Chapter 3
Deliberation with Refinement Methods

Dana S. Nau
University of Maryland
Planning and Acting

● **Planning:** prediction + search
  ▶ Search over predicted states, possible organizations of tasks and actions
  ▶ Uses descriptive models (e.g., PDDL)
    • predict what the actions will do
    • don’t include instructions for performing it

● **Acting:** performing
  ▶ Dynamic, unpredictable, partially observable environment
    • Adapt to context, react to events
  ▶ Uses operational models
    • instructions telling how to perform the tasks
    • usually hierarchical
Planning and Acting Integration

- Problem with integrating:
  - Consistency between two different models
  - Plan generation, verification, management

- Our solution
  - Both actor and planner use the same representation
    - Must be operational; descriptive models too abstract
  - New planning algorithms to reason with operational models
Outline

1. Motivation
2. *Representation*
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application
States and Commands

- **Objects**
  - Robots = \{r1, r2\}
  - Containers = \{c1, c2\}
  - Locations = \{loc0, loc1, loc2, loc3, loc4\}

- **Rigid relations (properties that won’t change)**
  - adjacent(loc0,loc1), adjacent(loc1,loc0), adjacent(loc1,loc2), adjacent(loc2,loc1), adjacent(loc2,loc3), adjacent(loc3,loc2), adjacent(loc3,loc4), adjacent(loc4,loc3)

- **State variables (fluents)**
  - where \( r \in \text{Robots}, \ c \in \text{Containers}, \ l \in \text{Locations} \)
  - \( \text{loc}(r) \in \text{Locations} \)
  - \( \text{cargo}(r) \in \text{Containers} \cup \{\text{empty}\} \)
  - \( \text{pos}(c) \in \text{Locations} \cup \text{Robots} \cup \{\text{unknown}\} \)
  - \( \text{view}(l) \in \{T, F\} \)
    - Whether a robot has looked at location \( l \)
    - If \( \text{view}(l) = T \) then \( \text{pos}(c) = l \) for every container \( c \) at \( l \)

- **Commands to the execution platform:**
  - take(\( r, o, l \)): \( r \) takes object \( o \) at location \( l \)
  - put(\( r, o, l \)): \( r \) puts \( o \) at location \( l \)
  - perceive(\( r, l \)): robot \( r \) perceives what objects are at \( l \)
  - move-to(\( r, l \)): robot \( r \) moves to location \( l \)
Tasks and Methods

- **Task**: an activity for the actor to perform
  - `taskname(arg_1, …, arg_k)`
- For each task, one or more refinement methods
  - Operational models telling how to perform the task

```plaintext
method-name(arg_1, …, arg_k)
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

```plaintext
m-fetch1(r,c)
task: fetch(r,c)
pre: pos(c) = unknown
body:
  if ∃l (view(l) = F) then
    move-to(r,l)
    perceive(r,l)
    if pos(c) = l then
      take(r,c,l)
    else fetch(r,c)
  else fail
```

