Amorphous Medium Language

Jacob Beal
MIT CSAIL
LSMAS Workshop, July 2005
The Big Picture

Programming large spatially distributed systems is too hard!

- Reduce complexity by **programming continuous space** and **compiling for discrete agents**
- Increase reuse by **functional composition** of processes
A Farming Problem
A Farming Problem
A Farming Problem
A Farming Problem

Solar powered UCI DuraNode
A Farming Problem

Solar powered UCI DuraNode
A Farming Problem

Wait a minute! Weren't we programming a farm?
We need an Abstraction Barrier!

What behavior we want from the space

How a network of agents reliably produce the behavior
Related Work

- Amorphous Computing
  - [Coore 99], [Nagpal 01], [Kondacs 03]

- Paintable Computing
  [Butera 02]

- GHT [Ratnasamy et al. 02], TinyDB [Madden et al. 02]

- Regiment [Newton & Walsh 04]
Related Work

- Amorphous Computing
  - [Coore 99], [Nagpal 01], [Kondacs 03]

- Paintable Computing
  - [Butera 02]

- GHT [Ratnasamy et al. 02], TinyDB [Madden et al. 02]

- Regiment [Newton & Walsh 04]
Related Work

- Amorphous Computing
  - [Coore 99], [Nagpal 01], [Kondacs 03]
- Paintable Computing
  - [Butera 02]
- GHT [Ratnasamy et al. 02], TinyDB [Madden et al. 02]
- Regiment [Newton & Welsh 04]
Related Work

- Amorphous Computing
  - [Coore 99], [Nagpal 01], [Kondacs 03]

- Paintable Computing
  [Butera 02]

- GHT [Ratnasamy et al. 02], TinyDB [Madden et al. 02]

- Regiment [Newton & Walsh 04]
Amorphous Medium Language

- Global program → local action → global behavior
- Implicit distribution, coordination, communication
  - Program continuous space
  - Compile for discrete devices
  - Infrastructure supplied robust coordination primitives
- Functional composition of programs

(defun pest-management ()
  (in-components
   (dilate 2 (select-region (> (sense :bugs) 0.2)))
   (reduce-region #'max (sense :bugs) 0)))
AML: Two Compatible Models

- Global Model (continuous regions)
  - reduce-region, in-components, select-region, dilate

- Discrete Model (agents, messages)
  - send, receive, sleep, sense, act
AML: Two 3 Compatible Models

- Global Model (continuous regions)
  - reduce-region, in-components, select-region, dilate

- Neighborhood Model (continuous neighborhoods)
  - exposed-state@, reduce-neighbors

- Discrete Model (agents, messages)
  - send, receive, sleep, sense, act
AML: Two 3 4 Compatible Models

- Global Model (continuous regions)
  - reduce-region, in-components, select-region, dilate

- Neighborhood AML (composable neighborhood processes, infrastructure)

- Neighborhood Model (continuous neighborhoods)
  - exposed-state@, reduce-neighbors

- Discrete Model (agents, messages)
  - send, receive, sleep, sense, act
Programming in AML

AML Code

X compiles to

N-AML Code

✓ compiles to

AML Primitives

written in

in progress...

Neighborhood Methods

✓ simulated on

Neighborhood Emulator

✓ Executed by

Discrete Kernel

✓
Compiling AML to N-AML

AML:
(defprocess pest-management ()
  (in-components
    (dilate 2 (select-region (> (sense :bugs) 0.2)))
    (reduce-region #'max (sense :bugs) 0)))

N-AML:
(defun pest-management ()
  (where
    (dilate 2 (lambda (place) (> (sense-bugs place) 0.2)))
    (gossiped-value #'max #'sense-bugs 0)))

• Compilation is straightforward, implementing reduce-region is difficult
Neighborhood Model

- Continuous region of space
- Each point is a separate agent
- Agents share (delayed) exposed state w. neighbors
Discrete Model

