Domain Model

State-transition system or *classical planning domain*:

- $\Sigma = (S,A,\gamma,\text{cost})$ or (S,A,γ)
  - S - finite set of *states*
  - A - finite set of *actions*
  - $\gamma: S \times A \rightarrow S$

    prediction (or *state-transition*) function
    - *partial* function: $\gamma(s,a)$ is not necessarily defined for every (s,a)
      - a is *applicable* in s iff $\gamma(s,a)$ is defined
      - $\text{Domain}(a) = \{s \in S \mid a \text{ is applicable in } s\}$
      - $\text{Range}(a) = \{\gamma(s,a) \mid s \in \text{Domain}(a)\}$
  - cost: $S \times A \rightarrow \mathbb{R}^+$ or cost: $A \rightarrow \mathbb{R}^+$
    - optional; default is $\text{cost}(a) \equiv 1$
    - money, time, something else

- *plan*:
  - a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$
- π is *applicable* in s_0 if the actions are applicable in the order given

  $\gamma(s_0, a_1) = s_1$

  $\gamma(s_1, a_2) = s_2$

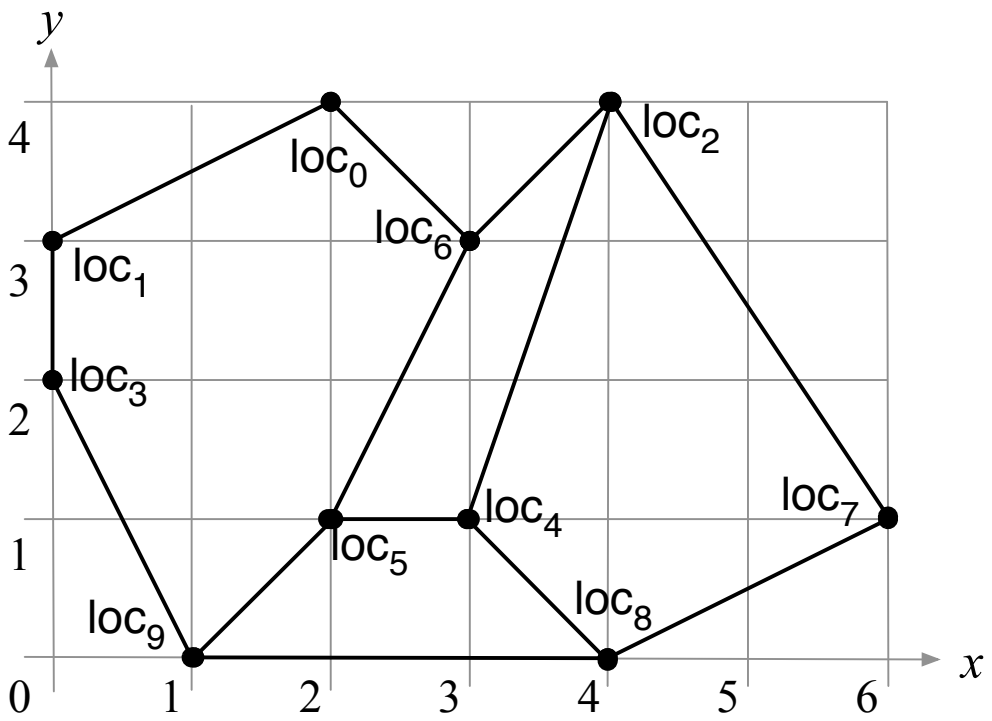
  $\ldots$

  $\gamma(s_{n-1}, a_n) = s_n$
  - In this case define $\gamma(s_0, \pi) = s_n$

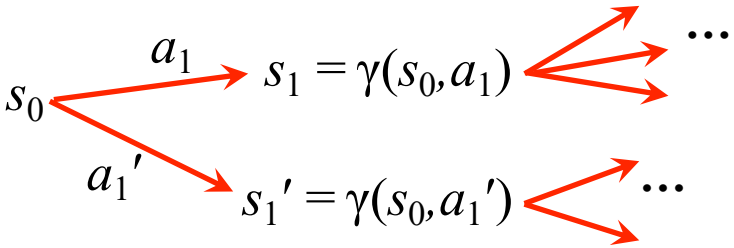
- *Classical planning problem*:
  - $P = (\Sigma, s_0, S_g)$
  - planning domain, initial state, set of goal states
- *Solution* for P:
  - a plan π such that that $\gamma(s_0,\pi) \in S_g$

Representing Σ

- If S and A are small enough
  - Give each state and action a name
  - For each s and a, store $\gamma(s,a)$ in a lookup table



- In larger domains, don't represent all states explicitly
  - Language for describing properties of states
  - Language for describing how each action changes those properties
  - Start with initial state, use actions to produce other states

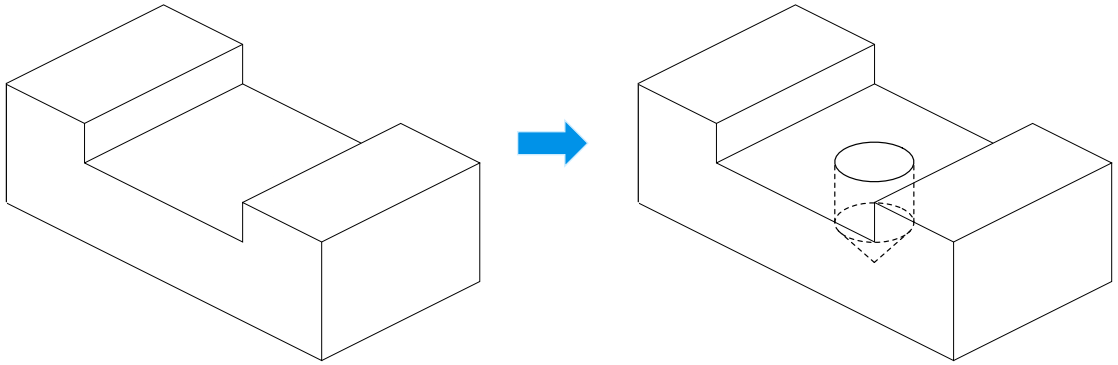


Domain-Specific Representation

- Tailor-made for a specific environment
- State: arbitrary data structure
- Action: (head, preconditions, effects, cost)
  - *head*: name and parameter list
    - Get actions by instantiating the parameters
  - *preconditions*:
    - Computational tests to predict whether an action can be performed
    - Should be necessary/sufficient for the action to run without error
  - *effects*:
    - Procedures that modify the current state
  - *cost*: procedure that returns a number
    - Can be omitted, default is cost ≡ 1

Example

- Drilling holes in a metal workpiece
  - A state
    - geometric model of the workpiece
      - *annotated* with dimensions, tolerances, etc.
    - capabilities and status of drilling machine and drill bit
  - Several actions
    - clamp the workpiece onto the drilling machine
    - load a drill bit into the machine
    - drill a hole



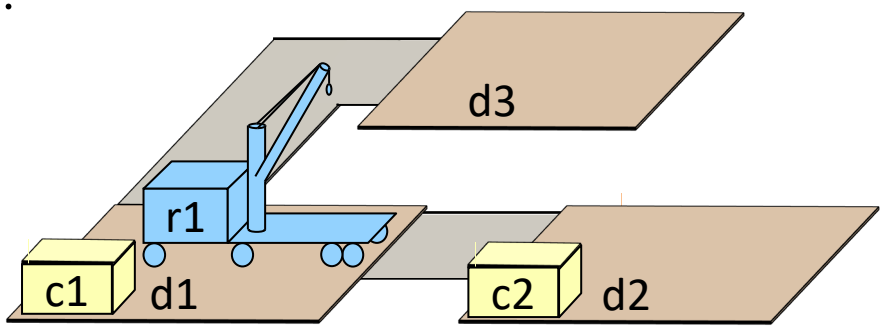
- Name: drill-hole
- Arguments:
  - ID codes for the machine and drill bit
  - annotated geometric model of the workpiece
  - description of the hole to be drilled
- Preconditions
  - *Capabilities*: can the machine and drill bit produce the desired hole?
  - *Current state*: Is the drill bit installed? Is the workpiece clamped onto the table? Etc.
- Effects
  - annotated geometric model of modified workpiece
- Cost
  - estimate of time or monetary cost

Discussion

- Advantage of domain-specific representation:
  - use whatever works best for that particular domain
- Disadvantage:
  - for each new domain, need new representation and deliberation algorithms
- Alternative: *domain-independent* representation
  - Try to create a "standard format" that can be used for many different planning domains
  - Deliberation algorithms that work for anything in this format
- *State-variable* representation
  - Simple formats for describing states and actions
  - Limited representational capability
    - But easy to compute, easy to reason about
  - Domain-independent search algorithms and heuristic functions that can be used in all state-variable planning problems

State-Variable Representation

- *E*: environment that we want to represent
- *B*: set of symbols called *objects*
  - names for objects in *E*, mathematical constants, …



- Example
  - $B = Robots \cup Containers \cup Locs \cup \{\mathtt{nil}\}$
    - $Robots = \{\mathtt{r1}\}$
    - $Containers = \{\mathtt{c1}, \mathtt{c2}\}$
    - $Locs = \{\mathtt{d1}, \mathtt{d2}, \mathtt{d3}\}$

- *B* only needs to include objects that matter at the current level of abstraction
- Can omit lots of details
  - physical characteristics of robots, containers, loading docks, roads, …

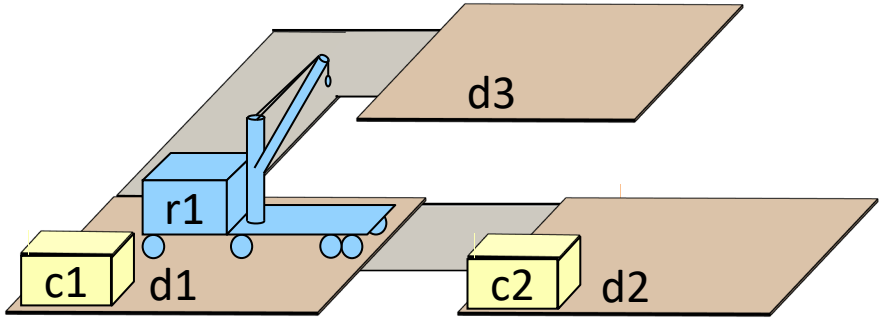
Rigid Properties

- Objects have two kinds of properties
  - *rigid* and *varying*
- *Rigid*: stays the same in every state
  - Can be described as a mathematical relation

    adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}
  - Or equivalently, a set of *ground atoms*

    adjacent(d1,d2), adjacent(d2,d1),

    adjacent(d1,d3), adjacent(d3,d1)
  - I'll use the two notations interchangeably

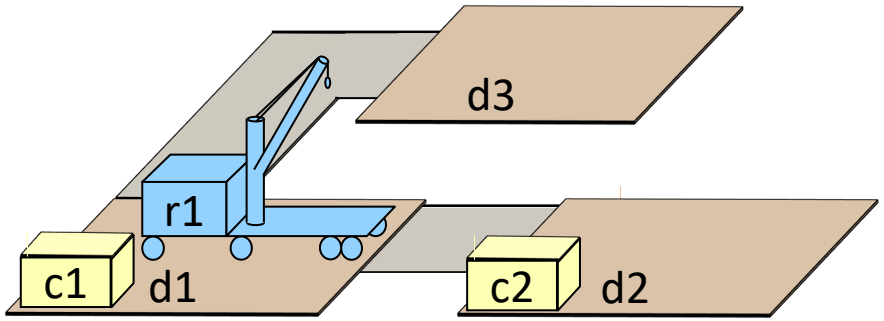


- Terminology from first-order logic:
  - *atom* ≡ *atomic formula* ≡ *positive literal* ≡ predicate symbol with list of arguments
    - *e.g.*, adjacent(x,d2)
  - an atom is *ground* (or *fully instantiated*) if it contains no variable symbols
    - *e.g.*, adjacent(d1,d2)
  - *negative literal* ≡ *negated atom* ≡ atom with a negation sign in front of it
    - *e.g.*, ¬ adjacent(x,d2)

Varying Properties

- *Varying* property (or *fluent*):
  - a property that may differ in different states
- Represent it using a *state variable*
  - a term that we can assign a value to
    - *e.g.,* loc(r1)
- Let X = {all state variables in the environment}

  e.g., X = {loc(r1), loc(c1), loc(c2), cargo(r1)}
- Each state variable $x \in X$ has a *range* = {all values that can be assigned to x}
  - Range(loc(r1)) = *Locs*
  - Range(loc(c1)) = Range(loc(c2)) = *Robots* ∪ *Locs*
  - Range(cargo(r1)) = *Containers* ∪ {nil}
- To abbreviate the "range" notation often I'll just say things like
  - loc(r1) ∈ *Locs*
  - loc(c1), loc(c2) ∈ *Robots* ∪ *Locs*

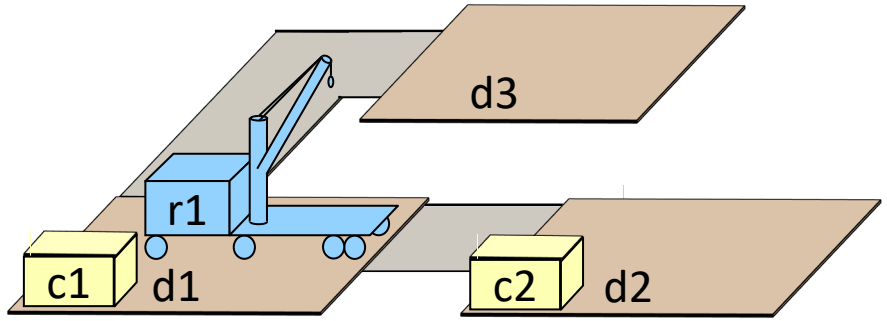


Instead of "domain", to avoid confusion with planning domains

States as Functions

- Represent each state s as a function that assigns values to state variables
  - For each state variable x, $s(x)$ is one x's possible values

s_1(loc(r1)) = d1, s_1(cargo(r1)) = nil,
s_1(loc(c1)) = d1, s_1(loc(c2)) = d2



- Mathematically, a function is a set of ordered pairs

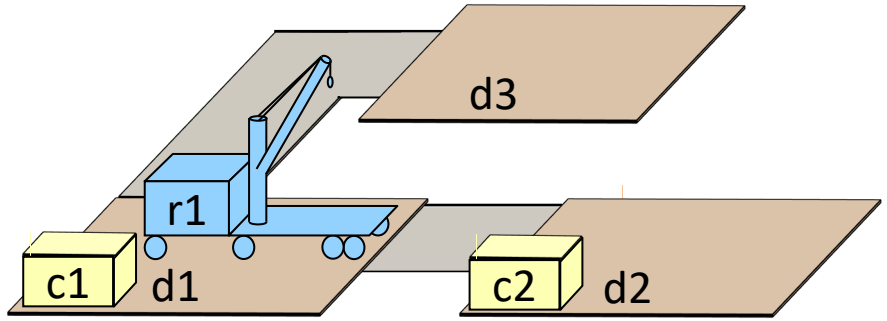
$$s_1 = \{(\text{loc(r1), d1}),\ (\text{cargo(r1), nil}),\ (\text{loc(c1), d1}),\ (\text{loc(c2), d2})\}$$

- Equivalently, write it as a set of *ground positive literals* (or *ground atoms*):

$$s_1 = \{\text{loc(r1)=d1},\ \text{cargo(r1)=nil},\ \text{loc(c1)=d1},\ \text{loc(c2)=d2}\}$$

  - Here, we're using '=' as a predicate symbol

Action Templates



- Action *template* or *schema*: a parameterized set of actions
  α = (head, pre, eff, cost)
  - head: *name, parameters*
  - pre: *precondition* literals
  - eff: *effect* literals
  - cost: *a number* (optional, default is 1)
- e.g.,
  - head = take(r,l,c)
  - pre = {cargo(r)=nil, loc(r)=l, loc(c)=l}
  - eff = {cargo(r)=c, loc(c)=r}
- Each parameter has a range of possible values:
  - Range(r) = *Robots* = {r1}
  - Range(l) = *Locs* = {d1,d2,d3}
  - Range(l) = Range(m) = *Locs* = {d1,d2,d3}
  - Range(c) = *Containers* = {c1,c2}
- But we'll usually write it more like pseudocode

move(r,l,m)
  pre: loc(r)=l, adjacent(l,m)
  eff: loc(r) ← m

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

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

$r \in$ *Robots* = {r1}

$l,m \in$ *Locs* = {d1,d2,d3}

$c \in$ *Containers* = {c1,c2}

Actions

- $\mathcal{A}$ = set of action templates

move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

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

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

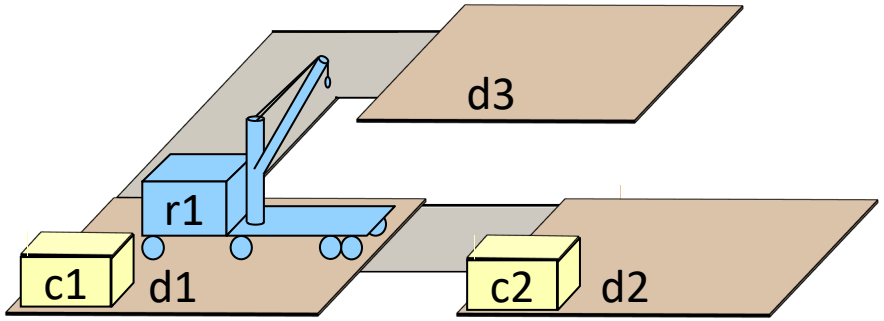
$r \in Robots = \{r1\}$

$l,m \in Locs = \{d1,d2,d3\}$

$c \in Containers = \{c1,c2\}$

- Action: *ground instance* of an $\alpha \in \mathcal{A}$
  - replace each parameter with something in its range
- A = {all actions we can get from $\mathcal{A}$}
  = {all ground instances of members of $\mathcal{A}$}

move(r1,d1,d2)
  pre: loc(r1)=d1, adjacent(d1,d2)
  eff: loc(r1) ← d2



Actions

- $\mathcal{A}$ = set of action templates

move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

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

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

$r \in Robots$ = {r1}

$l,m \in Locs$ = {d1,d2,d3}

$c \in Containers$ = {c1,c2}

- Action: *ground instance* of an $\alpha \in \mathcal{A}$
  - replace each parameter with something in its range
- A = {all actions we can get from $\mathcal{A}$}
  = {all ground instances of members of $\mathcal{A}$}

move(r1,d1,d2)
  pre: loc(r1)=d1, adjacent(d1,d2)
  eff: loc(r1) ← d2

Poll. Let:

$\mathcal{A}$ = {the action templates on this page}

A = {all ground instances of members of $\mathcal{A}$}

How many move actions in A?

Answers:

A. 1	F. 6
B. 2	G. 7
C. 3	H. 8
D. 4	I. 9
E. 5	J. other

Applicability

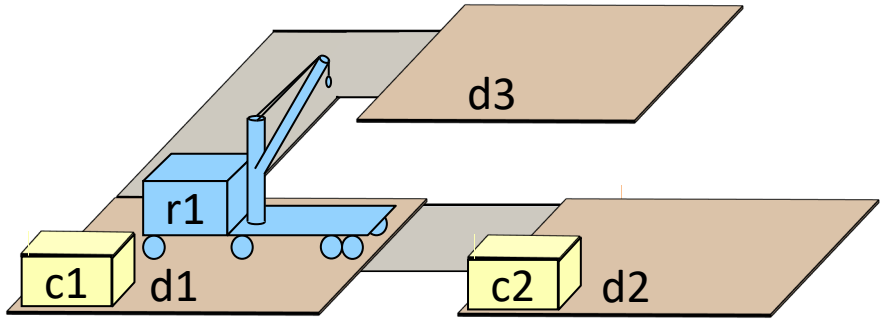
- a is *applicable* in s if
  - for every positive literal $l \in \text{pre}(a)$, $l \in s$ or l is in one of the rigid relations
  - for every negative literal $\neg l \in \text{pre}(a)$, $l \notin s$ and l isn't in any of the rigid relations

- Rigid relation

  adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}

- *State*

  s_1 = {loc(r1)=d1, cargo(r1)=nil, loc(c1)=d1}



- Action template

  move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

  Range(r) = *Robots*

  Range(l) = Range(m) = *Locs*

- Applicable:

  move(r1,d1,d2)
  pre: loc(r1)=d1, adjacent(d1,d2)
  eff: loc(r1) ← d2

- Not applicable:

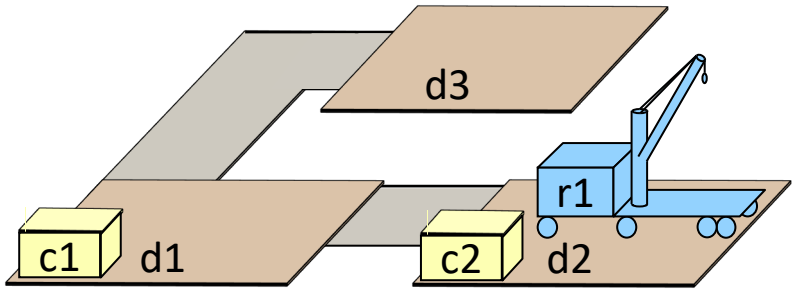
  move(r1,d2,d1)
  pre: loc(r1)=d2, adjacent(d2,d1)
  eff: loc(r1) ← d1

Poll: How many move actions are applicable in s_1?

A. 1	F. 6
B. 2	G. 7
C. 3	H. 8
D. 4	I. 9
E. 5	J. other

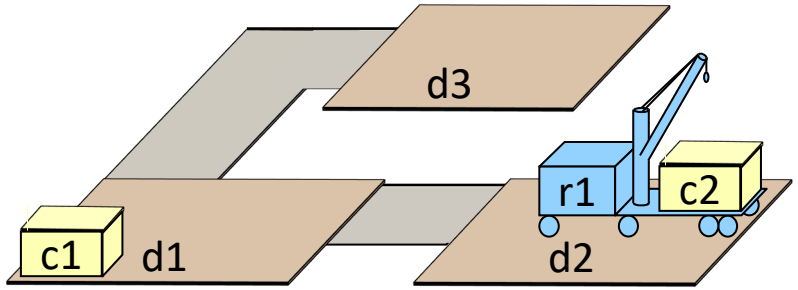
State-Transition Function

- If a is applicable in s:
  - $\gamma(s,a)$ = {every literal in s that isn't changed in eff(a)}
    ∪ {every literal in eff(a)}



- s_2 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d1, loc(c2)=d2}

- a = take(r1,d2,c2)
  pre: cargo(r1)=nil, loc(r1)=d2, loc(c2)=d2
  eff: cargo(r1) ← c2, loc(c2) ← r1



- $\gamma(s_2$, take(r1,d2,c2)) =
  {loc(r1)=d2, loc(c1)=d1, cargo(r1)=c2, loc(c2)=r1}
  from s_2 — from eff(a)

State-Variable Planning Domain

- Let

  B = finite set of objects

  R = finite set of rigid relations over B

  X = finite set of state variables

  - for every state variable x, Range(x) $\subseteq B$

  S = state space over X

  = {all value-assignment functions that have sensible interpretations}

  ?

  $\mathcal{A}$ = finite set of action templates

  - for every parameter y, Range(y) $\subseteq B$

  A = {all ground instances of action templates in $\mathcal{A}$}

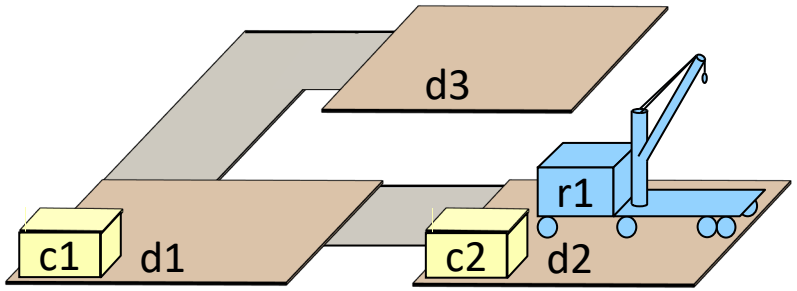
  $\gamma(s,a) = \{(x,w) \mid \text{eff}(a) \text{ contains the effect } x \leftarrow w\}$
  $\cup \{(x,w) \in s \mid x \text{ isn't the target of any effect in eff}(a)\}$

- Then $\Sigma = (S,A,\gamma)$ is a *state-variable planning domain*

Interpretations

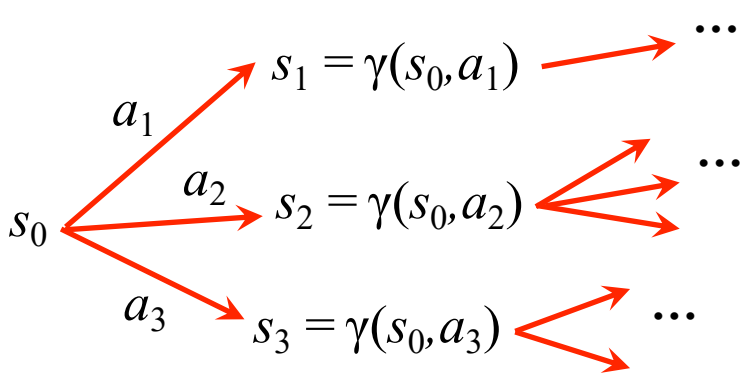
- Let s be a value-assignment function
  - s is a state only if the values make sense in the planning domain we're trying to represent
    - (relation to *model theory*)
- Can loc(c1)=r1 if cargo(r1)=nil?
  - Not in our intended *interpretation*
    - Mapping of symbols to what they represent

s_2 = {loc(r1)=d2,
cargo(r1)=nil,
loc(c1)=d1,
loc(c2)=d2}



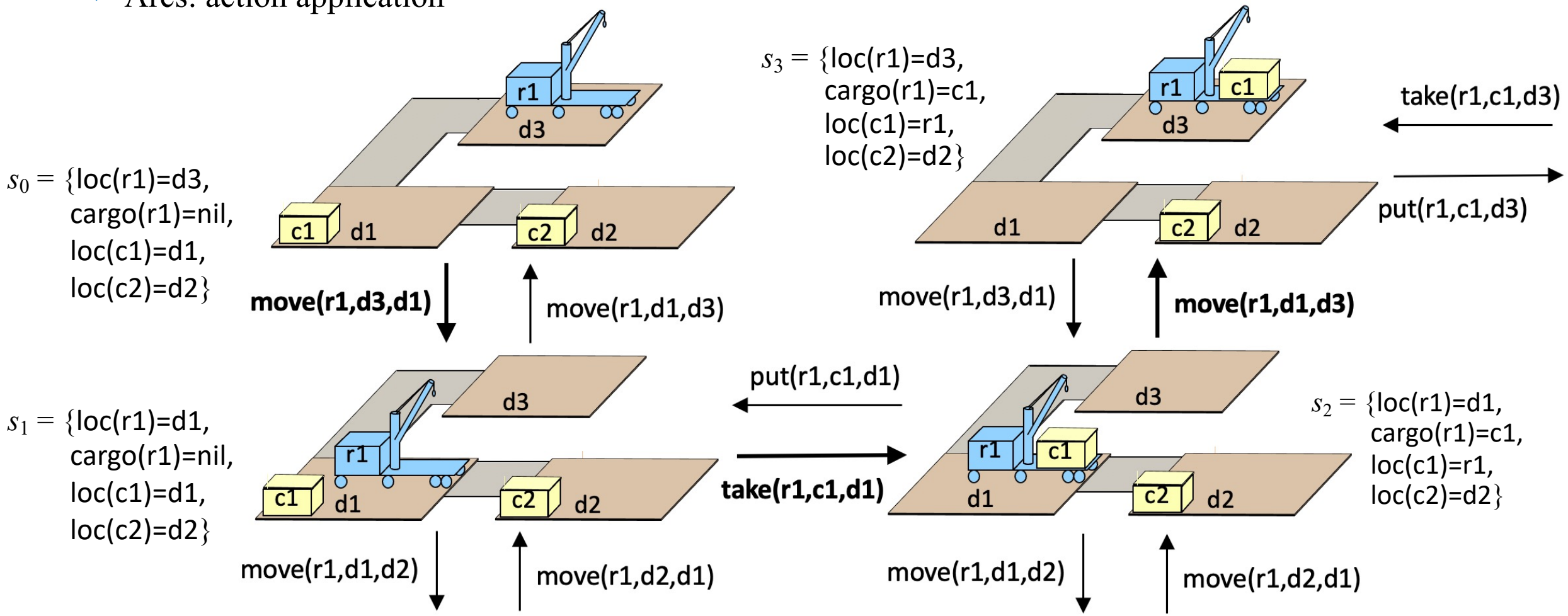
- Can both loc(c1)=r1 and loc(c2)=r1?
  - In our intended interpretation, can a robot carry more than one object at a time?

- How to enforce the intended interpretation?
- Explicitly
  - Mathematical axioms
  - Integrity constraints
- Implicitly
  - Write an initial state s_0 that satisfies the interpretation
  - Write the actions in such a way that whenever s satisfies the interpretation, $\gamma(s,a)$ will too



State Space

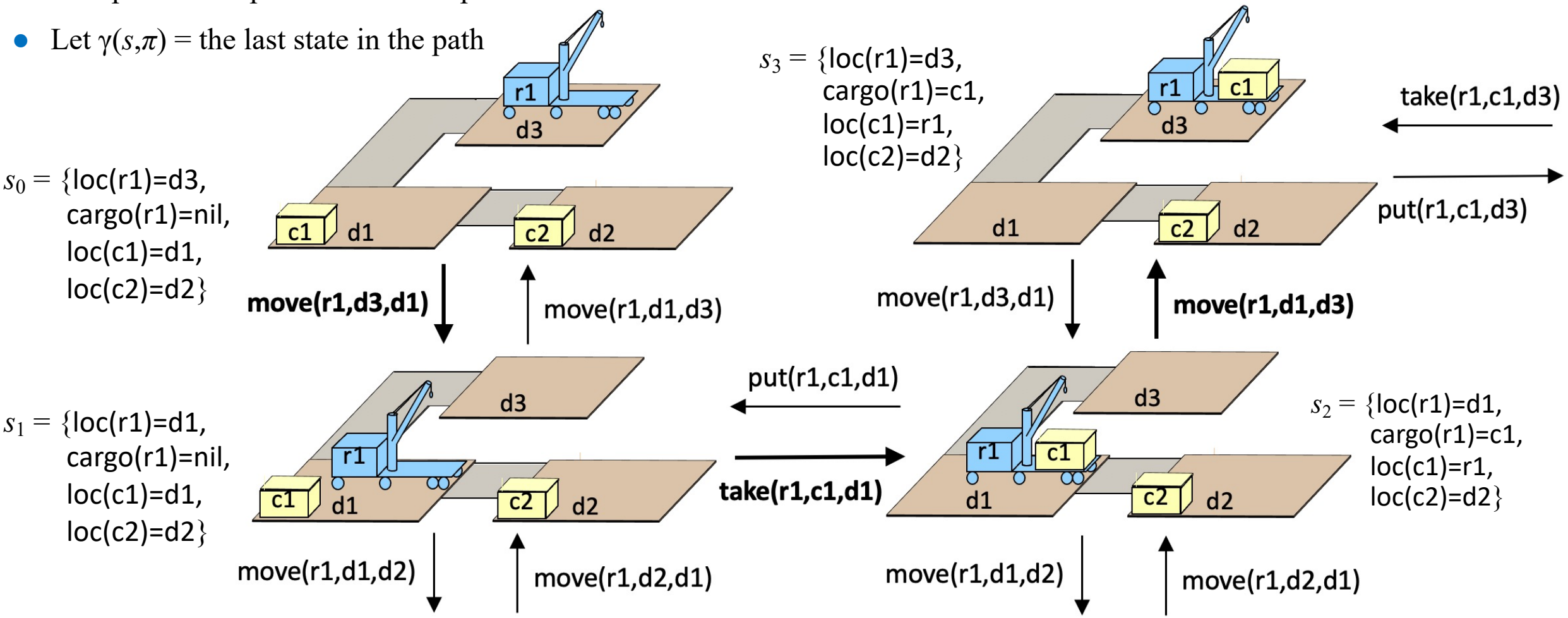
- *State Space*: a directed graph
  - Nodes = states of the world
  - Arcs: action application



Applying a Plan

- A plan π is applicable in a state s if we can apply the actions in the order that they appear in π
- This produces a path in the state space
- Let $\gamma(s,\pi)$ = the last state in the path

- If $\pi = \langle$move(r1,d3,d1), take(r1,d1,c1), move(r1,d1,d3)$\rangle$ then $\gamma(s_0,\pi) = s_3$



Planning Problems

- *State-variable planning problem*: a triple $P = (\Sigma, s_0, g)$, where
  - $\Sigma = (S,A,\gamma)$ is a state-variable planning domain
  - $s_0 \in S$ is the *initial state*
  - g is a set of ground literals called the *goal*

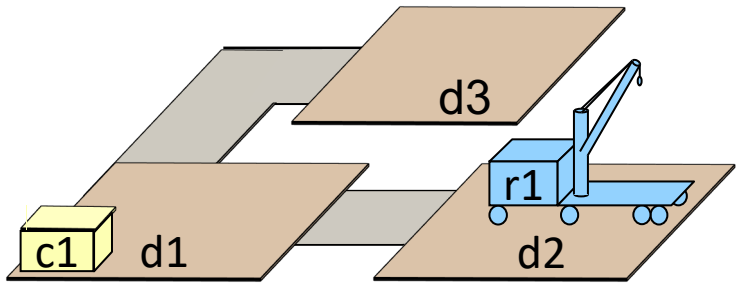
- S_g = {all states in S that satisfy g}
  = $\{s \in S \mid s \cup R$ contains every positive literal in g, and none of the negative literals in $g\}$

- If $\gamma(s_0,\pi)$ satisfies g (or equivalently, $\gamma(s_0,\pi) \in S_g$) then π is a *solution* for P

Poll: How many solutions of length 3?

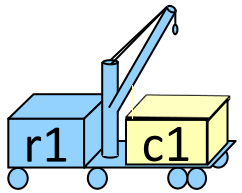
A. 1	B. 2	C. 3	D. 4	E. 5
F. 6	G. 7	H. 8	I. 9	J. other

adjacent = {(d1,d2), (d2,d1), (d1,d3), (d3,d1)}



s_0 = {loc(r1)=d2, cargo(r1)=nil, loc(c1)=d1}

g = {cargo(r1)=c1}



⟨move(r1,d2,d1), take(r1,d1,c1)⟩ is a solution of length 2

move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

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

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

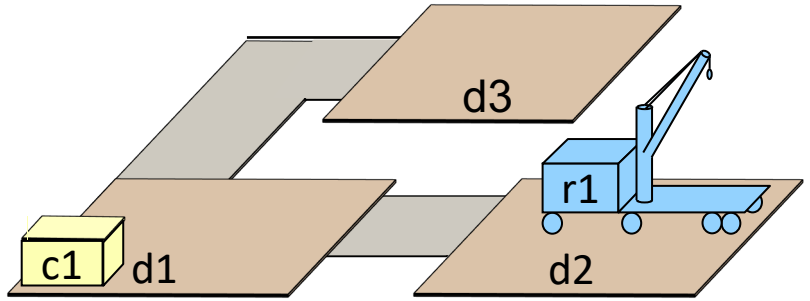
Range(r) = *Robots*
Range(l) = *Locs*
Range(m) = *Locs*

Classical Representation

- Motivation
  - The field of AI planning started out as automated theorem proving
  - It still uses a lot of that notation
- Classical representation is equivalent to state-variable representation
  - No distinction between rigid and varying properties
  - Both represented as logical predicates
  - Both are in the current state

adjacent(l,m)	- location l is adjacent to m
loc(r) = l ⟶ loc(r,l)	- robot r is at location l
loc(c) = r ⟶ loc(c,r)	- container c is on robot r
cargo(r) = c ⟶ loaded(r)	- there's a container on r

why not loaded(r,c)?



- State s = a set of ground atoms
  - Atom a is true in s iff $a \in s$

s_0 = {adjacent(d1,d2), adjacent(d2,d1), adjacent(d1,d3), adjacent(d3,d1), loc(c1,d1), loc(r1,d2)}

Poll: Should s_0 also contain ¬ loaded(r1) ?

A: yes B: no

C: unsure

Classical planning operators

- Action templates

move(r,l,m)
  pre: loc(r)=l, adjacent(l, m)
  eff: loc(r) ← m

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

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

Range(r) = *Robots* = {r1}

Range(l) = Range(m) = *Locs* = {d1,d2,d3}

Range(c) = *Containers* = {c1,c2}

- Classical planning operators

move(r,l,m)
  pre: loc(r,l), adjacent(l, m)
  eff: ¬loc(r,l), loc(r,m)

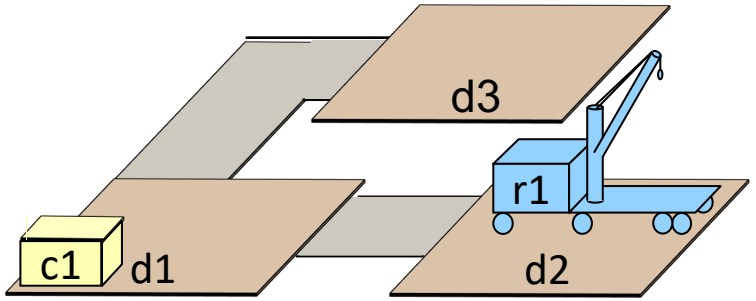
take(r,l,c)
  pre: ¬loaded(r), loc(r,l), loc(c,l)
  eff: loaded(r), ¬loc(c,l), loc(c,r)

put(r,l,c)
  pre: loc(r,l), loc(c,r)
  eff: ¬loaded(r), loc(c,l), ¬loc(c,r)

Poll: Does move really need to include ¬loc(r,l)?

A: yes B: no

C: unsure



Actions

- Planning operator:

  o: move(r,l,m)
  pre: loc(r,l), adjacent(l,m)
  eff: ¬loc(r,l), loc(r,m)

- Action:

  a_1: move(r1,d2,d1)
  pre: loc(r1,d2), adjacent(d2,d1)
  eff: ¬loc(r1,d2), loc(r1,d1)

- Let
  - $\text{pre}^-(a)$ = {a's negated preconditions}
  - $\text{pre}^+(a)$ = {a's non-negated preconditions}

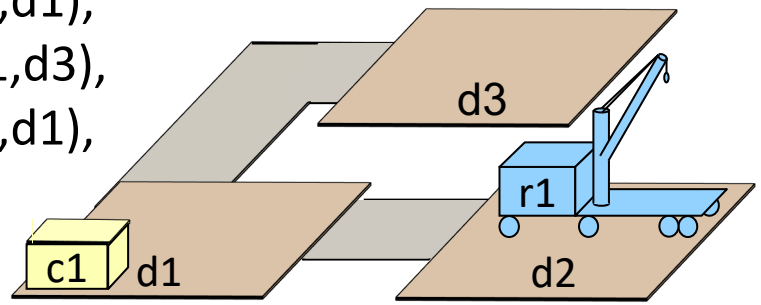
- a is applicable in state s iff

  $s \cap \text{pre}^-(a) = \emptyset$ and $\text{pre}^+(a) \subseteq s$

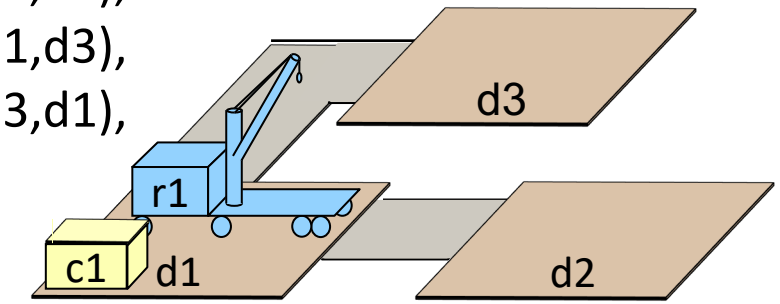
- If a is applicable in s then
  - $\gamma(s,a) = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$

meaning?

s_0 = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
loc(r1,d2)}



$\gamma(s_0, a_1)$ = {adjacent(d1,d2),
adjacent(d2,d1),
adjacent(d1,d3),
adjacent(d3,d1),
loc(c1,d1),
<u>loc(r1,d1)</u>}



Discussion

$$x(b_1,\ldots,b_{n-1})=b_n \;\Rightarrow\; P_x(b_1,\ldots,b_{n-1},b_n)$$

State-variable rep. ⇐ Classical rep.

$$x_P(b_1,\ldots,b_k)=1 \;\Leftarrow\; P(b_1,\ldots,b_k)$$

Poll: Could we instead use $x_P(b_1,\ldots,b_{k-1})=b_k$?

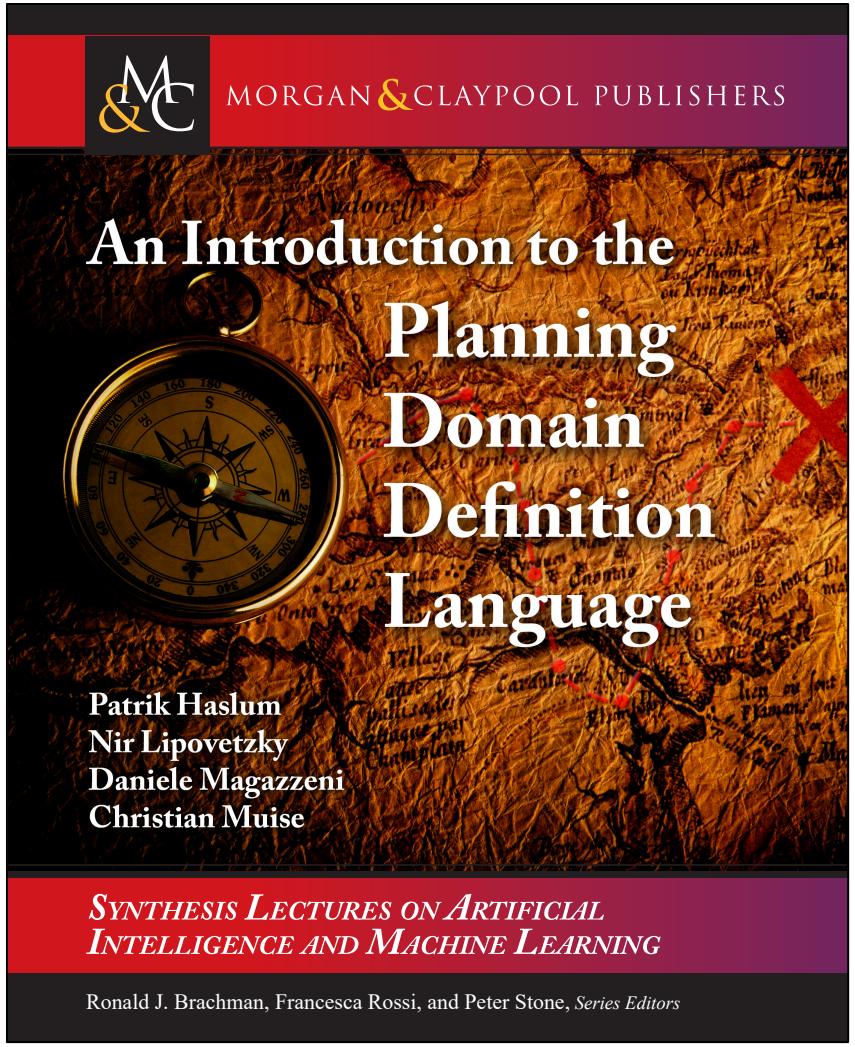
A: yes B: no

C: unsure

- Equivalent expressive power
  - Each can be converted to the other in linear time and space
- Classical representation
  - More natural for logicians
  - Don't require single-valued functions
- State variables
  - More natural for engineers and computer programmers
  - When changing a value, don't have to explicitly delete the old one
- Historically, classical representation has been more widely used
  - That's starting to change

PDDL

- Language for defining planning domains and problems
- Original version of PDDL ≈ 1996
  - Just classical planning
- Multiple revisions and extensions
  - Different subsets accommodate different kinds of planning
- We'll discuss the classical-planning subset
  - Chapter 2 of the PDDL book



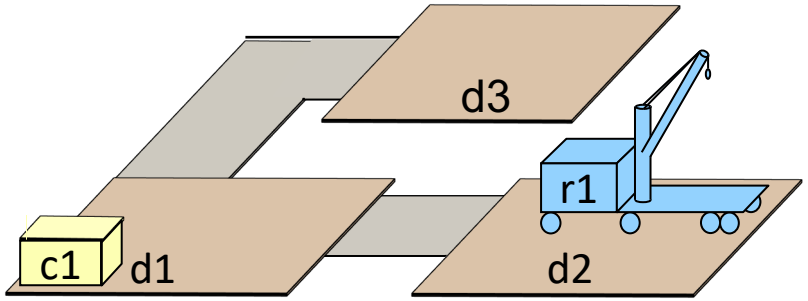
Example domain

- Classical representation:

move(r,l,m)
  Precond: loc(r,l), adjacent(l,m)
  Effects: ¬loc(r,l), loc(r,m)

take(r,l,c)
  Precond: loc(r,l), loc(c,l), ¬loaded(r)
  Effects: loc(c,r), ¬loc(c,l), loaded(r)

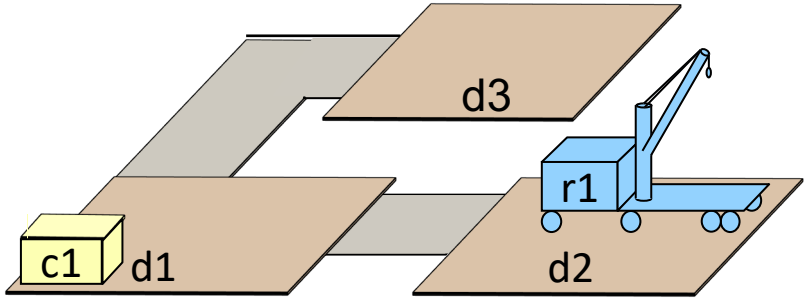
put(r,l,c)
  Precond: loc(r,l), loc(c,r)
  Effects: loc(c,l), ¬loc(c,r), ¬loaded(r)



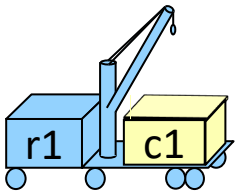

```
(define (domain example-domain-1)
    (requirements :negative-preconditions)

    (:action   move
     :parameters (?r ?l ?m)
     :precondition (and (loc ?r ?l)
                        (adjacent ?l ?m))
     :effect (and (not (loc ?r ?l))
                  (loc ?r ?m)))

    (:action take
     :parameters (?r ?l ?c)
     :precondition (and (loc ?r ?l)
                        (loc ?c ?l)
                        (not (loaded ?r)))
     :effect (and (not (loc ?c ?l))
                  (loc ?c ?r)
                  (loaded ?r)))

    (:action put
     :parameters (?r ?l ?c)
     :precondition (and (loc ?r ?l)
                        (loc ?c ?r))
     :effect (and (loc ?c ?l)
                  (not (loc ?c ?r))
                  (not (loaded ?r)))))
```

Example problem

- Classical representation:



s_0 = {adjacent(d1,d2), adjacent(d2,d1),
adjacent(d1,d3), adjacent(d3,d1),
loc(c1,d1), loc(r1,d2)}



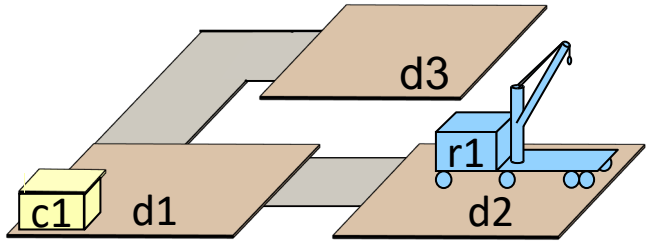
g = {loc(c1,r1)}

```
(define (problem example-problem-1)
    (:domain example-domain-1))

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

Typed domain

State-variable planning:

- Sets of objects
  - *B = Movable_objects ∪ Locs*
  - *Movable_objects = Robots ∪ Containers*
  - *Robots* = {r1}
  - *Containers* = {c1}
  - *Locs* = {d1, d2, d3}



- Parameter ranges
  - *r ∈ Robots*
  - *l,m ∈ Locs*
  - *c ∈ Containers*

```
(define (domain example-domain-2)
    (:requirements
        :negative-preconditions
        :typing)
    (:types
        location movable-obj - object
        robot container - movable-obj)

    (:predicates
        (loc ?r - movable-obj
             ?l - location)
        (loaded ?r - robot)
        (adjacent ?l ?m - location))
```

like saying *Locations, Movable_objects ⊆ B*

Robots, Containers ⊆ Movable_objects

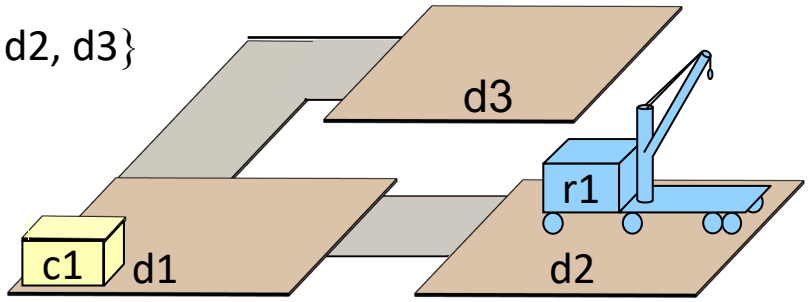
```
(:action move
 :parameters (?r - robot
              ?l ?m - location)
 :precondition (and (loc ?r ?l)
                    (adjacent ?l ?m))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action take
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?l)
                    (not (loaded ?r)))
 :effect (and (not (loc ?r ?l))
              (loc ?r ?m)))

(:action put
 :parameters (?r - robot
              ?l - location
              ?c - container)
 :precondition (and (loc ?r ?l)
                    (loc ?c ?r))
 :effect (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r)))))
```

like saying *r ∈ Robots, l ∈ Locs, c ∈ Containers*

Typed problem

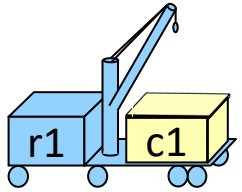
State-variable planning:

- Sets of objects
  - *B* = *Movable_objects* ∪ *Locs*
  - *Movable_objects* = *Robots* ∪ *Containers*
  - *Robots* = {r1}
  - *Containers* = {c1}
  - *Locs* = {d1, d2, d3}



s_0 = {adjacent(d1,d2), adjacent(d2,d1), adjacent(d1,d3), adjacent(d3,d1), loc(c1,d1), loc(r1,d2)}

g = {loc(c1,r1)}

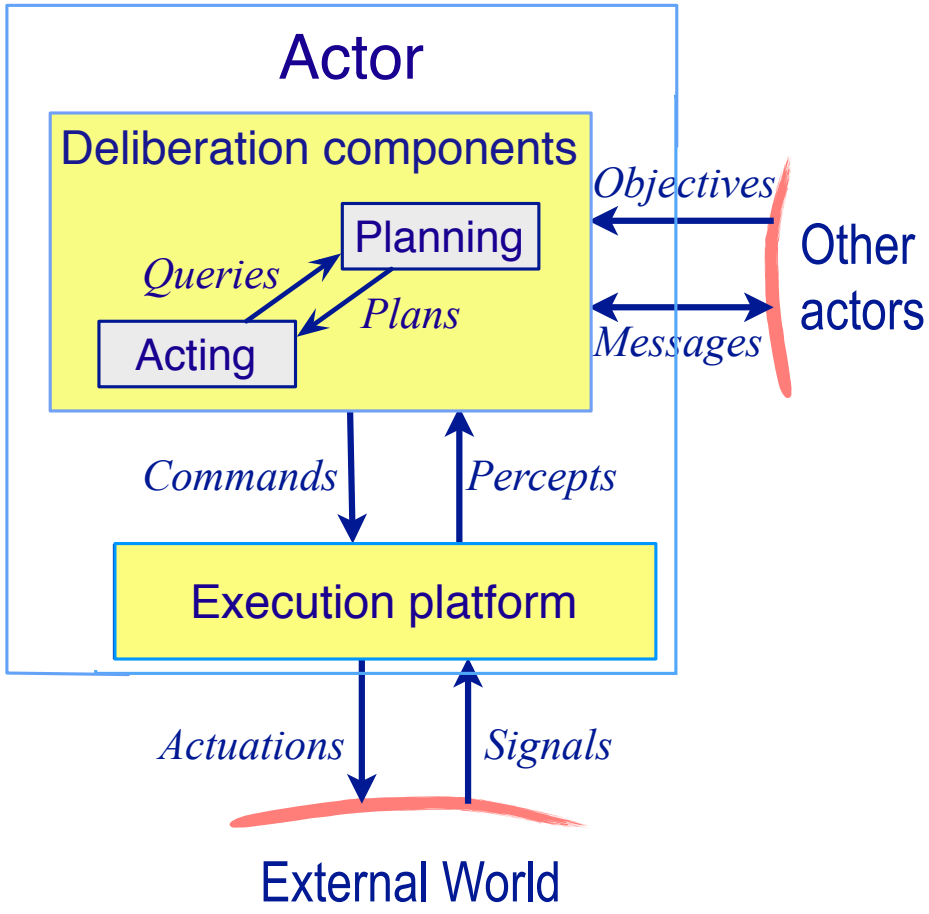



```
(define (problem example-problem-2)
    (:domain example-domain-2))

    (:objects
        r1 - robot
        c1 - container
        d1 d2 d3 - location)

    (:init
      (adjacent d1 d2)
      (adjacent d2 d1)
      (adjacent d1 d3)
      (adjacent d3 d1)
      (loc c1 d1)
      (loc r1 d2)