```plaintext
m-fetch2(r,c)
task: fetch(r,c)
pre: pos(c) ≠ unknown
body:
  if loc(r) = pos(c) then
    take(r,c,pos(c))
  else do
    move-to(r,pos(c))
    take(r,c,pos(c))
```

command
```
```
Outline

1. Motivation
2. Representation
3. *Acting* (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application
RAE (Refinement Acting Engine)

- Performs multiple tasks in parallel
  - Purely reactive, no lookahead

- For each task or event $\tau$, a refinement stack
  - execution stack
  - corresponds to current path in RAE’s refinement tree for $\tau$

- $\textit{Agenda} = \{\text{all current refinement stacks}\}$

\[
\begin{align*}
\tau_1, \tau_2, \tau_3
\end{align*}
\]

procedure RAE:

loop:

for every new external task or event $\tau$ do

choose a method instance $m$ for $\tau$

create a refinement stack for $\tau, m$

add the stack to $\textit{Agenda}$

for each stack $\sigma$ in $\textit{Agenda}$

call $\text{Progress}(\sigma)$

if $\sigma$ is finished then remove it
Representation

- **Objects**
  - **Robots** = \{r1, r2\}
  - **Containers** = \{c1, c2\}
  - **Locations** = \{loc1, loc2, loc3, loc4\}

- **Rigid relations** (properties that won’t change)
  - adjacent(loc0,loc1), adjacent(loc1,loc0),
    adjacent(loc1,loc2), adjacent(loc2,loc1),
    adjacent(loc2,loc3), adjacent(loc3,loc2),
    adjacent(loc3,loc4), adjacent(loc4,loc3)

- State variables (fluents)
  - where \( r \in \text{Robots}, \ c \in \text{Containers}, \ l \in \text{Locations} \)
    - \( \text{loc}(r) \in \text{Locations} \)
    - \( \text{cargo}(r) \in \text{Containers} \cup \{\text{nil}\} \)
    - \( \text{pos}(c) \in \text{Locations} \cup \text{Robots} \cup \{\text{unknown}\} \)
    - \( \text{view}(l) \in \{\text{T}, \text{F}\} \)
    - Whether a robot has looked at location \( l \)
    - If \( \text{view}(l) = \text{T} \) then \( \text{pos}(c) = l \) for every container \( c \) at \( l \)

- **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 \( l \)
  - \( \text{move-to}(r,l) \): robot \( r \) moves to location \( l \)
Example

Refinement tree

\[
\text{fetch}(r_0, c_2)
\]

procedure RAE:

loop:

for every new external task or event \( \tau \) do

choose a method instance \( m \) for \( \tau \)

create a refinement stack for \( \tau, m \)

add the stack to Agenda

for each stack \( \sigma \) in Agenda

call Progress(\( \sigma \))

if \( \sigma \) is finished then remove it

- Container locations unknown
- Partially observable
  - Robot only sees current location

\begin{itemize}
  \item Container locations unknown
  \item Partially observable
    \begin{itemize}
      \item Robot only sees current location
    \end{itemize}
\end{itemize}
Example

\[
m-fetch_1(r,c) \quad r = r_0, c = c_2
\]

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

\[
m-fetch_2(r,c)
\]

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

Candidates
\(= \{m-fetch_1(r_1,c_2), m-fetch_1(r_2,c_2)\}\)

procedure RAE:

- loop:
  - for every new external task or event \(\tau\) do
    - choose a method instance \(m\) for \(\tau\)
    - create a refinement stack for \(\tau, m\)
    - add the stack to \(Agenda\)
  - for each stack \(\sigma\) in \(Agenda\)
    - call \(\text{Progress}(\sigma)\)
    - if \(\sigma\) is finished then remove it

- Container locations unknown
- Partially observable
  - Robot only sees current location

Candidates
\(= \{m-fetch_1(r_1,c_2), m-fetch_1(r_2,c_2)\}\)
Example

\[ m\text{-}fetch1(r, c) \quad r = r_1, c = c_2 \]

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

\[ m\text{-}fetch2(r, c) \]

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

procedure RAE:

- loop:
  - for every new external task or event \( \tau \) do
    - choose a method instance \( m \) for \( \tau \)
    - create a refinement stack for \( \tau, m \)
    - add the stack to \textit{Agenda}
    - for each stack \( \sigma \) in \textit{Agenda}
      - call \text{Progress}(\sigma)
      - if \( \sigma \) is finished then remove it

Container locations unknown
Partially observable
  - Robot only sees current location

Candidates
\[ = \{ m\text{-}fetch(r_1,c_2), m\text{-}fetch(r_2,c_2) \} \]
Example

m-fetch1\((r,c)\) \(r = r_1, c = c_2\)

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

m-fetch2\((r,c)\)

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

### Candidates

\[
\{ \text{m-fetch}(r_1,c_2), \text{m-fetch}(r_2,c_2) \}
\]

### Procedure RAE

- **loop**:
  - for every new external task or event \(\tau\) do
    - choose a method instance \(m\) for \(\tau\)
    - create a refinement stack for \(\tau, m\)
    - add the stack to \(\text{Agenda}\)
  - for each stack \(\sigma\) in \(\text{Agenda}\)
    - call Progress\((\sigma)\)
    - if \(\sigma\) is finished then remove it

- Container locations unknown
- Partially observable
  - Robot only sees current location

**Refinement tree**

