Chapter 3

Deliberation with Refinement Methods

Dana S. Nau
University of Maryland
Motivation

- Deliberation is hierarchically organized
  - At high levels, the actions are abstract
  - At lower levels, more detail

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

identify type of door

move close to knob
grip knob
turn knob
maintain
move back
pull
monitor
monitor

open door

Nau – Lecture slides for Automated Planning and Acting
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, …
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?

- Identify type of door
- Move close to knob
- Grasp knob
- Turn knob
- Maintain
- Pull
- Pull handle
- Monitor
- Monitor
- Open door
- Close door
- Get out
- Close
- Monitor
- Monitor
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**: $Robots = \{r_{bt}\}$, $Containers = \{c_1, c_2, c_3, \ldots\}$, $Locations = \{loc_0, loc_1, loc_2, \ldots\}$

- **State variables**: syntactic terms to which we can assign values
  - $loc(r) \in Locations$
  - $load(r) \in Containers \cup \{nil\}$
  - $pos(c) \in Locations \cup Robots \cup \{unknown\}$
  - $view(r,l) \in \{T, F\}$ – 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
  - $\{loc(r_{bt}) = loc_0, pos(c_1) = loc_2, pos(c_3) = loc_4, pos(c_2) = unknown, \ldots\}$

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** $x \leftarrow expr$
  - 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
  - ...

Diagram:
- Tasks \(\xi\) flow to **Acting**.
- Commands flow from **Acting**.
- Events flow from **Environment**.
- Execution Platform connects all elements.
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

```plaintext
method-name(\text{arg}_1, \ldots, \text{arg}_k)

\text{task: task-identifier}
\text{pre: test}
\text{body: a program}
```

*Can extend to include events and goals
Opening a Door

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

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
    - unlash\((r,d)\)
  - throw-wide\((r,d)\)
  - end-monitor-status\((r,d)\)

m1-unlash\((r,d,l,o)\)

- task: unlash\((r,d)\)
- pre: $\text{loc}(r,l) \land \text{toward-side}(l,d) \land$
  - $\text{side}(d,left) \land \text{type}(d,rotate) \land \text{handle}(d,o)$
- body:
  - $\text{grasp}(r,o)$
  - $\text{turn}(r,o,\alpha_1)$
  - $\text{pull}(r,v_1)$
  - if door-status\((d)\) = cracked then $\text{ungrasp}(r,o)$
  - else fail

m1-throw-wide\((r,d,l,o)\)

- task: throw-wide\((r,d)\)
- pre: $\text{loc}(r,l) \land \text{toward-side}(l,d) \land$
  - $\text{side}(d,left) \land \text{type}(d,rotate) \land$
  - $\text{handle}(d,o) \land \text{door-status}(d) = \text{cracked}$
- body:
  - $\text{grasp}(r,o)$
  - $\text{pull}(r,v_1)$
  - $\text{move-by}(r,v_2)$
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:

\[ Agenda = \{ \text{current refinement stacks} \} \]

loop:

- if new external tasks/events then add them to \( Agenda \)
- for each stack in \( Agenda \)
  - Progress it, and remove it if it’s finished

\[ \text{Rae}(M) \]

\[ Agenda \leftarrow \emptyset \]

loop

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

  - read \( \tau \) in the input stream

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

  - if \( \text{Candidates} = \emptyset \) then output(“failed to address” \( \tau \))
  - else do
    - arbitrarily choose \( m \in \text{Candidates} \)
    - \[ Agenda \leftarrow Agenda \cup \{(\tau, m, \text{nil}, \emptyset)\} \]
    - for each \( \text{stack} \in Agenda \) do
      - \( \text{Progress}(\text{stack}) \)
      - if \( \text{stack} = \emptyset \) then \( Agenda \leftarrow Agenda \setminus \{\text{stack}\} \)

Stack element \((\tau,m,i,\text{tried})\)

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

Progress(stack)
(τ, m, i, tried) ← top(stack)
if \( i \neq \text{nil} \) 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 ← Instances(\( \mathcal{M}, \tau', \xi \))
if Candidates = \( \emptyset \) then Retry(stack)
else do
  arbitrarily choose \( m' \in \text{Candidates} \)
  stack ← push((\( \tau', m', \text{nil}, \emptyset \)), stack)
Just a decision tree:
Retry (subroutine)

\[ \text{Retry}(\text{stack}) \]
\[ (\tau, m, i, \text{tried}) \leftarrow \text{pop}(\text{stack}) \]
\[ \text{tried} \leftarrow \text{tried} \cup \{m\} \]
\[ \text{Candidates} \leftarrow \text{Instances}(M, \tau, \xi) \setminus \text{tried} \]
if \( \text{Candidates} \neq \emptyset \) then do
\hspace{1em} arbitrarily choose \( m' \in \text{Candidates} \)
\hspace{1em} \( \text{stack} \leftarrow \text{push}((\tau, m', \text{nil}, \text{tried}), \text{stack}) \)
else do
\hspace{1em} if \( \text{stack} \neq \emptyset \) then \text{Retry}(\text{stack})
\hspace{1em} else do
\hspace{2em} output(“failed to accomplish” \( \tau \))
\hspace{2em} \( \text{Agenda} \leftarrow \text{Agenda} \setminus \text{stack} \)

Another decision tree:

- pop top stack element \((\tau,m,i,tried)\)
- add \( m \) to \( \text{tried} \)

\[ \text{untried candidates for } \tau? \]

