

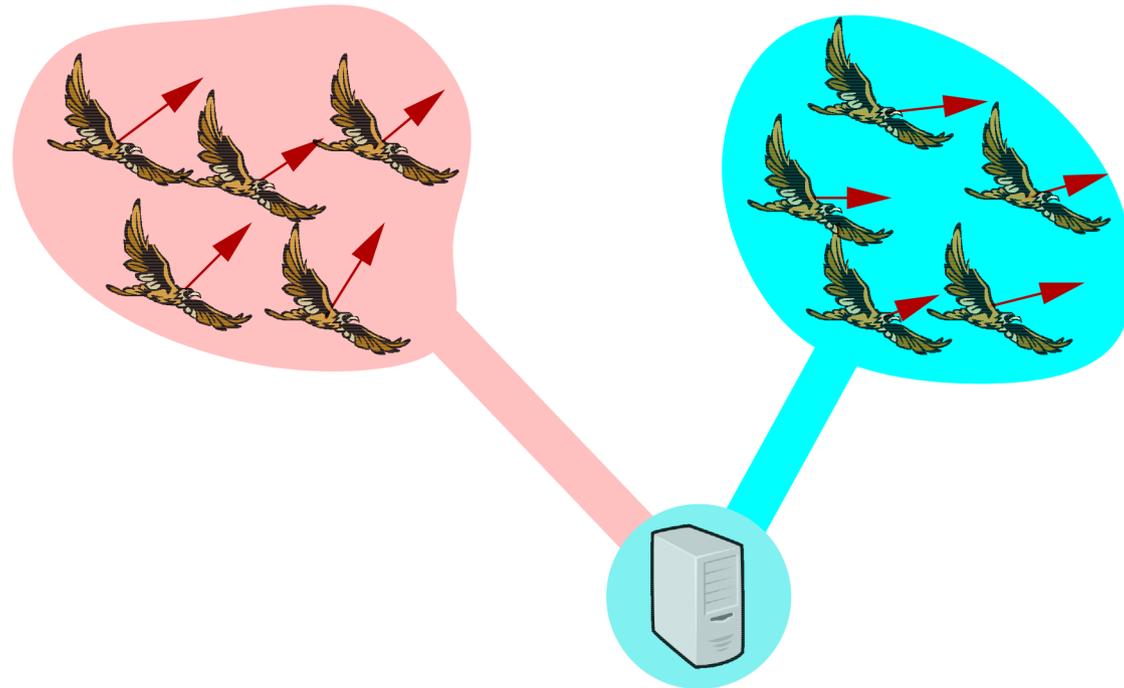
Dynamically Defined Processes for Spatial Computers

Jacob Beal

Spatial Computing Workshop 2009

Dynamic Allocation of State

Many applications must create state (e.g. objects, processes) in response to their environment



Consider tracking flocks of birds...

Why is this hard?



Are the visible birds part of the same flock?



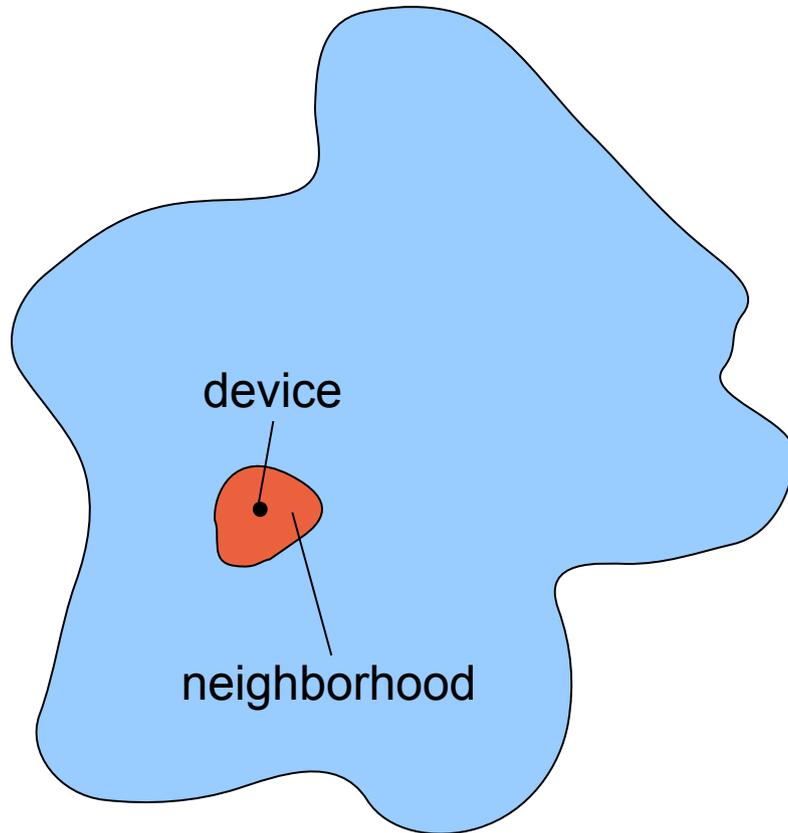
Outline

- Defining spatial processes
- Problem of independent creation
- Dynamically defining processes