```
Candidates = {m-fetch(r_1,c_2), m-fetch(r_2,c_2)}
```

\(r_0 = r_1\)

\(\tau\)

\(\sigma\)

\(m\)

\(\text{m-fetch1}(r_1,c_2)\)

\(\text{fetch}(r_0,c_2)\)

\(\text{lo}c_0\)

\(\text{lo}c_1\)

\(\text{lo}c_2\)

\(\text{lo}c_3\)

\(\text{lo}c_4\)

\(\text{c}1\)

\(\text{c}2\)

\(\text{r}1\)

\(\text{r}2\)
Example

\[m\text{-}fetch1(r,c) \quad r = r_1, \ c = c_2\]

- Container locations unknown
- Partially observable
- Robot only sees current location

### m-fetch1 (r,c)
- **Task:** fetch(r,c)
- **Pre:** pos(c) = unknown
- **Body:**
  - if \( \exists l \ (\text{view}(l) = F) \) then
    - move-to(r,l)
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail

### m-fetch2 (r,c)
- **Task:** fetch(r,c)
- **Pre:** pos(c) ≠ unknown
- **Body:**
  - if loc(r) = pos(c) then
    - take(r,c,pos(c))
  - else do
    - move-to(r,pos(c))
    - take(r,c,pos(c))

---

**Candidates**

\[= \{m\text{-}fetch1(r_1,c_2), m\text{-}fetch1(r_2,c_2)\}\]
Example

- Container locations unknown
- Partially observable
  - Robot only sees current location

m-fetch1\(r,c\)  \(r = r1, c = c2\)

- task: \(\text{fetch}(r,c)\)
- pre: \(\text{pos}(c) = \text{unknown}\)
- body:
  - \(l = \text{loc1}\)
  - if \(\exists l\) (\(\text{view}(l) = F\)) then
    - move-to\(r,l\)
    - perceive\(r,l\)
    - if \(\text{pos}(c) = l\) then
      - take\(r,c,l\)
    - else fetch\(r,c\)
  - else fail

m-fetch2\(r,c\)

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

Progress(\(\sigma\)):

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

Progression:

\(\tau'\leftarrow \text{next step of } m\)
m-fetch1(r,c) \ r = r1, c = c2

- task: fetch(r,c)
- pre: pos(c) = unknown
- body:
  \[ l = \text{loc1} \]
  \[ \text{if } \exists l (\text{view}(l) = F) \text{ then} \]
  \[ \text{move-to}(r,l) \]
  \[ \text{perceive}(r,l) \]
  \[ \text{if } \text{pos}(c) = l \text{ then} \]
  \[ \text{take}(r,c,l) \]
  \[ \text{else fetch}(r,c) \]
  \[ \text{else fail} \]

m-fetch2(r,c)

- task: fetch(r,c)
- pre: pos(c) \neq \text{unknown}
- body:
  \[ \text{if loc}(r) = \text{pos}(c) \text{ then} \]
  \[ \text{take}(r,c,\text{pos}(c)) \]
  \[ \text{else do} \]
  \[ \text{move-to}(r,\text{pos}(c)) \]
  \[ \text{take}(r,c,\text{pos}(c)) \]
  \[ \tau' \]

- Container locations unknown
- Partially observable
  - Robot only sees current location

Example

Refinement tree

![Refinement tree diagram]

Progress(\sigma):

- started \( m \)?
  - yes
  - no

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

- running command status?
  - succeeded
  - failed

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

- \( \tau' \) \leftarrow next step of \( m \)

- assignment
- type(\( \tau' \))
- command

- update state \( s \)
- candidates for \( \tau' \)?
  - yes
  - no

- send \( \tau' \) to the execution platform

- choose a candidate \( m' \)
  - push (\( \tau',m',... \)) onto \( \sigma \)
  - retry \( \tau \) using an untried candidate
Example

m-fetch1(r,c) \[ r = r1, c = c2 \]

- task: fetch(r,c)
- pre: pos(c) = unknown
- body:
  - \[ l = \text{loc1} \]
  - if \( \exists l \) (\( \text{view}(l) = \text{F} \)) then
    - move-to(r,l)
    - perceive(r,l)
  - if pos(c) = l then
    - take(r,c,l)
  - else fetch(r,c)
  - else fail

m-fetch2(r,c)

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

procedure RAE:

loop:

for every new external task or event \( \tau \) do

choose a method instance \( m \) for \( \tau \)

create a refinement stack for \( \tau, m \)

add the stack to Agenda

for each stack \( \sigma \) in Agenda

call Progress(\( \sigma \))

if \( \sigma \) is finished then remove it

Progress(\( \sigma \)):

- \( (\tau,m,i,\text{tried}) \leftarrow \text{top}(\sigma) \)

- started \( m \)?

no

yes

- \( m \)'s current step a command?

no

yes

running

command status?

success

more steps in \( m \)?

no

yes

failed

retry \( \tau \) using an untried candidate

- \( \tau' \leftarrow \text{next step of } m \)

- pop(\( \sigma \))

assignment

type(\( \tau' \))

command

- send \( \tau' \) to the execution platform

- update state \( s \)

- candidates for \( \tau' \)?

yes

no

- if yes:
  - choose a candidate \( m' \)
  - push \( (\tau',m',\ldots) \) onto \( \sigma \)
- retry \( \tau \) using an untried candidate
Example

$m$-fetch1($r,c$)  \( r = r_1, c = c_2 \)

- **task**: fetch($r,c$)
- **pre**: pos($c$) = unknown
- **body**:
  - \( l = \text{loc1} \)
  - if \( \exists l \) (view($l$) = F) then
    - move-to($r,l$)
    - perceive($r,l$)
    - if pos($c$) = $l$ then
      - take($r,c,l$)
    - else
      - fetch($r,c$)
  - else
    - fail

$m$-fetch2($r,c$)

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

---

Refinement tree

Progress($\sigma$):
- \((\tau,m,i,\text{tried}) \leftarrow \text{top}(\sigma)\)

- if \( \tau \) is \( m \)'s current step a command?
  - yes
    - return
    - succeeded
    - more steps in \( m \)?
      - yes
        - \( \tau \) ← next step of \( m \)
        - type($\tau'$)
        - send $\tau'$ to the execution platform
        - candidates for $\tau'$?
          - yes
            - choose a candidate $m'$
