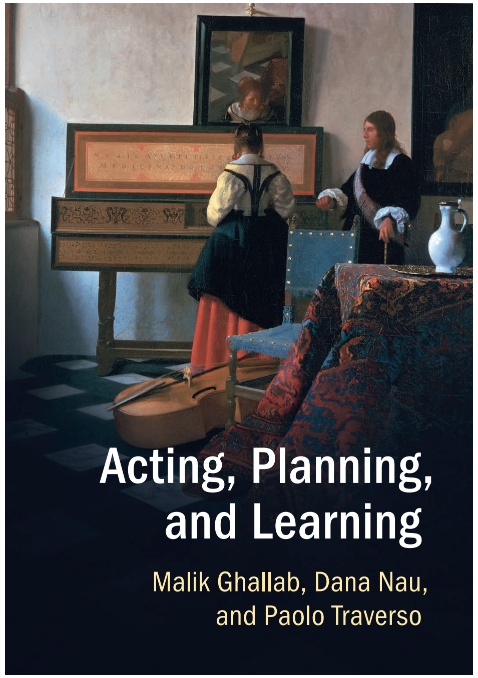
Last update: 11:10 PM, March 7, 2025



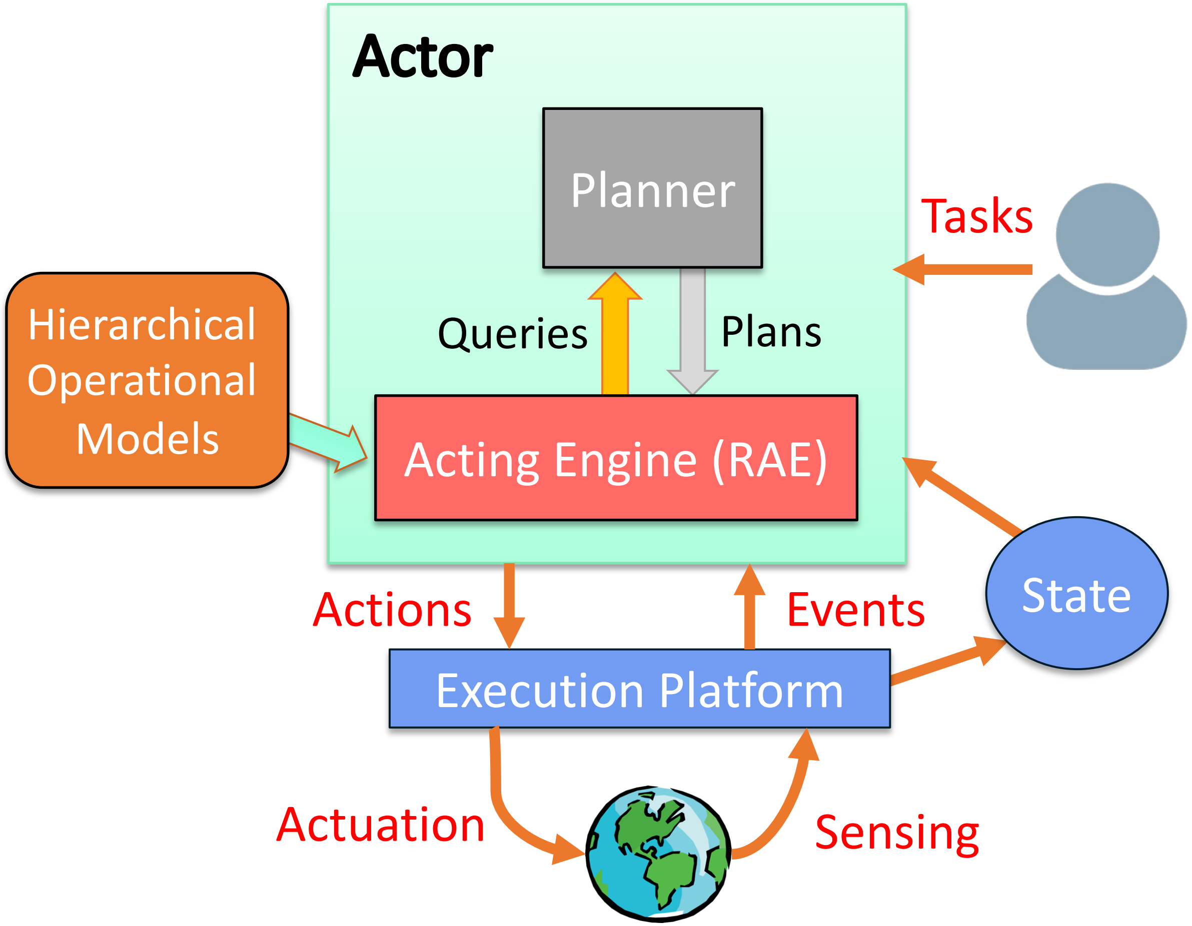
Chapters 15, 16
Hierarchical Refinement Planning, Learning

Dana S. Nau
University of Maryland

with contributions from
Mark “mak” Roberts

Outline

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



RAE (Ch. 14 Review)

RAE

$Agenda \leftarrow$ empty list

while True **do**

1 **for** each new task or event τ to be addressed **do**

2 observe current state ξ

3 $m \leftarrow \text{Guide}(\xi, \tau, \langle(\tau, \text{nil}, 1, \varnothing)\rangle, d_{max}, n_{ro})$

4 **if** $m = \varnothing$ **then** output(τ, "failed")

else $Agenda \leftarrow Agenda \cup \{\langle(\tau, m, 1, \varnothing)\rangle\}$

5 **for** each stack $\in$ *Agenda* **do**

6 observe current state ξ

stack $\leftarrow$ Progress(stack, ξ)

7 **if** stack $= \varnothing$ **then**

$Agenda \leftarrow Agenda \setminus$ stack

output(τ, "succeeded")

8 **else if** stack =failure **then**

$Agenda \leftarrow Agenda \setminus$ stack

output(τ, "failed")

An abstraction of RAE we will use:

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)

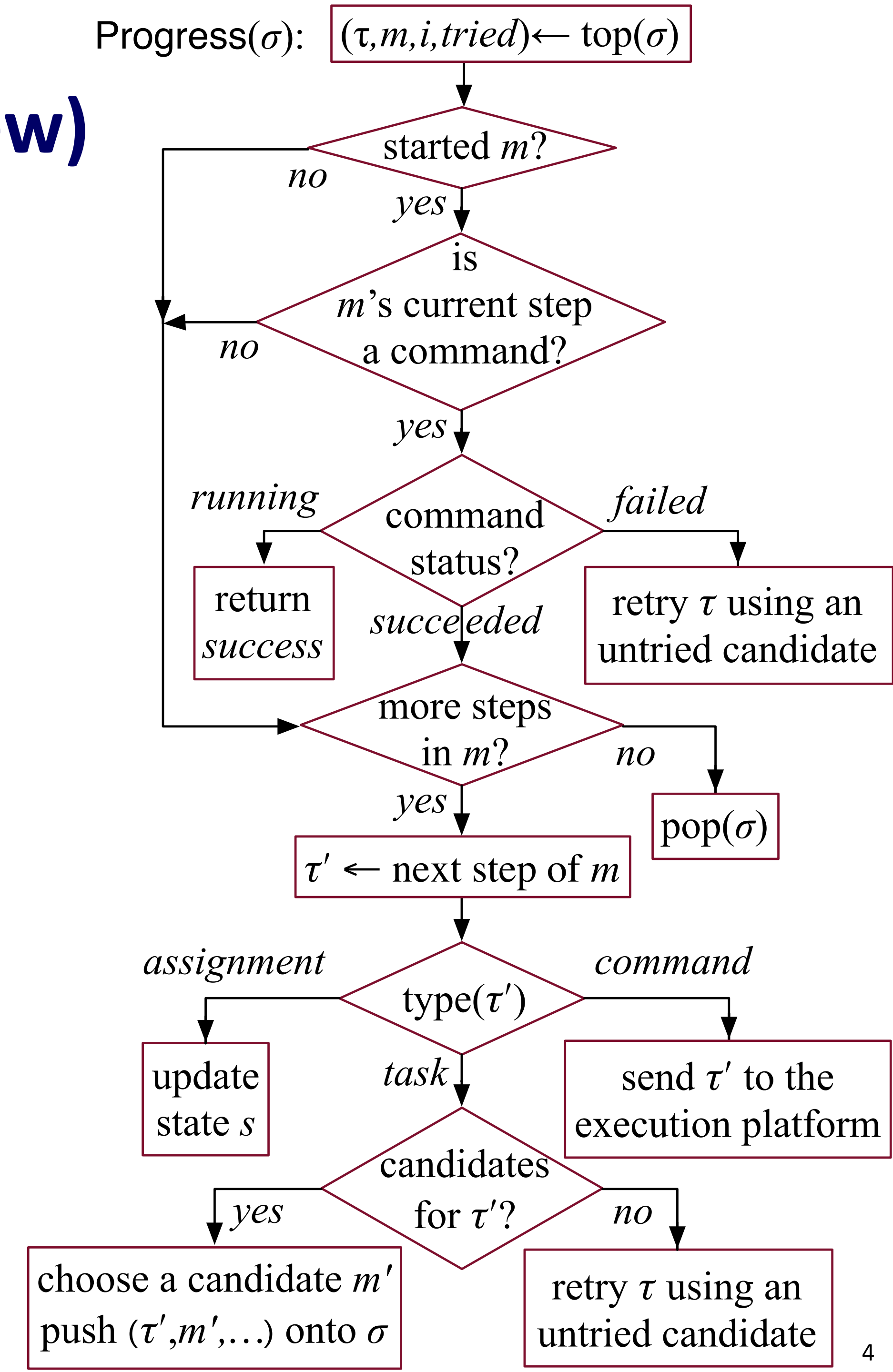
if σ is finished then remove it

In Ch. 14, Guide was a heuristic choice.

We will explore some possible ways to do Guide.

