UNIVERSITÄT ZU LÜBECK

Automated Planning and Acting – Refinement Methods

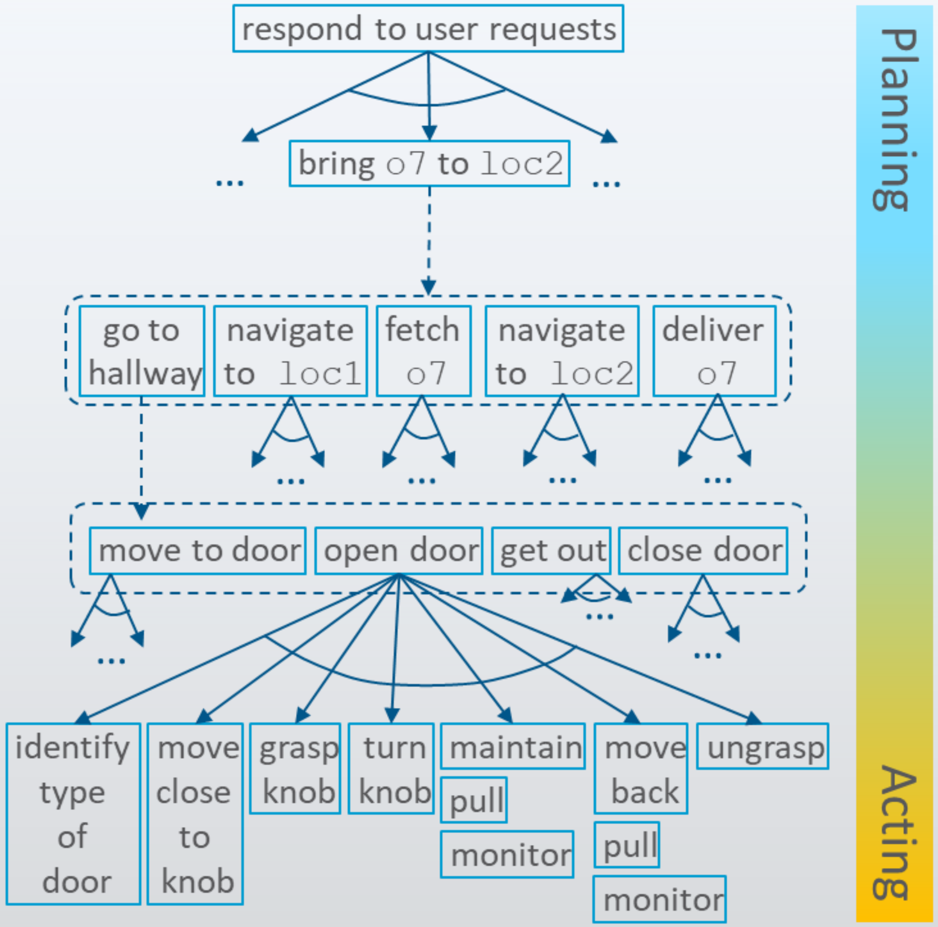
Institute of Information Systems

Mattis Hartwig

Content

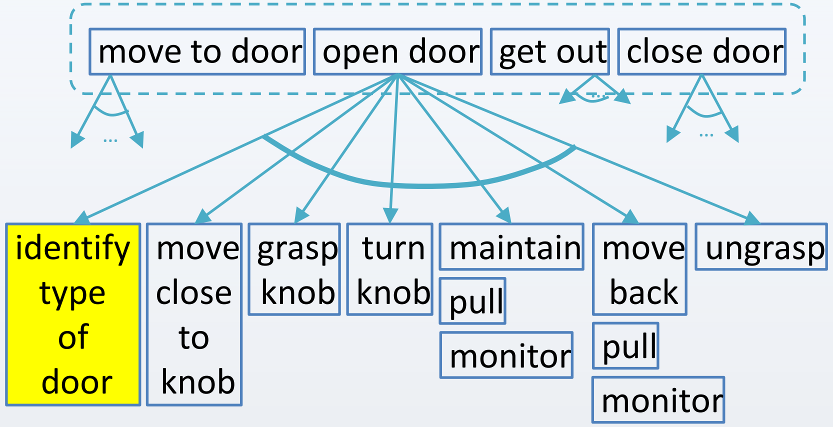
Motivation

- Hierarchically organized deliberation
  - At high levels, abstract actions
  - At lower levels, more detail
- Refine abstract actions into ways of carrying out those actions
  - How?

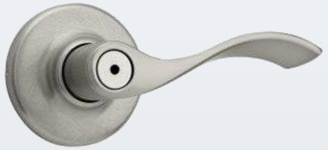


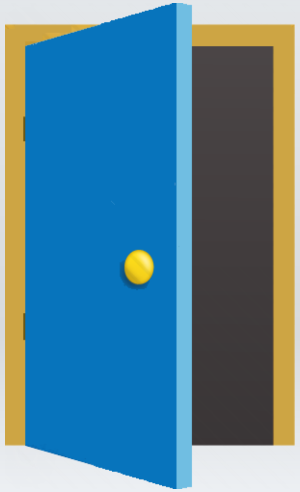
Opening a Door

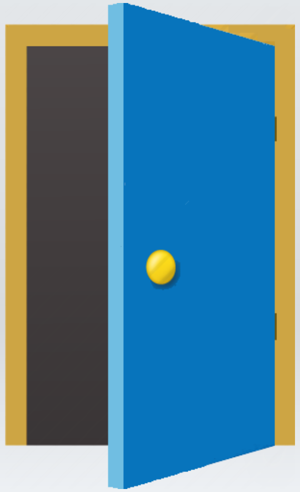
- Many different methods, depending on what kind of door
    - Sliding or hinged?
    - Hinge on left or right?
    - Open toward or away?
    - Knob, lever, push bar
    - Pull handle, push plate
    - Something else?



move to door
open door
get out
close door
…
…
…
identify type of door
move close to knob
grasp knob
turn knob
maintain
pull
monitor
move back
pull
monitor
ungrasp





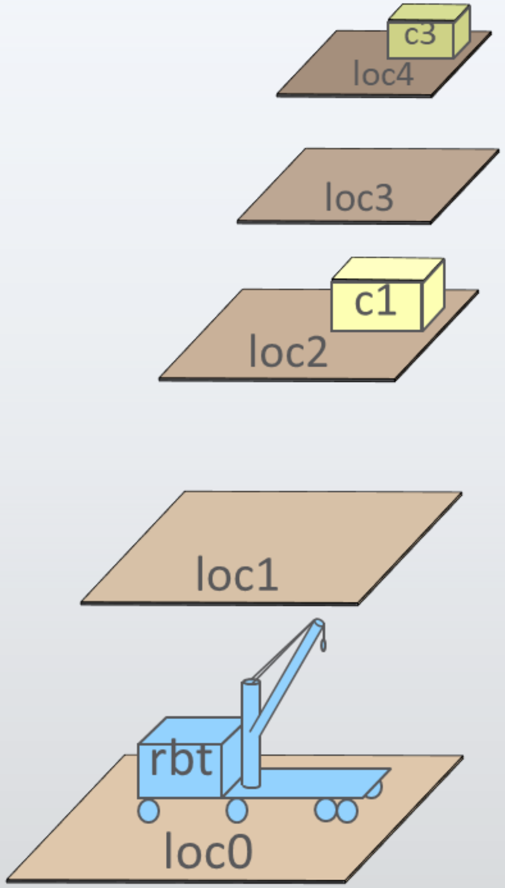


Assumptions

- Removes/weakens assumptions from classical planning
- Characteristics
    - Dynamic environment
    - Imperfect information
    - Overlapping actions
    - Nondeterminism
    - Hierarchy
    - Discrete and continuous variables

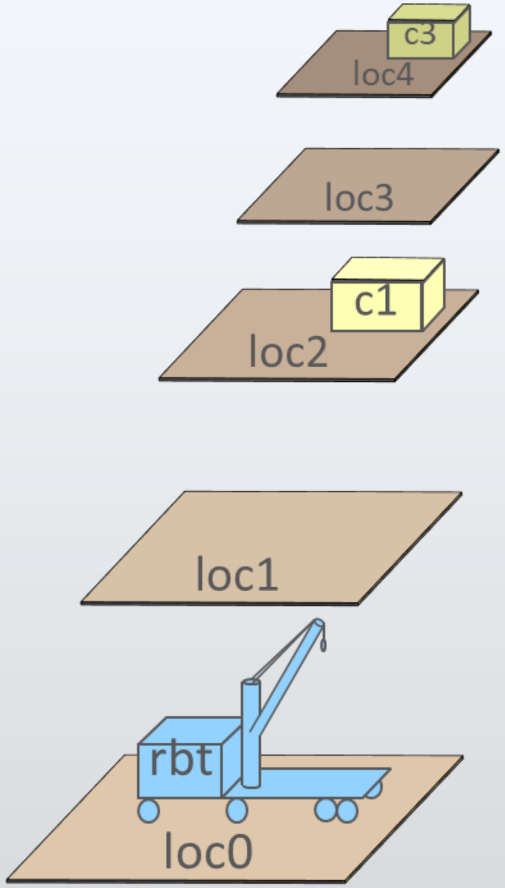
State-variable Representation (Recap)

- Objects:
  - $Robots = \{rbt\}$
  - $Containers = \{c1, c2, c3, \dots\}$
  - $Locations = \{loc0, loc1, loc2, \dots\}$
- State variables: syntactic terms to which we can assign values
  - $loc(r) \in Locations$
  - $load(r) \in Containers \cup \{nil\}$
  - $pos(c) \in Locations \cup Robots \cup \{unknown\}$
  - $view(r, l) \in \{T, F\}$
    - whether robot r has looked at location l
    - r can only see what is at its current location
- State: assign a value to each state variable
  - $\{loc(rbt) = loc0, pos(c1) = loc2,\ pos(c3) = loc4, pos(c2) = \ unknown, \dots\}$



