Chapter 3
Deliberation with Refinement Methods

Dana S. Nau
University of Maryland
Motivation

- Hierarchically organized deliberation
  - At high levels, abstract actions
  - At lower levels, more detail

- Refine abstract actions into ways of carrying out those actions
  - How?

Diagram:
- Respond to user requests
  - Bring o7 to room2
    - Go to hallway
    - Navigate to room1
    - Fetch o7
    - Navigate to room2
    - Deliver o7
  - Move to door
  - Open door
  - Get out
  - Close door
  - Identify type of door
  - Move close to knob
  - Grasp knob
  - Turn knob
  - Maintain
  - Move back
  - Ungrasp
  - Pull
  - Monitor

Planning
Acting
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
  - Hinge on left or right?

![Diagram of door opening process with steps and labels:]
- Identify type of door
- Move close to knob
- Grasp knob
- Turn knob
- Maintain
- Pull
- Monitor
- Move back
- Ungrasp
- Open door
- Get out
- Close door
- Respond to user requests
- Bring object to room
- Go to hallway
- Deliver object
- Navigate to room1
- Navigate to room2
- Move to door
- Fetch object
- Close door
- Identify type of door
- Open door
- Move to door
- Fetch object
- Navigate to room1
- Navigate to room2
- Move to door
- Fetch object
- Close door
- Identify type of door
- Open door
- Move to door
- Fetch object
- Navigate to room1
- Navigate to room2
- Move to door
- Fetch object
- Close door
- Identify type of door
- Open door
- Move to door
- Fetch object
- Navigate to room1
- Navigate to room2
- Move to door
- Fetch object
- Close door
- Identify type of door
- Open door
- Move to door
- Fetch object
- Navigate to room1
- Navigate to room2
- Move to door
- Fetch object
- Close door
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
  - Hinge on left or right?
  - Open toward or away?
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
  - Hinge on left or right?
  - Open toward or away?
  - Knob, lever, push bar, …
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
  - Hinge on left or right?
  - Open toward or away?
  - Knob, lever, push bar, pull handle, push plate, …

- Open door
  - identify type of door
  - move close to knob
  - grasp knob
  - turn knob
  - maintain
  - move back
  - pull
  - pull
  - monitor
  - monitor
  - ungrasp

- Opening a Door
  - go to hallway
  - deliver
  - bring
  - to
  - room2
  - move to door
  - fetch
  - navigate
to
  - room1
  - navigate
to
  - room2
  - get out
  - close door
  - respond to user requests
Opening a Door

- Many different methods, depending on what kind of door
  - Sliding or hinged?
  - Hinge on left or right?
  - Open toward or away?
  - Knob, lever, push bar, pull handle, push plate, something else?
Outline

3.1 Representation
   a. State variables, commands, refinement methods
   b. Example

3.2 Acting
   a. Rae (Refinement Acting Engine)
   b. Example
   c. Extensions

3.3 Planning
   a. Motivation and basic ideas
   b. Deterministic action models
   c. SeRPE (Sequential Refinement Planning Engine)

3.4 Using Planning in Acting
   a. Techniques
   b. Caveats
State-Variable Representation

Quick review:

- **Objects:** $\text{Robots} = \{r_1\}$, $\text{Containers} = \{c_1, c_2, c_3, \ldots\}$, $\text{Locations} = \{\text{loc0, loc1, loc2, \ldots}\}$
- **State variables:** syntactic terms to which we can assign values
  - $\text{loc}(r) \in \text{Locations}$
  - $\text{load}(r) \in \text{Containers} \cup \{\text{nil}\}$
  - $\text{pos}(c) \in \text{Locations} \cup \text{Robots} \cup \{\text{unknown}\}$
  - $\text{view}(r, l) \in \{T, F\}$ – whether robot $r$ has looked at location $l$
    - $r$ can only see what’s at its current location
- **State:** assign a value to each state variable
  - $\{\text{loc}(r_1) = \text{loc0}, \text{pos}(c_1) = \text{loc2}, \text{pos}(c_3) = \text{loc4}, \text{pos}(c_2) = \text{unknown}, \ldots\}$

Details: *Automated Planning and Acting*, Sections 2.1 and 3.1.1
State-Variable Representation

Extensions:

- **Range($x$)**
  - can be finite, infinite, continuous, discontinuous, vectors, matrices, other data structures

- **Assignment statement** $x \leftarrow \text{expr}$
  - expression that returns a ground value in Range($x$) and has no side-effects on the current state

- **Tests (e.g., preconditions)**
  - *Simple*: $x = v$, $x \neq v$, $x > v$, $x < v$
  - *Compound*: conjunction, disjunction, or negation of simple tests
**Commands**

- **Command**: primitive function that the execution platform can perform
  - `take(r,o,l)`: robot `r` takes object `o` at location `l`
  - `put(r,o,l)`: `r` puts `o` at location `l`
  - `perceive(r,l)`: robot `r` perceives what objects are at `l`
    - `r` can only perceive what’s at its current location
  - …
Tasks and Methods

- **Task**: an activity for the actor to perform
- For each task, a set of refinement methods
  - Operational models:
    - tell *how* to perform the task
    - don’t predict *what* it will do

```
method-name(arg1, ..., argk)
task: task-identifier
pre: test
body: a program
```

- assignment statements
- control constructs: if-then-else, while, …
- tasks
  - can extend to include events, goals
- commands to the execution platform
Opening a Door

- What kind:
  - Hinged on left, opens toward us, lever handle

m1-unlatch\((r,d,l,o)\)
- task: \(\text{unlatch}(r,d)\)
- pre: \(\text{loc}(r,l) \land \text{toward-side}(l,d) \land \text{side}(d,\text{left}) \land \text{type}(d,\text{rotate}) \land \text{handle}(d,o)\)
- body:
  - \(\text{grasp}(r,o)\)
  - \(\text{turn}(r,o,\text{alpha1})\)
  - \(\text{pull}(r,\text{val1})\)
  - if \(\text{door-status}(d) = \text{cracked}\) then
    - \(\text{ungrasp}(r,o)\)
  - else fail

m1-throw-wide\((r,d,l,o)\)
- task: \(\text{throw-wide}(r,d)\)
- pre: \(\text{loc}(r,l) \land \text{toward-side}(l,d) \land \text{side}(d,\text{left}) \land \text{type}(d,\text{rotate}) \land \text{handle}(d,o) \land \text{door-status}(d) = \text{cracked}\)
- body:
  - \(\text{grasp}(r,o)\)
  - \(\text{pull}(r,\text{val1})\)
  - \(\text{move-by}(r,\text{val2})\)
Outline

