Automated Planning and Acting – Refinement Methods

Institute of Information Systems

Mattis Hartwig
1. Planning and Acting with **Deterministic** Models
   Conventional AI planning

2. Planning and Acting with **Refinement** Methods
   Abstract activities $\rightarrow$ collections of less-abstract activities

3. Planning and Acting with **Temporal** Models
   Reasoning about time constraints

4. Planning and Acting with **Nondeterministic** Models
   Actions with multiple possible outcomes

5. **Standard** Decision Making
   Utility theory
   Markov decision process (MDP)

6. Planning and Acting with **Probabilistic** Models
   Actions with multiple possible outcomes, with probabilities

7. **Advanced** Decision Making
   Hidden goals
   Partially observable MDP (POMDP)
   Decentralised POMDP

8. **Human-aware** Planning
   Planning with a human in the loop

9. **Causal** Planning
   Causality & Intervention
   Implications for Causal Planning
Motivation

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

• Refine abstract actions into ways of carrying out those actions
  • How?
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?
Assumptions

• Removes/weakens assumptions from classical planning
• Characteristics
  • Dynamic environment
  • Imperfect information
  • Overlapping actions
  • Nondeterminism
  • Hierarchy
  • Discrete and continuous variables
State-variable Representation (Recap)

• Objects:
  • \( \text{Robots} = \{ \text{rbt} \} \)
  • \( \text{Containers} = \{ \text{c1, c2, c3, ...} \} \)
  • \( \text{Locations} = \{ \text{loc0, loc1, loc2, ...} \} \)

• State variables: syntactic terms to which we can assign values
  • \( \text{loc}(r) \in \text{Locations} \)
  • \( \text{load}(r) \in \text{Containers} \cup \{ \text{nil} \} \)
  • \( \text{pos}(c) \in \text{Locations} \cup \text{Robots} \cup \{ \text{unknown} \} \)
  • \( \text{view}(r, l) \in \{ T, F \} \)
    • whether robot \( r \) has looked at location \( l \)
    • \( r \) can only see what is at its current location

• State: assign a value to each state variable
  • \( \{ \text{loc(rbt)} = \text{loc0}, \text{pos(c1)} = \text{loc2}, \text{pos(c3)} = \text{loc4}, \text{pos(c2)} = \text{unknown}, \ldots \} \)
State-variable Representation: Extensions

- **Range** $\mathcal{R}(x)$
  - Can be finite, infinite, continuous, discontinuous, vectors, matrices, other data structures
- **Assignment statement** $x \leftarrow expr$
  - Expression that returns a ground value in $\mathcal{R}(x)$ 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 is at its current location

- **Event**: occurrence detected by execution platform
  - `event-name(args)`
  - Exogenous changes in the environment to which the actor may have to react
  - E.g., emergency signal, arrival of transportation vehicle

- For later
  - $\mathcal{M}$: library of methods
  - $\xi$: current state (abstraction)
Tasks and Methods

- **Task**: an activity for the actor to perform
  - Could be an abstract action of a plan
  - For each task, a set of refinement methods
  - **Operational models**:
    - Tell *how* to perform the task
    - Do not predict *what* it will do

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

- assignment statements
- control constructs: if-then-else, while, ...
- tasks (can extend to include events, goals)
- commands to the execution platform
Example: “open door” task

- What kind:
  - Hinged on left
  - Opens toward us
  - Lever handle

→ Refinement method

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

```
m1-throw-wide(r,d,l,o)
  task: throw-wide(r,d)
  pre: loc(r,l) \land toward-side(l,d) \land
        side(d,left) \land type(d,rotate) \land
        handle(d,o) \land door-status(d)=cracked
  body: grasp(r,o)
        pull(r,val1)
        move-by(r,val2)
```
Rae (Refinement Acting Engine)

- Based on OpenPRS
  - Programming language, open-source robotics software
  - Deployed in many applications
- Input
  - External tasks, events, current state $\xi$, library of methods $\mathcal{M}$
- 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
  $\text{Agenda} = \{\text{all current refinement stacks}\}$

![Diagram of Rae (Refinement Acting Engine)]
• Basic idea

**Loop:**
- if new external tasks/events then
  - Add them to Agenda
- for each stack in Agenda
  - Progress it
- Remove it if it’s finished

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

Stack element(\tau, \text{m}, i, \text{tried})
- \tau: task
- \text{m}: instance of a method in \mathcal{M}
- i: instruction pointer to step in body of \text{m}
- \text{tried}: method instances already tried
Progress (subroutine) – Just a Decision Tree

