Dynamically Defined Processes for Spatial Computers

Jacob Beal
BBN Technologies
Cambridge, MA 02138
Email: jakebeal@bbn.com

***Abstract*—A program executing on a spatial computer must be able to react to changes in its environment. For example, a sensor network tracking flocks of birds needs to be able to create a spatially-extended tracking process for each flock that it detects, and these processes should not interfere with one another. When dynamically defined processes are identified using fixed equivalence classes (e.g. UIDs), however, the *independent creation dilemma* means that no algorithm can safely create processes in less than $O(diameter/c)$ time, where c is the speed of information propagating through the network. This dilemma can be evaded by defining the extent of a process with a comparator that does not form an equivalence class. I an example of the dilemma and its resolution using the example of a sensor network tracking flocks of birds, as well as proposing an extension of the Proto spatial computing language[1] to handle dynamically defined processes.**

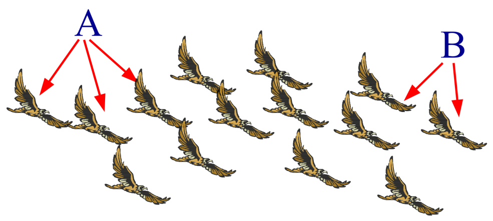
I. Introduction

An increasing number of systems being designed or deployed may be viewed as *spatial computers*—potentially large collections of devices distributed to fill some space, such that the difficulty of moving information between devices is strongly dependent on the distance separating them. Examples of spatial computers include sensor networks, robotic swarms, colonies of engineered bacteria, and peer-to-peer wireless networks.

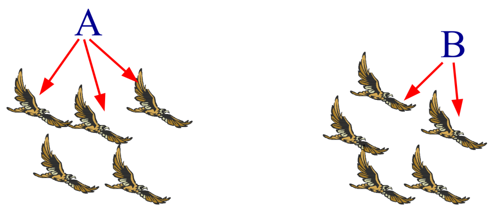
The program on a spatial computer is often intimately tied to its environment. For example, a sensor network might be emplaced to observe the nesting grounds of a rare species of bird, or a robotic swarm dispatched to search for injured people in a disaster area.

In many situations, a programmer would naturally wish for the program to react to changes in the environment by dynamically defining new state elements, such as objects or processes. For example, when the sensor network spots a flock of birds taking off, it might create a data object representing the flock and spawn a process that follows the flock around, recording observations. Likewise, when the robotic swarm spots an injured person, it might create a report of the person's location and condition and send this report in search of a nearby available rescue team.

If the program were executing on a centralized system, dynamically defining state in cases like these would be trivial. When a program is executing distributed across the devices of a spatial computer, however, we are placed on the horns of a dilemma. This *independent creation dilemma* means that when state elements are identified conventionally, any algorithm that can dynamically define state elements in less than $O(diameter/c)$ time must risk either creating duplicate state elements or creating state elements that interfere with one another (Figure 1).



(a) Independent detection of the same flock



(b) Observation of two different flocks

Fig. 1. An example of the *independent creation dilemma*: in a sensor network, device A and device B each observe part of a flock of birds and immediately create a tracking process that spreads out in space to follow the flock. If A and B are observing the same flock (a), then when the processes they launched encounter one another, they should merge. If, however, A and B are observing different flocks (b) the processes should stay separate. When there is space-like separation between the creation of the processes, it is not possible to determine whether the processes should merge at the time of their creation.

I first lay out the abstractions to be used in this discussion, then define the independent creation dilemma and prove that the dilemma applies whenever state element identifiers form an equivalence class. I then give examples of how more general tests for process membership can be used to resolve the dilemma in the case of tracking flocks of birds, as well as proposing an extension of the Proto spatial computing language[1] to handle dynamically defined processes.

A. Related Work

Processes that are created, distributed, and destroyed dynamically are fundamental to the family of viral approaches to spatial computing. Notable such approaches include TOTA[2], paintable computing[3], and Smart Messages[4]. In all of these approaches, however, the programmer acts at the level of the individual device and must build their own approach to the independent creation dilemma from scratch.