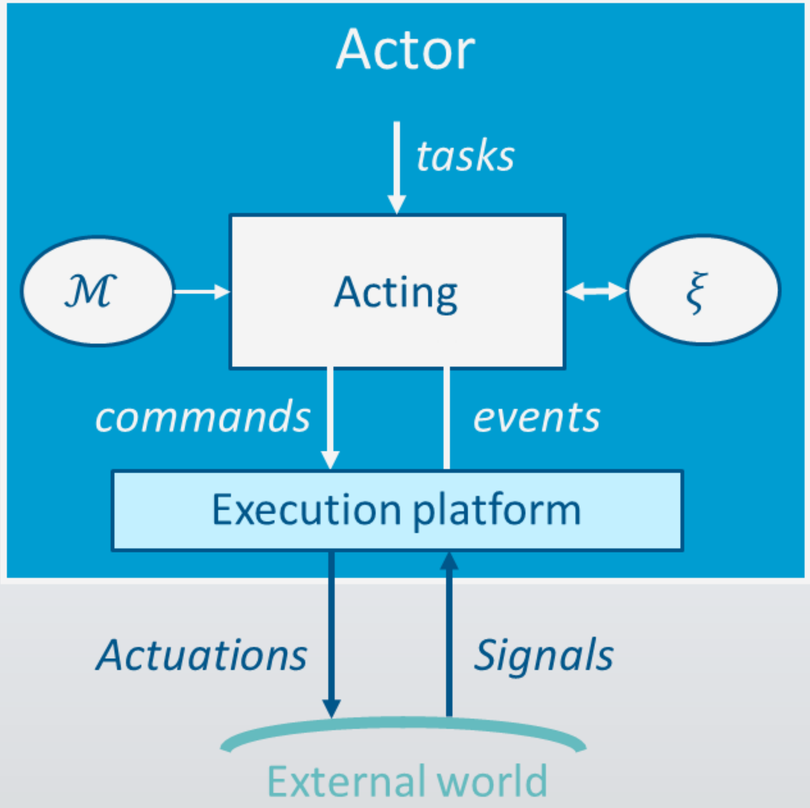
State-variable Representation: Extensions

- Range $\mathcal{R}(x)$
  - Can be finite, infinite, continuous, discontinuous, vectors, matrices, other data structures
- Assignment statement $x \leftarrow expr$
  - Expression that returns a ground value in $\mathcal{R}(x)$ and has no side-effects on the current state
- Tests (e.g., preconditions)
  - Simple: $x = v, x \neq v, x > v, x < v$
  - Compound: conjunction, disjunction, or negation of simple tests



Commands

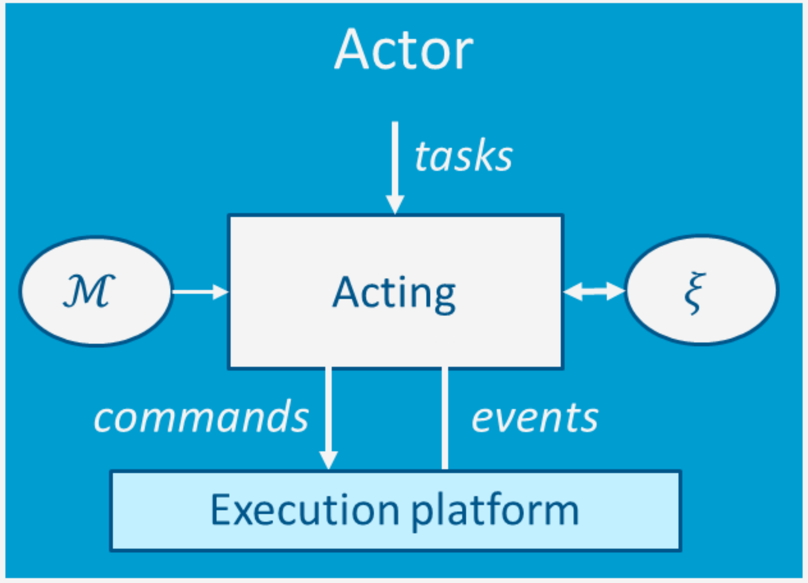
- **Command**: primitive function that the execution platform can perform
  - $take(r, o, l)$:
    robot r takes object o
    at location l
  - $put(r, o, l)$:
    r puts o at location l
  - $perceive(r, l)$:
    robot r perceives what objects are at l
    - r can only perceive what is at its current location
- **Event**: occurrence detected by execution platform
  - $event{-}name(args)$
  - Exogenous changes in the environment to which the actor may have to react
  - E.g., emergency signal, arrival of transportation vehicle



- For later
  - $\mathcal{M}$: library of methods
  - ξ: current state (abstraction)

Tasks and Methods

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



$method\!-\!name(arg_1, \ldots, arg_k)$

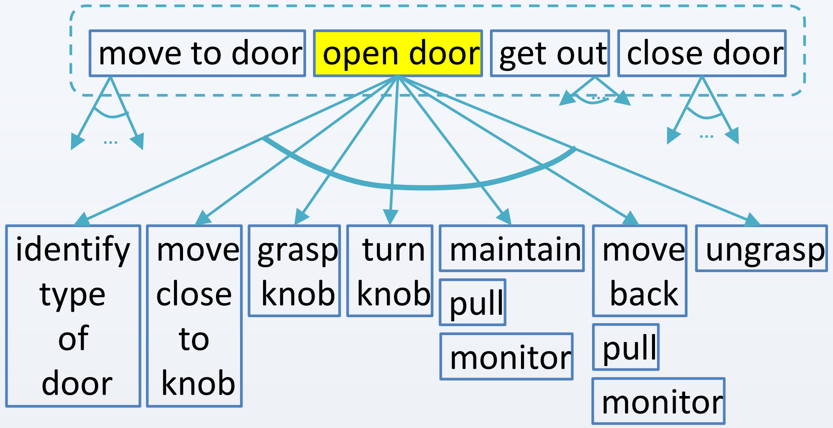
task: *task-identifier*

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

Example: “open door” task



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

→ Refinement method

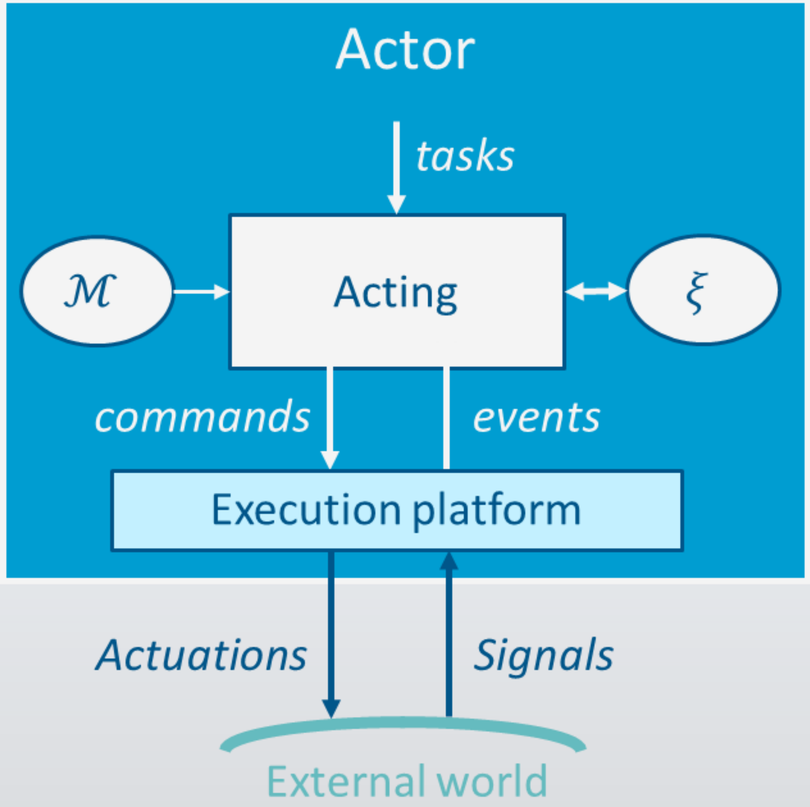
```
m-opendoor(r,d,l,h)
    task:   opendoor(r,d)
    pre:    loc(r) = l ∧ adj(l,d)
            ∧ handle(d,h)
    body:
        while ¬reachable(r,h) do
            move-close(r,h)
        monitor-status(r,d)
        if door-status(d)=closed then
            unlatch(r,d)
        throw-wide(r,d)
        end-monitor-status(r,d)
```

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

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

Rae (Refinement Acting Engine)

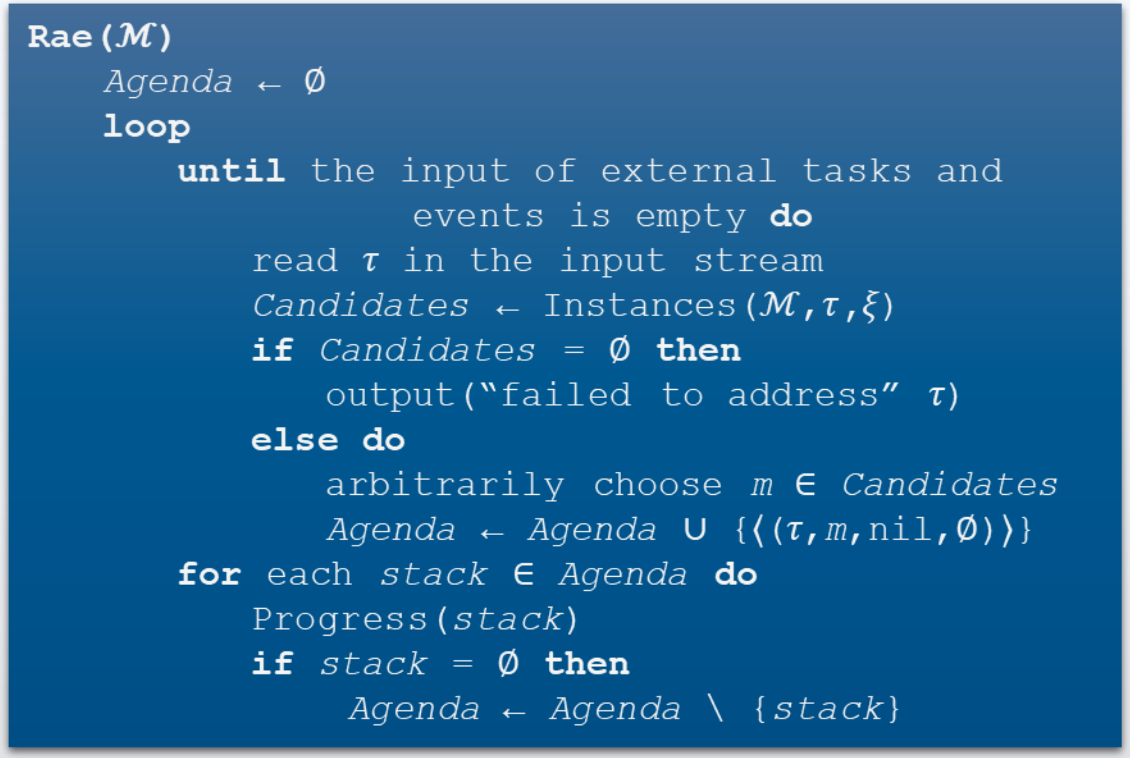
- Based on OpenPRS
  - Programming language,
    open-source robotics software
  - Deployed in many applications
- Input
  - External tasks, events, current state ξ, library of methods $\mathcal{M}$
- Output
  - Commands to execution platform
- Perform multiple tasks / events in parallel
  - Purely reactive, no lookahead
- For each task/event,
  a refinement stack
  - current path in Rae's search tree for the task / event
- Agenda
  = {all current refinement stacks}