Progress (Ch. 14 Review)

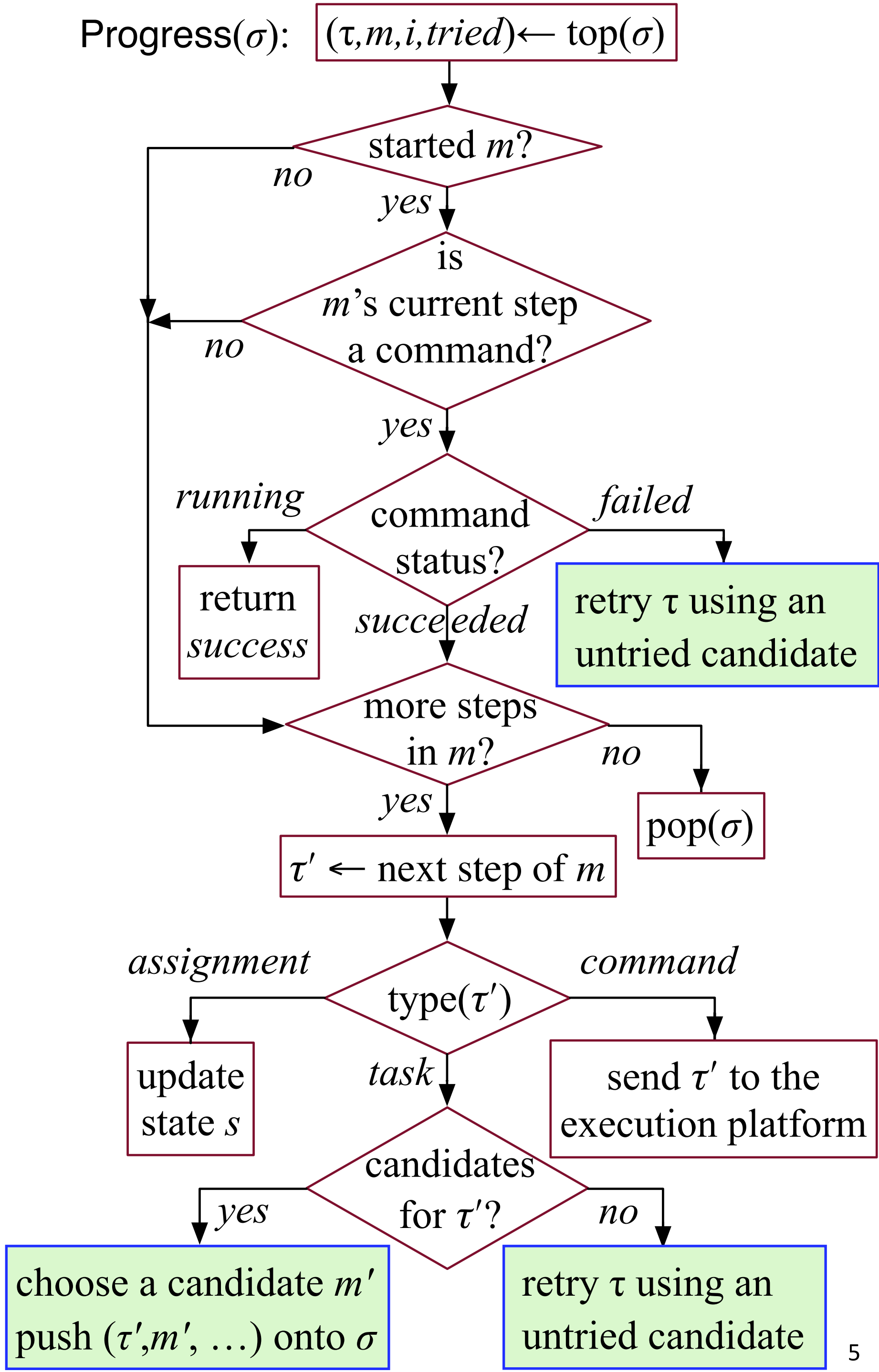
```
Progress(stack, ξ)
    (τ, m, i, tried) ← top(stack)
1   if m[i] is an already triggered action then          // i is the current step of m
        case exec-status (m[i])=
            running:  return stack
2           failed:   return Retry(stack)
            done:     return Next(stack, ξ)
3   else                                                  // i is the next step of m
        if m[i] is an assignment step then
            update ξ according to m[i]
            return Next(stack, ξ)
        if m[i] is an action a then
            trigger the execution of action a
            return stack
        if m[i] is a task τ′ then
4           observe current state ξ
5           m′ ← Guide(ξ, τ′, push((τ′, nil, 1, 0), stack), d_max, n_ro)
            if m′ = ∅ then return Retry(stack)
            else return push((τ′, m′, 1, ∅), stack)

Next(stack, ξ)
    repeat
        (τ, m, i, tried) ← top(stack)
        pop(stack)
1       if stack = ⟨⟩ then return ∅
    until i is not the last step of m
    j ← step following i in m depending on ξ
    return push((τ, m, j, tried), stack)
```



Planning for Rae?

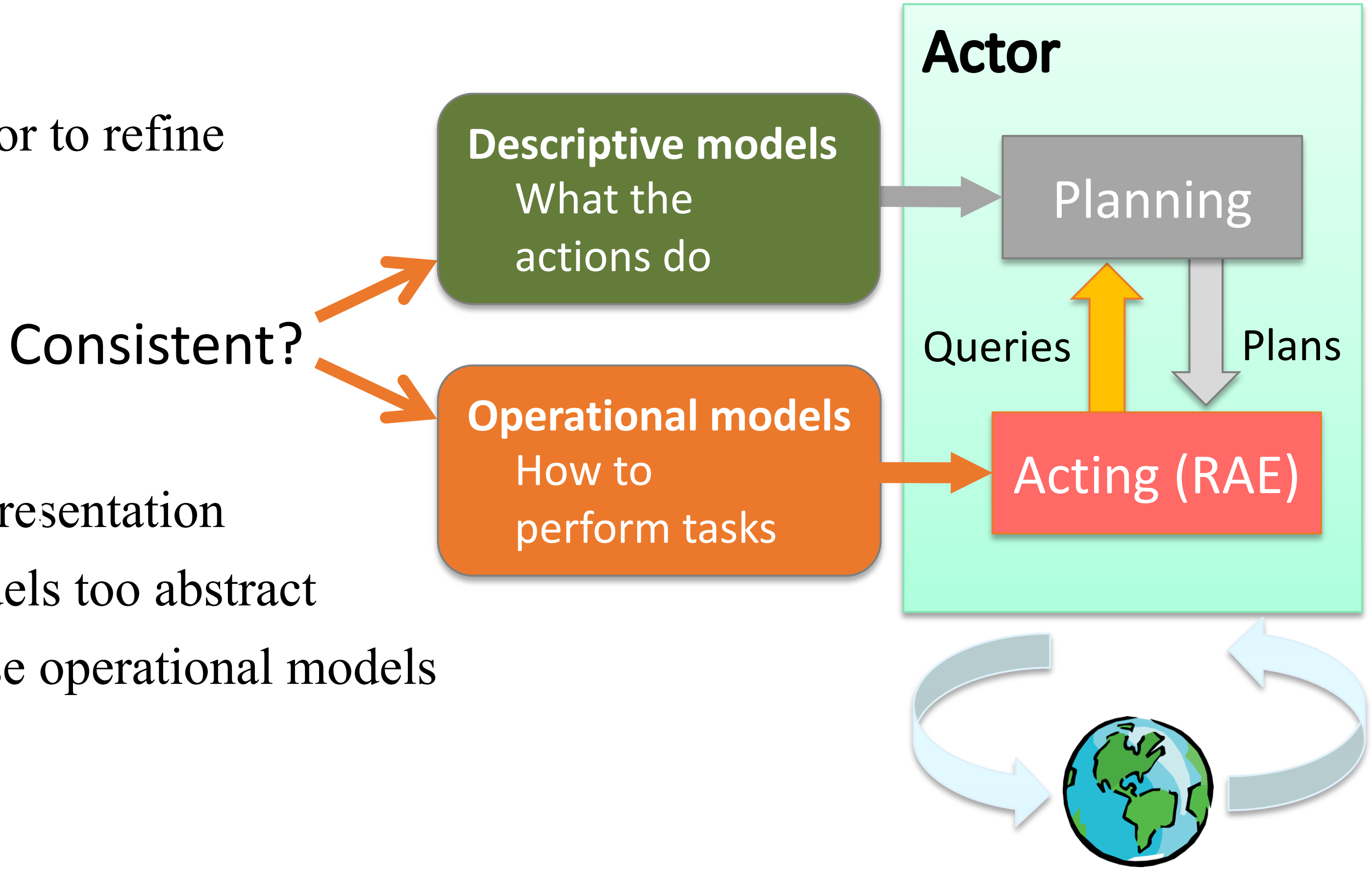
procedure RAE:
    loop:
        for every new external task or event τ do
            choose a method instance m for τ
            create a refinement stack for τ, m
            add the stack to *Agenda*
        for each stack σ in *Agenda*
          call Progress(σ)
          if σ is finished then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



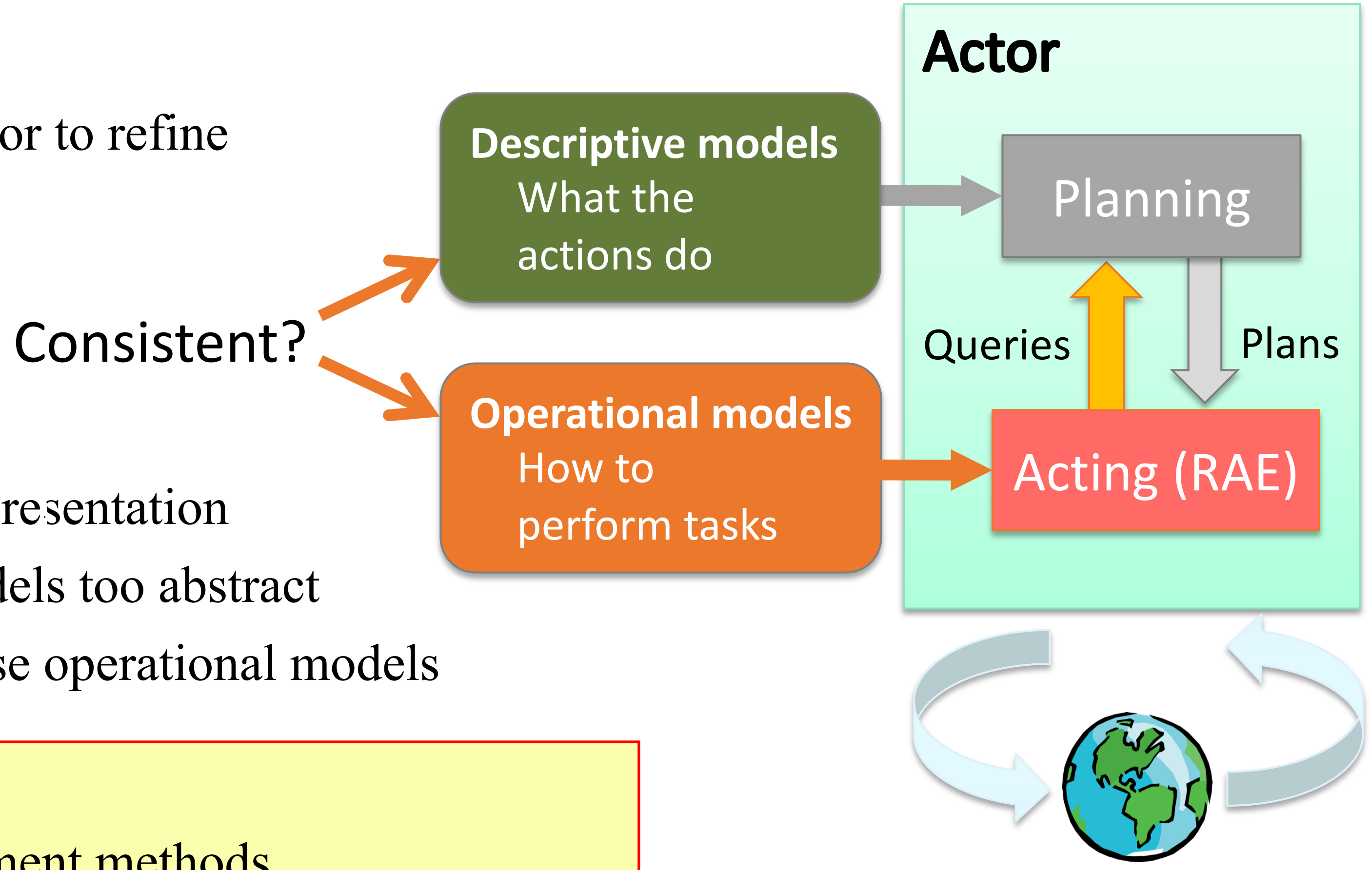
Planning and Acting Integration

- 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 actions, have a classical action model
  - DFS or GBFS search among alternatives to see which works best

SeRPE (Sequential Refinement Planning Engine)

Automated Planning and Acting
Ch. 3.3

$\mathcal{M}$ = {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 action, use a classical action model
- This has some problems …

Problems with SeRPE

Automated Planning and Acting
Ch. 3.3

Problem 1: difficult to implement

- Each time a method invokes a subtask, SeRPE makes a nondeterministic choice
- To implement deterministically
  - Each path in the search space is an execution trace of the body of a method
  - Need to backtrack over code execution

- Need to write a compiler that can do backtracking
  - Is it worth the effort?

