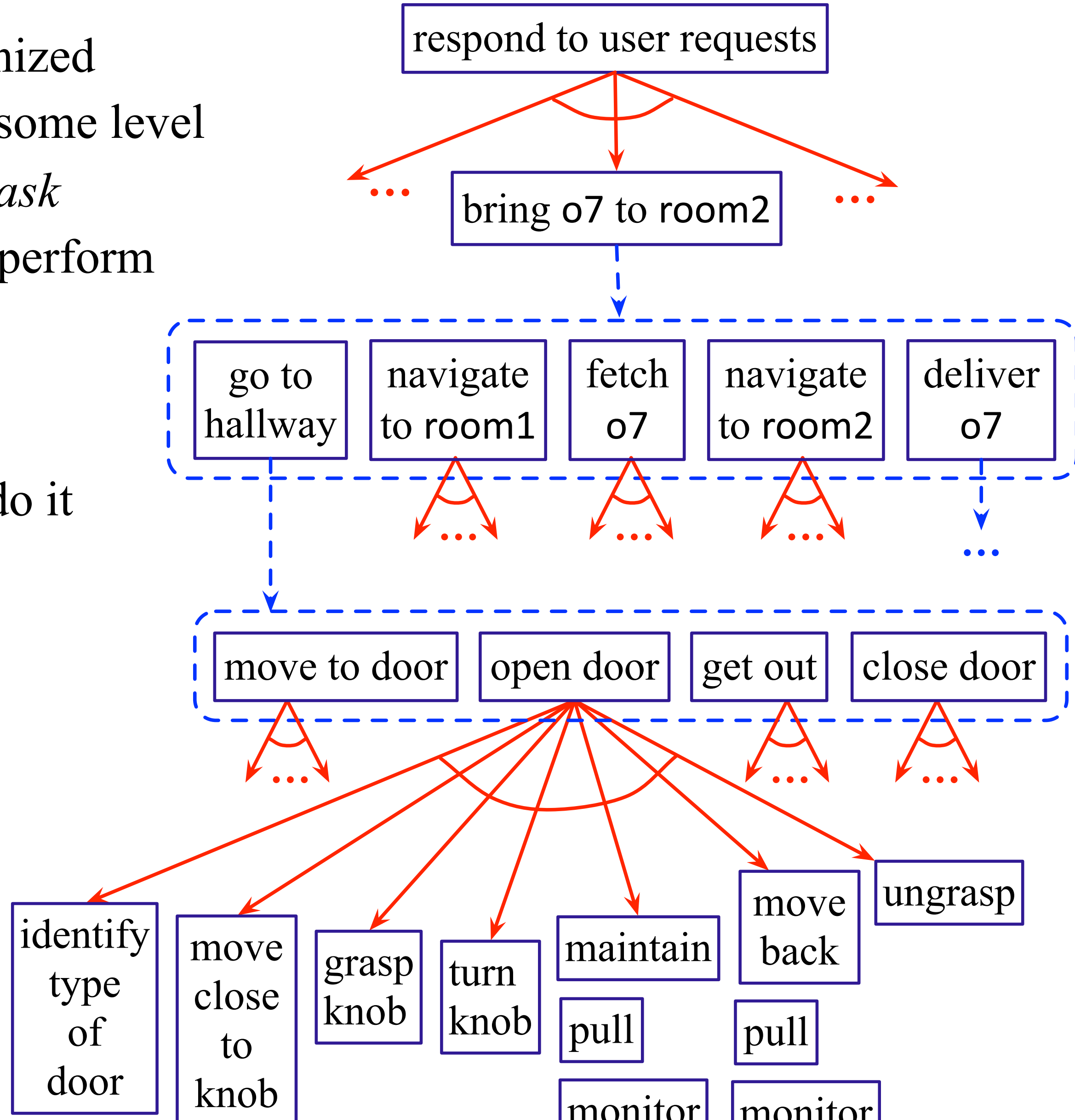
Chapter 3
Deliberation with Refinement Methods

Dana S. Nau
University of Maryland

Motivation

- Actor is hierarchically organized
  - Consider an action *a* at some level
  - One level down, *a* is a *task*
    - complex activity to perform
- Recursively refine tasks into subtasks
  - Dashed blue lines: state-space planner can do it
  - Solid red lines: need other techniques
- *Refinement method*: operational model
  - Program to compute a refinement
  - Actor can use it online
  - Can also use it for planning



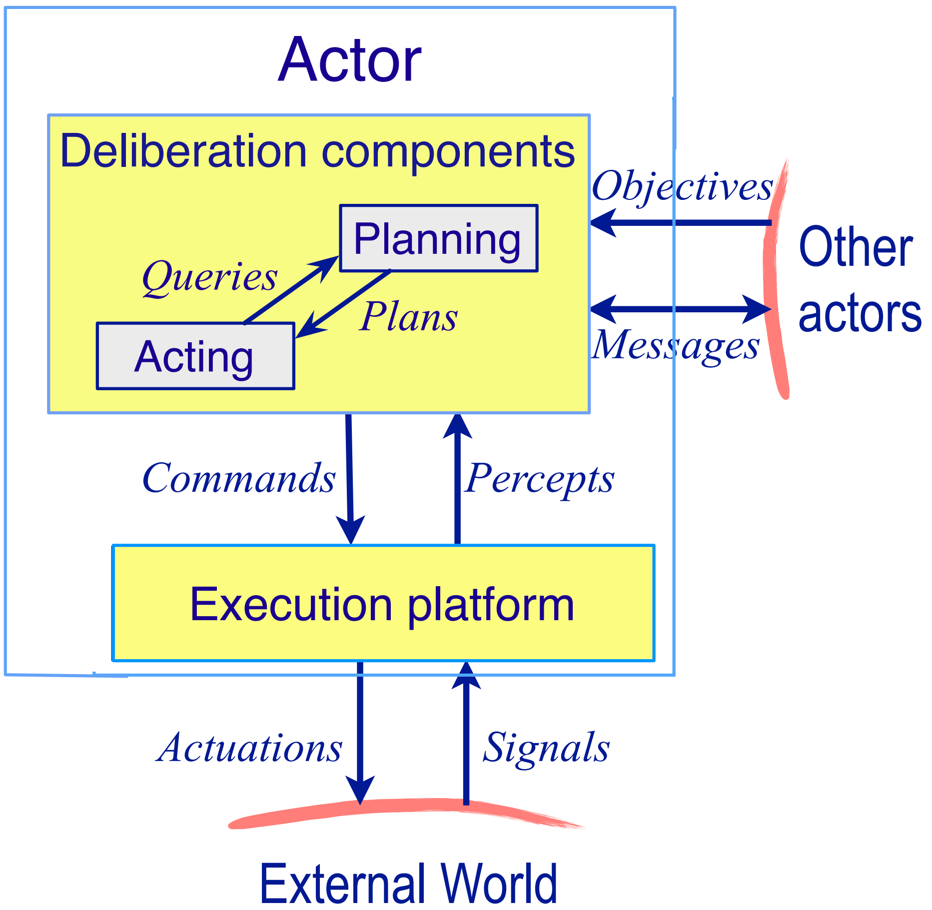
Outline

- 3.1 Operational models
- 3.2 Refinement Acting Engine (RAE)
- 3.3 Refinement planning (SeRPE)
- 3.4 Acting and refinement planning

3.1 Operational Models

Notation and terminology:

- ξ is the actor's current state
- Ξ = set of all possible ξ
- Execution platform keeps ξ up-to-date
  - may differ from the state s the actor gives the planner
- *Fact*: ground atom that's true in ξ
- *Task*: activity that the actor needs to carry out
- *Command*: primitive function that the execution platform can perform
- *Event*: occurrence detected by the execution platform
  - exogenous change in the environment



State-Variable Representation

Extensions:

- Range(x)
  - can be finite, infinite, continuous, discontinuous, vectors, matrices, other data structures
- Assignment statement $x \leftarrow expr$
  - expression that returns a ground value in Range(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

Example

- Objects
  - $Robots$ = {rbt}, $Containers$ = {c1, c2, …}, $Locations$ = {loc1, loc2, …}
- State variables
  - loc(r) $\in$ $Locations$
  - load(r) $\in$ $Containers \cup$ {nil}
  - pos(c) $\in$ $Locations \cup Robots \cup$ {unknown}
  - view(l) $\in$ {T, F}
    - Whether the robot has looked at location l or not
    - If view(l) = T then for every container c at location l, pos(c) = 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 location l

Refinement Methods

- Refinement method: a triple (*task or event*, *precondition*, *body*)
  - a task or event that the method can be used for
  - precondition that needs to be true for the method to be applicable
  - body: a program
- Write in the following format:

method-name(arg_1, …, arg_k)
  task: *task-identifier*
  pre: *test*
  body: *program*

method-name(arg_1, …, arg_k)
  event: *event-identifier*
  pre: *test*
  body: *program*

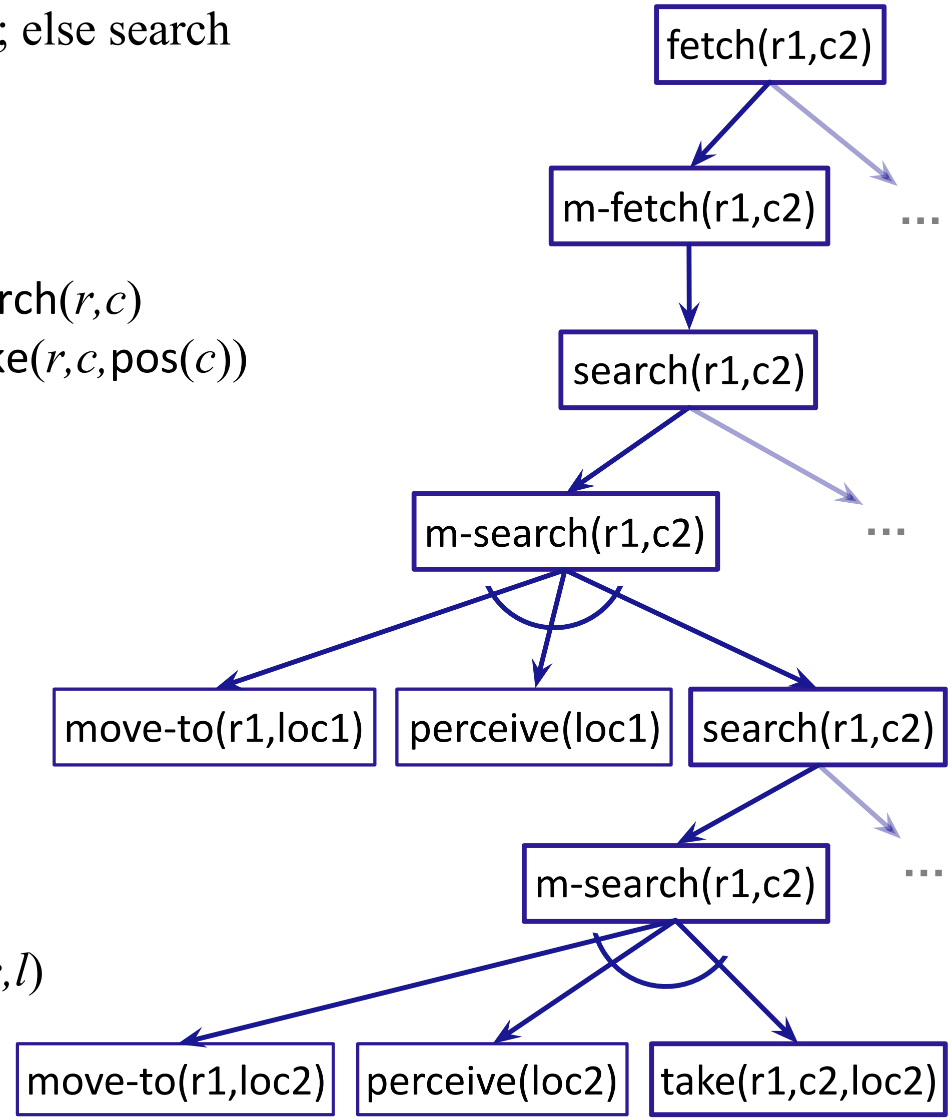
- *body*: sequence of steps
  - assignments, commands, tasks
  - control constructs: if-then-else, while, loop, etc.

