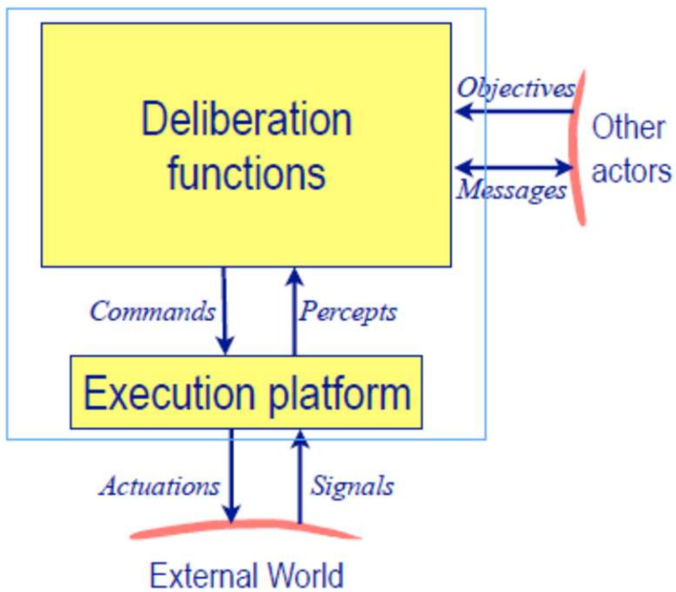
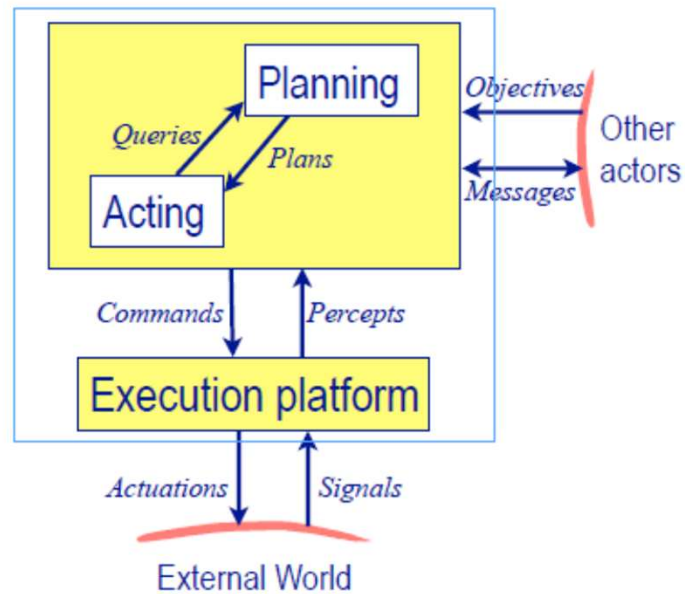
Planning and Acting

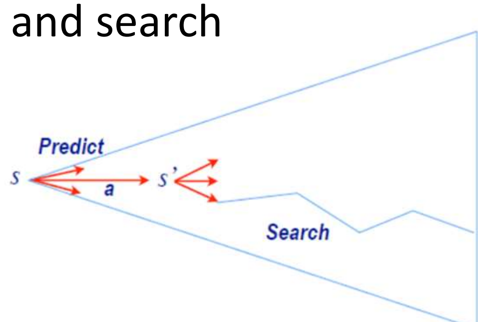


(a)

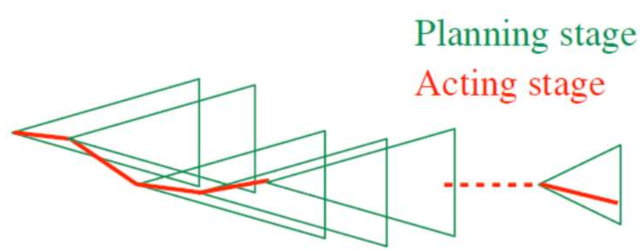


(b)

Prediction and search

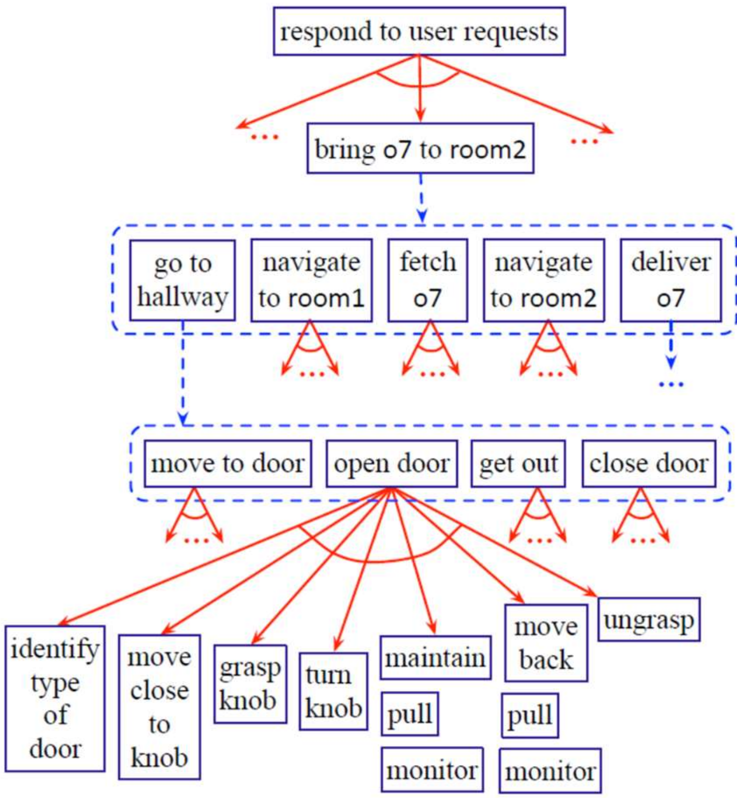


Receding horizon



Planning and Acting

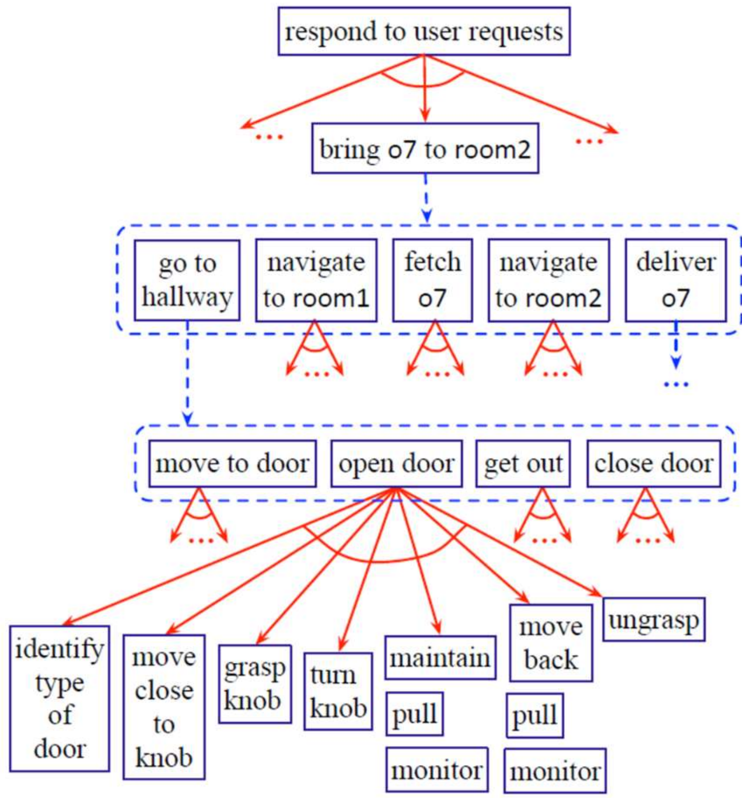
Multiple levels of deliberation and representation



- Hierarchically organized deliberation
- Continual online deliberation

Planning and Acting

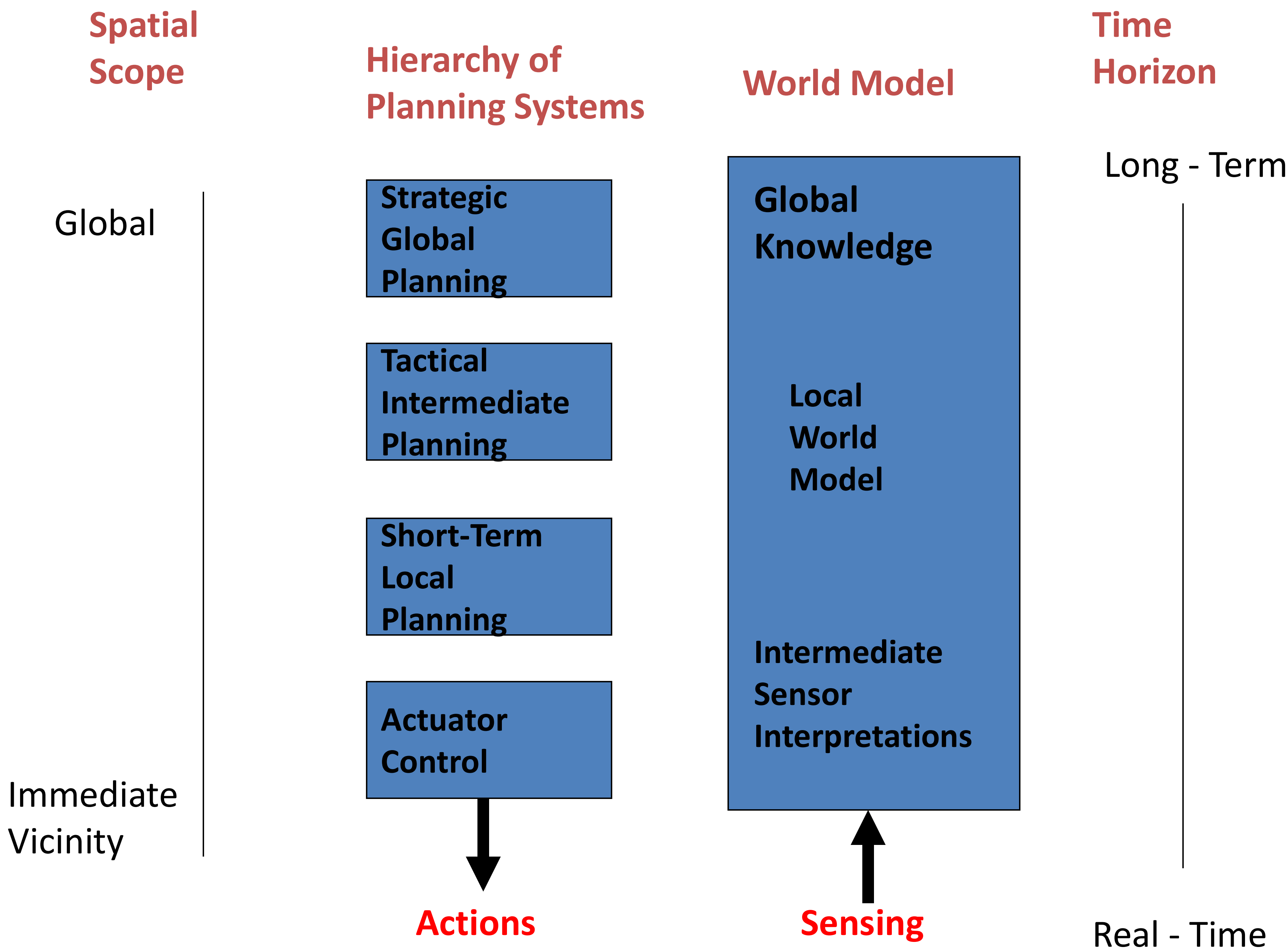
Multiple levels of deliberation and representation



- Hierarchically organized deliberation
- Continual online deliberation

Planner Hierarchy

- Hierarchical planning systems typically share a structured and clearly identifiable subdivision of functionality regarding **distinct program modules** that communicate with each other in a **predictable and predetermined manner**.
- At a hierarchical planner's highest level, the most global and least specific plan is formulated (deliberative planner).
- At the lowest levels, **rapid real-time** response is required, but **the planner is concerned only** with its immediate surroundings and has lost the sight of the big picture.


Spatial Scope
Hierarchy of Planning Systems
World Model
Time Horizon
Global
Immediate Vicinity
Strategic Global Planning
Tactical Intermediate Planning
Short-Term Local Planning
Actuator Control
Actions
Global Knowledge
Local World Model
Intermediate Sensor Interpretations
Sensing
Long - Term
Real - Time

Hierarchical Planners vs. BBS

Hierarchical Planners

- Rely heavily on world models,
- Can readily integrate world knowledge,
- Have a broad perspective and scope.

