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 $\tau_1$, $\tau_2$, ..., $\tau_n$
    - Several tasks/events, how to prioritize?
  - Candidates for $\tau_1$: $m_1$, $m_2$, ..., 
    - 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
  - \( \text{take}(r,o,l) \)
    - 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**

  ```plaintext
  take(r;o;l)
  pre: cargo(r) = nil, loc(r) = l, loc(o) = l
  eff: cargo(r) ← o, loc(o) ← r
  ```

  ```plaintext
  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**\((r;o,l)\):
    \begin{align*}
    \text{pre:} & \quad \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(o) = l \\
    \text{eff:} & \quad \text{cargo}(r) \leftarrow o, \text{loc}(o) \leftarrow r
    \end{align*}
  - **put**\((r;o,l)\):
    \begin{align*}
    \text{pre:} & \quad \text{loc}(r) = l, \text{loc}(o) = r \\
    \text{eff:} & \quad \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(o) \leftarrow l
    \end{align*}
  - **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
- Ongoing research on extending refinement planning to handle nondeterminism
  - [Patra et al., AAAI-2019]
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\((r,d,d')\)
    
    \[
    \text{pre: } \text{adjacent}(d,d'), \text{loc}(r)=d, \text{occupied}(d')=\text{F}
    \]
    
    \[
    \text{eff: } \text{loc}(r)=d', \text{occupied}(d)=\text{F}, \text{occupied}(d')=\text{T}
    \]
Tasks and Methods

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

  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($c$)=$p$ ∧ at($p,d$) ∧ at($p',d'$)
  ∧ $p \neq p'$ ∧ 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($r, c, top(p'), p', d$)

  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)

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

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

if Candidates = \(\emptyset\) then return failure

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

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

\(M = \{\text{methods}\}\)
\(A = \{\text{action models}\}\)

\(s = \text{initial state}\)
\(\tau = \text{task or goal}\)

Rae(\(M\))

Agenda \leftarrow \emptyset

loop

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

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

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

if 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 stack \(\in\) Agenda do

Progress(stack)

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

Which candidate method for \(\tau\)?

- SeRPE: Nondeterministic choice
  - backtracking point

How to implement?

- Hierarchical adaptation of backtracking, A*, GBFS, …

Which candidate method for \(\tau\)?

- RAE: Arbitrary choice
  - no search, purely reactive
SeRPE (Sequential Refinement Planning Engine)

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

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

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

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

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

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

\[
\text{Progress}(stack) \\
(\tau, m, i, \text{tried}) \leftarrow \text{top}(stack) \\
\text{if } i \neq \text{nil} \text{ and } m[i] \text{ is a command then do} \\
\hspace{1em} \text{case status}(m[i]) \\
\hspace{2em} \text{running: return} \\
\hspace{2em} \text{failure: Retry}(stack); \text{return} \\
\hspace{2em} \text{done: continue} \\
\text{if } i \text{ is the last step of } m \text{ then} \\
\hspace{1em} \text{pop}(stack) \quad // \text{remove } stack\text{’s top element} \\
\text{else do} \\
\hspace{2em} i \leftarrow \text{nextstep}(m, i) \\
\hspace{2em} \text{case type}(m[i]) \\
\hspace{3em} \text{assignment: update } \xi \text{ according to } m[i]; \text{return} \\
\hspace{3em} \text{command: trigger command } m[i]; \text{return} \\
\hspace{3em} \text{task or goal: continue} \\
\hspace{1em} \tau' \leftarrow m[i] \\
\hspace{1em} \text{Candidates} \leftarrow \text{Instances}(M, \tau', \xi) \\
