Architectures for Robot Control

15-494 Cognitive Robotics
David S. Touretzky &
Ethan Tira-Thompson

Carnegie Mellon
Spring 2009

Why Is Robot Control Hard?

Coste-Maniere and Simmons (ICRA 2000):

- High-level, complex goals
    - Assemble this water pump
    - Cook my breakfast
- Dynamic (changing) environment
- Robot has dynamic constraints of its own (don't fall over)
- Sensor noise and uncertainty
- Unexpected events (collisions, dropped objects, etc.)

Approaches To Control

1. Hierarchical: classic sense-plan-act

   - "Top-down" approach

   - Start with high level goals, decompose into subtasks

   - Not very flexible

2. Behavioral

   - "Bottom-up" approach

   - Start with lots of independent modules executing concurrently, monitoring sensor values and triggering actions.

   - Hard to organize into complex behaviors; gets messy quickly.

3. Hybrid

   - Deliberative at high level; reactive at low level

Levels of Control Problem

Robots pose _multiple_ control problems, at different levels.

- Low-level control:
    - Example: where to place a leg as robot takes its next step
    - Generally, continuous-valued problems
    - Short time scale (under a second); high frequency loop

- Intermediate level control:
    - Navigating to a destination, or picking up an object.
    - Continuous or discrete valued problems
    - Time scale of a few seconds

- High level control:
    - What is the plan for moving these boxes out of the room?
    - Discrete problems, long time scale (minutes)

Low-Level Control Issues

- Real-time performance requirement

  - Code to issue motor commands or process sensor readings must run every so many milliseconds.

- Safety: avoid states with disastrous consequences

  - Never turn on the rocket engine if the telescope is uncovered.

  - Never fail to turn off the rocket engine after at most n seconds.

  - Therac-25 accident (see IEEE Computer, July 1993)

  - Safety properties sometimes provable using temporal logic.

- Liveness: every request must eventually be satisfied

- Deadlock-free

"Reactive" Architectures

- Sensors directly determine actions.

- In its most extreme form, stateless control.

- "Let the world be its own model."

- Example: light-chasing robot:
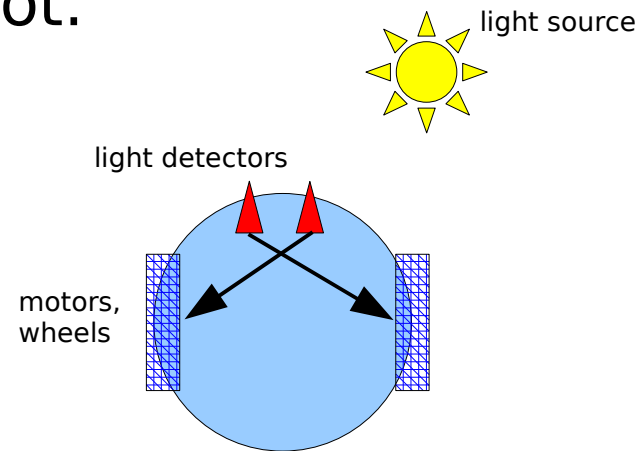
light source

light detectors

motors,
wheels

```
(behavior chase-light
  :period (1 ms)
  :actions
    ((set left-motor (right-sensor-value))
     (set right-motor (left-sensor-value))))
```

Overriding a Behavior

- If robot loses sight of the light, turn clockwise until the light comes back into view.



```
(behavior chase-light
  :period (1 ms)
  :actions
    ((set left-motor (right-sensor-value))
     (set right-motor (left-sensor-value))))
```

```
(behavior find-light
  :overrides (chase-light)
  :test (0? (+ (left-sensor-value)
              (right-sensor-value)))
  :actions
    ((set left-motor 0.5)))
```
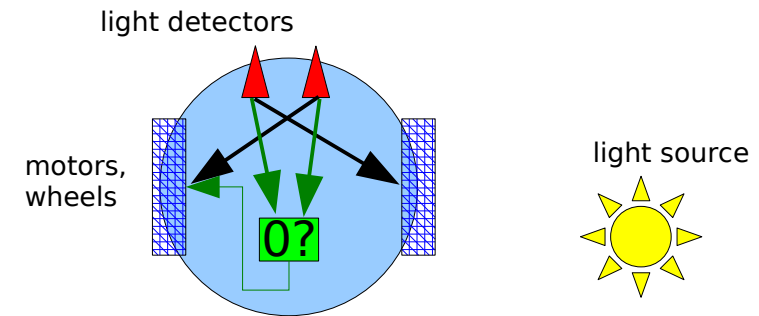
# Light Chasing in a State Machine Formalism

- States treated as equal alternatives.

