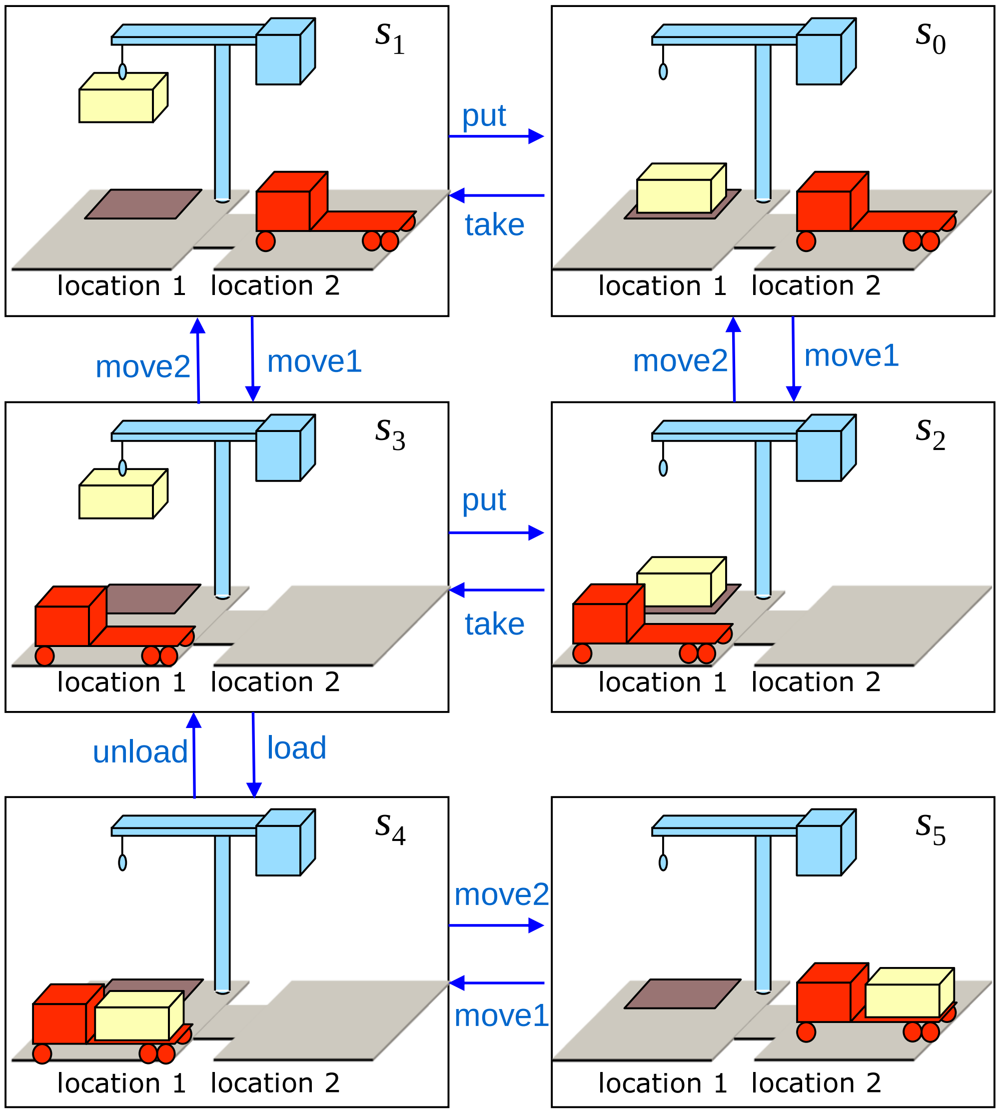
Chapter 8: Planning Representation And Algorithms

Hankui Zhuo

April 12, 2019

Quick Review of Classical Planning

- Classical planning requires all eight of the restrictive assumptions:
  - A0: Finite
  - A1: Fully observable
  - A2: Deterministic
  - A3: Static
  - A4: Attainment goals
  - A5: Sequential plans
  - A6: Implicit time
  - A7: Offline planning



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

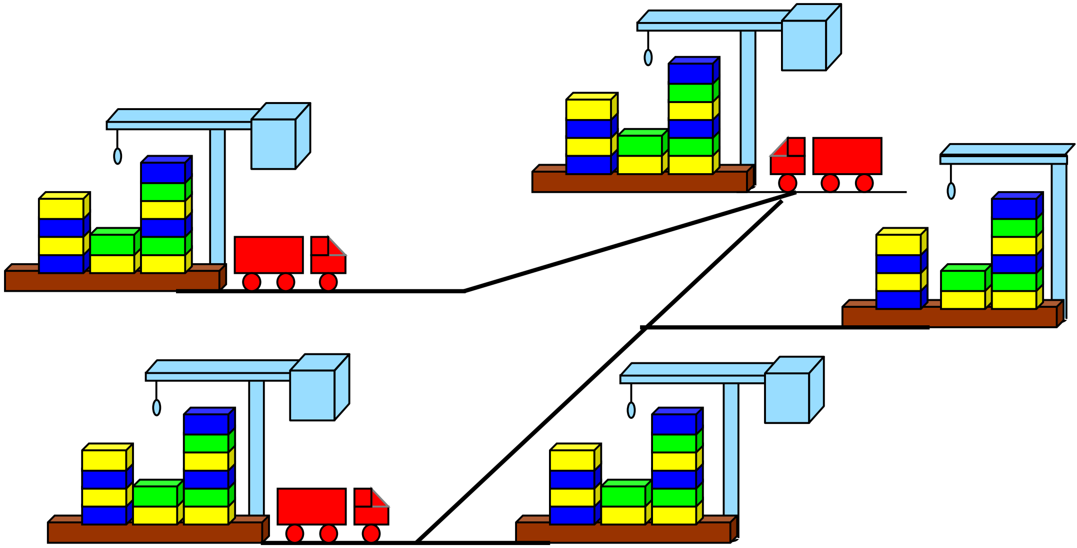
Outline

- Representation schemes
  - Classical representation
  - Set-theoretic representation
  - State-variable representation
  - Examples: DWR and the Blocks World
  - Comparisons

Classical Representation

- Start with a *function-free* first-order language
  - Finitely many predicate symbols and constant symbols, but *no* function symbols

- Example: the DWR domain
  - Locations: l1, l2, …
  - Containers: c1, c2, …
  - Piles: p1, p2, …
  - Robot carts: r1, r2, …
  - Cranes: k1, k2, …



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

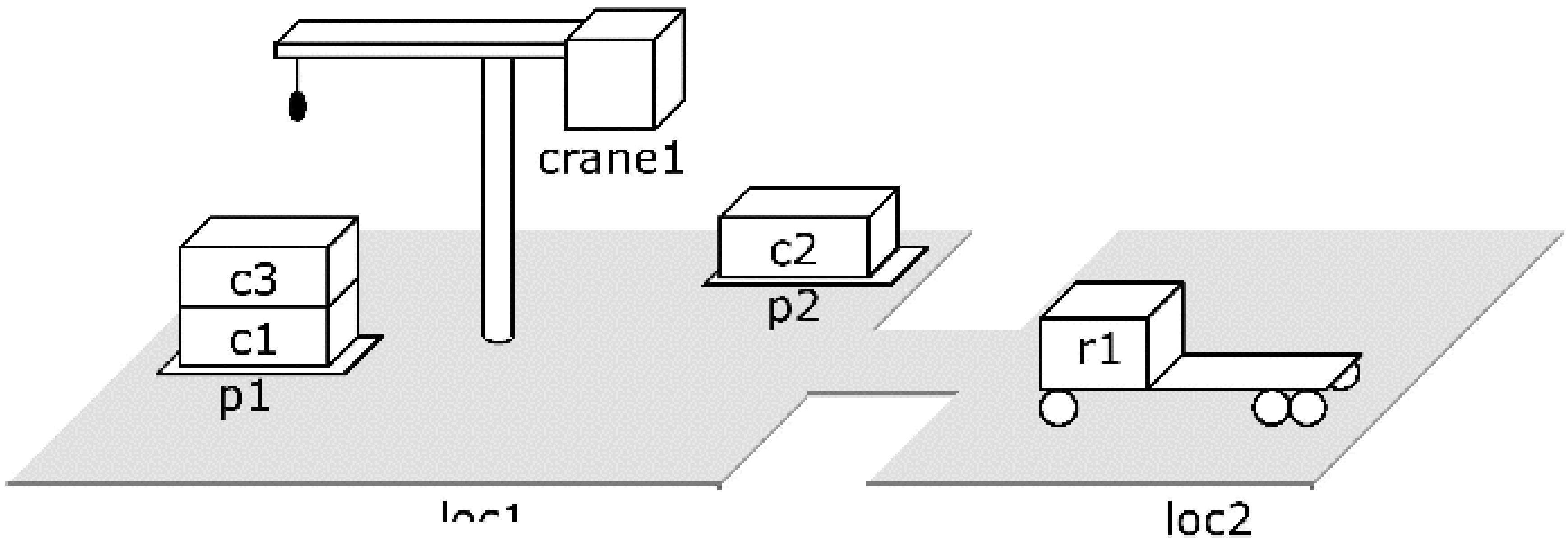


Figure 2.2: The DWR state s_1={attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

Operators

- *Operator*: a triple o=(name(o), precond(o), effects(o))
  - name(o) is a syntactic expression of the form $n(x_1,\ldots,x_k)$
    - » n: *operator symbol* - must be unique for each operator
    - » $x_1,\ldots,x_k$: variable symbols (parameters)
      - must include every variable symbol in o
  - precond(o): *preconditions*
    - » literals that must be true in order to use the operator
  - effects(o): *effects*
    - » literals the operator will make true

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

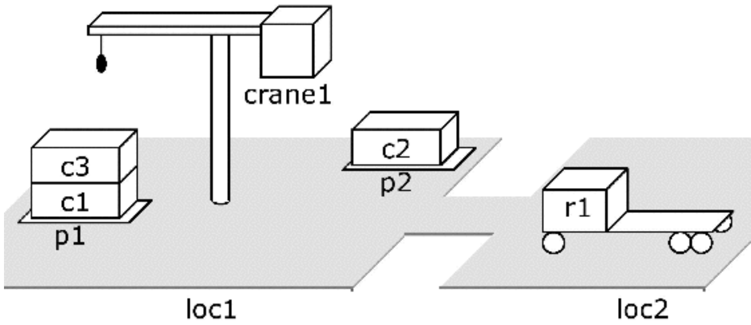
;; crane k at location l takes c off of d in pile p

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)

Actions

- *Action*: ground instance (via substitution) of an operator



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)

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,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)

Notation

- Let S be a set of literals. Then
  - S^+ = {atoms that appear positively in S}
  - S^- = {atoms that appear negatively in S}
- More specifically, let a be an operator or action. Then
  - $\text{precond}^+(a)$ = {atoms that appear positively in a's preconditions}
  - $\text{precond}^-(a)$ = {atoms that appear negatively in a's preconditions}
  - $\text{effects}^+(a)$ = {atoms that appear positively in a's effects}
  - $\text{effects}^-(a)$ = {atoms that appear negatively in a's effects}

$\mathsf{take}(k, l, c, d, p)$

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

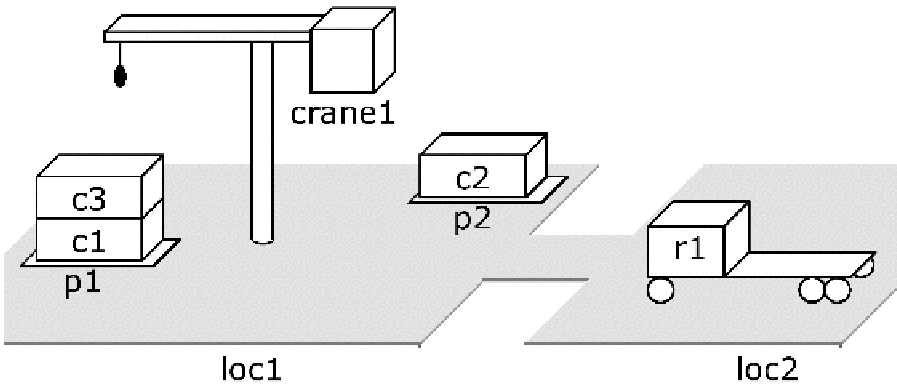
precond: $\mathsf{belong}(k, l), \mathsf{attached}(p, l), \mathsf{empty}(k), \mathsf{top}(c, p), \mathsf{on}(c, d)$

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

- $\text{effects}^+(\text{take}(k,l,c,d,p)$ = {holding(k,c), top(d,p)}
- $\text{effects}^-(\text{take}(k,l,c,d,p)$ = {empty(k), in(c,p), top(c,p), on(c,d)}

Applicability

- An action a is *applicable* to a state s if s satisfies precond(a),
  - i.e., if $\text{precond}^{+}(a) \subseteq s$ and $\text{precond}^{-}(a) \cap s = \varnothing$
- Here are an action and a state that it's applicable to:



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

;; crane crane1 at location loc1 takes c3 off c1 in pile p1

precond: belong(crane1,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)

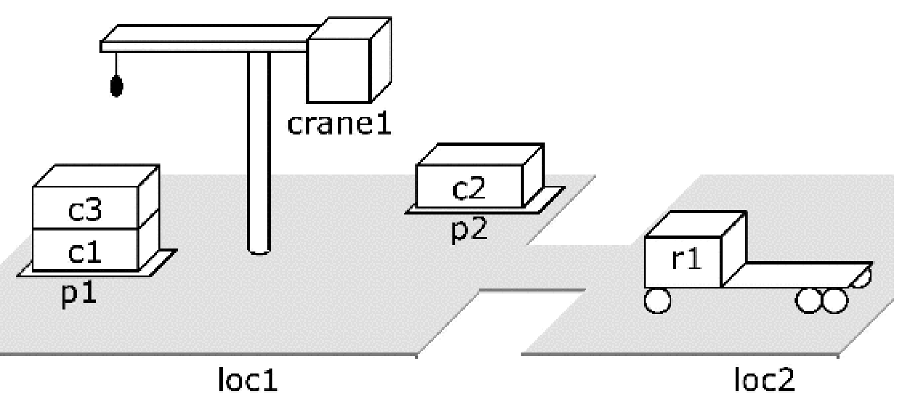
Result of Performing an Action

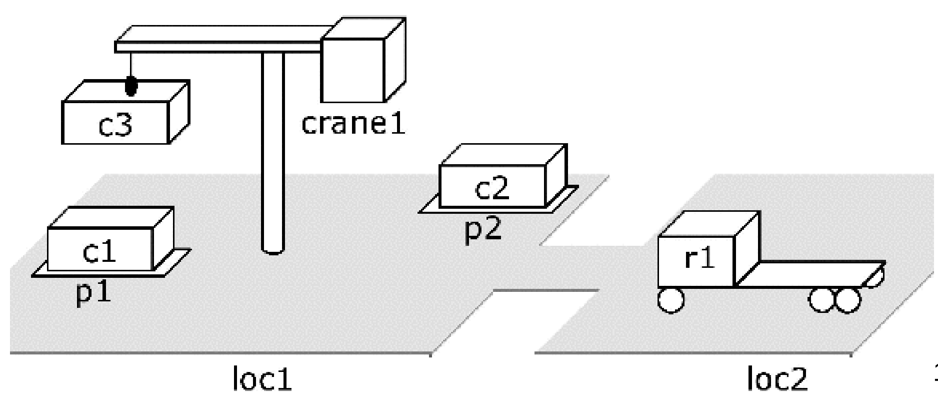
- If *a* is applicable to *s*, the result of performing it is

  $$\gamma(s,a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$$

  - Delete the negative effects, and add the positive ones

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,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)