Other spatial languages, such as the current version of Proto[1], [5] and pattern languages like Growing Point Language[6] and Origami Shape Language[7], generally avoid the dilemma by requiring all processes to be identified at compile time. The logical language Meld[8] is an exception, in that the logical assertions it produces could be used to produce dynamically defined processes, but has not been designed to support this and suffers from the same limitations as the viral approaches.

Data aggregation in sensor networks is a highly restricted version of the problem, where there is generally one destination and the question is when and how to merge independently created datums moving toward that destination. Data flows and decisions on when to aggregate are often dynamically determined (see for example, greedy incremental trees[9] or the structure-free approach in [10]), but there is generally little dynamism in where the program gathers what kinds of data, except via the intervention of a human user.

Finally, the distributed theory community has dealt with related problems on general networks, particularly in the case of IP networks, overlay networks, and relatively small mobile networks. The relatively low diameter of such networks, however, has favored the development of consensus-based algorithms such as RAMBO[11] and Virtual Mobile Nodes[12], that avoid the problem by letting the network act as though it were a single device.

II. Processes on a Spatial Computer

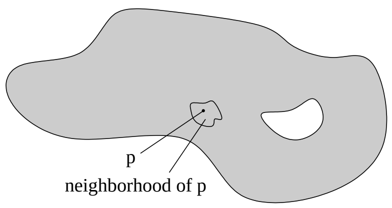
In order to have a clear discussion of dynamic state on a spatial computer, we first need a clear definition of spatially-extended state. Using the amorphous medium abstraction of a spatial computer, I define spatially-extended processes in terms of the sets of communicating local program instances. Any other spatially-extended state element may then be defined in terms of such a process: for example, a data object may be equivalently expressed as a process that holds a piece of state, and operations on the object may be expressed as the response of the process to a stream of "operate" messages.

A. Amorphous Medium Abstraction

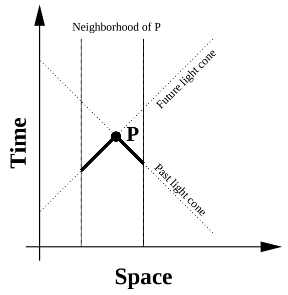
My definition of spatially-extended processes uses the *amorphous medium* abstraction of a spatial computer[13]. This continuous space abstraction is useful because many different varieties of spatial computer can all be viewed as discrete approximations of an amorphous medium.

An amorphous medium is a manifold M with an independent computing device at every point $m \in M$ in the manifold (Figure 2(a)). For simplicity, we assume this manifold is compact and Riemannian, though these conditions may be stronger than necessary.

Each point m in the manifold is associated with a neighborhood $N(m) \subseteq M$, which contains at least some $\epsilon-$ball around m. That is, given m we can choose some small number ϵ such that every point within ϵ of m is in the neighborhood $N(m)$.



(a) Amorphous Medium



(b) Information Propagation

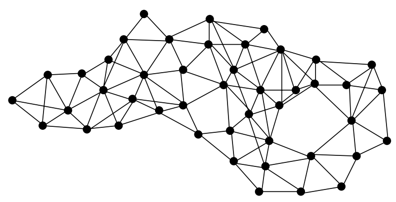

(c) Discrete Network

Fig. 2. An amorphous medium is a manifold where every point is a computational device that shares state with a neighborhood of other nearby devices. Information propagates through an amorphous medium at a fixed rate, so each device has access only to values in the intersection of its neighborhood and past light cone. An amorphous medium may be approximated by a mesh-like discrete network.

For simplicity, let us also assume that every neighborhood is also connected and compact, though again these conditions may be stronger than necessary. Information flows through the amorphous medium at a fixed velocity c, and each point m has access to the most recent information that has propagated to it from its neighbors.