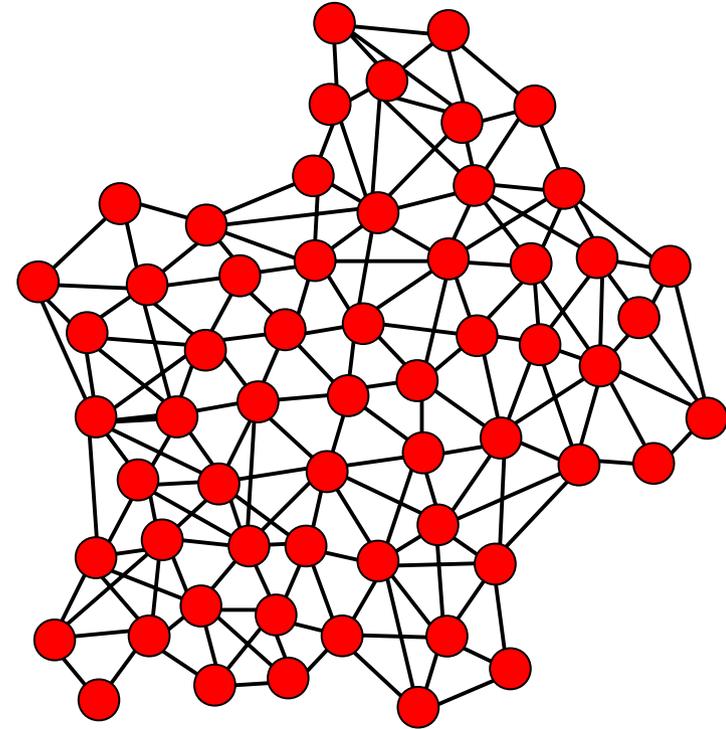
Related Work

- Viral Programming: dynamic but unconstrained
 - e.g. Paintable computing [Butera, '02], TOTA [Mamei & Zambonelli, '06]
- Distributed algorithms: safe but costly
 - e.g. Virtual Mobile Nodes [Dolev et al., '04]
- Data aggregation: highly specialized
 - e.g. greedy incremental trees [Intanagonwiwat, '01]
- Spatial languages: mostly compile-time
 - e.g. Proto [Beal & Bachrach, '06], Meld [Ashley-Rollman et al., '07], OSL [Nagpal, 01]

Spatial Focus: Amorphous Medium

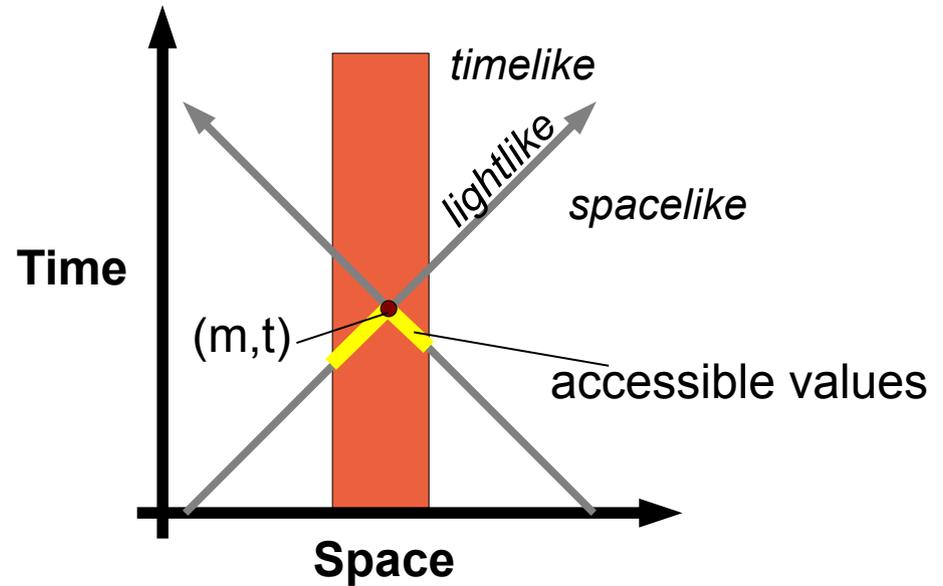
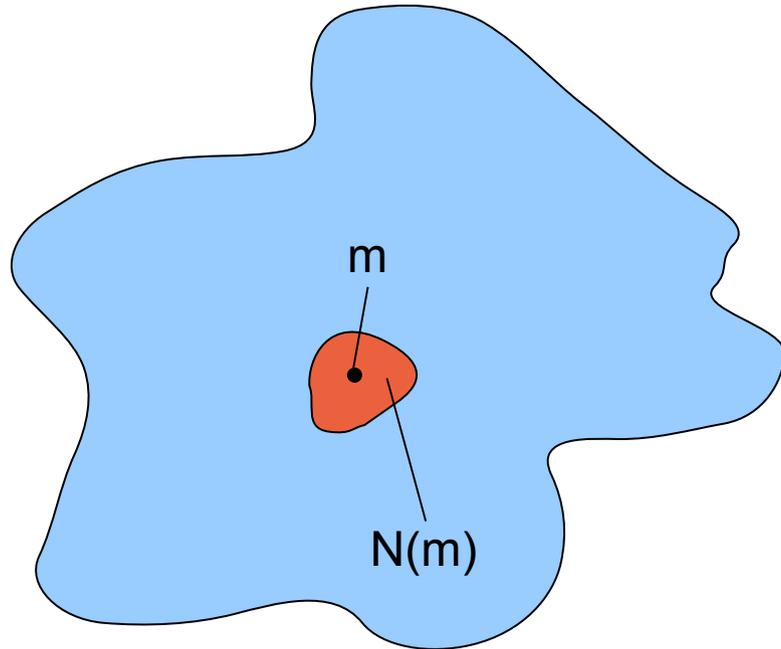


- Continuous space & time
- Infinite number of devices
- See neighbors' past state



