Advanced Topics Data Science and AI
Automated Planning and Acting

Refinement Methods

Tanya Braun

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

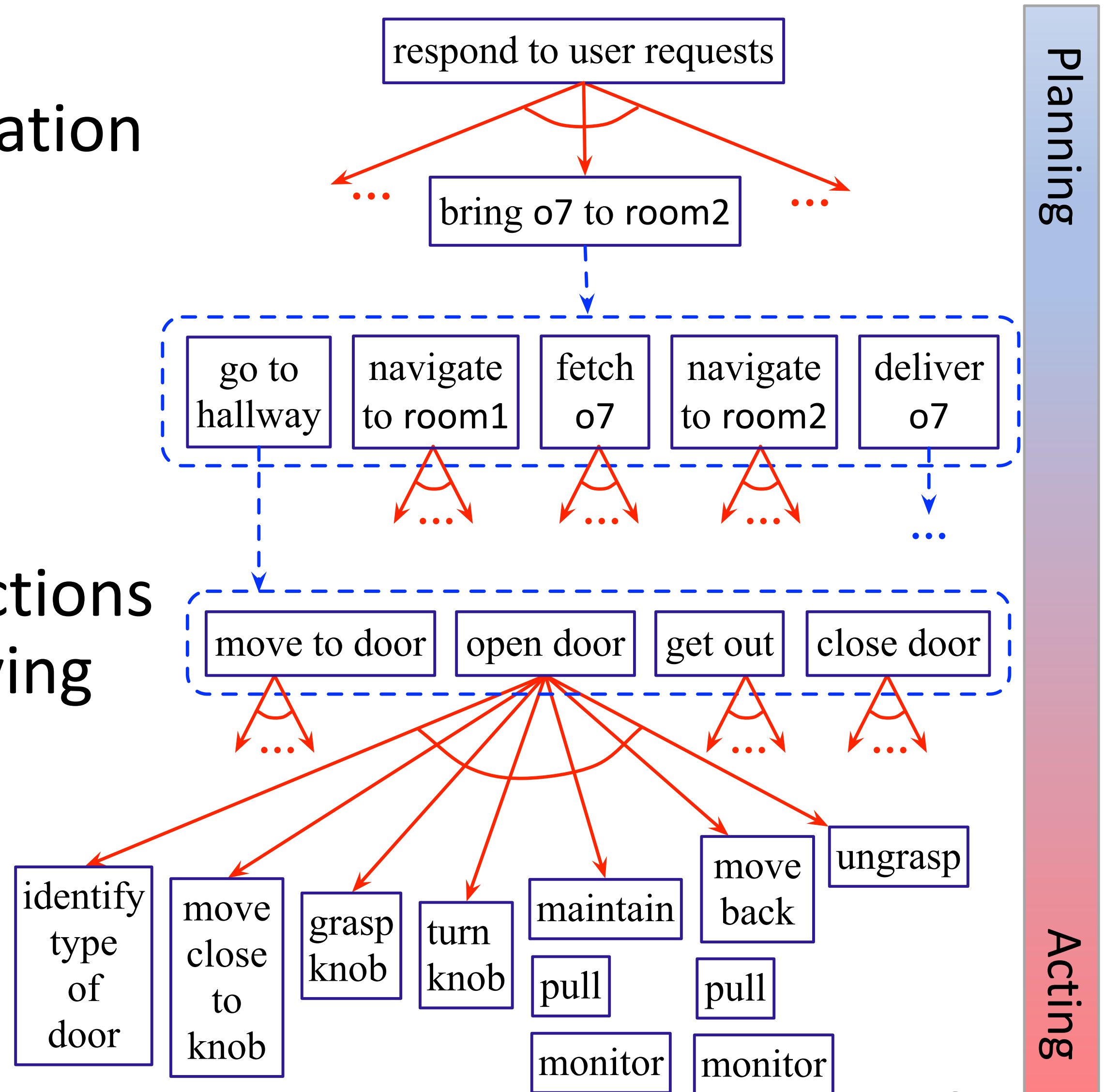
Content

1. Planning and Acting with **Deterministic** Models
2. Planning and Acting with **Refinement** Methods
   a. Operational Models
   b. Refinement-Acting Machine
   c. Refinement Planning
   d. Acting and Refinement Planning
3. Planning and Acting with **Temporal** Models
4. Planning and Acting with **Nondeterministic** Models
5. Making Simple Decisions
6. Making Complex Decisions
7. Planning and Acting with **Probabilistic** Models
8. Provably Beneficial AI

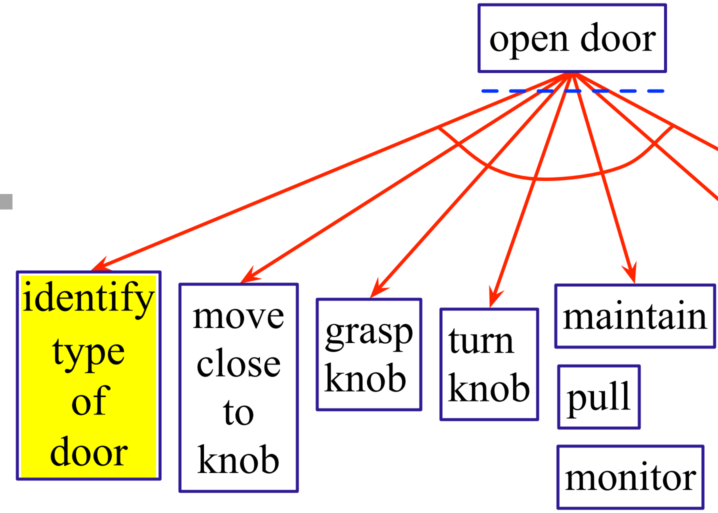
- Other: open world, perceiving, learning
  - If time permits

Motivation

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



Opening a Door



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

Assumptions

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

Outline per the Book

3.1 *Representation*
- State variables, commands, refinement methods
- Example

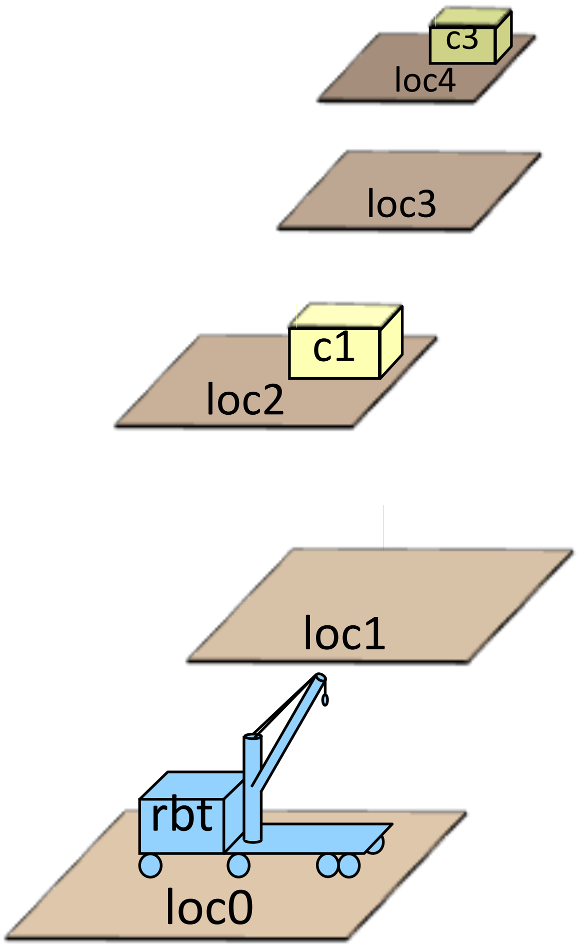
3.2 *Acting*
- Rae (Refinement Acting Engine)
- Example
- Extensions

3.3 *Planning*
- Motivation and basic ideas
- Deterministic action models
- SeRPE (Sequential Refinement Planning Engine)

3.4 *Using Planning in Acting*
- Techniques
- Caveats

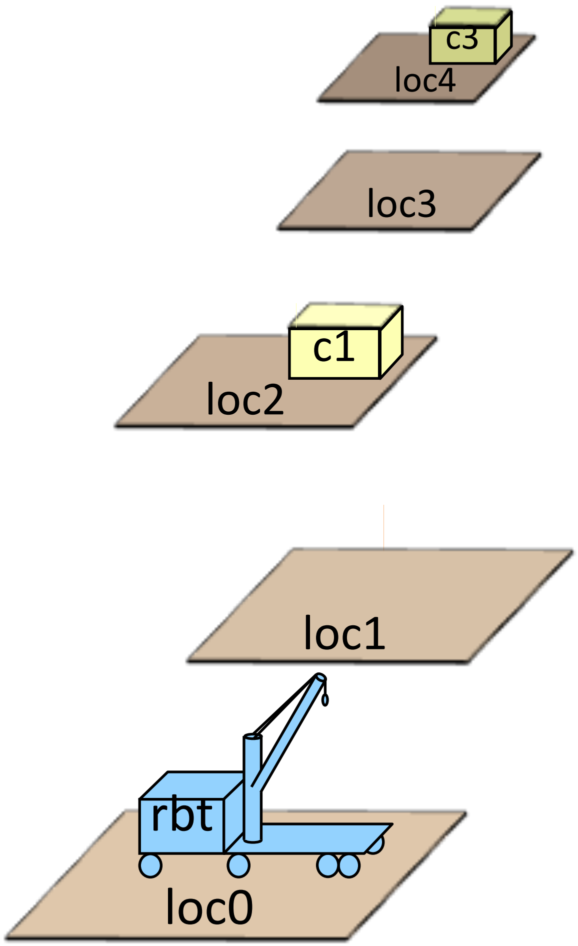
State-variable Representation (Recap)

- 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
  - $\{loc(rbt) = loc0, pos(c1) = loc2,$
    $pos(c3) = loc4, pos(c2) = \ unknown, \dots\}$



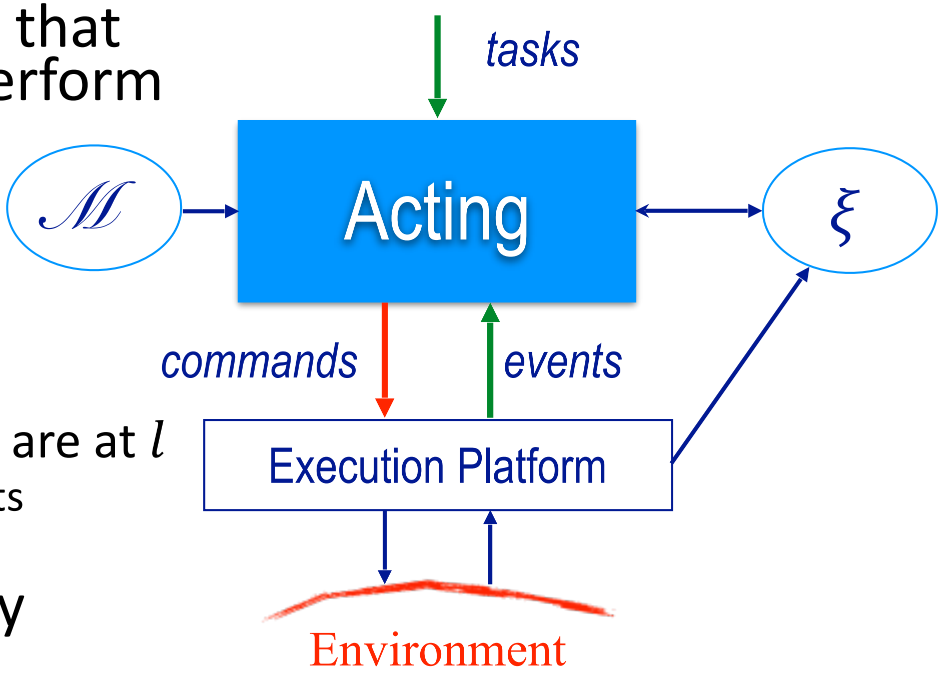
State-variable Representation: Extensions