    (:goal (loc c1 r1)))
```

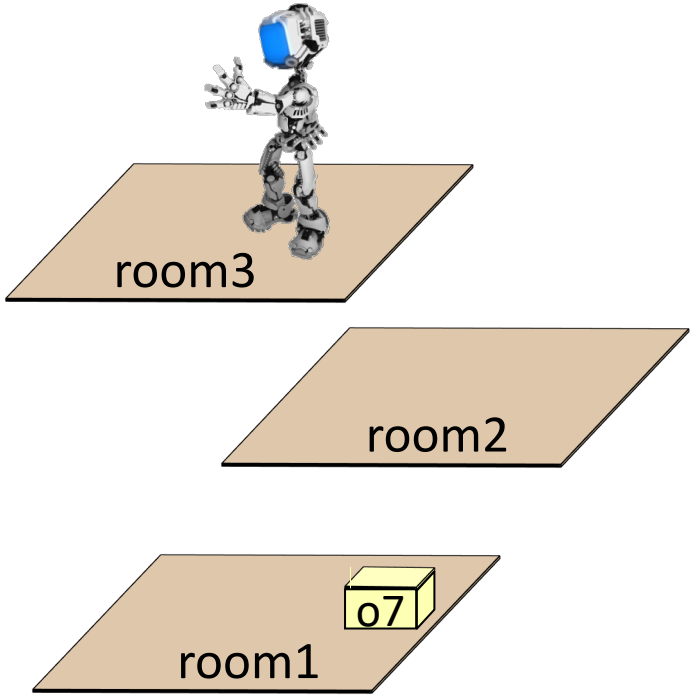
2.6 Incorporating Planning into an Actor

- For classical planning we assumed
  - Finite, static world, just one actor
  - No concurrent actions, no explicit time
  - Determinism, no uncertainty
  - Sequence of states and actions $\langle s_0, a_1, s_1, a_2, s_2, \ldots \rangle$
- Most real-world environments don't satisfy the assumptions

  ⇒ Errors in prediction
- OK if
  - errors occur infrequently, and
  - they don't have severe consequences
- What to do if an error *does* occur?



Service Robot



a_1 = go(r1,room3,hall)

a_2 = navigate(r1,hall,room1)

a_3 = take(r1,room1,o7)

a_4 = navigate(r1,room1,room2)

a_5 = put(r1,room2,o7)⟩

go(r,l,m)
  pre: adjacent(l,m), loc(r)=l
  eff: loc(r) ← m

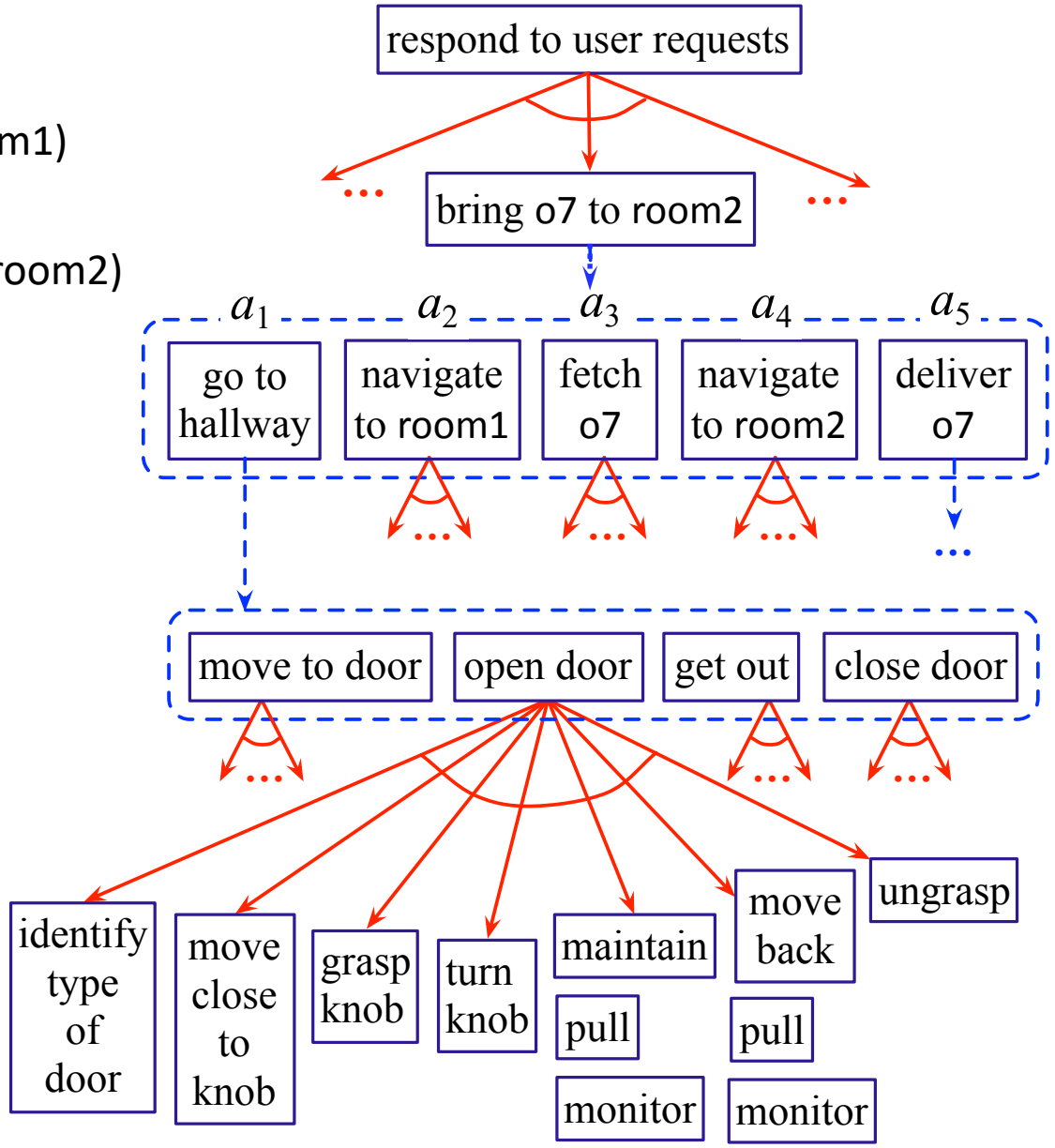
navigate(r,l,m)
  pre: ¬adjacent(l, m), loc(r)=l
  eff: loc(r) ← m

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

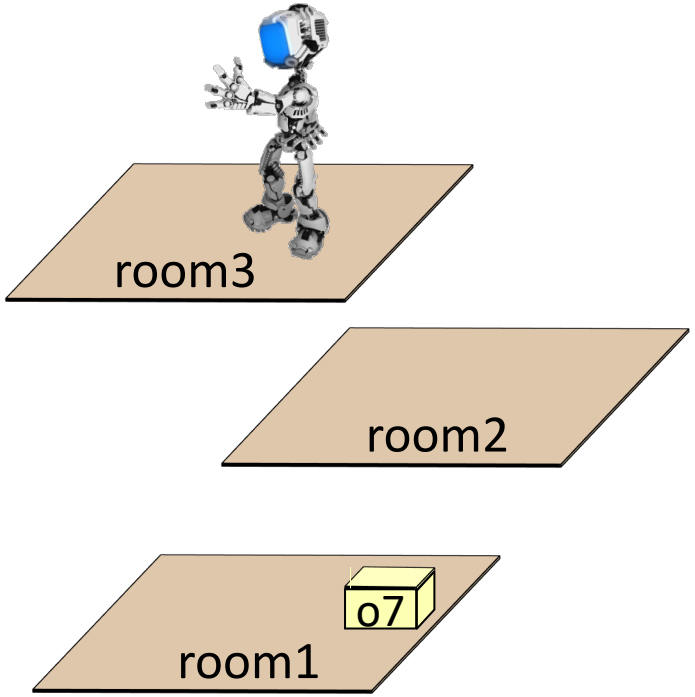
ignores how to get from l to m, e.g., opening the door

ignores how do navigation, localization

ignores how to find o, get access to it, grasp it, lift it



Service Robot



a_1 = go(r1,room3,hall)

a_2 = navigate(r1,hall,room1)

a_3 = take(r1,room1,o7)

a_4 = navigate(r1,room1,room2)

a_5 = put(r1,room2,o7)⟩

go(r,l,m)
  pre: adjacent(l,m), loc(r)=l
  eff: loc(r) ← m

navigate(r,l,m)
  pre: ¬adjacent(l, m), loc(r)=l
  eff: loc(r) ← m

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

ignores how to get from l to m, e.g., opening the door

ignores how do navigation, localization

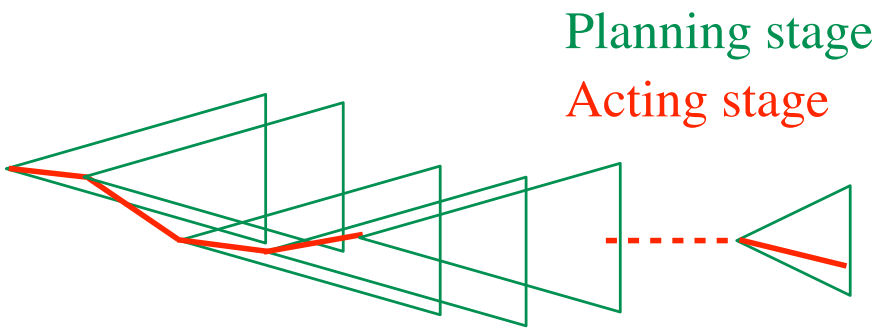
ignores how to find o, get access to it, grasp it, lift it

- Some things that can go wrong:
  - *Execution failures*
    - locked door
    - robot battery goes dead
  - *Unexpected events*
    - class ends, hallway gets crowded
    - hallway closed for maintenance
  - *Incorrect information*
    - navigation error, go to wrong place
  - *Missing information*
    - where is o7 ?
- How to detect and recover from errors?

Using Planning in Acting

the planner

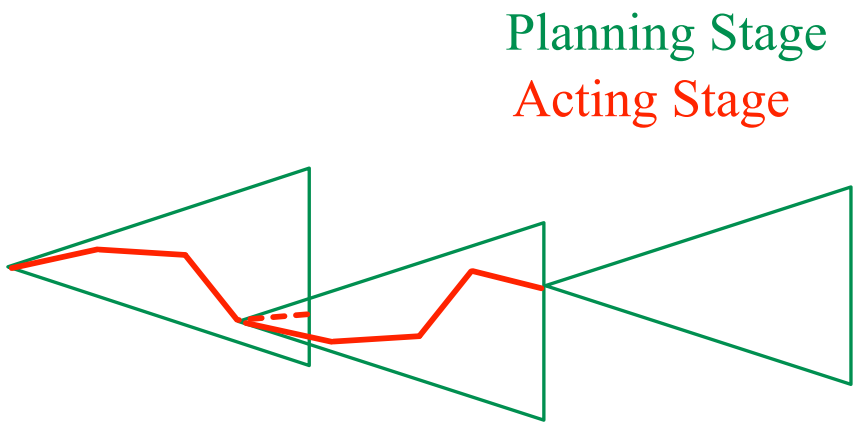
Run-Lookahead(Σ, g)
  $s \leftarrow$ abstraction of observed state ξ
  while $s \not\models g$ do
    $\pi \leftarrow$ Lookahead(Σ, s, g)
    if π = failure then return failure
    $a \leftarrow$ pop-first-action(π); perform(a)
    $s \leftarrow$ abstraction of observed state ξ



- Call Lookahead, obtain π, perform 1st action, call Lookahead again …
- Useful when unpredictable things are likely to happen
  - Replans immediately
- Also useful with *receding horizon* search (e.g., as in chess programs):
  - Lookahead looks a limited distance ahead
- Potential problem:
  - Lookahead needs to return quickly
  - Otherwise, may pause repeatedly while waiting for Lookahead to return
  - What if ξ changes during the wait?

Using Planning in Acting

Run-Lazy-Lookahead(Σ, g)
  $s \leftarrow$ abstraction of observed state ξ
  until s satisfies g do
    $\pi \leftarrow$ Lookahead(Σ, s, g)
    if π = failure then return failure
    until $\pi = \langle\,\rangle$ or $s \vDash g$ or Simulate(Σ, s, g, π) = failure do
      $a \leftarrow$ pop-first-action(π); perform(a)
      $s \leftarrow$ abstraction of observed state ξ

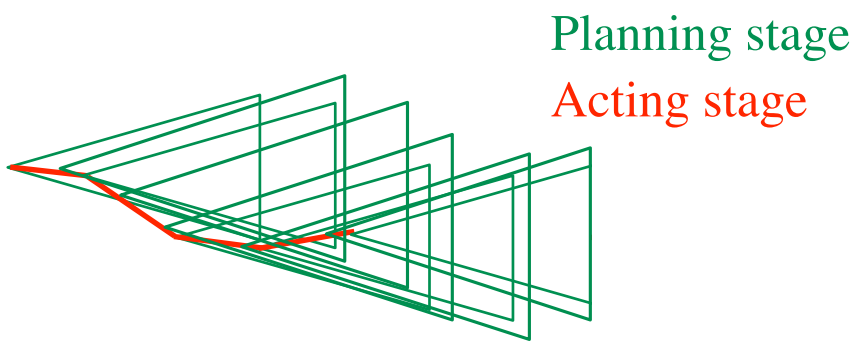


- Call Lookahead, execute the plan as far as possible, don't call Lookahead again unless necessary
- Simulate tests whether the plan will execute correctly
  - Lower-level refinement, physics-based simulation
- What if you don't have a simulation program?
  - Could write Simulate(…) to test whether $\gamma(s,\pi) \vDash g$
    - or test whether $s = \gamma(s', a)$, where s' is the previous state
- Potential problems
  - Simulate needs to return quickly
    - otherwise, may pause repeatedly, ξ may change
  - May might miss opportunities to replace π with a better plan

Using Planning in Acting

Run-Concurrent-Lookahead (basic idea)

- global s, π
- thread 1:
  - loop:
    - $s \leftarrow$ observed state
    - $\pi \leftarrow$ Lookahead(Σ,s,g)
- thread 2:
  - loop:
    - $a \leftarrow$ pop-first-element(π)
    - perform a
    - return if observed state $\models g$



- Motivation: plan and act in a dynamically changing environment
  - Want a recent plan, rather than the old one that Run-Lazy-Lookahead would use
  - Want to get it quickly, rather than waiting like Run-Lookahead
- But there are several problems with the pseudocode
  - It ignores some implementation details
    - how to do locking
    - whether each thread has correct values for π and s
  - If thread 2 performs any actions while Lookahead is running, we probably should restart Lookahead
    - Otherwise Lookahead will return a plan that's out-of-date
  - Another possibility:
    - If thread 2 is going to perform action a, have thread 1 run Lookahead(Σ, $\gamma(s,a)$, g)

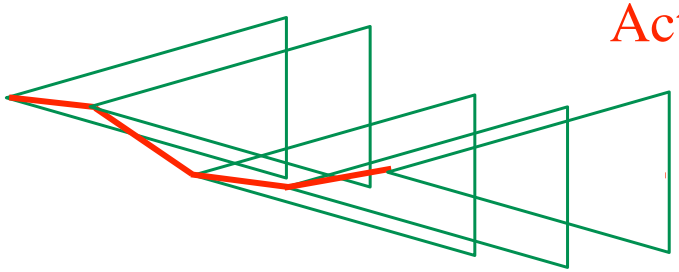
How to do Lookahead

Some possibilities (can also combine these)

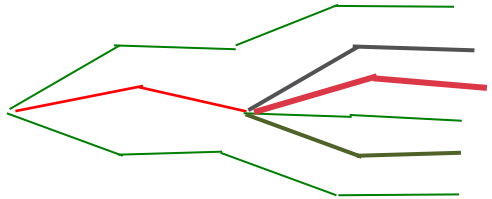
- **Full planning** (if the planner can solve the planning problem quickly enough)
- **Receding horizon**
  - Modify Lookahead to search just part of the way to g (see next page)
  - E.g., cut off search when one of the following exceeds a maximum threshold:
    - plan length, plan cost, computation time

Planning stage

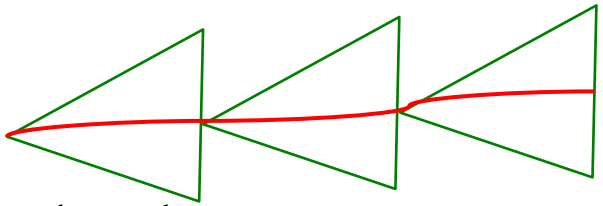
Acting stage



- **Sampling**
  - Modify Lookahead to do a *Monte Carlo rollout*
    - Depth-first search with random node selection and no backtracking
  - Call Lookahead several times, choose the plan that looks best
  - Best-known example of this: the UCT algorithm (see Chapter 6)



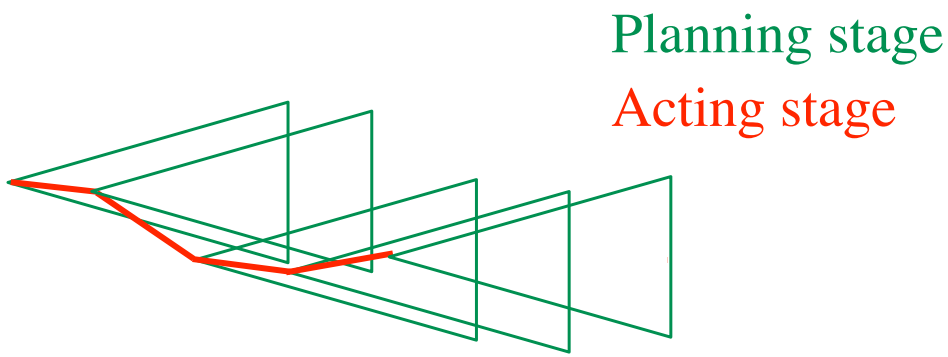
- **Subgoaling**
  - Tell Lookahead to plan for some subgoal g_1, rather than g itself
  - Once the actor has achieved g_1, tell Lookahead to plan for the next subgoal g_2
  - And so forth until the actor reaches g



Receding-Horizon Search

Deterministic-Search(Σ, s_0, g)
  Frontier ← $\{(\langle\rangle, s_0)\}$
  Expanded ← $\emptyset$
  while *Frontier* $\neq \emptyset$ do
    select a node $\nu = (\pi, s) \in$ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if s satisfies g then return π
    Children ← $\{(\pi.a, \gamma(s,a)) \mid s$ satisfies pre(a)$\}$
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

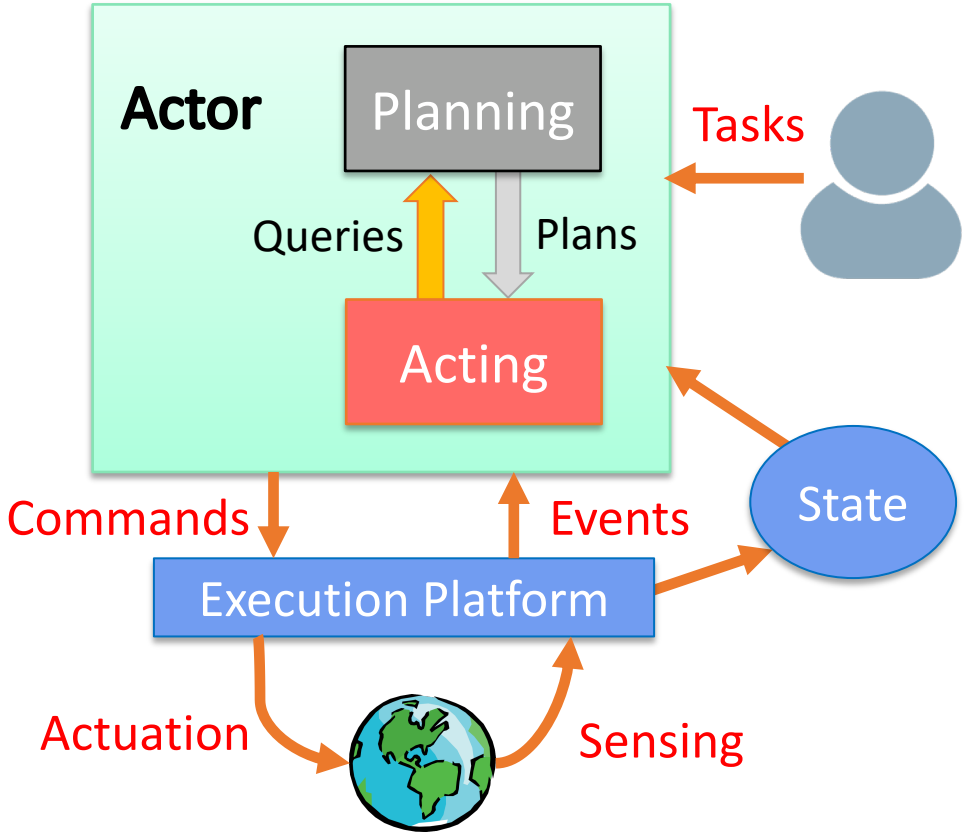
- Lookahead = modified version of Deterministic-Search
  - Before line (*i*), put something like one of these:
    - *time-based cutoff*: if time-left() = 0 then return π
    - *length-based cutoff*: if $|\pi| > l_{max}$ then return π
    - *cost-based cutoff*: if $f(\nu) > c_{max}$ then return π
    - *closeness to goal*: if $h(s) \leq \varepsilon$ then return π
  - Length-based and cost-based make sense if you're doing GBFS or AI, but not if you're doing DFS
  - Could modify DFBB to use π^* = least costly partial solution of length $\leq l_{max}$

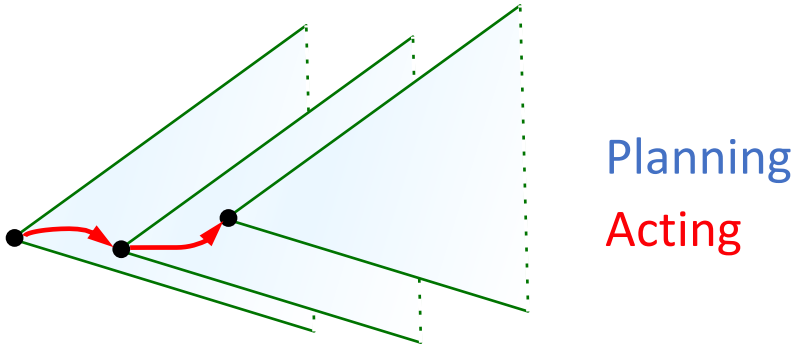


Planning and Acting

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

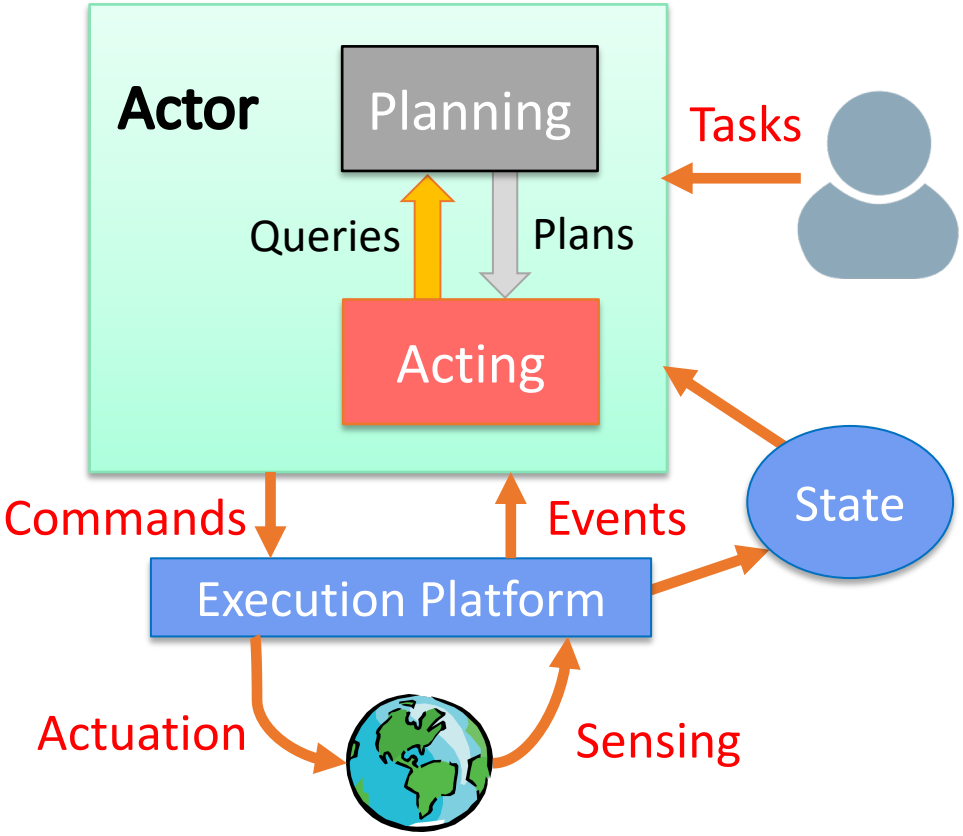
- **Acting**: *performing*
  - Dynamic, unpredictable, partially observable environment
    - Adapt to context, react to events
  - Uses *operational* models
    - instructions telling *how* to perform the tasks
    - usually hierarchical

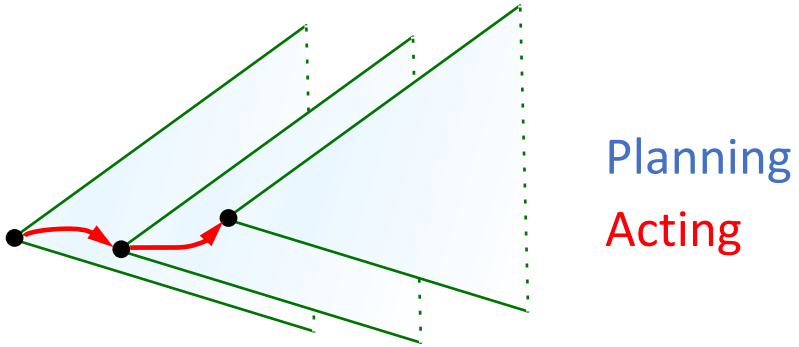




Outline

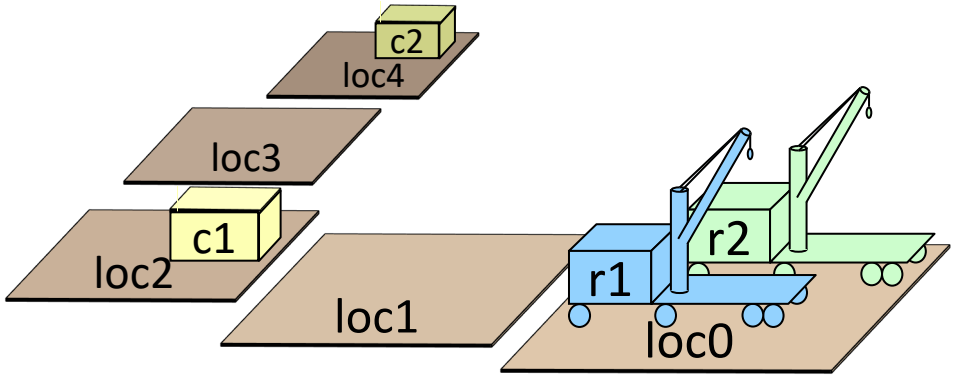
1. Motivation
2. *Representation*
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application





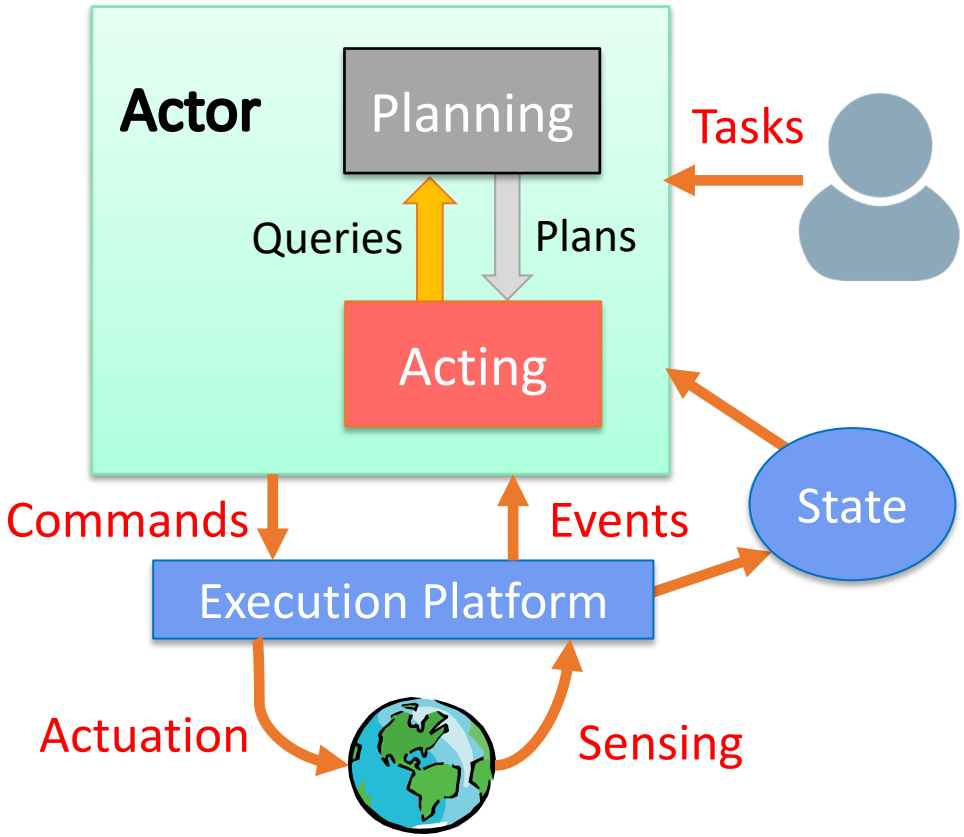
Example

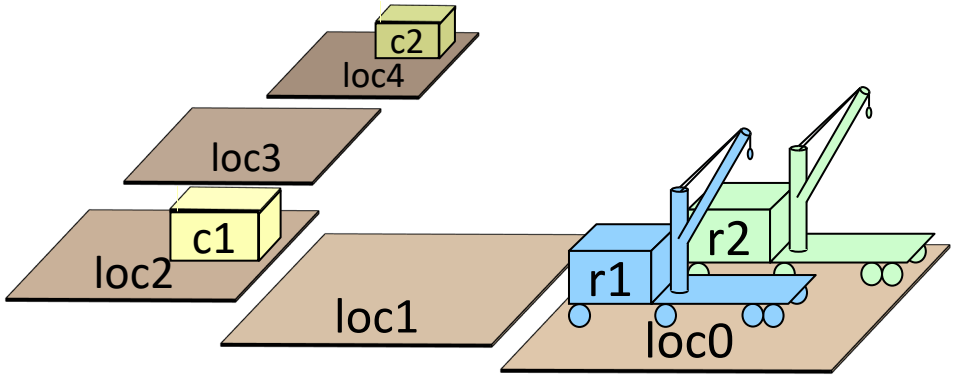
- Consider an actor that controls two robots
- Environment is *partially observable*
  - Each robot can only see what's at the current location



- Objects
  - *Robots* = {r1, r2}
  - *Containers* = {c1, c2}
  - *Locations* = {loc0, loc1, loc2, loc3, loc4}
- Rigid relations (properties that won't change)
  - adjacent(loc0,loc1), adjacent(loc1,loc0), adjacent(loc1,loc2), adjacent(loc2,loc1), adjacent(loc2,loc3), adjacent(loc3,loc2), adjacent(loc3,loc4), adjacent(loc4,loc3)
- State variables (fluents)
  - where $r \in$ *Robots*, $c \in$ *Containers*, $l \in$ *Locations*
  - loc(r) $\in$ *Locations*
  - cargo(r) $\in$ *Containers* $\cup$ {empty}
  - pos(c) $\in$ *Locations* $\cup$ *Robots* $\cup$ {unknown}
  - view(l) $\in$ {T, F}
    - Whether a robot has looked at location l
    - If view(l) = T then pos(c) = l for every container c at l

Example (continued)

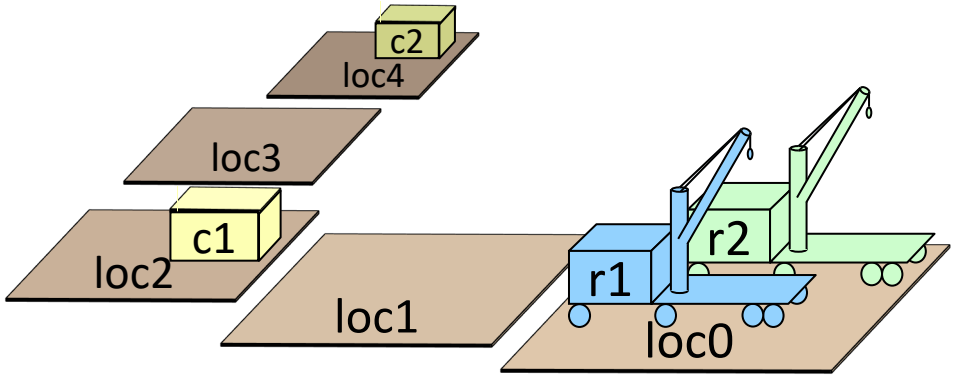




- Commands to the execution platform:
  - take(r,o,l): r takes object o at location l
  - put(r,o,l): r puts o at location l
  - perceive(r,l): robot r perceives what objects are at l
  - move-to(r,l): robot r moves to location l

Tasks and Methods

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



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

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

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

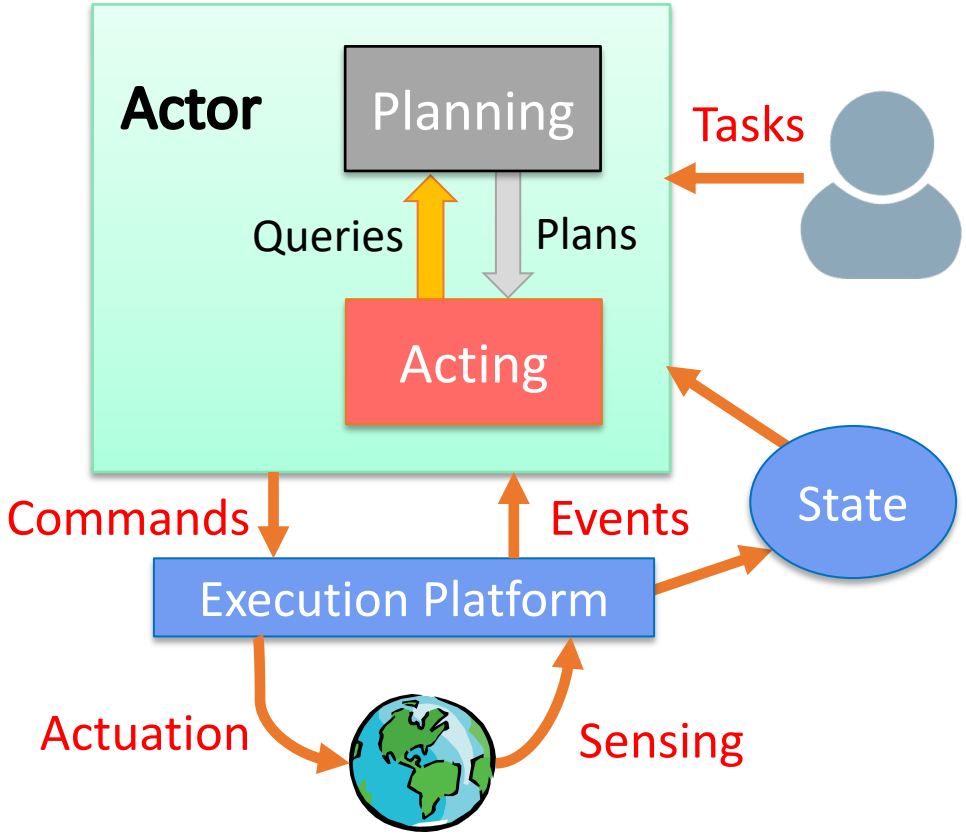
m-fetch2(r,c)
  task: fetch(r,c)
  pre: pos(c) ≠ unknown
  body:
    if loc(r) = pos(c) then
      take(r,c,pos(c))
    else do
      move-to(r,pos(c))
      take(r,c,pos(c))

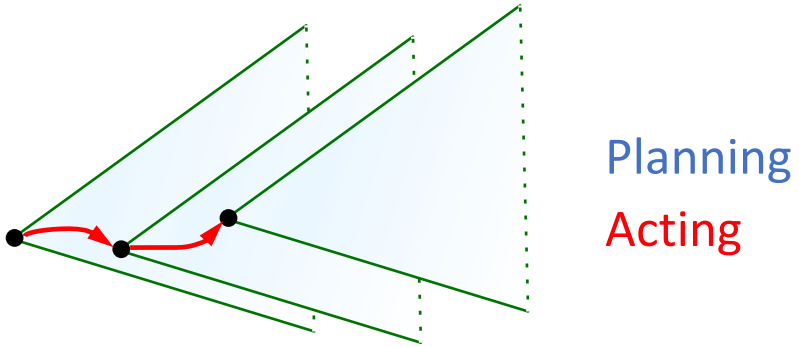
command

task

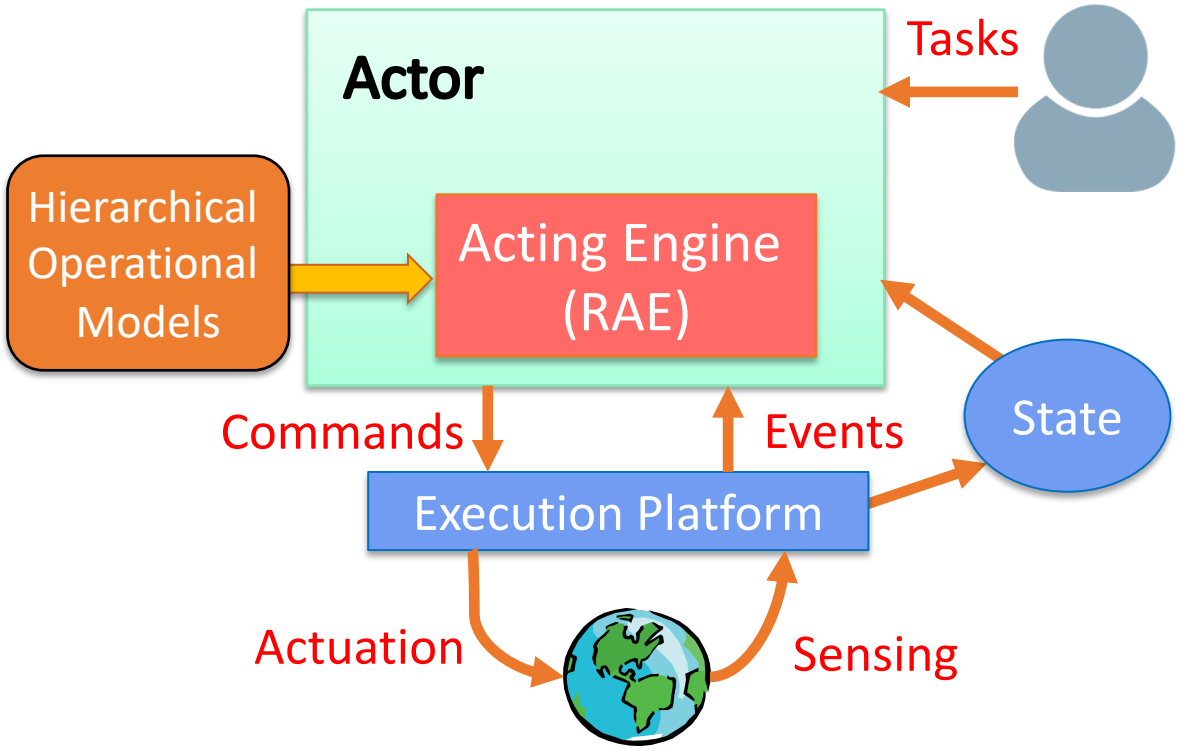
Outline

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

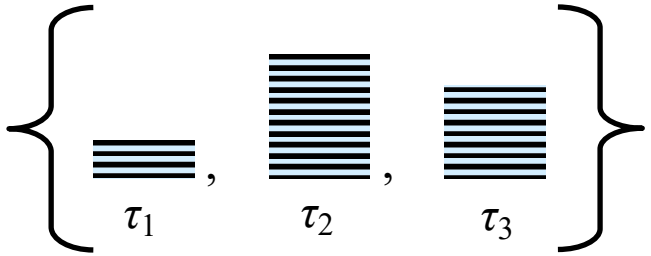




RAE (Refinement Acting Engine)



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

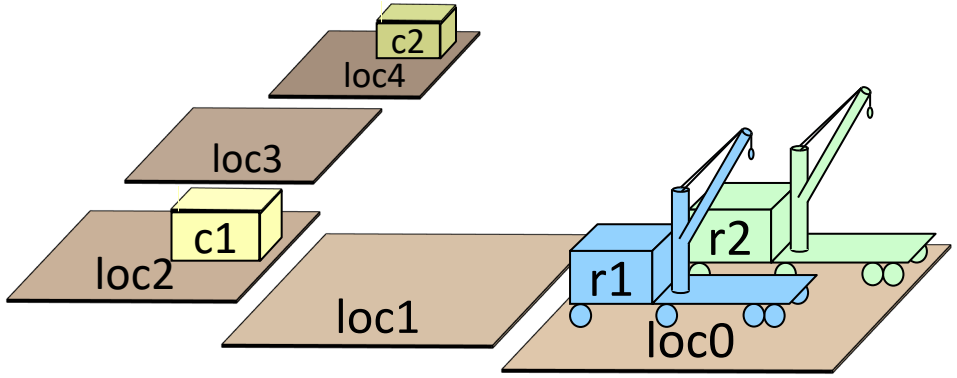


- Refinement stack for a task τ
  ⇔ current ~~path~~ in RAE's search tree for τ

refinement tree

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

Example (reminder)



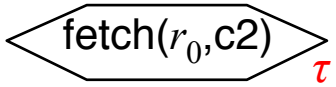
- Objects
  - *Robots* = {r1, r2}
  - *Containers* = {c1, c2}
  - *Locations* = {loc1, loc2, loc3, loc4}
- Rigid relations (properties that won't change)
  - adjacent(loc0,loc1), adjacent(loc1,loc0), adjacent(loc1,loc2), adjacent(loc2,loc1), adjacent(loc2,loc3), adjacent(loc3,loc2), adjacent(loc3,loc4), adjacent(loc4,loc3)
- State variables (fluents)
  - where $r \in$ *Robots*, $c \in$ *Containers*, $l \in$ *Locations*
  - loc(r) $\in$ *Locations*
  - cargo(r) $\in$ *Containers* $\cup$ {nil}
  - pos(c) $\in$ *Locations* $\cup$ *Robots* $\cup$ {unknown}
  - view(l) $\in$ {T, F}
    - Whether a robot has looked at location l
    - If view(l) = T then pos(c) = l for every container c at l
- Commands to the execution platform:
  - take(r,o,l): r takes object o at location l
  - put(r,o,l): r puts o at location l
  - perceive(r,l): robot r perceives what objects are at l
  - move-to(r,l): robot r moves to location l

Example

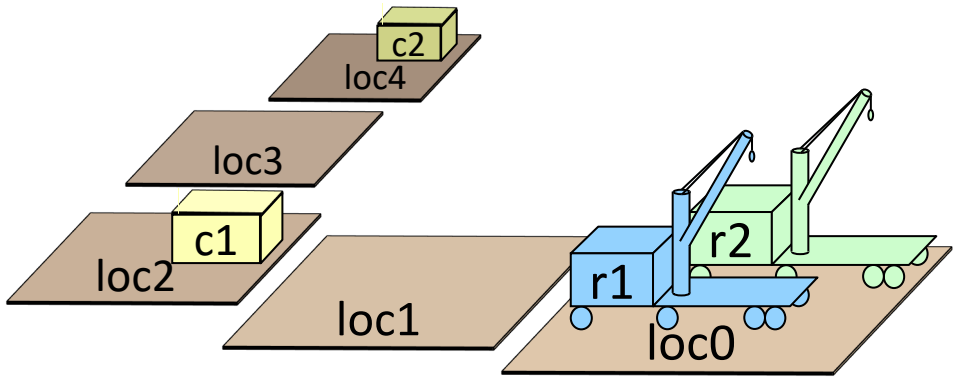
```
m-fetch1(r,c)
    task:   fetch(r,c)
    pre:    pos(c) = unknown
    body:
        if ∃l (view(l) = F) then
            move-to(r,l)
            perceive(r,l)
            if pos(c) = l then
                take(r,c,l)
            else fetch(r,c)
        else fail
```

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

Refinement tree



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




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

Example

m-fetch1(r,c) $r = r_0$, c = c2
task: fetch(r,c)
pre: pos(c) = unknown
body:
if ∃l (view(l) = F) then
move-to(r,l)
perceive(r,l)
if pos(c) = l then
take(r,c,l)
else fetch(r,c)
else fail

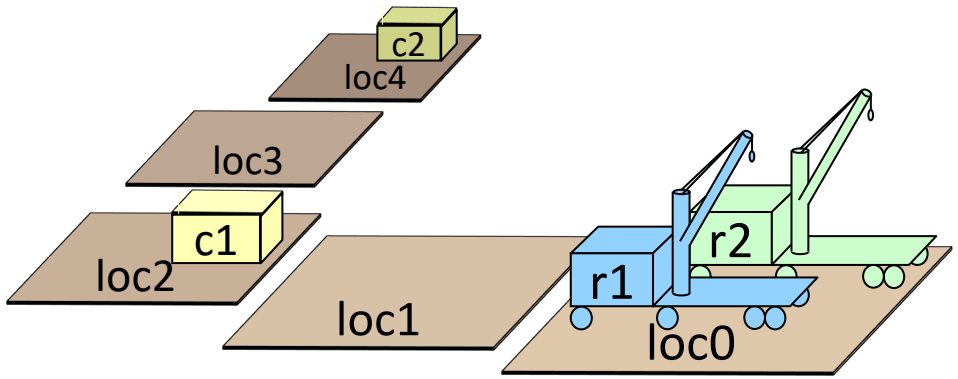
Refinement tree

fetch(r_0,c2) τ

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

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

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



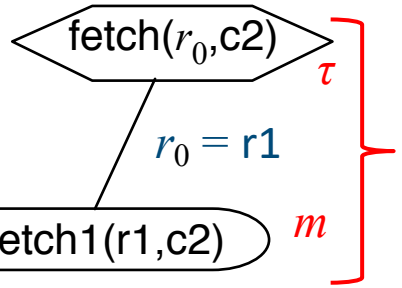
procedure RAE:
loop:
for every new external task or event τ do
choose a method instance m for τ
create a refinement stack for τ, m
add the stack to *Agenda*
for each stack σ in *Agenda*
call Progress(σ)
if σ is finished then remove it

Example

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

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

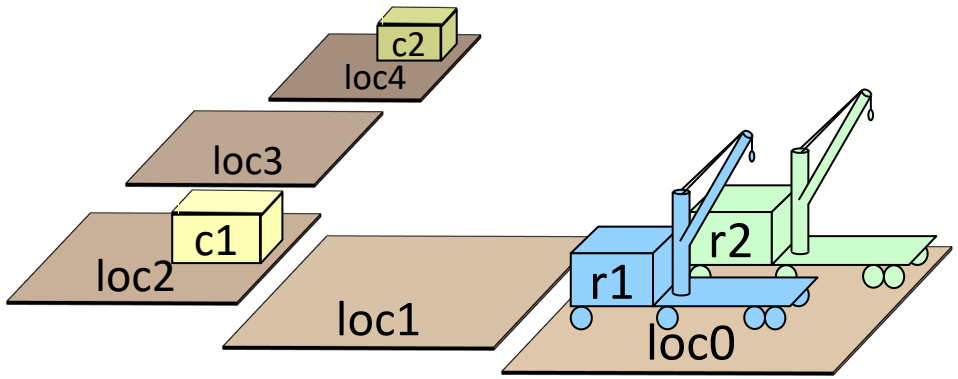
Refinement tree



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

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

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

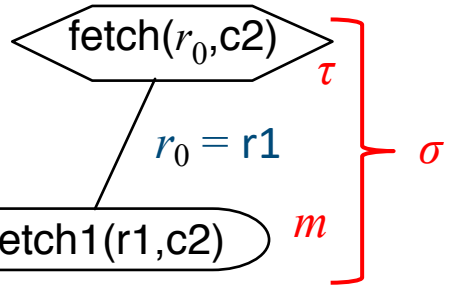


Example

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

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

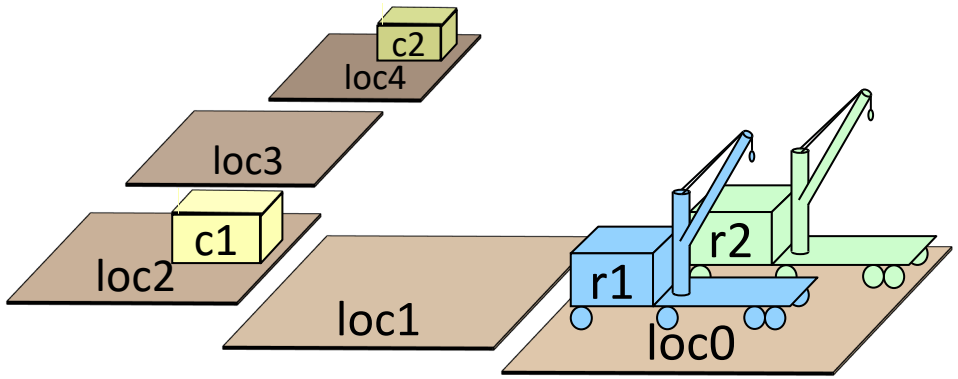
Refinement tree



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

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

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

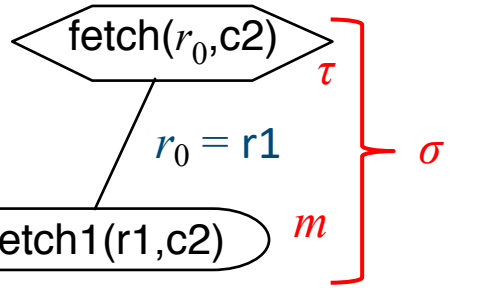


Example

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

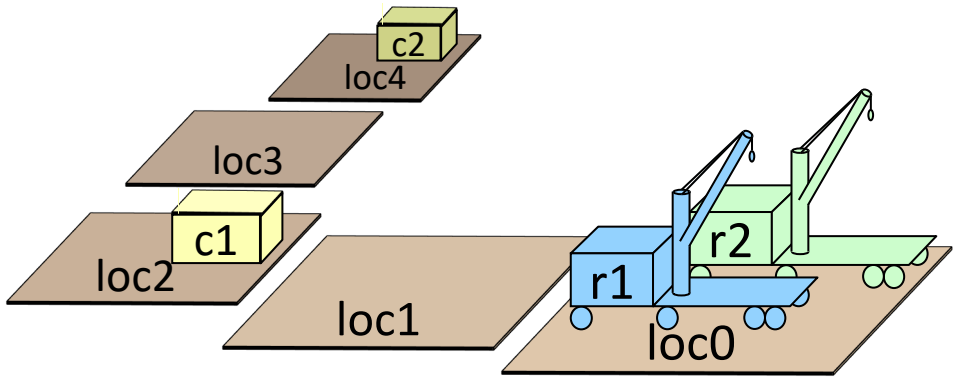
Candidates = {m-fetch(r1,c2), m-fetch(r2,c2)}

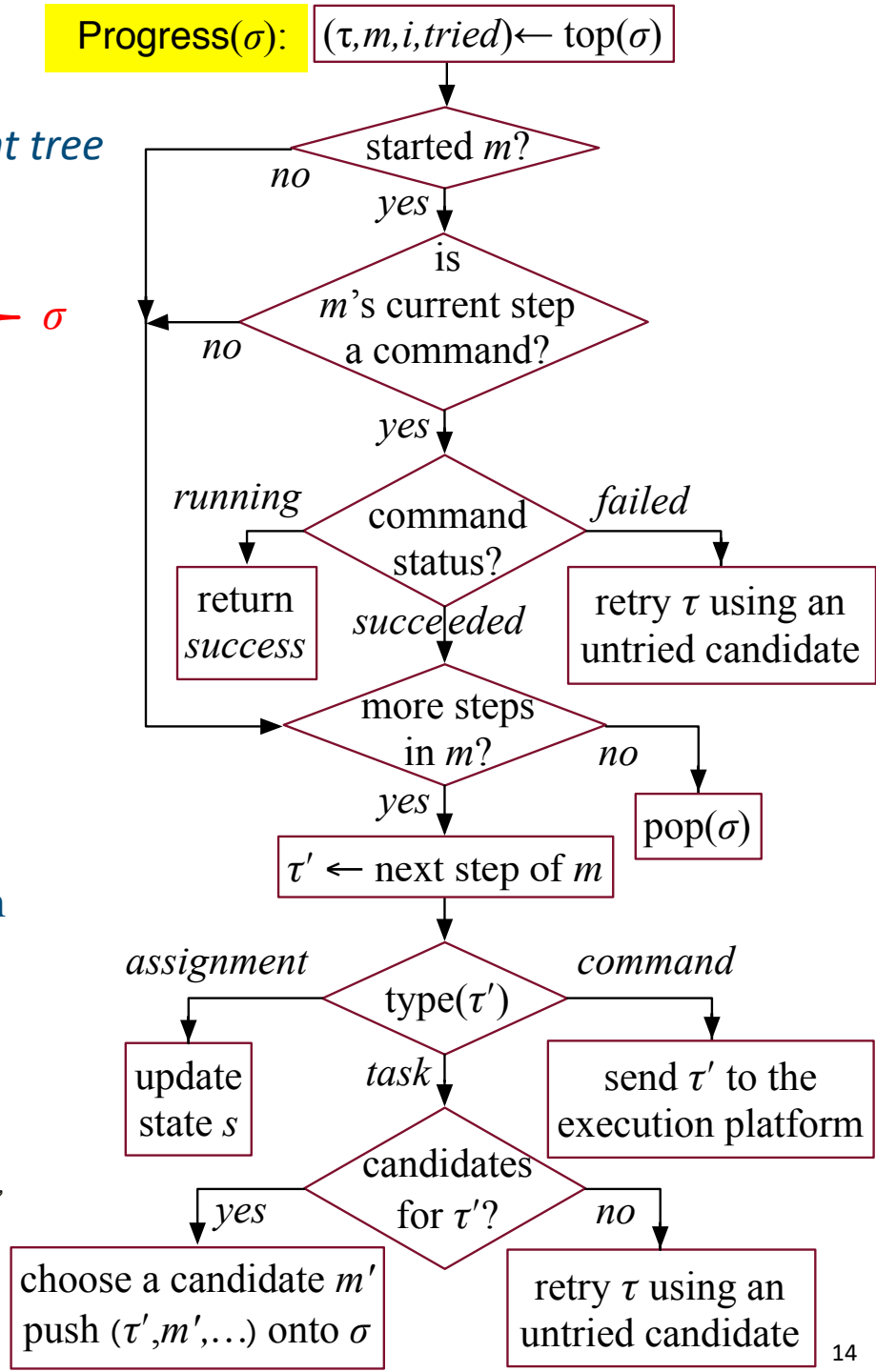
Refinement tree



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

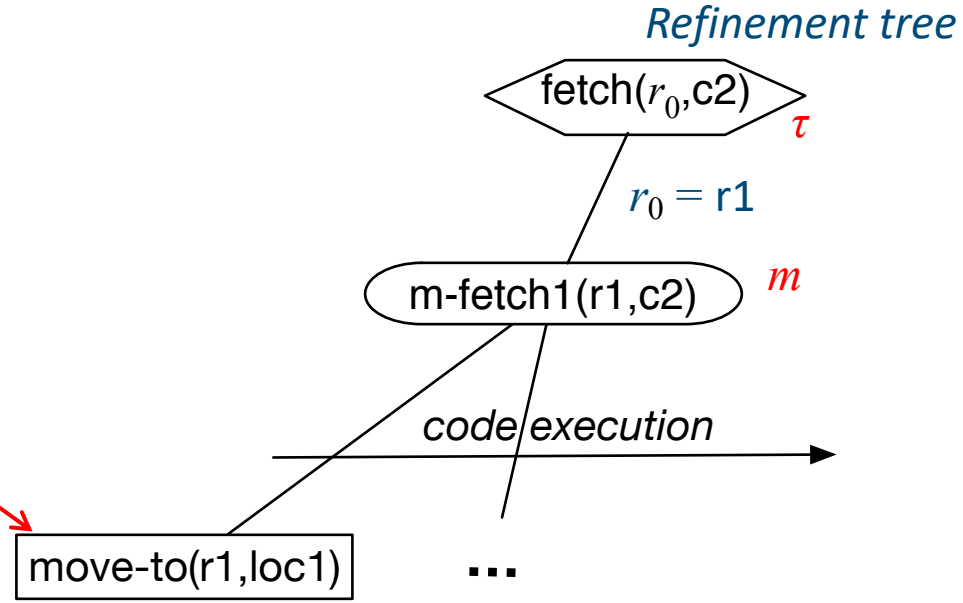
- Container locations unknown
- Partially observable
  - Robot only sees current location





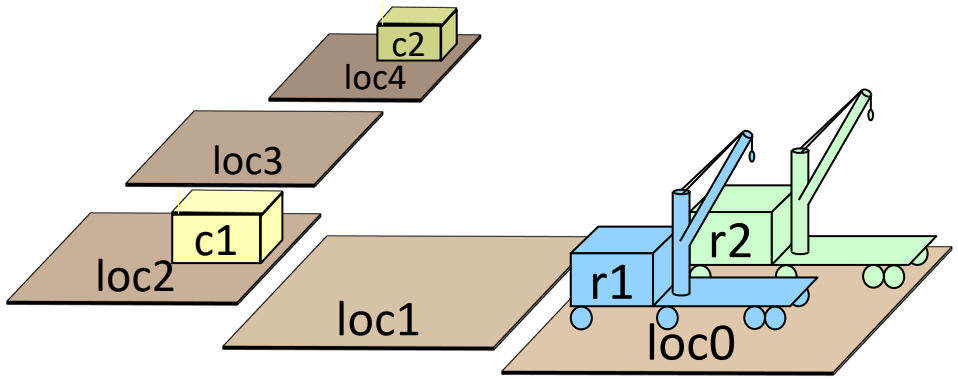
Example

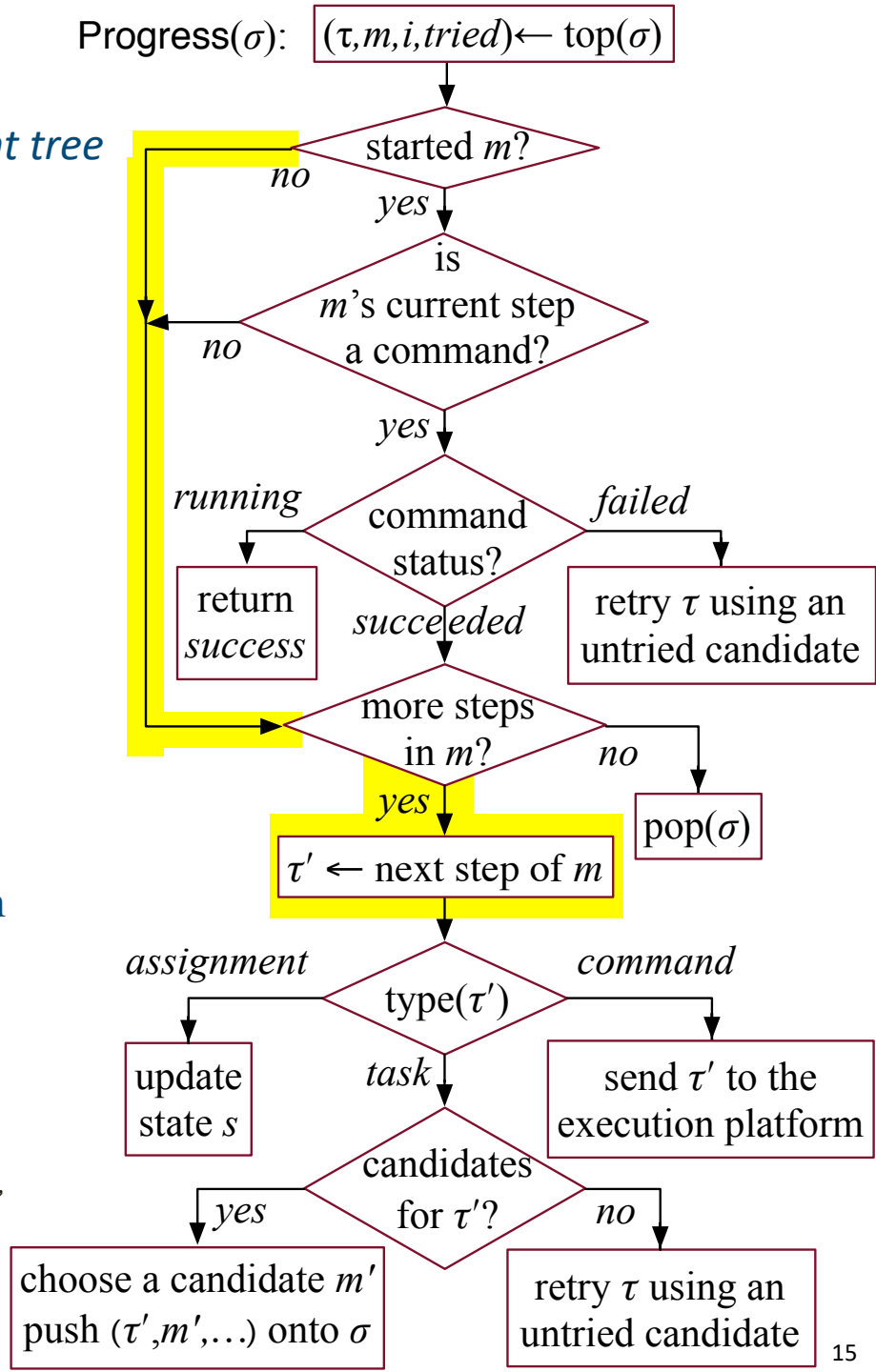
m-fetch1(r,c) r = r1, c = c2
- task: fetch(r,c)
- pre: pos(c) = unknown
- body: l = loc1
  - if ∃l (view(l) = F) then
    - move-to(r,l)
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail



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

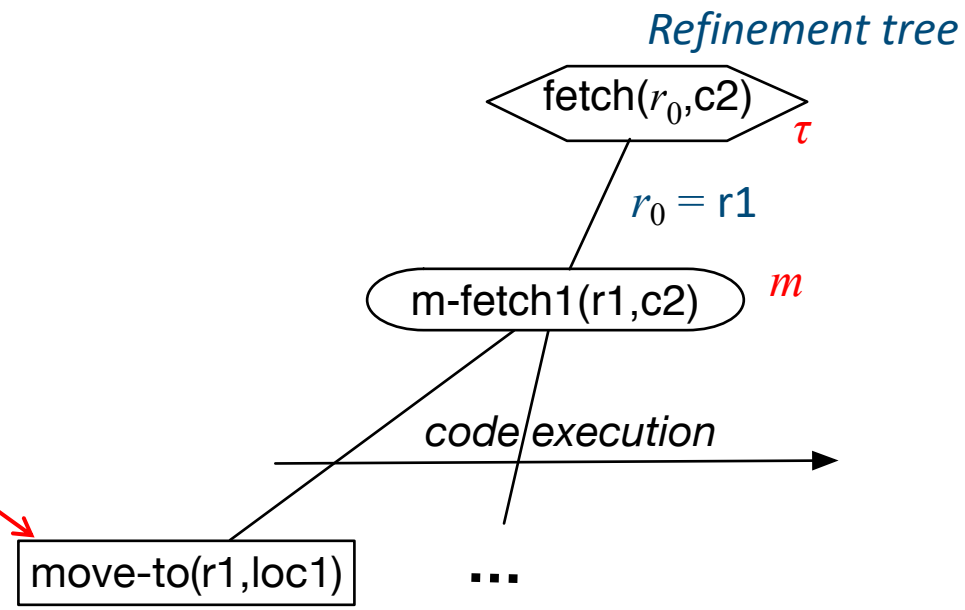
- Container locations unknown
- Partially observable
  - Robot only sees current location





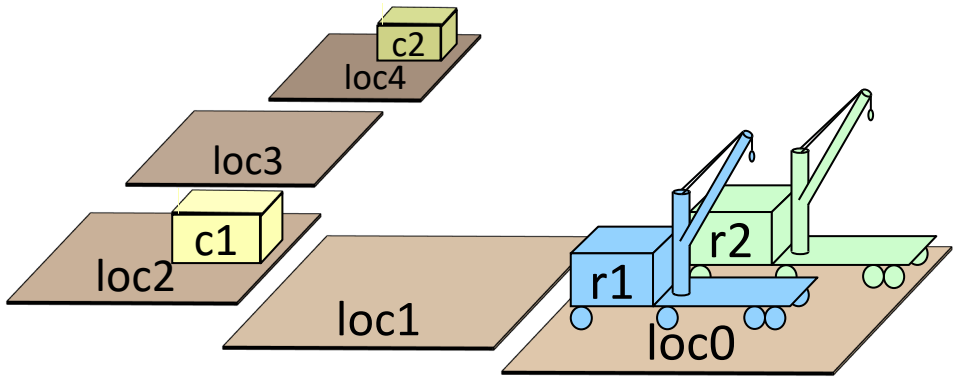
Example

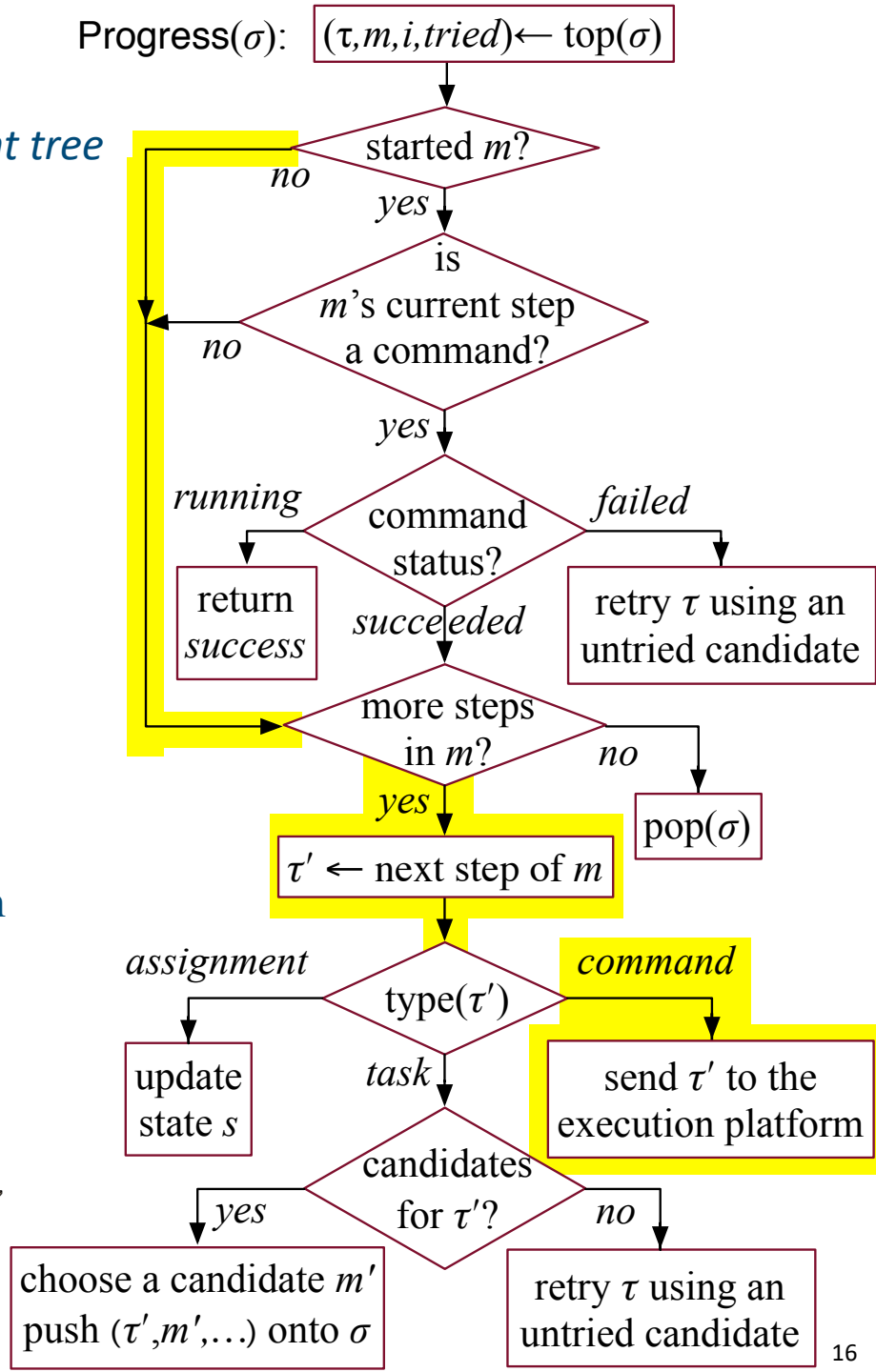
m-fetch1(r,c) r = r1, c = c2
- task: fetch(r,c)
- pre: pos(c) = unknown
- body: l = loc1
  - if ∃l (view(l) = F) then
    - move-to(r,l)
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail



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

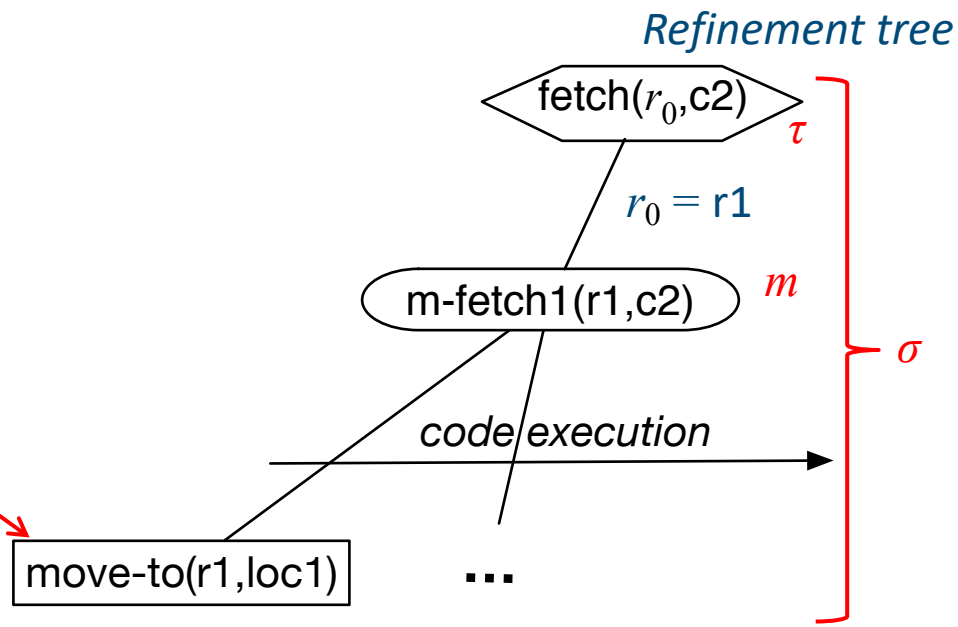
- Container locations unknown
- Partially observable
  - Robot only sees current location





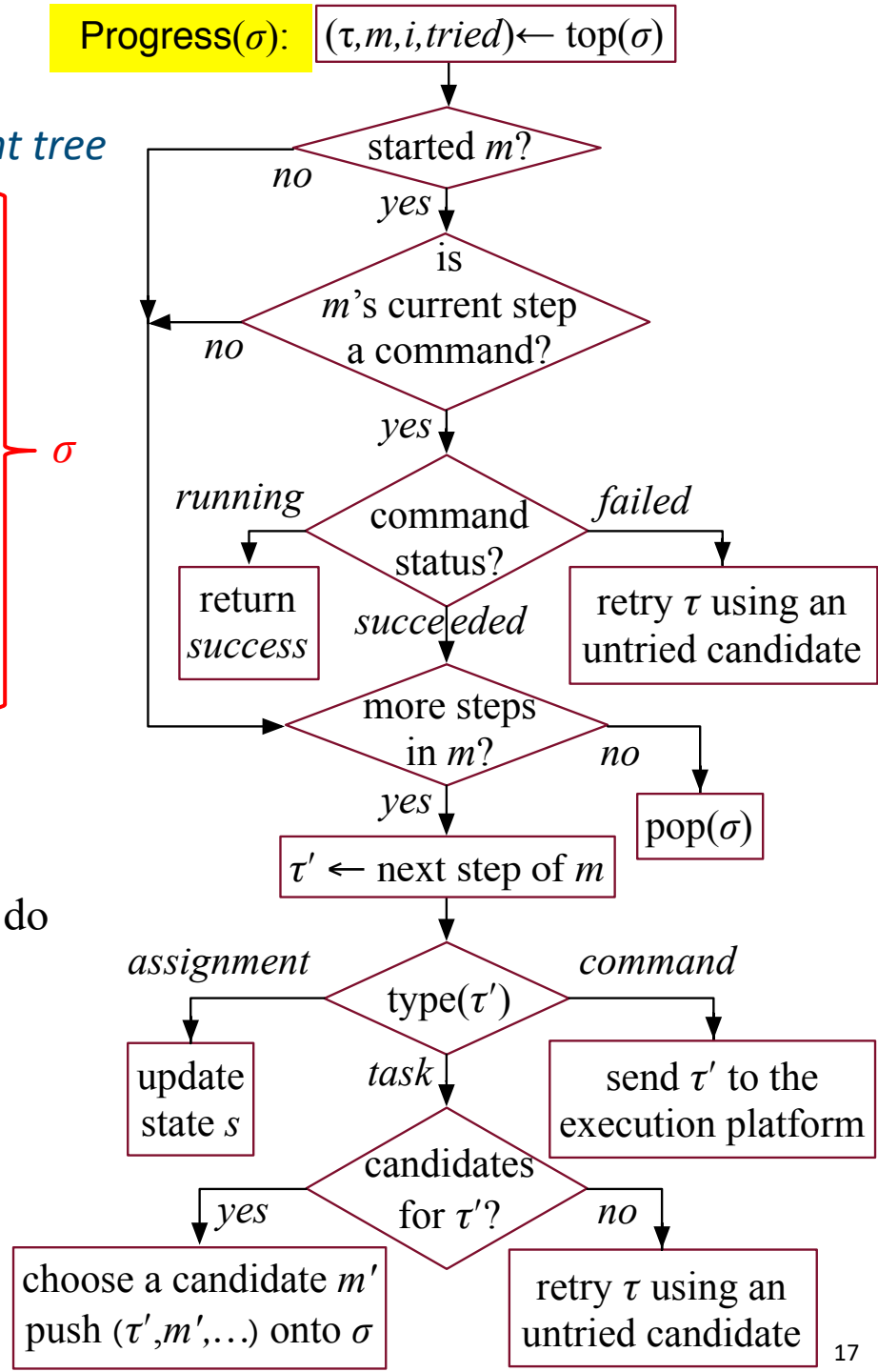
Example

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



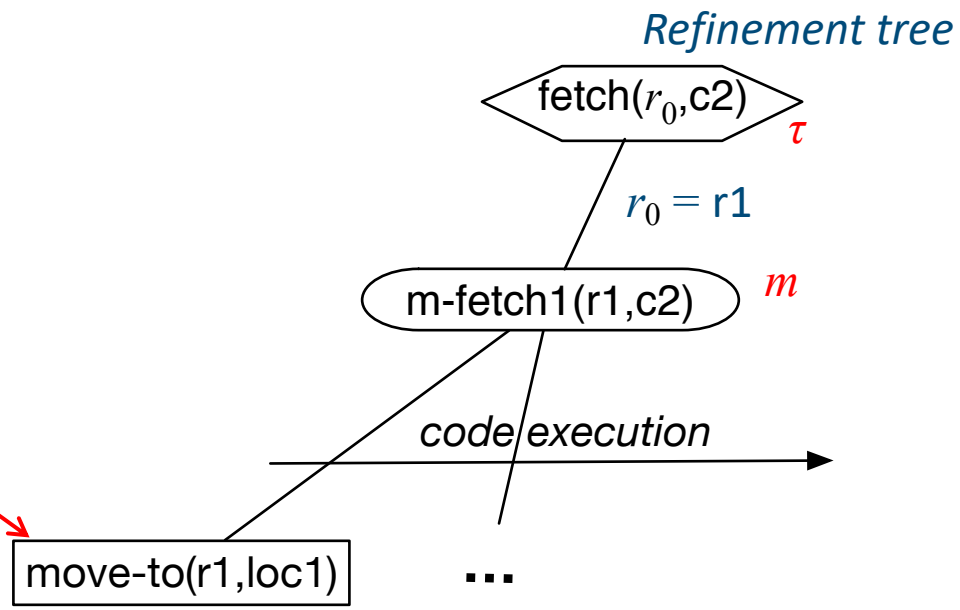
m-fetch2(r,c)
- task: fetch(r,c)
- pre: pos(c) ≠ unknown
- body:
  - if loc(r) = pos(c) then
    - take(r,c,pos(c))
  - else do
    - move-to(r,pos(c))
    - take(r,c,pos(c))

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

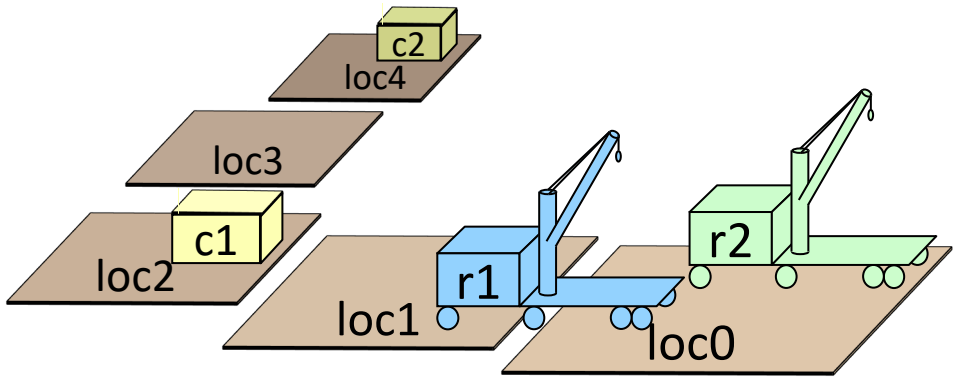


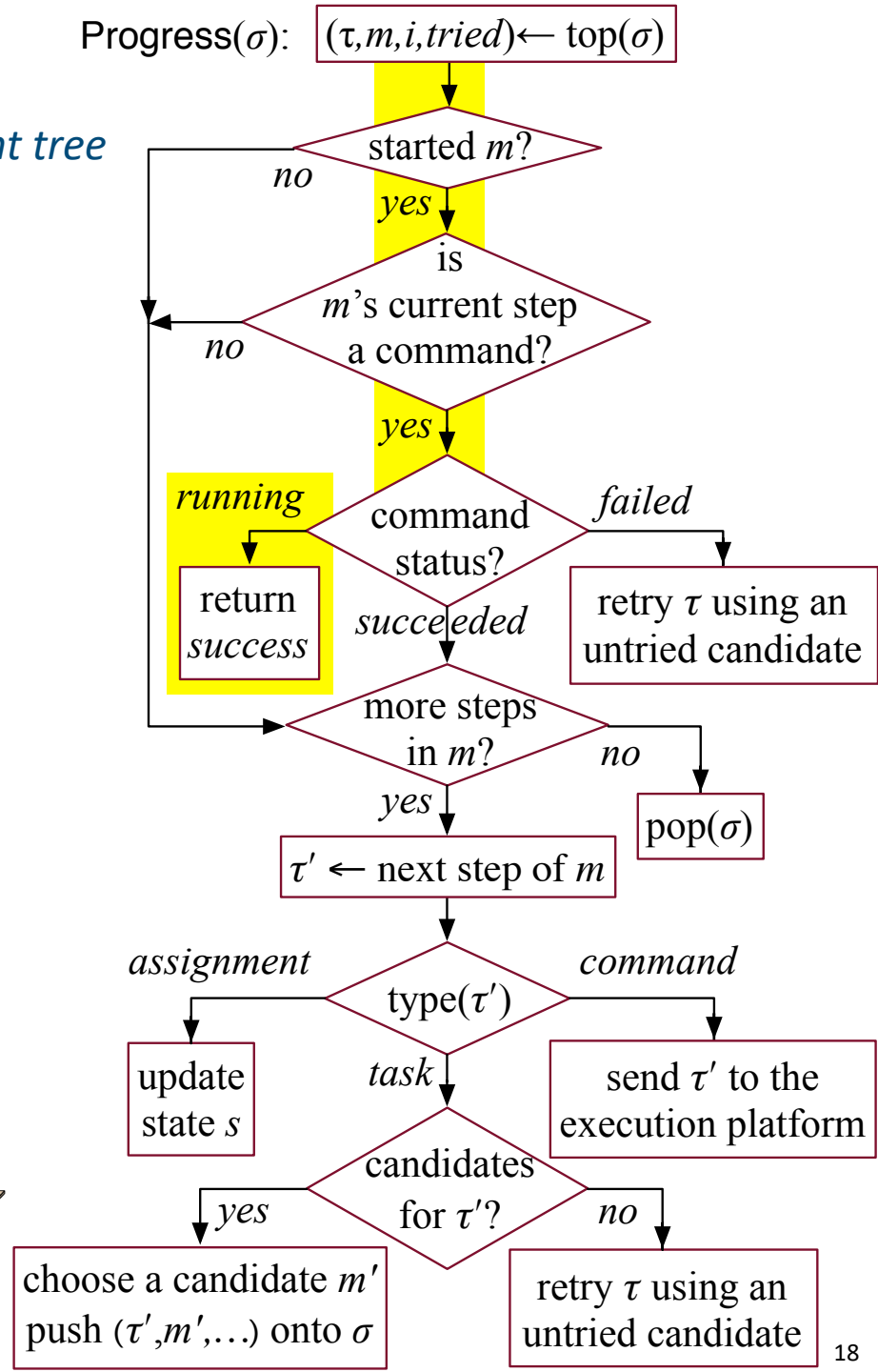
Example

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



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

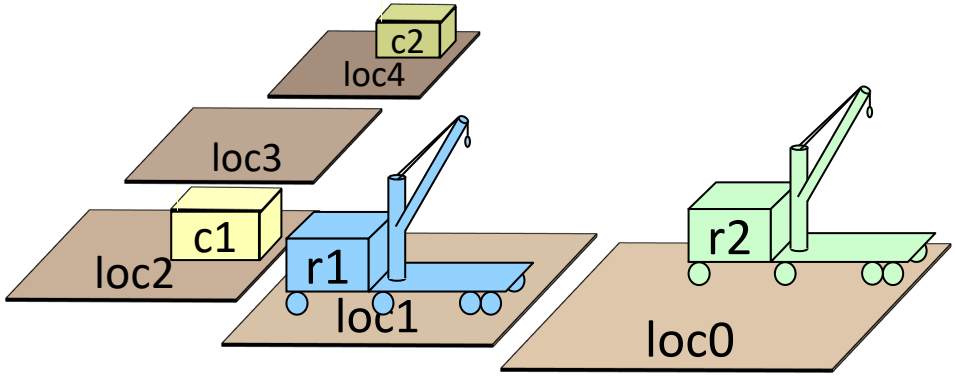




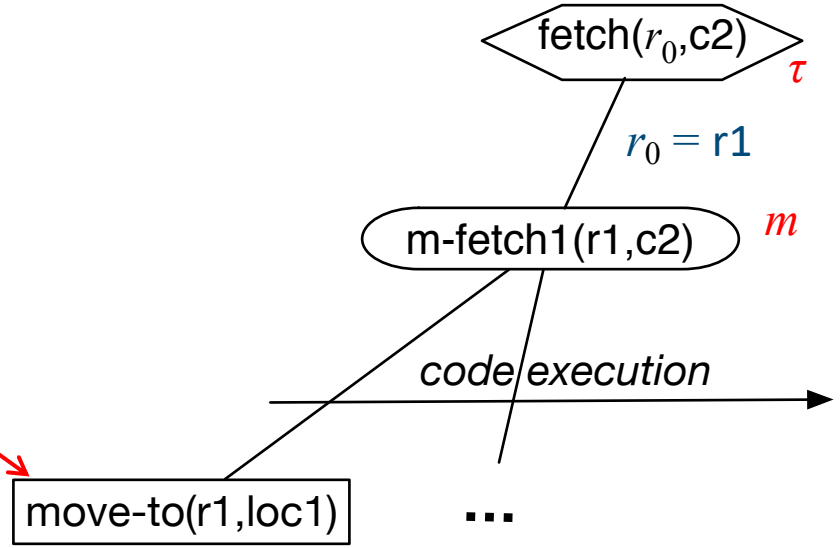
Example

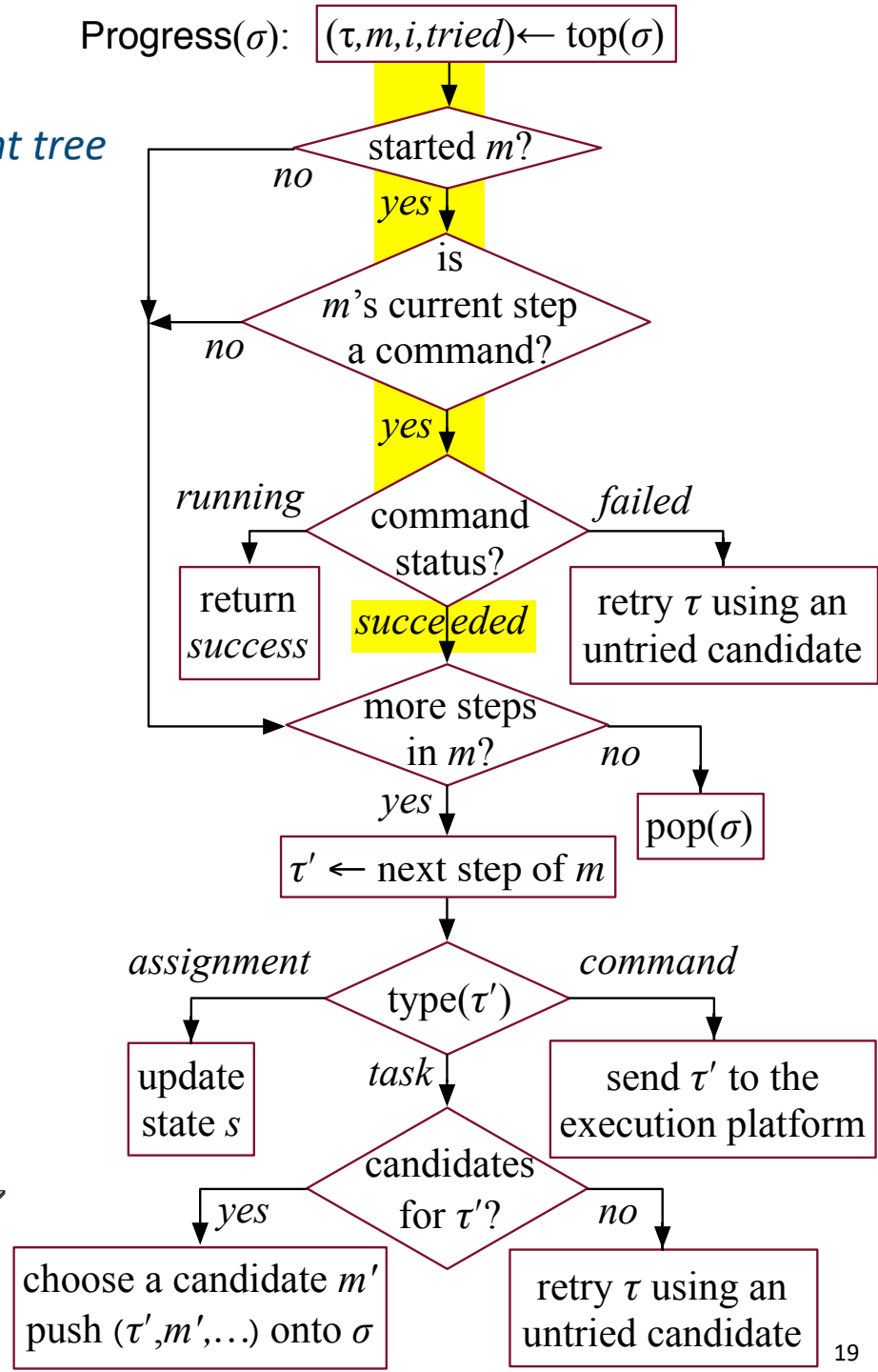
m-fetch1(r,c) r = r1, c = c2
- task: fetch(r,c)
- pre: pos(c) = unknown
- body: l = loc1
  - if ∃l (view(l) = F) then
    - move-to(r,l) ← *succeeded*
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail

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



Refinement tree

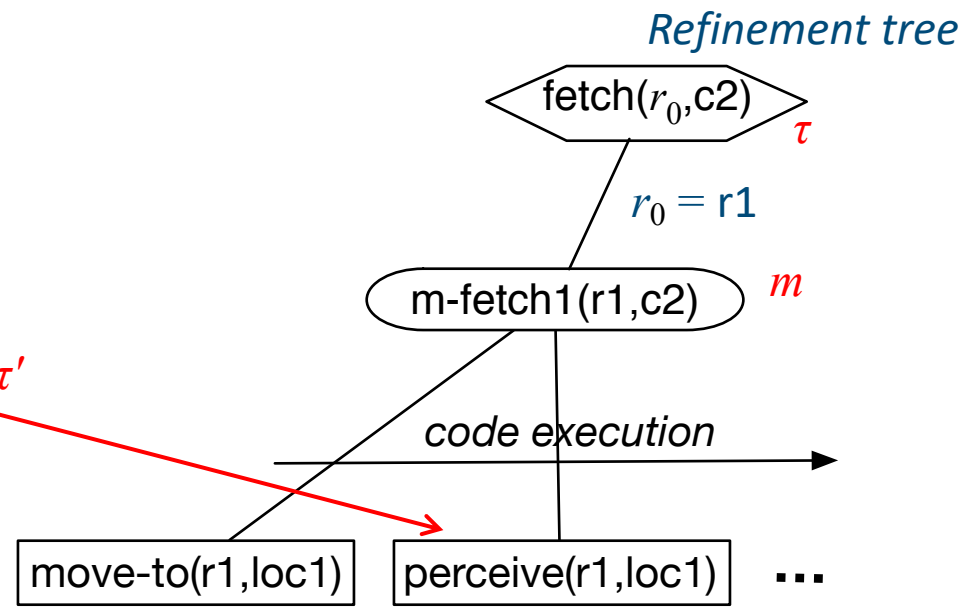


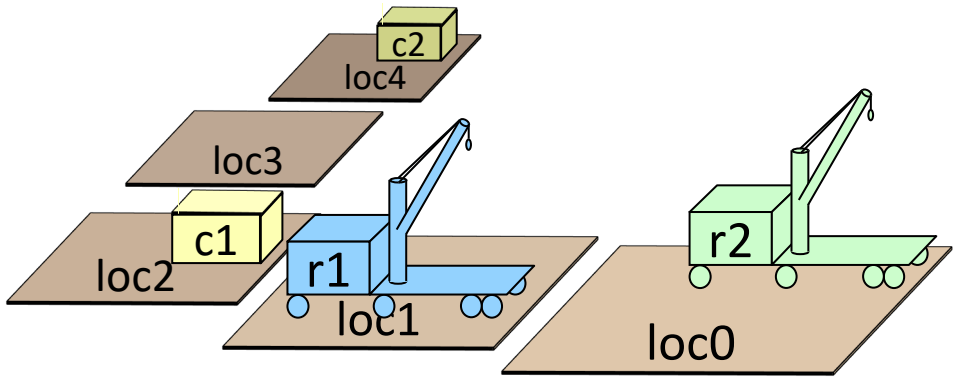


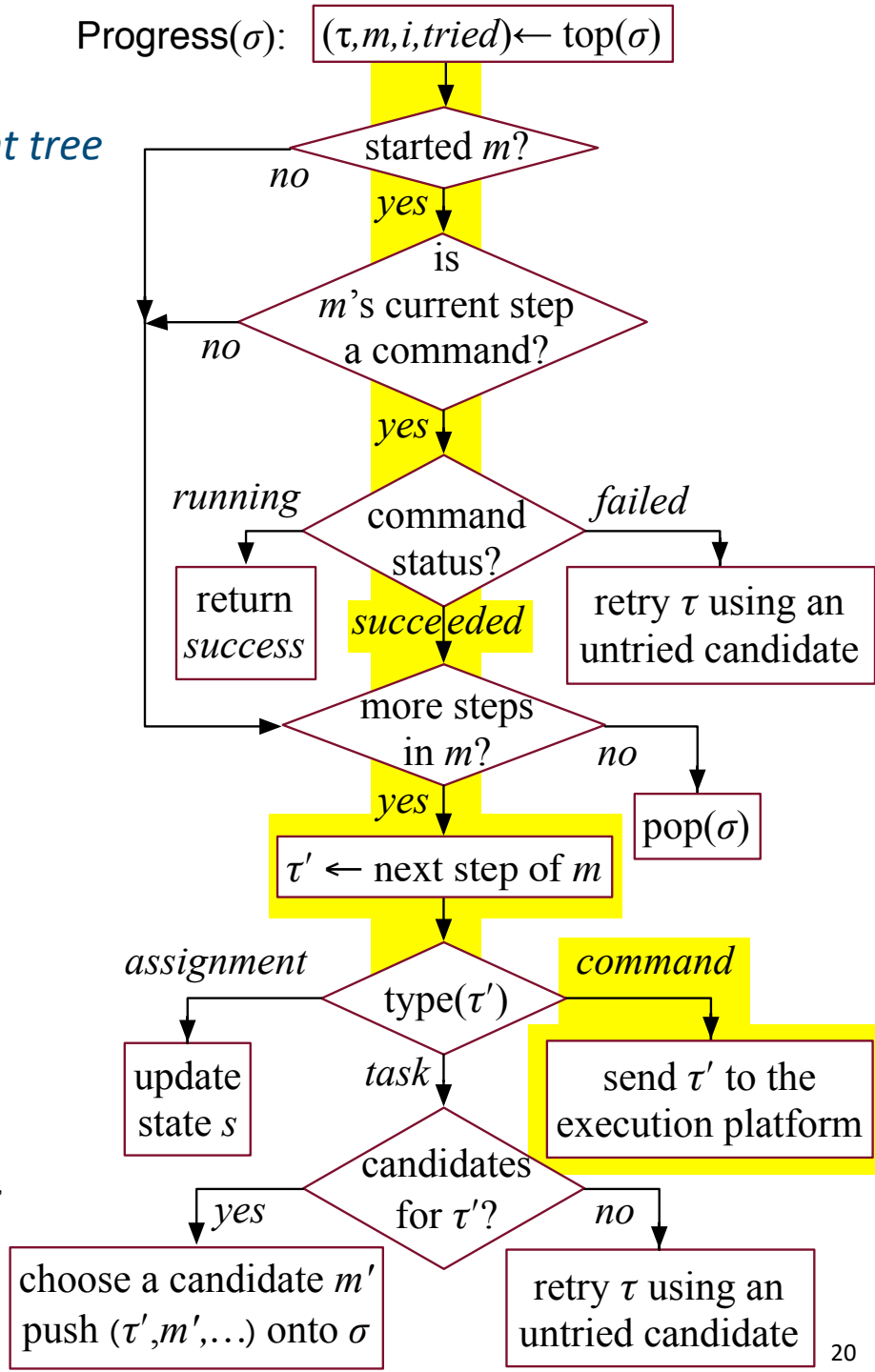
Example

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

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

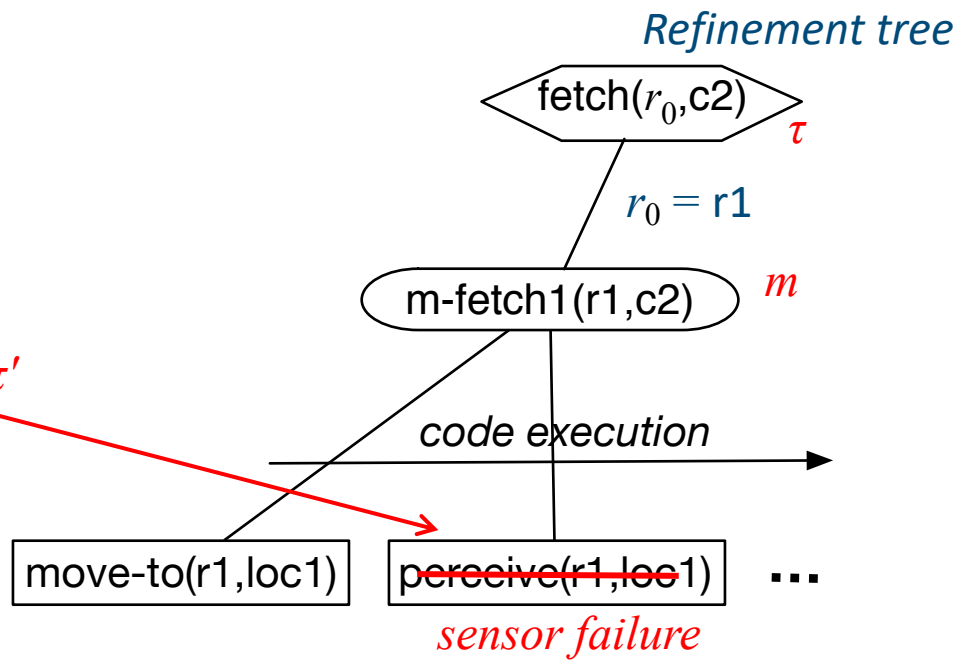


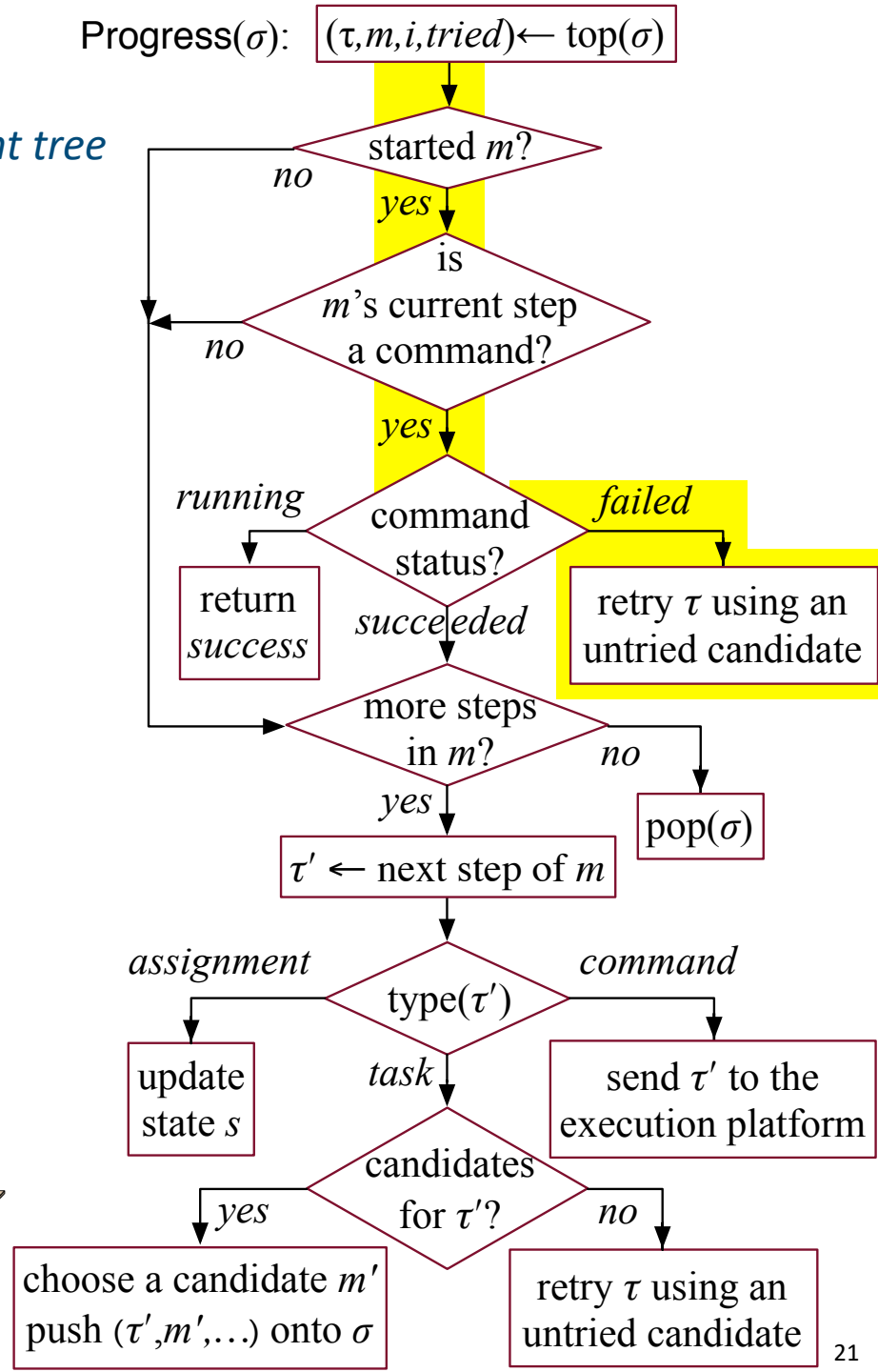




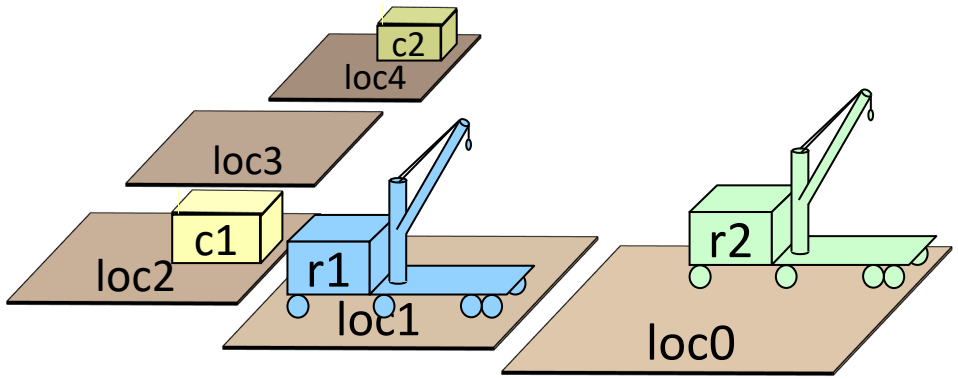
Example

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





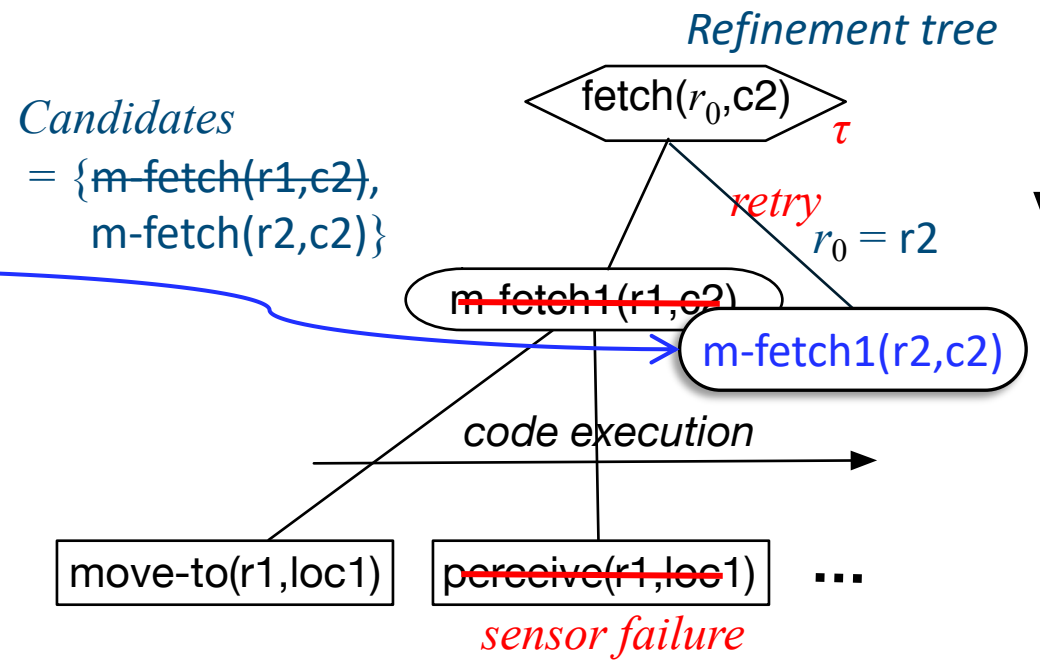
m-fetch2(r,c)
- task: fetch(r,c)
- pre: pos(c) ≠ unknown
- body:
  - if loc(r) = pos(c) then
    - take(r,c,pos(c))
  - else do
    - move-to(r,pos(c))
    - take(r,c,pos(c))

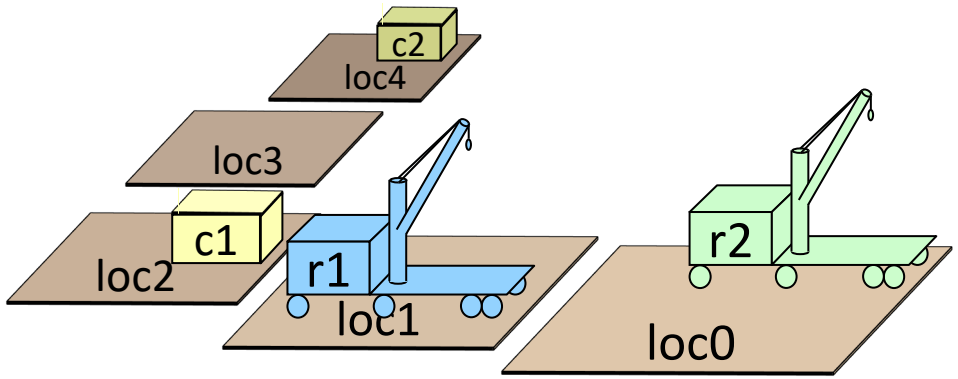


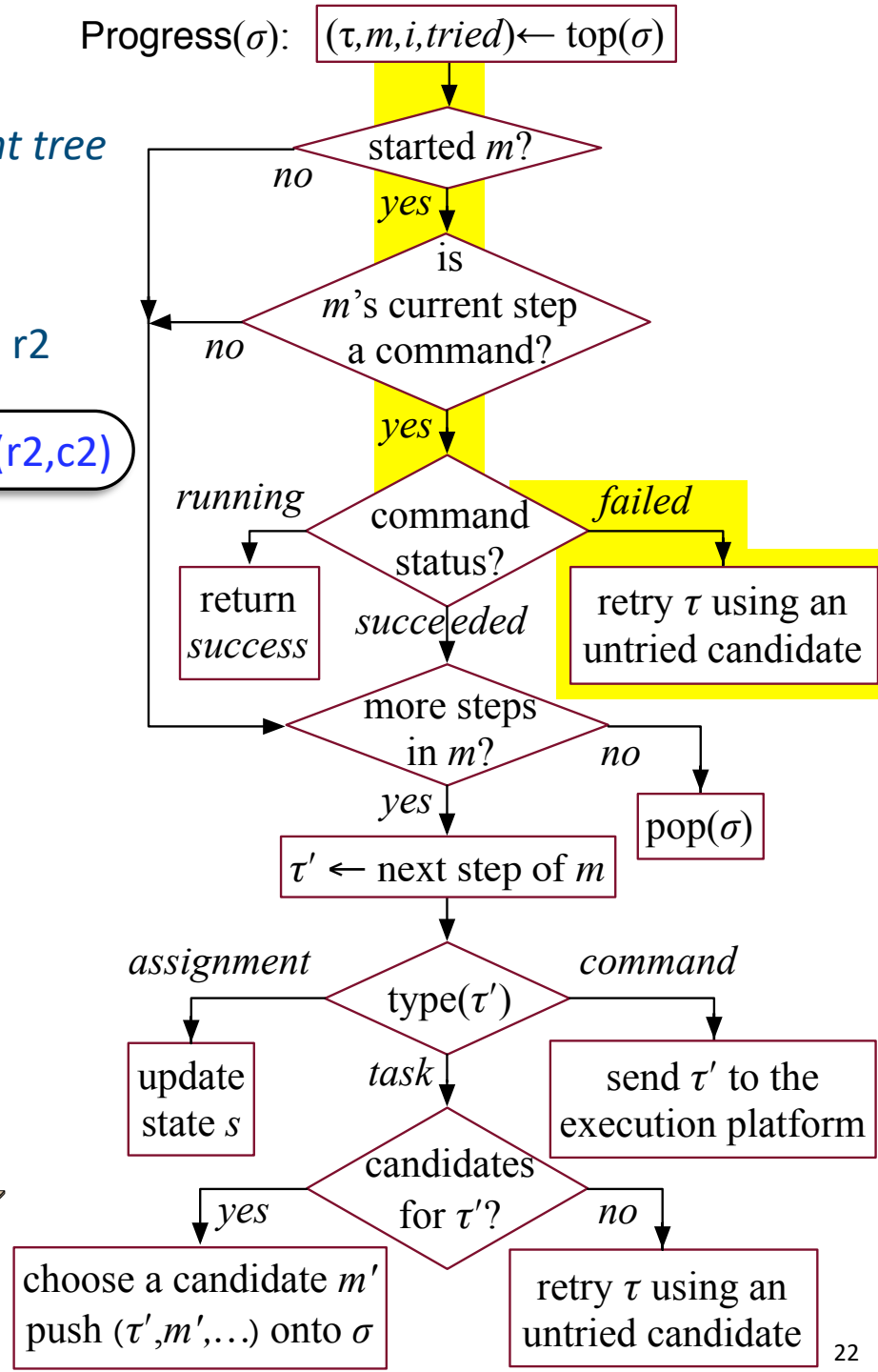
Example

m-fetch1(r,c) r = r2, c = c2
- task: fetch(r,c)
- pre: pos(c) = unknown
- body:
  - if ∃l (view(l) = F) then
    - move-to(r,l)
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail

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



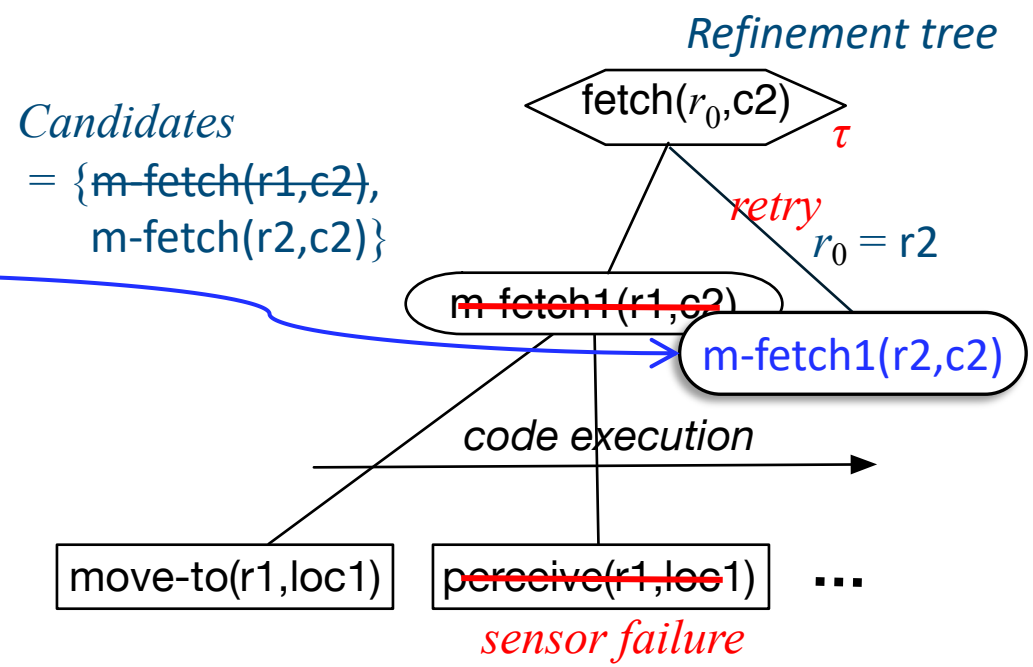




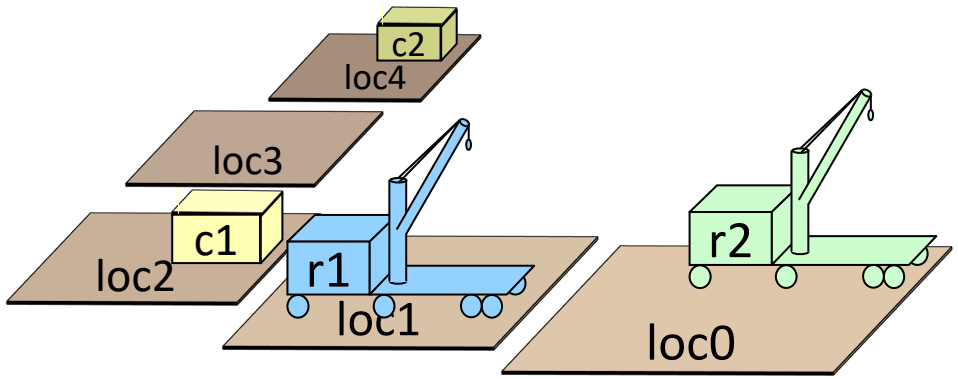
Example

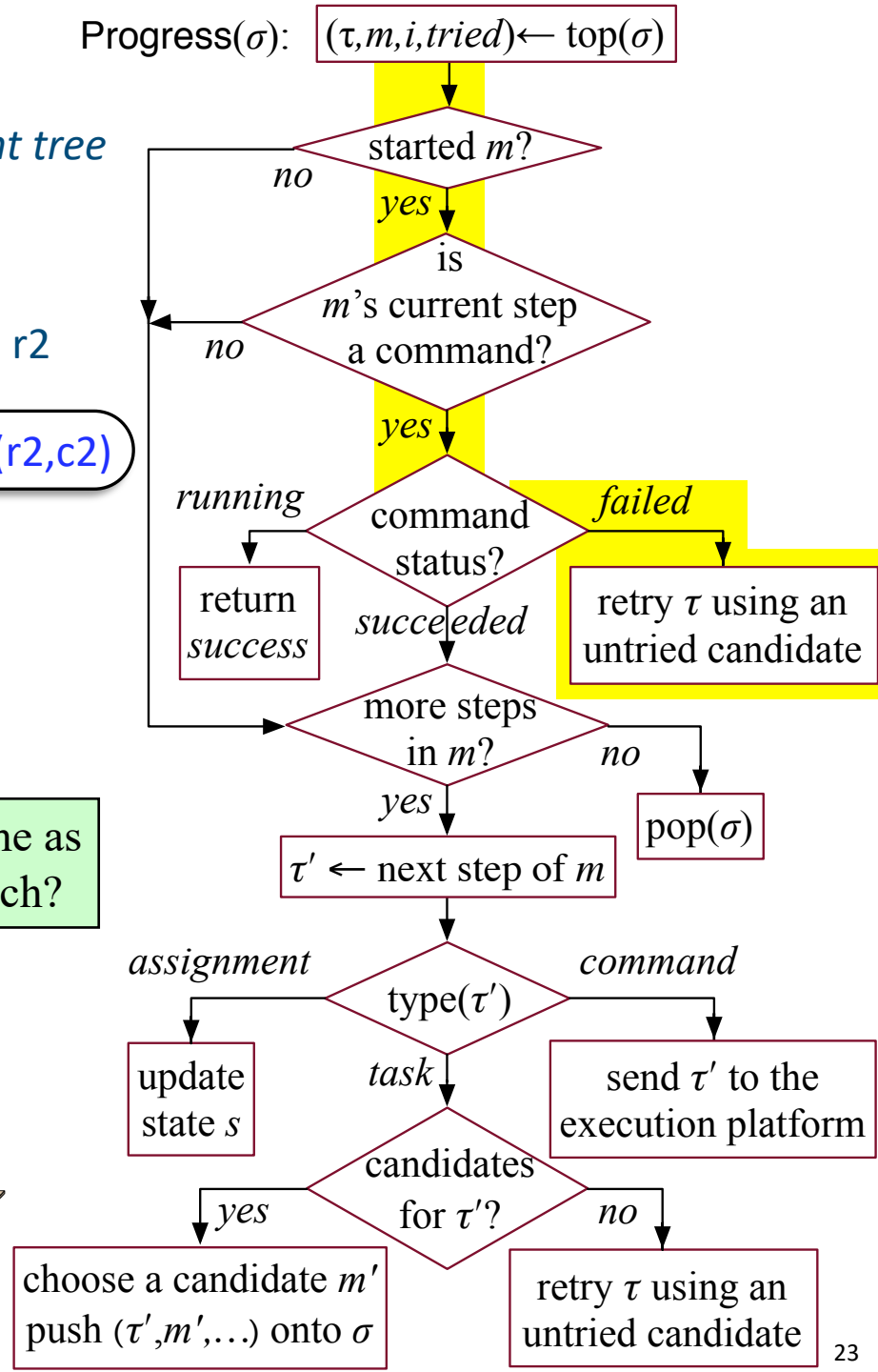
m-fetch1(r,c) r = r2, c = c2
- task: fetch(r,c)
- pre: pos(c) = unknown
- body:
  - if ∃l (view(l) = F) then
    - move-to(r,l)
    - perceive(r,l)
    - if pos(c) = l then
      - take(r,c,l)
    - else fetch(r,c)
  - else fail

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



Poll: Is this the same as a backtracking search?



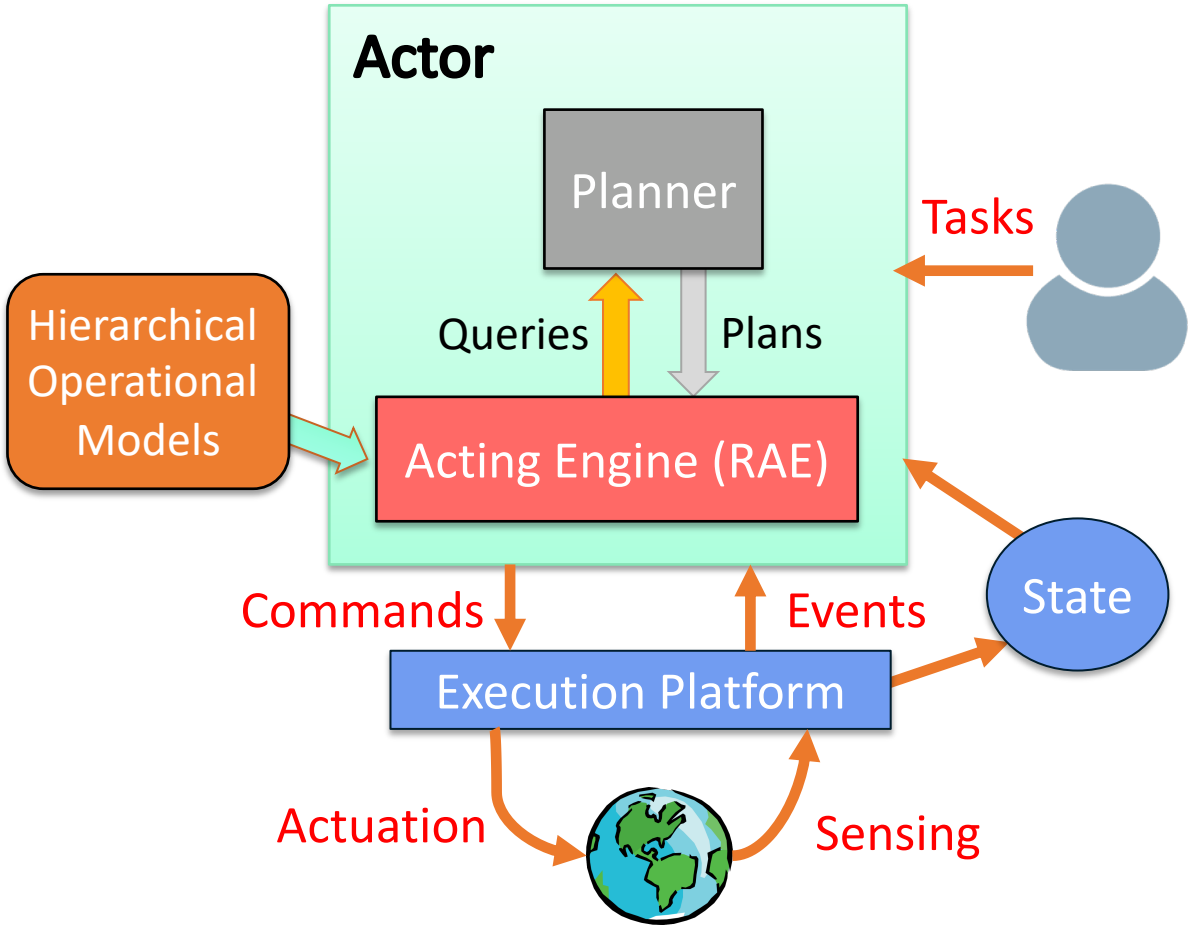


Extensions to RAE

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

Outline

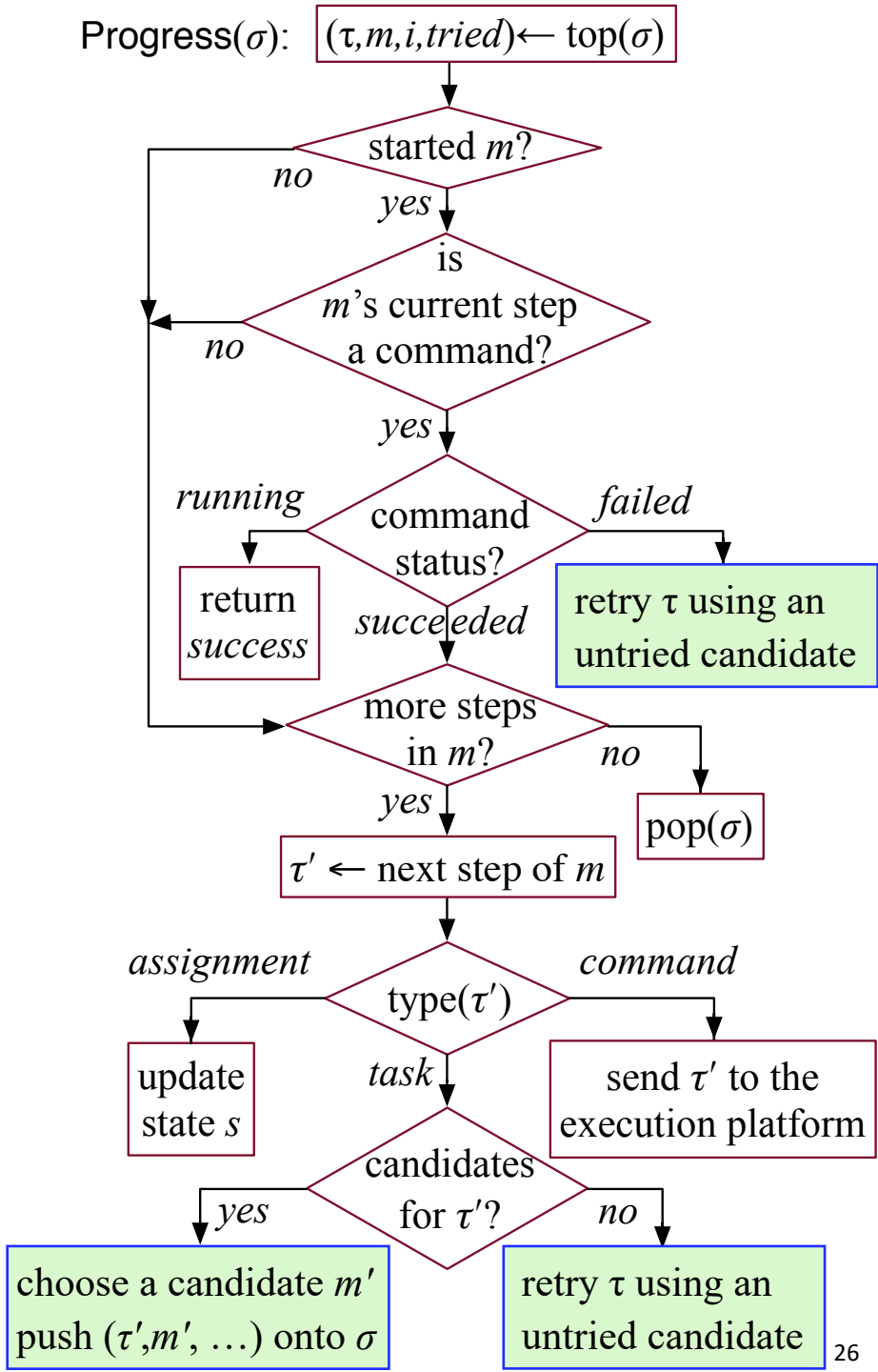
1. Motivation
2. Representation
3. Acting (Rae)
4. *Planning for Rae*
5. Acting with Planning (RAE+UPOM)
6. Learning
7. Evaluation, Application



Planning for Rae?

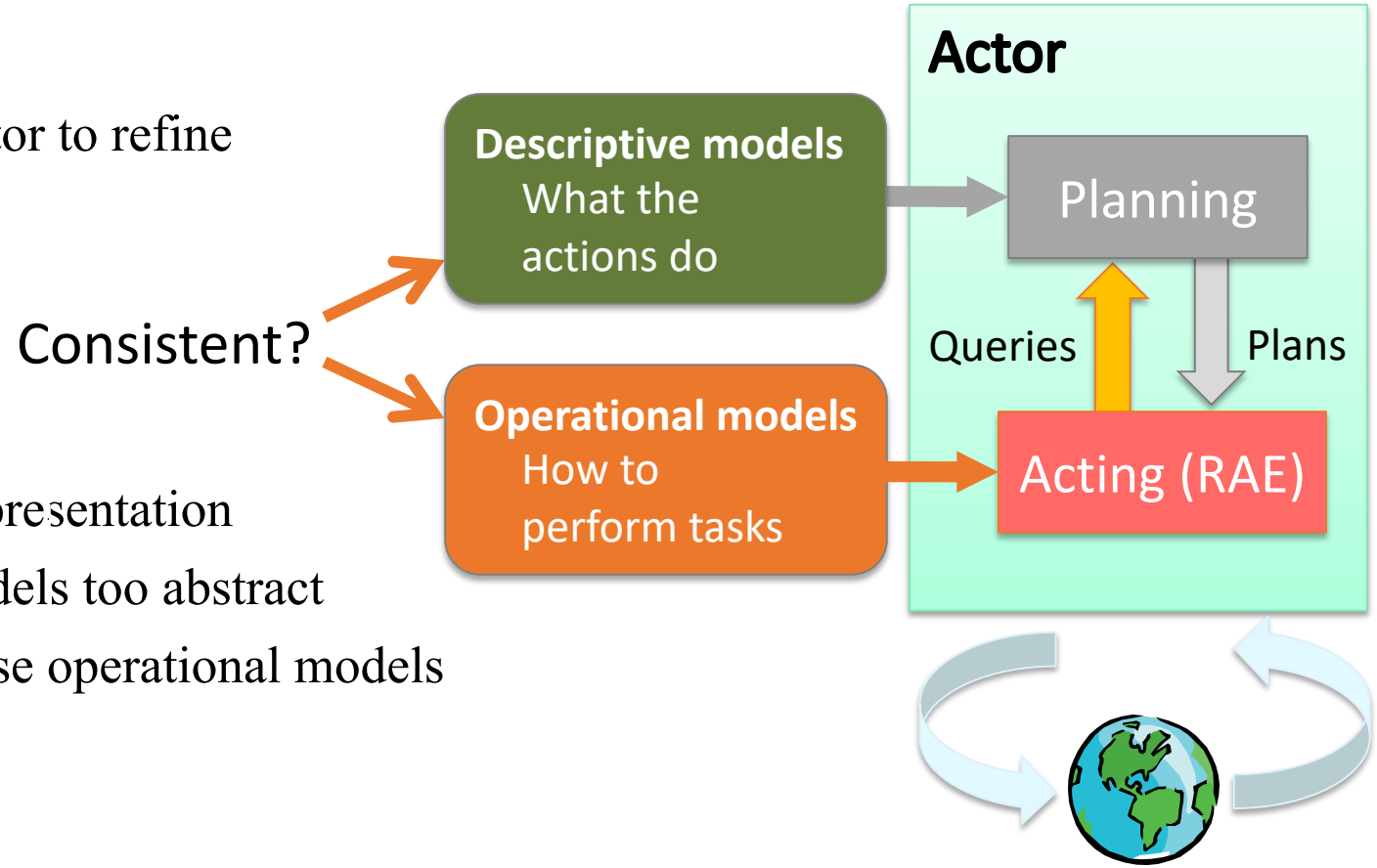
```
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to Agenda
        for each stack σ in Agenda
            call Progress(σ)
            if σ is finished then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