Progress(stack)

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

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

**case** status($m[i]$)

running: return

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

done: continue

if $i$ is the last step of $m$ then

\text{pop}(\text{stack})

else do

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

**case** type($m[i]$)

assignment: update $\xi$ according to $m[i]$; return
task or goal: continue

$t' \leftarrow m[i]$

Candidates $\leftarrow \text{Instances}(M, t', \xi)$

if Candidates $= \emptyset$ then

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

else do

arbitrarily choose $m' \in \text{Candidates}$

\text{stack} $\leftarrow \text{push}((\tau, m, \text{nil}, \emptyset), \text{stack})$
• Objects:
  • $Robots = \{rbt\}$
  • $Containers = \{c1, c2, c3, \ldots\}$
  • $Locations = \{loc0, loc1, loc2, \ldots\}$

• 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 is at its current location

• Commands to the execution platform:
  • $take(r, o, l)$: 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 loc. $l$
  • $move-to(r, l)$: robot $r$ moves to location $l$
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

\tau: fetch(r_1,c_2)
\rho: ?
i: (see method)
\text{tried}: \emptyset

Refinement stack
**Example**

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

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

Refinement stack:
- \( \tau: \) fetch(r1,c2)
- \( m: \) m-fetch(r1,c2)
- \( i: \) (see method)
- \( tried: \emptyset \)
Example

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

m-search(r,c)
  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
Example

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

m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body:
    if \exists l (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
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

Refinement stack

\[ \text{...} \]

- \( \tau: \) search(r1,c2)
- \( m: \) m-search(r1,c2)
- \( i: \) (see method)
- \( tried: \emptyset \)

\[ \text{...} \]

- \( \tau: \) fetch(r1,c2)
- \( m: \) m-fetch(r1,c2)
- \( i: \) (see method)
- \( tried: \emptyset \)

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

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

\[\tau: \text{search}(r1,c2)\]
\[m: \text{m-search}(r1,c2)\]
\[i: \text{(see method)}\]
\[\text{tried}: \emptyset\]

\[\tau: \text{fetch}(r1,c2)\]
\[m: \text{m-fetch}(r1,c2)\]
\[i: \text{(see method)}\]
\[\text{tried}: \emptyset\]

Sensor failure
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

τ: search(r1,c2)
m: ?
i: (see method)
tried:{m-search(r1,c2)}

τ: fetch(r1,c2)
m: m-fetch(r1,c2)
i: (see method)
tried:∅

Refinement stack

If other candidates for search(r1,c2), try them.
Retry (subroutine)

```
Retry(stack)
    (τ,m,i, tried) ← pop(stack)
    tried ← tried U {m}
    Candidates ← Instances(ℳ, τ, ξ) \ tried
    if Candidates ̸= ∅ then
        arbitrarily choose m' ∈ Candidates
        stack ← push((τ,m,nil,Ø), stack)
    else do
        if stack ̸= ∅ then
            Retry(stack)
        else do
            output(“failed to accomplish” τ)
            Agenda ← Agenda \ stack
```

Another decision tree
Quiz

When an agent has the task “going shopping to buy ingredients for a meal” what is probably no a refinement of that task?

a) Finding the way for the shopping mall
b) Doing movements
c) Perceiving items
d) Preparing the meal
Extensions to RAE

• Events
  • External inputs that are handled together with task (e.g. an emergency)
  • Are handled on the outer RAE loop and can also result in methods handling the event

• Goals
  • Special kind of tasks where the progress to achieve a specific goal condition is monitored

• Concurrent subtasks
  • Refinement stack for each one

• Controlling the progress of tasks
  • E.g. Pausing tasks under specific conditions

• For a task \( \tau \), which candidate to try first?
  • Refinement planning
Refinement Planning
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 do not 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 (Hierarchical Task Network) 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 $m\text{-search}(r1,c2)$ 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
  • \texttt{take}(r,o,l):
    robot \( r \) takes object \( o \) at location \( l \)
  • \texttt{put}(r,o,l):
    \( r \) puts \( o \) at location \( l \)
  • \texttt{perceive}(r,l):
    robot \( r \) perceives what objects are at location \( l \)
  • Can only perceive what is at its current location
  • If we knew this in advance, perception would not be necessary