- yes
  - choose one
  - push onto stack

- no
  - stack empty?
    - yes
      - fail
    - no
      - Retry
Retry (subroutine)

Retry(stack)
(τ, m, i, tried) ← pop(stack)
tried ← tried ∪ {m}
Candidates ← Instances(M, τ, ξ) \ tried
if Candidates ≠ ∅ then do
  arbitrarily choose m' ∈ Candidates
  stack ← push((τ, m', nil, tried), stack)
else do
  if stack ≠ ∅ then Retry(stack)
  else do
    output("failed to accomplish" τ)
    Agenda ← Agenda \ stack

Another decision tree:

- pop top stack element (τ, m, i, tried)
- add m to tried
- untried candidates for τ?
  - yes
    - choose one
    - push onto stack
  - no
    - stack empty?
      - yes
        - fail
      - no
        - Retry
Example

- Objects
  - $Robots = \{\text{rbt}\}$
  - $Containers = \{\text{c1, c2, ...}\}$
  - $Locations = \{\text{loc1, loc2, ...}\}$

- State variables
  - $\text{loc}(r) \in Locations$
  - $\text{load}(r) \in Containers \cup \{\text{nil}\}$
  - $\text{pos}(c) \in Locations \cup Robots \cup \{\text{unknown}\}$
  - $\text{view}(l) \in \{T, F\}$
    - Whether the robot has looked at location $l$ or not
    - If $\text{view}(l) = T$ then for every container $c$ at location $l$, $\text{pos}(c) = l$

- Commands to the execution platform:
  - $\text{take}(r,o,l)$: $r$ takes object $o$ at location $l$
  - $\text{put}(r,o,l)$: $r$ puts $o$ at location $l$
  - $\text{perceive}(r,l)$: robot $r$ perceives what objects are at location $l$
Example

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

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

Refinement stack

(fetch\((r1,c2), ?, ...\))
Example

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

\text{task: } \text{fetch}(r,c)
\text{pre: }
\text{body:}
\begin{align*}
&\text{if pos}(c) = \text{unknown then} \\
&\hspace{1em} \text{search}(r,c) \\
&\text{else if loc}(r) = \text{pos}(c) \text{ then} \\
&\hspace{1em} \text{take}(r,c,\text{pos}(c)) \\
&\text{else do} \\
&\hspace{2em} \text{move-to}(r,\text{pos}(c)) \\
&\hspace{2em} \text{take}(r,c,\text{pos}(c))
\end{align*}

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

\text{task: } \text{search}(r,c)
\text{pre: } \text{pos}(c) = \text{unknown}
\text{body:}
\begin{align*}
&\text{if } \exists l \ (\text{view}(r,l) = F) \text{ then} \\
&\hspace{1em} \text{move-to}(r,l) \\
&\hspace{1em} \text{perceive}(l) \\
&\hspace{2em} \text{if pos}(c) = l \text{ then} \\
&\hspace{3em} \text{take}(r,c,l) \\
&\hspace{3em} \text{else search}(r,c) \\
&\text{else fail}
\end{align*}

\text{Refinement stack}

\text{fetch}(r1,c2), \text{m-fetch}(r1,c2), \ldots

Nau – Lecture slides for \textit{Automated Planning and Acting}
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

Example

(search\((r1,c2), ?, ...\))
(fetch\((r1,c2), m-fetch\((r1,c2), ...\))

Refinement stack
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 ∃ 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

Refinement stack

(search(r1,c2), m-search(r1,c2), ...)
(fetch(r1,c2), m-fetch(r1,c2), ...)

m-fetch(r1,c2)

search(r1,c2)

m-search(r1,c2)
Example

**m-fetch**(r,c)

- task: `fetch(r,c)`
- pre:
- body:
  - if `pos(c) = unknown`
    - `search(r,c)`
  - else if `loc(r) = pos(c)`
    - `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`
      - `take(r,c,l)`
    - else `search(r,c)`
  - else fail

**Refinement stack**

(search(r1,c2), m-search(r1,c2), ...)
(fetch(r1,c2), m-fetch(r1,c2), ...)

m-fetch(r1,c2)

search(r1,c2)

move-to(r1,loc1)

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

\(\text{Refinement stack}\)

\((\text{search}(r1,c2), \text{m-search}(r1,c2), \ldots)\)
\((\text{fetch}(r1,c2), \text{m-fetch}(r1,c2), \ldots)\)
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 \( \exists l (\text{view}(r,l) = F) \) then
    move-to(r,l)
    perceive(l)
    if pos(c) = l then
      take(r,c,l)
    else
      search(r,c)
    else
      fail

Refinement stack

(sensor failure)

Nau – Lecture slides for Automated Planning and Acting
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

• If other candidates for search(r1,c2), try them
• Not same as backtracking
  ➢ Different current state

(fetch(r1,c2), m-fetch(r1,c2), ...)
(search(r1,c2), ?, ...)
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 \( \exists l (\text{view}(r,l) = F) \) then
      move-to(r,l)
      perceive(l)
      if pos(c) = l then
        take(r,c,l)
      else search(r,c)
    else fail

(fetch(r1,c2), m-fetch(r1,c2), ...)