\hspace{1em} \text{if } \text{Candidates} = \emptyset \text{ then Retry}(stack) \\
\hspace{1em} \text{else do} \\
\hspace{2em} \text{arbitrarily choose } m' \in \text{Candidates} \\
\hspace{2em} \text{stack} \leftarrow \text{push}((\tau', m', \text{nil,}\emptyset), \text{stack})
\]

Just a decision tree:

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

\( \xi \) update

\( \text{Progress-to-finish} \)
- Like Progress with a loop around it
- Simulates the commands
Progress-to-finish(\(M, A, s, \tau, m\))

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

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:

\(a \leftarrow \text{the descriptive model of } m[i] \text{ in } A\)
if \(s \models \text{pre}(a)\) then
  \(s \leftarrow \gamma(s, a); \quad \pi \leftarrow \pi.a\)
else return failure

if \(\tau\) is a goal then
  \(\pi' \leftarrow \text{SeRPE}(M, A, s, m[i])\)
  if \(\pi'\) = failure then return failure
  \(s \leftarrow \gamma(s, \pi'); \quad \pi \leftarrow \pi.\pi'\)

Simulate RAE’s goal monitoring

If \(m[i]\) is a command
  ➢ Use action model to predict outcome

If current step is a task
  ➢ Call SeRPE recursively
  ➢ Recursion stack \(\approx\) Rae’s refinement stack

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-put-in-pile(c_{1},p_{2}), \\
m2-put-in-pile(r,c_{1},p_{1},d,p',d') \}

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

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')

SeRPE(M, A, s, \tau)

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

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

s_0 = \{ loc(r_{1}) = d_{1}, cargo(r_{1}) = \text{nil}, occupied(d_{1}) = \text{T}, \\
          occupied(d_{2}) = \text{F}, occupied(d_{3}) = \text{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{SeRPE}(\mathcal{M}, \mathcal{A}, s, \tau)

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

\text{m2-put-in-pile} starts with \( c = c_1, p' = p_2, \) and \( r, d, p', d' \) unbound

- Bind the other variables here

\text{r}_1, \text{c}_1, \text{p}_1, \text{d}_1, \text{p}_2, \text{d}_2

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

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

\text{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}

\text{body: } \text{if} \quad \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} \quad \text{loc}(r) \neq d' \text{ then }

\text{navigate}(r, d')

\text{unload}(r, c, \text{top}(p'), p', d)
task
put-in-pile(c₁,p₂)

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

Progress-to-finish(M,A,s,τ,m)
i ← nil // instruction pointer for body(m)
π ← ⟨⟩ // plan produced from body(m)
loop
if τ is a goal and s ⊨ τ then return π
if i is the last step of m then
if τ is a goal and s ⊭ τ then return failure
return π
i ← nextstep(m,i)
case type(m[i])
assignment: update s according to m[i]
command:
a ← the descriptive model of m[i] in A
if s ⊨ pre(a) then
s ← γ(s,a); π ← π.a
else return failure

π' ← SeRPE(M,A,s,m[i])
if π' = failure then return failure
s ← γ(s,π'); π ← π.π'

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)

loc(r₁) = d₁ = d
Example

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

\[ \text{method} \]
\[ \text{m1-uncover}(c_1) \]
\[ \text{(no children)} \]

\[ \text{r}_1, c_1, p_1, d_1, p_2, d_2 \]
\[ \text{m2-put-in-pile}(r, c, p, d, p', d') \]
\[ \text{task: put-in-pile}(c, p') \]
\[ \text{pre: } \text{pile}(c)=p \land \text{at}(p,d) \land \text{at}(p',d) \land p \neq p' \land \text{cargo}(r)=\text{nil} \]
\[ \text{body: if } \text{loc}(r) \neq d \text{ then navigate}(r,d) \]
\[ \text{uncover}(c) \]
\[ \text{load}(r, c, \text{pos}(c), p, d) \]
\[ \text{if } \text{loc}(r) \neq d' \text{ then navigate}(r,d') \]
\[ \text{unload}(r,c,\text{top}(p'),p',d) \]

\[ \text{c} \]
\[ \text{m1-uncover}(c) \]
\[ \text{task: uncover}(c) \]
\[ \text{pre: } \text{top}(\text{pile}(c))=c \]
\[ \text{body: } // \text{empty} \]

\[ \text{m2-uncover}(r,c,c,p',d) \]
\[ \text{task: uncover}(c) \]