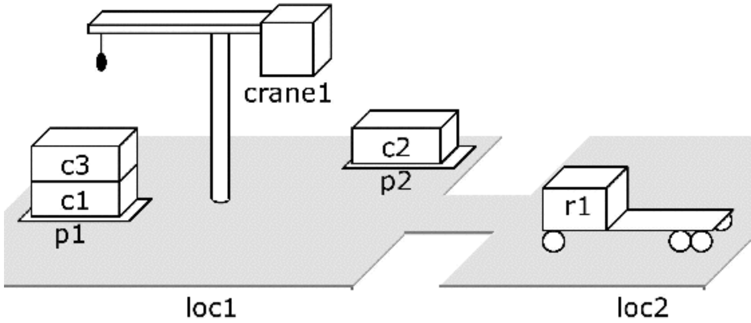


- Planning domain: language plus operators
  - Corresponds to a set of state-transition systems
  - Example: operators for the DWR domain

move(r, l, m)
;; robot r moves from location l to location m
precond: adjacent(l, m), at(r, l), $\neg$ occupied(m)
effects: at(r, m), occupied(m), $\neg$ occupied(l), $\neg$ at(r, l)

load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), $\neg$ holding(k, c), loaded(r, c), $\neg$ unloaded(r)

unload(k, l, c, r)
;; crane k at location l takes container c from robot r
precond: belong(k, l), at(r, l), loaded(r, c), empty(k)
effects: $\neg$ empty(k), holding(k, c), unloaded(r), $\neg$ loaded



put(k, l, c, d, p)
;; crane k at location l puts c onto d in pile p
precond: belong(k, l), attached(p, l), holding(k, c), top(d, p)
effects: $\neg$ holding(k, c), empty(k), in(c, p), top(c, p), on(c, d), $\neg$ top(d, p)

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)

Planning Problems

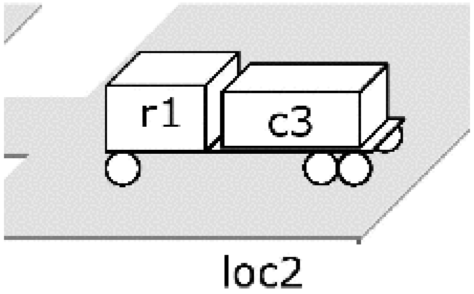
- Given a planning domain (language L, operators O)
  - *Statement* of a planning problem: a triple $P=(O,s_0,g)$
    - » O is the collection of operators
    - » s_0 is a state (the initial state)
    - » g is a set of literals (the goal formula)
  - The actual *planning problem*: $\mathrm{P} = (\Sigma,s_0,S_g)$
    - » s_0 and S_g are as above
    - » $\Sigma = (S,A,\gamma)$ is a state-transition system
    - » 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" when I mean the statement of the problem

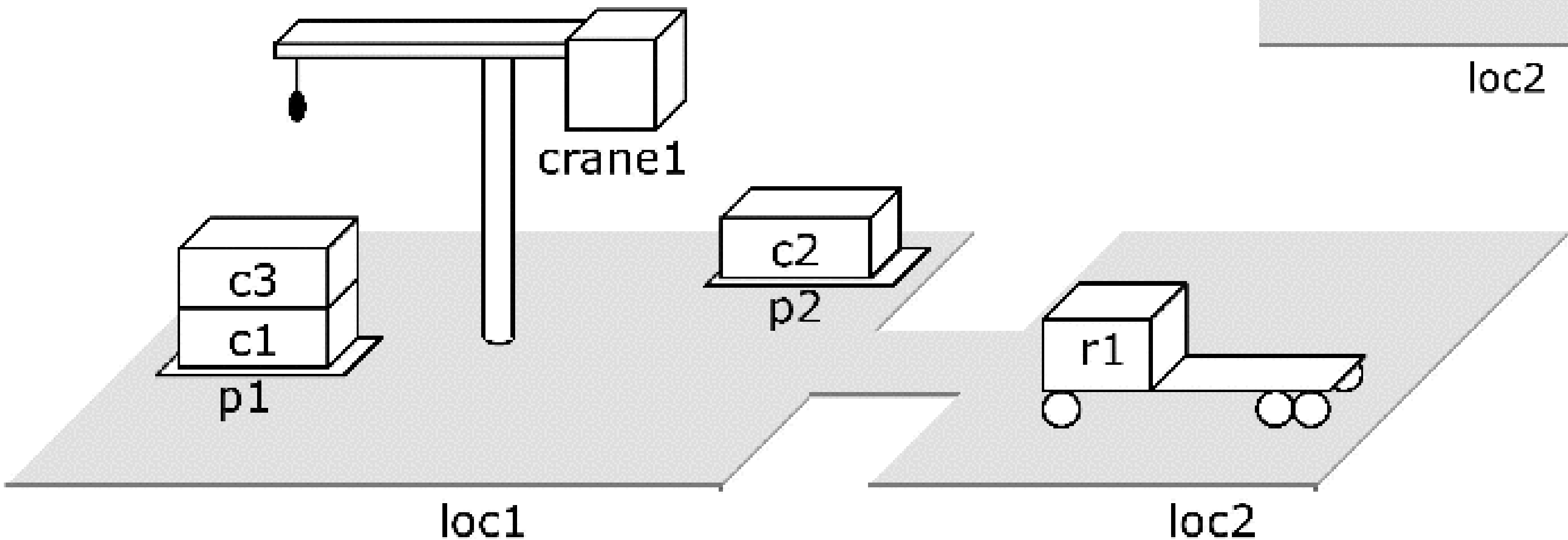
Plans and Solutions

- *Plan*: any sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is a ground instance of an operator in O
- The plan 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

Example

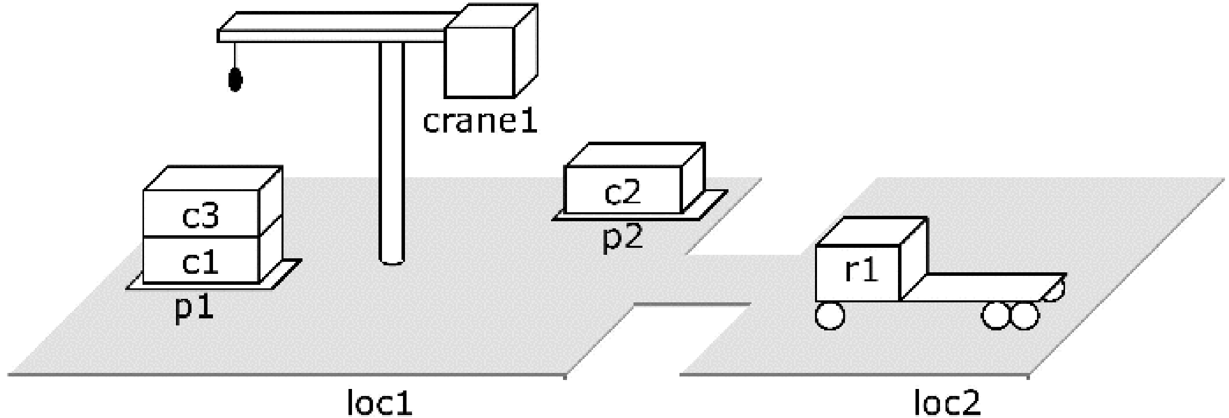
- Let $P_1 = (O, s_1, g_1)$, where
  - O is the set of operators given earlier
  - g_1={loaded(r1,c3), at(r1,loc2)}





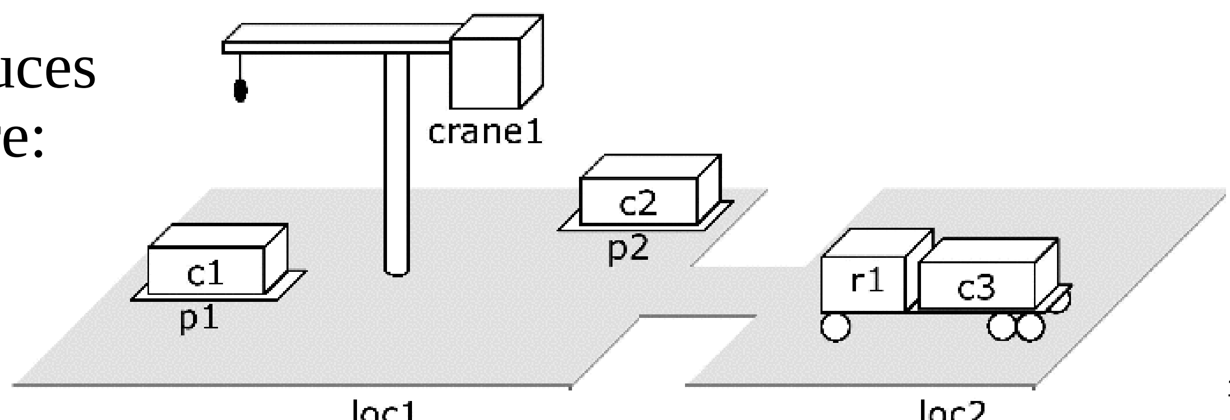
- s_1={attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

Example (continued)

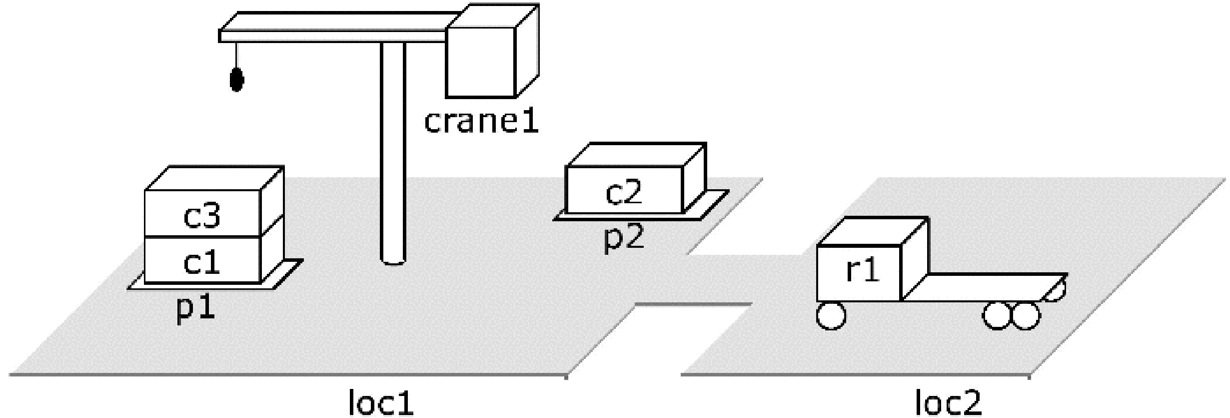


- Here are three solutions for P_1:
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

- Each of them produces the state shown here:

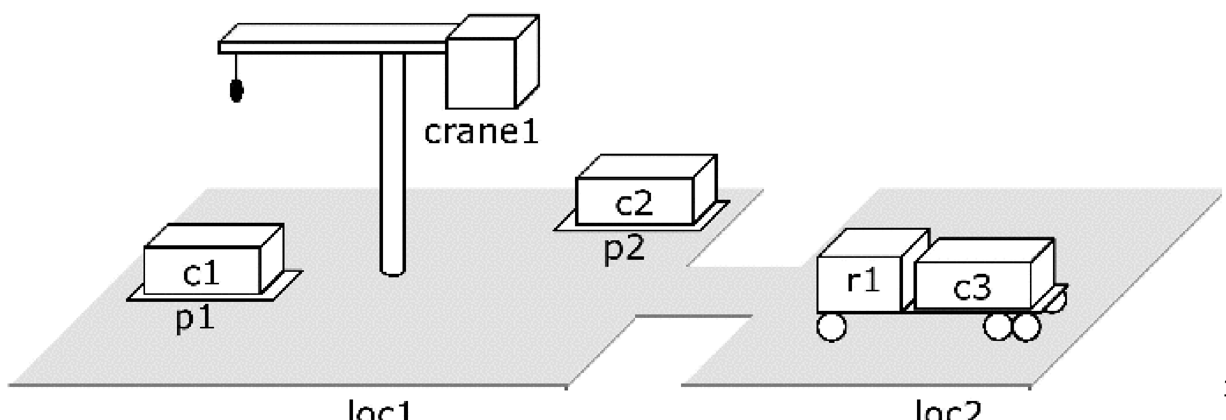


Example (continued)



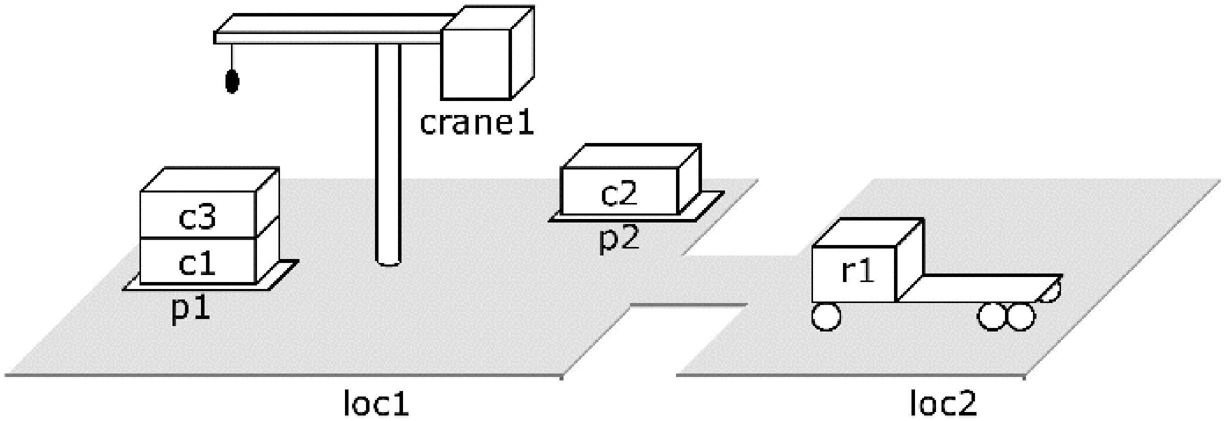
- The first is *redundant*: can remove actions and still have a solution
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), move(r1,loc1,loc2), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨take(crane1,loc1,c3,c1,p1), move(r1,loc2,loc1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩
  - ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1), load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

- The 2nd and 3rd are *irredundant* and *shortest*



Set-Theoretic Representation

- Like classical representation, but restricted to propositional logic



- States:
  - Instead of a collection of ground atoms …

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

  … use a collection of propositions (boolean variables):