BB Control Systems

- afford modular development,
- Real-time robust performance within a changing world,
- Incremental growth
- are tightly coupled with arriving sensory data.

Hybrid Control

- **The basic idea is simple**: we want the best of both worlds (if possible).
- The goal is to **combine** closed-loop and open-loop **execution.**
- That means to **combine reactive and deliberative control.**
- This implies combining the **different time-scales and representations.**
- This mix is called hybrid control.

Hybrid robotic architectures believe that a union of deliberative and behavior-based approaches can potentially yield the best of both worlds.

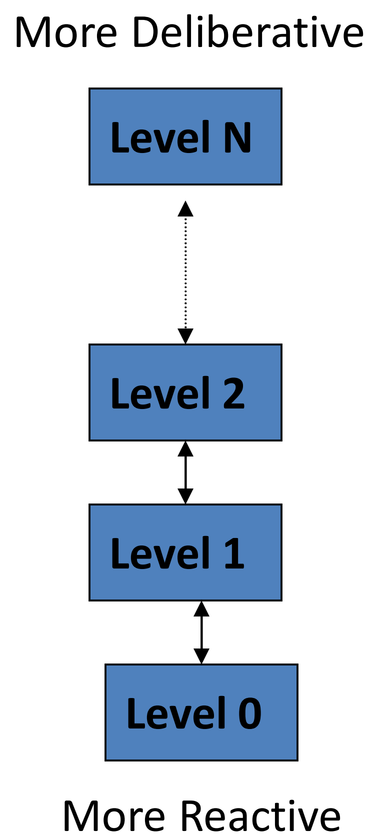
Organizing Hybrid Systems

Planning and reaction can be tied:

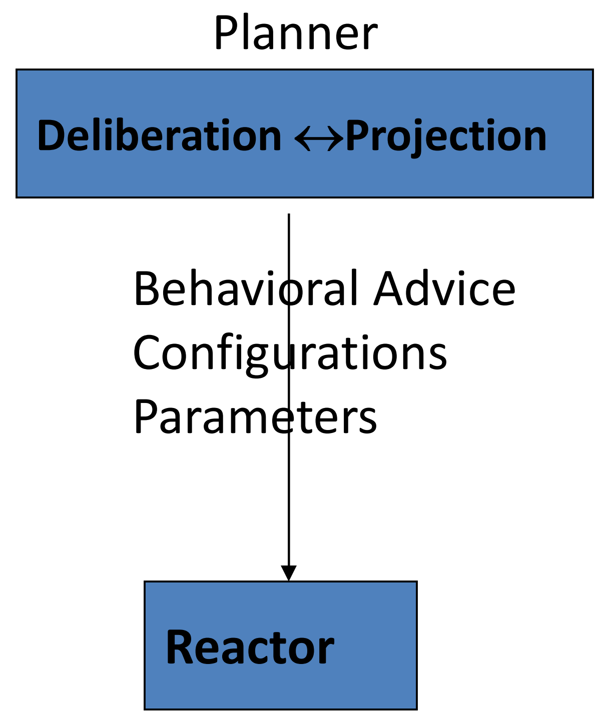
A: hierarchical integration - planning and reaction are involved with different activities, time scales

B: Planning to guide reaction - configure and set parameters for the reactive control system.

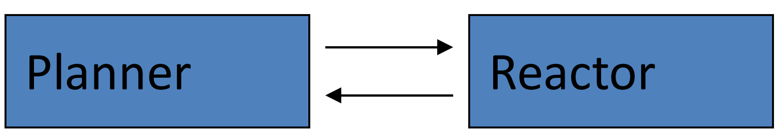
C: coupled - concurrent activities



A



B



C

Organizing Hybrid Systems

It was observed that the emerging architectural design of choice is:

– multi-layered hybrid comprising of

* a **top-down** planning system and
* a **lower-level** reactive system.

– **the interface** (middle layer between the two components) **design** is a central issue in differentiating different hybrid architectures.

In summary, a modern hybrid system typically consists of three components:

♦ **a reactive layer**

♦ **a planner**

♦ **a layer that puts the two together.**

=> Hybrid architectures are often called **three-layer architectures**.

The Magic Middle: Executive Control

- The middle layer has a hard job:
  1) compensate for the limitations of both the planner and the reactive system
  2) reconcile their different time-scales.
  3) deal with their different representations.
  4) reconcile any contradictory **commands** between the two.
- This is **the challenge** of hybrid systems

  => **achieving the right compromise between the two ends.**

AI Planning Paradigms

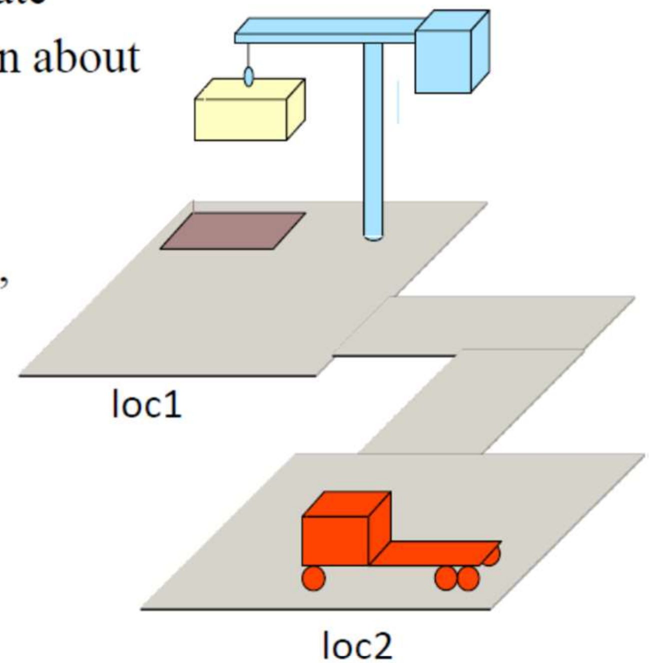
- Classical Planning
- Temporal Planning
- Conditional Planning
- Decision Theoretic Planning
- …
- Least-Commitment Planning
- HTN planning
- …

Three Main Types of Planners

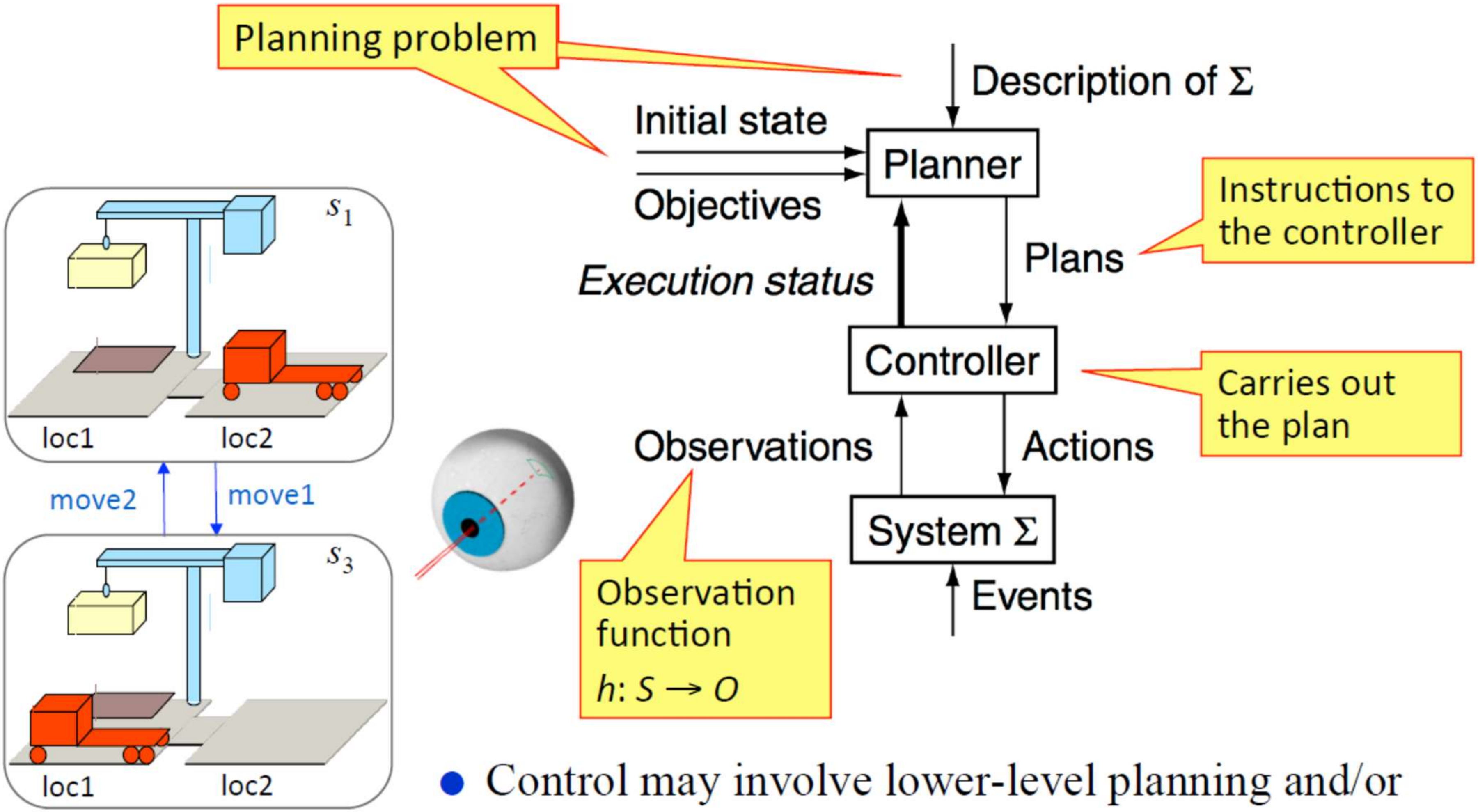
1. Domain-specific
   - Made or tuned for a specific planning domain
   - Won't work well (if at all) in other planning domains
2. Domain-independent
   - In principle, works in any planning domain
   - In practice, need restrictions on what kind of planning domain
3. Configurable
   - Domain-independent planning engine
   - Input includes info about how to solve problems in some domain

Abstraction

- Real world is absurdly complex, need to approximate
  - Only represent what the planner needs to reason about
- **State transition system** $\Sigma = (S, A, E, \gamma)$
  - S = {abstract states}
    - e.g., states might include a robot's location, but not its position and orientation
  - A = {abstract actions}
    - e.g., "move robot from loc2 to loc1" may need complex lower-level implementation
  - E = {abstract exogenous events}
    - Not under the agent's control
  - γ = state transition function
    - Gives the next state, or possible next states, after an action or event
    - $\gamma: S \times (A \cup E) \rightarrow S$ or $\gamma: S \times (A \cup E) \rightarrow 2^S$
- In some cases, avoid ambiguity by writing S_Σ, A_Σ, E_Σ, γ_Σ



Conceptual Model



- Control may involve lower-level planning and/or plan execution
  - e.g., how to move from one location to another

