Behavior Selection

Gal A. Kaminka

Behavior Selection

Reminder: Behavior Based Control

Two main branches of investigation:

- Behavior Selection (one behavior takes over)
  - Key question 1: How to select?
  - Key question 2: How to de-select?
- Behavior Fusion (combine multiple behaviors)

Our focus here: **Selection**

Behavior Selection: One Behavior at a Time

Remember key questions: How to select? How to de-select?

Behavior Selection: One Behavior at a Time

Remember key questions: How to select? How to de-select?

General ARBITRATE(W,B) behavior selection loop:

```
W knowledge base, g goal, B BEHAVIORS(templates)

C = INSTANTIATE(W,B) // All that can be grounded
c = SELECT(W,C)
START(c)
 While not TERMINATE(c): wait
STOP(c)
```

- INSTANTIATE() grounds behavior based on *W*
  - e.g., a single pass-to-player ungrounded behavior
  - instatiated for all open players–many grounded behaviors

Behavior Selection: One Behavior at a Time

Remember key questions: How to select? How to de-select?

General ARBITRATE(W,B) behavior selection loop:

```
W knowledge base, g goal, B BEHAVIORS(templates)

C = INSTANTIATE(W,B) // All that can be grounded
c = SELECT(W,C)
START(c)
 While not TERMINATE(c): wait
STOP(c)
```

- INSTANTIATE() grounds behavior based on *W*
  - e.g., a single pass-to-player ungrounded behavior
  - instatiated for all open players–many grounded behaviors
- START(), STOP(): start execution, stop it
  - Think: why not EXECUTE()?

Behavior Selection: One Behavior at a Time

Remember key questions: How to select? How to de-select?

General ARBITRATE(W,B) behavior selection loop:

```
W knowledge base, g goal, B BEHAVIORS(templates)
C = INSTANTIATE(W,B) // All that can be grounded
c = SELECT(W,C)
START(c)
 While not TERMINATE(c): wait
STOP(c)
```

- INSTANTIATE() grounds behavior based on *W*
  - e.g., a single pass-to-player ungrounded behavior
  - instatiated for all open players–many grounded behaviors
- START(), STOP(): start execution, stop it
  - Think: why not EXECUTE()?
- **Focus: SELECT(), TERMINATE()**
  - Leaving aside *INSTANTIATE()*

Techniques for SELECT(), TERMINATE()

- Global/centralized evaluation
  - Memoryless: decision lists, decision trees
  - Markovian: (non-) deterministic finite state machines, Petri-nets
  - Memory-based: recipes[1] (e.g., in BDI architectures)
- Local/distributed evaluation
  - Behaviors compete for control of agent
  - Typically: through activation functions

[1] {The association of *recipes* with behavior-based control is non-standard. I take full responsibility.}

Centralized/Global Behavior Selection

State-Based Selection

- Behaviors centrally instantiated
  - Conflicts handled via decision procedures
- Techniques differ by use of memory/context:
  - Memory-less: decision lists, decision trees
  - Markovian: (non-) deterministic finite state machines, Petri-nets
  - Memory-based: *recipes* (e.g., in BDI architectures)

Memory-less selection. . .

SELECT():

- Decision Lists
  - Sequence of IF-THEN rules: If <condition> THEN <behavior>
  - First condition to match selects behavior . . .
- Decision Trees
  - Variables are associated with vertices tree branches with values
  - (Grounded) behaviors set as leaves in tree
  - Variable tests root to leaf, then selects behavior at leaf

TERMINATE() (two alternatives):

- Behaviors self-terminate, **OR**
- Behaviors replaceable if decision changes

Memory-less selection... **A BAD IDEA**

- Decision lists (**Terrible, never use this**):
  - Ordering of rules is critical
  - Very difficult to get right, as system grows in complexity
  - Multiple checks of same variables

Memory-less selection... **A BAD IDEA**

- Decision lists (**Terrible, never use this**):
  - Ordering of rules is critical
  - Very difficult to get right, as system grows in complexity
  - Multiple checks of same variables
- Decision Trees (**Very bad, use only under protest**):
  - No dependency on ordering checks
  - Internal nodes test one variable at a time, depth may vary
  - Can be efficient, where only subset variables checked
    - Not always possible
  - Size of tree can grow exponentially

This is not just for robots!

