Connected objects

*From macro to micro IoT*

Julien BOURGEOIS (UFC/FEMTO-ST)
Seth Copen G O L D S T E I N (CMU/SCS)

Work funded by:
ANR/RGC (ANR-12-IS02-0004-01 and 3-ZG1F),
ANR (ANR-06-ROBO-0009),
ANR (ANR-2011-BS03-005),
DARPA (FA87501010215),
NSF (CNS-0428738) and
Intel Corporation
Microsoft
Acknowledgments

• All Claytronics team, in particular:
  – Flavio Cruz, Frank Pfenning, Emre Karagozler, Michael Ashley-Rollman, Brian Kirby

• All OMNI team, in particular:
  – André Naz, Benoit Piranda, Dominique Dhoutaut, Nicolas Boillot, Hicham Lakhlef, Hakim Mabed, Pierre Thalamy
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Introduction

- Microtechnology is now a mature technology
- MEMS can be produced by thousands of units
- Applications:
  Accelerometers
  STMicro LIS331DLH
Introduction

• Microtechnology is now a mature technology
• MEMS can be produced by thousands units
• Applications:

Digital Micromirror Device

TI
MEMS Classification

MEMS
- Sensor MEMS
  - Single MEMS
    - Static topology
  - Distributed MEMS
    - Dynamic topology
- Actuator MEMS
  - Single MEMS
    - Static topology
  - Distributed MEMS
    - Dynamic topology
- Sensor/Actuator MEMS
  - Single MEMS
    - Static topology
  - Distributed MEMS
    - Dynamic topology

Accelerometer

DMD
Flow of information

Distributed Intelligent MEMS

Sensor MEMS
- Static topology
- Dynamic topology

Actuator MEMS
- Static topology
- Dynamic topology

Sensor/Actuator MEMS
- Static topology
- Dynamic topology

Output only

Input only

Input/Output

Scalability issue

Distributed Intelligent MEMS
Introduction

• Microtechnology is now a mature technology
• MEMS can be produced by thousands units
• Need for embedded intelligence
• New challenges:
  – Coordination needs distribution paradigm
    • Communication
    • Programming
    • Control
  – Smooth integration of different technologies
• Scalability up to millions!
  – 1 m³ of micro-robots -> internet on your table!
Introduction

• One overriding design goal: Scaling
• Scale up in numbers (millions of units)
  – A software challenge
• Scale down in size (sub mm dimensions)
  – A hardware challenge
Internet of [micro]-things

Monolithic intelligent objects

Distributed intelligent MEMS objects

Macro world

Micro world

Smart Surface

Claytronics

Monolithic intelligent objects

Distributed intelligent MEMS objects

Smart Surface

Claytronics
Internet of [micro]-things

Monolithic intel. obj.

Macro world

Low density of communication
Few communicating objects
Single point of contact

Smart Surface

High density of communication
High number of communicating objects
No point of contact by default

Micro world

Distributed intelligent MEMS objects

Claytronics
Relations to existing fields of research

- Parallel computing
  - Scientific applications running in parallel
- Mobile computing
  - Mobility
- Ad hoc networking
  - Topology and energy saving
- Pervasive/ubiquitous computing/Internet of Things
  - Interactions between intelligent objects
  - Linked to real-world
- Distributed Intelligent MEMS
  - Scalability
  - Linked to/act on real-world
Concurrent Systems

- Embedded-Physical Distributed
  Distributed Intelligent MEMS/Claytronics

- Geographically Distributed
  Internet

- Cloud Computing

- Parallel
## Concurrent Systems

<table>
<thead>
<tr>
<th>Physical</th>
<th>Geographical</th>
<th>Cloud</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geophysical location</td>
<td>+++</td>
<td>++</td>
<td>---</td>
</tr>
<tr>
<td>Relative location</td>
<td>+++</td>
<td>+++</td>
<td>+</td>
</tr>
<tr>
<td>Faults</td>
<td>++++</td>
<td>+++</td>
<td>++</td>
</tr>
<tr>
<td>Number of Processors</td>
<td>+++</td>
<td>+++</td>
<td>+</td>
</tr>
<tr>
<td>Network structure</td>
<td>varies</td>
<td>varies</td>
<td>fixed</td>
</tr>
<tr>
<td>Network connectivity</td>
<td>---</td>
<td>---</td>
<td>+</td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Claytronics

