Outline

Planning systems

PDDL (Planning Domain Definition Language)

Planning algorithms

- Forward chaining
- Backward chaining
- Partial-order planning

Applications

Planning: Motivation

How to develop systems or 'agents'
that can make decisions on their own?

What is AI planning?

Planning is the task of finding a set of actions that will achieve a goal. A *planner* is a program that searches for a plan. It inputs a description of the world and the goals. The output is a plan. The simplest plan is a sequence of actions:

```
``do action1, do action2 ...''
```

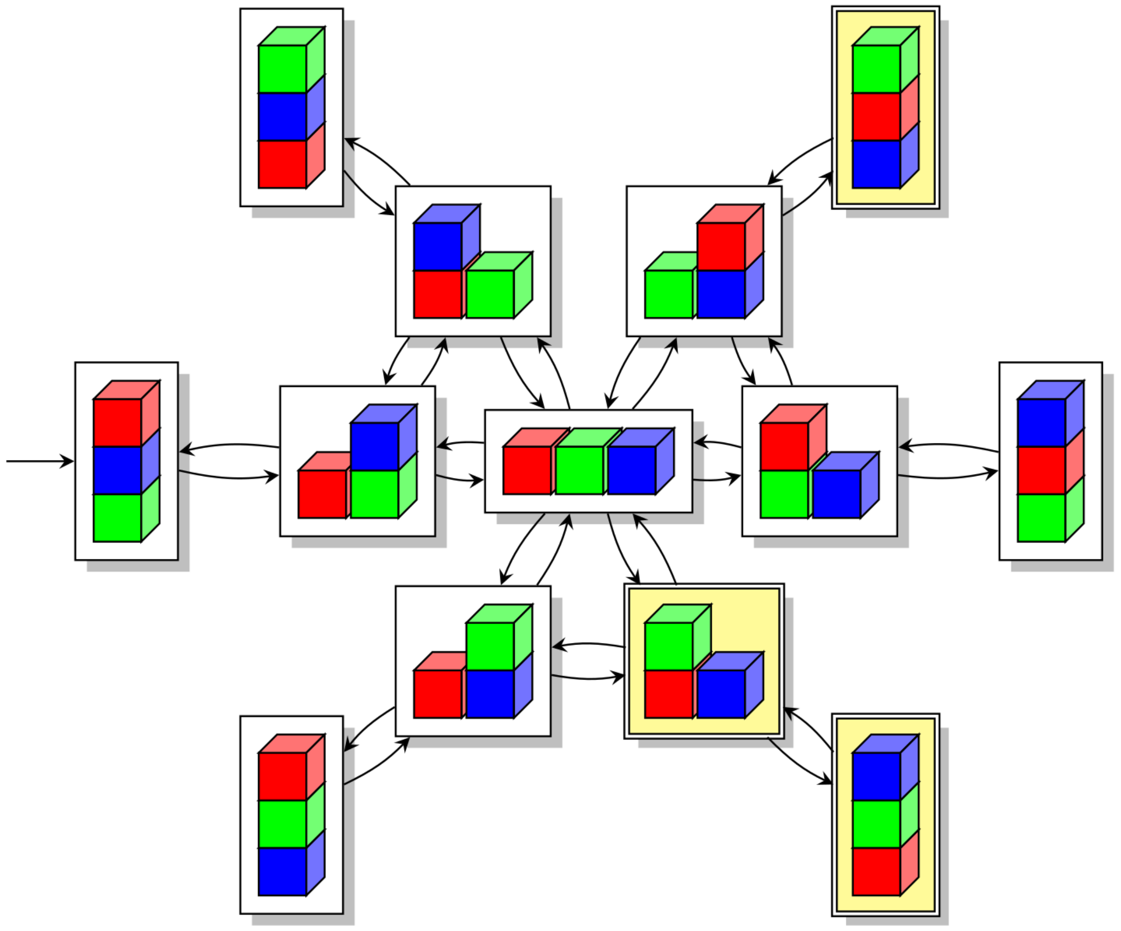
More complex plans may include branching actions: “if (condition) do action1 else do action2”

Definition (Plan)

A plan for a transition system is a sequence of actions occurring as labels on a path from the initial state to a goal state.

The cost of a plan $\langle a_1, \ldots, a_n \rangle$ is $\sum_{i=1}^{n} cost(a_i)$.

A plan is optimal if it has minimal cost.



Autonomous Behavior in AI

The key problem is to select **the action to do next**. This is the so-called **control problem**. Three approaches to this problem:

- **Programming-based:** Specify control by hand
- **Learning-based:** Learn control from experience
- **Model-based:** Specify problem by hand, derive control automatically

Planning is the **model-based approach to autonomous behavior** where agent controller derived from model of the actions, sensors, and goals.

Basic State Model: Classical Planning

- finite and discrete state space S
- a **known initial state** $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a **deterministic transition function** $s' = f(a, s)$ for $a \in A(s)$
- positive **action costs** $c(a, s)$

A **solution** is a sequence of applicable actions that maps s_0 into S_G, and it is **optimal** if it minimizes **sum of action costs** (e.g., $\#$ of steps)

A Basic Language for Classical Planning: Strips

- A **problem** in Strips is a tuple $P = \langle F, O, I, G \rangle$:
  - ▷ F stands for set of all **atoms** (boolean vars)
  - ▷ O stands for set of all **operators** (actions)
  - ▷ $I \subseteq F$ stands for **initial situation**
  - ▷ $G \subseteq F$ stands for **goal situation**

- Operators $o \in O$ **represented** by
  - ▷ the **Add** list $Add(o) \subseteq F$
  - ▷ the **Delete** list $Del(o) \subseteq F$
  - ▷ the **Precondition** list $Pre(o) \subseteq F$

From Language to Models

A Strips problem $P = \langle F, O, I, G \rangle$ determines **state model** $\mathcal{S}(P)$ where

- the states $s \in S$ are **collections of atoms** from F
- the initial state s_0 is I
- the goal states s are such that $G \subseteq s$
- the actions a in $A(s)$ are ops in O s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

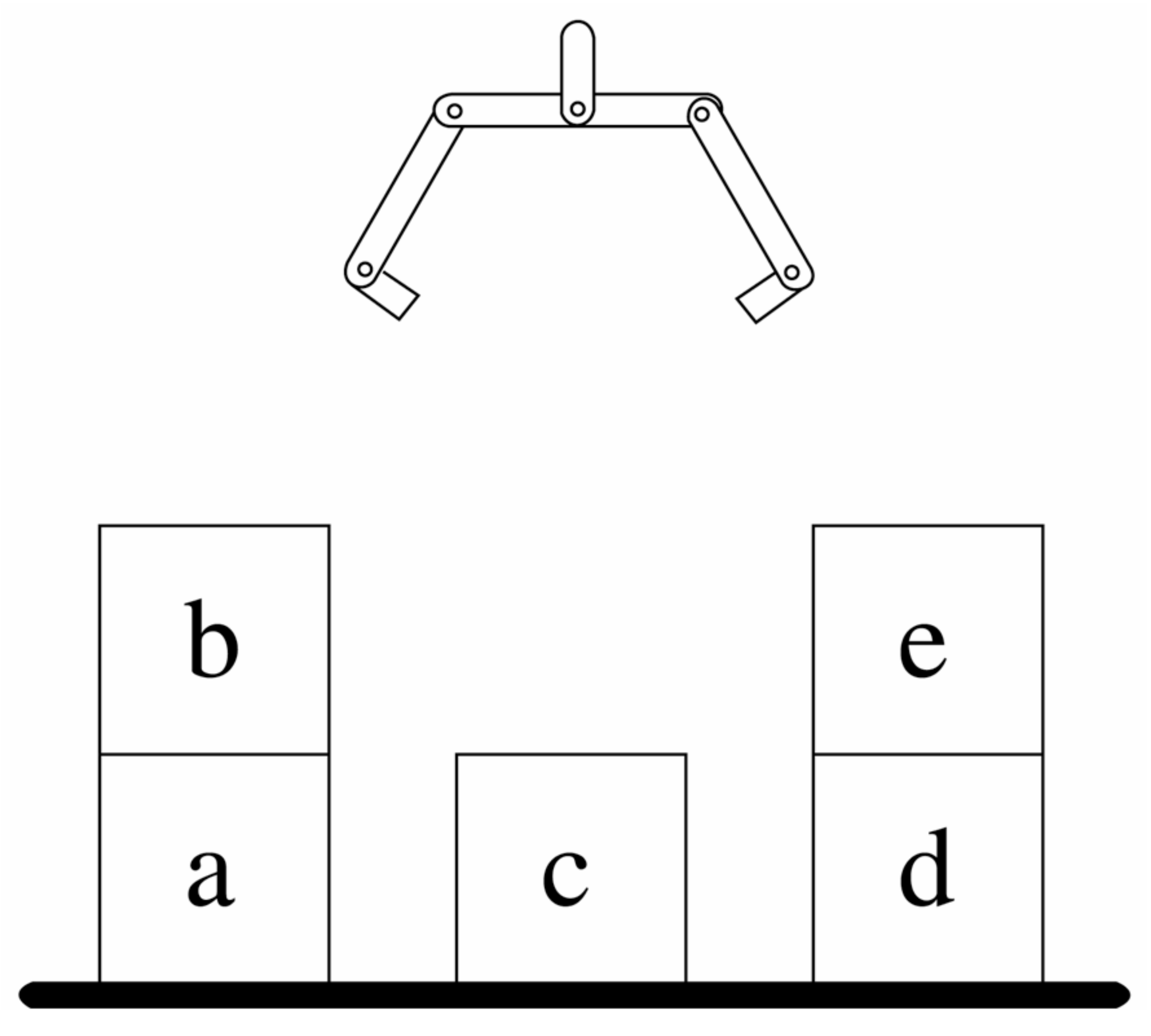
– (Optimal) **Solution** of P is (optimal) **solution** of $\mathcal{S}(P)$

– Slight language extensions often convenient: **negation**, **conditional effects**, **non-boolean variables**; some required for describing richer models (costs, probabilities, ...).

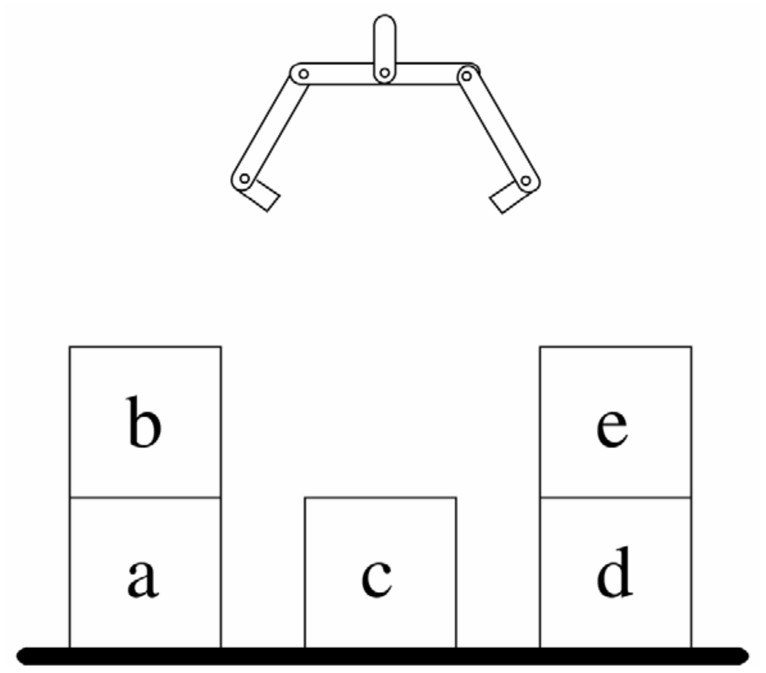
Characteristics of *classical planners*

- They operate on basic STRIPS actions.
- Important assumptions:
  - the agent is the only source of change in the world, otherwise the environment is static.
  - all the actions are deterministic.
  - the agent is omniscient: knows everything it needs to know about start state and effects of actions.
  - the goals are categorical, the plan is considered successful iff all the goals are achieved.

The blocks world



Represent this world using predicates



ontable(a)
ontable(c)
ontable(d)
on(b,a)
on(e,d)
clear(b)
clear(c)
clear(e)
gripping()

The robot arm can perform these tasks

- **pickup (W)**: pick up block W from its current location on the table and hold it
- **putdown (W)**: place block W on the table
- **stack (U, V)**: place block U on top of block V
- **unstack (U, V)**: remove block U from the top of block V and hold it

All assume that the robot arm can precisely reach the block.

How to define a planning problem

- **Create a *domain file:* contains the domain behavior, simply the operators**
- **Create a *problem file:* contains the initial state and the goal**

Example: Blocks in Strips (PDDL Syntax)

```
(define (domain BLOCKS)
  (:requirements :strips) ...
  (:action pick_up
          :parameters (?x)
          :precondition (and (clear ?x) (ontable ?x) (handempty))
          :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) ...)
  (:action put_down
           :parameters (?x)
           :precondition (holding ?x)
           :effect  (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action stack
          :parameters (?x ?y)
          :precondition (and (holding ?x) (clear ?y))
          :effect  (and (not (holding ?x)) (not (clear ?y)) (clear ?x)(handempty) .

(define (problem BLOCKS_6_1)
   (:domain BLOCKS)
   (:objects F D C E B A)
   (:init (CLEAR A) (CLEAR B) ...  (ONTABLE B) ... (HANDEMPTY))
   (:goal (AND (ON E F) (ON F C) (ON C B) (ON B A) (ON A D))))
```

A state space algorithm for STRIPS operators

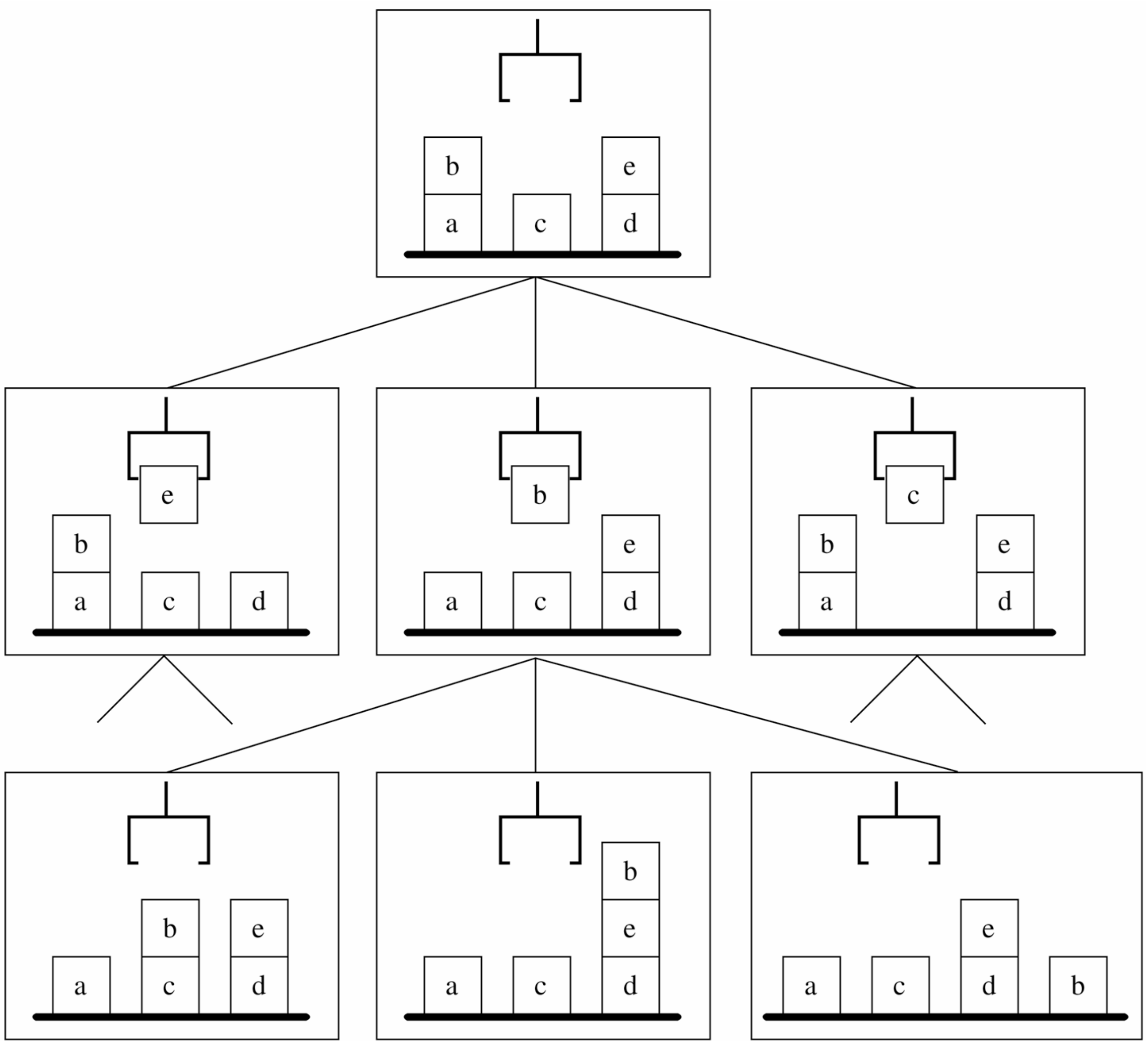
Search the space of situations (or states). This means each node in the search tree is a state.

The root of the tree is the start state.

Operators are the means of transition from each node to its children.

The goal test involves seeing if the set of goals is a subset of the current situation.

