Robust and Reversible Self-Reconfiguration

SCM’10, Technical Faculty

Ulrik Pagh Schultz, joint work with Mirko Bordignon, David Christensen & Kasper Søj Moduler Roboter Lab, PPR, UCS

- Reconfigurable robot: change physical shape to adapt to task/environment
- Self-reconfigurable robot: can change its own shape; in practice a modular robot
- Whole composed of physically connected modules
  - Real-time distributed embedded system with dynamic topology
  - Physical composition influences behavior
- Module typically has several distinct behaviors aka “roles”

M-TRAN
ATRON
Odin (deformable)

Programming language abstractions for roles and structure

Role: set of behaviors that are active in a given context

```cpp
role Read { #connections(*)==2; handle proximity(C3) { LeftWheel.reverse() ...
} }
role LeftWheel { #connections(WEST)==0; }
role RightWheel { #connections(EAST)==0; }
```

DynaRole (previous slide)
- Declarative specification of role selection
- Compiles to DCD-VM program fragments

DCD-VM
- Dynamic code distribution:
  - role-based code diffusion
  - runtime, non-intrusive
- Context management (neighbors + compass & 3D position from seed)
- Role-based behavior control & scheduling
- Role-based group communication
- Runs on TinyOS

Collaboration between roles, e.g. self-reconfiguration?
- Tolerance towards partial failures (robustness?)

Robust self-reconfiguration?

- Self-reconfigurable
- This talk: high-level [intermediate] programming language
- Robust towards partial failures
- Reversible self-reconfiguration for modular robot ensembles
- Flexible? Never done “Snake to car” before Versatile? Must be easy to program
- Robust? Communication is the weakest link
- Cheap? Must tolerate partial failures

Advantages:
- Versatile
- Cheap

Robust and reversible self-reconfiguration: overview

1. Implementation
   - execution using distributed state machine
   - state management
2. Properties
   - robustness and efficiency
   - reversibility
3. Conclusion and open issues
Language support for self-reconfiguration

- High-level distributed scripting / intermediate language
- Sequence of module operations with explicit sequencing / parallelism
- Extension to role-based language+VM ([ICRA’09]) (but not well integrated)

```plaintext
role M0 extends Module {
    ...
}
seq eight2car = {
    M0.connector[0].retract() | M3.connector[4].retract();
    M3.rotateFromTo(0,324);
    ...
}
```

(Compiles to native nesC running under tinyOS; VM support is future work)

Express self-reconfiguration as a sequence of steps executed by module structure
- Controlled parallelism using pending states (P)

---

State management

- Globally shared state
  - Store current and pending states in all modules
  - Continuously and independently of actions communicate local state to all neighbors
  - Merge incoming global state to ensure progression

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

- Start retract()

Globally shared state

- Store current and pending states in all modules
- Continuously and independently of actions communicate local state to all neighbors
- Merge incoming global state to ensure progression
Globally shared state

- Store current and pending states in all modules
- Continuously and independently of actions communicate local state to all neighbors
- Merge incoming global state to ensure progression
State management

Globally shared state

- Store current and pending states in all modules
- Continuously and independently of actions communicate local state to all neighbors
- Merge incoming global state to ensure progression

M1
M2
M3

<4,Ø>
<4,{1}>
<4,{1}>
<4,{1}>

Key property for state merge:

- A pending state is added before increment and propagates
- Comparing state values determines which pending states should be preserved when merging
- Pending state updates can propagate independently of the active state

Properties [1/2]: Robustness and efficiency

Robustness: partial failures
- Order of magnitude improvement!
- Communication:
  - continuous transmission of idempotent packets
  - broadcast communication
- Module reset:
  - idempotent operations
  - replication of global state

Efficiency:
- Time: continuous transmission ensures fastest safe progression
- Steps: massive opportunities for parallelization often unexploited
- Experiments: reversible experiments reduces need for reassembly

Properties [2/2]: Program reversibility

Reversible programs:
- facilitated by API design
- practical tool, not theoretical result (reverse, then generate)
- Not reversal in a purely semantic sense
- Perfect for self-reconfiguration

Details (skip - for now)

- Distributed state merge function
- Robustness experiments
- Generality of reversibility

Reversible programs:
- seq eight2car = { M0.connector[0].retract() | M3.connector[4].retract(); M3.rotateFromTo(0,324); ...
- seq car2eight = rev eight2car;
Assessment (open issues)

- **Good:**
  - order of magnitude improvement in robustness
  - scripting language so advantageous that robotics people want to use it (robustness and reversibility)
- **Not so good:**
  - pending state complex and insufficient
  - lacks underlying semantic model for proper integration with role-based features
  - no support for distributed decision of what sequence is applicable
  - no support for distributed algorithms

Conclusion and open issues

- **Robust and reversible self-reconfiguration**
  - robustness through shared state and idempotent messages and commands
  - reversibility through compiler and API design
- **Open issues**
  - design/implementation/generality of distributed [reversible] scripting language
  - robustness properties vs state management