Plans

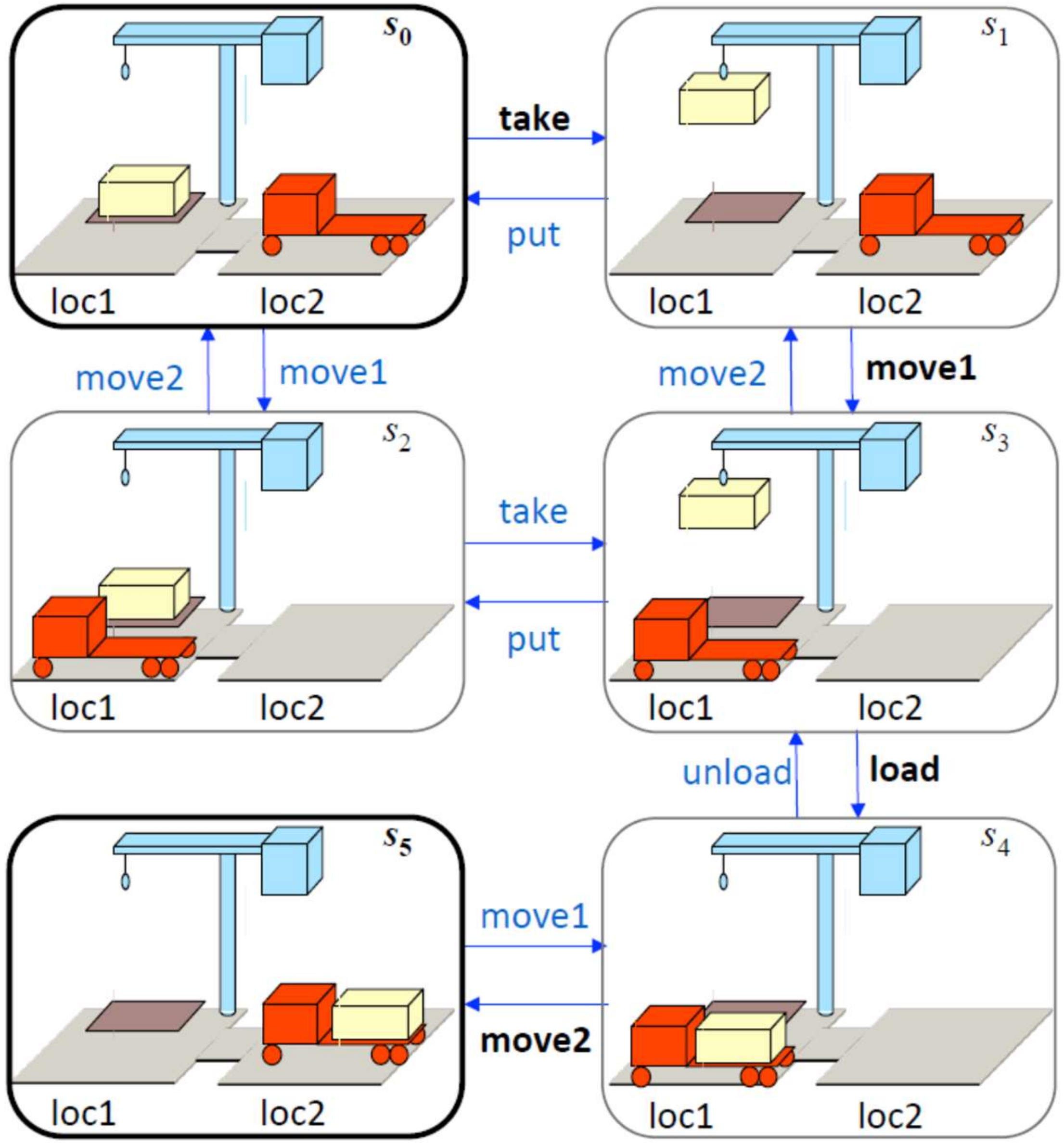
- **Classical plan**: a sequence of actions

  ⟨take, move1, load, move2⟩

- **Policy**: partial function from S into A

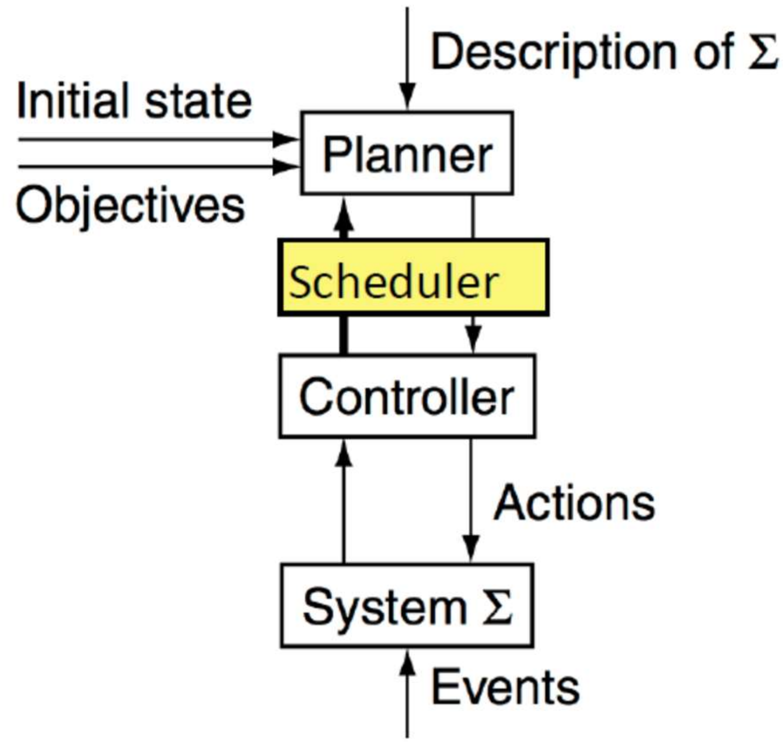
  $\{(s_0$, take),
  $(s_1$, move1),
  $(s_3$, load),
  $(s_4$, move2)$\}$

- Both, if executed starting at s_0, produce s_3



Dock Worker Robots (DWR) example

Planning Versus Scheduling



- Scheduling
  - Decide when and how to perform a given set of actions
    - Time constraints
    - Resource constraints
    - Objective functions
  - Typically NP-complete
- Planning
  - Decide what actions to use to achieve some set of objectives
  - Can be much worse than NP-complete; worst case is undecidable
- Scheduling problems may require replanning

Restrictive Assumptions

A0: Finite system:

- finitely many states, actions, events

A1: Fully observable:

- the controller always Σ's current state

A2: Deterministic:

- each action has only one outcome

A3: Static (no exogenous events):

- no changes but the controller's actions

A4: Attainment goals:

- a set of goal states S_g

A5: Sequential plans:

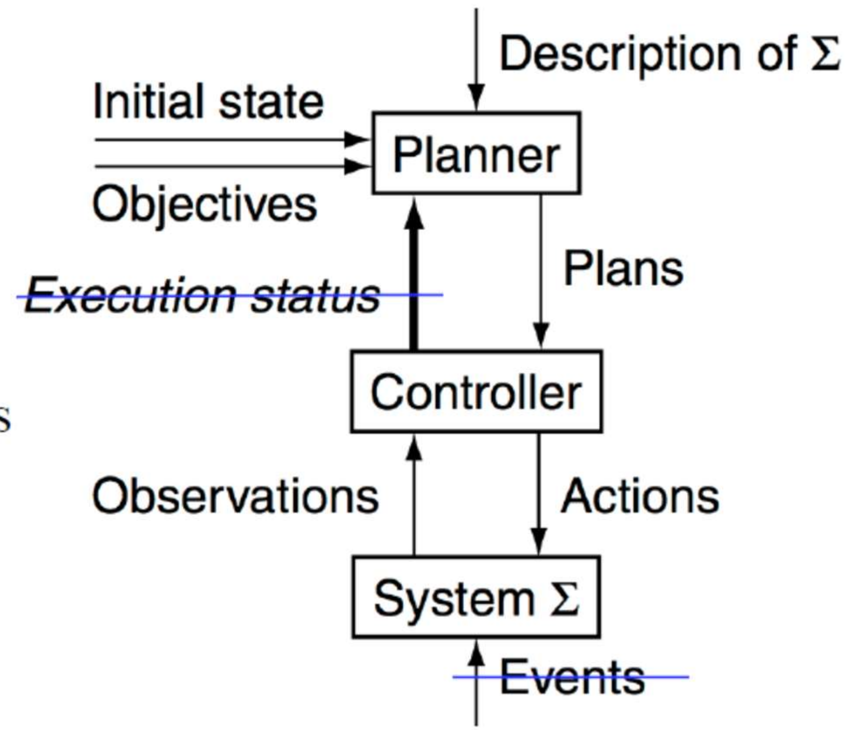
- a plan is a linearly ordered sequence of actions $(a_1, a_2, \ldots a_n)$

A6: Implicit time:

- no time durations; linear sequence of instantaneous states

A7: Off-line planning:

- planner doesn't know the execution status



Classical Planning (Chapters 2–9)

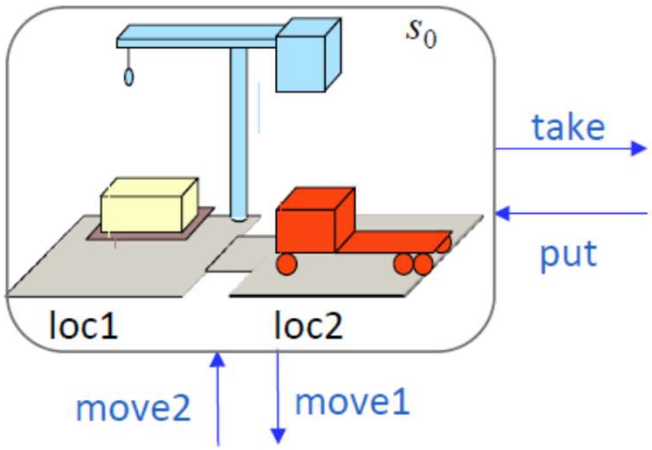
- Classical planning requires all eight restrictive assumptions
  - Offline generation of action sequences for a deterministic, static, finite system, with complete knowledge, attainment goals, and implicit time
- Reduces to the following problem:
  - Given a planning problem $\mathcal{P} = (\Sigma, s_0, S_g)$
  - Find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces a sequence of state transitions $(s_1, s_2, \ldots, s_n)$ such that s_n is in S_g.
- This is just path-searching in a graph
  - Nodes = states
  - Edges = actions
- Is this trivial?

Classical Planning (Chapters 2–9)

- Generalize the earlier example:

  5 locations,
  3 robot vehicles,
  100 containers,
  3 pallets to stack containers on

  - Then there are 10^{277} states



- Number of particles in the universe is only about 10^{87}
  - The example is more than 10^{190} times as large

- Automated-planning research has been heavily dominated by classical planning
  - Dozens (hundreds?) of different algorithms

Classical Planning Problem

Newell and Simon 1956

- Given the *actions* available in a task domain.
- Given a problem specified as:
  - an initial *state* of the world,
  - a set of *goals* to be achieved.

- Find a *solution* to the problem, i.e., a *way* to transform the initial state into a new state of the world where the goal statement is true.

Action Model, State, Goals

Classical Planning

- Action Model: complete, deterministic, correct, rich representation
- State: single initial state, fully known
- Goals: complete satisfaction

Several different planning algorithms

STRIPS Domain

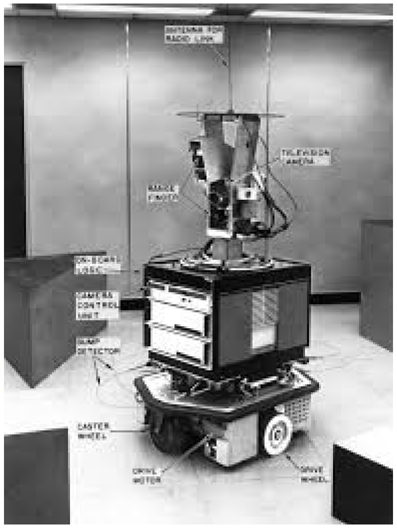
STanford Research Institute Problem Solver [Fikes, Nilsson, 1971]

Pickup_from_table(b)
- Pre: Block(b), Handempty
  Clear(b), On(b, Table)
- Add: Holding(b)
- Delete: Handempty,
  On(b, Table)

Pickup_from_block(b, c)
- Pre: Block(b), Handempty
  Clear(b), On(b, c), Block(c)
- Add: Holding(b), Clear(c)
- Delete: Handempty,
  On(b, c)

Putdown_on_table(b)
- Pre: Block(b), Holding(b)
- Add: Handempty,
  On(b, Table)
- Delete: Holding(b)

Putdown_on_block(b, c)
- Pre: Block(b), Holding(b)
  Block(c), Clear(c), $b \neq c$
- Add: Handempty, On(b, c)
- Delete: Holding(b), Clear(c)



Init: On(a,Table), On(b,table), On(c,table)

Goal: On(a,table),On(b,a), On(c,b)

STRIPS-like Domain

Observation-1
target — pointing
instruments — calibrated
Observation-2
Observation-3
Observation-4
…

TakeImage (?target, ?instr):
- Pre: Status(?instr, Calibrated), Pointing(?target)
- Eff: Image(?target)

Calibrate (?instrument):
- Pre: Status(?instr, On), Calibration-Target(?target), Pointing(?target)
- Eff: ¬Status(?inst, On), Status(?instr, Calibrated)

Turn (?target):
- Pre: Pointing(?direction), ?direction ≠ ?target
- Eff: ¬Pointing(?direction), Pointing(?target)