CATOM = Claytonic Atom

~meters (2006)

~decimeters (2007)

~centimeters (2007)

~milimeters (2012)
Catom: a rolling cylinder.

Shell: SiO$_2$ film + Aluminum

Chip: HV SOI CMOS die
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Meld programming

Programming ensembles is hard:

– Need to distribute computation
– Need to manage communication
– Need to keep nodes in a coherent state
– Needs to be fault tolerant
  (nodes and links fail)

*Or, sensor networks?
Meld programming

• Our goal: Program an ensemble as a single entity

• Let compiler and runtime handle issues
  – Automatically distribute computation
  – Automatically manage communication
  – Automatically maintain coherence
  – Automatically handle faults

• Based on prior work derived from Datalog
  – P2 for programming overlay networks [Loo et. al., SIGMOD 2006]
  – SNLog for programming sensor networks [Chu et. al., VLDB 2006]
Meaning of a Logic Program

• Standard forward-chaining logic programming language
  – Axioms never change
  – Once something is true, it stays true
  – Proving a fact is side-effect free
  – Run program until nothing more to prove
  – “Meaning” is final set of facts
Suppose we have a robotic system where the robots can move around one another. How can we get them to a particular location (e.g. 4) with 3 lines of code?
Walk example

**Rule 1:**
dist(S, DS) :- at(S, PS), destination(PD), DS = |PS-PD|.

**Rule 2:**
farther(S, T) :- neighbor(S, T),
    dist(S, DS), dist(T, DT), DS >= DT.

**Rule 3:**
moveAround(S,T,P) :- farther(S,T), vacant(T,P),
    dist(S,DS), destination(PD),
    DS > |P-PD|.
Walk example: Rule 1 dist

\[
\text{dist}(S, DS) :- \text{at}(S, PS), \text{destination}(PD),
\quad DS = |PS-PD|.
\]

\[
\text{dist}(s, DS) :- \text{at}(s, <0>), \text{destination}(PD),
\quad DS = |<0>-PD|.
\]

\[
\text{dist}(s, DS) :- \text{at}(s, <0>), \text{destination}(<4>),
\quad DS = |<0>-<4>|.
\]

\[
\text{dist}(s, 4) :- \text{at}(s, <0>), \text{destination}(<4>),
\quad 4 = |<0>-<4>|.
\]
Walk example: Rules

- Rules are created:
  destination(<4>)
  at(s, <0>)
  at(t, <1>)
  neighbor(s,t)
  neighbor(t,s)
  vacant(t,<2>)
  dist(s, 4)
  dist(t, 3)
  ...
Walk example: Rule 2, farther

farther(S, T) :- neighbor(S, T), dist(S, DS),
    dist(T, DT), DS >= DT.

farther(s, t) :- neighbor(s, t), dist(s, DS),
    dist(t, DT), DS >= DT.

farther(s, t) :- neighbor(s, t), dist(s, 4),
    dist(t, DT), 4 >= DT.

farther(s, t) :- neighbor(s, t), dist(s, 4),
    dist(t, 3), 4 >= 3.

farther(s, t) :- neighbor(s, t), dist(s, 4),
    dist(t, 3), 4 >= 3.
Walk example: Rules