• Action model

\begin{align*}
\text{take}(r,o,l) & \quad \text{pre: } \text{cargo}(r) = \text{nil}, \text{loc}(r) = l, \text{loc}(o) = l \\
& \quad \text{eff: } \text{cargo}(r) \leftarrow o, \text{loc}(o) \leftarrow r
\end{align*}

\begin{align*}
\text{put}(r,o,l) & \quad \text{pre: } \text{loc}(r) = l, \text{loc}(o) = r \\
& \quad \text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{loc}(o) \leftarrow l
\end{align*}

\begin{align*}
\text{perceive}(r,l) & \quad ?
\end{align*}
Limitation

• Most environments are inherently nondeterministic
  • Deterministic action models will not always make the right prediction

• Why use them?
  • Deterministic models ⇒ much simpler planning algorithms
    • Use when errors are infrequent and do not have severe consequences
    • Actor can fix the errors online
SeRPE (Sequential Refinement Planning Engine)

- SeRPE inputs
  - $\mathcal{M} = \{\text{methods}\}$
  - $\mathcal{A} = \{\text{action models}\}$
  - $s = \text{initial state}$
  - $\tau = \text{task or goal}$
- Which candidate method for $\tau$?
  - SeRPE:
    - Nondeterministic choice
    - Backtracking point
    - How to implement?
      - Hierarchical adaptation of backtracking, A*, GBFS, ...
  - RAE
    - Arbitrary choice
    - No search, purely reactive

### Algorithm

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

- **Rae($\mathcal{M}$)**
  - $Agenda \leftarrow \emptyset$
  - loop
  - until the input of external tasks and events is empty do
    - read $\tau$ in the input stream
    - $Candidates \leftarrow \text{Instances}(\mathcal{M}, \tau, s)$
    - 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
      - $Agenda \leftarrow Agenda \setminus \{\text{stack}\}$
SeRPE (Sequential Refinement Planning Engine)

- SeRPE
  - One external task
  - Simulate progressing it all the way to the end
- Rae
  - Several external tasks
  - Each time through loop, progress each one by one step

```plaintext
SeRPE(ℳ,𝒜,s,τ)
Candidates ← Instances(ℳ,τ,s)
if Candidates = ∅ then
  return failure
nondeterministically choose m ∈ Candidates
return Progress-to-finish(ℳ,𝒜,s,τ,m)

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

• Like Rae progress with a loop around it
• Simulates the commands

Progress-to-finish(ℳ, A, s, τ, m)
i ← nil; π ← ⟨⟩
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 ← descriptive model of m[i] in A
    if s ⊨ pre(a) then
      s ← γ(s, a); π ← π.a
    else
      return failure
  task or goal:
  π' ← SeRPE(ℳ, A, s, m[i])
  if π' = failure then
    return failure
  s ← γ(s, π'); π ← π. π'
Progress-to-finish

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

\[ i \leftarrow \text{nil}; \pi \leftarrow \langle \rangle \]

\text{loop}

\[ \begin{align*}
\text{if } \tau \text{ is a goal and } s \models \tau & \text{ then} \\
\text{return } \pi \\
\text{if } i \text{ is the last step of } m & \text{ then} \\
\text{if } \tau \text{ is a goal and } s \nvdash \tau & \text{ then} \\
\text{return failure} \\
\text{return } \pi \\
i \leftarrow \text{nextstep}(m, i) \\
\text{case } \text{type}(m[i]) \\
\text{assignment:} \\
\text{update } s \text{ according to } m[i] \\
\text{command:} \\
a \leftarrow \text{descriptive model of } m[i] \text{ in } A \\
\text{if } s \models \text{pre}(a) & \text{ then} \\
\quad s \leftarrow \gamma(s, a); \pi \leftarrow \pi.a \\
\text{else} \\
\quad \text{return failure} \\
\text{task or goal:} \\
\quad \pi' \leftarrow \text{SeRPE}(M, A, s, m[i]) \\
\text{if } \pi' = \text{failure} & \text{ then} \\
\quad \text{return failure} \\
\quad s \leftarrow \gamma(s, \pi'); \pi \leftarrow \pi. \pi' 
\end{align*} \]

- **Inputs**
  - \( M = \{\text{methods}\}, A = \{\text{action models}\} \)
  - \( s = \text{initial state}, \tau = \text{task or goal}, m = \text{chosen method} \)
- **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, no Retry (Rae)**
  - **A failure means SeRPE could not find a solution**
  - **Implementation: hierarchical adaptations of backtracking, A*, GBFS, ...**
Heuristics For SeRPE

- *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