            - push ($\tau'$,$m'$,\ldots) onto $\sigma$
          - no
            - retry $\tau$ using an untried candidate
      - no
        - retry $\tau$ using an untried candidate
    - \( \tau \) ← failed
    - \( \tau \) ← next step of \( m \)
    - \( \tau \) ← failed
    - pop($\sigma$)

- no
  - started \( m \)?
    - yes
      - return
      - succeeded
      - more steps in \( m \)?
        - yes
          - \( \tau \) ← next step of \( m \)
          - type($\tau'$)
          - send $\tau'$ to the execution platform
          - candidates for $\tau'$?
            - yes
              - choose a candidate $m'$
              - push ($\tau'$,$m'$,\ldots) onto $\sigma$
            - no
              - retry $\tau$ using an untried candidate
        - no
          - retry $\tau$ using an untried candidate
Example

Refinement tree

\[ \text{Progress(}\sigma\text{): } (r,m,i,\text{tried}) \leftarrow \text{top(}\sigma\text{)} \]

\[ \begin{array}{c}
\text{no} \\
\text{started } m\text{?} \\
\text{no} \\
\text{m’s current step a command?} \\
\text{yes} \\
\text{command status?} \\
\text{running} \\
\text{return succeeded} \\
\text{more steps in } m\text{?} \\
\text{yes} \\
\text{no} \\
\text{retry } \tau \text{ using an untried candidate} \\
\text{yes} \\
\text{no} \\
\text{pop(}\sigma\text{)} \\
\end{array} \]

\[ \begin{array}{c}
\text{yes} \\
\text{\( \tau' \leftarrow \text{next step of } m \)} \\
\text{assignment} \\
\text{type(}\tau'\text{)} \\
\text{command} \\
\text{update state } s \\
\text{candidates for } \tau'\text{?} \\
\text{yes} \\
\text{no} \\
\text{retry \( \tau \text{ using an untried candidate}} \\
\text{yes} \\
\text{no} \\
\end{array} \]

\[ \begin{array}{c}
\text{choose a candidate } m' \\
\text{push (}\tau',m',\ldots\text{) onto } \sigma \\
\end{array} \]

\[ \begin{array}{c}
m-\text{fetch1}(r,c) \quad r = r1, \ c = c2 \\
\text{task: } \text{fetch}(r,c) \\
\text{pre: } \text{pos}(c) = \text{unknown} \\
\text{body:} \quad l = \text{loc1} \\
\quad \text{if } \exists l \ (\text{view}(l) = F) \text{ then} \\
\quad \text{move-to}(r,l) \\
\quad \text{perceive}(r,l) \\
\quad \text{if } \text{pos}(c) = l \text{ then} \\
\quad \text{take}(r,c,l) \\
\quad \text{else fetch}(r,c) \\
\quad \text{else fail} \\
\end{array} \]

\[ \begin{array}{c}
m-\text{fetch2}(r,c) \\
\text{task: } \text{fetch}(r,c) \\
\text{pre: } \text{pos}(c) \neq \text{unknown} \\
\text{body:} \quad \text{if } \text{loc}(r) = \text{pos}(c) \text{ then} \\
\quad \text{take}(r,c,\text{pos}(c)) \\
\quad \text{else do} \\
\quad \text{move-to}(r,\text{pos}(c)) \\
\quad \text{take}(r,c,\text{pos}(c)) \\
\end{array} \]
Example

m-fetch1(r,c)  \( r = r_1, c = c_2 \)
- task: fetch(r,c)
- pre: pos(c) = unknown
- body:
  - if \( \exists l \) (view(l) = F) then
    - move-to(r,l)
    - perceive(r,l)
  - if pos(c) = l then
    - take(r,c,l)
  - else fetch(r,c)
- else fail

m-fetch2(r,c)  \( r = r_1, c = c_2 \)
- task: fetch(r,c)
- pre: pos(c) ≠ unknown
- body:
  - if loc(r) = pos(c) then
    - take(r,c,pos(c))
  - else do
    - move-to(r,pos(c))
    - take(r,c,pos(c))

Progress(\( \sigma \)):
- started \( m \)?
- yes
  - \( m \)'s current step a command?
    - yes
      - command status?
        - yes
          - more steps in \( m \)?
            - yes
              - succeeded
            - no
              - retry \( \tau \) using an untried candidate
        - no
          - returned
          - updated state \( s \)
    - no
      - no more steps in \( m \)?
        - yes
          - retry \( \tau \) using an untried candidate
        - no
          - failed

Refinement tree

\[ \tau' \leftarrow \text{next step of } m \]

Assignment
- type(\( \tau' \))
- task
- candidates for \( \tau' \)?
  - yes
    - choose a candidate \( m' \)
      - push (\( \tau', m', \ldots \)) onto \( \sigma \)
  - no
    - retry \( \tau \) using an untried candidate
m-fetch1(r,c)  \( r = r1, c = c2 \)

- **task:** fetch\((r,c)\)
- **pre:** \( \text{pos}\(c\) = unknown\)
- **body:**
  - \( l = \text{loc1} \)
  - if \( \exists l\ (\text{view}\(l\) = F) \) then
    - move-to\((r,l)\)
    - perceive\((r,l)\)
  - if \( \text{pos}\(c\) = l \) then
    - take\((r,c,l)\)
  - else fetch\((r,c)\)
  - else fail

---

m-fetch2(r,c)

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

---

**Example**

- **Refinement tree**
  - \( \text{fetch}\((r0,c2)\) \)
  - \( r0 = r1 \)
  - \( m\)-fetch1\((r1,c2)\)

---

**Progress(\(\sigma\)):**

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

**type(\(\tau\))**

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

---

**Progress(\(\sigma\)):**

- (\(\tau,m,i,\text{tried}\)) \leftarrow \text{top}(\sigma)

---

** Fuse state s**

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

---

**Nau – Lecture slides for Automated Planning and Acting**
m-fetch1(r,c)  \( r = r2, c = c2 \)

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

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

**Body:**
- if \( \exists l \) (view(l) = F) then
  - move-to(r,l)
- perceive(r,l)
- if pos(c) = l then
  - take(r,c,l)
- else fetch(r,c)
- else fail

**Example**

