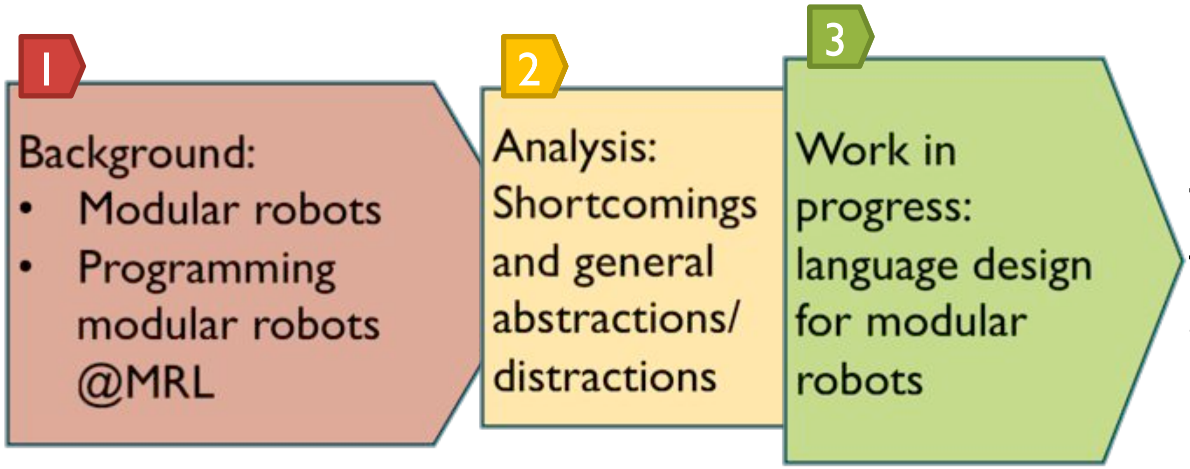
Language Abstractions for Modular Robots

Ulrik Pagh Schultz

Modular Robotics Lab http://modular.mmmi.sdu.dk

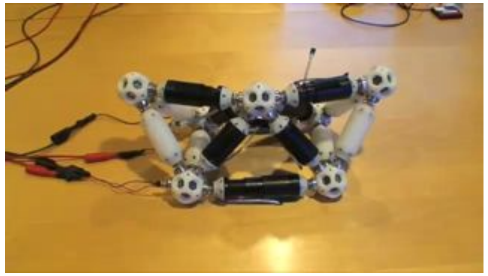
University of Southern Denmark

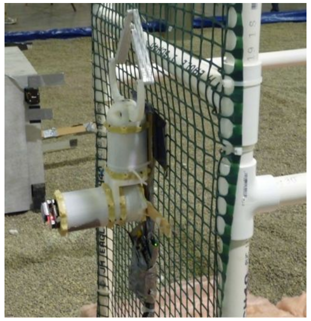


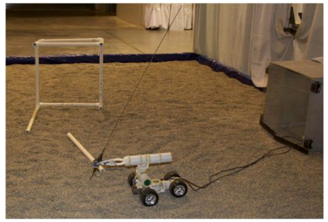
The search for the right abstractions

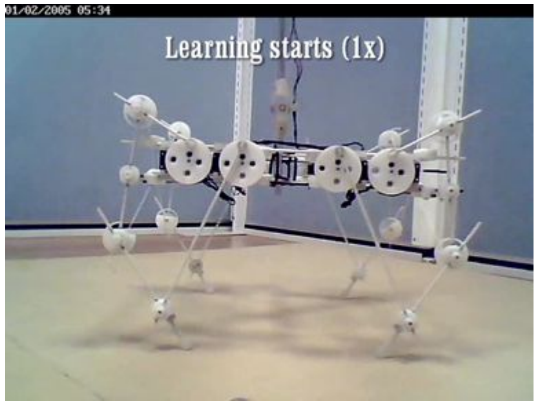
Reconfigurable modular robots

- Robot constructed from multiple physical modules (mechatronic links)
- Many variations, typically:
  - local (limited) CPU
  - neighbor-to-neighbor communication
  - *physically interlocked connections*
- Advantages: versatile, mass production, fault tolerance?
- Applications: construction kit (ICRA Contingency Challenge), …