- Approximate with:
- Discrete network of devices
 - Signals transmit state

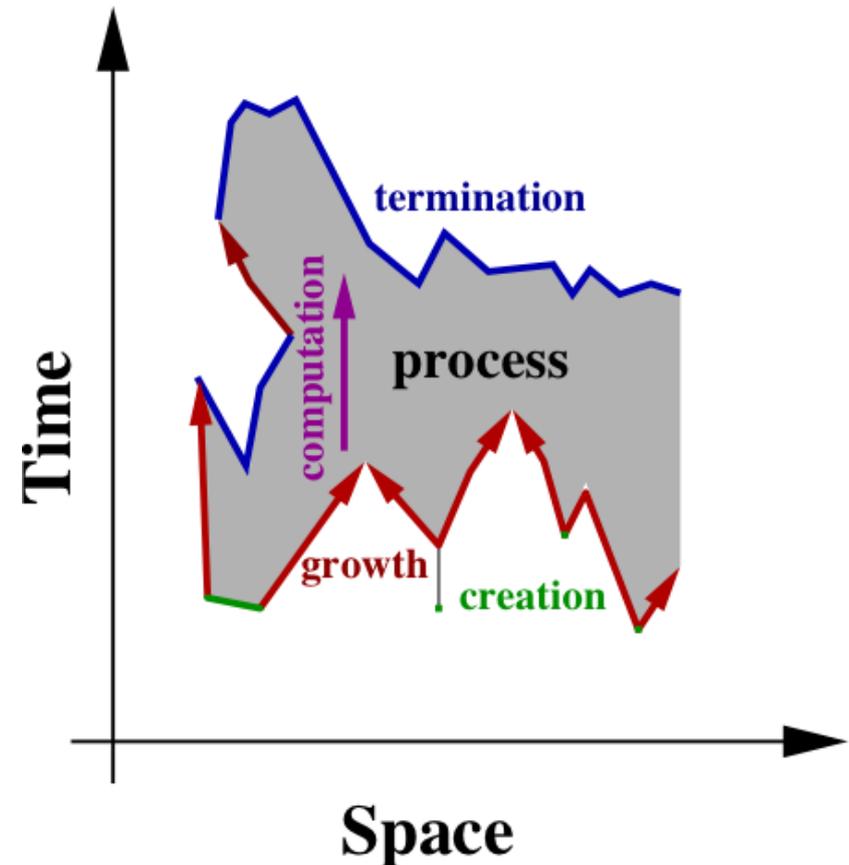
Amorphous Medium Definition (Simple)



- Compact, Riemannian manifold M , time interval T
- $N(m)$ contains ε -ball around m ; connected, compact
- Information flows at c
 - Interval between (m,t) and (m',t') : $s^2 = c^2(t-t')^2 - d(m,m')^2$

Definition of Process

- Let p be an executing instance of a program at a point m
- p' on $m' \in N(m)$ is in the same process if p can use state from p'
- Specifiable by 5 behaviors: creation, growth, sharing, computation, termination



Outline

- Defining spatial processes
- **Problem of independent creation**
- Dynamically defining processes

Problem of Independent Creation



Are the visible birds part of the same flock?



UIDs can't distinguish processes

Theorem: if instances of processes form an equivalence class \sim , no algorithm for creating program instances exists that can guarantee safe creation in less than $O(\text{diameter}/c)$ time

- Proof sketch:
 - Time bound \rightarrow space-like separation possible
 - choice of \sim only affected by causally related points
 - Algorithm must fail on one of:
 - m and m' create P
 - m and m' create P'
 - m creates P , m' creates P'

Outline

- Defining spatial processes
- Problem of independent creation
- **Dynamically defining processes**

Solution: dynamically determined extent

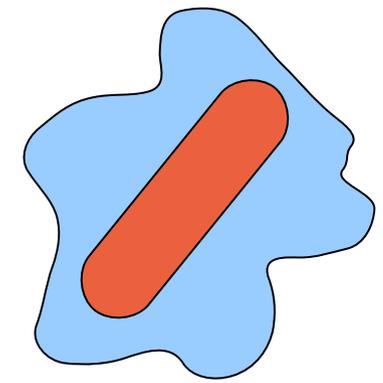
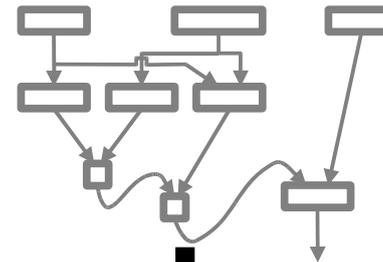
Instead of identifying processes with UIDs specify neighborhood flow directly.

Let's make this concrete...