\[ \text{pre: } \text{pile}(c)=p \land \text{top}(p) \neq c \land \ldots \]
\[ \text{task} \]
\[ \text{put-in-pile}(c_1, p_2) \]

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

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

\[ \text{method} \]
\[ \text{m1-uncover}(c_1) \]
\[ (\text{no children}) \]

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

---

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

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

\[ \ldots \]

\[ \text{body: if loc}(r) \neq d \text{ then navigate}(r, d) \]
\[ \text{uncover}(c) \]
\[ \boxed{\text{load}(r, c, \text{pos}(c), p, d) \text{ action}} \]
\[ \text{if loc}(r) \neq d' \text{ then} \]
\[ \text{navigate}(r, d') \]
\[ \text{unload}(r, c, \text{top}(p'), p', d) \]
Example

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

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

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

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

(no children)

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

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

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

... body: if loc\((r) \neq d\) then navigate\((r, d)\)

uncover\((c)\)

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

if loc\((r) \neq d'\) then

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

action

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

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

- **Task**: uncover($c_1$) (no children)

- **Method**

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

- **Action**
  
  - load($r_1$, $c_1$, nil, $p_1$, $d_1$)
  
  - unload($r_1$, $c_1$, $c_3$, $p_2$, $d_2$)

- **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($p'$), $p'$, $d'$)
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
  - 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]
Interleaving

Want to move $c_1$ to $p_2$, using this plan …

$\langle \text{load}(r_1,c_1,c_2,p_1,d_1), \text{move}(r_1,d_1,d_2), \text{unload}(r_1,c_1,p_3,\text{nil},d_2) \rangle$

… and move $c_3$ to $p_1$ using this plan:

$\langle \text{load}(r_2,c_3,\text{nil},p_2,d_2), \text{move}(r_2,d_2,d_3), \text{move}(r_2,d_3,d_1), \text{unload}(r_2,c_3,c_2,p_1,d_1) \rangle$

For it to work, must interleave the plans

$\langle \text{load}(r_2,c_3,\text{nil},p_2,d_2), \text{move}(r_2,d_2,d_3), \text{load}(r_1,c_1,c_2,p_1,d_1), \text{move}(r_1,d_1,d_2), \text{unload}(r_1,c_1,p_3,\text{nil},d_2), \text{move}(r_2,d_3,d_1), \text{unload}(r_2,c_3,c_2,p_1,d_1) \rangle$

\begin{align*}
\text{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'
\end{align*}

\begin{align*}
\text{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
\end{align*}

\begin{align*}
\text{move}(r,d,d') \\
\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*}

\textbf{Poll:} can SeRPE do that? \\
1. yes; 2. no
Interleaved Refinement Tree (IRT) Procedure

- SeRPE doesn't allow the 'concurrent' programming construct
- Partial fix: 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
  \(\pi \leftarrow \text{Lookahead}(M, A, s, \tau)\)
  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 \( s \not\equiv g \) 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 \( \text{Simulate}(s,g,\pi) \neq \text{failure} \) 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 the horizon, 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, 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)
  - plan by simulating RAE on a single external task/event/goal
  - Deterministic actions
    - OK if we’re confident of outcome, can recover if things go wrong
  - Interleaved plans (brief example)

- Acting and planning
  - Lookahead: search part of the search space, return a partial solution
  - Refine-Lookahead, Refine-Lazy-Lookahead, Refine-Concurrent-Lookahead
    - Like Run-Lookahead, Run-Lazy-Lookahead, Run-Concurrent-Lookahead, but call SeRPE

- Caveats
  - Current state may not be what we expect
  - Possible ways to handle that