Methods for Tasks

- Example task: fetch container *c*
  - If you know *c*'s location, go get it; else search

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(*l*) = F) then
          move-to(*r*,*l*)
          perceive(*l*)
          if pos(*c*) = *l* then take(*r*,*c*,*l*)
          else search(*r*,*c*)
      else fail



Methods for Events

- Example event: emergency
  - If you aren't already handing another emergency, then
    - stop what you're doing, go to where emergency is, do something about it

- Another state variable
  - emergency-handling(r) $\in$ {T, F}
    - whether robot r is handling an emergency

m-emergency(r,l,i)

i is the ID number of the event

event: emergency(l,i)

pre: emergency-handling(r) = F

body: emergency-handling(r) $\leftarrow$ T
    if load(r) $\neq$ nil then put(r,load(r))
    move-to(l)
    address-emergency(l,i)

Example: Opening a Door

- State variables:
  - reachable(r,o) ∈ {T, F}
    - is o close enough to grasp?
  - door-status(d) ∈ {closed, cracked, open, unknown}

- Rigid relations:
  - adjacent(l,d) location l is adjacent to door d
  - toward-side(l,d) location l is on the side where d's door hinges are
  - away-side(l,d) location l is on the opposite side of d's door hinges
  - handle(d,o) o is the handle of door d
  - type(d,rotates) door d rotates
  - type(d,slide) door d slides
  - side(d,left) door d slides left, or rotates left when you pull it
  - side(d,right) door d slides right, or rotates right when you pull it

Example: Opening a Door

- Commands
  - move-close(r,o): r moves to a position where reachable(r,o) = T
  - move-by(r,λ): r moves with magnitude and direction given by vector λ
  - grasp(r,o): r grasps object o
  - ungrasp(r,o): r release grasp on o
  - turn(r,o,α): r turns a grasped object o by angle $\alpha \in [-\pi,+\pi]$
  - pull(r,λ): r pulls its arm by vector λ
  - push(r,λ): r pushes its arm by λ
  - monitor-status(r,d): r focuses its perception to keep door-status updated
    (starts a process)
  - end-monitor-status(r,d): r stops monitoring d
    (terminates a process)

Example: Opening a Door

- Move close to door handle, unlatch door, pull it open

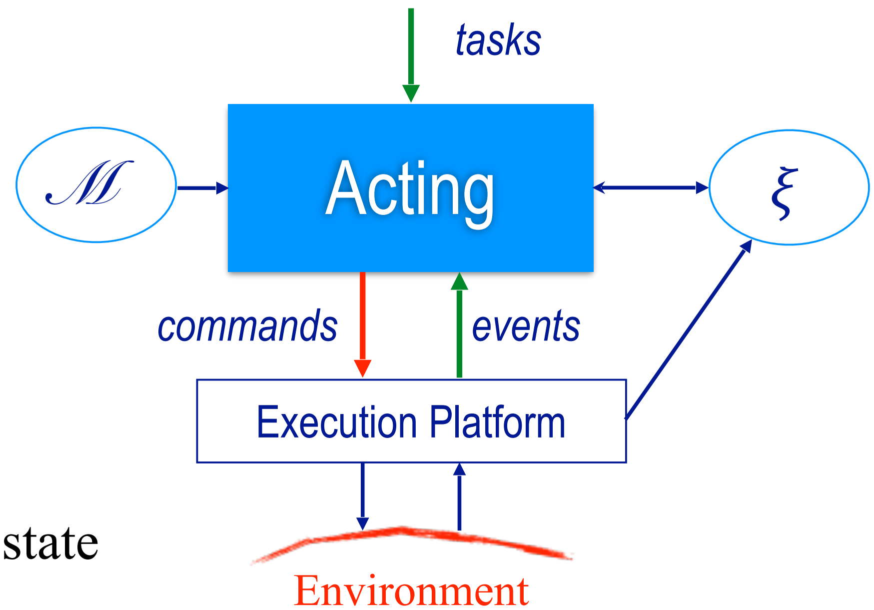
m-opendoor(r,d,l,h)
  task: opendoor(r,d)
  pre: loc(r) = l ∧ adjacent(l,d)
    ∧ handle(d,h)
  body: while ¬reachable(r,h) do
      move-close(r,h)
    monitor-status(r,d)
    if door-status(d)=closed then
      unlatch(r,d)
    throw-wide(r,d)
    end-monitor-status(r,d)

- Methods for hinged door that opens to the left, toward the robot
  - Need other methods for doors that slide, rotate to the right, or rotate away from the robot

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)

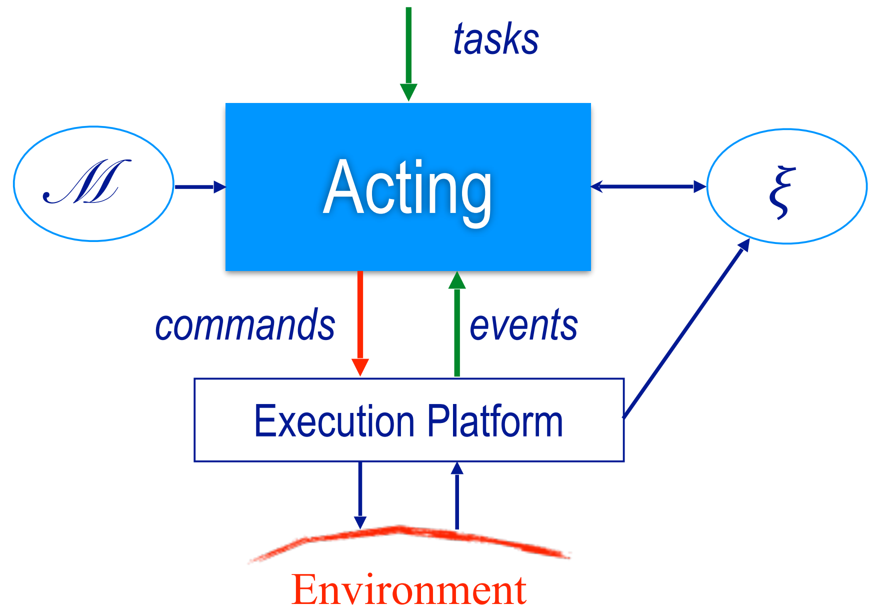
3.2 Refinement Acting Engine (RAE)



- Based on OpenPRS
  - Programming language, open-source robotics software
- Input:
  - facts, external tasks, events, current state
- Output:
  - commands to execution platform

Refinement Acting Engine (RAE)

- Perform external tasks/events in parallel
  - For each, a *refinement stack*
- *Agenda* = {current stacks}



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

Example

Call stack:
RAE

Refinement tree:

fetch(r1,c2)
$\downarrow$
m-fetch(r1,c2)

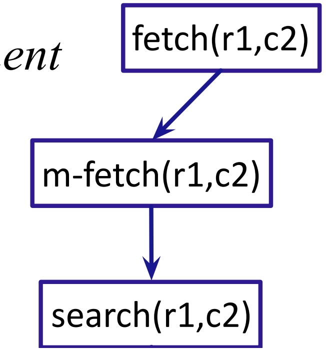
Rae($\mathcal{M}$)
  $Agenda \leftarrow \varnothing$
  loop
    until the input stream of external tasks and events is empty do
      read τ in the input stream
      $Candidates \leftarrow \mathsf{Instances}(\mathcal{M}, \tau, \xi)$
      if $Candidates = \varnothing$ then output(“failed to address”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

- τ = fetch(r1,c2)
- $Candidates$ = {m-fetch(r1,c2)}
- m = m-fetch(r1,c2)
- $Agenda$ = {⟨(fetch(r1,c2), m-fetch(r1,c2), nil, $\varnothing$)⟩}
- *nothing else in the stream*
- $stack$ = ⟨(fetch(r1,c2), m-fetch(r1,c2), nil, $\varnothing$)⟩
- *Call* Progress($stack$)

Progress($stack$)
  $(\tau, m, i, tried) \leftarrow$ top($stack$)
  if $i \neq$ nil and $m[i]$ is a command then do
    case status($m[i]$)
      running: return
      failure: Retry($stack$); return
      done: continue
  if i is the last step of m then
      pop($stack$)   // remove $stack$'s top element
  else do
    $i \leftarrow$ nextstep(m, i)
    case type($m[i]$)
      assignment: update ξ according to $m[i]$; return
      command: trigger command $m[i]$; return
      task or goal: continue
    $\tau' \leftarrow m[i]$
    $Candidates \leftarrow$ Instances($\mathcal{M}, \tau', \xi$)
    if $Candidates = \varnothing$ then Retry($stack$)
    else do
      arbitrarily choose $m' \in Candidates$
      $stack \leftarrow$ push((τ', m', nil, $\varnothing$), $stack$)

Call stack:
RAE
Progress [1]

Refinement tree:



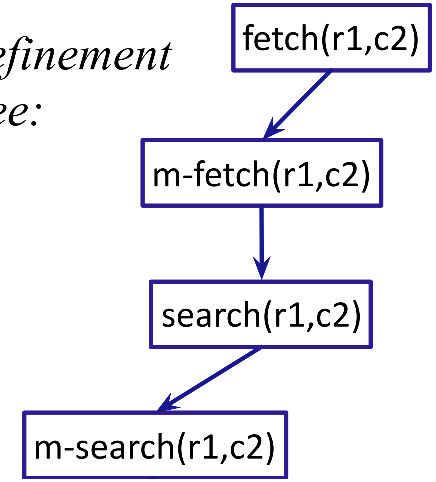
- *stack* = ⟨(fetch(r1,c2), m-fetch(r1,c2), nil, ∅)⟩
- (τ,*m*,*i*,*tried*) = (fetch(r1,c2), m-fetch(r1,c2), ~~nil~~, ∅)
- $i \leftarrow$ nextstep(m-fetch(r1,c2),nil) = i1
- τ' = *m*[i1] = search(r1,c2)

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*))

Progress($stack$)
  $(\tau, m, i, tried) \leftarrow$ top($stack$)
  if $i \neq$ nil and $m[i]$ is a command then do
    case status($m[i]$)
      running: return
      failure: Retry($stack$); return
      done: continue
  if i is the last step of m then
      pop($stack$)   // remove $stack$'s top element
  else do
    $i \leftarrow$ nextstep(m, i)
    case type($m[i]$)
      assignment: update ξ according to $m[i]$; return
      command: trigger command $m[i]$; return
      task or goal: continue
    $\tau' \leftarrow m[i]$
    $Candidates \leftarrow$ Instances($\mathcal{M}, \tau', \xi$)
    if $Candidates = \varnothing$ then Retry($stack$)
    else do
      arbitrarily choose $m' \in Candidates$
      $stack \leftarrow$ push($(\tau', m'$, nil, $\varnothing)$, $stack$)

Call stack:
RAE
Progress [1]

Refinement tree:



- τ' = search(r1,c2)
- *Candidates* = {m-search(r1,c2)}
- m' = m-search(r1,c2)
- *stack* = ⟨(search(r1,c2), m-search(r1,c2), nil, $\varnothing$), (fetch(r1,c2), m-fetch(r1,c2), i1, $\varnothing$)⟩
- *Return to* RAE

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))

Rae($\mathcal{M}$)

$Agenda \leftarrow \varnothing$

loop

  until the input stream of external tasks and events is empty do

    read τ in the input stream

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

    if $Candidates = \varnothing$ then output("failed to address" τ)

    else do

      arbitrarily choose $m \in Candidates$

      $Agenda \leftarrow Agenda \cup \{\langle(\tau, m, \text{nil}, \varnothing)\rangle\}$

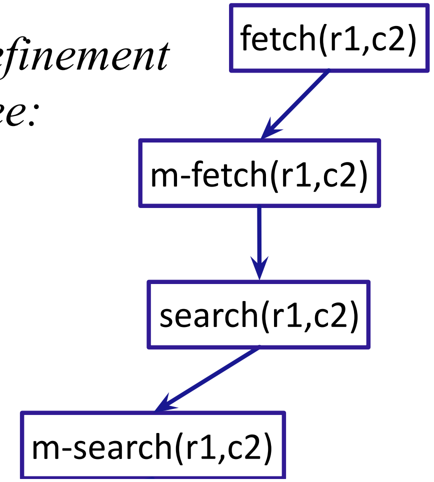
  for each $stack \in Agenda$ do

    Progress($stack$)

    if $stack = \varnothing$ then $Agenda \leftarrow Agenda \setminus \{stack\}$

Call stack:
RAE

Refinement tree:



- *Returned from* Progress(*stack*)
- *stack* =
  ⟨(search(r1,c2), m-search(r1,c2), nil, ∅),
  (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)⟩
- *finish "for", iterate outer loop*
- *input stream empty*
- *Call* Progress(*stack*)

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))

Progress($stack$)

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

  if $i \neq$ nil and $m[i]$ is a command then do

    case status($m[i]$)

      running: return

      failure: Retry($stack$); return

      done: continue

  if i is the last step of m then

      pop($stack$)   // remove $stack$'s top element

  else do

    $i \leftarrow$ nextstep(m, i)

    case type($m[i]$)

      assignment: update ξ according to $m[i]$; return

      command: trigger command $m[i]$; return

      task or goal: continue

    $\tau' \leftarrow m[i]$

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

    if $Candidates = \varnothing$ then Retry($stack$)

    else do

      arbitrarily choose $m' \in Candidates$

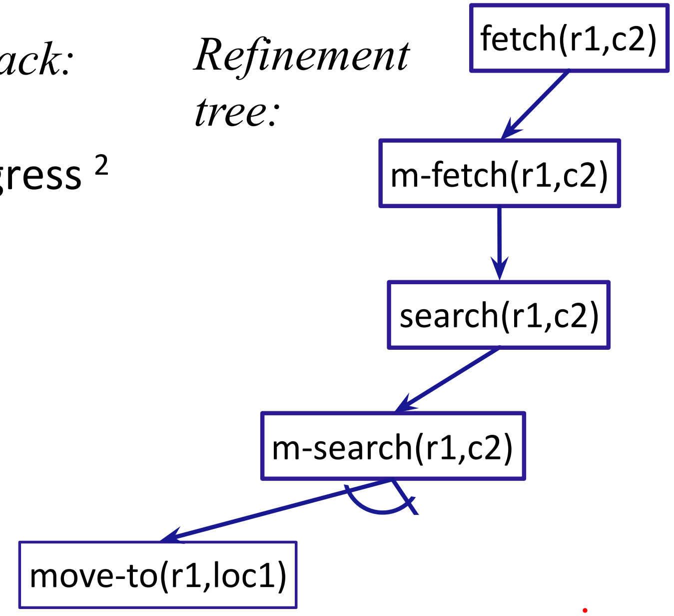
      $stack \leftarrow$ push((τ', m', nil, $\varnothing$), $stack$)

Call stack:

RAE

Progress [2]

Refinement tree:



- *stack*
  = ⟨(search(r1,c2), m-search(r1,c2), nil, $\varnothing$),
  (fetch(r1,c2), m-fetch(r1,c2), i1, $\varnothing$)⟩
- $l \leftarrow$ loc1; $i \leftarrow$ i2
- *trigger* m[i2] = move-to(r1,loc1)
- *return to* RAE; RAE calls Progress again

i

r=r1; c=c2

m-search(r,c)

  task: search(r,c)

  pre: pos(c) = unknown

  body: if ∃ l (view(l)=F) then

      move-to(r,l)

      perceive(l)

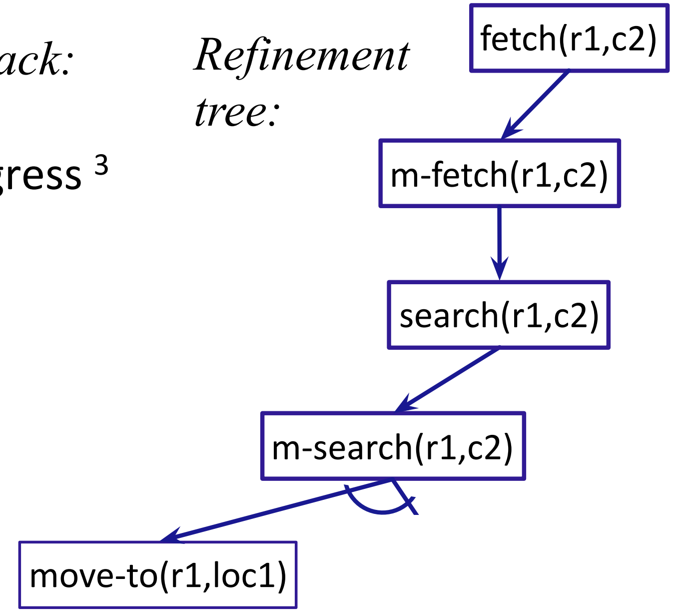
      if pos(c)=l then take(r,c,l) else search(r,c)

    else fail

Progress($stack$)
  $(\tau, m, i, tried) \leftarrow$ top($stack$)
  if $i \neq$ nil and $m[i]$ is a command then do
    $\boxed{\text{case status}(m[i])}$
      $\boxed{\text{running: return}}$
      failure: Retry($stack$); return
      done: continue
  if i is the last step of m then
      pop($stack$)   // remove $stack$'s top element
  else do
    $i \leftarrow$ nextstep(m, i)
    case type($m[i]$)
      assignment: update ξ according to $m[i]$; return
      command: trigger command $m[i]$; return
      task or goal: continue
    $\tau' \leftarrow m[i]$
    $Candidates \leftarrow$ Instances($\mathcal{M}, \tau', \xi$)
    if $Candidates = \varnothing$ then Retry($stack$)
    else do
      arbitrarily choose $m' \in Candidates$
      $stack \leftarrow$ push((τ', m', nil, $\varnothing$),$stack$)

Call stack:
RAE
Progress [3]

Refinement tree:



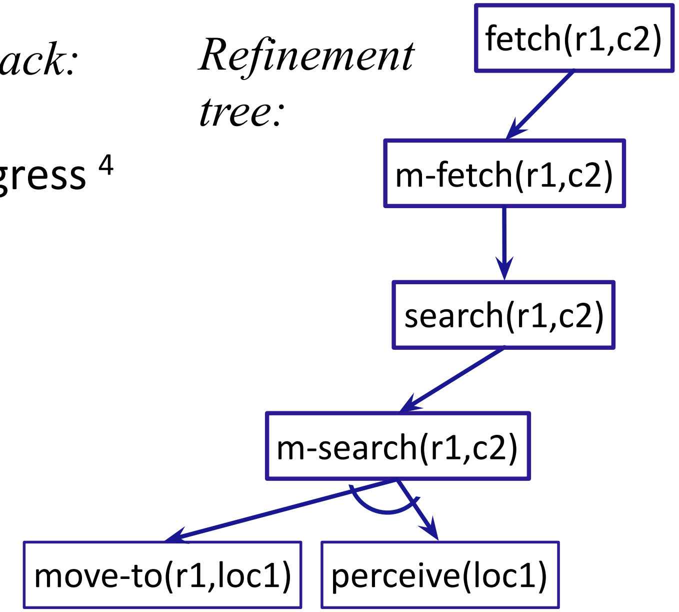
- $stack$ =
  ⟨(search(r1,c2), m-search(r1,c2), i2, ∅),
  (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)⟩
  i
- status(m[i2]) = running
- *return to* RAE; RAE calls Progress again

r=r1; c=c2
m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body: if ∃ l (view(l)=F) then
      $\boxed{\text{move-to}(r,l)}$
      perceive(l)
      if pos(c)=l then take(r,c,l) else search(r,c)
    else fail