- 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



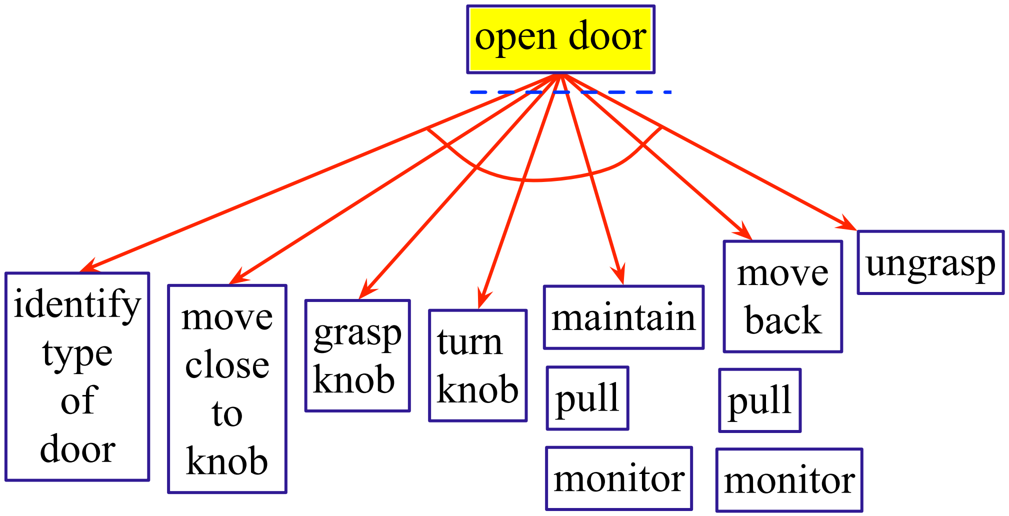
Commands

- 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
  - $event\text{–}name(args)$
  - Exogenous changes in the environment to which the actor may have to react
  - E.g., emergency signal, arrival of transportation vehicle



Tasks and Methods

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

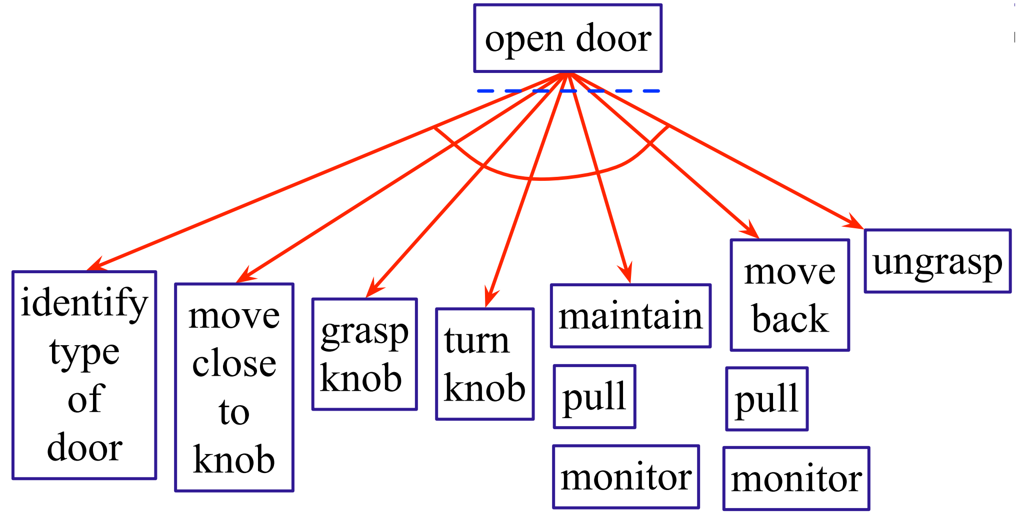


$method\text{–}name(arg_1, \ldots, arg_k)$
task: *task-identifier*
pre: *test*
body: *a program*

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

Opening a Door

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



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

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-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)

Intermediate Summary

- 3.1 Operational models
  - Tasks, events
  - Commands to the execution platform
  - Extensions to state-variable representation
  - Refinement method: name, task/event, preconditions, body
  - Example: opening a door

Outline per the Book

3.1 *Representation*

- State variables, commands, refinement methods
- Example

3.2 *Acting*

- Rae (Refinement Acting Engine)
- Example
- Extensions

3.3 *Planning*

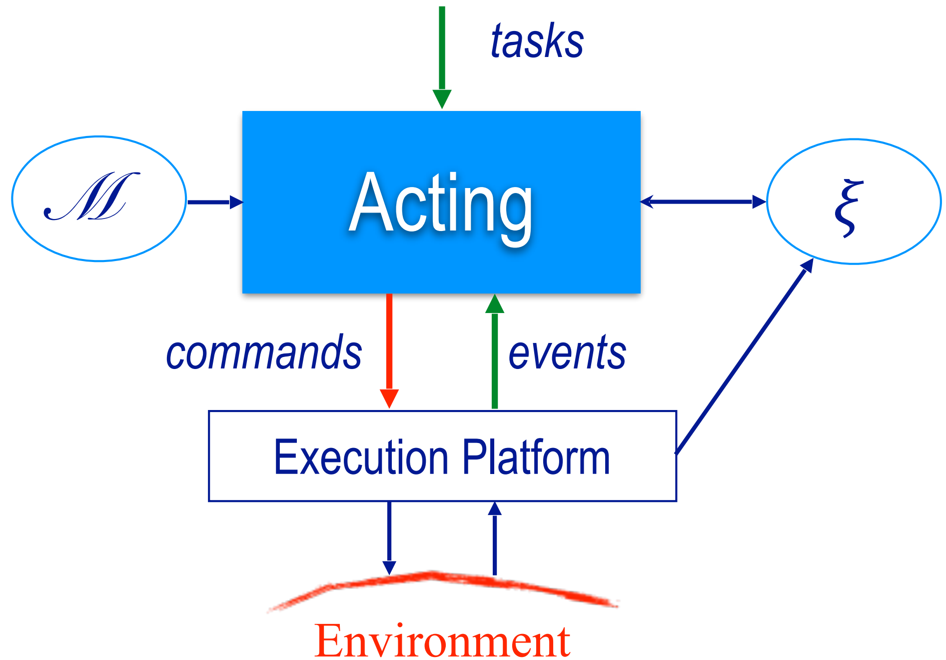
- Motivation and basic ideas
- Deterministic action models
- SeRPE (Sequential Refinement Planning Engine)

3.4 *Using Planning in Acting*

- Techniques
- Caveats

Rae (Refinement Acting Engine)

- Based on OpenPRS
  - Programming language, open-source robotics software
  - Deployed in many applications
- Input
  - External tasks, events, current state, 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 (Refinement Acting Engine)

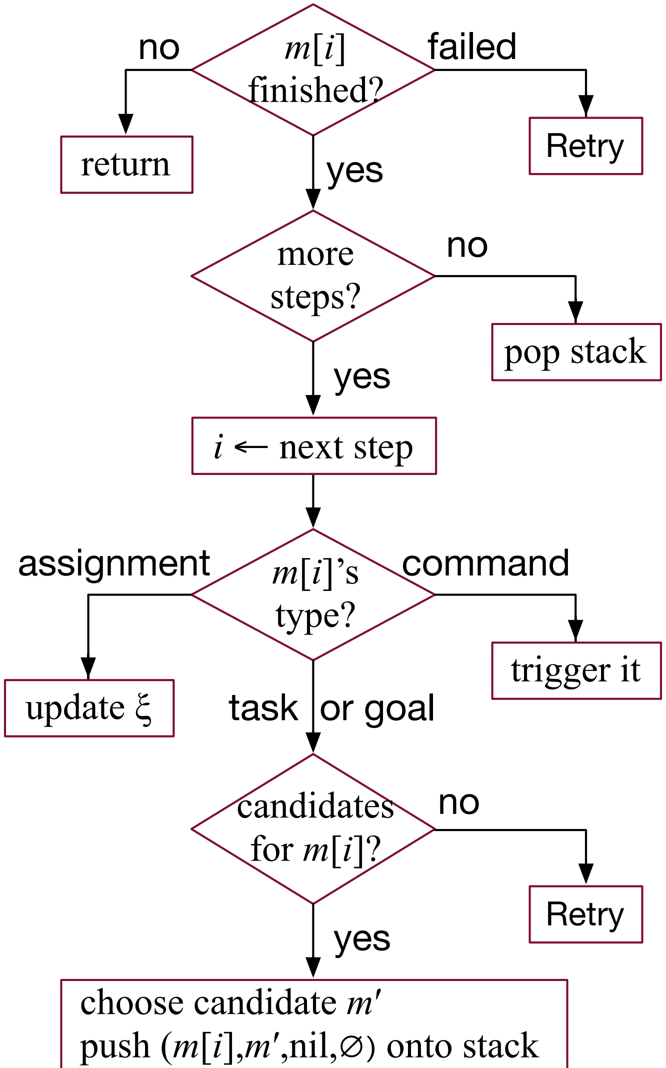
- Basic idea

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

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

```
Stack element(τ,m,i,tried)
    τ           task
    m           instance of a method in ℳ
    i           instruction pointer to
                         step in body of m
    tried       method instances already tried
```

Progress (subroutine)

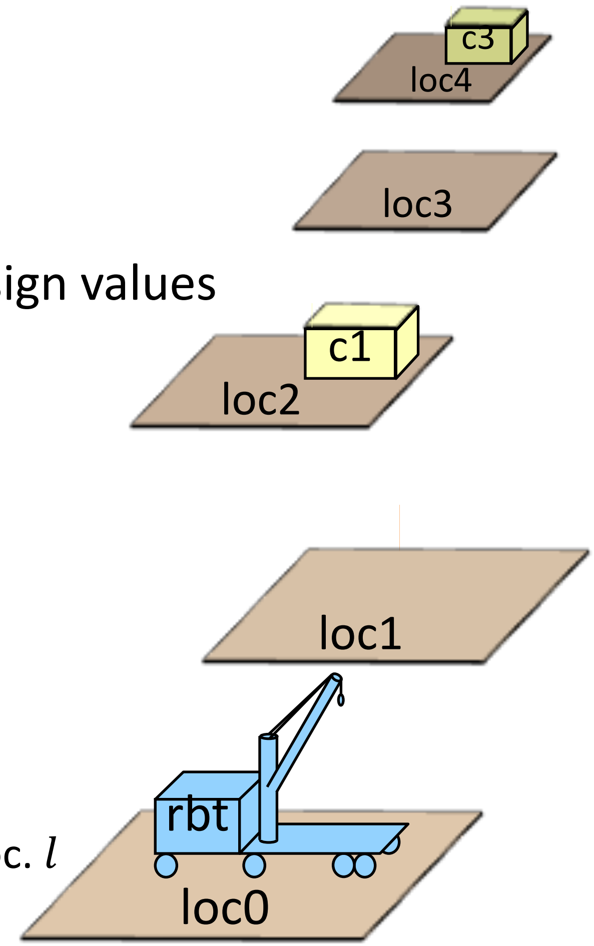



```
Progress(stack)
    (τ,m,i,tried) ← top(stack)
    if i ≠ nil and m[i] is a command then
        case status(m[i])
            running: return
            failure: Retry(stack); return
            done: continue
    if i is the last step of m then
        pop(stack)
    else do
        i ← nextstep(m,i)
        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)
```

Just a decision tree

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 ∃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: ∅

Refinement stack

fetch(r1,c2)

?

