A Prolog Implementation of IndiGolog for Real Robots

Sebastian Sardina
Cognitive Robotics Groups
Department of Computer Science
University of Toronto
Talk Overview: 3 parts

- Briefly go over P-IndiGolog, a Prolog implementation of the IndiGolog agent architecture

- Introduce EVOLUTION ER1 robot and explain how it can be controlled using P-IndiGolog

- Explain how to implement the simulated Wumpus World scenario using P-IndiGolog
Reactive Robotics Architecture

Reactive robots:

given what is currently sensed and (a small amount of internal state), decide on commands to effectors

\[
\langle \text{sense outlet } \land \text{ power low } \Rightarrow \text{ recharge} \rangle \ ||
\langle \text{sense trash } \Rightarrow \text{ pickup trash} \rangle \ ||
\langle (\text{otherwise}) \Rightarrow \text{ explore} \rangle
\]

Headless!

cf. [Brooks 86], ...
Cognitive Robotics Architecture

- **description of initial state**
- **prereqs & effects of primitive actions**
- **high-level program**
- **sensing results**
- **primitive actions**
- **exogenous events**

- **sensing results**
- **primitive actions**
- **Motion planning**
- **Obstacle avoidance**

- **high-level program interpreter**

- **robotic perception**
- **robotic controller**

- **sensors**
- **effectors**
P-IndiGolog Overview

Cognitive Robotics agent architecture implementation

- Realization of IndiGolog (incremental Golog)
- Based on LeGolog (LEGO® Mindstorm™ with IndiGolog: by Maurice Pagnucco & Hector Levesque)
- Completely implemented in Prolog (SWI/ECLIPSE)

So far, it provides an interleaved framework of

- Execution
- Sensing
- Exogenous events
- Local planning
P-IndiGolog Design

The user/programmer gets to choose:

- The agent programming language (e.g., ConGolog)
  - Define trans/4 and final/2
- The action theory and temporal projector
  - Define eval/3: eval(φ,H,B)
- The external devices/environments used
  - E.g.: specific robot (ER1), simulator, Internet?
- Some other options
  - E.g.: how to handle exogenous events: ignore/abort step?
P-IndiGolog Design (cont.)

It is divided in three main modules:

- **Device managers (rcx, er1, web, simulator, etc.)**
  - Understands each particular device or external environment

- **Main module:**
  - Main cycle
  - Environment manager
  - Transition system + temporal projector

- **Domain application**
  - Theory of action: states the dynamics of the world
  - High-level program: dictates the agent behaviour
  - Domain execution directives
TCP/IP SOCKET COMMUNICATION

- INTERNET
  Environment for the Internet and OS
  env_int.pl

- SIMULATOR
  Environment for simulating actions and exog events
  env_sim.pl

- GOLEM
  Environment for RWI B21
  env_golem.pl

- RCX
  Environment for running the Lego (event_after)
  env_rcx.pl

- ER1
  Environment for running the ER1

Four types of messages:
1. Execute an action
2. Receive sensing outcome
3. Receive exogenous action
4. System messages

ENVIRONMENT MANAGER
Manages all the environments and communicates with the main cycle
env_man.pl

abortedStep/1
indi_exog/1

MAIN CYCLE
IndiGolog main execution cycle
indigolog.pl

executeAction/4
exog_occurs/1

SINGLE STEP
Trans/4 and Final/2 implementation
transfinal.pl

DOMAIN EXEC
Environments to load and how to execute actions
main_xxx.pl

DOMAIN AXIOMS
Actions, effects, initial database, etc.

DOMAIN PROGRAMS
Main program and all necessary procedures

PROJECTOR
Projects a formula wrt some history
eval_ttt.pl
eval/3

axioms of the domain
The Main Cycle

Mandatory Rolling forward?

Collect and handle Exog. Events

Type of step?
- trans
- wait
- test sim. action
- domain action
- execute action
- wait for exog. event
- optional rolling forward?

