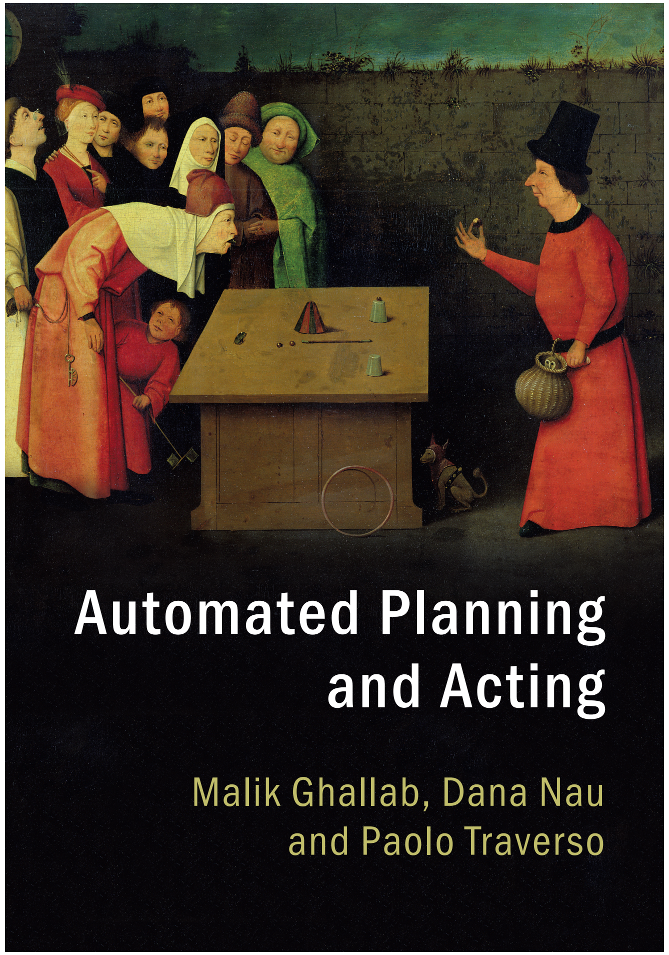
Last update: 3:24 PM, March 10, 2022

Chapter 3 Deliberation with Refinement Methods

Dana S. Nau
University of Maryland

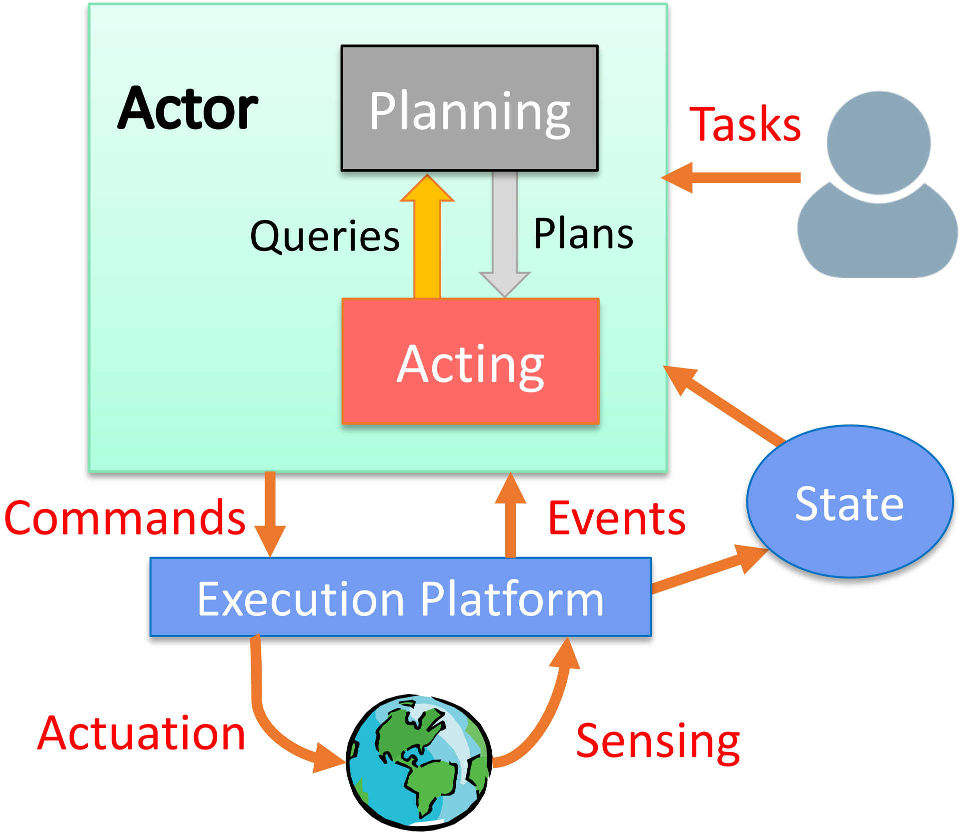

Automated Planning and Acting
Malik Ghallab, Dana Nau and Paolo Traverso

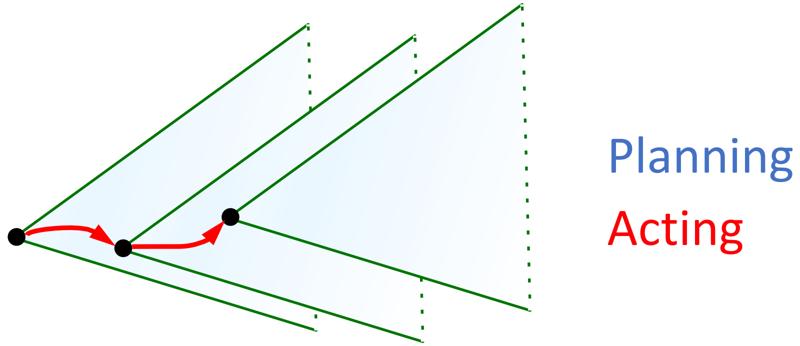
http://www.laas.fr/planning

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

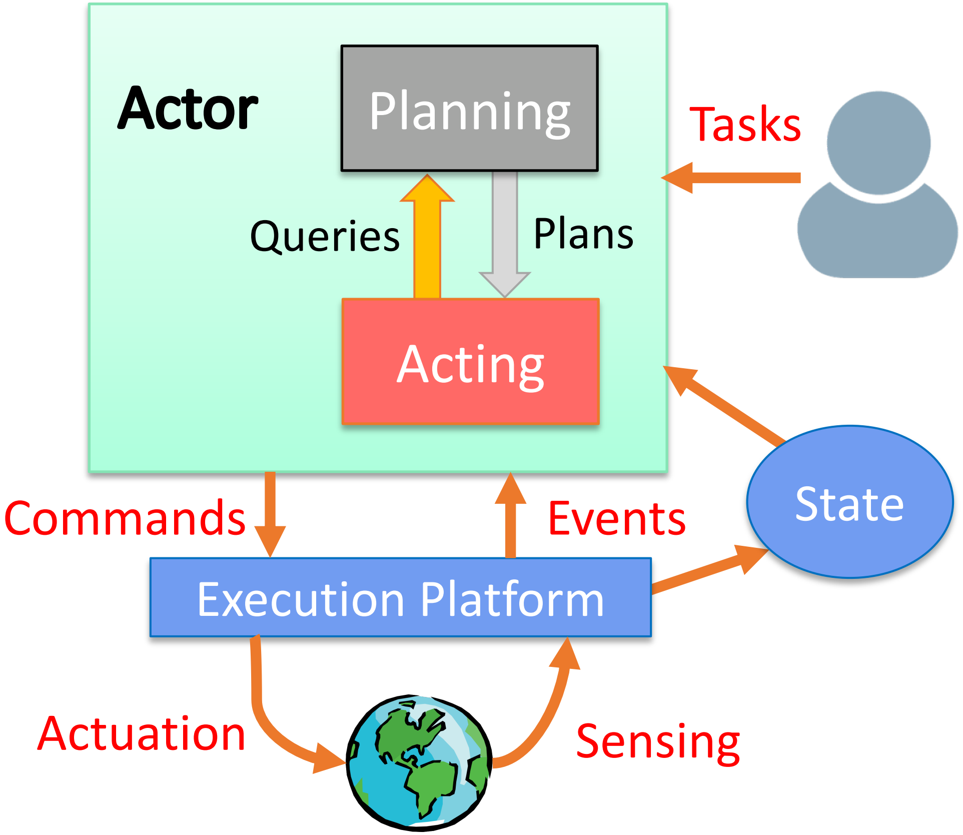
- **Acting**: *performing*
  - Dynamic, unpredictable, partially observable environment
    - Adapt to context, react to events
  - Uses *operational* models
    - instructions telling *how* to perform the tasks
    - usually hierarchical

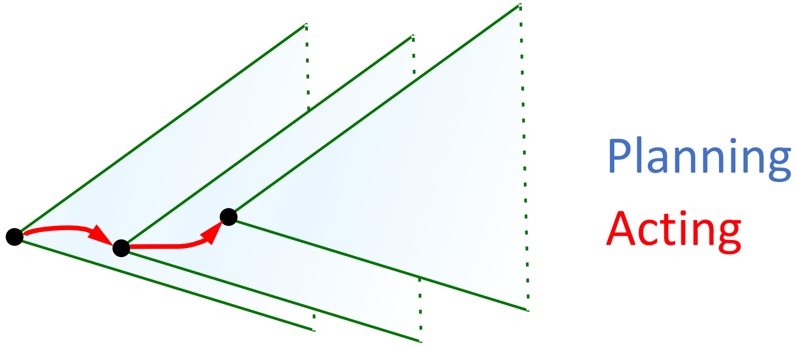




Outline

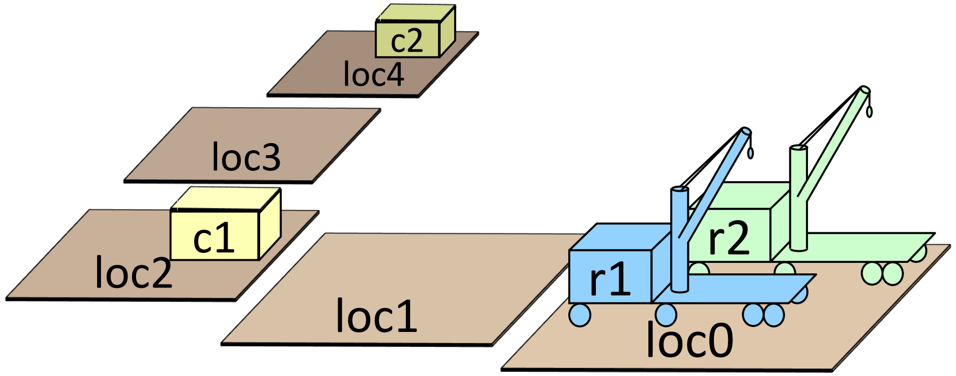
1. Motivation
2. *Representation*
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application





Example

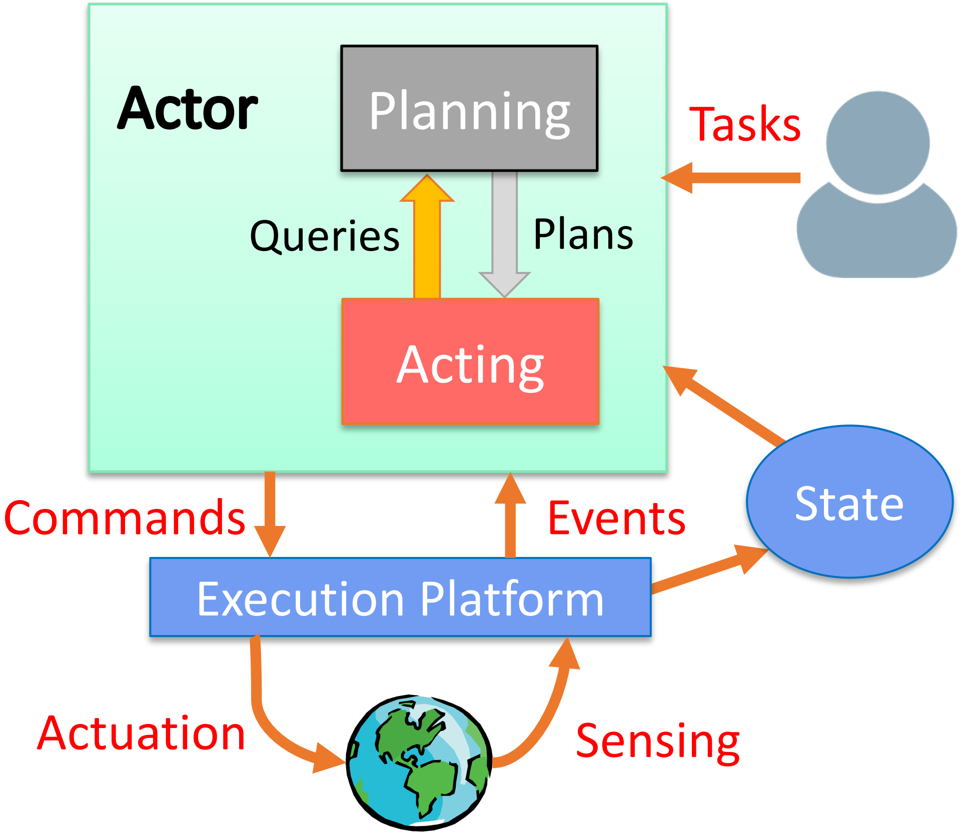
- Consider an actor that controls two robots
- Environment is *partially observable*
  - Each robot can only see what's at the current location

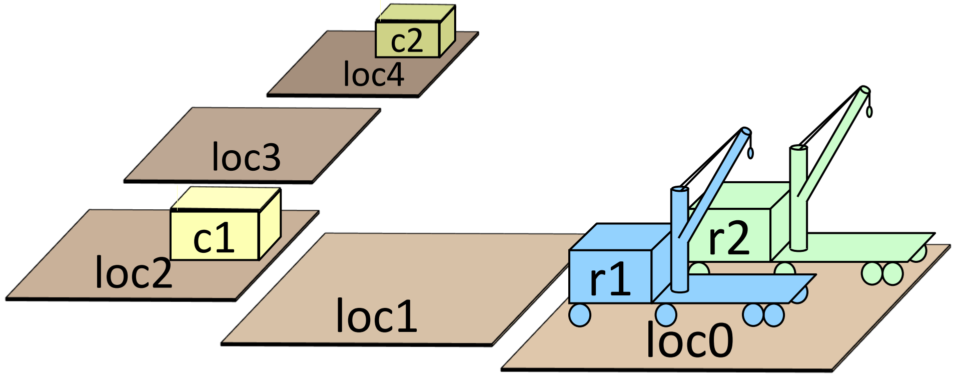


- Objects
  - *Robots* = {r1, r2}
  - *Containers* = {c1, c2}
  - *Locations* = {loc0, loc1, loc2, loc3, loc4}
- Rigid relations (properties that won't change)
  - adjacent(loc0,loc1), adjacent(loc1,loc0), adjacent(loc1,loc2), adjacent(loc2,loc1), adjacent(loc2,loc3), adjacent(loc3,loc2), adjacent(loc3,loc4), adjacent(loc4,loc3)

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

Example (continued)

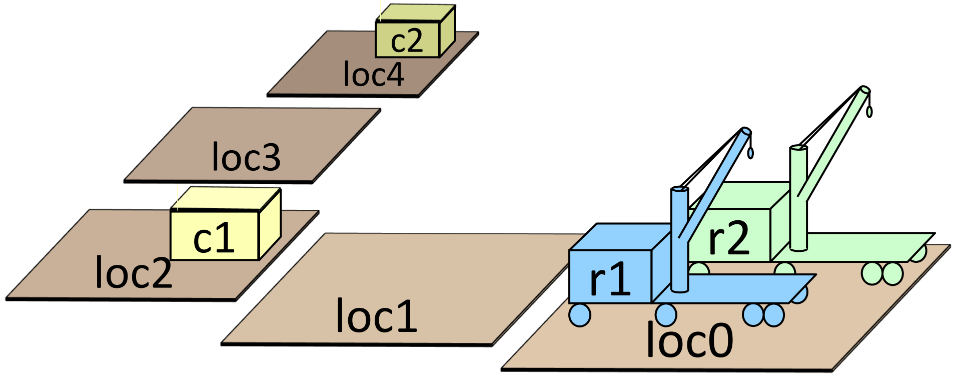




- Commands to the execution platform:
  - take(r,o,l): r takes object o at location l
  - put(r,o,l): r puts o at location l
  - perceive(r,l): robot r perceives what objects are at l
  - move-to(r,l): robot r moves to location l

Tasks and Methods

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



```
method-name($arg_1, \ldots, arg_k$)
    task:   *task-identifier*
    pre:    *test*
    body:
            *a program*
```

- assignment statements
- control constructs:
  - if-then-else, while, … .
- tasks
  - can extend this to include events, goals
- commands to the execution platform

```
m-fetch1($r,c$)
    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
```

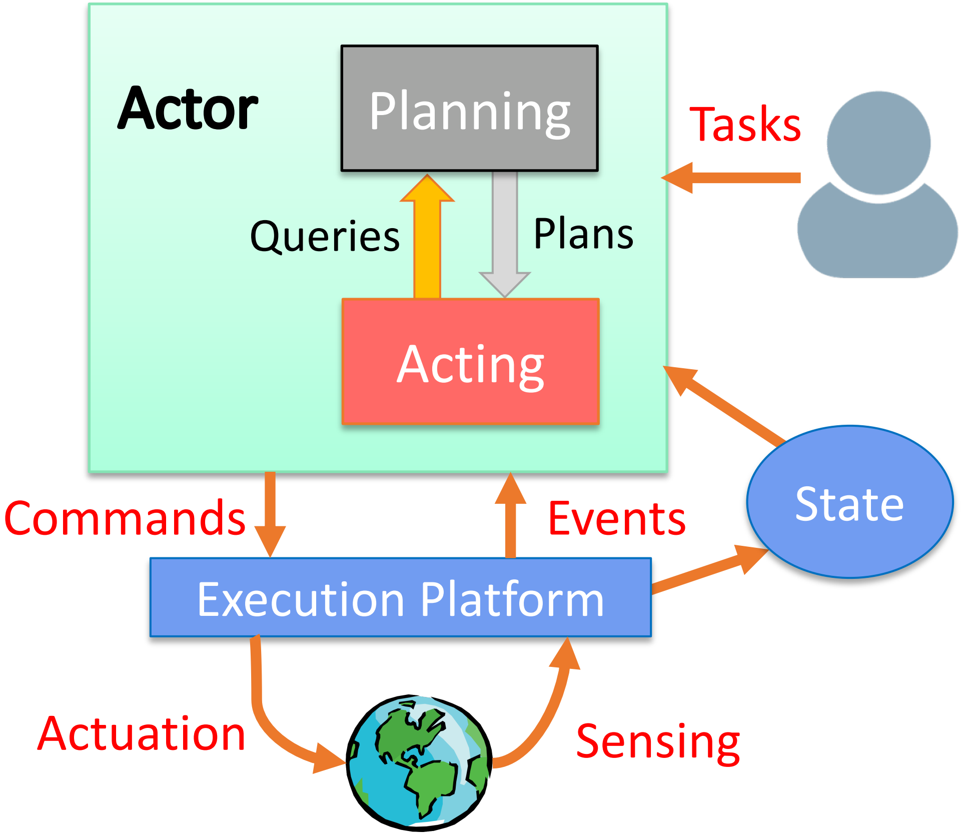
```
m-fetch2($r,c$)
    task:  fetch($r,c$)
    pre:   pos($c$) $\neq$ unknown
    body:
        if loc($r$) = pos($c$) then
            take($r,c$,pos($c$))
        else do
            move-to($r$,pos($c$))
            take($r,c$,pos($c$))
```

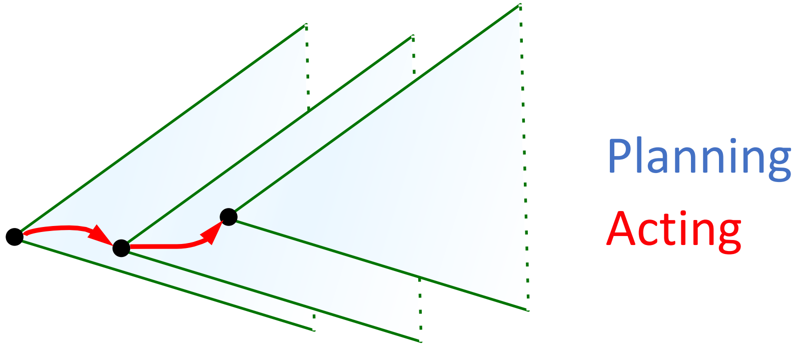
command

task

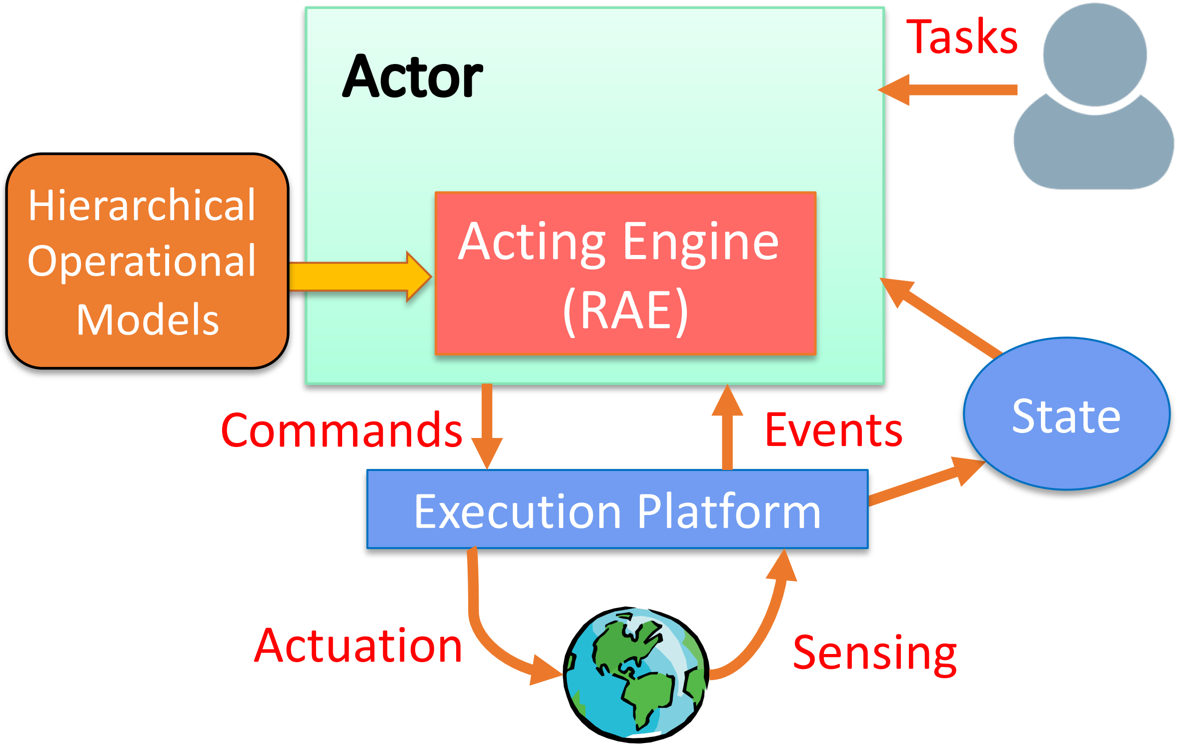
Outline

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

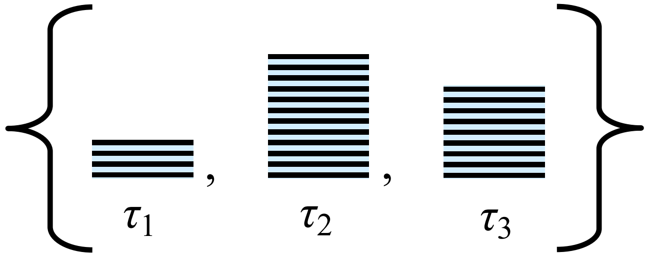




RAE (Refinement Acting Engine)



- Performs multiple tasks in parallel
  - Purely reactive, no lookahead
- For each task or event τ, a *refinement stack*
  - execution stack
- *Agenda* = {all current refinement stacks}

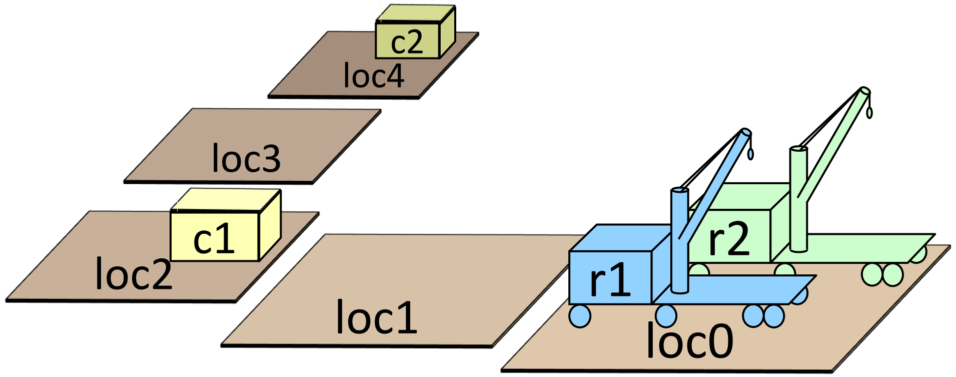


- Refinement stack for a task τ
  ⇔ current ~~path~~ in RAE's search tree for τ

  refinement tree

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

Example (reminder)