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

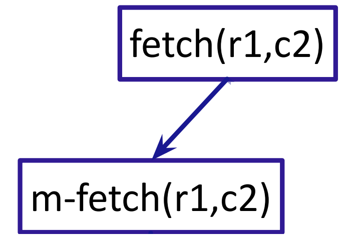
Example

```
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

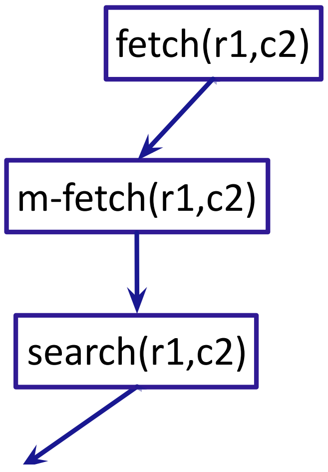
```
m-fetch(r,c)
    task: fetch(r,c)
    pre:
    body:
        if pos(c) = unknown then
            search(r,c)
        else if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

```
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

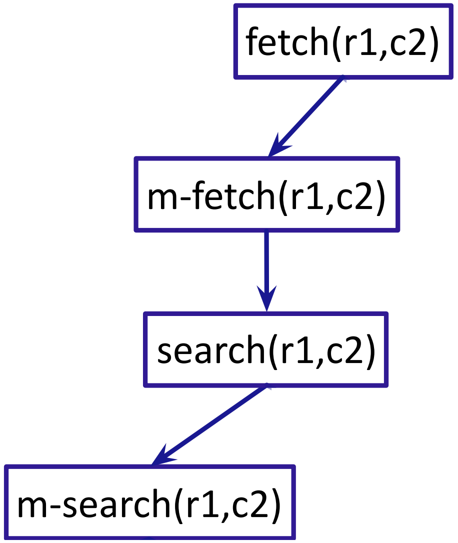
```
m-fetch(r,c)
    task: fetch(r,c)
    pre:
    body:
        if pos(c) = unknown then
            search(r,c)
        else if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

```
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



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))
```

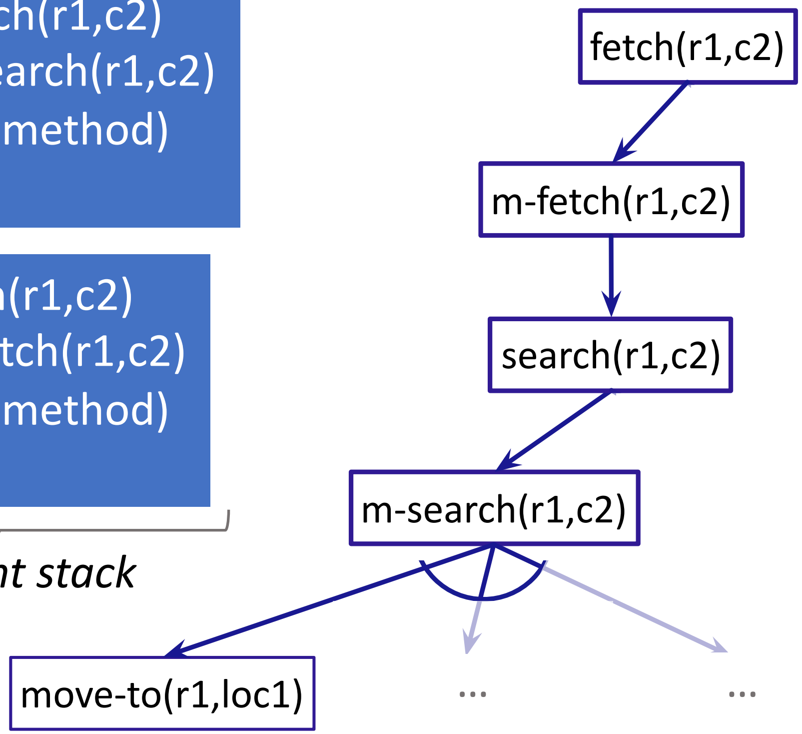
```
m-search(r,c)
    task:   search(r,c)
    pre:    pos(c) = unknown
    body:
        if ∃l (view(r,l) = F) then
            move-to(r,l)
            perceive(l)
            if pos(c) = l then
                take(r,c,l)
            else search(r,c)
        else fail
```

...

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



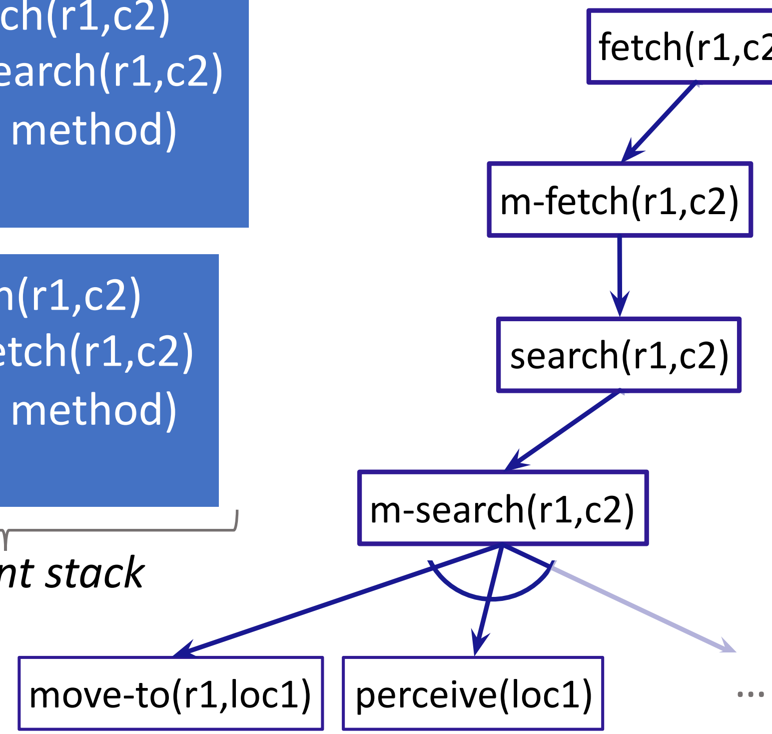
Example

m-fetch(*r*,*c*)
    task: fetch(*r*,*c*)
    pre:
    body:
        if pos(*c*) = unknown then
            $\boxed{\text{search}(r,c)}$
        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*)
            $\boxed{\text{perceive}(l)}$
            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
```

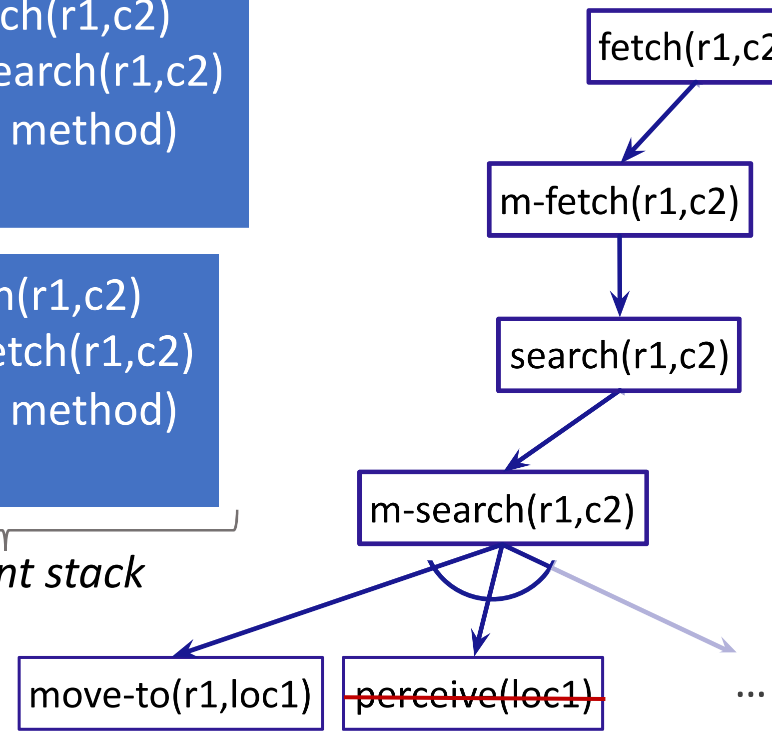
sensor failure

...

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

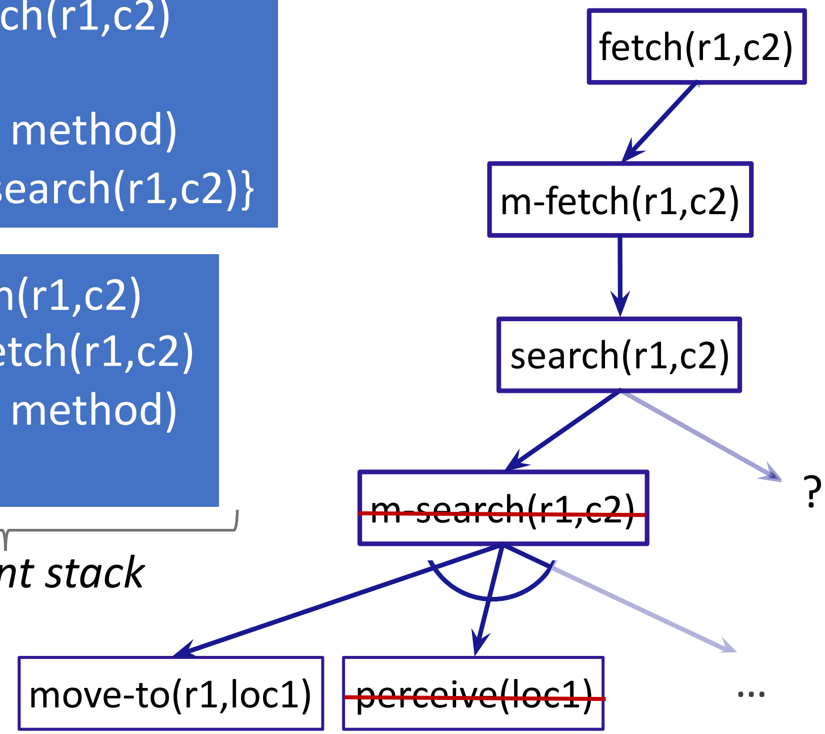
```
m-fetch(r,c)
    task:  fetch(r,c)
    pre:
    body:
        if pos(c) = unknown then
            search(r,c)
        else if loc(r) = pos(c) then
            take(r,c,pos(c))
        else do
            move-to(r,pos(c))
            take(r,c,pos(c))
```

~~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)}

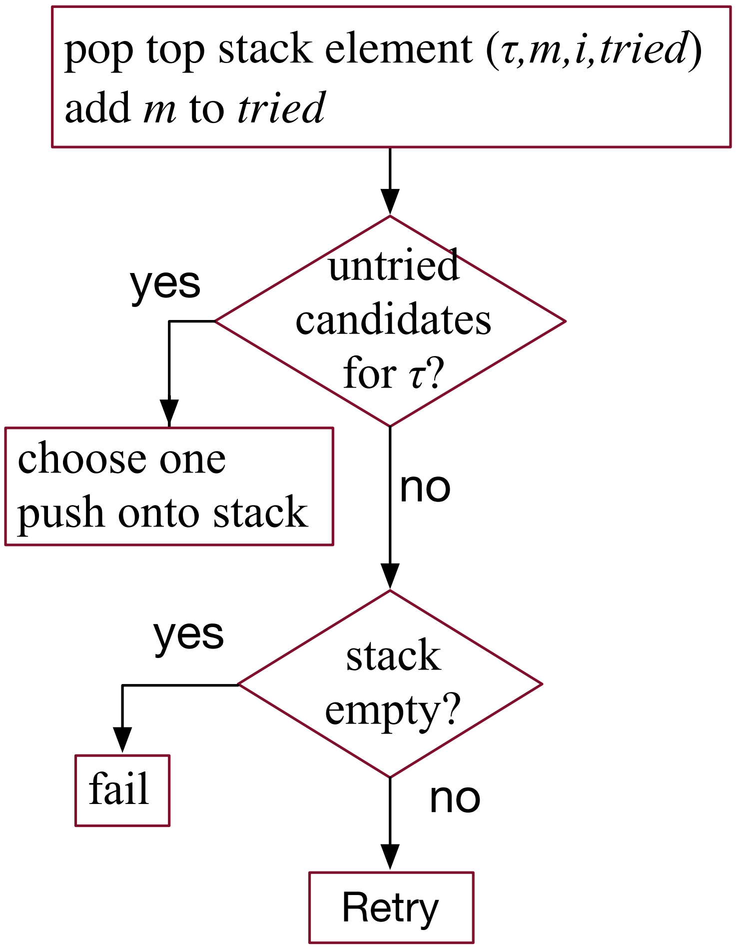
τ:	fetch(r1,c2)
m:	m-fetch(r1,c2)
i:	(see method)
tried:	∅

Refinement stack



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

Retry (subroutine)