- Farther rule is added:
  - destination(<4>)
  - at(s, <0>)
  - at(t, <1>)
  - neighbor(s,t)
  - neighbor(t,s)
  - vacant(t,<2>)
  - dist(s, 4)
  - dist(t, 3)
  - farther(s,t)
Walk example: Rule 3, moveAround

moveAround(S,I,P) :- farther(S,I), vacant(I,P),
                 dist(S,DS), destination(PD), DS > |P-PD|.

moveAround(s,t,P) :- farther(s,t), vacant(t,P),
                    dist(s,DS), destination(PD), DS > |P-PD|.

moveAround(s,t,<2>) :- farther(s,t), vacant(t,<2>),
                       dist(s,DS), destination(PD), DS > |<2>-PD|.

moveAround(s,t,<2>) :- farther(s,t), vacant(t,<2>),
                       dist(s,4), destination(PD), 4 > |<2>-PD|.

moveAround(s,t,<2>) :- farther(s,t), vacant(t,<2>),
                       dist(s,4), destination(<4>), 4 > |<2>-<4>|

moveAround(s,t,<2>) :- farther(s,t), vacant(t,<2>),
                       dist(s,4), destination(<4>), 4 > |<2>-<4>|. 
Walk example

• moveAround(s,t,<2>) is an action, not a fact
  – Causes robot s to move to <2>

• 3 lines of code makes two robots reach their goal
  – Program is provable and sound
  – Fault-tolerant
It works also with 3 catoms!
Or more!
For new applications
Mapping an area with wireless walkers
An example: temperatures
An example: temperatures
Goal: find local hotspots
Local Maximum Temperature

sensor readings, such as temperature, are modeled as axioms

/* compare temperature to nearby nodes */

localMax(A) :- temperature(A, T),
               forall closeNode(A, B, _)
               [temperature(B, T’), T’ < T].
Local Maximum Temperature

if the temperature at A is greater than that of every closeNode, then it is a local maxima

/* compare temperature to nearby nodes */

localMax(A) :- temperature(A, T),
               forall closeNode(A, B, _)
               [temperature(B, T'), T' < T].
Local Maximum Temperature

/* find nearby nodes */
type virtual neighbor

closeNode(module, module, min < int).

closeNode(A, B, 1):
- neighbor(A, B).

closeNode(A, C, N+1):
- neighbor(B, A),
  closeNode(B, C, N),
  N<3.

/* compare temperature to nearby nodes */
type localMax(module).
localMax(A) :- temperature(A, T),
  forall closeNode(A, B, _)
    [temperature(B, T’), T’ < T].
/* find nearby nodes */

closeNode(A, B, 1) :- neighbor(A, B).

/* compare temperature to nearby nodes */
type localMax(module).
localMax(A) :- temperature(A, T),
   forall closeNode(A, B, _)
   [temperature(B, T') -> T < T'].
Local Maximum Temperature

/* find nearby nodes */

closeNode(A, B, 1) :- neighbor(A, B).

/* compare temperature to nearby nodes */
type localMax(module).
localMax(A) :- temperature(A, T),
forall closeNode(A, B, _),
[temperature(B, T'), T' < T].

every immediate neighbor is a closeNode at distance 1
Local Maximum Temperature

/* find nearby nodes */

closeNode(A, B, 1) :- neighbor(A, B).
closeNode(A, C, N+1) :-
    neighbor(B, A), closeNode(B, C, N), N+1<3.

/* compare temperature to nearby nodes */
type localMax(module).
localMax(A) :- temperature(A, T),
   forall closeNode(A, B, C, N, T') [
        temperature(B, T'), T' < T].

neighbors of closeNodes are also close, up to distance 2
/* find nearby nodes */

closeNode(A, B, 1) :- neighbor(A, B).
closeNode(A, C, N+1) :-
    neighbor(B, A), closeNode(B, C, N), N+1<3.

Local Maximum Temperature
Local Maximum Temperature

/* find nearby nodes */

type virtual neighbor

closeNode(module, module, min <int>).
closeNode(A, B, 1) :- neighbor(A, B).
closeNode(A, C, N+1) :-
    neighbor(B, A), closeNode(B, C, N), N+1<3.