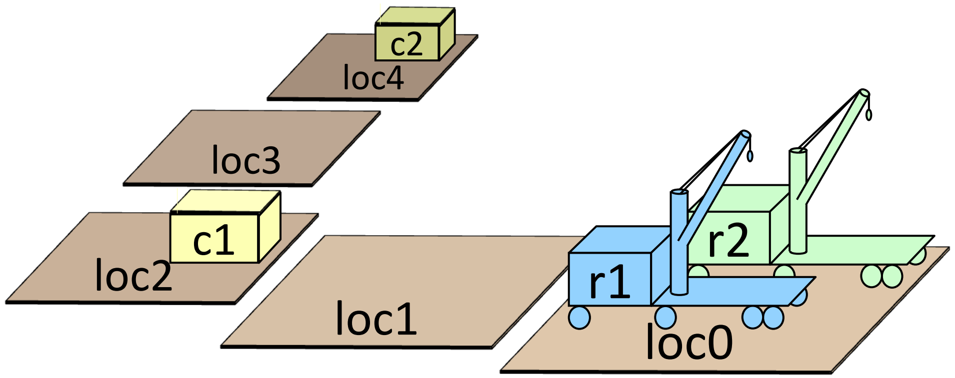
```
m-foo(k)
    task:   foo(k)
    pre:    …
    body:
        for i ← 1 to k:
            bar(i)
            baz(i)
```

Example:

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

Problems with SeRPE

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



take(r,o,l)

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

put(r,o,l)

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

perceive(r,l):

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

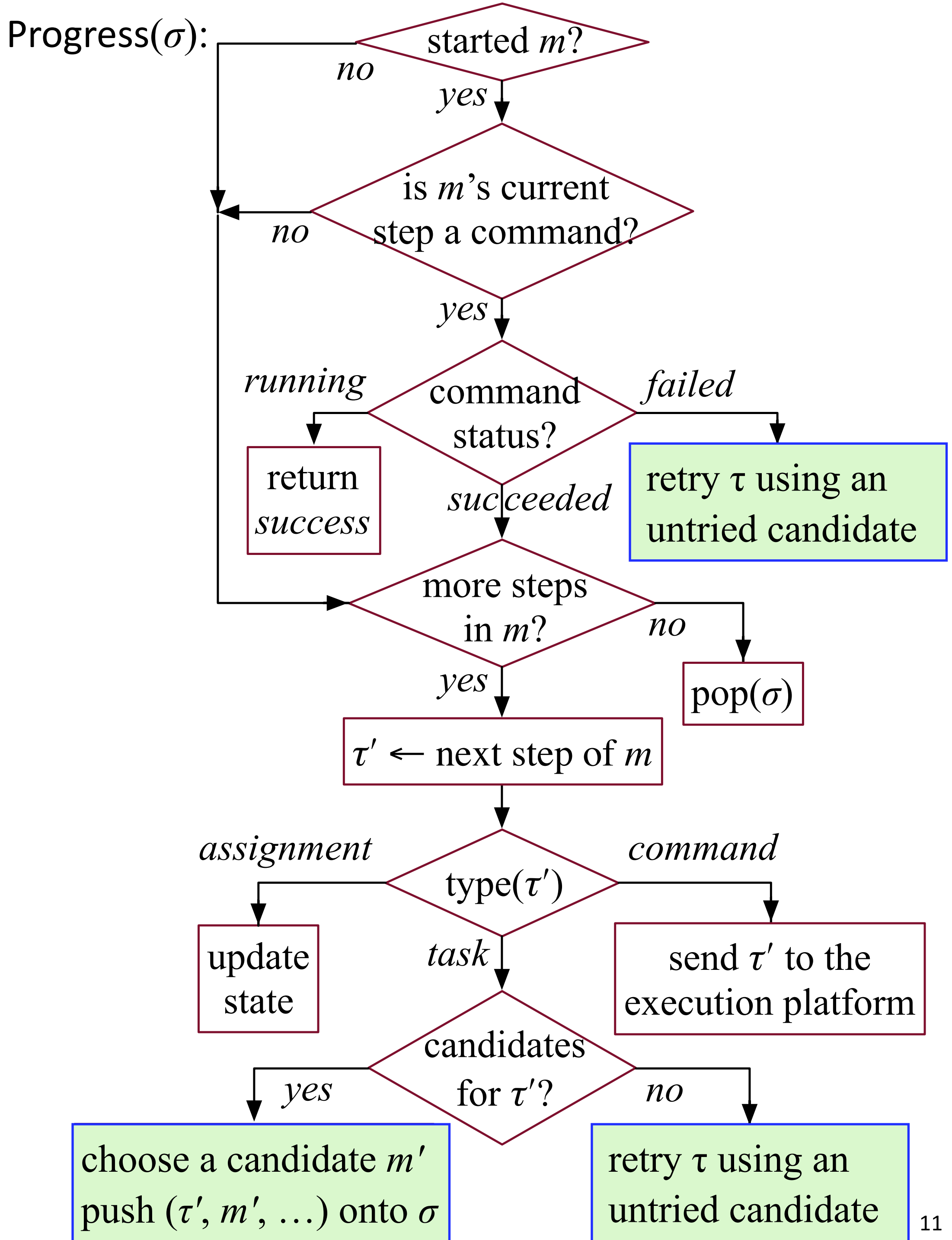
Planning for Rae

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

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

Poll: is this a reasonable approach?