3.1 Representation
   a. State variables, commands, refinement methods
   b. Example

3.2 Acting
   a. Rae (Refinement Acting Engine)
   b. Example
   c. Extensions

3.3 Planning
   a. Motivation and basic ideas
   b. Deterministic action models
   c. SeRPE (Sequential Refinement Planning Engine)

3.4 Using Planning in Acting
   a. Techniques
   b. Caveats
Rae (Refinement Acting Engine)

- Based on OpenPRS
  - Programming language, open-source robotics software
  - Deployed in many applications

- Input: external tasks, events, current state
- Output: commands to execution platform

- Perform multiple tasks/events in parallel
  - Purely reactive, no lookahead

- For each task/event, a refinement stack
  - current path in Rae’s search tree for the task/event
- Agenda = \{all current refinement stacks\}
Rae (Refinement Acting Engine)

Basic idea: \( \text{Agenda} = \{ \text{current refinement stacks} \} \)

loop:
if new external tasks/events then
  for each one, add a refinement stack to \( \text{Agenda} \)
  for each stack in \( \text{Agenda} \)
    Progress it, and remove it if it’s finished

until the input stream of external tasks and events is empty do
  read \( \tau \) in the input stream
  \( \text{Candidates} \leftarrow \text{Instances}(\mathcal{M}, \tau, \xi) \)
  if \( \text{Candidates} = \emptyset \) then output(“failed to address” \( \tau \))
  else do
    arbitrarily choose \( m \in \text{Candidates} \)
    \( \text{Agenda} \leftarrow \text{Agenda} \cup \{ (\tau, m, \text{nil}, \emptyset) \} \)
  for each \( \text{stack} \in \text{Agenda} \) do
    \text{Progress}(\text{stack})
    if \( \text{stack} = \emptyset \) then \( \text{Agenda} \leftarrow \text{Agenda} \setminus \{ \text{stack} \} \)

Stack element \((\tau, \text{m}, i, \text{tried})\)
\( \tau \): task
\( m \): method
\( i \): instruction pointer
\( \text{tried} \): methods already tried
Progress (subroutine)

Progress(stack)

(τ, m, i, tried) ← top(stack)

if i ≠ nil and m[i] is a command then do

case status(m[i])
    running: return
    failure: Retry(stack); return
    done: continue

if i is the last step of m then
    pop(stack)  // remove stack’s top element
else do
    i ← nextstep(m, i)
    case type(m[i])
        assignment: update ξ according to m[i]; return
        command: trigger command m[i]; return
        task or goal: continue

τ′ ← m[i]
Candidates ← Instances(ℳ, τ′, ξ)

if Candidates = ∅ then Retry(stack)
else do
    arbitrarily choose m' ∈ Candidates
    stack ← push((τ', m', nil, ∅), stack)
Poll. Is it possible to have situations where both of these have the answer “no”?
- Objects
  - $Robots = \{r1\}$
  - $Containers = \{c1, c2, \ldots\}$
  - $Locations = \{loc1, loc2, \ldots\}$
- State variables
  - $loc(r) \in Locations$
  - $load(r) \in Containers \cup \{nil\}$
  - $pos(c) \in Locations \cup Robots \cup \{unknown\}$
  - $view(l) \in \{T, F\}$
    - Whether the robot has looked at location $l$ or not
    - If $view(l) = T$ then for every container $c$ at location $l$, $pos(c) = l$
- Commands to the execution platform:
  - $take(r, o, l)$: $r$ takes object $o$ at location $l$
  - $put(r, o, l)$: $r$ puts $o$ at location $l$
  - $perceive(r, l)$: robot $r$ perceives what objects are at location $l$
  - $move-to(r, l)$: robot $r$ moves to location $l$
Example

m-fetch(r,c)
  task: fetch(r,c)
  pre: pos(c) = unknown
  body:
    if pos(c) = unknown then
      search(r,c)
    else if loc(r) = pos(c) then
      take(r,c,pos(c))
    else do
      move-to(r,pos(c))
      take(r,c,pos(c))

m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body:
    if ∃ l (view(r,l) = F) then
      move-to(r,l)
      perceive(l)
      if pos(c) = l then
        take(r,c,l)
      else search(r,c)
    else fail

τ = fetch(r1,c2)
Candidates = {m-fetch(r1,c2)}

Add to Agenda:

<table>
<thead>
<tr>
<th>τ</th>
<th>m</th>
<th>i</th>
<th>tried</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ = fetch(r1,c2)</td>
<td>m-fetch(r1,c2)</td>
<td>nil</td>
<td>(∅)</td>
</tr>
</tbody>
</table>

Rae(\(M\))
Agenda \(\leftarrow ∅\)
loop
  until the input stream of external tasks and events is empty
  read \(τ\) in the input stream
    Candidates \(\leftarrow\) Instances(\(M\), \(τ\), \(ξ\))
    if Candidates = ∅ then output(“failed to address” \(τ\))
    else do
      arbitrarily choose \(m \in\) Candidates
      Agenda \(\leftarrow\) Agenda \(∪\) \{\((\tau, m, \text{nil}, \emptyset)\)\}
      for each stack \(\in\) Agenda do
        Progress(stack)
      if stack = ∅ then Agenda \(\leftarrow\) Agenda \(\setminus\) \{stack\}
Example

**Refinement stack**

\[ \tau: \text{fetch}(r_1,c_2) \]
\[ m: \text{m-fetch}(r_1,c_2) \]
\[ i: \text{nil} \]
\[ \text{tried: } \emptyset \]

**m-fetch\((r;c)\)**

- **task:** fetch\((r;c)\)
- **pre:**
- **body:**
  - if \(\text{pos}(c) = \text{unknown}\) then
    - search\((r;c)\)
  - else if \(\text{loc}(r) = \text{pos}(c)\) then
    - take\((r,c,\text{pos}(c))\)
  - else do
    - move-to\((r,\text{pos}(c))\)
    - take\((r,c,\text{pos}(c))\)