```
Candidates = \{ m-fetch(r1,c2), m-fetch(r2,c2) \}
```

m-fetch2(r,c)

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

**Pre:** pos(c) \( \neq \) unknown

**Body:**
- if loc(r) = pos(c) then
  - take(r,c,pos(c))
- else do
  - move-to(r,pos(c))
  - take(r,c,pos(c))

Refinement tree:

- `fetch(r0,c2)`

Code execution:

- move-to(r1,loc1)
- perceive(r1,loc1)

Sensor failure:

- retry

```
\tau = \text{m's current step in m?}
\text{command status?}
\text{retrieved}\text{ }\text{using an untried candidate}
```

Progress(\( \sigma \)):

- \((\tau, m, i, \text{tried}) \leftarrow \text{top}(\sigma)\)
- \text{started } m? \text{?}
- \text{running}\text{ ?}
- \text{more steps in } m? \text{?}
- \text{pop}(\sigma)\text{ ?}

Assignment:

- type(\( \tau' \))
- task
- send \( \tau' \) to the execution platform

Candidates for \( \tau' \)?

- yes
- no

No more steps in \( m \)?

- yes
- no

Choose a candidate \( m' \)

- push (\( \tau', m', \ldots \)) onto \( \sigma \)

Retry \( \tau \) using an untried candidate

- yes
- no
Example

\[ \text{m-fetch1}(r,c) \quad r = r_2, \ c = c_2 \]

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

\[ \text{Candidates} = \{ \text{m-fetch1}(r_1,c_2), \text{m-fetch1}(r_2,c_2) \} \]

\[ \text{m-fetch2}(r,c) \]

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

\[ \text{Is this the same as a backtracking search?} \]

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

- started m?
  - \[ \text{no} \]
  - \[ \text{yes} \]
  - m’s current step a command?
    - \[ \text{no} \]
    - \[ \text{runs} \]
    - \[ \text{return} \]
    - \[ \text{succeeded} \]
    - \[ \text{more steps in m?} \]
      - \[ \text{no} \]
      - \[ \text{retry} \]
      - \[ \text{pop}(\sigma) \]
    - \[ \text{yes} \]
  - \[ \text{retry} \] \( \tau \) using an untried candidate

- \[ \text{no more steps in m?} \]
  - \[ \text{yes} \]
    - \[ \text{return} \]
    - \[ \text{succeeded} \]
    - \[ \text{more steps in m?} \]
      - \[ \text{no} \]
      - \[ \text{retry} \]
      - \[ \text{pop}(\sigma) \]
    - \[ \text{yes} \]
      - \[ \text{retry} \] \( \tau \) using an untried candidate

- \[ \text{no} \]
  - \[ \text{pop}(\sigma) \]

- \[ \text{yes} \]
  - \[ \text{retry} \] \( \tau \) using an untried candidate
Extensions to RAE

- Methods for events
  - e.g., an emergency
- Methods for goals
  - special kind of task: achieve(goal)
  - sets up a monitor to see if the goal has been achieved
- Concurrent subtasks
Outline

1. Motivation
2. Representation
3. Acting (Rae)
4. *Planning for Rae*
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application
Planning for Rae

procedure RAE:
    loop:
        for every new external task or event \( \tau \) do
            choose a method instance \( m \) for \( \tau \)
            create a refinement stack for \( \tau, m \)
            add the stack to Agenda
        for each stack \( \sigma \) in Agenda
            call Progress(\( \sigma \))
        if \( \sigma \) is finished then remove it

- Bad choice may lead to
  - more costly solution
  - failure - need to recover, sometimes unrecoverable

- Solution:
  - call a planner, choose the method instance it suggests

Progress(\( \sigma \)):
- (\( \tau, m, i, \text{tried} \)) \leftarrow \text{top}(\( \sigma \))

- started \( m \)?
  - yes
  - no

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

- command status?
  - succeeded
  - running
  - failed

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

- \( \tau' \leftarrow \text{next step of } m \)
- pop(\( \sigma \))

assignment
- type(\( \tau' \))
- command
- task
- send \( \tau' \) to the execution platform

- update state \( s \)
- candidates for \( \tau' \)?
- yes
- no

- choose a candidate \( m' \)
- push (\( \tau', m', \ldots \)) onto \( \sigma \)
- retry \( \tau \) using an untried candidate
Planning and Acting Integration

- Recall:
  - To maintain consistency, we want both actor and planner to use the same operational models.

- Idea 1:
  - Planner uses Rae’s tasks and refinement methods.
  - DFS or GBFS search among alternatives to see which works best.
  - Use classical action models to predict outcomes of commands.

Diagram showing the integration of planning and acting with operational and descriptive models.
SeRPE (Sequential Refinement Planning Engine)

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

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

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

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

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

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

Like Rae with just one external task

- Progress it all the way to the end, like Progress with a loop around it
- Plan rather than act
  - For each command, apply a classical action model
  - Don’t do retries
  - Classical action model would give same outcome every time

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

\( i \leftarrow \text{nil} \) // instruction pointer for body(\( m \))

\( \pi \leftarrow \langle \rangle \) // 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 type(m[i])

assignment: update \( s \) according to \( m[i] \)

command:

\( a \leftarrow \text{the descriptive model of } m[i] \) 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 \( \pi' \) = failure then return failure

\( s \leftarrow \gamma(s, \pi'); \pi \leftarrow \pi.\pi' \)
Limitations

- **Problem 1**: difficult to implement
  - Each time a method invokes a subtask, SeRPE makes a nondeterministic choice
  - To implement deterministically
    - Need snapshot of current execution state
    - Would need to write our own compiler or interpreter
  - Is it worth the effort?

- **Problem 2**: limitations of classical action models

- Classical action models for the *fetch* example

```plaintext
take(r,o,l)
// robot r takes object o at location l
pre: cargo(r) = nil, loc(r) = l, loc(o) = l
eff: cargo(r) ← o, loc(o) ← r
```

```plaintext
put(r,o,l)
// r puts o at location l
pre: loc(r) = l, loc(o) = r
eff: cargo(r) ← nil, loc(o) ← l
```