A) Yes   B) No   C) It depends

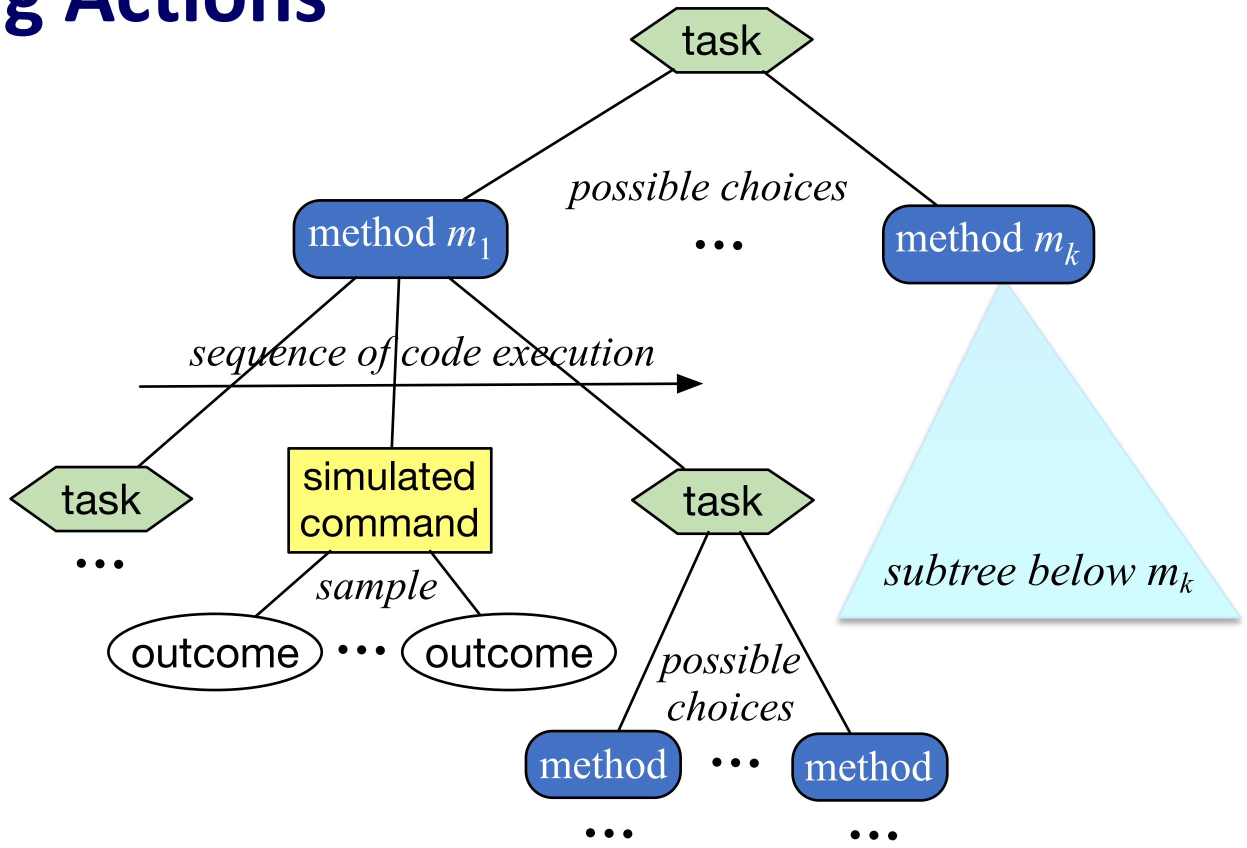


Simulating Actions

- Simplest case:
  - probabilistic action template

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

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



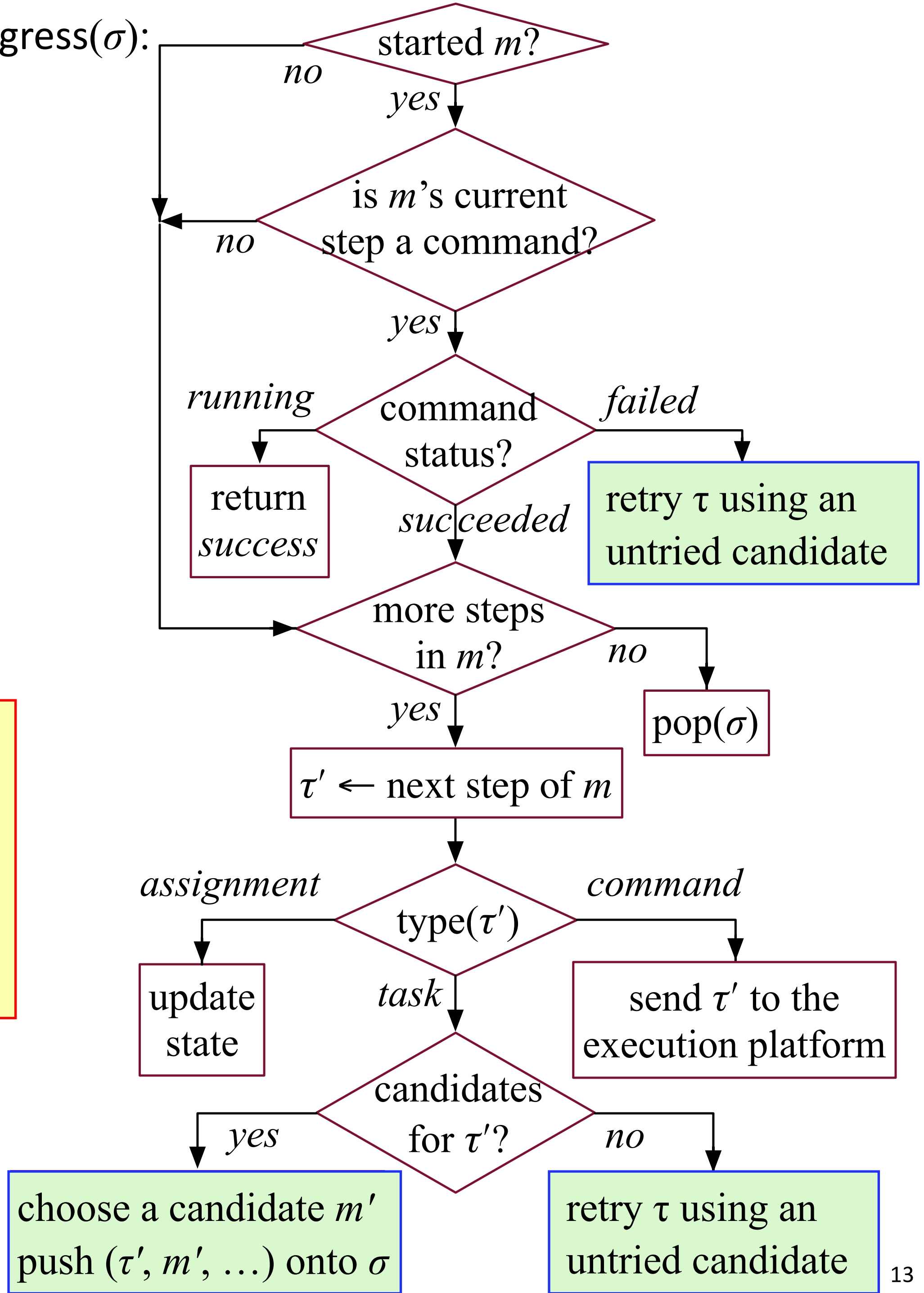
- More general:
  - Arbitrary computation, e.g., physics-based simulation
  - Run the code to get simulated effects

Planning for Rae

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

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

Progress(σ):



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

Patra, Mason, Kumar, Ghallab, Nau, and Traverso. Deliberative acting, planning and learning with hierarchical operational models. Art. Intel. Journal. Vol. 299, 2021. https://doi.org/10.1016/j.artint.2021.103523

Planner

Plan-with-UPOM (task τ):

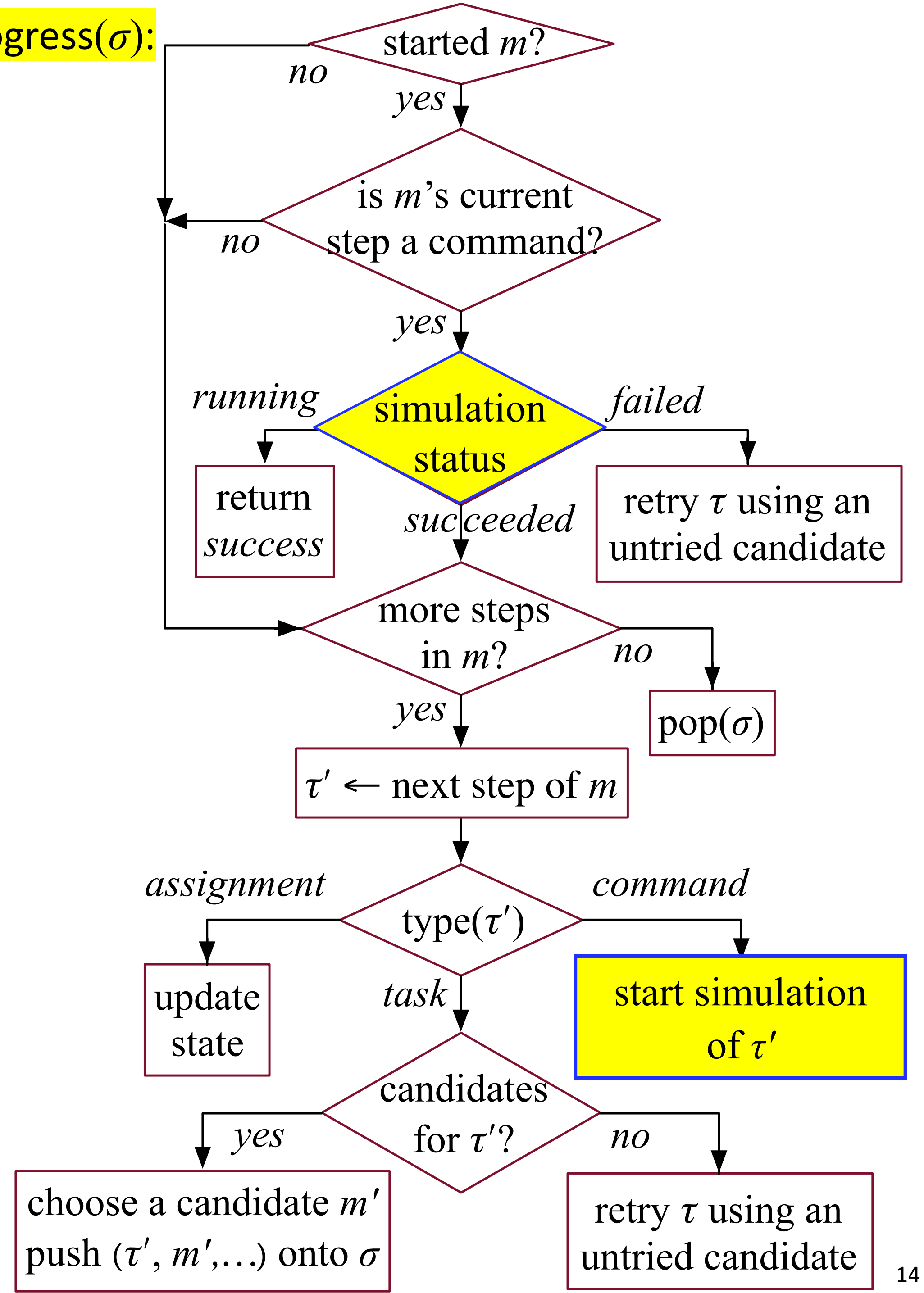
    Candidates ← {method instances relevant for τ}
    for i ← 1 to n
        call UPOM(τ)
        update estimates of methods' expected utility
    return the $m \in$ *Candidates* that has
        the highest estimated utility

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

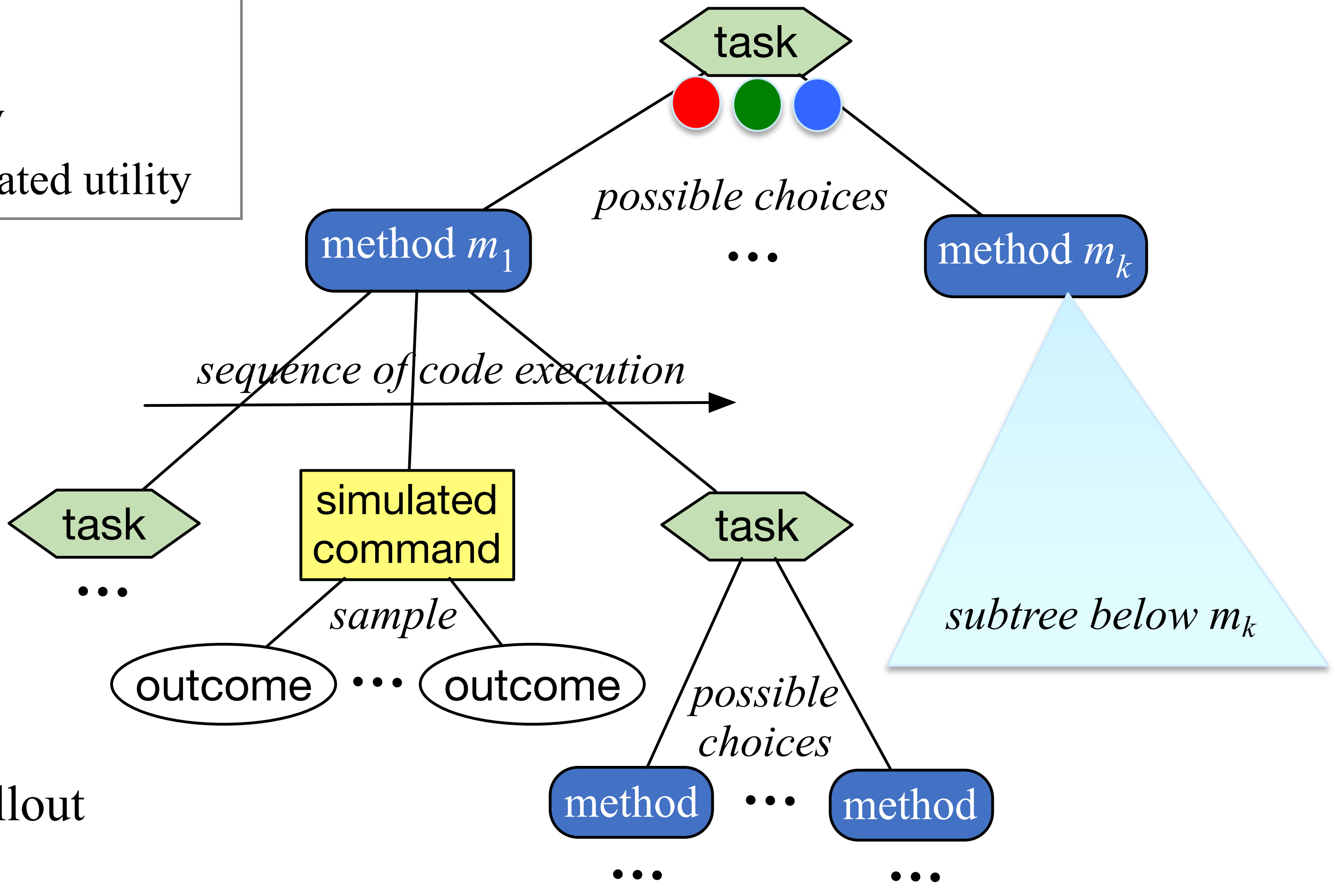
Simulate-Progress(σ):