**m-search\((r;c)\)**

- **task:** search\((r,c)\)
- **pre:** \(\text{pos}(c) = \text{unknown}\)
- **body:**
  - if \(\exists l (\text{view}(r,l) = F)\) then
    - move-to\((r,l)\)
    - perceive\((l)\)
    - if \(\text{pos}(c) = l\) then
      - take\((r,c,l)\)
    - else search\((r,c)\)
  - else fail

**Search tree**

- **Progress**
  - \(m'\) started?
  - yes
  - no
    - is \(m'\)’s current step a command?
      - yes
      - no
        - more steps in \(m'\)?
          - yes
          - no
            - no more steps in \(m'\)?
              - yes
                - \(\tau' \leftarrow \text{next step of } m'\)
              - no
                - \(\text{return} \)
                - \(\text{succeeded}\)
                - \(\text{failed}\)
                - \(\text{running}\)
                - \(\text{command status}\)?
                  - yes
                    - \(\text{return}\)
                    - \(\text{succeeded}\)
                    - \(\text{failed}\)
                  - no
                    - \(\text{more steps in } m'\)?
                      - yes
                      - no
                        - yes
                          - pop\((\sigma)\)
                        - no
                          - \(\text{retry } \tau' \text{ using an untried candidate}\)

- type\((\tau')\)
  - task
  - assignment
  - send \(\sigma\) to the execution platform
  - candidates for \(\tau'\)?
    - yes
    - no
      - choose a candidate \(m'\)
      - push \((\tau', m', \ldots)\) onto stack
  - retry \(\tau\) using an untried candidate
Example

### m-fetch(r,c)

**task:** fetch(r,c)

**pre:**

**body:**

if pos(c) = unknown then
  search(r,c)
else if loc(r) = pos(c) then
  take(r,c,pos(c))
else do
  move-to(r,pos(c))
take(r,c,pos(c))

### m-search(r,c)

**task:** search(r,c)

**pre:** pos(c) = unknown

**body:**

if $\exists l$ (view(r,l) = F) then
  move-to(r,l)
  perceive(l)
  if pos(c) = l then
    take(r,c,l)
  else search(r,c)
else fail

---

Progress (stack):

- **started $m'$?**
  - yes
  - no

- **is $m$'s current step a command?**
  - yes
  - no

- **command status?**
  - running
  - failed
  - success

- **more steps in $m$?**
  - yes
  - no

- **$\tau'$ ← next step of $m$**

- **type($\tau'$)**

- **assignment**

- **command**

- **send $\alpha$ to the execution platform**

- **update state**

- **candidates for $\tau'$?**
  - yes
  - no

- **choose a candidate $m'$**

- **push ($\tau'$, $m'$, ...) onto stack**

- **retry $\tau$ using an untried candidate**

---

**Refinement stack**

Candidates = {m-search(r,c)}
m-fetch(r,c)
  task: fetch(r,c)
  pre: pos(c) = unknown
  body:
    if pos(c) = unknown then
      search(r,c)
    else if loc(r) = pos(c) then
      take(r,c,pos(c))
    else do
      move-to(r,pos(c))
      take(r,c,pos(c))

m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body:
    if \( \exists l \) (view(r,l) = F) then
      move-to(r,l)
      perceive(l)
      if pos(c) = l then
        take(r,c,l)
      else search(r,c)
    else fail

\( \tau: \text{search}(r_1,c_2) \)
\( m: \text{m-search}(r_1,c_2) \)
\( i: \text{nil} \)
\( \text{tried} : \emptyset \)

\( \tau: \text{fetch}(r_1,c_2) \)
\( m: \text{m-fetch}(r_1,c_2) \)
\( i: \text{(instruction pointer)} \)
\( \text{tried} : \emptyset \)

Example

Refinement stack

Search tree

Progress(stack):

- \( \tau' \leftarrow \text{next step of } m \)
- assignment
- type(\( \tau' \))
- command
- send \( \tau' \) to the execution platform
- candidates for \( \tau' \)
- yes
- \( \text{retry } \tau \) using an untried candidate
- no
- \( \text{no more steps in } m \)
- pop(\( \sigma \))
- more steps in \( m \)?
- yes
- succeeded
- \( \text{return success} \)
- running
- command status?
- yes
- failed
- \( \text{return failed} \)
- no
- started \( m' \)?
- yes
- \( \text{retry } \tau \) using an untried candidate
- no
Example

\[ m\text{-fetch}(r,c) \]

<table>
<thead>
<tr>
<th>Task</th>
<th>fetch((r,c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>pos((c) = \text{unknown})</td>
</tr>
<tr>
<td>Body</td>
<td>if pos((c) = \text{unknown}) then search((r,c)) else if loc((r) = \text{pos}(c)) then take((r,c,\text{pos}(c))) else do move-to((r,\text{pos}(c))) take((r,c,\text{pos}(c)))</td>
</tr>
</tbody>
</table>

\[ m\text{-search}(r,c) \]

<table>
<thead>
<tr>
<th>Task</th>
<th>search((r,c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>pos((c) = \text{unknown})</td>
</tr>
<tr>
<td>Body</td>
<td>if (\exists l) (view((r,l) = F)) then move-to((r,l)) perceive((l)) if pos((c) = l) then take((r,c,l)) else search((r,c)) else fail</td>
</tr>
</tbody>
</table>

\[ \tau: \text{search} (r_1,c_2) \]
\[ m: \text{m-search} (r_1,c_2) \]
\[ i: \text{(instruction pointer)} \]
\[ \text{tried}: \emptyset \]

Search tree

\[ \tau: \text{fetch} (r_1,c_2) \]
\[ m: \text{m-fetch} (r_1,c_2) \]
\[ i: \text{(instruction pointer)} \]
\[ \text{tried}: \emptyset \]

Progress(stack):

1. is \(m\)'s current step a command?
   - yes
   - no

2. command status?
   - succeeded
   - running
   - failed

3. more steps in \(m\)?
   - yes
   - no

4. \(\tau\) ← next step of \(m\)