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

- Instead of operators like this one,

$\mathsf{take}(k, l, c, d, p)$
;; crane k at location l takes c off of d in pile p
precond: $\mathsf{belong}(k, l), \mathsf{attached}(p, l), \mathsf{empty}(k), \mathsf{top}(c, p), \mathsf{on}(c, d)$
effects: $\mathsf{holding}(k, c), \neg\,\mathsf{empty}(k), \neg\,\mathsf{in}(c, p), \neg\,\mathsf{top}(c, p), \neg\,\mathsf{on}(c, d), \mathsf{top}(d, p)$

take all of the operator instances,

e.g., this one,

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,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)

and rewrite ground atoms as propositions

take-crane1-loc1-c3-c1-p1
precond: belong-crane1-loc1, attached-p1-loc1,
empty-crane1, top-c3-p1, on-c3-c1
delete: empty-crane1, in-c3-p1, top-c3-p1, on-c3-p1
add: holding-crane1-c3, top-c1-p1

Comparison

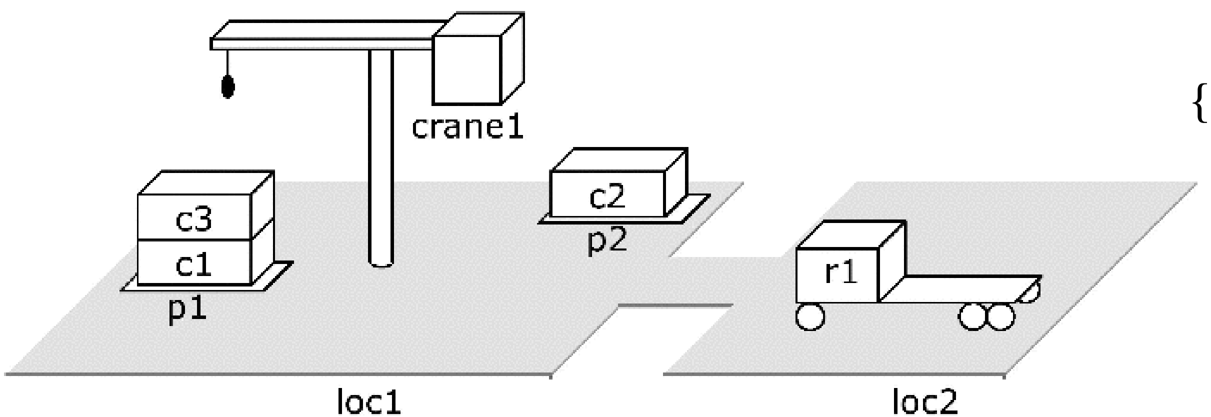
- A set-theoretic representation is equivalent to a classical representation in which all of the atoms are ground

- Exponential blowup
  - If a classical operator contains n atoms
    and each atom has arity k,
    then it corresponds to c^{nk} actions where $c = |\{\text{constant symbols}\}|$

State-Variable Representation

- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to *state variables*
  - Like fields in a record structure
- Classical and state-variable representations take similar amounts of space
  - Each can be translated into the other in low-order polynomial time

move(r, l, m)
 ;; robot r at location l moves to an adjacent location m
 precond: $\text{rloc}(r) = l, \text{adjacent}(l, m)$
 effects: $\text{rloc}(r) \leftarrow m$

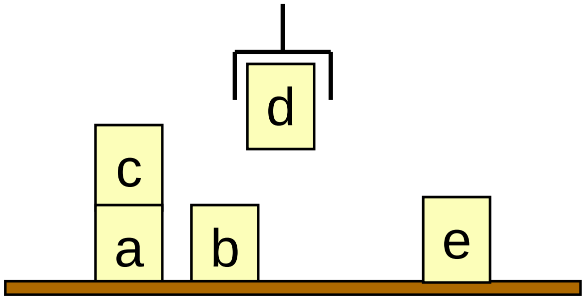


{top(p1)=c3,
cpos(c3)=c1,
cpos(c1)=pallet,
holding(crane1)=nil,
rloc(r1)=loc2,
loaded(r1)=nil, …}

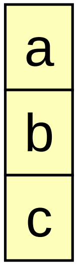
Example: The Blocks World

- Infinitely wide table, finite number of blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- Want to move blocks from one configuration to another
  - e.g.,

initial state



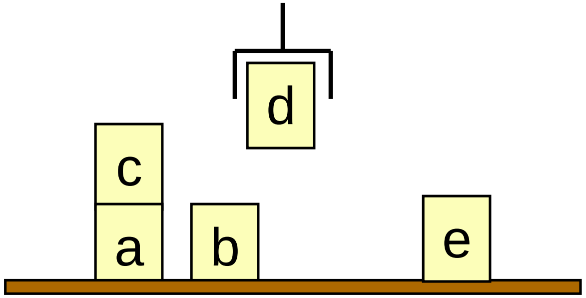
goal



- Can be expressed as a special case of DWR
  - But the usual formulation is simpler
- I'll give classical, set-theoretic, and state-variable formulations
  - For the case where there are five blocks

Classical Representation: Symbols

- Constant symbols:
  - The blocks: a, b, c, d, e
- Predicates:
  - ontable(*x*) - block *x* is on the table
  - on(*x*,*y*) - block *x* is on block *y*
  - clear(*x*) - block *x* has nothing on it
  - holding(*x*) - the robot hand is holding block *x*
  - handempty - the robot hand isn't holding anything



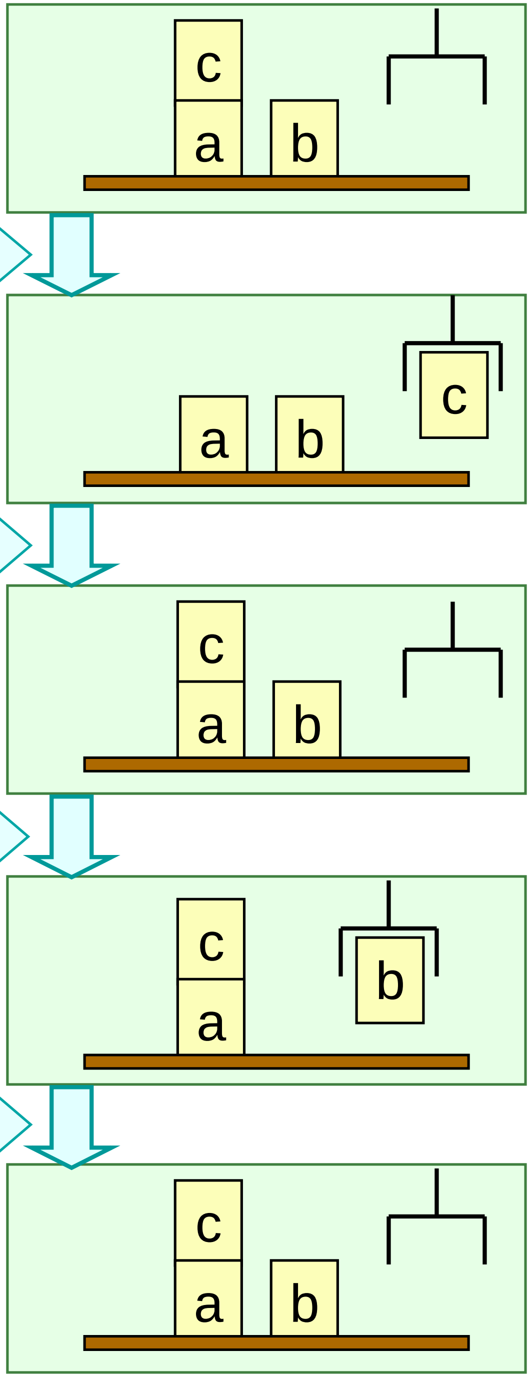
Classical Operators

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)

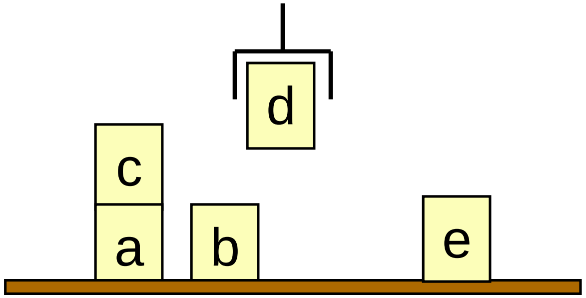
putdown(x)
Precond: holding(x)
Effects: ~holding(x), ontable(x), clear(x), handempty



Set-Theoretic Representation: Symbols

- For five blocks, there are 36 propositions
- Here are 5 of them:

ontable-a	- block a is on the table
on-c-a	- block c is on block a
clear-c	- block c has nothing on it
holding-d	- the robot hand is holding block d
handempty	- the robot hand isn't holding anything



Set-Theoretic Actions

Fifty different actions

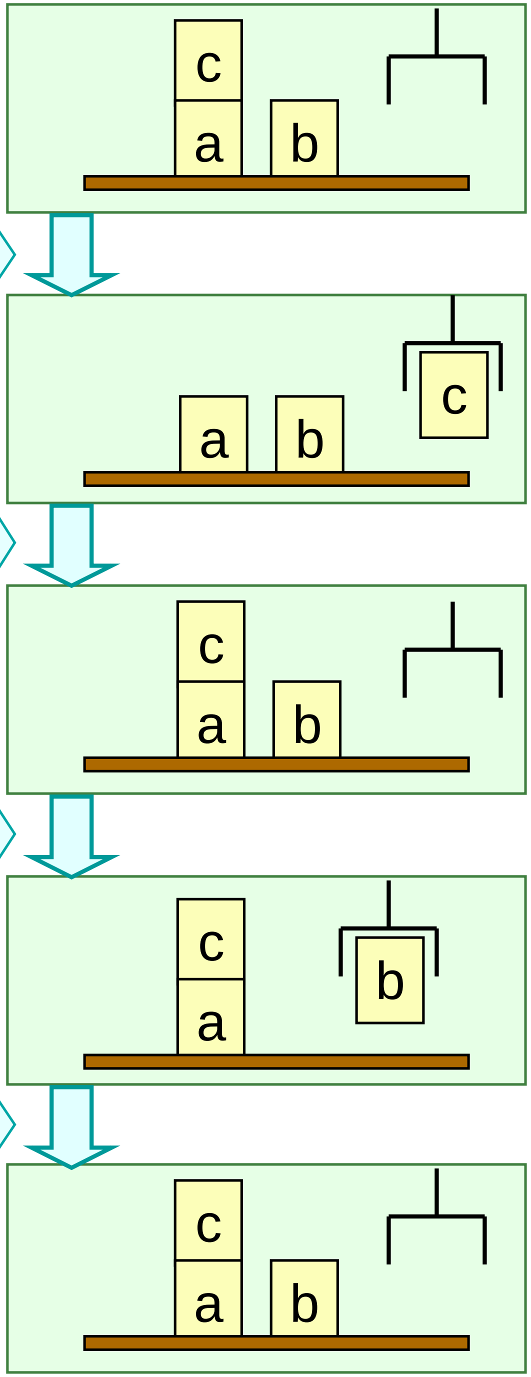
Here are four of them:

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-c
- Pre: ontable-c, clear-c, handempty
- Del: ontable-c, clear-c, handempty
- Add: holding-c

putdown-c
- Pre: holding-c
- Del: holding-c
- Add: ontable-c, clear-c, 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

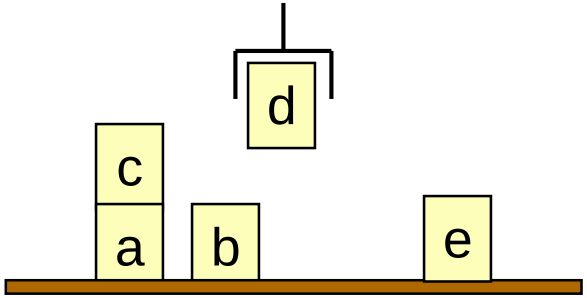
  pos(*x*) = nil if block *x* is being held

  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



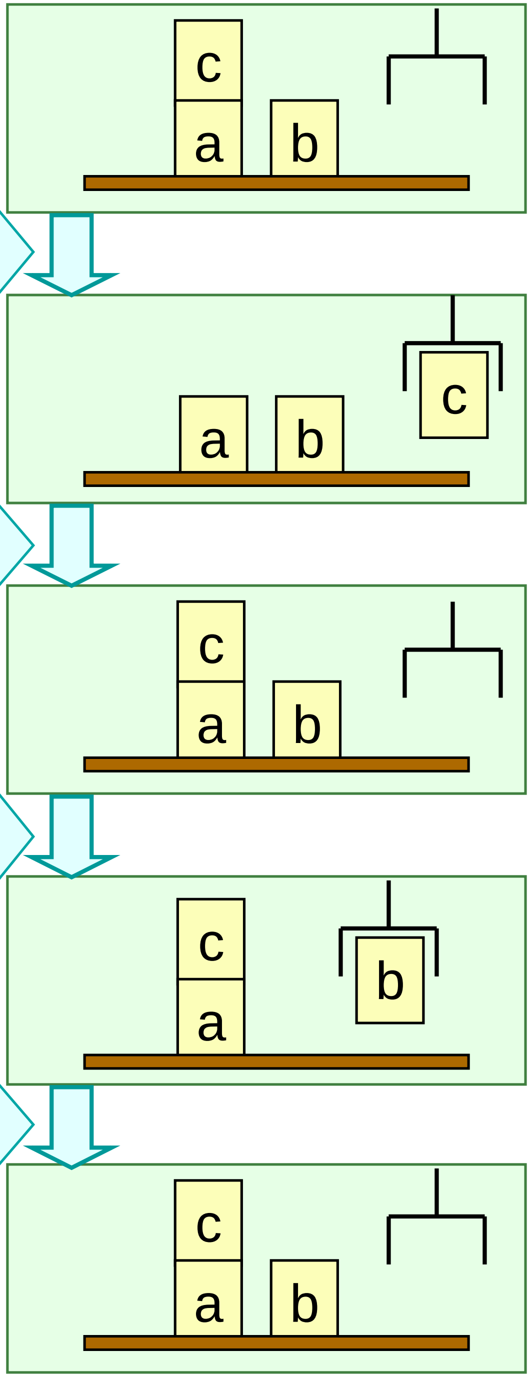
State-Variable Operators

unstack(*x* : block, *y* : block)
Precond: pos(*x*)=*y*, clear(*y*)=0, clear(*x*)=1, holding=nil
Effects: pos(*x*)=nil, clear(*x*)=0, holding=*x*, clear(*y*)=1

stack(*x* : block, *y* : block)
Precond: holding=*x*, clear(*x*)=0, clear(*y*)=1
Effects: holding=nil, clear(*y*)=0, pos(*x*)=y, clear(*x*)=1