Monte-Carlo rollouts

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

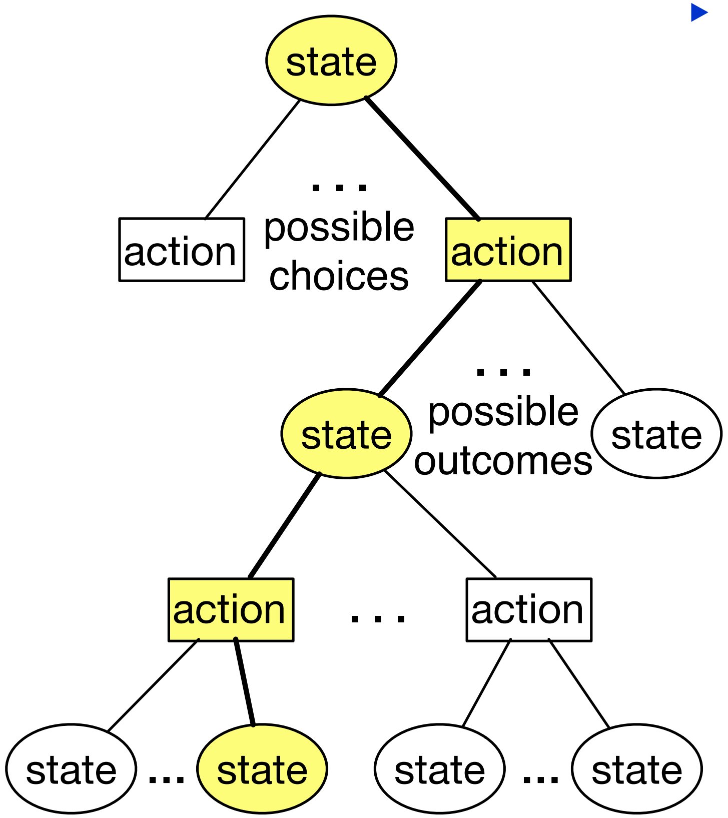
UPOM(τ):
    choose a method instance m for τ
    create refinement stack σ for τ and m
    loop while Simulate-Progress(σ) $\neq$ *failure*
        if σ is completed then return (m, *utility*)
    return *failure*



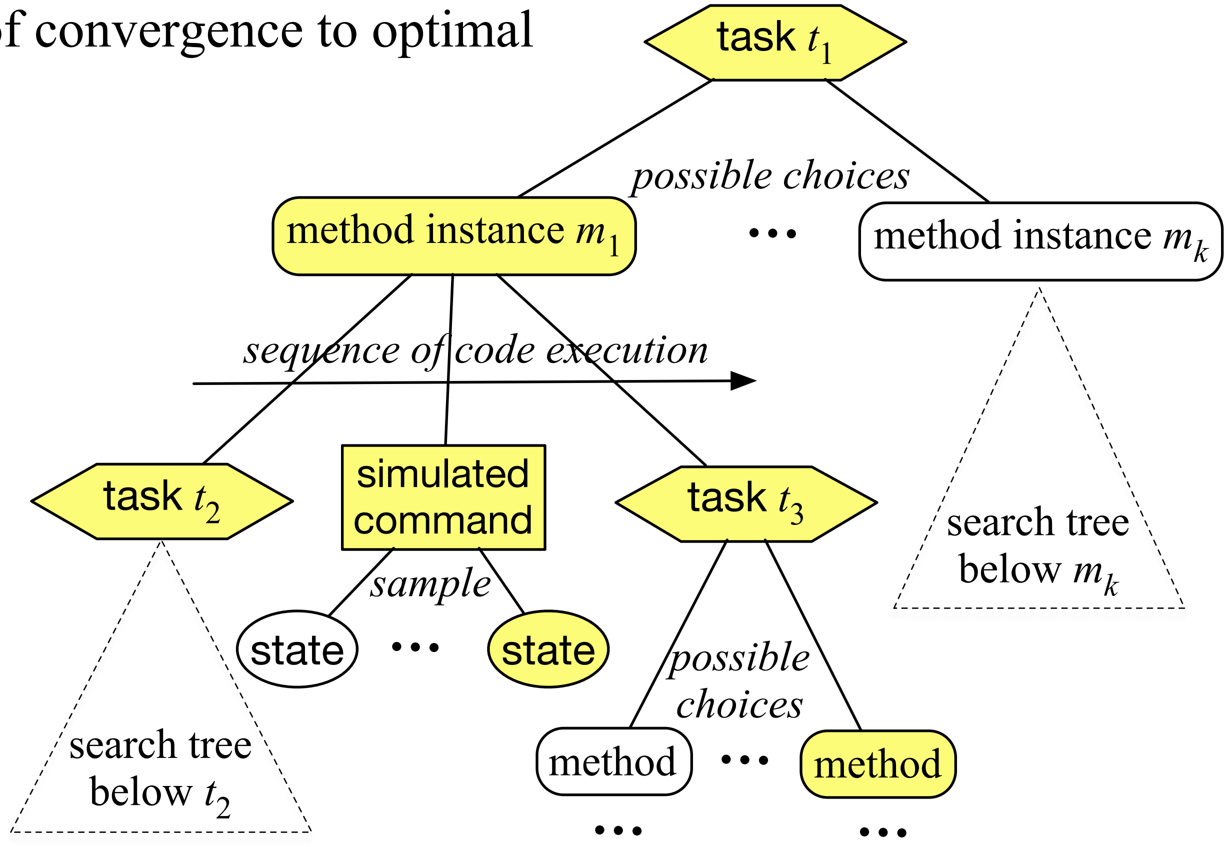
- Each call to UPOM does a Monte Carlo rollout
  - Simulated execution of RAE on τ

UCT and UPOM

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

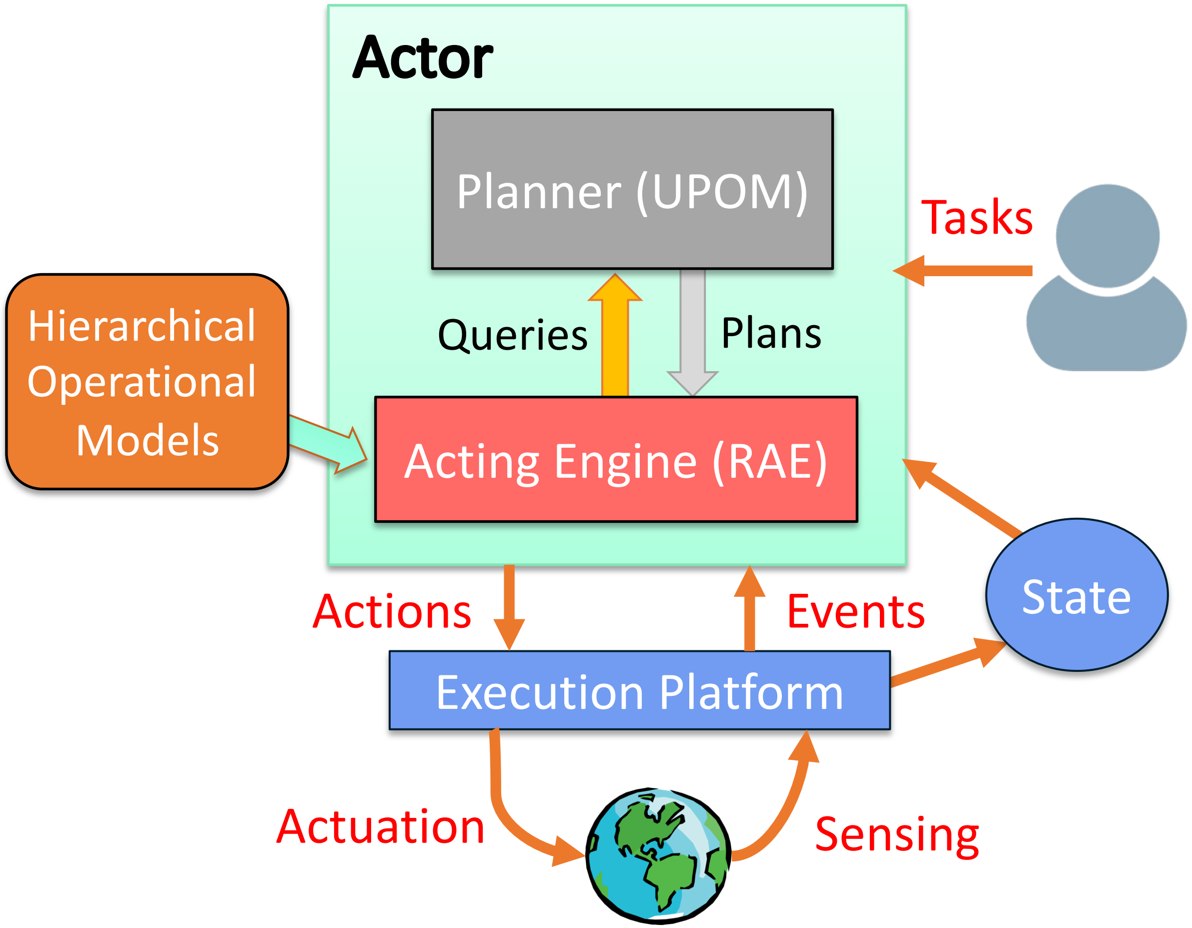


- UPOM search tree more complicated
  - tasks, method instances, actions, code execution
- If no exogenous events,
  - Can map it to UCT search of a complicated MDP
  - Proof of convergence to optimal



Outline

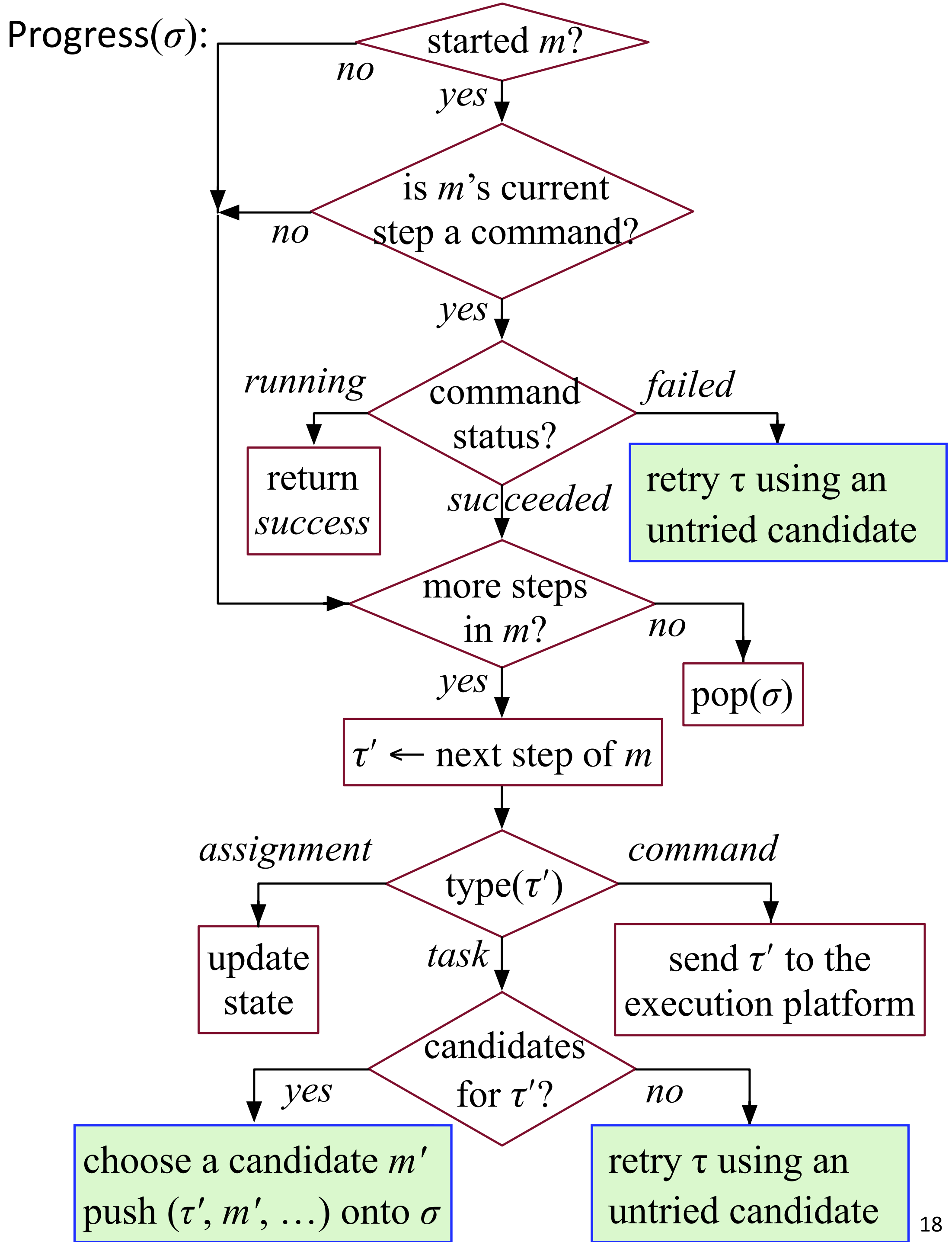
1. Planning for Rae
2. *Acting with Planning* (RAE+UPOM)
3. Learning
4. Evaluation, Application