The choice of c to indicate the speed of information is a deliberate echo of relativistic notation. Adding a time dimension, we may consider a space-time volume $M \times T$, where T is a time interval. The space-time interval between any two points (m, t) and (m', t') in this volume is defined as $s^2 = c^2(t - t')^2 - d(m, m')^2$. When $s^2 < 0$, the interval is space-like, meaning that events at (m, t) cannot influence events at (m', t'). When the interval is light-like ($s^2 = 0$) or time-like ($s^2 > 0$), then the earlier event can influence the later event. The set of accessible neighbor values may thus also be defined as those lying in the intersection of the neighborhood $N(m)$ and the past light cone of m at time t (Figure 2(b)).

A program p for an amorphous medium may thus be defined, without reference to any particular amorphous medium M, as a rule for evolving a value at a generic point m with respect to the most recently arrived information from its neighborhood $N(m)$, beginning at an initial time t_0. When p is computed on M, the value $p|M(m, t)$ of every point m evolves as specified by p.

Another way to look at computation is to make the dimension of time explicit. A valid computation of p on M for the time interval T is thus any assignment of state to the space-time volume $M \times T$ that is consistent with the evolution specified by p.

There are other, more complex formulations of the amor-

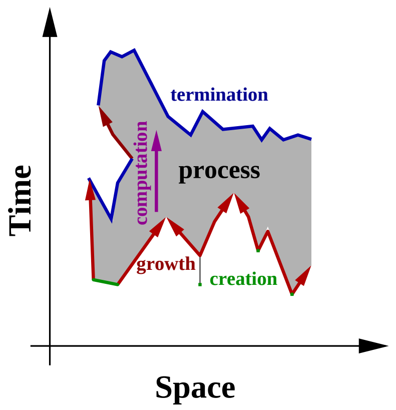


Fig. 3. Space-time anatomy of a process: a process (grey region) is initially created at one or more locations (green minima and space-like bottom edge), then may grow along time-like trajectories (red edges). Computation at a device takes place along a vertical line (e.g. purple arrow) and at any point can potentially be influenced by any values in the process with past time-like separation. The process can terminate independently at any point (blue edge).

phous medium that consider moving devices, a changing volume of space, or information that flows at a variable rate bounded by minimum c^- and maximum c^+. We will not consider these more complex cases in this discussion, but expect that it is possible to extend the results herein.

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 (Figure 2(c)). 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 [14] and [15].

B. Process Definition

On a single computing device, a process is an instance of a program being executed. If we consider that device as a single point m in an amorphous medium, then a program instance p on m is a single-device process spanning the sub-space $m \times [t_0, t_\Omega]$, where the program instance is created on m at time t_0 and terminates at time t_Ω.

The extent of a process P that spans many devices may then be defined in terms of information flow: if a device $m \in M$ is running a program instance p at time t, then a program instance p' on its neighbor $m' \in N(m)$ is part of the same process if and only if p can incorporate state from p' at time $t - d(m, m')/c$ into its computation.

Note that since we assume that m' is a neighbor of m, there is no physical obstacle to p and p' being part of the same process—it is only a question of which local program instances are specified to share state and which program instances are not. Note that the set of devices over which a process P extends need not be connected at any given point in time—it is enough that the program instances have the potential to communicate at some time in the past or future.

What do we need to know in order to describe the behavior of a process? Consider a collection of program instances across space and time that form a process P. Figure 3 shows an example process as a region of space-time. The points in this region can be categorized into four classes:

- The process is *created* independently on individual devices by other programs. Creation occurs at all minima of the region and any portion where the minimum-time edge is space-like.
- The process *grows* outward from devices where it was created to their neighbors along light-like or time-like trajectories.
- The process is *terminated* on devices independently, forming the maximum-time edges of the region.
- In the interior, the process *evolves* with respect to a point (m, t) and the state of neighboring program instances of P in its past light cone.

A process P may thus be fully specified by five values: conditions for creation, growth, and termination, a function specifying evolution, and a test for whether two neighboring program instances are part of the same process.