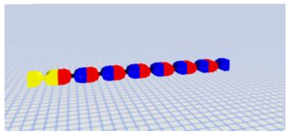


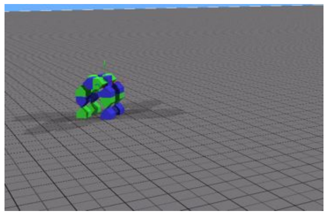


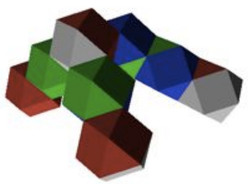


http://modular.mmmi.sdu.dk

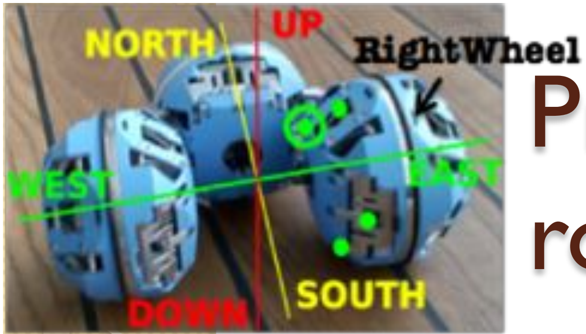
Self-reconfigurable modular robots







- Typically: modules can physically move around in the structure
- Control: off-line/on-line
- Advantages: adaptation, fault tolerance?
- Challenges:
  - morphology (connector design)
  - motion constraints
  - *communication (global versus local)*
  - *control (distributed, dynamic, unreliable)*
  - *adapting behavior to physical shape (and vice-versa)*
  - …



Programming adaptive behaviors: roles

```
abstract role Wheel extends Module {
 …
 require self.center == $EAST_WEST;
 require sizeof(self.connected(side)) == 1;
 behavior move() {
  self.$TURN_CONTINUOUSLY(turn_dir);
 }
 command evade() {
  self.$TURN_CONTINUOUSLY(evasion_dir);
  self.sleepcs(25);
 }
}

role RightWheel extends Wheel { … }

role Head extends Module {
 require self.center == $NORTH_SOUTH;
 startup initialize() {
  handle $PROXIM_1 $PROXIM_3 {
   Wheel.evade(0);
…
```

DynaRole [ICRA'09]

- Role = hierarchy of behaviors in context
- Spatial constraints for activation and deployment
  - global "compass" wrt origin
  - local constraints
- Virtual machine, quick role-driven reprogramming etc

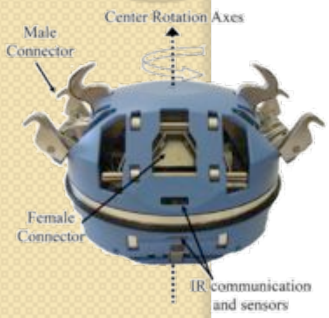
Programming self-reconfiguration: group behaviors

- Self-reconfiguration = group sequential behavior
- Robust local/global execution in the presence of partial hardware failure
- Manage physical parallelism easily
- Automatic derivation of reverse sequence

[Robotica'11]

```
sequence eight2car {
  M0.Connector[$CONNECTOR_0].retract() &
  M3.Connector[$CONNECTOR_4].retract();
  M3.Joint.rotateFromToBy(0,324,false,150);
  …
}
…
car2eight = reverse eight2car;
car2snake = car2eight + eight2snake;
snake2car = reverse car2snake;
```