```plaintext
perceive(r,l):
// robot r sees what objects are at l ?
```

- No good model for perceive
  - If we knew the outcome in advance, perception wouldn’t be necessary
Simulating commands

- Simplest case:
  - probabilistic action template
    \[ a(x_1, \ldots, x_k) \]
    
    \[ \text{pre: } \ldots \]
    
    \[ (p_1) \text{ effects}_1: e_{11}, e_{12}, \ldots \]
    
    \[ \ldots \]
    
    \[ (p_m) \text{ effects}_m: e_{m1}, e_{m2}, \ldots \]
  
  - Choose effects \( i \) at random with probability \( p_i \) and use it to update the current state

- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get simulated effects
**Planning for Rae**

procedure RAE:

loop:

for every new external task or event $\tau$ do

  choose a method instance $m$ for $\tau$

  create a refinement stack for $\tau$, $m$

  add the stack to Agenda

for each stack $\sigma$ in Agenda

  call Progress($\sigma$)

if $\sigma$ is finished then remove it

- Idea 2: simulation with multithreading or multiprocessing
  - Run Rae in simulated environment
  - Simulate commands as shown on previous page
  - To choose among candidate method instances, try all of them in parallel
- Planner returns the candidate $m$ having the highest expected utility

**Poll:** is this a reasonable approach?
Planning for Rae

procedure RAE:

loop:
  for every new external task or event $\tau$ do
    choose a method instance $m$ for $\tau$
    create a refinement stack for $\tau, m$
    add the stack to Agenda
  for each stack $\sigma$ in Agenda
    call Progress($\sigma$)
    if $\sigma$ is finished then remove it

- Idea 3: simulation with Monte Carlo rollouts
  - Multiple runs, random choices and outcomes in each run
  - Maintain statistics to estimate expected utility of each choice
  - Return the candidate $m$ having has the highest estimated utility

Plan-with-UPOM (task $\tau$):

$\text{Candidates} \leftarrow \{\text{method instances relevant for } \tau\}$

for $i \leftarrow 1$ to $n$

call UPOM($\tau$)

update estimates of methods’ expected utility

return the $m \in \text{Candidates}$ that has

the highest estimated utility

UPOM($\tau$):

choose a method instance $m$ for $\tau$

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

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

if $\sigma$ is completed then return $(m, \text{utility})$

return failure

- Each call to UPOM does a Monte Carlo rollout
  - Simulated execution of RAE on $\tau$

Simulate-Progress($\sigma$):

started $m$?

is $m$’s current step a command?

running

simulation status

more steps in $m$?

return success

succeeded

more steps in $m$?

yes

$\tau' \leftarrow$ next step of $m$

no

pop($\sigma$)

no more steps in $m$?

return success

succeeded

more steps in $m$?

yes

$\tau' \leftarrow$ next step of $m$

no

pop($\sigma$)

no more steps in $m$?

return success

succeeded

more steps in $m$?

yes

$\tau' \leftarrow$ next step of $m$

no

pop($\sigma$)

no more steps in $m$?

return success

succeeded

more steps in $m$?

yes

$\tau' \leftarrow$ next step of $m$

no

pop($\sigma$)

no more steps in $m$?
Monte-Carlo rollouts

Plan-with-UPOM (task $\tau$):

$Candidates \leftarrow \{\text{method instances relevant for } \tau\}$

for $i \leftarrow 1$ to $n$

1. call UPOM($\tau$)

2. update estimates of methods’ expected utility

return the $m \in Candidates$ that has

the highest estimated utility

UPOM($\tau$):

1. choose a method instance $m$ for $\tau$

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

3. loop while Simulate-Progress($\sigma$) $\neq$ failure

   a. if $\sigma$ is completed then return ($m$, utility)

4. return failure

- Each call to UPOM does a Monte Carlo rollout
  - Simulated execution of RAE on $\tau$

Nau – Lecture slides for Automated Planning and Acting
Digression: background on Monte Carlo rollouts

- Multi-arm bandit problem
- Statistical model of sequential experiments
  - Name comes from a traditional slot machine (one-armed bandit)
- Multiple actions $a_1, a_2, \ldots, a_n$
  - Each $a_i$ provides a reward from an unknown probability distribution $p_i$
  - Assume each $p_i$ is *stationary*
    - Same every time, regardless of history
  - Objective: maximize expected utility of a sequence of actions
- Exploitation vs exploration dilemma:
  - *Exploitation*: choose action that has given you high rewards in the past
  - *Exploration*: choose action that’s less familiar, in hopes that it might produce a higher reward
UCB (Upper Confidence Bound) Algorithm

- Assume all rewards are between 0 and 1
  - If they aren’t, normalize them
- For each action $a$, let
  - $r(a)$ = average reward you’ve gotten from $a$
  - $n(a)$ = number of times you’ve tried $a$
  - $n_t = \sum_a n(a)$
  - $Q(a) = r(a) + \sqrt{\frac{2 \ln n_t}{n(a)}}$

UCB:
- if there are any untried actions:
  - $\tilde{a} \leftarrow$ any untried action
- else:
  - $\tilde{a} \leftarrow \arg\max_a Q(a)$
  - perform $\tilde{a}$
  - update $r(\tilde{a})$, $n(\tilde{a})$, $n_t$, $Q(a)$

- Theorem (given some assumptions):
  As the number of calls to UCB → ∞,
  UCB’s choice at each call → optimal choice
UCT Algorithm

● MDP: state space in which each action has probabilistic outcomes
  ▶ We’ll discuss this in Chapter 6
● UCT algorithm: Monte Carlo rollouts on an MDP
● At each state $s$,
  ▶ Use UCB to choose an action at random
    • Balances exploration vs exploitation at $s$