- Dozens to billions of simple, unreliable agents
- Distributed through space, communicating by local broadcast
- Agents may be added or removed
- No global services (e.g. time, naming, routing, coordinates)
- Relatively cheap power, memory, processing
- Agents are immobile*

*or slow and dense enough to run Virtual Stationary Nodes [Dolev et al. 05]
Places

• Places are Points/Agents

• State at a place:
  – Unique ID
  – Sensors (e.g. bug-detector, temperature)
  – Actuators (e.g. pesticide sprayer, LEDs)
  – Running process
Neighborhood Abstraction

- A process interacts with its neighborhood by:
  - setting the state exposed to its neighbors
  - sampling the state exposed by its neighbors
Process Execution Model

- **terminatep@**
  - TRUE
  - FALSE

- **propagate@**
  - TRUE
  - FALSE

- **integrate-foreign-state@**
  - **execute-round@**

- **exposed-state@**

- **neighborhood abstraction**
  - **neighbor-values**
  - **exposed-state**
Handling Process Time

- Time is partially synchronous
  - Discrete rounds at each place
  - Different places have different clocks
- A process is an object containing state at round $T$
  - Round 0 specified as arguments, initial forms
  - Evolution specified as transition from $T$ to $T+1$
Starting and Stopping Processes

- Processes run when installed at a place
  - Places lacking a process attempt to install from neighbor's exposed state

- Processes are tested for termination each round
Handling Interaction with Neighbors

- Exposed state = constructor args, designated slots
  - Terminated processes expose nothing
- Incremental integration between rounds
  - special neighborhood forms
    - e.g. \texttt{(reduce-nbrs #'max v 0)}
      - may implicitly use hidden state
  - result returned during next transition between rounds
A process produces a time-series of values

- A composed process is a tree of processes containing other processes in their state
  - Compose by filling process slots (implicit or explicit)
  - Execution protocol must distribute through the tree
Local functions as processes

(sense-bugs place) accesses a sensor

(exposed-state@ place #'sense-bugs) \rightarrow #'(function #'sense-bugs)

(propagate@ place 'function state) \rightarrow #'sense-bugs

(integrate-foreign-state@ place #'sense-bugs state) \rightarrow \text{nil [no effect]}

(execute-round@ place #'sense-bugs state) \rightarrow ('sense-bugs place)

(terminatep@ place #'sense-bugs) \rightarrow \text{nil}

- An ordinary function is a process that ignores its neighbors and never terminates.
Describing Processes

(defnonlocal where ((f nonlocal) (target nonlocal)))
(declare (termination (terminatep target)))
(declare (exposing live))
(declare (integration (target (and live target))))
(with-state ((live nil))
  (setf live (evaluate f))
  (if live (evaluate target) nil)))

- where executes the target process only when the $f$ process returns true
N-AML: Constructor & Class Def'n

(defun where (f target)
  (let ((#:INST (make-instance 'where :f f :target target)))
    (with-slots (f target live) #:INST
      (setf live nil))
    #:INST)))

(defnonlocal where ((f nonlocal) (target nonlocal))
  (declare (termination (terminatep target)))
  (declare (exposing live))
  (declare (integration (target (and live target))))
  (with-state ((live nil))
    (setf live (evaluate f))
    (if live (evaluate target) nil)))

(defun where (f target)
  (let ((#:INST (make-instance 'where :f f :target target)))
    (with-slots (f target live) #:INST
      (setf live nil))
    #:INST)))

(defclass where (nonlocal)
  ((f :accessor where-f :initarg :f)
   (target :accessor where-target :initarg :target)
   (live :accessor where-live :initarg :live)))
(defnonlocal where ((f nonlocal) (target nonlocal))
 (declare (termination (terminatep target)))
 (declare (exposing live))
 (declare (integration (target (and live target)))))
 (with-state ((live nil))
 (setf live (evaluate f))
 (if live (evaluate target) nil))

(defmethod execute-round@ ((#:PLACE place) (#:INST where))
 (with-slots (f target live) #:INST
 (progn
 (setf live (execute-round@ #:PLACE f))
 (if live
 (execute-round@ #:PLACE target) nil)))))