Programming spatial awareness: morphology

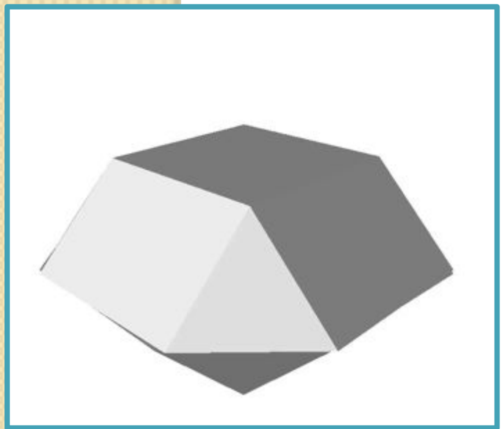


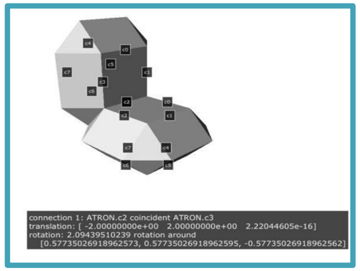
M3L: Modular Mechatronics Modeling Language

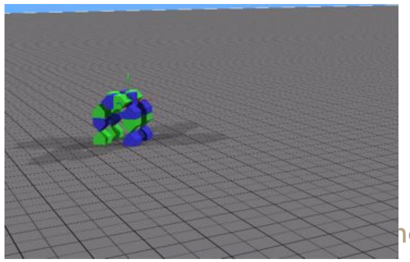
- DSL for arbitrary modular mechatronic system
- Automatic runtime forwards kinematics

[GPCE'10,IROS'11]

```
module ATRON:
    point c0: coords=(0,1,-1),gender="male",extended=[bool]
    point …

    link north: grouping=(c0,c1,c2,c3)
    link south: grouping=(c4,c5,c6,c7)

    axis north_axis: origin=(0,0,0), direction=(0,1,0)
    axis south_axis: origin=(0,0,0), direction=(0,-1,0)

    joint center: type=revolute, value=[-inf,inf],
              pair=((north,north_axis),(south,south_axis))
```







Assessment

Useful abstractions:

- Roles = intra-modular state and behaviors (but how to express?)
- Sequences = fixed self-reconfiguration (but how to generalize?)
- M3L+Roles+Labels = morphology-independent programming (but maybe too complex?)

Missing abstractions:

- Global representation of control
- Interaction (influence) between roles
- Groups of modules:
  - interacting
  - adaptive behaviors
  - composing behaviors
- *Key question: what are the right abstractions?*

Analysis

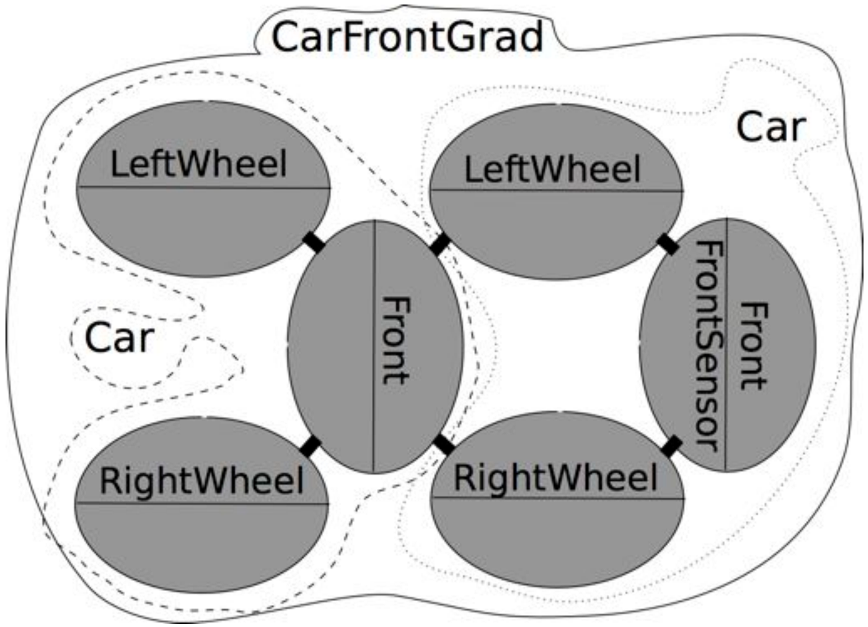
- Module: behavior in a context (=state)
  - a module plays a role given by its context
  - role defines (local) reactive behaviors
- Group (ensemble): module interactions
  - reactive and proactive behaviors according to context
  - execution by distributed state and consensus
  - precise algorithm across modules
- Global (robot): probabilistic (next talk…)