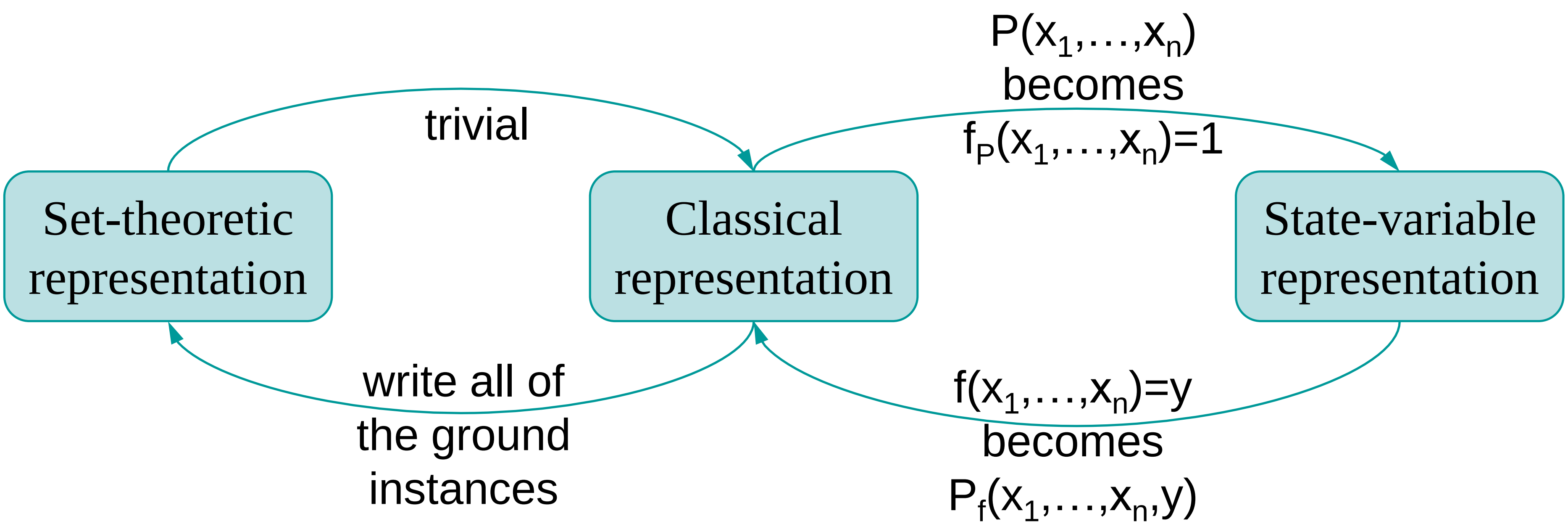
pickup(*x* : block)
Precond: pos(*x*)=table, clear(*x*)=1, holding=nil
Effects: pos(*x*)=nil, clear(*x*)=0, holding=*x*

putdown(*x* : block)
Precond: holding=*x*
Effects: holding=nil, pos(*x*)=table, clear(*x*)=1



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, except when converting to set-theoretic (where we get an exponential blowup)



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
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

Planning Algorithms

Motivation

- Nearly all planning procedures are search procedures
- Different planning procedures have different search spaces
- *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

Outline

- State-space planning
  - Forward search
  - Backward search
  - Lifting
  - STRIPS
  - Block-stacking

Forward-search(O, s_0, g)

$s \leftarrow s_0$

$\pi \leftarrow$ the empty plan

loop

if s satisfies g then return π

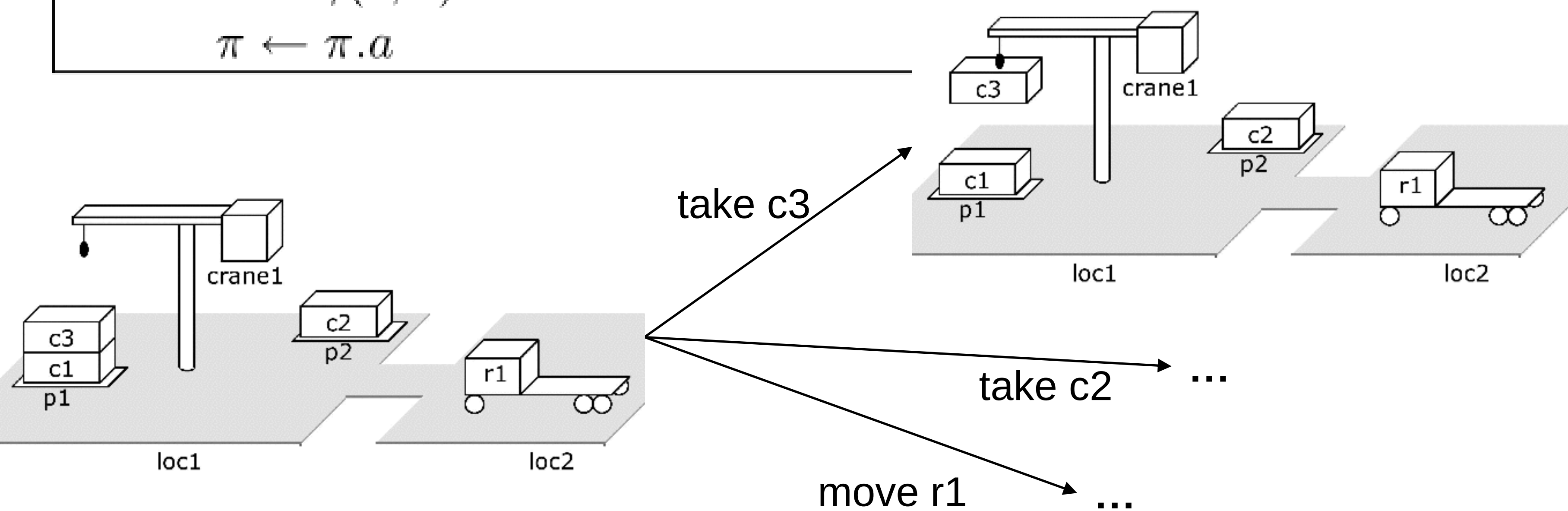
$E \leftarrow \{a | a$ is a ground instance an operator in O, and $\mathrm{precond}(a)$ is true in $s\}$

if $E = \emptyset$ then return failure

nondeterministically choose an action $a \in E$

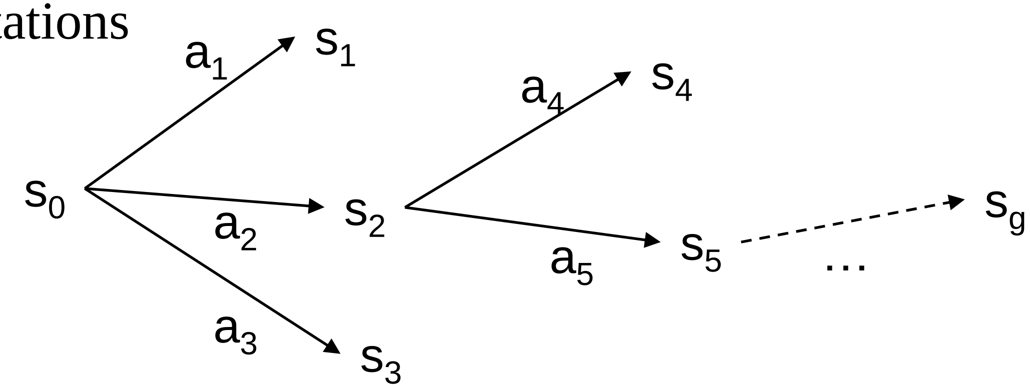
$s \leftarrow \gamma(s, a)$

$\pi \leftarrow \pi.a$



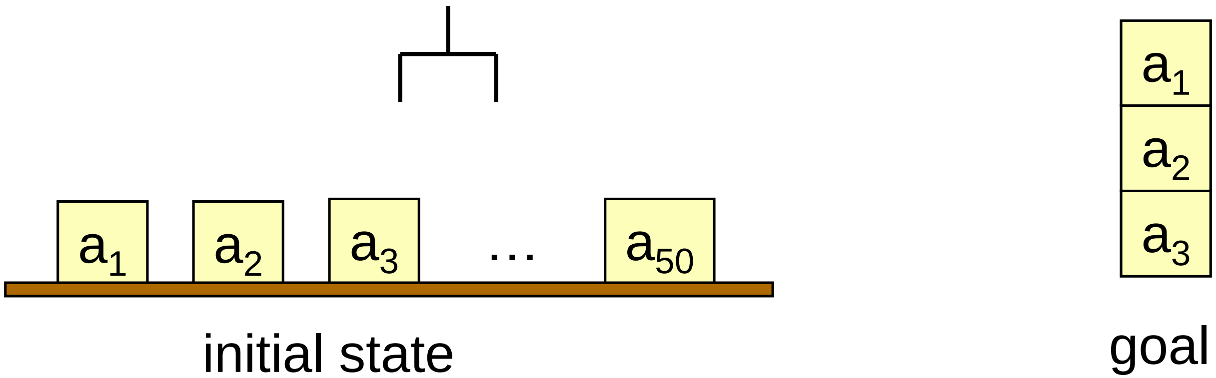
Deterministic Implementations

- 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
  - 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

Backward Search

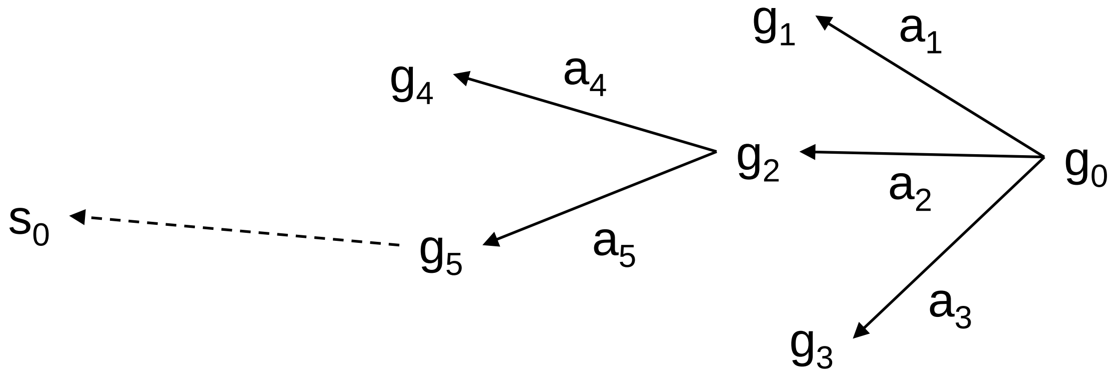
- For forward search, we started at the initial state and computed state transitions
  - new state = $\gamma(s,a)$
- For backward search, we start at the goal and compute inverse state transitions
  - new set of subgoals = $\gamma^{-1}(g,a)$
- To define $\gamma^{-1}(g,a)$, must first define *relevance*:
  - 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

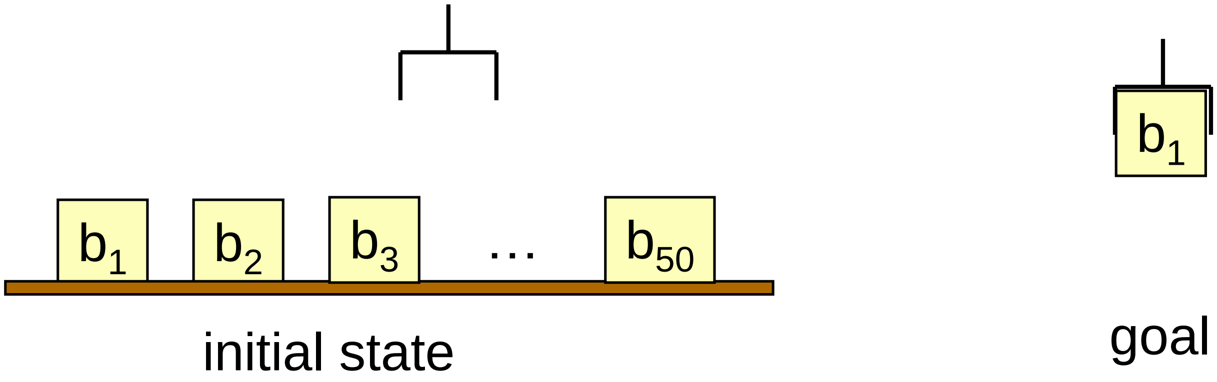
- If a is relevant for g, then
  - $\gamma^{-1}(g,a) = (g - \text{effects}(a)) \cup \text{precond}(a)$
- Otherwise $\gamma^{-1}(g,a)$ is undefined

- 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



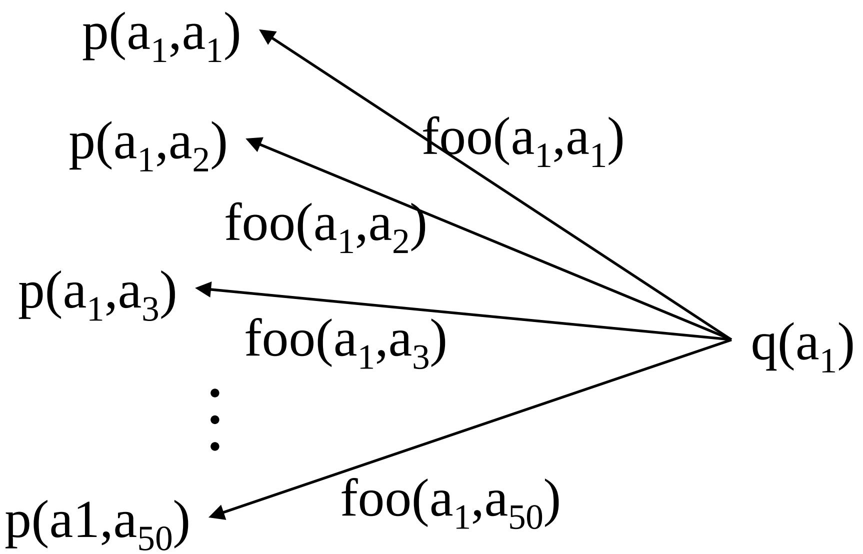
- Backward search can *also* have a very large branching factor
  - E.g., an operator o that is relevant for g may have many ground instances a_1, a_2, …, a_n such that each a_i's input state might be unreachable from the initial state
- As before, deterministic implementations can waste lots of time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

Lifting

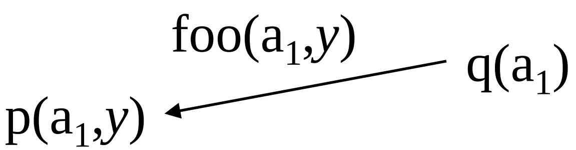
foo(x,y)

precond: p(x,y)

effects: q(x)