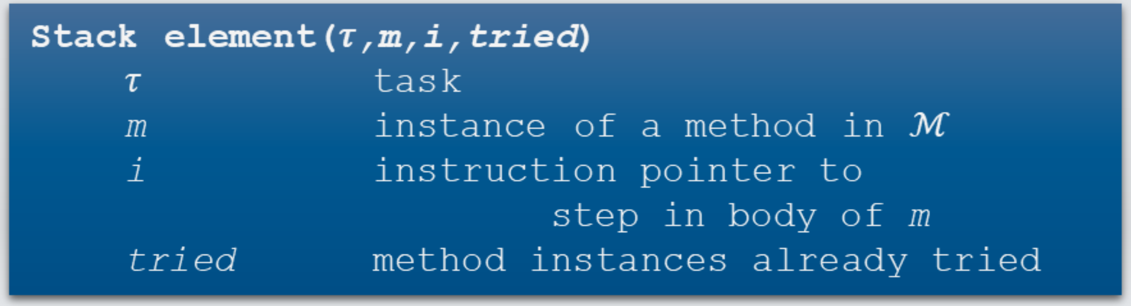
RAE

- Basic idea

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

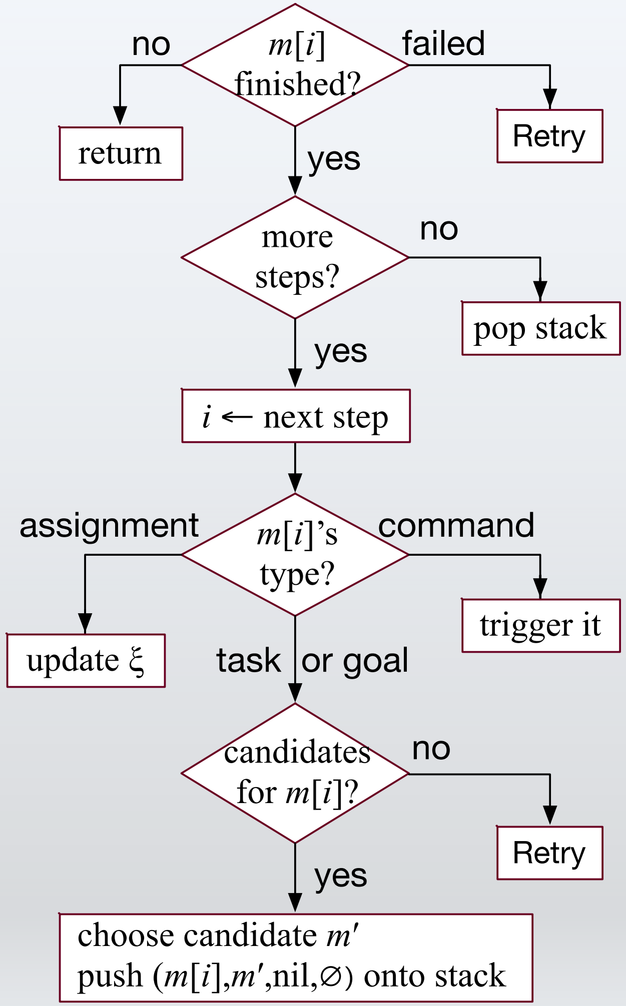


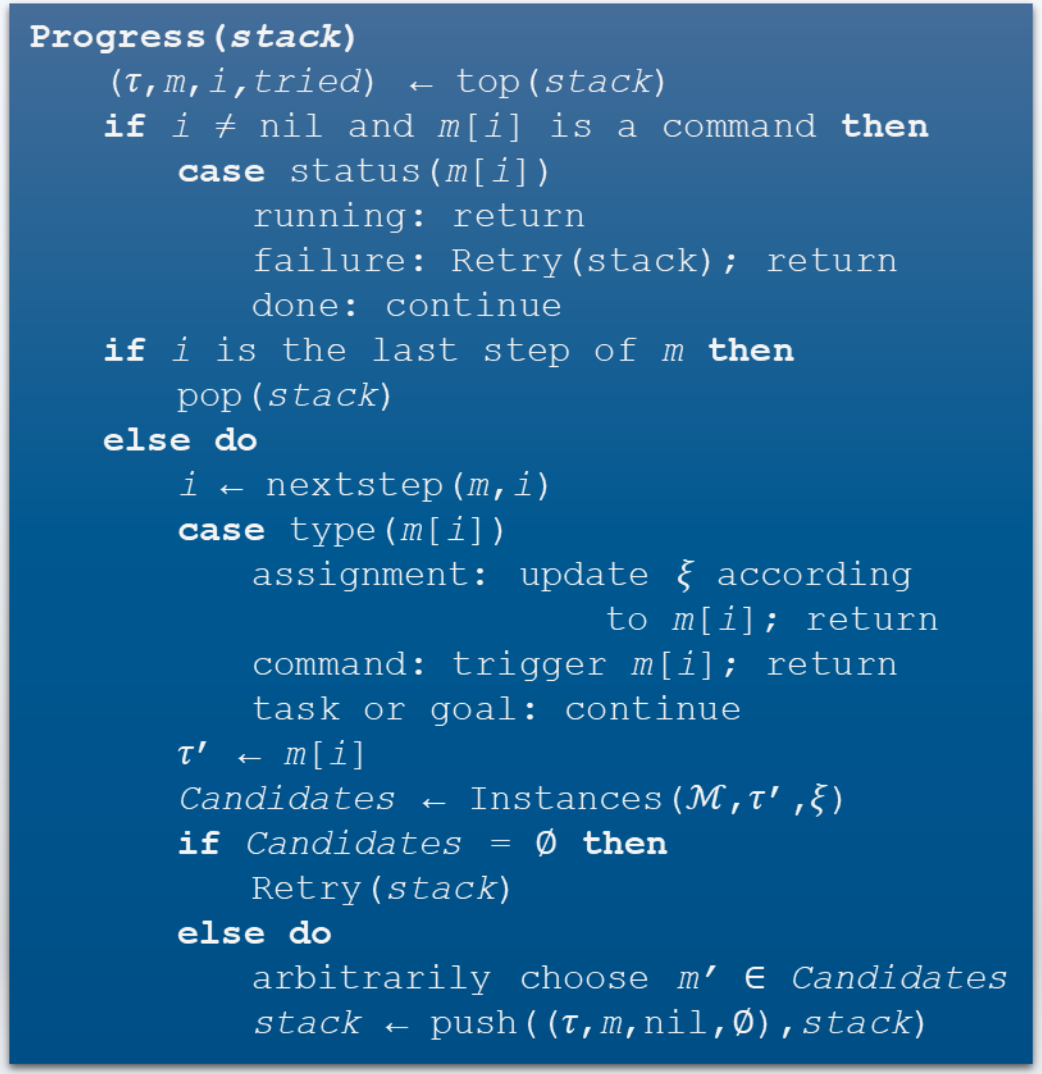
```
Rae($\mathcal{M}$)
    Agenda ← ∅
    loop
        until the input of external tasks and
                events is empty do
            read $\tau$ in the input stream
            Candidates ← Instances($\mathcal{M}$,$\tau$,$\xi$)
            if Candidates = ∅ then
                output(“failed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



```
Stack element($\tau$,$m$,$i$,tried)
    $\tau$          task
    $m$          instance of a method in $\mathcal{M}$
    $i$          instruction pointer to
                     step in body of $m$
    tried      method instances already tried
```

Progress (subroutine) – Just a Decision Tree

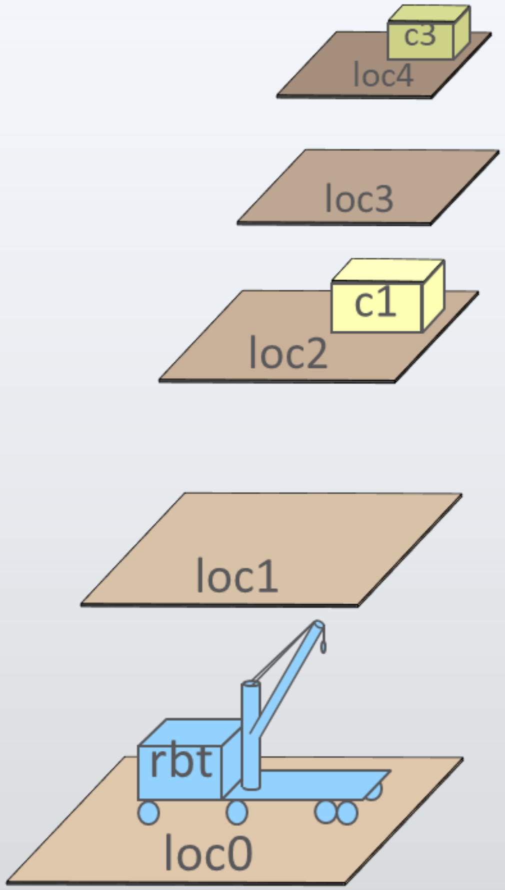