Proto

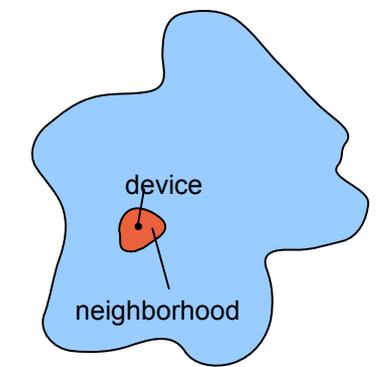
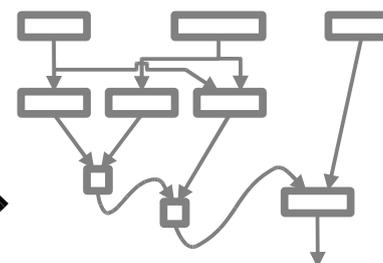
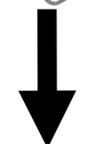
```
(def distance-to (src) ...)  
(def distance (src dst) ...)  
(def dilate (src n)  
  (<= (distance-to src) n))  
(def channel (src dst width)  
  (let* ((d (distance src dst))  
         (trail (<= (+ (distance-to src)  
                       (distance-to dst))  
                    d)))  
    (dilate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

evaluation →

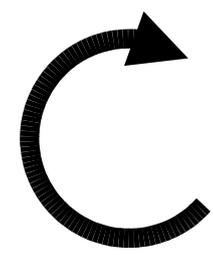


Global
Local
Discrete

global to local compilation



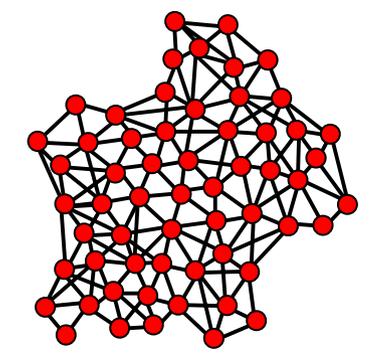
platform specificity & optimization



discrete approximation

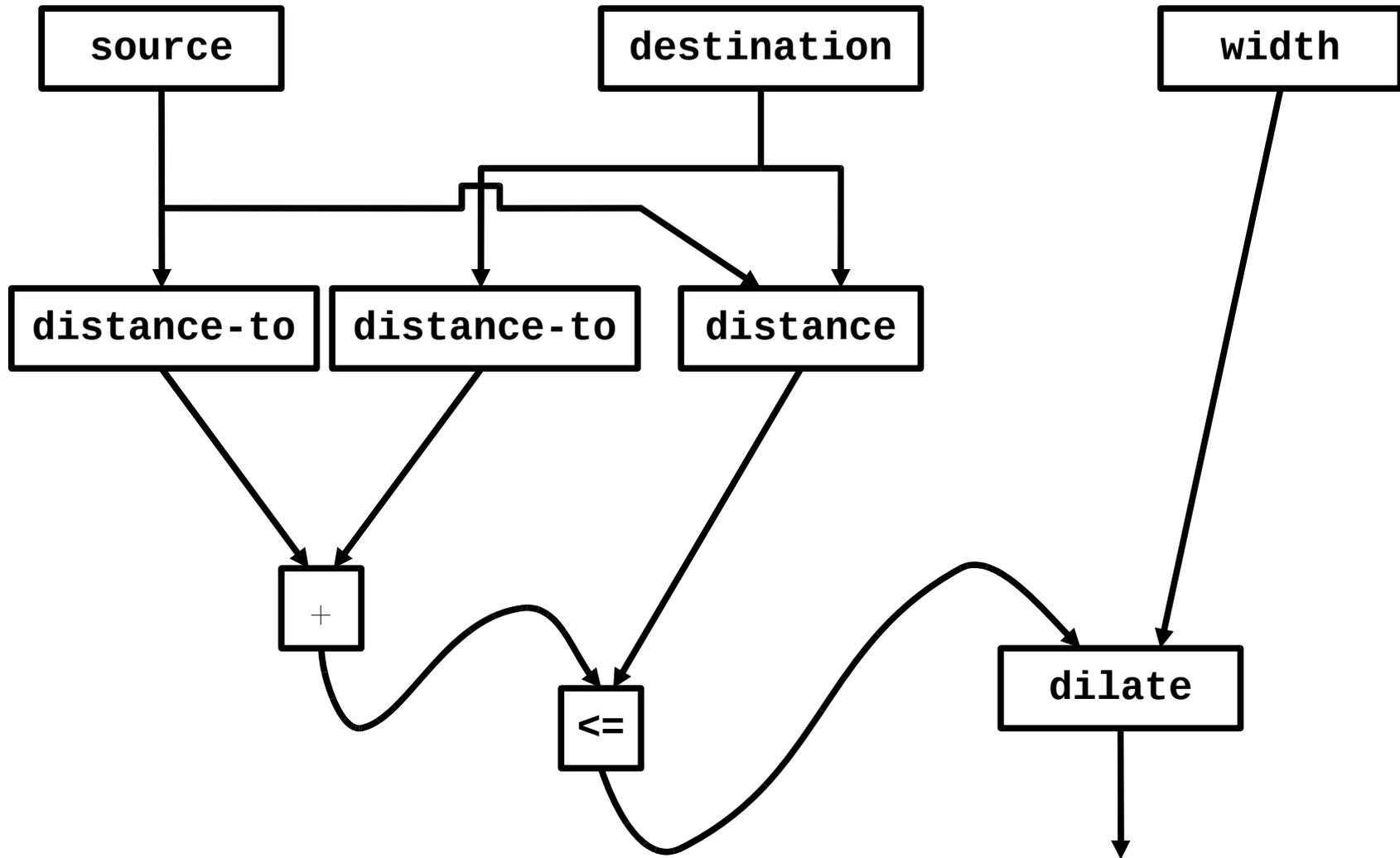


Device
Kernel

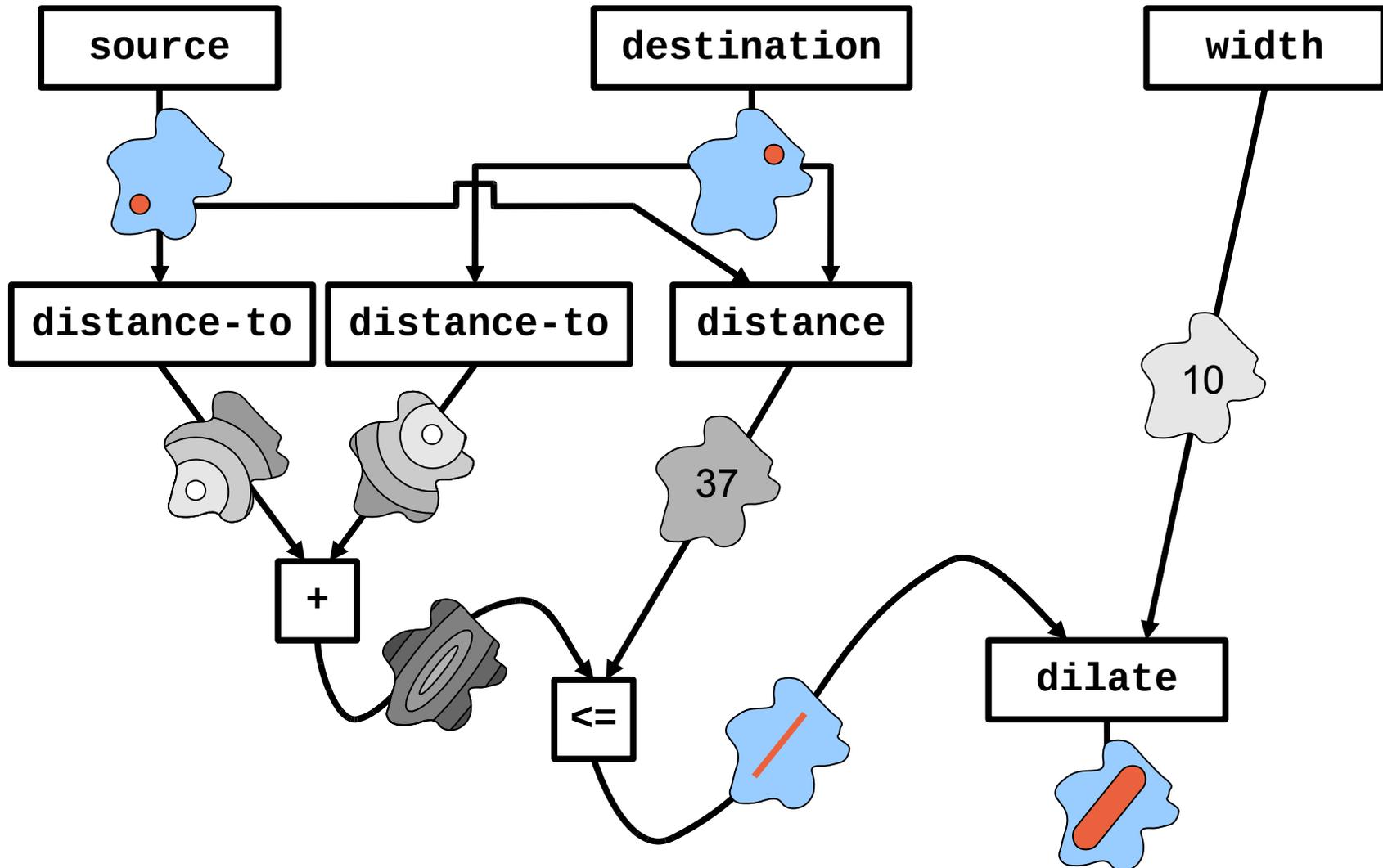


<http://stpg.csail.mit.edu/proto.html>

Computing with fields

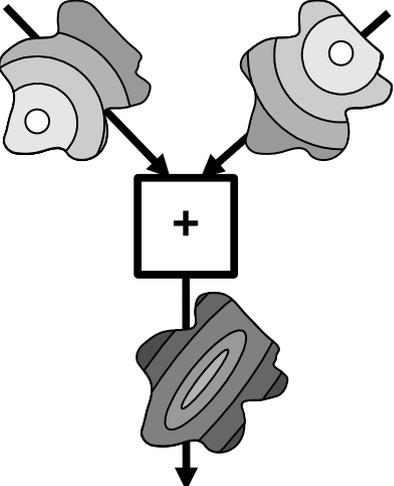


Computing with fields