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



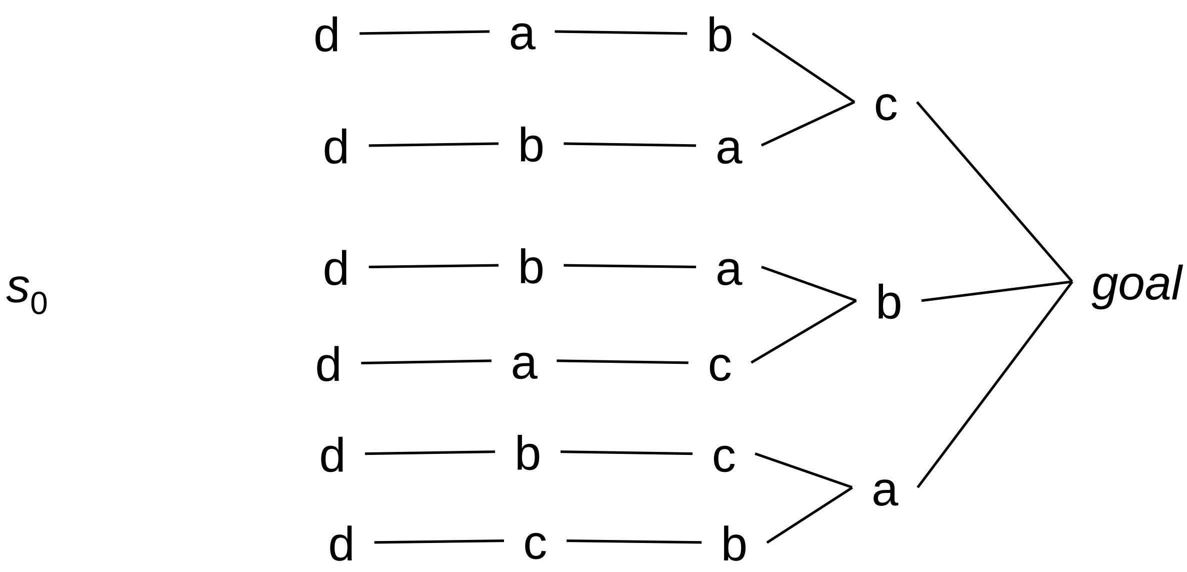
Lifted Backward Search

- 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

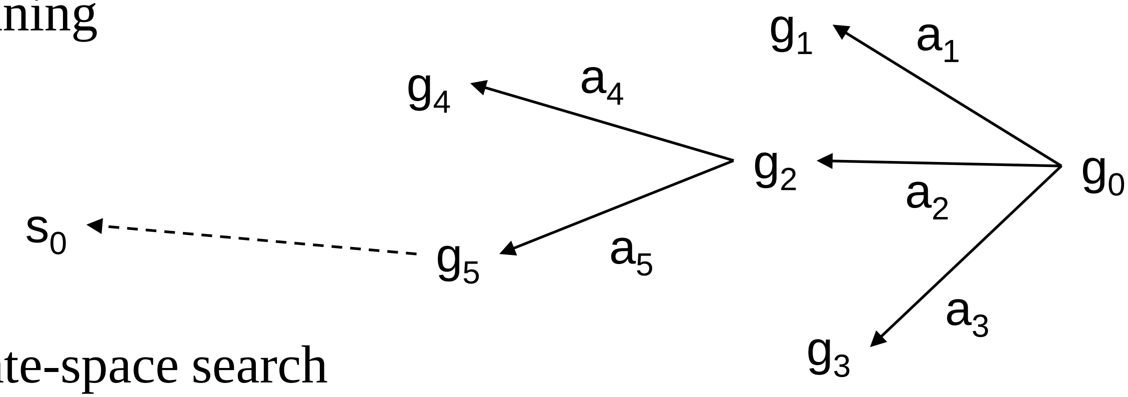
- Lifted-backward-search generates a smaller search space than Backward-search, but it still can be quite large
  - Suppose actions *a*, *b*, and *c* are independent, action *d* must precede all of them, and there's no path from s_0 to *d*'s input state
  - We'll try all possible orderings of *a*, *b*,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



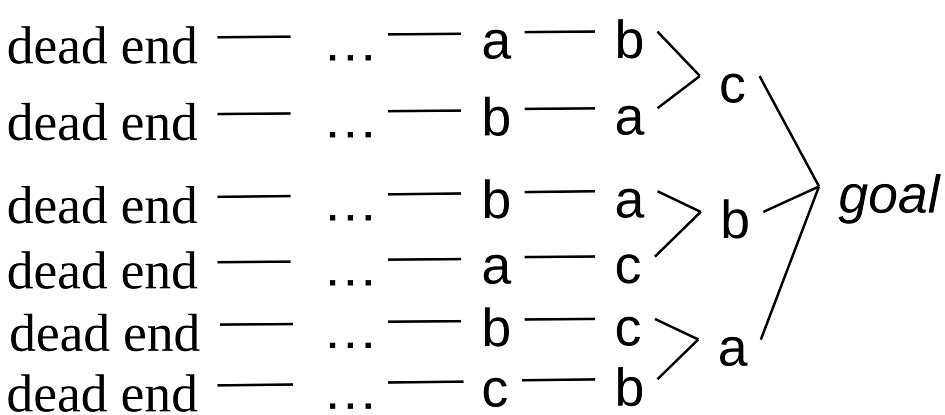
Plan-space planning

Motivation

- State-Space Planning



- Problem with state-space search
  - In some cases we may try many different orderings of the same actions before realizing there is no solution



- *Least-commitment strategy:* don’t commit to orderings, instantiations, etc., until necessary

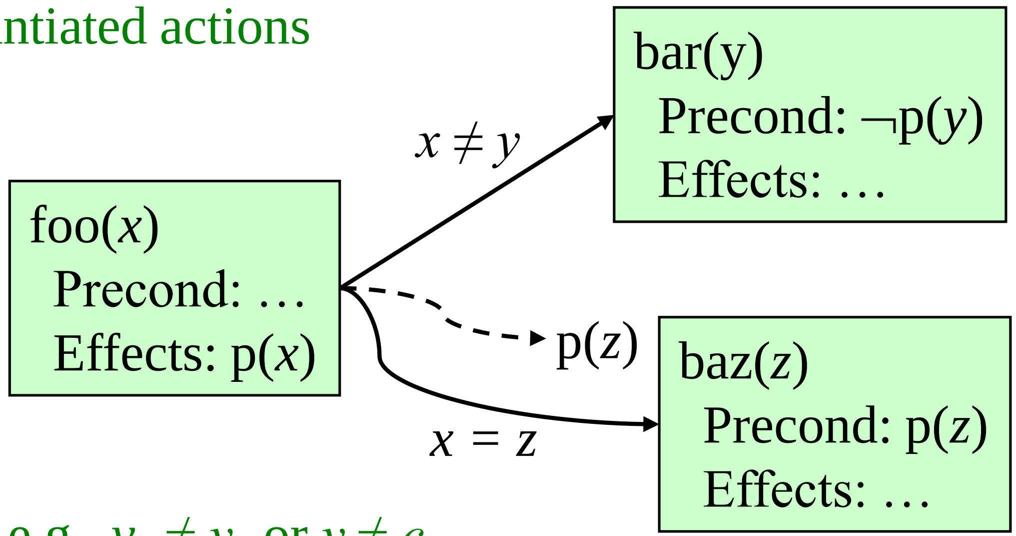
Outline

- Basic idea
- Open goals
- Threats
- The PSP algorithm
- Long example
- Comments

Plan-Space Planning - Basic Idea

- Backward search from the goal

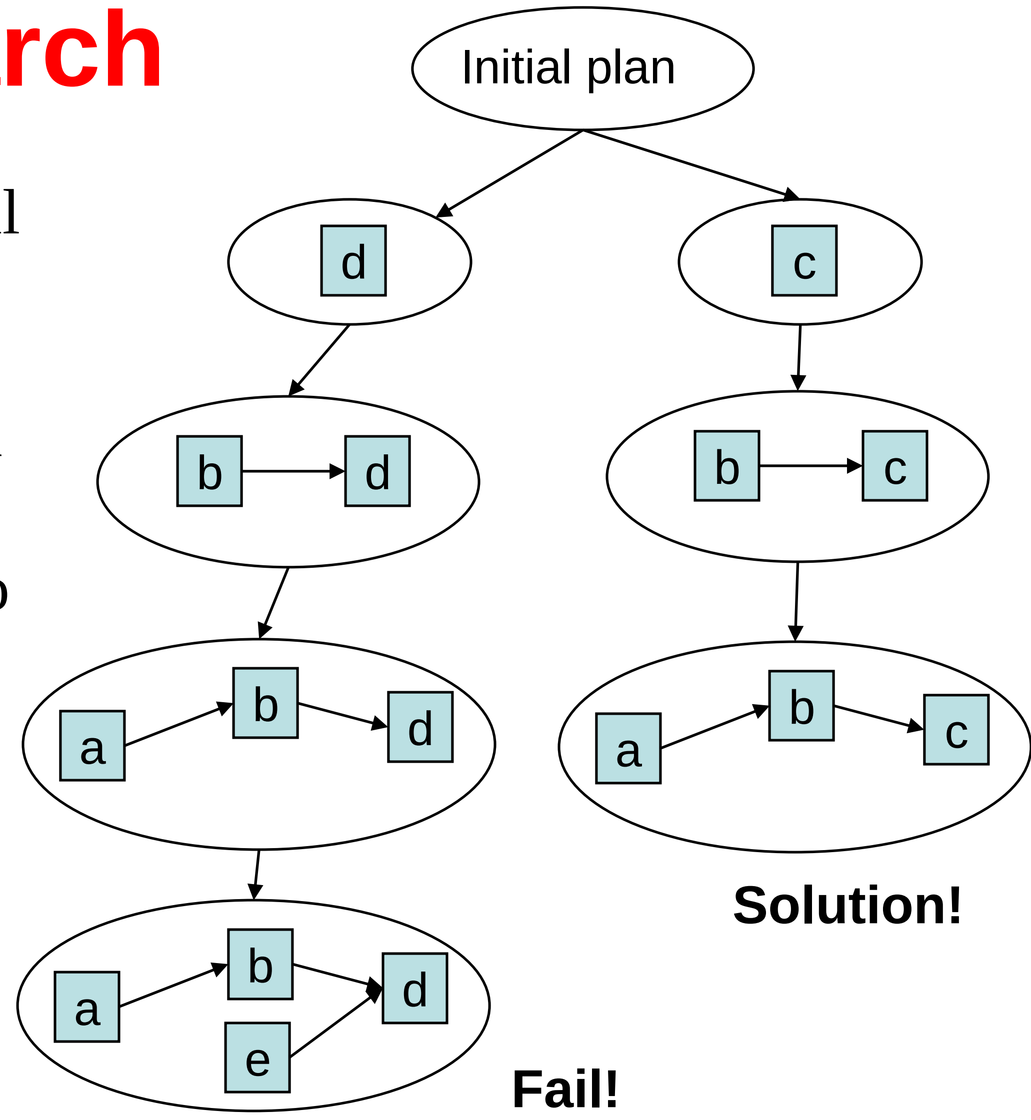
- Each node of the search space is a *partial plan*
  - » A set of partially-instantiated actions
  - » A set of constraints



- Types of constraints:
  - ◆ *precedence constraint*: *a* must precede *b*
  - ◆ *binding constraints*:
    - » inequality constraints, e.g., $v_1 \neq v_2$ or $v \neq c$
    - » equality constraints (e.g., $v_1 = v_2$ or $v = c$) and/or substitutions
  - ◆ *causal link*:
    - » use action *a* to establish the precondition *p* needed by action *b*

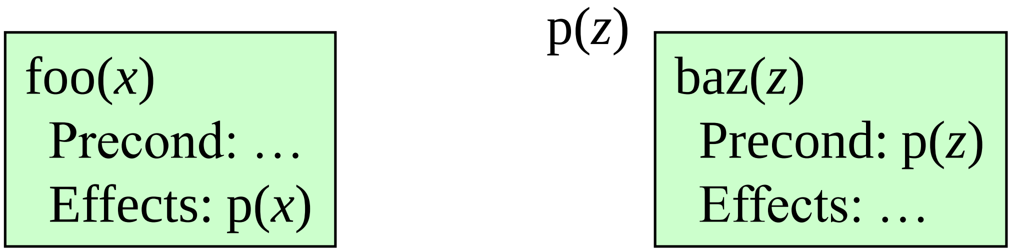
Plan-Space Search

- Start with an initial plan (will explain this later)
- Make more and more refinements to the plan, until a solution is found
- If there are no refinements to a plan during search, then backtrack
- Questions:
  - How to tell we have a solution?
    - » No more *flaws* in the plan
  - How to refine a plan?
    - » Resolve the existing *flaws*

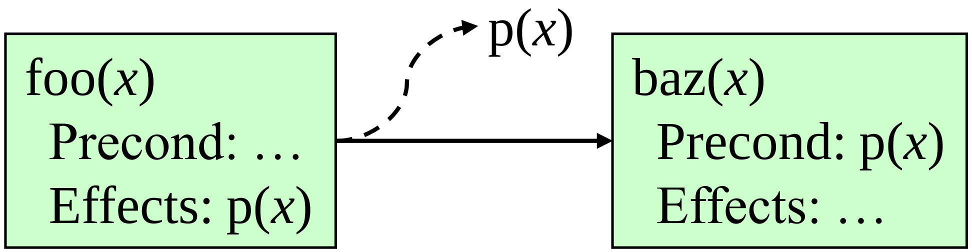


Flaws: 1. Open Goals

- Open goal:
  - An action a has a precondition p that we haven't decided how to establish

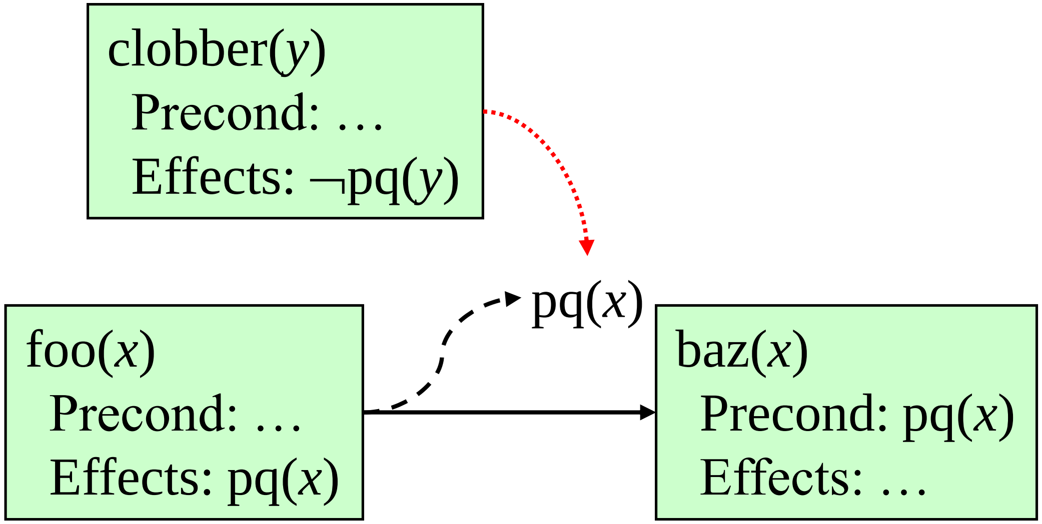


- Resolving the flaw:
  - Find an action b
    - (either already in the plan, or insert it)
  - that can be used to establish p
    - can precede a and produce p
  - Instantiate variables and/or constrain variable bindings
  - Create a causal link