Refinement stack

Sensor failure

move-to(r1,loc1) perceive(loc1)
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\((r,c,\text{pos}(c))\)
    else do
      move-to\((r,\text{pos}(c))\)
      take\((r,c,\text{pos}(c))\)

m-search\((r,c)\)
  task: search\((r,c)\)
  pre: \(\text{pos}(c)\) = 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

Example

- If other candidates for fetch\((r_1,c_2)\), try them
- Not same as backtracking
  - Different current state
Extensions to RAE

Events:

```
method-name(arg₁, …, argₖ)
  event: event-identifier
  pre: test
  body: program
```

Example: an emergency

- If r isn’t already handing another emergency, then
  - stop what it’s doing
  - go handle the emergency

```plaintext
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 a
    else do
      arbitrarily choose m ∈ Candidates
      Agenda ← Agenda ∪ {⟨(τ, m, nil, ∅)⟩}
    for each stack ∈ Agenda do
      Progress(stack)
      if stack = ∅ then Agenda ← Agenda \ {stack}
      m-emergency(r,l,i)
      event: emergency(l,i)
      pre: emergency-handling(r) = F
      body: emergency-handling(r) ← T
      if load(r) ≠ nil then
        put(r,load(r))
        move-to(l)
        address-emergency(l,i)
```

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

Goals:

\[
\text{method-name}(arg_1, \ldots, arg_k)
\]
\[
\text{task: achieve(condition)}
\]
\[
\text{pre: test}
\]
\[
\text{body: program}
\]

Rae(\mathcal{M})

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

loop

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

- Write goal 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
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:

\[ \ldots \]
\[ \{ \text{concurrent: } \tau_1, \tau_2, \ldots, \tau_n \} \]
\[ \ldots \]

Agenda = \{stack_1, stack_2, \ldots, stack_n\}

Candidates = 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

- When dealing with an event or task, Rae may need to make either/or choices
  - Agenda: tasks $\tau_1, \tau_2, \ldots, \tau_n$
    - Several tasks/events, how to prioritize?
  - Candidates for $\tau_1$: $m_1, m_2, \ldots$,
    - Several candidate methods or commands, which one to try first?
- Rae immediately executes commands
  - Bad choices may be costly
    - or irreversible
Refinement Planning

- Basic idea:
  - Go step by step through Rae, but don’t send commands to execution platform
  - For each command, use a descriptive action model to predict the next state
    - Tells *what*, not *how*
  - Whenever we need to choose a method
    - Try various possible choices, explore consequences, choose best

- Generalization of HTN planning
  - HTN planning: body of a method is a list of tasks
  - Here: body of method is the same program Rae uses
  - Use it to *generate* a list of tasks
Example

- Suppose we learn in advance that the sensor isn’t available
  - Planner infers that m-search(r1,c2) will fail
  - If another method is available, use it
  - Otherwise, planner will infer that the actor can’t do search(r1,c2)
Descriptive Action Models

- Predict the outcome of performing a command
  - Preconditions-and-effects representation

- **Command:**
  - `take(r, o, l)`: robot $r$ takes object $o$ at location $l$

- **Action model**

  ```
  take(r, o, l)
  pre: cargo(r) = nil, loc(r) = l, loc(o) = l
  eff: cargo(r) ← o, loc(o) ← r
  ```
Descriptive Action Models

- Predict the outcome of performing a command
  - Preconditions-and-effects representation

- **Command:**
  - `take(r;o;l)`:
    - Robot `r` takes object `o` at location `l`
  - `put(r;o;l)`:
    - `r` puts `o` at location `l`

- **Action model**
  - `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`
Descriptive Action Models

- Predict the outcome of performing a command
  - Preconditions-and-effects representation

**Command:**

- 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 sees what objects are at l
  - can only perceive what’s at its current location

**Action model**

- 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):
  - ?
  - If we knew this in advance, perception wouldn’t be necessary

Can’t do the *fetch* example
Limitation

- Most environments are inherently nondeterministic
  - Deterministic action models won’t always make the right prediction
- Why use them?
  - Deterministic models => much simpler planning algorithms
    - Use when errors are infrequent and don’t have severe consequences
    - Actor can fix the errors online
Planning/Acting at Different Levels

- Deterministic models may work better at some levels than others

- May want
  - Rae at some levels
  - Rae+planner at some levels
  - planner at some levels

- In some cases, might want the planner to reason about nondeterministic outcomes
  - Chapters 5 and 6
Simple Deterministic Domain

- Robot can move containers

- Action models:

  load\( (r,c,c',p,d) \)
  \[
  \text{pre: } \text{at}(p,d), \text{cargo}(r)=\text{nil}, \text{loc}(r)=d, \text{pos}(c)=c', \text{top}(p)=c
  \]
  \[
  \text{eff: } \text{cargo}(r) \leftarrow c, \text{pile}(c) \leftarrow \text{nil}, \text{pos}(c) \leftarrow r, \text{top}(p) \leftarrow c'
  \]

  unload\( (r,c,c',p,d) \)
  \[
  \text{pre: } \text{at}(p,d), \text{pos}(c)=r, \text{loc}(r)=d, \text{top}(p)=c'
  \]
  \[
  \text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{pile}(c) \leftarrow p, \text{pos}(c) \leftarrow c', \text{top}(p) \leftarrow c
  \]

  move\( (r,d,d') \)
  \[
  \text{pre: } \text{adjacent}(d,d'), \text{loc}(r)=d, \text{occupied}(d')=\text{false}
  \]
  \[
  \text{eff: } \text{loc}(r)=d', \text{occupied}(d)=\text{false}, \text{occupied}(d')=\text{true}
  \]
Tasks and Methods