5. type(\(\tau\))
   - command
   - assignment

6. send \(a\) to the execution platform
   - yes
   - no

7. update state
   - yes
   - no

8. choose a candidate \(m'\)
   - push \((\tau', m', \ldots)\) onto stack
   - retry \(\tau\) using an untried candidate

9. retry \(\tau\) using an untried candidate

Refinement stack

\[ \text{move-to}(r_1,\text{loc}1) \]

...
Example

m-fetch(r;c)
task: fetch(r;c)
pre: pos(r) = unknown
body: if pos(c) = unknown then
    search(r;c)
else if loc(r) = pos(c) then
    take(r;c,pos(c))
else do
    move-to(r,pos(c))
    take(r;c,pos(c))

m-search(r;c)
task: search(r;c)
pre: pos(c) = unknown
body: if ∃l(view(r,l) = F) then
    move-to(r,l)
    perceive(l)
    if pos(c) = l then
        take(r,c,l)
    else search(r,c)
else fail

Progress(stack):

τ: search(r1,c2)
m: m-search(r1,c2)
i: (instruction pointer)
tried: ∅

τ: fetch(r1,c2)
m: m-fetch(r1,c2)
i: (instruction pointer)
tried: ∅

τ: search(r1,c2)
m: m-fetch(r1,c2)

\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{search}(r1,c2) \]

\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{search}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]

\[ \tau: \text{search}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
\[ \tau: \text{fetch}(r1,c2) \]
Example

m-fetch(r,c)
  task: fetch(r,c)
  pre: pos(c) = unknown
  body:
  if pos(c) = unknown then search(r,c)
  else if loc(r) = pos(c) then take(r,c,pos(c))
  else do move-to(r,pos(c))
       take(r,c,pos(c))

m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body:
  if \( \exists l \text{ (view}(r,l) = F) \) then
    move-to(r,l)
    perceive(l)
  if pos(c) = l then take(r,c,l)
  else fail

\[ \tau: \text{search}(r_1,c_2) \]
\[ \text{m: m-search}(r_1,c_2) \]
\[ \text{i: (instruction pointer)} \]
\[ \text{tried: } \emptyset \]

\[ \tau: \text{fetch}(r_1,c_2) \]
\[ \text{m: m-fetch}(r_1,c_2) \]
\[ \text{i: (instruction pointer)} \]
\[ \text{tried: } \emptyset \]
Example

m-fetch(r,c)
task: fetch(r,c)
pre: pos(c) = unknown
body: if pos(c) = unknown then
  
  search(r,c)
else if loc(r) = pos(c) then
take(r,c,pos(c))
else do
  move-to(r,pos(c))
take(r,c,pos(c))

m-search(r,c)

failed

m-fetch(r,c)
task: fetch(r,c)
pre: pos(c) = unknown
body: if \( \exists l \) (view(r,l) = F) then
  move-to(r,l)
perceive(l)
if pos(c) = l then
take(r,c,l)
else search(r,c)
else fail

\( \tau: \) search(r1,c2)
\( m: \) m-search(r1,c2)
\( i: \) (instruction pointer)
\( \text{tried}: \) \( \emptyset \)

\( \tau: \) fetch(r1,c2)
\( m: \) m-fetch(r1,c2)
\( i: \) (instruction pointer)
\( \text{tried}: \) \( \emptyset \)

Progress(stack):

- started \( m' \)?
  - yes
    - is \( m' \)'s current step a command?
    - yes
      - command status?
        - yes
          - return success
        - no
          - more steps in \( m' \)?
            - yes
              - \( \tau' \) \( \leftarrow \) next step of \( m' \)
            - no
              - pop(\( \sigma \))
    - no
      - no candidates for \( \tau' \)?
        - yes
          - retry \( \tau \) using an untried candidate
        - no
          - \( \tau' \) \( \leftarrow \) next step of \( m \)

Assignment

- type(\( \tau' \))
  - command
    - send \( \alpha \) to the execution platform
      - yes
        - candidates for \( \tau' \)?
          - yes
            - choose a candidate \( m' \)
              - push (\( \tau' \), \( m' \),...) onto stack
            - no
          - no
            - retry \( \tau \) using an untried candidate
        - no
          - \( \tau' \) \( \leftarrow \) next step of \( m \)
      - no
        - \( \tau \) \( \leftarrow \) next step of \( m \)
### Retry (subroutine)

**Progress**(*stack*)

\[
(\tau, m, i, \text{tried}) \leftarrow \text{top}(\text{stack})
\]

if \(i \neq \text{nil}\) and \(m[i]\) is a command then do

  - **case** status\((m[i])\)
    - **running**: return
    - **failure**: Retry(*stack*); return
    - **done**: continue

if \(i\) is the last step of \(m\) then

  - pop(*stack*) // remove *stack*'s top element

else do

  - \(i \leftarrow \text{nextstep}(m, i)\)
  - **case** type\((m[i])\)
    - **assignment**: update \(\xi\) according to \(m[i]\); return
    - **command**: trigger command \(m[i]\); return
    - **task** or **goal**: continue

  - \(\tau' \leftarrow m[i]\)

**Candidates** \(\leftarrow \text{Instances}(\mathcal{M}, \tau', \xi)\)

  - if **Candidates** = \(\emptyset\) then Retry(*stack*)
    - arbitrarily choose \(m' \in \text{Candidates}\)
    - \(\text{stack} \leftarrow \text{push}((\tau, m', \text{nil}, \text{tried}), \text{stack})\)
  - else do

**Retry**(*stack*)

\[
(\tau, m, i, \text{tried}) \leftarrow \text{pop}(\text{stack})
\]

\(\text{tried} \leftarrow \text{tried} \cup \{m\}\)

**Candidates** \(\leftarrow \text{Instances}(\mathcal{M}, \tau, \xi)\) \(\setminus \text{tried}\)

if **Candidates** \(\neq \emptyset\) then do

  - arbitrarily choose \(m' \in \text{Candidates}\)
    - \(\text{stack} \leftarrow \text{push}((\tau, m', \text{nil}, \text{tried}), \text{stack})\)
  - else do