(defmethod terminatep@ ((#:PLACE place) (#:INST where))
 (with-slots (f target live) #:INST
 (terminatep@ #:PLACE target)))
N-AML: Communication

(defnonlocal where ((f nonlocal) (target nonlocal))
  (declare (termination (terminatep target)))
  (declare (exposing live))
  (declare (integration (target (and live target))))
  (with-state ((live nil))
    (setf live (evaluate f))
    (if live (evaluate target) nil))

(defmethod propagate@ ((#:PLACE place) (#:TYPE (eql where)) #:EXP)
  (when (type-match where #:EXP)
    (destructuring-bind (f target live) (cdr #:EXP)
      (setf f (propagate@ #:PLACE (when (consp f) (first f)) f))
      (setf target
        (propagate@ #:PLACE (when (consp (and live target))
          (first (and live target)))
        (and live target)))
      (unless (terminatep@ #:PLACE target)
        (make-instance 'where :f f :target target :live nil))))

(defmethod integrate-foreign-state@
  ((#:PLACE place) #:INST where #:EXP)
  (when (type-match where #:EXP)
    (with-slots (f target live) #:INST
      (destructuring-bind (#:F #:TARGET #:LIVE) (cdr #:EXP)
        (integrate-foreign-state@ #:PLACE f #:F)
        (integrate-foreign-state@ #:PLACE target
          (and #:LIVE #:TARGET))))
    t))

(defmethod exposed-state@ ((#:PLACE place) #:INST where)
  (unless (terminatep@ #:PLACE #:INST)
    (with-slots (f target live) #:INST
      (list where
        (exposed-state@ #:PLACE f
          (exposed-state@ #:PLACE target) live))))
Putting the pieces together...

(defun pest-management ()
  (where
   (dilate 2 (lambda (place) (> (sense-bugs place) 0.2)))
   (gossiped-value #'max #'sense-bugs 0)))

• Compound process created by functional composition of constructors
Simulated Execution

- Implemented in Allegro CommonLISP
- Runs 5000+ agent simulations
Future Work

• Optimization
• Discrete→Mica2 Motes
• Finish Global→Neighborhood
  – Update compiler
  – Improved reduce-region primitive
AML Contributions

• Abstraction barrier between what and how in large multi-agent systems
  – Computation models bridging from continuous regions of space to agents passing messages
  – Language supporting functional composition of processes
  – Primitives scalable to extremely large systems
AML Contributions

• Abstraction barrier between what and how in large multi-agent systems
  – Computation models bridging from continuous regions of space to agents passing messages
  – Language supporting functional composition of processes
  – Primitives scalable to extremely large systems

Thanks to Hal Abelson, Jonathan Bachrach, Gerry Sussman
Appendices
Isn't AML too expensive to use?

• Plentiful opportunities for optimization
• Communication is limited by congestion
  – Maximum density of bits, not number
• Power isn't as tight a constraint as often perceived
  – Powered networks: user-deployed (RoofNet), solar (DuraNode), embedded distribution (biological tissue)
  – Lots of research on energy storage
  – Tradeoff between control response and power usage
Using neighbor state

(defnonlocal dilate (n (source nonlocal)))
(declare (termination (terminatep source)))
(declare (exposing r))
(with-state ((r (1+ n)))
  (merge-nbrs (r) (setf r (min r (nbr r))))
  (when (evaluate source) (setf r -1))
  (incf r)
  (<= r n)))

• The *dilate* process returns *true* within *n* units of points where the *source* process returns *true*
Using neighbor state

(defnonlocal gossiped-value
  (fuser (source nonlocal) &optional (base unspecified))
(declare (exposing value))
(declare (termination nil))
(with-state ((value base))
  (setf value
    (funcall fuser (evaluate source) (reduce-nbrs fuser value base)))))

• The *gossiped-value* process uses *fuser* to reduce the values of the *source* process to the same summary value at every place