```
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

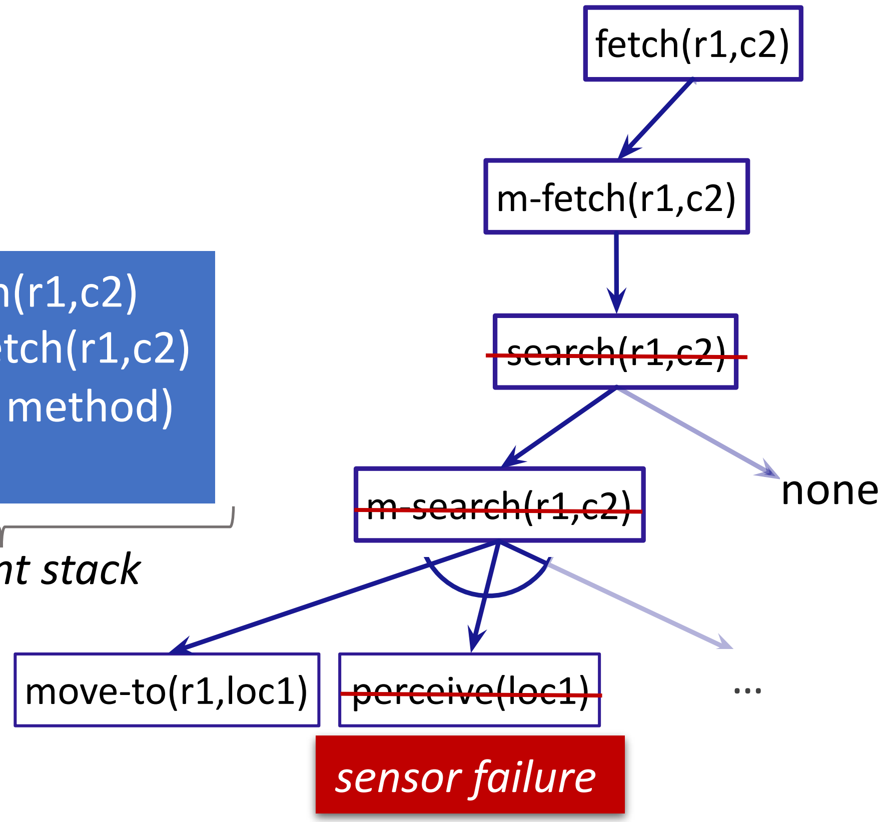
Example

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



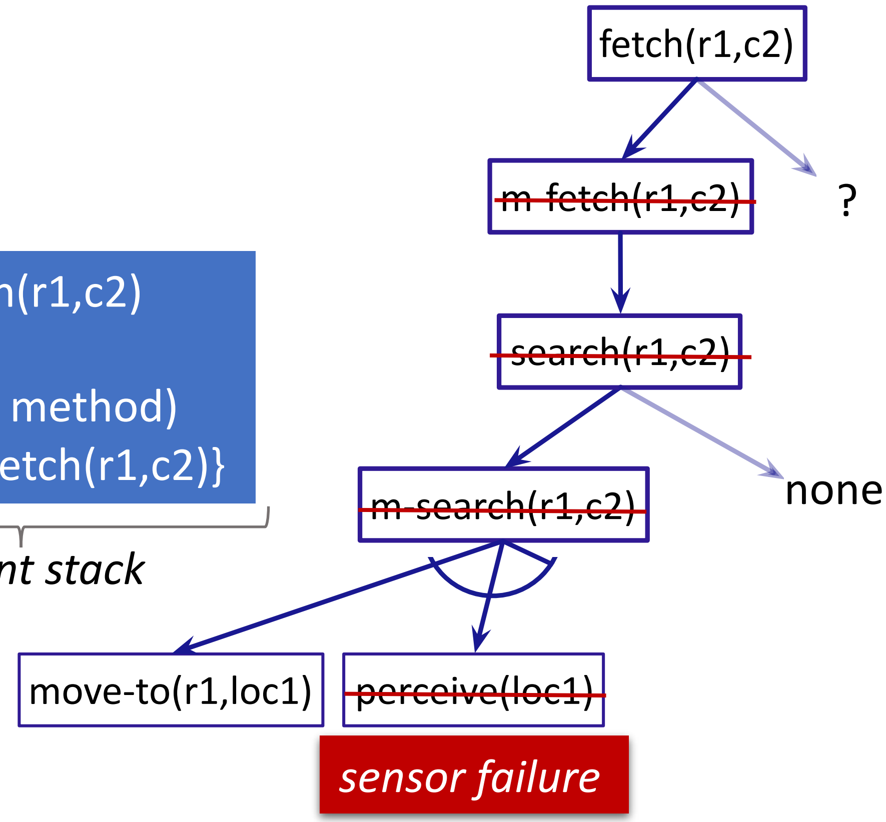
Example

```
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: fetch(r1,c2)
m: ?
i: (see method)
tried:{m-fetch(r1,c2)}

Refinement stack



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

Extensions to Rae: Events

- Events:

$method\text{–}name(arg_1, \ldots, arg_k)$
  event: *event-identifier*
  pre: *test*
  body: *a program*

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

m-emergency(*r*,*l*,*i*)
  event: emergency(*l*,*i*)
  pre: emergency-handling(*r*) = F
  body: emergency-handling(*r*) ← T
    if load(*r*) ≠ nil then
      put(*r*,load(*r*))
    move-to(*l*)
    address-emergency(*l*,*i*)

- Example: an emergency
  - If r is not already handling another emergency, then
    - stop what it is doing
    - go handle the emergency

Extensions to Rae: Goals

- Write as a special kind of task
  - Like other tasks, but includes monitoring (modify Progress)

$method\text{-}name(arg_1, \ldots, arg_k)$
task: achieve(*condition*)
pre: *test*
body: *a program*

m-fetch(*r*,*c*)
 task: fetch(*r*,*c*)
 pre:
 body:
  achieve(pos(*c*)≠unknown)
  move-to(*r*,pos(*c*))
  take(*r*,*c*,pos(*c*))
m-find-where(*r*,*c*)
 goal: achieve(pos (*c*)≠unknown)
 pre:
 body:
  while there is a loc. *l* s.t. view(*l*) = F do
   move-to(*l*)
   preceive(*l*)

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

- if *condition* becomes true before finishing body(*m*), stop early
- if *condition* isn't true after finishing body(*m*),
  fail and try another method

Other Extensions to Rae

- Concurrent subtasks:
  - Refinement stack for each one

body of a method
...
{concurrent: $\tau_1, \tau_2, \ldots, \tau_n$}
...

- Controlling the progress of tasks:
  - E.g., suspend a task for a while
  - If there are multiple stacks, which ones get higher priority?
    - Application-specific heuristics

$Agenda = \{stack_1, stack_2, \ldots, stack_n\}$

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

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

Intermediate Summary

- 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, Agenda* \ *stack*
  - Refinement trees
  - Concurrent tasks: for each, a refinement stack
  - Goal: achieve(condition), uses monitoring
  - Controlling progress, heuristics

Outline per the Book

3.1 *Representation*

- State variables, commands, refinement methods
- Example

3.2 *Acting*

- Rae (Refinement Acting Engine)
- Example
- Extensions

3.3 *Planning*

- Motivation and basic ideas
- Deterministic action models
- SeRPE (Sequential Refinement Planning Engine)

3.4 *Using Planning in Acting*

- Techniques
- Caveats

Motivation

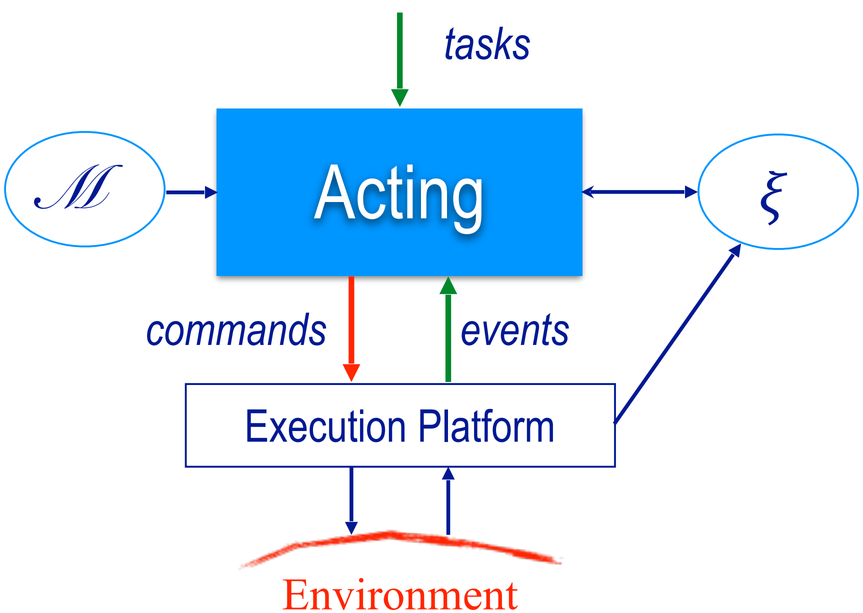
- When dealing with an event or task, Rae may need to make either/or choices
  - *Agenda*: tasks $\tau_1, \tau_2, \ldots, \tau_n$
    - Several tasks/events, how to prioritize?
  - Candidates for τ_1: $m_1, m_2, \ldots$
    - Several candidate methods or commands, which one to try first?
- Rae immediately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

Refinement Planning

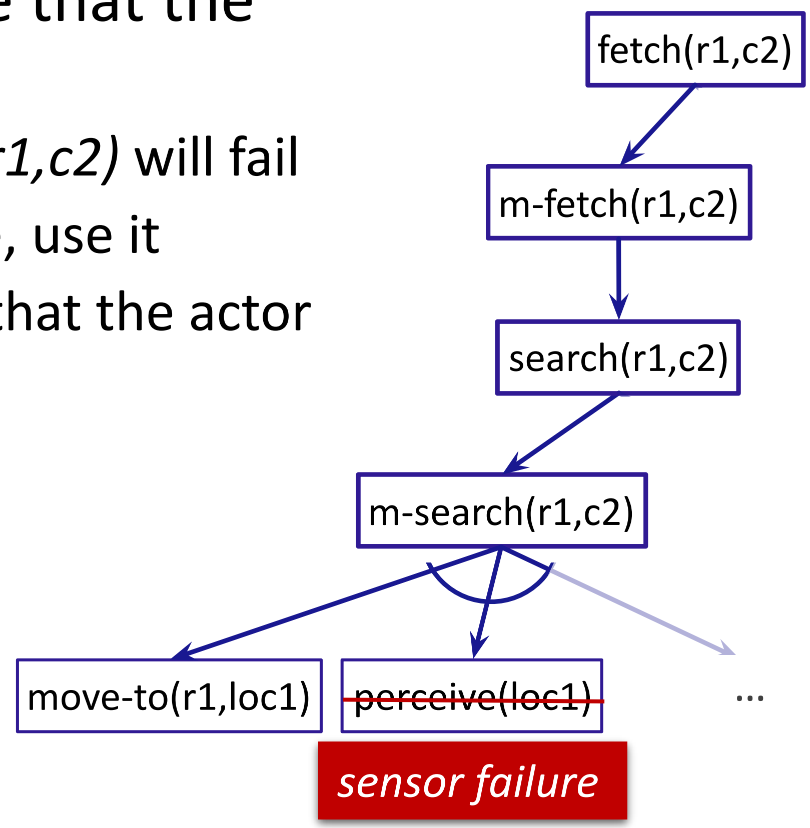
- Basic idea:
  - Go step by step through Rae, but do not send commands to execution platform
  - For each command, use a descriptive action model to predict the next state
    - Tells *what*, not *how*
  - Whenever we need to choose a method
    - Try various possible choices, explore consequences, choose best
- Generalization of HTN 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)*



Search tree:



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

- Action model

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

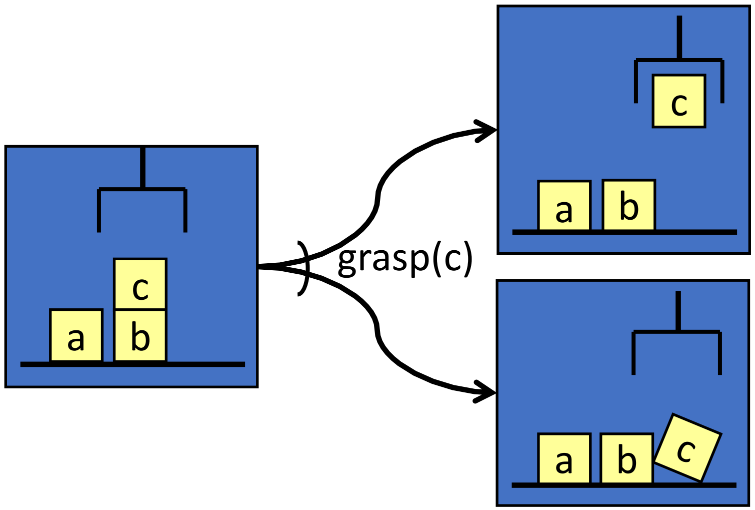
put(*r*,*o*,*l*)
  pre: loc(*r*) = *l*, loc(*o*) = *r*
  eff: cargo(*r*) ← nil, loc(*o*) ← *l*

perceive(*r*,*l*)
  ?

  - If we knew this in advance, perception would not be necessary

Limitation

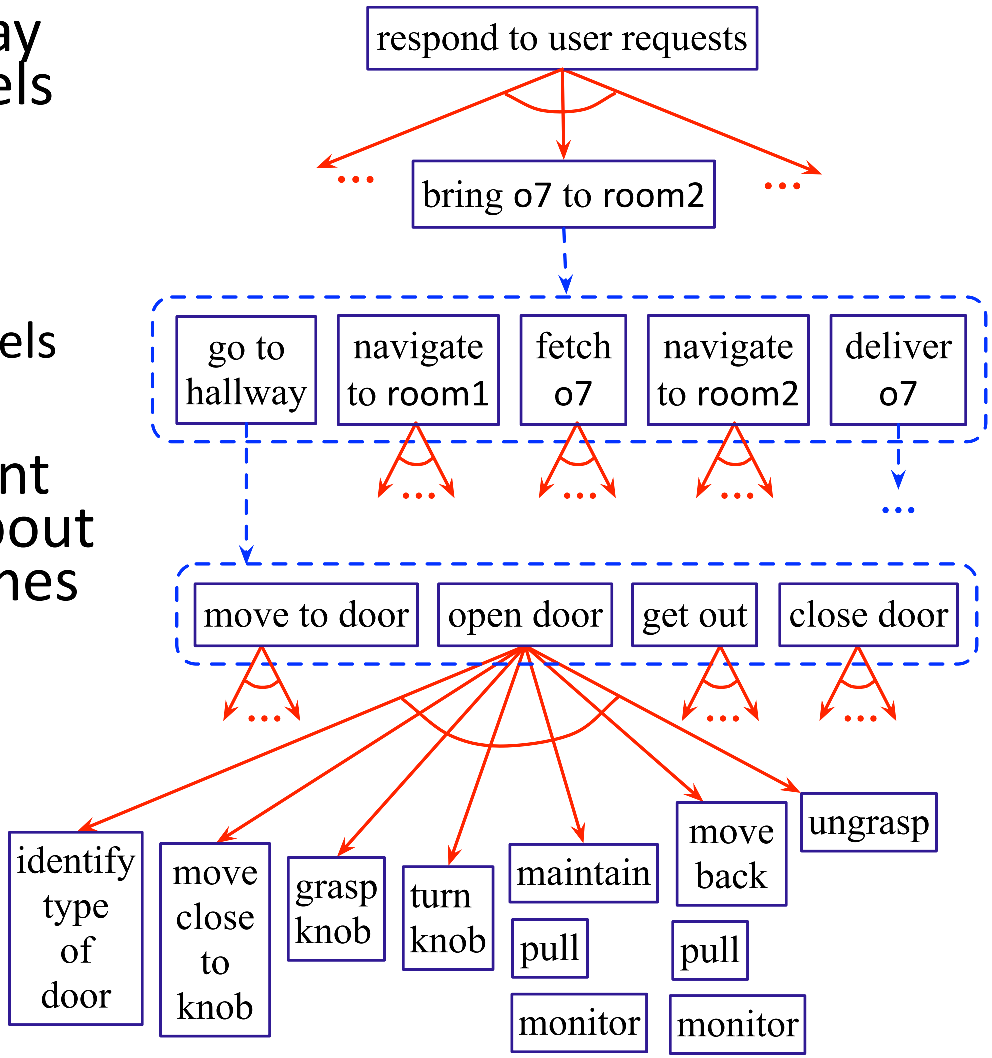
- Most environments are inherently nondeterministic
  - Deterministic action models will not always make the right prediction



- 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

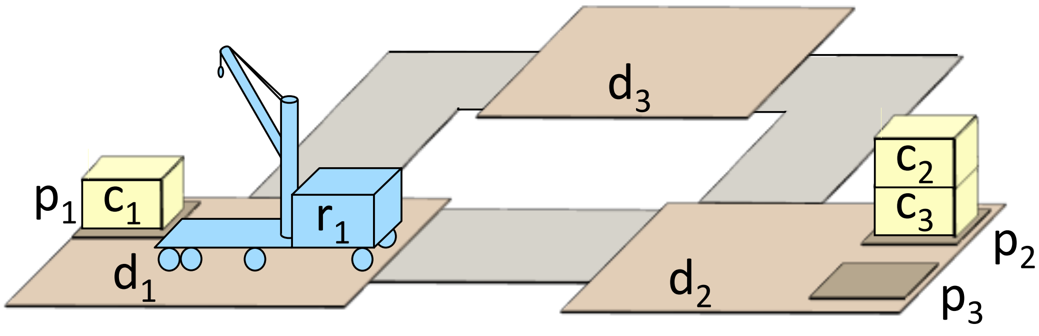
Planning/Acting at Different Levels

- Deterministic models may work better at some levels than others
- May want
  - Rae at some levels
  - Rae+planner at some levels
  - planner at some levels
- In some cases, might want the planner to reason about nondeterministic outcomes
  - Later in lecture (Book: Ch. 5 + 6)
- Ongoing research on extending refinement planning to handle nondeterminism
  [Patra *et al.*, AAAI-2019]



Simple Deterministic Domain

- Robot can move containers
  - Action models:
    - $load(r, c, c', p, d)$
      - 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: $loc(r) = d', occupied(d) = F, occupied(d') = T$

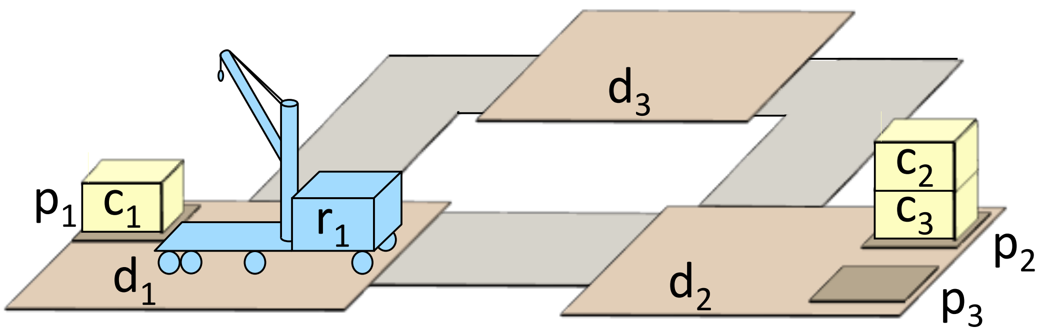


Tasks and Methods

- Task: put-in-pile(c,p') – put c into pile p' if not there
  - If c is not in p':
    - Find a route to c, follow it to c
    - Uncover c, load c onto r
    - Move to p', unload c
  - If c is already in p', do nothing

m2-put-in-pile(*r*,*c*,*p*,*d*,*p'*,*d'*)
task: put-in-pile(*c*,*p'*)
pre: pile(*c*)=*p* ∧ at(*p*,*d*) ∧ at(*p'*,*d'*) ∧ *p* ≠ *p'* ∧ cargo(*r*)=nil
body: if loc(*r*) ≠ *d* then navigate(*r*,*d*)
uncover(*c*)
load(*r*, *c*, pos(*c*), *p*, *d*)
if loc(*r*) ≠ *d'* then navigate(*r*,*d'*)
unload(*r*, *c*, top(*p'*), *p'*, *d*)

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

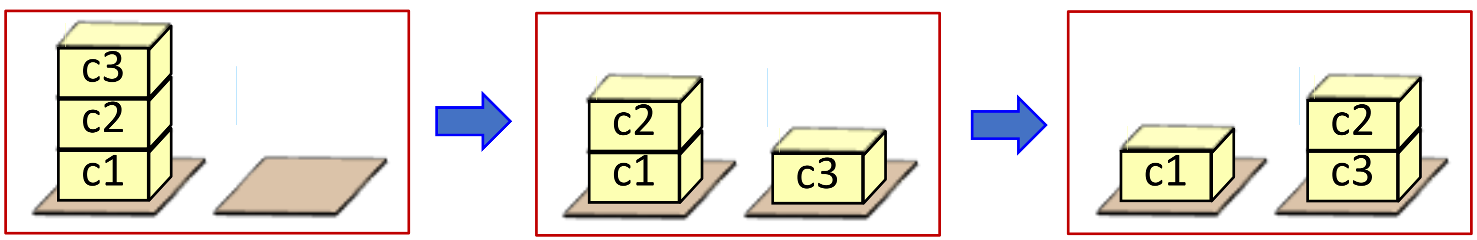


Tasks and Methods

- Task: uncover(c) – remove everything that is on c
  - While something is on c
    - Remove whatever is on top of the stack
  - If nothing is on c, do nothing

m2-uncover(r,c,c,p',d)
    task: uncover(c)
    pre: pile(c)=p ∧ top(p)≠c
        ∧ at(p,d) ∧ at(p',d) ∧ $p' \neq p$
        ∧ loc(r)=d ∧ cargo(r)=nil
    body: while top(p) ≠ c do
            c' ← top(p)
            load(r,c',pos(c'),p,d)
            unload(r,c',top(p'),p',d)

m1-uncover(c)
    task: uncover(c)
    pre: top(pile(c))=c
    body: *// empty*



SeRPE (Sequential Refinement Planning Engine)

- SeRPE

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

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

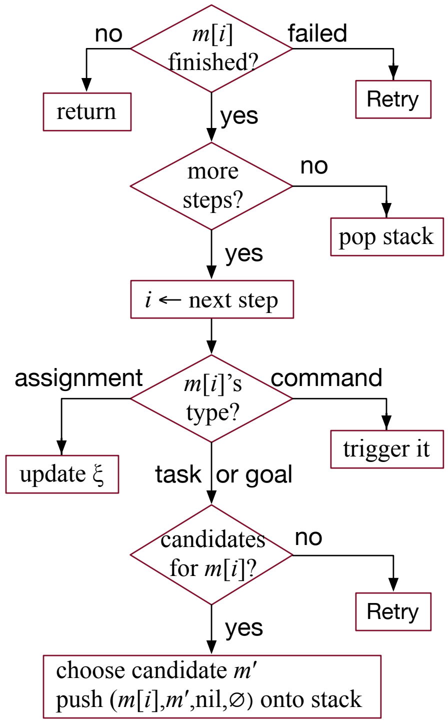
SeRPE (Sequential Refinement Planning Engine)

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

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

Rae’s Progress Subroutine



- Progress-to-finish
  - Like Progress with a loop around it
  - Simulates the commands