/* compare temperature to nearby nodes */

type localMax(module).

localMax(A) :- temperature(A, T),
    forall closeNode(A, B, _)
    [temperature(B, T'), T' < T].

using an aggregate, we consider only the closest distance for each closeNode
Local Maximum Temperature

/* find nearby nodes */

type virtual neighbor

    closeNode(module, module, min <int>).

closeNode(A, B, 1) :- neighbor(A, B).

closeNode(A, C, N+1) :-
    neighbor(B, A), closeNode(B, C, N), N+1<3.

/* compare temperature to nearby nodes */

type localMax(module).

localMax(A) :- temperature(A, T),
    forall closeNode(A, B, _)
    [temperature(B, T'), T' < T].

virtual neighbor indicates communications partners;
Local Maximum Temperature

/* find nearby nodes */
type virtual neighbor
    closeNode(module, module, min <int>).
closeNode(A, B, 1) :- neighbor(A, B).
closeNode(A, C, N+1) :-
    neighbor(B, A), closeNode(B, C, N), N+1<3.

/* compare temperature to nearby nodes */
type localMax(module).
localMax(A) :- temperature(A, T),
    forall closeNode(A, B, _)
    [temperature(B, T'), T' < T].
Programmer’s Viewpoint

- 1 program, running on a single ensemble
- Run forward-chaining until saturation
Axioms

\[
\begin{align*}
\text{neighbor}(a, b) \\
\text{neighbor}(a, c) \\
\text{neighbor}(b, a) \\
\text{neighbor}(b, d) \\
\text{neighbor}(c, a) \\
\text{neighbor}(c, d) \\
\text{neighbor}(d, b) \\
\text{neighbor}(d, c) \\
temperature(a, 68) \\
temperature(b, 66) \\
temperature(c, 73) \\
temperature(d, 65)
\end{align*}
\]
Run to Saturation

closeNode(a,b,1)
closeNode(a,c,1)
closeNode(b,a,1)
closeNode(b,d,1)
closeNode(c,a,1)
closeNode(c,d,1)
closeNode(d,b,1)
closeNode(d,c,1)
closeNode(b,c,2)
closeNode(c,b,2)
closeNode(a,d,2)
closeNode(d,a,2)
localMax(c)
...
What happens if temperature changes?

- neighbor(a,c)
- neighbor(c,a)
- temperature(a,68)
- temperature(c,73)
- closeNode(a,c,1)
- closeNode(c,a,1)
- localMax(c)
• Retract all affected facts

neighbor(a,c)
neighbor(c,a)
...
temperature(a,68)
temperature(c,73)
...
closeNode(a,c,1)
closeNode(c,a,1)
localMax(c)
Sensor Values Change

- Introduce new axioms, derive new facts

```
neighbor(a,c)
neighbor(c,a)
...
temperature(a,75)
temperature(c,73)
...
closeNode(a,c,1)
closeNode(c,a,1)
localMax(a)
```
Fact Retraction

• Retract axioms
• Rerun rules on retracted axioms, removing derived facts
• Use reference counting to keep derived facts with alternate derivations

aka Truth Maintenance
Fault Tolerance

- What if a link goes down?

```
neighbor(a,c)
neighbor(c,a)
... 
temperature(a,68)
temperature(c,73)
... 
closeNode(a,c,1)
closeNode(c,a,1)
localMax(c)
```
Fault Tolerance

- We just retract the affected facts

\[
\text{neighbor}(a,c) \\
\text{neighbor}(c,a) \\
\text{temperature}(a,68) \\
\text{temperature}(c,73) \\
\text{closeNode}(a,c,1) \\
\text{closeNode}(c,a,1) \\
\text{localMax}(c) \\
\text{localMax}(a)
\]
• A node fails?