Now, the following graph makes much more sense



Problems in representation

***Frame problem*: List everything that does not change. It no more is a significant problem because what is not listed as changing (via the add and delete lists) is assumed to be not changing.**

***Qualification problem*: Can we list every precondition for an action? For instance, in order for PICKUP to work, the block should not be glued to the table, it should not be nailed to the table, …**

It still is a problem. A partial solution is to prioritize preconditions, i.e., separate out the preconditions that are worth achieving.

Problems in representation (cont’d)

Ramification problem**: Can we list every result of an action? For instance, if a block is picked up its shadow changes location, the weight on the table decreases, ...**

It still is a problem. A partial solution is to code rules so that inferences can be made. For instance, allow rules to calculate where the shadow would be, given the positions of the light source and the object. When the position of the object changes, its shadow changes too.

The gripper domain

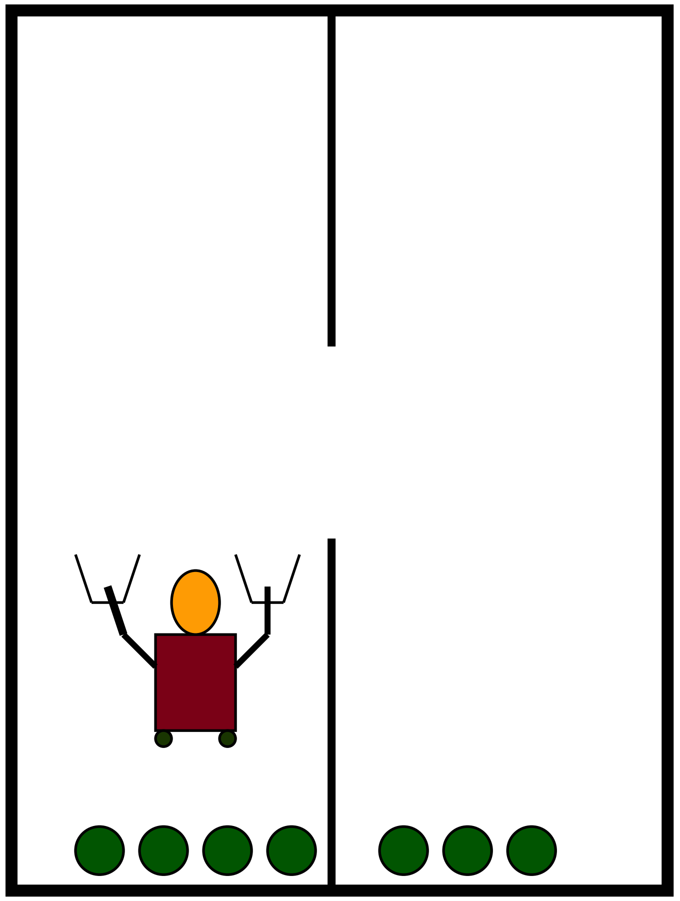
The agent is a robot with two grippers (left and right)

There are two rooms (rooma and roomb)

There are a number of balls in each room

Operators:

- PICK
- DROP
- MOVE



A “deterministic” plan

Pick ball1 rooma right

Move rooma roomb

Drop ball1 roomb right

Remember: the plans are generated “offline,” no observability, nothing can go wrong.

The gripper domain is interesting because parallelism is possible: can pick with both grippers at the same time.

The *domain definition* for the gripper domain

```
(define (domain gripper-strips)
  (:predicates    (room ?r)          (ball ?b)
                  (gripper ?g)       (at-robby ?r)
                  (at ?b ?r)         (free ?g)
                  (carry ?o ?g))
```

name of the domain

name of the action

"?" indicates a variable

combined add and delete lists

```
 (:action move
      :parameters (?from ?to)
      :precondition (and (room ?from) (room ?to)
                         (at-robby ?from))
      :effect (and (at-robby ?to)
                   (not (at-robby ?from))))
```

The *domain definition* for the gripper domain (cont'd)

```
(:action pick
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room)
          (gripper ?gripper) (at ?obj ?room)
          (at-robby ?room) (free ?gripper))
        :effect (and (carry ?obj ?gripper)
          (not (at ?obj ?room)) (not (free ?gripper))))
```

The *domain definition* for the gripper domain (cont'd)

```
(:action drop
          :parameters (?obj ?room ?gripper)
          :precondition (and (ball ?obj) (room ?room)
             (gripper ?gripper) (at-robby ?room)
             (carrying ?obj ?gripper))
          :effect (and (at ?obj ?room) (free ?gripper)
              (not (carry ?obj ?gripper))))))
```

An example *problem definition* for the gripper domain

```
(define (problem strips-gripper2)
  (:domain gripper-strips)

  (:objects rooma roomb ball1 ball2 left right)

  (:init (room rooma)          (room roomb)
         (ball ball1)          (ball ball2)
         (gripper left)        (gripper right)

         (at-robby rooma)
         (free left)           (free right)
         (at ball1 rooma)      (at ball2 rooma) )

  (:goal (at ball1 roomb)))
```

Why is planning a hard problem?

It is due to the large branching factor and the overwhelming number of possibilities.

There is usually no way to separate out the relevant operators. Take the previous example, and imagine that there are 100 balls, just two rooms, and two grippers. Again, the goal is to take 1 ball to the other room.

How many PICK operators are possible in the initial situation?

```
pick
        :parameters (?obj ?room ?gripper)
```

That is only one part of the branching factor, the robot could also move without picking up anything.

Why is planning a hard problem? (cont'd)

Also, goal interactions is a major problem. In planning, goal-directed search seems to make much more sense, but unfortunately cannot address the exponential explosion. This time, the branching factor increases due to the many ways of resolving the interactions.

When subgoals are compatible, i.e., they do not interact, they are said to be *linear* (or *independent,* or *serializable*).

Life is easier for a planner when the subgoals are independent because then divide-and-conquer works.

How to deal with the exponential explosion?

Use goal-directed algorithms

Use domain-independent heuristics

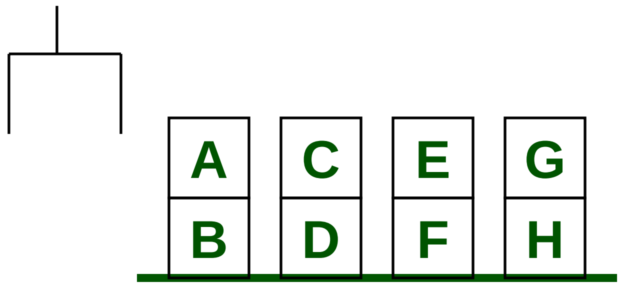
Use domain-dependent heuristics
 (we need a language to specify them)

A sampler of planning algorithms

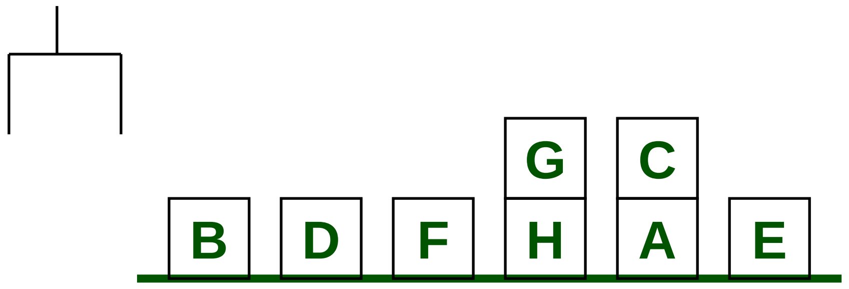
- **Forward chaining**
  - **Work in a state space**
  - **Start with the initial state, try to reach the goal state using forward progression**
- **Backward chaining**
  - **Work in a state space**
  - **Start with the goal state, try to reach the initial state using backward regression**
- **Partial order planning**
  - **Work in a plan space**
  - **Start with an empty plan, work from the goal to reach a complete plan**

Forward chaining

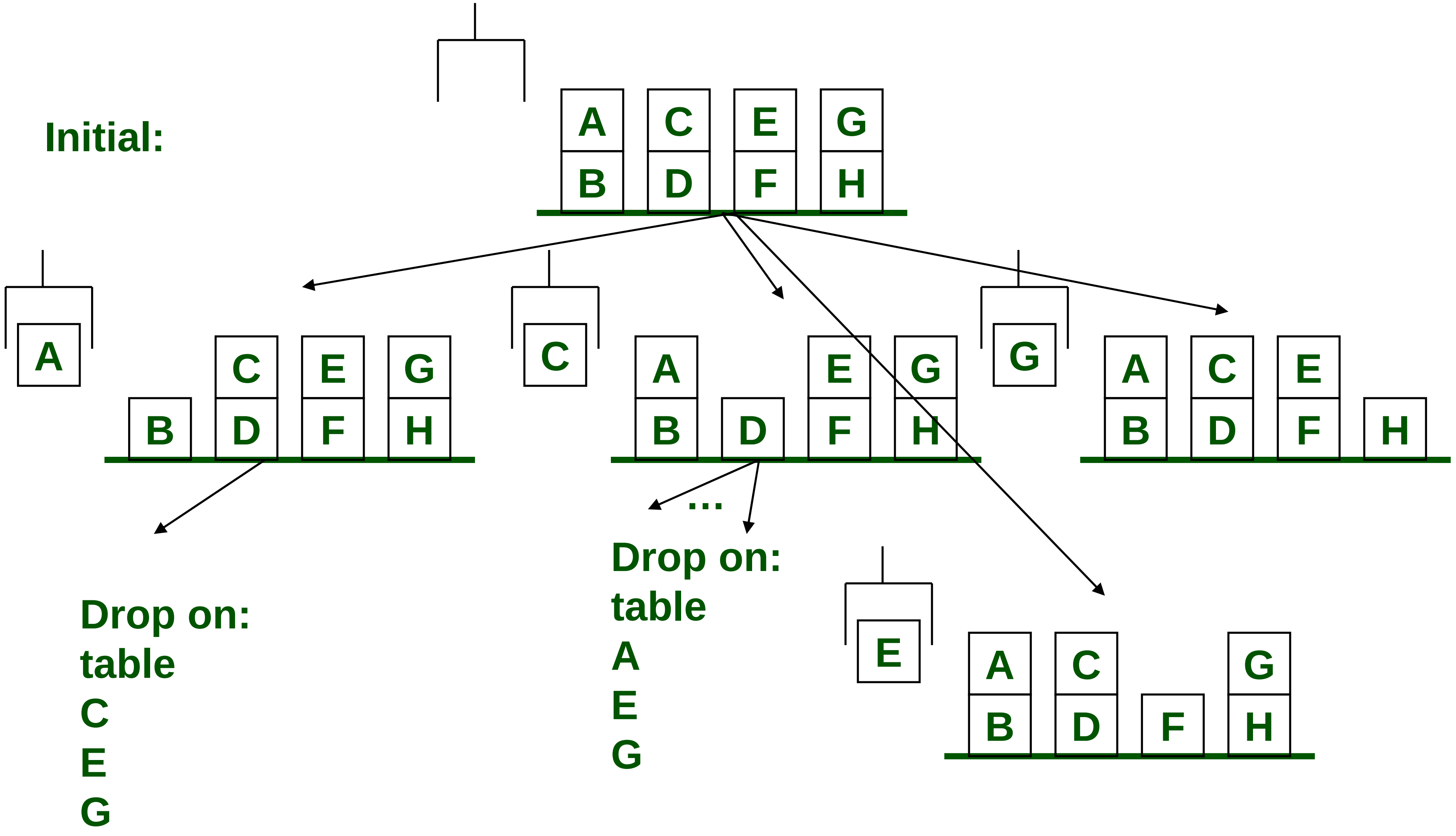
Initial:



Goal :



1st and 2nd levels of search

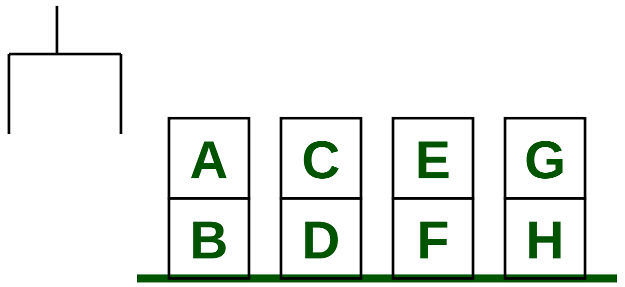


Results

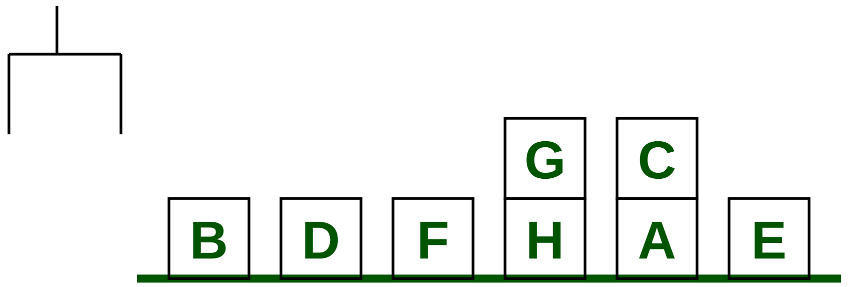
- A plan is:
  - unstack (A, B)
  - putdown (A)
  - unstack (C, D)
  - stack (C, A)
  - unstack (E, F)
  - putdown (F)
- Notice that the final locations of D, F, G, and H need not be specified
- Also notice that D, F, G, and H will never need to be moved. But there are states in the search space which are a result of moving these. Working backwards from the goal might help.

Backward chaining

Initial:



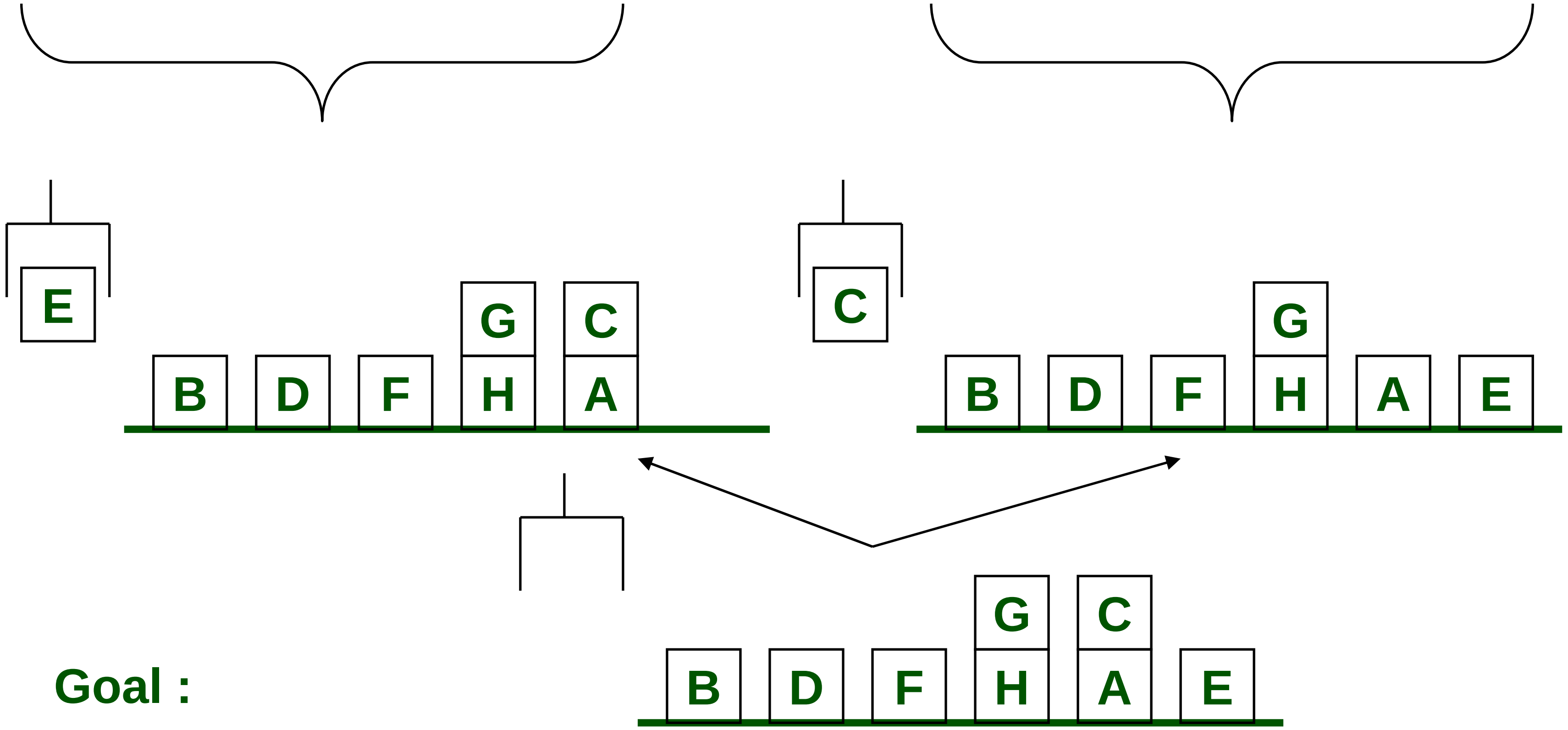
Goal :



1st level of search

For E to be on the table,
the last action must be
putdown(E)