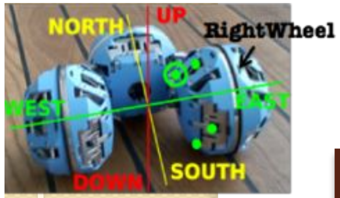
Ensembles, roles, and modules

"Rocoin" overview:

- *Ensemble:* dynamic distributed scope that encapsulates shared state and behavior
- *Role:* local state and behavior
- Basic principles:
  - interactions through state changes
  - distributed state and execution by merging

[SCW'12,RC'12 wip]

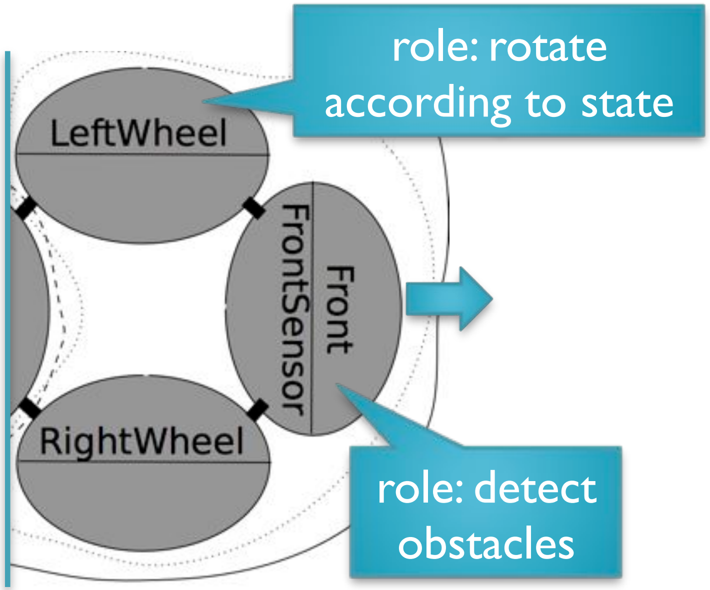




3

Ensemble: obstacle avoidance

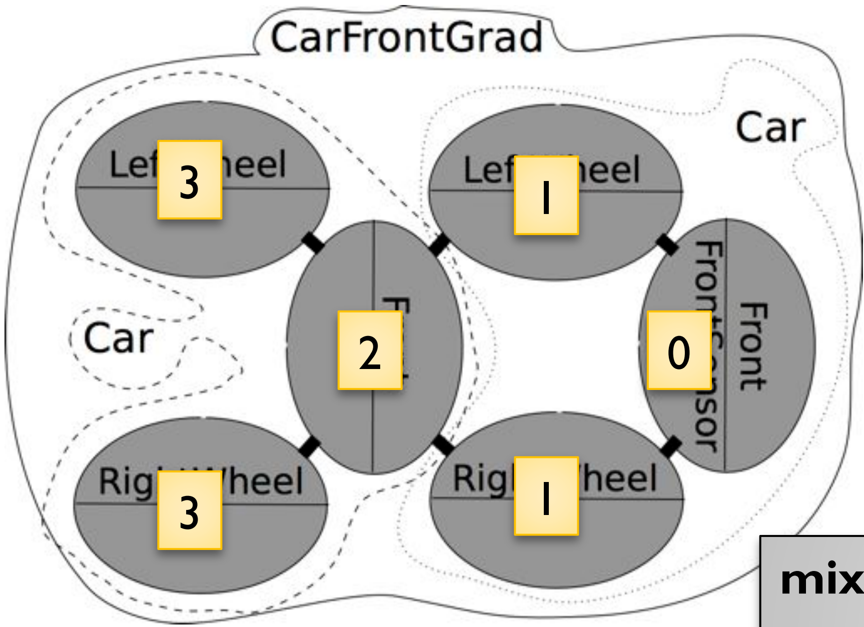
```
ensemble Car {
  state obstacle { None, Left, Right, Center } = None;
  when (obstacle==None) Wheel.drive.Forward;
  else Wheel.drive.Evade;
}
```