- Task: put-in-pile(c,p') – put c into pile p' if it isn’t there already

m1-put-in-pile(c,p')
  task: put-in-pile(c,p')
  pre: pile(c)=p'
  body: // empty

If c is already in p', do nothing

m2-put-in-pile(r,c,p,d,p',d')
  task: put-in-pile(c,p')
  pre: pile(c)=p \land at(p,d) \land at(p',d') \land p \neq p' \land cargo(r)=nil
  body: if loc(r) \neq d then navigate(r,d)
  uncover(c)
  load(r, c, pos(c), p, d)
  if loc(r) \neq d' then navigate(r,d')
  unload(r, c, top(p'), p', d)

If c isn’t in p'
  - find a route to c, follow it to c
  - uncover c, load c onto r
  - move to p', unload c
Tasks and Methods

- Task: uncover(c) – remove everything that’s on c

m1-uncover(c)

  task: uncover(c)
  pre: top(pile(c))=c
  body: // empty

If nothing is on c, do nothing

m2-uncover(r,c,c,p',d)

  task: uncover(c)
  pre: pile(c)=p ∧ top(p)≠c
       ∧ at(p,d) ∧ at(p',d) ∧ p'≠p
       ∧ loc(r)=d ∧ cargo(r)=nil
  body: while top(p) ≠ c do
        c' ← top(p)
        load(r,c',pos(c'),p,d)
        unload(r,c',top(p'),p',d)

while something is on c

  ➢ remove whatever is at the top of the stack
SeRPE (Sequential Refinement Planning Engine)

\[ \text{SeRPE}(M, A, s, \tau) \]

- Candidates ← Instances(M, \tau, s)
- if Candidates = \emptyset then return failure
- nondeterministically choose \( m \in \text{Candidates} \)
- return Progress-to-finish(M, A, s, \tau, m)

\[ \text{Rae}(M) \]

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

- Which candidate method for \( \tau \)?
- SeRPE: Nondeterministic choice
  - backtracking point
- Implementation:
  - hierarchical adaptation of backtracking, A*, GBFS, …

- Which candidate method for \( \tau \)?
- RAE: Arbitrary choice
  - no search, purely reactive
SeRPE (Sequential Refinement Planning Engine)

SeRPE($\mathcal{M}, \mathcal{A}, s, \tau$)

- Candidates $\leftarrow$ Instances($\mathcal{M}, \tau, s$)
- if Candidates $= \emptyset$ then return failure
- nondeterministically choose $m \in$ Candidates
- return Progress-to-finish($\mathcal{M}, \mathcal{A}, s, \tau, m$)

\[\begin{align*}
\mathcal{M} &= \{\text{methods}\} \\
\mathcal{A} &= \{\text{action models}\} \\
s &= \text{initial state} \\
\tau &= \text{task or goal}
\end{align*}\]

Rae($\mathcal{M}$)

- Agenda $\leftarrow \emptyset$
- loop
- until the input stream of external tasks and events is empty do
- \hspace{1cm} read $\tau$ in the input stream
- \hspace{1cm} Candidates $\leftarrow$ Instances($\mathcal{M}, \tau, \xi$)
- \hspace{1cm} if Candidates $= \emptyset$ then output("failed to address" $\tau$)
- \hspace{1cm} else do
- \hspace{2cm} arbitrarily choose $m \in$ Candidates
- \hspace{2cm} Agenda $\leftarrow$ Agenda $\cup \{(\tau, m, \text{nil}, \emptyset)\}$
- \hspace{1cm} for each stack $\in$ Agenda do
- \hspace{2cm} Progress(stack)
- \hspace{2cm} if stack $= \emptyset$ then Agenda $\leftarrow$ Agenda $\setminus \{\text{stack}\}$

- Multiple tasks, multiple stacks
- Each time through loop, progress each by one step

- Just one task, one stack
- Progress it as many steps as possible
RAE’s Progress subroutine

\[
\text{Progress}(stack) \\
(\tau, m, i, \text{tried}) \leftarrow \text{top}(stack) \\
\text{if } i \neq \text{nil} \text{ and } m[i] \text{ is a command then do} \\
\quad \text{case status}(m[i]) \\
\quad \quad \text{running: return} \\
\quad \quad \text{failure: } \text{Retry}(stack); \text{return} \\
\quad \quad \text{done: continue} \\
\text{if } i \text{ is the last step of } m \text{ then} \\
\quad \text{pop}(stack) \quad // \text{ remove stack’s top element} \\
\text{else do} \\
\quad i \leftarrow \text{nextstep}(m, i) \\
\quad \text{case type}(m[i]) \\
\quad \quad \text{assignment: update } \xi \text{ according to } m[i]; \text{return} \\
\quad \quad \text{command: trigger command } m[i]; \text{return} \\
\quad \quad \text{task or goal: continue} \\
\quad \tau' \leftarrow m[i] \\
\] Candidates \leftarrow \text{Instances}(M, \tau', \xi) \\
\text{if } \text{Candidates} = \emptyset \text{ then } \text{Retry}(stack) \\
\text{else do} \\
\quad \text{arbitrarily choose } m' \in \text{Candidates} \\
\quad \text{stack } \leftarrow \text{push}((\tau', m', \text{nil}, \emptyset), \text{stack})
\]

Just a decision tree:

- **Progress-to-finish** simulates calling this repeatedly

\[
\begin{align*}
\text{no} & \quad m[i] \quad \text{failed} \\
& \quad \text{return} \\
& \quad \text{yes} \\
& \quad \text{more steps?} \\
& \quad \text{no} \\
& \quad \text{pop stack} \\
& \quad \text{yes} \\
& \quad i \leftarrow \text{next step} \\
\end{align*}
\]

- assignment
- \(m[i]\)'s type?
- command
- task or goal
- candidates for \(m[i]\)?
- no
- Retry
- yes

\[
\begin{align*}
\text{update } \xi & \\
\text{trigger it} & \\
\text{no} & \\
\text{yes} & \\
\text{choose candidate } m' \\
\text{push } (m[i], m', \text{nil}, \emptyset) \text{ onto stack}
\end{align*}
\]
SeRPE’s Progress-to-finish

\[ \text{Progress-to-finish}(\mathcal{M}, \mathcal{A}, s, \tau, m) \]
\[ i \leftarrow \text{nil} \quad \text{// instruction pointer for body}(m) \]
\[ \pi \leftarrow \emptyset \quad \text{// plan produced from body}(m) \]
loop
  \[ \text{if } \tau \text{ is a goal and } s \models \tau \text{ then return } \pi \]
  \[ \text{if } i \text{ is the last step of } m \text{ then} \]
  \[ \quad \text{if } \tau \text{ is a goal and } s \not\models \tau \text{ then return failure} \]
  \[ \quad \text{return } \pi \]
\[ i \leftarrow \text{nextstep}(m, i) \]
\[ \text{case } \text{type}(m[i]) \]
  \[ \text{assignment: update } s \text{ according to } m[i] \]
  \[ \text{command:} \]
  \[ a \leftarrow \text{the descriptive model of } m[i] \text{ in } \mathcal{A} \]
  \[ \text{if } s \models \text{pre}(a) \text{ then} \]
  \[ \quad s \leftarrow \gamma(s, a); \quad \pi \leftarrow \pi . a \]
  \[ \text{else return failure} \]
\[ \text{task or goal:} \]
  \[ \pi' \leftarrow \text{SeRPE}(\mathcal{M}, \mathcal{A}, s, m[i]) \]
  \[ \text{if } \pi' = \text{failure} \text{ then return failure} \]
  \[ s \leftarrow \gamma(s, \pi'); \quad \pi \leftarrow \pi . \pi' \]

\[ s = \text{initial state} \]
\[ \tau = \text{task or goal} \]
\[ m = \text{chosen method} \]

- Simulate RAE’s goal monitoring
- If \( m[i] \) is a command
  - Predict outcome using a descriptive action model
- If current step is a task
  - Call SeRPE recursively
  - SeRPE looks for a plan whose simulated execution succeeds
- For failures, don’t have Rae’s Retry
  - If SeRPE failed, this means it couldn’t find a solution
  - Implementation: hierarchical adaptations of backtracking, A*, GBFS, …
$\tau = \text{put-in-pile}(c_1, p_2)$

Candidates = \{\text{m1-put-in-pile}(c_1, p_2),
               \text{m2-put-in-pile}(r_1, c_1, p_1, d_1, p_2, d_2), \ldots \}

SeRPE($\mathcal{M}, \mathcal{A}, s, \tau$)
Candidates $\leftarrow$ Instances($\mathcal{M}, \tau, s$)
if Candidates = $\emptyset$ then return failure
non-deterministically choose $m \in$ Candidates
return Progress-to-finish($\mathcal{M}, \mathcal{A}, s, \tau, m$)

s_0 = \{loc(r_1)=d_1, \text{cargo}(r_1)=\text{nil}, \text{occupied}(d_1)=\text{T},
          \text{occupied}(d_2)=\text{F}, \text{occupied}(d_3)=\text{F},
          \text{pos}(c_1)=\text{nil}, \text{pos}(c_2)=c_3, \text{pos}(c_3)=\text{nil},
          \text{pile}(c_1)=p_1, \text{pile}(c_2)=p_2, \text{pile}(c_3)=p_2,
          \text{top}(p_1)=c_1, \text{top}(p_2)=c_2, \text{top}(p_3)=\text{nil}\}$
**Example**

Task

\[ \text{put-in-pile}(c_1, p_2) \]

Method

\[ \text{m2-put-in-pile}(r_1, c_1, p_1, d_1, p_2, d_2) \]

Bottom of refinement tree = top of refinement stack

### SeRPE(\(M, A, s, \tau\))

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

if \(\text{Candidates} = \emptyset\) then return failure

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

return Progress-to-finish(\(M, A, s, \tau, m\))

Implementation would use partially instantiated expressions

- Instantiate \(c = c_1, p' = p_2\) here
- Instantiate the other variables here

---

**Diagram**

- \(r, c, p, d, p', d'\)
- \(r_1, c_1, p_1, d_1, p_2, d_2\)
- \(\text{m2-put-in-pile}(r, c, p, d, p', d')\)
- Task: \(\text{put-in-pile}(c, p')\)
- Pre: \(\text{pile}(c) = p \land \text{at}(p, d) \land \text{at}(p', d) \land p \neq p' \land \text{cargo}(r) = \text{nil}\)
- Body: if \(\text{loc}(r) \neq d\) then \(\text{navigate}(r, d)\)
  - \(\text{uncover}(c)\)
  - \(\text{load}(r, c, \text{pos}(c), p, d)\)
- if \(\text{loc}(r) \neq d'\) then
  - \(\text{navigate}(r, d')\)
- \(\text{unload}(r, c, \text{top}(p'), p', d)\)
**task**

\[ \text{put-in-pile}(c_1, p_2) \]