- State is discrete, but control signal is continuous.

- "Find Light" has to know which state to return control to when the light is found.

- Usually not parallel (but can be).
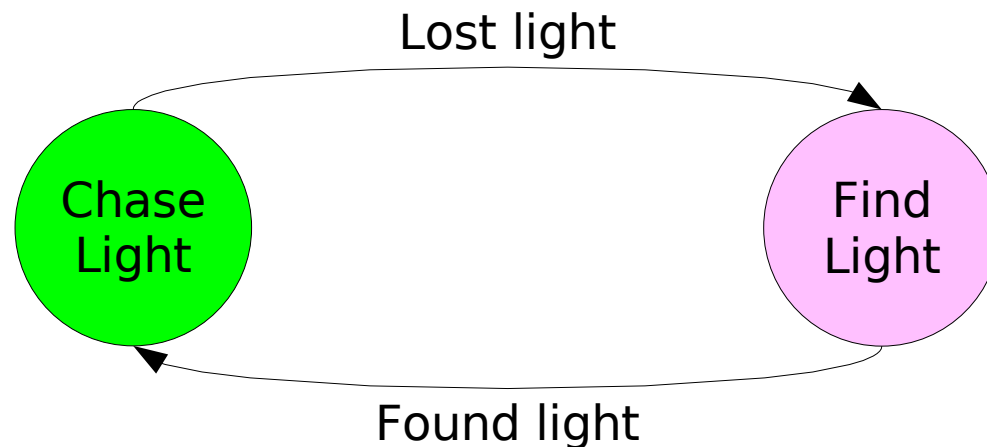
Rod Brooks' Subsumption Idea

- In 1986 Rod Brooks proposed the "subsumption" architecture, a kind of reactive controller.

- Robot control program is a collection of little autonomous modules (state machines).

- Hierarchy of layers of control.

- Some modules override (subsume) inputs or outputs of lower layer modules.
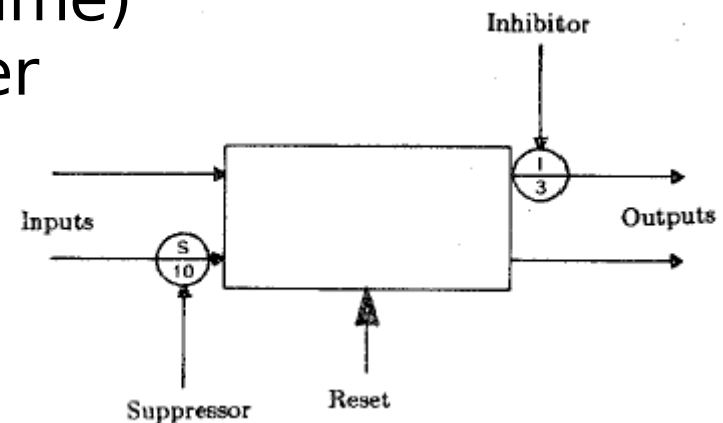
Genghis: Six-Legged Walker