- Ensemble controls overall behavior
- State changes:
  - shared variables
  - shared behavior execution
  - role state assignments

Robust to packet loss and state loss

Ensemble: gradients




```
ensemble GradientField {
  int g = MAX_INT update {
    min = MAX_INT;
    for(ng: GradientField.g)
      min = Math_min(min,ng);
    g = (min+1);
  };
}
```

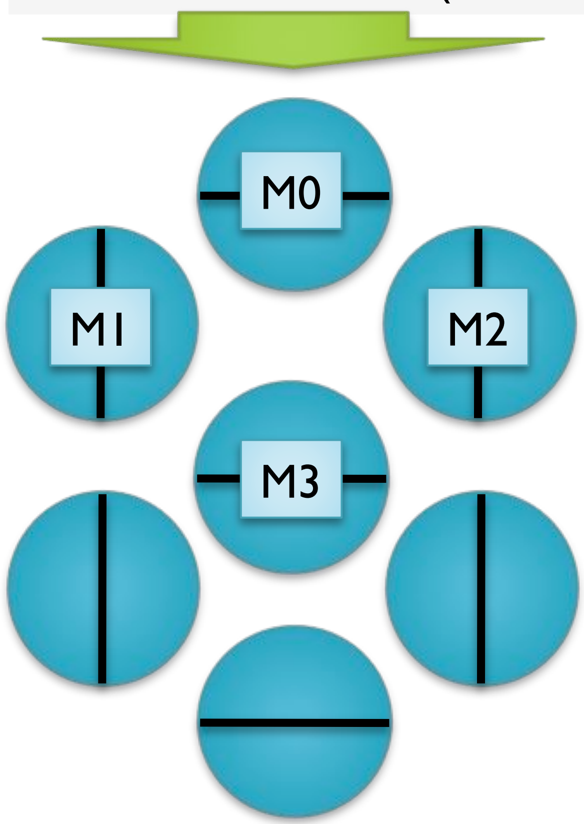
```
mixin role Source within GradientField {
  g.update { g = 0; }
}
```

```
ensemble  CarFrontGrad extends GradientField {
  require subrole(Wheel)||subrole(Front);
}
role Front extends Source { … }
```

- Critical issue: ensemble formation
- Proposed solution: restrict based on state

Ensemble: distributed execution

```
M0.connector[0].retract() |
M3.connector[4].retract();
M3.rotateFromTo(0,324); ...
```



- Previous work:
  - distributed state machine "program counter" is shared state
  - fixed merge function
- Current work:
  - condition = implicit / explicit consensus
  - loops: careful
  - programmer control over merge function (reflection)

Ensemble: reversibility

- *Not yet remotely possible!*
- Neural network + genetic algorithms + magic [Christensen]
- Key question: can we enable any programmer to do this?

- Reversibility for error recovery
  - reverse at any program point
  - easy solution: step counter part of shared state
  - merge: forward = MAX, reverse = MIN
- Reversibility because we can
  - arbitrary controller
  - reversible role behaviors

Open questions

- Execution model visible at language level?
- Separation of concerns also valid for execution flow
- Declarative versus operational
- General-purpose vs DSL

Goal: morphogenesis