- Print servers
- GUI client event handlers
- Web-based applications
- Trading agents
- Decision-making support systems

Markovian State-Based Selection

- Key: Add state variable that tracks *state of execution*
  - **NOT** the same as *state of world*
- Execution state determines context
  - Choice between behaviors depends on state of execution
- In general: discrete event systems can be used as model
  - Finite State Machines
  - Petri nets
  - Markov decision processes

We start with finite state machines

Deterministic finite state machines

- Every behavior represented as a state
- Events cause behaviors to SELECT(), TERMINATE()
  - Events computed by perception processes
- Very efficient: focus attention only on needed events

Example: Foraging Robot

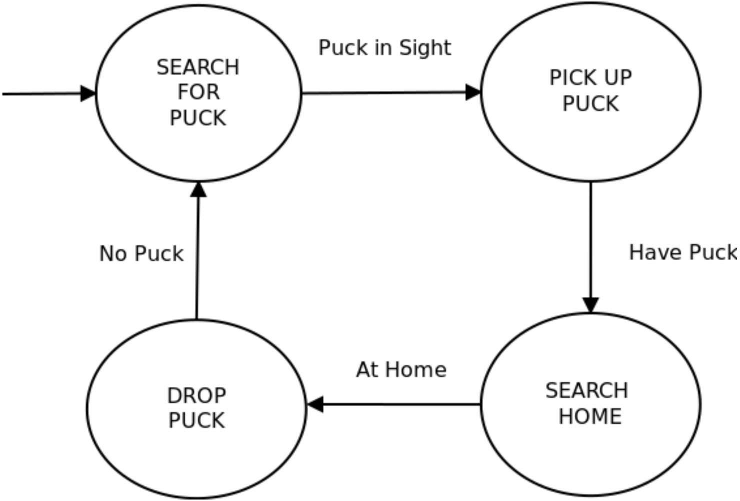

Figure 1: Foraging Robot (Simple)

Example: Foraging Robot (better)

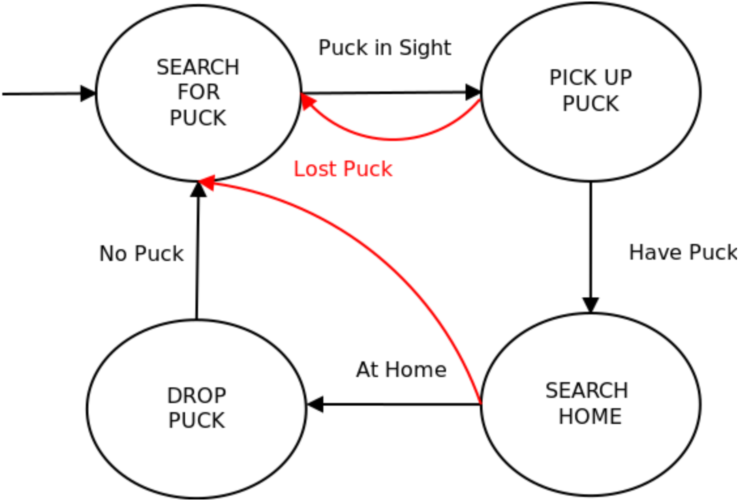

Figure 2: Foraging Robot (Complex)

Deterministic FSM Behavior Agent

```
W knowledge base, g goal, B behavior FSM

Let b = starting behavior in B  // starting state
START(b) // start execution of b
Let W' be PERCEIVE(W) // W' is updated
E = new beliefs in W' // (W'-W)
if E matches event on outgoing transition to b':
    STOP(b) // stop execution of b
    Let b = b'  // update b
    goto 4
goto 5
```

Note this replaces the entire “while goal not achieved” loop

Also, grounded behaviors only. Why?

Deterministic FSM Behavior Agent

```
W knowledge base, g goal, B behavior FSM

Let b = starting behavior in B  // starting state
START(b) // start execution of b
Let W' be PERCEIVE(W) // W' is updated
E = new beliefs in W' // (W'-W)
if E matches event on outgoing transition to b':
    STOP(b) // stop execution of b
    Let b = b'  // update b
    goto 4
goto 5
```

Note this replaces the entire "while goal not achieved" loop

Also, grounded behaviors only. Why?

(Because ungrounded behaviors may have several instantiations—leading to non-determinism on outgoing transitions)

Deterministic FSMs Pros and Cons

- Mechanism easy to implement, efficient
- Focused attention on perception
  - Only looking for events that match transitions
- Works well in practice for *smallish* tasks
  - Simple robot tasks, servers: a few dozen states

Deterministic FSMs Pros and Cons

- Mechanism easy to implement, efficient
- Focused attention on perception
  - Only looking for events that match transitions
- Works well in practice for *smallish* tasks
  - Simple robot tasks, servers: a few dozen states

But: Difficult to work with in large tasks (100s states)

Challenges to scalability of Deterministic FSMs

Reality forces non-determinism

- Ungrounded behaviors
- Incomplete event specs (factored, multiple attributes)
- Opportunism, interruption, interleaving

Hinder Modularity

- Changes require global refactoring
- All partial orderings needed