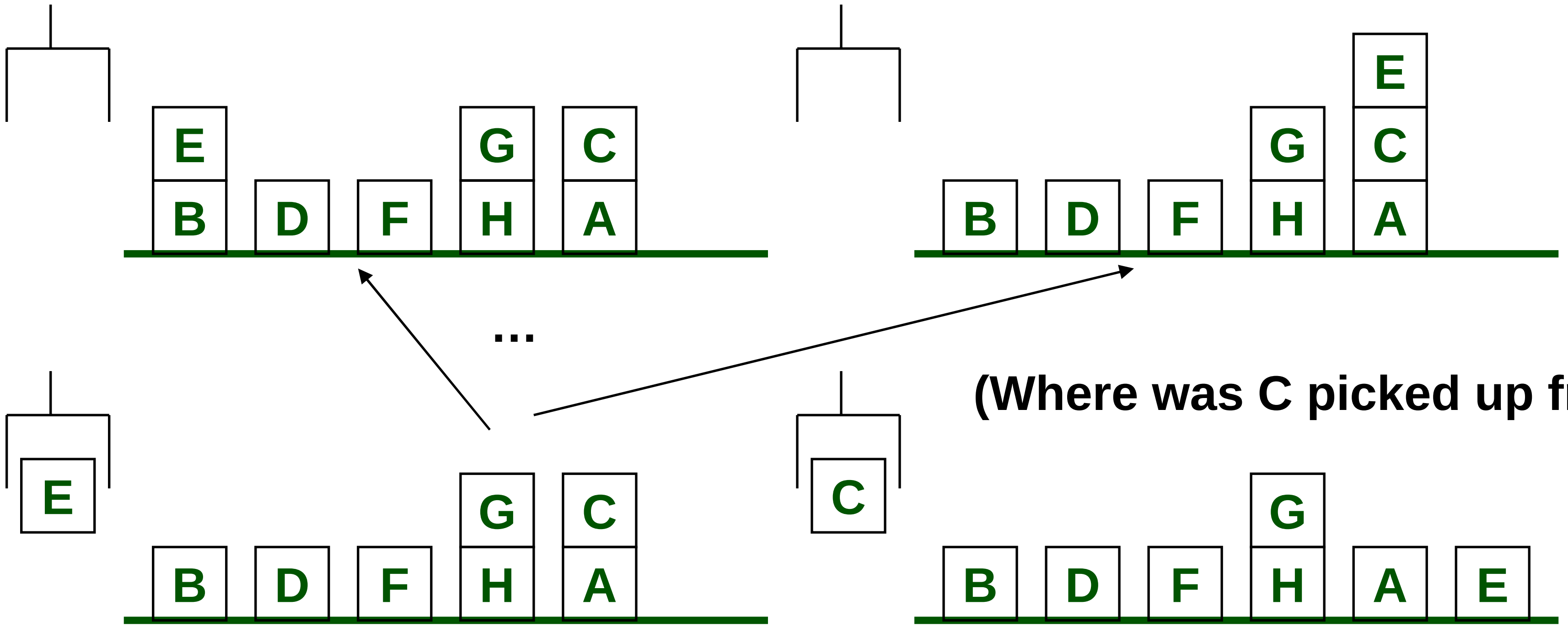
For C to be on A,
the last action must be
stack(C,A)



Goal :

2nd level of search

Where was E picked up from?



Results

- The same plan can be found
  - unstack (A, B)
  - putdown (A)
  - unstack (C, D)
  - stack (C, A)
  - unstack (E, F)
  - putdown (F)
- Now, the final locations of D, F, G, and H need to be specified
- Notice that D, F, G, and H will never need to be moved. But observe that from the second level on the branching factor is still high

Partial-order planning (POP)

- Notice that the resulting plan has two parallelizable threads:

unstack (A,B)		unstack (E, F)
putdown (A)		putdown (F)
unstack (C,D)	&	
stack (C,A)		

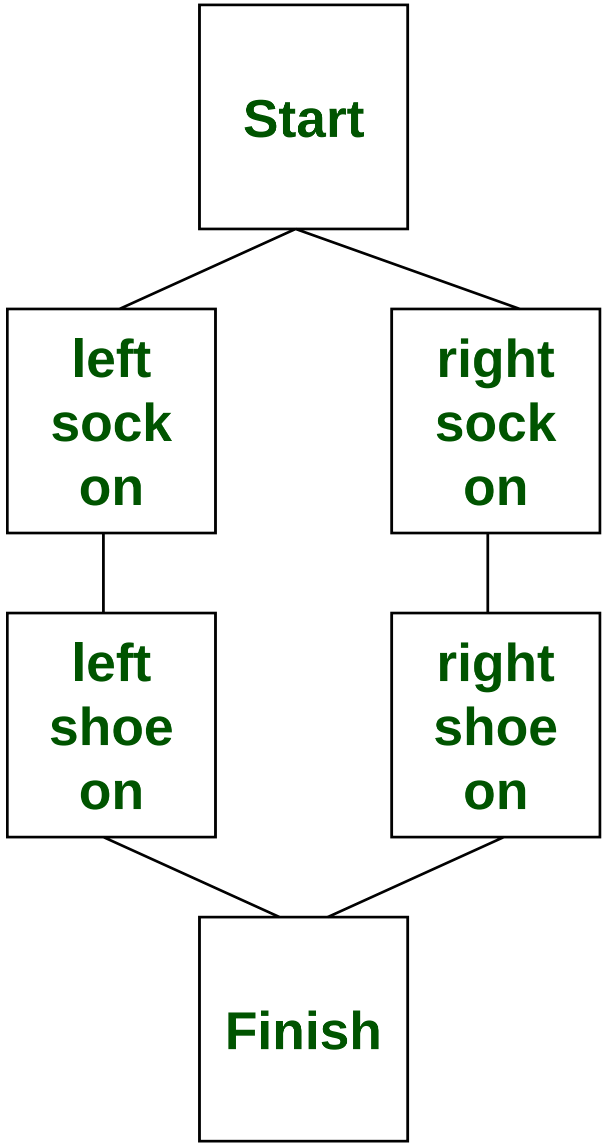
- These steps can be interleaved in 3 different ways:

unstack (E, F)	unstack (A,B)	unstack (A,B)
putdown (F)	putdown (A)	putdown (A)
unstack (A,B)	unstack (E, F)	unstack (C,D)
putdown (A)	putdown (F)	stack (C,A)
unstack (C,D)	unstack (C,D)	unstack (E, F)
stack (C,A)	stack (C,A)	putdown (F)

Partial-order planning (cont'd)

- **Idea: Do not order steps unless it is necessary**
- **Then a partially ordered plan represents several totally ordered plans**
- **That decreases the search space**
- **But still the planning problem is not solved, good heuristics are crucial**

Partial-order planning (cont'd)



Start	Start	Start	Start	Start	Start
Left sock on	Right sock on	Left sock on	Right sock on	Left sock on	Right sock on
Left shoe on	Right shoe on	Right sock on	Left sock on	Right sock on	Left sock on
Right sock on	Left sock on	Left shoe on	Right shoe on	Right shoe on	Left shoe on
Right shoe on	Left shoe on	Right shoe on	Left shoe on	Left shoe on	Right shoe on
Finish	Finish	Finish	Finish	Finish	Finish

Partially ordered plans

Partially ordered collection of steps with

- START step has the initial state description as its effect
- FINISH step has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

Partially ordered plans (cont'd)

A partially ordered plan is a 5-tuple (A, O, C, OC, UL)

- A is the set of actions that make up the plan. They are partially ordered.
- O is a set of ordering constraints of the form $A \prec B$. It means A comes before B.
- C is the set of causal links in the form (A, p, B) where A is the *supplier action*, where B is the *consumer action*, and p is the condition supplied. It is read as "*A achieves p for B*."

Partially ordered plans (cont'd)

A partially ordered plan is a 5-tuple (A, O, C, OC, UL)

- OC is a set of open conditions, i.e., conditions that are not yet supported by causal links. It is of the form p for A where p is a condition and A is an action.
- UL is a set of unsafe links, i.e., causal links whose conditions might be undone by other actions.

Partially ordered plans (cont'd)

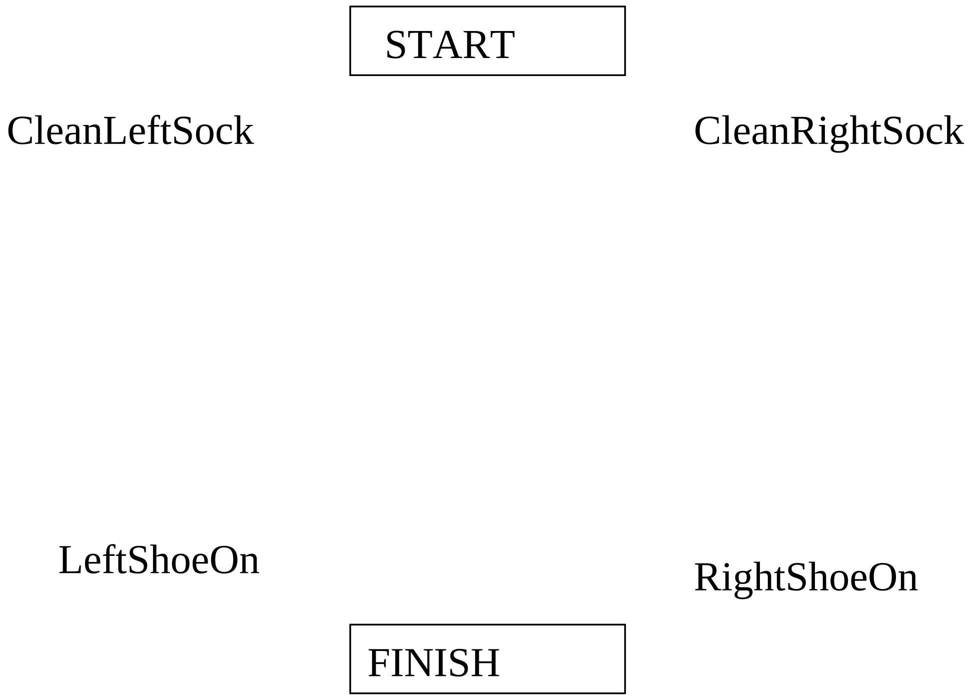
A plan is *complete* iff every precondition is achieved, and there are no unsafe links. A precondition is *achieved* iff it is the effect of an earlier step and no *possibly intervening* step undoes it

In other words, a plan is complete when $OC \cup UL = \emptyset$.

$OC \cup UL$ is referred to as the *flaws* in a plan.

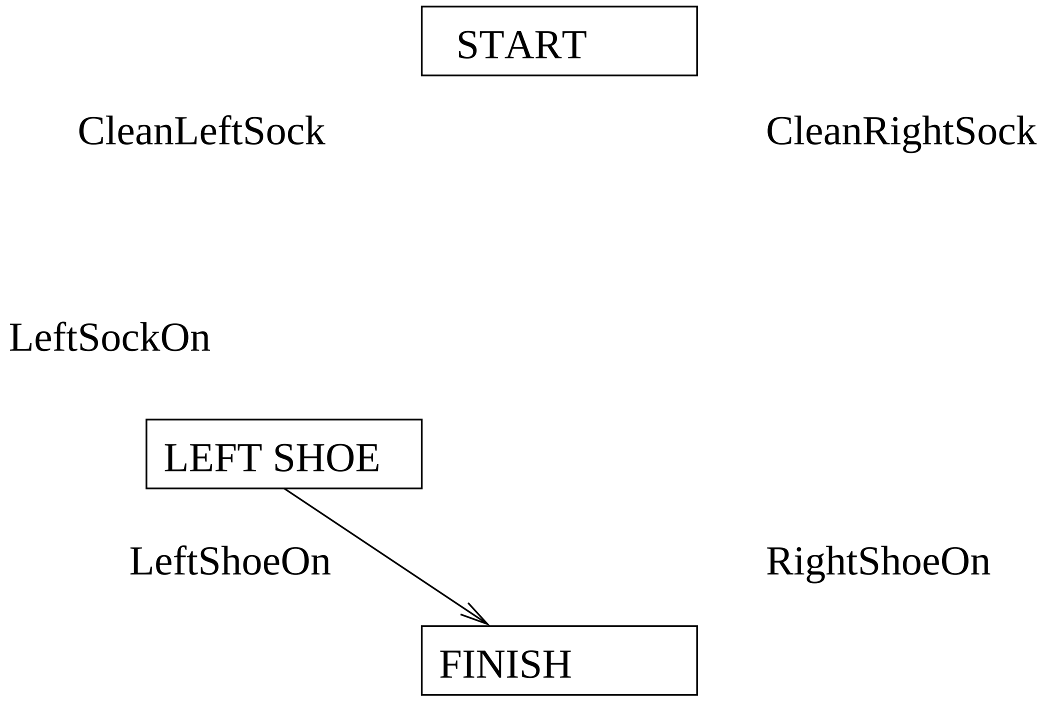
When a causal link is established, the corresponding condition is said to be *closed*.

Example



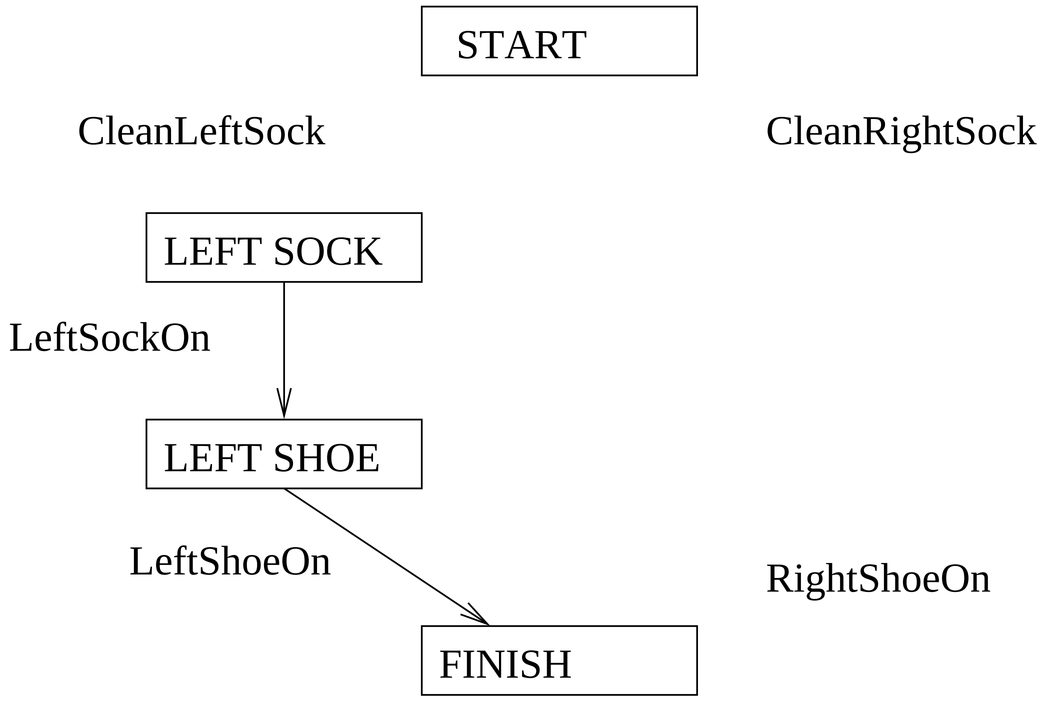
OC=
LeftShoeOn for FINISH
RightShoeOn for FINISH

Example (cont’d)



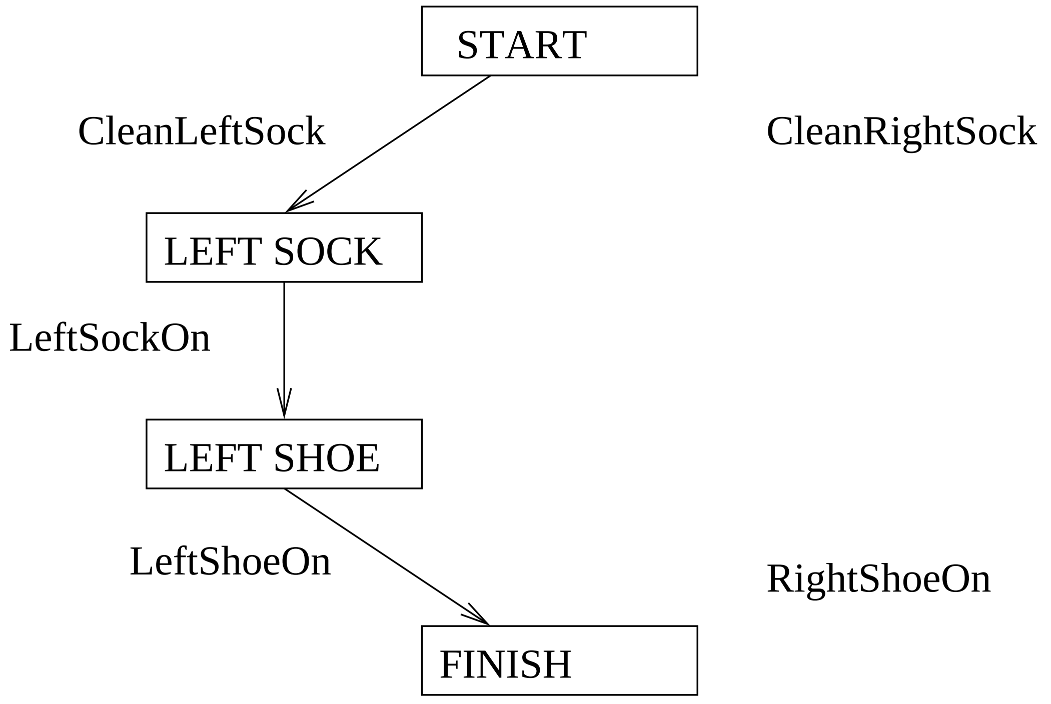
OC=
RightShoeOn for FINISH
LeftSockOn for LEFTSHOE