Progress($stack$)
  $(\tau, m, i, tried) \leftarrow$ top($stack$)
  if $i \neq$ nil and $m[i]$ is a command then do
    case status($m[i]$)
      running: return
      failure: Retry($stack$); return
      done: continue
  if i is the last step of m then
      pop($stack$)   // remove $stack$'s top element
  else do
    $i \leftarrow$ nextstep(m, i)
    case type($m[i]$)
      assignment: update ξ according to $m[i]$; return
      command: trigger command $m[i]$; return
      task or goal: continue
    $\tau' \leftarrow m[i]$
    $Candidates \leftarrow$ Instances($\mathcal{M}, \tau', \xi$)
    if $Candidates = \varnothing$ then Retry($stack$)
    else do
      arbitrarily choose $m' \in Candidates$
      $stack \leftarrow$ push((τ', m', nil, $\varnothing$),$stack$)

Call stack:
  RAE
  Progress [4]

Refinement tree:



- *stack*
  i
  = ⟨(search(r1,c2), m-search(r1,c2), i2, ∅),
  (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)⟩
- status(m[i2]) = done; $i \leftarrow$ i3
- *trigger* perceive(loc1)
- *return to* RAE; RAE calls Progress again

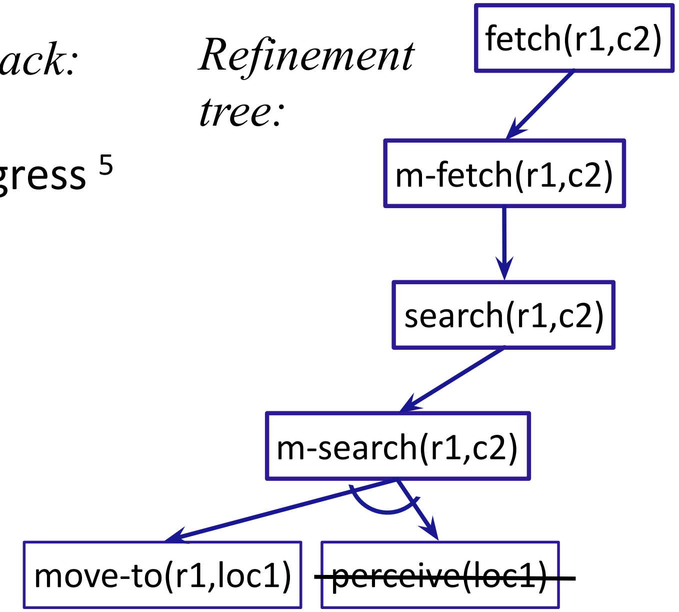
r=r1; c=c2

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

Progress($stack$)
  $(\tau, m, i, tried) \leftarrow$ top($stack$)
  if $i \neq$ nil and $m[i]$ is a command then do
    case status($m[i]$)
      running: return
      failure: Retry($stack$); return
      done: continue
  if i is the last step of m then
      pop($stack$)   // remove $stack$'s top element
  else do
    $i \leftarrow$ nextstep(m, i)
    case type($m[i]$)
      assignment: update ξ according to $m[i]$; return
      command: trigger command $m[i]$; return
      task or goal: continue
    $\tau' \leftarrow m[i]$
    $Candidates \leftarrow$ Instances($\mathcal{M}, \tau', \xi$)
    if $Candidates = \varnothing$ then Retry($stack$)
    else do
      arbitrarily choose $m' \in Candidates$
      $stack \leftarrow$ push($(\tau', m', \text{nil}, \varnothing), stack$)

Call stack:
RAE
Progress [5]

Refinement tree:



- *stack*
  = ⟨(search(r1,c2), m-search(r1,c2), i3, ∅),
  (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)⟩
- m[i3] = perceive(loc1)
- status(m[i3]) = failure (*sensor failure*)
- *call* Retry

i

r=r1; c=c2

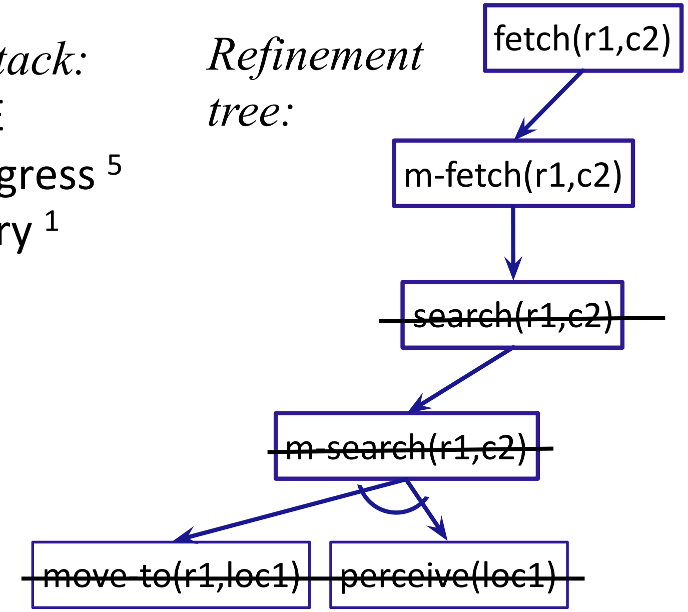
m-search(r,c)
  task: search(r,c)
  pre: pos(c) = unknown
  body: if ∃ l (view(l)=F) then
      move-to(r,l)
      perceive(l)
      if pos(c)=l then take(r,c,l) else search(r,c)
    else fail

Retry($stack$)

$(\tau, m, i, tried) \leftarrow$ pop($stack$)
$tried \leftarrow tried \cup \{m\}$
$Candidates \leftarrow$ Instances($\mathcal{M}, \tau, \xi$)$\setminus$ $tried$
if $Candidates \neq \varnothing$ then do
  arbitrarily choose $m' \in Candidates$
    $stack \leftarrow$ push((τ, m', nil, $tried$),$stack$)
else do
  if $stack \neq \varnothing$ then Retry($stack$)
  else do
    output("failed to accomplish" τ)
    $Agenda \leftarrow Agenda \setminus stack$

Call stack:
RAE
Progress 5
Retry 1

Refinement tree:



- $stack$ = ⟨~~(search(r1,c2), m-search(r1,c2), i3, ∅)~~, (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)⟩
- (τ, m, i, $tried$) ← (search(r1,c2), m-search(r1,c2), i3, $\boxed{\varnothing}$)
- $tried$ ← {m-search(r1,c2)}
- Instances($\mathcal{M}$, τ, ξ) ={m-search(r1,c2)}
- $Candidates$ ←
  Instances($\mathcal{M}$, τ, ξ) \ $tried$
  = ∅
- *Call* Retry($stack$)

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

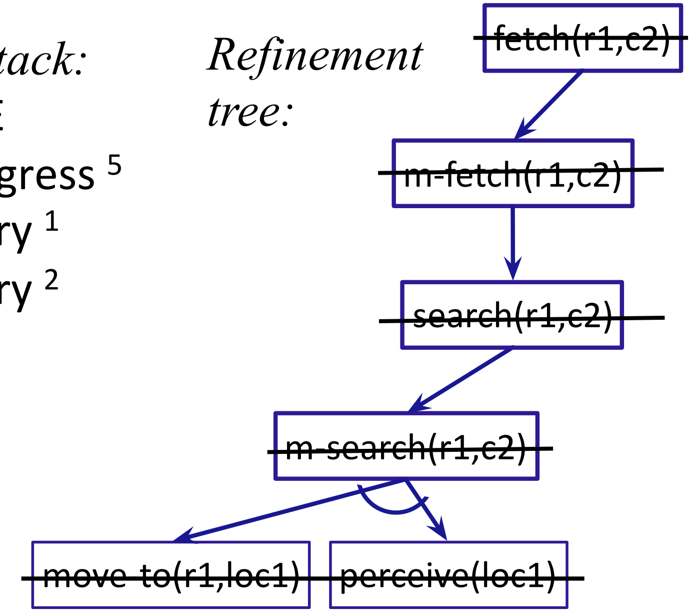
Retry($stack$)

$(\tau, m, i, tried) \leftarrow$ pop($stack$)
$tried \leftarrow tried \cup \{m\}$
$Candidates \leftarrow$ Instances($\mathcal{M}, \tau, \xi$)$\setminus tried$

if $Candidates \neq \varnothing$ then do
    arbitrarily choose $m' \in Candidates$
        $stack \leftarrow$ push((τ, m', nil, $tried$),$stack$)
else do
    if $stack \neq \varnothing$ then Retry($stack$)
    else do
        output("failed to accomplish" τ)
        $Agenda \leftarrow Agenda \setminus stack$

Call stack:
RAE
Progress [5]
Retry [1]
Retry [2]

Refinement tree:



- *stack* = ⟨~~(fetch(r1,c2), m-fetch(r1,c2), i1, ∅)~~⟩
- (τ, m, i, *tried*)
  ← (fetch(r1,c2), m-fetch(r1,c2), i1, ∅)
- *tried* ← {m-fetch(r1,c2)}
- Instances($\mathcal{M}$, τ, ξ) = {m-fetch(r1,c2)}
- *Candidates* ← Instances($\mathcal{M}$, τ, ξ) \ *tried* = ∅
- output("failed to accomplish" fetch(r1,c2))
- *Return to first* Retry
  - *Return to* Progress
    - *Return to* RAE

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))

Example

Call stack:
RAE

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

- *Agenda* $= \varnothing$
- *repeatedly execute loop, waiting for input*

Goals in RAE

- Special kind of task: achieve(*condition*)
  - Like other tasks, but includes monitoring
- Can easily modify Progress to do this
  - Stop executing body(m) early if *condition* becomes true
  - fail if *condition* isn't true after executing body(m)
    - triggers a Retry
  - m-fetch(r,c)
    task: fetch(r,c)
    pre:
    body: achieve(pos(c) $\neq$ unknown)
    goto(pos(c))
    take(r,c)
  - m-find-where(r,c)
    goal: achieve(pos(c) $\neq$ unknown)
    pre:
    body: while $\exists\, l$ (view(l)=F) do
    goto(l)
    perceive(l)

- What we had before:

```
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(l)=F) then
               move-to(r,l); perceive(l)
               if pos(c)=l then take(r,c,l)
               else search(r,c)
         else fail
```

Extensions to RAE

- Section 3.2.4 discusses additional extensions to RAE:
  - Controlling the progress of tasks, e.g., suspend a task for a while
  - Concurrent subtasks
    - m-foo(x,y,z)
      task: foo(x,y,z)
      pre:
      body: . . .
      $\{\text{concurrent: } \langle \nu_{1,1},\ldots,\nu_{1,n}\rangle\ \langle \nu_{2,1},\ldots,\nu_{2,m}\rangle\ \ldots\ \langle \nu_{k,1},\ldots,\nu_{k,l}\rangle\}$
    - Each $\langle \nu_{i,1},\ldots,\nu_{i,j}\rangle$ is a sequence of steps, like the body of a method
    - Split the current stack into k sub-stacks, one for each $\langle \nu_{i,1},\ldots,\nu_{i,j}\rangle$
  - If there are multiple stacks, which ones get higher priority?
    - Application-specific heuristics
  - For a task t, which method to try first?
    - Refinement planning (Section 3.3)