- Four places where Rae and Progress choose a method instance for a task
- Bad choice may lead to
  - more costly solution
  - failure - need to recover, sometimes unrecoverable
- Solution:
  - call a planner, choose the method instance it suggests



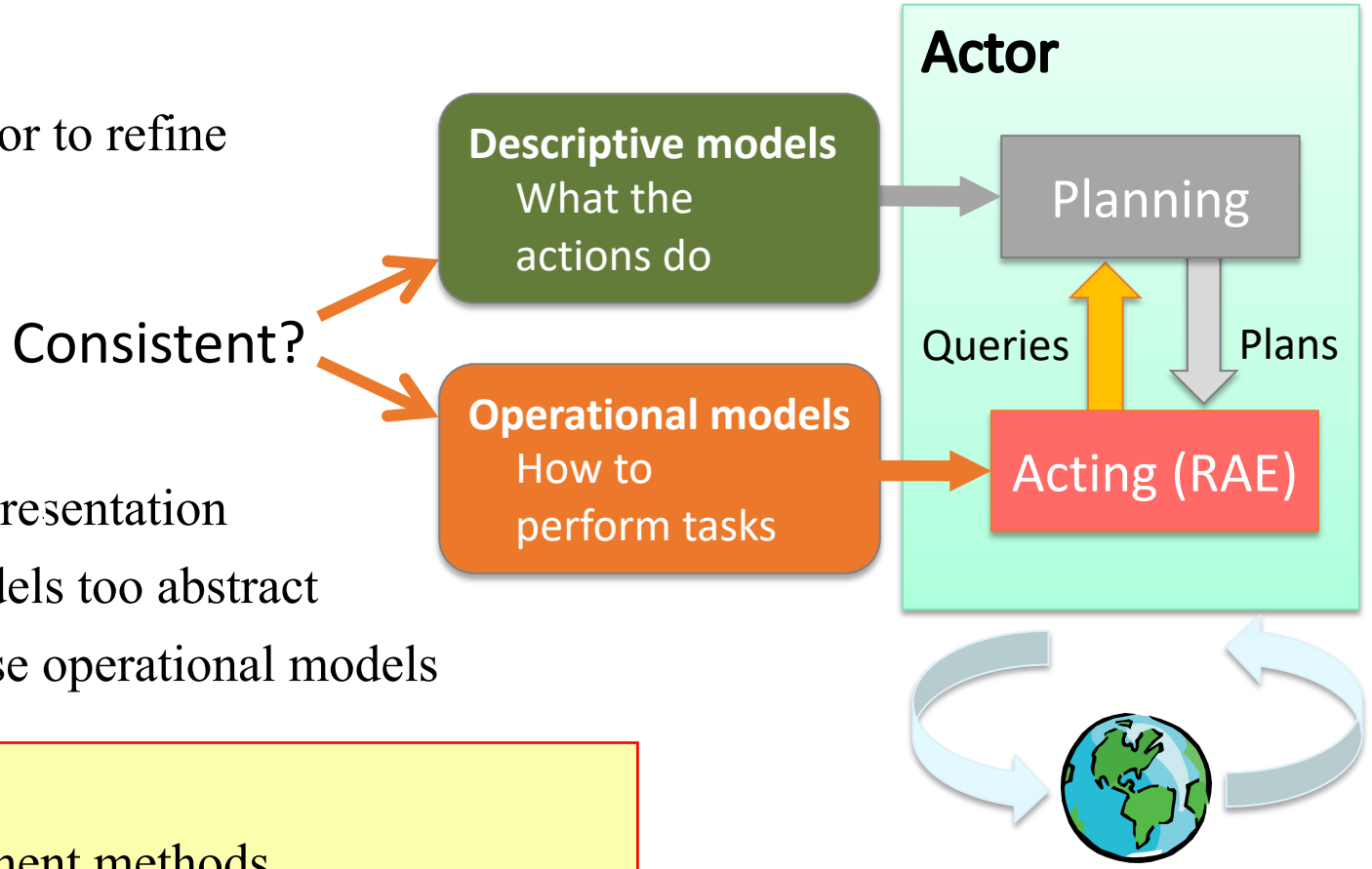
Planning and Acting Integration

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



Planning and Acting Integration

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



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

SeRPE (Sequential Refinement Planning Engine)

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

```
SeRPE(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)
```

```
Progress-to-finish(M, A, s, τ, m)
  i ← nil        // instruction pointer for body(m)
  π ← ⟨⟩         // plan produced from body(m)
  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 ← the 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′
```

- Like Rae with just one external task
  - Progress it all the way to the end, like Progress with a loop around it
  - Plan rather than act
    - For each command, apply a classical action model
- But SeRPE there are problems …

Problems with SeRPE

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

Problem 1: difficult to implement

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

Example:

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

Problems with SeRPE

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

take(r,o,l)

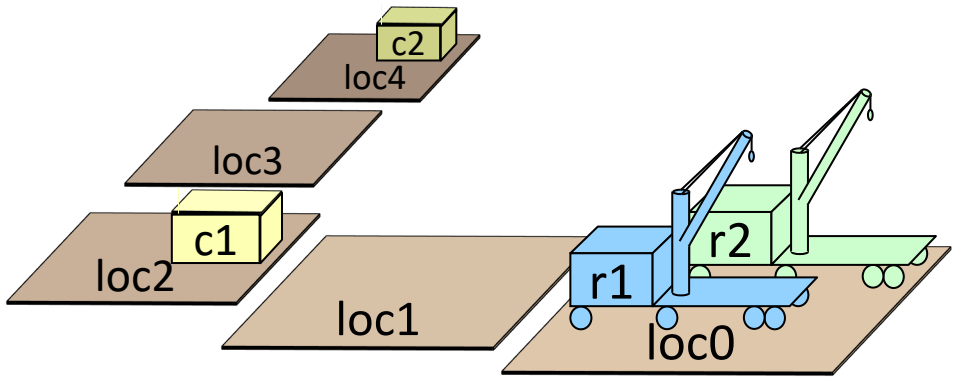
// robot r takes object o at location l
pre: cargo(r) = nil, loc(r) = l, loc(o) = l
eff: cargo(r) ← o, loc(o) ← r

put(r,o,l)

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

perceive(r,l):

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

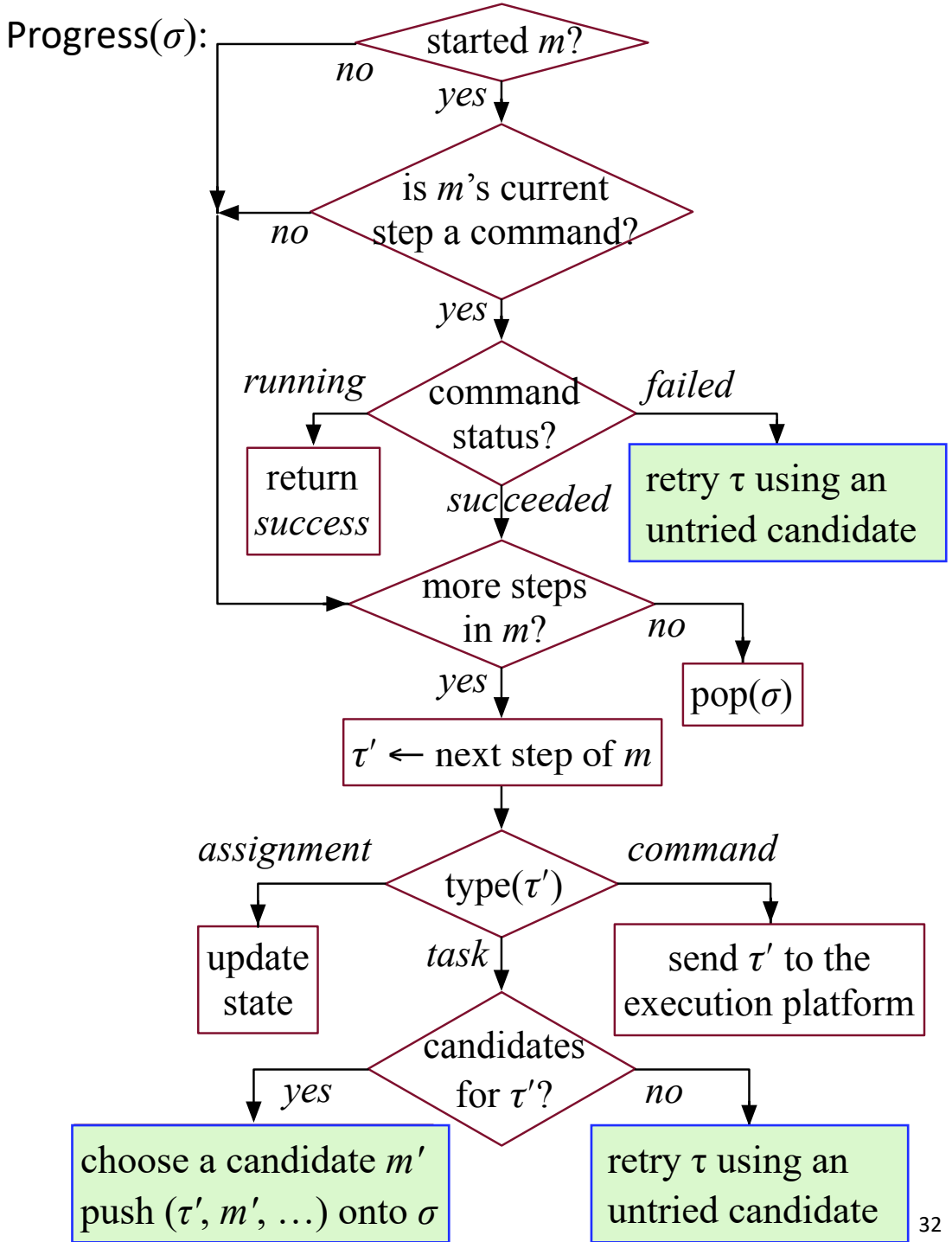


Planning for Rae

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

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

Poll: is this a reasonable approach?

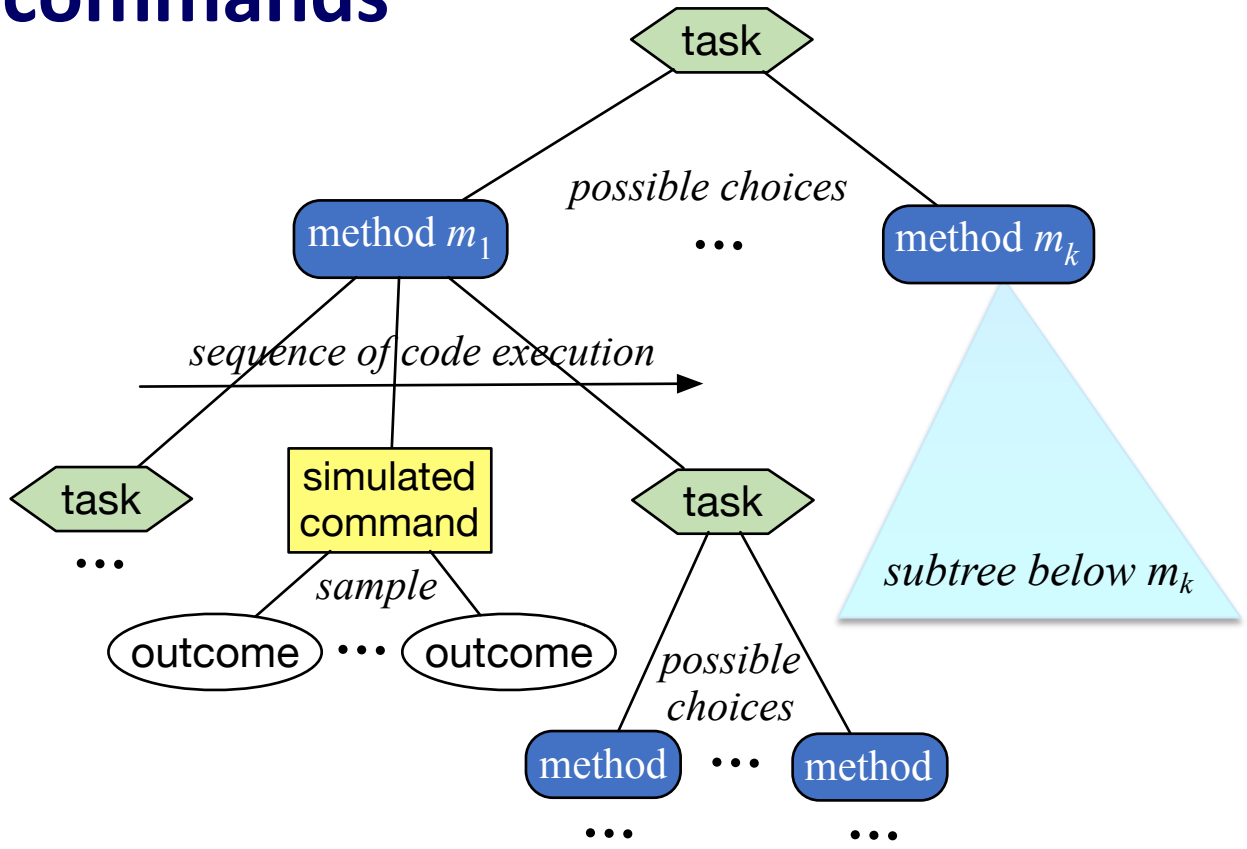


Simulating commands

- Simplest case:
  - probabilistic action template

$$
\begin{array}{l}
a(x_1, \ldots, x_k) \\
\quad \text{pre: } \ldots \\
\quad\quad (p_1)\ \text{effects}_1\text{:}\ \ e_{11}, e_{12}, \ldots \\
\quad\quad\quad \ldots \\
\quad\quad (p_m)\ \text{effects}_m\text{:}\ \ e_{m1}, e_{m2}, \ldots
\end{array}
$$

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

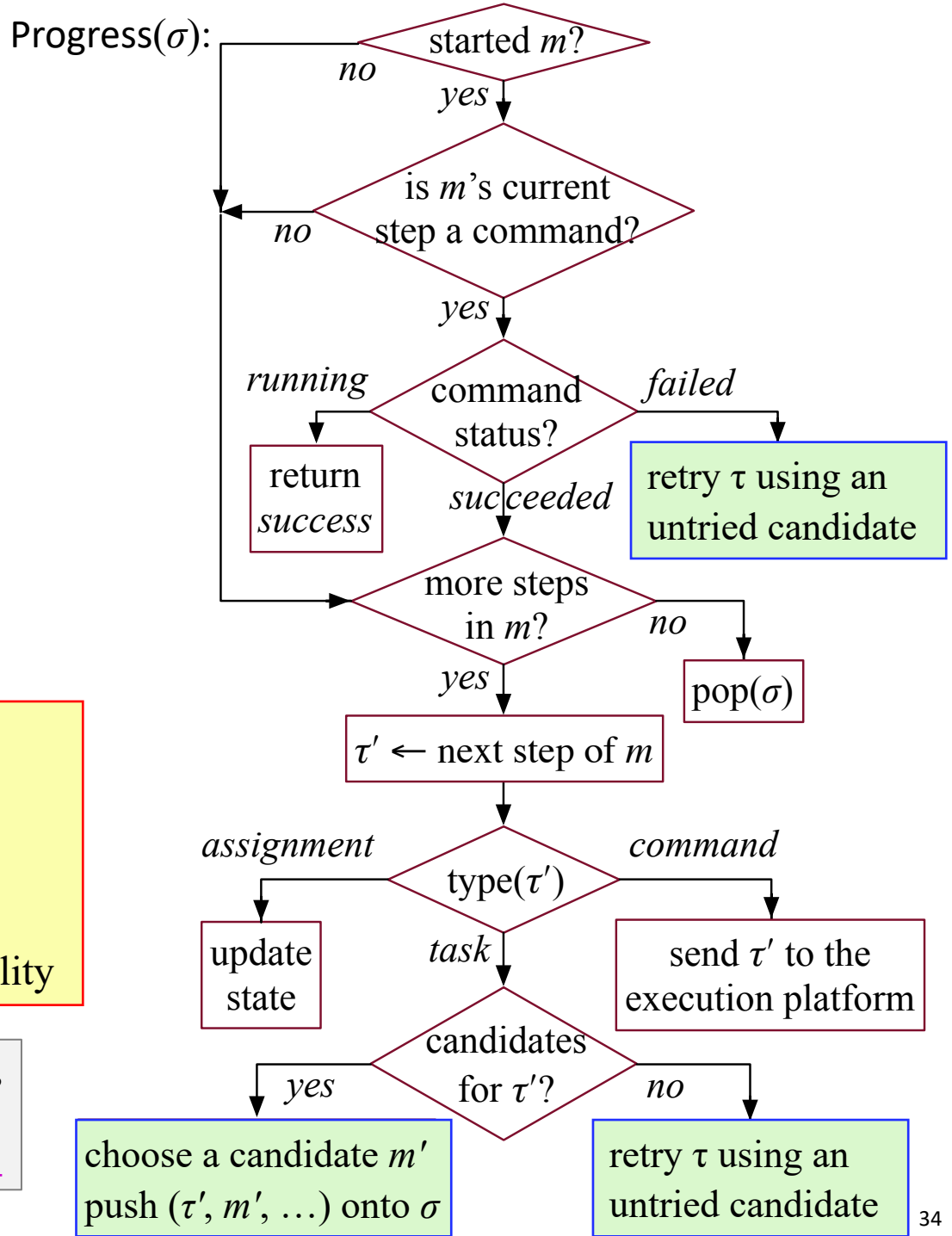


- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get simulated effects

Planning for Rae

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

- Idea 3: simulation with Monte Carlo rollouts
  - Multiple runs
    - Random choices and outcomes in each run
  - Maintain statistics to estimate each choice's expected utility
  - Return the method instance m that has the highest estimated utility



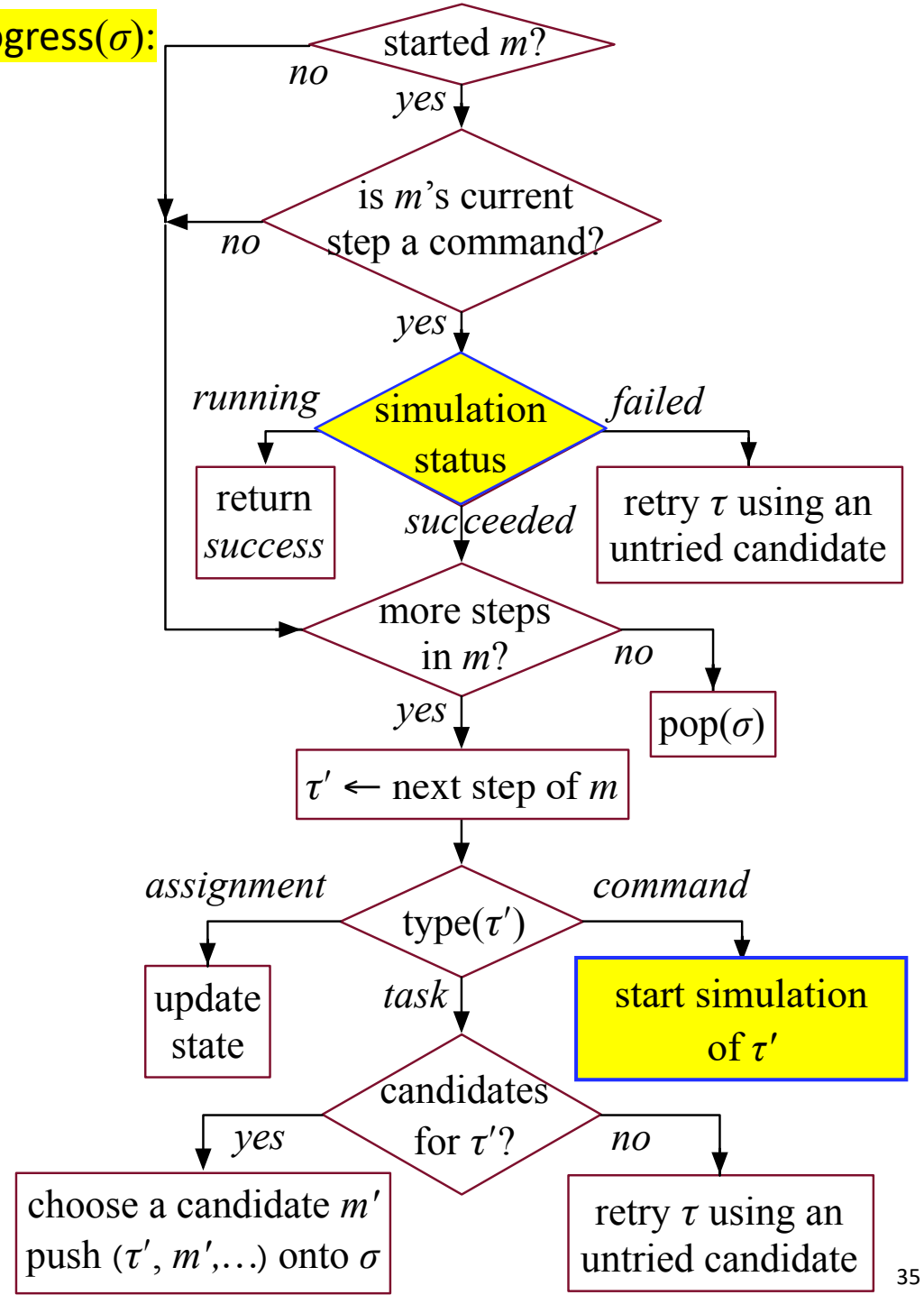
Patra, Mason, Kumar, Traverso, Ghallab, and Nau. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. *ICAPS*, 2020. Best student paper honorable mention award. https://doi.org/10.1609/aaai.v33i01.33017691

Planner

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

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

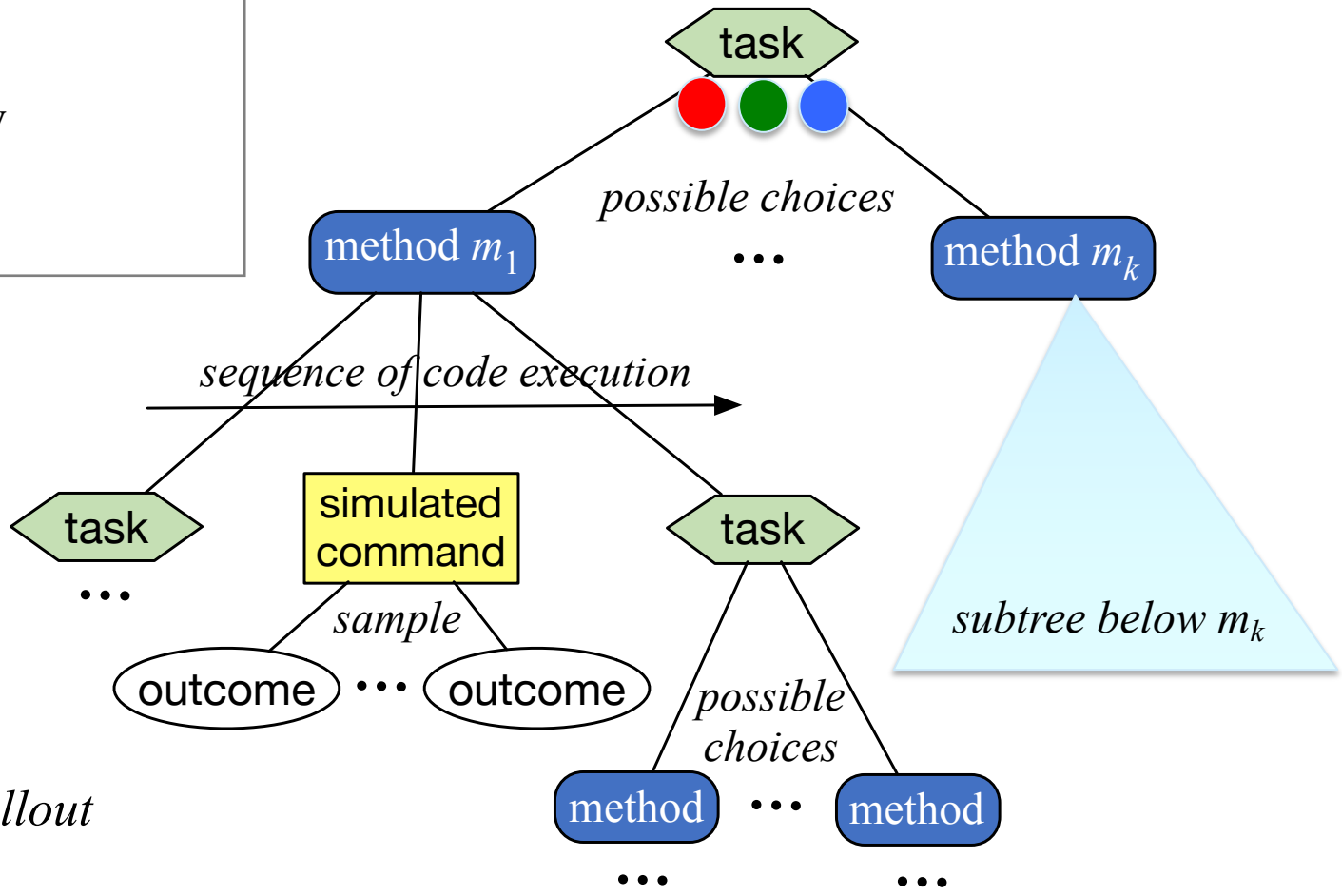
- Each call to UPOM does a Monte Carlo rollout
  - Simulated execution of RAE on τ



Monte-Carlo rollouts

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

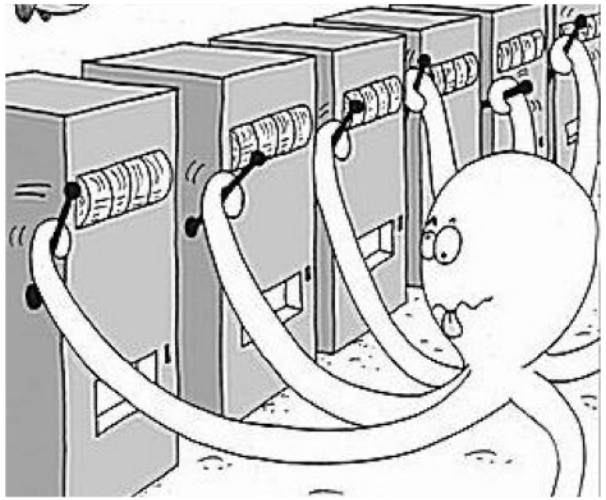
UPOM(τ):
  choose a method instance m for τ
  create refinement stack σ for τ and m
  loop while Simulate-Progress(σ) $\neq$ *failure*
    if σ is completed then return (m, *utility*)
  return *failure*



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

Digression: Monte Carlo rollouts

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



UCB (Upper Confidence Bound) Algorithm

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

UCB:

if there are any untried actions:

$\tilde{a} \leftarrow$ any untried action

else:

$\tilde{a} \leftarrow \text{argmax}_a\, Q(a)$

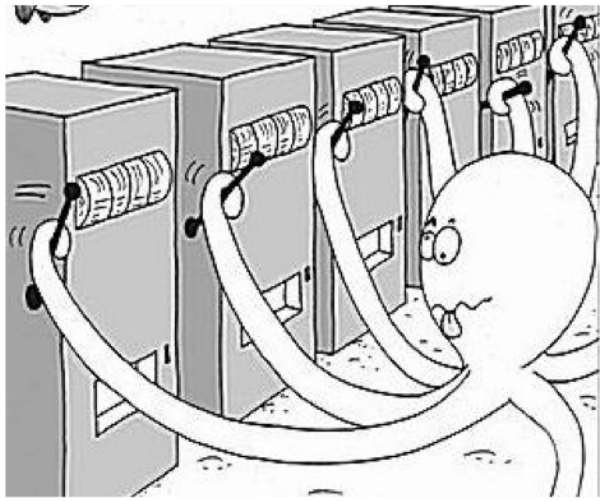
perform $\tilde{a}$

update $r(\tilde{a})$, $n(\tilde{a})$, n_t, $Q(\tilde{a})$

- Theorem (given some assumptions):

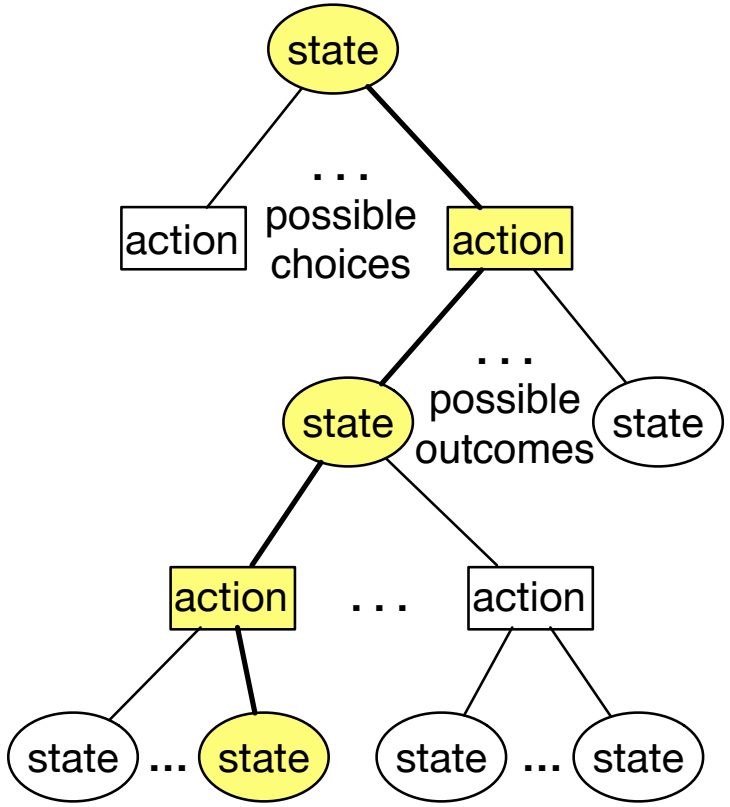
  As the number of calls to UCB $\rightarrow \infty$,

  UCB's choice at each call $\rightarrow$ optimal choice



UCT Algorithm

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



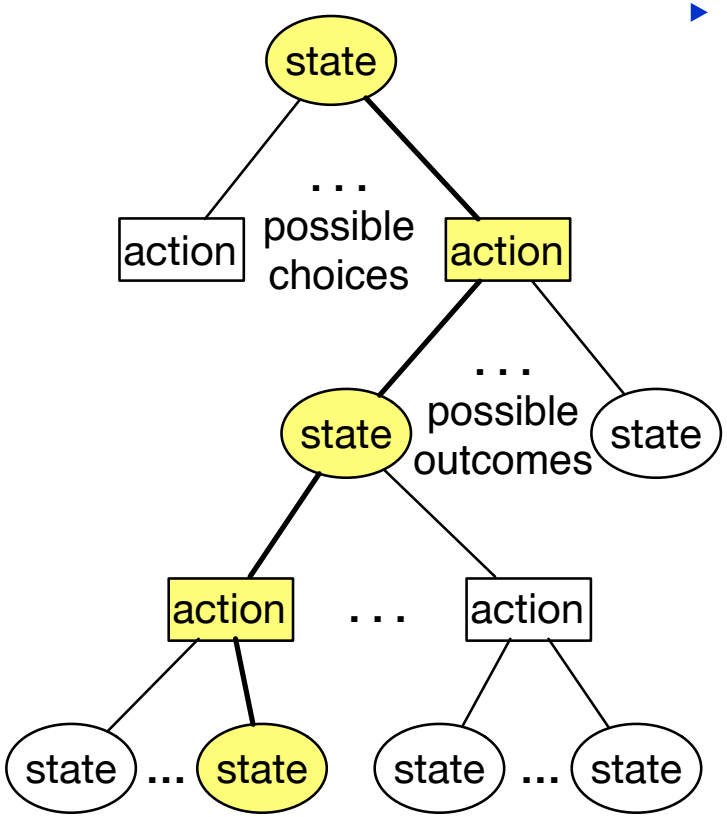
- How to use UCT:
  - Call it many times, return action with highest expected utility
- Theorem:

  As number of calls to UCT → ∞,

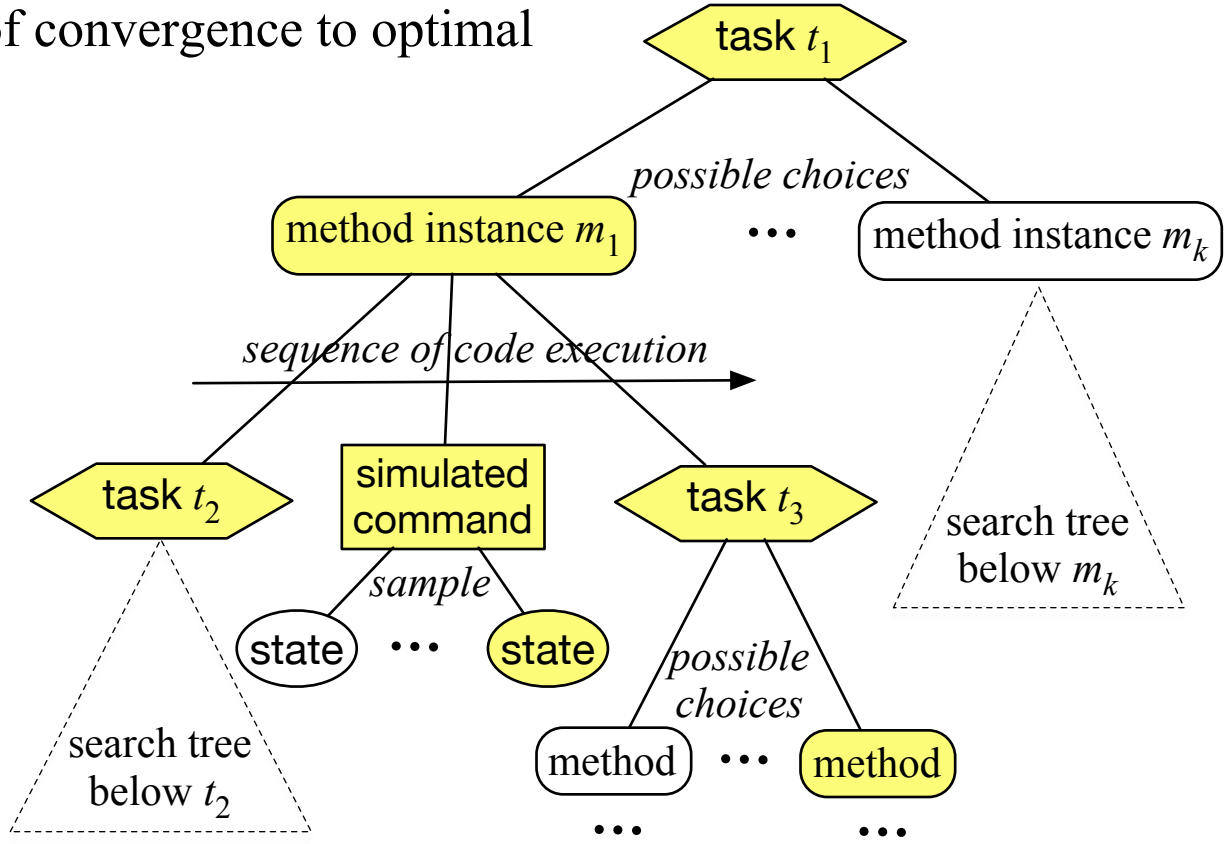
  choice converges to optimal

Convergence

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

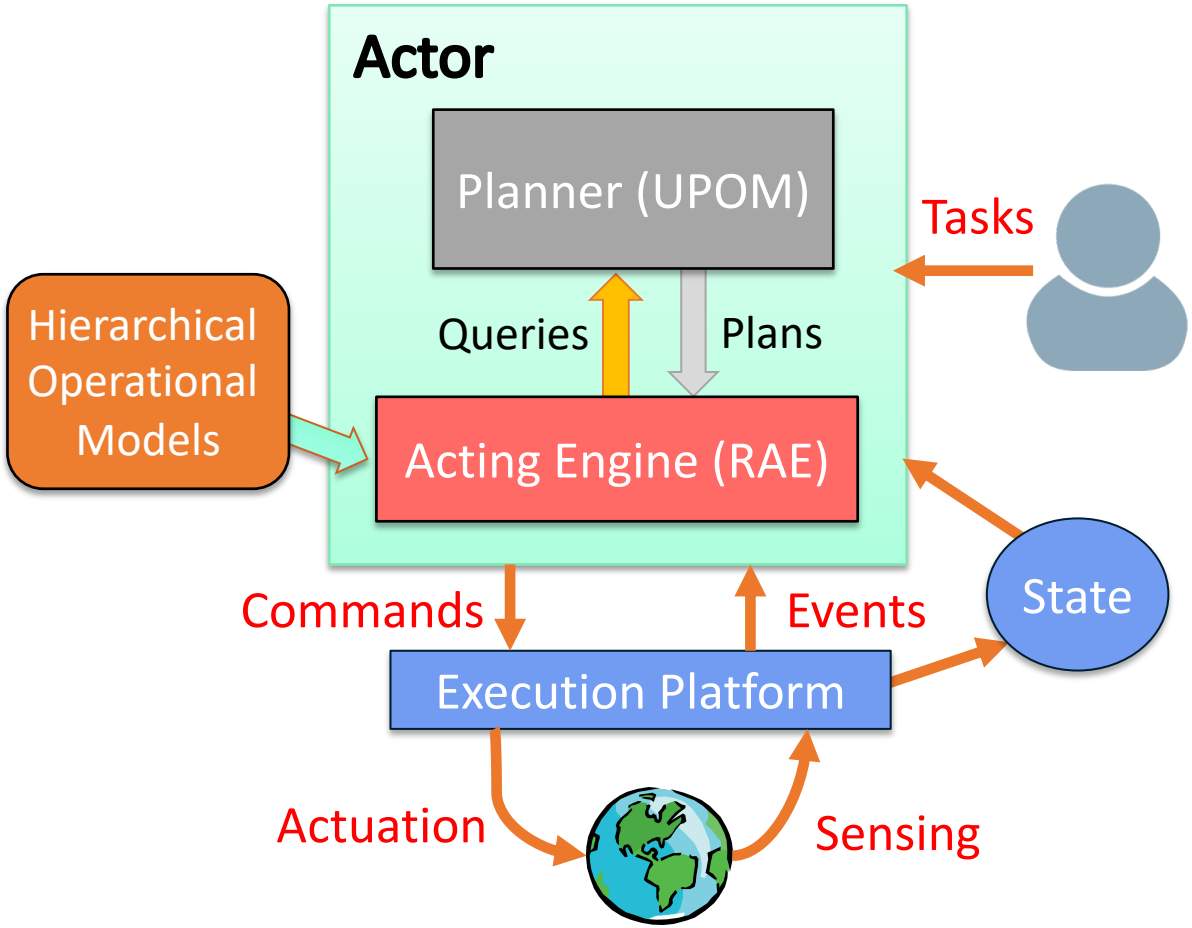


- UPOM search tree more complicated
  - tasks, method instances, commands, code execution
- If no exogenous events,
  - Can map it to UCT search of a complicated MDP
  - Proof of convergence to optimal