Example (cont'd)



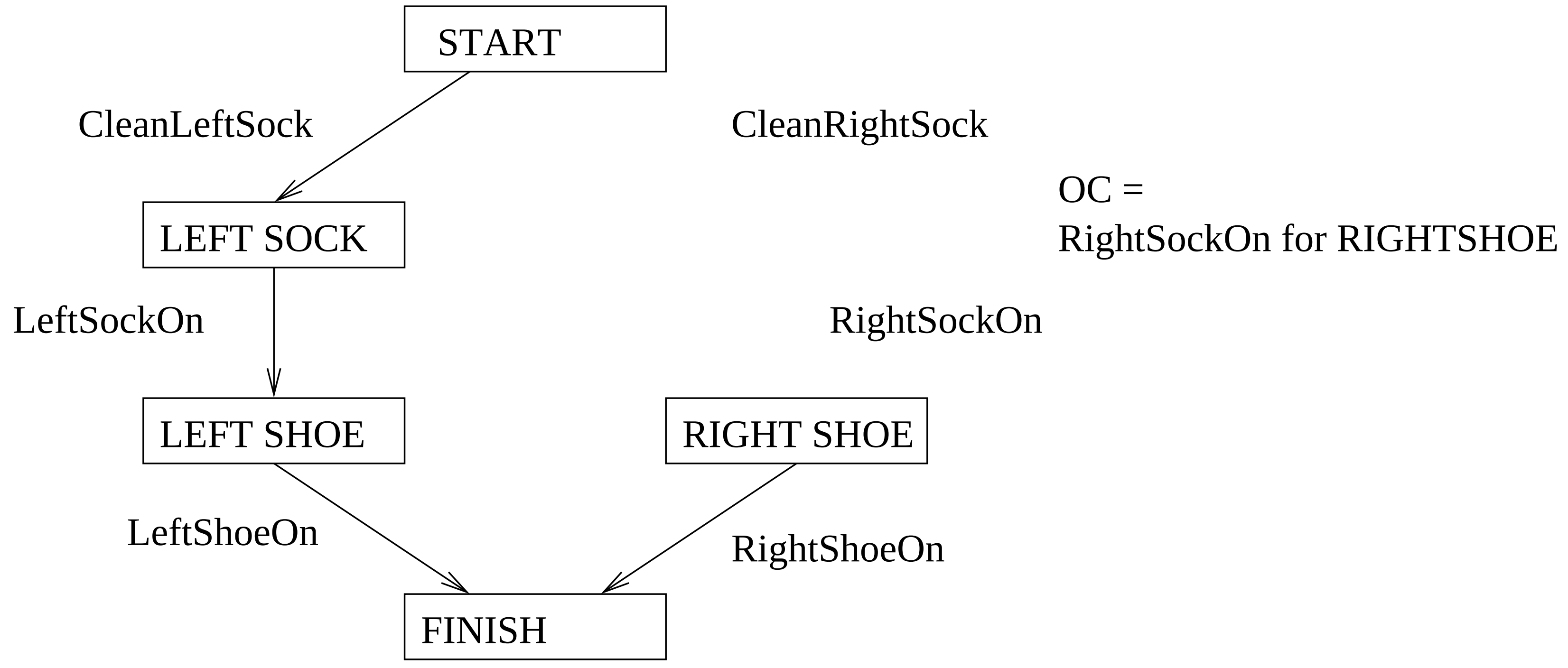
OC =
CleanLeftSock for LEFTSOCK
RightShoeOn for FINISH

Example (cont’d)

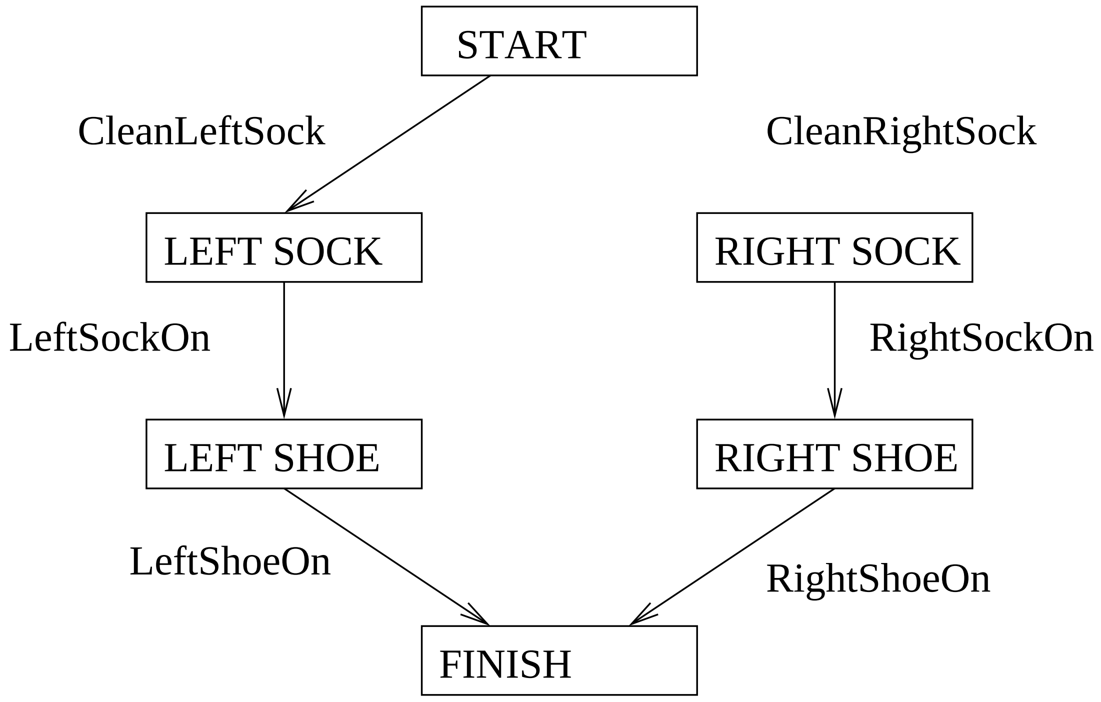


OC =
RightShoeOn for FINISH

Example (cont’d)

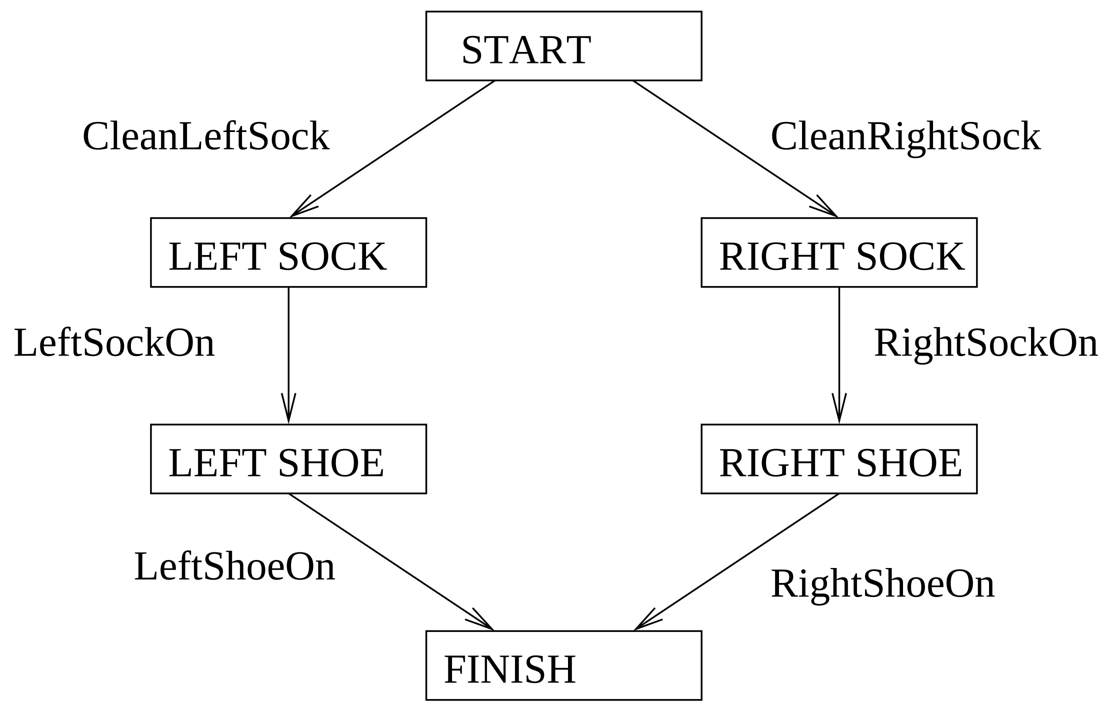


Example (cont'd)



OC =
CleanRightSock for RIGHTSOCK

Example (cont’d)



OC=
{ }

Planning process

Operators on partial plans:

- close open conditions:
  - add a link from an existing action to an open condition
  - add a step to fulfill an open condition
- resolve threats:
  - order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans

Backtrack if an open condition is unachievable or if a conflict is unresolvable

POP is a search in the plan space

```
function TREE-SEARCH (problem)
returns a solution, or failure

    initialize the frontier using the initial state of problem
    loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state
        then return the corresponding solution
      else expand the chosen node and add the resulting
        nodes to the frontier
    end
```

POP algorithm specifics

The initial state, goal state and the operators are given. The planner converts them to required structures.

Initial state:
MAKE-MINIMAL-PLAN (*initial*,*goal*)

Goal-Test:
SOLUTION?(*plan*)

SOLUTION? returns true iff OC and UL are both empty.

Successor function:
The successors function could either close an open condition or resolve a threat.

POP algorithm specifics (cont'd)

```
function SUCCESSORS (plan)
returns a set of partially ordered plans

  flaw-type ← SELECT-FLAW-TYPE (plan)
  if flaw-type is an open condition then
    $S_{need}, c$ ← SELECT-SUBGOAL (plan)
    return CLOSE-CONDITION (plan, operators, $S_{need}$,c)
  if flaw-type is a threat then
    $S_{threat}, S_i, c, S_j$ ← SELECT-THREAT(plan)
    return RESOLVE-THREAT (plan, $S_{threat}, S_i, c, S_j$)
```

POP algorithm specifics (cont'd)

```
function CLOSE-CONDITION (plan, operators, $S_{need}$,c)
returns a set of partially ordered plans

   plans $\leftarrow \emptyset$
  for each $S_{add}$ from operators or STEPS(plan)
   that has $c$ has an effect do
    new-plan $\leftarrow$ plan
    if $S_{add}$ is a newly added step from operators then
      add $S_{add}$ to STEPS (new-plan)
      add START $\prec S_{add} \prec$ FINISH to ORDERINGS (new-plan)
    add the causal link $(S_{add}, c, S_{need})$ to LINKS (new-plan)
    add the ordering constraint $(S_{add} \prec S_{need})$ to
       ORDERINGS (new-plan)
    add new-plan to plans
  end

   return new-plans
```

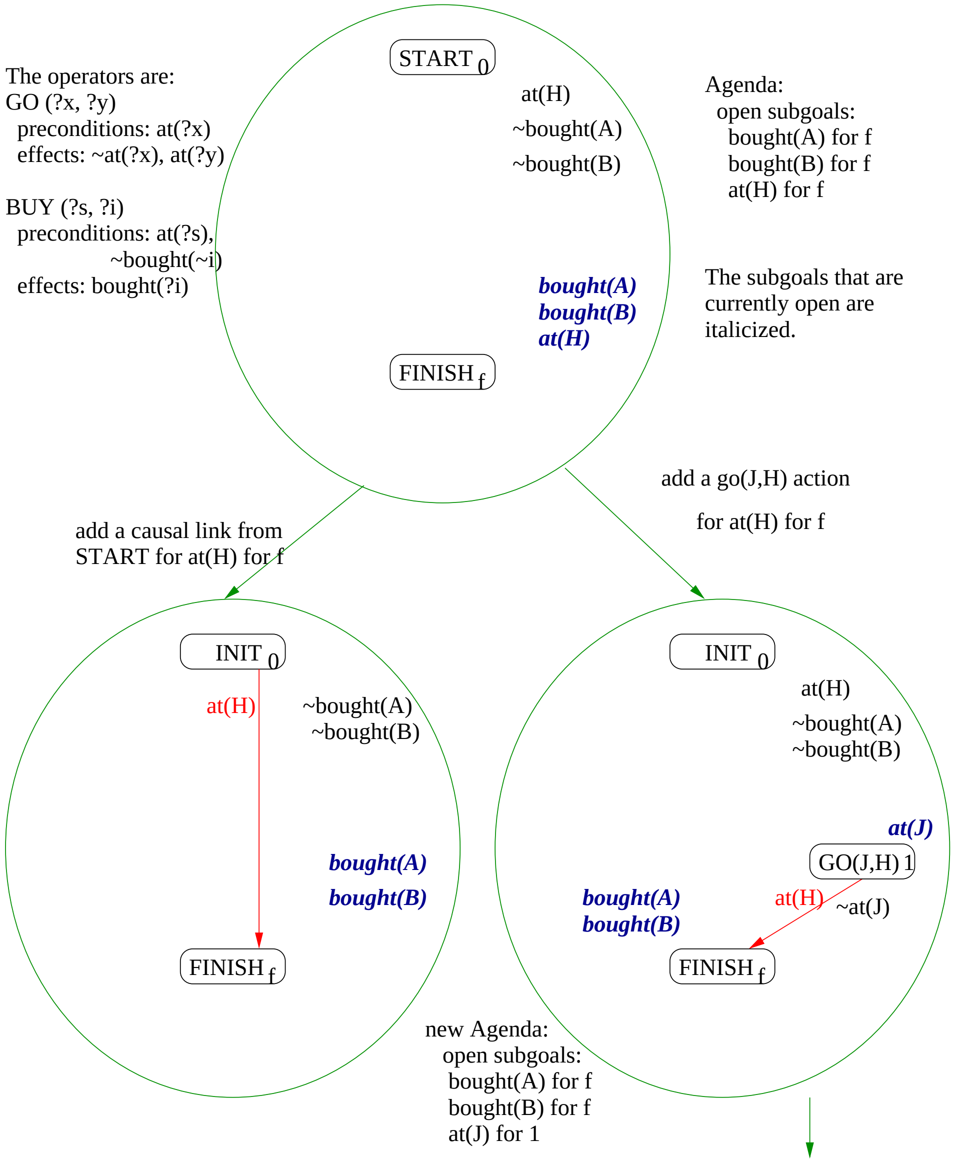
POP algorithm specifics (cont'd)

```
**function** RESOLVE-THREAT (*plan*, $S_{threat}, S_i, c, S_j$)
**returns** *a set of partially ordered plans*

    plans ← ∅
  //Demotion:
  new-plan ← plan
  add the ordering constraint $(S_{threat} \prec S_i)$ to ORDERINGS (new-plan)
  if new-plan is consistent then
    add new-plan to plans
  //Promotion:
  new-plan ← plan
  add the ordering constraint $(S_j \prec S_{threat})$ to ORDERINGS (new-plan)
  if new-plan is consistent then
    add new-plan to plans

  return new-plans
```

Shopping example

The operators are:
GO (?x, ?y)
  preconditions: at(?x)
  effects: ~at(?x), at(?y)

BUY (?s, ?i)
  preconditions: at(?s),
    ~bought(~i)
  effects: bought(?i)