3.3 Refinement planning (SeRPE)

- When dealing with an event or task, RAE may need to make either/or choices
  - $Agenda = \{\langle\tau_1, ...\rangle, \langle\tau_2, ...\rangle, \ldots, \langle\tau_n, ...\rangle\}$
    - Several tasks/events, which one to address first?
  - $Candidates = \{m_1, m_2, \ldots, \}$
    - Several candidate methods or commands, which one to try first?
- RAE immediately executes commands
  - Bad choices may be costly or irreversible

- Look ahead to predict the consequences
  - Instead of executing commands immediately,
    simulate effects using descriptive action models (like in Chap. 2)

Sequential Refinement Planning Engine (SeRPE)

- Refines tasks similarly to RAE, except:
  - ➢ Where RAE executes a command that changes the world state, SeRPE instead uses a descriptive action model to predict the next world state

Restrictive assumptions:

- All action models need to have deterministic outcomes (as in Chapter 2)
  - ➢ e.g., predict exactly what grasp, turn, pull will do
- Don't allow concurrent tasks
  - {concurrent: $\langle v_{1,1},\ldots,v_{1,n}\rangle\ \langle v_{2,1},\ldots,v_{2,m}\rangle\ \ldots\ \langle v_{k,1},\ldots,v_{k,l}\rangle$}
  - ➢ We'll relax this restriction later

m1-unlatch(r,d,l,o)
  task: unlatch(r,d)
  pre: …
  body: grasp(r,o)
    turn(r,o,alpha1)
    pull(r,val1)
    if door-status(d)=cracked
    then ungrasp(r,o)
    else fail

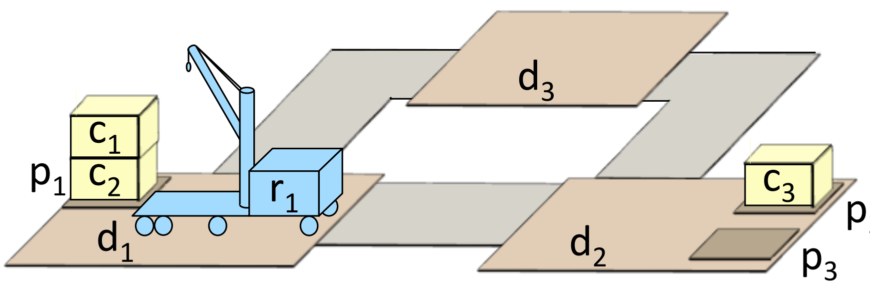
Planning Problem

- Planning problem
  $P = (\mathcal{M},\mathcal{A},s_0,\tau)$
  - $\mathcal{M}$: refinement methods
  - $\mathcal{A}$: action templates
  - s_0: initial state
  - τ: task or goal
    - put-in-pile(c_1,p_2)

Example:

Rigid relations

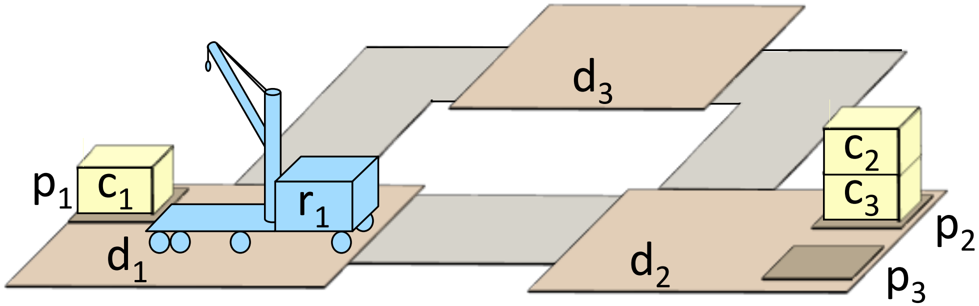
- at(p_1,d_1), at(p_2,d_2), at(p_3,d_2)
- adjacent(d_i ,d_j) for every i, j



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}

Action Templates

- From Example 2.12:



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)←c, pile(c)←nil, pos(c)←r, top(p)←c'

unload(r,c,c',p,d)

  pre: at(p,d), pos(c)=r, loc(r)=d, top(p)=c'

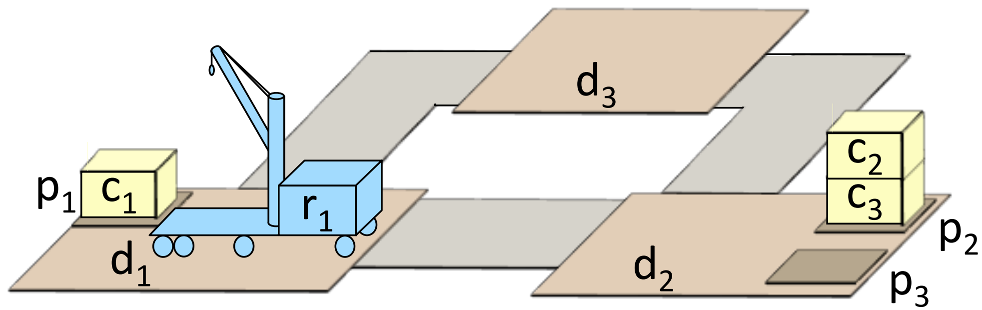
  eff: cargo(r)←nil, pile(c)←p, pos(c)←c', top(p)←c

move(r,d,d')

  pre: adjacent(d,d'), loc(r)=d, occupied(d')=F

  eff: loc(r)=d', occupied(d)=F, occupied(d')=T

Tasks and Methods



put-in-pile(c, p') – put c into pile p' if it isn't already there

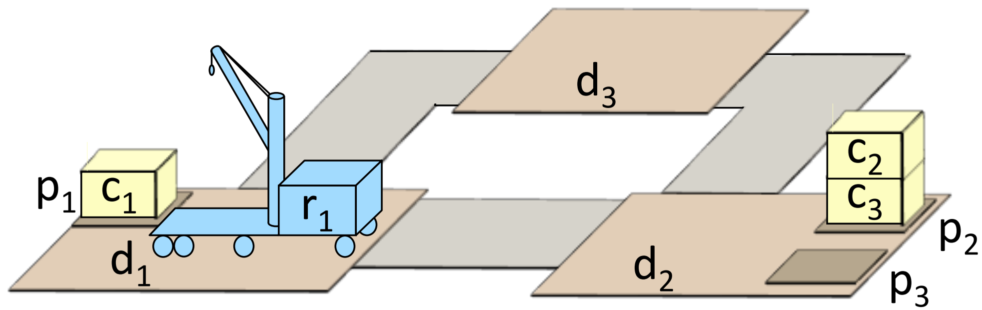
m1-put-in-pile(c,p')

- 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') ∧ $p \neq p'$ ∧ cargo(r)=nil
- body: if loc(r) $\neq d$ then navigate(r,d)
  uncover(c)
  load(r, c, pos(c), p, d)
  if loc(r) $\neq d'$ then navigate(r,d')
  unload(r, c, top(p'), p', d)

Tasks and Methods



uncover(c) – ensure no containers are on c

m1-uncover(c)

task: uncover(c)

pre: top(pile(c))=c

body: // *empty*

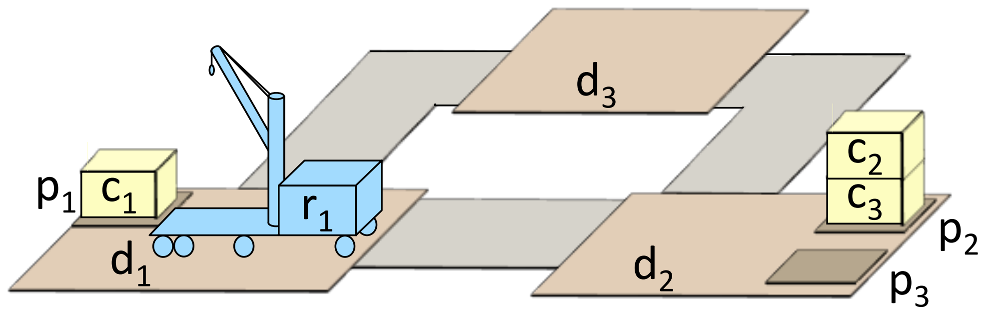
m2-uncover(r,c,c,p',d)

task: uncover(c)

pre: pile(c)=p ∧ top(p)≠c
∧ at(p,d) ∧ at(p',d) ∧ $p' \neq p$
∧ 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)

Tasks and Methods



navigate(r, d') – move r along some route that ends at loading dock d'

m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d'
- body: // *empty*

m2-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))
- body: move(r, loc(r), d')

m3-navigate(r,d,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ $d \neq d'$ ∧ adjacent(loc(r),d))
- body: move(r, loc(r), d)
  navigate(r, d')

SeRPE Algorithm

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 τ is a goal and $s \models \tau$ then return π
    if i is the last step of m then
      if τ is a goal and $s \not\models \tau$ then return failure
      return π
    $i \leftarrow$ 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
        $s \leftarrow \gamma(s, \pi')$; $\pi \leftarrow \pi.\pi'$

Example

τ = put-in-pile(c_1,p_2)

Candidates = {~~m1-put-in-pile(c_1,p_2)~~,
m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2), …}

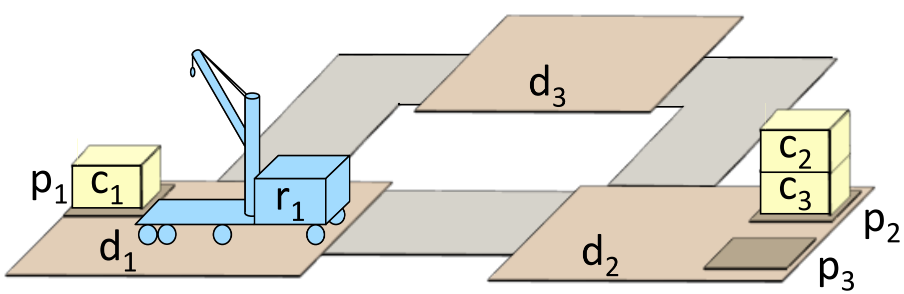
Here, implementation would use a lifted expression m2-put-in-pile(r,c_1,p_1,d,p',d'). It would instantiate variables here:

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($\mathcal{M}, \mathcal{A}, s, \tau, m$)

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}

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)
    ∧ $p \neq p'$ ∧ cargo(r)=nil
  body: if loc(r) $\neq d$ then navigate(r,d)
    uncover(c)
    load(r, c, pos(c), p, d)
    if loc(r) $\neq d'$ then
      navigate(r,d')
    unload(r, c, top(p'), p', d)



Example

task
put-in-pile(c_1,p_2)