- 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* ∈ *Robots*, *c* ∈ *Containers*, *l* ∈ *Locations*
  - loc(*r*) ∈ *Locations*
  - cargo(*r*) ∈ *Containers* ∪ {nil}
  - pos(*c*) ∈ *Locations* ∪ *Robots* ∪ {unknown}
  - view(*l*) ∈ {T, F}
    - Whether a robot has looked at location *l*
    - If view(*l*) = T then pos(*c*) = *l* for every container *c* at *l*

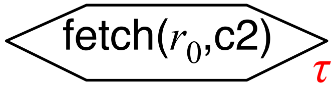
- Commands to the execution platform:
  - take(*r,o,l*): *r* takes object *o* at location *l*
  - put(*r,o,l*): *r* puts *o* at location *l*
  - perceive(*r,l*): robot *r* perceives what objects are at *l*
  - move-to(*r,l*): robot *r* moves to location *l*

Example

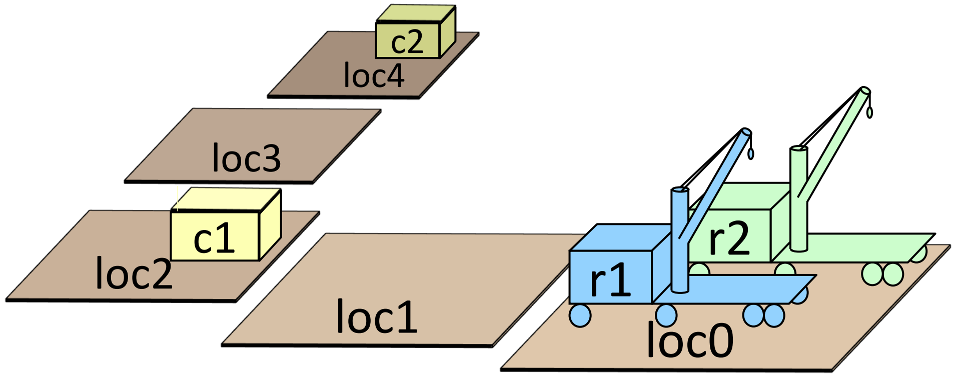
```
m-fetch1($r$,$c$)
    task:   fetch($r$,$c$)
    pre:    pos($c$) = unknown
    body:
        if $\exists l$ (view($l$) = F) then
            move-to($r$,$l$)
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                take($r$,$c$,$l$)
            else fetch($r$,$c$)
      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

Refinement tree



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

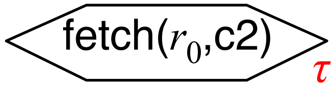


```
procedure RAE:
    loop:
        for every new external task or event $\tau$ do
            choose a method instance $m$ for $\tau$
            create a refinement stack for $\tau$, $m$
            add the stack to Agenda
        for each stack $\sigma$ in Agenda
            call Progress($\sigma$)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

Example

```
m-fetch1($r$,$c$)    $r = r_0$, $c$ = 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
```

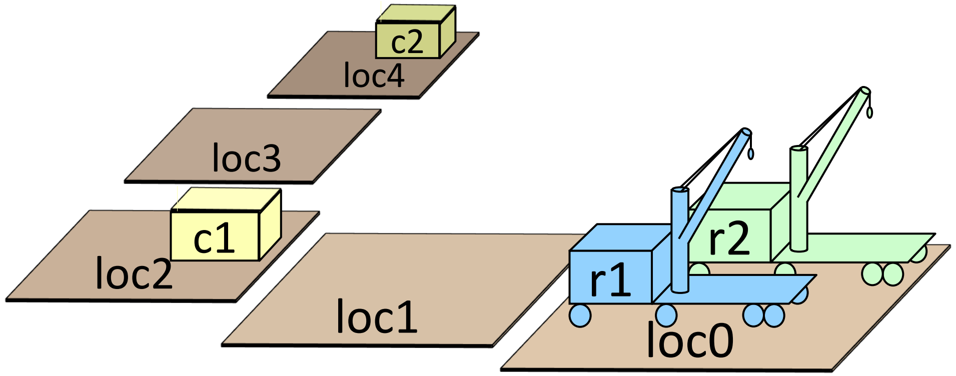
Refinement tree



Candidates
= {m-fetch1(r1,c2),
m-fetch1(r2,c2)}

```
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($c$))
```

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

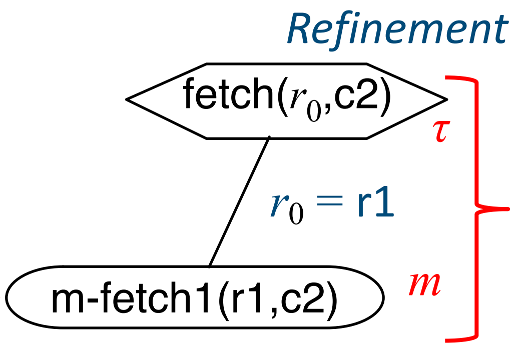


```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance $m$ for τ
            create a refinement stack for τ, $m$
            add the stack to $Agenda$
        for each stack $\sigma$ in $Agenda$
            call Progress($\sigma$)
       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

Example

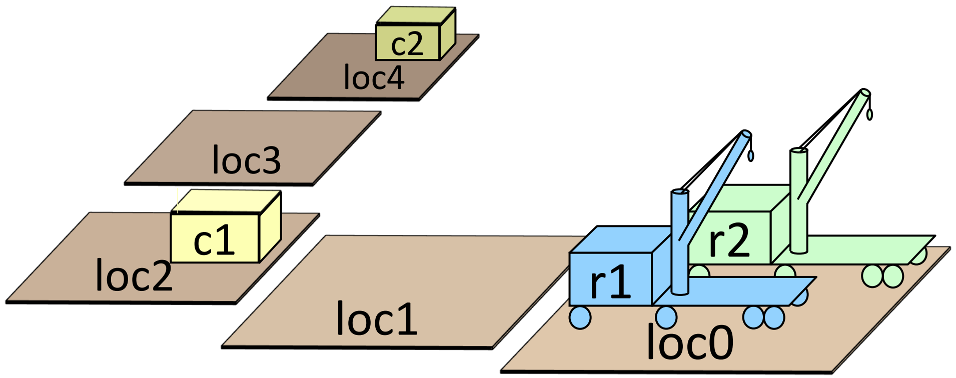
```
m-fetch1($r$,$c$)   $r$ = r1, $c$ = c2
    task:   fetch($r$,$c$)
    pre:    pos($c$) = unknown
    body:
        if $\exists l$ (view($l$) = F) then
            move-to($r$,$l$)
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                take($r$,$c$,$l$)
            else fetch($r$,$c$)
        else fail
```

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($c$))
```

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



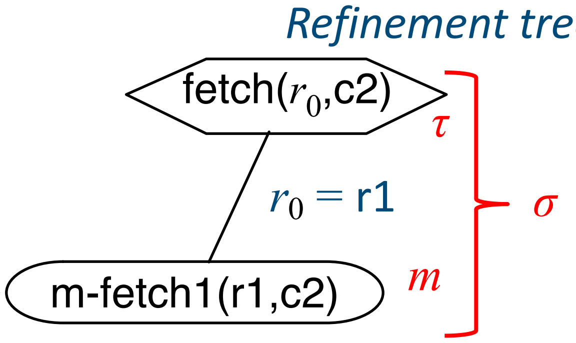
```
procedure RAE:
    loop:
        for every new external task or event $\tau$ do
            choose a method instance $m$ for $\tau$
            create a refinement stack for $\tau$, $m$
            add the stack to Agenda
        for each stack $\sigma$ in Agenda
            call Progress($\sigma$)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

Example

```
m-fetch1($r$,$c$)   $r$ = r1, $c$ = 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
```

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

Refinement tree



fetch(r_0,c2)

τ

r_0 = r1

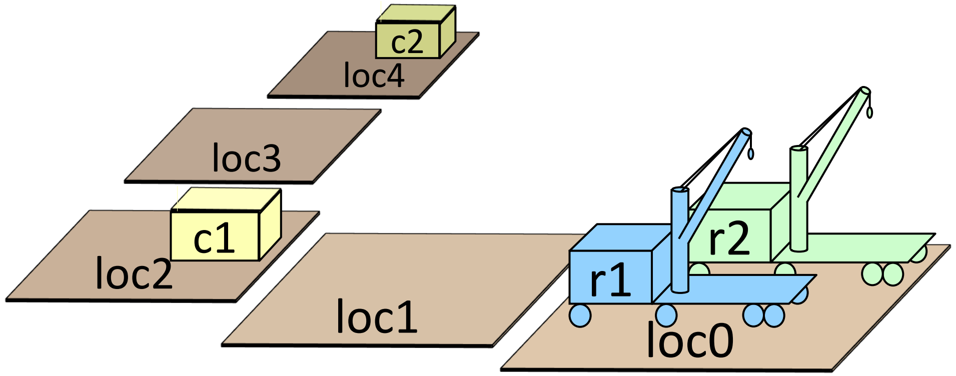
σ

m-fetch1(r1,c2)

m

```
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($c$))
```

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



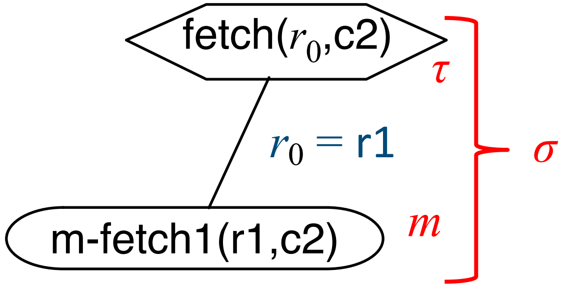
```
procedure RAE:
    loop:
        for every new external task or event $\tau$ do
            choose a method instance $m$ for $\tau$
            create a refinement stack for $\tau$, $m$
            add the stack to $Agenda$
        for each stack $\sigma$ in $Agenda$
            call Progress($\sigma$)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

Example

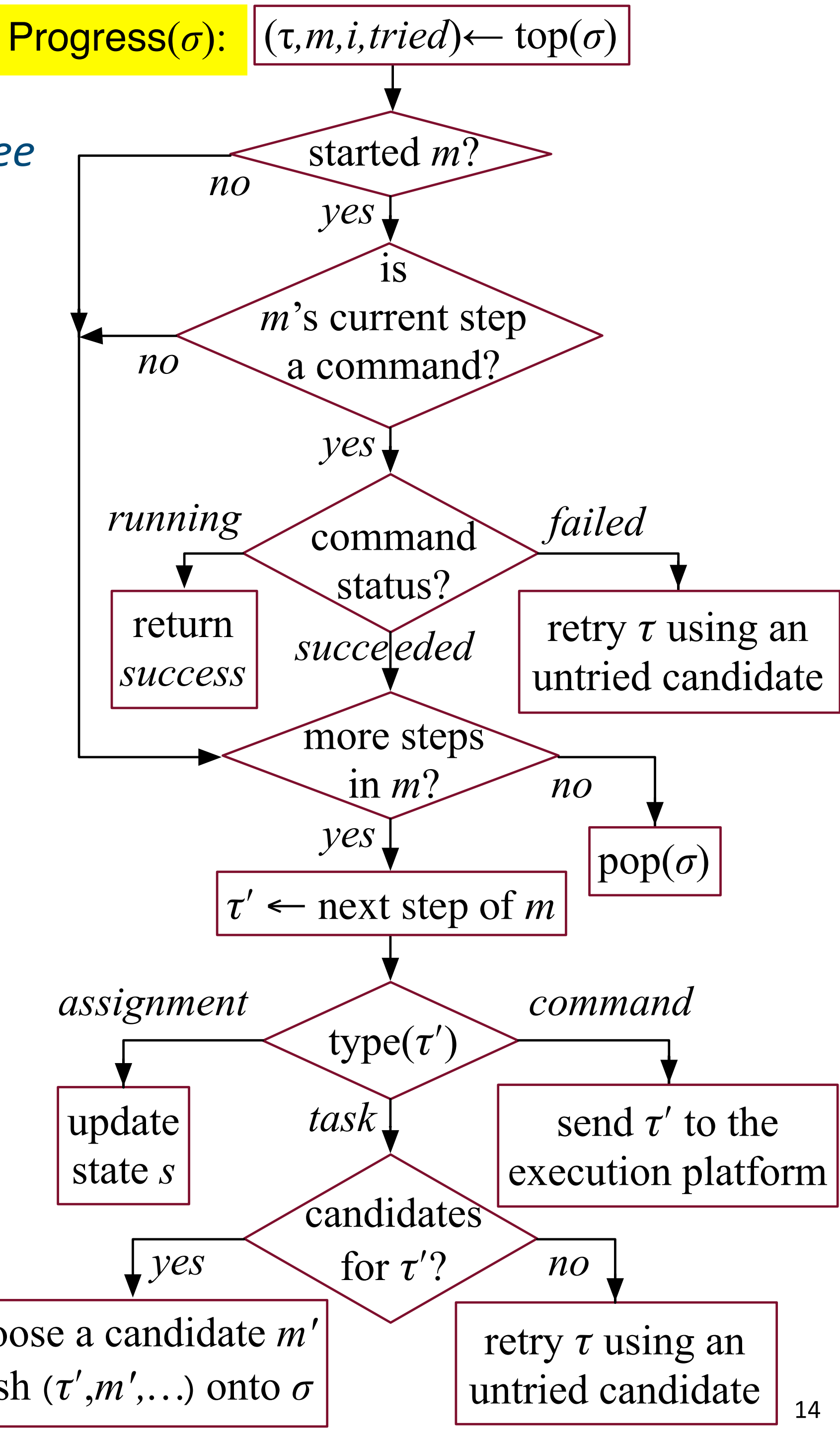
```
m-fetch1(r,c)        r = r1, c = c2
    task:   fetch(r,c)
    pre:    pos(c) = unknown
    body:
        if ∃l (view(l) = F) then
            move-to(r,l)
            perceive(r,l)
            if pos(c) = l then
                take(r,c,l)
            else fetch(r,c)
        else fail
```

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

Refinement tree



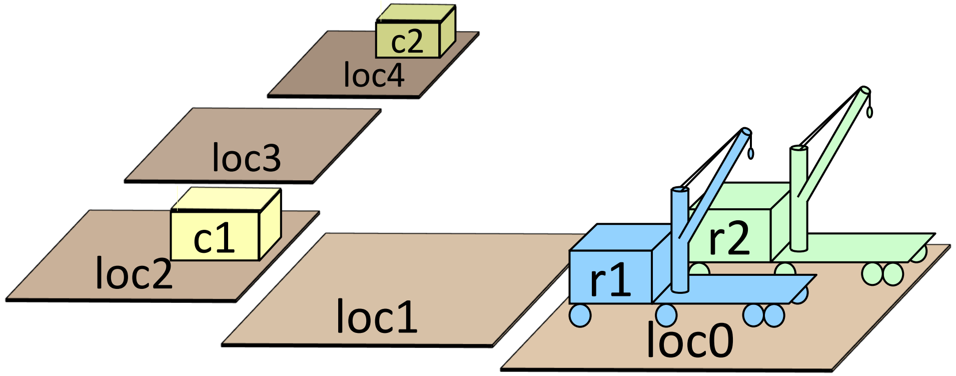
Progress(σ):



(τ,m,i,$tried$)← top(σ)
started m?
no
yes
is m's current step a command?
no
yes
running
command status?
failed
return success
succeeded
retry τ using an untried candidate
more steps in m?
no
yes
pop(σ)
τ' ← next step of m
assignment
type(τ')
command
update state s
task
send τ' to the execution platform
candidates for τ'?
yes
no
choose a candidate m' push (τ',m',…) onto σ
retry τ using an untried candidate

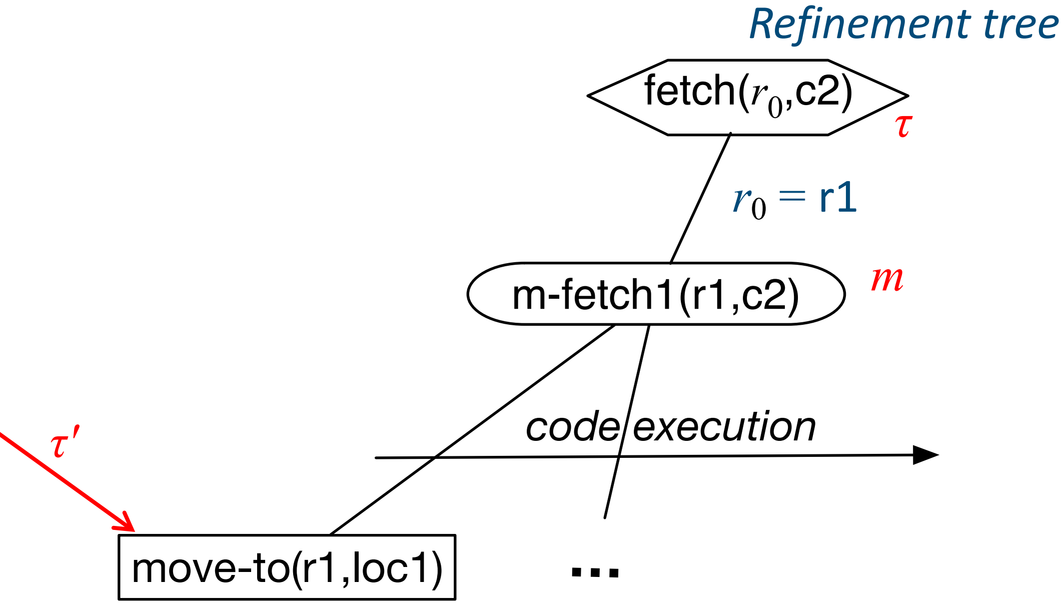
```
m-fetch2(r,c)
    task:   fetch(r,c)
    pre:    pos(c) ≠ unknown
    body:
        if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

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



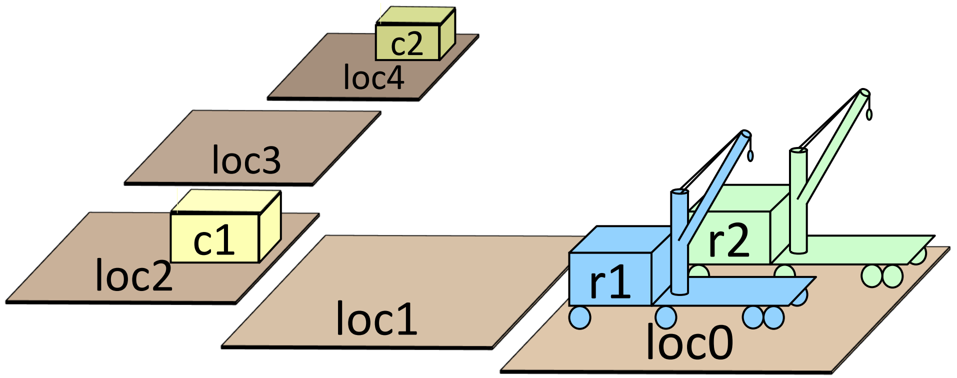
Example

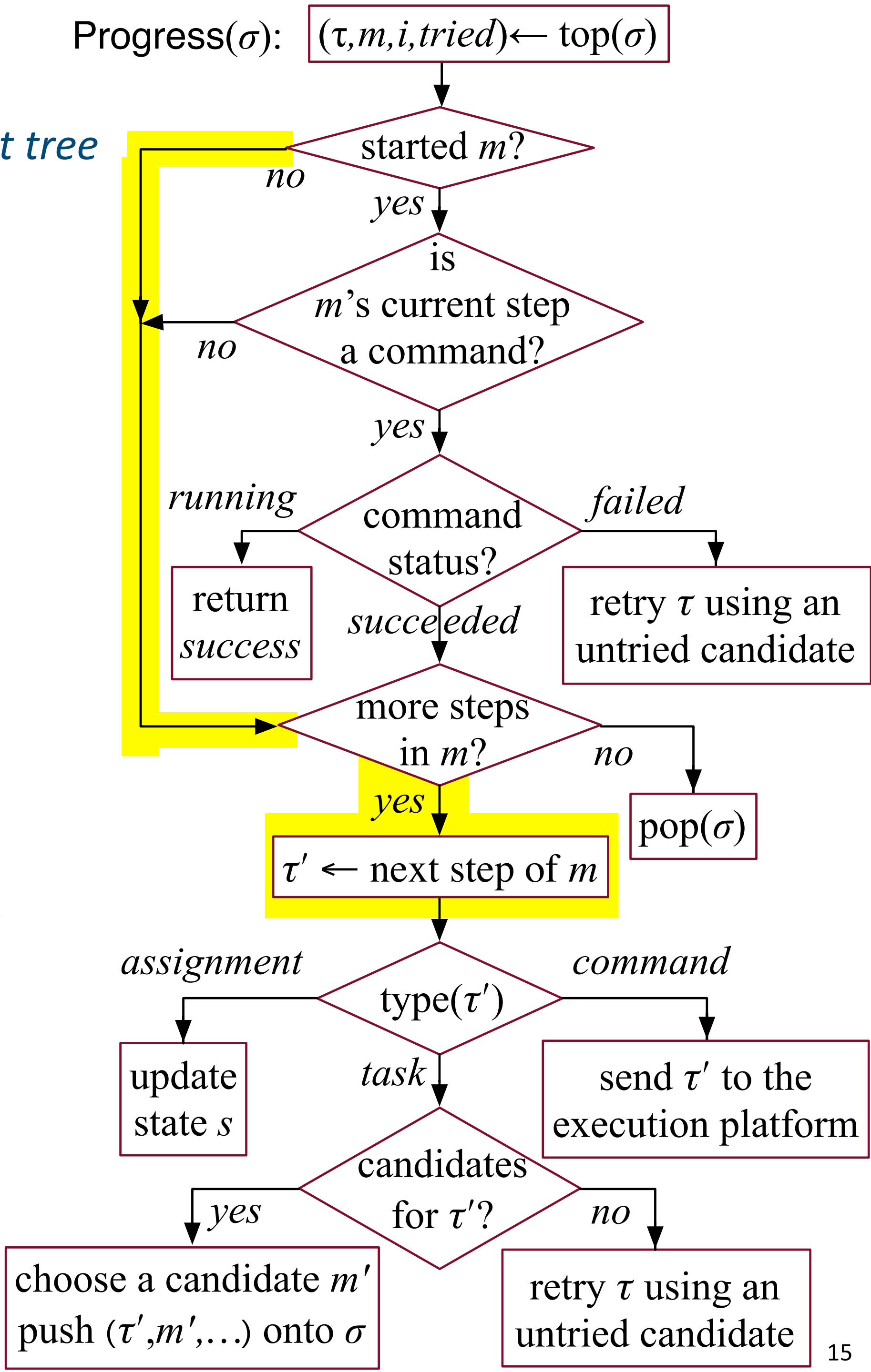
```
m-fetch1($r$,$c$)   $r$ = r1, $c$ = c2
    task:   fetch($r$,$c$)
    pre:    pos($c$) = unknown
    body:                      $l$ = loc1
        if $\exists l$ (view($l$) = F) then
            move-to($r$,$l$)
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                take($r$,$c$,$l$)
            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($c$))
```

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



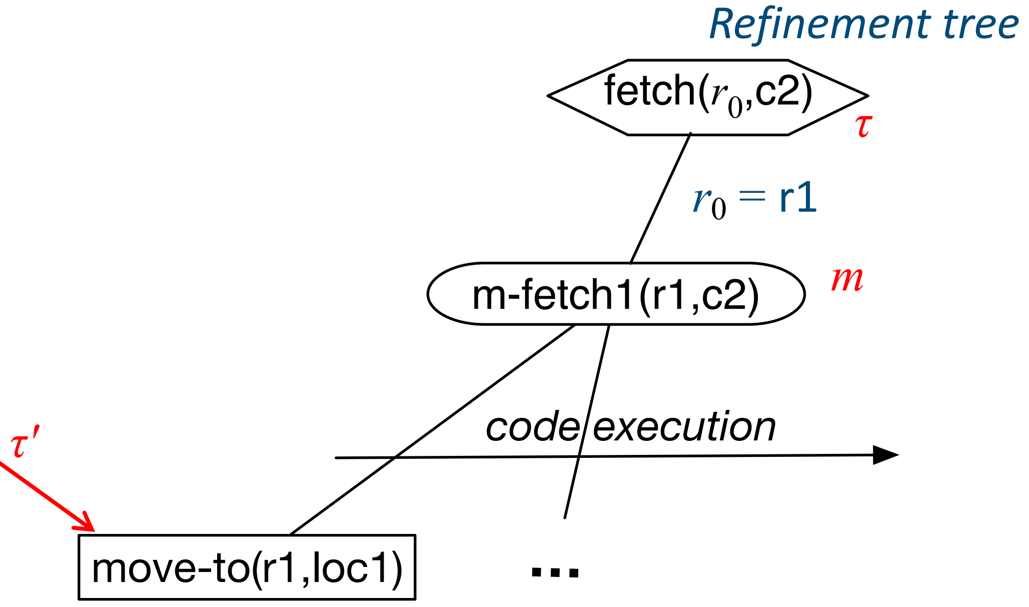


Progress(σ):
(τ,m,i,$tried$)← top(σ)
started m?
no
yes
is m's current step a command?
no
yes
command status?
running
failed
return success
succeeded
retry τ using an untried candidate
more steps in m?
no
yes
pop(σ)
τ' ← next step of m
type(τ')
assignment
command
task
update state s
send τ' to the execution platform
candidates for τ'?
yes
no
choose a candidate m' push (τ',m',…) onto σ
retry τ using an untried candidate

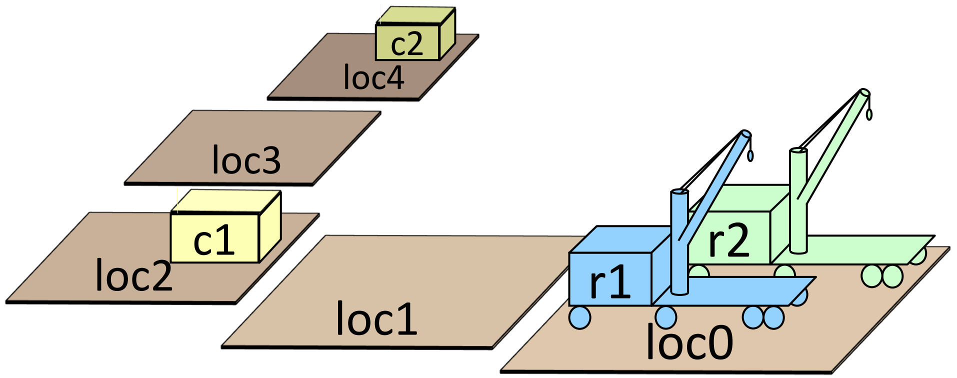
Example

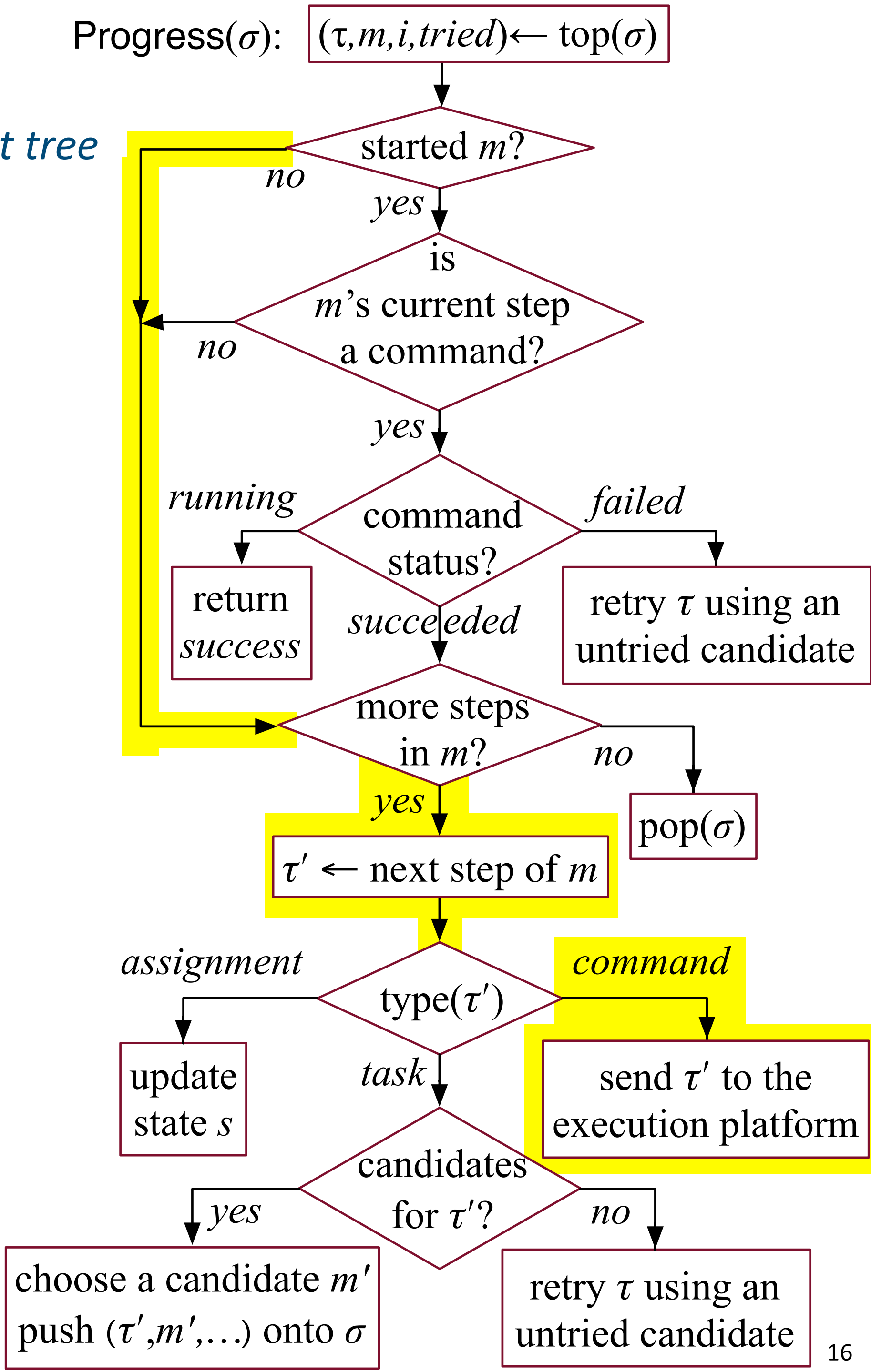
```
m-fetch1($r$,$c$)        $r$ = r1, $c$ = c2
    task:   fetch($r$,$c$)
    pre:    pos($c$) = unknown
    body:                        $l$ = loc1
        if $\exists l$ (view($l$) = F) then
            move-to($r$,$l$)
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                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