```
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)
```

Progress-to-finish

```
Progress-to-finish(𝓜,𝓐,s,τ,m)
    i ← nil; π ← ⟨⟩
    loop
        if τ is a goal and s ⊨ τ then
            return π
        if i is the last step of m then
            if τ is a goal and s ⊭ τ then
                return failure
            return π
        i ← nextstep(m,i)
        case type(m[i])
            assignment:
                update s according to m[i]
            command:
                a ← descriptive model of m[i] in 𝓐
                if s ⊨ pre(a) then
                    s ← γ(s,a); π ← π.a
                else
                    return failure
            task or goal:
                π′ ← SeRPE(𝓜,𝓐,s,m[i])
                if π′ = failure then
                    return failure
                s ← γ(s,π′); π ← π. π′
```

- 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, do not have Rae's Retry
  - If SeRPE failed, this means it could not find a solution
  - Implementation: hierarchical adaptations of backtracking, A*, GBFS, …

task

put-in-pile(c_1,p_2)

Example

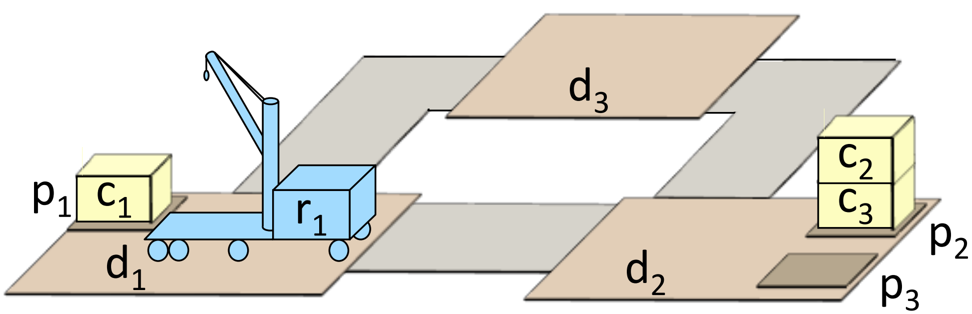
Candidates = {~~m1-put-in-pile(c_1,p_2)~~,
m2-put-in-pile(*r*,c_1,p_1,*d*,*p′*,*d′*)}

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

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

m2-put-in-pile(*r*,*c*,*p*,*d*,*p′*,*d′*)
- task: put-in-pile(*c*,*p′*)
- pre: pile(*c*)=*p* ∧ at(*p*,*d*) ∧ at(*p′*,*d′*) ∧ *p* ≠ *p′* ∧ cargo(*r*)=nil
- body: if loc(*r*) ≠ *d* then navigate(*r*,*d*)
  uncover(*c*)
  load(*r*, *c*, pos(*c*), *p*, *d*)
  if loc(*r*) ≠ *d′* then navigate(*r*,*d′*)
  unload(*r*, *c*, top(*p′*), *p′*, *d*)

s_0 = {loc(r_1)=d_1, cargo(r_1)=nil, occupied(d_1)=T,
occupied(d_2)=F, occupied(d_3)=F,
pos(c_1)=nil, pos(c_2)=c_3, pos(c_3)=nil,
pile(c_1)=p_1, pile(c_2)=p_2, pile(c_3)=p_2,
top(p_1)=c_1, top(p_2)=c_2, top(p_3)=nil}



Example

task

put-in-pile(c_1,p_2)

method

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

Refinement tree

- The SeRPE pseudocode doesn't return this, but can easily be modified to do so

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

r_1,c_1,p_1,d_1,p_2,d_2

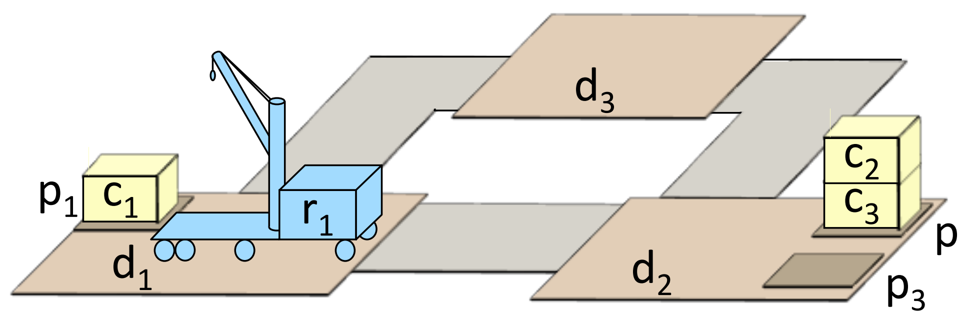
m2-put-in-pile(*r, c, p, d, p′, d′*)

task: put-in-pile(*c,p′*)

pre: pile(*c*)=*p* ∧ at(*p*,*d*) ∧ at(*p′*,*d′*) ∧ *p* ≠ *p′* ∧ cargo(*r*)=nil

body: if loc(*r*) ≠ *d* then navigate(*r,d*)
uncover(*c*)
load(*r, c,* pos(*c*), *p, d*)
if loc(*r*) ≠ *d′* then navigate(*r,d′*)
unload(*r, c,* top(*p′*), *p′, d*)

- m2-put-in-pile starts with *c*=c_1, *p′*=p_2, and *r, d, p′, d′* unbound
- Bind the other variables here



task

put-in-pile(c_1,p_2)

method

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