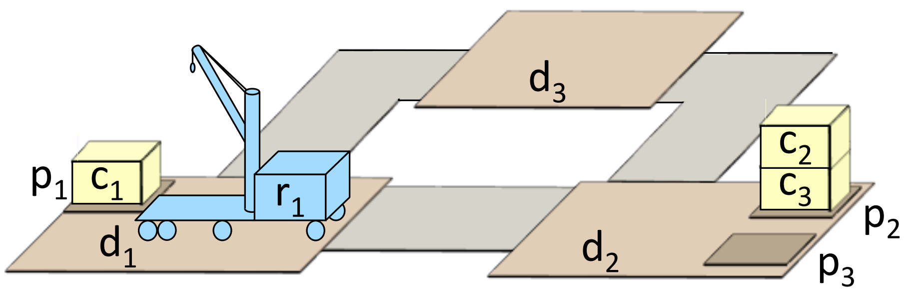
method
m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2)

Refinement tree

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($\mathcal{M}, \mathcal{A}, s, \tau, m$)

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)
    ∧ $p \neq p'$ ∧ cargo(r)=nil
  body: if loc(r) $\neq d$ then navigate(r,d)
    uncover(c)
    load(r, c, pos(c), p, d)
    if loc(r) $\neq d'$ 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}

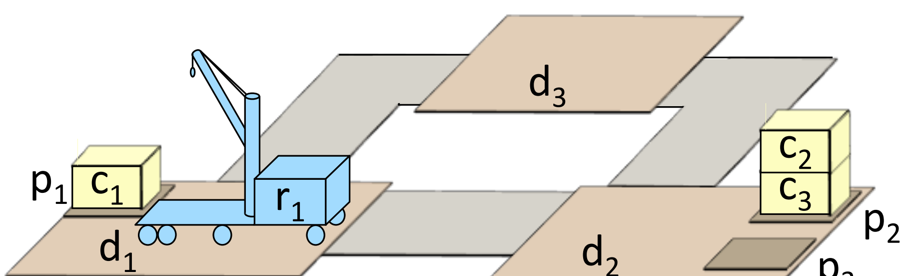


Example

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 τ is a goal and $s \models \tau$ then return π
    if i is the last step of m then
      if τ is a goal and $s \not\models \tau$ then return failure
      return π
    $i \leftarrow$ 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
        $s \leftarrow \gamma(s, \pi')$; $\pi \leftarrow \pi.\pi'$

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

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)
    ∧ $p \neq p'$ ∧ cargo(r)=nil
  body: if loc(r) $\neq d$ then navigate(r,d)
    uncover(c)
    load(r, c, pos(c), p, d)
    if loc(r) $\neq d'$ then
      navigate(r,d')
    unload(r, c, top(p'), p', d)



Example

task
put-in-pile(c_1,p_2)

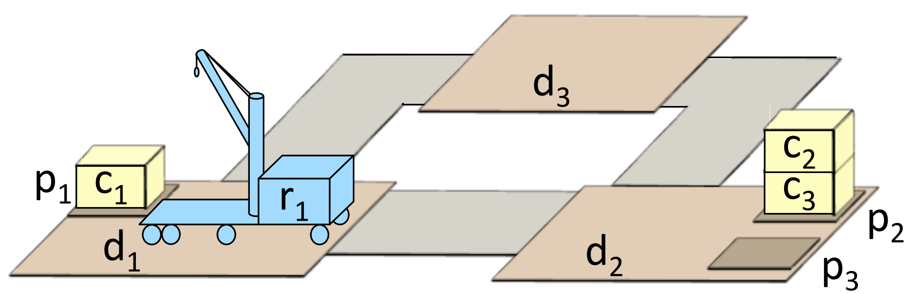
method
m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2)

task
uncover(c_1)

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) ∧ $p \neq p'$ ∧ cargo(r)=nil
body: if loc(r) $\neq d$ then navigate(r,d)
$\boxed{\text{uncover}(c)}$
load(r, c, pos(c), p, d)
if loc(r) $\neq d'$ 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)

task

uncover(c_1)

method

m1-uncover(c_1)

(no children)

m1-uncover(c)

task: uncover(c)

pre: $\boxed{\text{top(pile}(c))=c}$

body: // *empty*

m2-uncover(r,c,c,p',d)

task: uncover(c)

move(r_1,d_1,d_2)

pre: pile(c)=p ∧ top(p)≠c ∧ …

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) ∧ $p \neq p'$ ∧ cargo(r)=nil

body: if loc(r) $\neq d$ then navigate(r,d)

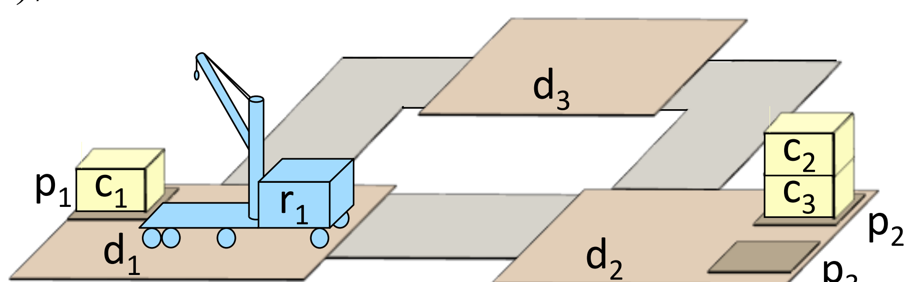
$\boxed{\text{uncover}(c)}$

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

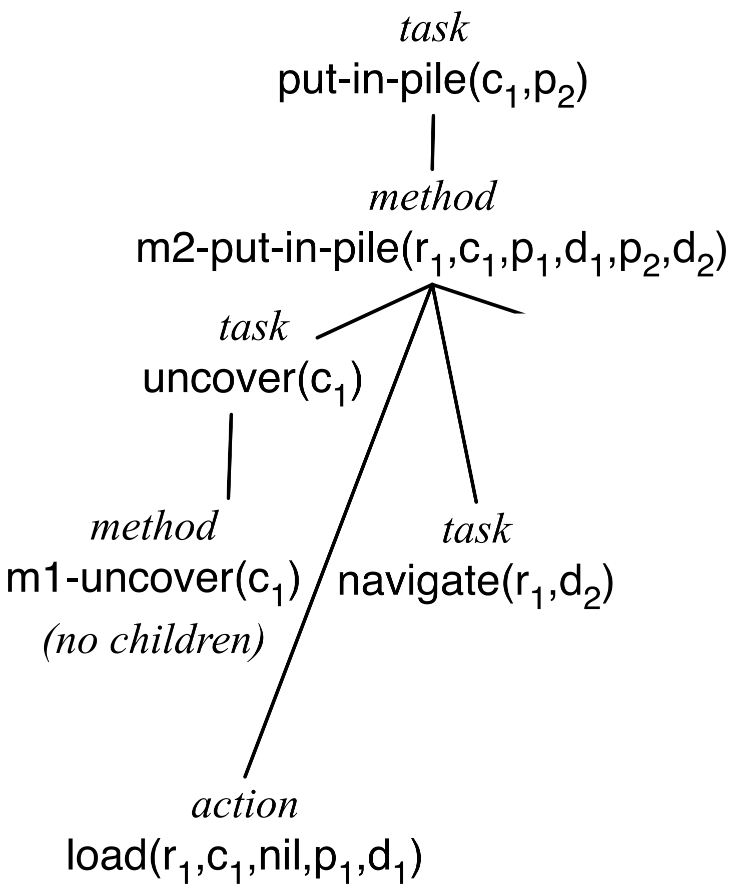
if loc(r) $\neq d'$ then

navigate(r,d')

unload(r, c, top(p'), p', d)



Example



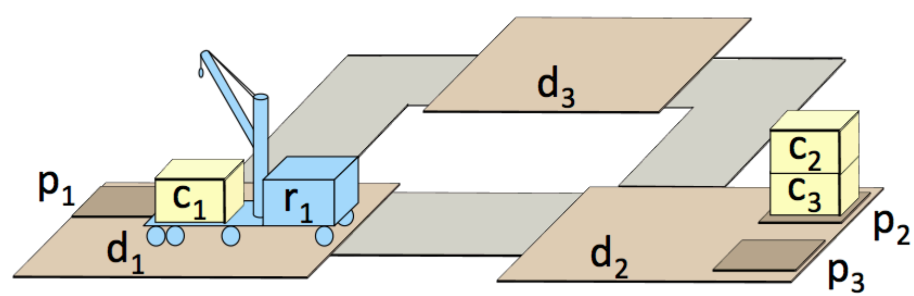
$r_1, c_1, p_1, d_1, p_2, d_2$

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

task: put-in-pile(c,p')

pre: pile(c)=p ∧ at(p,d) ∧ at(p',d) ∧ $p \neq p'$ ∧ cargo(r)=nil

body: if loc(r) $\neq d$ then navigate(r,d)
uncover(c)
load(r, c, pos(c), p, d)
if loc(r) $\neq d'$ 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)

task
navigate(r_1,d_2)

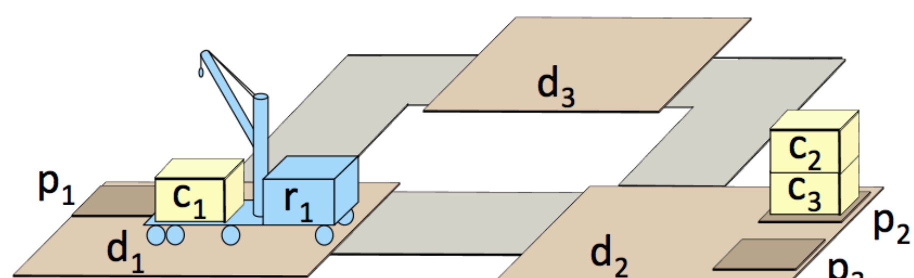
action
load(r_1,c_1,nil,p_1,d_1)