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




Progress(σ): (τ,m,i,$tried$) ← top(σ)

started m? — no / yes

is m's current step a command? — no / yes

command status? — running: return *success*; failed: retry τ using an untried candidate; succeeded

more steps in m? — no: pop(σ); yes

τ' ← next step of m

type(τ') — assignment: update state s; command: send τ' to the execution platform; task

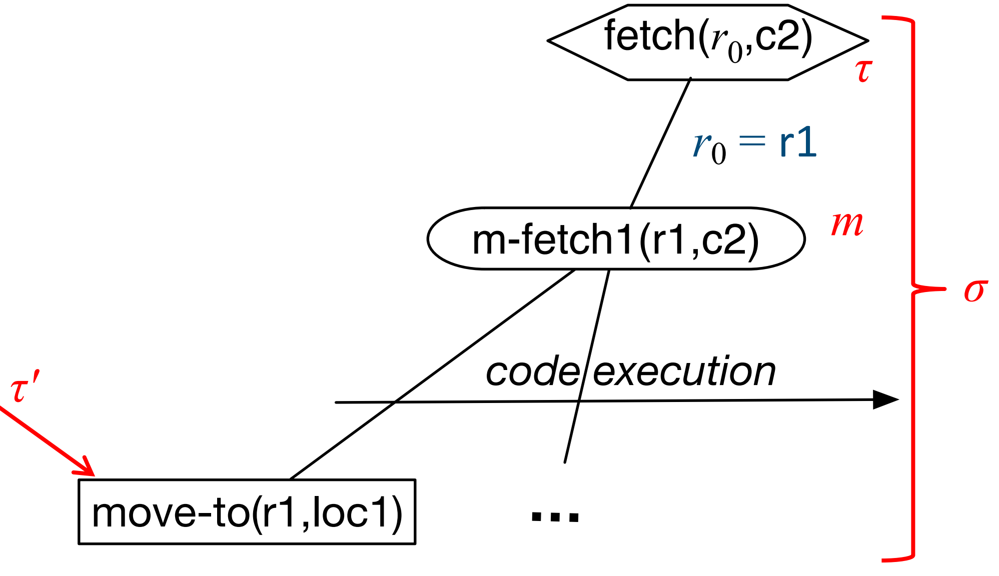
candidates for τ'? — yes: choose a candidate m' push (τ',m',…) onto σ; no: retry τ using an untried candidate

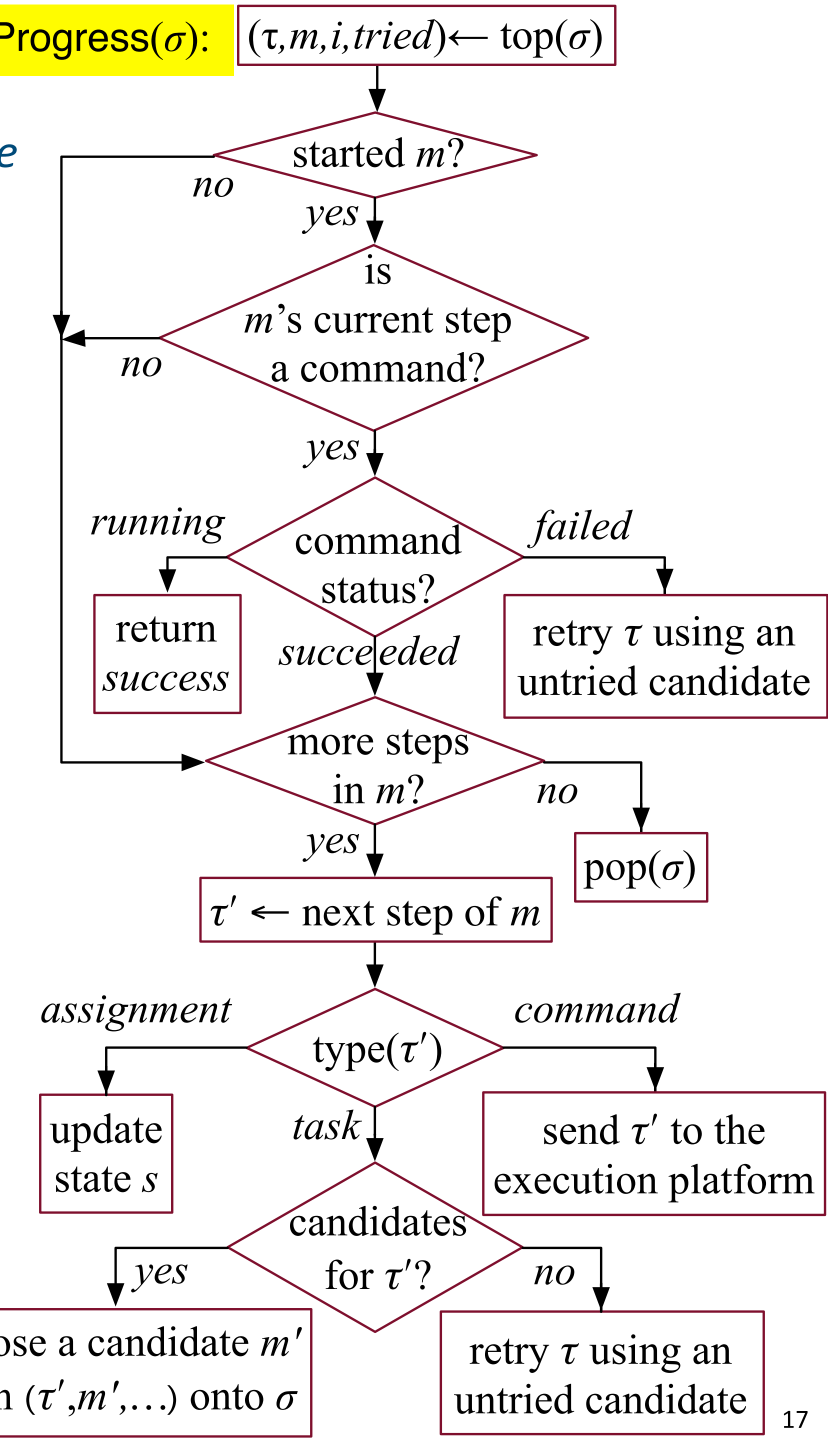
Example

```
m-fetch1(r,c)        r = r1, c = c2
    task:   fetch(r,c)
    pre:    pos(c) = unknown
    body:                    l = loc1
        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
```

τ′

Refinement tree





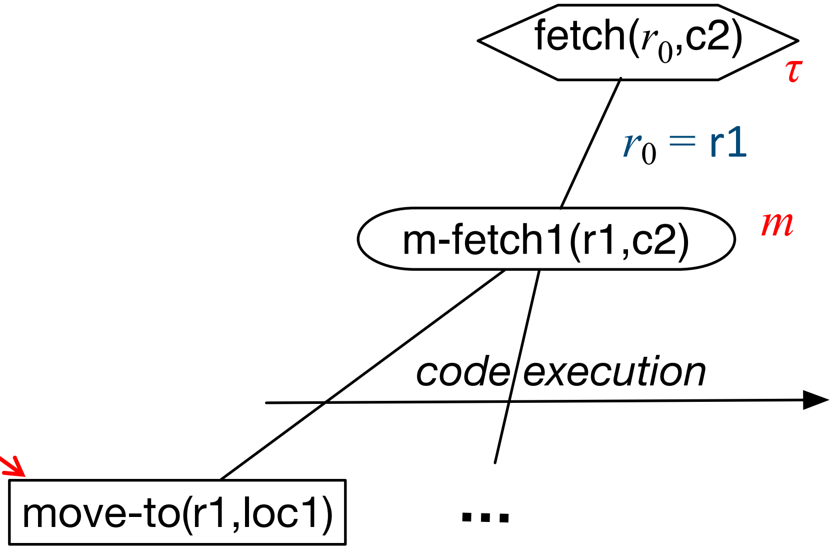
```
m-fetch2(r,c)
    task:   fetch(r,c)
    pre:    pos(c) ≠ unknown
    body:
        if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

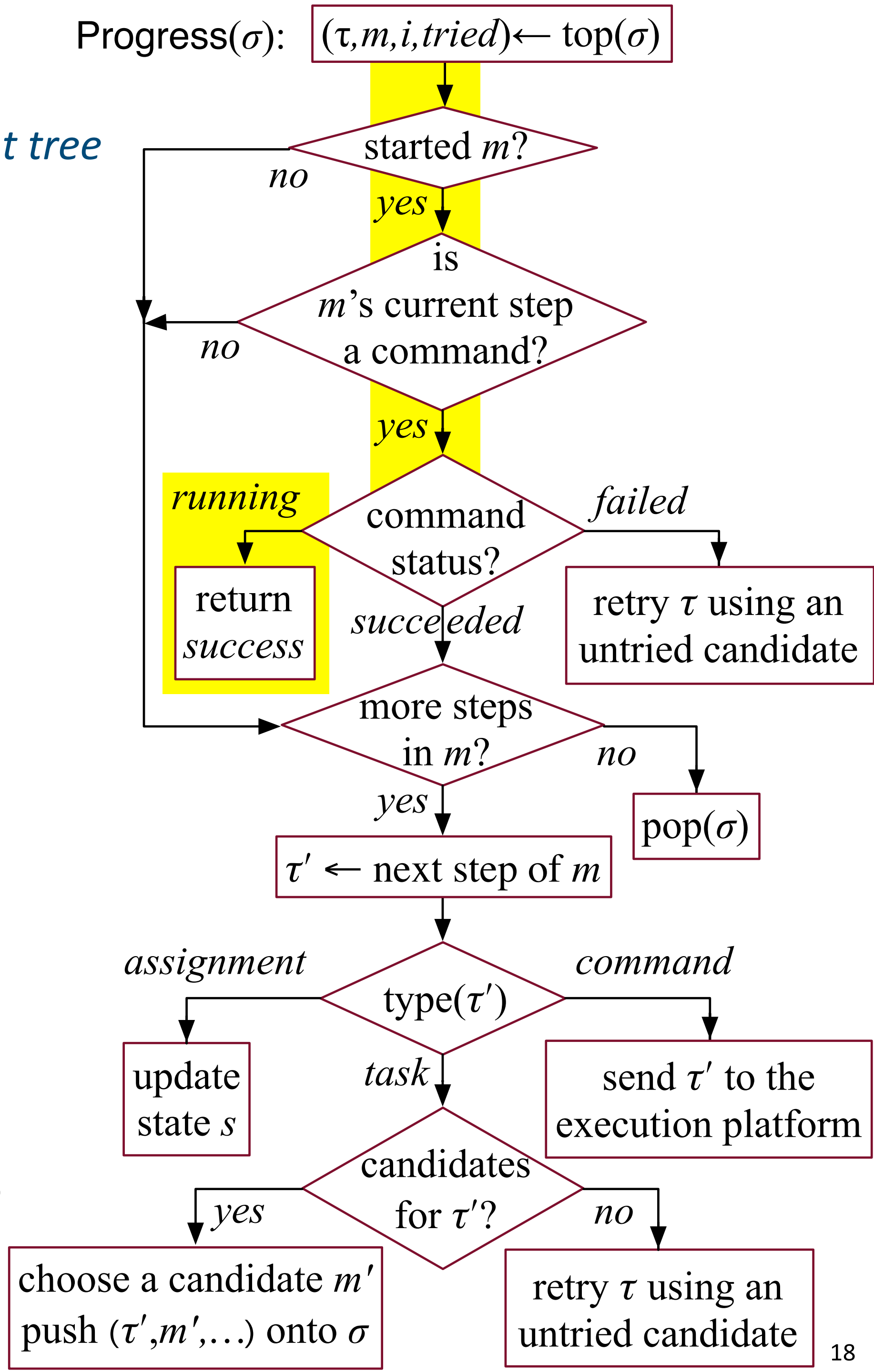
```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to Agenda
        for each stack σ in Agenda
            call Progress(σ)
            if σ is finished then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

Example

```
m-fetch1($r$,$c$)        $r$ = r1, $c$ = c2
    task:   fetch($r$,$c$)
    pre:    pos($c$) = unknown
    body:                    $l$ = loc1
        if ∃$l$ (view($l$) = F) then
            move-to($r$,$l$)   ← running …
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                    take($r$,$c$,$l$)
            else fetch($r$,$c$)
        else fail
```

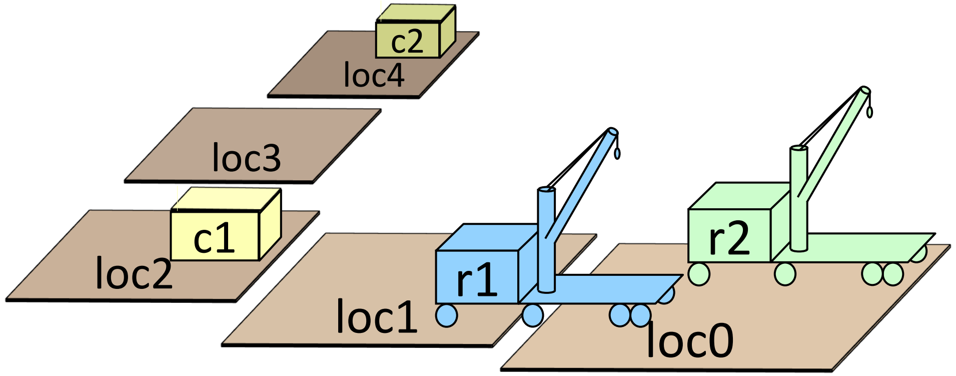
Refinement tree





Progress(σ):
(τ,m,i,$tried$)← top(σ)
started m?
no
yes
is m's current step a command?
no
yes
command status?
running
return success
failed
retry τ using an untried candidate
succeeded
more steps in m?
no
pop(σ)
yes
τ' ← next step of m
type(τ')
assignment
update state s
command
send τ' to the execution platform
task
candidates for τ'?
yes
choose a candidate m' push (τ',m',…) onto σ
no
retry τ using an untried candidate

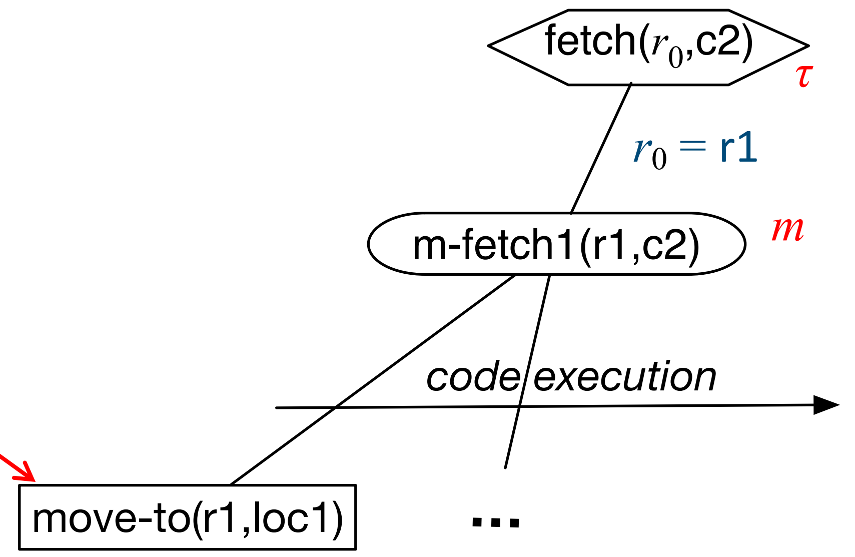
```
m-fetch2($r$,$c$)
    task:   fetch($r$,$c$)
    pre:    pos($c$) ≠ unknown
    body:
        if loc($r$) = pos($c$) then
            take($r$,$c$,pos($c$))
        else do
            move-to($r$,pos($c$))
            take($r$,$c$,pos($c$))
```



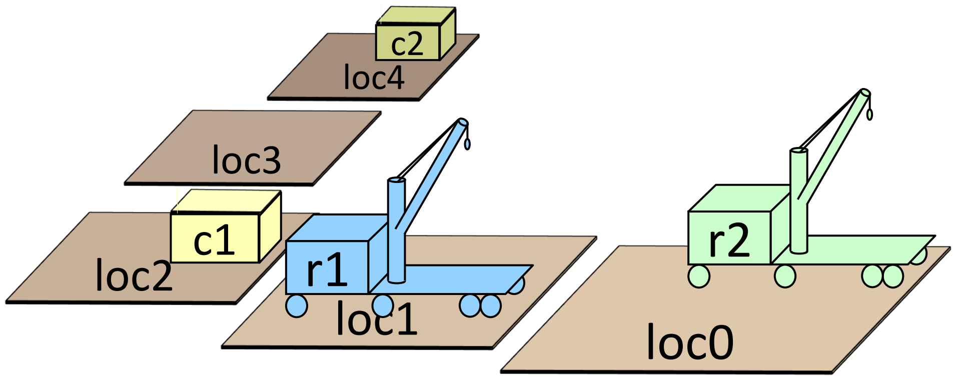
Example

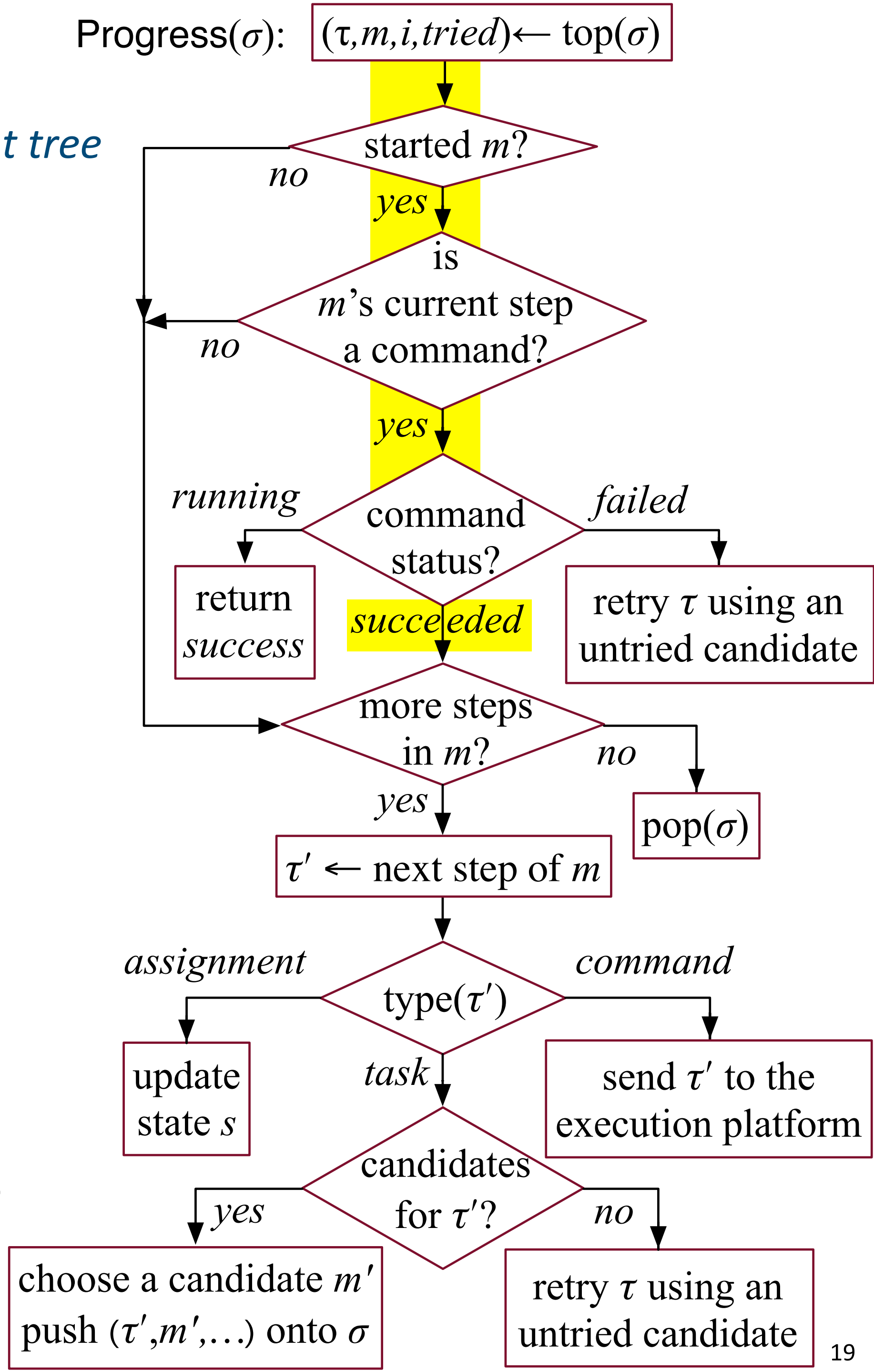
```
m-fetch1($r$,$c$)    $r$ = r1, $c$ = c2
    task:  fetch($r$,$c$)
    pre:   pos($c$) = unknown
    body:                    $l$ = loc1
        if $\exists l$ (view($l$) = F) then
            move-to($r$,$l$)   ← succeeded
            perceive($r$,$l$)
            if pos($c$) = $l$ then
                take($r$,$c$,$l$)
            else fetch($r$,$c$)
        else fail
```

