Concurrency and Coordination in Robotics

Hexapod Control using SCOOP

Ganesh Ramanathan, Benjamin Morandi, Sebastian Nanz
Chair of Software Engineering, ETH Zürich

2010

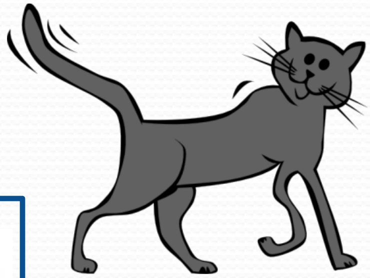
Outline

- The problem of Hexapod locomotion
- Specification: the biological model
- Implementations
  - Sequential programming
  - Multi-threaded programming
  - SCOOP (Simple Concurrent Object-Oriented Programming)
- Demonstration

Legs and Locomotion

Distributed control

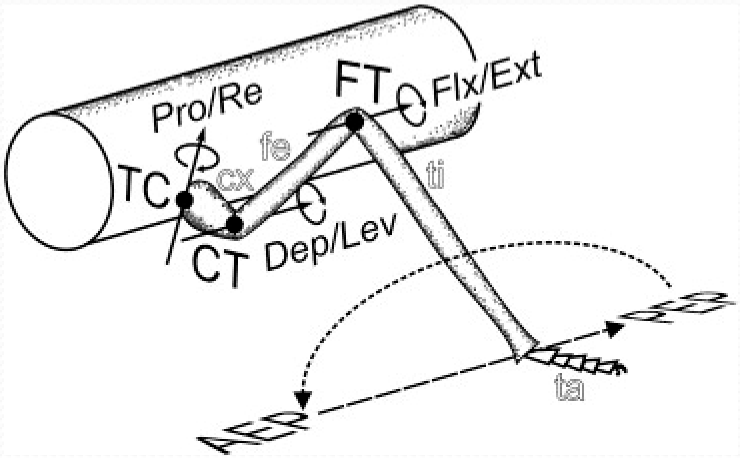
Load sensing



Centralised control

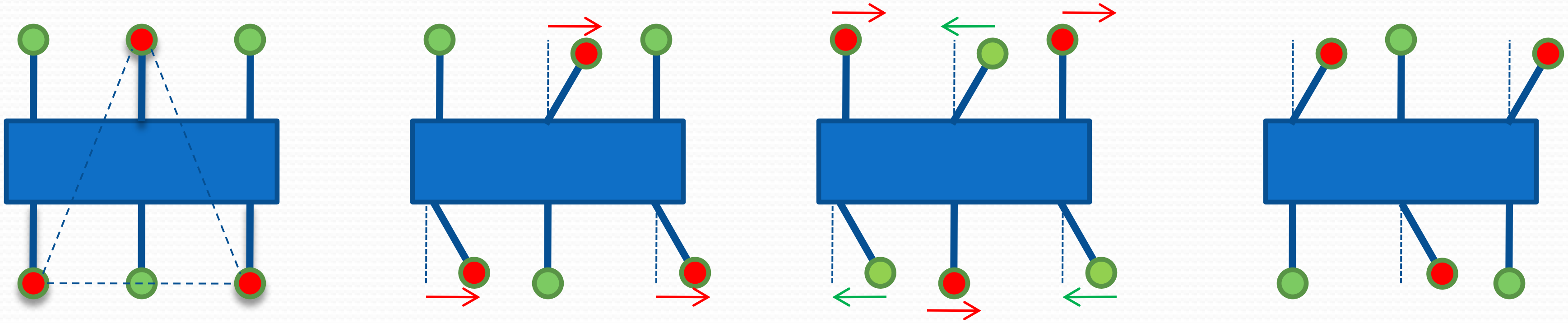
Balance sensing

Hexapod Locomotion



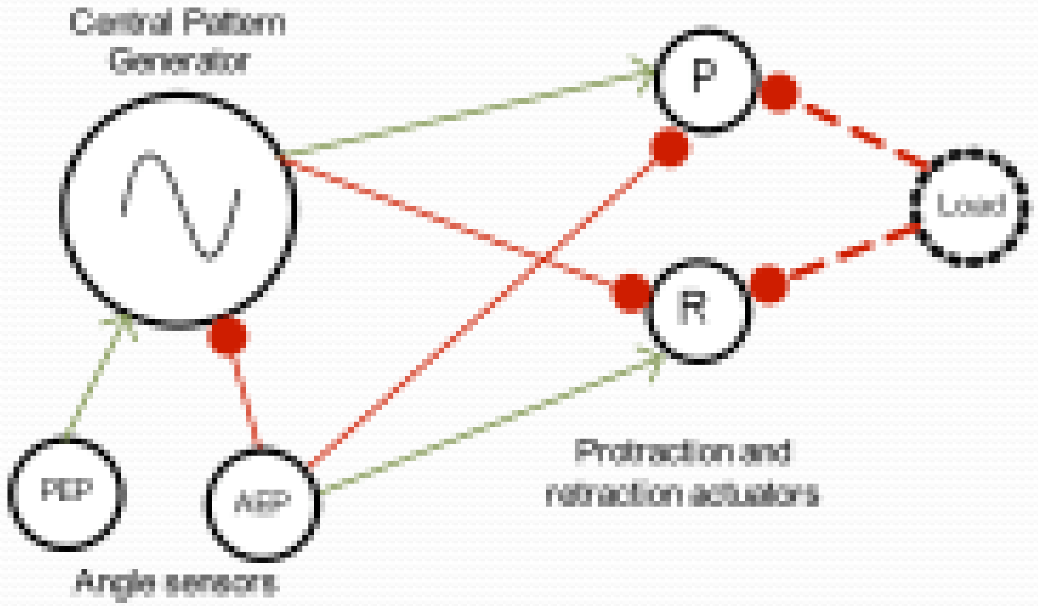
- The hexapod should maintain the static stability by keeping the center of gravity within the bounds of the grounded legs.
