Chapter 3
Deliberation with Refinement Methods

Section 3.3: Refinement Planning
Section 3.4: Acting and Refinement Planning

Dana S. Nau
University of Maryland
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
Refinement Planning

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)

Search tree:

- fetch(r1,c2)
  - m-fetch(r1,c2)
    - search(r1,c2)
      - m-search(r1,c2)
        - move-to(r1,loc1)
          - perceive(loc1)
      - search(r1,c2)
        - sensor failure
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: \(\text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(o) = l\)
    - eff: \(\text{cargo}(r) \leftarrow o, \text{loc}(o) \leftarrow r\)
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: $\text{cargo}(r) = \text{nil}$, $\text{loc}(r) = l$, $\text{loc}(o) = l$
    - eff: $\text{cargo}(r) \leftarrow o$, $\text{loc}(o) \leftarrow r$

  - `put(r;o,l)`
    - pre: $\text{loc}(r) = l$, $\text{loc}(o) = r$
    - eff: $\text{cargo}(r) \leftarrow \text{nil}$, $\text{loc}(o) \leftarrow l$

  - `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:

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

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

\[ \text{move}(r,d,d') \]
- pre: \( \text{adjacent}(d,d'), \text{loc}(r)=d, \text{occupied}(d')=\text{F} \)
- eff: \( \text{loc}(r)=d', \text{occupied}(d)=\text{F}, \text{occupied}(d')=\text{T} \)
Tasks and Methods

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

\begin{align*}
\text{m1-put-in-pile}(c,p') \\
\text{task:} & \quad \text{put-in-pile}(c,p') \\
\text{pre:} & \quad \text{pile}(c)=p' \\
\text{body:} & \quad // \text{empty}
\end{align*}

If \(c\) is already in \(p'\), do nothing

\begin{align*}
\text{m2-put-in-pile}(r,c,p,d,p',d') \\
\text{task:} & \quad \text{put-in-pile}(c,p') \\
\text{pre:} & \quad \text{pile}(c)=p \land \text{at}(p,d) \land \text{at}(p',d') \land p \neq p' \land \text{cargo}(r)=\text{nil} \\
\text{body:} & \quad \text{if loc}(r) \neq d \text{ then navigate}(r,d) \\
& \quad \text{uncover}(c) \\
& \quad \text{load}(r, c, \text{pos}(c), p, d) \\
& \quad \text{if loc}(r) \neq d' \text{ then navigate}(r,d') \\
& \quad \text{unload}(r, c, \text{top}(p'), p', d)
\end{align*}

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: \( \text{top} \( (\text{pile} \( (c) \)) \)=c \)
- body: \( // \text{empty} \)

If nothing is on \( c \), do nothing

**m2-uncover\( (r,c,c,p',d) \)**
- task: uncover\( (c) \)
- pre: \( \text{pile} \( (c) \)=p \land \text{top} \( (p) \)\( \neq c \)
\[ \land \text{at} \( (p,d) \) \land \text{at} \( (p',d) \) \land p' \neq p \]
\[ \land \text{loc} \( (r) \)=d \land \text{cargo} \( (r) \)=nil \]
- body: while \( \text{top} \( (p) \) \neq c \) do
  \[ c' \leftarrow \text{top} \( (p) \) \]
  \[ \text{load} \( (r,c',\text{pos} \( (c') \),p,d) \]  
  \[ \text{unload} \( (r,c',\text{top} \( (p') \),p',d) \]

while something is on \( c \)

- remove whatever is at the top of the stack
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(\(\mathcal{M}, \mathcal{A}, s, \tau, m\))

\[ M = \{\text{methods}\} \]
\[ A = \{\text{action models}\} \]
\[ s = \text{initial state} \]
\[ \tau = \text{task or goal} \]

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

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

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

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

$\mathcal{M} = \{$methods$\}$
$\mathcal{A} = \{$action models$\}$
$s = \text{initial state}$
$\tau = \text{task or goal}$

Rae($\mathcal{M}$)

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

- One external task
- Simulate progressing it all the way to the end

- Several external tasks
- Each time through loop, progress each one by one step
RAE’s Progress subroutine

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

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

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

- running: return
- failure: \text{Retry}(\text{stack}) \text{; return}
- done: continue

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

\( \text{pop}(\text{stack}) \) // remove stack’s top element

else do

\( i \leftarrow \text{nextstep}(m, i) \)

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

- assignment: update \( \xi \) according to \( m[i] \); return
- command: trigger command \( m[i] \); return
- task or goal: continue

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

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

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

else do

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

\( \text{stack} \leftarrow \text{push}((\tau', m', \text{nil}, \emptyset), \text{stack}) \)

Just a decision tree:

- \( m[i] \) finished?
  - yes: return
  - no:
    - more steps?
      - yes:
        - \( i \leftarrow \text{next step} \)
        - assignment
        - command
        - task or goal
        - candidates for \( m[i] \)?
          - no: \text{Retry}
          - yes:
            - choose candidate \( m' \)
            - push \((m[i], m', \text{nil}, \emptyset)\) onto stack
      - no: pop stack
    - retry failed

- Put a loop around this
- Simulate the commands
Progress-to-finish ($M, A, s, \tau, m$)

1. Simulate RAE’s goal monitoring
2. If $m[i]$ is a command
   - Use action model to predict outcome
3. If current step is a task
   - Call SeRPE recursively
   - Recursion stack $\approx$ Rae’s refinement stack
4. 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, …
Example

Candidates = \{m1\text{-put\text{-}in\text{-}pile}(c_1,p_2), m2\text{-put\text{-}in\text{-}pile}(r,c_1,p_1,d,p',d')\}

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

m2\text{-put\text{-}in\text{-}pile}(r,c,p,d,p',d')
  task: put\text{-}in\text{-}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)

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

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

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

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

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

**Refinement tree**
- The SeRPE pseudocode doesn’t return this, but can easily be modified to do so

\[ \text{m2-put-in-pile}(r, c, p, d, p', d') \]

**Task:** 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\((r, c, \text{top}(p'), p', d)\)

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

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

- \(m2\) starts with \(c=c_1, p'=p_2\), and \(r, d, p', d'\) unbound
- Bind the other variables here
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)

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

r_1,c_1,p_1,d_1,p_2,d_2
m2-put-in-pile(r, c, p, d, p', d')
task: 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 \text{ then navigate}(r,d)
\quad \text{uncover}(c)
\quad \text{load}(r, c, \text{pos}(c), p, d)
\quad \text{if } \text{loc}(r) \neq d' \text{ then}
\quad \quad \text{navigate}(r,d')
\quad \text{unload}(r, c, \text{top}(p'), p', d)

\text{loc}(r_1) = d_1 = d
Example

(task
  put-in-pile(c₁,p₂)
)

(method
  m2-put-in-pile(r₁,c₁,p₁,d₁,p₂,d₂)
)

(task
  uncover(c₁)
)

(method
  m1-uncover(c₁)
  (no children)
)

(r₁,c₁,p₁,d₁,p₂,d₂)

m2-put-in-pile(r,c,p,d,p',d')
(task: put-in-pile(c,p')
pre: pile(c)=p ∧ at(p,d) ∧ at(p',d)
 ∧ p ≠ p' ∧ cargo(r)=nil
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)
)

(c₁)

m1-uncover(c)
(task: uncover(c)
pre: top(pile(c))=c
body: // empty
)

m2-uncover(r,c,c,p',d)
(task: uncover(c)
pre: pile(c)=p ∧ top(p)≠c ∧...
)

[r₁, c₁, p₁, d₁, p₂, d₂]
Example

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

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

\[
\text{task} \quad \text{uncover}(c_1)
\]

\[
\text{method} \quad m1-\text{uncover}(c_1)
\]

\[
\text{(no children)}
\]

\[
\text{action} \quad \text{load}(r_1, c_1, \text{nil}, p_1, d_1)
\]

\[
\text{action} \quad \text{unload}(r_1, c_1, c_3, p_2, d_2)
\]

\[
\text{method} \quad m2-\text{navigate}(r_1, d_2)
\]

\[
\text{action} \quad \text{move}(r_1, d_1, d_2)
\]

\[
\text{r}_1, c_1, p_1, d_1, p_2, d_2
\]

\[
m2-\text{put-in-pile}(r, c, p, d, p', d')
\]

\[
\ldots
\]

\[
\text{body: } \text{if} \ \text{loc}(r) \neq d \ \text{then} \ \text{navigate}(r, d)
\]

\[
\text{uncover}(c)
\]

\[
\text{load}(r, c, \text{pos}(c), p, d)
\]

\[
\text{if} \ \text{loc}(r) \neq d' \ \text{then}
\]

\[
\text{navigate}(r, d')
\]

\[
\text{unload}(r, c, \text{top}(p'), p', d)
\]
Example

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

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

m2-navigate(r,d') \(r_1,d_2\)
 task: navigate(r, d')
 pre: loc(r)\neq d' \land 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)\neq d' \land d \neq d' \\
\wedge adjacent(loc(r),d))
 body: move(r, loc(r), d)
 navigate(r, d')

This is just an example. One really should use a motion-planning algorithm.

\(r_1,c_1,p_1,d_1,p_2,d_2\)

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

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)
Alternative 1

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

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

m3-navigate\((r, d, d')\)
- task: navigate\((r, d')\)
- pre: loc\((r) \neq d' \land d \neq d' \land \text{adjacent(loc}(r), d))\)
- body: move\((r, \text{loc}(r), d)\)
  navigate\((r, d')\)
Alternative 1

```
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) \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)
```
Alternative 2

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

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

m3-navigate\((r,d,d')\)
- task: navigate\((r, d')\)
- pre: loc\((r)\)\(\neq d'\) ∧ \(d \neq d'\) ∧ 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_3,d_2)\}\}

Uncover \((c)\)
load\((r, c, pos(c), p, d)\)
if loc\((r)\)\(\neq d'\) then
    navigate\((r,d')\)
    unload\((r,c,top(p'),p',d)\)

Body: if loc\((r)\)\(\neq d\) then ...