```
Progress-to-finish(ℳ,𝒜,s,τ,m)
    i ← nil; π ← ⟨⟩
    loop
        if τ is a goal and s ⊨ τ then
            return π
        if i is the last step of m then
            if τ is a goal and s ⊭ τ then
                return failure
            return π
        i ← nextstep(m,i)
        case type(m[i])
            assignment:
                update s according to m[i]
            command:
                a ← descriptive model of m[i] in 𝒜
                if s ⊨ pre(a) then
                    s ← γ(s,a); π ← π.a
                else
                    return failure
            task or goal:
                π′ ← SeRPE(ℳ,𝒜,s,m[i])
                if π′ = failure then
                    return failure
```

$\text{loc}(r_1) = d_1 = d$

r_1,c_1,p_1,d_1,p_2,d_2

m2-put-in-pile(*r, c, p, d, p′, d′*)

task: put-in-pile(*c,p′*)

pre: pile(*c*)=*p* ∧ at(*p,d*) ∧ at(*p′,d′*) ∧ *p* ≠ *p′* ∧ cargo(*r*)=nil

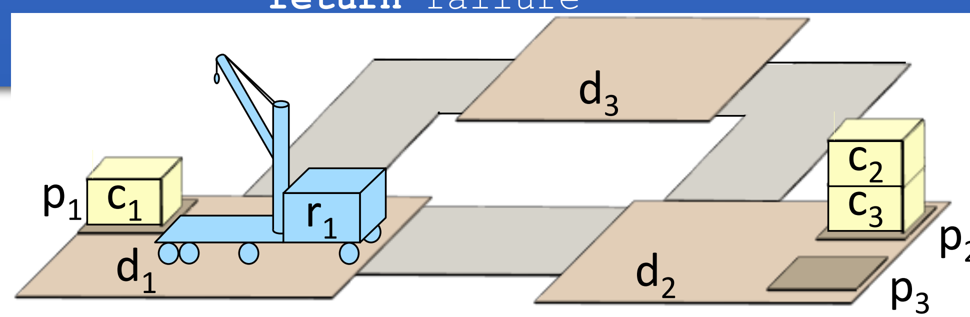
body: ~~if loc(*r*) ≠ *d*~~ then navigate(*r,d*)

uncover(*c*)

load(*r, c*, pos(*c*), *p, d*)

if loc(*r*) ≠ *d′* then navigate(*r,d′*)

unload(*r, c*, top(*p′*), *p′, d*)



task

put-in-pile(c_1,p_2)

method

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

task

uncover(c_1) … …

method

m1-uncover(c_1)

(no children)

m1-uncover(*c*)

task: uncover(*c*)

pre: top(pile(*c*))=*c*

body: *// empty*

m2-uncover(*r*,*c*,*c*,*p′*,*d*)

task: uncover(*c*)

pre: pile(*c*)=*p* ∧ top(*p*)≠*c*
∧ at(*p*,*d*) ∧ at(*p′*,*d*) ∧ *p′* ≠ *p*
∧ loc(*r*)=*d* ∧ cargo(*r*)=nil

body: while top(*p*) ≠ *c* do
c′ ← top(*p*)
load(*r*,*c′*,pos(*c′*),*p*,*d*)
unload(*r*,*c′*,top(*p′*),*p′*,*d*)

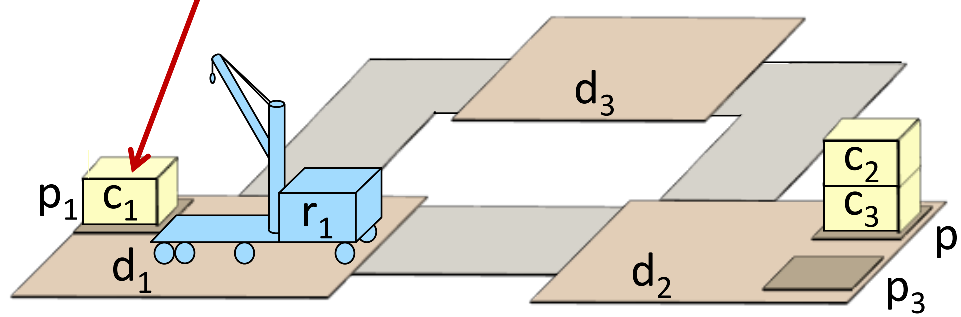
r_1,c_1,p_1,d_1,p_2,d_2

m2-put-in-pile(*r*, *c*, *p*, *d*, *p′*, *d′*)

task: put-in-pile(*c*,*p′*)

pre: pile(*c*)=*p* ∧ at(*p*,*d*) ∧ at(*p′*,*d′*)
∧ *p* ≠ *p′* ∧ cargo(*r*)=nil

body: if loc(*r*) ≠ *d* then navigate(*r*,*d*)
uncover(*c*)
load(*r*, *c*, pos(*c*), *p*, *d*)
if loc(*r*) ≠ *d′* then navigate(*r*,*d′*)
unload(*r*, *c*, top(*p′*), *p′*, *d*)



Example

task
put-in-pile(c_1,p_2)

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

task
uncover(c_1)

...

method
m1-uncover(c_1)
(no children)

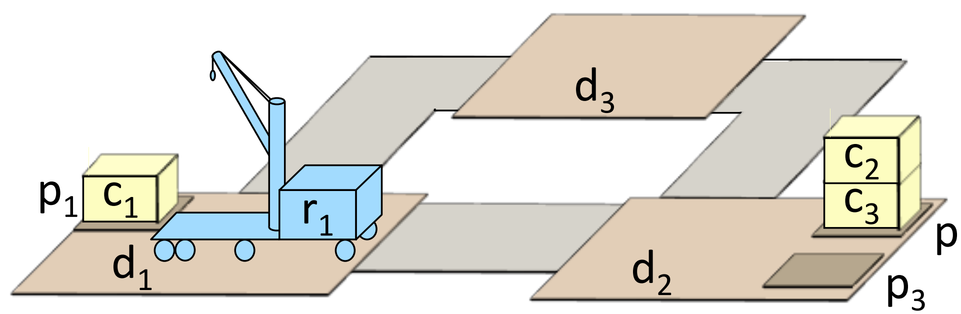
action
load(r_1,c_1,nil,p_1,d_1)

$r_1, c_1, p_1, d_1, p_2, d_2$

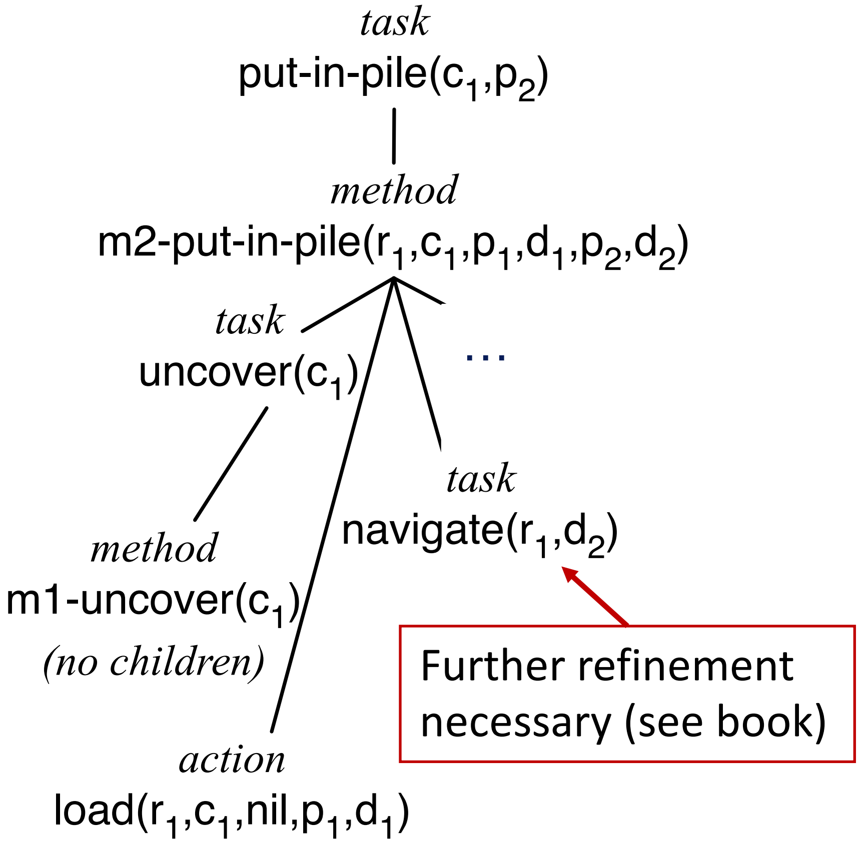
m2-put-in-pile(*r, c, p, d, p′, d′*)

...

body: if loc(*r*) ≠ *d* then navigate(*r,d*)
uncover(*c*)
load(*r, c*, pos(*c*), *p, d*) action
if loc(*r*) ≠ *d′* then navigate(*r,d′*)
unload(*r, c*, top(*p′*), *p′, d*)



Example

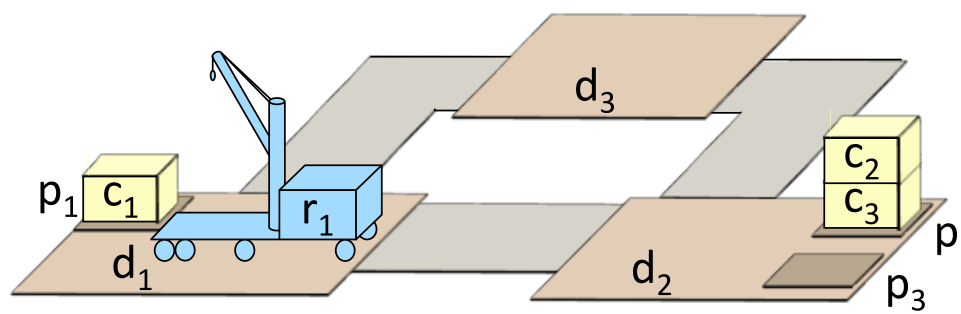


$r_1, c_1, p_1, d_1, p_2, d_2$

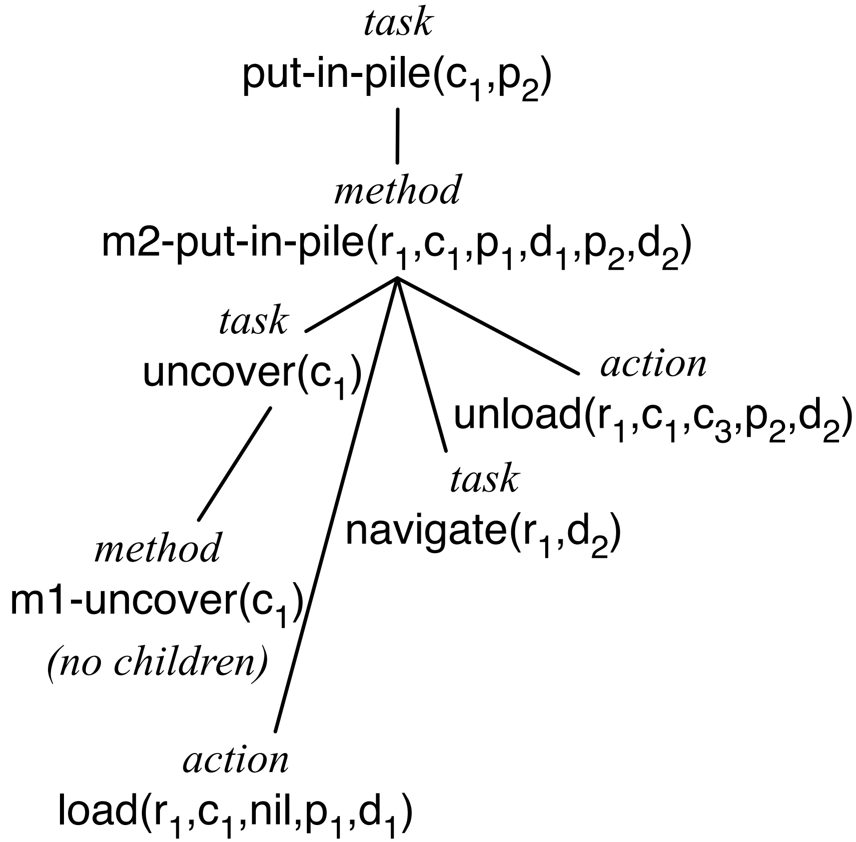
m2-put-in-pile(*r, c, p, d, p′, d′*)

...

body: if loc(*r*) ≠ *d* then navigate(*r,d*)
uncover(*c*)
load(*r, c*, pos(*c*), *p, d*) task
if loc(*r*) ≠ *d′* then navigate(*r,d′*)
unload(*r, c*, top(*p′*), *p′, d*)



Example

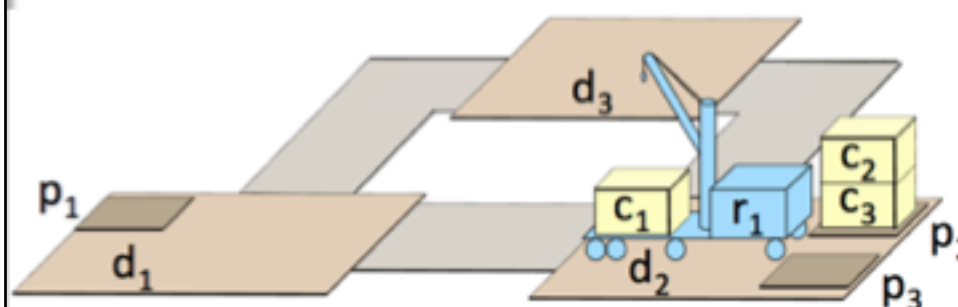


r_1,c_1,p_1,d_1,p_2,d_2