**fail**

**output**("failed to accomplish" \(\tau\))

**Agenda** \(\leftarrow \text{Agenda} \setminus \text{stack}\)

**Repeat**
Example

\text{m-fetch}(r,c)
\text{task: fetch}(r,c)
\text{pre: }
\text{body:}
\text{if pos}(c) = \text{unknown then search}(r,c)
\text{else if loc}(r) = \text{pos}(c) then take(r,c,pos(c))
\text{else do move-to}(r,pos(c))
take(r,c,pos(c))
\text{m-search}(r,c)
\text{task: search}(r,c)
\text{pre: pos}(c) = \text{unknown}
\text{body:}
\text{if } \exists l \ (\text{view}(r,l) = F) \text{ then move-to}(r,l)
\text{perceive}(l)
\text{if pos}(c) = l \text{ then take}(r,c,l)
\text{else search}(r,c)
\text{else fail}

\text{Poll: Is this the same as backtracking?}
1. yes
2. no
Example

- **m-fetch(r,c)** failed
  - task: fetch(r,c)
  - pre: pos(c) = unknown
  - body:
    - if pos(c) = unknown then
      - search(r,c) failed
    - else if loc(r) = pos(c) then
      - take(r,c,pos(c))
    - else do
      - move-to(r,pos(c))
      - take(r,c,pos(c))
  
- **m-search(r,c)** failed
  - task: search(r,c)
  - pre: pos(c) = unknown
  - body:
    - if $\exists l$ (view(r,l) = F) then
      - move-to(r,l) succeeded
    - else if pos(c) = l then
      - take(r,c,l)
    - else search(r,c)
    - else fail

Progress(stack):
- started $m'$?
  - yes
  - no
- is $m$'s current step a command?
  - yes
  - no
- command status?
  - succeeded
  - failed
- more steps in $m$?
  - yes
  - no
- pop($\sigma$)
- $\tau' \leftarrow$ next step of $m$

Refinement stack
- $\tau$: fetch(r1,c2)
- $m$: m-fetch(r1,c2)
- $i$: (instruction pointer)
- tried: $\emptyset$

Search tree
- code execution
- move-to(r1,loc1)
- perceive(loc1)

Assignment
- task
- send $\alpha$ to the execution platform

Update state
- candidates for $\tau'$
- yes
- no

Choose a candidate $m'$
- push ($\tau'$, $m'$,...) onto stack
- retry $\tau$ using an untried candidate
- retry $\tau$ using an untried candidate

Nau – Lecture slides for Automated Planning and Acting
Extensions to Rae

Events:

- method-name\( (a_1, \ldots, a_k) \)  
  - event: event-identifier  
  - pre: \( \text{test} \)  
  - body: program

Rae\( (M) \)  
\[
\text{Agenda } \leftarrow \emptyset \\
\text{loop}
\]

\[
\text{until the input stream of external tasks and events is empty do}
\]

\[
\text{read } \tau \text{ in the input stream}
\]

\[
\text{Candidates } \leftarrow \text{Instances} (M, \tau, \xi)
\]

if Candidates = \emptyset then output (“failed to address” \( \tau \))

else do

\[
\text{arbitrarily choose } m \in \text{Candidates}
\]

\[
\text{Agenda } \leftarrow \text{Agenda} \cup \{ (\tau, m, \text{nil, } \emptyset) \}
\]

for each stack \( \in \text{Agenda} \) do

\[
\text{Progress} (\text{stack})
\]

if stack = \emptyset then Agenda \( \leftarrow \text{Agenda} \setminus \{ \text{stack} \} \)

- Example: an emergency

  - If \( r \) isn’t already handling another emergency, then
    - stop what it’s doing
    - go handle the emergency

\[
\text{m-emergency}(r, l, i)
\]

\[
\text{event: emergency}(l, i)
\]

\[
\text{pre: emergency-handling}(r) = F
\]

\[
\text{body: emergency-handling}(r) \leftarrow T
\]

\[
\text{if load}(r) \neq \text{nil then}
\]

\[
\text{put}(r, \text{load}(r))
\]

\[
\text{move-to}(l)
\]

\[
\text{address-emergency}(l, i)
\]
Extensions to Rae

- **Goals:**
  - Write as a special kind of task
  - `achieve(condition)`

- **Like other tasks, but includes monitoring**
  - Modify Progress
  - If `condition` becomes true before finishing body(`m`), stop early
  - If `condition` isn’t true after finishing body(`m`), fail and try another method

**Example:**

```
method-name(arg1, ..., arg_k)
  task: achieve(condition)
  pre: test
  body: program
```

```
m-fetch(r,c)
  task: fetch(r,c)
  pre:
  body:
    if pos(c) = unknown then
      search(r,c) achieve(pos(c)≠unknown)
    else if loc(r) = pos(c) then
      take(r,c,pos(c))
    else do
      move-to(r,pos(c))
      take(r,c,pos(c))
```

```
Rae(M)

Agenda ← ∅

loop
  until the input stream of external tasks and events is empty do
    read τ in the input stream
    Candidates ← Instances(M, τ, ξ)
    if Candidates = ∅ then output(“failed to address” τ)
    else do
      arbitrarily choose m ∈ Candidates
      Agenda ← Agenda ∪ \{(τ, m, nil, ∅)}
      for each stack ∈ Agenda do
        Progress(stack)
        if stack = ∅ then Agenda ← Agenda \ {stack}
```

Nau – Lecture slides for Automated Planning and Acting
Extensions to Rae

- Concurrent subtasks:
  - refinement stack for each one

- Controlling the progress of tasks:
  - e.g., suspend a task for a while
  - If there are multiple stacks, which ones get higher priority?
    - Application-specific heuristics