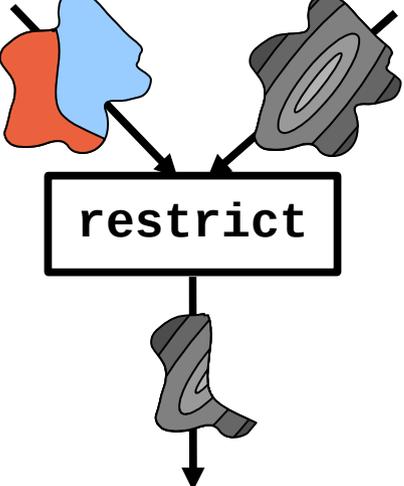


Four Families of Primitives

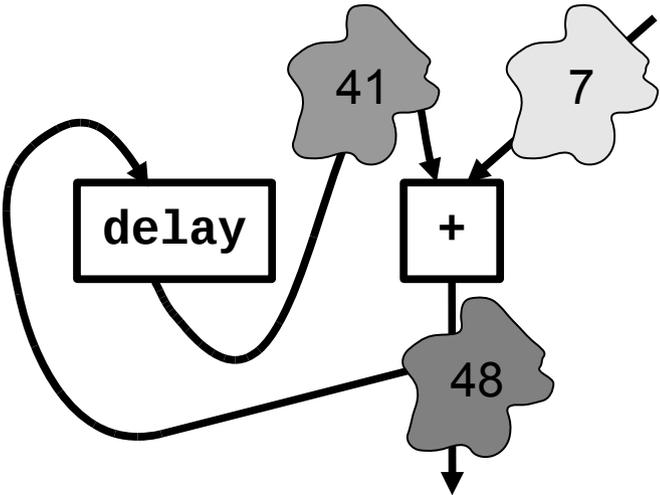
Pointwise



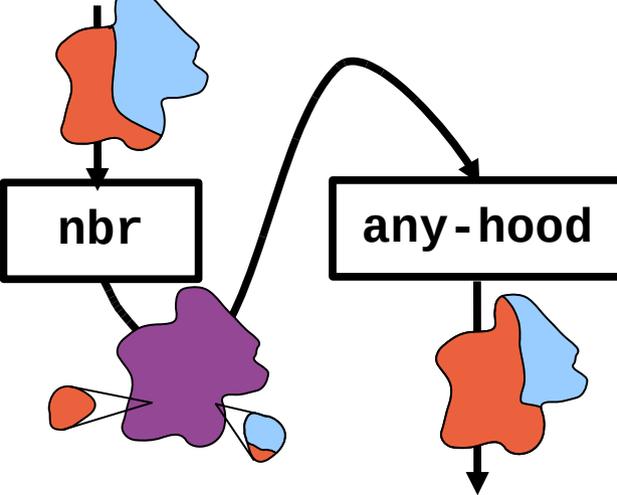
Restriction



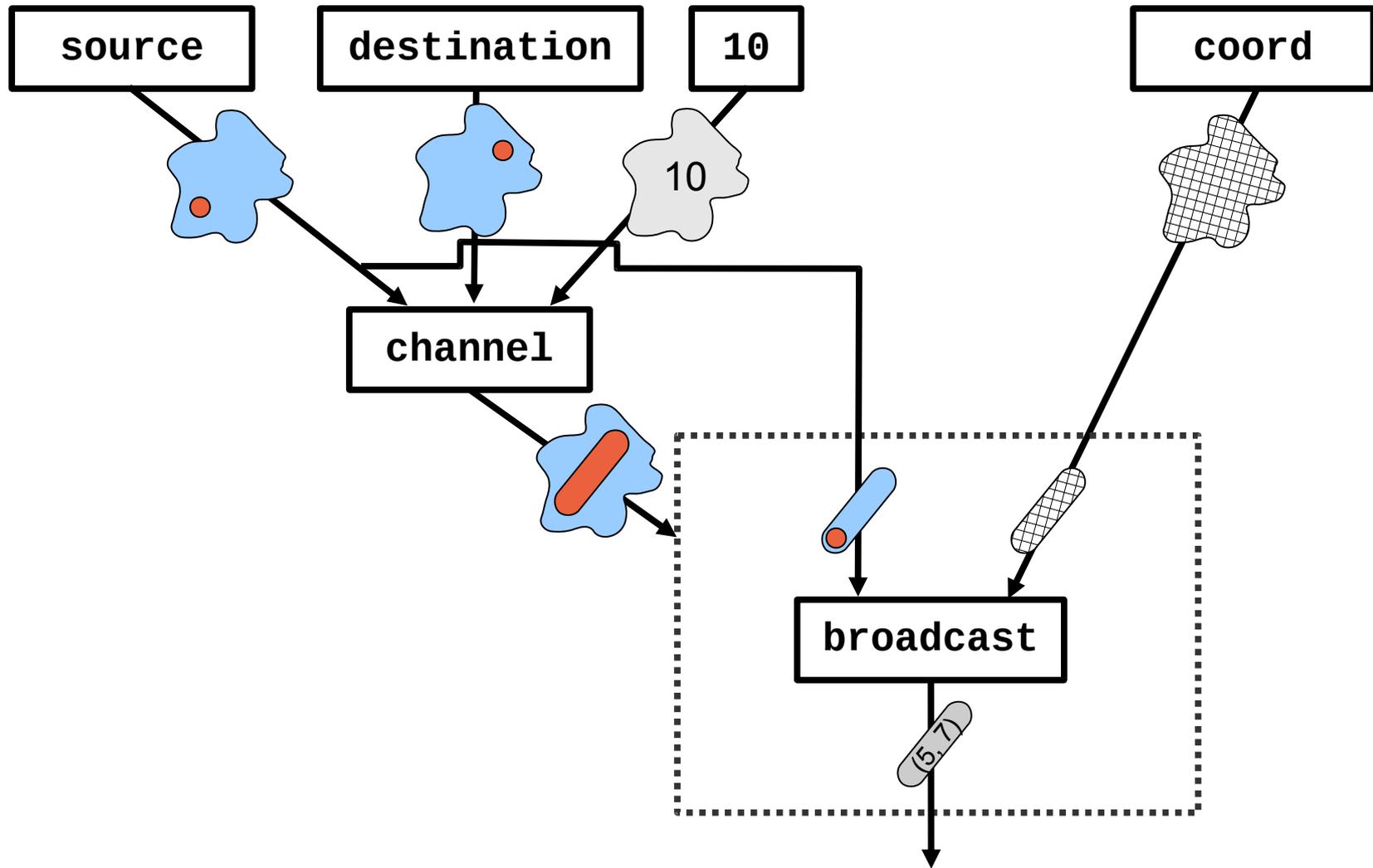
Feedback



Neighborhood



Branching = Restriction



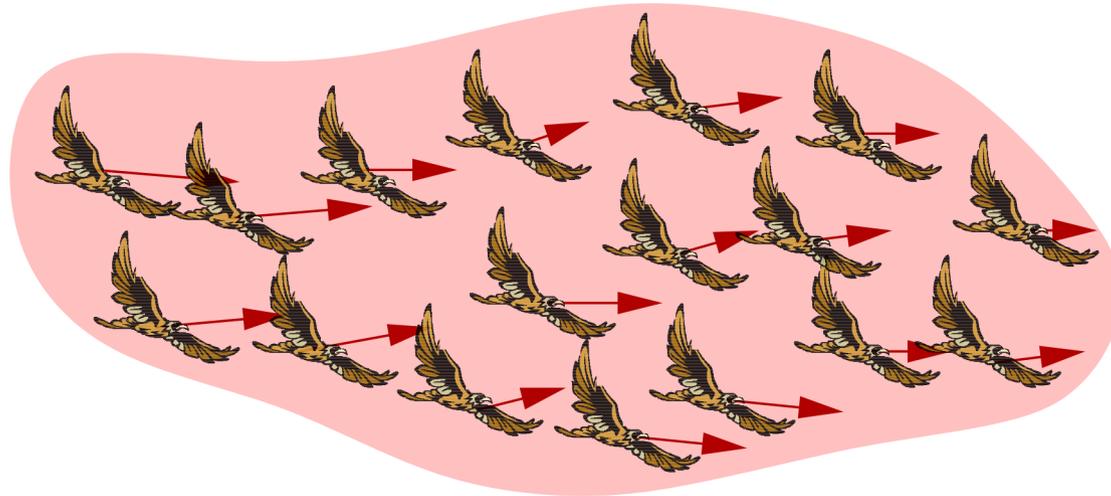
Processes will dynamically determine restriction

Possible Proto process primitives:

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