neighbor(a,c)
neighbor(c,a)
...
temperature(a,68)
temperature(c,73)
...
closeNode(a,c,1)
closeNode(c,a,1)
localMax(c)
Same thing! Retract affected facts.

neighbor(a,c)
neighbor(c,a)

... temperature(a,68)
... temperature(c,73)

... closeNode(a,c,1)
closeNode(c,a,1)
localMax(c)
localMax(a)
Support for actions

• Actions have side effects
  – Derive an action instead of a fact
  – Perform the action
  – **Treat module move as a fault**, treat the position of the module like sensors

• In effect we run a program to either
  – saturation or
  – the derivation of an action.

• If an action is derived, we perform it and “restart” the program.

• And it scales…
Gates-Hillman shape change

type create(catom, point).
type destroy(catom, catom).
type give(catom, catom).
type resources(catom, int).

extern int inTargetShape(point).
type in(catom).
type out(catom).

type state(catom, min int).
type parent(catom, catom, int).
type child(catom, catom).
type false(catom, catom).
type possibleParent(catom, first catom, min int).
type notChild(catom, catom).

in(Module) :- position(Module, Spot), 1 = inTargetShape(Spot).

out(Module) :- position(Module, Spot), 0 = inTargetShape(Spot).

state(Module, Neutral) :- position(Module, _), Neutral = 2.

state(Module2, Path) :- neighbor(Module1, Module2, _), state(Module1, Final), Final = 0, out(Module2), aPath = 1.

state(Module2, Path) :- neighbor(Module1, Module2, _), state(Module1, Path), Path = 1.

state(Module2, Final) :- neighbor(Module1, Module2, _), state(Module1, Final), Final = 0, in(Module2).

possibleParent(Module2, Module1, Dist2) :- neighbor(Module1, Module2, _), state(Module2, Path), Path = 1, parent(Module1, _, Dist1), Dist2 = Dist1 + 1.

possibleParent(Module2, Module1, Dist) :- neighbor(Module1, Module2, _), state(Module2, Path), Path = 1, state(Module1, Final), Final = 0, Dist = 1.

child(Module2, Module1) :- neighbor(Module1, Module2, _), possibleParent(Module1, Module2, _).

parent(Module1, Module2, Dist). :- neighbor(Module2, Module1, _), child(Module2, Module1), possibleParent(Module1, Module2, Dist).

notChild(Module1, Module2) :- neighbor(Module1, Module2, _), parent(Module2, Module3, Dist2), parent(Module1, _, Dist1), Dist2 = Dist1 + 1, Module1 != Module3.

notChild(Module1, Module2) :- neighbor(Module1, Module2, _), state(Module2, Final), Final = 0.

destroy(Module1, Module2) :- state(Module1, Path), Path = 1, neighbor(Module1, Module2, _), resources(Module1, Destroy), Destroy = 0, resources(Module2, Destroy), Destroy = 0,forall neighbor(Module1, N, _) notChild(Module1, N),forall child(Module1, Child) false(Module1, Child).

give(Module1, Module2) :- neighbor(Module1, Module2, _), resources(Module1, Create), Create = 1, resources(Module2, Destroy), Destroy = 0.

create(Module, Spot) :- state(Module, Final), 0 = Final, // in final state vacant(Module, Spot), 0 = inTargetShape(Spot), resources(Module, Create), 1 = Create.
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Simulation environments

- DPRSim: the Dynamic Physical Rendering Simulator
  - Not maintained anymore
- BBSim: Blinky blocks Simulator
  - Only for Blinky Blocks
- VisibleSim:
  - Multi-targets (Blinky Blocks, Smart blocks, RoboBlocks, Claytronics (TODO))
  - Interactive
  - Include debugging
Required properties

- Versatile
  - multi-targets simulator
- Scalable
  - Up to millions
- Deterministic
  - Reproducibility for analyzing and debugging
- Simulate the environment
  - Sensing and actuation
- Programming model agnostic
  - Meld
  - C
  - Java
  - Etc.
Simulator architecture

- Object oriented: inheritance helps to share code between multiple platforms.
- Applications just have to derive a class representing the code being executed in each element.

