mobile robots shown in Figure 4. Each "SwarmBot" is autonomous and is equipped with bump sensors, light sensors, and an infra-red inter-robot communication and localization system [14]. The inter-robot localization system enables each robot to determine the positions of its neighbors relative to its own local coordinate system. The infra-red communication system is used to maintain the neighborhood table.

V. Experiments

We tested `elect-leader`, `distance-to`, `cluster-to`, and `dilate` on the robots. Data was collected from the robots using a ceiling-mounted vision tracking system that recorded the positions of each robot over time. Telemetry from each robot was recorded to monitor each robot's internal state.

A. `elect-leader` and `distance-to`

The `distance-to` function measures the distance between any point in the medium space and a source region. We ran the following code:

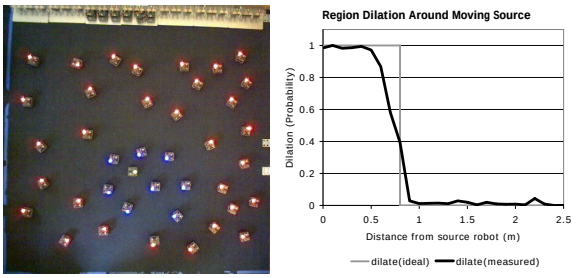

Fig. 6. Dilate results on robots. The lefthand picture shows a swarm of 40 robots running dilate. The righthand graph shows the actual dilated region versus the ideal region.

```
(distance-to (elect-leader (id)))
```

which elects a robot to be the source region, and then measures the distance to that robot from all other robots. Figure 5 compares the estimated distance to the source region to the actual distance. The distance estimate is accurate over the entire workspace. Because the paths for messages are constrained to only travel over the communication graph, the distance estimate will be an overestimate of the actual path. The longest path through the network was four communications hops.

B. `dilate`

The `dilate` function uses the `distance-to` function to defined a region around a source. We tested the following code:

```
(dilate (elect-leader (id)) 0.8)
```

to produce a region of 0.8 meters around the leader. The picture in Figure 6 shows a snapshot of `dilate` running on robots. The graph shows the extent of the dilation region around the leader robot as it is driven around using radio control. The black line shows the probability of a neighboring robot considering itself part of the dilation region. The transition point is shifted to smaller radii because `distance-to` is an overestimate of the actual distance. We suspect that the slope of the transition is caused by the voids in the network and the convergence speed of `distance-to` relative to the speed of the robot.

C. `cluster-to`

The `cluster-to` produced a vector field which is used to drive regions towards source regions. We tested the following code:

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

which drives an "anti-leader" robot to a leader robot. The left plot in Figure 7 shows five paths

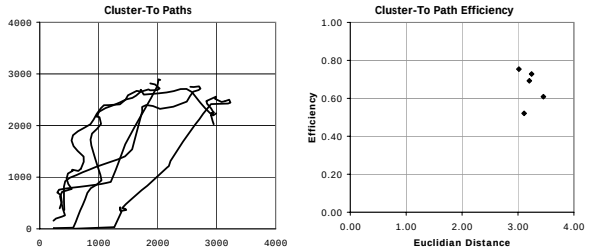

Fig. 7. Cluster-to results on robots. The lefthand plot shows five paths towards the source the bottom left, and the righthand plot shows the efficiency of these paths, where the efficiency is the ration of shortest possible path length to actual path length.

from various starting positions. The right plot shows the path efficiency for each of these paths, where path efficiency is the ratio of shortest possible path length to actual path length. In twenty runs, the robot always converged to the source.

VI. Challenges, Conclusions, and Future Work

In this paper, we introduce a continuous spatial computer abstraction to programming multi-agent behaviors. Our approach is built upon an Amorphous Medium Abstraction which frees the programmer from needing to consider individual robots. The Protoswarm language uses this abstraction to provide the user with a high-level programming model. We describe several core algorithms written in Protoswarm useful for constructing larger applications. We tested program fragments on 40 to 10000 in simulation and on a physical swarm of 40 robots. In all cases, the programs behaved as expected and the resulting behaviors were robust and scalable.

The power of our approach is that we can write scalable applications once and deploy them approximately on a number of multi-robot platforms with each platform incurring a certain approximation error. We think it is important to characterize this error, but at this time we are unable to make formal or statistical bounds on it or guarantees on correctness of high level programs. We have been working steadily on this challenge and hope to have results soon.

Although the demonstrated examples are limited, the programming model is a promising tool for multi-robot systems. In the future, we hope to expand the list of group level behaviors and applications and deploy the model on a wider range of multi-robot systems.

VII. Acknowledgments

The authors would like to thank all our sponsors. In particular, this work was funded under NSF grant CCF-0621897. We would like to acknowledge Jacob Beal's contribution to this work both in helping flush out the robotics approach and for his key role in the development of the Amorphous Medium Abstraction and the Protoswarm language.

References

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

[2] Jonathan Bachrach and Jacob Beal. Programming a sensor network as an amorphous medium. In *Distributed Computing in Sensor Systems (DCOSS) 2006 Poster*, June 2006.

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

[4] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.

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

[6] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[7] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

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

[9] E. Klavins. A language for modeling and programming cooperative control systems. In *Proceedings of the International Conference on Robotics and Automation*, 2004.

[10] M. Kloetzer and C. Belta. Hierarchical abstractions for robotic swarms. In *IEEE International Conference on Robotics and Automation*, 2006.

[11] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acqusitional query processing system for sensor networks. In *ACM TODS*, 2005.

[12] N. Margolus. CAM-8: A computer architecture based on cellular automata. In *Pattern Formation and Lattice-Gas Automata*, 1993.

[13] M. Mataric and M. Marjanovic. 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)*, pages 698–707, May 1993.

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

[15] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system, a toolkit for building multi-agent simulations. Technical Report Working Paper 96-06-042, Santa Fe Institute, 1996.

[16] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[17] J. Palmer and Jr. G.L. Steele. Connection machine model cm-5 system overview. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[18] W. Spears, D. Spears, J. Hamann, and R. Heil. Distributed, physics-based control of swarms of vehicles. *Autonomous Robots*, 17(2-3), August 2004.