RAE + UPOM

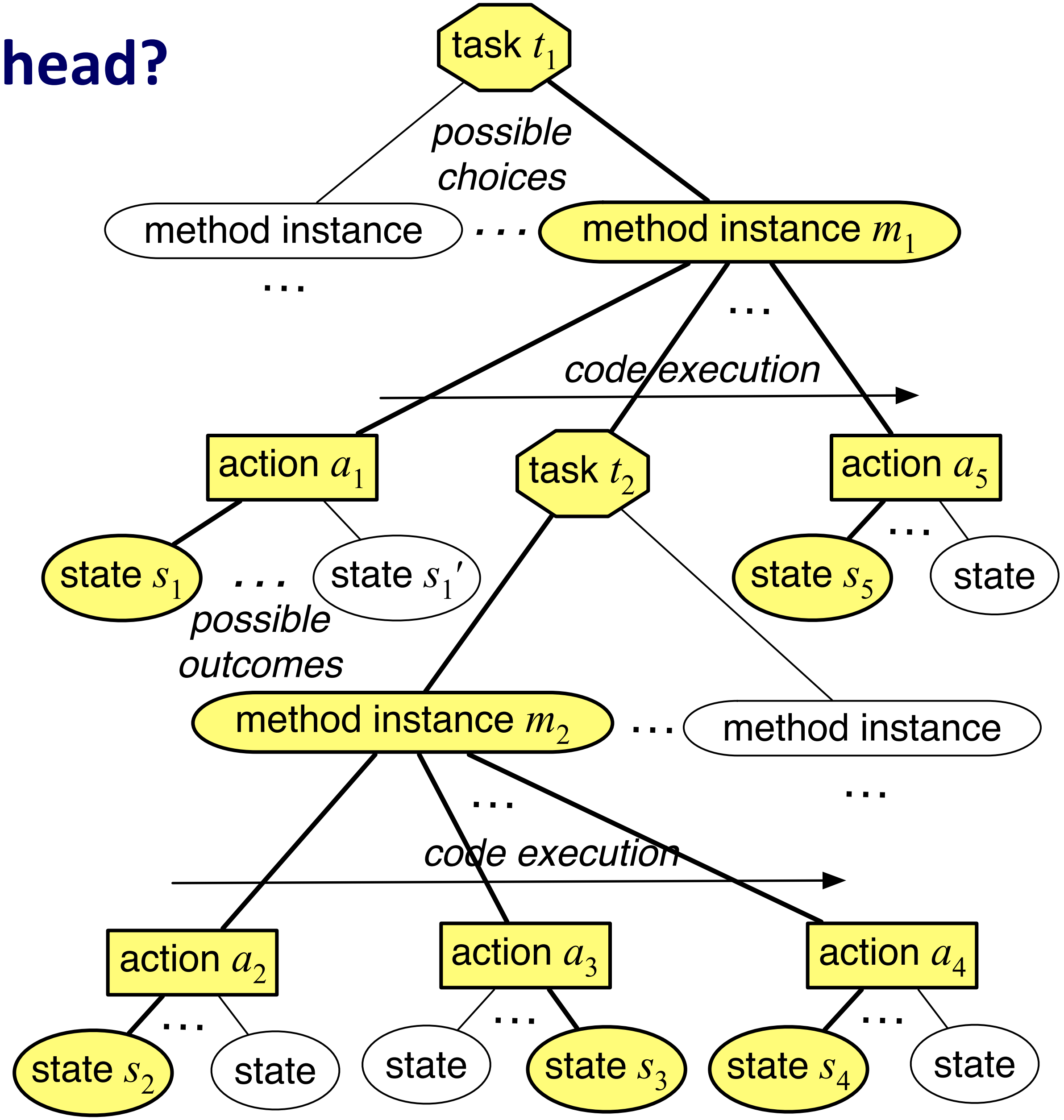
procedure RAE:
  loop:
    for every new external task or event τ do
      choose a method instance *m* for τ
      create a refinement stack for τ, *m*
      add the stack to *Agenda*
    for each stack σ in *Agenda*
      call Progress(σ)
      if σ is finished then remove it

- 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/



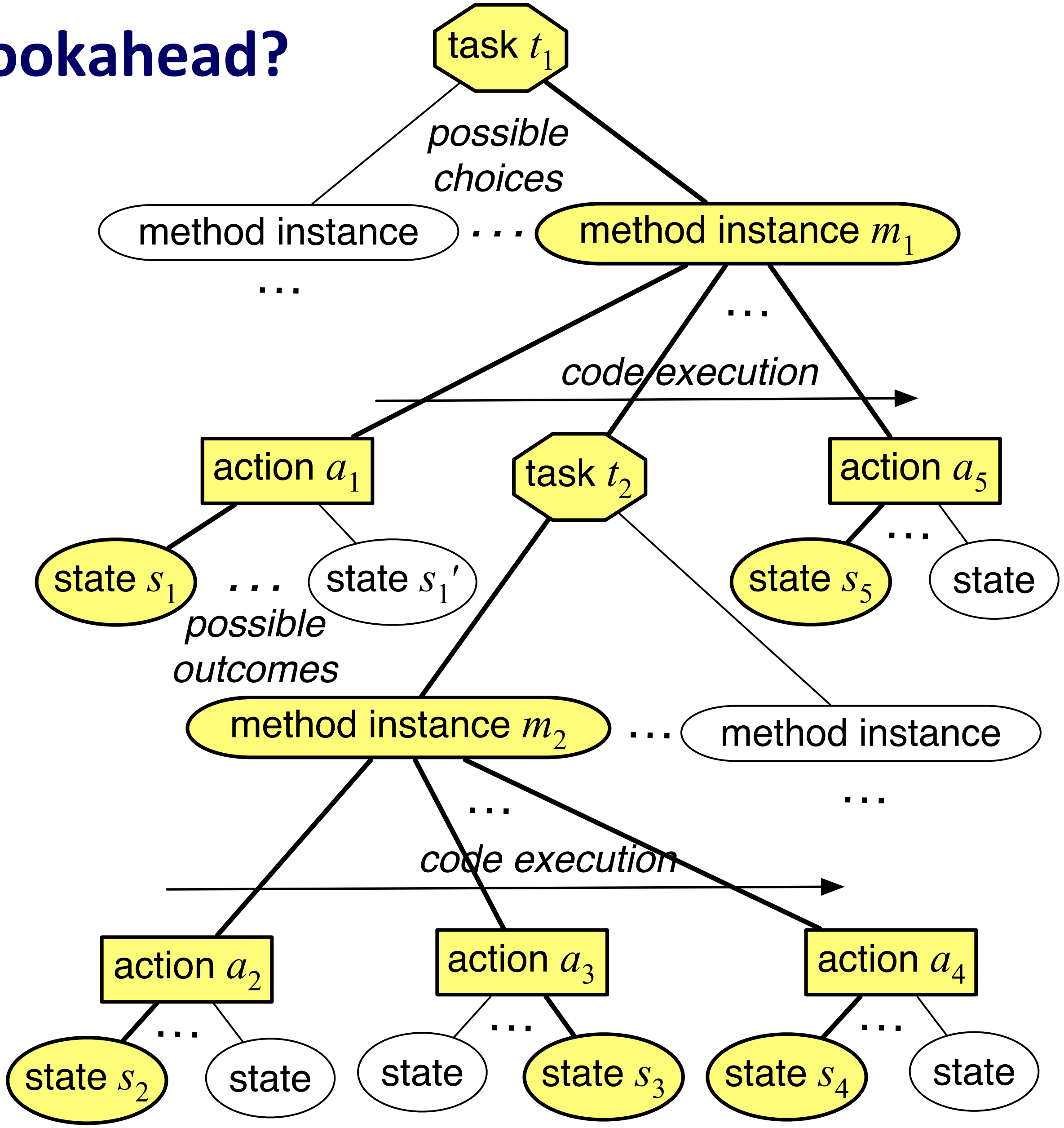
Could we use UPOM with HTN-Run-Lookahead?

- 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
    - $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
- Problem
  - Run-lookahead calls UPOM′, gets π, executes a_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



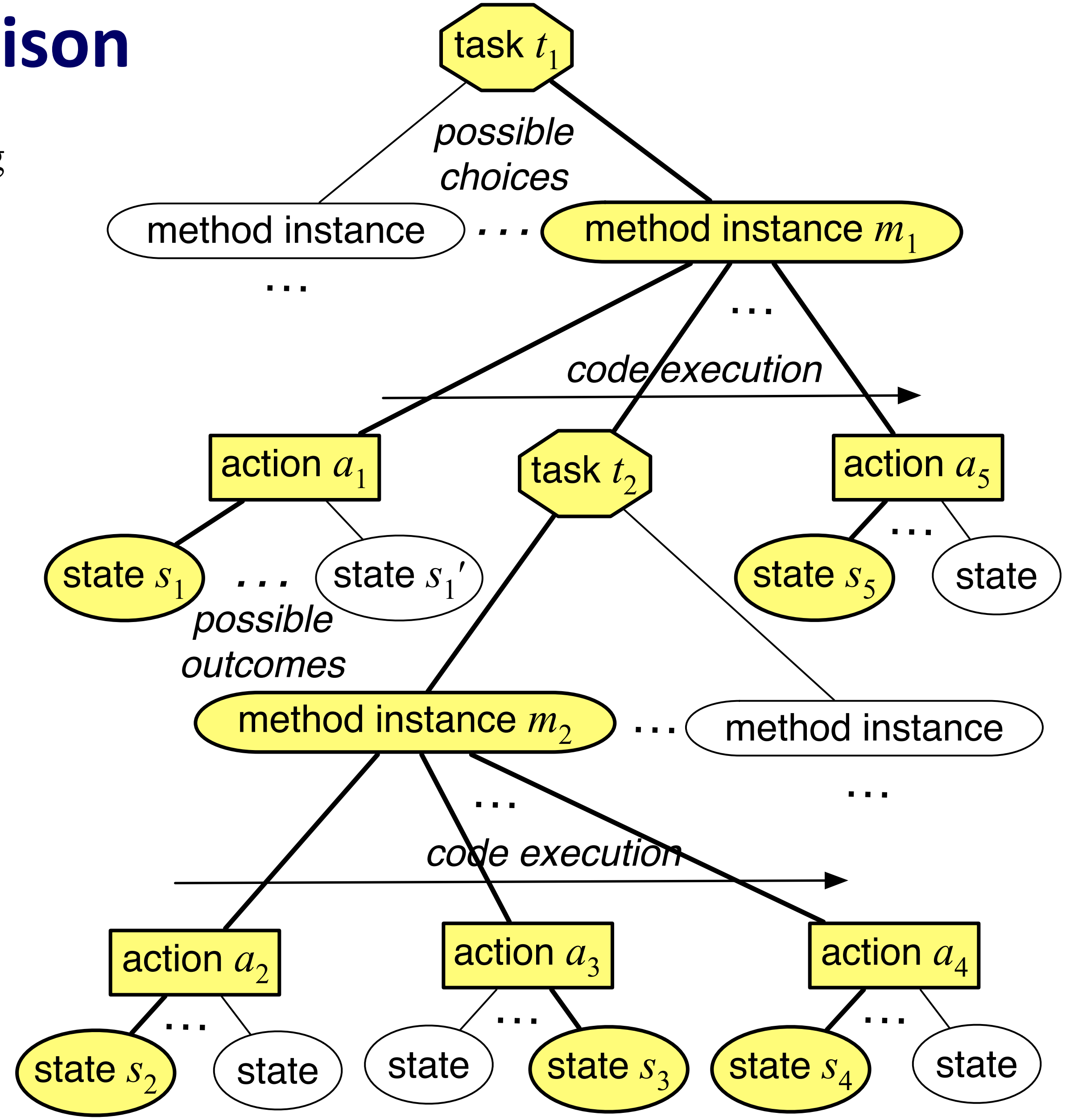
Could we use UPOM with HTN-Run-Lazy-Lookahead?