It is tempting to strengthen this definition slightly, such that the set of program instances in a process P form an equivalence class. In this case, given any chain of program instances that share state, $p_0, p_1, ..., p_k$, we may assume that if p_0 and p_k are on neighboring devices, that they will communicate. This stronger definition is also equivalent to identifying a process by a unique identifier and determining if program instances can communicate by comparing their UIDs. As we will see, however, this strengthening of the definition leads to problems when it is not known at the time of creation whether information should be able to flow between two program instances.

III. Independent Creation Dilemma

The *independent creation dilemma* is a problem of identity that tends to emerge whenever we have many devices acting independently. Let us explain it by means of an illustrative example.

Assume we have a sensor network emplaced to observe the nesting grounds of a rare species of bird. Each time a flock of birds takes off and begins to fly around the area, we want to launch a distributed process that follows along with the flock and tracks its motion (Figure 1).

When a number of birds take off and begin to flock, they may be observed by many devices at once. This should result in one tracking process, not a vast number of duplicates. On the other hand, two different groups of birds might take off and begin to flock in different places at approximately the same time. This should result in two different tracking processes that will not merge or interfere with one another with one another (assuming communication, memory, and processing resources are not scarce) if the flocks later cross paths but do not combine into one larger flock.

The straightforward way to ensure that the local program instances form the desired processes is to associate each process with an equivalence class, such as a unique identifier (UID). Every program instance in a process then holds a copy of the unique identifier for the process. When a device sends a message about a flock tracking process to a neighbor, including the UID for that flock, the neighbor can compare UIDs with the program instances already executing on it to determine whether to route the incoming message to one of its current program instances or to treat the message as part of a novel, as yet unseen process.

On a single-processor system, UIDs are usually assigned explicitly to processes in the form of a numeric identifier and implicitly to objects as the address of the object in memory. Conventional multi-processor systems operate essentially the same way, as do most distributed systems, though the unique identifiers (e.g. IP addresses, URIs, email addresses) often have meaningful substructure. The rules for comparing them, however, remain essentially the same.

On a spatial computer, however, we cannot assign UIDs to processes appropriately unless we make process creation very slow. At the time when a device observes a flock of birds taking off and the flock extends to the edge of its sensor range, it cannot know whether there are other devices far away observing the same flock, because it cannot directly observe the extent of the flock and there has not been enough time for messages to carry that information to it from elsewhere.

As a result, the device cannot distinguish between the case of one large flock and two disconnected flocks unless it waits for at least long enough for information to propagate across the diameter of the flock. While this may be a reasonable time to wait on some networks, it is not on a spatial computer, because the diameter of the spatial computer may be high due to the number of devices involved and the locality of communication and the flock may span an appreciable fraction of the diameter.

We are thus faced with a dilemma caused by the fact that some devices must decide independently on the UID for a flock process: either creating a process takes a potentially long time, or the system must cope with processes having different UIDs when they should have the same UIDs, or else cope with processes having the same UID when they should have different UIDs.

Let us generalize this with a definition of the *safe creation* of a process identified by an equivalence class $\sim$. For such a process to be safely created, every independent creation of a program instance in the process must choose the same equivalence class $\sim$ and no program instance of a different process can choose the same equivalence class. If the first condition is violated, then processes can be duplicated. If the second condition is violated, then processes can interfere.

Given this definition, we can prove that it is in general impossible to both quickly and safely create a process identified by an equivalence class:

Theorem 3.1: If the program instances of each process P form an equivalence class $\sim$, no algorithm for creating program instances exists that can guarantee safe creation of a process in less than $O(diameter/c)$ time.

Proof: Assume we have an algorithm that is guaranteed to safely create a program instance in less than $O(diameter/c)$ time. Since it executes in less than $O(diameter/c)$ time, then for any amorphous medium M, it must be possible to choose two points (m,t) and (m',t') in the space-time volume $M \times T$ such that executions of the algorithm beginning at these points have space-like separation. Now consider the algorithm executed at these two points in one of three conditions, either both creating the program instances in process P, both creating program instances in process P', or (m,t) creating an instance of P and (m',t') creating an instance of P'.