```
Progress(stack)
    (τ,m,i,tried) ← top(stack)
    if i ≠ nil and m[i] is a command then
        case status(m[i])
            running: return
            failure: Retry(stack); return
            done: continue
    if i is the last step of m then
        pop(stack)
    else do
        i ← nextstep(m,i)
        case type(m[i])
            assignment: update ξ according
                             to m[i]; return
            command: trigger m[i]; return
            task or goal: continue
        τ' ← m[i]
        Candidates ← Instances(M,τ',ξ)
        if Candidates = ∅ then
            Retry(stack)
        else do
            arbitrarily choose m' ∈ Candidates
            stack ← push((τ,m,nil,∅),stack)
```

Example

- Objects:
  - $Robots = \{rbt\}$
  - $Containers = \{c1, c2, c3, \ldots\}$
  - $Locations = \{loc0, loc1, loc2, \ldots\}$
- State variables: syntactic terms to which we can assign values
  - $loc(r) \in Locations$
  - $load(r) \in Containers \cup \{nil\}$
  - $pos(c) \in Locations \cup Robots \cup \{unknown\}$
  - $view(r, l) \in \{T, F\}$
    - whether robot r has looked at location l
    - r can only see what is at its current location
- Commands to the execution platform:
  - $take(r, o, l)$: robot r takes object o at location l
  - $put(r, o, l)$: r puts o at location l
  - $perceive(r, l)$: robot r perceives what objects are at loc. l
  - $move{-}to(r, l)$: robot r moves to location l



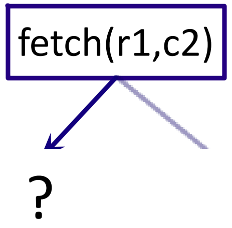
Example

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

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

τ: fetch(r1,c2)
m: ?
i: (see method)
tried: $\emptyset$

Refinement stack



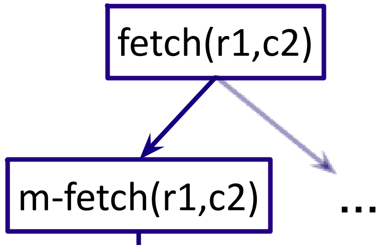
Example

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

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

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

Refinement stack



Example

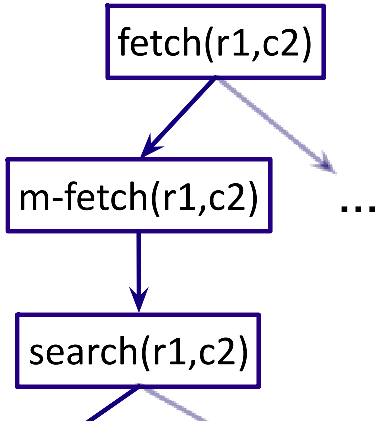
```
m-fetch(r,c)
    task:  fetch(r,c)
    pre:
    body:
        if pos(c) = unknown then
            search(r,c)
        else if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

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

τ: **search(r1,c2)**
m: ?
i: (see method)
tried:∅

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

Refinement stack



Example

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

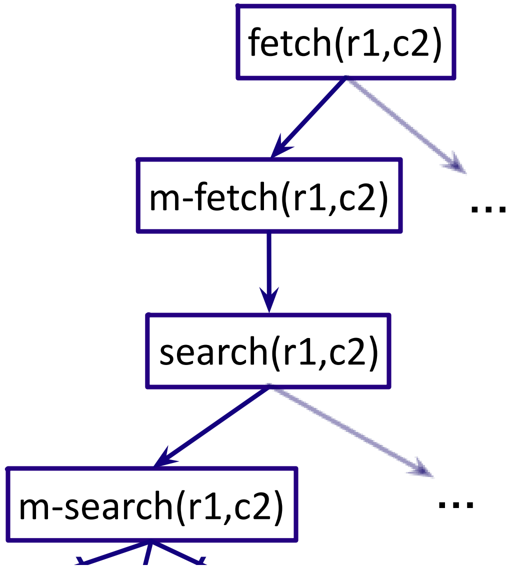
```
m-search(r,c)
    task:   search(r,c)
    pre:    pos(c) = unknown
    body:
        if ∃l (view(r,l) = F) then
            move-to(r,l)
            perceive(l)
            if pos(c) = l then
                take(r,c,l)
            else search(r,c)
        else fail
```

...

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

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

Refinement stack


fetch(r1,c2)
m-fetch(r1,c2)
...
search(r1,c2)
m-search(r1,c2)
...

Example

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

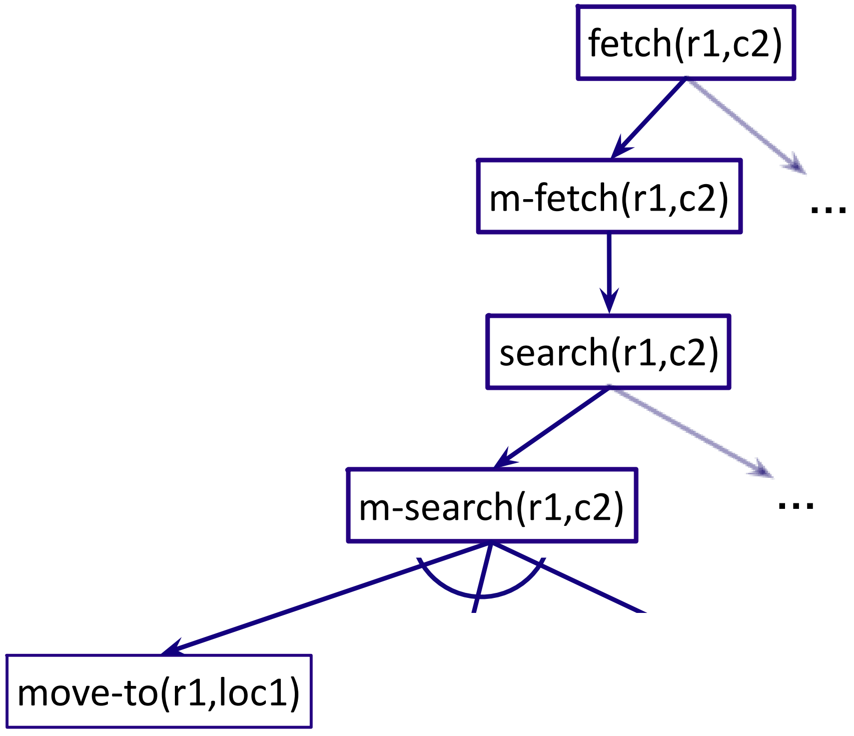
```
m-search(r,c)
    task:   search(r,c)
    pre:    pos(c) = unknown
    body:
        if ∃l (view(r,l) = F) then
            move-to(r,l)
            perceive(l)
            if pos(c) = l then
                take(r,c,l)
            else search(r,c)
        else fail
```

...

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

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

Refinement stack



Example

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

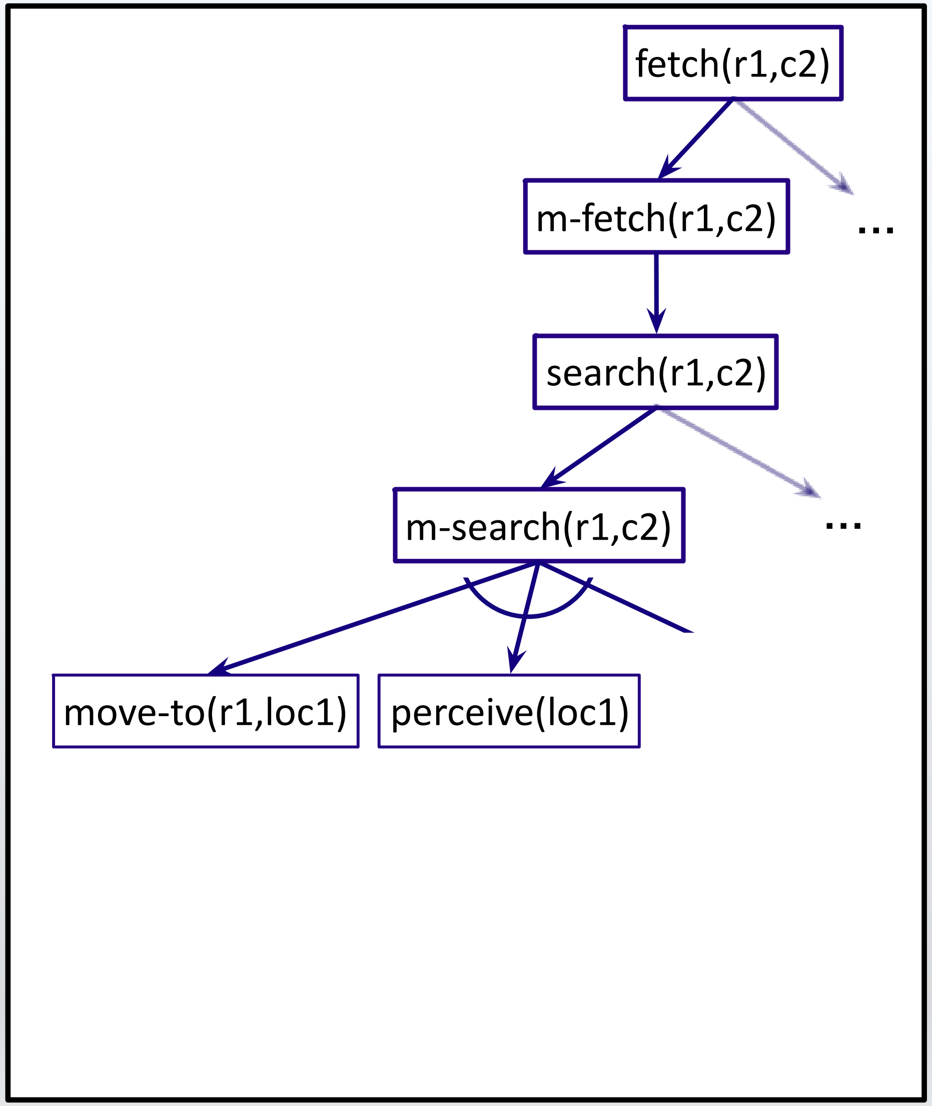
```
m-search(r,c)
    task:   search(r,c)
    pre:    pos(c) = unknown
    body:
        if $\exists l$ (view(r,l) = F) then
            move-to(r,l)
            perceive(l)
            if pos(c) = l then
                take(r,c,l)
            else search(r,c)
        else fail
```

...