  ▶ Action has random outcome

● How to use UCT:
  ▶ Call it many times, return action with highest expected utility
● Theorem:
  As number of calls to UCT $\rightarrow \infty$, choice converges to optimal
**Convergence**

- **UCT algorithm:**
  - Monte Carlo rollouts on MDPs
  - Call it many times, choice converges to optimal

- **UPOM search tree more complicated**
  - tasks, method instances, commands, code execution

- If no exogenous events,
  - Can map it to UCT search of a complicated MDP
  - Proof of convergence to optimal
Outline

1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. *Acting with Planning* (RAE+UPOM)
6. Learning
7. Evaluation, Application
RAE + UPOM

procedure RAE:
  loop:
    for every new external task or event \( \tau \) do
      choose a method instance \( m \) for \( \tau \)
      create a refinement stack for \( \tau, m \)
      add the stack to Agenda
    for each stack \( \sigma \) in Agenda
      call Progress(\( \sigma \))
      if \( \sigma \) is finished then remove it

- Whenever RAE needs to choose a method instance
  - call Plan-with-UPOM, use the method instance it returns

- Open-source Python implementation: [https://bitbucket.org/sunandita/RAE/](https://bitbucket.org/sunandita/RAE/)

Progress(\( \sigma \)):

- \( \text{started } m? \)
  - \( \text{yes} \)
    - \( \text{is } m\text{'s current step a command?} \)
      - \( \text{yes} \)
        - return \text{command status}
      - \( \text{failed} \)
        - retry \( \tau \) using an untried candidate
    - \( \text{no} \)
      - \( \text{return success} \)
      - \( \text{more steps in } m? \)
        - \( \text{yes} \)
          - \( \tau \leftarrow \text{next step of } m \)
          - \( \text{assignment} \)
            - \( \text{type}(\tau') \)
              - \( \text{command} \)
                - \( \text{task} \)
                  - \( \text{send } \tau' \text{ to the execution platform} \)
                    - \( \text{yes} \)
                      - \( \text{update state} \)
                        - \( \text{candidates for } \tau'? \)
                          - \( \text{yes} \)
                            - choose a candidate \( m' \)
                              - push \( (\tau', m', ...) \) onto \( \sigma \)
                          - \( \text{no} \)
                            - retry \( \tau \) using an untried candidate
        - \( \text{no} \)
          - \( \text{pop}(\sigma) \)
            - \( \text{no more steps in } m? \)
              - \( \text{yes} \)
                - \( \tau \leftarrow \text{next step of } m \)
                - \( \text{assignment} \)
                  - \( \text{type}(\tau') \)
                    - \( \text{command} \)
                      - \( \text{task} \)
                        - \( \text{send } \tau' \text{ to the execution platform} \)
                          - \( \text{yes} \)
                            - \( \text{update state} \)
                              - \( \text{candidates for } \tau'? \)
                                - \( \text{yes} \)
                                  - choose a candidate \( m' \)
                                    - push \( (\tau', m', ...) \) onto \( \sigma \)
                                - \( \text{no} \)
                                  - retry \( \tau \) using an untried candidate
                          - \( \text{no} \)
                          - \( \text{retry } \tau \) using an untried candidate
Run-Lookahead + UPOM?

• Suppose we try to use Run-Lookahead with a modified version of UPOM (call it UPOM’)
  ▸ Instead of returning method instance \( m_1 \), return the actions in the last Monte Carlo rollout
    • \( \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \)
    • corresponding commands: \( c_1, c_2, c_3, c_4, c_5 \)

• Big problem
  ▸ Run-lookahead calls UPOM’, gets \( \pi \), executes \( c_1 \), then calls UPOM’ again
  ▸ This time, UPOM’ needs to plan for \( t_1 \) in state \( s_1 \) rather than \( s_0 \)
  ▸ There might not be an applicable method

• If we want to use Run-Lookahead, we need methods that can start at any state along the way

Suppose we try to use Run-Lookahead with a modified version of UPOM (call it UPOM’)

• Instead of returning method instance \( m_1 \), return the actions in the last Monte Carlo rollout
  • \( \pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle \)
  • corresponding commands: \( c_1, c_2, c_3, c_4, c_5 \)

Big problem

• Run-lookahead calls UPOM’, gets \( \pi \), executes \( c_1 \), then calls UPOM’ again
• This time, UPOM’ needs to plan for \( t_1 \) in state \( s_1 \) rather than \( s_0 \)
• There might not be an applicable method

If we want to use Run-Lookahead, we need methods that can start at any state along the way
Run-Lazy Lookahead + UPOM?

- Things are better if we use Run-Lazy-Lookahead
- Run-lookahead calls UPOM’, UPOM’ returns $\pi = \{a_1, a_2, a_3, a_4, a_5\}$
- Run-Lookahead executes $c_1, c_2, c_3, c_4, c_5$, won’t call UPOM’ again unless something unexpected happens, e.g.,
  - command $c_2$ has an execution failure
  - $c_2$ produces a state in which $c_3$ is inapplicable
  - or an exogenous event makes $c_3$ inapplicable
    - Method $m_2$ fails; we need to replan task $t_2$
- Need to modify Run-Lazy-Lookahead so that when a failure occurs, it knows which task to replan
Comparison

- Rae + UPOM has tighter coupling between planning and acting