**method**

\[ \text{m2-put-in-pile}(r_1, c_1, p_1, d_1, p_2, d_2) \]

Bottom of refinement tree = top of refinement stack

---

\[ r_1, c_1, p_1, d_1, p_2, d_2 \]

**m2-put-in-pile**\( (r, c, p, d, p', d') \)

**task:** \( \text{put-in-pile}(c, p') \)

**pre:** \[ \text{pile}(c) = p \land \text{at}(p, d) \land \text{at}(p', d) \land p \neq p' \land \text{cargo}(r) = \text{nil} \]

**body:**

- if \( \text{loc}(r) \neq d \) then navigate\( (r, d) \)
- uncover\( (c) \)
- load\( (r, c, \text{pos}(c), p, d) \)
- if \( \text{loc}(r) \neq d' \) then navigate\( (r, d') \)
- unload\( (r, c, \text{top}(p'), p', d) \)

---

**Progress-to-finish**\( (\mathcal{M}, \mathcal{A}, s, \tau, m) \)

\[ i \leftarrow \text{nil} \quad \text{// instruction pointer for body}(m) \]
\[ \pi \leftarrow \langle \rangle \quad \text{// plan produced from body}(m) \]

**loop**

- if \( \tau \) is a goal and \( s \models \tau \) then return \( \pi \)
- if \( i \) is the last step of \( m \) then
  - if \( \tau \) is a goal and \( s \not\models \tau \) then return failure
  - return \( \pi \)
- \( i \leftarrow \text{nextstep}(m, i) \)
- case \( \text{type}(m[i]) \)
- **assignment:** update \( s \) according to \( m[i] \)
- **command:**
  - \( a \leftarrow \text{the descriptive model of } m[i] \text{ in } \mathcal{A} \)
  - if \( s \models \text{pre}(a) \) then
    - \( s \leftarrow \gamma(s, a); \ \pi \leftarrow \pi.a \)
  - else return failure
- **task or goal:**
  - \( \pi' \leftarrow \text{SeRPE}(\mathcal{M}, \mathcal{A}, s, m[i]) \)
  - if \( \pi' = \text{failure} \) then return failure
  - \( s \leftarrow \gamma(s, \pi'); \ \pi \leftarrow \pi.\pi' \)
Example

### Task: `put-in-pile(c_1,p_2)`

**Method:**

`m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2)`

### Task: `uncover(c_1)`

**Method:**

`m1-uncover(c_1)`

(no children)

### Task: `navigate(r_1,d_2)`

**Action:**

`load(r_1,c_1,pos(c_1),p_1,d_1)`

if `loc(r_1) ≠ d_1` then

`navigate(r_1,d_1)`

`uncover(c_1)`

`unload(r_1,c_1,top(p_2),p_2,d_2)`

### Task: `uncover(c)`

**Pre:**

`top(pile(c))=c`

**Body:**

// empty

### Task: `uncover(c,c,c,p_1',d)`

**Pre:**

`pile(c)=p_1 ∧ top(p_1)≠c ∧ ...`

**Body:**

// empty
Example

m1-navigate\((r,d')\)
- task: navigate\((r, d')\)
- pre: \(\text{loc}(r)=d'\)
- body: // empty

m2-navigate\((r,d')\)
- task: navigate\((r, d')\)
- pre: \(\text{loc}(r)\neq d' \land \text{adjacent}(\text{loc}(r),d')\)
- body: move\((r, \text{loc}(r), d')\)

m3-navigate\((r,d,d')\)
- task: navigate\((r, d')\)
- pre: \(\text{loc}(r)\neq d' \land d \neq d' \land \text{adjacent}(\text{loc}(r),d)\)
- body: move\((r, \text{loc}(r), d)\)
  navigate\((r, d')\)

Candidates = \{m2-navigate\((r_{1},d_{2})\), m3-navigate\((r_{1},d_{3},d_{2})\)\}
Example

m1-navigate\( (r, d') \)
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) = d' \\
\text{body:} & \quad \text{// empty}
\end{align*}
\]

m2-navigate\( (r, d') \)
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) \neq d' \land \text{adjacent}(\text{loc}(r), d') \\
\text{body:} & \quad \text{move}(r, \text{loc}(r), d')
\end{align*}
\]

m3-navigate\( (r, d, d') \)
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) \neq d' \land d \neq d' \\
& \quad \land \text{adjacent}(\text{loc}(r), d) \\
\text{body:} & \quad \text{move}(r, \text{loc}(r), d) \\
& \quad \text{navigate}(r, d')
\end{align*}
\]
Example

task put-in-pile(c_1, p_2)

method m2-put-in-pile(r_1, c_1, p_1, d_1, p_2, d_2)

task uncover(c_1)

method m1-uncover(c_1) (no children)

action load(r_1, c_1, nil, p_1, d_1)

method m2-navigate(r_1, d_2)

action move(r_1, d_1, d_2)

body: if loc(r) ≠ d then navigate(r, d)

uncover(c)

load(r, c, pos(c), p, d)

if loc(r) ≠ d' then

navigate(r, d')

unload(r, c, top(p'), p', d)
Example

\[\text{m1-navigate}(r,d')\]
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) = d' \\
\text{body:} & \quad \text{// empty}
\end{align*}
\]

\[\text{m2-navigate}(r,d')\]
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) \neq d' \land \text{adjacent}(	ext{loc}(r),d') \\
\text{body:} & \quad \text{move}(r, \text{loc}(r), d')
\end{align*}
\]

\[\text{m3-navigate}(r,d,d')\]
\[
\begin{align*}
\text{task:} & \quad \text{navigate}(r, d') \\
\text{pre:} & \quad \text{loc}(r) \neq d' \land d \neq d' \\
& \quad \land \text{adjacent}(	ext{loc}(r),d) \\
\text{body:} & \quad \text{move}(r, \text{loc}(r), d) \\
& \quad \text{navigate}(r, d')
\end{align*}
\]