When the algorithm begins to run at a point, its choice of equivalence class can only be influenced by points in $M \times T$ that are causally related to the execution of the algorithm and that causally relate to some point in the execution of the algorithm. Because (m,t) and (m',t') have space-like separation, the regions of space-time that influence the choice of equivalence class cannot overlap. Thus it is not possible for both (m,t) to distinguish between the all-P and the mixed case and for (m',t') to distinguish between the all-P' case and the mixed case. Thus, by symmetry, the algorithm must fail in at least one of the three cases. ■

Thus we see that it is not possible to create processes both safely and efficiently when the membership of program instances in the same process is determined using fixed unique identifiers or other equivalence-class based systems.

Note that even $O(diameter/c)$ is a highly optimistic time bound: on any spatial computer with non-trivial communication or device failures, choosing an equivalence class becomes a consensus problem, subject to the wide variety of impossibility results and poor time bounds for distributed consensus algorithms.

IV. Specifying Dynamic Processes

Processes can avoid the independent creation dilemma by avoiding or deferring commitment to a particular identifier. There are many different ways in which this might be done, and which of the many ways is appropriate depends on the particulars of the program. Let us now illustrate how a programmer may handle the independent creation dilemma using two examples in our bird monitoring domain: tracking a flock of birds and reporting the movements of flocks to a base station.

These examples as expressed using an extension I propose for Proto. Proto[1], [5] is a LISP-like functional language for programming spatial computers based on the amorphous medium. In Proto, the programmer specifies a computation globally using four families of space-time operations. The computation is then compiled to a local program suitable for execution on an amorphous medium, and may be approximated discretely on the devices of spatial computer.

A. The `procs` *Operator*

The current version of Proto requires that the complete structure of a program be determinable at compile-time. To allow for dynamically defined processes, I propose a new construct, `procs`, on which a new family of Proto operators dealing with dynamic state can begin to be constructed:

```
    (procs (elt sources)
        ((var init evolve) ...)
      (same? run? &optional terminate?)
      body-expr ...)
```

Each appearance of the `procs` construct in a Proto program implies a family of processes. Every device in the spatial computer stores a set of program instances, which it shares with its neighbors.

The first expression declares the `sources` of new program instances—a stream of sets of potential new program instances— and a variable `elt` for accessing values within any particular instance under consideration.

The second expression, a set of `(var init evolve)` declarations, creates and maintains state for a program instance (equivalent to a standard Proto `letfed`): state variable `var` is initialized to the value of the `init` expression and evolves forward in arbitrarily small steps of time `dt` according to the expression `evolve`. Any state variable can appear in `evolve`, where it receives its prior value. Other state can be established within the `body` of the process, but these state variables are lexically scoped to be usable in determining which program instances run where, while state within the `body` expression is not.

The `same?`, `run?`, and optional `terminate?` expressions are used to manage the extent of processes. If `terminate?` is not supplied, then it is assumed to be `(not run?)`. Within these expressions, the `nbr` operator pulls values from all other program instances in the neighborhood, including other instances at the same device. At each instant of execution, a device updates its set of program instances as follows:

1) A potential new program instance is created for each element of `sources`, and its tentative state is set by `init` values for the state variable `vars`. If the new instance is not `same?` as any existing local program instance, and if the `run?` expression returns true, it is added to the set of local program instances.
2) Each program instance on a neighbor is compared with the local set of program instances. If it is not `same?` as any local instances, then a potential new program instance is created. If the `run?` expression returns true, it is added to the set of local program instances. Otherwise, it is discarded.
3) For each local program instance, the `terminate?` expression is evaluated. If it returns true, the program instance is removed from the set of local program instances.
4) Finally, the `evolve` and `body` expressions are evaluated for each program instance in the updated local set of program instances. The values produced by the body evaluations are collected into a set, which is the value produced by the whole `procs` expression.