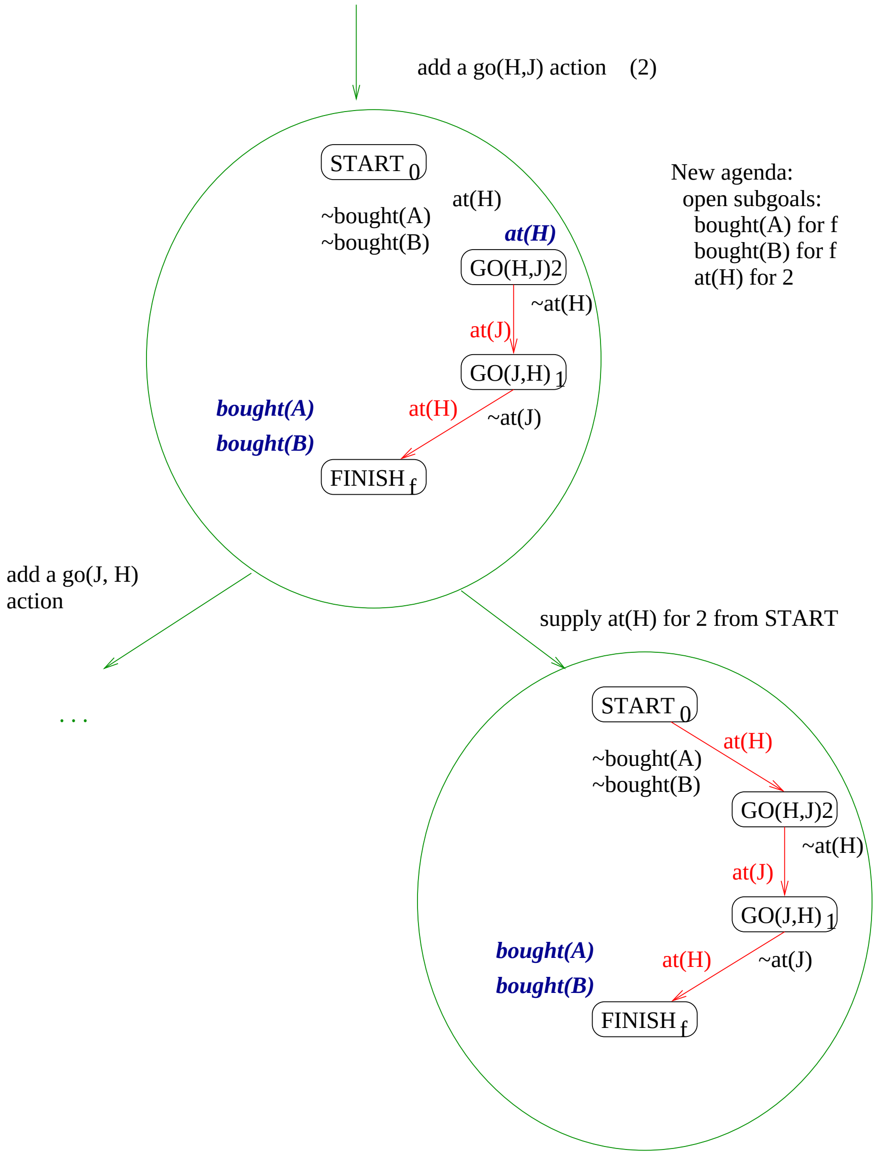


Agenda:
  open subgoals:
    bought(A) for f
    bought(B) for f
    at(H) for f

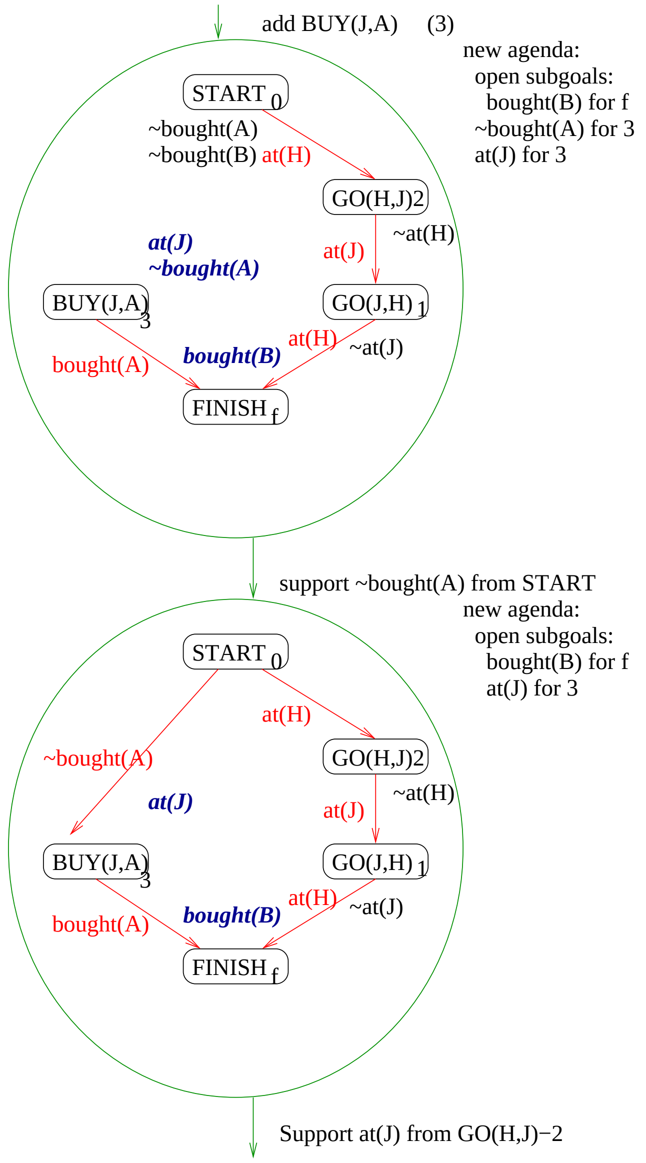
The subgoals that are currently open are italicized.

new Agenda:
  open subgoals:
    bought(A) for f
    bought(B) for f
    at(J) for 1

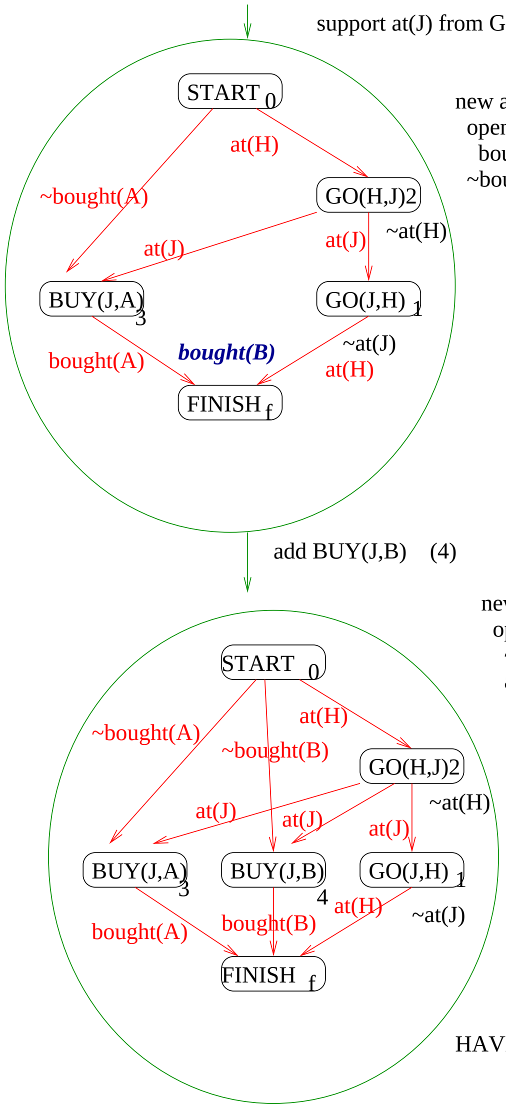
Shopping example (cont'd)



Shopping example (cont’d)



Shopping example (cont'd)



support at(J) from GO(H,J)−2

new agenda:
open subgoals:
bought(B) for f
~bought(A) for 3

add BUY(J,B) (4)

new agenda:
open subgoals:
~bought(B) for 4
at(J) for 4

support ~bought(B) for 4 from START

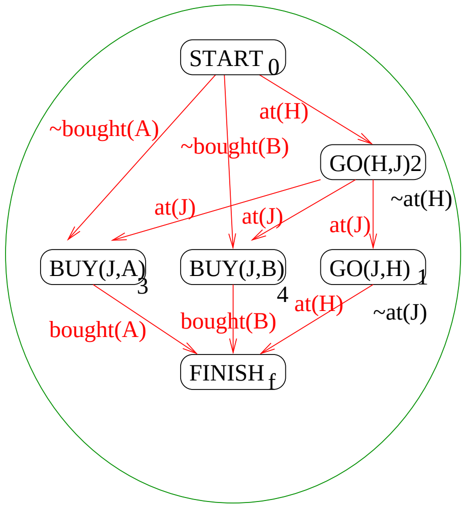
new agenda:
open subgoals:
at(J) for 4

support ~at(J) for 4 from GO(H,J)−2

new agenda:
open subgoals:
none

HAVEN'T CONSIDERED THE THREATS YET

Shopping example (cont’d)

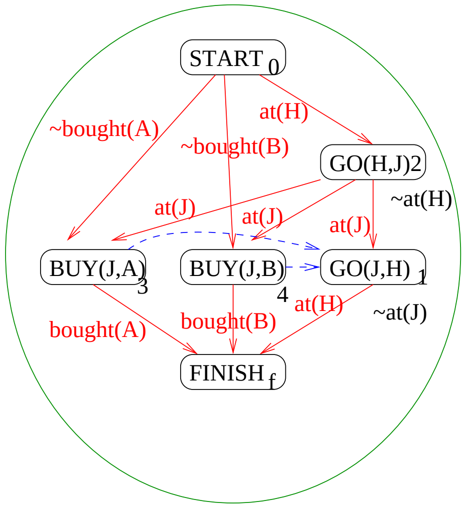


Now, the solution is a possible ordering of this plan. Those are:

```
2 3 4 1
2 3  1 4
2 4 3 1
2 4 1 3
2 1 3 4
2 1 4 3
```

It should not be possible to order GO(J,H) before any of the BUY actions.

Shopping example (cont'd)

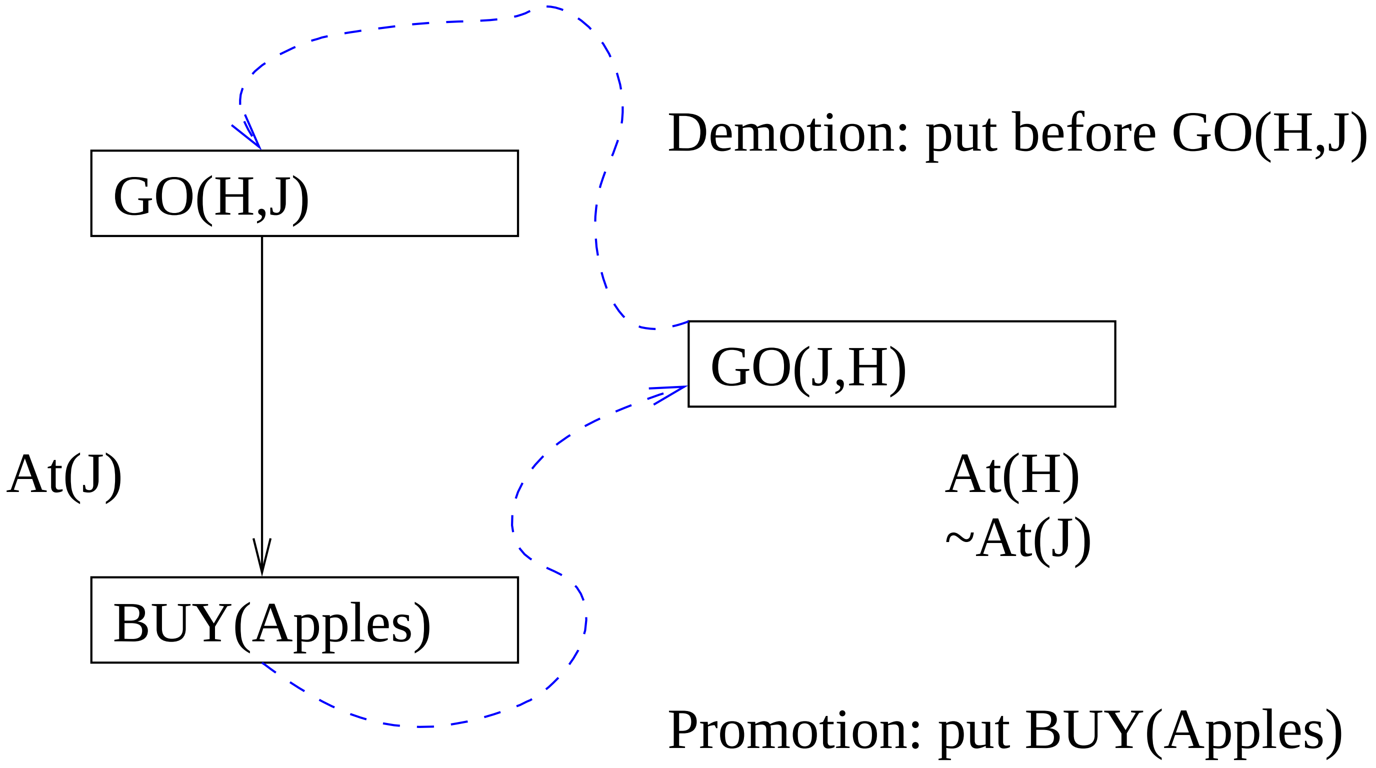


This is a correct partially ordered plan.
It is complete.
The possible total orders are:
2 3 4 1
2 4 3 1

The agent has to go to Jim's first.
It order of getting the items does not matter.
Then it has to go back home.

Threats and promotion/demotion

A threatening step is a potentially intervening step that destroys the condition achieved by a causal link. E.g., GO(J,H) threatens At(J)



Properties of POP

- Nondeterministic algorithm: backtracks at choice points on failure:
  - choice of S_{add} to achieve S_{need}
  - choice of demotion or promotion for threat resolution
  - selection of S_{need} is irrevocable
- POP is sound, complete, and systematic (no repetition)
- Extensions for disjunction, universals, negation, conditionals
- Particularly good for problems with many loosely related subgoals

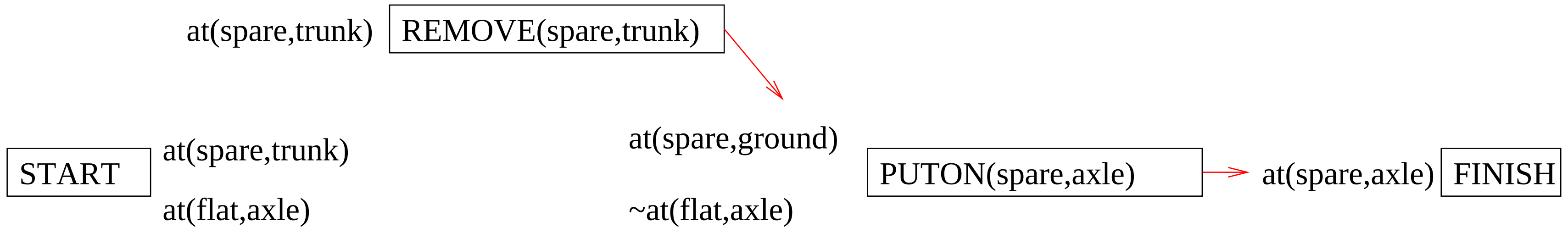
Additional POP examples

- The flat tire example shows the effect of inserting an “impossible” action.
- The Sussman anomaly shows that “divide-and-conquer” is not always optimal. POP can find the optimal plan.

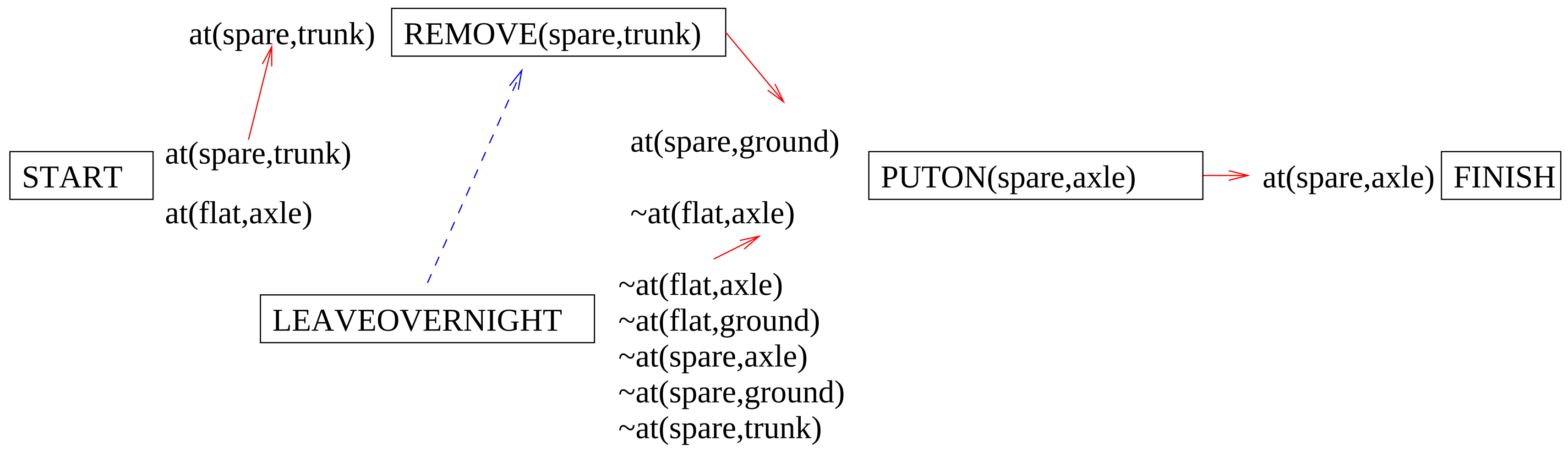
The flat tire domain

Init(At(Flat,Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(REMOVE(spare,trunk),
    Precond: At(spare,trunk)
    Effect: ¬At(spare,trunk) ∧ At(spare,ground)
Action(REMOVE(flat,axle),
    Precond: At(flat,axle)
    Effect: ¬At(flat,axle) ∧ At(flat,ground)
Action(PUTON(spare,axle),
    Precond: At(spare,ground) ∧ ¬ at(flat,axle)
    Effect: ¬At(spare,ground) ∧ At(spare,axle)
Action(LEAVEOVERNIGHT
    Precond:
    Effect: ¬At(spare,ground) ∧ ¬ At(spare,axle)
            ¬At(spare,trunk) ∧ ¬ At(flat,ground)
            ¬At(flat,axle)