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

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

Refinement stack



Example

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

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

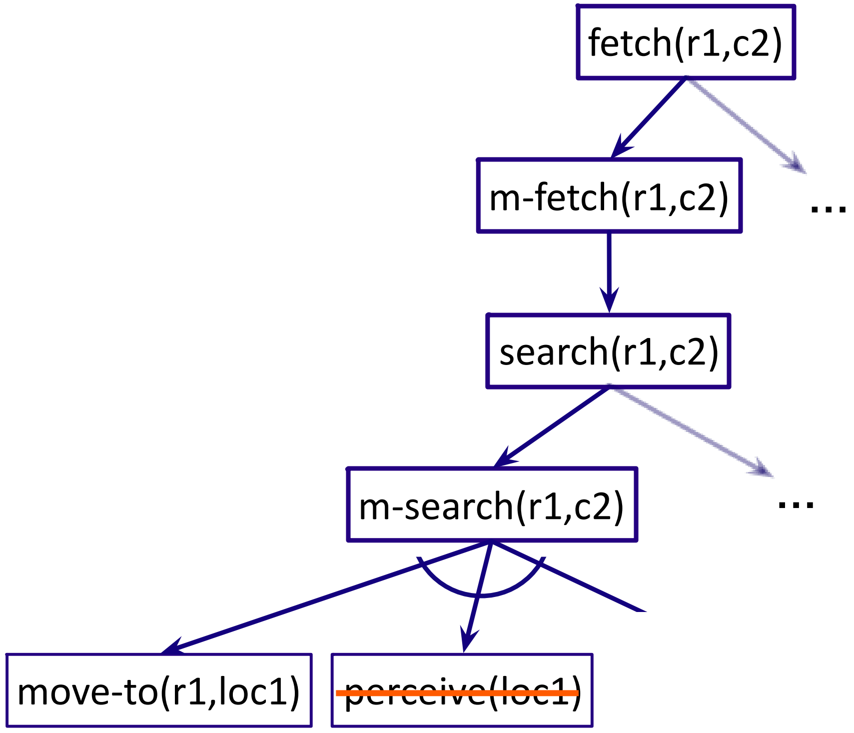
sensor failure

...

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

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

Refinement stack



Example

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

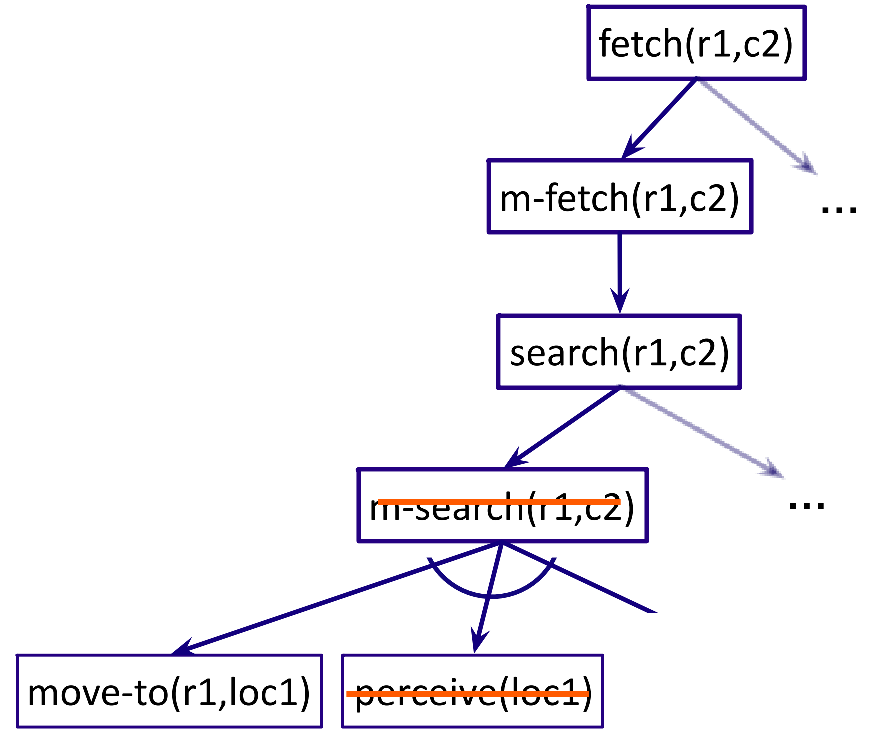
~~m-search(r,c)~~

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

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

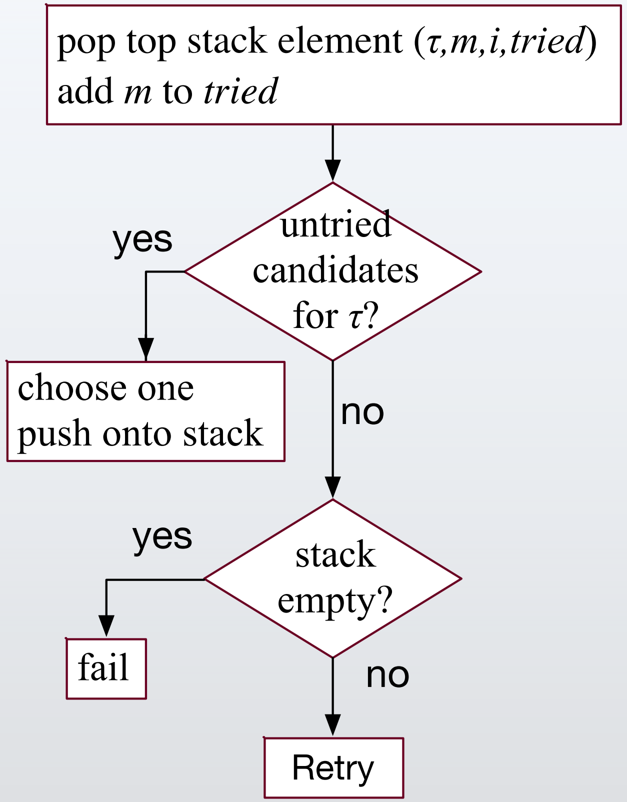
τ: fetch(r1,c2)
m: m-fetch(r1,c2)
i: (see method)
tried:∅

Refinement stack



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

Retry (subroutine)



pop top stack element (τ,*m*,*i*,*tried*)
add *m* to *tried*

untried candidates for τ?

yes

choose one
push onto stack

no

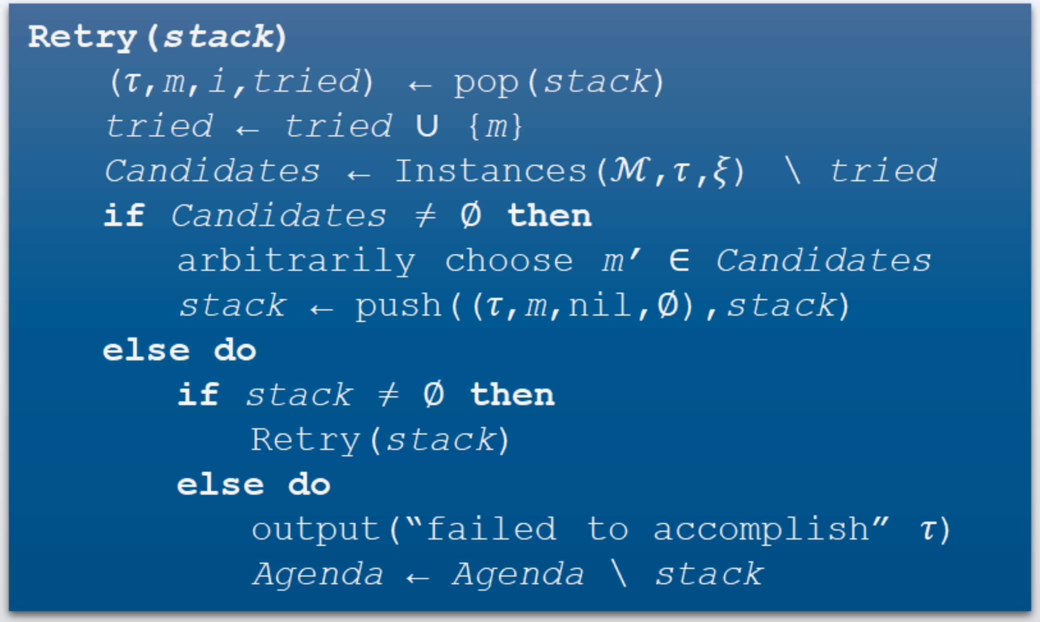
stack empty?

yes

fail

no

Retry



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

Another decision tree

Quiz

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

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

Extensions to RAE

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

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

- Concurrent subtasks
  - Refinement stack for each one

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

- For a task τ, which candidate to try first?
  - Refinement planning

Refinement Planning

Motivation

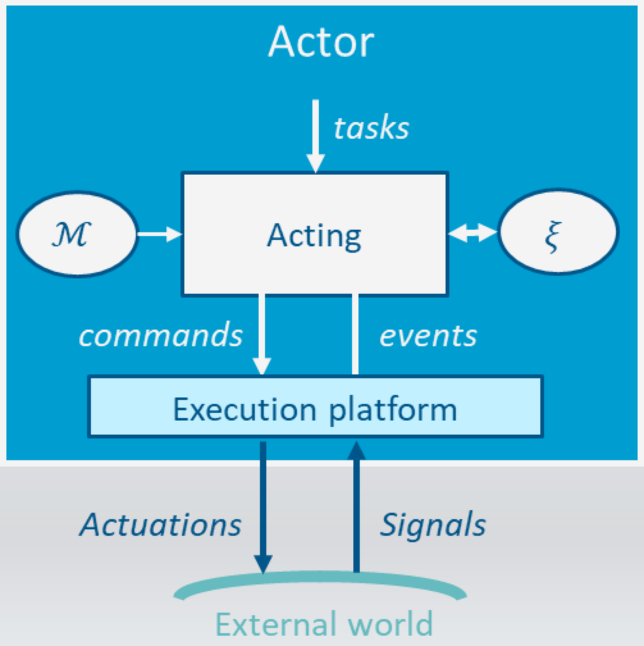
- When dealing with an event or task, Rae may need to make either/or choices
  - *Agenda*: tasks $\tau_1, \tau_2, \dots, \tau_n$
    - Several tasks/events, how to prioritize?
  - Candidates for τ_1: $m_1, m_2, \dots$
    - Several candidate methods or commands, which one to try first?
- Rae immediately executes commands
  - Bad choices may be costly or irreversible