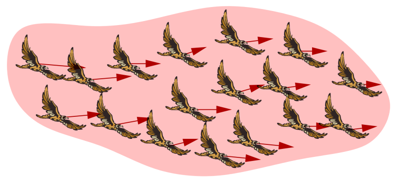


Fig. 4. The extent of a process tracking a flock is defined to be the set of birds moving in approximately the same direction.

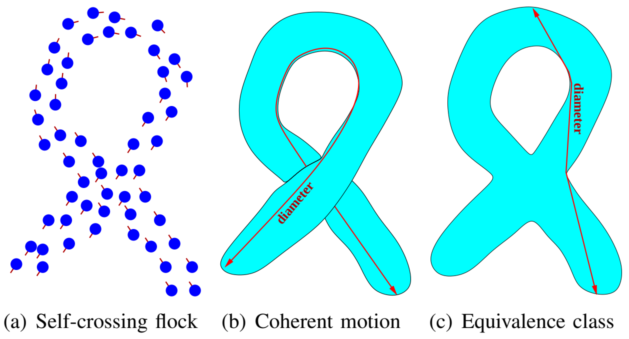


Fig. 5. Defining a flock in terms of coherent bird motion means that the extent of the tracking process matches closely with our intuitions about flocks even in extreme circumstances, such as when the leading portion of the flock crosses over the trailing portion of the flock (a). The coherent motion definition leads to a process that overlaps itself (b), while an equivalence class definition leads to the head and tail portions of the flock merging (c), giving the tracking process a difference shape than the flock being tracked. The difference between these two cases is highlighted by indicating the diameter of the process with a red arrow.

B. Example: Tracking a Flock

A flock of birds is a coherent group of birds moving together in approximately the same direction and velocity. We can use this definition of a flock to define the extent of a process that tracks flocks:

```
(def close-vec (base other err)
  (< (len (- base other)) (* err (len base))))

(def track-flocks ()
  (procs (bird-vec bird-vecs)
      ((flock-vec
        bird-vec
        (average (filter
                   (lambda (v) (close-vec flock-vec v 0.1))
                  bird-vecs))))
    ((close-vec flock-vec (nbr flock-vec) 0.1)
     (find-if (lambda (v) (close-vec flock-vec v 0.1))
              bird-vecs))
    (measure-shape)))
```

where `bird-vecs` is a sensor field that returns the velocities of nearby birds observed from this device and `measure-shape` is some function for estimating the size and shape of a flock.

Here, the extent of a flock process is defined in terms of the average motion of birds in the flock. A device attempts to launch a program instance for each of the birds it sees. These combine together to form one program instance for each collection of birds moving within 10% of the same vector, which run as long as similarly moving birds are observed. A tracking process for a flock thus extends over any collection of birds where birds are locally moving in approximately the same direction (Figure 4).

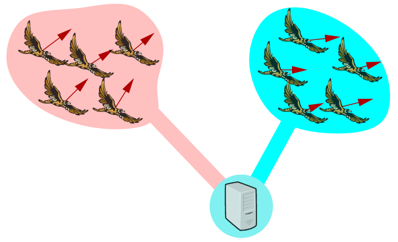


Fig. 6. When reporting tracking data from a flock to a base station, the flock must be given an identifier so that its tracking data can be distinguished from that of other flocks.

Defining a flock in terms of coherent bird motion means that the extent of the tracking process matches closely with our intuitions about flocks even in extreme circumstances. For example, if the leading portion of a flock loops back and crosses over its tail (Figure 5(a)), the definition from coherent motion means that the head and tail cross over one another without interacting, putting two program instances from the same process at the intersection (Figure 5(b)). If processes were defined using an equivalence class like a UID, on the other hand, the head and tail would merge, giving the tracking process a different shape than the flock being tracked (Figure 5(c)). Likewise, if the flock splits in two or if two separate flocks come together and merge, the tracking process will do so as well.