Refinement tree



```
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($c$))
```



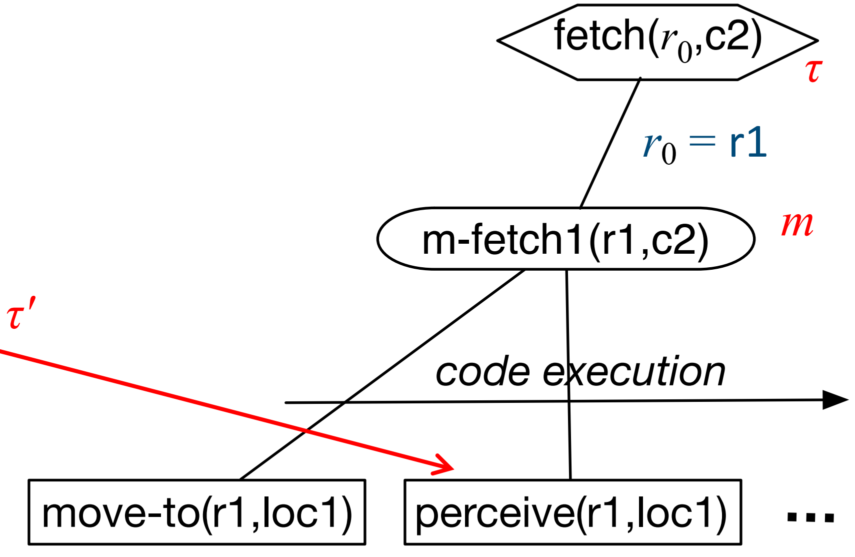


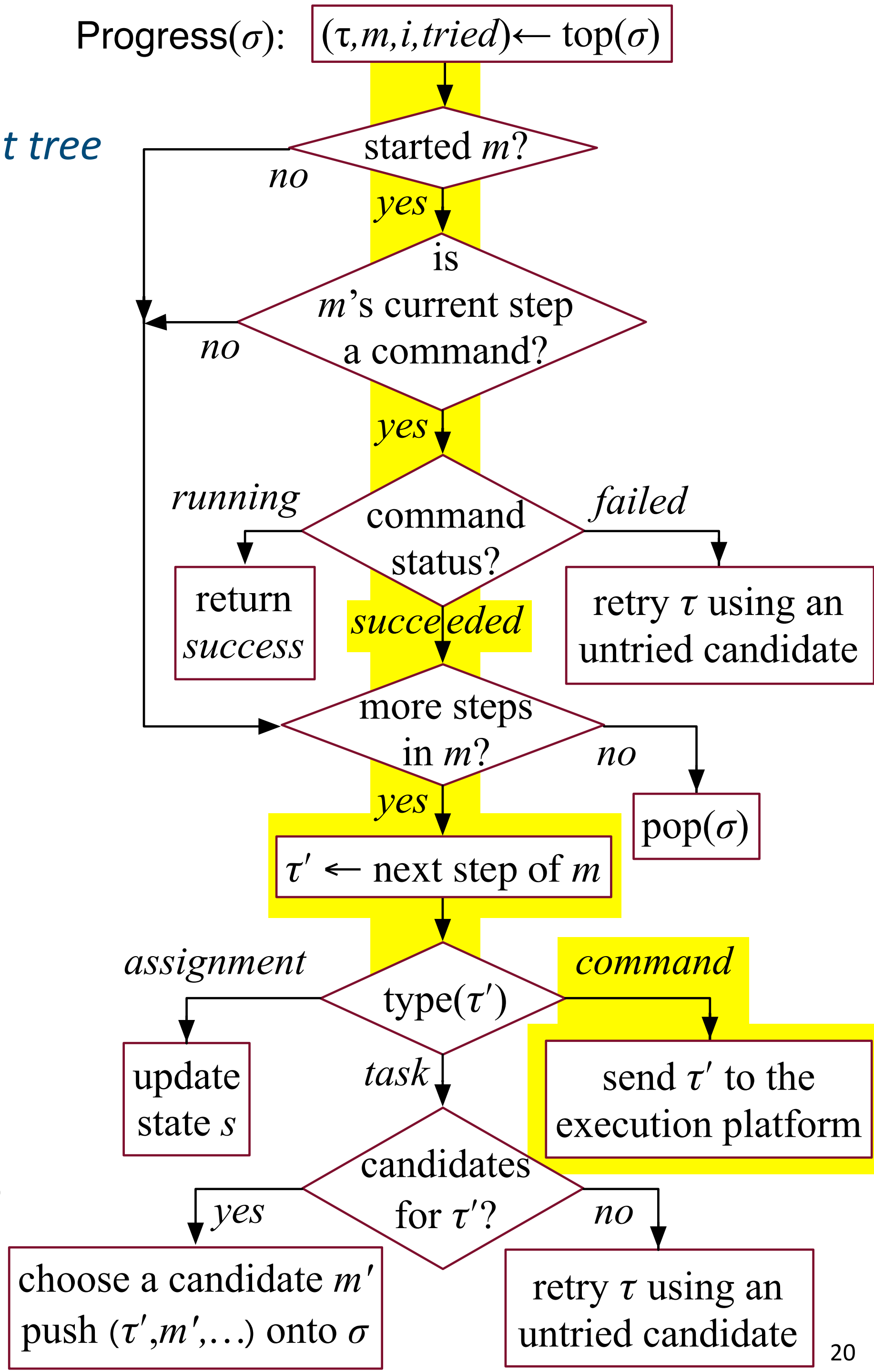
Progress(σ):
(τ,m,i,$tried$)← top(σ)
started m?
no
yes
is m's current step a command?
no
yes
running
command status?
failed
return success
succeeded
retry τ using an untried candidate
more steps in m?
no
yes
pop(σ)
τ' ← next step of m
assignment
type(τ')
command
update state s
task
send τ' to the execution platform
candidates for τ'?
yes
no
choose a candidate m' push (τ',m',…) onto σ
retry τ using an untried candidate

Example

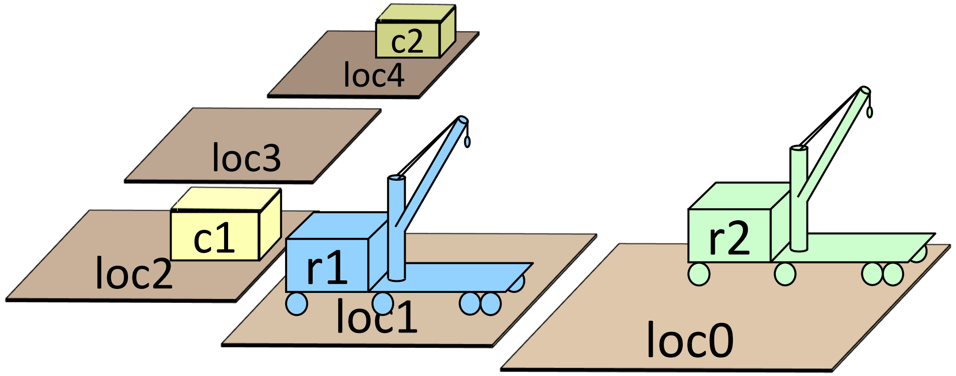
```
m-fetch1(r,c)   r = r1, c = c2
    task:   fetch(r,c)
    pre:    pos(c) = unknown
    body:                 l = loc1
      if ∃l (view(l) = F) then
          move-to(r,l)
          perceive(r,l)
          if pos(c) = l then
              take(r,c,l)
          else fetch(r,c)
      else fail
```

Refinement tree



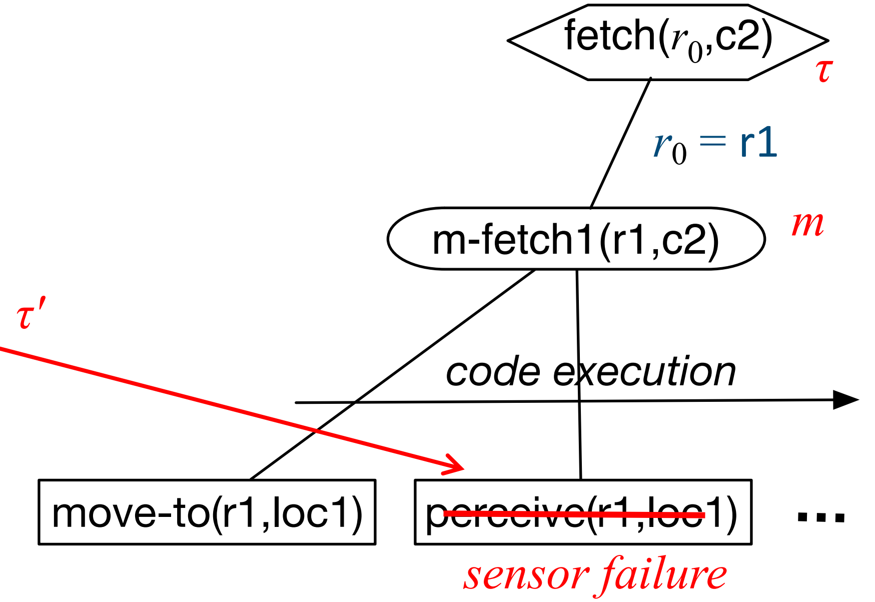

Progress(σ):
(τ,m,i,$tried$)← top(σ)
started m?
no
yes
is m's current step a command?
no
yes
running
command status?
failed
return success
succeeded
retry τ using an untried candidate
more steps in m?
no
yes
pop(σ)
τ' ← next step of m
assignment
type(τ')
command
update state s
task
send τ' to the execution platform
candidates for τ'?
yes
no
choose a candidate m' push (τ',m',…) onto σ
retry τ using an untried candidate

```
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))
```

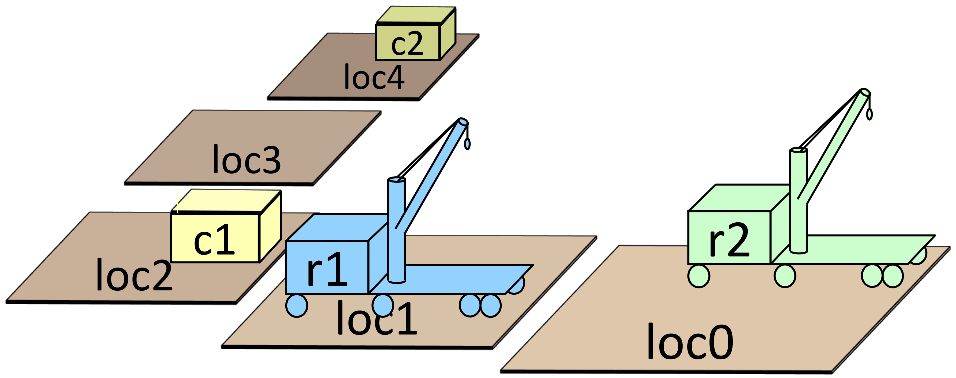


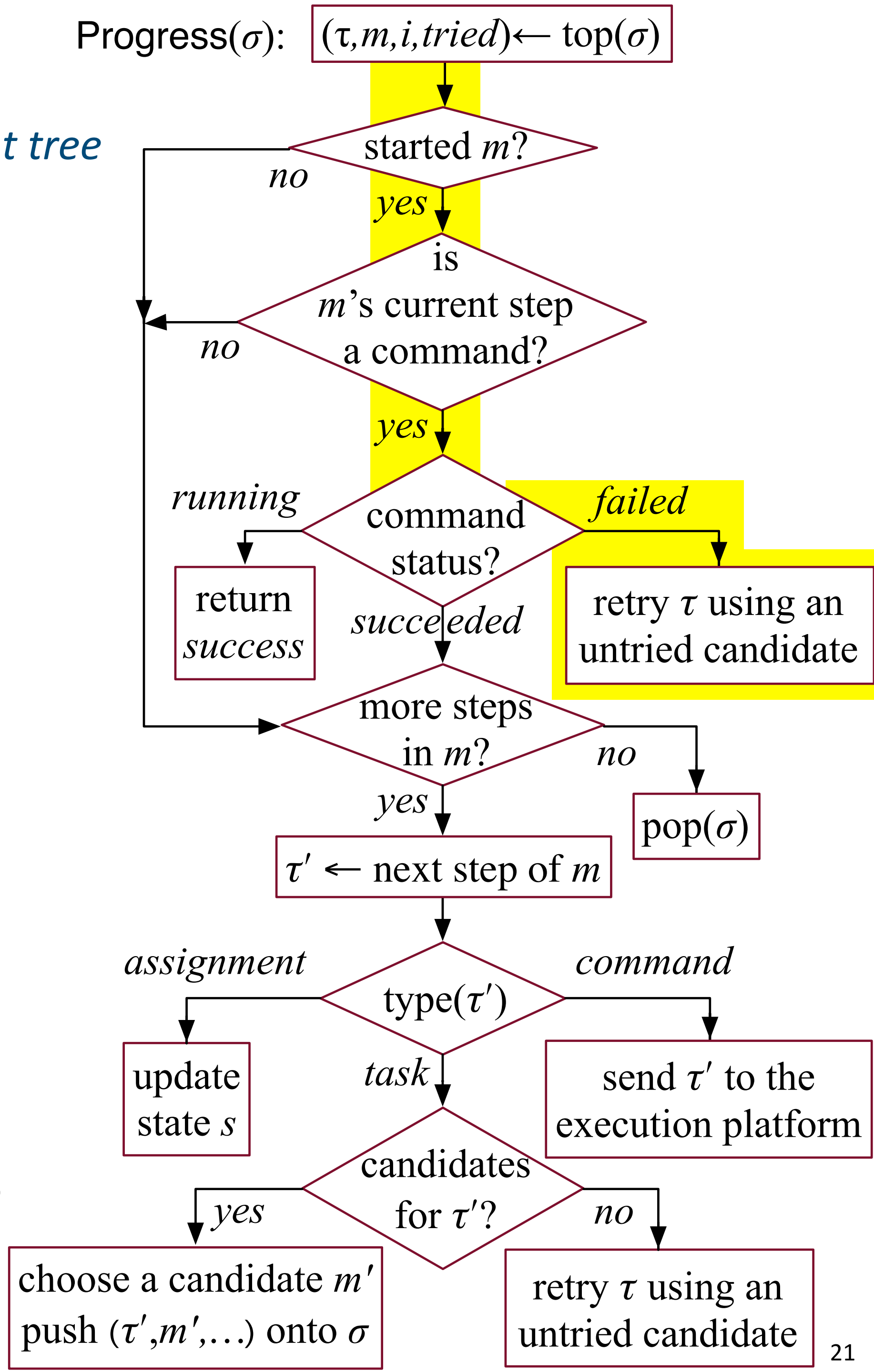
Example

```
m-fetch1(r,c)    r = r1, c = c2
    task:   fetch(r,c)
    pre:    pos(c) = unknown
    body:                 l = loc1
        if ∃l (view(l) = F) then
            move-to(r,l)
            perceive(r,l) ← failed
            if pos(c) = l then
                    take(r,c,l)
            else fetch(r,c)
        else fail
```



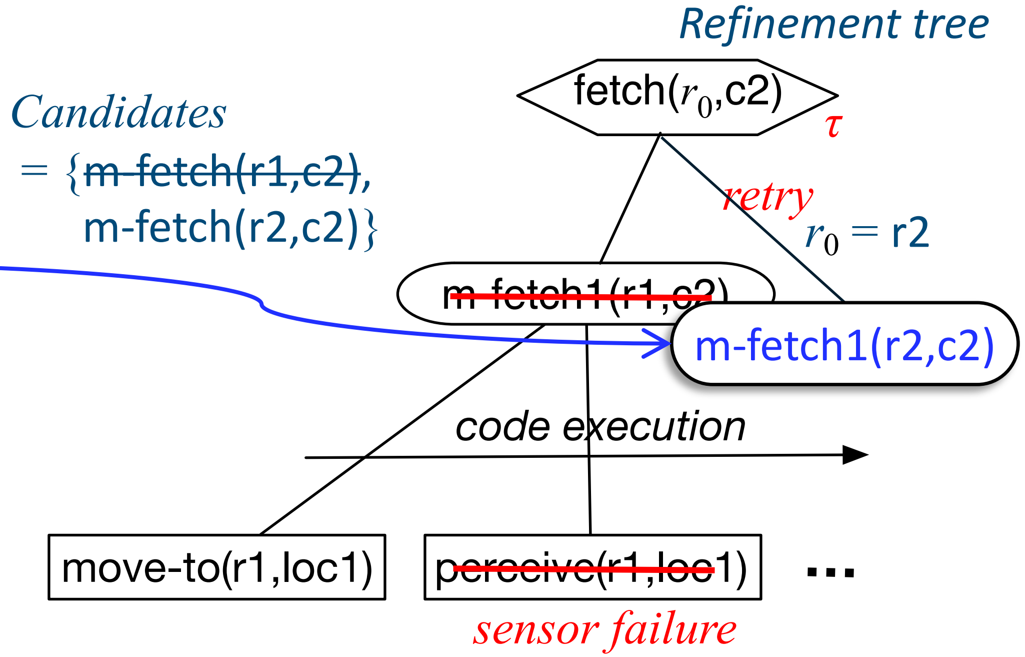
```
m-fetch2(r,c)
    task:   fetch(r,c)
    pre:    pos(c) ≠ unknown
    body:
        if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```





Example

```
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
```



Refinement tree

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

fetch(r_0,c2)

τ

retry

r_0 = r2

~~m-fetch1(r1,c2)~~

m-fetch1(r2,c2)

code execution

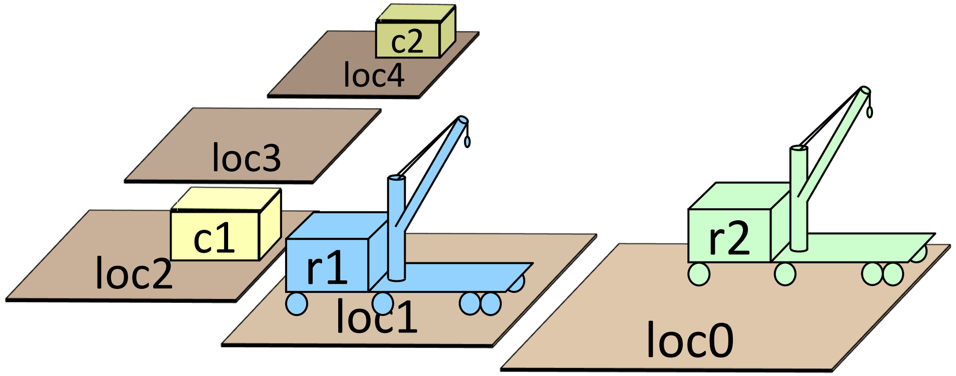
move-to(r1,loc1)

~~perceive(r1,loc1)~~

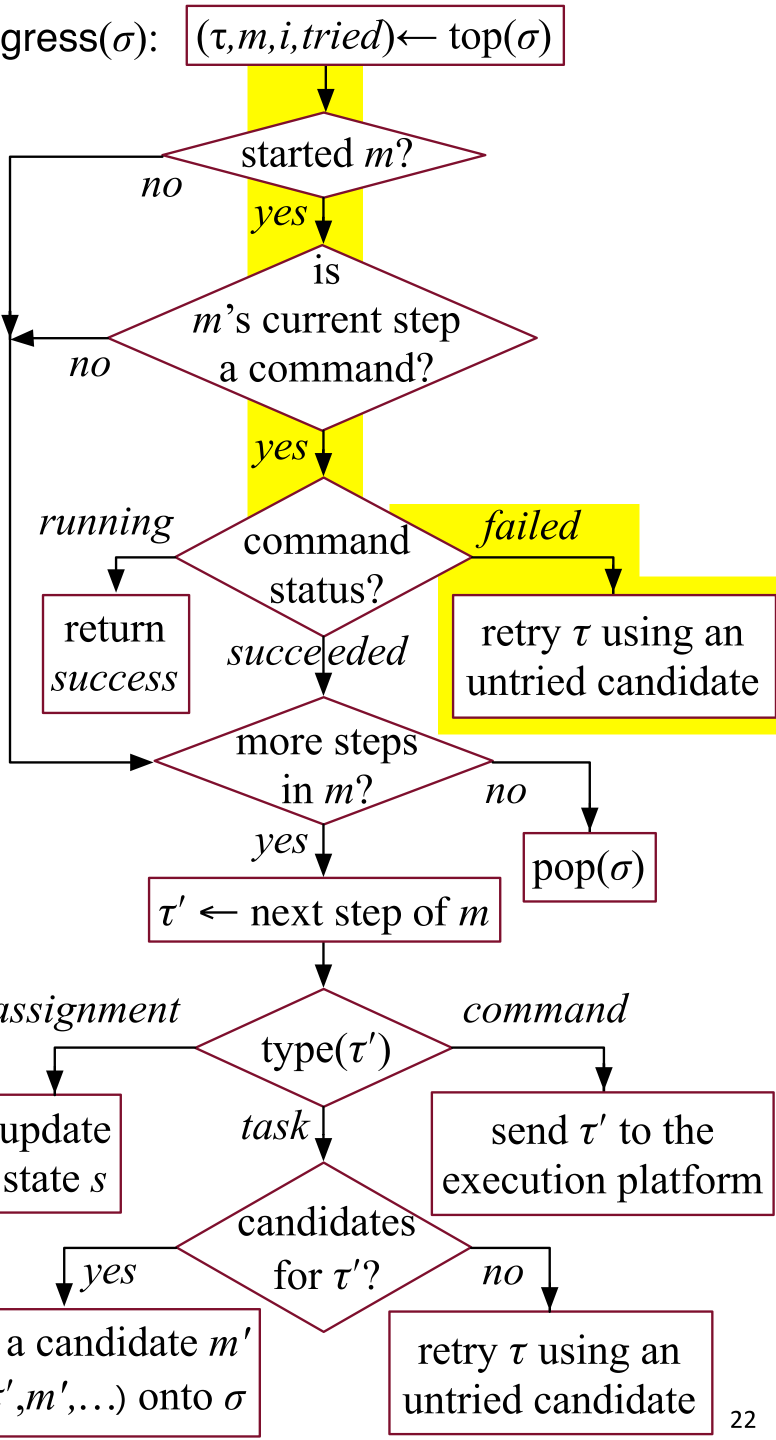
...

sensor failure

```
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($c$))
```



Progress(σ):



(τ,m,i,$tried$) ← top(σ)

started m?

no

yes

is m's current step a command?

no

yes

command status?

running

return success

failed

retry τ using an untried candidate

succeeded

more steps in m?

no

pop(σ)

yes

τ' ← next step of m

type(τ')

assignment

update state s

command

send τ' to the execution platform

task

candidates for τ'?

yes

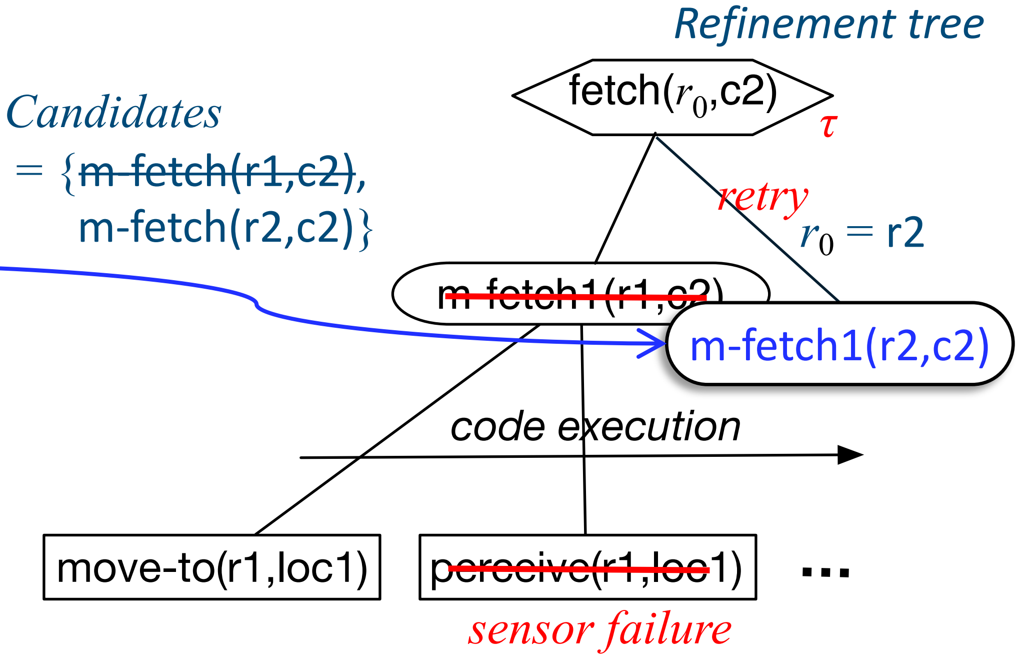
choose a candidate m' push (τ',m',...) onto σ

no

retry τ using an untried candidate

Example