- Run-Lazy-Lookahead calls UPOM′,
  UPOM′ returns $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
- Run-Lazy-Lookahead executes a_1, a_2, a_3, a_4, a_5,
  won't call UPOM′ again unless
  something unexpected happens, e.g.,
  - action a_2 has an execution failure
  - a_2 produces a state in which a_3 is inapplicable
  - an exogenous event makes a_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



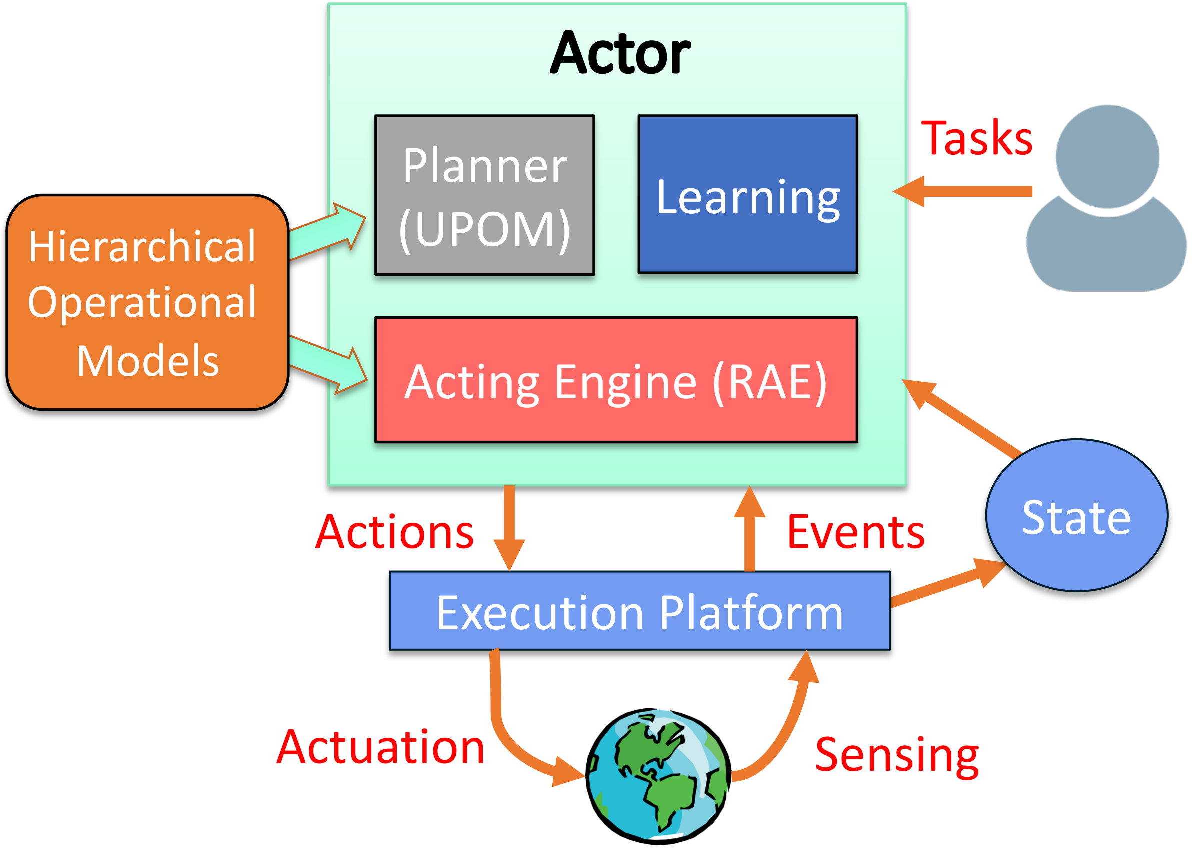
Comparison

- 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 $\pi = \langle a_1, a_2, a_3, a_4, a_5 \rangle$
    - Run-Lazy-Lookahead executes a_1, gets state $s_1{}'$ (not s_1)
      - Suppose this makes 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 a_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



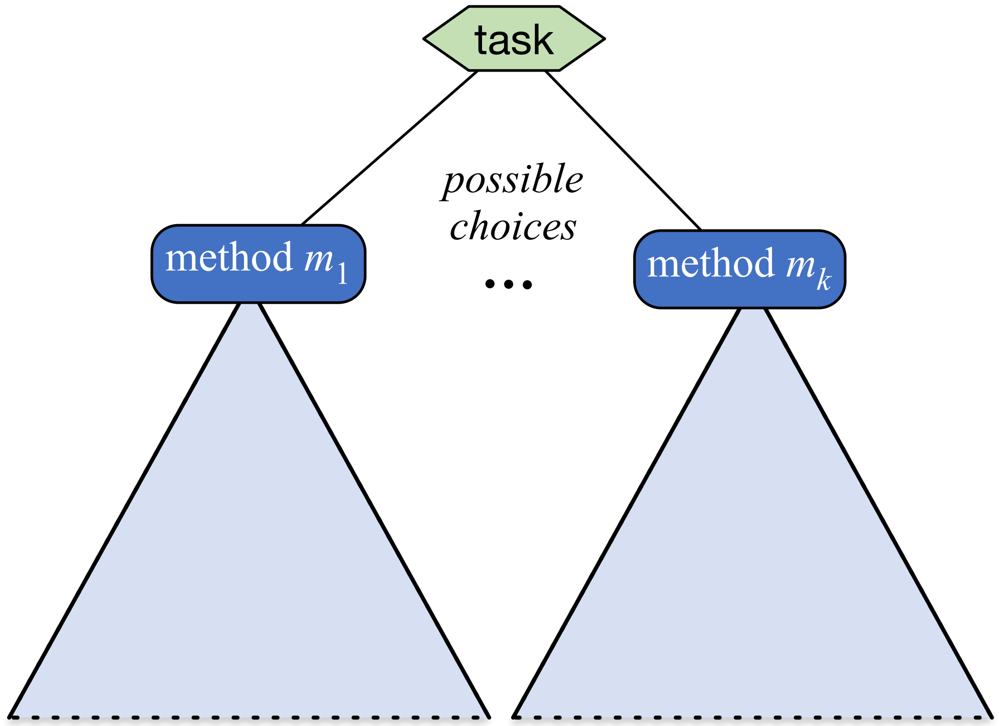
Outline

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



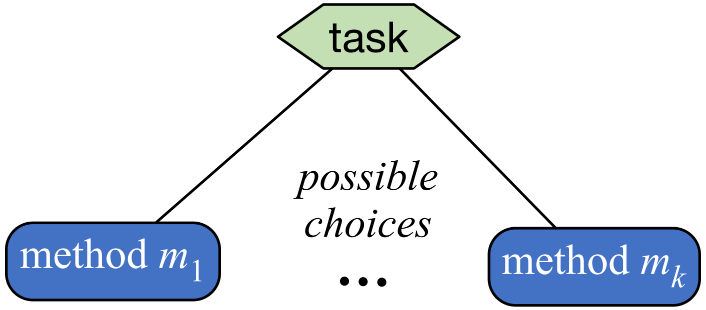
Motivation

- 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

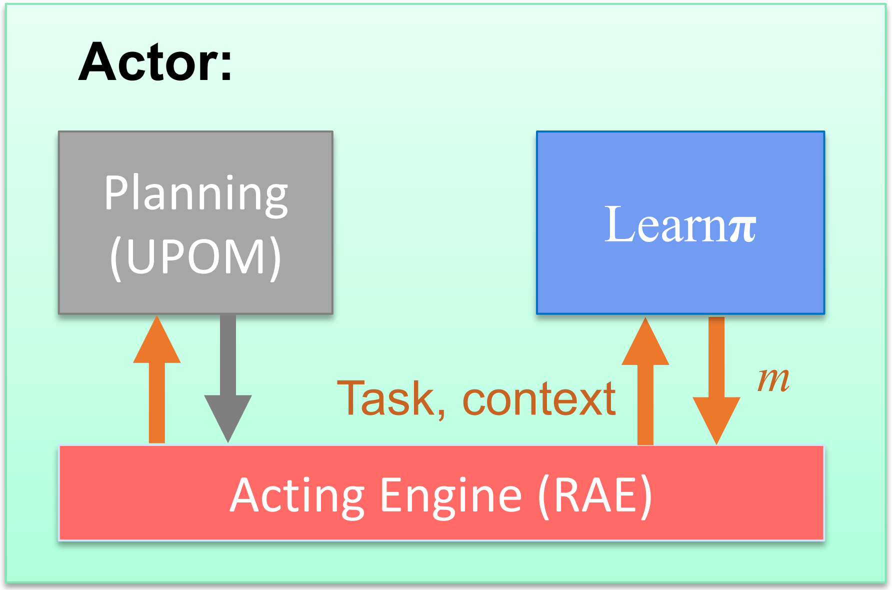


Integration with Learning

- Gather training data from acting-and-planning traces of RAE and Plan-with-UPOM
- Train classifiers (feed-forward neural nets)