15-494 Cognitive Robotics
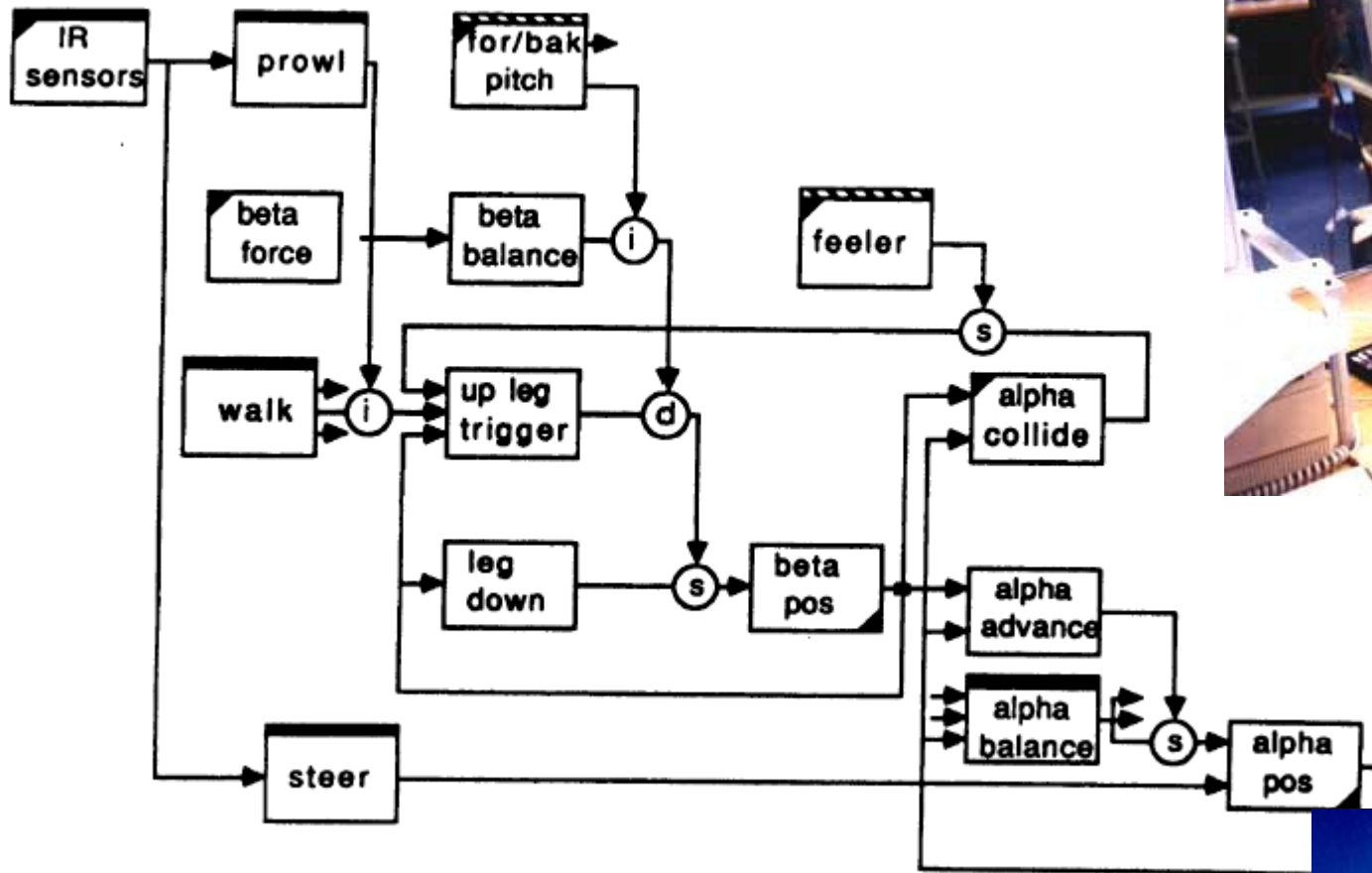
Hannibal (Breazeal)
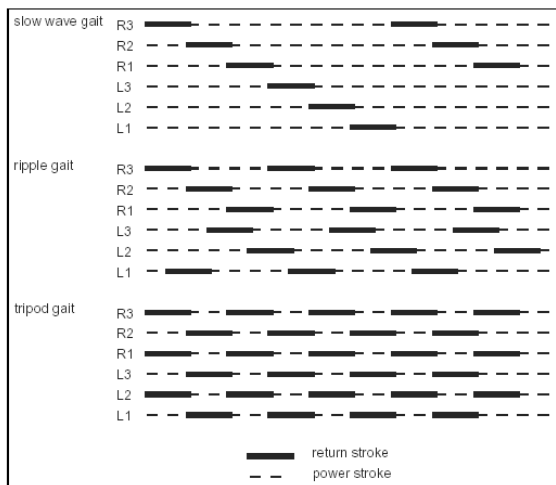