```
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
```


Refinement tree

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

fetch(r_0,c2) τ

retry r_0 = r2

~~m-fetch1(r1,c2)~~

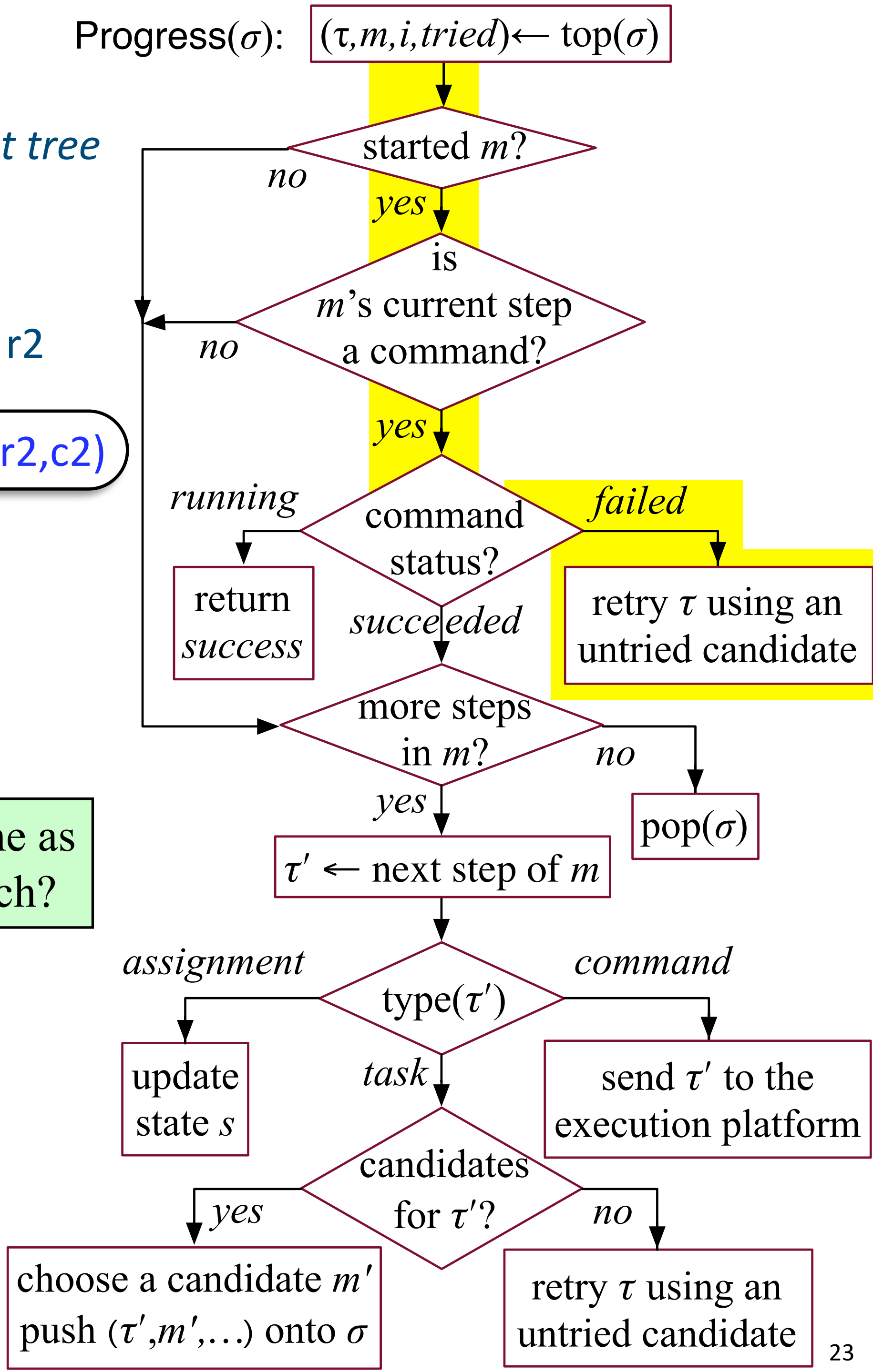
m-fetch1(r2,c2)

code execution

move-to(r1,loc1)

~~perceive(r1,loc1)~~ …

sensor failure


Progress(σ): (τ,m,i,$tried$)← top(σ)

started m? *no* / *yes*

is m's current step a command? *no* / *yes*

command status? *running* → return *success*; *failed* → retry τ using an untried candidate; *succeeded*

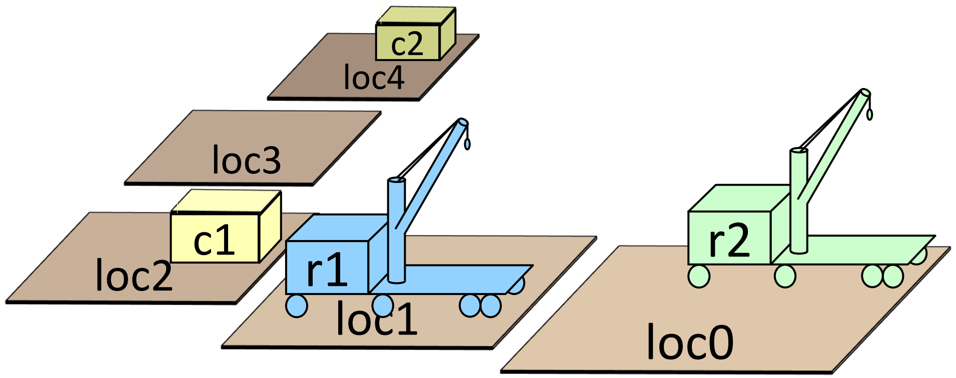
more steps in m? *no* → pop(σ); *yes* → τ' ← next step of m

type(τ'): *assignment* → update state s; *command* → send τ' to the execution platform; *task* →

candidates for τ'? *yes* → choose a candidate m' push (τ',m',…) onto σ; *no* → retry τ using an untried candidate

```
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($c$))
```

Poll: Is this the same as a backtracking search?

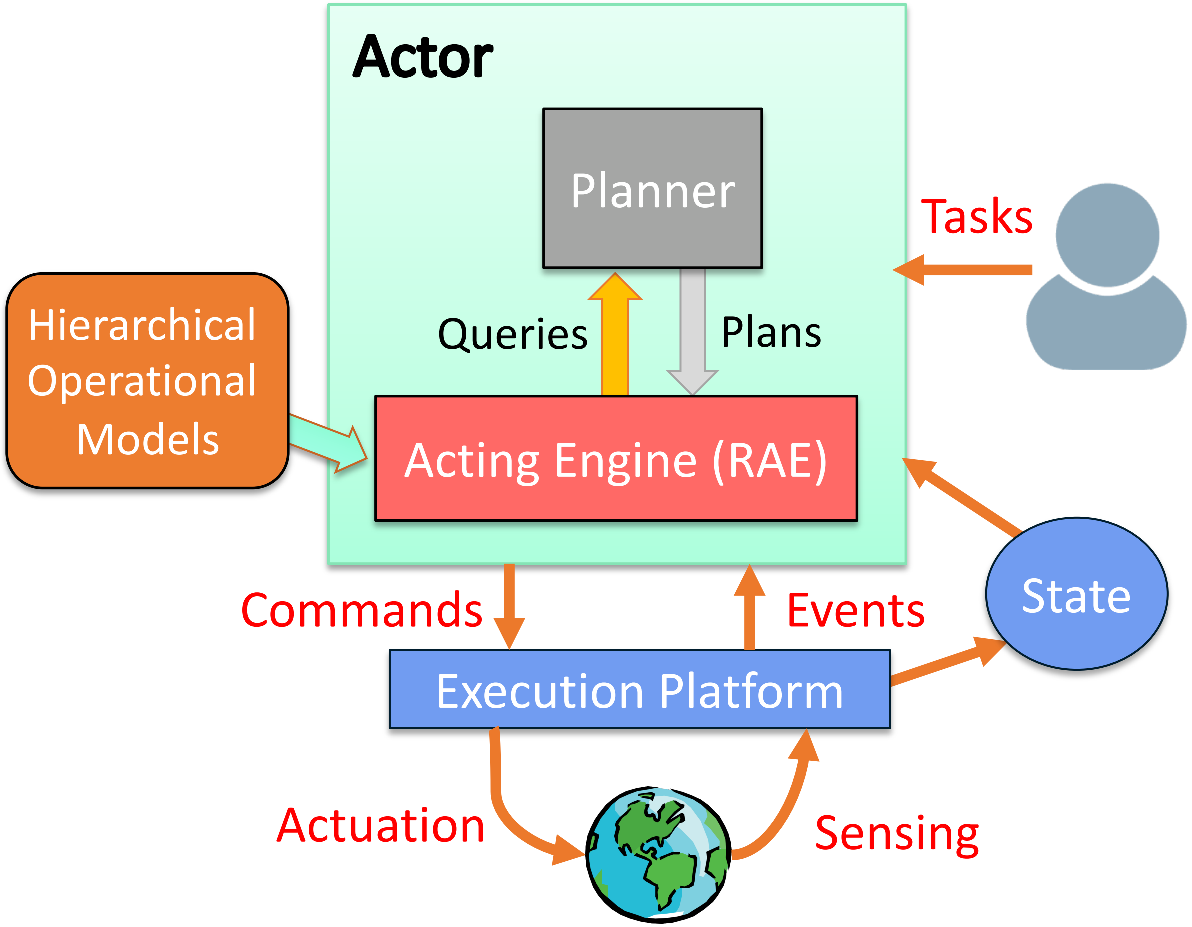


Extensions to RAE

- 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

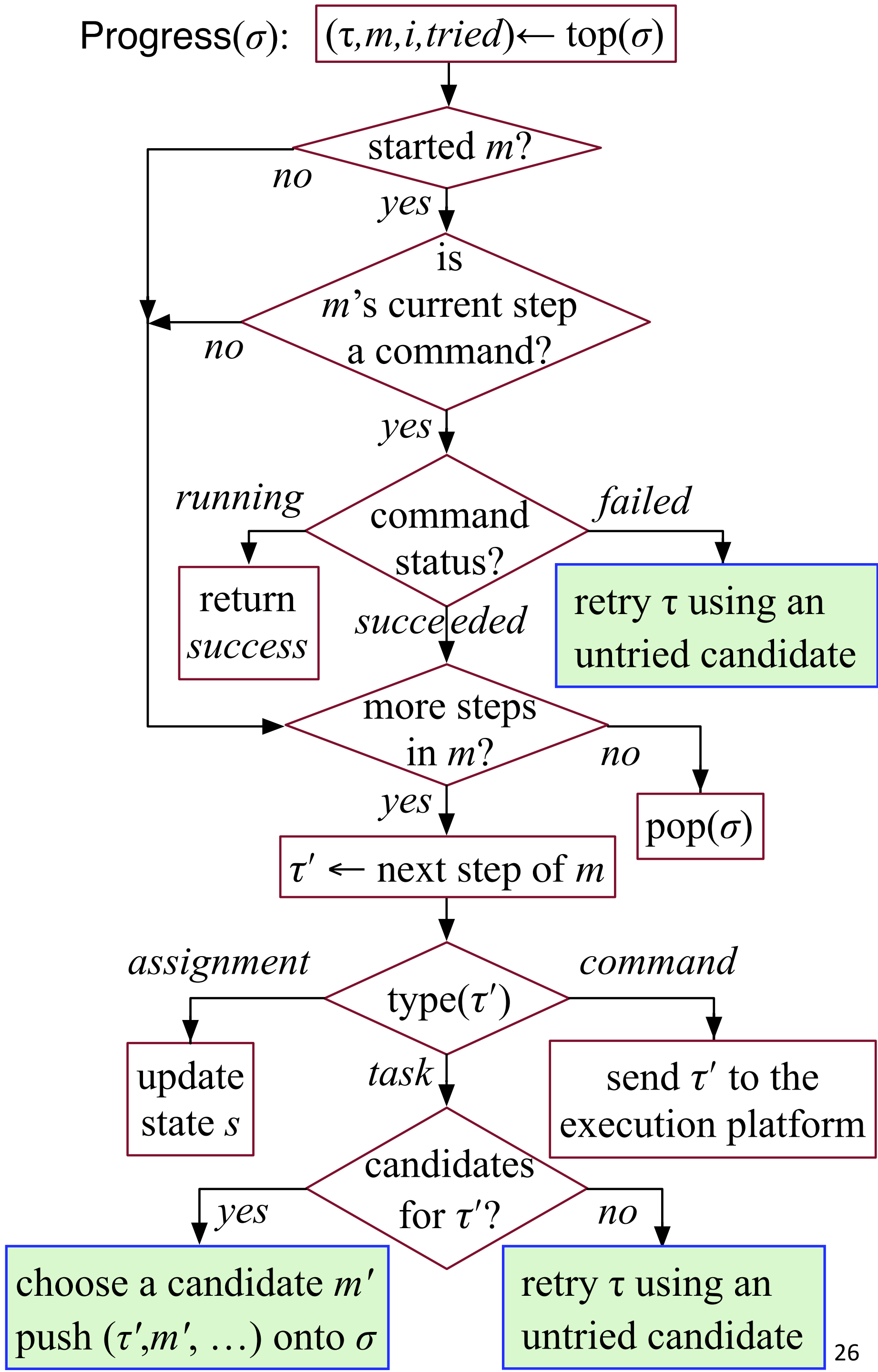
1. Motivation
2. Representation
3. Acting (Rae)
4. *Planning for Rae*
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application



Planning for Rae?

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

- Four places where Rae and Progress choose a method instance for a task
- 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(σ): $(\tau, m, i, tried) \leftarrow \mathrm{top}(\sigma)$

started m?

no

yes

is m's current step a command?

no

yes

running

command status?

failed

return *success*

succeeded

retry τ using an untried candidate

more steps in m?

no

yes

pop(σ)

$\tau' \leftarrow$ next step of m

assignment

type(τ')

command

update state s

task

send τ' to the execution platform

candidates for τ'?

yes

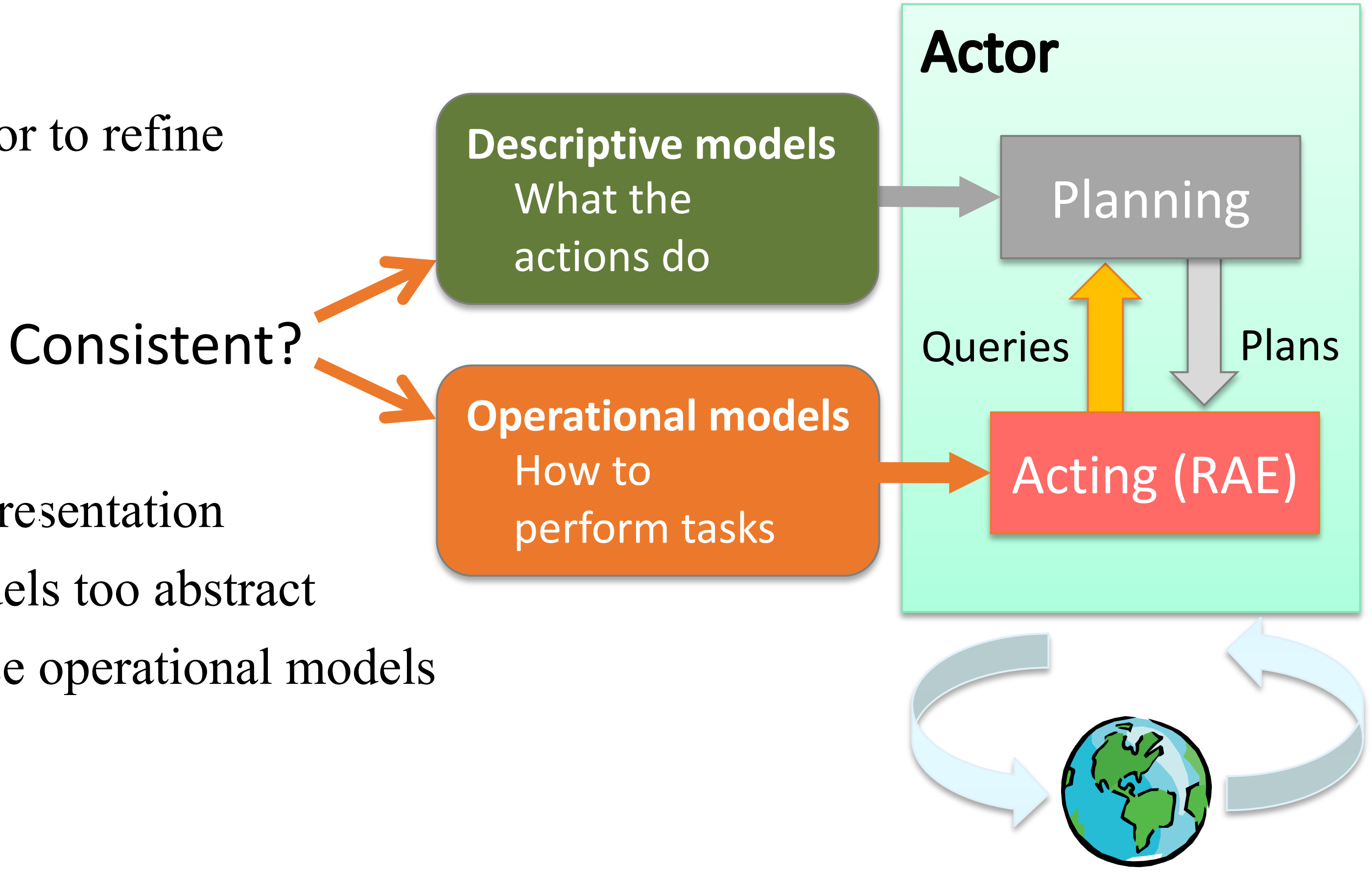
no

choose a candidate m' push $(\tau', m', \ldots)$ onto σ

retry τ using an untried candidate

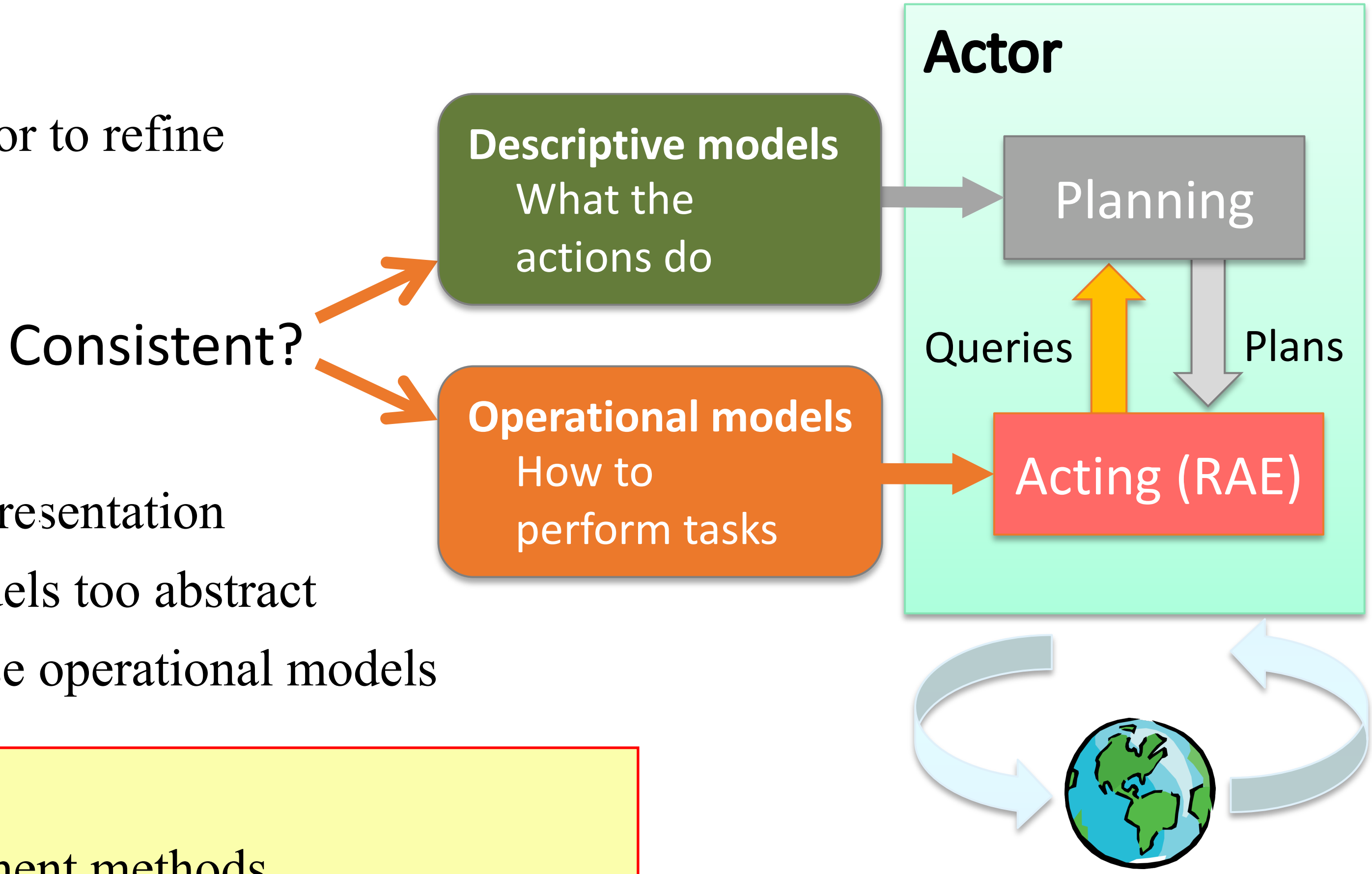
Planning and Acting Integration

- Planner's action models are abstractions
  - The planned actions are tasks for the actor to refine
- Consistency problem:
  - How to get action models that describe what the actor will do?
- One possible solution:
  - Actor and planner both use the same representation
    - Must be operational; descriptive models too abstract
    - Need planning algorithms that can use operational models



Planning and Acting Integration

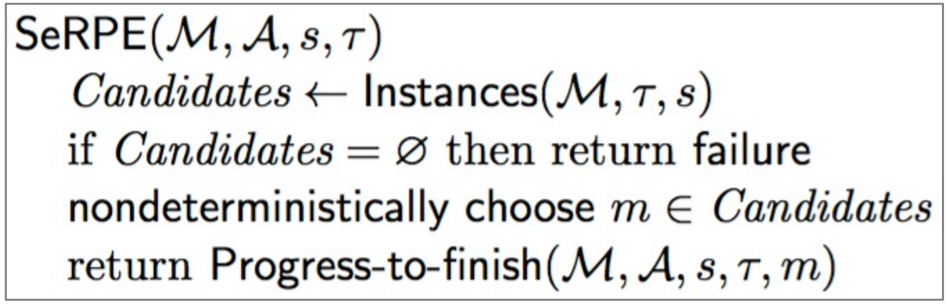
- Planner’s action models are abstractions
  - The planned actions are tasks for the actor to refine
- Consistency problem:
  - How to get action models that describe what the actor will do?
- One possible solution:
  - Actor and planner both use the same representation
    - Must be operational; descriptive models too abstract
    - Need planning algorithms that can use operational models



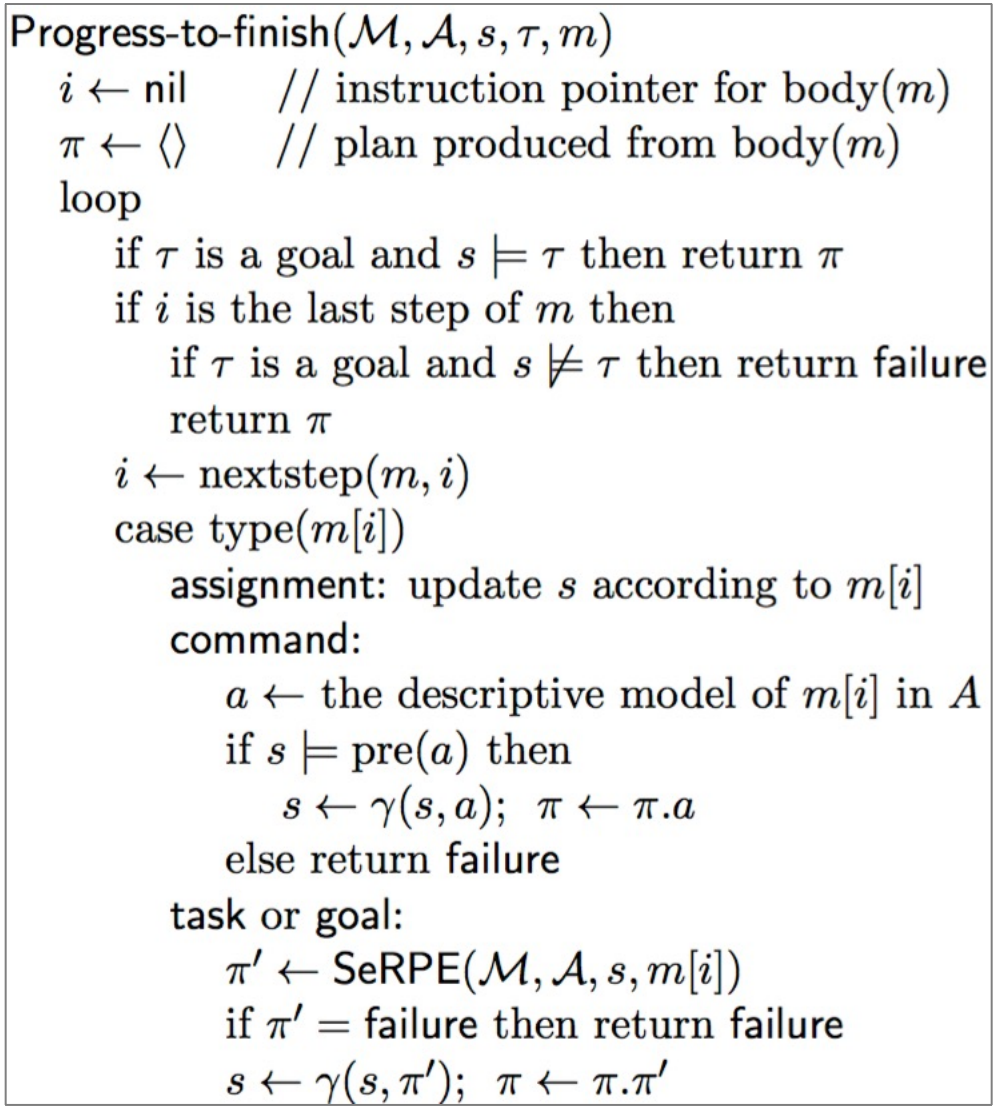
- Idea 1:
  - Planner uses Rae’s tasks and refinement methods
  - For each of Rae’s commans, have a classical action model
  - DFS or GBFS search among alternatives to see which works best

SeRPE (Sequential Refinement Planning Engine)

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



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