Representations: Motivation

- In most problems, far too many states to try to represent all of them explicitly as s_0, s_1, s_2, …
- Represent each state as a set of features
  - e.g.,
    - » a vector of values for a set of variables
    - » a set of ground atoms in some first-order language L
- Define a set of *operators* that can be used to compute state-transitions
- Don't give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed

Classical Representation

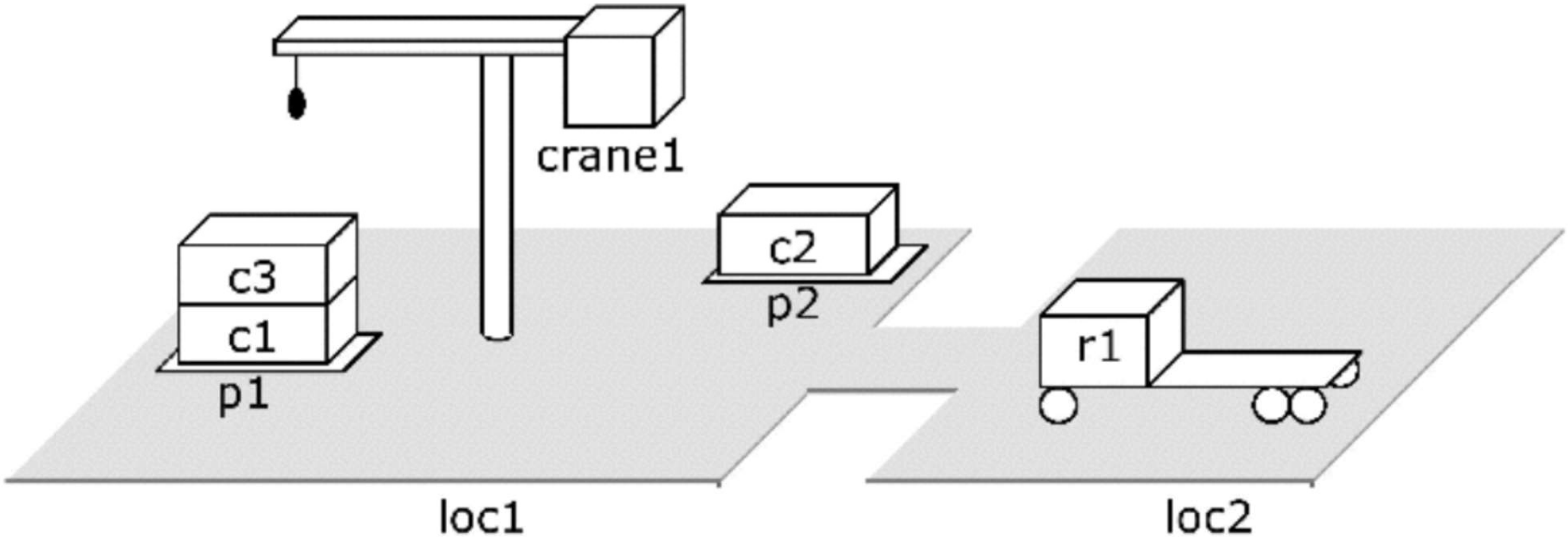
- *Atom*: predicate symbol and args
  - Use these to represent both fixed and dynamic relations

adjacent(l,l')	attached(p,l)	belong(k,l)
occupied(l)	at(r,l)	
loaded(r,c)	unloaded(r)	
holding(k,c)	empty(k)	
in(c,p)	on(c,c')	
top(c,p)	top(pallet,p)	

- *Ground* expression: contains no variable symbols - e.g., in(c1,p3)
- *Unground* expression: at least one variable symbol - e.g., in(c1,x)
- *Substitution*: $\theta = \{x_1 \leftarrow v_1,\ x_2 \leftarrow v_2,\ \ldots,\ x_n \leftarrow v_n\}$
  - Each x_i is a variable symbol; each v_i is a term
- *Instance* of e: result of applying a substitution θ to e
  - Replace variables of e simultaneously, not sequentially

States

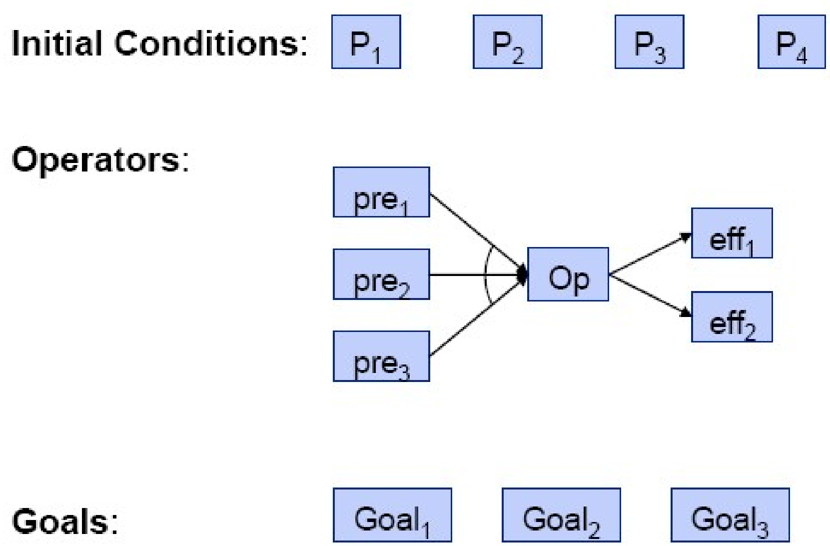
- *State*: a set s of ground atoms
  - The atoms represent the things that are true in one of Σ's states
  - Only finitely many ground atoms, so only finitely many possible states



s_1 = {attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1),
on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,palet),
belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2),
adjacent(loc2,loc1), at(r1,loc2), occupied(loc2, unloaded(r1)}

Planning Problem

- **Planning Domain:**
  - Operators as preconditions and effects
- **Planning Problem:**
  - Initial State, Planning Domain, Goals



Planning Domain

- Frame Problem:
  - How to represent unchanged facts?
  - Example: I go from home (state S) to the store (state S'). In S': The house is still there, Rome is still the largest city in Italy, my shoes are the same, etc..
  - Path Planning has not this issue (sub-symbolic representation)

- Ramification Problem:
  - How to represent indirect effect of the actions
  - I go from home (state S) to the store (state S'). In S':
    The number of people in the store went up by 1,
    The contents of my pockets are now in the store, etc..

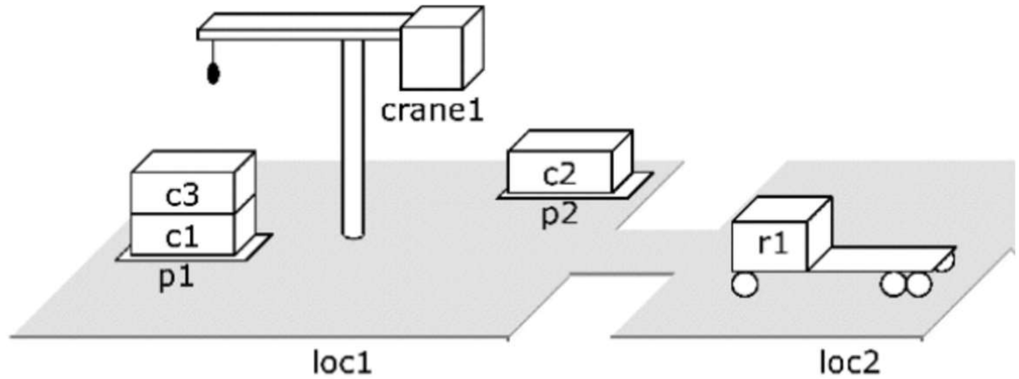
STRIPS Domain

States:
- Set of well-formed formulas (wffs: conjunction of literals)

Set of Actions, each represented with:
– Preconditions (list of predicates that should hold)
– Delete list (list of predicates that will become invalid)
– Add list (list of predicates that will become valid) Actions thus allow variables

A goal condition:
- Well-formed formula

Actions



take(k, l, c, d, p)

;; crane k at location l takes c off of d in pile p

precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)

effects: holding(k, c), $\neg$empty(k), $\neg$in(c, p), $\neg$top(c, p), $\neg$on(c, d), top(d, p)

- An *action* is a ground instance (via substitution) of an operator
  - Let $\theta = \{k \leftarrow$ crane1, $l \leftarrow$ loc1, $c \leftarrow$ c3, $d \leftarrow$ c1, $p \leftarrow$ p1$\}$
  - Then (take(k,l,c,d,p))θ is the following action:

    take(crane1,loc1,c3,c1,p1)

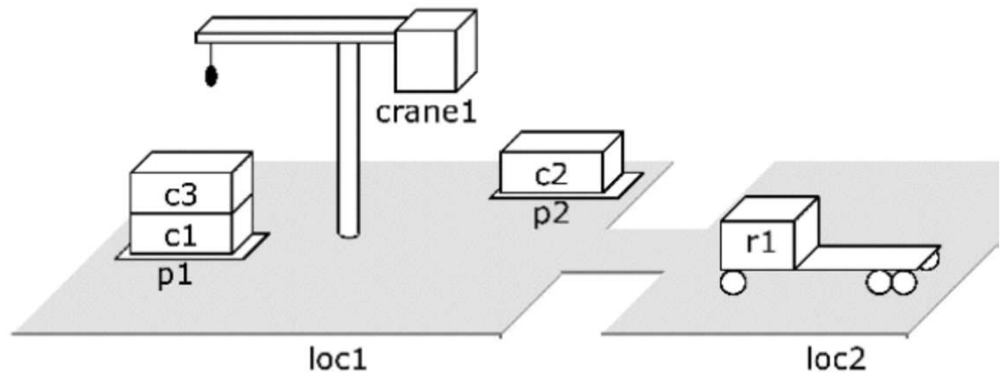
    precond: belong(crane,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

    effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

  - i.e., crane **crane1** at location **loc1** takes **c3** off of **c1** in pile **p1**

Applicability

- Let s be a state and a be an action
- a is *applicable* to (or *executable* in) if s satisfies precond(a)
  - $\text{precond}^{+}(a) \subseteq s$
  - $\text{precond}^{-}(a) \cap s = \varnothing$



- An action:

  take(crane1,loc1,c3,c1,p1)

  precond: belong(crane,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

  effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

- A state it's applicable to

  s_1 = {**attached(p1,loc1)**, in(c1,p1), in(c3,p1), **top(c3,p1)**, **on(c3,c1)**, on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,palet), **belong(crane1,loc1)**, **empty(crane1)**, adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2, unloaded(r1)}

Planning Problems

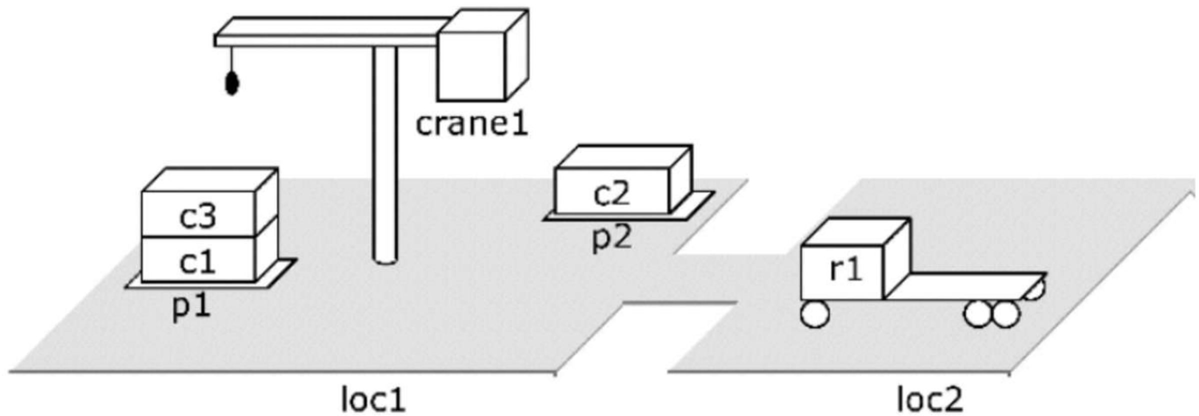
- Given a planning domain (language L, operators O)
  - *Statement* of a planning problem: a triple $P=(O,s_0,g)$
    - » O is the collection of operators
    - » s_0 is a state (the initial state)
    - » g is a set of literals (the goal formula)
  - Planning problem: $\mathcal{P} = (\Sigma, s_0, S_g)$
    - » s_0 = initial state
    - » S_g = set of goal states
    - » $\Sigma = (S, A, \gamma)$ is a state-transition system that satisfies all of the restrictive assumptions in Chapter 1
    - » S = {all sets of ground atoms in L}
    - » A = {all ground instances of operators in O}
    - » γ = the state-transition function determined by the operators
- I'll often say "planning problem" to mean the statement of the problem