- Dragging of feet should be avoided.
- Three degrees of freedom per leg, load sensor on feet, forward and rear angle sensing

The Tripod Gait



- Alternating protraction and retraction of tripod pairs
  - Begin protraction only if partner legs are down
  - Depress legs only if partner legs have retracted
  - Begin retraction when partner legs are up

Biological Model

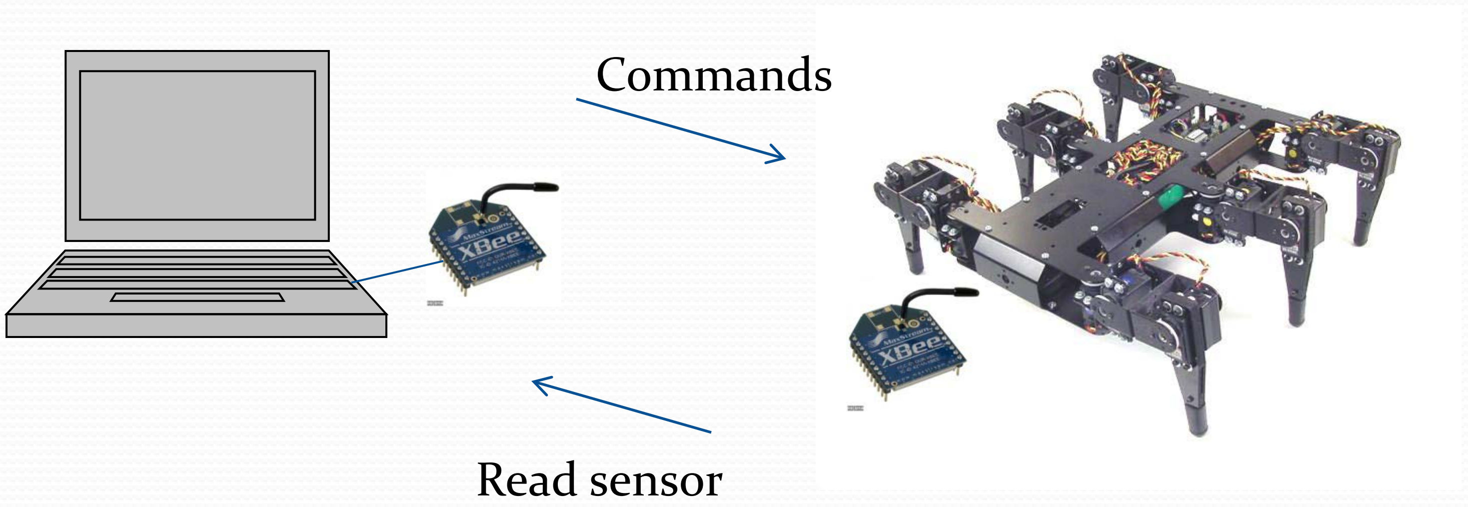


- Sensory Inputs allow or inhibit transmission of pulses from the Pattern Generator

The Hexapod Robot

- Hind legs have force sensors on feet and retraction limit switches.

The Hexapod Robot



- The control program (SCOOP based or other variants) runs on the PC and transmits command to the on-board servo controller.
- It also polls the inputs to obtain sensor information.