- For a task $\tau$, which candidate to try first?
  - Refinement planning

---

**body of a method:**

```
... {concurrent: $\tau_1, \tau_2, \ldots, \tau_n$}
...```

**Agenda** = \{stack$_1$, stack$_2$, ..., stack$_n$\}

**Candidates** = \(\text{Instances}(\tau, \mathcal{M}, \xi)\)
Outline

3.1 Representation
   a. State variables, commands, refinement methods
   b. Example

3.2 Acting
   a. Rae (Refinement Acting Engine)
   b. Example
   c. Extensions

3.3 Planning
   a. Motivation and basic ideas
   b. Deterministic action models
   c. SeRPE (Sequential Refinement Planning Engine)

3.4 Using Planning in Acting
   a. Techniques
   b. Caveats
Motivation

- **Which candidate to choose?**
- **Bad choices may lead to**
  - more costly solutions
  - failure, need to recover
  - unrecoverable failures

- **Idea: look ahead using Rae’s refinement methods**

Rae(\(\mathcal{M}\))

\[\text{Agenda} \leftarrow \emptyset\]

loop

until the input stream of external tasks and events is empty do

read \(\tau\) in the input stream

\[\text{Candidates} \leftarrow \text{Instances}(\mathcal{M}, \tau, \xi)\]

if \(\text{Candidates} = \emptyset\) then output(“failed to address” \(\tau\))

else do

arbitrarily choose \(m \in \text{Candidates}\)

\[\text{Agenda} \leftarrow \text{Agenda} \cup \{((\tau, m, \text{nil}, \emptyset))\}\]

for each \(\text{stack} \in \text{Agenda}\) do

\[\text{Progress}(\text{stack})\]

if \(\text{stack} = \emptyset\) then \(\text{Agenda} \leftarrow \text{Agenda} \setminus \{\text{stack}\}\)

Retry(\(\text{stack}\))

\[(\tau, m, i, \text{tried}) \leftarrow \text{pop}(\text{stack})\]

\(\text{tried} \leftarrow \text{tried} \cup \{m\}\)

\[\text{Candidates} \leftarrow \text{Instances}(\mathcal{M}, \tau, \xi) \setminus \text{tried}\]

if \(\text{Candidates} \neq \emptyset\) then do

arbitrarily choose \(m' \in \text{Candidates}\)

\[\text{stack} \leftarrow \text{push}((\tau, m', \text{nil}, \text{tried}), \text{stack})\]

else do

if \(\text{stack} \neq \emptyset\) then Retry(\(\text{stack}\))

else do

output(“failed to accomplish” \(\tau\))

\[\text{Agenda} \leftarrow \text{Agenda} \setminus \text{stack}\]

\[\text{Progress}(\text{stack})\]

\[(\tau, m, i, \text{tried}) \leftarrow \text{top}(\text{stack})\]

if \(i \neq \text{nil}\) and \(m[i]\) is a command then do

case \(\text{status}(m[i])\)

running: return

failure: \(\text{Retry}(\text{stack})\); return

none: continue

the last step of \(m\) then

\[\text{op}(\text{stack}) \quad \text{// remove stack’s top element}\]

nextstep(\(m, i\))

\[\text{type}(m[i])\]

assignment: update \(\xi\) according to \(m[i]\); return

command: trigger command \(m[i]\); return

ask or goal: continue

\[\text{Candidates} \leftarrow \text{Instances}(\mathcal{M}, \tau', \xi)\]

if \(\text{Candidates} = \emptyset\) then Retry(\(\text{stack}\))

else do

arbitrarily choose \(m' \in \text{Candidates}\)

\[\text{stack} \leftarrow \text{push}((\tau', m', \text{nil}, \emptyset), \text{stack})\]
How to Do the Planning?

- **SeRPE planner** (described in the book)
  - Step-by-step execution of Rae’s refinement methods, except:
    - Doesn’t execute the commands
      - Classical action models to predict outcomes
    - Backtracking search over all possible choices

- **SeRPE generalizes a planner, SHOP, that uses HTN planning**
  - depth-first search through space of refinements
  - body of a method:
    - list of tasks and classical actions

- **Body of a SeRPE method**:
  - the same program Rae uses
  - SeRPE uses it to generate tasks and actions

m-opendoor($r,d,l,h$)
- task: `opendoor($r,d$)`
- pre: $\text{loc}(r) = l \land \text{adjacent}(l,d) \land \text{handle}(d,h)$
- body:
  - while $\neg\text{reachable}(r,h)$ do
    - move-close($r,h$)
    - monitor-status($r,d$)
  - if door-status($d$) = closed then
    - unlatch($r,d$)
    - throw-wide($r,d$)
    - end-monitor-status($r,d$)
How to Do the Planning?

- Backtracking in SHOP:
  - method body: $\langle \tau_1, \ldots, \tau_{i-1}, \tau_i, \ldots, \tau_n \rangle$
  - revert to $s_{i-1}$ and $\tau_{i-1}$

- Backtracking in SeRPE:
  - Backtrack over statements in a computer program
  - Need a language interpreter capable of handling this
  - A group of students worked on this for several months, didn’t get it to work properly

- Additional concern:
  - Classical action models too limited for some of the environments where we’d like to use SeRPE

- Result: stopped work on implementing SeRPE

m-opendoor($r,d,l,h$)

  task: opendoor($r,d$)
  pre: $\text{loc}(r) = l \land \text{adjacent}(l,d) \land \text{handle}(d,h)$
  body:
  while $\neg \text{reachable}(r,h)$ do
    move-close($r,h$)
    monitor-status($r,d$)
    if door-status($d$) = closed then
      unlatch($r,d$)
      throw-wide($r,d$)
  end-monitor-status($r,d$)
Classical Action Models

- **Commands:**
  - `take(r,o,l)`: robot `r` takes object `o` at location `l`
  - `put(r,o,l)`: `r` puts `o` at location `l`
  - `perceive(r,l)`: robot `r` perceives what objects are at `l`
    - can only perceive what’s at its current location

- **Action models:**
  - `take(r,o,l)`
    - pre: `cargo(r) = nil, loc(r) = l, loc(o) = l`
    - eff: `cargo(r) ← o, loc(o) ← r`
  - `put(r,o,l)`
    - pre: `loc(r) = l, loc(o) = r`
    - eff: `cargo(r) ← nil, loc(o) ← l`
  - `perceive(r,l)`: ?

Poll: can we write a reasonable classical action model for perceive?
Why use classical action models?

- Given a state $s$, a classical action $a$ always predicts the same outcome
  - Doesn’t deal with nondeterministic outcomes, partial observability, exogenous events

- Can tolerate prediction errors if they’re infrequent and don’t have severe consequences
  - Much simpler planning algorithms
  - Actor can fix the errors online

- But not if the errors are more frequent and the consequences are worse
  - Need to reason about multiple outcomes
How to Do the Planning?

- SeRPE planner (described in the book)