Flaws: 2. Threats

- Threat: a deleted-condition interaction
  - Action *a* establishes a precondition (e.g., pq(*x*)) of action *b*
  - Another action *c* is capable of deleting *pq*
- Resolving the flaw:
  - impose a constraint to prevent *c* from deleting *pq*
- Three possibilities:
  - Make *b* precede *c*
  - Make *c* precede *a*
  - Constrain variable(s) to prevent *c* from deleting *pq*



The PSP Procedure

PSP(π)

$\quad$ $flaws \leftarrow$ OpenGoals(π) $\cup$ Threats(π)

$\quad$ if $flaws = \emptyset$ then return(π)

$\quad$ select any flaw $\phi \in flaws$

$\quad$ $resolvers \leftarrow$ Resolve(ϕ, π)

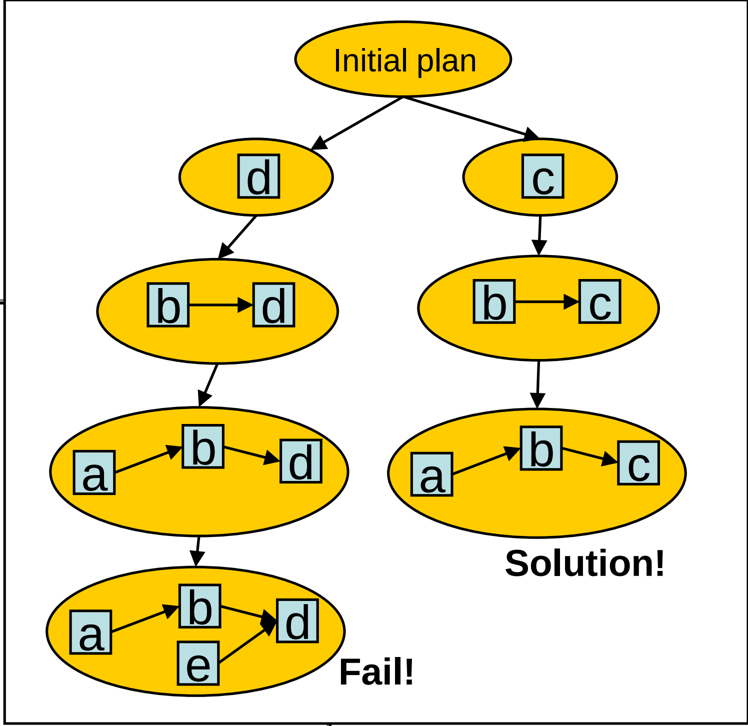
$\quad$ if $resolvers = \emptyset$ then return(failure)

$\quad$ nondeterministically choose a resolver $\rho \in resolvers$

$\quad$ $\pi' \leftarrow$ Refine(ρ, π)

$\quad$ return(PSP(π'))

end



- PSP is both sound and complete
- It returns a partially ordered solution plan
  - Any total ordering of this plan will achieve the goals
  - Or could execute actions in parallel if the environment permits it

Example

- Similar (but not identical) to an example in Russell and Norvig's *Artificial Intelligence: A Modern Approach* (1st edition)
- Operators:
  - **Start**

    Precond: none

    Effects: At(Home), sells(HWS,Drill), Sells(SM,Milk), Sells(SM,Banana)
  - **Finish**

    Precond: Have(Drill), Have(Milk), Have(Banana), At(Home)

    Effects: none
  - **Go(*l*,*m*)**

    Precond: At(*l*)

    Effects: At(*m*), ¬At(*l*)
  - **Buy(*p*,*s*)**

    Precond: At(*s*), Sells(*s*,*p*)

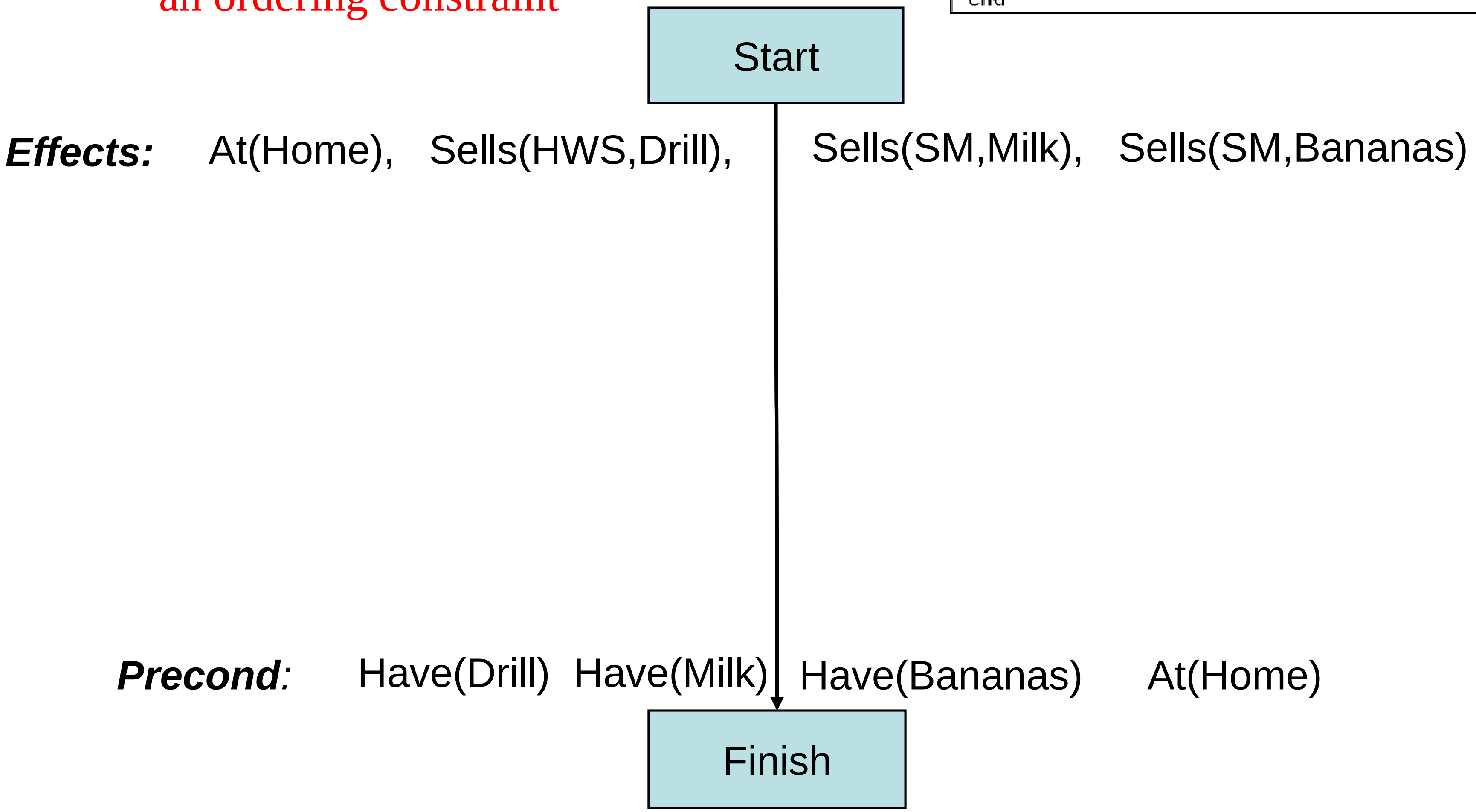
    Effects: Have(*p*)

Start and **Finish** are dummy actions that we'll use instead of the initial state and goal

Example (continued)

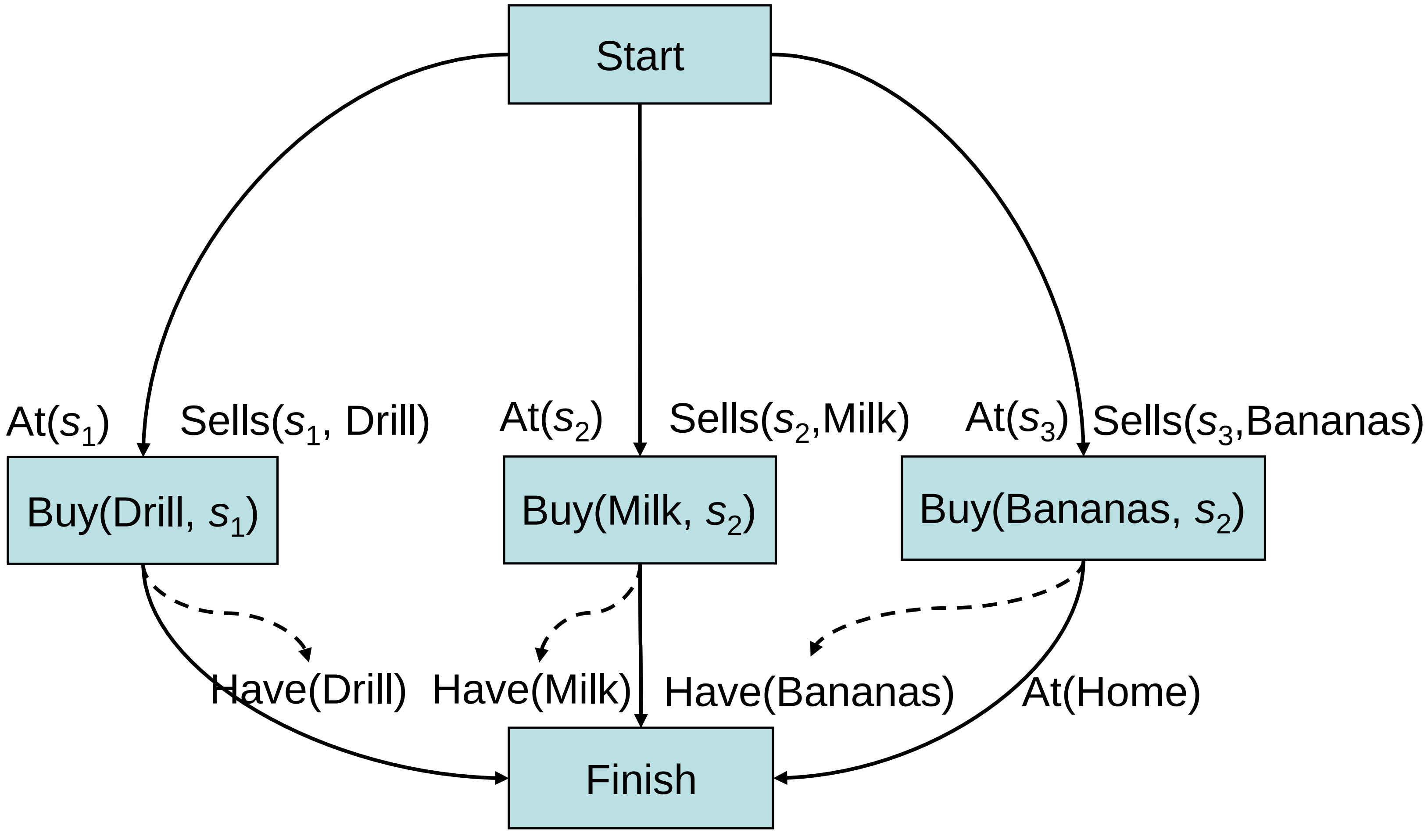
```
PSP(π)
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then return(π)
    select any flaw ϕ ∈ flaws
    resolvers ← Resolve(ϕ, π)
    if resolvers = ∅ then return(failure)
    nondeterministically choose a resolver ρ ∈ resolvers
    π′ ← Refine(ρ, π)
    return(PSP(π′))
end
```

- Need to give PSP a plan π as its argument
  - Initial plan: **Start**, **Finish**, and an ordering constraint



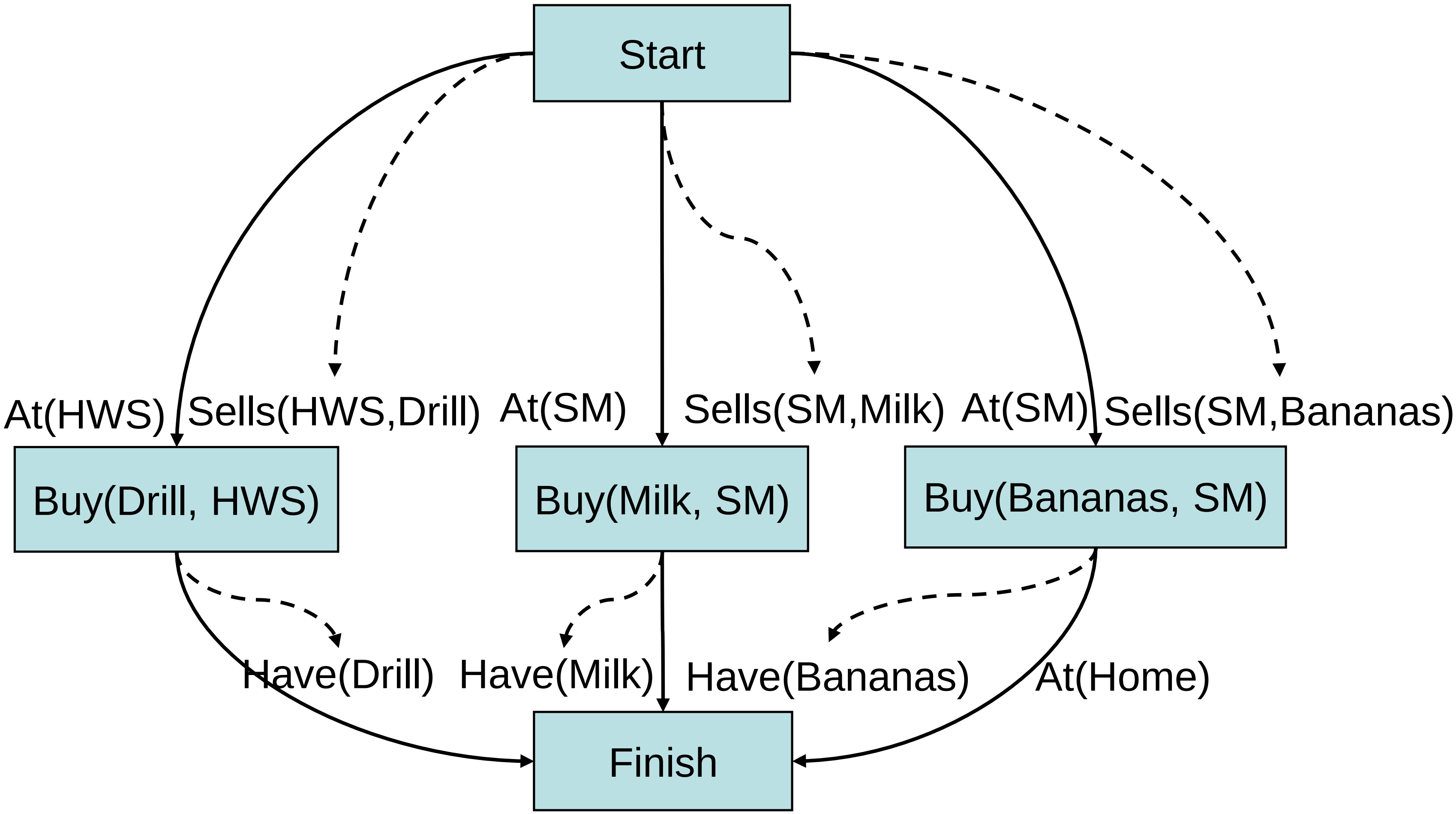
Example (continued)