Candidates ={…,
m2-navigate(r_1,d_2), …,
m3-navigate(r_1,d_3,d_2), …}

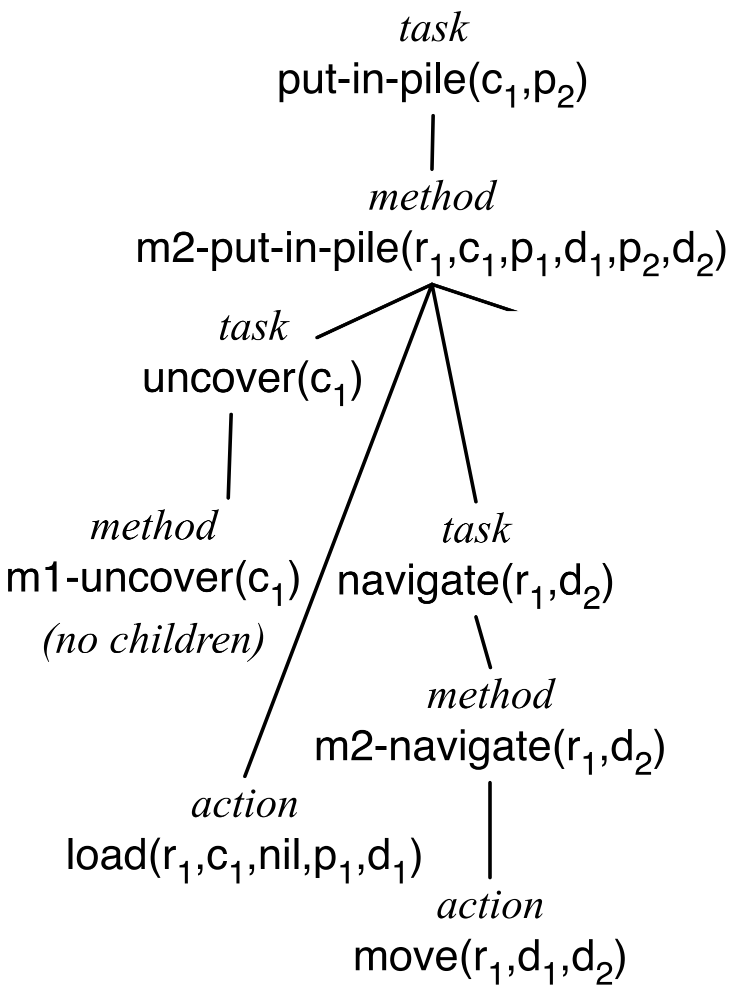
m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d'
- body: *// empty*

m2-navigate(r,d') r_1,d_2
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))
- body: move(r, loc(r), d')

m3-navigate(r,d,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ $d \neq d'$
  ∧ adjacent(loc(r),d))
- body: move(r, loc(r), d)
  navigate(r, d')



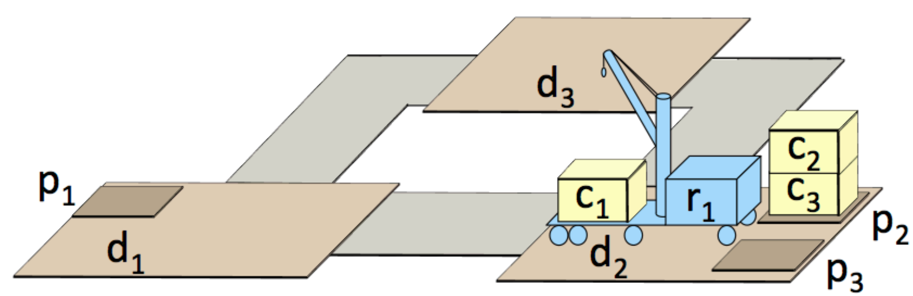
Example



m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d'
- body: *// empty*

m2-navigate(r,d') r_1,d_2
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))
- body: $\boxed{\text{move}(r, \text{loc}(r), d')}$

m3-navigate(r,d,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ $d \neq d'$ ∧ adjacent(loc(r),d))
- body: move(r, loc(r), d)
  navigate(r, 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)
    - *task* navigate(r_1,d_2)
      - *method* m2-navigate(r_1,d_2)
        - *action* move(r_1,d_1,d_2)
    - *action* unload(r_1,c_1,c_3,p_2,d_2)

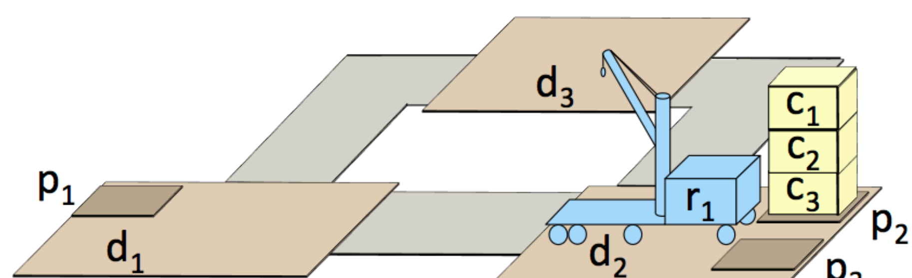
r_1,c_1,p_1,d_1,p_2,d_2

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) ∧ $p \neq p'$ ∧ cargo(r)=nil

body: if loc(r) $\neq d$ then navigate(r,d)
uncover(c)
load(r, c, pos(c), p, d)
if loc(r) $\neq d'$ then
navigate(r,d')
$\boxed{\text{unload}(r, c, \text{top}(p'), p', d)}$



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)

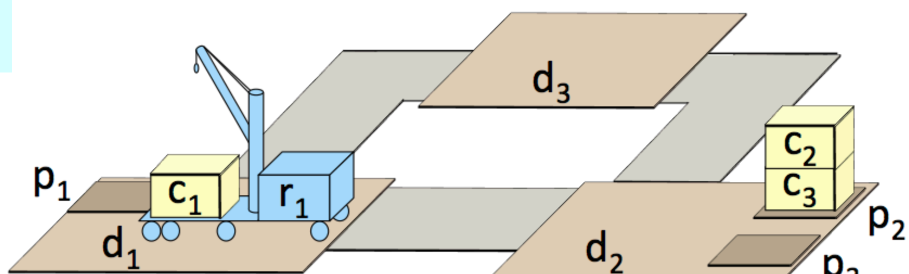
task
navigate(r_1,d_2)

Candidates ={…,
m2-navigate(r_1,d_2), …,
m3-navigate(r_1,d_3,d_2), …}

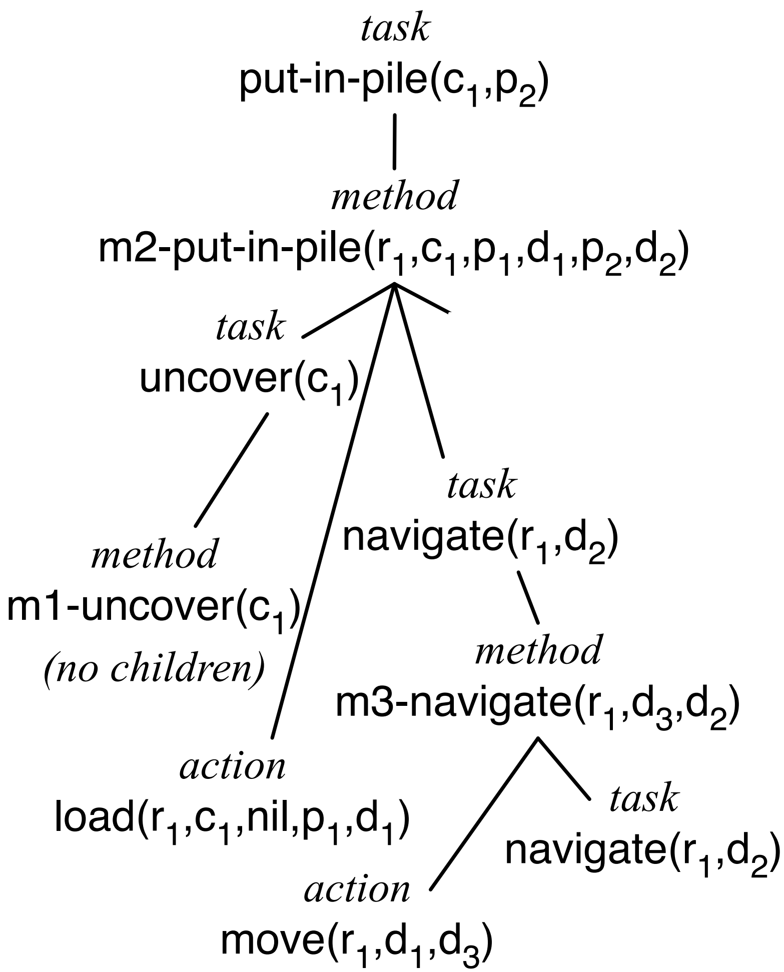
m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)$=d'$
- body: // *empty*

m2-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))
- body: move(r, loc(r), d')

m3-navigate(r,d,d') r_1,d_3,d_2
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ $d \neq d'$
  ∧ adjacent(loc(r),d))
- body: move(r, loc(r), d)
  navigate(r, d')



Example



m1-navigate(r,d')

task: navigate(r, d')

pre: loc(r)=d'

body: *// empty*

m2-navigate(r,d')

task: navigate(r, d')

pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))

body: move(r, loc(r), d')

m3-navigate(r,d,d') r_1,d_3,d_2

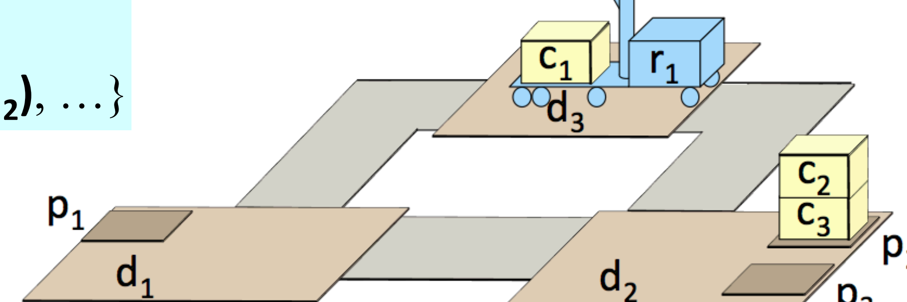
task: navigate(r, d')

pre: loc(r)$\neq d'$ ∧ $d \neq d'$

∧ adjacent(loc(r),d))

body: move(r, loc(r), d)
navigate(r, d')

Candidates ={…,
m2-navigate(r_1,d_2), …}



Example

task
put-in-pile(c_1,p_2)

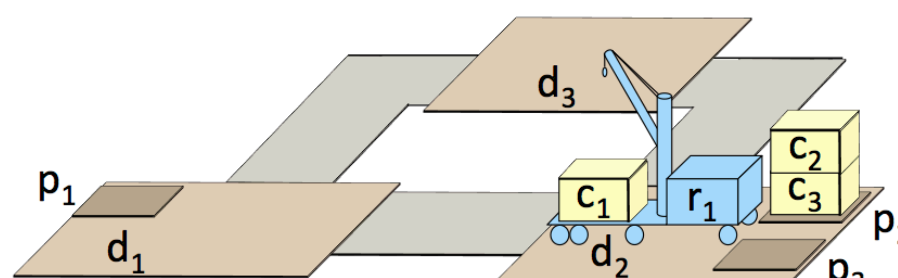
method
m2-put-in-pile(r_1,c_1,p_1,d_1,p_2,d_2)