  - works better than Run-Lazy-Lookahead + UPOM’
- Example
  - Case 1: Run-Lazy-Lookahead calls UPOM’ for $t_1$ in state $s_0$
    - UPOM’ returns $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
    - corresponding commands: $c_1, c_2, c_3, c_4, c_5$
    - Run-Lazy-Lookahead executes $c_1$, gets state $s_1'$ (not $s_1$)
      - Suppose this makes action $a_2$ redundant
      - Run-Lazy-Lookahead doesn’t have a way to detect this; continues with the rest of $\pi$
  - Case 2: Rae calls UPOM for $t_1$ in state $s_0$
    - UPOM returns $m_1$, Rae executes $c_1$, gets state $s_1'$
    - Rae calls UPOM for $t_2$ in state $s_1'$
      - UPOM might return a better method instance
      - Or maybe UPOM returns $m_2$, but $m_2$’s body includes an if-test to omit $a_2$ if it’s redundant
1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application
Motivation

- Plan-with-UPOM is called by RAE, runs online
  - Time constraints might not allow complete search

- Case 1: no time to search at all
  - need a choice function

- Case 2: enough time to do partial search
  - Receding horizon
    - Cut off search at depth $d_{\text{max}}$ or when we run out of time
    - At leaf nodes, use heuristic function to estimate expected utility

- Learning algorithms:
  - Learn$\pi$: learns a choice function
  - LearnH: learns a heuristic function
Integration with Learning

- Gather training data from acting-and-planning traces of RAE and Plan-with-UPOM
- Train classifiers (multi-layered perceptrons)

Learn $\pi$
- Learns function for choosing a method
- Given current task and context (state and other information), choose $m$ from the set of available refinement methods
- Useful if there isn’t enough time to use UPOM
Integration with Learning

- Gather training data from acting-and-planning traces of RAE and Plan-with-UPOM
- Train classifiers (multi-layered perceptrons)

LearnH
- Learns a heuristic function to guide UPOM’s search
- UPOM can use it to estimate expected utility at leaf nodes
- Useful if there isn’t enough time to search all the way to the end

Actor:

Planning (UPOM)  LearnH  Learn\(\pi\)

m, task, context  \(h(m)\)

Task, context  m

Acting Engine (RAE)
Outline

1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application
## Experimental Evaluation

- Five different domains, different combinations of characteristics
- Evaluation criteria: efficiency (reciprocal of cost), successes vs failures
- Result: Planning and learning help
  - RAE operates better with UPOM or learning than without
  - RAE’s performance improves with more planning

| Domain | $|\mathcal{T}|$ | $|\mathcal{M}|$ | $|\bar{\mathcal{M}}|$ | $|\mathcal{A}|$ | Dynamic events | Dead ends | Sensing | Robot collaboration | Concurrent tasks |
|--------|-----------|-----------|-------------|-----------|----------------|------------|--------|-------------------|-----------------|
| S&R    | 8         | 16        | 16          | 14        | ✓              | ✓          | ✓      | ✓                 | ✓               |
| Explore| 9         | 17        | 17          | 14        | ✓              | ✓          | ✓      | ✓                 | ✓               |
| Fetch  | 7         | 10        | 10          | 9         | ✓              | ✓          | ✓      | –                 | ✓               |
| Nav    | 6         | 9         | 15          | 10        | ✓              | –          | ✓      | ✓                 | ✓               |
| Deliver| 6         | 6         | 50          | 9         | ✓              | ✓          | –      | ✓                 | ✓               |
Prototype Application

● Software-defined networks
  ▪ Decoupled control and data layers
  ▪ Prone to high-volume, fast-paced online attacks
  ▪ Need automated attack recovery

● Prototype solution using RAE+UPOM
  ▪ Expert writes recovery procedures as refinement methods

Prototype Application

- SDN domain modeled with 11 tasks, 28 refinement methods, 13 commands
- 50 autogenerated problems each containing one or more random attacks
- Utility optimized is *resilience*
  - Linear combination of efficiency (reciprocal of cost) and success ratio

- Experimental results
  - Improved efficiency, retry ratio, success ratio, resilience compared to human expert

Billions of Data Points

Millions of Alerts

High-volume, fast-paced Cyber Events

Cyber Warriors

Complex Systems to Defend
Summary

● 3.1 Operational models
  ▶ ξ versus s, tasks, events,
  ▶ Commands to the execution platform
  ▶ Extensions to state-variable representation
  ▶ Refinement method
    • name, task/event, preconditions, body
  ▶ Example: fetch a container

● 3.2 Refinement Acting Engine (RAE)
  ▶ Purely reactive: select a method and apply it
  ▶ Rae: input stream, Candidates, Instances, Agenda, refinement stacks
  ▶ Progress:
    • command status, nextstep, type of step
  ▶ Retry: Candidates \ tried
    • comparison to backtracking
  ▶ Refinement trees

● 3.3 Refinement planning
  ▶ plan by simulating Rae on a single external task/event/goal
  ▶ SeRPE uses classical action models
  ▶ UPOM simulates the actor’s commands, does Monte Carlo rollouts

● 3.4 Acting and planning
  ▶ Rae + UPOM
  ▶ Comparison: Run-Lazy-Lookahead + UPOM’
  ▶ A little about learning, experimental evaluation, prototype application

● Open-source Python implementation of Rae and UPOM:
  ▶ https://bitbucket.org/sunandita/RAE/