- The first three refinement steps
  - These are the only possible ways to establish the Have preconditions



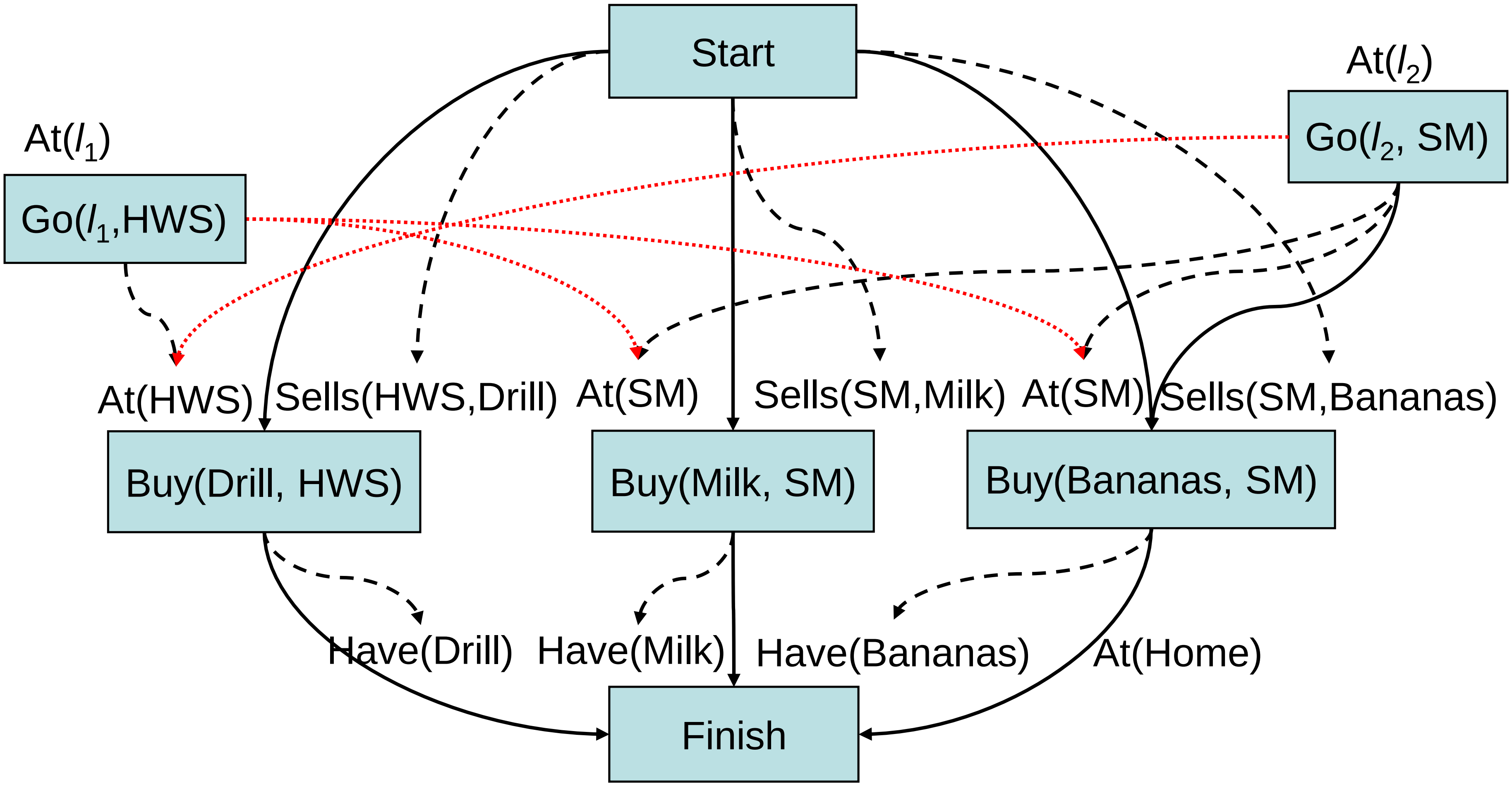
Example (continued)

- Three more refinement steps
  - The only possible ways to establish the Sells preconditions



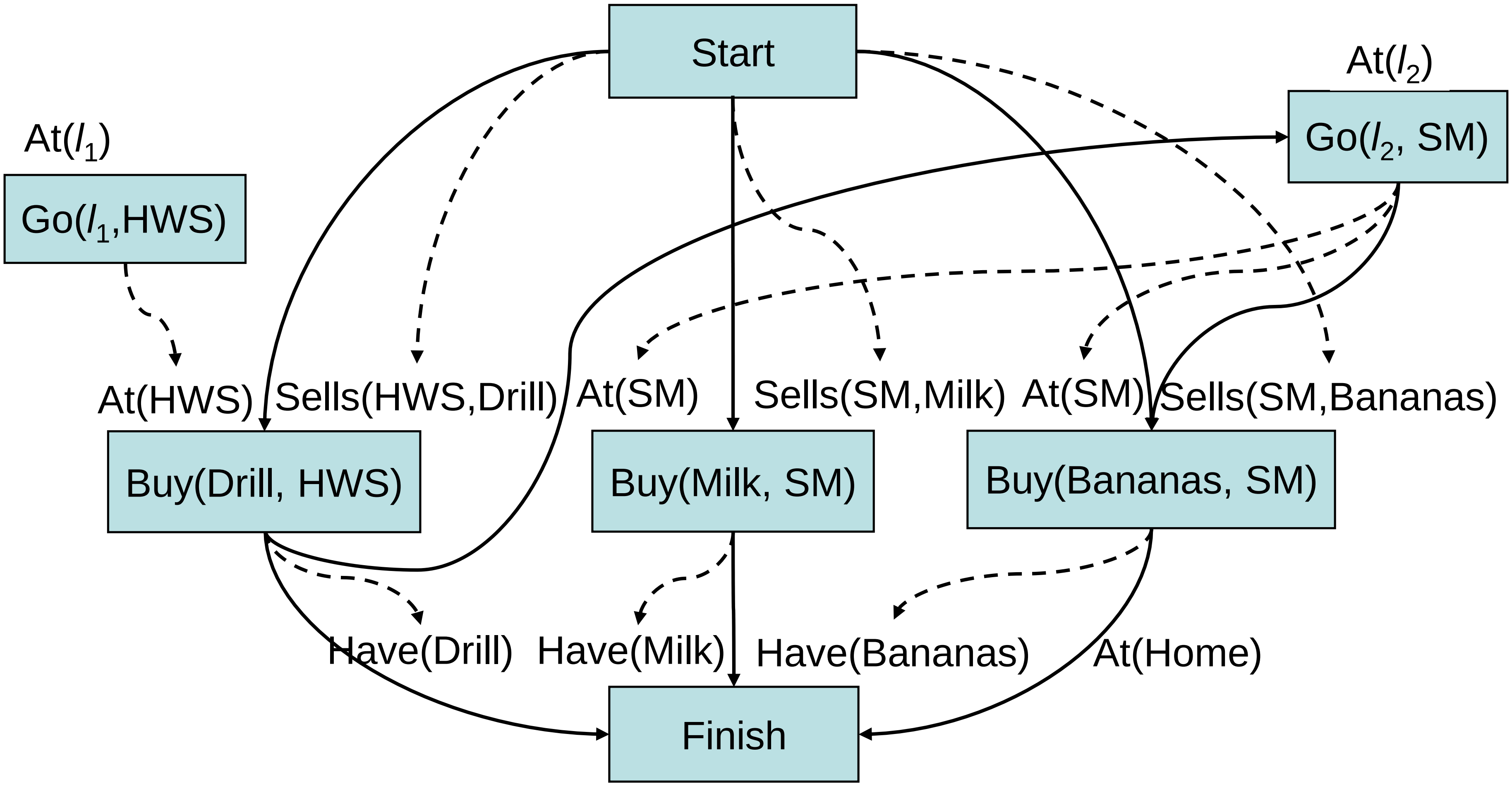
Example (continued)

- Two more refinements: the only ways to establish At(HWS) and At(SM)
  - This time, several threats occur



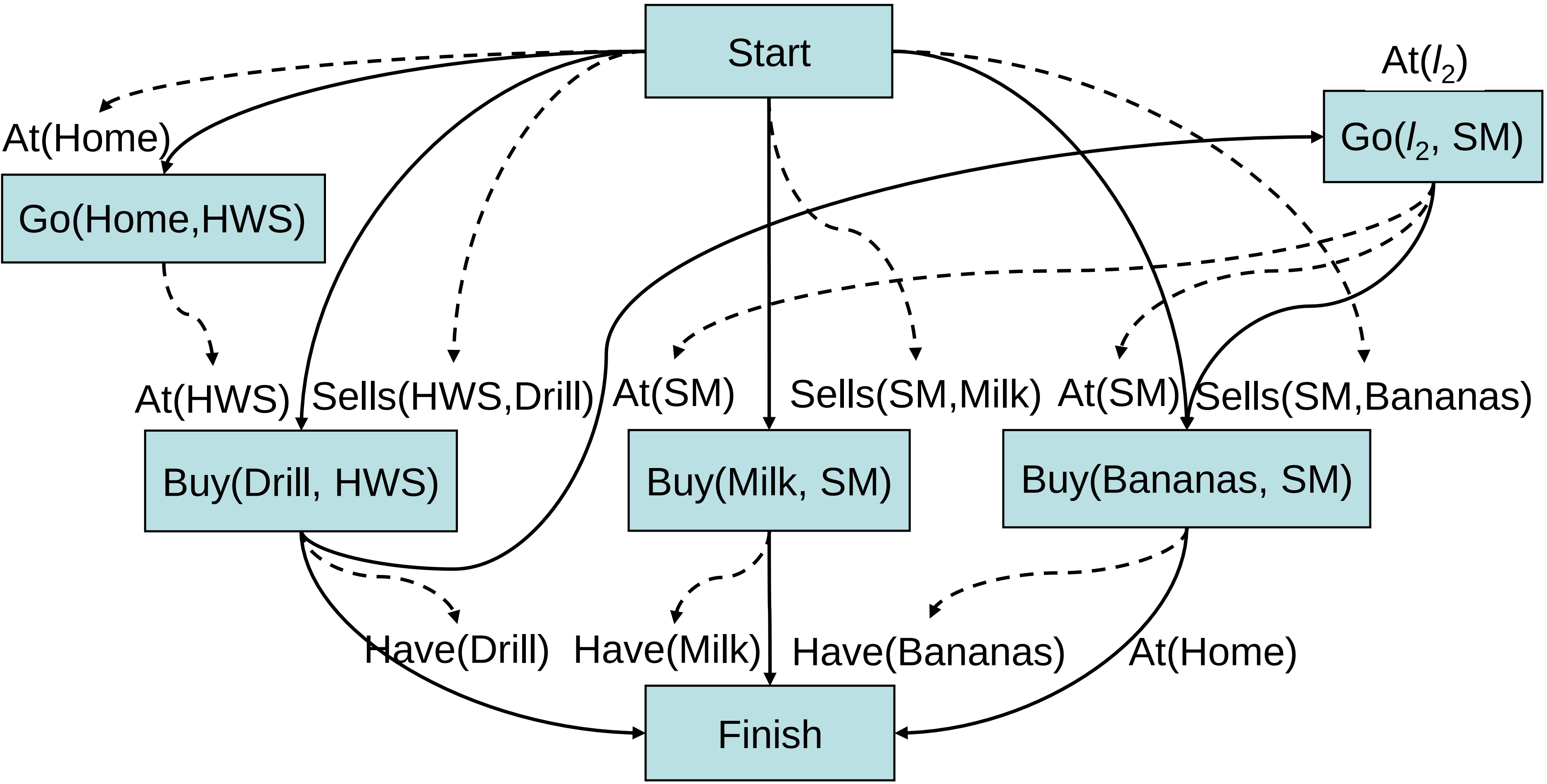
Example (continued)

- Finally, a nondeterministic choice: how to resolve the threat to At(s_1)?
  - Our choice: make Buy(Drill) precede Go(SM)
  - This also resolves the other two threats



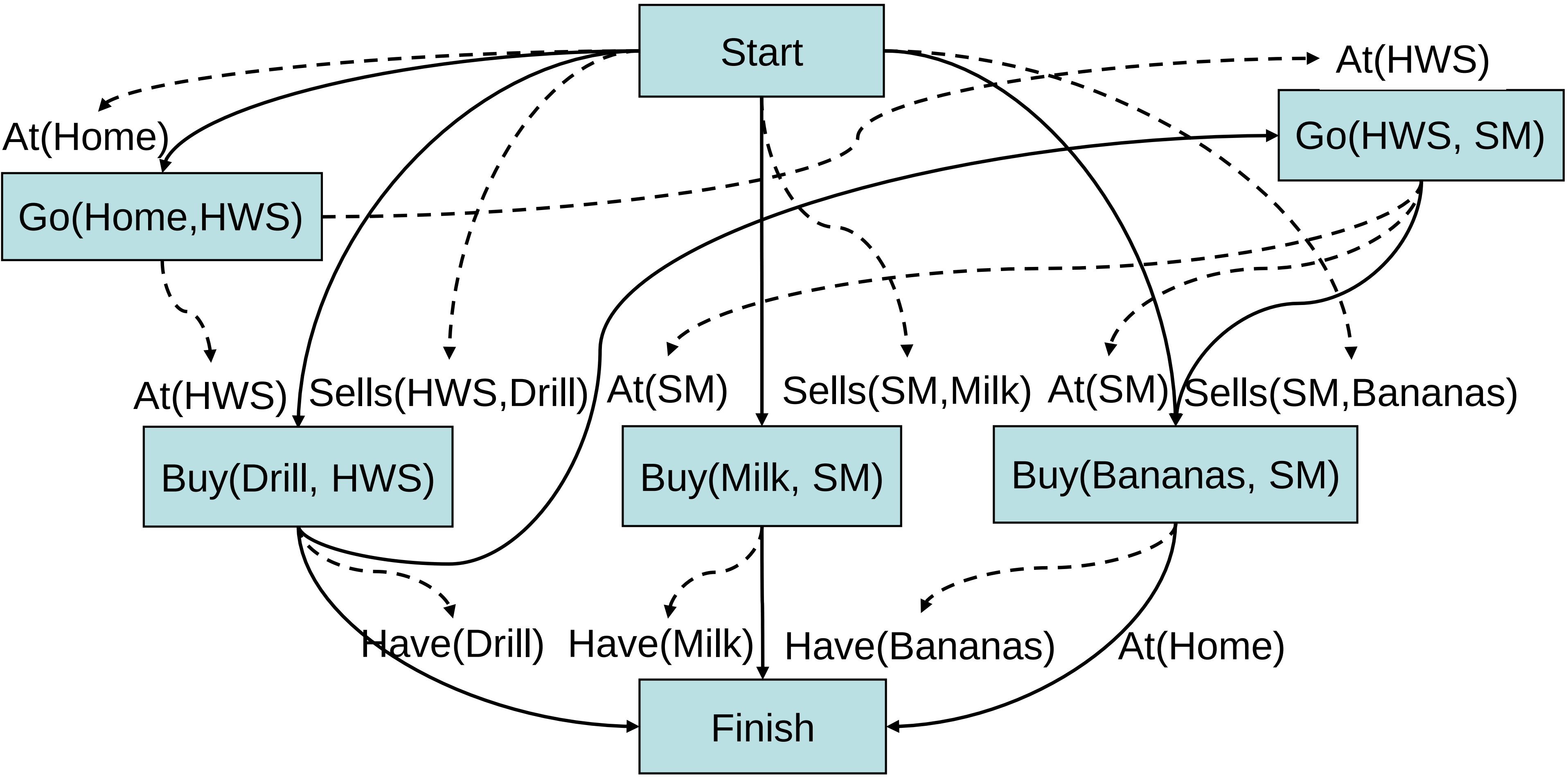
Example (continued)