Implementation: Sequential Program

```
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true)
{
    lead.Raise();
    lag.Retract();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
```

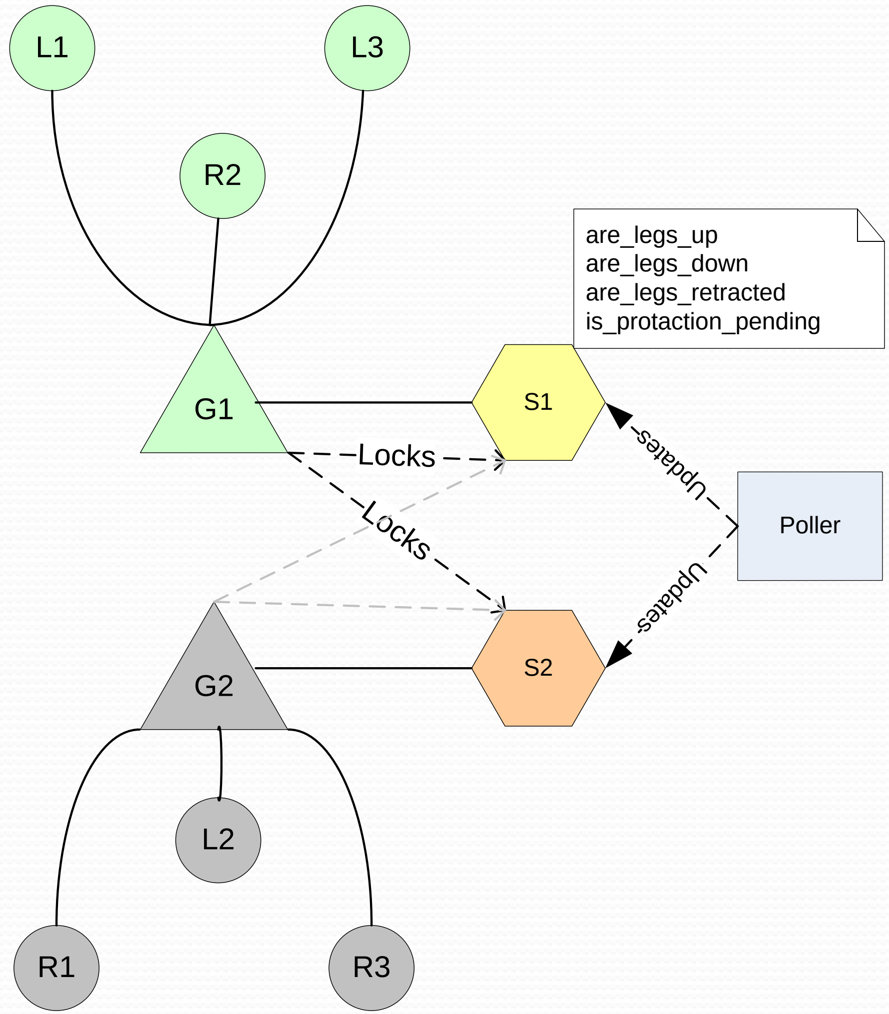
Implementation: Multi-Threaded Program

```
private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState !=ThreadState.
            AbortRequested)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        // Waiting for partner retraction
        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Partner.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
```

SCOOP Overview

- Each object is associated with an abstract processor (its handler)
- Feature calls can only be executed by the handling processor, providing mutual exclusion on a per-object basis
- Locking is expressed by the formal argument list of a routine, providing mutual exclusion on a set of objects
- Synchronization is expressed by wait conditions (preconditions with wait semantics)

Implementation: SCOOP



Implementation: SCOOP

```
walk
    do
        checklegs (my_signaler)
        from until my_signaler.stop_requested
        loop
            begin_protraction (partner_signaler, my_signaler)
            ensure_protraction (my_signaler)
            complete_protraction (partner_signaler)
            execute_retraction (partner_signaler, my_signaler)
        end
    end
```

Implementation: SCOOP

```
begin_protraction(partner, me:separate LEG_GROUP_SIGNALER) is
    --
    require
      my_legs_retracted : me.legs_retracted
      partner_down : partner.legs_down
      partner_not_protracting : not partner.protraction_pending
    do
      io.put_string (group_name)
      io.put_string (" : begin_protraction ")
      io.put_new_line

      tripod.lift

      me.set_protraction_pending(true)
    end
```

Main benefits of SCOOP

- Freedom of data races
- Wait conditions are an intuitive mechanism for implementing coordination
- Small semantical gap between specification and implementation

Demonstration