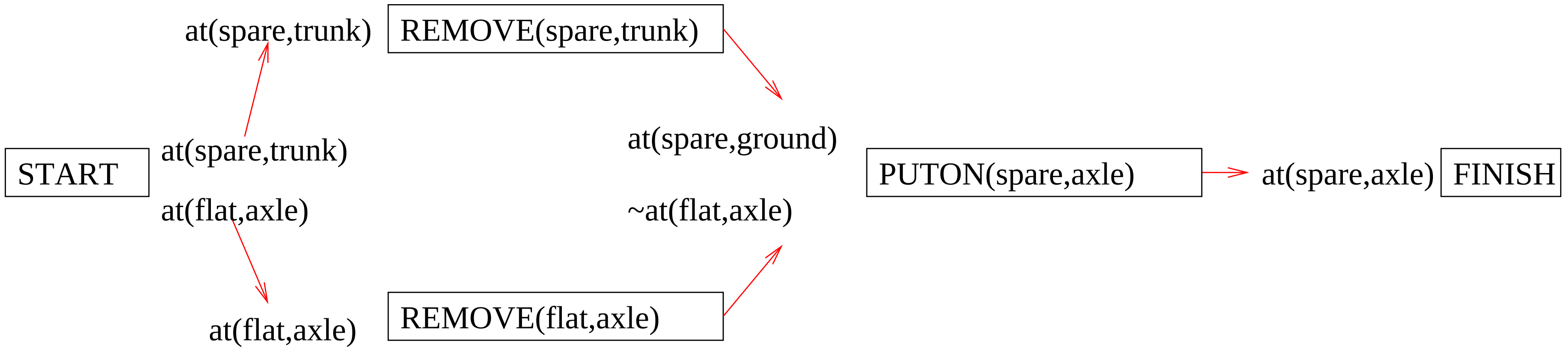
The flat tire plan



The flat tire plan (cont’d)



The flat tire plan (cont'd)



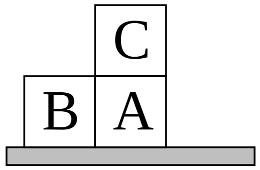
Sussman anomaly

Clear(x)   On(x,z)   Clear(y)

$\boxed{\text{PUTON(x,y)}}$

~On(x,z)   ~Clear(y)

Clear(z)   On(x,y)



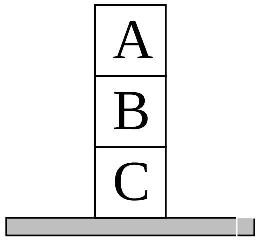
Clear(x)   On(x,z)

$\boxed{\text{PUTONTABLE(x)}}$

~On(x,z)

Clear(z)   On(x,Table)

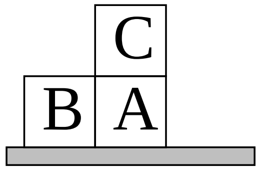
+ several inequality constraints



Sussman anomaly (cont'd)

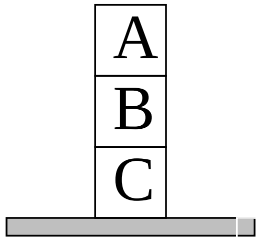
START

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



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

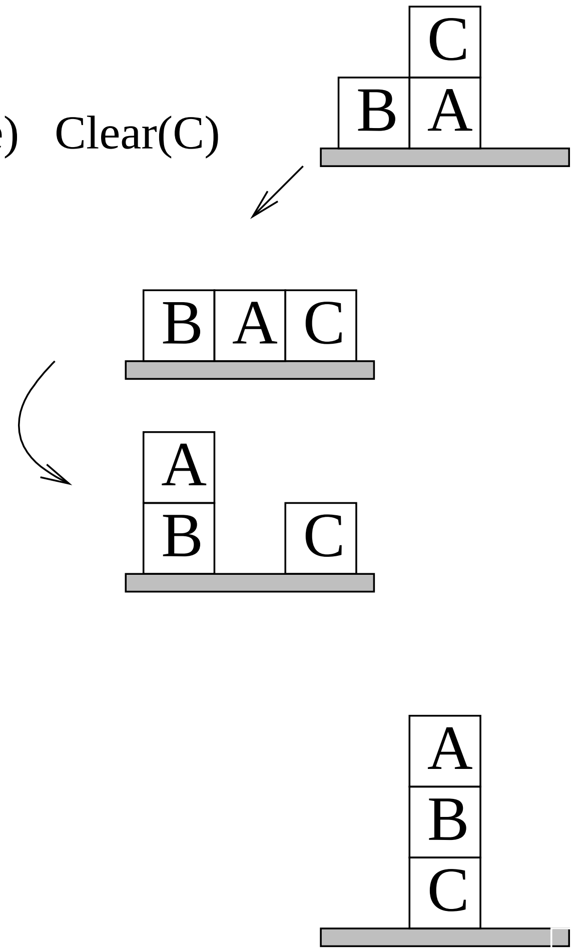
FINISH



Sussman anomaly (cont'd)

START

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

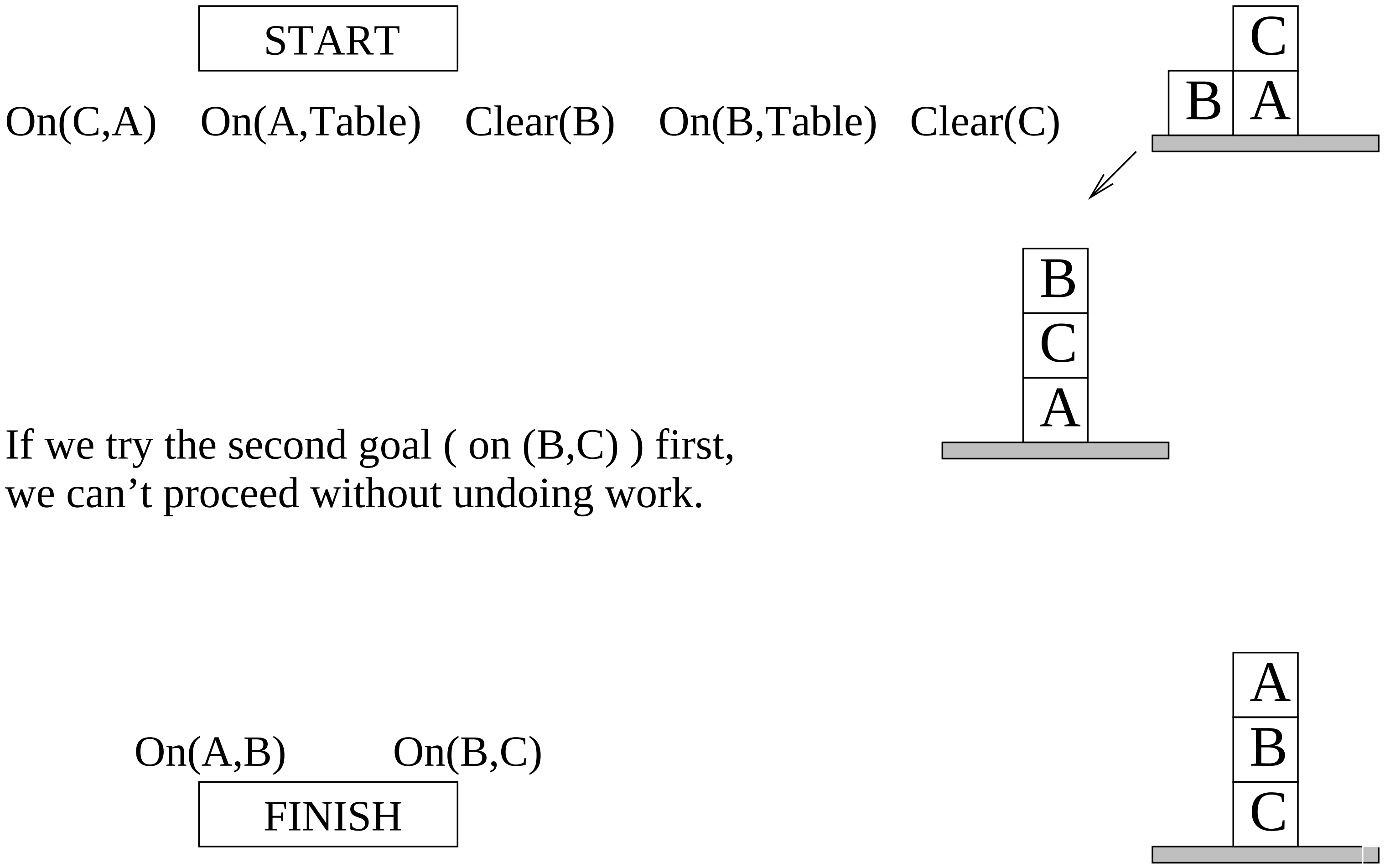


If we try the first goal (on(A,B)) first,
we can't proceed without undoing work

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

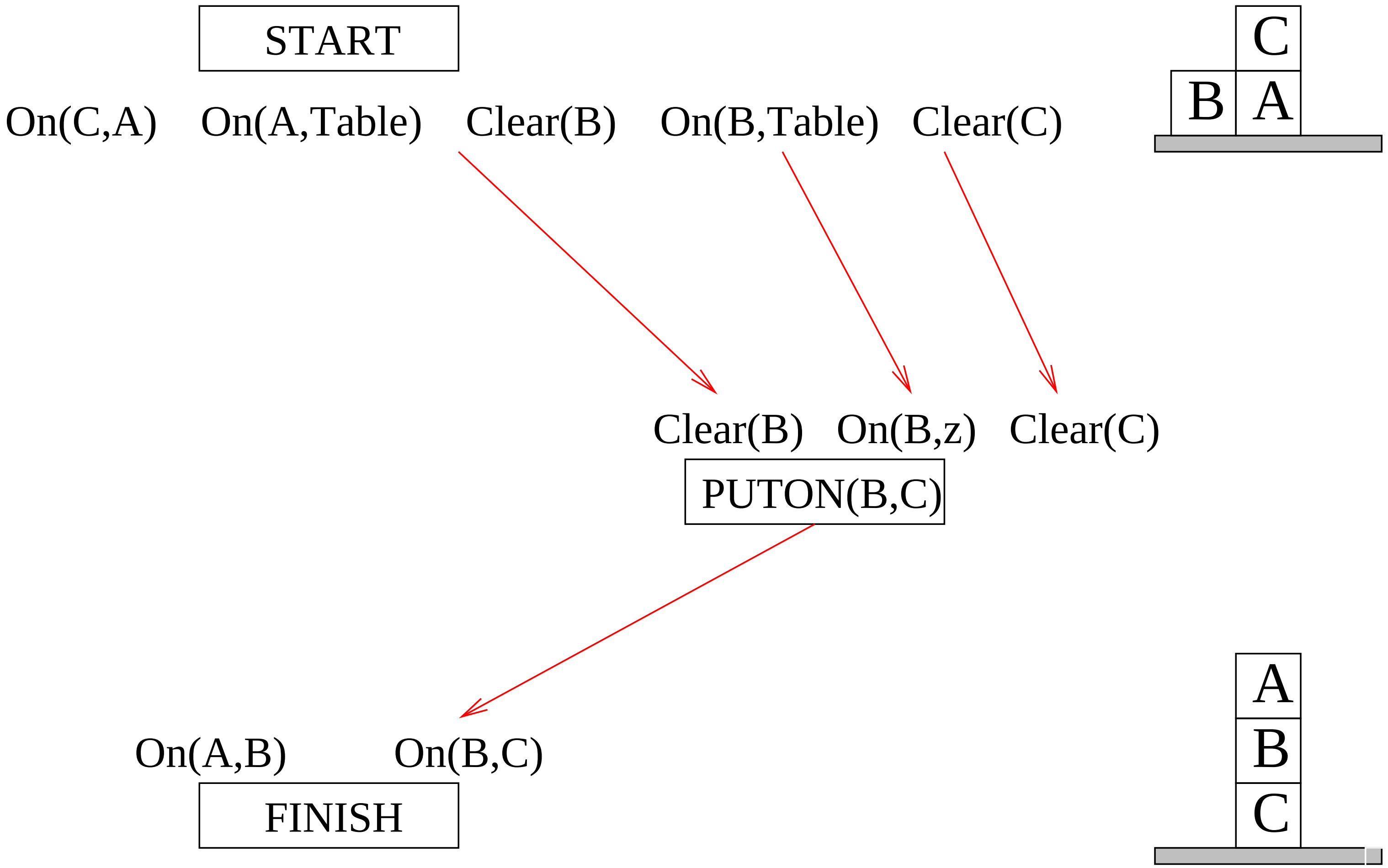
FINISH

Sussman anomaly (cont'd)

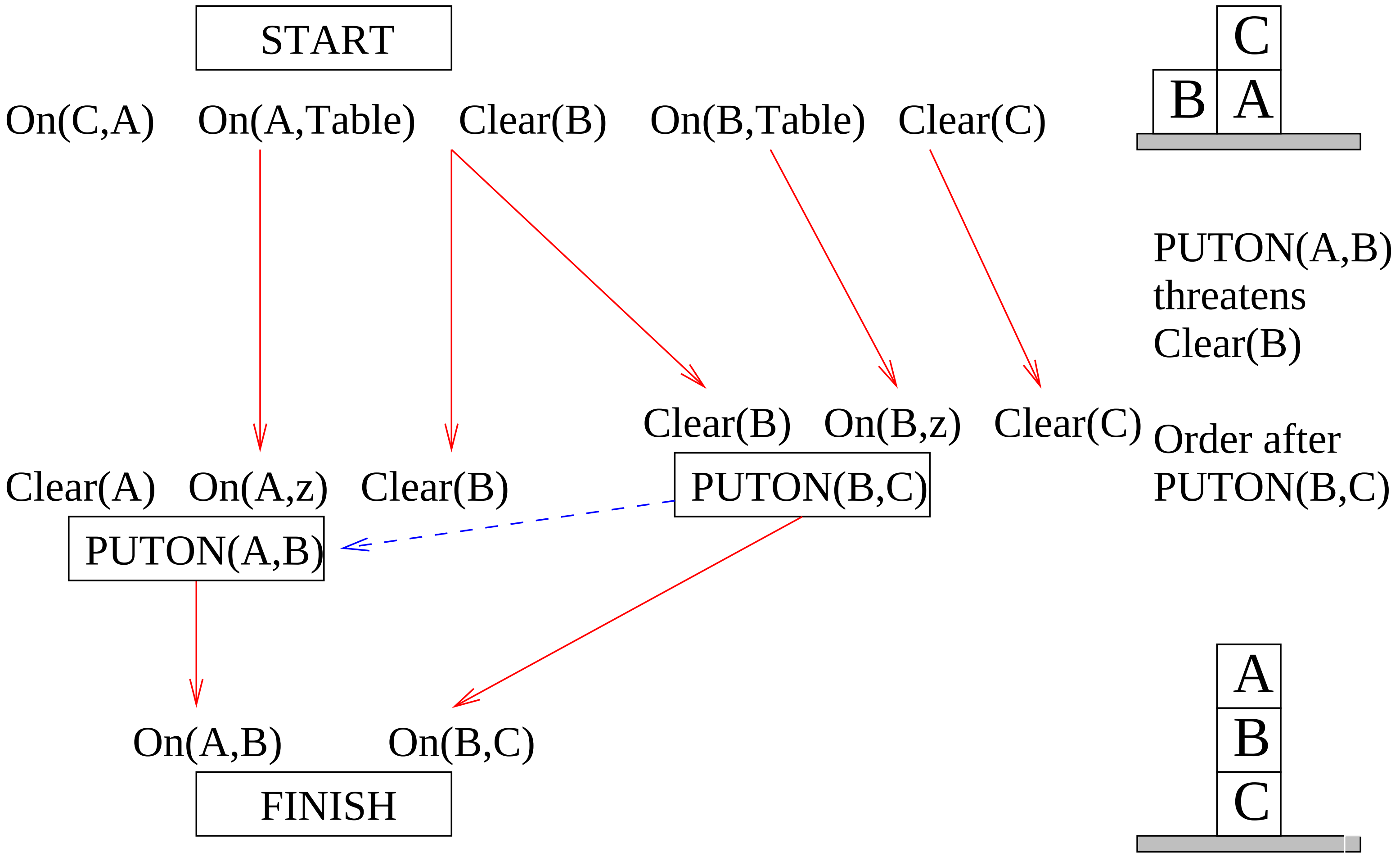


If we try the second goal (on (B,C)) first,
we can't proceed without undoing work.