  - Step-by-step execution of Rae’s refinement methods, except:
    - Doesn’t execute the commands
      - Classical action models to predict outcomes

---

m-opendoor\((r,d,l,h)\)

- task: \textit{opendoor}(r,d)
- pre: \textit{loc}(r) = l \land \textit{adjacent}(l,d) \land \textit{handle}(d,h)
- body:
  - while \neg \textit{reachable}(r,h) do
    - \textit{move-close}(r,h)
    - \textit{monitor-status}(r,d)
  - if \textit{door-status}(d) = \textit{closed} then
    - \textit{unlatch}(r,d)
    - \textit{throw-wide}(r,d)
    - \textit{end-monitor-status}(r,d)

Instead: simulated execution of commands

---

Nau – Lecture slides for Automated Planning and Acting
Is simulated execution feasible?

- AI planning uses descriptive models to address two problems
  - Search space size
    - AI planners search a huge search space, need fast predictions
  - Effort needed to create detailed operational models
    - Especially if you aren’t a domain expert

- How to avoid those problems?
  - Search space size
    - The operational models focus the search
    - Computers much more powerful than 20 years ago ⇒ easier to compute detailed simulations quickly
  - Creating operational models
    - Perhaps use the actor’s control software?

`m-opendoor(r,d,l,h)`

- task: `opendoor(r,d)`
- pre: `loc(r) = l \land adjacent(l,d) \land handle(d,h)`
- body:
  - while  ¬reachable(r,h) do
    - move-close(r,h)
    - monitor-status(r,d)
  - if door-status(d) = closed then
    - unlatch(r,d)
    - throw-wide(r,d)
    - end-monitor-status(r,d)
**RAEplan**

- Simulate RAE on a single task

RAEplan(task $\tau$):

- $Candidates = \{\text{applicable method instances}\}$
- nondeterministically choose $m \in Candidates$
- create refinement stack $\sigma$ for $\tau$ and $m$
- loop while $Progress(\sigma) \neq failure$
  - if $\sigma$ is completed then return $m$
  - return $failure$

Progress($\sigma$):

- started $m$?
  - yes
  - $is m's$ current step a command?
    - yes
      - $running command status?$
        - failed
          - retry $\tau$ using an untried candidate
        - succeeded
          - $tau' \leftarrow next step of m$
    - no
      - $no more steps in m?$
        - yes
          - $tau' \leftarrow next step of m$
        - no
          - $pop(\sigma)$
  - no
    - $tau' \leftarrow next step of m$
**RAEplan**

- Simulate RAE on a single task

**RAEplan**(task $\tau$):

$Candidates = \{\text{applicable method instances}\}$

nondeterministically choose $m \in Candidates$

create refinement stack $\sigma$ for $\tau$ and $m$

loop while $Progress(\sigma) \neq failure$

if $\sigma$ is completed then return $m$

return $failure$

Which candidate to choose next?

How to deal with uncertainty about outcome?

Progress($\sigma$):

- started $m$?
  - yes
  - $is \, m's \, current \, step \, a \, command?$
    - yes
    - $command \, status?$
      - success
      - running
      - failed
    - more steps in $m$?
      - yes
      - $\tau' \leftarrow \text{next step of } m$
      - retry $\tau$ using an untried candidate
    - no
      - pop($\sigma$)
  - no
    - no
    - no
    - no
    - no
    - no
    - ended

start simulation of $a$

which candidate to choose next?

choose a candidate $m'$

push ($\tau'$, $m'$, ...) onto stack

retry $\tau$ using an untried candidate

command

type($\tau'$)

assignment

update state

candidates for $\tau'$?

yes

no

retry $\tau$ using an untried candidate

start simulation

no

success

succeeded

failed

no

running

command status?
**RAEplan**

- Simulate RAE on a single task

RAEplan(task $\tau$):

*Candidates* = {applicable method instances}

- nondeterministically choose $m \in Candidates$
- create refinement stack $\sigma$ for $\tau$ and $m$
- loop while Progress($\sigma$) $\neq$ failure
  - if $\sigma$ is completed then return $m$
  - return *failure*

- Idea 1: depth-first backtracking
  - Backtrack through code execution
    - every method body, every command simulation
  - Every possible point where there’s
    - more than one possible candidate
    - more than one possible outcome

- Backtracking is hard in SeRPE
  - Even harder here

Progress($\sigma$):

- **started $m$?**
  - no
  - is $m$’s current step a command?
    - no
      - more steps in $m$?
        - yes
          - retry $\tau$ using an untried candidate
        - no
          - pop($\sigma$)
    - yes
      - $\tau'$ ← next step of $m$

- command status?
  - failed
  - retry $\tau$ using an untried candidate
  - succeeded
  - more steps in $m$?
    - yes
      - $\tau'$ ← next step of $m$
    - no
      - pop($\sigma$)

- assignment
  - command
    - type($\tau'$)
      - update state
        - task
          - candidates for $\tau'$?
            - yes
              - choose a candidate $m'$
                - push ($\tau'$, $m'$, ... ) onto stack
            - no
              - retry $\tau$ using an untried candidate
RAEplan

- Simulate RAE on a single task

RAEplan(task \(\tau\)):

\(\text{Candidates} = \{\text{applicable method instances}\}\)

nondeterministically choose \(m \in \text{Candidates}\)

create refinement stack \(\sigma\) for \(\tau\) and \(m\)

loop while \(\text{Progress}(\sigma) \neq \text{failure}\)

if \(\sigma\) is completed then return \(m\)

return \text{failure}

- Idea 2: multithreading or multiprocessing
  
  - At every point where we need to choose a candidate, try all of them in parallel

Poll: is this a reasonable approach?
RAEplan

- Simulate RAE on a single task

RAEplan(task $\tau$):

- $Candidates = \{\text{applicable method instances}\}$
- Nondeterministically choose $m \in Candidates$
- Create refinement stack $\sigma$ for $\tau$ and $m$
- Loop while $Progress(\sigma) \neq \text{failure}$
  - If $\sigma$ is completed then return $m$
  - Return $\text{failure}$

- Idea 3: Monte Carlo rollouts
  - Multiple runs
    - Random choices and outcomes in each run
  - Return the $m \in Candidates$ that has the highest expected utility

---