Diagram
Alternative 2

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

*Candidates = {m2-navigate(r, d), m3-navigate(r, d, d')}*
Alternative 2

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($r$)$\neq$ d'$ \land$ adjacent(loc($r$),$d'$)
  body: move($r$, loc($r$), $d'$)

m3-navigate($r,d,d'$)
  task: navigate($r, d'$)
  pre: loc($r$)$\neq$ d'$ \land$ d$\neq$ d'
      \land$ adjacent(loc($r$),$d$)
  body: move($r$, loc($r$), $d$)
      navigate($r, d'$)
Alternative 2

Task
put-in-pile(c₁,p₂)

Method
m2-put-in-pile(r₁,c₁,p₁,d₁,p₂,d₂)

Task
uncover(c₁)

Method
m1-uncover(c₁)
(no children)

Action
load(r₁,c₁,nil,p₁,d₁)

Task
navigate(r₁,d₂)

Method
m3-navigate(r₁,d₃,d₂)

Action
move(r₁,d₁,d₃)

Method
m2-navigate(r₁,d₂)

Action
move(r₁,d₃,d₂)

Method

Task
navigate(r₁,d₂)

Method
m2-navigate(r₁,d₂)

Action
move(r₁,d₃,d₂)

Action
unload(r₁,c₁,c₃,p₂,d₂)

Task
uncover(c₁)

Method

Task
put-in-pile(c₁,p₂)

Method
m2-put-in-pile(r₁,c₁,p₁,d₁,p₂,d₂)

Task
uncover(c₁)

Method
m1-uncover(c₁)
(no children)

Action
load(r₁,c₁,nil,p₁,d₁)

Task
navigate(r₁,d₂)

Method
m3-navigate(r₁,d₃,d₂)

Action
move(r₁,d₁,d₃)

Method
m2-navigate(r₁,d₂)

Action
move(r₁,d₃,d₂)

Action
unload(r₁,c₁,top(p'),p',d)

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)
Heuristics For SeRPE

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

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

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

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

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

- \textit{Ad hoc} approaches:
  - domain-specific estimates
  - 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 \textit{et al.}, ECAI-2016]
Want to move $c_1$ to $p_2$, using this plan …
\[
\langle \text{load}(r_1,c_1,c_2,p_1,d_1), \text{move}(r_1,d_1,d_2), \text{unload}(r_1,c_1,p_3,nil,d_2) \rangle
\]
… and move $c_3$ to $p_1$ using this plan:
\[
\langle \text{load}(r_2,c_3,nil,p_2,d_2), \text{move}(r_2,d_2,d_3), \text{move}(r_2,d_3,d_1), \text{unload}(r_2,c_3,c_2,p_1,d1) \rangle
\]
For it to work, need to interleave the plans

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

load($r,c,c',p,d$)
\[
\begin{align*}
\text{pre: } & \text{at}(p,d), \text{cargo}(r)=\text{nil}, \\
& \text{loc}(r')=d, \text{pos}(c)=c', \text{top}(p)=c \\
\text{eff: } & \text{cargo}(r)\leftarrow c, \text{pile}(c)\leftarrow \text{nil}, \\
& \text{pos}(c)\leftarrow r, \text{top}(p)\leftarrow c'
\end{align*}
\]
unload($r,c,c',p,d$)
\[
\begin{align*}
\text{pre: } & \text{at}(p,d), \text{pos}(c)=r, \text{loc}(r)=d, \\
& \text{top}(p)=c' \\
\text{eff: } & \text{cargo}(r)\leftarrow \text{nil}, \text{pile}(c)\leftarrow p, \\
& \text{pos}(c)\leftarrow c', \text{top}(p)\leftarrow c
\end{align*}
\]
move($r,d,d'$)
\[
\begin{align*}
\text{pre: } & \text{adjacent}(d,d'), \text{loc}(r)=d, \\
& \text{occupied}(d')=\text{F} \\
\text{eff: } & \text{loc}(r)=d', \text{occupied}(d)=\text{F}, \\
& \text{occupied}(d')=\text{T}
\end{align*}
\]
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 ($s \leftarrow$ observed state) $\not\equiv g$ do
  $\pi \leftarrow$ Lookahead($M, A, s, \tau$)
  if $\pi = \text{failure}$ then return failure
  $a \leftarrow \text{pop-first-action}(\pi)$; 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 \( s \not\equiv g \) do

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

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

while \( \pi \neq \langle \rangle \) and \( s \not\equiv g \) and 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

- 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