```plaintext
SeRPE(\mathcal{M}, \mathcal{A}, s, \tau)
Candidates ← Instances(\mathcal{M}, \tau, s)
if Candidates = \emptyset then
    return failure
nondeterministically choose m ∈ Candidates
return Progress-to-finish(\mathcal{M}, \mathcal{A}, s, \tau, m)
```

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
  - $\langle \text{load}(r_2, c_3, \text{nil}, p_2, d_2), \text{move}(r_2, d_2, d_3), \text{load}(r_1, c_1, c_2, p_1, d_1), \text{move}(r_1, d_1, d_2), \text{unload}(r_1, c_1, p_3, \text{nil}, d_2), \text{move}(r_2, d_3, d_1), \text{unload}(r_2, c_3, c_2, p_1, d_1) \rangle$
  - $\text{load}(r, c, c', p, d)$
    - pre: $at(p, d), \text{cargo}(r) = \text{nil}, \text{loc}(r) = d, \text{pos}(c) = c', \text{top}(p) = c$
    - 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: $at(p, d), \text{pos}(c) = r, \text{loc}(r) = d, \text{top}(p) = c'$
    - 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: $\text{adj}(d, d'), \text{loc}(r) = d, \text{occupied}(d') = F$
    - eff: $\text{loc}(r) = d', \text{occupied}(d) = F, \text{occupied}(d') = T$
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
Acting and Refinement Planning

- Hierarchical acting with refinement planning
  - REAP: a RAE-like actor uses SeRPE-like planning at all levels

- 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
  - Lookahead same as before
    - Receding horizon, sampling, subgoaling
Using Planning in Acting

- Lookahead: modified version of SeRPE
  - Searches part of the search space, returns a partial plan
  - Useful when unpredictable things are likely to happen
  - Always re-plans immediately
- Potential problem:
  - May pause repeatedly while waiting for Lookahead to return
  - What if $s$ changes during the wait?

```plaintext
Refine-Lookahead($M, A, \tau$)
while ($s$ ← abstraction of observed state $\xi$) $\not= \tau$ do
  $\pi$ ← SeRPE-Lookahead($M, A, s, \tau$)
  if $\pi$ = failure then
    return failure
  $a$ ← pop-first-action($\pi$)
  perform $a$
```

Planning stage
Acting stage
Using Planning in Acting

- Call Lookahead, execute the plan as far as possible, do not 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 re-plan
  - Might not notice problems until it’s too late
  - Might miss opportunities to replace $\pi$ with a better plan

Refine-Lazy-Lookahead($\mathcal{M}, \mathcal{A}, \tau$)

\[
s \leftarrow \text{abstraction of observed state } \xi
\]

while $s \not\equiv \tau$ do

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

if $\pi = \text{failure}$ then

return failure

while $\pi \neq \langle \rangle$ and $s \not\equiv \tau$ and

Simulate($\Sigma, s, \tau, \pi$)

$\neq \text{failure}$ do

$a \leftarrow \text{pop-first-action}(\pi)$

perform $a$

$s \leftarrow \text{abstraction of observed state } \xi$

Planning stage

Acting stage
Using Planning in Acting

- **Objective:**
  - Balance trade-offs between Refine-Lookahead and Refine-Lazy-Lookahead
  - More up-to-date plans than Refine-Lazy-Lookahead, but without waiting for Lookahead to return

```
Refine-Concurrent-Lookahead(ℳ,𝒜,τ)
  π ← ⟨⟩
  s ← abstraction of observed state ξ
  // threads 1 and 2 run concurrently

thread 1:
  loop
  π ← SeRPE-Lookahead(ℳ,𝒜,s,τ)

thread 2:
  loop
  if s ≠ τ then
    return success
  else if π = failure then
    return failure
  else if π ≠ ⟨⟩ and s ≠ τ and
    Simulate(Σ,s,τ,π) ≠ failure then
    a ← pop-first-action(π)
    perform a
    s ← abstraction of observed state ξ
```
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 there next time
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 is 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

• 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
1. Planning and Acting with **Deterministic** Models
   Conventional AI planning
2. Planning and Acting with **Refinement** Methods
   Abstract activities $\rightarrow$ collections of less-abstract activities
3. Planning and Acting with **Temporal** Models
   Reasoning about time constraints
4. Planning and Acting with **Nondeterministic** Models
   Actions with multiple possible outcomes
5. **Standard** Decision Making
   Utility theory
   Markov decision process (MDP)
6. Planning and Acting with **Probabilistic** Models
   Actions with multiple possible outcomes, with probabilities
7. **Advanced** Decision Making
   Hidden goals
   Partially observable MDP (POMDP)
   Decentralised POMDP
8. **Human-aware** Planning
   Planning with a human in the loop
9. **Causal** Planning
   Causality & Intervention
   Implications for Causal Planning