Outline

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

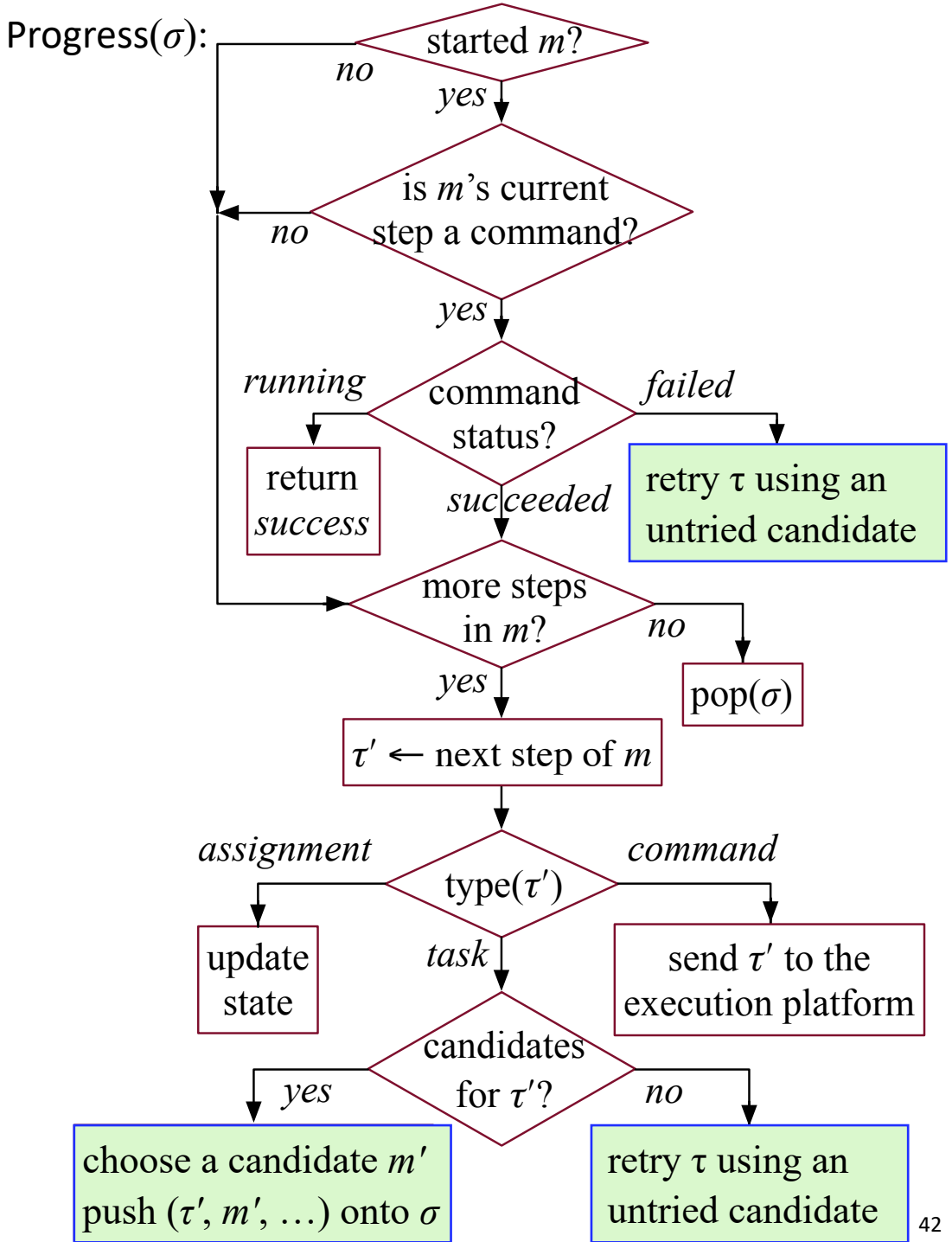


RAE + UPOM

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

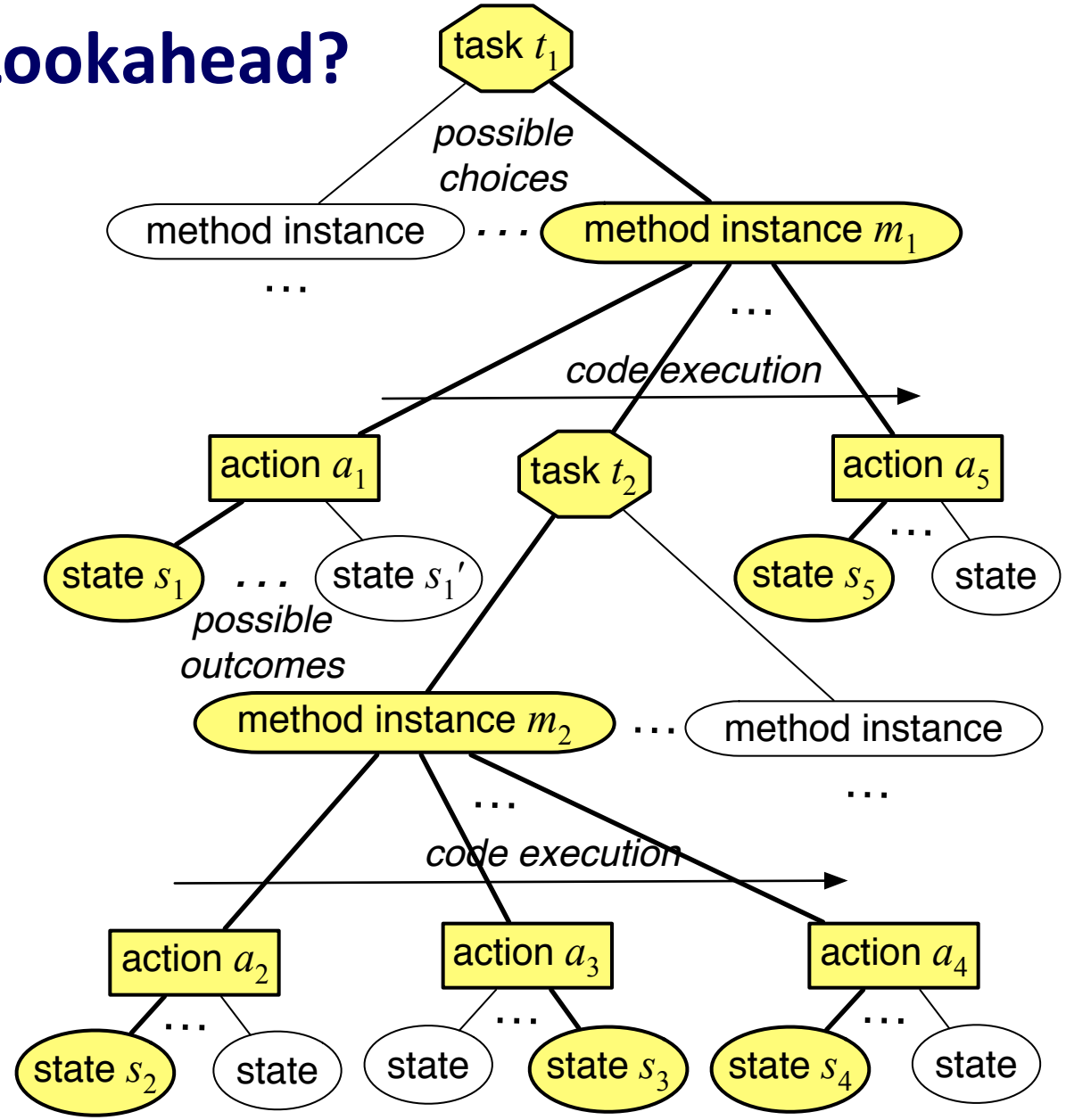
- Whenever RAE needs to choose a method instance
  - call Plan-with-UPOM, use the method instance it returns

- Open-source Python implementation: https://bitbucket.org/sunandita/RAE/



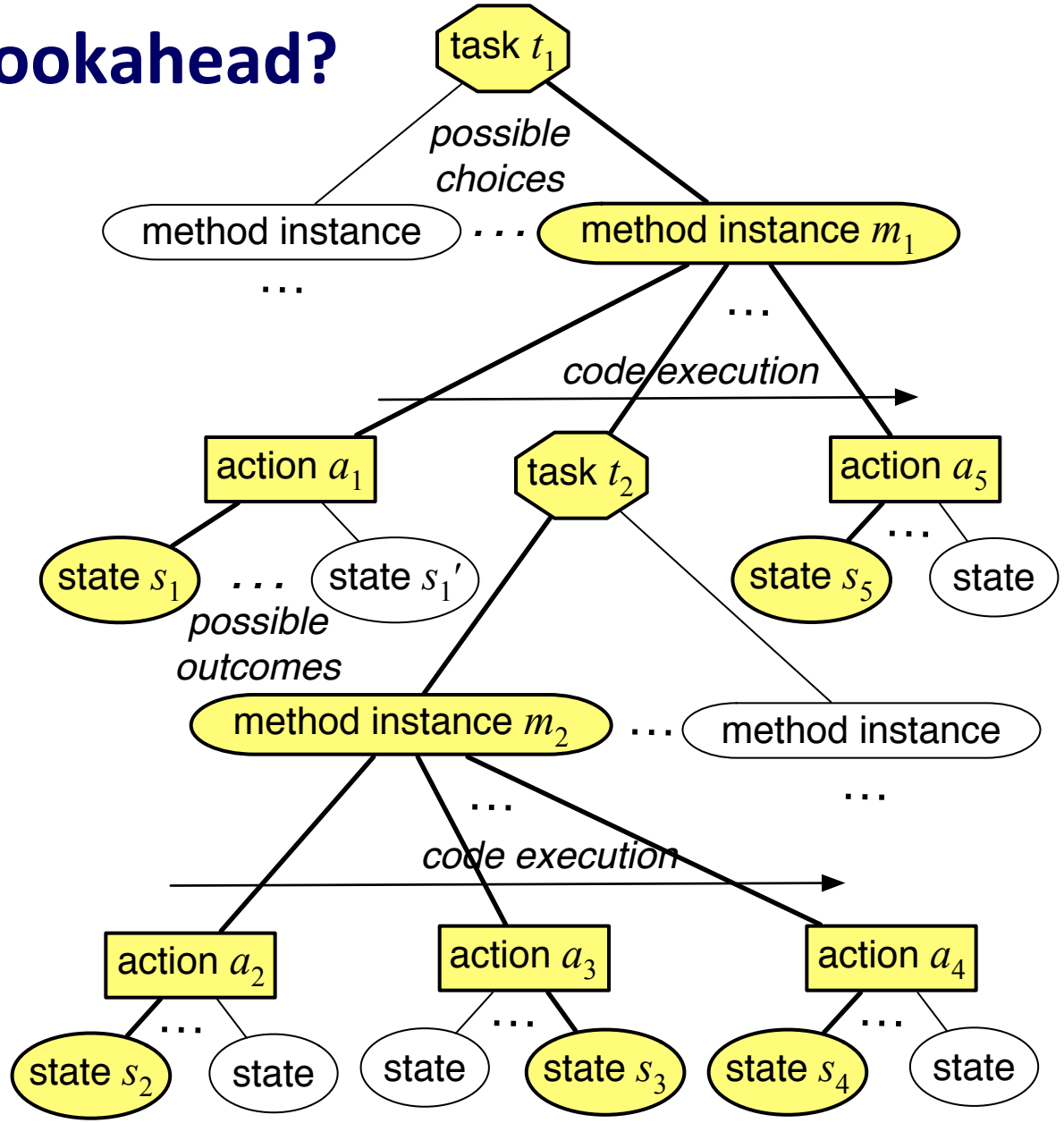
Can we use UPOM with Run-Lookahead?

- Suppose we try to use Run-Lookahead with a modified version of UPOM (call it UPOM′)
  - Instead of returning method instance m_1, return the actions in the last Monte Carlo rollout
    - $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
    - corresponding commands: c_1, c_2, c_3, c_4, c_5
- Problem
  - Run-lookahead calls UPOM′, gets π, executes c_1, then calls UPOM′ again
  - This time, UPOM′ needs to plan for t_1 in state s_1 rather than s_0
  - There might not be an applicable method
- If we want to use Run-Lookahead, we need to ensure that methods can work in unexpected states



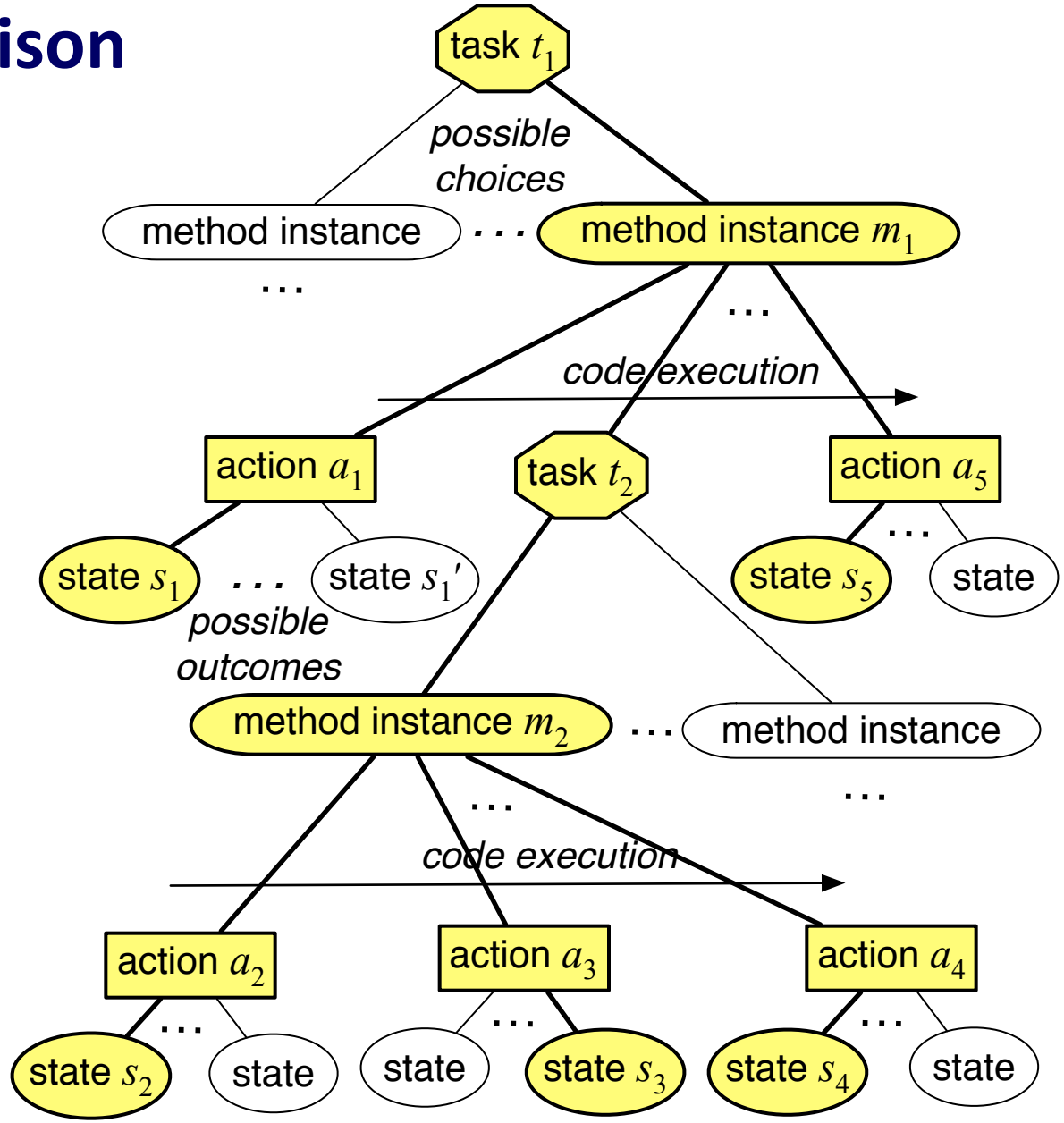
Can we use UPOM with Run-Lazy-Lookahead?

- Run-Lazy-Lookahead calls UPOM′, UPOM′ returns $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
- Run-Lazy-Lookahead executes c_1, c_2, c_3, c_4, c_5, won't call UPOM′ again unless something unexpected happens, e.g.,
  - command c_2 has an execution failure
  - c_2 produces a state in which c_3 is inapplicable
  - or an exogenous event makes c_3 inapplicable
  - Method m_2 fails; we need to replan task t_2
- Need to modify Run-Lazy-Lookahead so that when a failure occurs, it knows which task to replan
  - Need to modify the methods to work in unexpected states



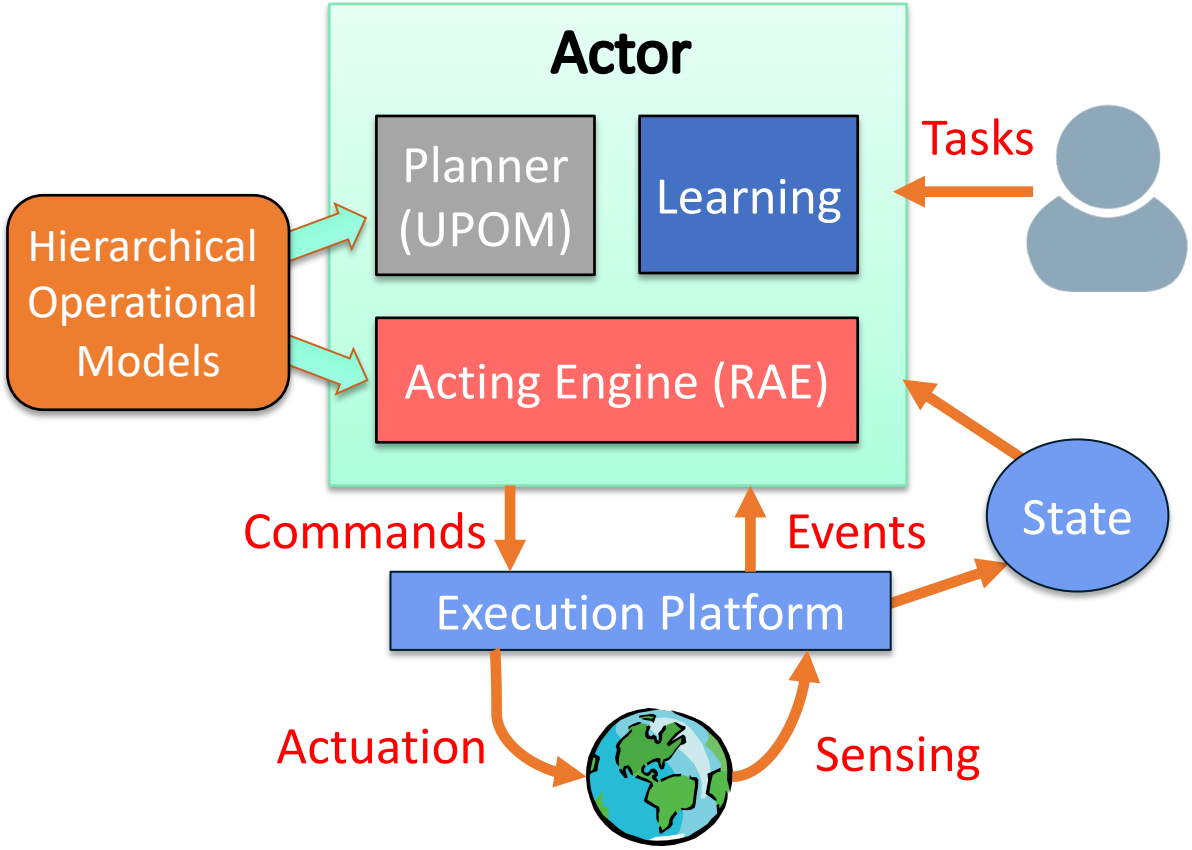
Comparison

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



Outline

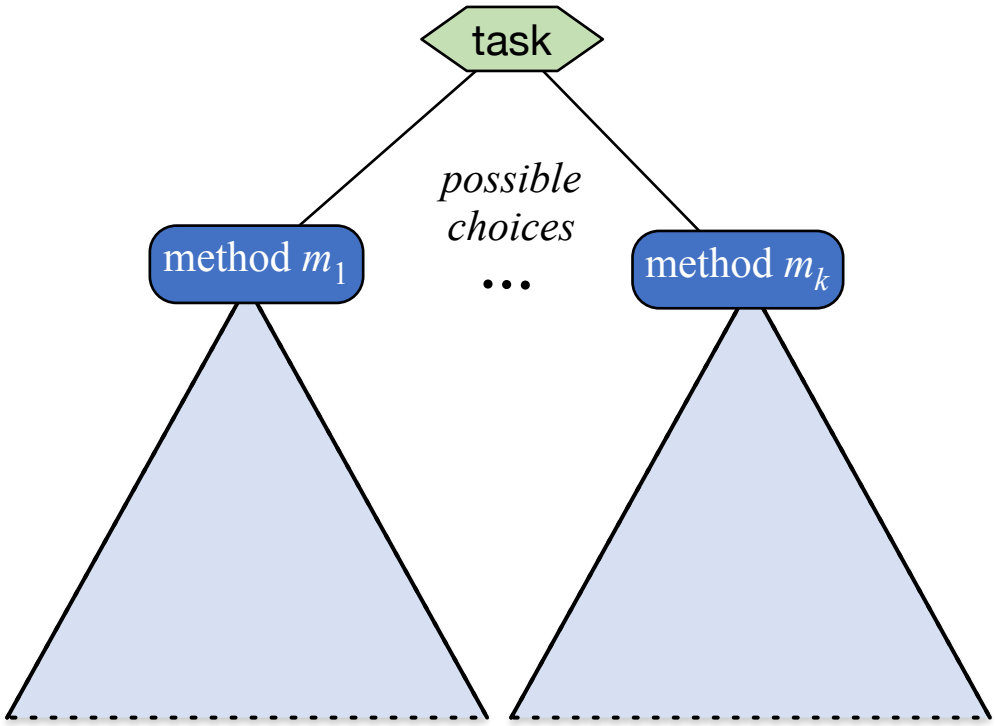
1. Motivation
2. Representation
3. Acting (Rae)
4. Planning for Rae
5. Acting with Planning (RAE+UPOM)
6. *Learning*
7. Evaluation, Application



Motivation

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

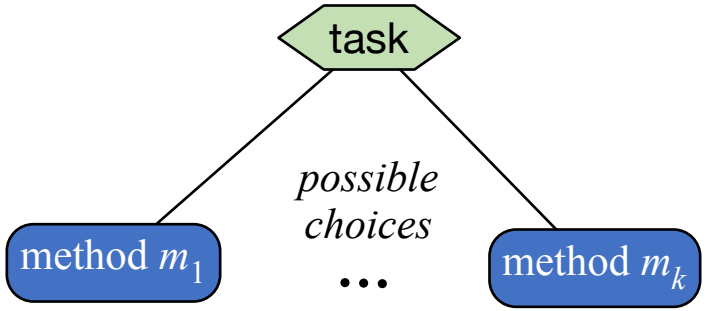
- Case 1: no time to search at all
  - need a choice function
- Case 2: enough time to do partial search
  - Receding horizon
    - Cut off search at depth d_{max} or when we run out of time
    - At leaf nodes, use heuristic function to estimated expected utility



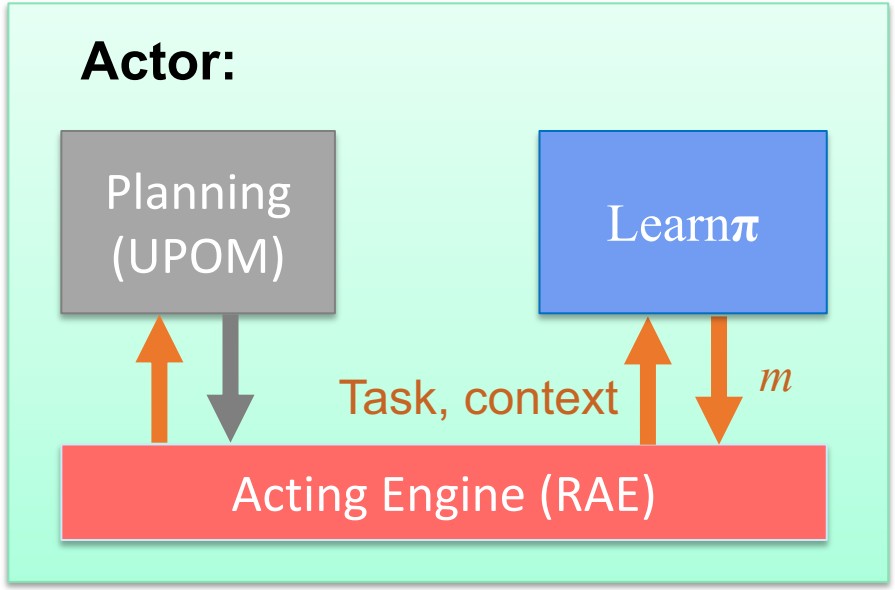
- Learning algorithms:
  - Learn$\boldsymbol{\pi}$: learns a choice function
  - LearnH: learns a heuristic function

Integration with Learning

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

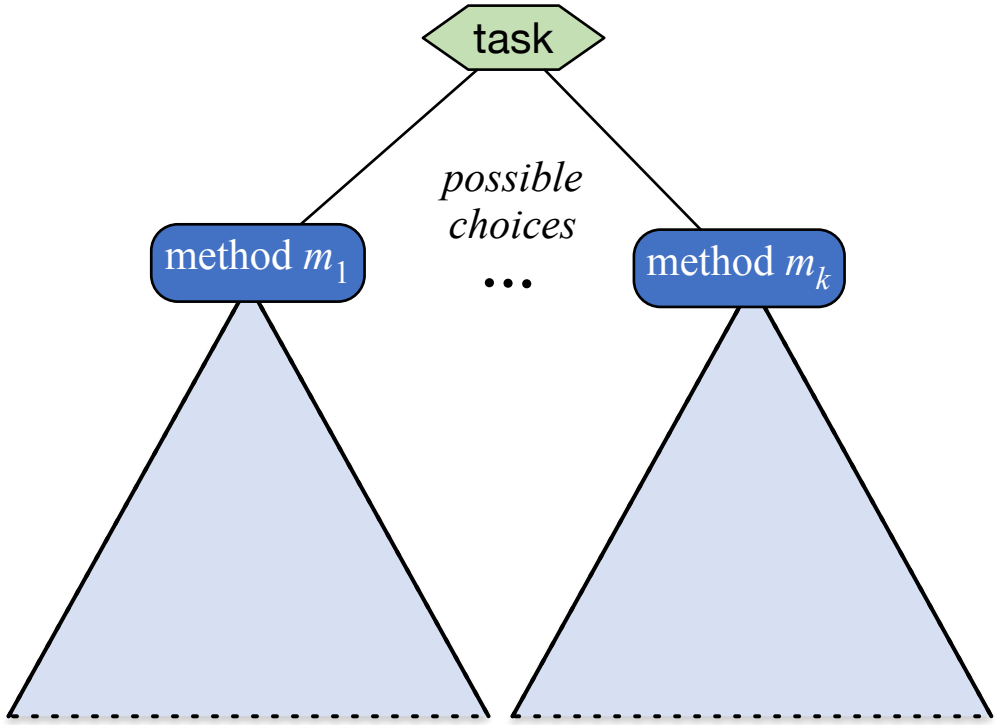


- Learn$\boldsymbol{\pi}$
  - Learns function for choosing a method
  - Given current task and context (state and other information), choose m from the set of available refinement methods
  - Useful if there isn't enough time to use UPOM

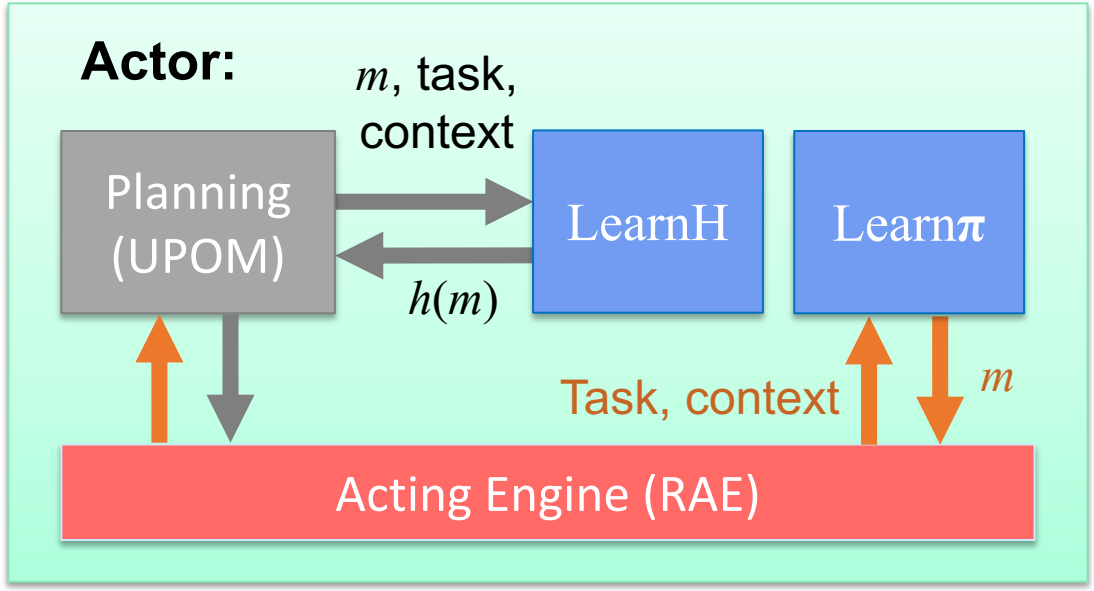


Integration with Learning

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

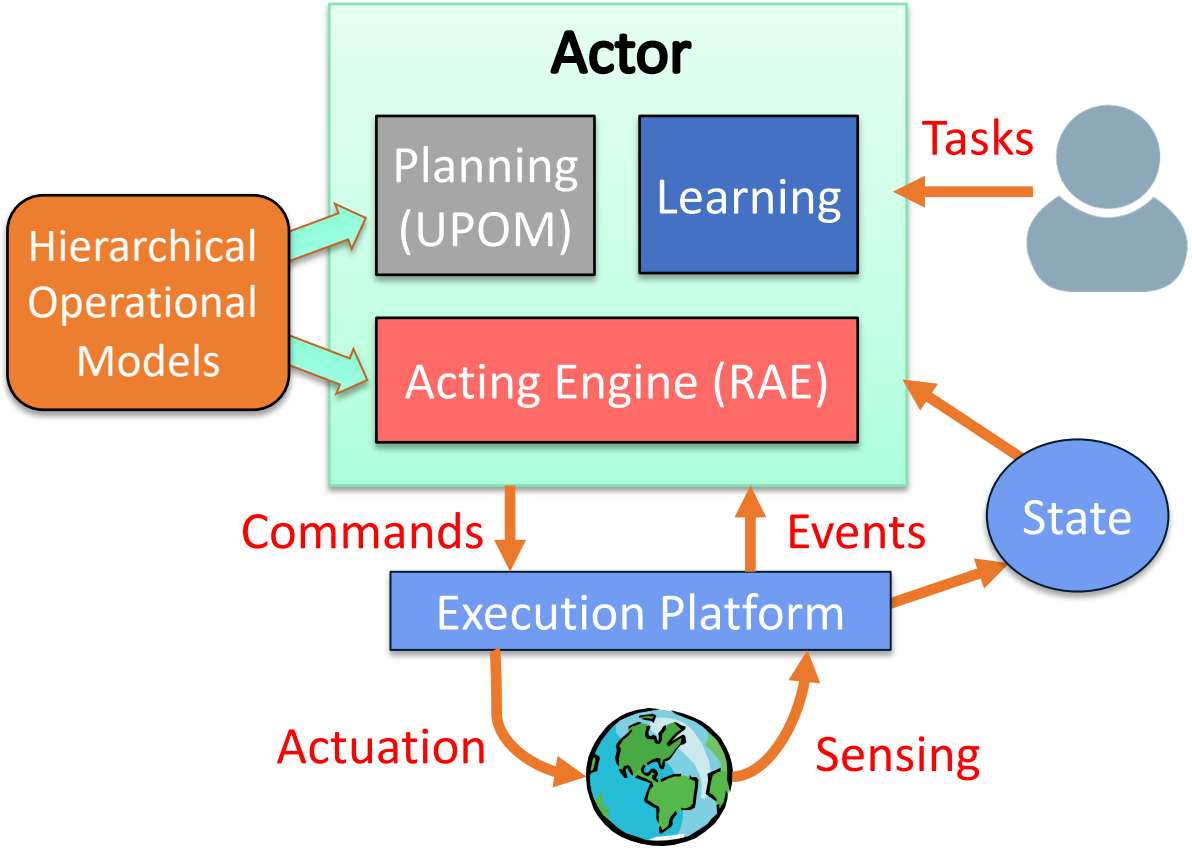


- LearnH
  - Learns a heuristic function to guide UPOM's search
  - UPOM can use it to estimate expected utility at leaf nodes
  - Useful if there isn't enough time to search all the way to the end



Outline

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



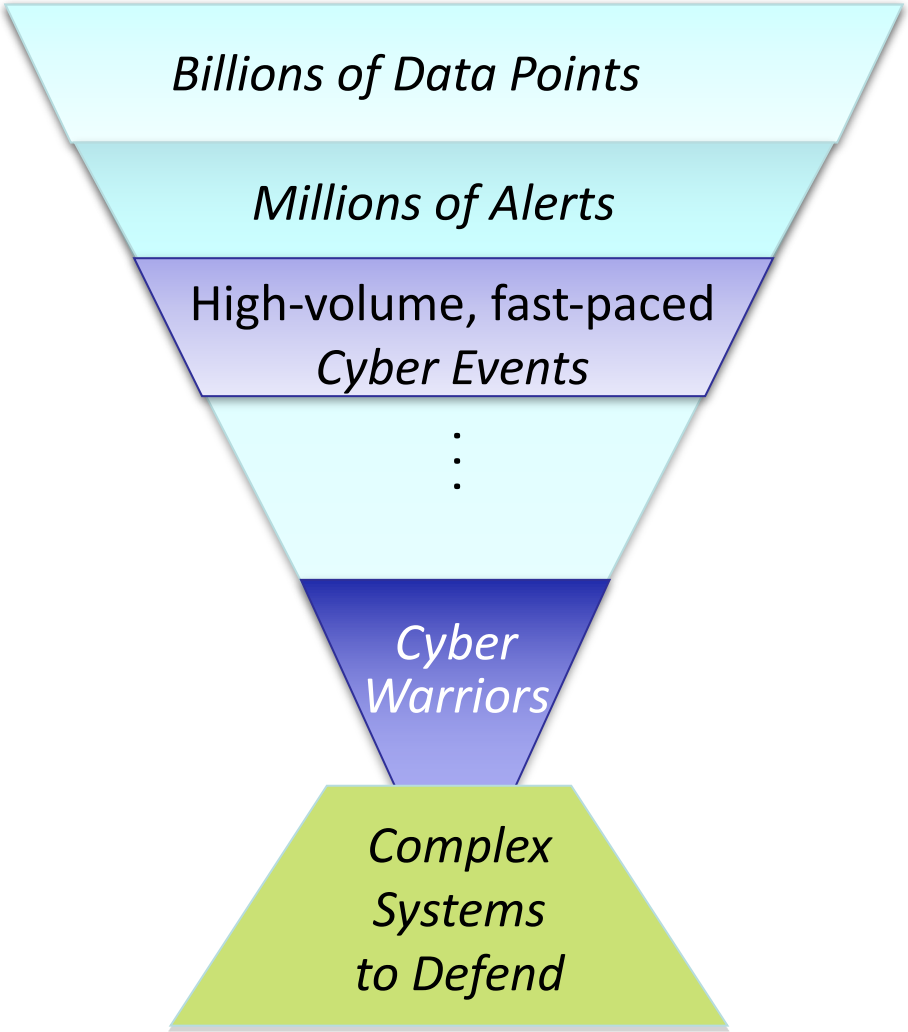
Experimental Evaluation

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

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

Prototype Application

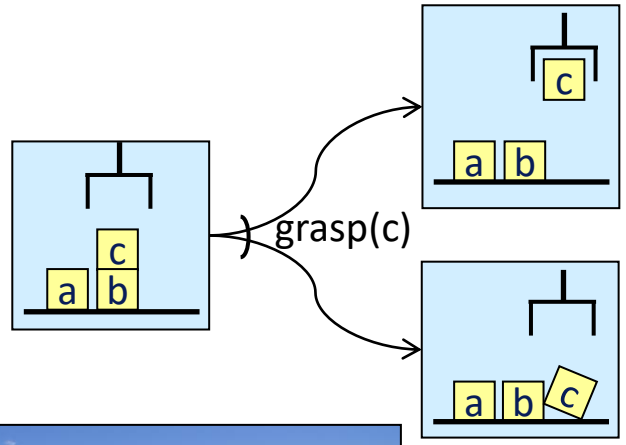
- Software-defined networks
  - Decoupled control and data layers
  - Prone to high-volume, fast-paced online attacks
  - Need automated attack recovery
- Prototype solution using RAE+UPOM
  - Expert writes recovery procedures as refinement methods
- Experimental results
  - Improved efficiency, retry ratio, success ratio, resilience compared to human expert



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

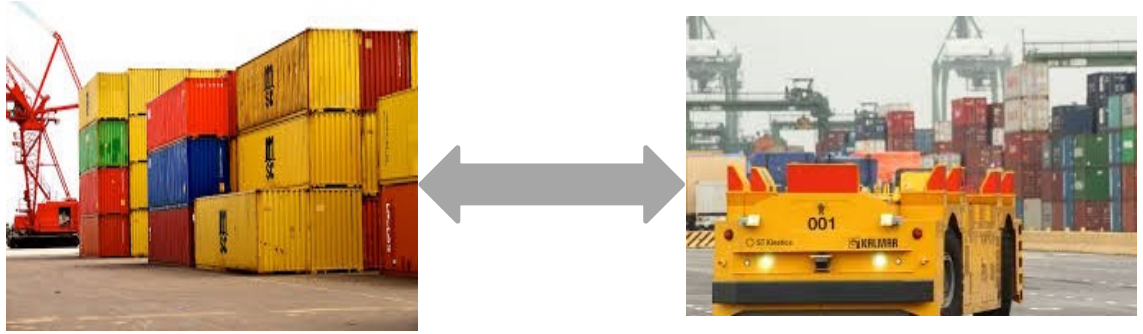
Motivation

- We've assumed action a in state s has just one possible outcome
  - $\gamma(s,a)$

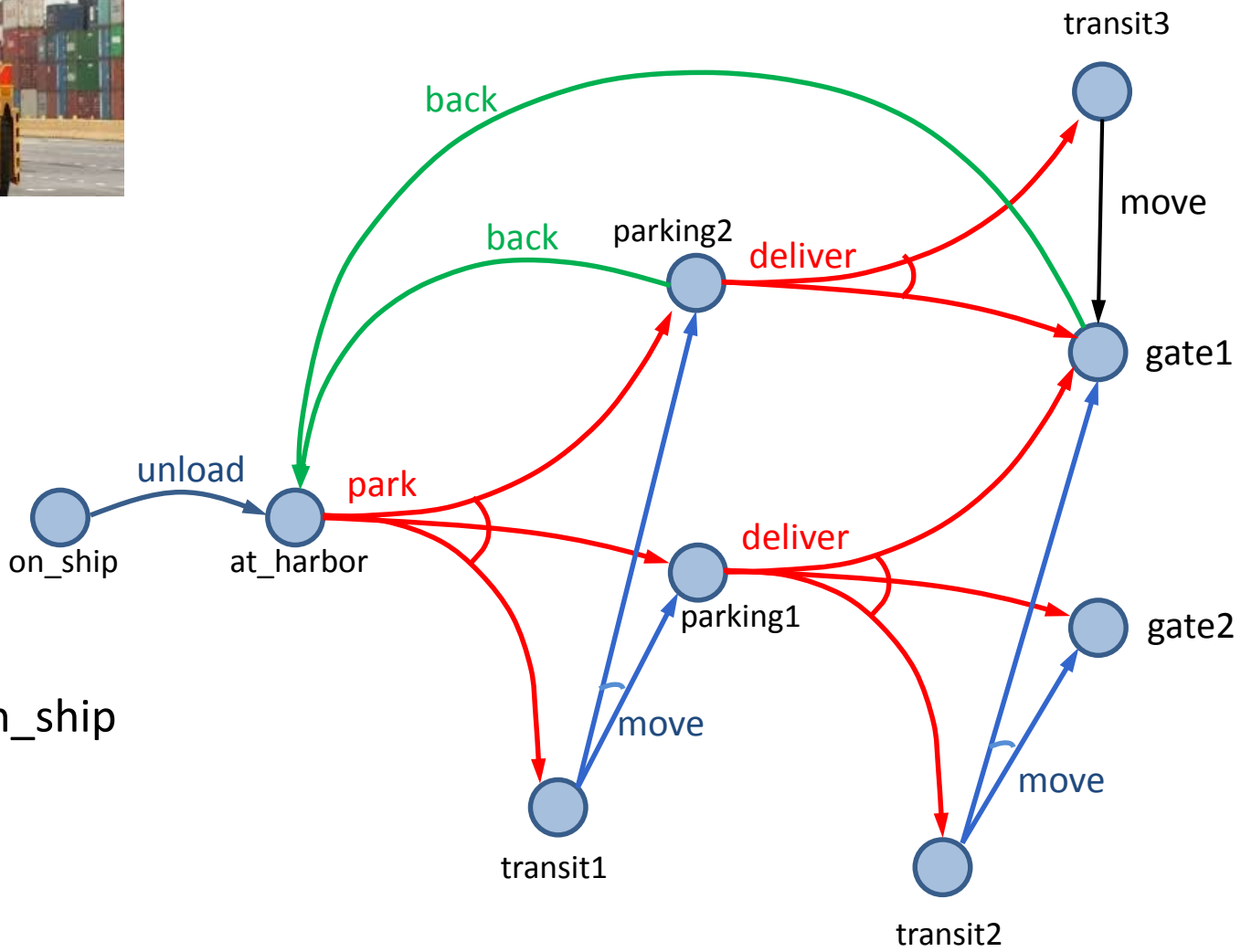


- Often more than one possible outcome
  - Unintended outcomes
  - Exogenous events
  - Inherent uncertainty

Example

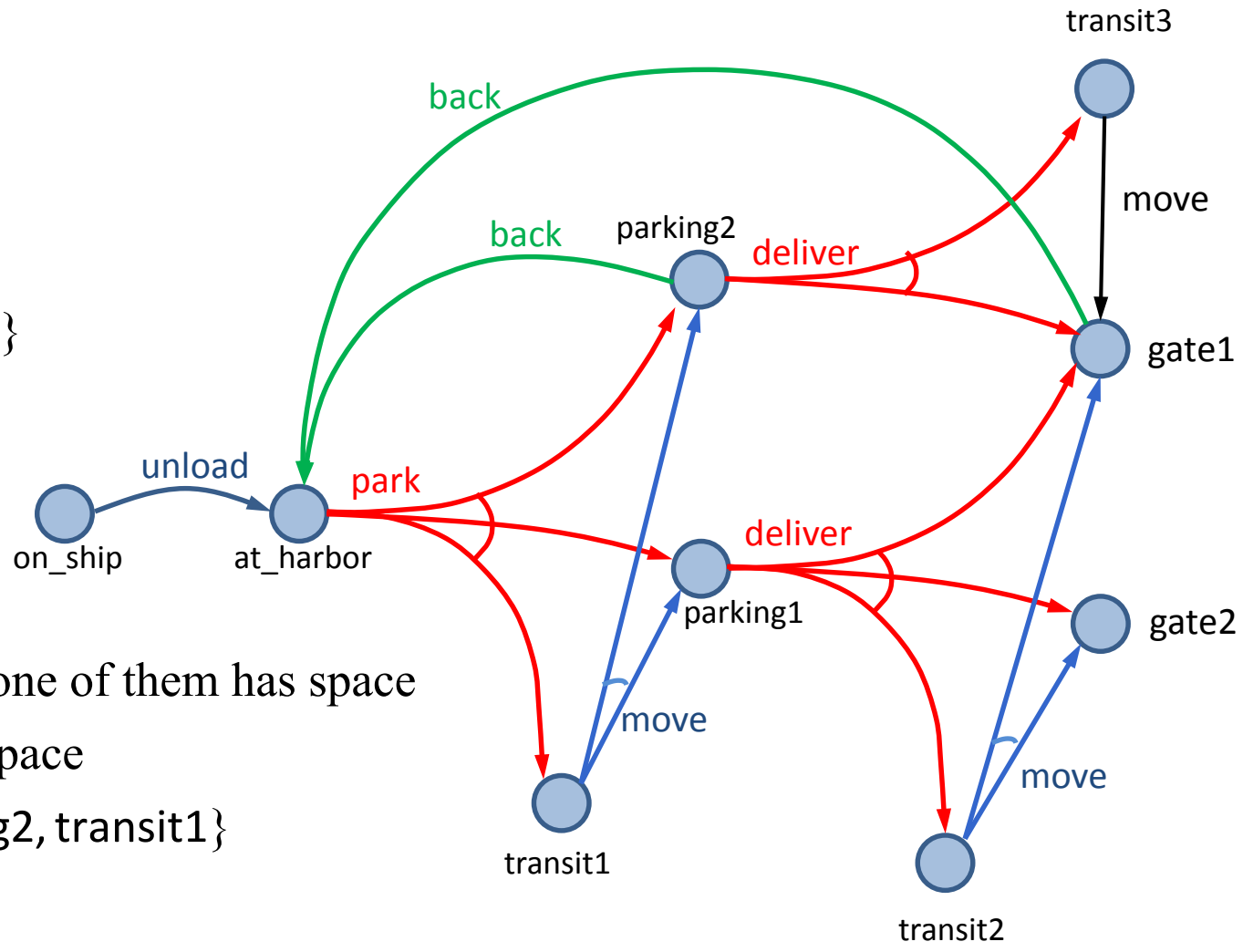


- Very simple harbor management domain
  - Unload a single item from a ship
  - Move it around a harbor
- One state variable: pos(item)
  - Simplified names for states
  - For {pos(item)=on_ship}, just write on_ship



Nondeterministic Planning Domains

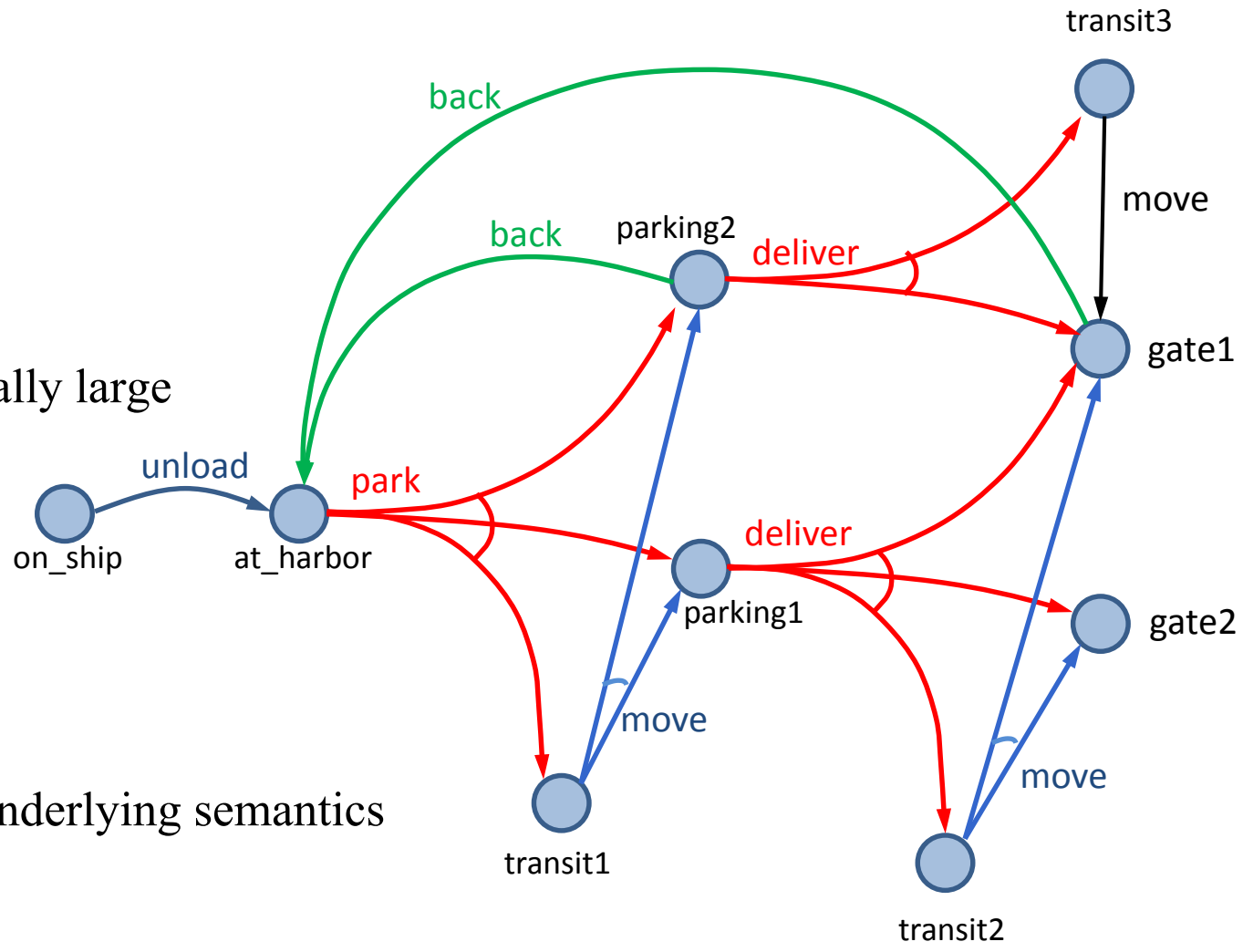
- 3-tuple (S, A, γ)
  - S and A – finite sets of states and actions
  - $\gamma: S \times A \rightarrow 2^S$
- $\gamma(s,a)$ = {all possible "next states" after applying action a in state s}
  - a is applicable in state s iff $\gamma(s,a) \neq \emptyset$
- Applicable(s) = {all actions applicable in s}
  $= \{a \in A \mid \gamma(s,a) \neq \emptyset\}$
- Example:
  - Applicable(at_harbor) = {park}
  - park has three possible outcomes
    - put item in parking1 or parking2 if one of them has space
    - or in transit1 if there's no parking space
  - γ(at_harbor, park) = {parking1, parking2, transit1}



Nondeterministic Planning Domains

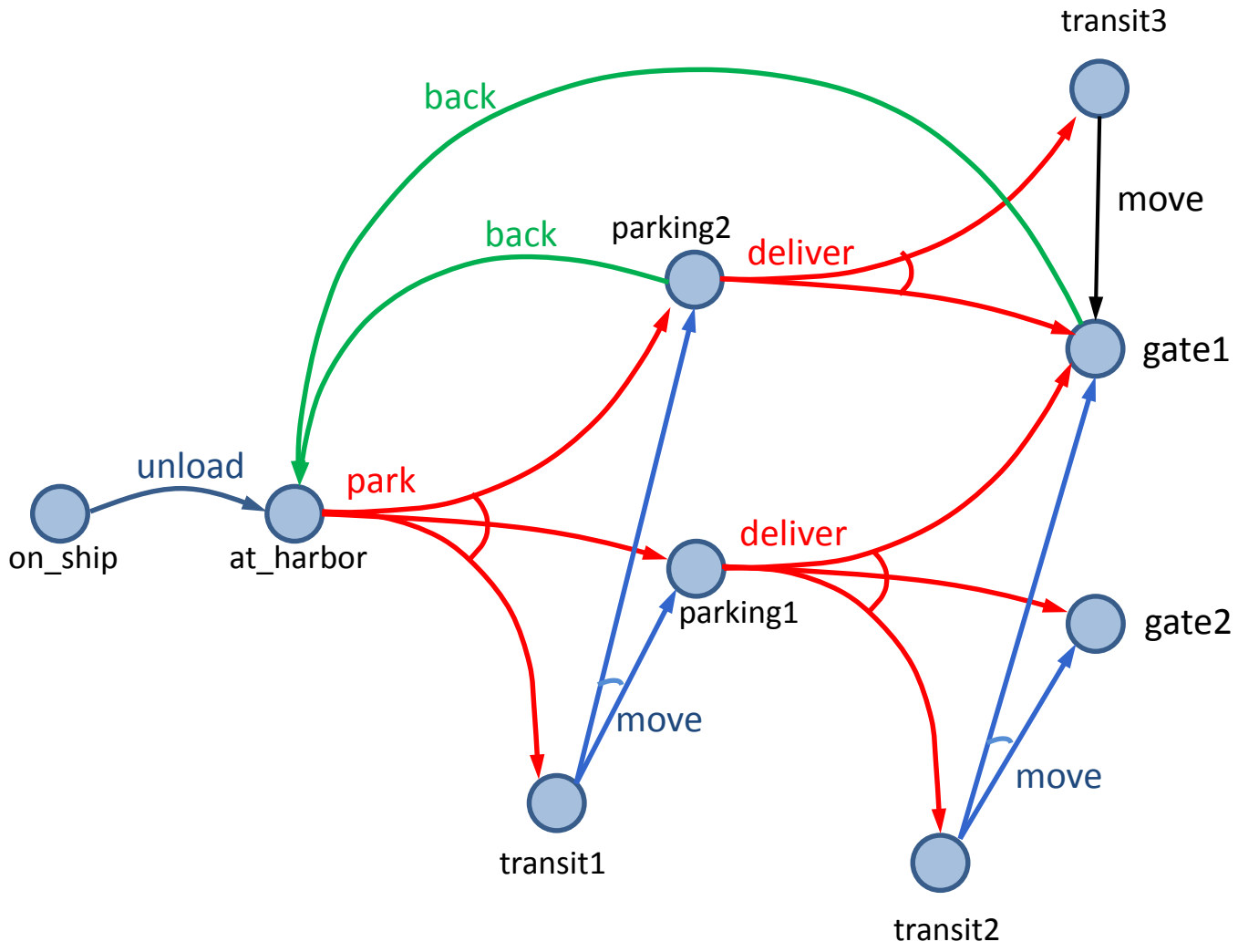
- One possible action representation:
  - like classical, but with n mutually exclusive "effects" lists
- e.g., park:

  pre: pos(item) = at_harbor
  eff_1: pos(item) ← parking1
  eff_2: pos(item) ← parking2
  eff_3: pos(item) ← transit1
- Problem:
  - number of effects lists may be combinatorially large
  - Suppose a can cause any possible combination of effects $e_1, e_2, \ldots, e_k$
  - Need eff_1 , eff_2 , …, eff_{2^k}
    - One for for each combination
  - Section 5.4: a way to alleviate this
- For now, ignore most of that, just look at the underlying semantics
  - states, actions ⇔ nodes, edges in a graph



Nondeterministic Planning Domains

- For deterministic planning problems, search space was a graph
- Now it's an AND/OR graph
  - *OR branch*:
    - several applicable actions, which one to choose?
  - *AND branch*:
    - multiple possible outcomes, must handle all of them
- Analogy to PSP in Chapter 2
  - *OR* branch ⇔ action selection
  - *AND* branch ⇔ flaw selection



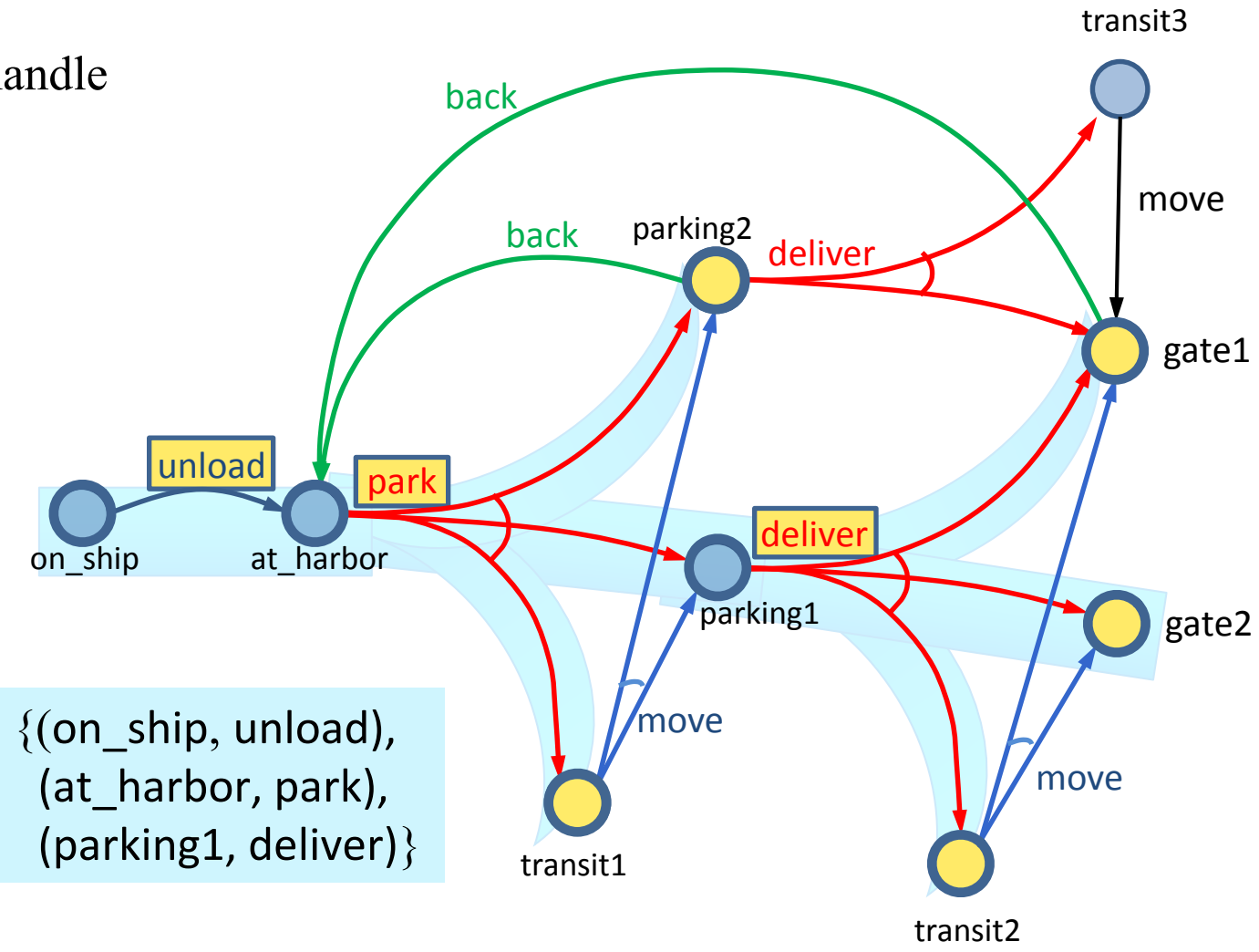
~~Plans~~ Policies

- Need something more general than a sequence of actions ⟨unload, park, deliver⟩
  - After park, what do we do next?
    - multiple possible outcomes, must handle all of them
- *Policy*: a *partial* function $\pi : S \nrightarrow A$
  - i.e., $\mathrm{Dom}(\pi) \subseteq S$
  - For every $s \in \mathrm{Dom}(\pi)$, require $\pi(s) \in \mathrm{Applicable}(s)$
- Meaning: perform $\pi(s)$ whenever we're in state s
- Two equivalent notations:

π_1(on_ship) = unload,
π_1 (at_harbor) = park,
π_1 (parking1) = deliver

$\pi_1 = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver)$\}$

- That's just the notation
  - implementation could be quite different



Definitions Over Policies

- *Transitive closure*:
  - $\hat{\gamma}(s,\pi) = \{\text{all states reachable from } s \text{ using } \pi\}$
  - $\hat{\gamma}(s,\pi) = S_0 \cup S_1 \cup S_2 \cup \ldots$

    $S_0 = \{s\}$

    $S_1 = S_0 \cup \{\gamma(s_0,\pi(s_0)) \mid s_0 \in S_0\}$

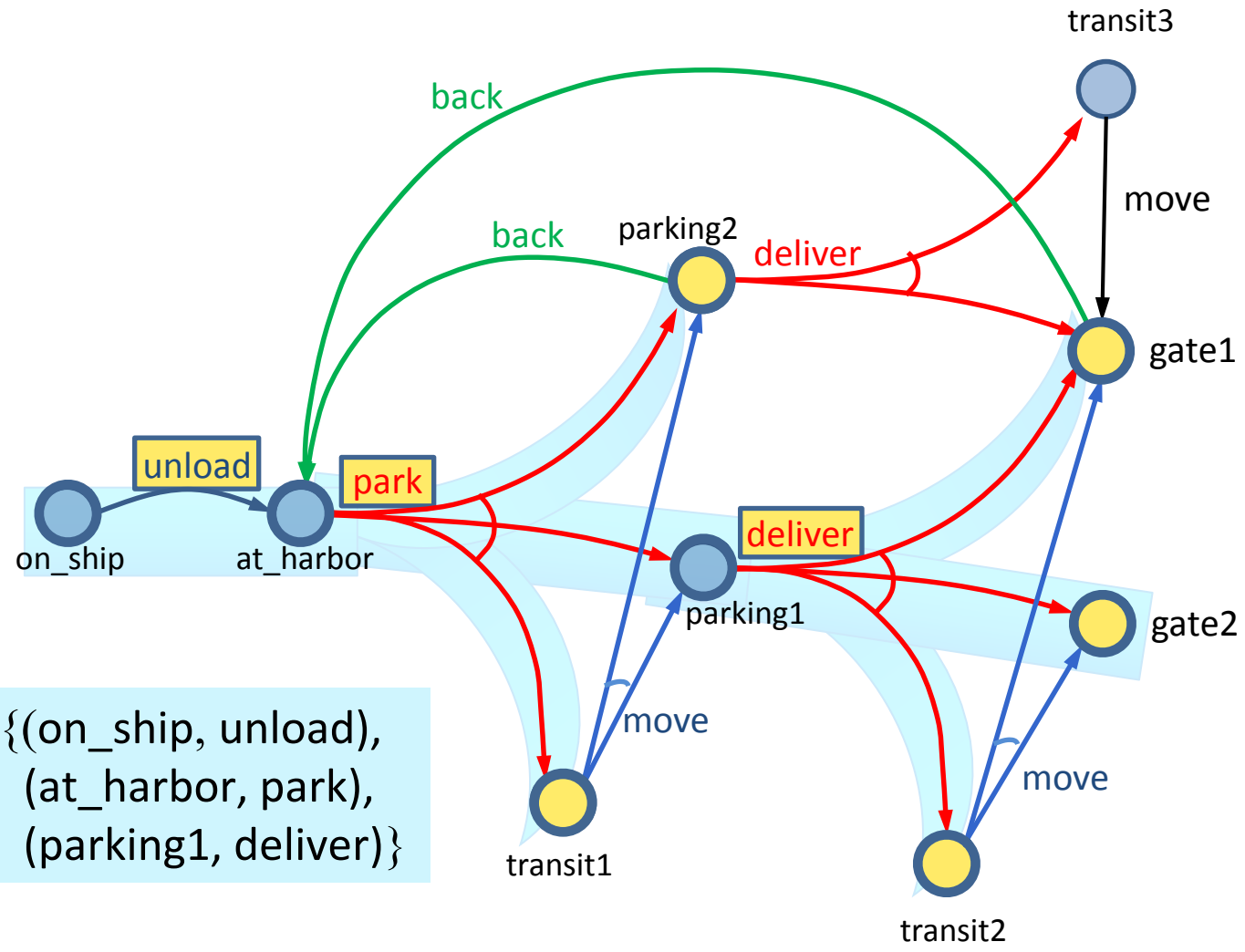
    $S_2 = S_1 \cup \{\gamma(s_1,\pi(s_1)) \mid s_1 \in S_1\}$

    …

- *Reachability graph*: $\text{Graph}(s,\pi) = (V,E)$
  - $V = \hat{\gamma}(s,\pi)$
  - $E = \{(s_1,s_2) \mid s_1 \in V,\ s_2 \in \gamma(s_1,\pi(s_1))\}$

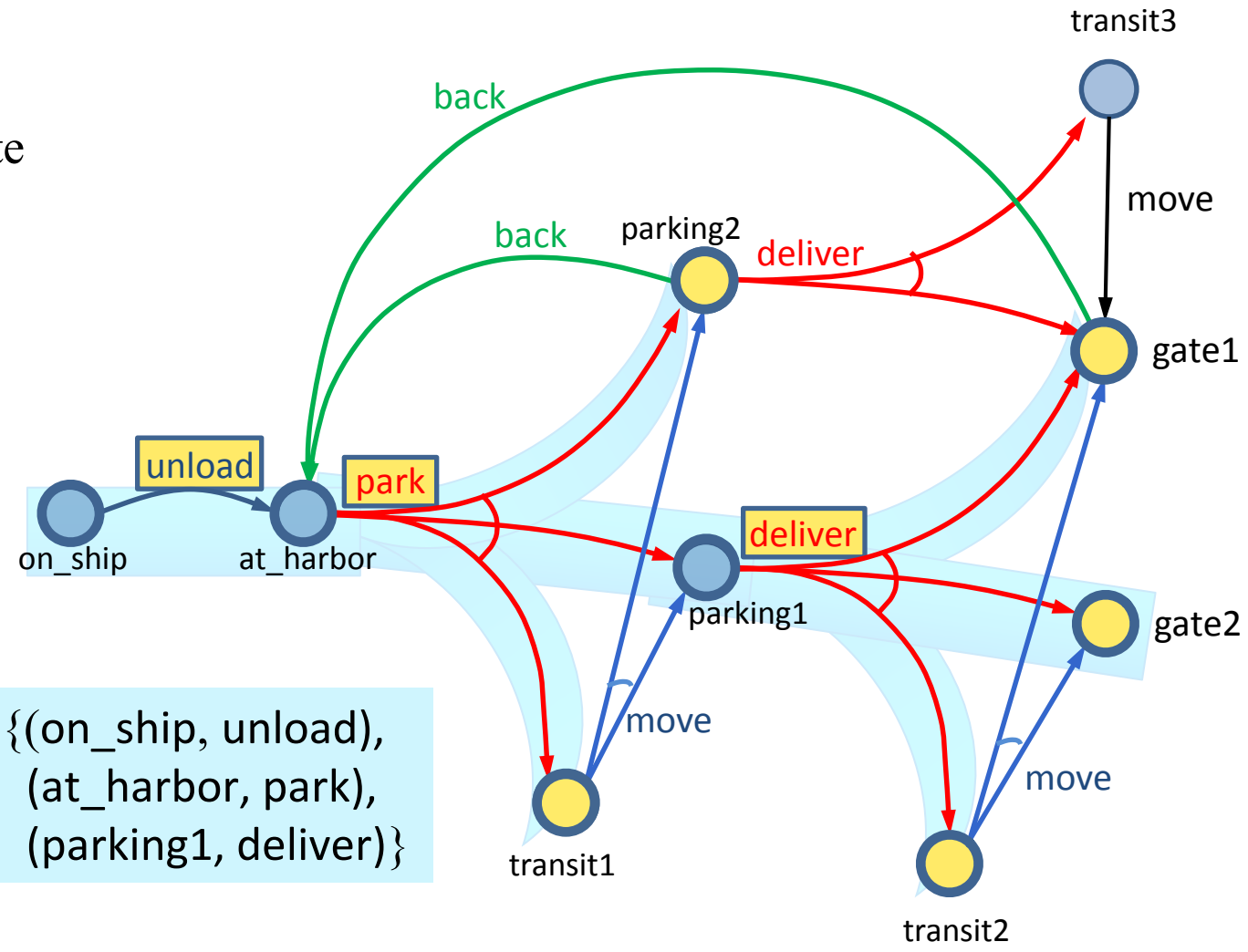
- $leaves(s,\pi) = \hat{\gamma}(s,\pi) \setminus \text{Dom}(\pi)$
  - may be empty

$\pi_1 = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver)$\}$



Performing a Policy

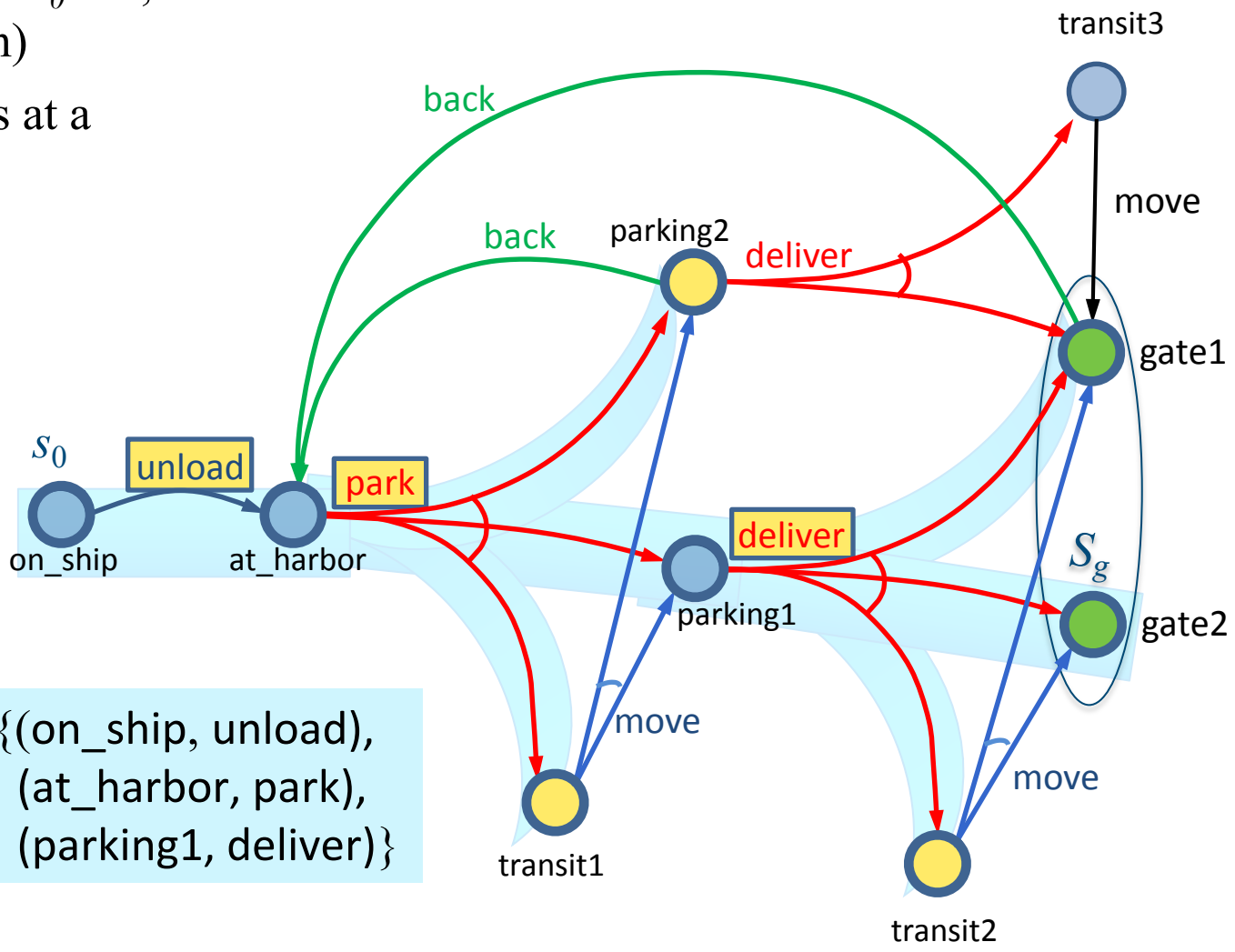
- PerformPolicy(π)
  - $s \leftarrow$ observe current state
  - while $s \in \mathrm{Dom}(\pi)$ do
    - perform action $\pi(s)$
    - $s \leftarrow$ observe current state



$\pi_1 = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver)$\}$

Planning Problems and Solutions

- Planning problem $P = (\Sigma, s_0, S_g)$
  - planning domain $\Sigma = (S, A, \gamma)$, initial state $s_0 \in S$, set of goal states $S_g \subseteq S$ (shown in green)
- π is a *solution* if at least one execution ends at a goal
  - $leaves(s,\pi) \cap S_g \neq \emptyset$
- A solution π is *safe* if $\forall s \in \hat{\gamma}(s_0,\pi)$, $leaves(s,\pi) \cap S_g \neq \emptyset$
  - at every state in $\hat{\gamma}(s_0,\pi)$, at least one of the execution paths from s using π stops at a goal state.
- Otherwise, *unsafe* solution
  - Is π_1 safe or unsafe?

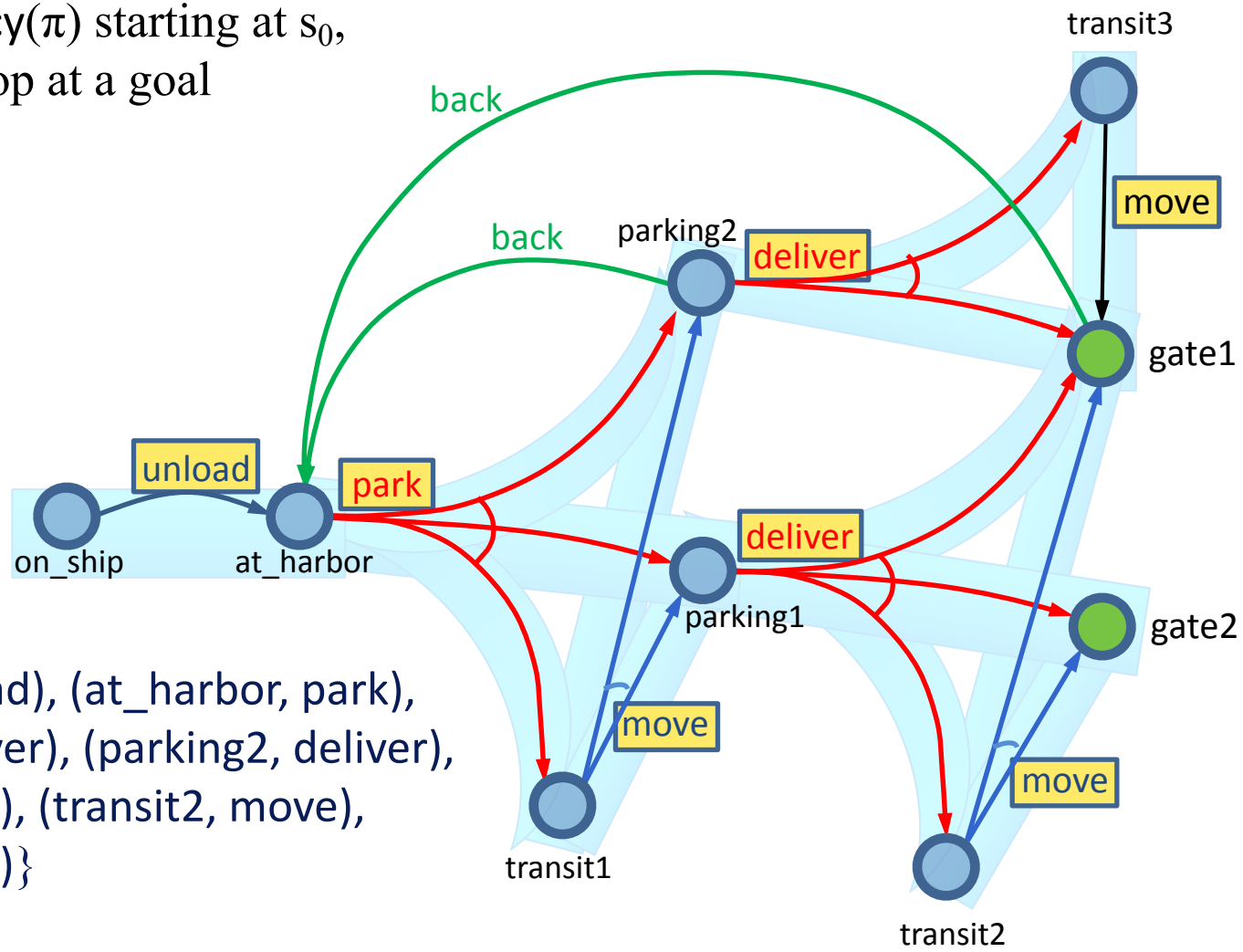


π_1 = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}

Safe Solutions

- PerformPolicy(π)
  - $s \leftarrow$ observe current state
  - while $s \in \mathrm{Dom}(\pi)$ do
    - perform action $\pi(s)$
    - $s \leftarrow$ observe current state

- *Acyclic* safe solution
  - Graph(s_0,π) is acyclic, and *leaves*(s,π) $\subseteq S_g$
- If we run PerformPolicy(π) starting at s_0, we're guaranteed to stop at a goal

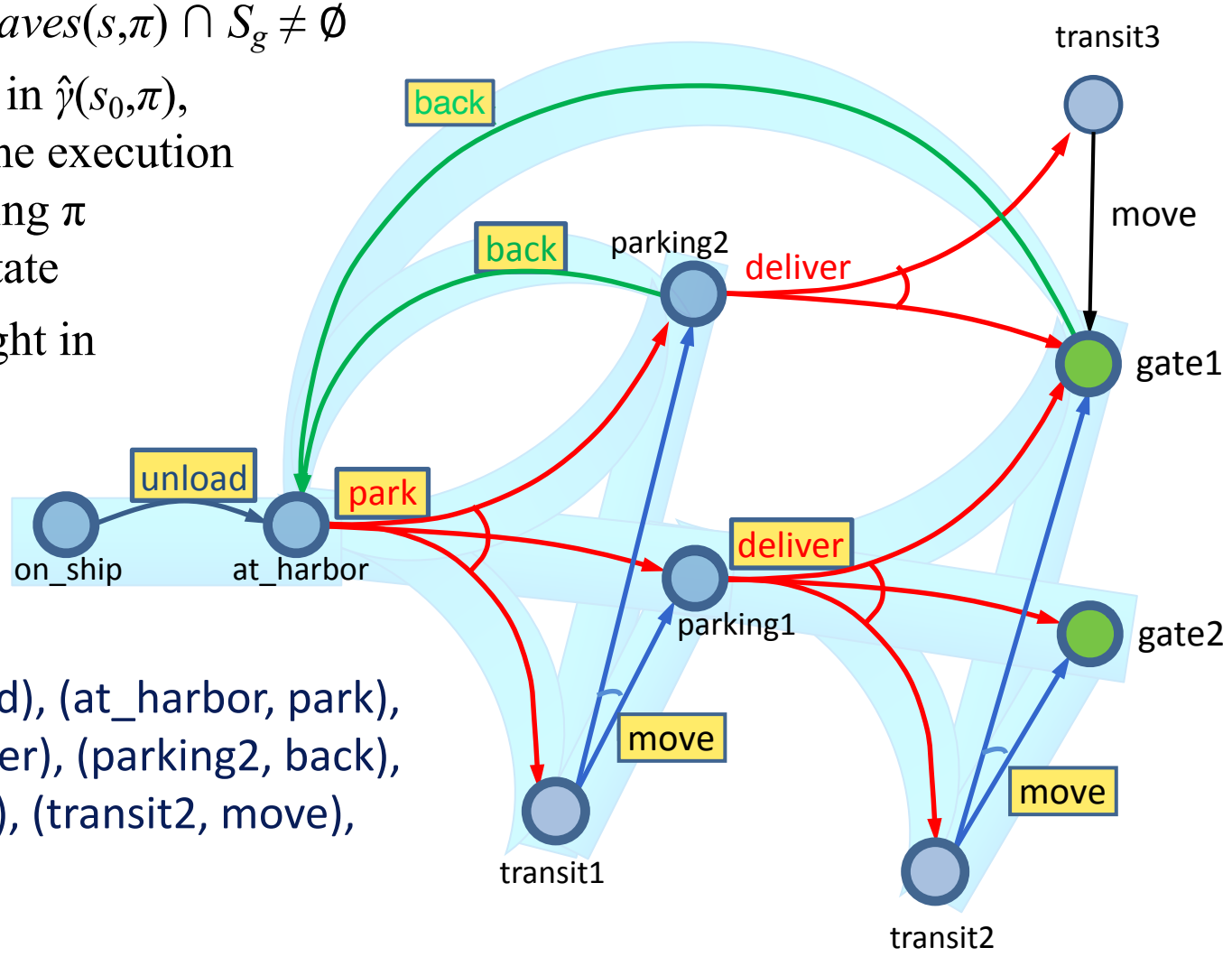


π_2 = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, deliver), (transit1, move), (transit2, move), (transit3, move)}

Safe Solutions

- PerformPolicy(π)
  - $s \leftarrow$ observe current state
  - while $s \in \mathrm{Dom}(\pi)$ do
    - perform action $\pi(s)$
    - $s \leftarrow$ observe current state

- *Cyclic* safe solution
  - Graph(s_0, π) is cyclic, and $leaves(s,\pi) \subseteq S_g$, and $\forall s \in \hat{\gamma}(s_0,\pi)$, $leaves(s,\pi) \cap S_g \neq \emptyset$
    - At every state s in $\hat{\gamma}(s_0,\pi)$, at least one of the execution paths from s using π ends at a goal state
  - Will never get caught in a dead end



π_3 = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, back), (transit1, move), (transit2, move), (gate1, back)}

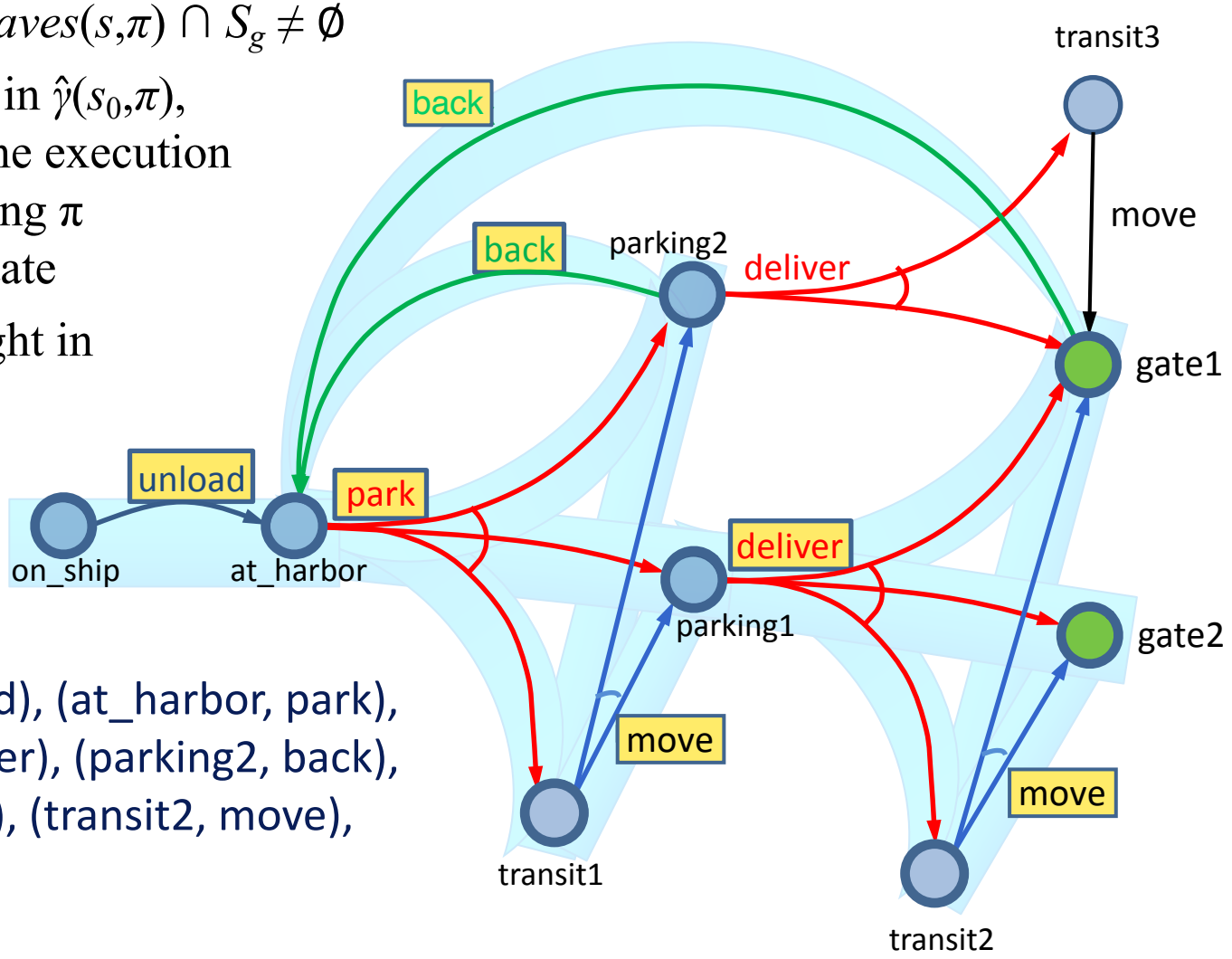
Poll: Let π be a cyclic safe solution. Suppose we run PerformPolicy(π) starting at s_0.

1. Are there situations where we can be sure π will reach a goal?
2. Are there situations where we can't be sure π will reach a goal?

Safe Solutions

- PerformPolicy(π)
  $s \leftarrow$ observe current state
  while $s \in$ Dom(π) do
    perform action $\pi(s)$
    $s \leftarrow$ observe current state

- *Cyclic* safe solution
  - Graph(s_0, π) is cyclic, and *leaves*(s,π) $\subseteq S_g$, and $\forall s \in \hat{\gamma}(s_0,\pi)$, *leaves*($s$,$\pi$) $\cap S_g \neq \emptyset$
    - At every state s in $\hat{\gamma}(s_0,\pi)$, at least one of the execution paths from s using π ends at a goal state
  - Will never get caught in a dead end
  - Every "fair" execution will reach a goal

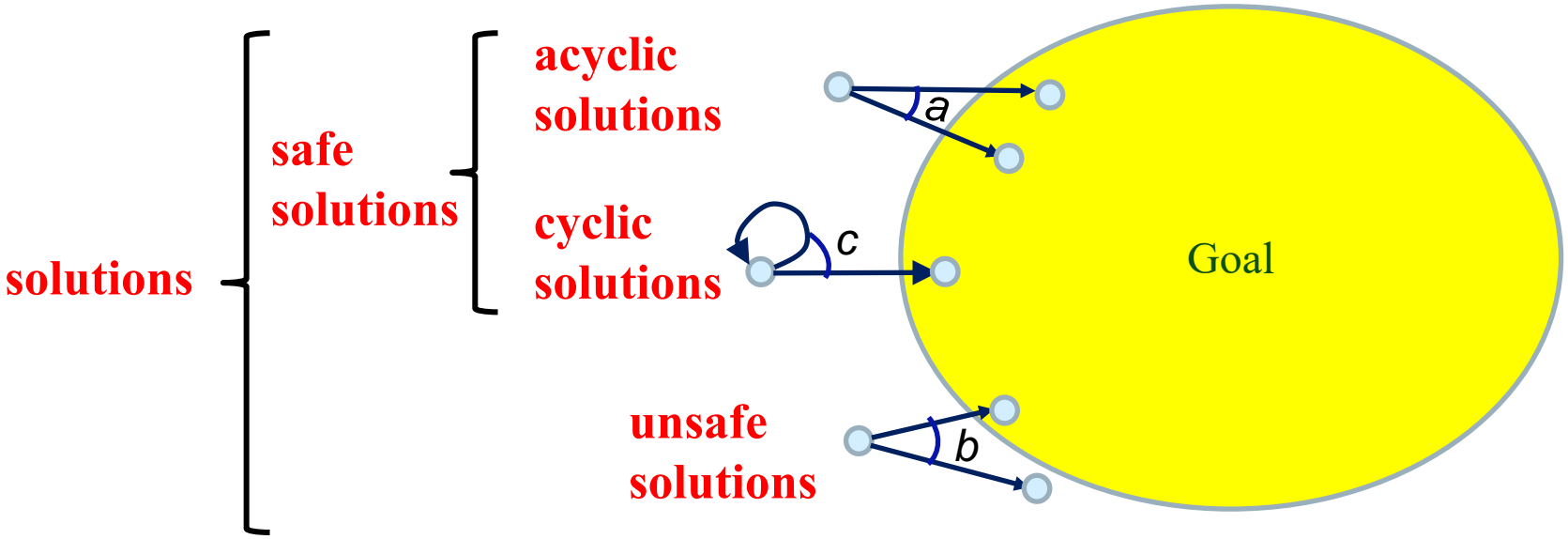


π_3 = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, back), (transit1, move), (transit2, move), (gate1, back)}

Poll:

1. Can you think of a real-world situation in which all executions are "fair"?
2. Can you think of a real-world situation in which there are "unfair" executions?

Kinds of Solutions



Finding (Unsafe) Solutions

For comparison:

Forward-search (Σ, s_0, g)
  $s \leftarrow s_0$; $\pi \leftarrow \langle\rangle$
  loop
    if s satisfies g then return π
    $A' \leftarrow \{a \in A \mid a$ is applicable in $s\}$
    if $A' = \emptyset$ then return failure
    nondeterministically choose $a \in A'$
    $s \leftarrow \gamma(s,a)$; $\pi \leftarrow \pi.a$

Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$; $s \leftarrow s_0$; $Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow$ Applicable(s)
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
(*) nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a$; $Visited \leftarrow Visited \cup \{s'\}$; $s \leftarrow s'$

Decide which state to plan for

Cycle-checking

Poll: which should (*) be?
A. nondeterministically choose
B. arbitrarily choose

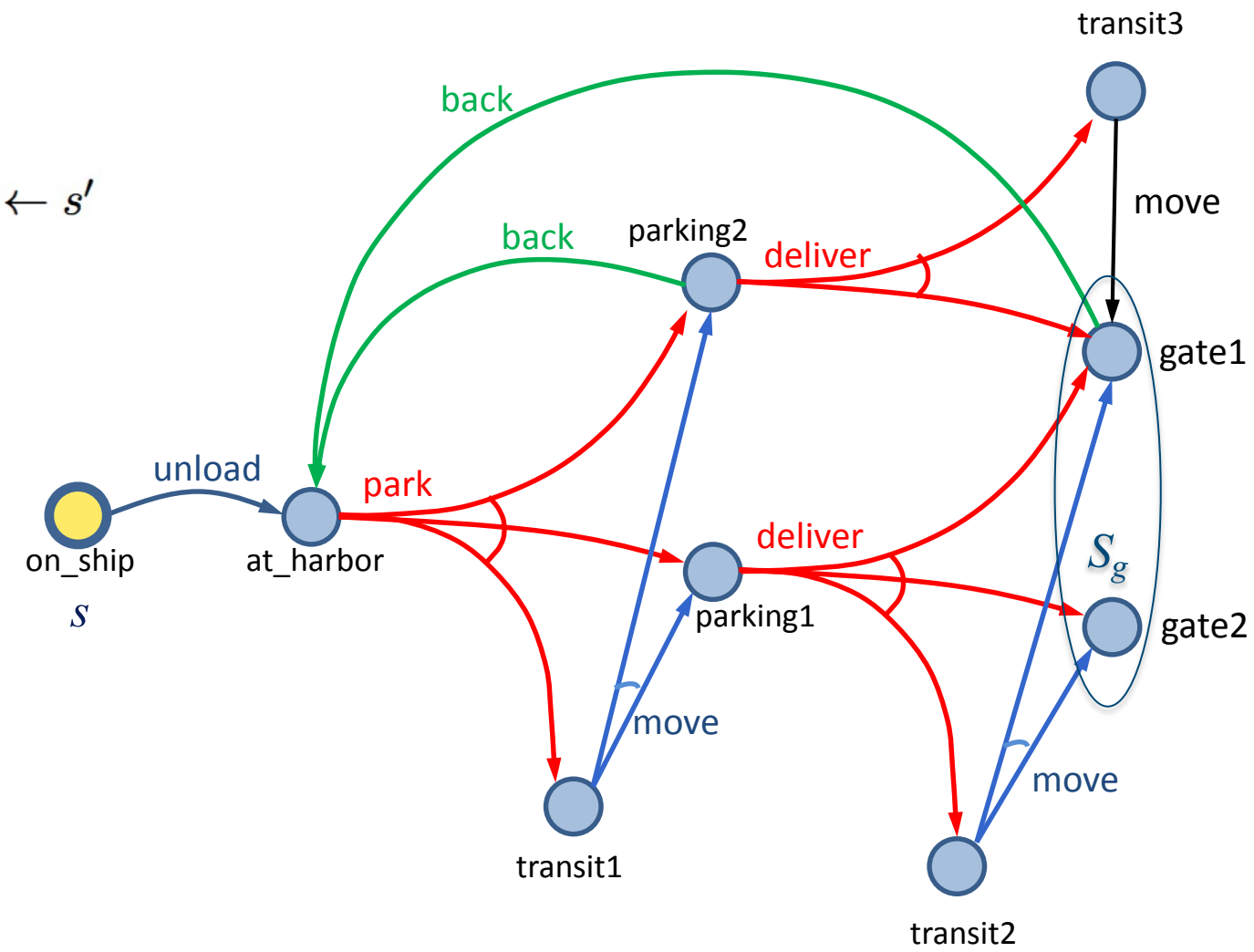
Example

Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing;\ \ s \leftarrow s_0;\ \ Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow$ Applicable(s)
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a;\ \ Visited \leftarrow Visited \cup \{s'\};\ s \leftarrow s'$

s = on_ship

$\pi = \{\}$

$Visited = \{$on_ship$\}$



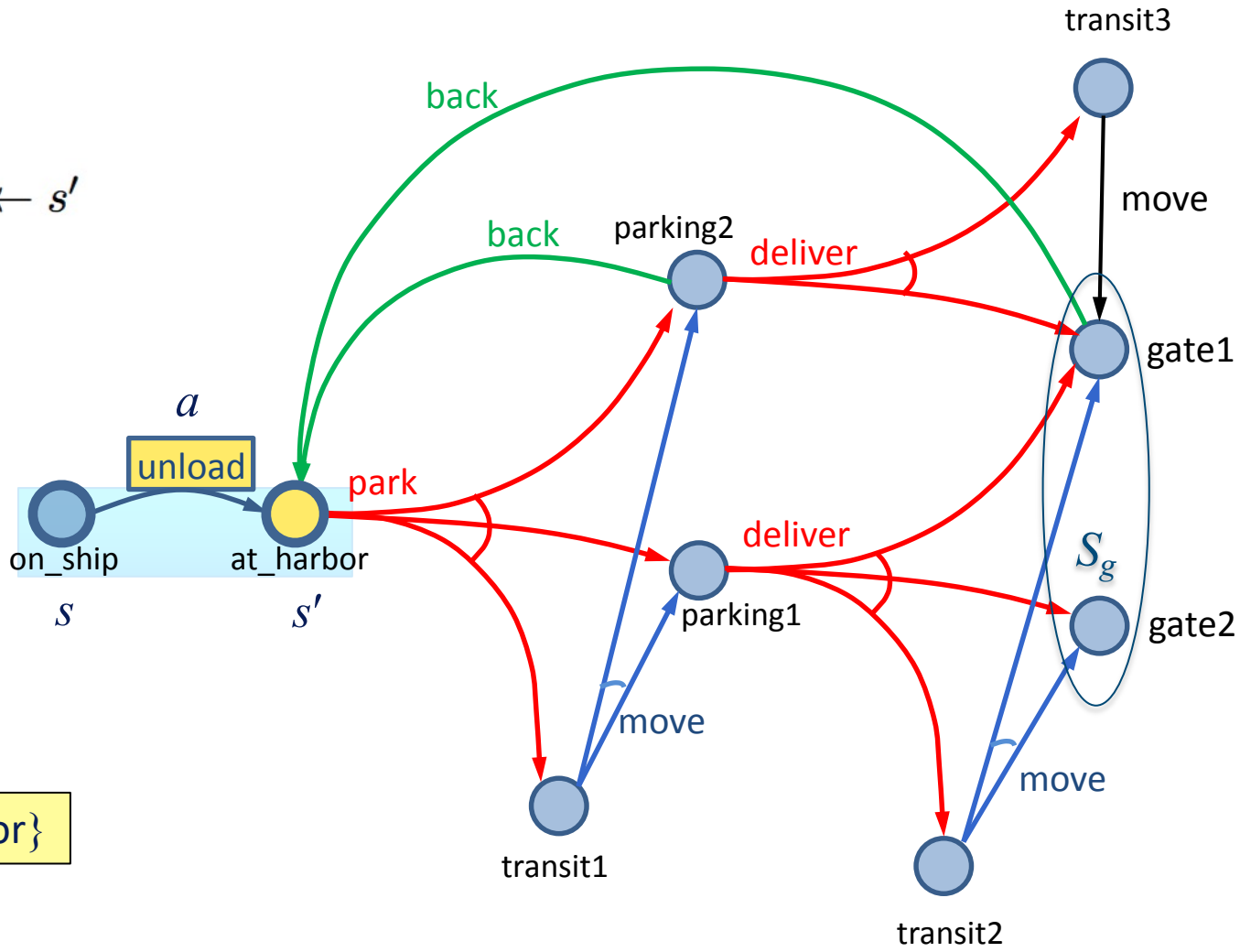
Example

Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing;\ \ s \leftarrow s_0;\ \ Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow \text{Applicable}(s)$
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a;\ \ Visited \leftarrow Visited \cup \{s'\};\ s \leftarrow s'$

s = on_ship
a = unload
$\gamma(s,a)$ = {at_harbor}
$s \leftarrow s'$ = at_harbor

π = {(on_ship, unload)}

$Visited$ = {on_ship, at_harbor}



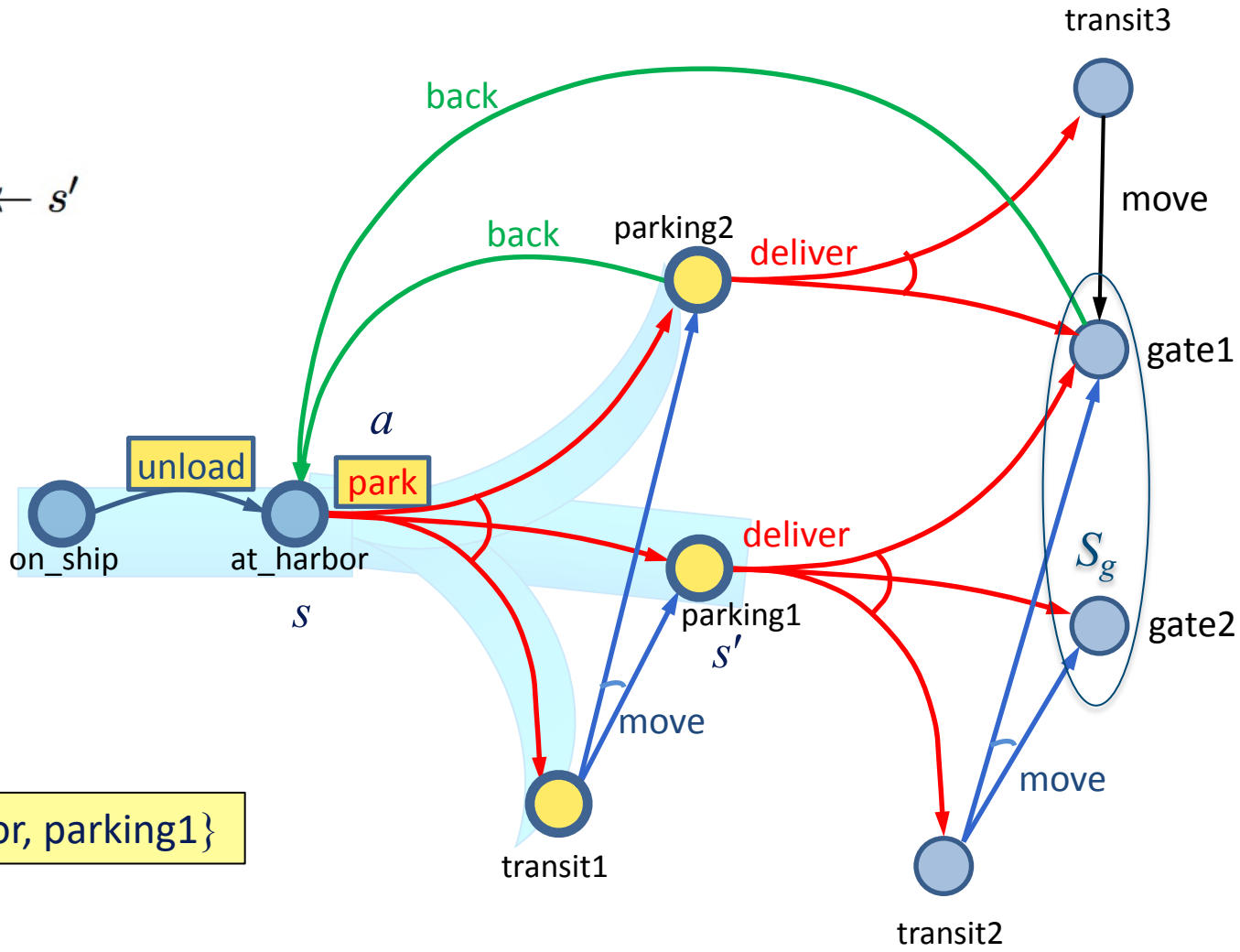
Example

Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing;\ \ s \leftarrow s_0;\ \ Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow$ Applicable(s)
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a;\ \ Visited \leftarrow Visited \cup \{s'\};\ s \leftarrow s'$

s = at_harbor
a = park
$\gamma(s,a)$ = {parking1, parking2, transit1}
$s \leftarrow s'$ = parking1

π = {(on_ship, unload), (at_harbor, park)}

$Visited$ = {on_ship, at_harbor, parking1}



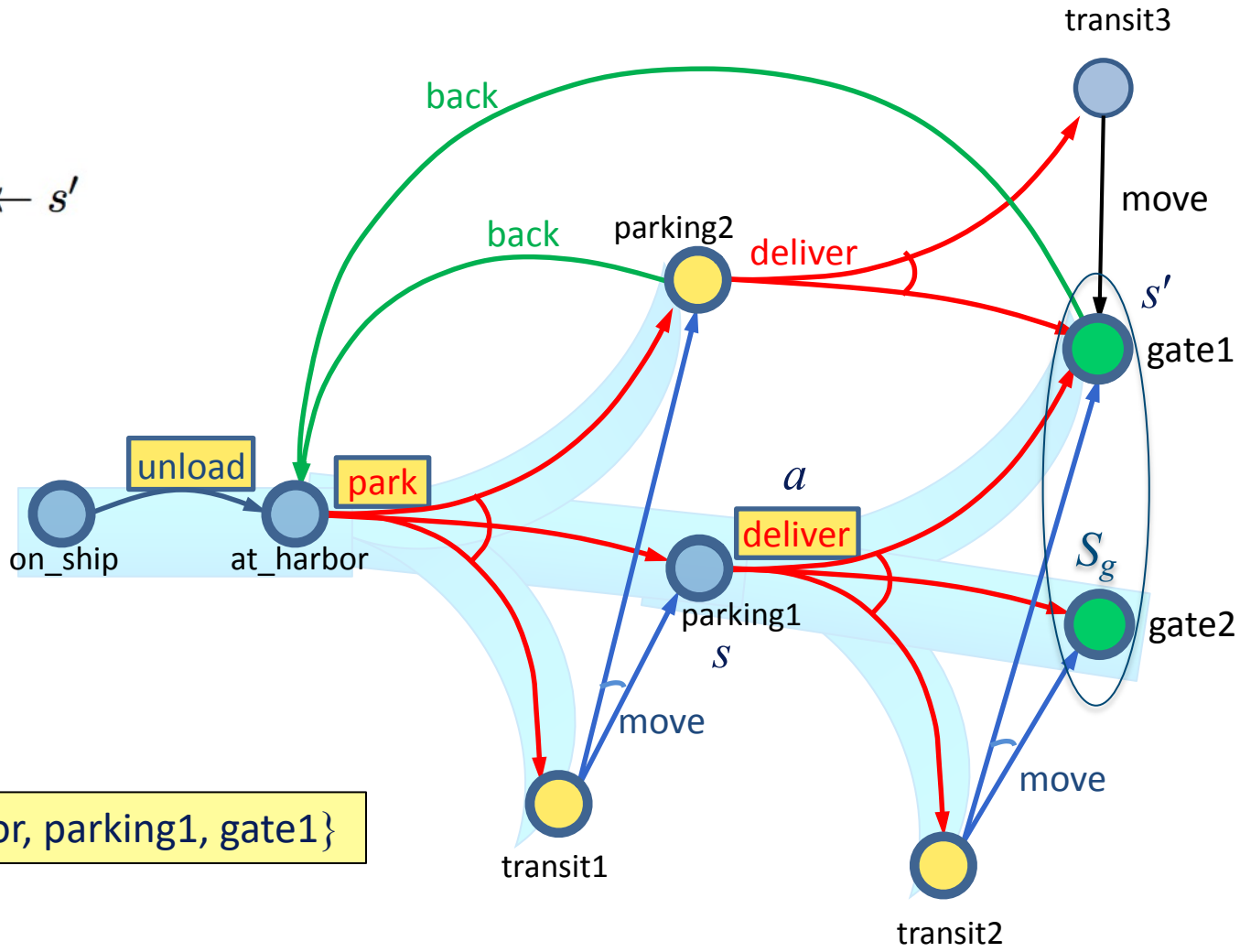
Example

Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing;\ \ s \leftarrow s_0;\ \ Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow \text{Applicable}(s)$
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a;\ \ Visited \leftarrow Visited \cup \{s'\};\ s \leftarrow s'$

s = parking1
a = deliver
$\gamma(s,a)$ = {gate1, gate2, transit2}
$s \leftarrow s'$ = gate1

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}

$Visited$ = {on_ship, at_harbor, parking1, gate1}



Example

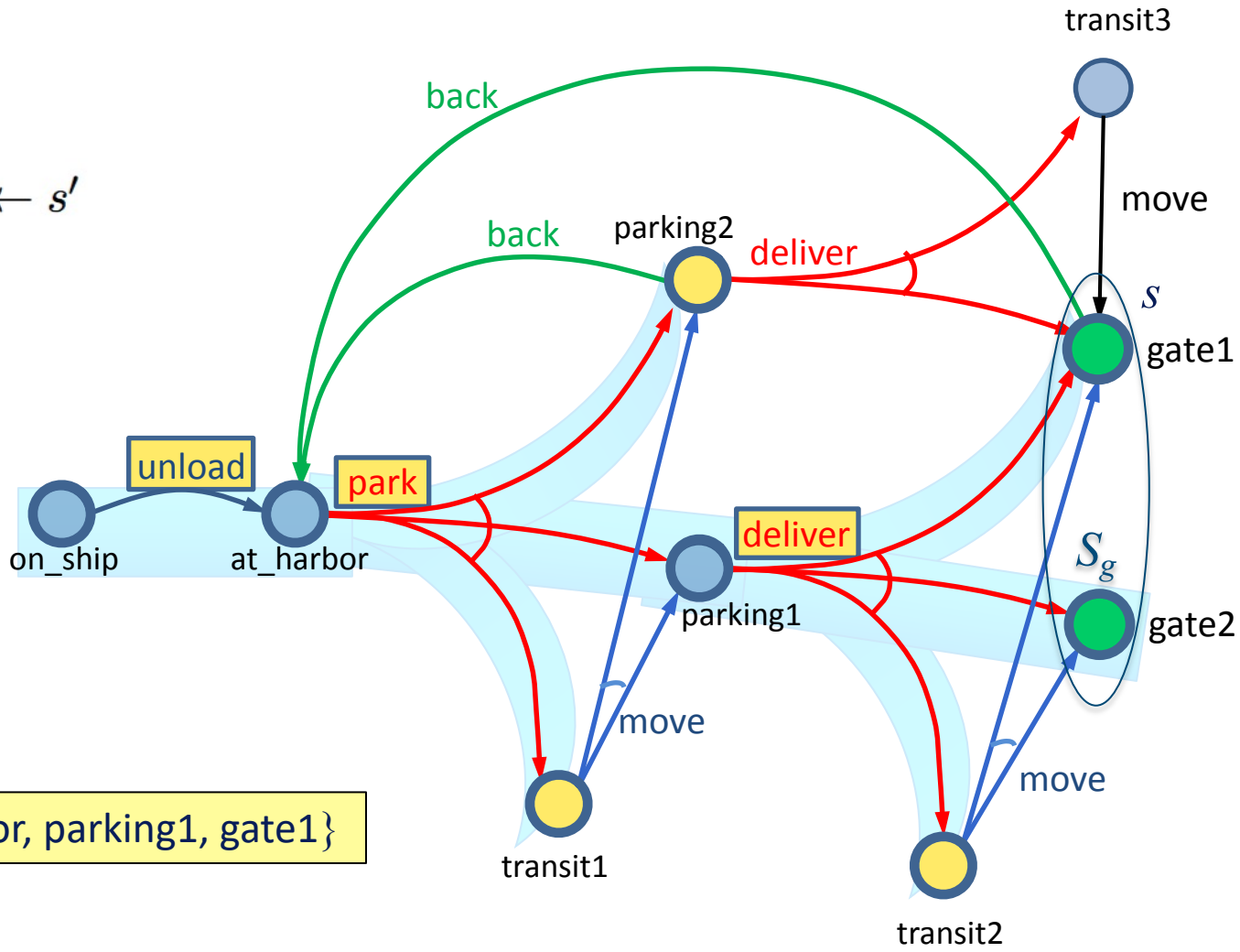
Find-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing;\ \ s \leftarrow s_0;\ \ Visited \leftarrow \{s_0\}$
  loop
    if $s \in S_g$ then return π
    $A' \leftarrow$ Applicable(s)
    if $A' = \varnothing$ then return failure
    nondeterministically choose $a \in A'$
    nondeterministically choose $s' \in \gamma(s, a)$
    if $s' \in Visited$ then return failure
    $\pi(s) \leftarrow a;\ \ Visited \leftarrow Visited \cup \{s'\};\ s \leftarrow s'$

s = gate1

gate1 is a goal,
so return π

π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver)}

$Visited$ = {on_ship, at_harbor, parking1, gate1}



Find-Acyclic-Solution

```
Find-Acyclic-Solution (Σ, s_0, S_g)
    π ← ∅
    Frontier ← {s_0}
    for every s ∈ Frontier \ S_g do
        Frontier ← Frontier \ {s}
        if Applicable(s) = ∅ then return failure
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s, a)
        Frontier ← Frontier ∪ (γ(s, a) \ Dom(π))
        if has-loops(π, a, Frontier) then return failure
    return π
```

Keep track of unexpanded states, as in A*

Add all states s such that $\pi(s)$ isn't already defined

Check for cycles:

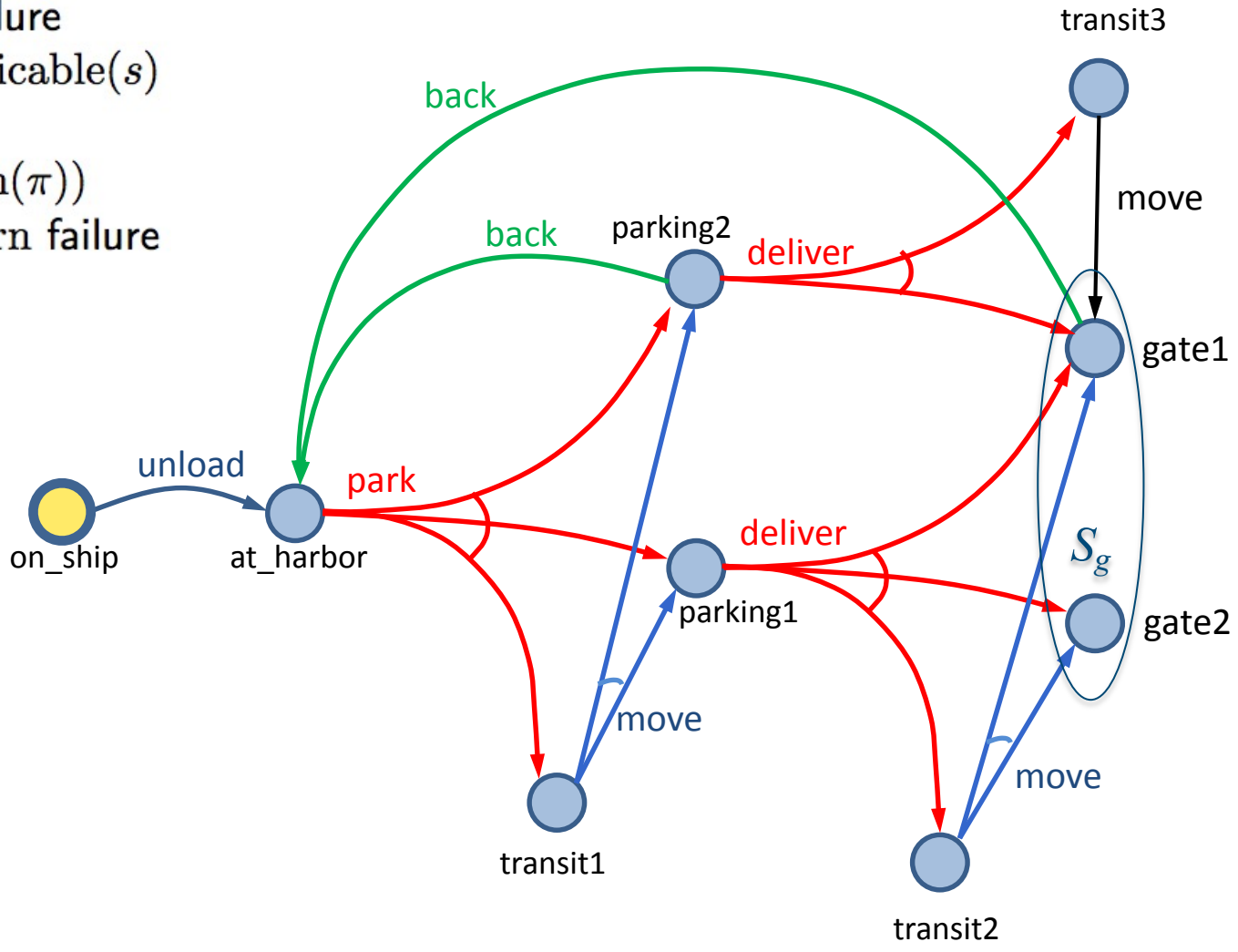
- Does $\gamma(s,a)$ include a state s' that is a π-ancestor of s?
  - for each $s' \in \gamma(s,a) \cap \mathrm{Dom}(\pi)$, is $s \in \hat{\gamma}(s',\pi)$?

Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{\text{on_ship}\}$

$\pi = \{\}$



Example

Find-Acyclic-Solution (Σ, s_0, S_g)

$\pi \leftarrow \varnothing$

$Frontier \leftarrow \{s_0\}$

for every $s \in Frontier \setminus S_g$ do

  $Frontier \leftarrow Frontier \setminus \{s\}$

  if $\text{Applicable}(s) = \varnothing$ then return failure

  nondeterministically choose $a \in \text{Applicable}(s)$

  $\pi \leftarrow \pi \cup (s, a)$

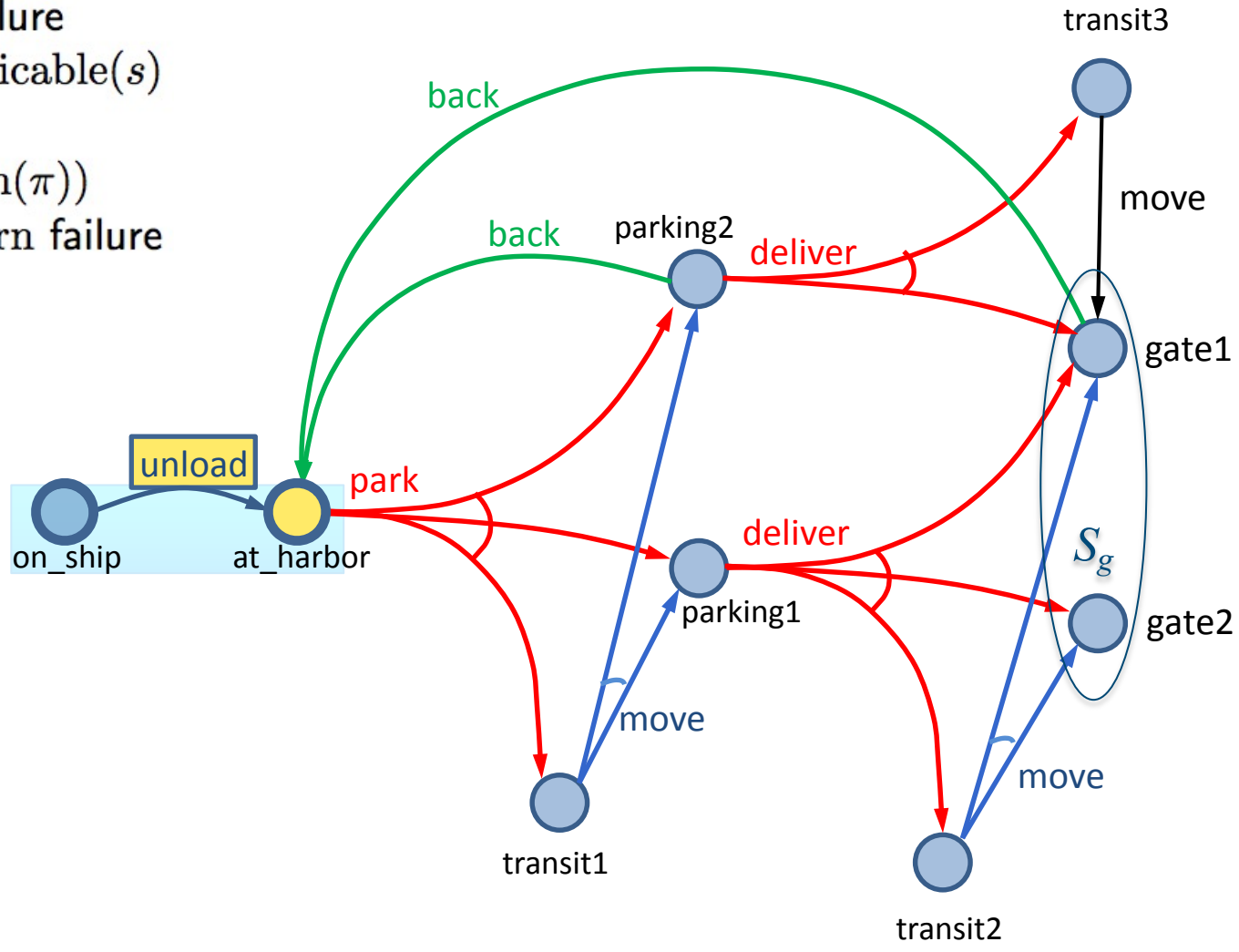
  $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$

  if has-loops$(\pi, a, Frontier)$ then return failure

return π

$Frontier = \{$at_harbor$\}$

$\pi = \{$(on_ship, unload)$\}$

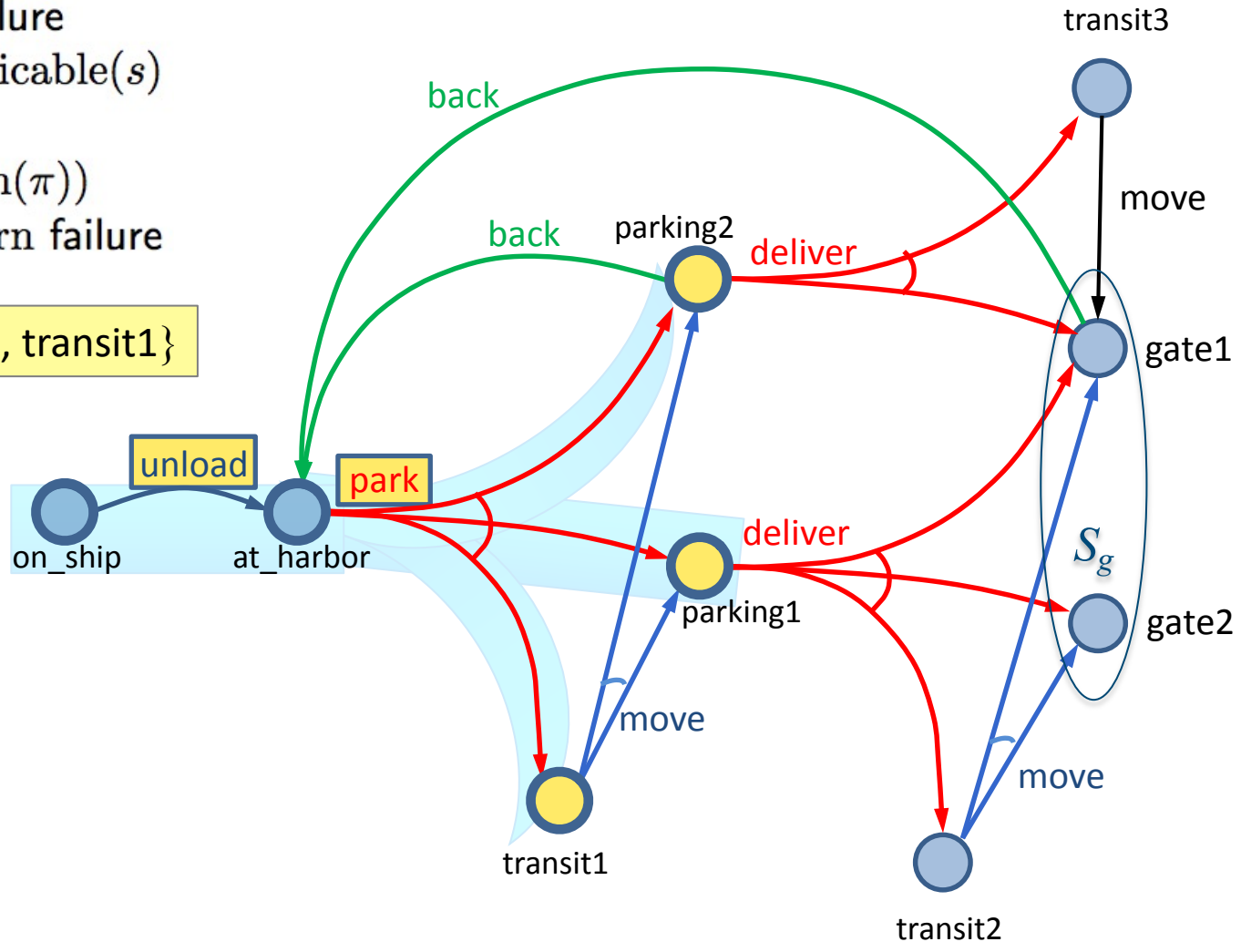


Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{$parking1, parking2, transit1$\}$

$\pi = \{$(on_ship, unload), (at_harbor, park)$\}$

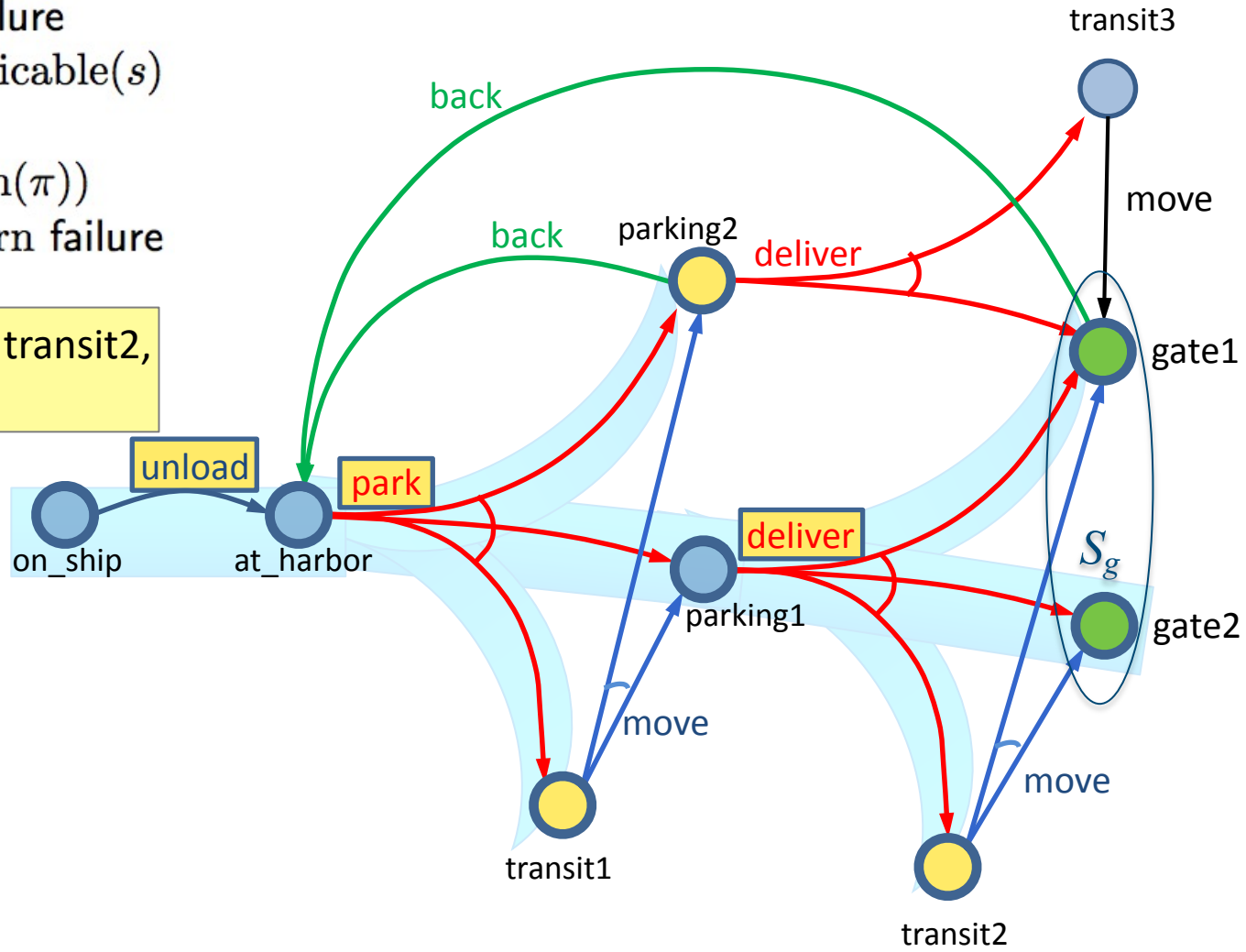


Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier$ = {parking2, transit1, transit2, gate1, gate2}

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}



Example

Find-Acyclic-Solution (Σ, s_0, S_g)

$\pi \leftarrow \varnothing$

$Frontier \leftarrow \{s_0\}$

for every $s \in Frontier \setminus S_g$ do

  $Frontier \leftarrow Frontier \setminus \{s\}$

  if $\text{Applicable}(s) = \varnothing$ then return failure

  nondeterministically choose $a \in \text{Applicable}(s)$

  $\pi \leftarrow \pi \cup (s, a)$

  $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$

  if has-loops$(\pi, a, Frontier)$ then return failure

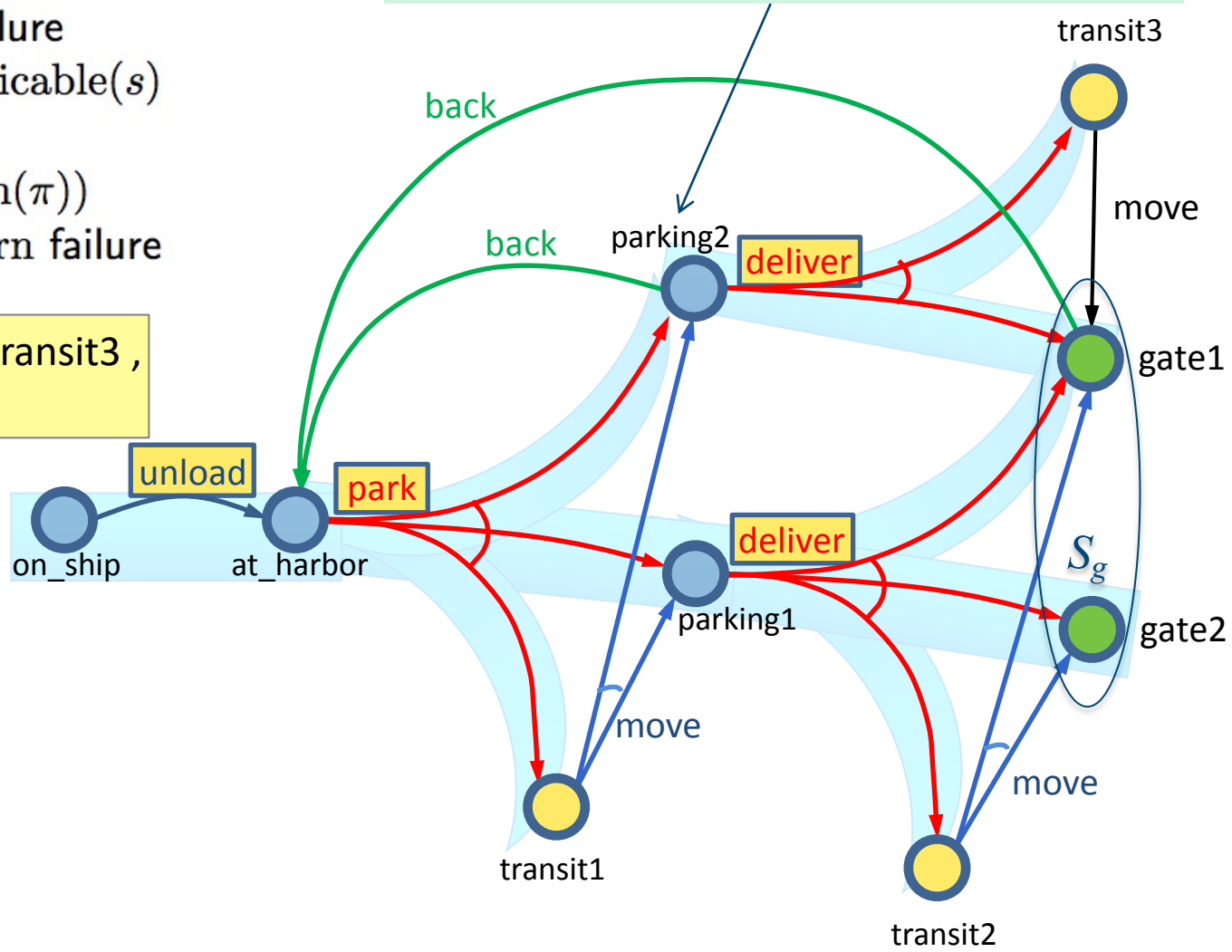
return π

$Frontier = \{$transit1, transit2, transit3 , gate1, gate2$\}$

$\pi = \{$(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, deliver)$\}$

nondeterministically choose back or deliver

- back ⇒ cycle, so return failure
- deliver ⇒ no cycle, so continue

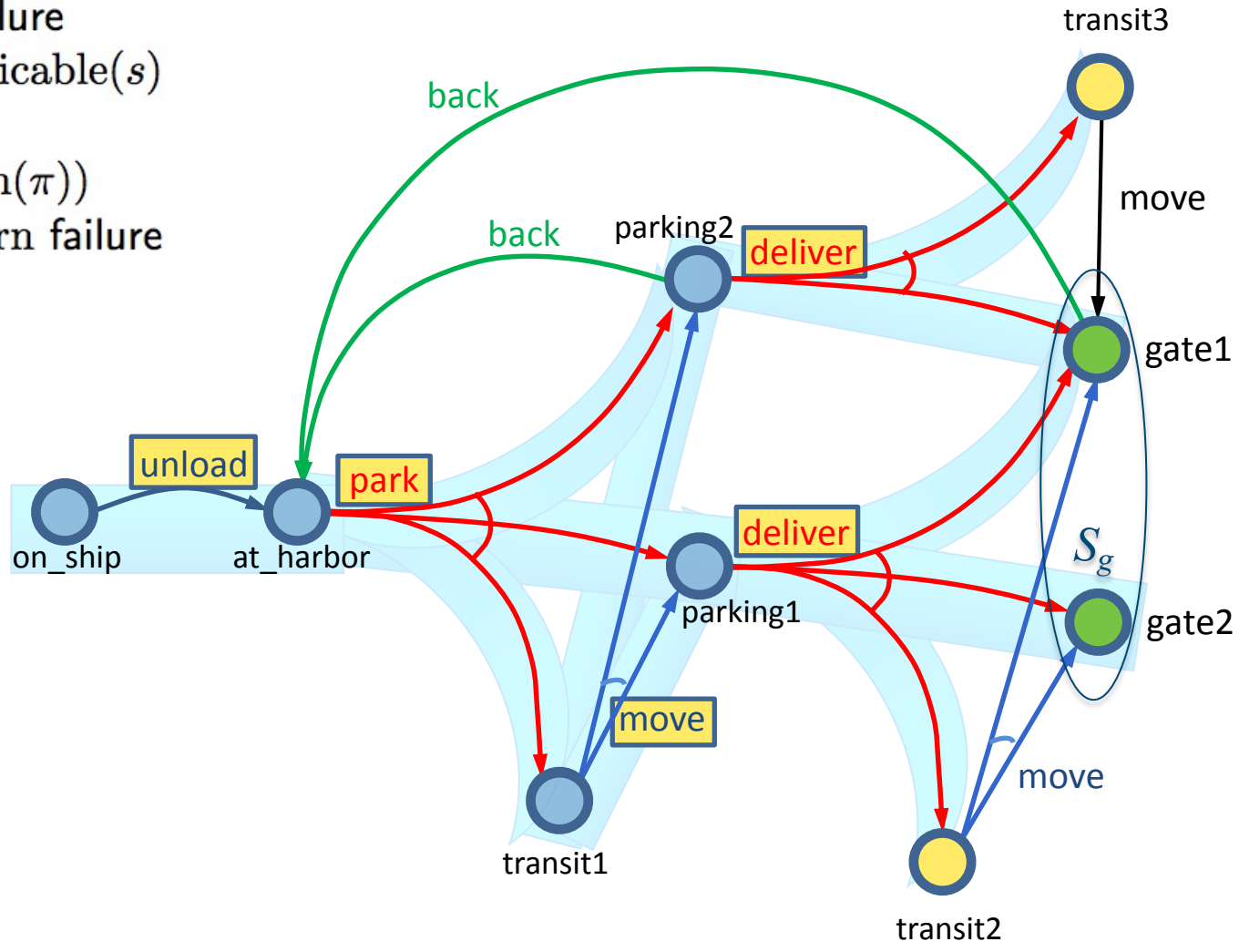


Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier$ = {transit2, transit3, gate1, gate2}

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, deliver), (transit1, move)}

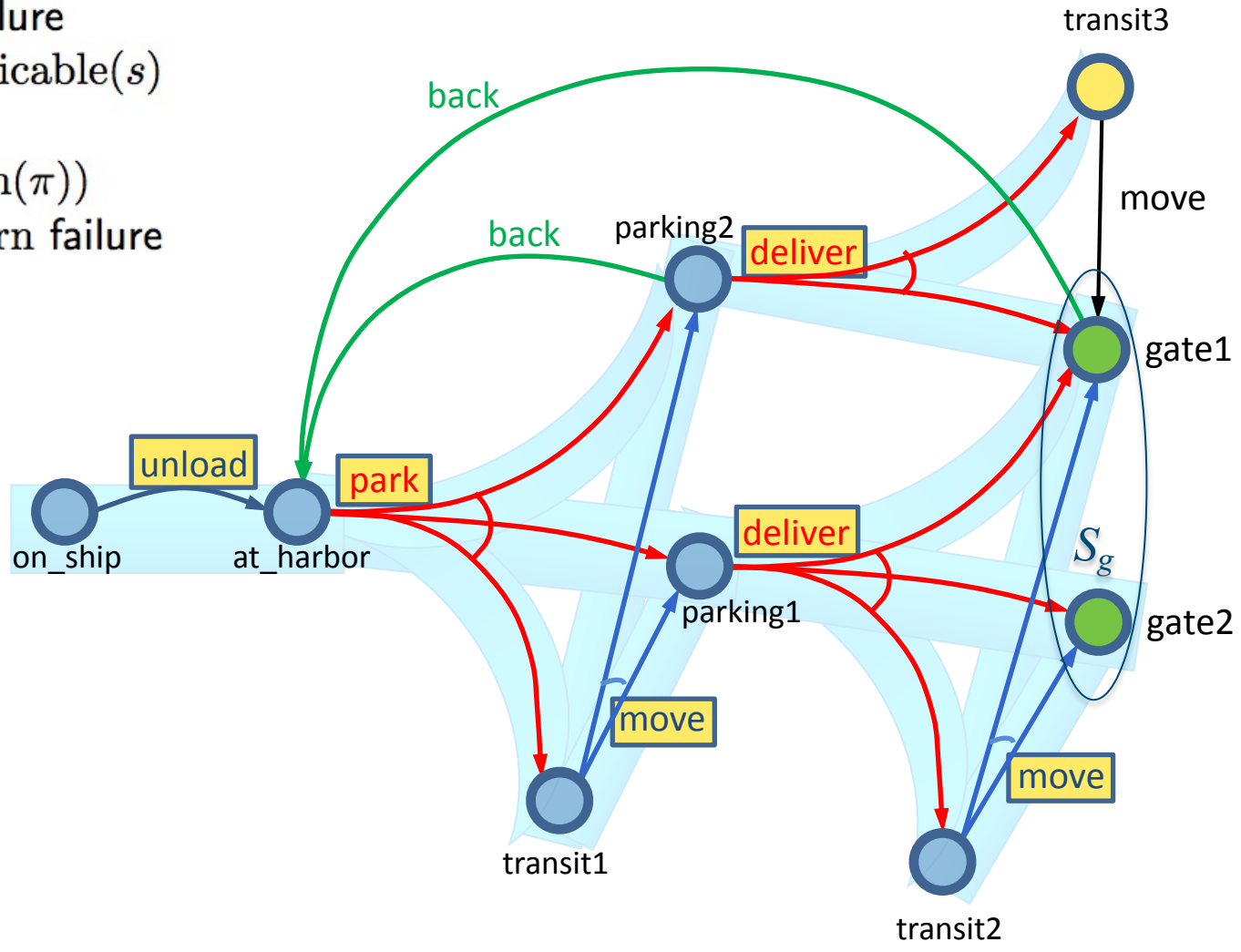


Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{$transit3, gate1, gate2$\}$

$\pi = \{$(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, deliver), (transit1, move), (transit2, move)$\}$



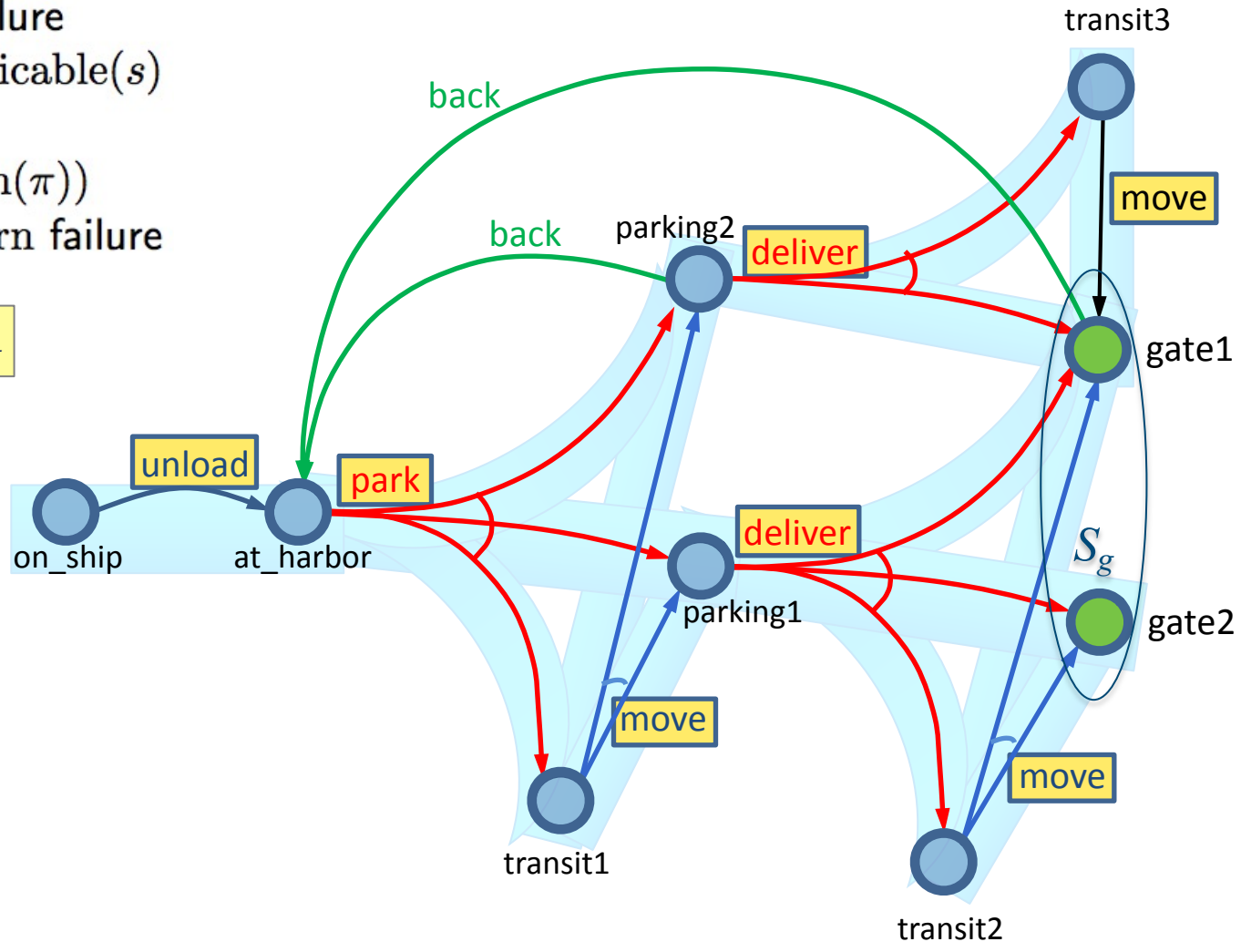
Example

Find-Acyclic-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{\text{gate1, gate2}\} \subseteq S_g$

Found a solution

$\pi = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit1, move),
(transit2, move),
(transit3, move)$\}$



Find-Safe-Solution

```
Find-Safe-Solution (Σ, s_0, S_g)
    π ← ∅
    Frontier ← {s_0}
    for every s ∈ Frontier \ S_g do
        Frontier ← Frontier \ {s}
        if Applicable(s) = ∅ then return failure
        nondeterministically choose a ∈ Applicable(s)
        π ← π ∪ (s, a)
        Frontier ← Frontier ∪ (γ(s, a) \ Dom(π))
        if has-unsafe-loops(π, a, Frontier) then return failure
    return π
```

Like Find-Acyclic-Solution except here:

Check for *unsafe* cycles:

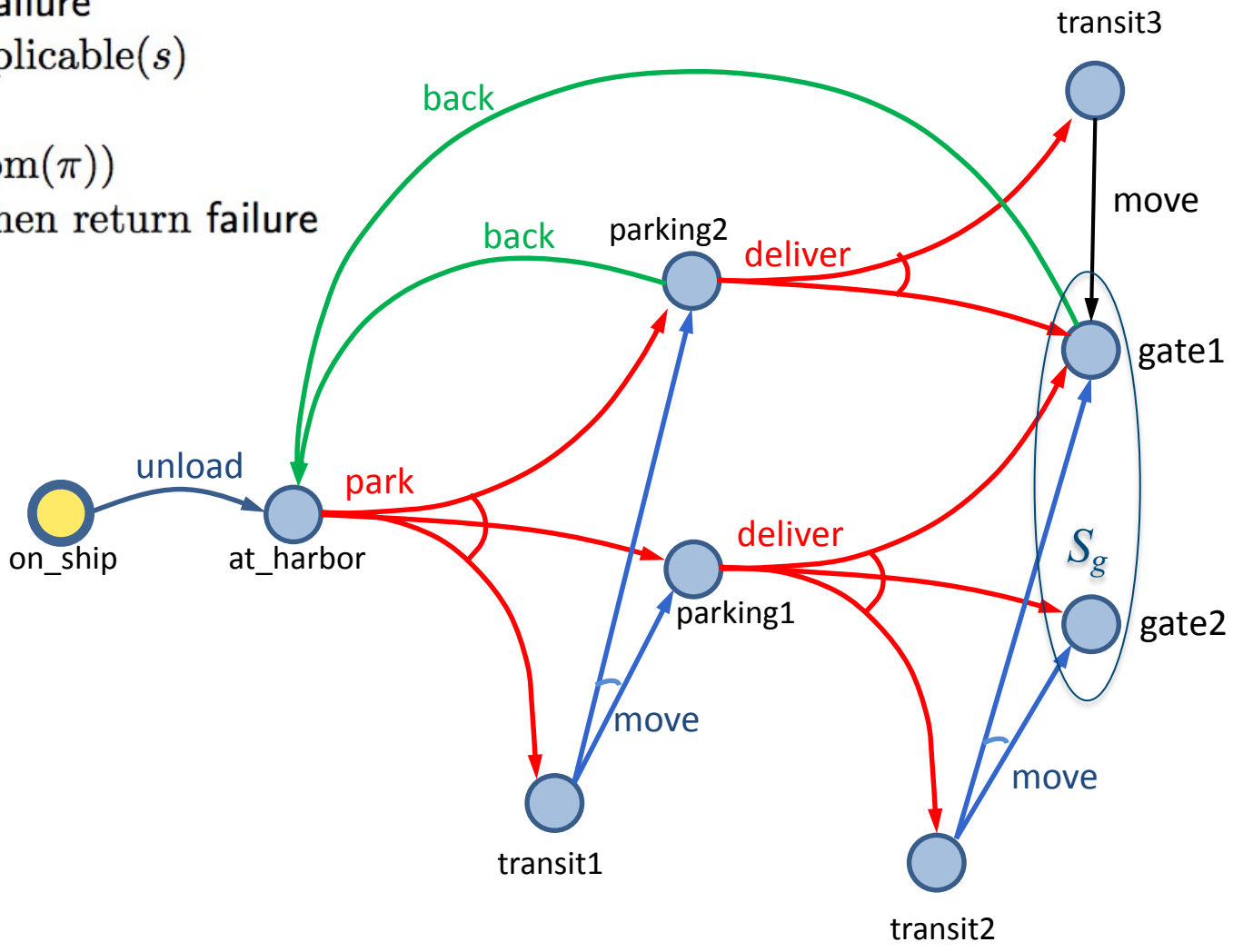
- Does $\gamma(s,a)$ include a state s' from which π can't take us to the frontier?
  - For each $s' \in \gamma(s,a) \cap \mathrm{Dom}(\pi)$, is $\hat{\gamma}(s',\pi) \cap \mathit{Frontier} = \emptyset$?
- If so, then π contains a cycle that can't be escaped

Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{\text{on_ship}\}$

$\pi = \{\}$

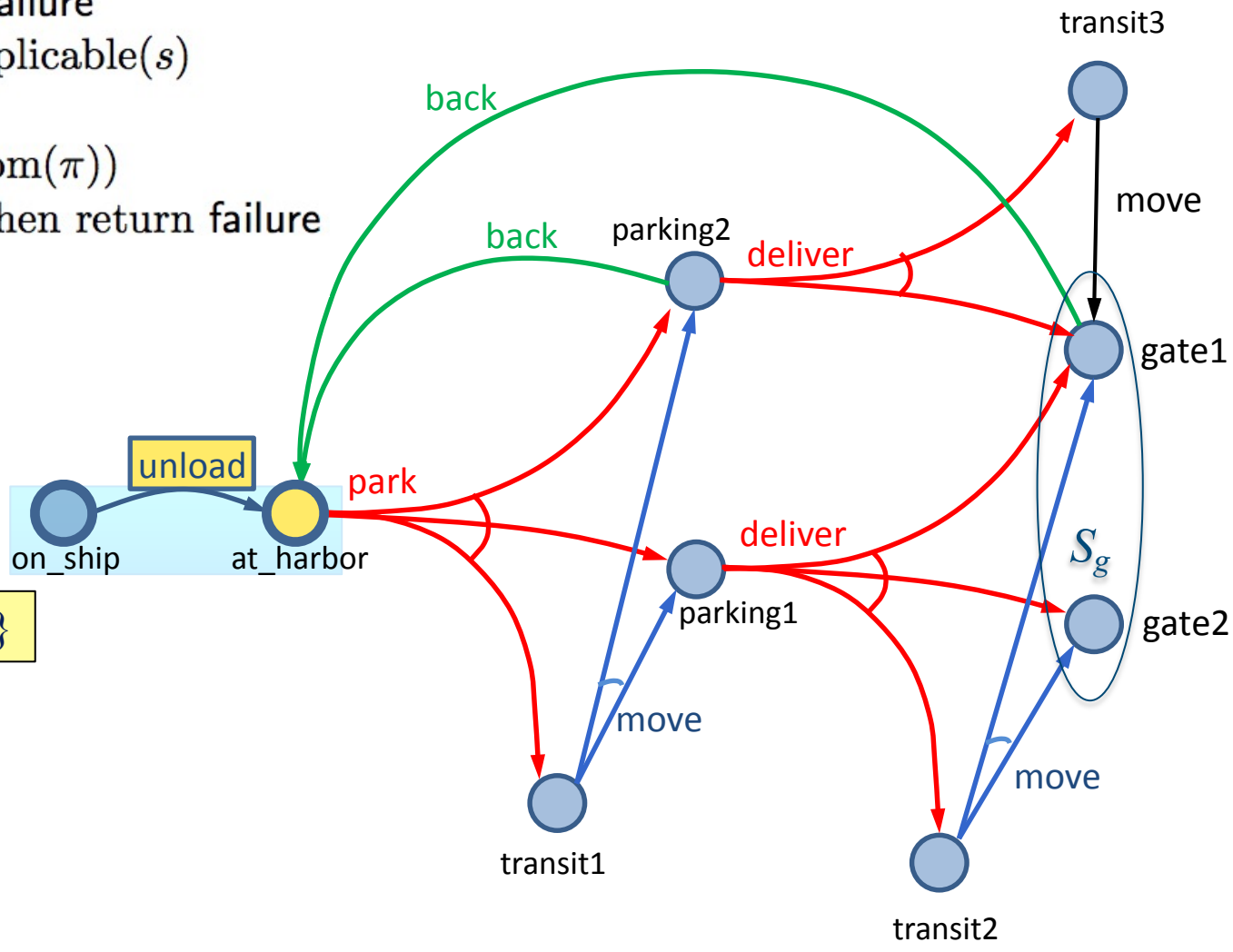


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{$at_harbor$\}$

$\pi = \{$(on_ship, unload)$\}$

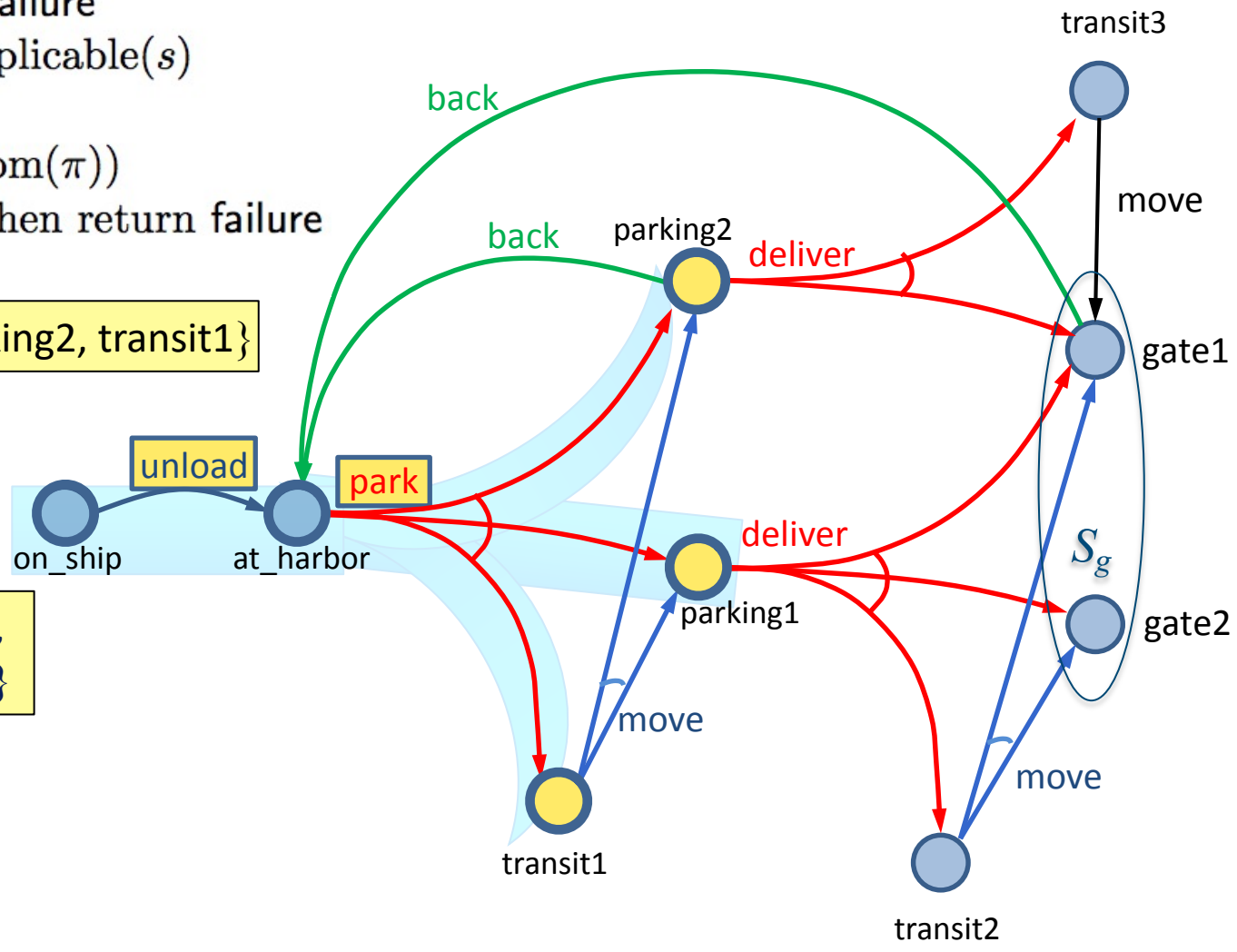


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{$parking1, parking2, transit1$\}$

$\pi = \{$(on_ship, unload), (at_harbor, park)$\}$

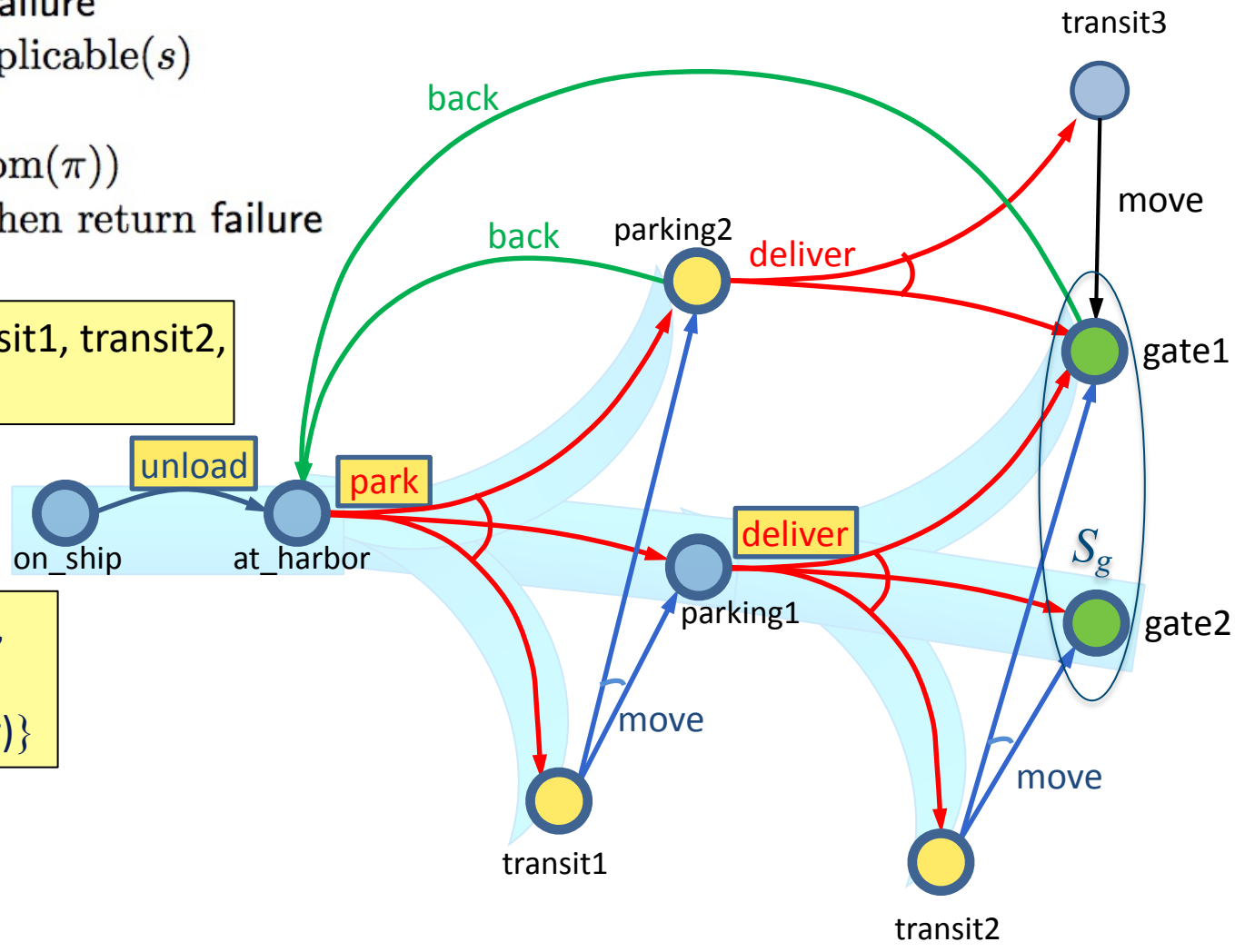


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier$ = {parking2, transit1, transit2, gate1, gate2}

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}



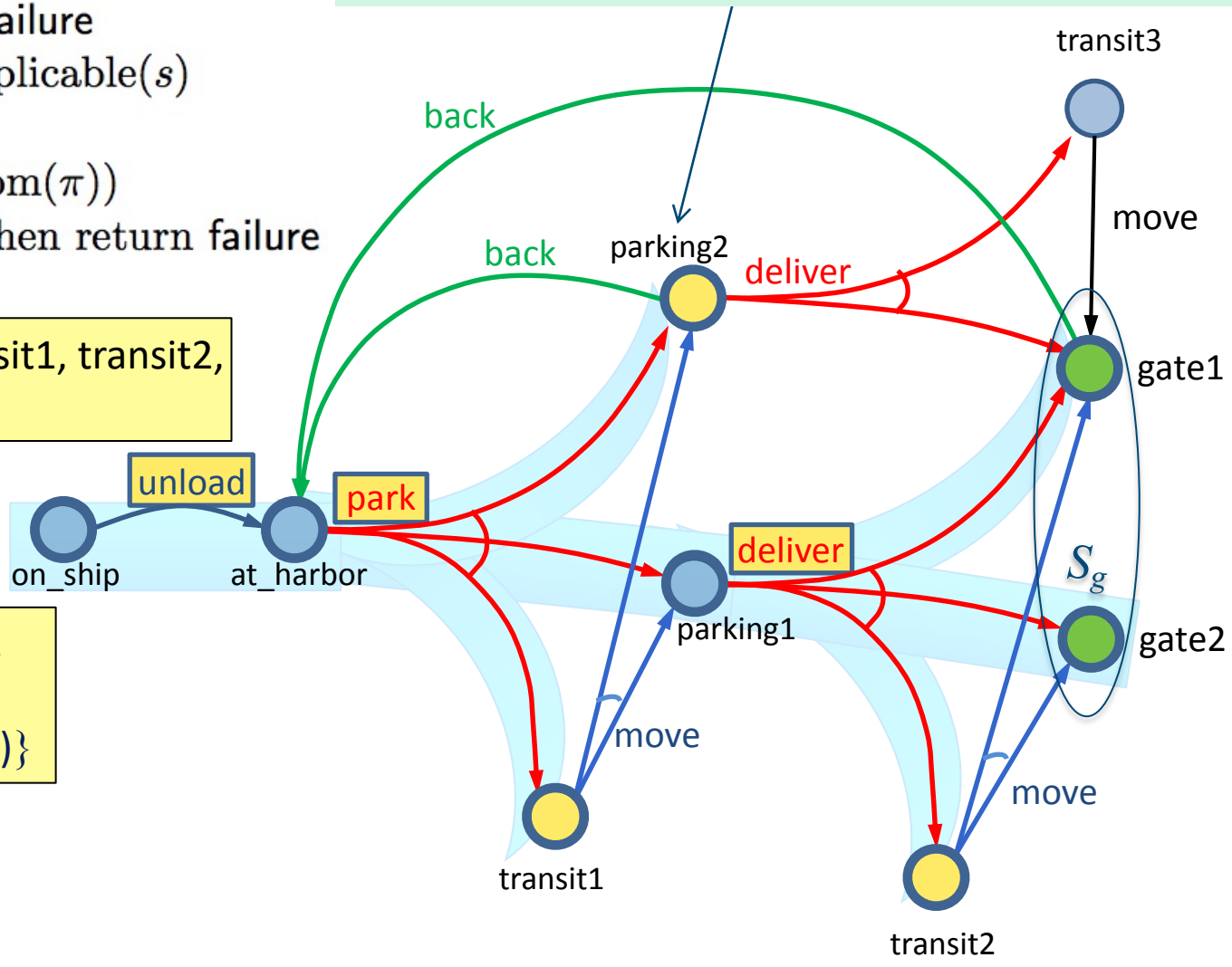
Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if Applicable$(s) = \varnothing$ then return failure
    nondeterministically choose $a \in$ Applicable(s)
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \mathrm{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

Nondeterministically choose either back or deliver
- back is OK because cycle is escapable

$Frontier$ = {parking2, transit1, transit2, gate1, gate2}

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver)}

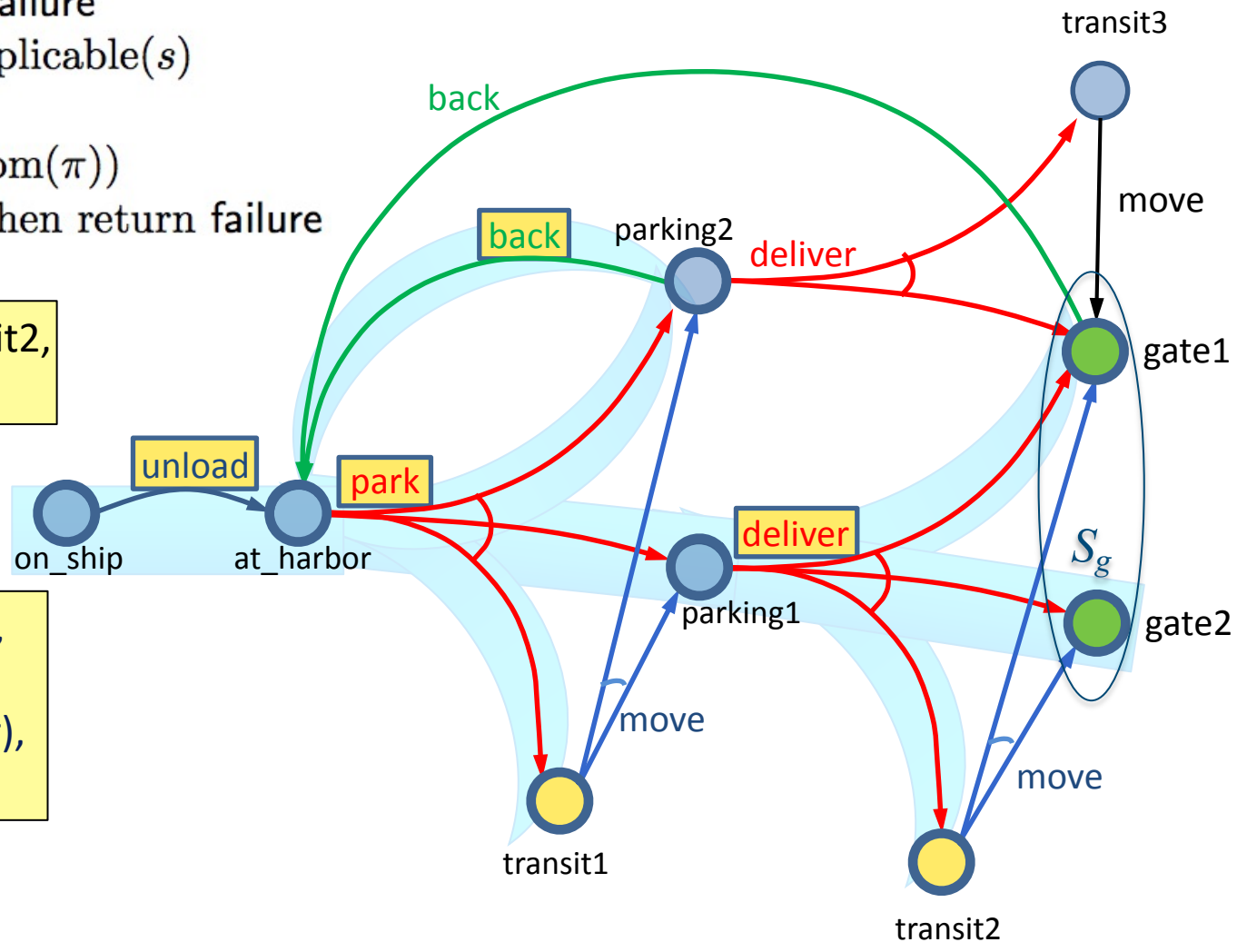


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier$ = {transit1, transit2, gate1, gate2}

π = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, back)}

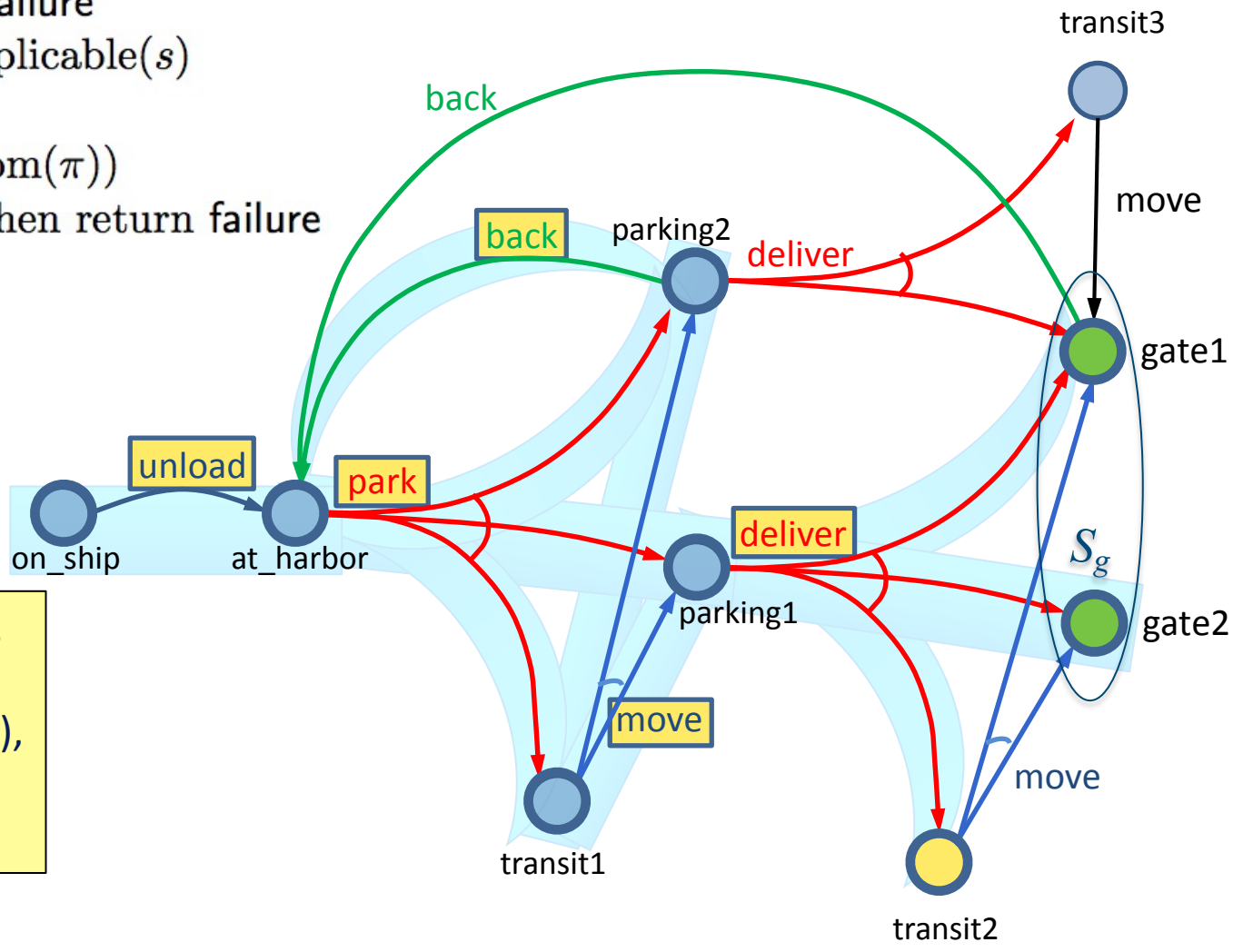


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{$transit2, gate1, gate2$\}$

$\pi = \{$(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, back), (transit1, move)$\}$

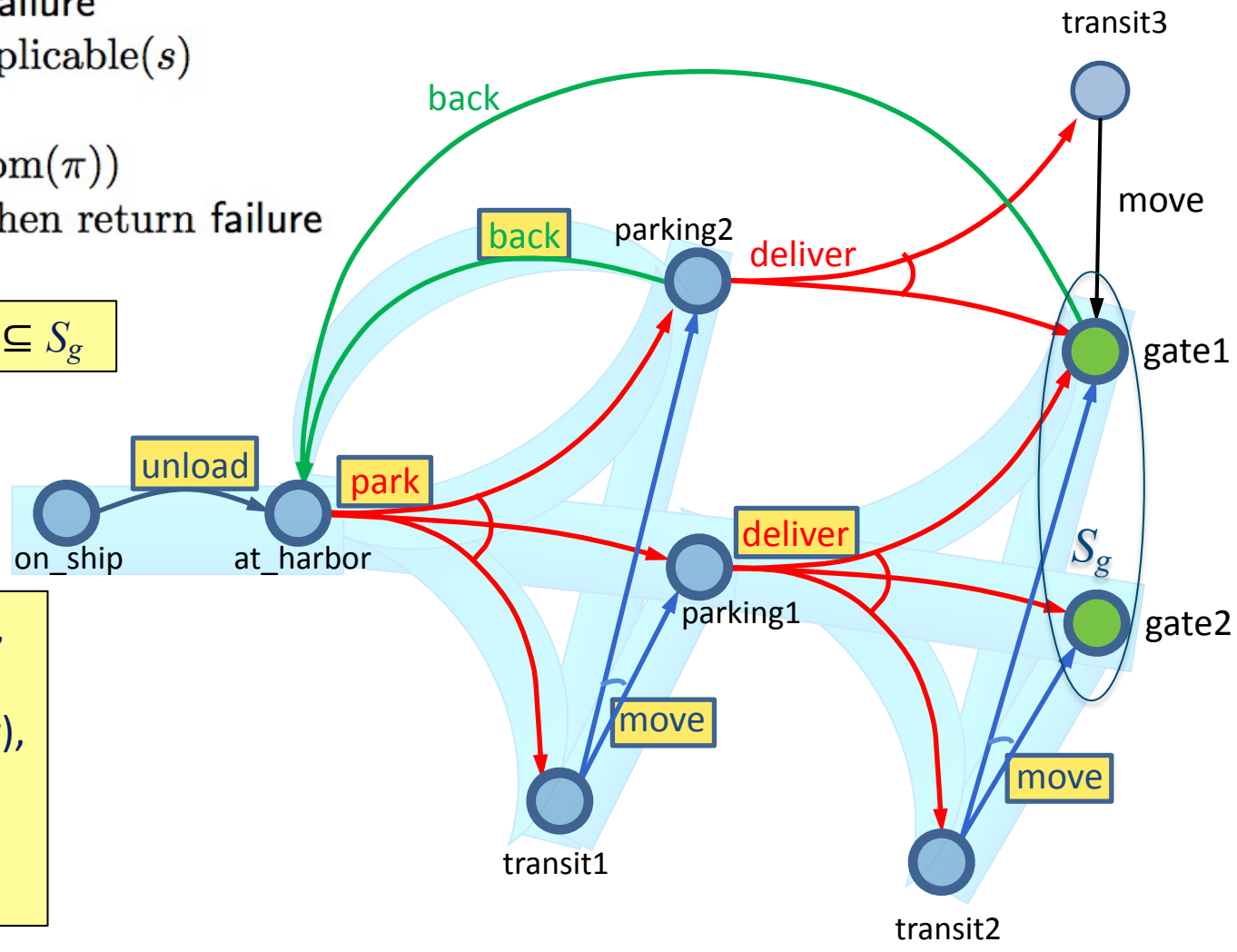


Example

Find-Safe-Solution (Σ, s_0, S_g)
  $\pi \leftarrow \varnothing$
  $Frontier \leftarrow \{s_0\}$
  for every $s \in Frontier \setminus S_g$ do
    $Frontier \leftarrow Frontier \setminus \{s\}$
    if $\text{Applicable}(s) = \varnothing$ then return failure
    nondeterministically choose $a \in \text{Applicable}(s)$
    $\pi \leftarrow \pi \cup (s, a)$
    $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus \text{Dom}(\pi))$
    if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
  return π

$Frontier = \{\text{gate1, gate2}\} \subseteq S_g$

$\pi = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, back),
(transit1, move),
(transit2, move)$\}$



Guided-Find-Safe-Solution

- Motivation: much easier to find solutions if they don't have to be safe
  - Find-Safe-Solution needs plans for all possible outcomes of actions
  - Find-Solution only needs a plan for one of them
- Idea:
  - loop
    - Find a (possibly unsafe) solution π
    - For each each leaf node of π
      - If the leaf node isn't a goal,
        - find a (possibly unsafe) solution and incorporate it into π

Guided-Find-Safe-Solution

Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'

π is a solution. Return the part that's reachable from s_0.