Candidates = \{\text{m2-navigate}(r_1,d_2), \text{m3-navigate}(r_1,d_3,d_2)\}

\[
\begin{align*}
\text{body:} & \quad \text{if} \ \text{loc}(r) \neq d \ \text{then ...} \\
\text{uncover}(c) & \quad \text{load}(r, c, \text{pos}(c), p, d) \\
& \quad \text{if} \ \text{loc}(r) \neq d' \ \text{then} \\
& \quad \text{navigate}(r,d') \\
& \quad \text{unload}(r,c,\text{top}(p'),p',d)
\end{align*}
\]
Example

m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d'
- body: // empty

m2-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d' ∧ adjacent(loc(r),d'))
- body: move(r, loc(r), d')

m3-navigate(r,d,d') r_1,d_3,d_2
- task: navigate(r, d')
- pre: loc(r)=d' ∧ d ≠ d'
  ∧ adjacent(loc(r),d'))
- body: move(r, loc(r), d)
  navigate(r, d')

Candidates = {m2-navigate(r_1,d_2), m3-navigate(r_1,d_1,d_2)}

body: if loc(r) ≠ d then …
  uncover(c)
  load(r,c, pos(c), p, d)
  if loc(r) ≠ d' then
    navigate(r,d')
  unload(r,c,top(p'),p',d)
Example

m1-navigate(r,d')
  task:  navigate(r, d')
  pre:   loc(r)=d'
  body:  // empty

m2-navigate(r,d') r_1,d_2
  task:  navigate(r, d')
  pre:   loc(r)≠d' ∧ adjacent(loc(r),d'))
  body:  move(r, loc(r), d')

m3-navigate(r,d,d')
  task:  navigate(r, d')
  pre:   loc(r)≠d' ∧ d ≠ d'
         ∧ adjacent(loc(r),d))
  body:  move(r, loc(r), d)
          navigate(r, d')

m2-put-in-pile ...
uncover(c)
load(r, c, pos(c), p, d)
if loc(r) ≠ d' then
  navigate(r,d')
unload(r,c,top(p'),p',d)
Example

**Task**
- put-in-pile($c_1,p_2$)

**Method**
- m2-put-in-pile($r_1,c_1,p_1,d_1,p_2,d_2$)

**Task**
- uncover($c_1$)

**Method**
- m1-uncover($c_1$) (no children)

**Task**
- navigate($r_1,d_2$)

**Method**
- m3-navigate($r_1,d_3,d_2$)

**Action**
- load($r_1,c_1,nil,p_1,d_1$)

**Action**
- move($r_1,d_1,d_3$)

**Method**
- m2-navigate($r_1,d_2$)

**Action**
- unload($r_1,c_1,c_3,p_2,d_2$)

**Action**
- move($r_1,d_3,d_2$)

**Example**

```
m2-put-in-pile ...
uncover(c)
load(r, c, pos(c), p, d)
if loc(r) ≠ d' then
  navigate(r,d')
unload(r,c,top(p'),p',d)
```
Heuristics For SeRPE

SeRPE($\mathcal{M}, \mathcal{A}, s, \tau$)

Candidates $\leftarrow$ Instances($\mathcal{M}, \tau, s$)

if Candidates $= \emptyset$ then return failure

nondeterministically choose $m \in$ Candidates

return Progress-to-finish($\mathcal{M}, \mathcal{A}, s, \tau, m$)

- Ad hoc approaches:
  - domain-specific estimates
  - keep statistical data on how well each method works
  - try methods (or actions) in the order that they appear in $\mathcal{M}$ (or $\mathcal{A}$)

- Ideally, would want to implement using heuristic search (e.g., GBFS)
  - What heuristic function?
  - Open problem

- SeRPE is a generalization of HTN planning
  - In some cases classical-planning heuristics can be used, in other cases they become intractable [Shivashankar et al., ECAI-2016]
Interleaved Refinement Tree (IRT) Procedure

- SeRPE doesn’t allow the ‘concurrent’ programming construct

- Assume same initial state as before
- Want to move \(c_1\) to \(p_2\), and \(c_3\) to \(p_1\) using these plans:
  \[
  \langle \text{load}(r_1,c_1,c_2,p_1,d_1), \text{move}(r_1,d_1,d_2), \text{unload}(r_1,c_1,p_3,nil,d_2) \rangle
  \]
  \[
  \langle \text{load}(r_2,c_3,nil,p_2,d_2), \text{move}(r_2,d_2,d_3), \text{move}(r_2,d_3,d_1), \text{unload}(r_2,c_3,c_2,p_1,d1) \rangle
  \]
- Need to interleave, otherwise they’ll fail
Interleaved Refinement Tree (IRT) Procedure