Patra, Traverso, Ghallab, and Nau. *Acting and planning using operational models*. AAAI, 2019

Patra, Mason, Kumar, Traverso, Ghallab, and Nau. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. ICAPS, 2020
Simulating an action

- Simplest case:
  - Descriptive model, e.g., probabilistic action template
    \[ a(x_1, \ldots, x_k) \]
    - pre: …
    - \((p_1)\text{ eff}_1: e_{11}, e_{12}, \ldots\)
    - \((p_m)\text{ eff}_m: e_{m1}, e_{m2}, \ldots\)
    - \(\sum p_i = 1\)
  - Choose randomly, each \text{eff}_i has probability \(p_i\)
  - Use \text{eff}_i to update the current state

- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get predicted effects
Monte Carlo Rollouts

Probabilistic actions (Chapter 5):

- At each state, choose action
- At each action, random outcome

In RAEplan:

- At each task
  - Choose method instance
  - Body generates tasks and commands
- At each command
  - run simulation
  - get simulated outcome
Summary of Experimental Results

Four different domains, different combinations of characteristics

Evaluation criteria:
- Efficiency, successes vs failures, how many retries

Result: planning helps
- Rae operates better with RAEplan than without
- Rae operates better with more planning than with less planning

<table>
<thead>
<tr>
<th>Domain</th>
<th>Dynamic events</th>
<th>Dead ends</th>
<th>Sensing</th>
<th>Robot collaboration</th>
<th>Concurrent tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>EE</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SD</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IP</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Outline

3.1 Representation
   a. State variables, commands, refinement methods
   b. Example

3.2 Acting
   a. Rae (Refinement Acting Engine)
   b. Example
   c. Extensions

3.3 Planning
   a. Motivation and basic ideas
   b. Deterministic action models
   c. SeRPE (Sequential Refinement Planning Engine)

3.4 Using Planning in Acting
   a. Techniques
   b. Caveats
3.4 Acting and Refinement Planning

- Hierarchical acting with refinement planning
  - RAE can call a planner each time it wants to choose a method

- Non-hierarchical actor with refinement planning
  - Refine-Lookahead
  - Refine-Lazy-Lookahead
  - Refine-Concurrent-Lookahead
  - Essentially the same as
    - Run-Lookahead
    - Run-Lazy-Lookahead
    - Run-Concurrent-Lookahead
  - But Lookahead is a refinement planner instead of a classical planner
How to do Lookahead

- **Receding horizon**
  - Cut off search before reaching $g$
    - e.g., depth $d_{max}$ or when we run out of time
  - At leaf nodes, use heuristic function

- **Subgoaling**
  - Instead of planning for $g$, plan for a subgoal $g'$
  - When actor is finished with $g_i$, call planner on next subgoal $g_{i+1}$

- **Sampling**
  - RAEplan’s Monte Carlo rollouts already do this
Example

- **Killzone 2**
  - video game
- **SHOP-style HTN methods**
  - Method body is a sequence of tasks and actions
- Used to plan enemy actions at the squad level
  - Don’t want the best possible plan
  - Need believable actions, very quickly
- Use subgoaling, e.g., “get to shelter”
  - Solution plan is maybe 4–6 actions long
  - Planning is very shallow
    - ≈ just pick an applicable method
    - Perhaps more like RAE than RAEplan?
- Replan several times per second as the world changes
Caveats

• Start in state $s_0$, want to accomplish task $\tau$
  - Refinement method $m$:
    - task: $\tau$
    - pre: $s_0$
    - body: $a_1, a_2, a_3$

• Actor uses Refine-Lookahead
  - Lookahead returns $\pi = \langle a_1, a_2, a_3 \rangle$
  - Actor performs $a_1$, calls Lookahead again
  - No applicable method for $\tau$ in $s_1$, Lookahead returns failure

• Fixes
  (1) When writing refinement methods, make them general enough to work in different states
  (2) In some cases Lookahead might be able to fall back on classical planning until it finds something that matches a method
  (3) Keep snapshot of the search tree at $s_1$, resume next time it’s called
Caveats

- Start in state $s_0$, want to accomplish task $\tau$
  - Refinement method $m$:
    - task: $\tau$
    - pre: $s_0$
    - body: $a_1, a_2, a_3$

- Actor uses Refine-Lazy-Lookahead
  - Lookahead returns $\pi = \langle a_1, a_2, a_3 \rangle$
  - Unexpected event during $a_1$, outcome isn’t $s_1$; actor calls Lookahead again
  - No applicable method for $\tau$ in $s_4$, Lookahead returns failure

- Can use (1), (2) on the previous slide, and (3) with this modification:
  - Keep snapshot of Lookahead’s search tree after each action
    - Restart it immediately after $a_1$, using $s_4$ as current state

- Also: make recovery methods for unexpected states
  - e.g., fix flat tire, get back on the road

Poll: will this also work for Refine-Lookahead?
Summary

- 3.1 Operational models
  - $\xi$ versus $s$, tasks, events,
  - Commands to the execution platform
  - Extensions to state-variable representation
  - Refinement method: name, task/event, preconditions, body
  - Examples: fetch container, emergency, opening a door

- 3.2 Refinement Acting Engine (RAE)
  - Purely reactive: select a method and apply it
  - Rae: input stream, Candidates, Instances, Agenda, refinement stacks
  - Progress: command status, nextstep, type of step
  - Retry: Candidates\backslash tried, comparison to backtracking
  - Refinement trees
  - Concurrent tasks: for each, a refinement stack
  - Goal: achieve(condition), uses monitoring
  - Controlling progress, heuristics
Summary

- **3.3 Refinement planning**
  - plan by simulating RAE on a single external task/event/goal
  - SeRPE uses deterministic actions
  - RAEplan simulates the actor’s commands, does Monte Carlo rollouts

- **3.4 Acting and planning**
  - Refine-Lookahead, Refine-Lazy-Lookahead, Refine-Concurrent-Lookahead
    - Like Run-Lookahead, Run-Lazy-Lookahead, Run-Concurrent-Lookahead
  - Caveats
    - Current state may not be what we expect
    - Possible ways to handle that