```
Progress-to-finish($\mathcal{M}, \mathcal{A}, s, \tau, m$)
    $i \leftarrow$ 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 $\pi$
        $i \leftarrow$ nextstep($m, i$)
        case type($m[i]$)
            assignment: update $s$ according to $m[i]$
            command:
                $a \leftarrow$ the descriptive model of $m[i]$ in $A$
                if $s \models$ pre($a$) then
                    $s \leftarrow \gamma(s, a)$;  $\pi \leftarrow \pi.a$
                else return failure
            task or goal:
                $\pi' \leftarrow$ SeRPE($\mathcal{M}, \mathcal{A}, s, m[i]$)
                if $\pi'$ = failure then return failure
                $s \leftarrow \gamma(s, \pi')$;  $\pi \leftarrow \pi.\pi'$
```

- 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
- But SeRPE there are problems …

Problems with SeRPE

Problem 1: difficult to implement

- Each time a method invokes a subtask, SeRPE makes a nondeterministic choice
- To implement deterministically
  - Each path in the search space is an execution trace of the body of a method
  - Need to backtrack over code execution
- Need to write a compiler that can do backtracking
  - Is it worth the effort?

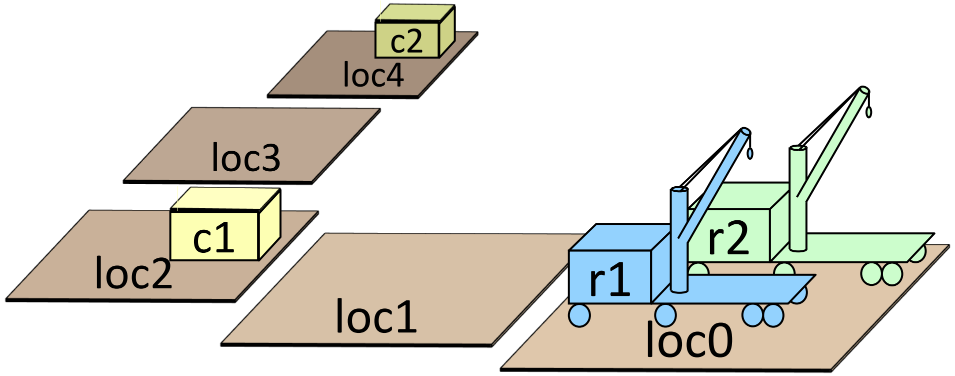
```
m-foo(k)
    task:   foo(k)
    pre:    …
    body:
        for i ← 1 to k:
            bar(i)
            baz(i)
```

Example:

- Suppose that
  - Each task has two applicable methods
  - When i=2, the 1st method for baz(2) fails
- Backtracking:
  - Try 2nd method for baz(2)
  - If it fails, try 2nd method for bar(2)
  - If it fails, backtrack to i = 1
    - Try 2nd method for baz(1)
    - If it fails, try 2nd method for bar(1)
  - If it fails, backtrack to task foo(k) …

Problems with SeRPE

- *Problem 2*: limitations of classical action models
  - e.g., the *fetch* example
- We don't know in advance what **perceive**'s effects will be
  - If we did, **perceive** wouldn't actually be needed



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

put(r,o,l)

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

perceive(r,l):

// robot r sees what objects are at l
pre: loc(r) = l
eff: ***?***

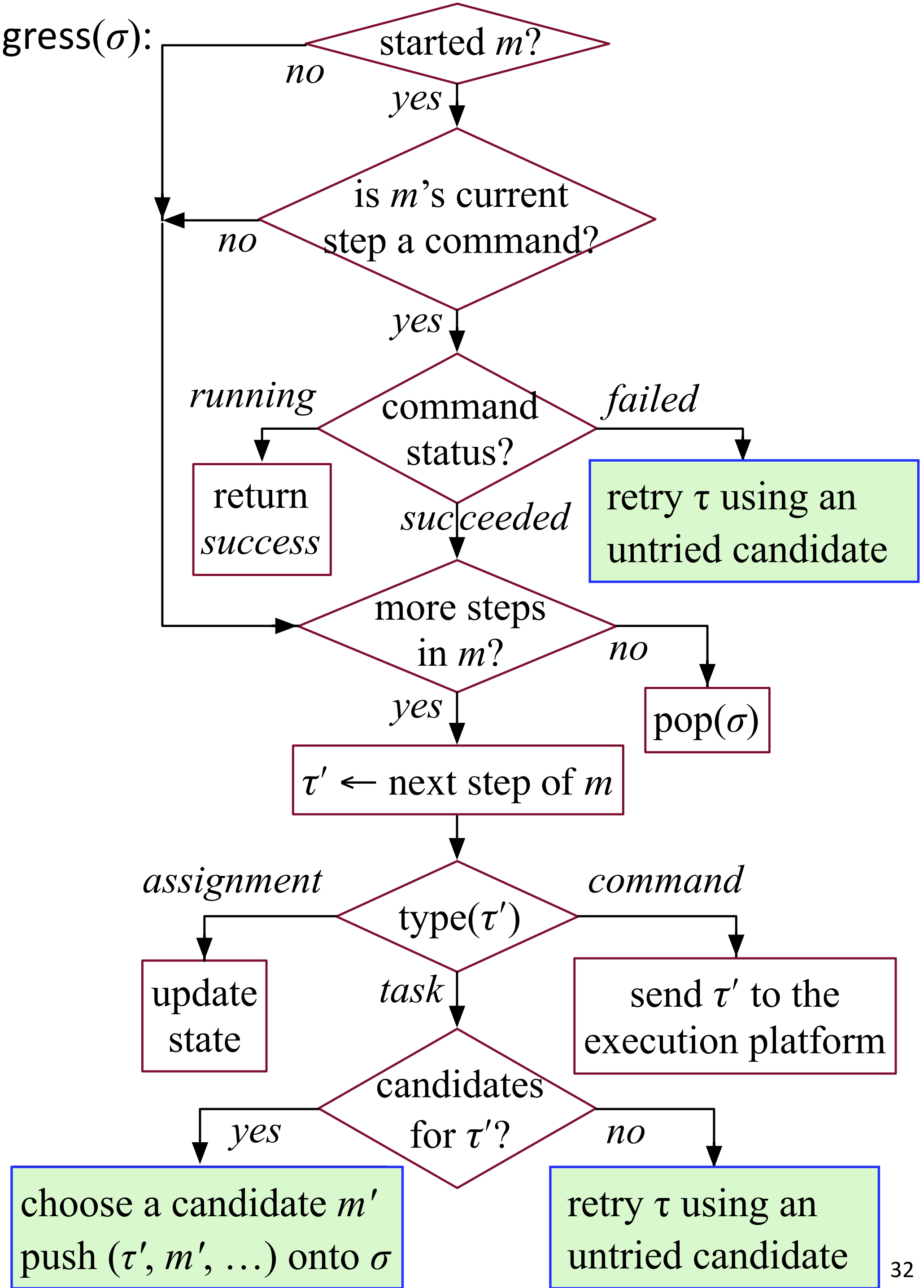
Planning for Rae

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

- Idea 2: simulation with multithreading or multiprocessing
  - Run Rae in simulated environment
    - Simulate the commands (see next page)
  - To choose among method instances, try all of them in parallel
- Planner returns the method instance *m* having the highest expected utility (≈ least expected cost)

Poll: is this a reasonable approach?

Progress(σ):

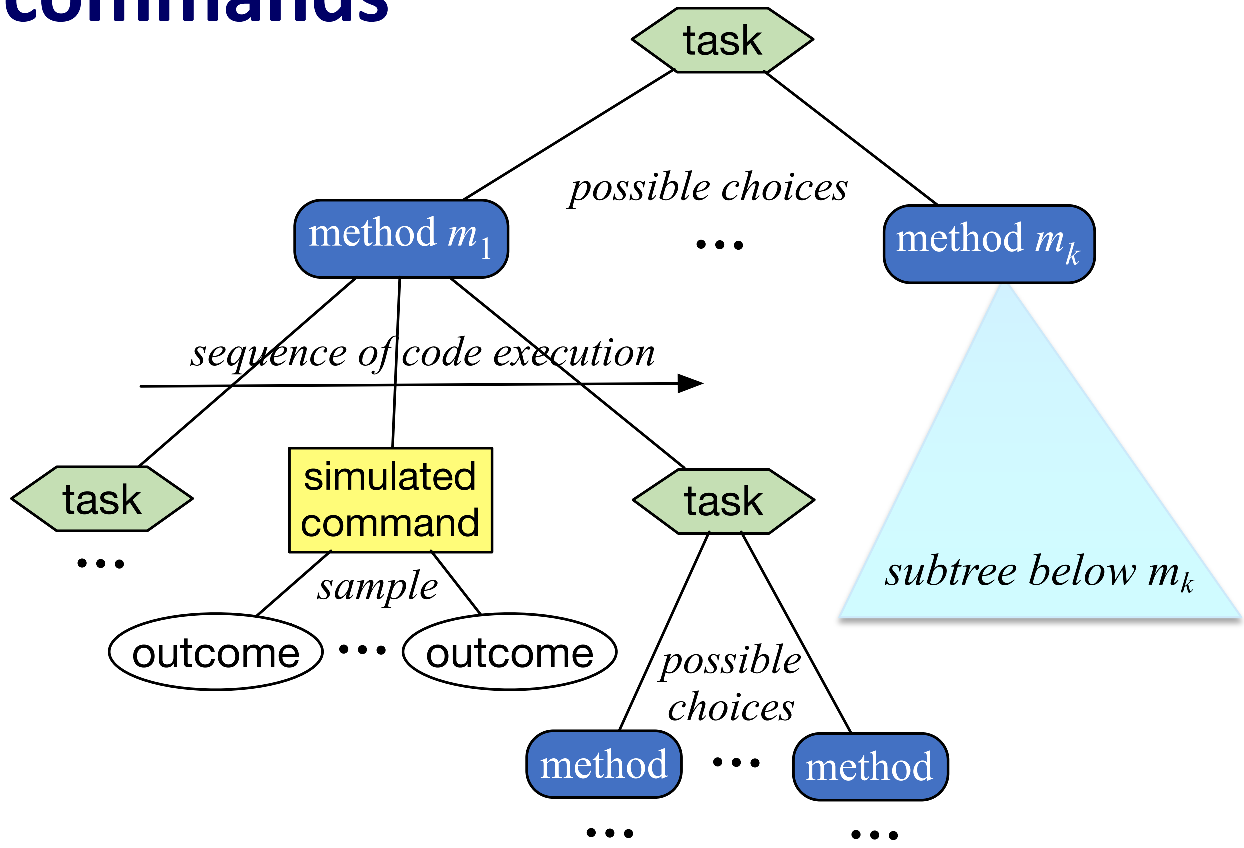


Simulating commands

- Simplest case:
  - probabilistic action template

```
$a(x_1, \ldots, x_k)$
        pre: …
         $(p_1)$ effects$_1$:  $e_{11}, e_{12}, \ldots$
              …
        $(p_m)$ effects$_m$:  $e_{m1}, e_{m2}, \ldots$
```

  - Choose effects$_i$ at random with probability p_i and use it to update the current state



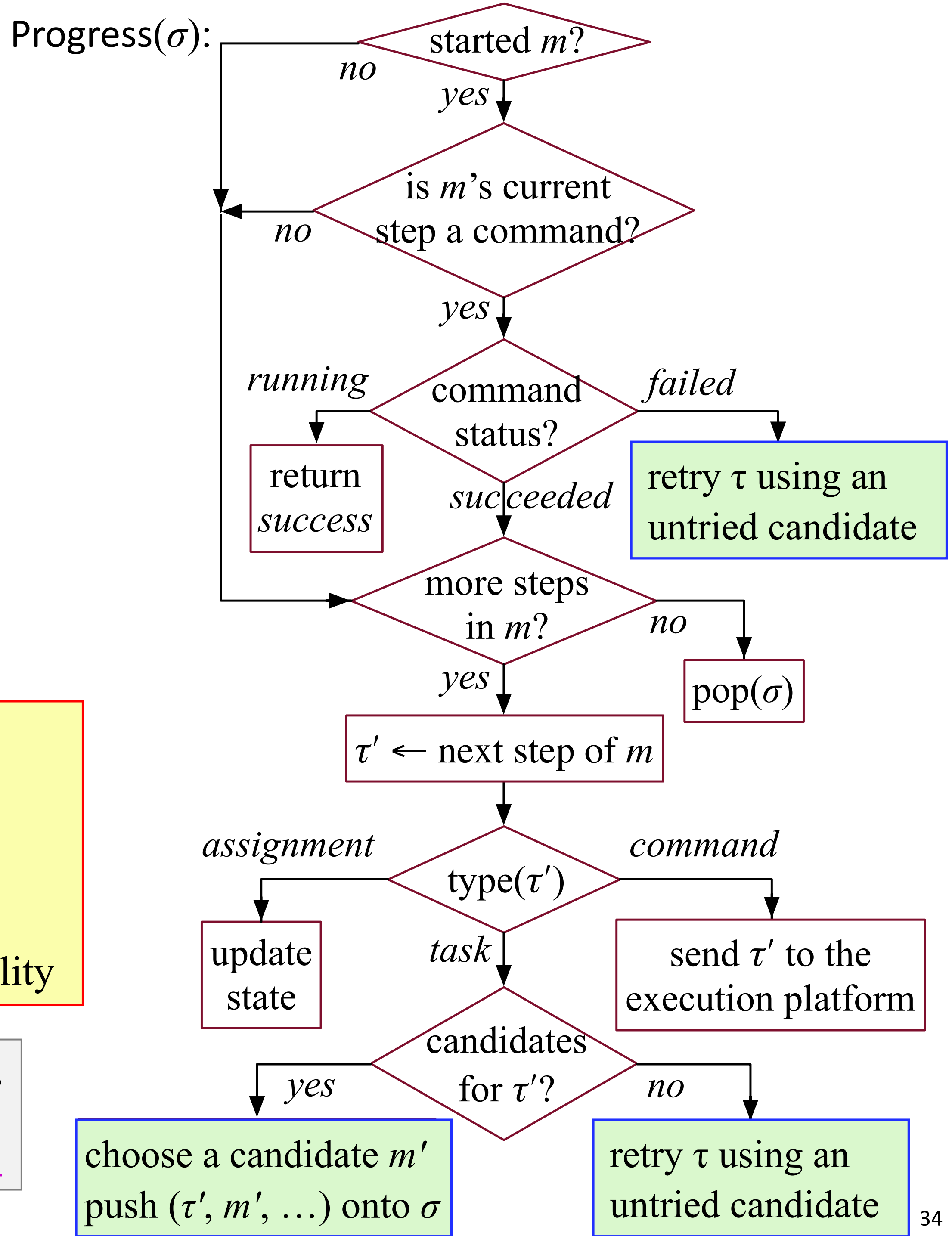
- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get simulated effects

Planning for Rae

```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance $m$ for τ
            create a refinement stack for τ, $m$
            add the stack to $Agenda$
        for each stack $σ$ in $Agenda$
            call Progress($σ$)
            if $σ$ 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 each choice's expected utility
  - Return the method instance *m* that has the highest estimated utility

Patra, Mason, Kumar, Traverso, Ghallab, and Nau. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. *ICAPS*, 2020. Best student paper honorable mention award. https://doi.org/10.1609/aaai.v33i01.33017691



Progress($σ$):

started m?

no

yes

is m's current step a command?

no

yes

running

command status?

failed

return *success*

succeeded

retry τ using an untried candidate

more steps in m?

no

yes

pop($σ$)

$τ' \leftarrow$ next step of m

assignment

type($τ'$)

command

update state

task

send $τ'$ to the execution platform

candidates for $τ'$?

yes

no

choose a candidate m' push ($τ'$, m', …) onto $σ$

retry τ using an untried candidate

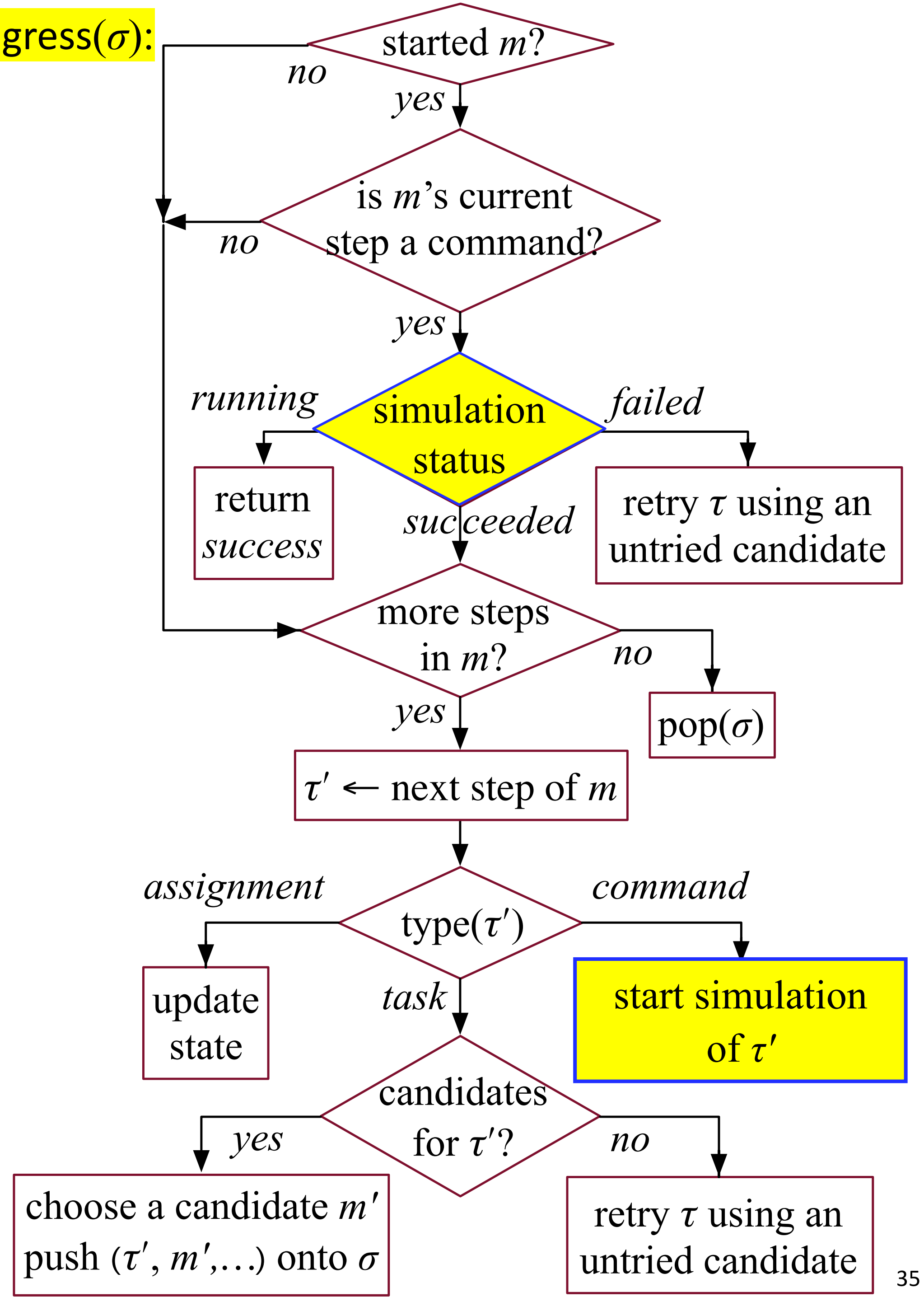
Planner

```
Plan-with-UPOM (task $\tau$):
    $Candidates \leftarrow$ {method instances relevant for $\tau$}
    for $i \leftarrow 1$ to $n$
        call UPOM($\tau$)
        update estimates of methods' expected utility
    return the $m \in Candidates$ 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 Simulate-Progress($\sigma$) $\neq$ failure
        if $\sigma$ is completed then return ($m$, utility)
    return failure
```

- Each call to UPOM does a Monte Carlo rollout
    - Simulated execution of RAE on τ

Simulate-Progress(σ):

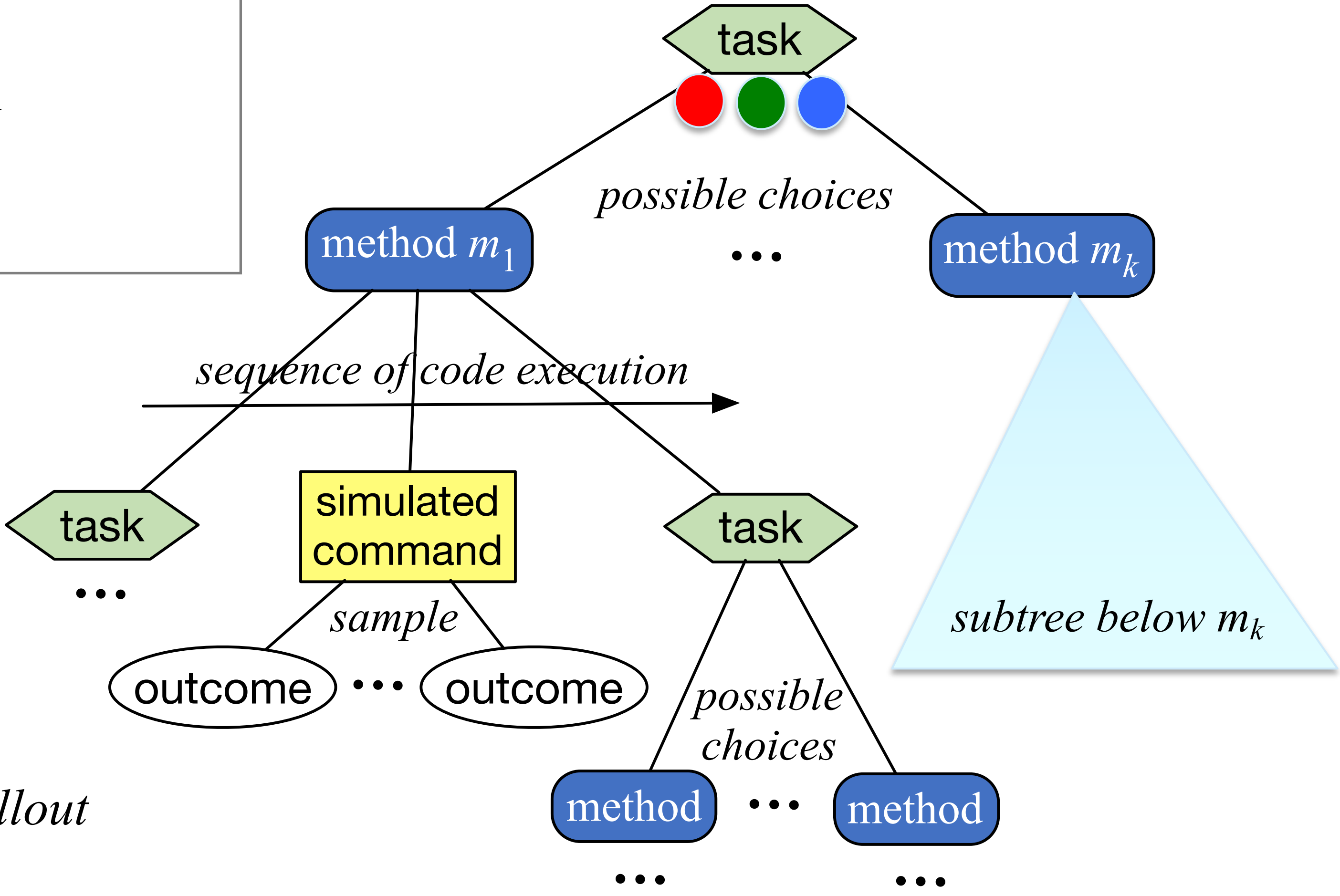

started m?
no
yes
is m's current step a command?
no
yes
simulation status
running
failed
return success
retry τ using an untried candidate
succeeded
more steps in m?
no
pop(σ)
yes
$\tau' \leftarrow$ next step of m
type(τ')
assignment
command
update state
task
start simulation of τ'
candidates for τ'?
yes
no
choose a candidate m' push (τ', m',…) onto σ
retry τ using an untried candidate

Monte-Carlo rollouts

```
Plan-with-UPOM (task $\tau$):
    $Candidates$ ← {method instances relevant for $\tau$}
    for $i$ ← 1 to $n$
        call UPOM($\tau$)
        update estimates of methods' expected utility
    return the $m \in Candidates$ that has
        the highest estimated utility
```

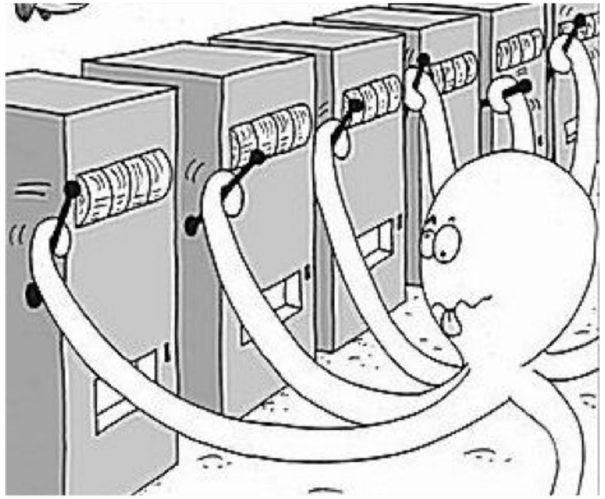
```
UPOM($\tau$):
    choose a method instance $m$ for $\tau$
    create refinement stack $\sigma$ for $\tau$ and $m$
    loop while Simulate-Progress($\sigma$) $\neq$ $failure$
        if $\sigma$ is completed then return ($m$, $utility$)
    return $failure$
```