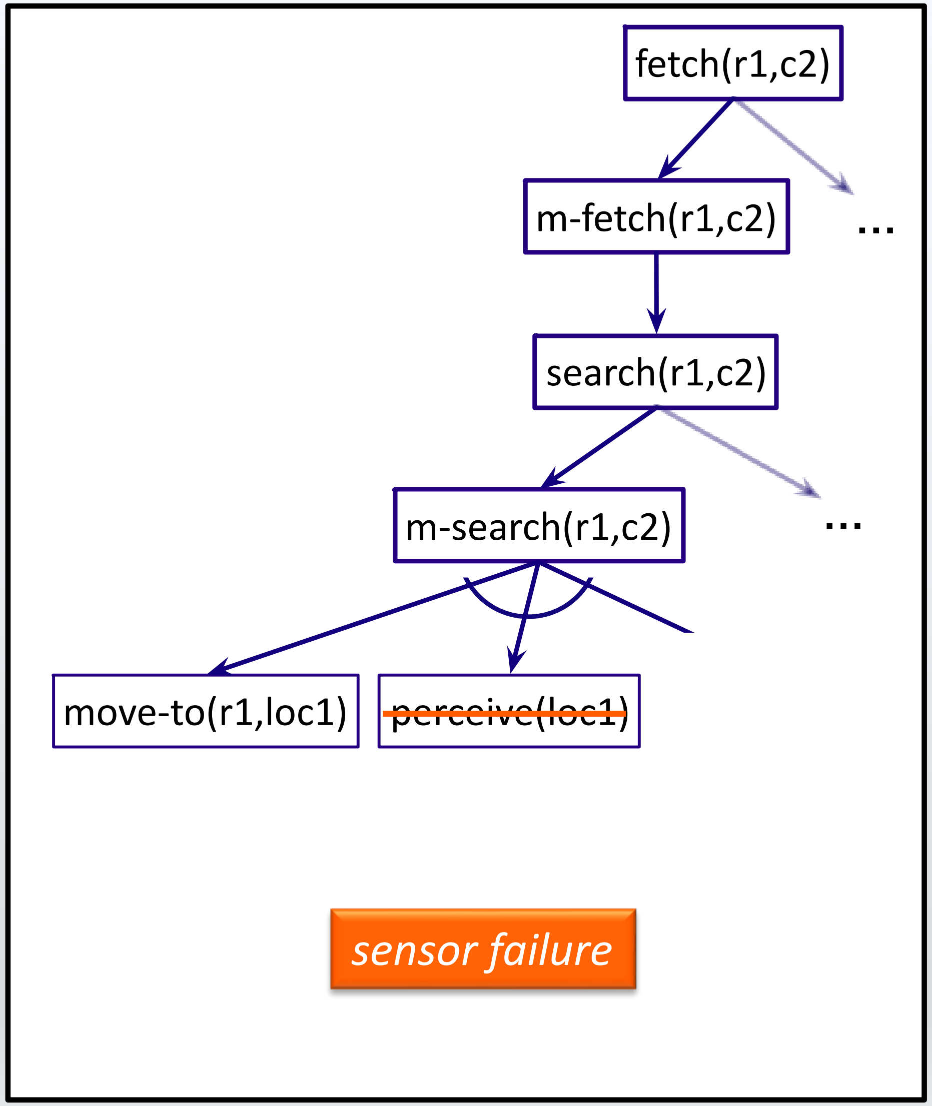
Refinement Planning

- Basic idea:
  - Go step by step through Rae, but do not send commands to execution platform
  - For each command, use a descriptive action model to predict the next state
    - Tells *what*, not *how*
  - Whenever we need to choose a method
    - Try various possible choices, explore consequences, choose best
- Generalization of HTN (Hierarchical Task Network) planning
  - HTN planning: body of a method is a list of tasks
  - Here: body of method is the same program Rae uses
  - Use it to *generate* a list of tasks

Refinement Planning: Example

- Suppose we learn in advance that the sensor isn't available
  - Planner infers that *m-search(r1,c2)* will fail
  - If another method is available, use it
  - Otherwise, planner will infer that the actor can't do *search(r1,c2)*





Descriptive Action Models

- Predict the outcome of performing a command
  - Preconditions-and-effects representation
- Command
  - $take(r, o, l)$: robot r takes object o at location l
  - $put(r, o, l)$: r puts o at location l
  - $perceive(r, l)$: robot r perceives what objects are at location l
    - Can only perceive what is at its current location
    - If we knew this in advance, perception would not be necessary

- Action model

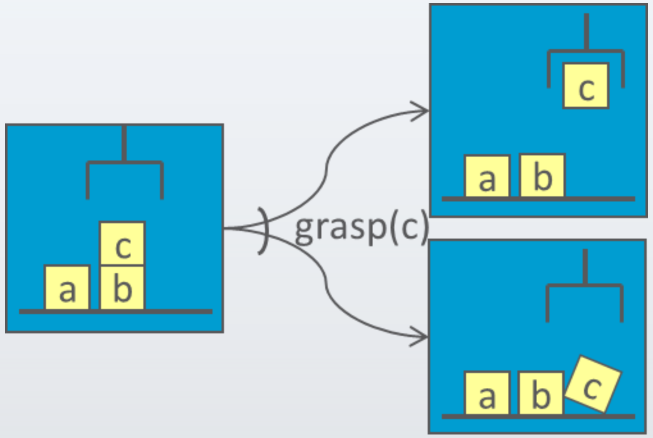
take(r,o,l)
pre: cargo(r) = nil, loc(r) = l, loc(o) = l
eff: cargo(r) ← o, loc(o) ← r

put(r,o,l)
pre: loc(r) = l, loc(o) = r
eff: cargo(r) ← nil, loc(o) ← l

perceive(r,l)
?

Limitation

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



- Why use them?
  - Deterministic models ⇒ much simpler planning algorithms
    - Use when errors are infrequent and do not have severe consequences
    - Actor can fix the errors online

SeRPE (Sequential Refinement Planning Engine)

- SeRPE inputs

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

- Which candidate method for τ?
  - SeRPE:
    - Nondeterministic choice
      - Backtracking point
    - How to implement?
      - Hierarchical adaptation of backtracking, A*, GBFS, ...
  - RAE
    - Arbitrary choice
      - No search, purely reactive

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

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

SeRPE (Sequential Refinement Planning Engine)

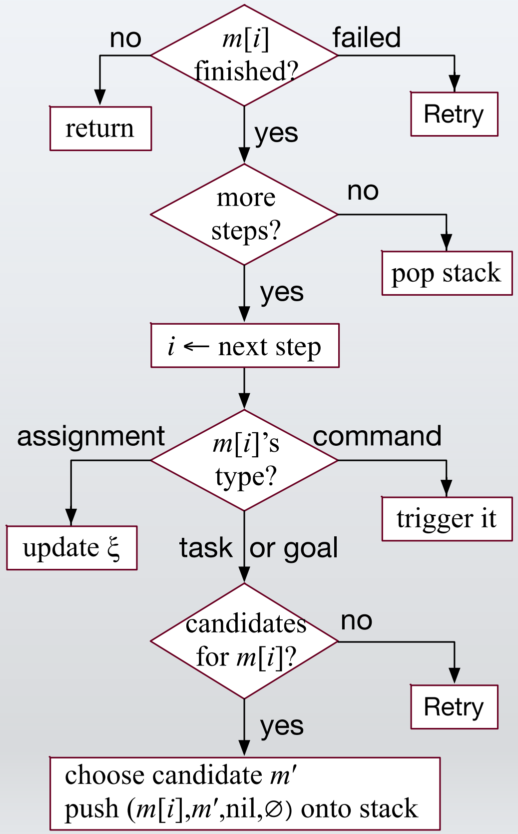
- SeRPE
  - One external task
  - Simulate progressing it all the way to the end
- Rae
  - Several external tasks
  - Each time through loop, progress each one by one step

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

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

Progress-to-finish

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



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

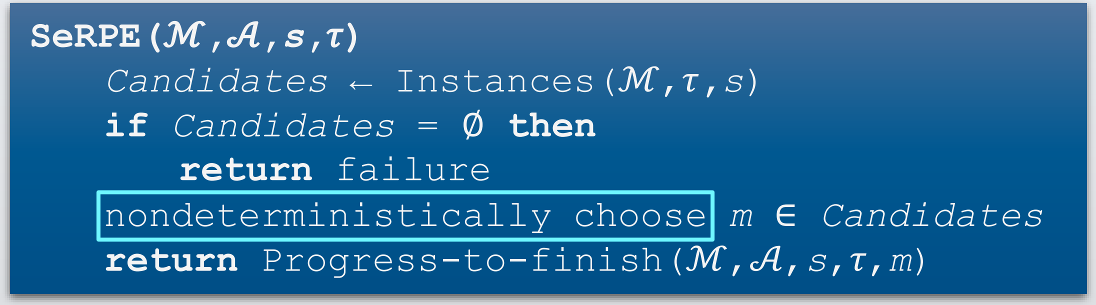
Progress-to-finish

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

- Inputs
  - $\mathcal{M}$ = {methods} , $\mathcal{A}$ = {action models} , s = initial state, τ = task or goal, m = chosen method
- Simulate Rae's goal monitoring
- If $m[i]$ is a command
  - Use action model to predict outcome
- If current step is a task
  - Call SeRPE recursively
  - Recursion stack ≈ Rae's refinement stack
- For failures, no Retry (Rae)
  - A failure means SeRPE could not find a solution
  - Implementation: hierarchical adaptations of backtracking, A*, GBFS, …