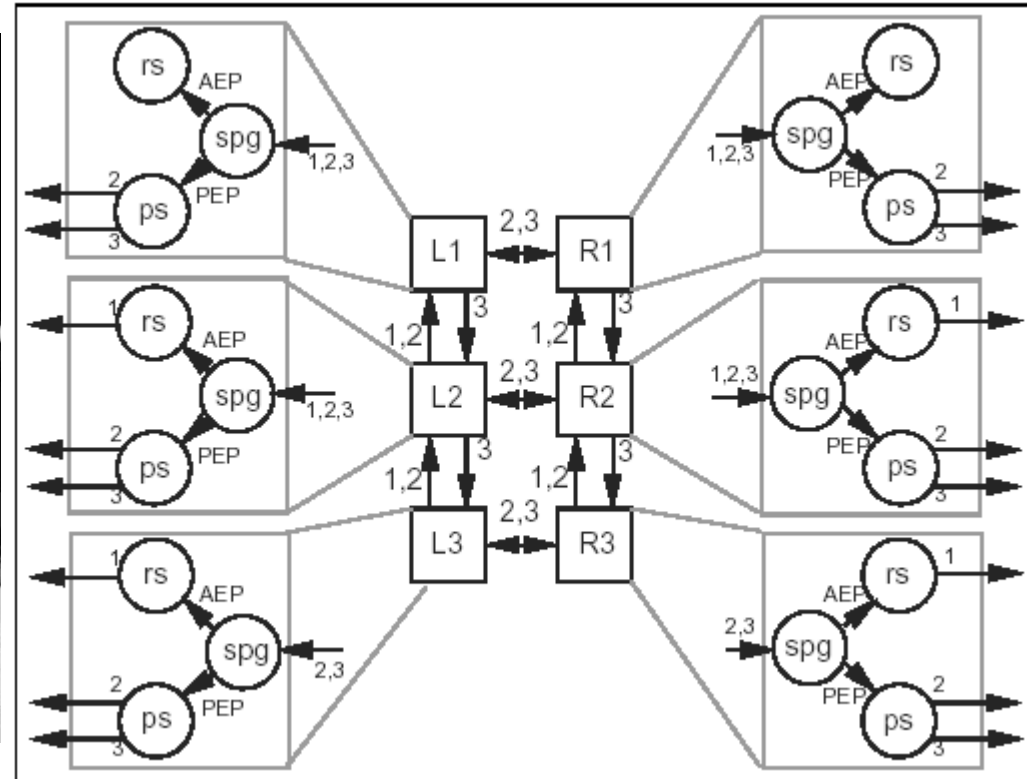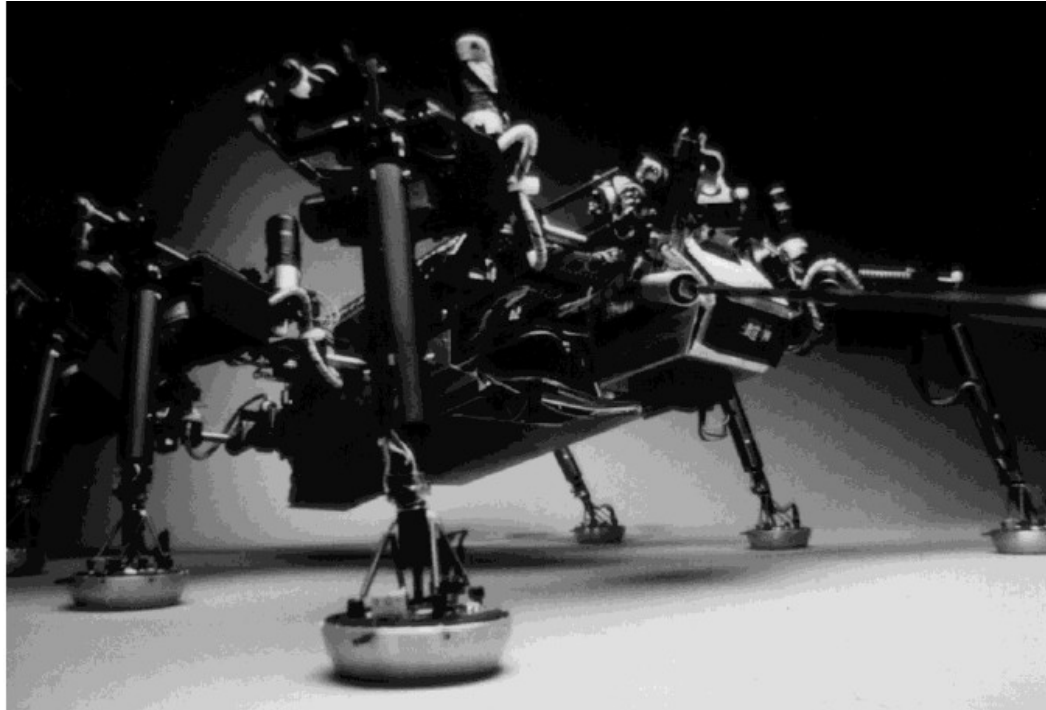
Three Distinct Insect Gaits:
(1) slow wave, (2) ripple,
(3) tripod