- Each call to UPOM does a *Monte Carlo rollout*
    - Simulated execution of RAE on τ



Digression: Monte Carlo rollouts

- Multi-arm bandit problem
  - Statistical model of sequential experiments
  - Name derived from *one-armed bandit* (slot machine)
- 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 an action that has given you high rewards in the past
  - ***Exploration***: choose an action that's less familiar, in hopes that it might produce a higher reward



UCB (Upper Confidence Bound) Algorithm

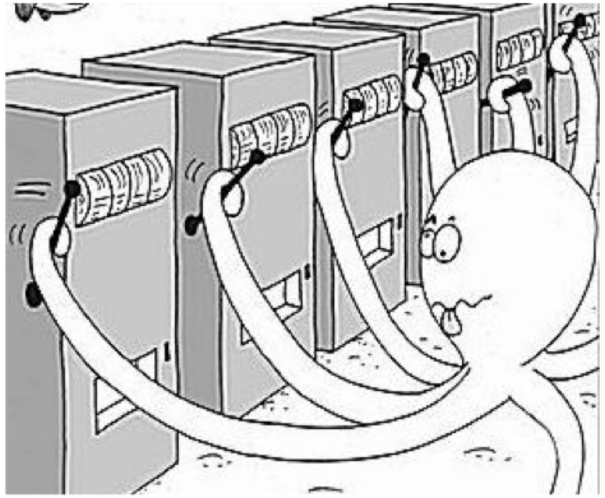
- Assume all rewards are between 0 and 1
  - If they aren't, normalize them
- For each action a, let
  - $r(a)$ = average reward you've gotten from a
  - $n(a)$ = number of times you've tried a
  - $n_t = \sum_a n(a)$
  - $Q(a) = r(a) + \sqrt{2(\ln n_t)/n(a)}$

```
UCB:
    if there are any untried actions:
        $\tilde{a}$ ← any untried action
    else:
        $\tilde{a}$ ← argmax$_a$ $Q(a)$
    perform $\tilde{a}$
    update $r(\tilde{a})$, $n(\tilde{a})$, $n_t$, $Q(\tilde{a})$
```

- Theorem (given some assumptions):

  As the number of calls to UCB → ∞,

  UCB's choice at each call → optimal choice



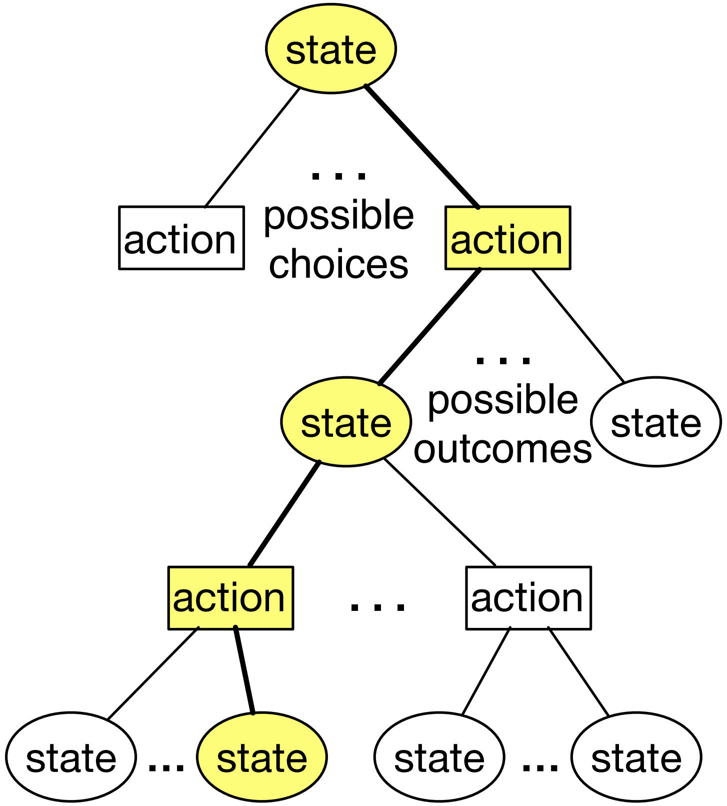
UCT Algorithm

- 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’s outcome ⇒ next state *s*

- How to use UCT:
  - Call it many times, return action with highest expected utility
- Theorem:

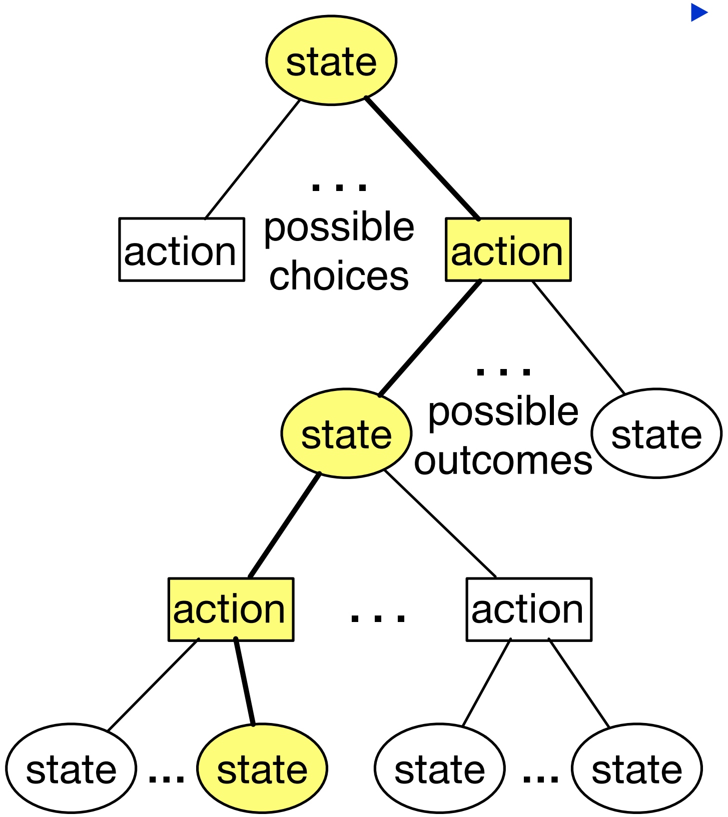
  As number of calls to UCT → ∞,

  choice converges to optimal

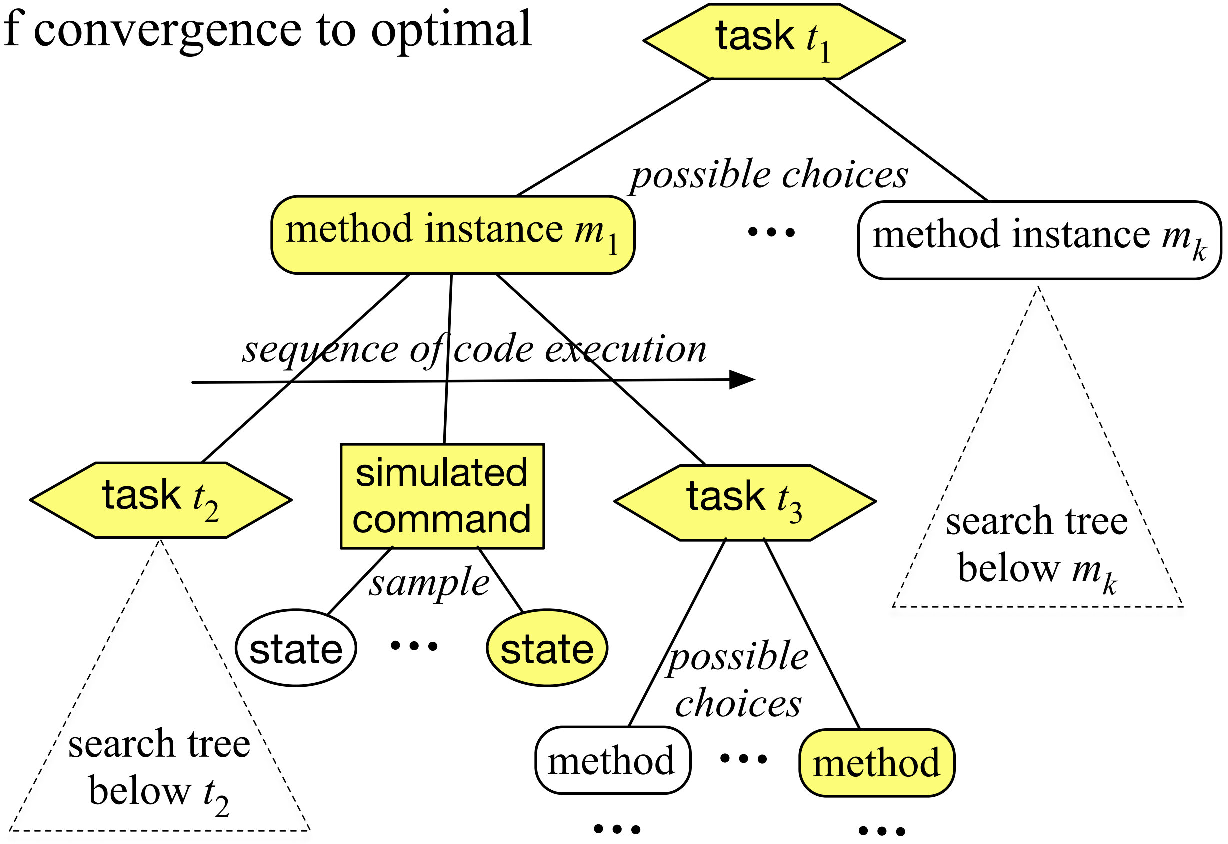


Convergence

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

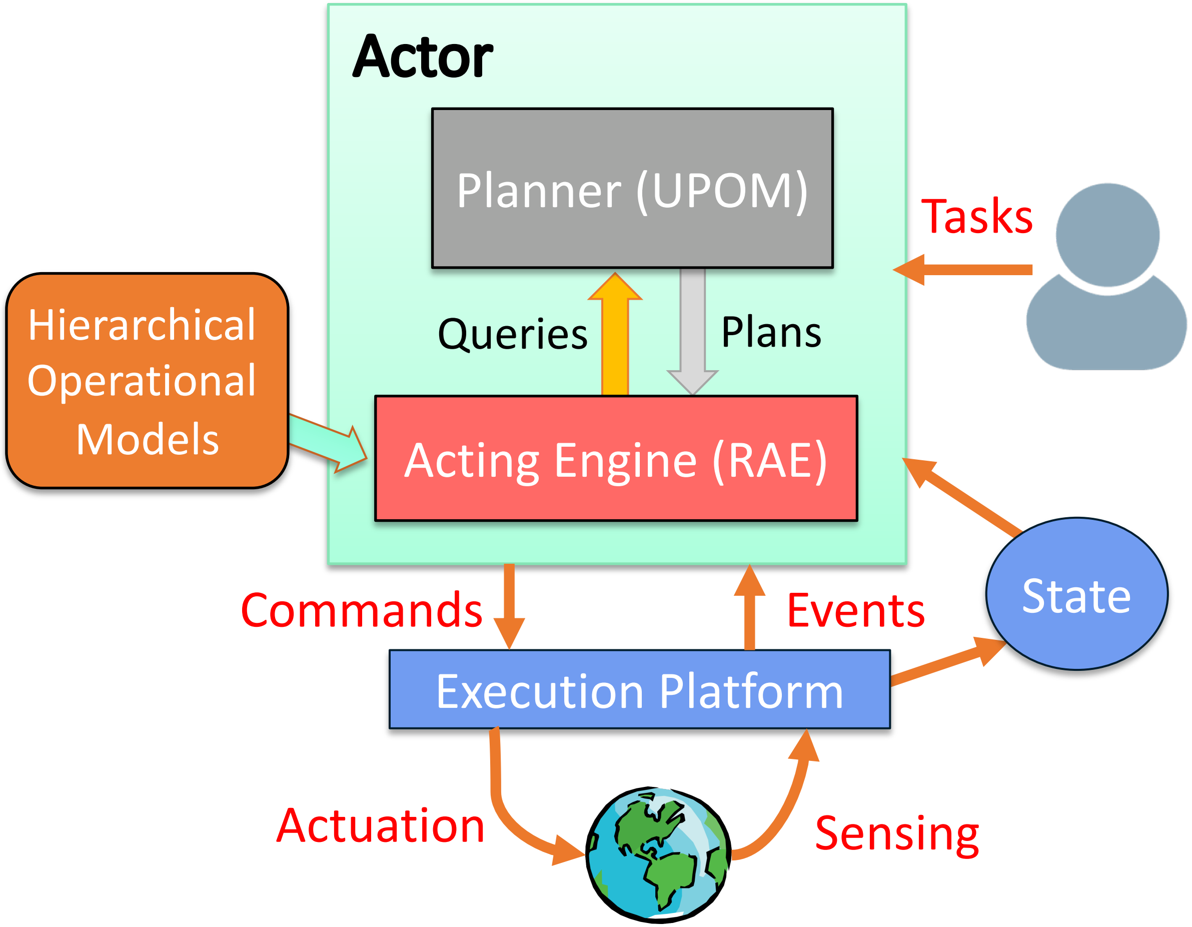


- UPOM search tree more complicated
  - tasks, method instances, commands, code execution
- If no exogenous events,
  - Can map it to UCT search of a complicated MDP
  - Proof of convergence to optimal



Outline

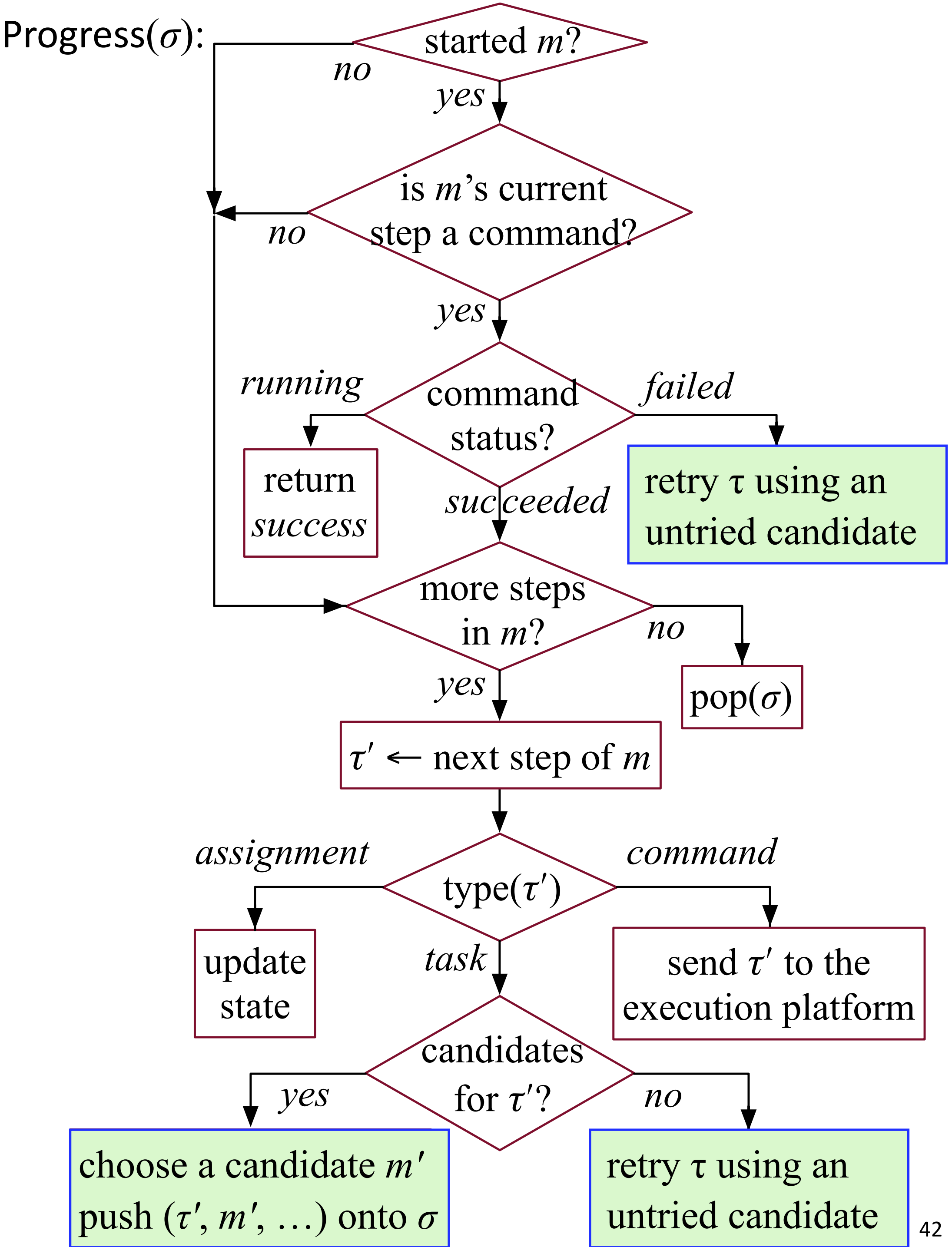
1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. *Acting with Planning* (RAE+UPOM)
6. Learning
7. Evaluation, Application