Heuristics For SeRPE

- *Ad hoc* approaches:
    - Domain-specific estimates
    - Statistical data on how well each method works
    - Try methods (or actions) in the order that they appear in $\mathcal{M}$ (or $\mathcal{A}$)
- Ideally, would want to implement using heuristic search (e.g., GBFS)
    - What heuristic function? Open problem

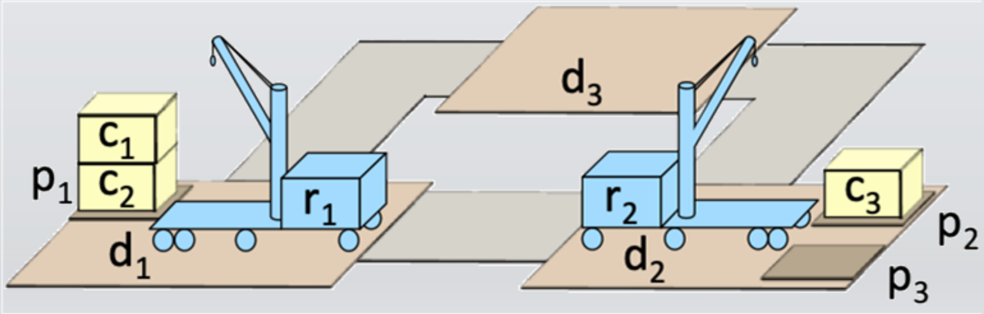


```
SeRPE($\mathcal{M}$,$\mathcal{A}$,$s$,$\tau$)
    $Candidates$ ← Instances($\mathcal{M}$,$\tau$,$s$)
    if $Candidates = \emptyset$ then
        return failure
    nondeterministically choose $m \in Candidates$
    return Progress-to-finish($\mathcal{M}$,$\mathcal{A}$,$s$,$\tau$,$m$)
```

Interleaving

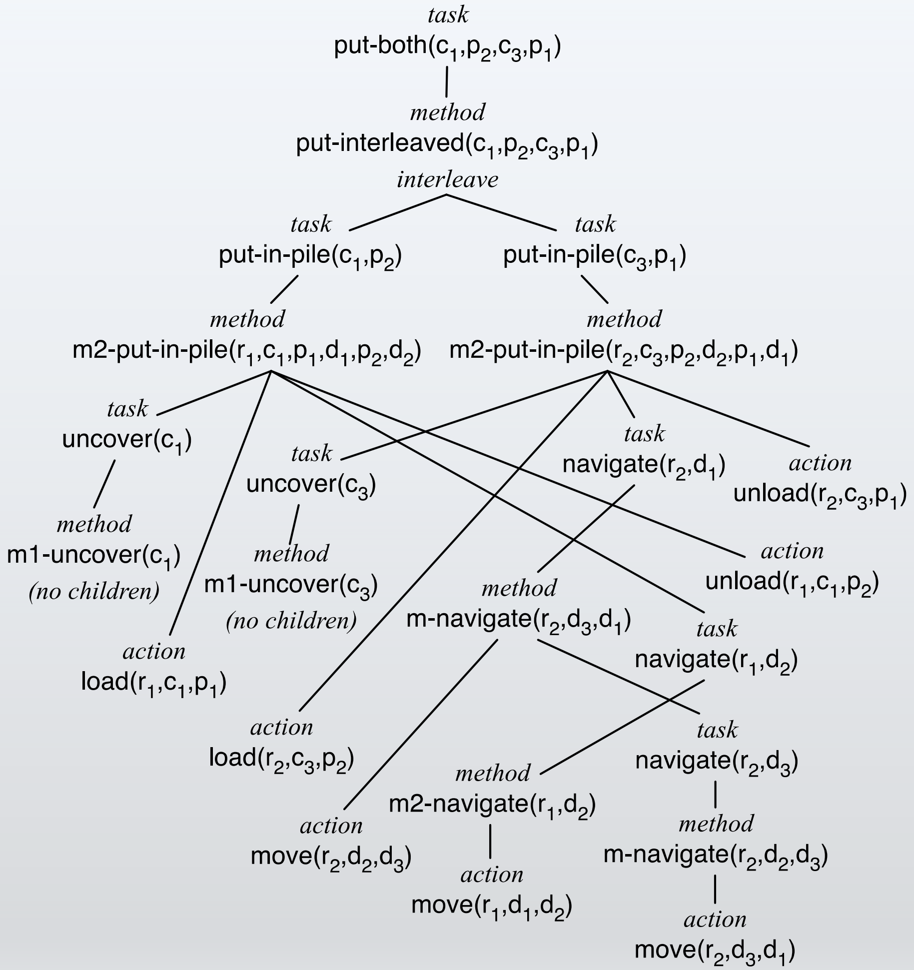
- Want to move c_1 to p_2, using this plan
  - $\langle load(r_1, c_1, c_2, p_1, d_1),$
    $move(r_1, d_1, d_2),$
    $unload(r_1, c_1, p_3, nil, d_2)\rangle$
- … and move c_3 to p_1 using this plan:
  - $\langle load(r_2, c_3, nil, p_2, d_2),$
    $move(r_2, d_2, d_3),$
    $move(r_2, d_3, d_1),$
    $unload(r_2, c_3, c_2, p_1, d_1)\rangle$
- For it to work, must interleave the plans
  - $\langle\ load(r_2, c_3, nil, p_2, d_2),$
    $move(r_2, d_2, d_3),$
    $load(r_1, c_1, c_2, p_1, d_1),$
    $move(r_1, d_1, d_2),$
    $unload(r_1, c_1, p_3, nil, d_2),$
    $move(r_2, d_3, d_1),$
    $unload(r_2, c_3, c_2, p_1, d_1)\rangle$

- $load(r, c, c', p, d)$
  - pre: $at(p, d), cargo(r) = nil, loc(r) = d, pos(c) = c', top(p) = c$
  - eff: $cargo(r) \leftarrow c, pile(c) \leftarrow nil,\ \ pos(c) \leftarrow r, top(p) \leftarrow c'$
- $unload(r, c, c', p, d)$
  - pre: $at(p, d), pos(c) = r, loc(r) = d,\ \ top(p) = c'$
  - eff: $cargo(r) \leftarrow nil, pile(c) \leftarrow p,\ \ pos(c) \leftarrow c', top(p) \leftarrow c$
- $move(r, d, d')$
  - pre: $adj(d, d'), loc(r) = d,\ \ occupied(d') = F$
  - eff: $loc(r) = d', occupied(d) = F, occupied(d') = T$



Interleaved Refinement Tree (IRT) Procedure

- SeRPE doesn't allow the 'concurrent' programming construct
- Partial fix: extend SeRPE to interleave plans for different tasks
- Details: Section 3.3.2



task
put-both(c_1,p_2,c_3,p_1)

method
put-interleaved(c_1,p_2,c_3,p_1)
interleave

task
put-in-pile(c_1,p_2)

task
put-in-pile(c_3,p_1)

method
m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2)

method
m2-put-in-pile(r_2,c_3,p_2,d_2,p_1,d_1)

task
uncover(c_1)

task
uncover(c_3)

task
navigate(r_2,d_1)

action
unload(r_2,c_3,p_1)

method
m1-uncover(c_1)
(no children)

method
m1-uncover(c_3)
(no children)

action
unload(r_1,c_1,p_2)

method
m-navigate(r_2,d_3,d_1)

task
navigate(r_1,d_2)

action
load(r_1,c_1,p_1)

action
load(r_2,c_3,p_2)

task
navigate(r_2,d_3)

method
m2-navigate(r_1,d_2)

action
move(r_2,d_2,d_3)

method
m-navigate(r_2,d_2,d_3)

action
move(r_1,d_1,d_2)

action
move(r_2,d_3,d_1)

Acting and Refinement Planning

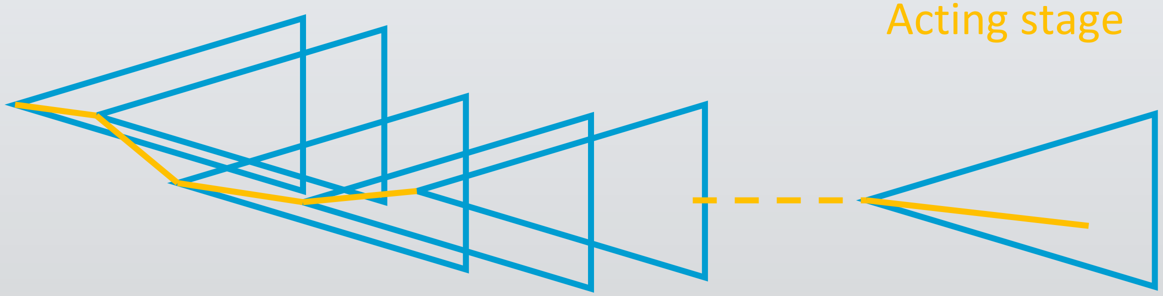
- Hierarchical acting with refinement planning
  - REAP: a RAE-like actor uses SeRPE-like planning at all levels

- Non-hierarchical actor with refinement planning
    - Refine-Lookahead
    - Refine-Lazy-Lookahead
    - Refine-Concurrent-Lookahead
  - Essentially the same as
    - Run-Lookahead
    - Run-Lazy-Lookahead
    - Run-Concurrent-Lookahead
  - But they call SeRPE instead of a classical planner
  - Lookahead same as before
    - Receding horizon, sampling, subgoaling

Using Planning in Acting

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

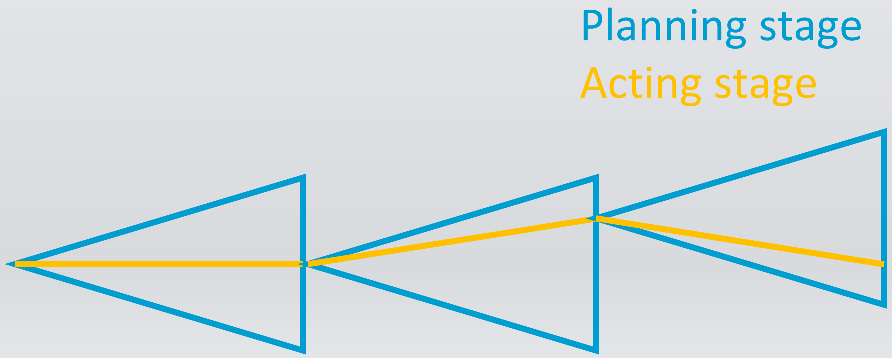
```
Refine-Lookahead($\mathcal{M}$,$\mathcal{A}$,$\tau$)
    while ($s \leftarrow$ abstraction of
            observed state $\xi$) $\nvDash \tau$ do
        $\pi \leftarrow$ SeRPE-Lookahead($\mathcal{M}$,$\mathcal{A}$,$s$,$\tau$)
        if $\pi$ = failure then
            return failure
        $a \leftarrow$ pop-first-action($\pi$)
        perform $a$
```