m2-put-in-pile(*r, c, p, d, p′, d′*)

...

body: if loc(*r*) ≠ *d* then navigate(*r,d*)
uncover(*c*)
load(*r*, *c*, pos(*c*), *p*, *d*)
if loc(*r*) ≠ *d′* then navigate(*r,d′*)
unload(*r*, *c*, top(*p′*), *p′*, *d*) action



Heuristics For SeRPE

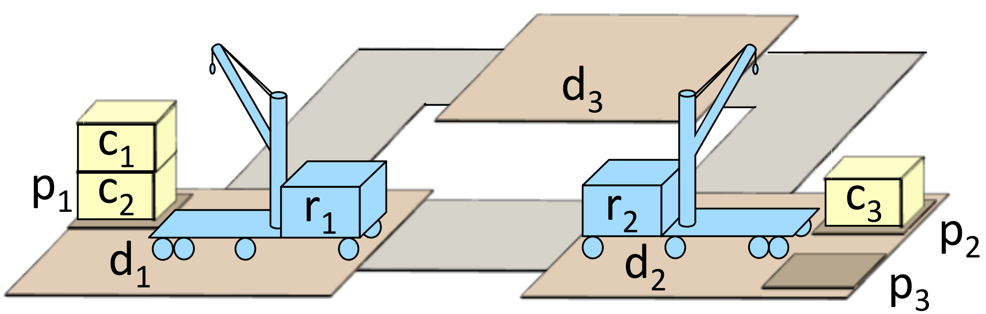
```
SeRPE(𝓜,𝓐,s,τ)
    Candidates ← Instances(𝓜,τ,ξ)
    if Candidates = ∅ then
        return failure
    nondeterministically choose m ∈ Candidates
    return Progress-to-finish(𝓜,𝓐,s,τ,m)
```

- *Ad hoc* approaches:
  - Domain-specific estimates
  - Statistical data on how well each method works
  - Try methods (or actions) in the order that they appear in $\mathcal{M}$ (or $\mathcal{A}$)
- Ideally, would want to implement using heuristic search (e.g., GBFS)
  - What heuristic function? Open problem
- SeRPE is a generalization of HTN planning
  - In some cases classical-planning heuristics can be used, in other cases they become intractable [Shivashankar *et al.*, ECAI-2016]

Interleaving

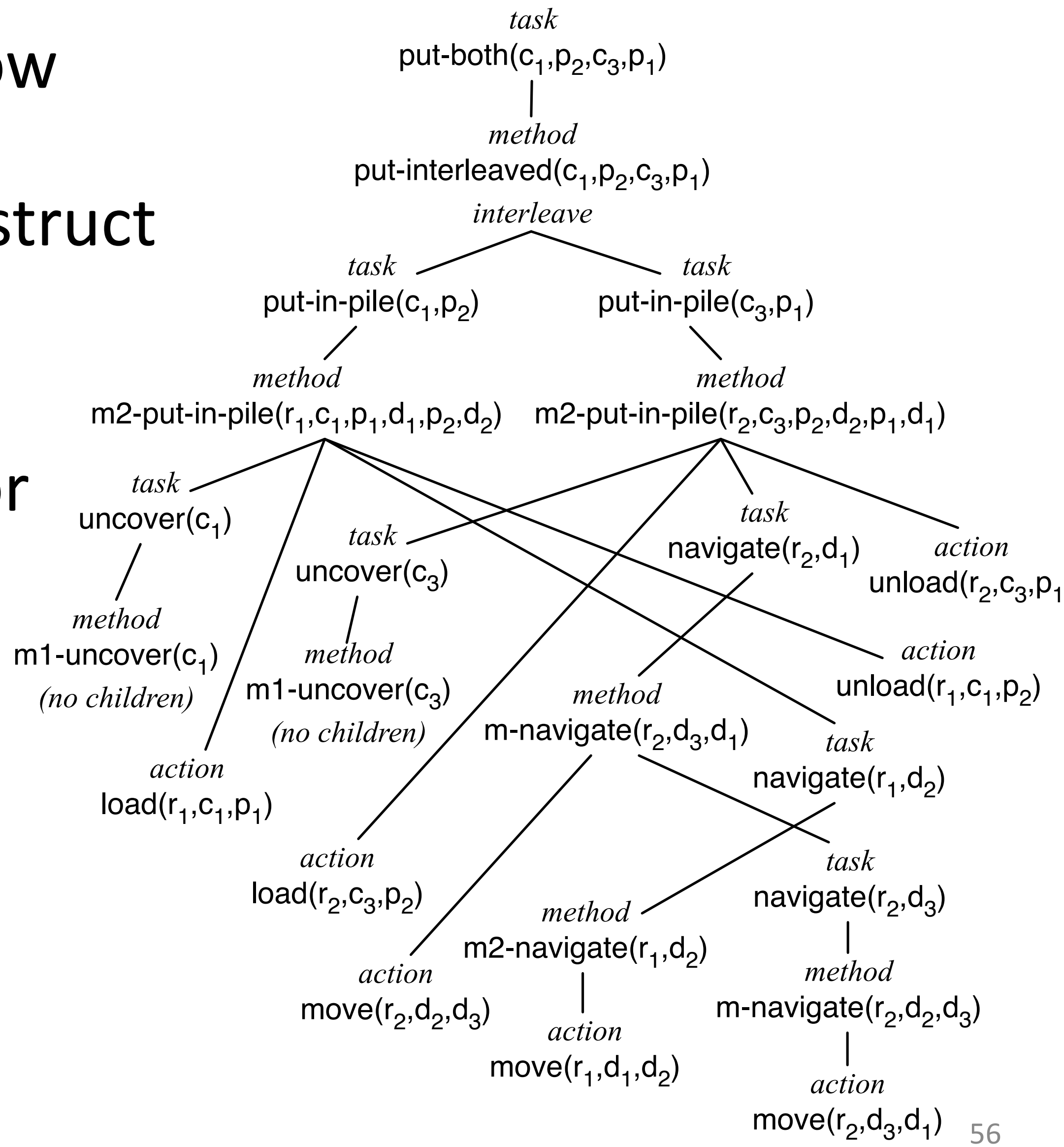
- Want to move c_1 to p_2, using this plan …
  - $\langle load(r_1, c_1, c_2, p_1, d_1),$ $move(r_1, d_1, d_2),$ $unload(r_1, c_1, p_3, nil, d_2)\rangle$
- … and move c_3 to p_1 using this plan:
  - $\langle load(r_2, c_3, nil, p_2, d_2),$ $move(r_2, d_2, d_3),$ $move(r_2, d_3, d_1),$ $unload(r_2, c_3,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



Summary

- Refinement planning (SeRPE)
  - Plan by simulating RAE on a single external task/event/goal
  - Deterministic actions
    - OK if we are confident of outcome, can recover if things go wrong
  - Interleaved plans (brief example)

Outline per the Book

3.1 *Representation*
- State variables, commands, refinement methods
- Example

3.2 *Acting*
- Rae (Refinement Acting Engine)
- Example
- Extensions

3.3 *Planning*
- Motivation and basic ideas
- Deterministic action models
- SeRPE (Sequential Refinement Planning Engine)

3.4 *Using Planning in Acting*
- Techniques
- Caveats

Acting and Refinement Planning

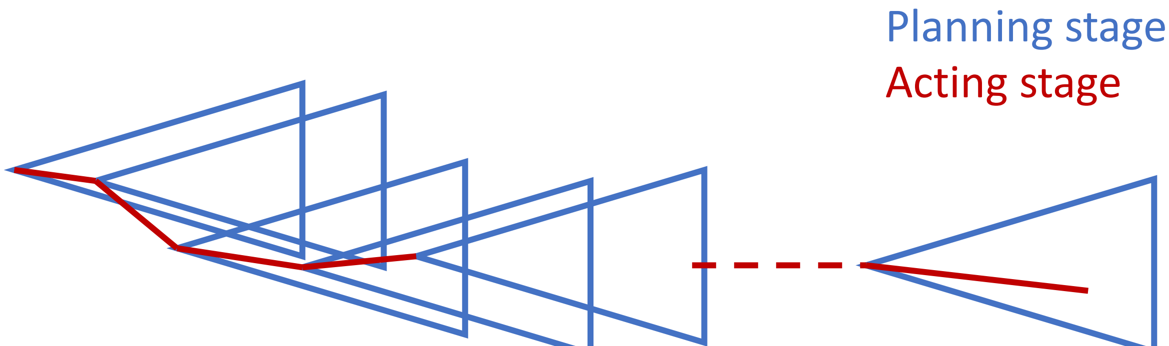
- Hierarchical acting with refinement planning
  - REAP: a RAE-like actor uses SeRPE-like planning at all levels

- 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?

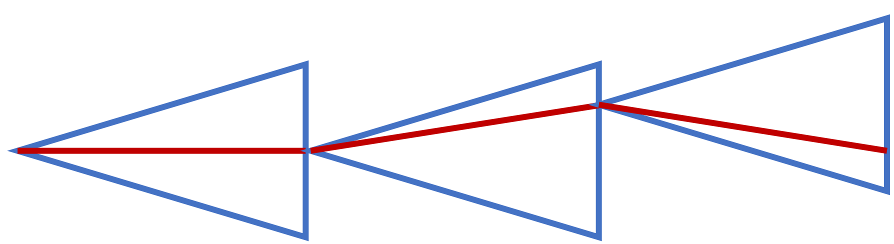
```
Refine-Lookahead(𝓜,𝓐,τ)
    while (s ← abstraction of
            observed state ξ) ⊭ τ do
        π ← SeRPE-Lookahead(𝓜,𝓐,s,τ)
        if π = failure then
            return failure
        a ← pop-first-action(π)
        perform a
```



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(ℳ,𝒜,τ)
    s ← abstraction of
            observed state ξ
    while s ⊭ τ do
        π ← SeRPE-Lookahead(ℳ,𝒜,s,τ)
        if π = failure then
            return failure
        while π ≠ ⟨⟩ and s ⊭ τ and
                Simulate(Σ,s,τ,π)
                    ≠ failure do
            a ← pop-first-action(π)
            perform a
            s ← abstraction of
                    observed state ξ
```