Plans and Solutions

- Let $P=(O,s_0,g)$ be a planning problem
- *Plan*: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n\rangle$ such that each a_i is an instance of an operator in O
- π is a *solution* for $P=(O,s_0,g)$ if it is executable and achieves g
  - i.e., if there are states $s_0, s_1, \ldots, s_n$ such that
    - » $\gamma(s_0,a_1) = s_1$
    - » $\gamma(s_1,a_2) = s_2$
    - » …
    - » $\gamma(s_{n-1},a_n) = s_n$
    - » s_n satisfies g

Set-Theoretic Representation

- Like classical representation, but restricted to propositional logic
  - Equivalent to a classical representation in which all of the atoms are ground



- **States:**
  - Instead of ground atoms, use propositions (boolean variables):

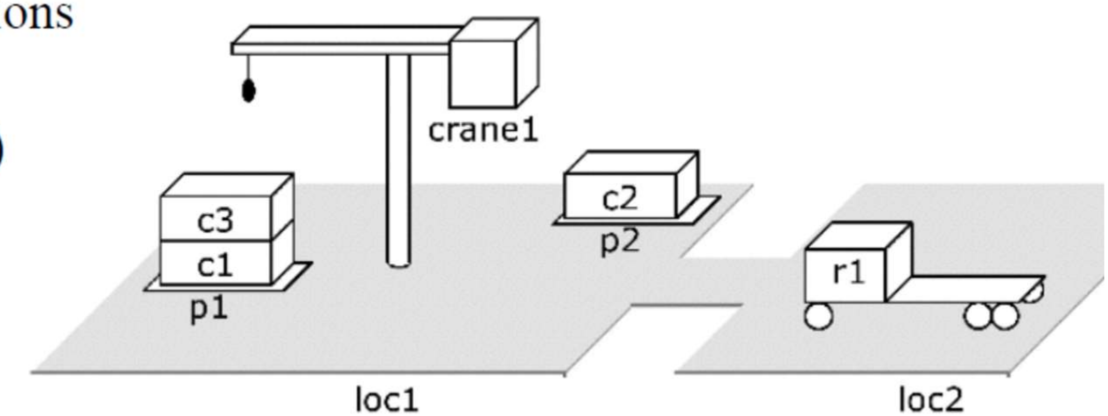
{on(c1,pallet), on(c1,r1), on(c1,c2), …, at(r1,l1), at(r1,l2), …}

⇩

{on-c1-pallet, on-c1-r1, on-c1-c2, …, at-r1-l1, at-r1-l2, …}

Exponential Blowup

- Suppose the language contains c constant symbols
- Let o be a classical operator with k parameters
- Then there are c^k ground instances of o
  - Hence c^k set-theoretic actions
- Example:
  take(crane1,loc1,c3,c1,p1)
  - $k = 5$
  - 1 crane, 2 locations, 3 containers, 2 piles
    - » 8 constant symbols
  - $8^5 = 32768$ ground instances



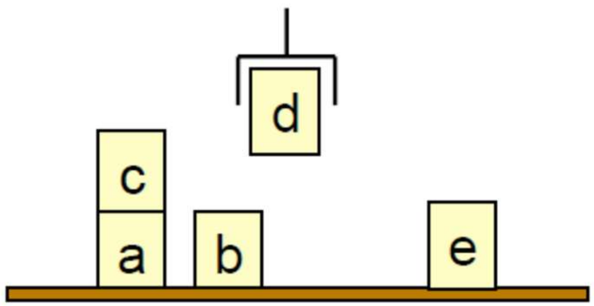
- Can reduce this by assigning data types to the parameters
    - » e.g., first arg must be a crane, second must be a location, etc.
    - » Number of ground instances is now 1 * 2 * 3 * 3 * 2 = 36
  - Worst case is still exponential

Example: The Blocks World

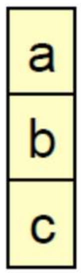
- Infinitely wide table, finite number of children's blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There's a robot gripper that can hold at most one block

- Want to move blocks from one configuration to another
  - e.g.,

initial state



goal



- Like a special case of DWR with one location, one crane, some containers, and many more piles than you need

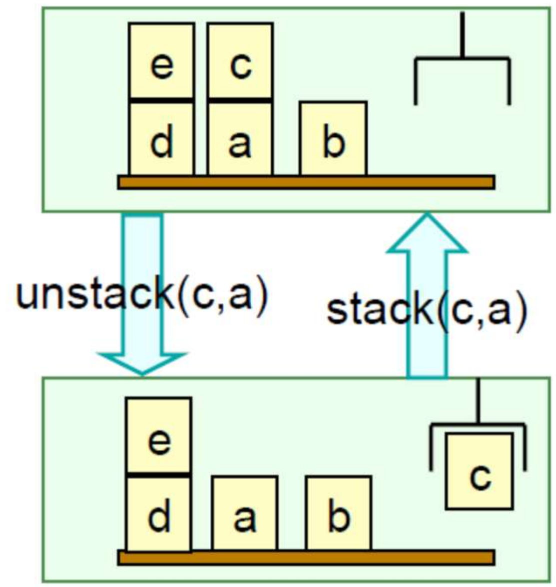
Classical Operators

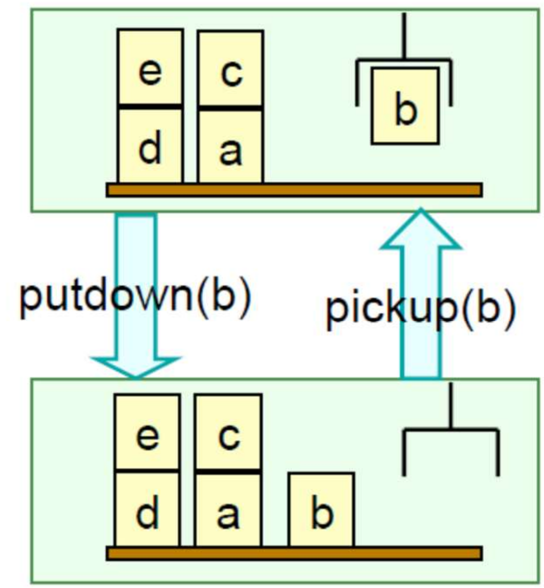
unstack(x,y)
    Precond: on(x,y), clear(x), handempty
    Effects: ¬on(x,y), ¬clear(x), ¬handempty,
                holding(x), clear(y)

stack(x,y)
    Precond: holding(x), clear(y)
    Effects: ¬holding(x), ¬clear(y),
                on(x,y), clear(x), handempty

pickup(x)
    Precond: ontable(x), clear(x), handempty
    Effects: ¬ontable(x), ¬clear(x),
                ¬handempty, holding(x)

putdown(x)
    Precond: holding(x)
    Effects: ¬holding(x), ontable(x),
                clear(x), handempty





Set-Theoretic Actions

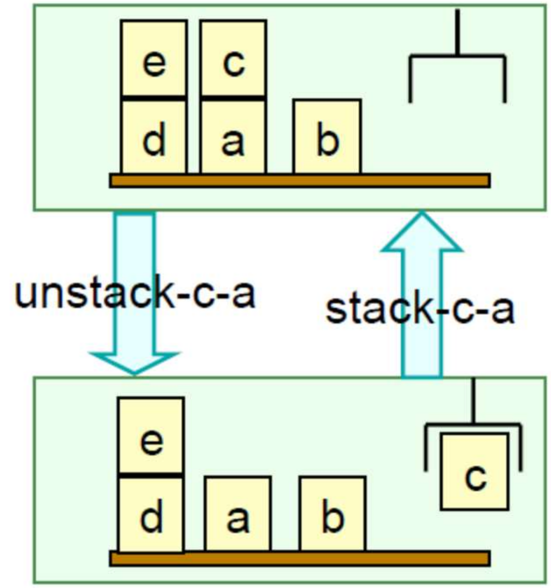
- 60 actions
- 50 if we exclude nonsensical ones, e.g., unstack-e-e
- Here are four of them:

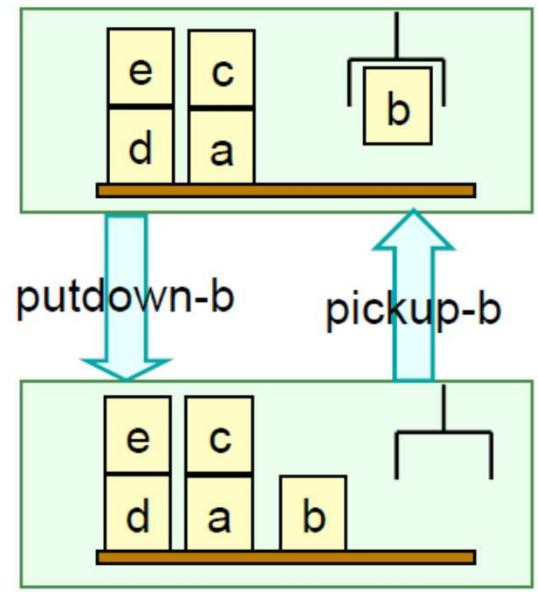
unstack-c-a
- Pre: on-c-a, clear-c, handempty
- Del: on-c-a, clear-c, handempty
- Add: holding-c, clear-a

stack-c-a
- Pre: holding-c, clear-a
- Del: holding-c, clear-a
- Add: on-c-a, clear-c, handempty

pickup-b
- Pre: ontable-b, clear-b, handempty
- Del: ontable-b, clear-b, handempty
- Add: holding-b

putdown-b
- Pre: holding-b
- Del: holding-b
- Add: ontable-b, clear-b, handempty





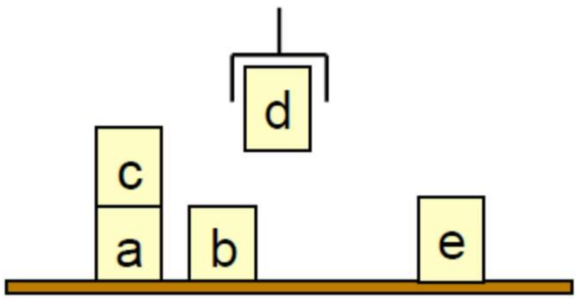
State-Variable Representation: Symbols

- Constant symbols:

  a, b, c, d, e of type block

  0, 1, table, nil of type other

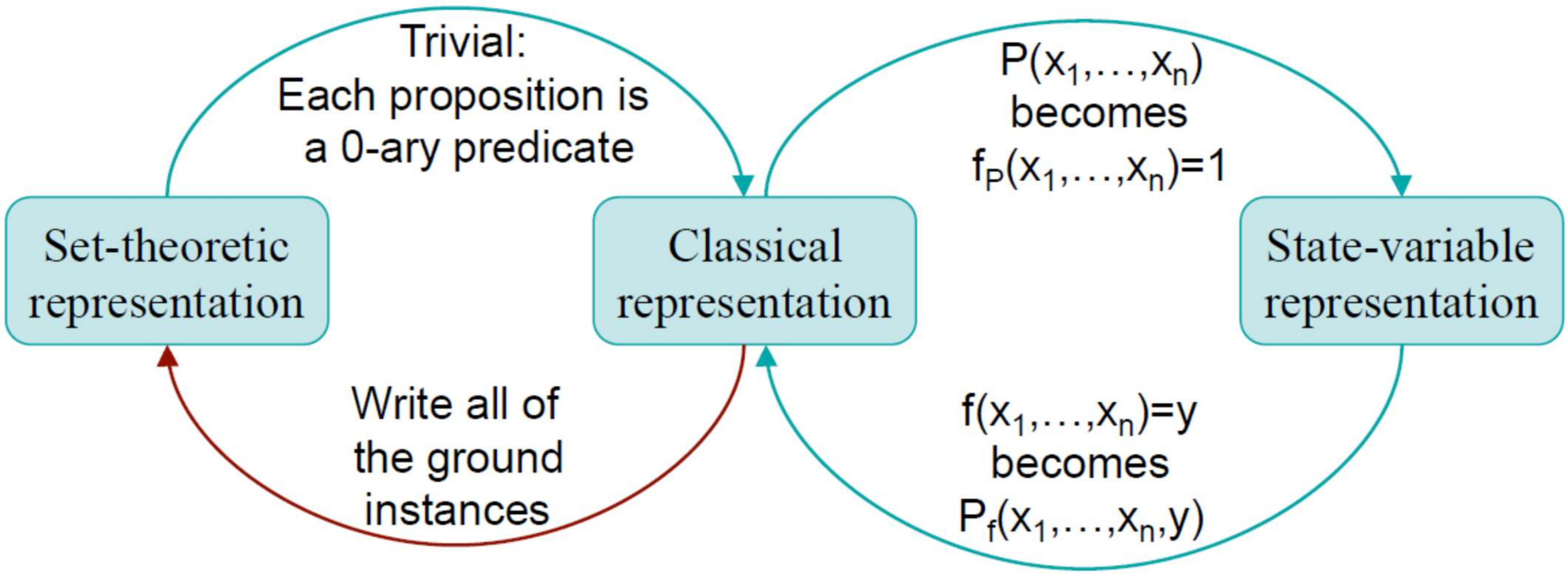
- State variables:



pos(x) = y	if block x is on block y
pos(x) = table	if block x is on the table
pos(x) = nil	if block x is being held
clear(x) = 1	if block x has nothing on it
clear(x) = 0	if block x is being held or has another block on it
holding = x	if the robot hand is holding block x
holding = nil	if the robot hand is holding nothing