Sussman anomaly (cont'd)



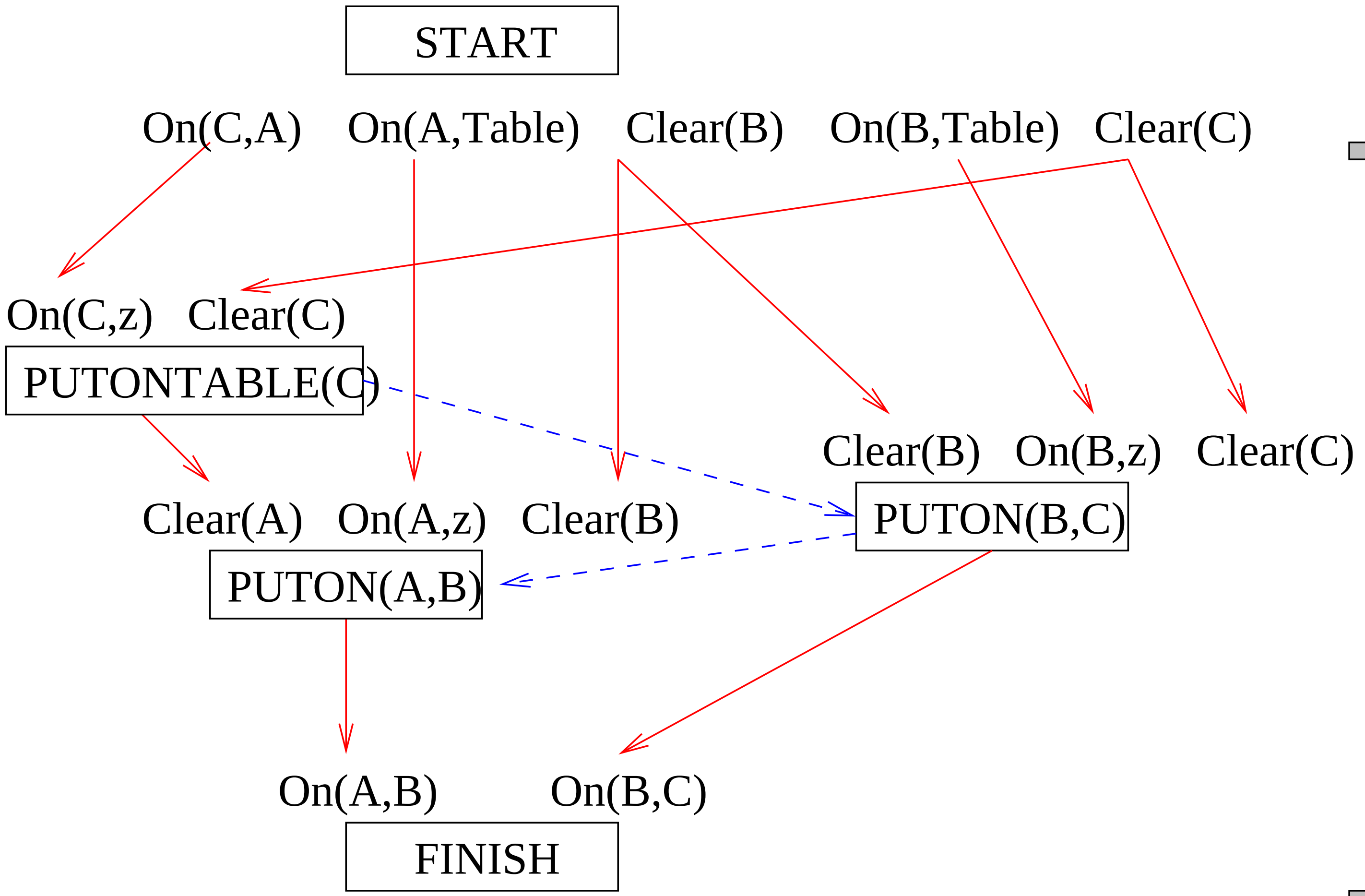
Sussman anomaly (cont'd)

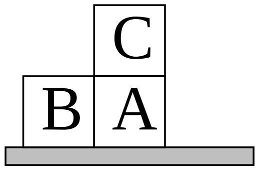


PUTON(A,B) threatens Clear(B)

Order after PUTON(B,C)

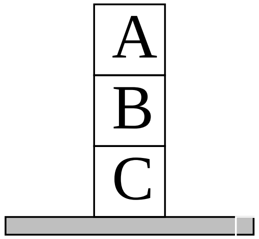
Sussman anomaly (cont’d)





PUTON(B,C) threatens Clear(C)

order after PUTON(A,B)



Applications of planning

- **Robotics**
  - **Shakey, the robot at SRI was the initial motivator.**
  - **However, several other techniques are used for path-planning etc.**
  - **Most robotic systems are *reactive.***
- **Games**
The story is a plan and a different one can be constructed for each game.
- **Web applications**
Formulating query plans, using web services.
- **Crisis response**
Oil spill, forest fire, emergency evacuation.

Applications of planning (cont'd)

- **Space**
  Autonomous spacecraft, self-healing systems.
- **Device control**
  Elevator control, control software for modular devices.
- **Military planning.**
- **And many others.**

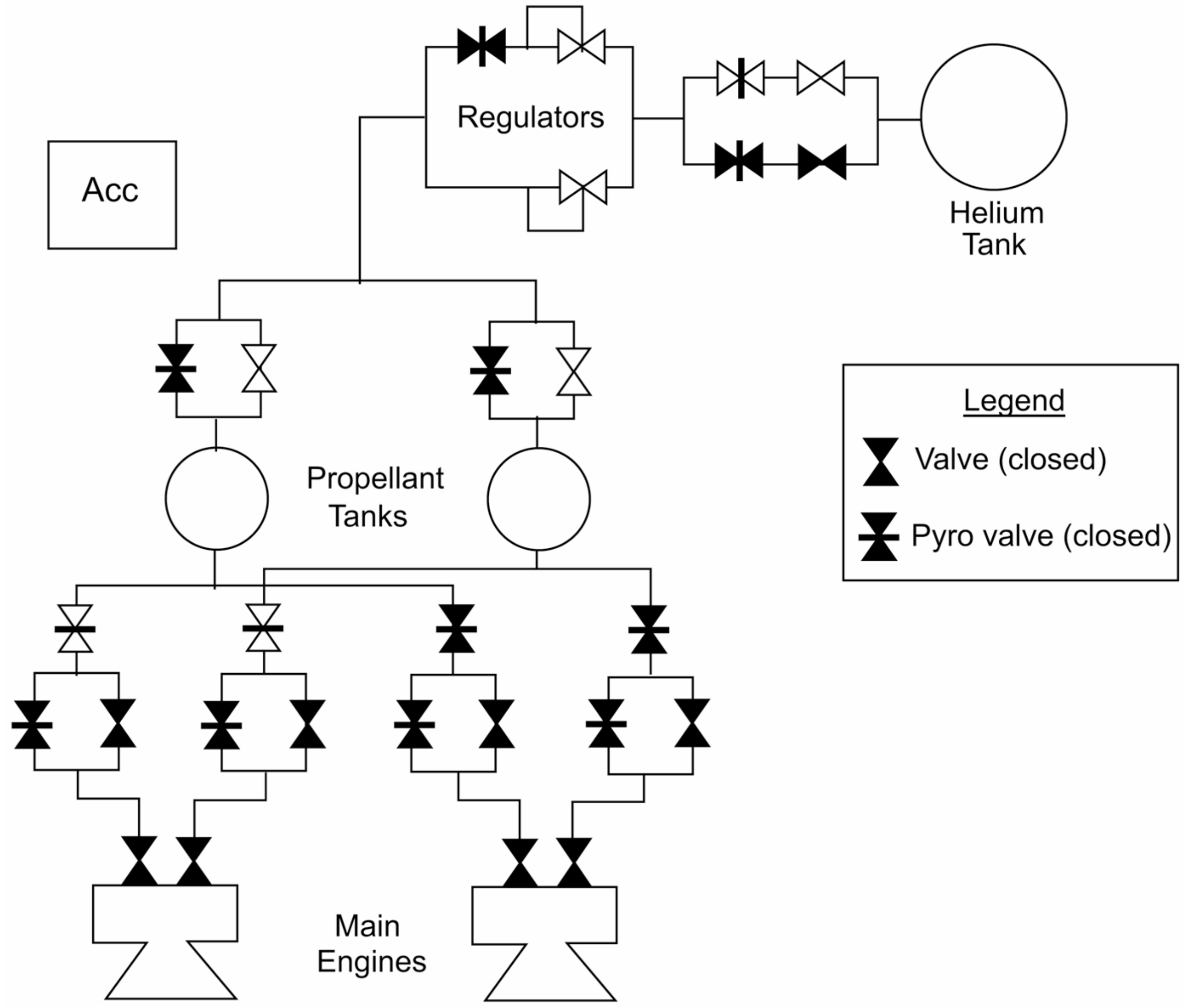
Model-based reactive configuration management (Williams and Nayak, 1996a)

Intelligent space probes that autonomously explore the solar system.

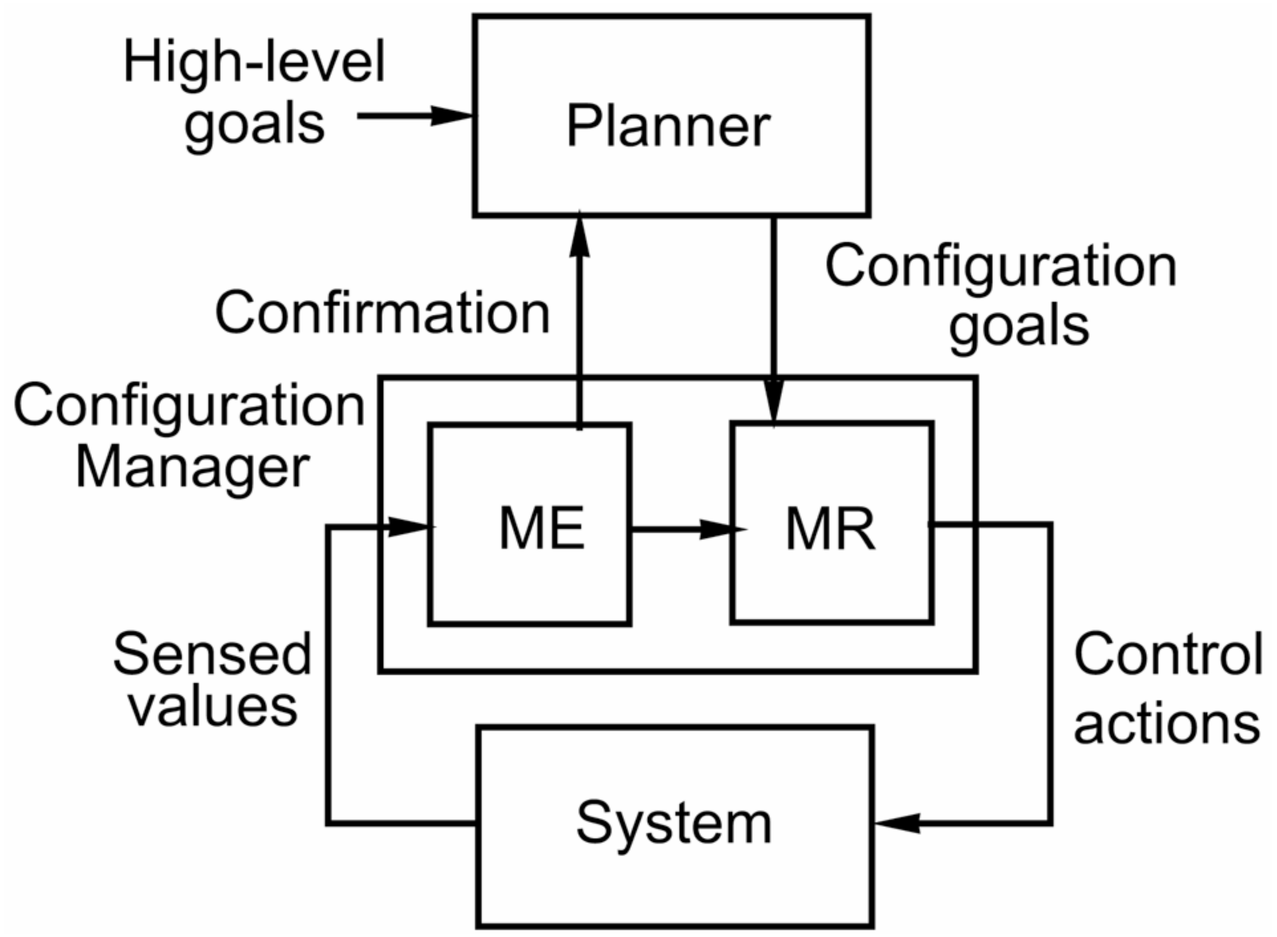
The spacecraft needs to:

- **radically reconfigure its control regime in response to failures,**
- **plan around these failures during its remaining flight.**

A schematic of the simplified Livingstone propulsion system (Williams and Nayak ,1996)

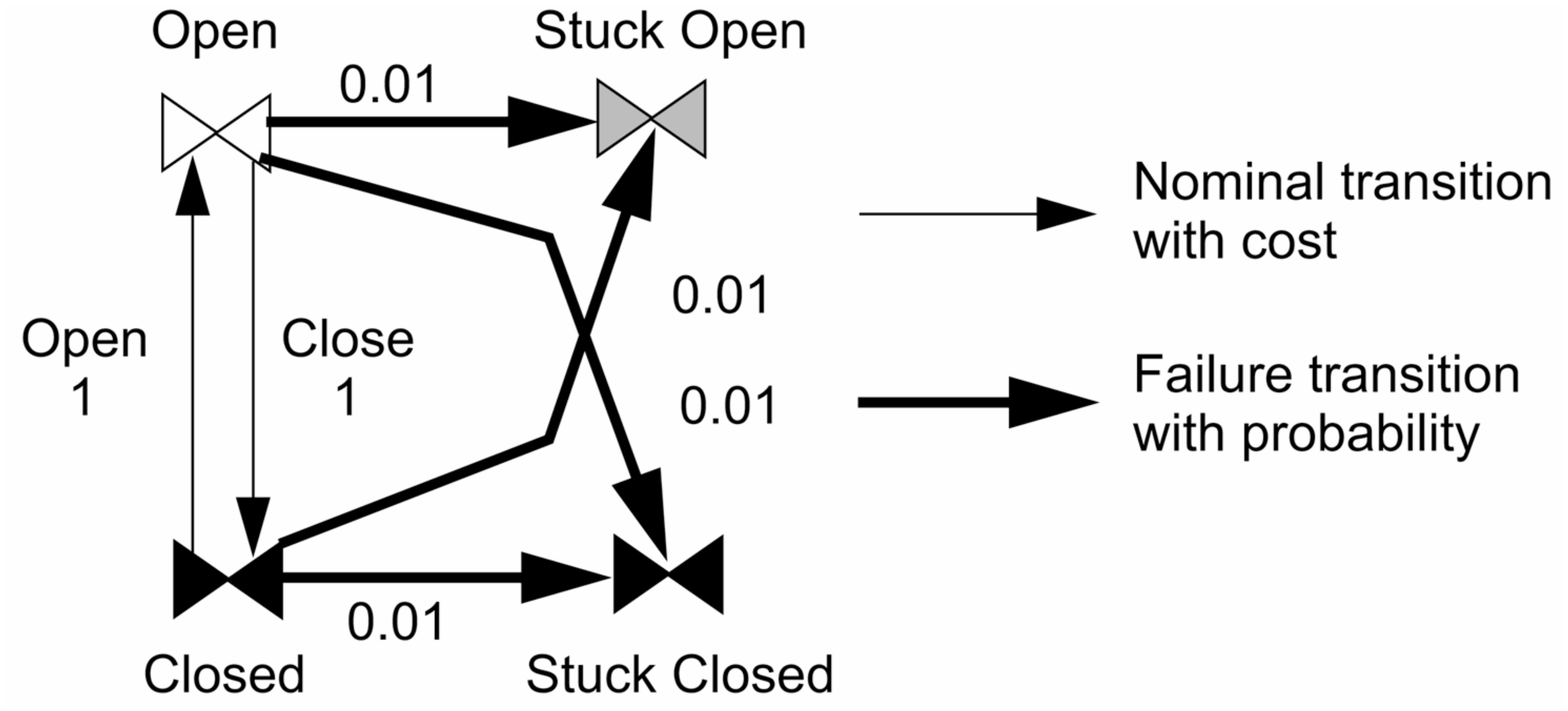


A model-based configuration management system (Williams and Nayak, 1996)