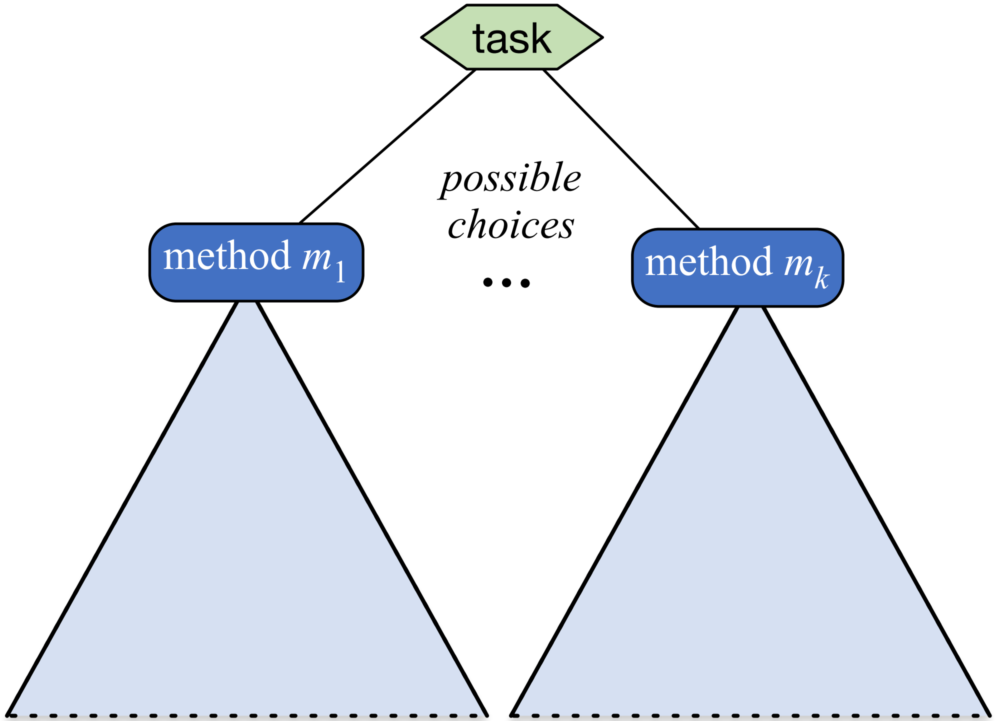


- Learnπ
  - Learns function for choosing a method
  - Given current task and context (state and other information), choose m from the set of available refinement methods
  - Useful if there isn't enough time to use UPOM

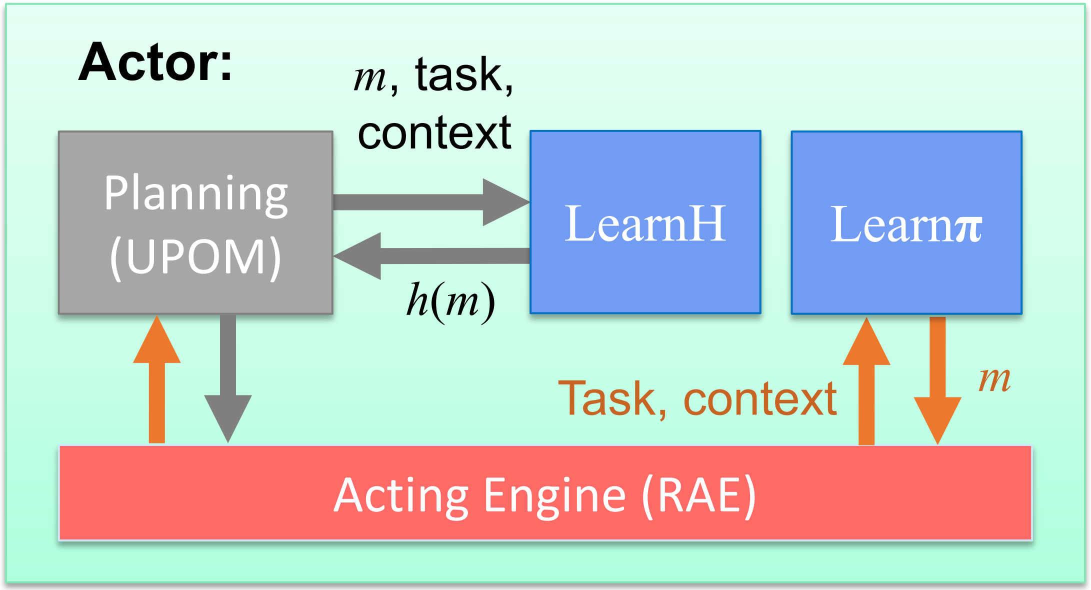


Integration with Learning

- Gather training data from acting-and-planning traces of RAE and Plan-with-UPOM
- Train classifiers (feed-forward neural nets)

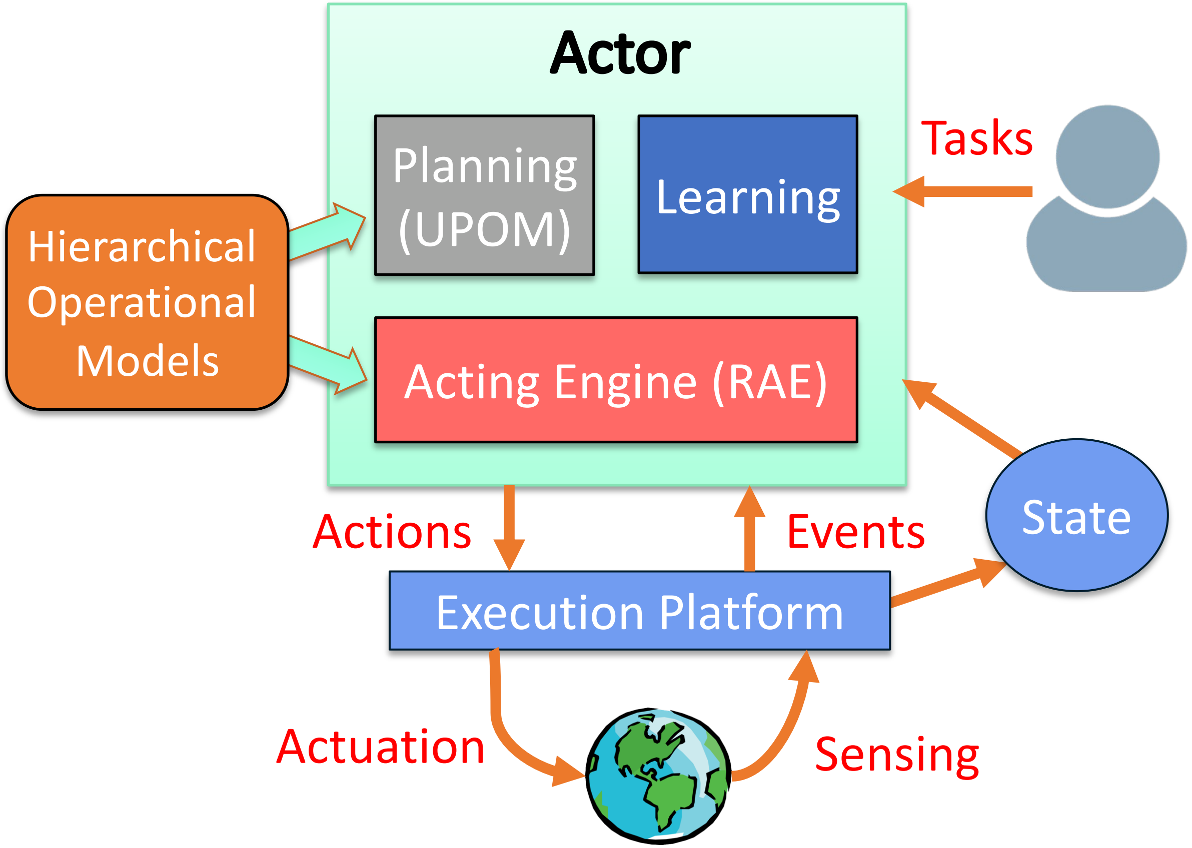


- LearnH
  - Learns a heuristic function to guide UPOM's search
  - UPOM can use it to estimate expected utility at leaf nodes
  - Useful if there isn't enough time to search all the way to the end



Outline

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



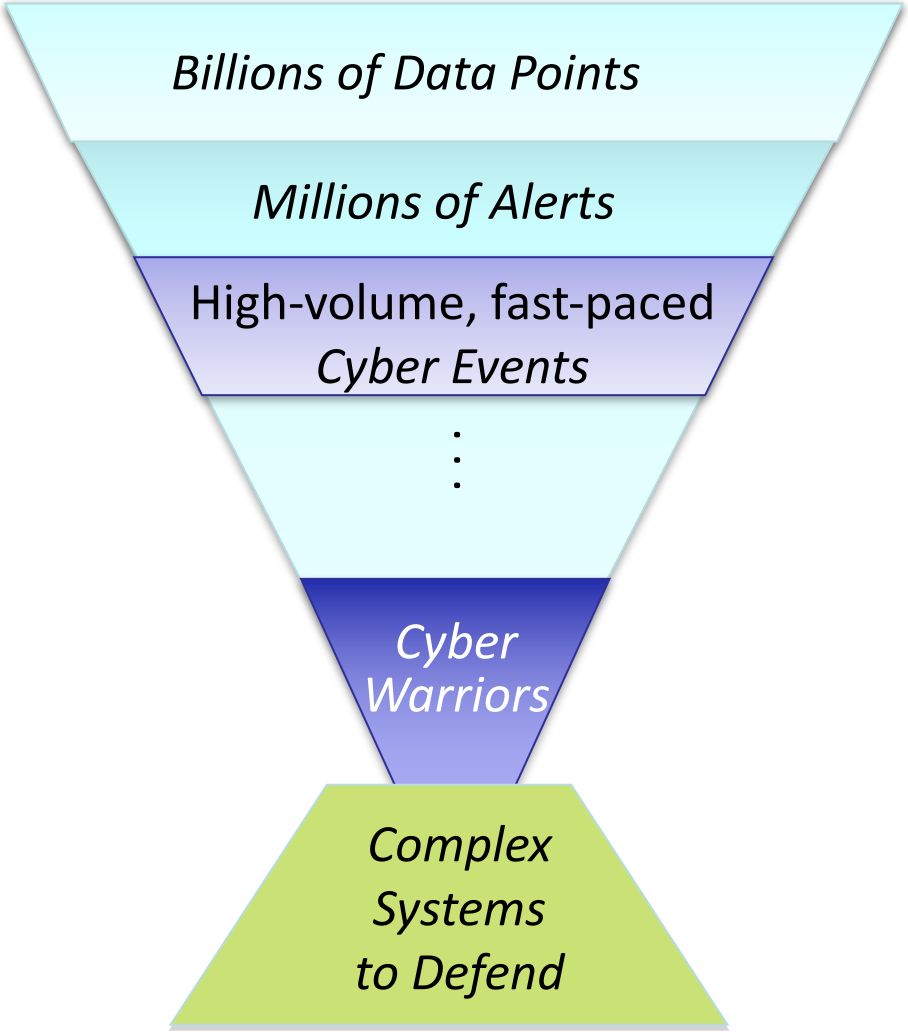
Experimental Evaluation

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

Summary

Chapter 15: Hierarchical Refinement Planning

- Plan by simulating Rae on a single external task/event/goal
  - SeRPE uses classical action models
  - UPOM simulates the actor's actions, does Monte Carlo rollouts
- Acting and planning
  - Rae + UPOM
  - Comparison: Run-Lazy-Lookahead + UPOM′
  - Open-source Python implementation:
    - https://bitbucket.org/sunandita/RAE/

- Chapter 16: Learning
  - Learning a function to choose a method
  - Learning heuristics to guide search

- Additional material not in the book
  - Experimental evaluation
  - Prototype application