Coping With a Noisy World

- URBI (Baillie, 2005) provides a ~ operator to test if a condition has held true for a certain duration.

- Onleave test is true when condition ceases to hold.

- You can build a state machine from these primitives.

```
// Main behavior
whenever (ball.visible ~ 100ms) {
    headPan  = headPan + ball.a * camera.xfov * ball.x &
    headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};

at (!ball.visible ~ 100ms)
    search:  {
        { headPan'n  = 0.5 smooth:1s &
          headTilt'n = 1 smooth:1s } |
        { headPan'n  = 0.5 sin:period ampli:0.5 &
          headTilt'n = 0.5 cos:period ampli:0.5 }
    };
at (ball.visible) stop search;

// Sound behavior
at (ball.visible ~ 100ms) speaker = found
onleave speaker = lost;
```
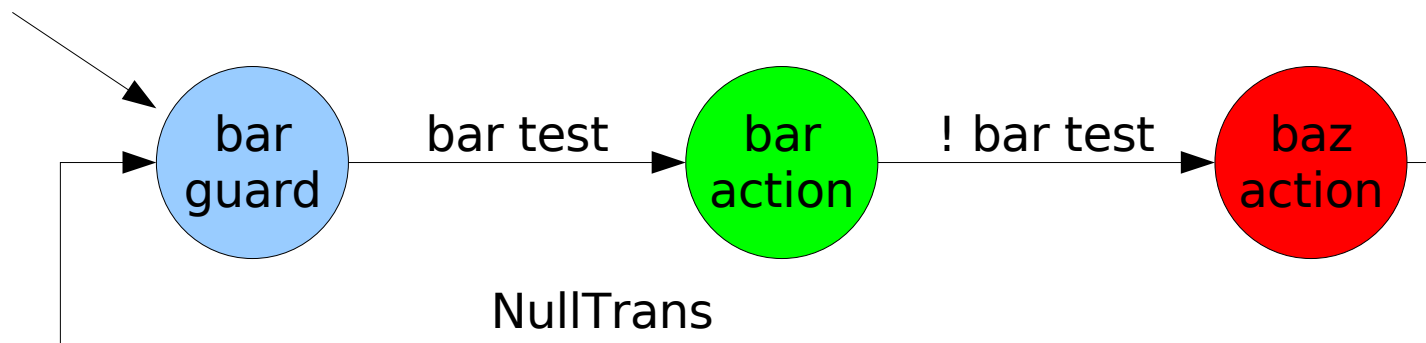
Guarded Commands vs. Finite State Machines

whenever (foo_test) foo_action;
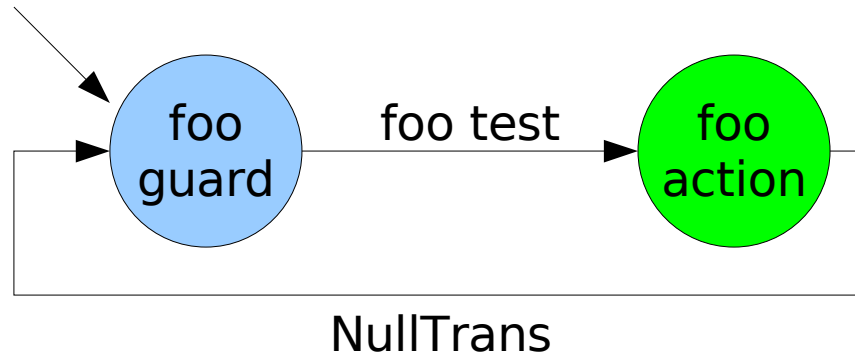
at (bar_test) bar_action; onleave baz_action;

Why Is Complex State Bad?

- Can be expensive to compute (vision)

- Error-prone: what if you make a map, and it's wrong?

- Goes stale quickly: the world constantly changes

- But...

  – Non-trivial intelligent behavior can't be achieved without complex world state.

  – You really do need a map of the environment.

  – Can't use a subsumption architecture to play chess.

  – Or even chase a ball well...

Chase Ball 1

- Cooperation between two simple processes:

    - Point the camera at the ball

    - Walk in the direction the camera is pointing

- Each process can execute independently.

- Purely reactive control.

Chase Ball 2

- If we lose sight of the ball, must look for it.