```
(instances variable)
```

Example: tracking a flock

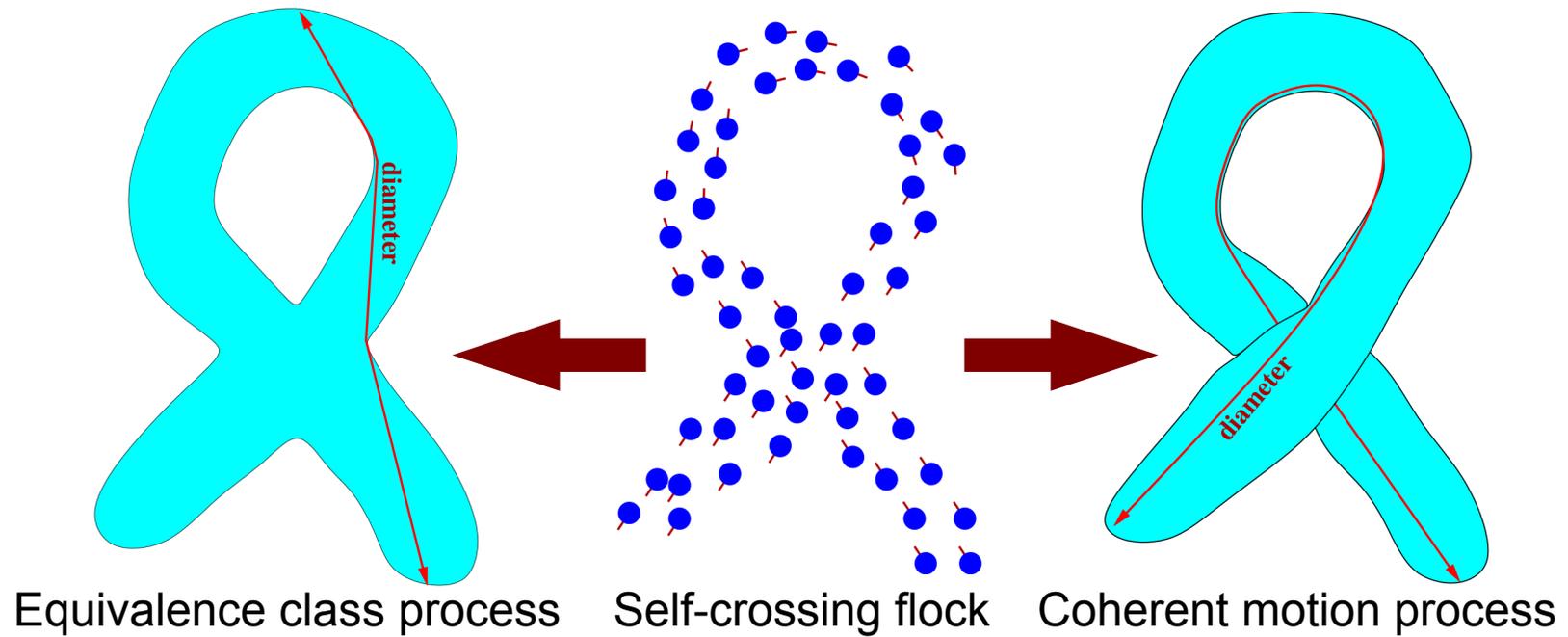