Using Planning in Acting

- Call Lookahead, execute the plan as far as possible, do not call Lookahead again unless necessary
- Simulate does a simulation of the plan
  - Can be more detailed than SeRPE's action models
    - e.g., physics-based simulation
- Potential problem: may wait too long to re-plan
  - Might not notice problems until it's too late
  - Might miss opportunities to replace π with a better plan

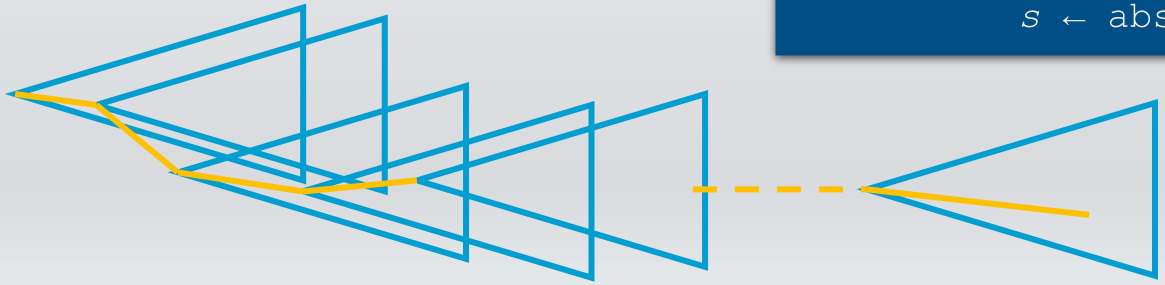
```
**Refine-Lazy-Lookahead**($\mathcal{M}$,$\mathcal{A}$,$\tau$)
    $s$ ← abstraction of
                 observed state $\xi$
    while $s \not\models \tau$ do
        $\pi$ ← SeRPE-Lookahead($\mathcal{M}$,$\mathcal{A}$,$s$,$\tau$)
        if $\pi$ = failure then
            return failure
        while $\pi \neq \langle\rangle$ and $s \not\models \tau$ and
                 Simulate($\Sigma$,$s$,$\tau$,$\pi$)
                     $\neq$ failure do
            $a$ ← pop-first-action($\pi$)
            perform $a$
            $s$ ← abstraction of
                     observed state $\xi$
```



Using Planning in Acting

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

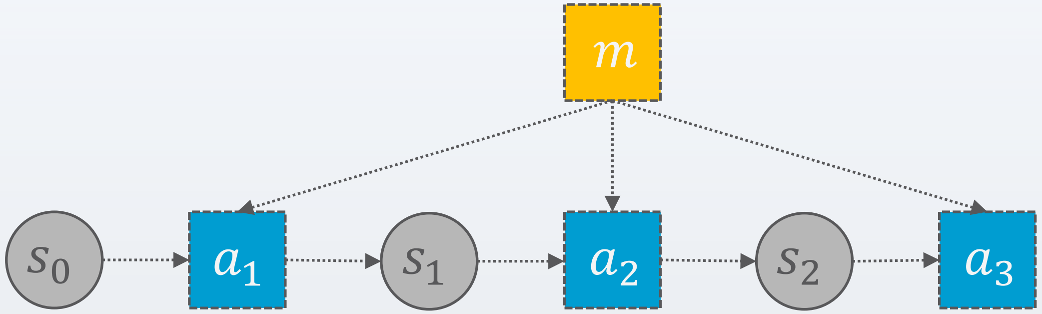
```
Refine-Concurrent-Lookahead($\mathcal{M}$,$\mathcal{A}$,$\tau$)
    $\pi \leftarrow \langle\rangle$
    $s \leftarrow$ abstraction of observed state $\xi$
    // threads 1 and 2 run concurrently
thread 1:
    loop
        $\pi \leftarrow$ SeRPE-Lookahead($\mathcal{M}$,$\mathcal{A}$,$s$,$\tau$)
thread 2:
    loop
        if $s \models \tau$ then
            return success
        else if $\pi$ = failure then
            return failure
        else if $\pi \neq \langle\rangle$ and $s \not\models \tau$ and
                Simulate($\Sigma$,$s$,$\tau$,$\pi$) $\neq$ failure then
            $a \leftarrow$ pop-first-action($\pi$)
            perform $a$
            $s \leftarrow$ abstraction of observed state $\xi$
```



Planning stage

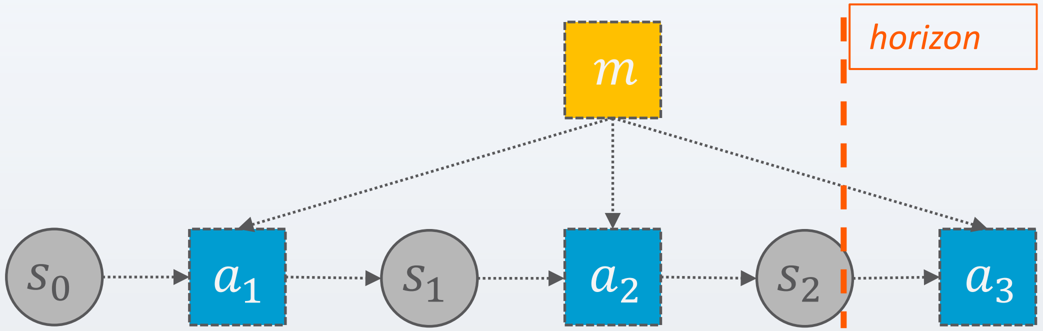
Acting stage

Caveats



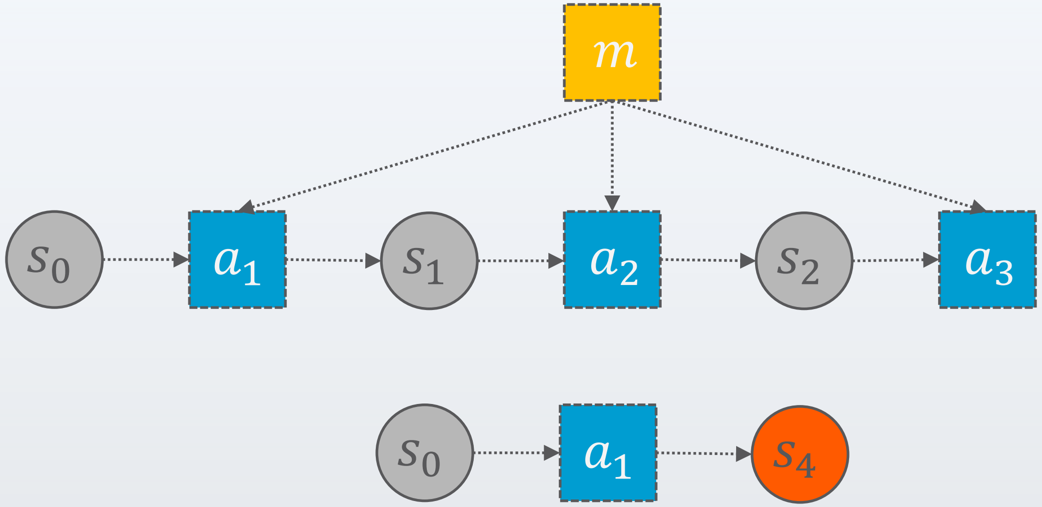
- Start in state s_0,
  want to accomplish task τ
  - Refinement method *m*:
    - task: τ
    - pre: s_0
    - body: a_1, a_2, a_3
- Actor uses Run-Lookahead
  - Lookahead = SeRPE, returns $\langle a_1, a_2, a_3 \rangle$
  - Actor performs a_1, calls Lookahead again
  - No applicable method for τ in s_1, SeRPE returns failure
- Fixes
  - When writing refinement methods, make them general enough to work in different states
  - In some cases, Lookahead might be able to fall back on classical planning until it finds something that matches a method
  - Keep snapshot of SeRPE's search tree at s_1, resume there next time

Caveats



- Start in state s_0,
  want to accomplish task τ
  - Refinement method *m*:
    - task: τ
    - pre: s_0
    - body: a_1, a_2, a_3
- Actor uses Run-Lazy-Lookahead
  - Lookahead = SeRPE with receding horizon, returns $\langle a_1, a_2 \rangle$
  - Actor performs them, calls Lookahead again
  - No applicable method for τ in s_2, SeRPE returns failure
- Can use the same fixes on previous slide, with one modification
  - Keep snapshot of SeRPE's search tree at the horizon, resume next time it is called

Caveats



- Start in state s_0,
  want to accomplish task τ
  - Refinement method *m*:
    - task: τ
    - pre: s_0
    - body: a_1, a_2, a_3
- Actor uses Run-Lazy-Lookahead
  - Lookahead = SeRPE, returns $\langle a_1, a_2, a_3 \rangle$
  - While acting, unexpected event
  - Actor calls Lookahead again
  - No applicable method for τ in s_4, SeRPE returns failure
- Can use most of the fixes on last two slides, with this modification
  - Keep snapshot of SeRPE's search tree after each action
    - Restart it immediately after a_1, using s_4 as current state
  - Also: make recovery methods for unexpected states
    - E.g., fix flat tire, get back on the road

Summary

- Acting and planning
  - Lookahead: search part of the search space, return a partial solution
  - Refine-Lookahead, Refine-Lazy-Lookahead, Refine-Concurrent-Lookahead
    - Like Run-Lookahead, Run-Lazy-Lookahead, Run-Concurrent-Lookahead, but call SeRPE
  - Caveats
    - Current state may not be what we expect
    - Possible ways to handle that

Content

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