- *task* uncover(c_1)
  - *method* m1-uncover(c_1) *(no children)*
- *action* load(r_1,c_1,nil,p_1,d_1)
- *task* navigate(r_1,d_2)
  - *method* m3-navigate(r_1,d_3,d_2)
    - *action* move(r_1,d_1,d_3)
    - *task* navigate(r_1,d_2)
      - *method* m2-navigate(r_1,d_2)
        - *action* move(r_1,d_3,d_2)

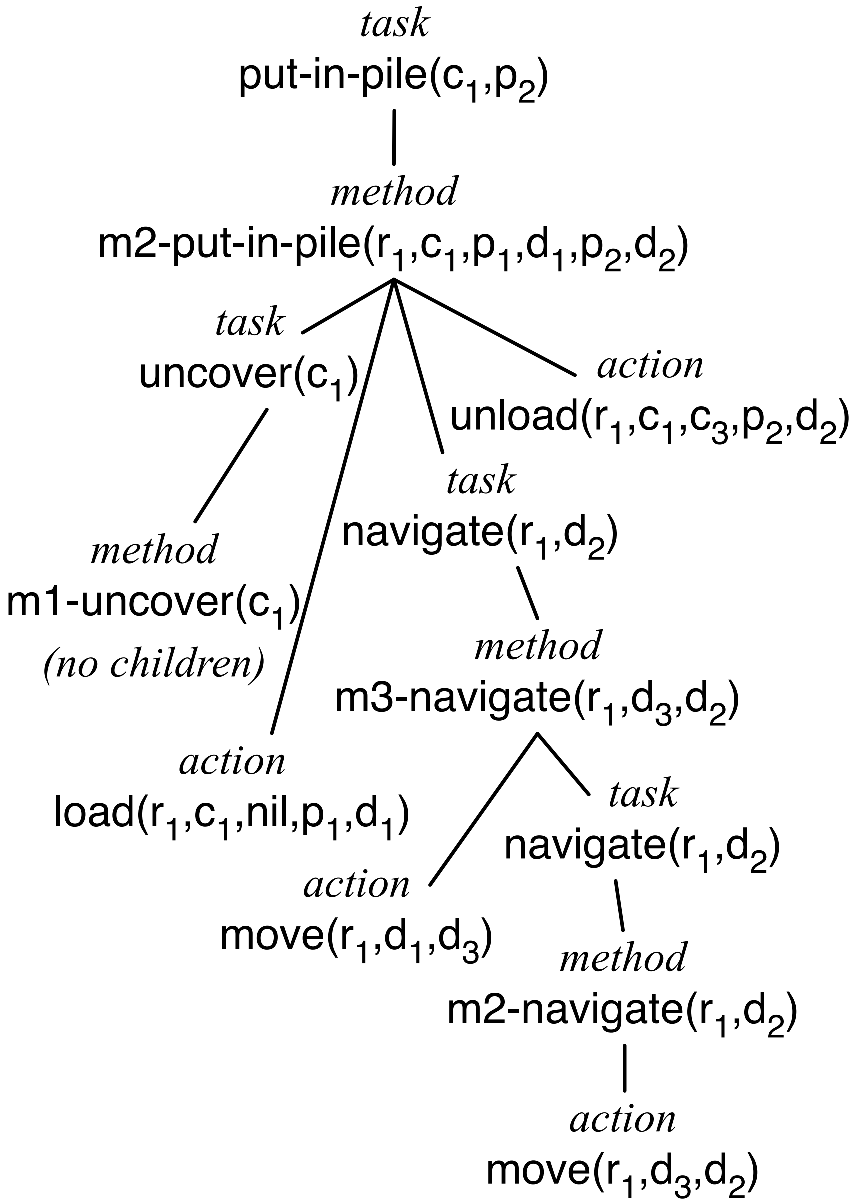
m1-navigate(r,d')
- task: navigate(r, d')
- pre: loc(r)=d'
- body: // *empty*

m2-navigate(r,d') r_1,d_2
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ adjacent(loc(r),d'))
- body: $\boxed{\text{move}(r, \text{loc}(r), d')}$

m3-navigate(r,d,d')
- task: navigate(r, d')
- pre: loc(r)$\neq d'$ ∧ $d \neq d'$
  ∧ adjacent(loc(r),d))
- body: move(r, loc(r), d)
  navigate(r, d')



Example



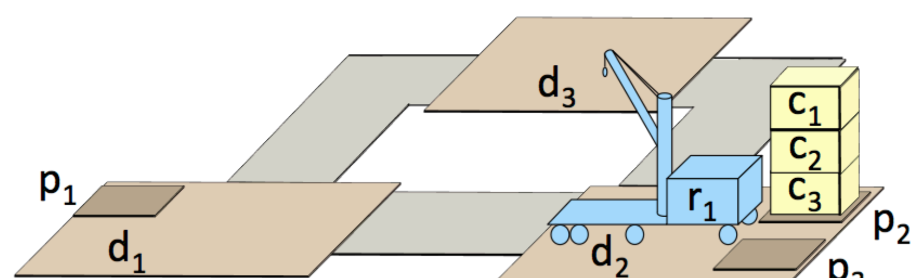
r_1,c_1,p_1,d_1,p_2,d_2

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

task: put-in-pile(c,p')

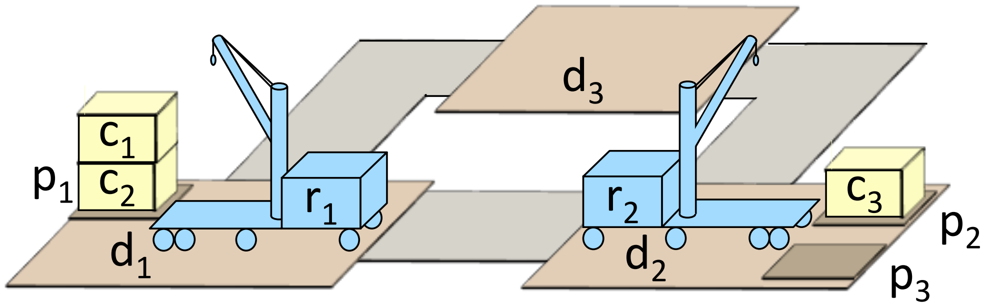
pre: pile(c)=p ∧ at(p,d) ∧ at(p',d) ∧ $p \neq p'$ ∧ cargo(r)=nil

body: if loc(r) $\neq d$ then navigate(r,d)
uncover(c)
load(r, c, pos(c), p, d)
if loc(r) $\neq d'$ then
navigate(r,d')
$\boxed{\text{unload}(r, c, \text{top}(p'), p', d)}$



Interleaved Refinement Tree (IRT) Procedure

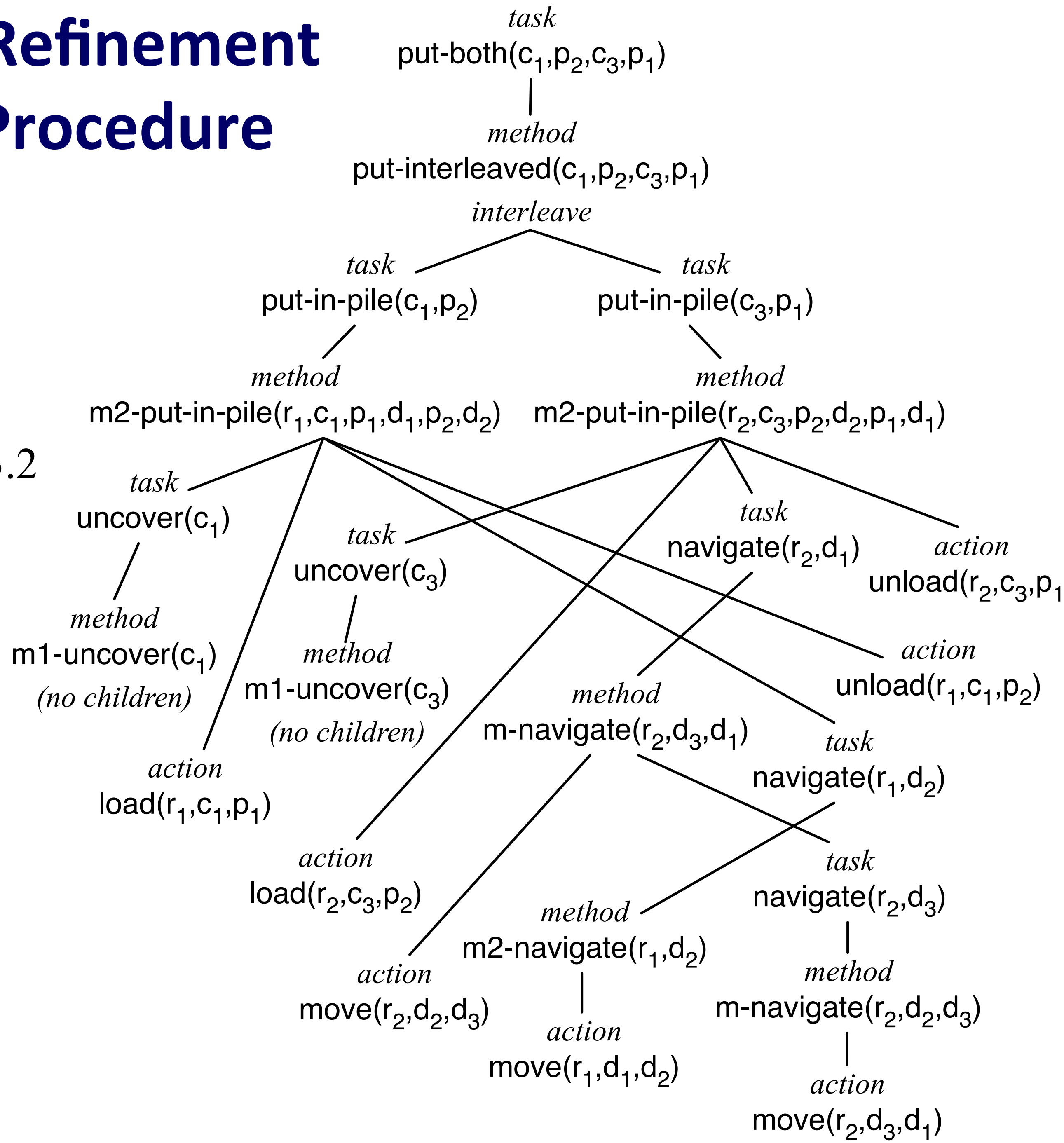
- SeRPE doesn't allow the 'concurrent' programming construct



- Assume same initial state as before
- Want to move c_1 to p_2, and c_3 to p_1 using these plans:
  $\langle$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)$\rangle$
  $\langle$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,d1)$\rangle$
- Need to interleave, otherwise they'll fail

Interleaved Refinement Tree (IRT) Procedure

- IRT extends SeRPE to interleave plans for different tasks
- Details: Section 3.3.2



3.4 Acting and Refinement Planning

- (to be added)