- Nondeterministic choice: how to establish At(l_1)?
  - We'll do it from Start, with l_1=Home



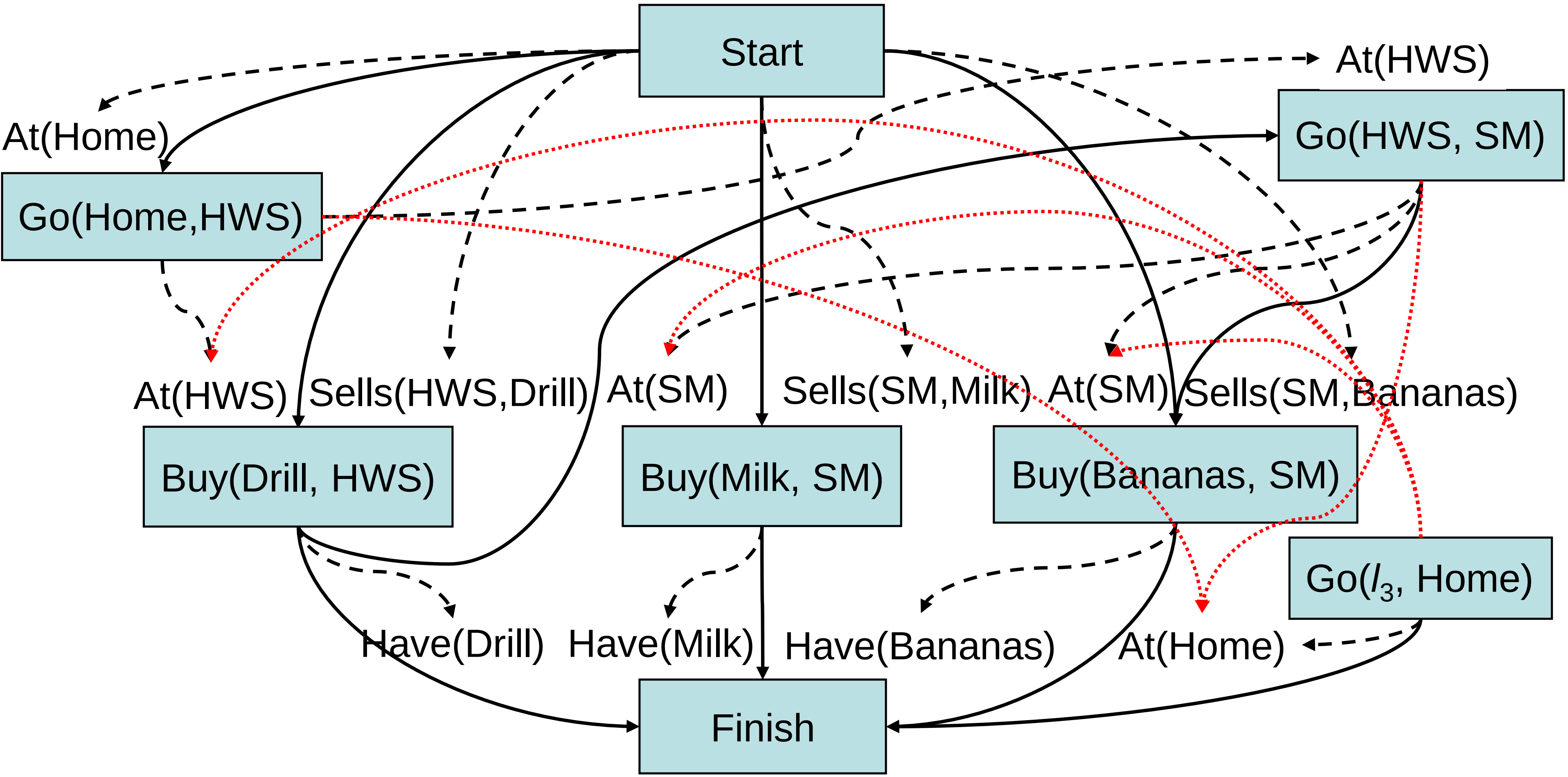
Example (continued)

- Nondeterministic choice: how to establish At(l_2)?
  - We'll do it from Go(Home,HWS), with l_2= HWS



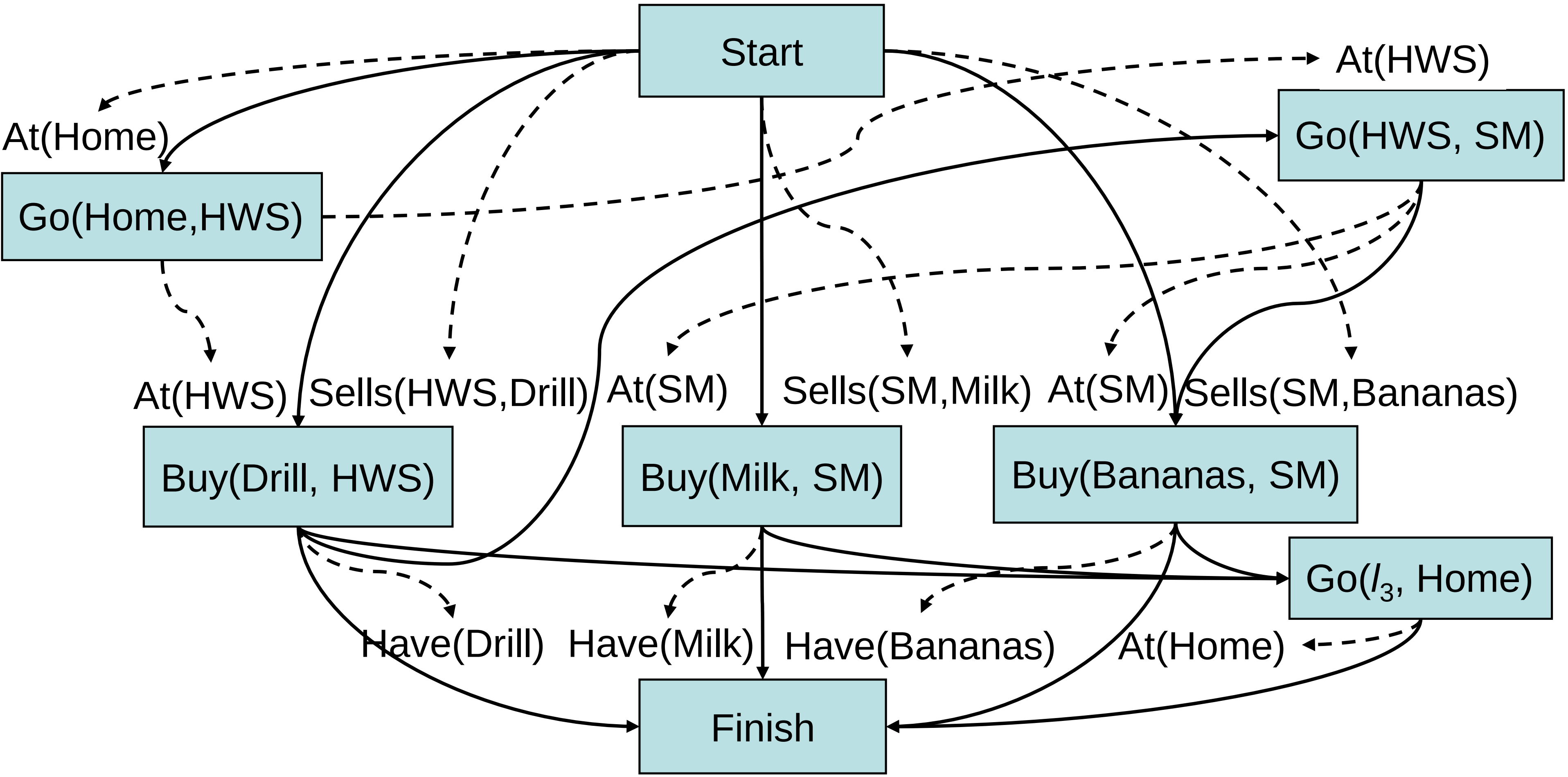
Example (continued)

- The only possible way to establish At(Home) for Finish
  - This creates a bunch of threats



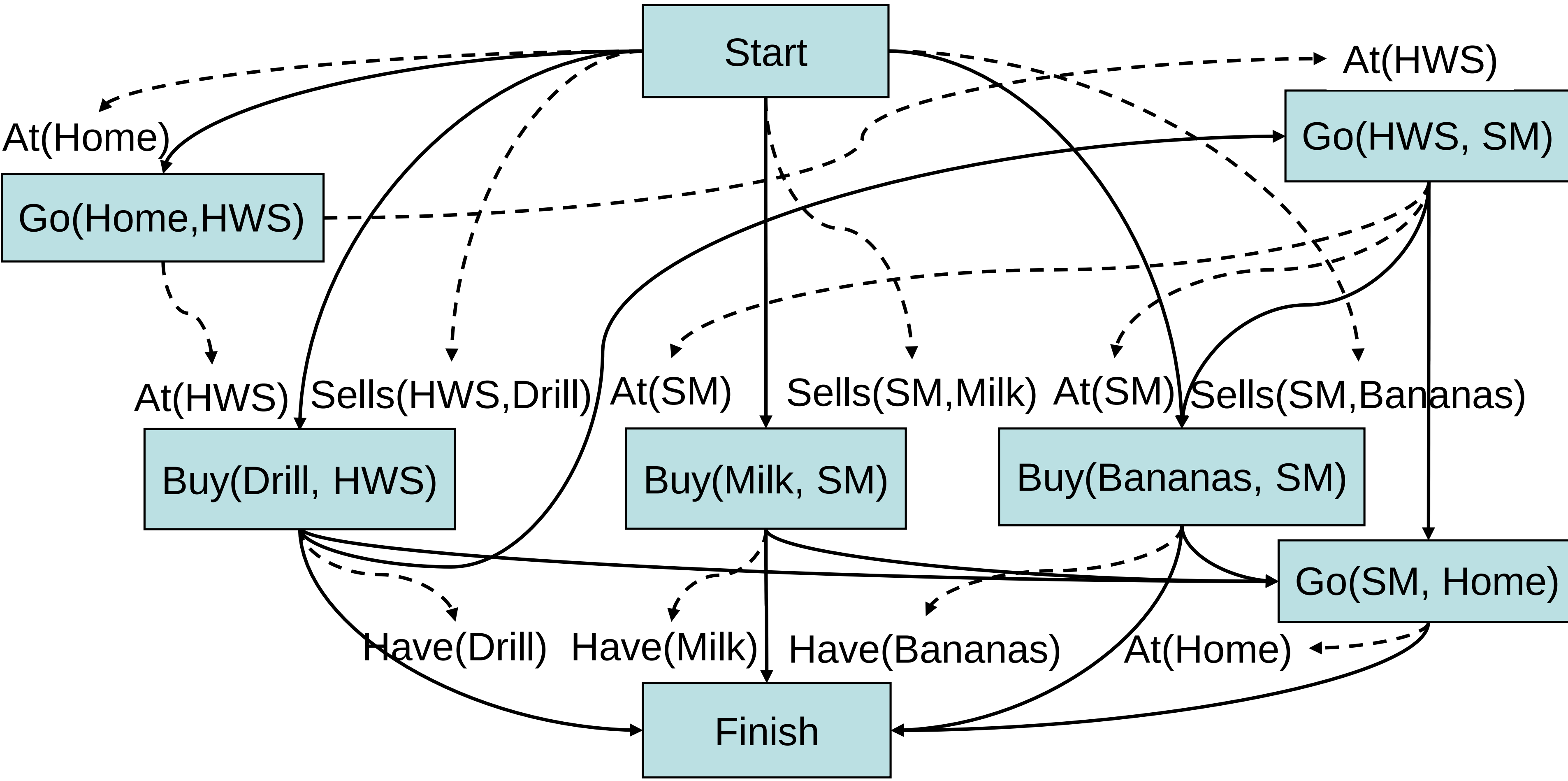
Example (continued)

- To remove the threats to At(SM) and At(HWS), make them precede Go(l_3,Home)
  - This also removes the other threats

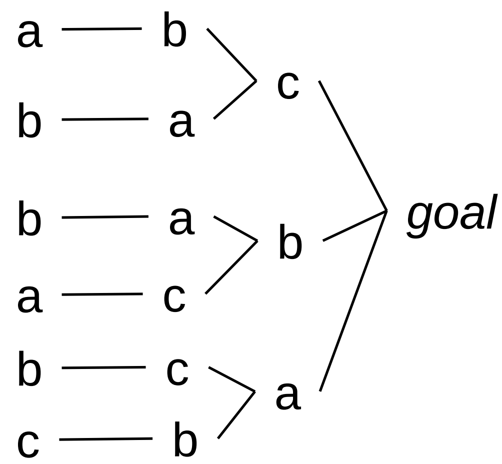


Final Plan

- Establish At(l_3) with l_3=SM



Discussion



- How to choose which flaw to resolve first and how to resolve it?
  - There are lots of things!!!
- PSP doesn't commit to orderings and instantiations until necessary
  - Avoids generating search trees like this one:
- Problem: how to prune infinitely long paths?
  - In state-space planning, loop detection is based on recognizing states we've seen before
  - In a partially ordered plan, we don't know the states
- Can we prune if we see the same *action* more than once?

  … — go(b,a) — go(a,b) – go(b,a) — at(a)

  - No. Sometimes we might need the same action several times in different states of the world (see next slide)

Example

- 3-digit binary counter starts at 000, want to get to 110

  s_0 = {d3=0, d2=0, d1=0}

  g = {d3=1, d2=1, d1=0}

- Operators to increment the counter by 1:

  incr0

  Precond: d_1=0

  Effects: d_1=1

  incr01

  Precond: d_2=0, d_1=1

  Effects: d_2=1, d_1=0

  incr011

  Precond: d_3=0, d_2=1, d_1=1

  Effects: d_3=1, d_2=0, d_1=0

A Weak Pruning Technique

- Can prune all partial plans of n or more actions, where $n = |\{\text{all possible states}\}|$
  - This doesn't help very much

- There's no good pruning technique for plan-space planning
  - Russell and Norvig's *Artificial Intelligence: A Modern Approach* describes a preliminary approach
  - In early 90's, researchers have looked at learning pruning rules during PSP planning

Thanks and Questions!