COMP. NEXT STEP

TERMINATE!

final

exog. event
Main Cycle: first phase *indigo/2*

```
indigo(E,H) :-
    handle_rolling(H,H2), !,
    handle_exog(H2,H3), !,
    mayEvolve(E,H3,E2,H4,S), !,
    (S=trans -> indigo2(H3,E2,H4) ;
     S=final -> report_message(program, 'Success') ;
     S=exog -> report_message(program, 'Restart'), indigo(E,H3) ;
     S=failed -> report_message(program, 'Program fails.')
).
```
mayEvolve/5: transition step...

% Vanilla Prolog (not aware of exog. events happening!)
mayEvolve(E1,H1,E2,H2,S) :- mayEvolve2(E1,H1,E2,H2,S).

mayEvolve2(E1,H1,E2,H2,final) :- final(E1,H1).
mayEvolve2(E1,H1,E2,H2,trans) :- trans(E1,H1,E2,H2).
mayEvolve2(E1,H1,E2,H2,failed).

% SWI/ECLIPSE/SICSTUS Prolog (require events)
mayEvolve(E1,H1,E2,H2,S) :-
    catch(bodyCatch(E1,H1,E2,H2,S)), exogAction, (retractall(flag),S=exog)).

bodyCatch(E1,H1,E2,H2,S) :-
    assert(flag), % Assert flag
    mayEvolve2(E1,H1,E2,H2,S),
    retract(flag). % Retract flag
Main Cycle: second phase

indigo2(H,E,H) :- indigo(E,H). % The case of Trans for tests

indigo2(H,E,[sim(_)|H]) :- !, indigo(E,H). % Drop simulated actions

indigo2(H,_,[abort|H]) :- !, indigo(?(false),H).

indigo2(H,E,[wait|H]) :- !, pause_or_roll(H,H1), % Wait for events!
  doWaitForExog(H1,H2), indigo(E,H2).

indigo2(H,E,[stop_interrupts|H]) :- !, indigo(E,[stop_interrupts|H]).

indigo2(H,E,[A|H]) :- indixeq(A, H, H1), indigo(E, H1).

% Execute action Act at history H, with new history H2
indixeq(Act, H, H2) :-
  type_action(Act, Type), !, % Type=sensing / nonsensing
  execute_action(Act, H, Type, S), !, % Environment manager!
  handle_sensing(Act, [Act|H], S, H2),
  update_now(H2).
Environment Manager

Connects the main cycle with the external world:

- Communicates with every used device manager
  - Uses TCP/IP sockets

- Instructs the execution of actions in devices
  - User states how/where each HL-action is executed
  - Sensing outcome is collected for each action

- Collects exogenous actions from devices
  - Asynchronous
  - Signal main cycle if necessary!
Environment Manager (cont.)

How to implement the EM to run asynchronously?

1. Multi-threads + Events
   - 2 threads: main cycle + environment manager
   - Requires multi-threading support (e.g., SWI)

2. Software signals / interrupts
   - Requires BSD, not too clean...

3. After-events
   - An event is triggered regularly
   - Requires event-after support (e.g., ECLIPSE)
Two Case Examples...

✔ Controlling the EVOLUTION ER1 robot

✔ Controlling an agent in the Wumpus World
Evolution ER1 Robot Platform

Promising as a research tool
- Inexpensive ($1000 + Laptop)
- Easy to set up (USB)
- Easy to upgrade, modular
- **Sensors**: camera, mic, IR
- **Actuators**: motors, speech, gripper
- Wireless connectivity

Two control software tools:
- **RCC**: simple, CAPI (Windows)
- **ERSP**: sophisticated (Linux)
P-IndiGolog on ER1

P-IndiGolog
Agent Architecture

TCP/IP
Sockets

sensing / events

primitive actions

events
gripper status
ir all
ir left/right/center
sense
position
etc...
moves
move rotate
move towards
talk
send email
gripper
open/close
etc...
sensors / events
effectors

Built-in obstacle and collision avoidance
ERSP 3.0

The Evolution Robotics Software Platform 3.0 (ERSP™) is a comprehensive development platform with four primary areas of functionality: vision, obstacle avoidance, interaction, and architecture. ERSP 3.0 includes library APIs, developer tools, and applications to aid you in the robot development process and allow you to move to higher-level programming quickly.
ERSP 3.0: Four Modules

- **ER Vision**
  - Object recognition, motion analysis, and colour segmentation

- **ER Navigation: SLAM**
  - Mapping, localization, and path-planning, obstacle and cliff detection and avoidance with webcams