RAE + UPOM

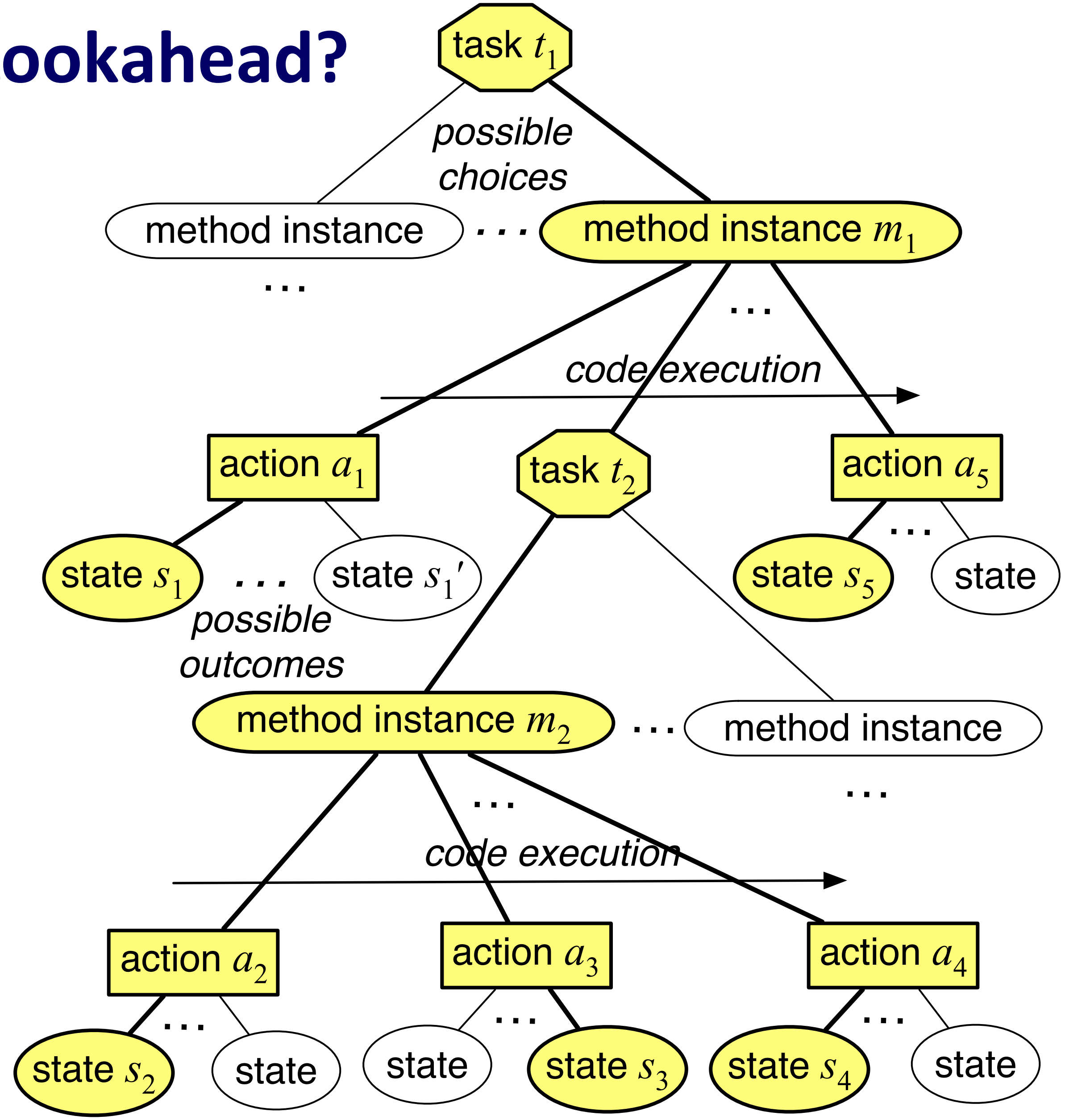
```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance $m$ for τ
            create a refinement stack for τ, $m$
            add the stack to $Agenda$
        for each stack $\sigma$ in $Agenda$
          call Progress($\sigma$)
          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/



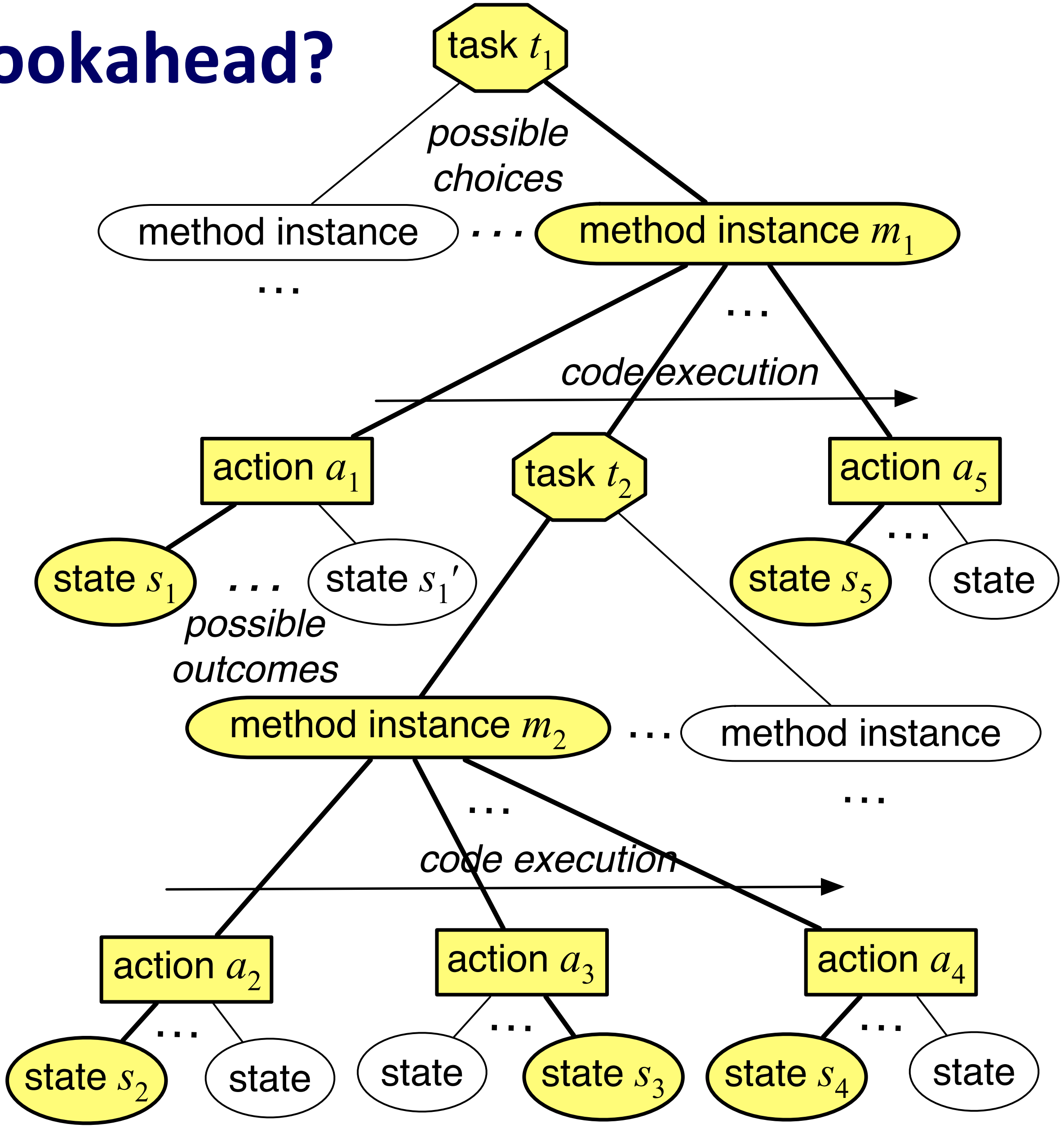
Can we use UPOM with Run-Lookahead?

- 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
- Problem
  - Run-lookahead calls UPOM′, gets π, 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 to ensure that methods can work in unexpected states



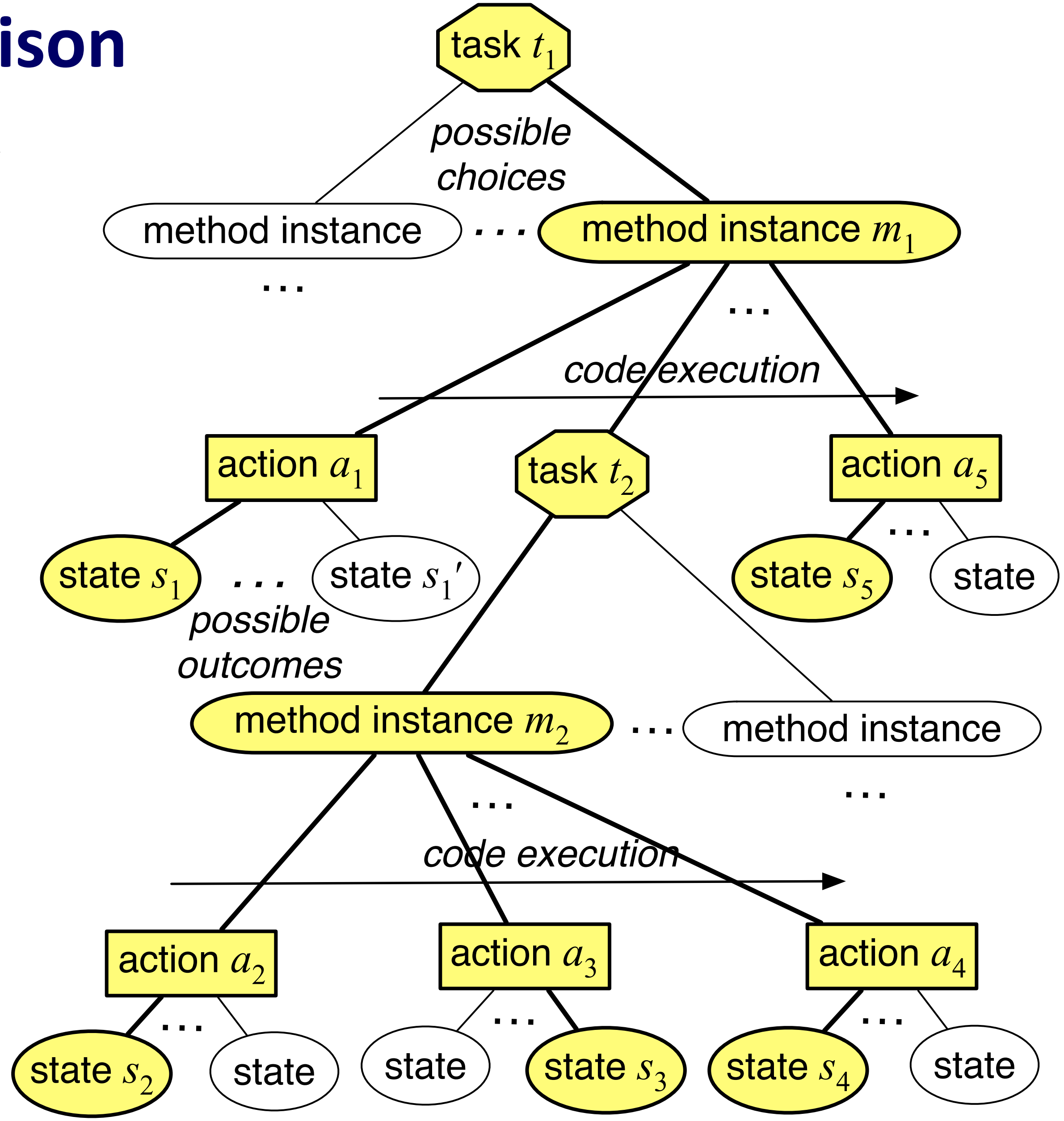
Can we use UPOM with Run-Lazy-Lookahead?

- Run-Lazy-Lookahead calls UPOM′, UPOM′ returns $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
- Run-Lazy-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
  - Need to modify the methods to work in unexpected states



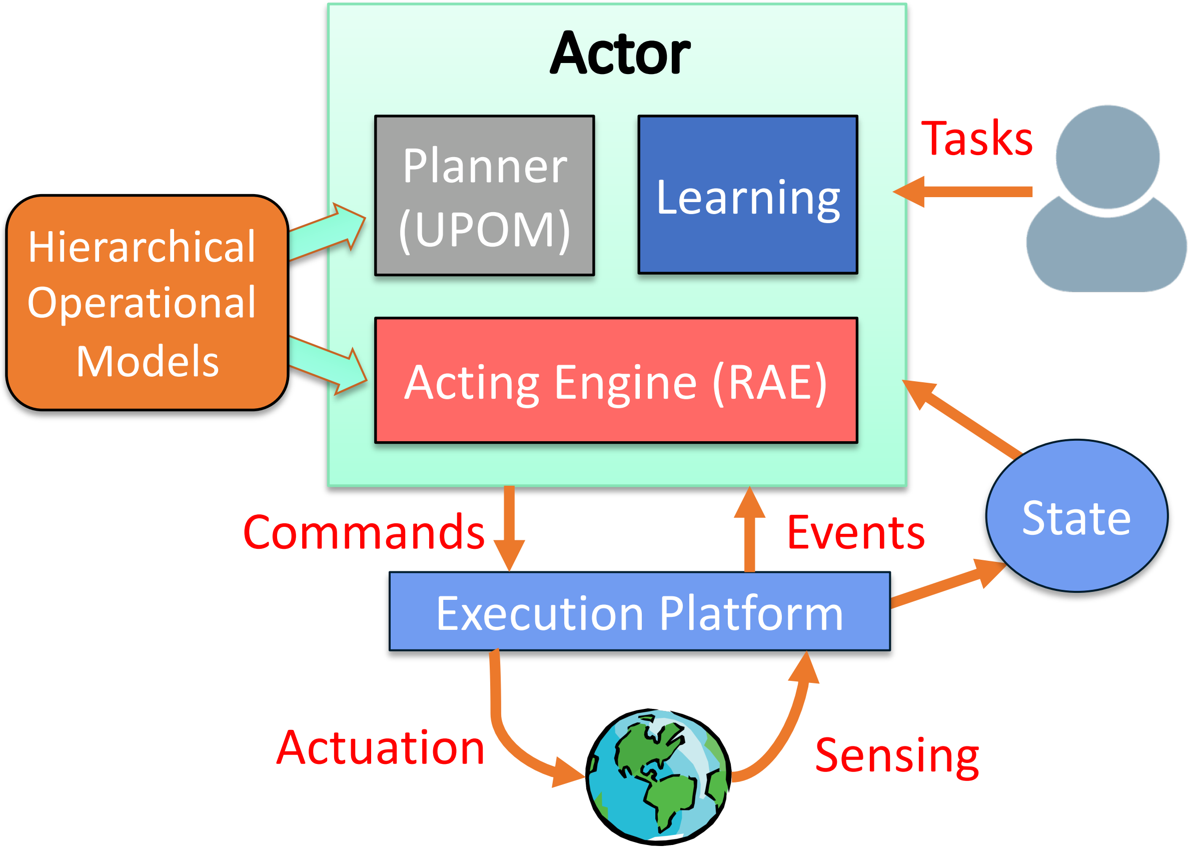
Comparison

- 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 π
  - 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



Outline

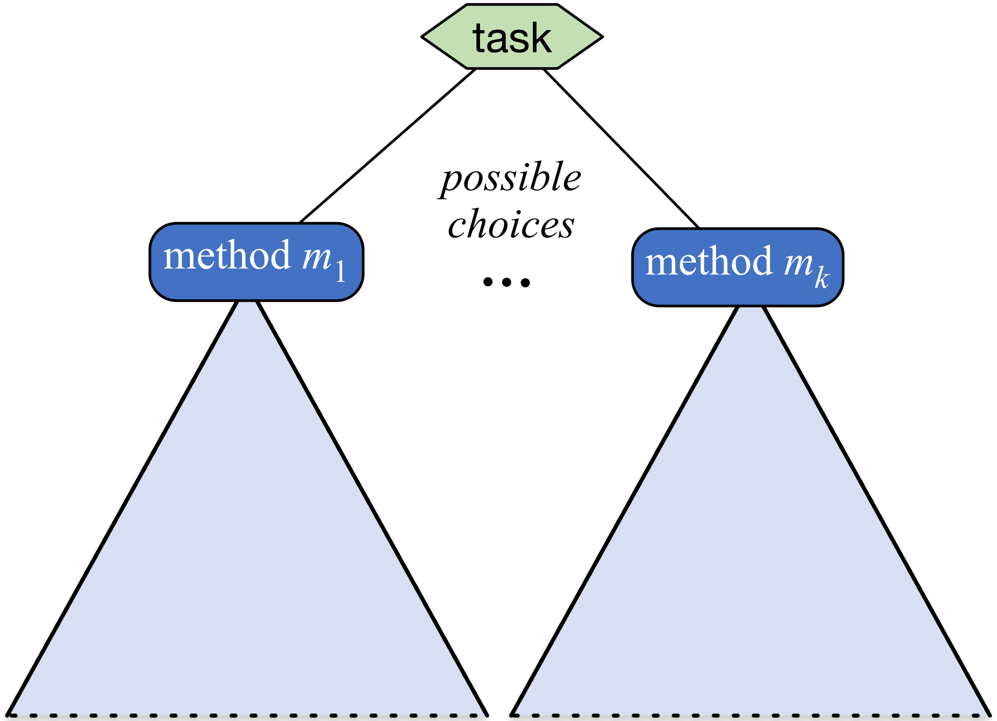
1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. *Learning*
7. Evaluation, Application



Motivation

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

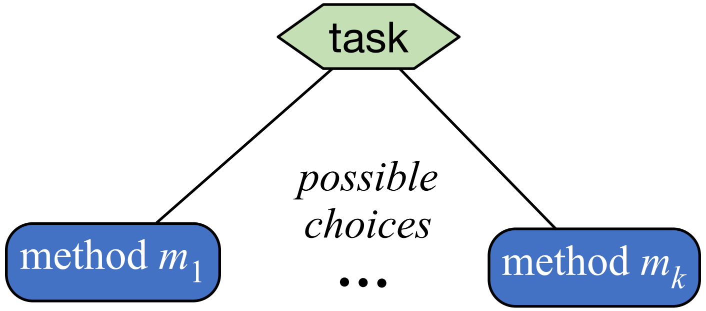
- Case 1: no time to search at all
  - need a choice function
- Case 2: enough time to do partial search
  - Receding horizon
    - Cut off search at depth d_{max} or when we run out of time
    - At leaf nodes, use heuristic function to estimated expected utility



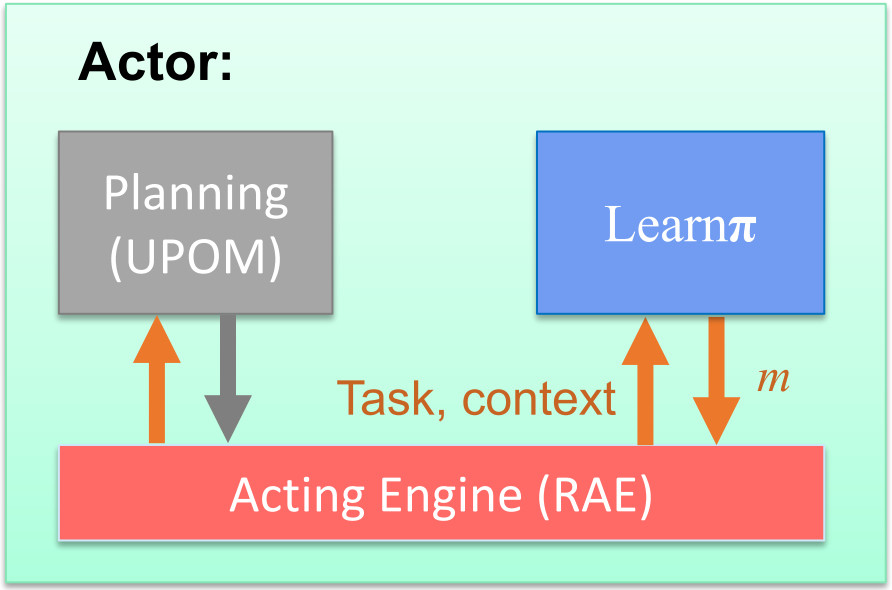
- Learning algorithms:
  - Learn**π**: learns a choice function
  - LearnH: learns a heuristic function

Integration with Learning

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

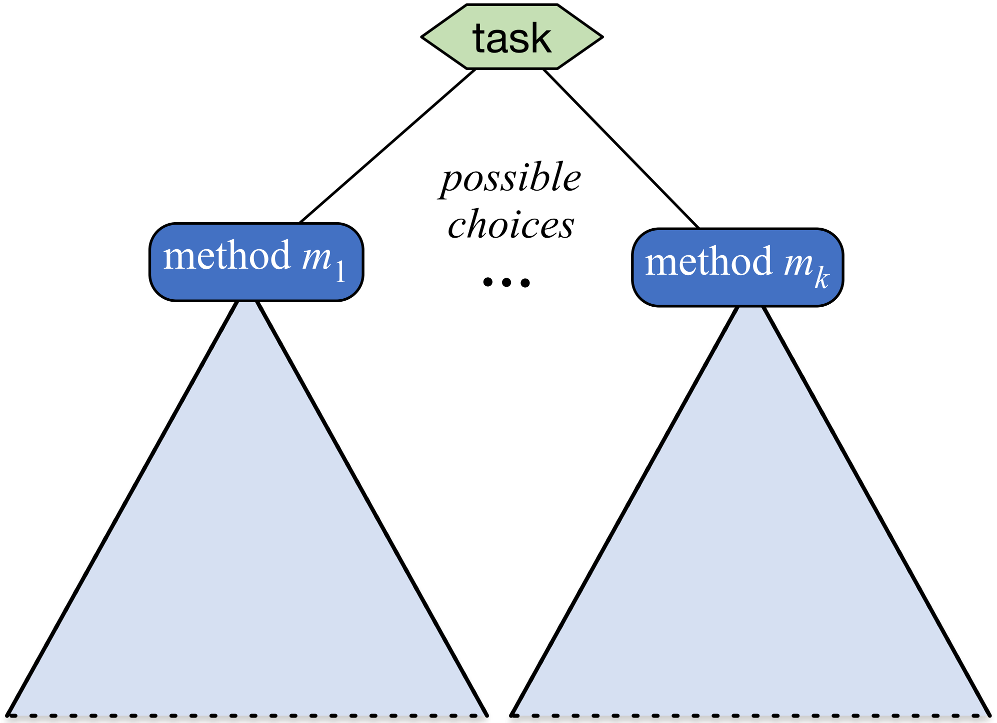


- Learnπ
  - Learns function for choosing a method
  - Given current task and context (state and other information), choose m from the set of available refinement methods
  - Useful if there isn't enough time to use UPOM

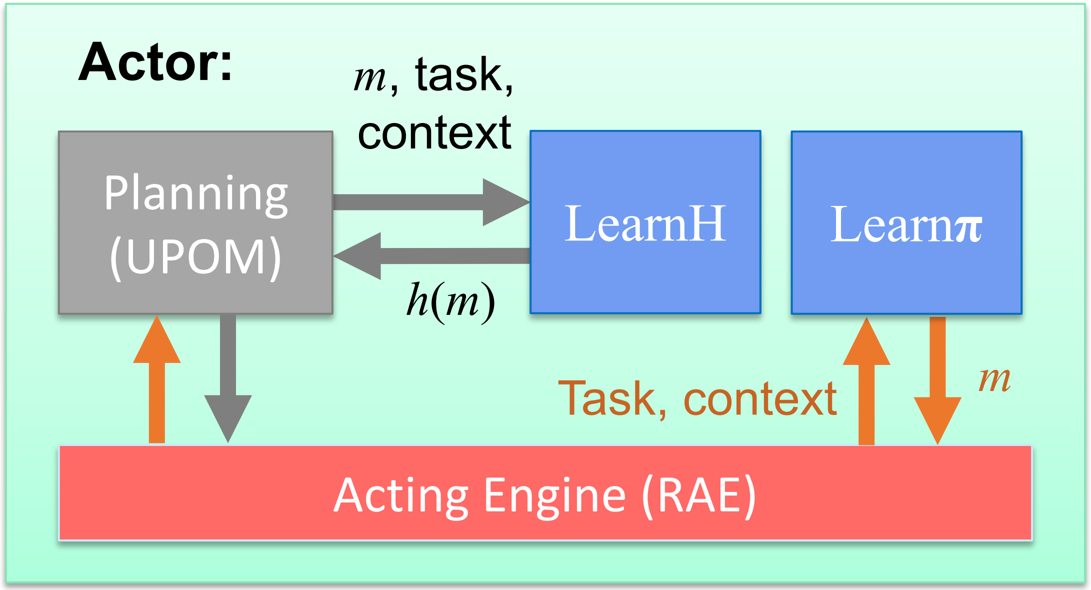


Integration with Learning

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

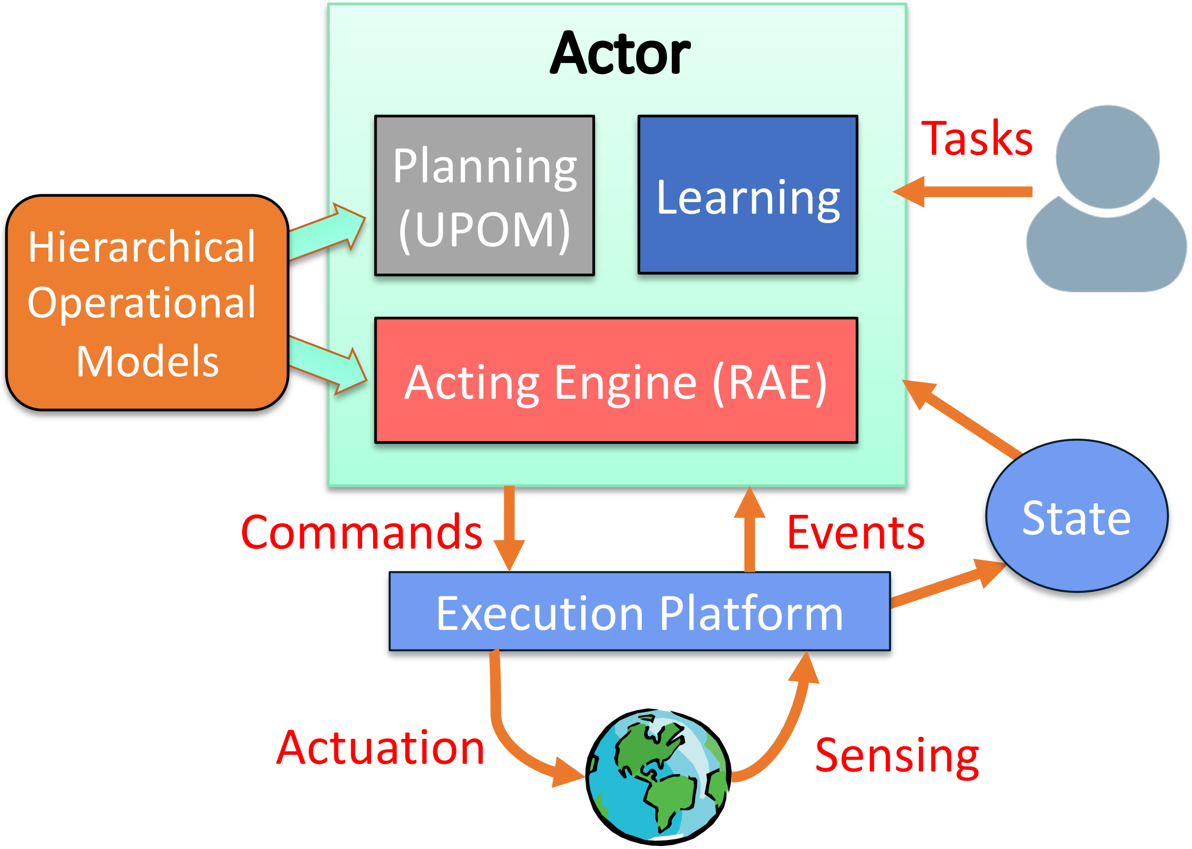


- LearnH
  - Learns a heuristic function to guide UPOM's search
  - UPOM can use it to estimate expected utility at leaf nodes
  - Useful if there isn't enough time to search all the way to the end

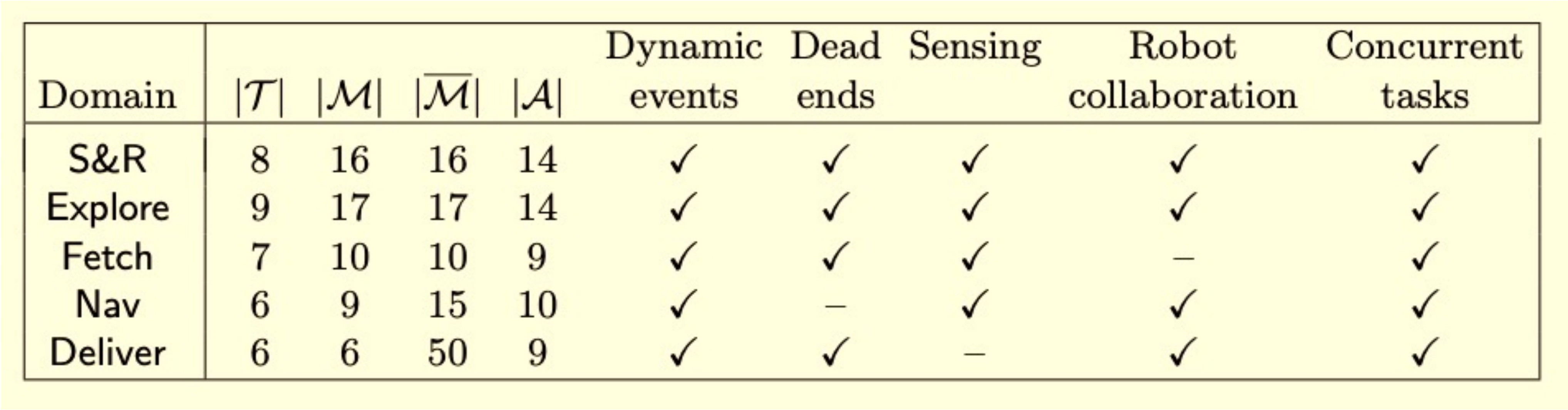


Outline

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



Experimental Evaluation



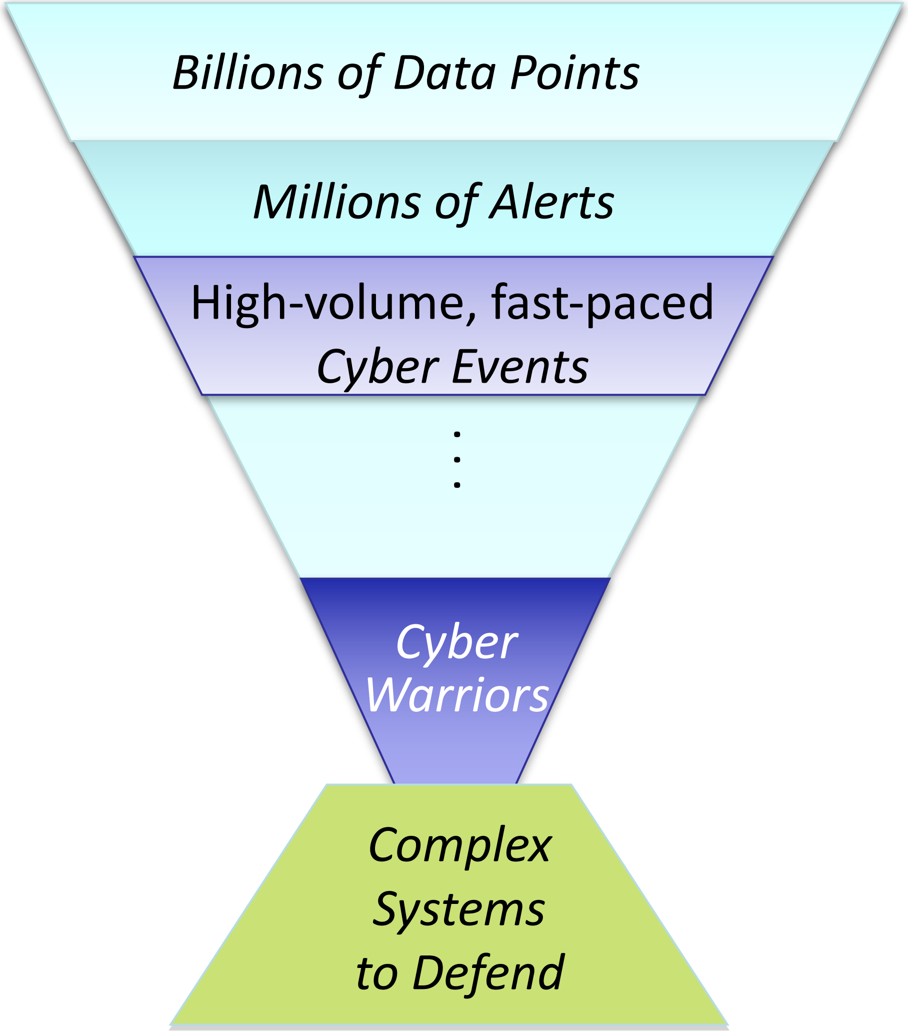
Domain	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{M}\rvert$	$\lvert\overline{\mathcal{M}}\rvert$	$\lvert\mathcal{A}\rvert$	Dynamic events	Dead ends	Sensing	Robot collaboration	Concurrent tasks
S&R	8	16	16	14	✓	✓	✓	✓	✓
Explore	9	17	17	14	✓	✓	✓	✓	✓
Fetch	7	10	10	9	✓	✓	✓	–	✓
Nav	6	9	15	10	✓	–	✓	✓	✓
Deliver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

- 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

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

S. Patra, A. Velasquez, M. Kang, and D. Nau. Using online planning and acting to recover from cyberattacks on software-defined networks. In *Proc. Innovative Applications of AI Conference (IAAI)*, Feb. 2021. https://www.cs.umd.edu/~nau/papers/patra2021using.pdf


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/