C. Example: Reporting on a Flock

In the example of tracking a flock, the identity of the flock process is entirely implicit and locally defined. If we wish to report the tracking data gathered from a flock to some base station, on the other hand, we need to give an identifier to a whole flock so that devices that are far away from the flock can distinguish between it and other far away flocks.

This does not invoke the independent creation dilemma, however, since the identifier need not be available until reporting data from the flock is also available. Assuming we have a self-stabilizing algorithm for generating a probably-unique identifier (meaning that if the flock splits or merges, the UID will eventually adjust appropriately), we can run this algorithm in each flock process and produce a tuple of this unique identifier and the tracking data for the flock.

Tracking data can then be relayed back to a base station by a program that uses the unique identifier to distinguish between streams of data:

```
(report-data-stream (data-set base)
  (procs (data data-set)
    ((uid (1st data) uid)
     (src true (find uid (map 1st data-set)) diameter))
    ((= uid (nbr uid))
     (dilate src diameter))
    (channelcast
     src base 2
     (2nd (find uid data-set :key 1st)))))
```

Here each flock becomes a region where `src` is true. The process then spreads throughout the space to find a `base` to report to, and relays the data to it via a spatial channel 2 meters wide. The `dilate` expression restricts the process to run only so long as it is within `diameter` units of the source: when a flock disappears, the estimated distance of other locations to the source rises steadily and this expression thus assures that the reporting process will be eliminated within $O(diameter/c)$ of the time when the flock disappears. This function can thus be used to send non-interfering reports of flocks back to a base station.

V. Conclusion

Dynamically defined processes on a spatial computer cannot be safely and quickly created when their identity is determined using fixed equivalence classes such as UIDs. Processes can avoid this problem, however, by avoiding or deferring commitment to a particular identifier. I have illustrated this an example of tracking a flock of birds, and proposed an extension of the Proto spatial computing language to support dynamically defined processes. Future work includes integration of the `procs` command with the Proto compiler and virtual machine. Looking farther ahead, the `procs` command may serve as the foundation for a family of new dynamic state operations in Proto and, more generally, a new synthesis of techniques for understanding and implementing dynamic processes on spatial computers.

References

[1] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, pp. 10–19, March/April 2006.

[2] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: the TOTA approach," *ACM Transactions on Software Engineering and Methodology*, 2008.

[3] W. Butera, "Programming a paintable computer," Ph.D. dissertation, MIT, 2002.

[4] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode, "Spatial programming using smart messages: Design and implementation," in *IEEE International Conference on Distributed Computing Systems*, 2004.

[5] J. Bachrach, J. Beal, and T. Fujiwara, "Continuous space-time semantics allow adaptive program execution," in *IEEE SASO 2007*, July 2007.

[6] D. Coore, "Botanical computing: A developmental approach to generating inter connect topologies on an amorphous computer," Ph.D. dissertation, MIT, 1999.

[7] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, 2001.

[8] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, 2007.

[9] C. Intanagonwiwat, D. Estrin, and R. Goviindan, "Impact of network density on data aggregation in wireless sensor networks," University of Southern California, Tech. Rep. 01-750, November 2001.

[10] K.-W. Kai-Wei Fan, S. Liu, and P. Sinha, "Structure-free data aggregation in sensor networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 8, pp. 929–942, August 2007.

[11] N. Lynch and A. Shvartsman., "RAMBO: A reconfigurable atomic memory service for dynamic networks," in *DISC*, 2002, pp. 173–190.

[12] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch, "Virtual mobile nodes for mobile ad hoc networks," in *DISC*, 2004.

[13] J. Beal, "Programming an amorphous computational medium," in *Unconventional Programming Paradigms International Workshop*, September 2004.

[14] J. Bachrach, J. Beal, J. Horowitz, and D. Qumsiyeh, "Empirical characterization of discretization error in gradient-based algorithms," in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) 2008*, October 2008.

[15] J. Beal, J. Bachrach, and M. Tobenkin, "Constraint and restoring force," MIT CSAIL, Tech. Rep. MIT-CSAIL-TR-2007-044, August 2007.