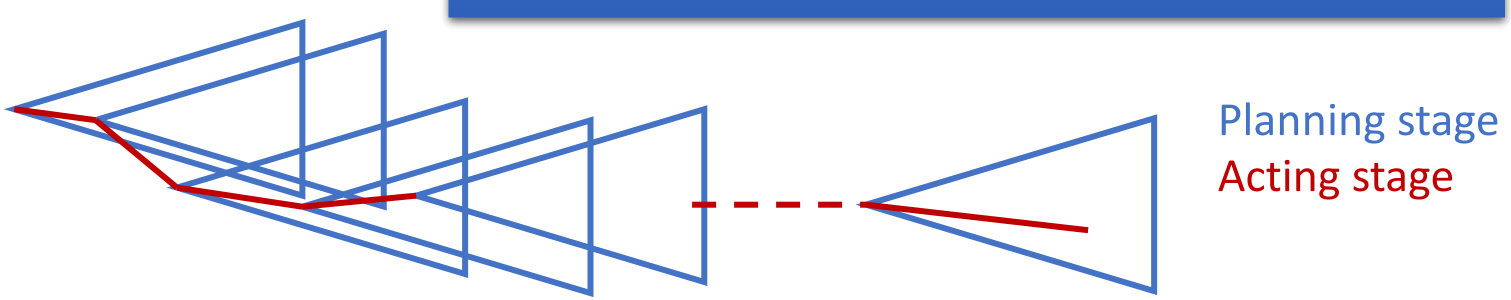


Planning stage
Acting stage

Using Planning in Acting

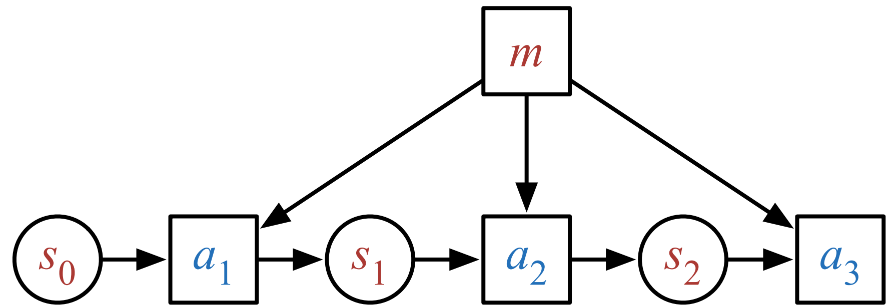
- Objective:
  - Balance trade-offs between Refine-Lookahead and Refine-Lazy-Lookahead
  - More up-to-date plans than Refine-Lazy-Lookahead, but without waiting for Lookahead to return

```
Refine-Concurrent-Lookahead(𝓜,𝓐,τ)
    π ← ⟨⟩
    s ← abstraction of observed state ξ
    // threads 1 and 2 run concurrently
thread 1:
    loop
        π ← SeRPE-Lookahead(𝓜,𝓐,s,τ)
thread 2:
    loop
        if s ⊭ τ then
            return success
        else if π = failure then
            return failure
        else if π ≠ ⟨⟩ and s ⊭ τ and
                Simulate(Σ,s,τ,π) ≠ failure then
            a ← pop-first-action(π)
            perform a
            s ← abstraction of observed state ξ
```



Caveats

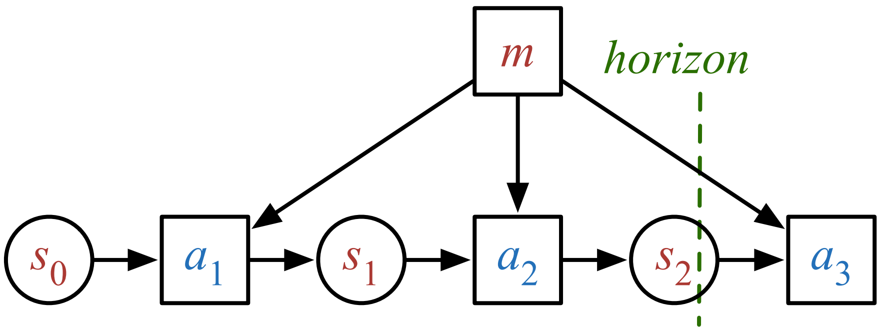
- Start in state s_0,
  want to accomplish task τ
  - Refinement method m:
    - task: τ
    - pre: s_0
    - body: a_1, a_2, a_3



- 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

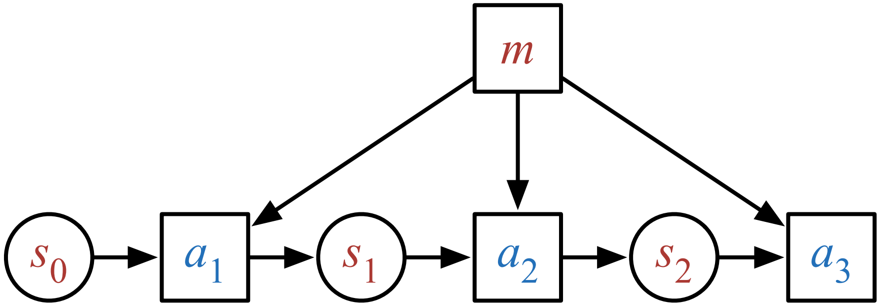
- Start in state s_0, want to accomplish task τ
  - Refinement method m:
    - task: τ
    - pre: s_0
    - body: a_1, a_2, a_3



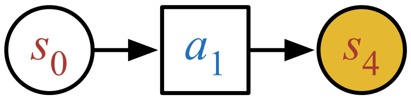
- Actor uses Run-Lazy-Lookahead
  - Lookahead = SeRPE with receding horizon, returns $\langle a_1, a_2 \rangle$
  - Actor performs them, calls Lookahead again
  - No applicable method for τ in s_2, SeRPE returns failure
- Can use the same fixes on previous slide, with one modification
  - Keep snapshot 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

Outline per the Book

3.1 *Representation*

- State variables, commands, refinement methods
- Example

3.2 *Acting*

- Rae (Refinement Acting Engine)
- Example
- Extensions

3.3 *Planning*

- Motivation and basic ideas
- Deterministic action models
- SeRPE (Sequential Refinement Planning Engine)

3.4 *Using Planning in Acting*

- Techniques
- Caveats

$\Longrightarrow$ Next: Planning and Acting with Temporal Models