Expressive Power

- Any problem that can be represented in one representation can also be represented in the other two
- Can convert in linear time and space in all cases except one:
  - Exponential blowup when converting to set-theoretic



Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons
- Set-theoretic representation
  - Can take much more space than classical representation
  - Useful in algorithms that manipulate ground atoms directly
    - » e.g., planning graphs (Chapter 6), satisfiability (Chapters 7)
  - Useful for certain kinds of theoretical studies
- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers and most computer scientists
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

PDDL Domain

Planning Domain Definition Language
(standard language for classical AI planning)

Components of a PDDL planning task:

- Objects: Things of interest
- Predicates: Relevant properties of objects (can be true or false)
- Initial state: The initial state of the world
- Goal specification: Desiderata
- Actions/Operators: Means to change the state of the world

Planning Domain: predicates and actions.
Planning Problem: initial state and goal specification.

PDDL Domain

Planning Domain Definition Language
(standard language for classical AI planning)

Planning Domain:

(define (domain <domain name>)
<PDDL code for predicates>
<PDDL code for first action>
[...]
<PDDL code for last action>
)

Planning Problem

(define (problem <problem name>)
(:domain <domain name>)
<PDDL code for objects>
<PDDL code for initial state>
<PDDL code for goal specification>
)

(:objects rooma roomb ball1 ball2 ball3 ball4 left right)

(:predicates (ROOM ?x) (BALL ?x) (GRIPPER ?x) (at-robby ?x) (at-ball ?x ?y) (free ?x) (carry ?x ?y))

(:init (ROOM rooma) (ROOM roomb) (BALL ball1) (BALL ball2) (BALL ball3) (BALL ball4) (GRIPPER left) (GRIPPER right) (free left) (free right) (at-robby rooma) (at-ball ball1 rooma) (at-ball ball2 rooma) (at-ball ball3 rooma) (at-ball ball4 rooma))

(:goal (and (at-ball ball1 roomb) (at-ball ball2 roomb) (at-ball ball3 roomb) (at-ball ball4 roomb)))

PDDL Domain

Planning Domain Definition Language
(standard language for classical AI planning)

Planning Domain:

```
(define (domain <domain name>)
<PDDL code for predicates>
<PDDL code for first action>
[...]
<PDDL code for last action>
)
```

Planning Problem:

```
(define (problem <problem name>)
(:domain <domain name>)
<PDDL code for objects>
<PDDL code for initial state>
<PDDL code for goal specification>
)
```

```
(:action move :parameters (?x ?y)
:precondition (and (ROOM ?x) (ROOM ?y) (at-
robby ?x)) :effect (and (at-robby ?y) (not (at-
robby ?x))))
```

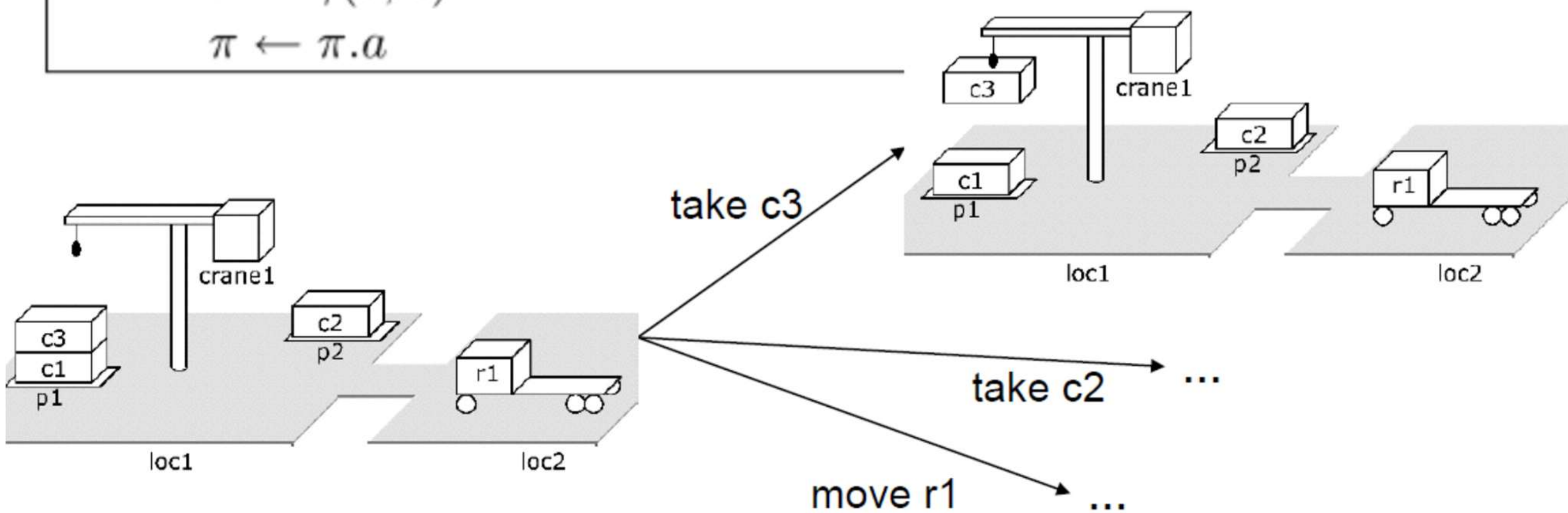
```
(:action pick-up :parameters (?x ?y ?z)
:precondition (and (BALL ?x) (ROOM ?y)
(GRIPPER ?z) (at-ball ?x ?y) (at-robby ?y) (free
?z)) :effect (and (carry ?z ?x) (not (at-ball ?x
?y)) (not (free ?z))))
```

Motivation

- Nearly all planning procedures are search procedures
- Different planning procedures have different search spaces
  - Two examples:
- *State-space planning*
  - Each node represents a state of the world
    - » A plan is a path through the space
- *Plan-space planning*
  - Each node is a set of partially-instantiated operators, plus some constraints
    - » Impose more and more constraints, until we get a plan

Forward-search(O, s_0, g)

$s \leftarrow s_0$
$\pi \leftarrow$ the empty plan
loop
  if s satisfies g then return π
  $E \leftarrow \{a | a$ is a ground instance an operator in O,
    and $\mathrm{precond}(a)$ is true in $s\}$
  if $E = \emptyset$ then return failure
  nondeterministically choose an action $a \in E$
  $s \leftarrow \gamma(s, a)$
  $\pi \leftarrow \pi.a$

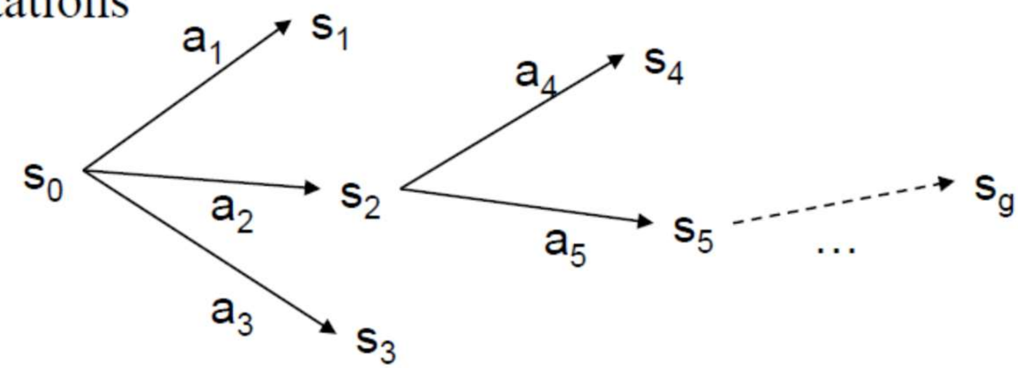


Properties

- Forward-search is *sound*
  - for any plan returned by any of its nondeterministic traces, this plan is guaranteed to be a solution
- Forward-search also is *complete*
  - if a solution exists then at least one of Forward-search's nondeterministic traces will return a solution.

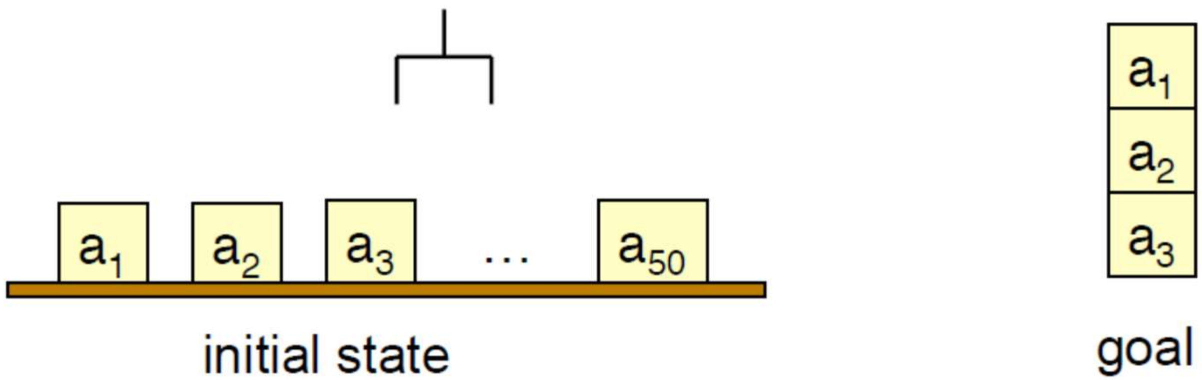
Deterministic Implementations

- Some deterministic implementations of forward search:
  - breadth-first search
  - depth-first search
  - best-first search (e.g., A*)
  - greedy search



- Breadth-first and best-first search are sound and complete
  - But they usually aren't practical because they require too much memory
  - Memory requirement is exponential in the length of the solution
- In practice, more likely to use depth-first search or greedy search
  - Worst-case memory requirement is linear in the length of the solution
  - In general, sound but not complete
    - » But classical planning has only finitely many states
    - » Thus, can make depth-first search complete by doing loop-checking

Branching Factor of Forward Search



- Forward search can have a very large branching factor
  - E.g., many applicable actions that don't progress toward goal
- Why this is bad:
  - Deterministic implementations can waste time trying lots of irrelevant actions
- Need a good heuristic function and/or pruning procedure
  - See Section 4.5 (Domain-Specific State-Space Planning) and Part III (Heuristics and Control Strategies)

Backward Search

- For forward search, we started at the initial state and computed state transitions
  - new state = $\gamma(s,a)$
- For backward search, we start at the goal and compute inverse state transitions
  - new set of subgoals = $\gamma^{-1}(g,a)$
- To define $\gamma^{-1}(g,a)$, must first define *relevance*:
  - An action a is relevant for a goal g if
    - » a makes at least one of g's literals true
      - $g \cap \text{effects}(a) \neq \varnothing$
    - » a does not make any of g's literals false
      - $g^{+} \cap \text{effects}^{-}(a) = \varnothing$ and $g^{-} \cap \text{effects}^{+}(a) = \varnothing$

Inverse State Transitions

- If a is relevant for g, then
  - $\gamma^{-1}(g,a) = (g - \text{effects(a)}) \cup \text{precond}(a)$
- Otherwise $\gamma^{-1}(g,a)$ is undefined

- Example: suppose that
  - g = {on(b1,b2), on(b2,b3)}
  - a = stack(b1,b2)
- What is $\gamma^{-1}(g,a)$?