- Now we introduce some internal state:
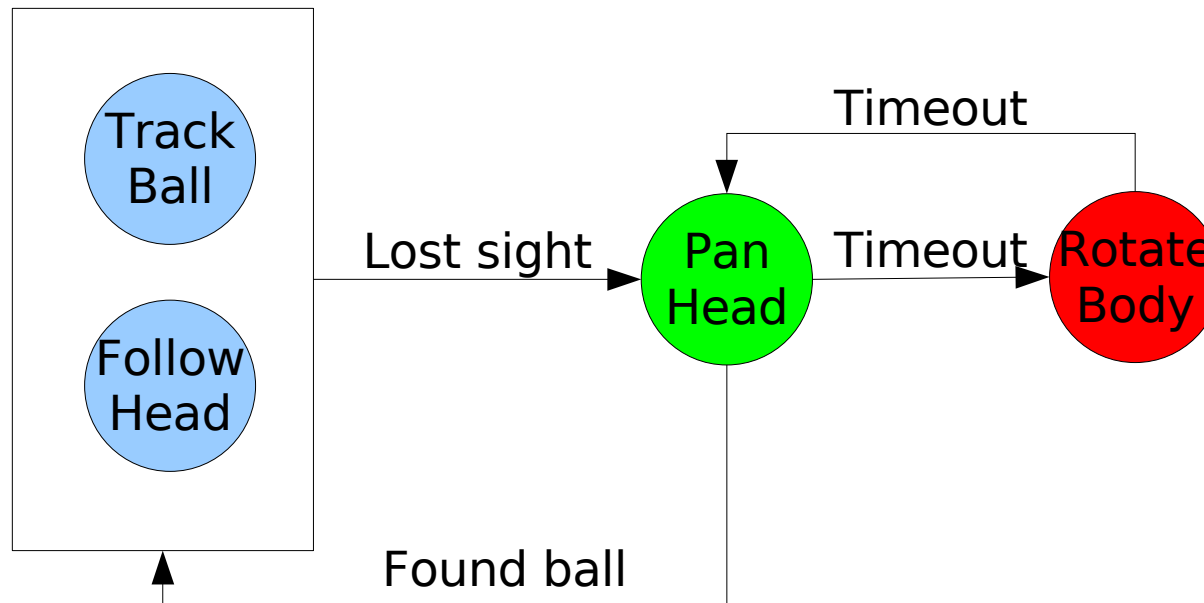
Chase Ball 3

- More intelligent search: direction of turn should depend on where the ball was last seen.

- Now we need to maintain world state (ball location).

Chase Ball 4

- Must avoid obstacles while chasing the ball.
  - May need to move the head to look for obstacles.
  - Attention divided between ball tracking and obstacle checking.

- May need to detour around obstacles.
  - Subgoal "detouring" temporarily overrides "chasing".

- Where will the ball be when the detour is completed?
  - Mapping, trajectory extrapolation…

Say "goodbye" to reactive control!

Mid-Level Control: Task Control Languages

- Takes the robot through a sequence of actions to achieve some simple task.

- Must be able to deal with failures, unexpected events.

- There are many architectures for mid-level control. Various design tradeoffs:

    - Specialized language vs. extensions to Lisp or C

    - Client/server vs. publish/subscribe communication model

    - Provide special exception states, or treat all states the same?

    - How to provide for and manage concurrency.

- Lots of languages/tools: RAPs, TCA, PRS, Propice, ESL, MaestRo, TDL, Orccad, ControlShell, 3T, Circa.

Gat's ESL

- ESL: Execution Support Language (Gat, AAAI 1992; AAAI Fall Symposium, 1996) provides special primitives for handling failures and limiting retries.

```
(defun move-object-to-table ()
  (with-recovery-procedures
      ((:dropped-object :retries 2)
        (locate-dropped-object)
        (retry))
    (pick-up-object)
    (move-to-table)
    (put-down-object)))

(defun pick-up-object ()
  (open-gripper)
  (move-gripper-to-object)
  (close-gripper)
  (raise-arm)
  (if (gripper-empty)
    (fail :dropped-object)))
```

ESL (Continued)

- Cleanup procedures are necessary to ensure safe state after failure.

```
(with-cleanup-procedure
    ((shut-down-motors)
     (close-camera-port))
  (do-some-thing-that-might-fail))
```

- Deadlock prevention: ESL includes "resource locking" primitives for mutual exclusion and deadlock prevention.