Choose any leaf s that isn't a goal. Find a (possibly unsafe) solution π' for s.

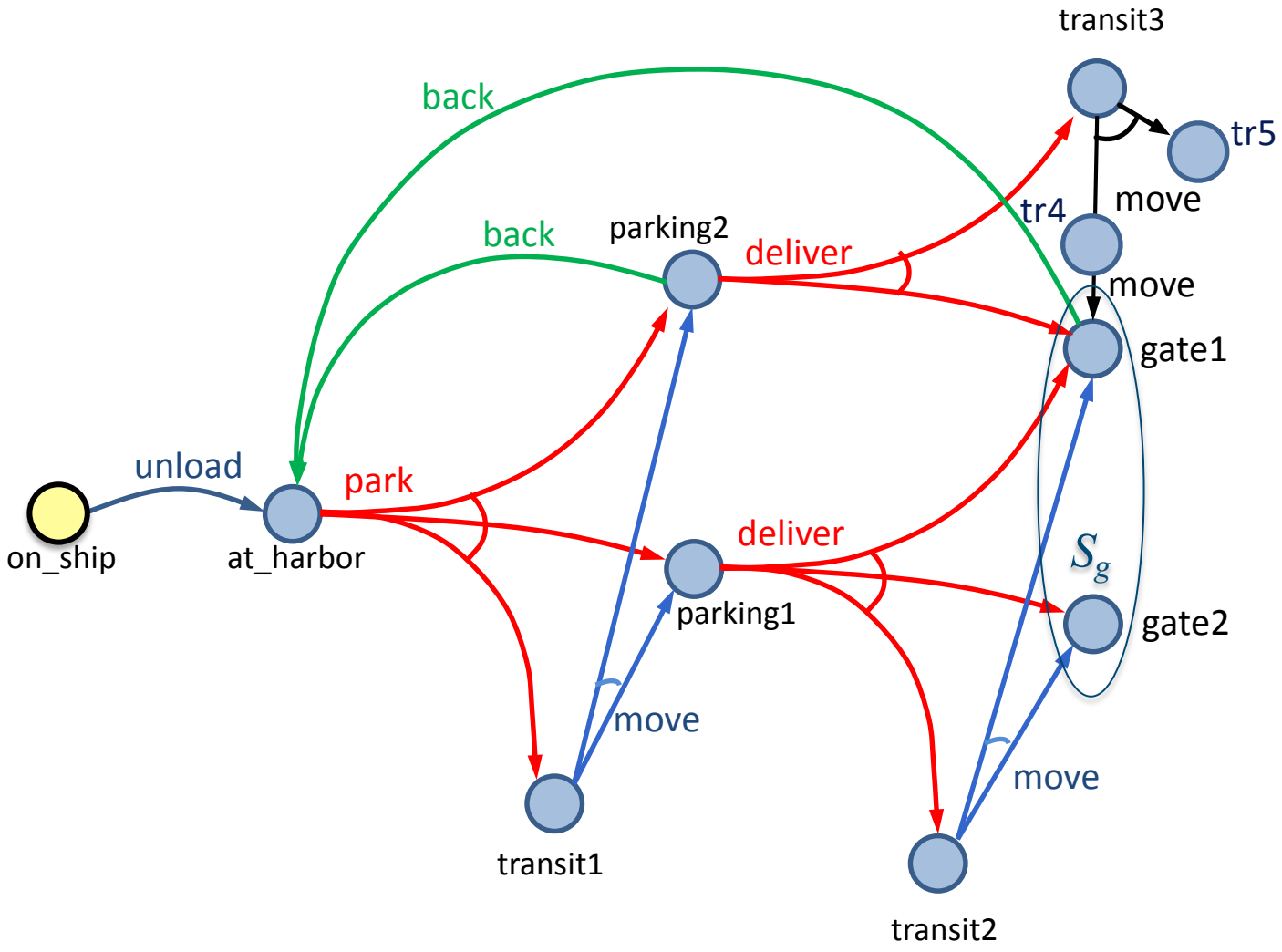
For each (s,a) in π', add to π unless π already has an action at s

s is unsolvable. For each (s',a) that can produce s, modify π and Σ so we'll never use a at s'

Guided-Find-Safe-Solution (Σ, s_0, S_g)

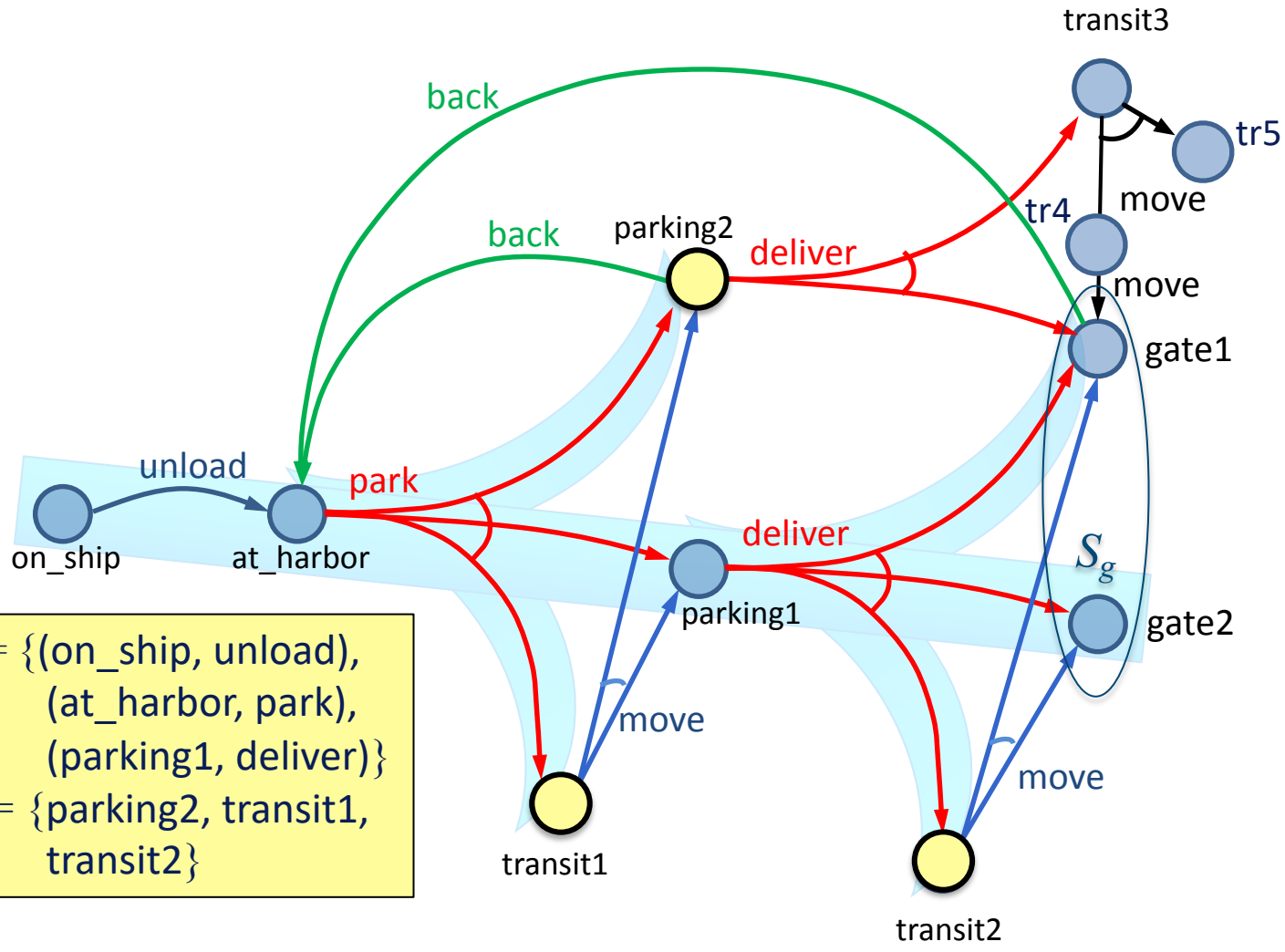
if $s_0 \in S_g$ then return$(\varnothing)$
if $Applicable(s_0) = \varnothing$ then return(failure)
$\pi \leftarrow \varnothing$
loop
  $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
  if $Q = \varnothing$ then do
    $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
    return(π)
  select arbitrarily $s \in Q$
  $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
  if $\pi' \neq$ failure then do
    $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
  else for every s' and a such that $s \in \gamma(s', a)$ do
    $\pi \leftarrow \pi \setminus \{(s', a)\}$
    make a not applicable in s'

Example



Example

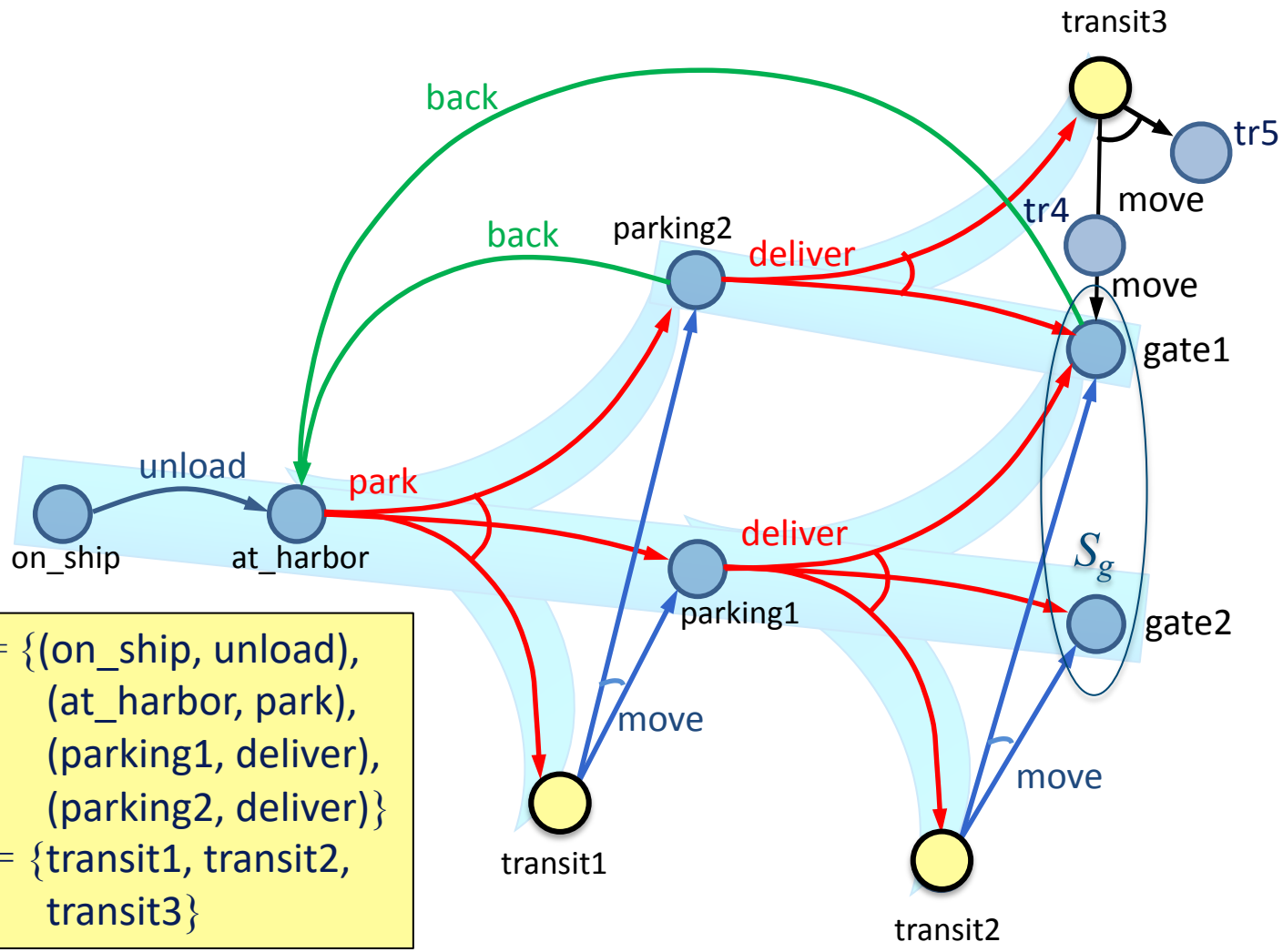
Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



$\pi = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver)$\}$
$Q = \{$parking2, transit1,
transit2$\}$

Example

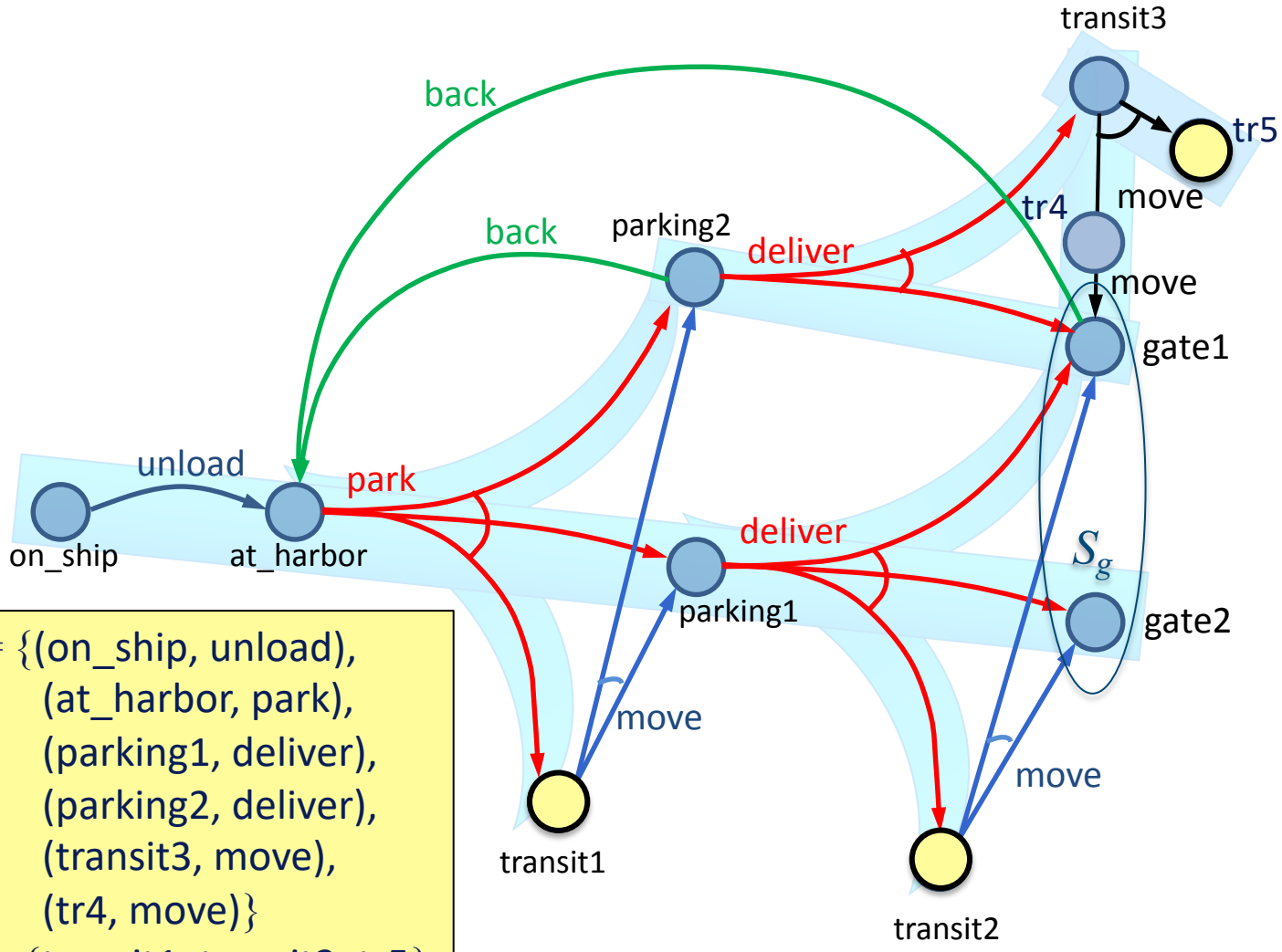
Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver)}
Q = {transit1, transit2,
transit3}

Example

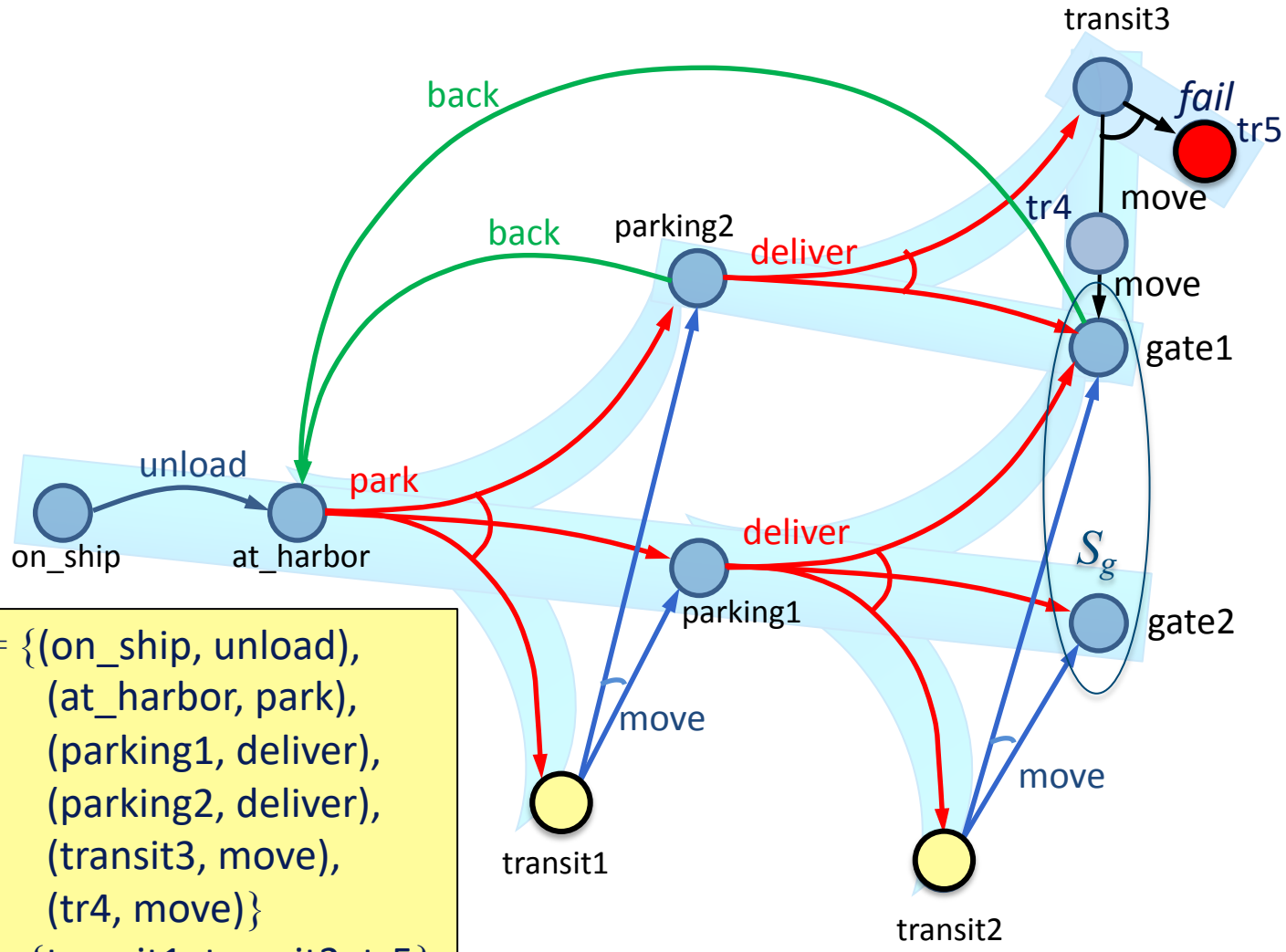
Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(tr4, move)}
Q = {transit1, transit2, tr5}

Example

Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



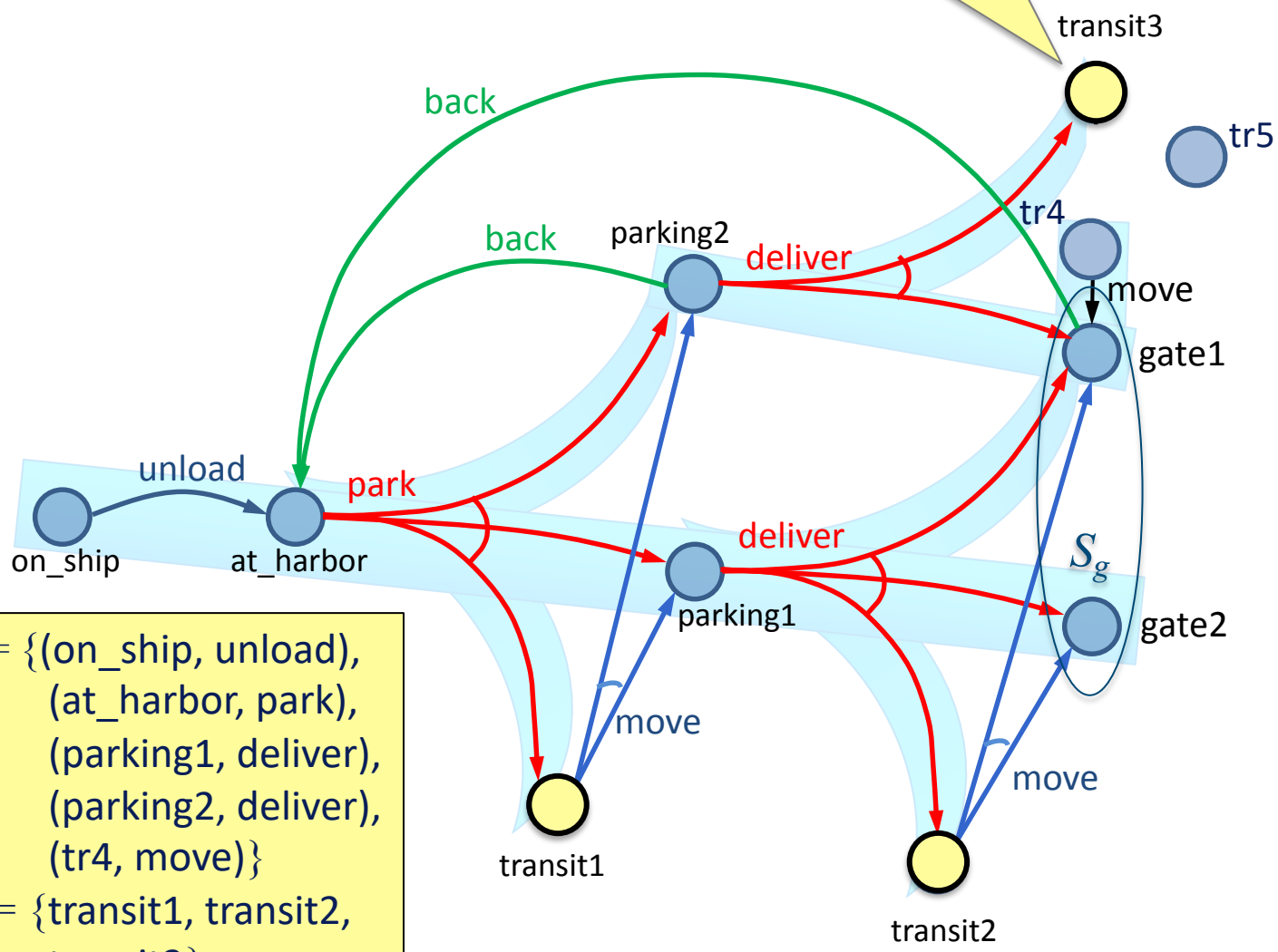
π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(transit3, move),
(tr4, move)}
Q = {transit1, transit2, tr5}

Guided-Find-Safe-Solution (Σ, s_0, S_g)

if $s_0 \in S_g$ then return$(\varnothing)$
if $Applicable(s_0) = \varnothing$ then return(failure)
$\pi \leftarrow \varnothing$
loop
  $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
  if $Q = \varnothing$ then do
    $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
    return(π)
  select arbitrarily $s \in Q$
  $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
  if $\pi' \neq$ failure then do
    $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
  else for every s' and a such that $s \in \gamma(s', a)$ do
    $\pi \leftarrow \pi \setminus \{(s', a)\}$
    make a not applicable in s'

Example

Modify Σ_d to make move inapplicable at transit3

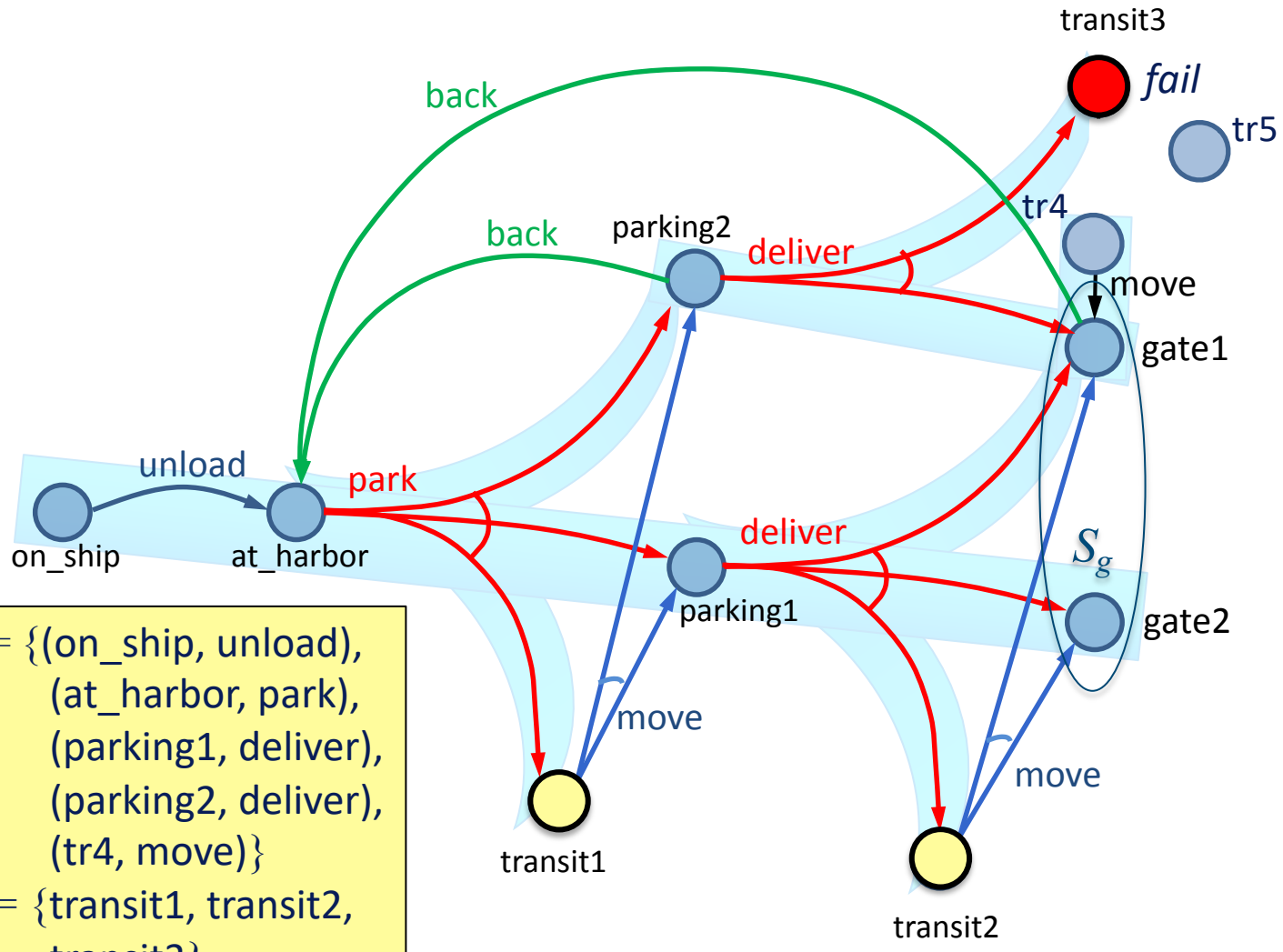


π = {(on_ship, unload), (at_harbor, park), (parking1, deliver), (parking2, deliver), (tr4, move)}

Q = {transit1, transit2, transit3}

Example

Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'

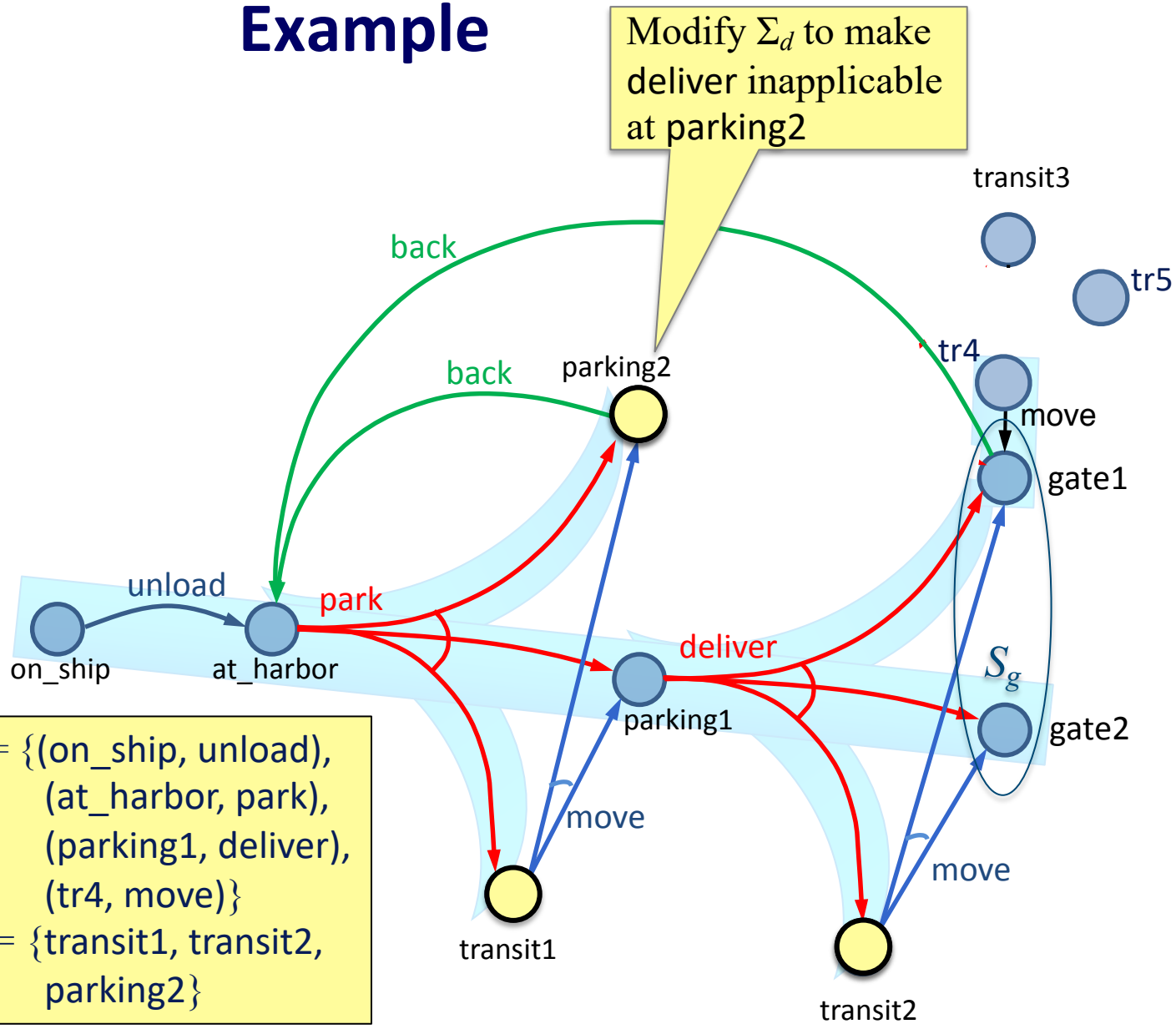


π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(parking2, deliver),
(tr4, move)}
Q = {transit1, transit2,
transit3}

Guided-Find-Safe-Solution (Σ, s_0, S_g)

if $s_0 \in S_g$ then return$(\varnothing)$
if $Applicable(s_0) = \varnothing$ then return(failure)
$\pi \leftarrow \varnothing$
loop
  $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
  if $Q = \varnothing$ then do
    $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
    return(π)
  select arbitrarily $s \in Q$
  $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
  if $\pi' \neq$ failure then do
    $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
  else for every s' and a such that $s \in \gamma(s', a)$ do
    $\pi \leftarrow \pi \setminus \{(s', a)\}$
    make a not applicable in s'

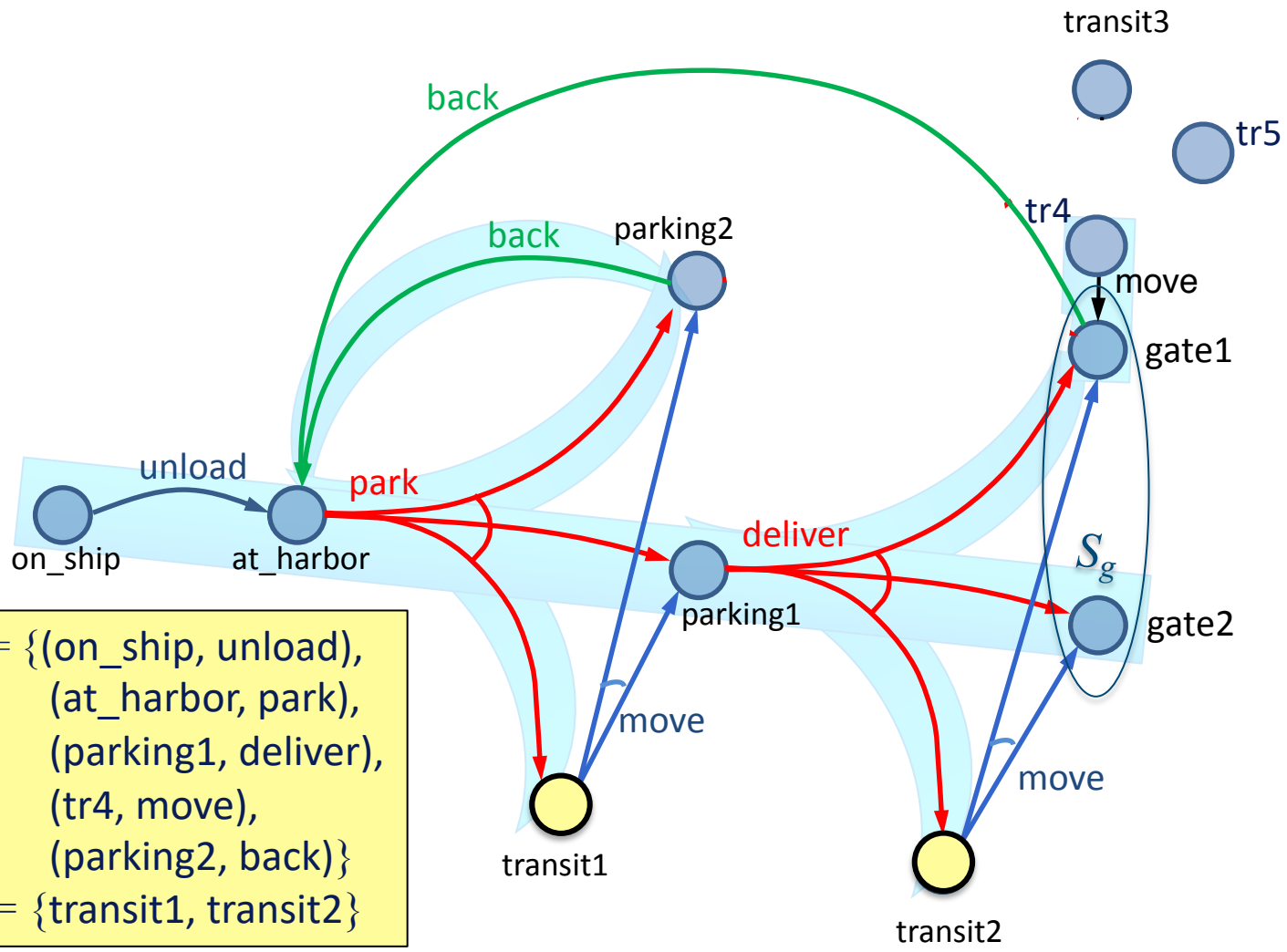
Example



$\pi = \{$(on_ship, unload), (at_harbor, park), (parking1, deliver), (tr4, move)$\}$
$Q = \{$transit1, transit2, parking2$\}$

Example

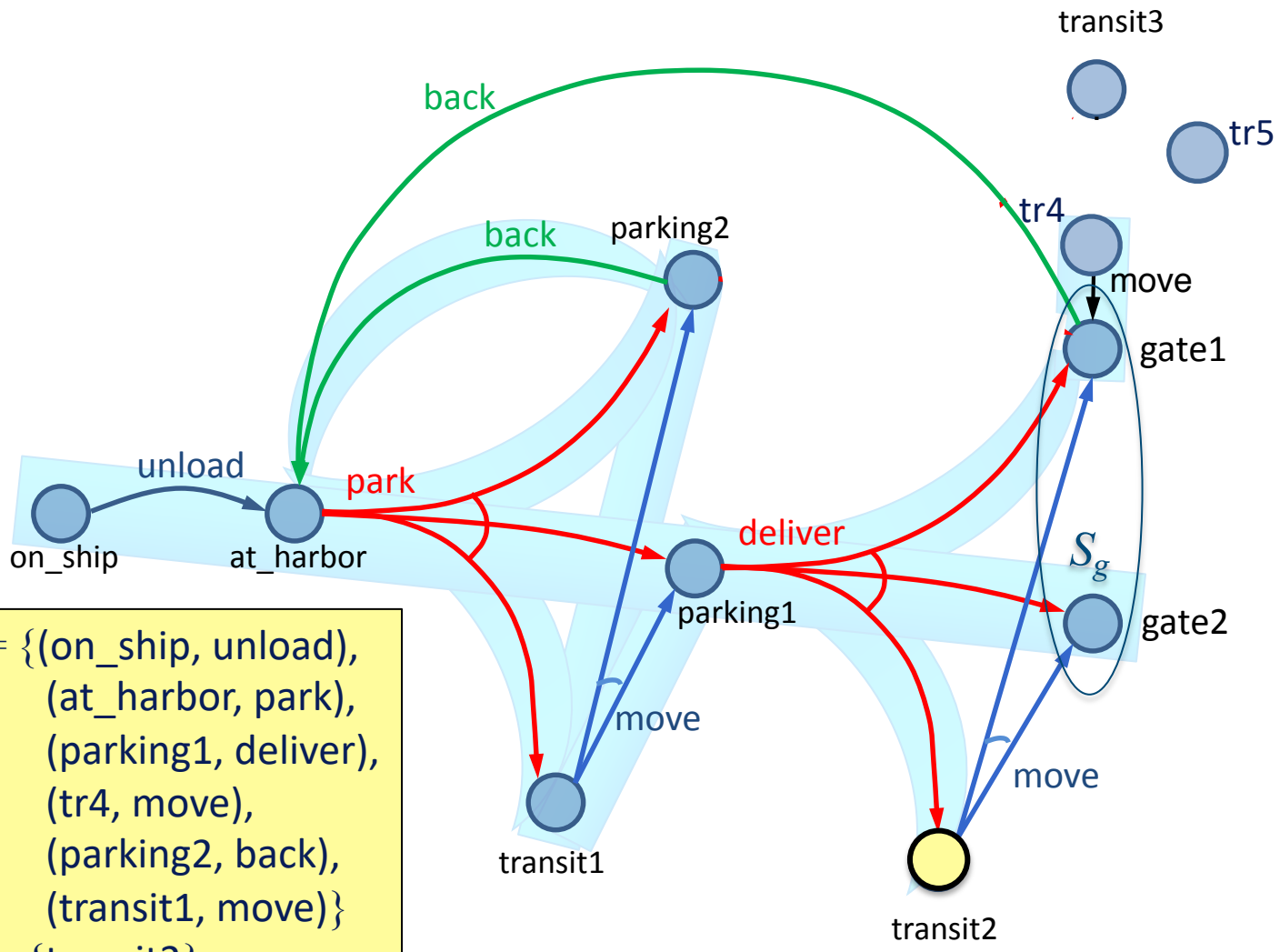
Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(tr4, move),
(parking2, back)}
Q = {transit1, transit2}

Example

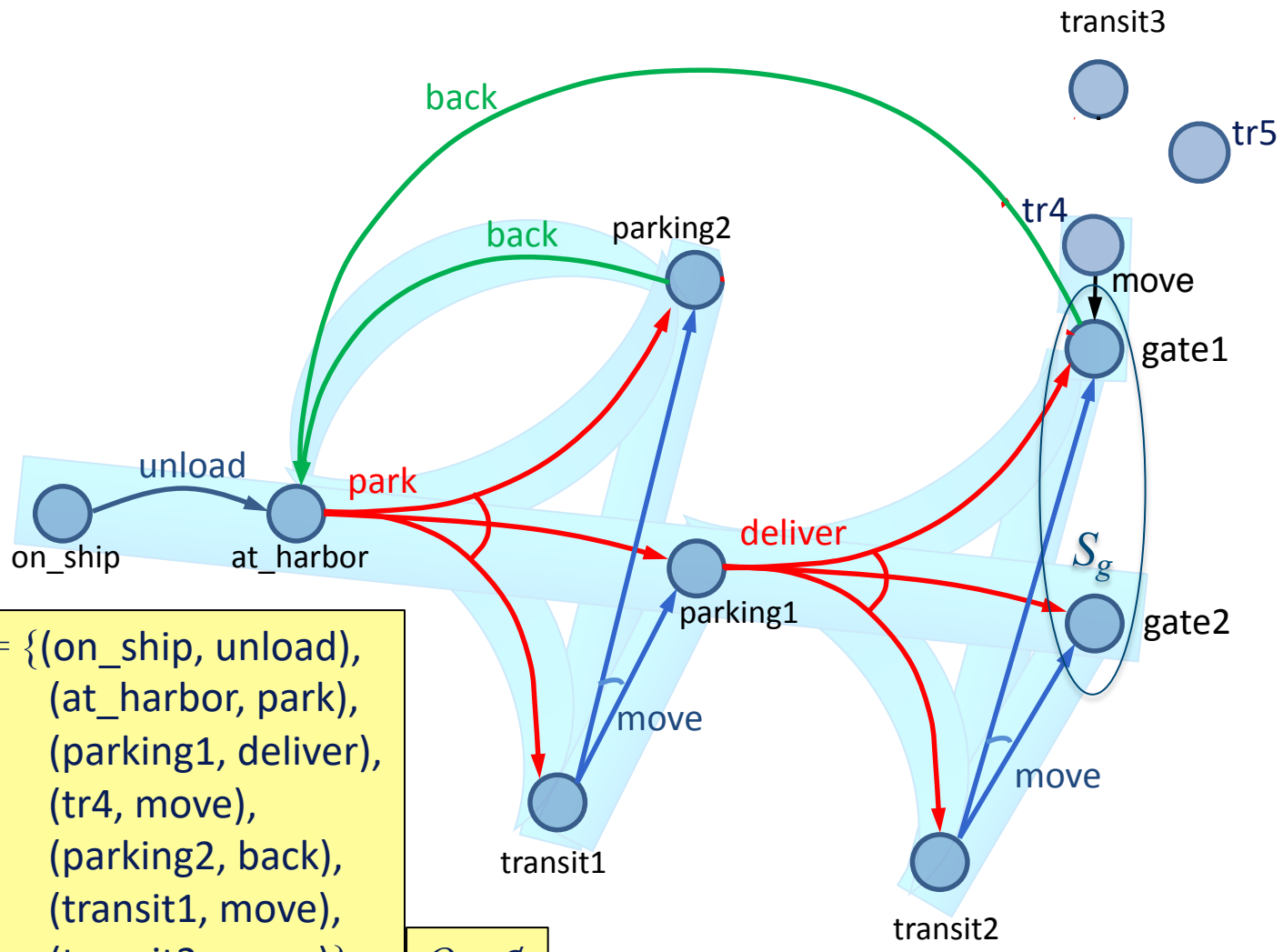
Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



π = {(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(tr4, move),
(parking2, back),
(transit1, move)}
Q = {transit2}

Example

Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'



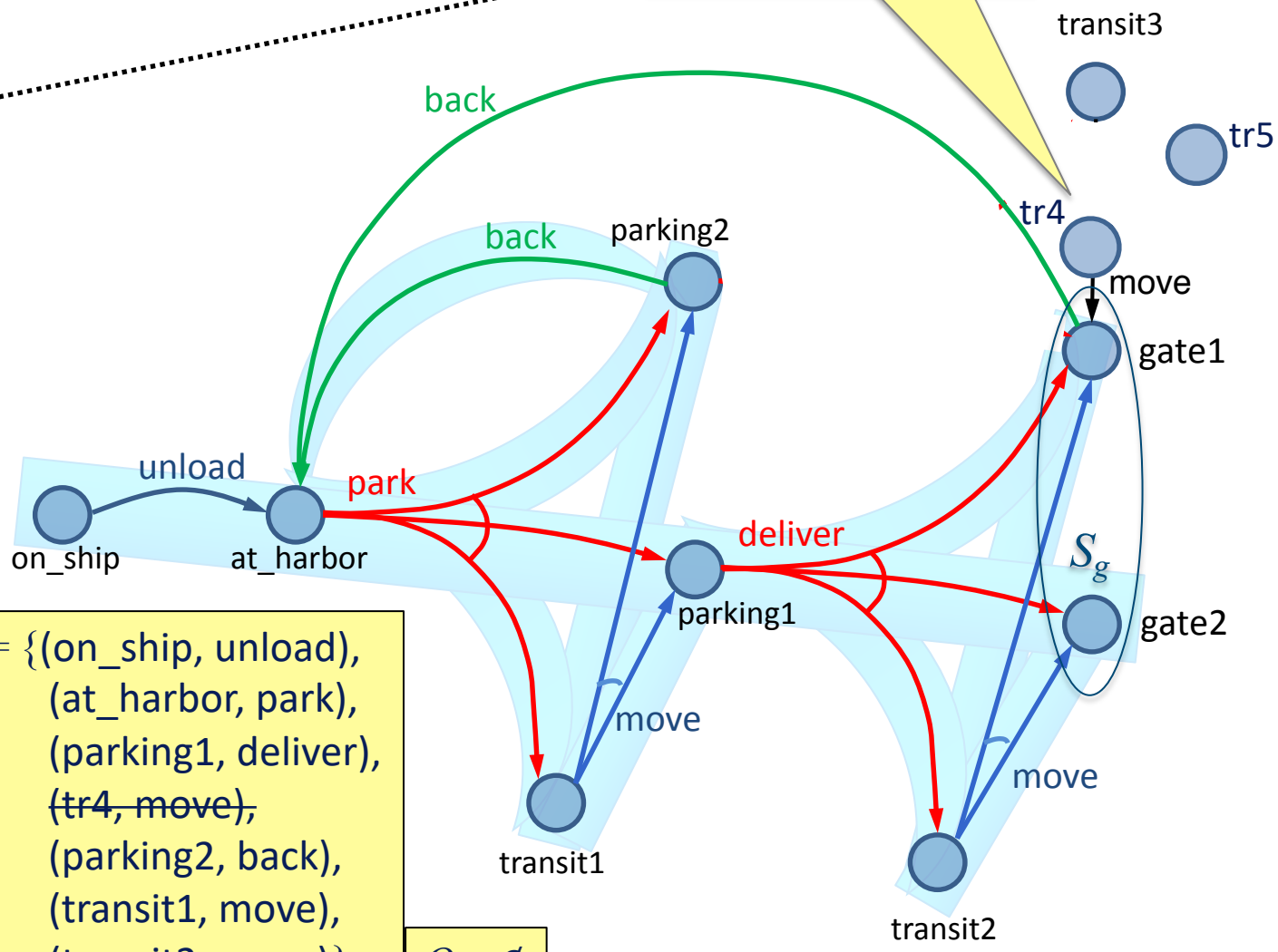
$\pi = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
(tr4, move),
(parking2, back),
(transit1, move),
(transit2, move)$\}$

$Q = \emptyset$

Example

Guided-Find-Safe-Solution (Σ, s_0, S_g)
  if $s_0 \in S_g$ then return$(\varnothing)$
  if $Applicable(s_0) = \varnothing$ then return(failure)
  $\pi \leftarrow \varnothing$
  loop
    $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
    if $Q = \varnothing$ then do
      $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
      return(π)
    select arbitrarily $s \in Q$
    $\pi' \leftarrow$ Find-Solution(Σ, s, S_g)
    if $\pi' \neq$ failure then do
      $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin \mathrm{Dom}(\pi)\}$
    else for every s' and a such that $s \in \gamma(s', a)$ do
      $\pi \leftarrow \pi \setminus \{(s', a)\}$
      make a not applicable in s'

Remove (tr4,move) from π because π can't ever reach tr4



$\pi = \{$(on_ship, unload),
(at_harbor, park),
(parking1, deliver),
~~(tr4, move),~~
(parking2, back),
(transit1, move),
(transit2, move)$\}$

$Q = \emptyset$

Guided-Find-Safe-Solution (Σ, s_0, S_g)

```
if s_0 ∈ S_g then return(∅)
if Applicable(s_0) = ∅ then return(failure)
π ← ∅
loop
    Q ← leaves(s_0, π) \ S_g
    if Q = ∅ then do
        π ← π \ {(s, a) ∈ π | s ∉ γ̂(s_0, π)}
        return(π)
    select arbitrarily s ∈ Q
    π' ← Find-Solution(Σ, s, S_g)
    if π' ≠ failure then do
        π ← π ∪ {(s, a) ∈ π' | s ∉ Dom(π)}
    else for every s' and a such that s ∈ γ(s', a) do
        π ← π \ {(s', a)}
        make a not applicable in s'
```

Discussion

- How to implement it?
  - Need implementation of Find-Solution
  - Need it to be very efficient
    - We'll call it many times
- Idea: instead of Find-Solution, use a classical planner
  - Any of the algorithms from Chapter 2
  - Efficient algorithms, search heuristics
- Need to convert the actions into something the classical planner can use …