Backward-search(O, s_0, g)

$\pi \leftarrow$ the empty plan

loop

if s_0 satisfies g then return π

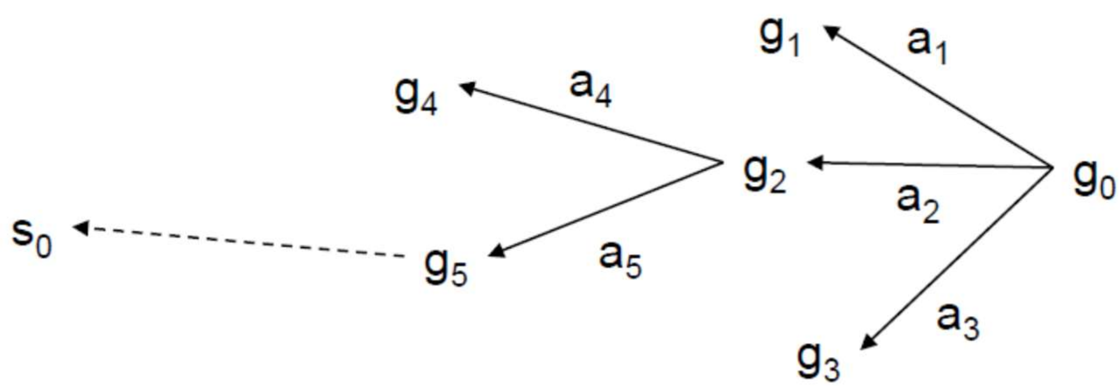
$A \leftarrow \{a | a$ is a ground instance of an operator in O and $\gamma^{-1}(g, a)$ is defined$\}$

if $A = \emptyset$ then return failure

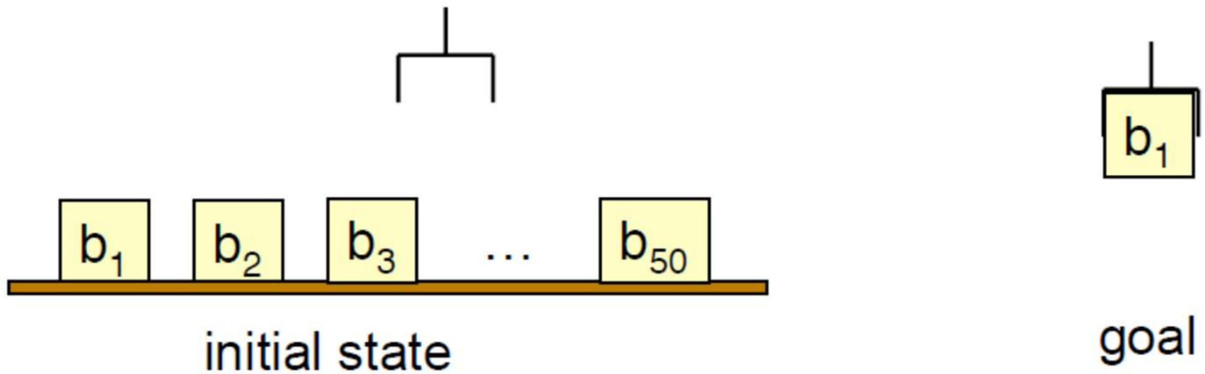
nondeterministically choose an action $a \in A$

$\pi \leftarrow a.\pi$

$g \leftarrow \gamma^{-1}(g, a)$

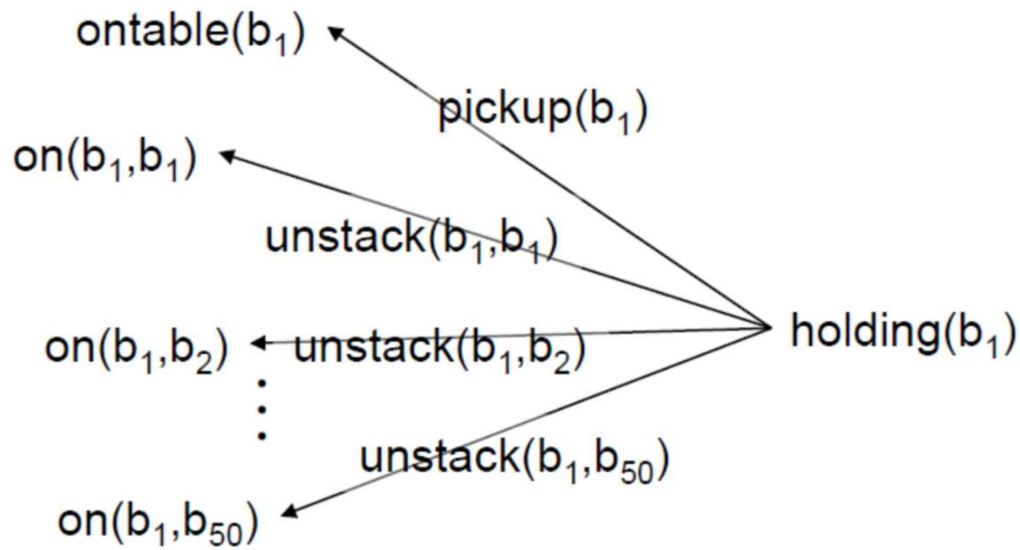


Efficiency of Backward Search

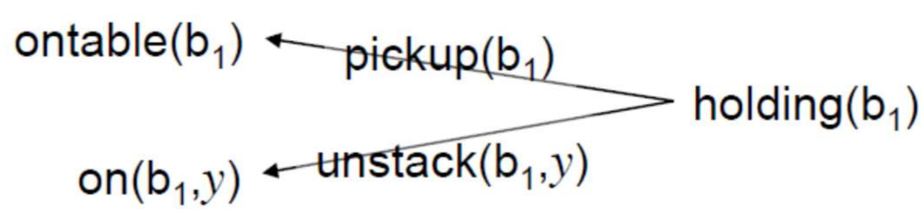


- Backward search can *also* have a very large branching factor
  - E.g., an operator *o* that is relevant for *g* may have many ground instances $a_1, a_2, \ldots, a_n$ such that each a_i's input state might be unreachable from the initial state
- As before, deterministic implementations can waste lots of time trying all of them

Lifting



- Can reduce the branching factor of backward search if we *partially* instantiate the operators
  - this is called *lifting*



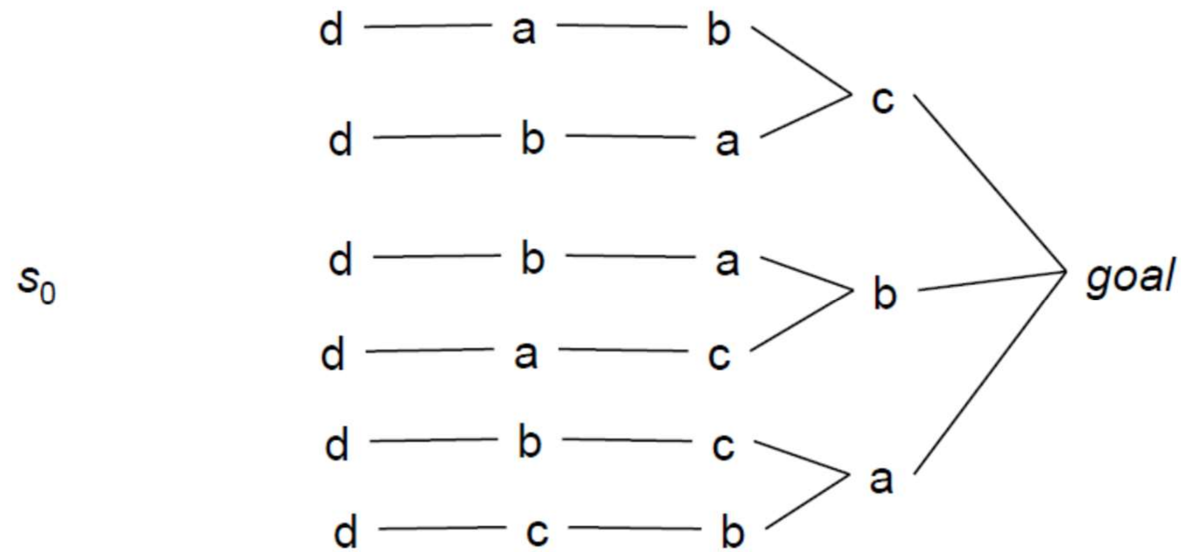
Lifted Backward Search

- More complicated than Backward-search
  - Have to keep track of what substitutions were performed
- But it has a much smaller branching factor

```
Lifted-backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        A ← {(o, θ)|o is a standardization of an operator in O,
                θ is an mgu for an atom of g and an atom of effects⁺(o),
                and γ⁻¹(θ(g), θ(o)) is defined}
        if A = ∅ then return failure
        nondeterministically choose a pair (o, θ) ∈ A
        π ← the concatenation of θ(o) and θ(π)
        g ← γ⁻¹(θ(g), θ(o))
```

The Search Space is Still Too Large

- Lifted-backward-search generates a smaller search space than Backward-search, but it still can be quite large
  - Suppose actions *a*, *b*, and *c* are independent, action *d* must precede all of them, and there's no path from s_0 to *d*'s input state
  - We'll try all possible orderings of *a*, *b*, and *c* before realizing there is no solution
  - More about this in Chapter 5 (Plan-Space Planning)



STRIPS

- Basic idea: given a compound goal $g = \{g_1, g_1, \ldots\}$, try to solve each g_i separately
  - Works if the goals are *serializable* (can be solved in some linear order)

$\pi \leftarrow$ the empty plan

do a modified backward search from g:

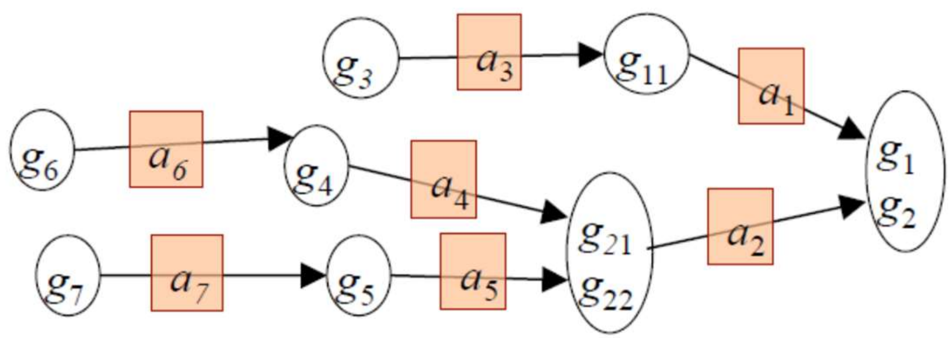
instead of $\gamma^{-1}(s,a)$, each new set of subgoals is just precond(a)

whenever you find an action that's executable in the current state,

go forward on the current search path as far as possible, executing actions and appending them to π

repeat until all goals are satisfied

$\pi = \langle \pi_1, \pi_2 \rangle$ or $\langle \pi_2, \pi_1 \rangle$
$\pi_2 = \langle \pi_{11}, \pi_{12}, a_2 \rangle$ or $\langle \pi_{12}, \pi_{11}, a_2 \rangle$
$\pi_{21} = \langle a_7, a_4 \rangle$
$\pi_{22} = \langle a_7, a_5 \rangle$



Linear Planning

- A linear planner is a classical planner such that:
  - no importance distinction of goals
  - all (sub)goals are assumed to be independent
  - (sub)goals can be achieved in arbitrary order

- Plans that achieve subgoals are combined by placing *all steps* of one subplan *before or after all* steps of the others (=non-interleaved)

Linear Planning

- Means-Ends analysis
  - What means (operators) are available to achieve the ends (goals)
  - Difference between goal and current state
  - Operator to reduce the difference
  - Means-ends analysis on new subgoals

STRIPS Planning

- STRIPS (*initial-state, goals*)
  - *state* = *initial-state*; *plan* = []; *stack* = []
  - Push *goals* on *stack*
  - Repeat until *stack* is empty
    - If top of *stack* is goal that matches *state*, then pop *stack*
    - Else if top of stack is a **conjunctive goal** *g*, then
      - **Select** an ordering for the subgoals of *g*, and push them on *stack*
    - Else if top of *stack* is a **simple goal** *sg*, then
      - **Choose** an operator *o* whose add-list matches goal *sg*
      - Replace goal *sg* with operator *o*
      - Push the preconditions of *o* on the *stack*
    - Else if top of *stack* is an **operator** *o*, then
      - *state* = apply(*o*, *state*)
      - *plan* = [*plan*; *o*]

Simmons, Veloso : Fall 2001

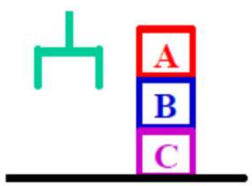
Linear Planning

- Advantage:
  - Goals are solved one at a time (ok if independent)
  - Sound
- Disadvantage
  - Suboptimal solutions (number of operators in the plan)
  - incomplete

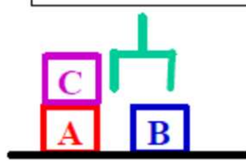
The "Sussman Anomaly"

1.