- Synchronization: "checkpoints" allow one process to wait until another has caught up.
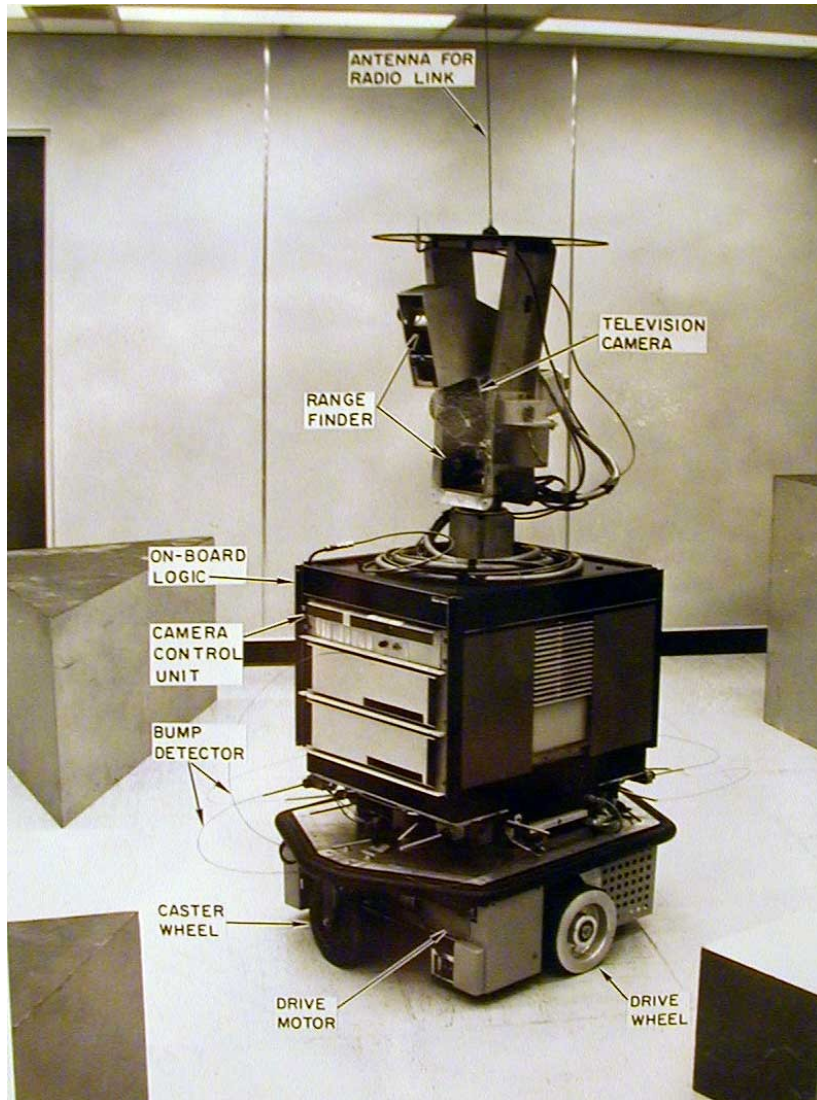
High Level Control: Planning

"Deliberative" architectures may run slowly, infrequently.

- Path planning for navigation.

- Planning as problem solving: achieve A-B-C by moving only one block at a time (gripper can't hold two blocks).
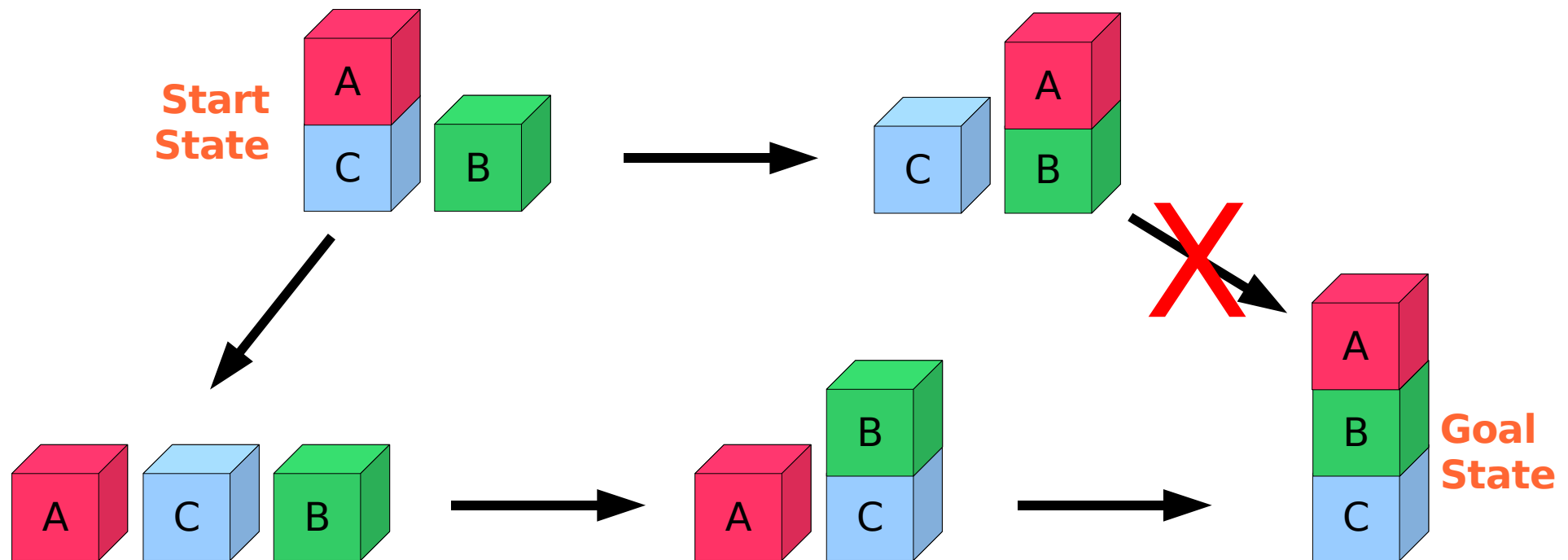
Shakey the Robot (1968) And The STRIPS Planner

Go ...