Elements common to all targets

Elements derived for each target

Each application derives its own BlockCode and make use of the appropriate simulator
Physical simulation

- Depending on the simulated platform, various kinds of physics can be implemented.
- For example, levitating and controlling air jets for the SmartBlocks platform.
- Applications can interact with conveyed objects
Physical simulation

- Depending on the simulated plateform, various kinds of physics can be implemented.
- For example, levitating and controlling air jets for the SmartBlocks plateform.
- Applications can interact with conveyed objects.
Physical simulation

- Depending on simulated platform, various kinds of physics can be implemented.
- For targets allowing 3D stacking of elements, the simulator is able to take gravity and « feasibility » into account.
Networking simulation
Debugging
Debugging

- Set or remove break point at specified node
- Set or remove break point at specified actions
  - Fact retraction, action, sense, etc.
- Dump the state of the system
- Run, stop or continue the simulation
- Stop the simulation at a certain time
- Etc.
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Hardware

• Connectivity infrastructure
  – USB → Ensemble communication
  – Easy ensemble reprogramming

• Wired power adapter

• Inertial Measurement Unit
  – Heterogeneous Module
  – Provides Acceleration and Gyroscopic data
Software

• Versatile Application Programming Interface (API)

• System messages
  – Periodically sent to allow for state updates
  – Invisible to user

• Application messages
  – User-triggered
  – Automatically queued, sent, retried (if necessary), and acknowledged
  – Active messaging (received messages handled automatically)

• Logging system
  – Debugging purpose
Outline

- Introduction
- Claytronics
- Meld
- Simulation environment
- Hardware platform
- New results
- Conclusion
Synchronization

• Inspired from
  • NTP and PTP for the communication delay estimation
  • RBS and FSTP for the correction of the clock skew with a linear model

• Blinky Blocks Time Protocol
  – Requirements
    • Benchmarking of the communications delay
    • Model of the clock skew
  – During runtime
    • Election of the time master
    • Construction of a spanning tree
    • Synchronization of the clocks
      – Calibration phase
      – Runtime phase
Test on a BB platform

Synchronizing a radius of 27 blocks
Octaedron $27\sqrt{2}$ blocks: **27 775 blocks!**
Results

Sans BBTP          Avec BBTP
Map-less reconfiguration

- Without map (predefined positions) of the target shape ⇒ $O(1)$ of memory ⇒ scalable algorithm
- Distributed and parallel algorithm
- From chain configuration to square configuration
- Predicting of the number of movements for each node ⇒ energy-aware
- Optimal number of movements
- Best complexity of messages so far: $N/2$ messages
- Programmed in MELD inside DPRSIM
Map-less reconfiguration
Simulation of nano-communication networks

- Investigating integration of nano-communication networks in micro-robots
- Design of new simulator integrated in DPRSim/VisibleSim
Simulation of nano-communication networks

- Investigating integration of nano-communication networks in micro-robots
New applications

- Cooperative mapping of environments
Outline

• Introduction
• Claytronics
• Meld
• Simulation environment
• Hardware platform
• New results
• Conclusion
Conclusion

• Meld integrates sensors, actions, and fault tolerance cleanly via fact-retraction
• Meld provides a method for programming an ensemble as a single entity
• Automatically distributes state & computation, manages communication, manages mutation, manages deadlines, etc.
• Amenable to proving properties about the program
Conclusion

• Use Meld for compute and coordination

• A program is:
  – The computation
  – Scheduling coordination
  – Locality coordination

• Meld fact types:
  – Logical performs computation
  – Routing data structure organization
  – Sensor get locality/scheduling info
  – Actuation change locality/scheduling info