```
(def close-vec (base other err)
  (< (len (- base other)) (* err (len base))))

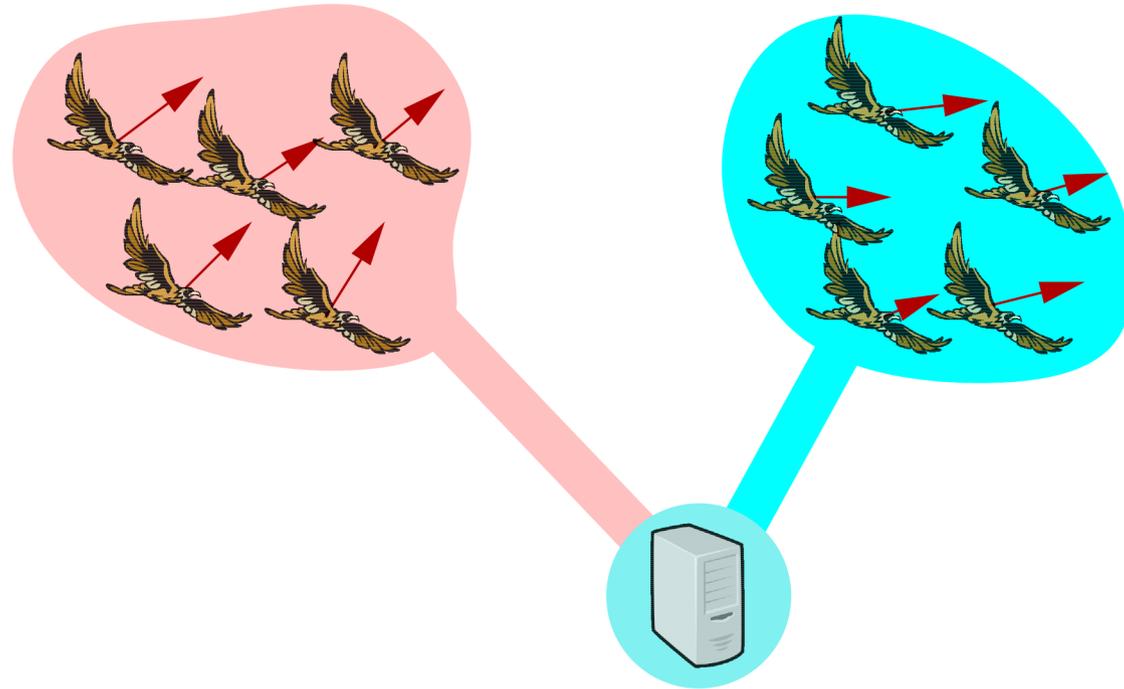
(def track-flocks (bird-vecs)
  (procs (bird-vec bird-vecs)
    ((flock-vec
      bird-vec
      (average (filter
                (lambda (v) (close-vec flock-vec v 0.1))
                bird-vecs))))
    ((close-vec flock-vec (nbr flock-vec) 0.1)
     (find-if (lambda (v) (close-vec flock-vec v 0.1))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

flock identity = similarly moving birds

Implication: self-crossing!



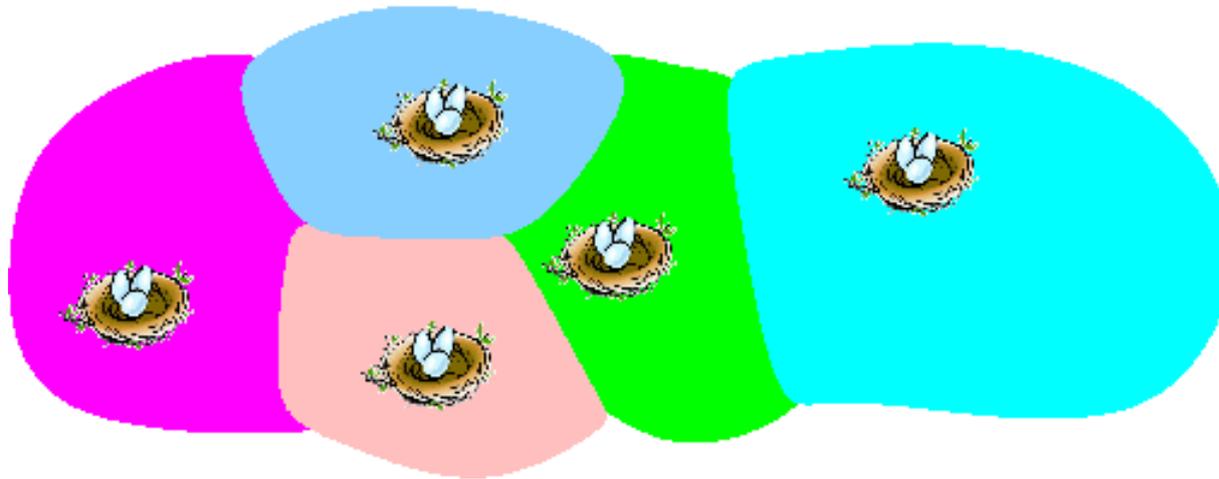
Example: reporting on flocks



```
(def report-data-stream (data-set base)
  (procs (data data-set)
    ((uid (1st data) uid)
      (src true (find uid (map 1st data-set))))
    ((= uid (nbr uid))
      (dilate src diameter))
    (channel-cast src base 2 (2nd data))))
```

use a reporting UID calculated by flock

Example: finding the nearest nest



```
(def voronoi (source payload-fn)
  (procs ((src-id (if source (tup (mid)) nil)))
    ((d (distance-to (= (mid) src-id)))
      ((= src-id (nbr src-id))
        (= d (apply min (instances d))))
      (payload-fn src-id d)))

  (voronoi (nest) (lambda (id d) (measure-shape))))
```

Processes compete on distance to nest

Contributions

- Defined spatially-extended processes
- Proved process IDs are impractical
- Proposed general process primitive for Proto
 - exa: weakening transitivity to define a flock

Open Questions

- What are good primitives for expressing dynamic process formation?
- What sorts of dynamic process-based algorithms are useful for various tasks?
- How can reportable identity be tracked for a process that splits and rejoins its parts?