Go to object bx
GOTOB(bx)

Preconditions:	TYPE(bx,OBJECT),(∃rx)[INROOM(bx,rx) ∧ INROOM(ROBOT,rx)]
Deletions:	AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions:	*NEXTTO(ROBOT,bx)

Go to door dx.
GOTOD(dx)

Preconditions:	TYPE(dx,DOOR),(∃rx)(∃ry)[INROOM(ROBOT,rx) ∧ CONNECTS(dx,rx,ry)]
Deletions:	AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions:	*NEXTTO(ROBOT,dx)

Go to coordinate location (x,y).
GOTOL(x,y)

Preconditions:	(∃rx)[INROOM(ROBOT,rx) ∧ LOCINROOM(x,y,rx)]
Deletions:	AT(ROBOT,$1,$2), NEXTTO(ROBOT,$1)
Additions:	*AT(ROBOT,x,y)

Go through door dx into room rx.
GOTHRUDR(dx,rx)

Preconditions:	TYPE(dx,DOOR), STATUS(dx,OPEN), TYPE(rx,ROOM), NEXTTO(ROBOT,dx) (∃rx)[INROOM(ROBOT,ry) ∧ CONNECTS(dx,ry,rx)]
Deletions:	AT(ROBOT,$1,$2), NEXTTO(ROBOT$1), INROOM(ROBOT,$1)
Additions:	*INROOM(ROBOT,rx)

Really High Level Control

- Can potentially use cognitive modeling architectures such as SOAR (Newell) or ACT-R (Anderson) to control robots.

- RoboSoar (Laird and Rosenbloom, 1990): plan-then-compile architecture.

  - Generate high level plan.

  - Then compile into reactive rules for execution.

- ACT-R has been used in simulated worlds.

- Grubb and Proctor (2006): Tekkotsu interface for ACT-R

Gat's Three-Level Architecture

- Gat (Artificial Intelligence and Mobile Robots, ch. 8, 1998) proposed a different three-level architecture:

- The Controller:

  - collection of reactive "behaviors"

  - each behavior is fast and has minimal internal state

- The Sequencer

  - decides which primitive behavior to run next

  - doesn't do anything that takes a long time to compute, because the next behavior must be specified soon

- The Deliberator

  - slow but smart

  - can either produce plans for the sequencer, or respond to queries from it

What Does Tekkotsu Provide?

- State machine formalism can be used for reactive control or a more hybrid approach.

- Behaviors can execute in parallel; event-based communication follows a publish/subscribe model.

- Main/Motion dichotomy – but Motion is only for ultra-low-level control.

- Could move really slow, higher level deliberative code out of Main to another process.

Tekkotsu Subsystems

- The Lookout controls the head:

  - visual search

  - target tracking

  - obstacle detection

- The Pilot controls the body:

  - walking

  - rotating in place

  - trajectory following

- The Grasper will control the arm

  - grasping, pushing, toppling, flipping, etc.

Potential for Lookout/Pilot Interactions

- The Lookout may need to turn the body in order to conduct a visual search, when head motion alone isn't enough.

  – Lookout makes a request to the Pilot for a turn.

- The Pilot may need to ask the Lookout to locate some landmarks so it can self-localize.

  – Pilot makes a request to the Lookout for a search.

- Interactions must be managed to prevent deadlock, infinite loops.

- But the user shouldn't have to worry about this.

Robot Cooperation

- An even higher level of control is cooperation among multiple robots working as a team.

- Tekkotsu allows robots to communicate by subscribing to each other's events.

```
DoStart:
  int ip = EventRouter::stringToIntIP("172.16.0.4");
  erouter->addRemoteListener(this, ip, EventBase::motmanEGID);

processEvent:
  if ( event.getHostID() == ip )
    cout << "Got remote event " << event.getDescription() << endl;
```

- Can also subscribe to state updates using

  ```
  requestRemoteStateUpdates(ip, type, interval)
  ```

- This is only a low-level form of coordination, but cooperation could be built on top of this.