- Extend SeRPE to interleave plans for different tasks
- Details: Section 3.3.2
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
  - REAP: a RAE-like actor uses SeRPE-like planning at all levels
  - Complicated, we’ll skip it

- 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 they call SeRPE instead of a classical planner
Using Planning in Acting

Refine-Lookahead

while \((s \leftarrow \text{observed state}) \neq g\) do
\[
\pi \leftarrow \text{Lookahead}(\mathcal{M}, \mathcal{A}, s, \tau)
\]
if \(\pi = \text{failure}\) then return failure
\(a \leftarrow \text{pop-first-action}(\pi); \text{perform}(a)\)

- Lookahead: modified version of SeRPE (discuss later)
  - Searches part of the search space, returns a partial plan

- Useful when unpredictable things are likely to happen
  - Always replans immediately

- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if \(s\) changes during the wait?
Using Planning in Acting

Refine-Lazy-Lookahead

\[ s \leftarrow \text{observed state} \]

while \( s \neq g \) do

\[ \pi \leftarrow \text{Lookahead}(\mathcal{M}, \mathcal{A}, s, \tau) \]

if \( \pi = \text{failure} \) then return failure

while \( \pi \neq \langle \rangle \) and \( s \neq g \) and \( \text{Simulate}(s, g, \pi) \neq \text{failure} \) do

\[ a \leftarrow \text{pop-first-action}(\pi); \text{perform}(a); s \leftarrow \text{observed state} \]

- Call Lookahead, execute the plan as far as possible, don’t call Lookahead again unless necessary
- Simulate does a simulation of the plan
  - Can be more detailed than SeRPE’s action models
    - e.g., physics-based simulation
- Potential problem: may wait too long to replan
  - Might not notice problems until it’s too late
  - Might miss opportunities to replace \( \pi \) with a better plan
Using Planning in Acting

Refine-Concurrent-Lookahead

\( \pi \leftarrow \langle \rangle; s \leftarrow \text{observed state} \)

thread 1:

loop

\( \pi \leftarrow \text{Lookahead}(\mathcal{M},\mathcal{A},s,\tau) \)

thread 2:

loop

if \( s \models g \) then return success

else if \( \pi = \text{failure} \) then return failure

else if \( \pi \neq \langle \rangle \) and \( \text{Simulate}(s,g,\pi) \neq \text{failure} \) do

\( a \leftarrow \text{pop-first-action}(\pi); \text{perform}(a); s \leftarrow \text{observed state} \)

Objective:

- Balance tradeoffs between Run-Lookahead and Run-Lazy-Lookahead
- More up-to-date plans than Run-Lazy-Lookahead, but without waiting for Lookahead to return
How to do Lookahead

- **Receding horizon**
  - Cut off search before reaching $g$
    - e.g., if plan’s length exceeds $l_{\text{max}}$
    - or if plan’s cost exceeds $c_{\text{max}}$
    - or when we’re running out of time
  - Horizon “recedes” on the actor’s successive calls to the planner

- **Sampling**
  - Try a few (e.g., randomly chosen) depth-first rollouts, take the one that looks best

- **Subgoaling**
  - Instead of planning for ultimate goal $g$, plan for a subgoal $g_i$
  - When it’s finished with $g_i$, actor calls planner on next subgoal $g_{i+1}$

- Can use combinations of these
Example

- **Killzone 2**
  - video game
- **SeRPE-like planner**
  - Domain-specific
  - Plans enemy actions at the squad level
- Don’t want to get the best possible plan
  - Need actions that appear believable and consistent to human users
  - Need them very quickly
- Use subgoaling
  - e.g., “get to shelter”
  - solution plan is maybe 4–6 actions long
- 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 Run-Lookahead
  - Lookahead = SeRPE, returns $\langle a_1, a_2, a_3 \rangle$
  - Actor performs $a_1$, calls Lookahead again
  - No applicable method for $\tau$ in $s_1$, SeRPE returns failure

- Fixes
  - When writing refinement methods, make them general enough to work in different states
  - In some cases Lookahead might be able to fall back on classical planning until it finds something that matches a method
  - Keep snapshot of SeRPE’s 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 Run-Lazy-Lookahead
  - Lookahead = SeRPE with receding horizon, returns $\langle a_1, a_2 \rangle$
  - Actor performs them, calls Lookahead again
  - No applicable method for $\tau$ in $s_2$, SeRPE returns failure

- Can use the same fixes on previous slide, with one modification
  - Keep snapshot of SeRPE’s search tree at horizon
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 Run-Lazy-Lookahead
  - Lookahead = SeRPE, returns $\langle a_1, a_2, a_3 \rangle$
  - While acting, unexpected event
  - Actor calls Lookahead again
  - No applicable method for $\tau$ in $s_4$, SeRPE returns failure

- Can use most of the fixes on last two slides, with this modification:
  - Keep snapshot of SeRPE’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
Summary

- **Representation:**
  - state variables, commands/actions, refinement methods

- **Refinement Acting Engine (RAE):**
  - Purely reactive
    - For each task, event, or goal, select a method and apply it

- **Refinement planning (SeRPE):**
  - Simulate RAE’s operation on a single task/event/goal
  - Deterministic actions
    - OK if we’re confident of outcome, can recover if things go wrong

- **Acting and planning:**
  - Lookahead: search part of the search space, return a partial solution
    - Several techniques for doing that
  - Caveats
    - Current state may not be what we expect
    - Possible ways to handle that
Any questions?