Stack	State
On(A, B) On(B, C)	Clear(B)
On(A, B)	Clear(C)
On(B, C)	On(C, A)
	On(A, Table)
	On(B, Table)
	Handempty



Goal



Initial State

2.

Stack	State
On(A, B) On(B, C)	Clear(B)
On(B, C)	Clear(C)
Put_Block(A, B)	On(C, A)
Holding(A) Clear(B)	On(A, Table)
Holding(A)	On(B, Table)
Clear(B)	Handempty

The "Sussman Anomaly"

3.

Stack
On(A, B) On(B, C)
On(B, C)
Put_Block(A, B)
Holding(A) Clear(B)
Holding(A)
Pick_Table(A)
Handempty Clear(A) On(A, Table)
Clear(A)
Handempty
On(A, Table)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

4.

Stack
On(A, B) On(B, C)
On(B, C)
Put_Block(A, B)
Holding(A) Clear(B)
Holding(A)
Pick_Table(A)
Handempty Clear(A) On(A, Table)
Pick_Block(C, A)
Handempty Clear(C) On(C, A)

State

Clear(B)
Clear(C)
On(C, A)
On(A, Table)
On(B, Table)
Handempty

The "Sussman Anomaly"

5. Stack

On(A, B) On(B, C)
On(B, C)
Put_Block(A, B)
Holding(A) Clear(B)
Holding(A)
Pick_Table(A)
Handempty Clear(A) On(A, Table)

[Pick(C,A)]

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)

6. Stack

On(A, B) On(B, C)
On(B, C)
Put_Block(A, B)
Holding(A) Clear(B)
Holding(A)
Pick_Table(A)
Handempty Clear(A) On(A, Table)
Put_Table(C)
Holding(C)

[Pick(C,A)]

State

Clear(B)
Clear(C)
On(A, Table)
On(B, Table)
Holding(C)

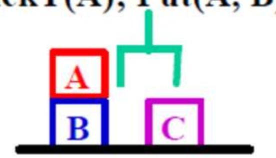
The "Sussman Anomaly"

7. Stack

On(A, B) On(B, C)

On(B, C)

**[Pick(C,A); PutT(C);
PickT(A); Put(A, B)]**



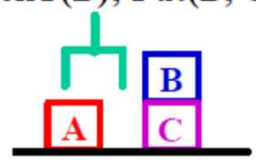
State

Clear(C)
On(B, Table)
On(C, Table)
Clear(A)
On(A, B)
Handempty

8. Stack

On(A, B) On(B, C)

**[Pick(C,A); PutT(C);
PickT(A); Put(A, B);
Pick(A, B); PutT(A);
PickT(B); Put(B, C)]**



State

On(C, Table)
Clear(B)
Clear(A)
On(A, Table)
On(B, C)
Handempty

9. Stack

On(A, B) On(B, C)

On(A, B)

On(B, C)

**[Pick(C,A); PutT(C);
PickT(A); Put(A, B);
Pick(A, B); PutT(A);
PickT(B); Put(B, C)]**

State

On(C, Table)
Clear(B)
Clear(A)
On(A, Table)
On(B, C)
Handempty

10. Stack

On(A, B) On(B, C)

**[Pick(C,A); PutT(C);
PickT(A); Put(A, B);
Pick(A, B); PutT(A);
PickT(B); Put(B, C);
PickT(A); Put(A, B)]**

State

On(C, Table)
Clear(A)
On(B, C)
On(A, B)
Handempty

The Register Assignment Problem

- Interchange the values stored in two registers
  - State-variable formulation:
    - » registers r1, r2, r3

s_0: {value(r1)=3, value(r2)=5, value(r3)=0}

g: {value(r1)=5, value(r2)=3}

Operator: assign(r,v,r',v')

precond: value(r)=v, value(r')=v'

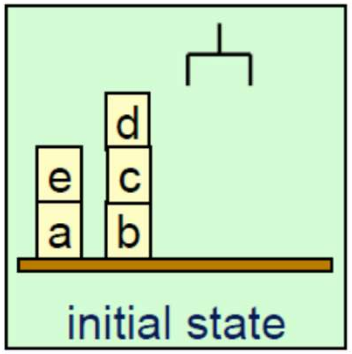
effects: value(r)=v'

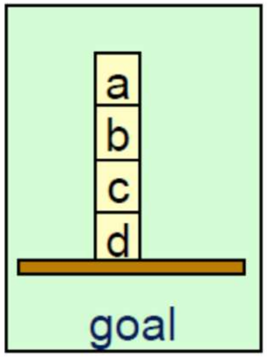
- STRIPS cannot solve this problem at all

Block-Stacking Algorithm

- All of the possible situations in which a block x needs to be moved:
  - s contains ontable(x) and g contains on(x,y) - e.g., a
  - s contains on(x,y) and g contains ontable(x) - e.g., d
  - s contains on(x,y) and g contains on(x,z) for some $y \neq z$ - e.g., c
  - s contains on(x,y) and y needs to be moved - e.g., e

loop
  if there is a clear block x that needs to be moved
      and x can be moved to a place where it won't need to be moved
    then move x to that place
  else if there's a clear block x that needs to be moved
    then move x to the table
  else if the goal is satisfied **then return** the plan
  else return failure
repeat





Non-Linear Planning

- **Basic Idea**
  - Goal set instead of goal stack
  - Search space all possible subgoal orderings
  - Goal interactions by interleaving

- **Advantages**
  - Sound, complete, can be optimal with respect to plan length (depending on search strategy employed)

- **Disadvantages**
  - Larger search space

Non-Linear Planning

NLP (*initial-state, goals*)

- *state* = *initial-state*; *plan* = []; *goalset* = *goals*; *opstack* = []
- Repeat until *goalset* is empty
  - **Choose** a goal *g* from the *goalset*
  - If *g* does not match *state*, then
    - **Choose** an operator *o* whose add-list matches goal *g*
    - Push *o* on the *opstack*
    - Add the preconditions of *o* to the *goalset*
  - While all preconditions of operator on top of *opstack* are met in *state*
    - Pop operator *o* from top of *opstack*
    - *state* = apply(*o*, *state*)
    - *plan* = [*plan*; *o*]

Simmons, Veloso : Fall 2001

Heuristics for Forward-Chaining Planning

Several classical planning style are available:
- http://icaps-conference.org/index.php/Main/Competitions

Forward-chaining planners:
- solving an abstraction of the original, hard, planning problem

The most widely used abstraction involves planning using `relaxed actions', where the delete effects of the original actions are ignored.

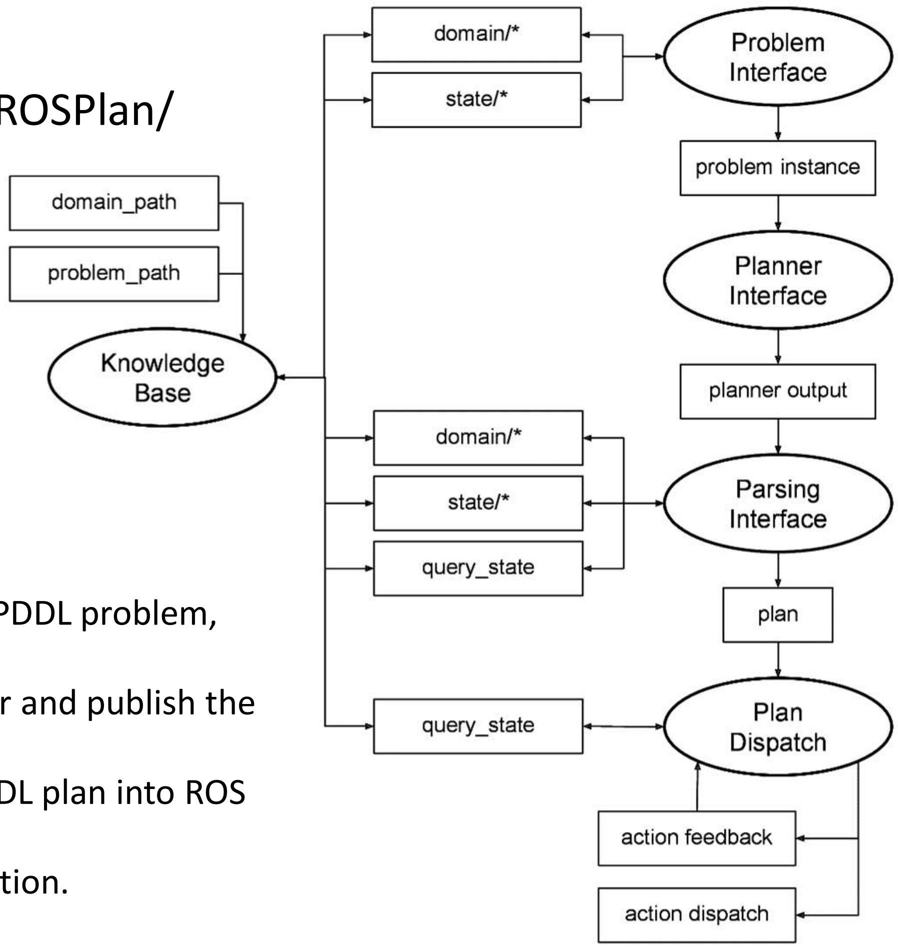
Examples:
FF [Hoffmann & Nebel 2001], HSP [Bonet & Geffner 2000], UnPOP [McDermott 1996] use relaxed actions as the basis for their heuristic estimates

FF was the first to count the number of relaxed actions in a relaxed plan connecting the goal to the initial state

ROSPlan

The ROSPlan framework provides a collection of tools for AI Planning in a ROS system. ROSPlan has a variety of nodes which encapsulate planning, problem generation, and plan execution

https://kcl-planning.github.io/ROSPlan/

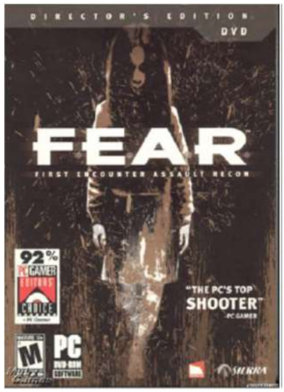


- **Knowledge Base** stores a PDDL model
- **Problem Interface** used to generate a PDDL problem, publish it on a topic, or write it to file
- **Planner Interface** used to call a planner and publish the plan to a topic, or write it to file
- **Parsing Interface** used to convert a PDDL plan into ROS messages, ready to be executed.
- **Plan Dispatch** encapsulates plan execution.

STRIPS and Games

Behavior of Non Player Characters (NPCs) can be described by abstract actions defined in a symbolic world model, e.g. First-Person Shooter (FPS) games

F.E.A.R. (short for First Encounter Assault Recon) is a horror-themed first-person shooter developed by Monolith Productions



– Gamespot’s Best AI Award in 2005

– Ranked 2nd in the list of most influential AI games

The agents’ behavior is a function of the generated plans based on goals, state, and available actions

Jeff Orkin: Three States and a Plan: The AI of F.E.A.R. *Proceedings of the Game Developer's Conference (GDC)*

Olivier Bartheye and Eric Jacopin: A PDDL-Based Planning Architecture to Support Arcade Game Playing

Summary

- If classical planning is extended to allow function symbols
  - Then we can encode arbitrary computations as planning problems
    - » Plan existence is semidecidable
    - » Plan length is decidable
- Ordinary classical planning is quite complex
    - » Plan existence is EXPSPACE-complete
    - » Plan length is NEXPTIME-complete
  - But those are *worst case* results
    - » If we can write domain-specific algorithms, most well-known planning problems are much easier