- **ER Interaction**
  - Software for robot-human interaction (e.g., person detection and tracking, robot emotions)

- **ER Architecture**
  - Infrastructure for Rapid Robot Development & Control
SLAM on ERSP 3.0
Running ER1 with IndiGolog

Fluent “state”:

- causes_val(moveFwd(_), state, moving, true).
- causes_val(turnLeft, state, moving, true).
- causes_val(turnRight, state, moving, true).
- causes_val(arrive, state, stopped, true).
- causes_val(getStuck, state, stopped, true).
- causes_val(stop_abnormally, state, suspended, true).

Action Preconditions:

- poss(moveFwd(_), neg(state=moving)).
- poss(turnRight, neg(state=moving)).
- poss(freeze, true).
- poss(forgetObject(O), or(sawObject(O), objectLost(O))).
- poss(say(_), and(neg(talking), neg(silent)))).
An IndiGolog Controller for ER1

proc(mainControl(3),
    [talk('ER1 controller initiated successfully!'),
    setObjectConfidence(20), senseOn(objects), setPower(moving, 40),
    prioritized_interrupts(
        [interrupt(talking, wait),
        interrupt(o, sawObject(o),
            [talk(['Hey!, I have just seen ', o]), forgetObject(o)]),
        interrupt(o, objectLost(o),
            [talk(['I have just lost the object ', o]), forgetObject(o)]),
        interrupt(state=moving, wait),
        interrupt(true, [talk('Starting a new round'),
            pi(n, [getNumber(10,30,n), setLinearVelocity(n)]),
            rndet(goSquare(right, 200),
                [turnRight, goSquare(left,200), turnLeft]),
            talk('Another round finished')]),
    ])
) % END OF INTERRUPTS    ]).
IndiGolog on ER1: A trace

- trans(p,h,p',[move(2)|h])
- indixeq(move(2), h, H2)
- execute_action(move(2), h, ns, S)
- trans(p',[move(2)|h],p'',[wait,move(2)|h])
- doWaitForEvent([wait,move(2)|h], H3)
- indigo([wait,move(2)|h], [arrived,wait,move(2)|h])

ER1 device manager
env_er1.pl

Socket

Main cycle+
Env. Man.

send_command_to_er1(move(2), R)

Socket

CAPI

move 200 cm
ER1

send_command_to_er1(events, E)

E = “move done”

H3 = [arrived,wait,move(2)|h]
Why ER1 is useful for us?

- It’s simple to deal with (laptop + USB devices)
- Low-level control is already done!
  - Good interface for primitive actions
  - Events management
- Good communication via TCP/IP
  - Not the case with Lego RCX!
- Complex tasks are already implemented
  - Object/colour recognition
  - Obstacle avoidance via IR and camera
  - Object/colour tracking (*move towards* ....)
  - Sound/voice recognition and speech
Wumpus World in IndiGolog

- **Fluents:** locA, dirA, locW, isPit(L), aliveW, noGold, inDungeon, ...
- **Agent actions:** moveFwd, turn, smell, exit, pickGold, shoot, senseBreeze, senseGold
- **Exog. actions:** scream

```
proc mainControl
  ⟨d, l : locW = l ∧ aliveW = true ∧
    aligned(locA, dir, locW) → shoot(d)⟩
  ⟨isGold(locA) = true → pickGold⟩
  ⟨inDungeon = true →
    {smell; senseBreeze; senseGold
     {?(noGold = 0); explore} | {goto(g(1,1)); climb}}⟩
endProc
```
The Wumpus World
Conclusions

- ER1 is a promising tool for research: simple, cheap, and powerful.
- ERSP toolkit can provide an excellent starting point for our Cognitive Robotics applications.
- IndiGolog can be already successfully used to control ER1.

We welcome everybody interested in working with ER1 and IndiGolog! :-}
Interesting Problems with ER1

- Discover ERSP (it's already installed and working!)
- Take full advantage of SLAM and vision capabilities

- Implement a real-world Wumpus World!
- Find known signs/objects in a room, approach them, and read them (e.g., numbers and directions)
  - First look for object color (long range)
  - Can use two behaviors in priorities
  - If nothing can be found, move around the room