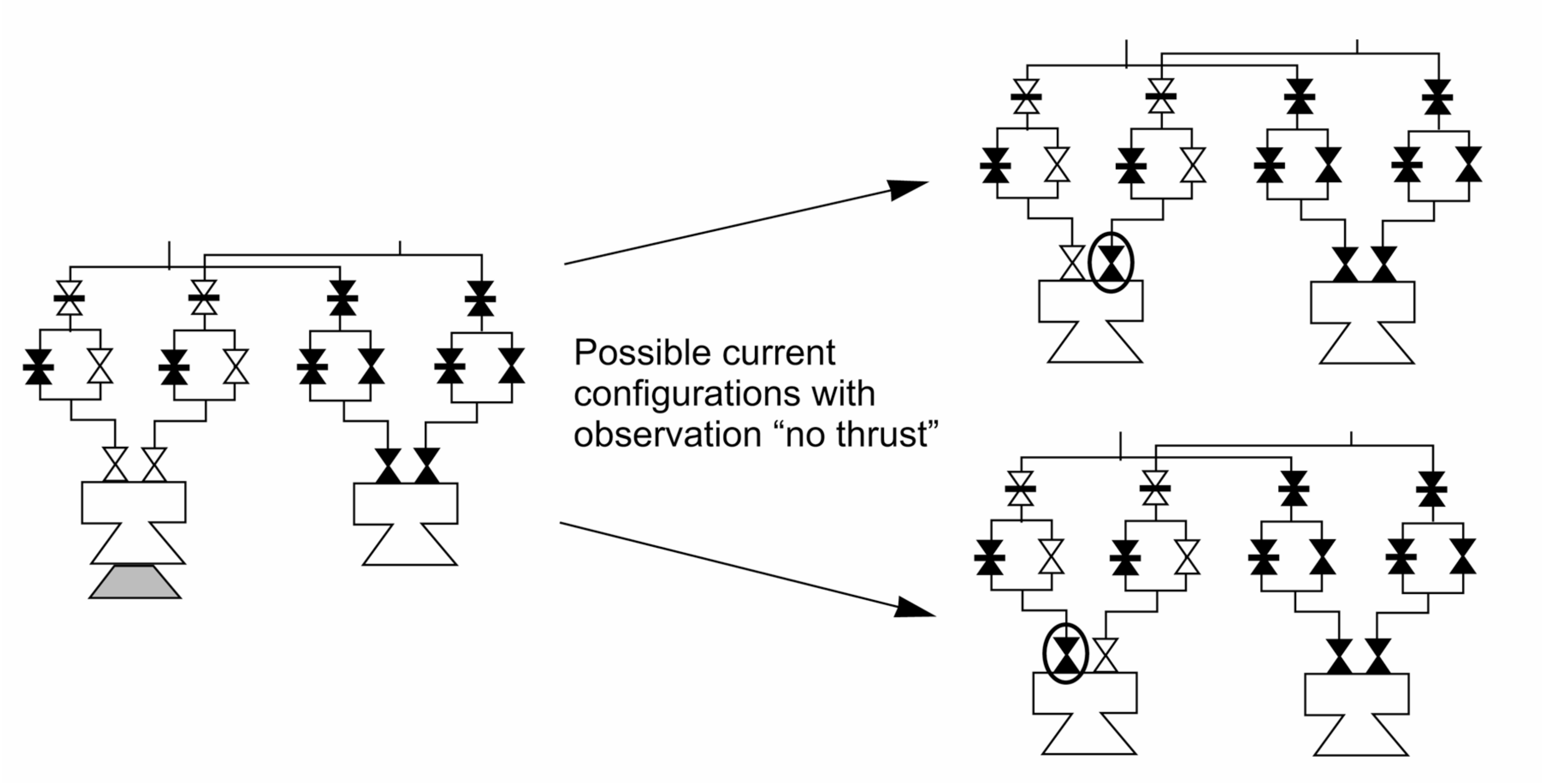


ME: mode estimation **MR: mode reconfiguration**

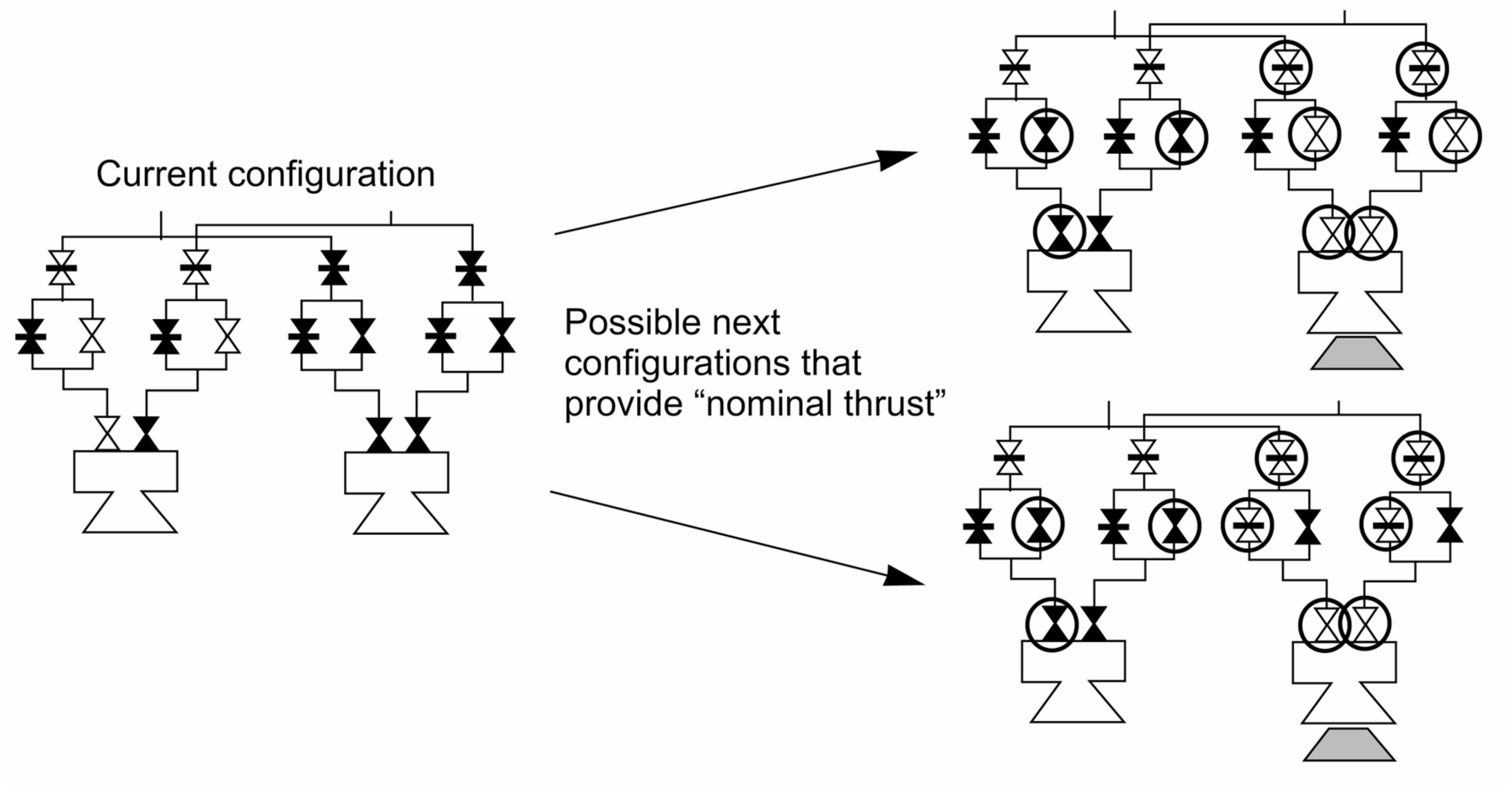
The transition system model of a valve (Williams and Nayak, 1996a)


Open
Stuck Open
0.01
Nominal transition with cost
0.01
Open 1
Close 1
Failure transition with probability
0.01
0.01
Closed
Stuck Closed

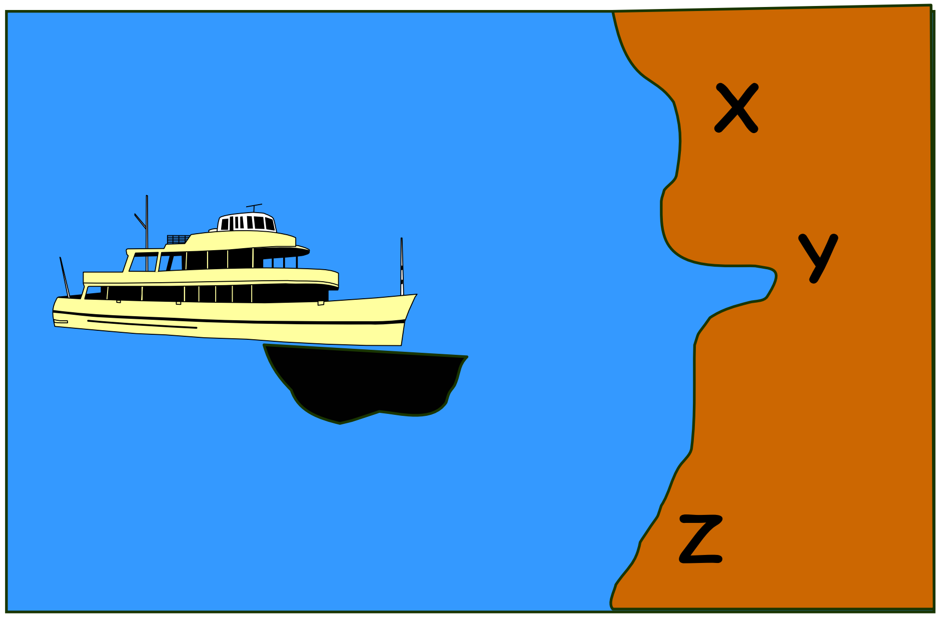
Mode estimation (Williams and Nayak, 1996a)


Possible current configurations with observation “no thrust”

Mode reconfiguration (MR)
(Williams and Nayak, 1996a)



Oil spill response planning

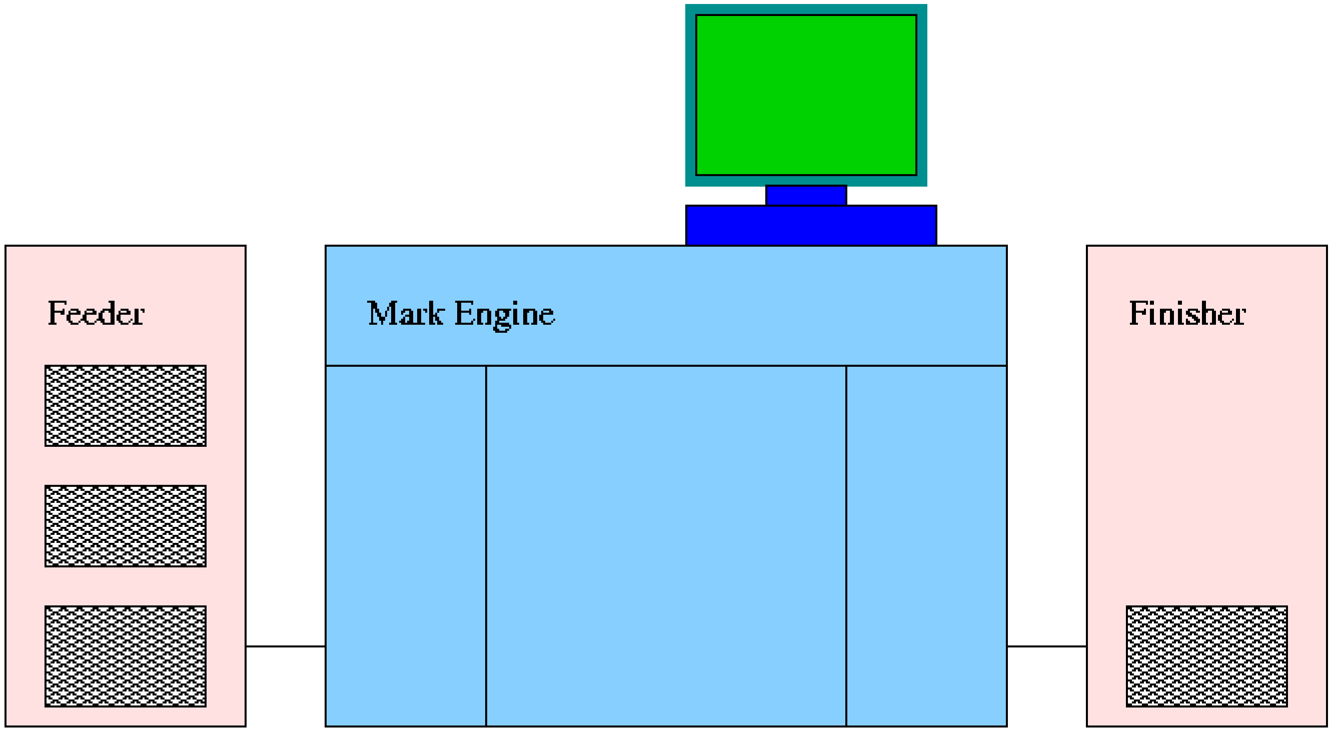


(Desimone & Agosto 1994)

Main goals: stabilize discharge, clean water, protect sensitive shore areas

The objective was to estimate the equipment required rather than to execute the plan

A modern photocopier

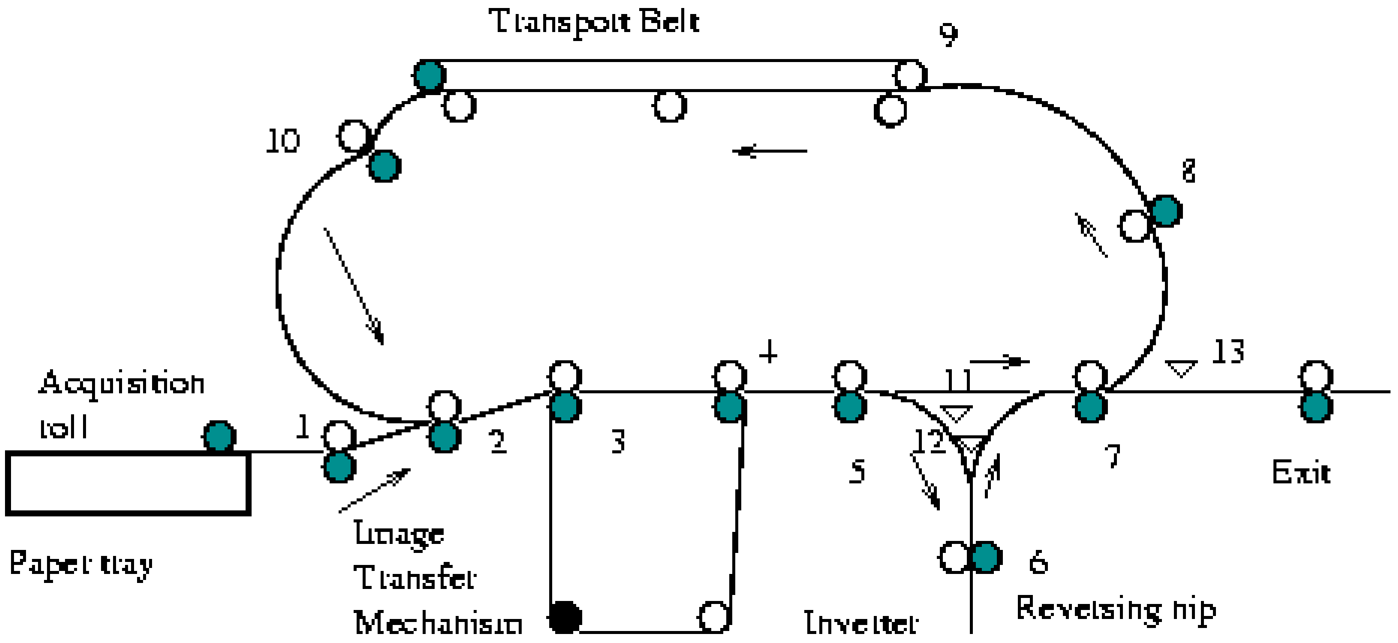


(From a paper by Fromherz et al. 2003)

Main goal: produce the documents as requested by the user

Rather than writing the control software, write a controller that produces and executes plans

The paper path



Monkeys can plan

Why shouldn’t computers?

The problem statement: A monkey is in a laboratory room containing a box, a knife and a bunch of bananas. The bananas are hanging from the ceiling out of the reach of the monkey. How can the monkey obtain the bananas?

VHPOP coding of the monkey and bananas problem

```
(define (domain monkey-domain)
  (:requirements :equality)
  (:constants monkey box knife glass water
    waterfountain)
  (:predicates (on-floor) (at ?x ?y) (onbox ?x)
    (hasknife) (hasbananas) (hasglass) (haswater)
    (location ?x)

  (:action go-to
    :parameters (?x ?y)
    :precondition (and (not = ?y ?x)) (on-floor)
      (at monkey ?y)
    :effect (and (at monkey ?x) (not (at monkey ?y))))
```

VHPOP coding (cont'd)

```
(:action climb
  :parameters (?x)
  :precondition (and (at box ?x) (at monkey ?x))
  :effect (and (onbox ?x) (not (on-floor))))

(:action push-box
  :parameters (?x ?y)
  :precondition (and (not (= ?y ?x)) (at box ?y)
    (at monkey ?y) (on-floor))
  :effect (and (at monkey ?x) (not (at monkey ?y))
    (at box ?x) (not (at box ?y))))
```

VHPOP coding (cont'd)

```
(:action get-knife
  :parameters (?y)
  :precondition (and (at knife ?y) (at monkey ?y))
  :effect (and (hasknife) (not (at knife ?y))))

(:action grab-bananas
  :parameters (?y)
  :precondition (and (hasknife) (at bananas ?y)
    (onbox ?y) )
  :effect (hasbananas))
```

VHPOP coding (cont'd)

```
(:action pick-glass
  :parameters (?y)
  :precondition (and (at glass ?y) (at monkey ?y))
  :effect (and (hasglass) (not (at glass ?y))))

(:action get-water
  :parameters (?y)
  :precondition (and (hasglass) (at waterfountain ?y)
    (ay monkey ?y) (onbox ?y))
  :effect (haswater))
```

Problem 1: monkey-test1.pddl

```
(define (problem monkey-test1)
  (:domain monkey-domain)
  (:objects p1 p2 p3 p4)
  (:init (location p1) (location p2)
         (location p3) (location p4)
         (at monkey p1) (on-floor)
         (at box p2) (at bananas p3)
         (at knife p4))
  (:goal (hasbananas)))
```

```
go-to p4 p1
get-knife p4
go-to p2 p4
push-box p3 p2
climb p3
grab-bananas p3                  time = 30 msec.
```

Problem 2: monkey-test2.pddl

```
(define (problem monkey-test2)
  (:domain monkey-domain)
  (:objects p1 p2 p3 p4 p6)
  (:init (location p1) (location p2)
          (location p3) (location p4) (location p6)
          (at monkey p1) (on-floor)
          (at box p2) (at bananas p3) (at knife p4)
          (at waterfountain p3) (at glass p6))
  (:goal (and (hasbananas) (haswater))))
```

```
go-to p4 p1                    go-to p2 p6
get-knife p4                   push-box p3 p2
go-to p6 p4                    climb p3
pick-glass p6                  get-water p3
                               grab-bananas p3
```

time = 70 msec.

Planning representation

Suppose that the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Can this goal be solved by a STRIPS-style system?

```
;
; Modelling the Wumpus World in PDDL: 1st try...
; by: Patrik Haslum
; Source web page:
;   http://users.cecs.anu.edu.au/~patrik/pddlman/wumpus.html
;

(define (domain wumpus-a)
  (:requirements :strips) ;; maybe not necessary

  (:predicates
   (adj ?square-1 ?square-2)
   (pit ?square)
   (at ?what ?square)
   (have ?who ?what)
   (dead ?who))

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (adj ?from ?to)
                       (not (pit ?to))
                       (at ?who ?from))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?arrow ?victim ?where-victim)
    :precondition (and (have ?who ?arrow)
                       (at ?who ?where)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (at ?victim ?where-victim))
                 (not (have ?who ?arrow)))
    )
)
```

```
(define (problem wumpus-a-1)
  (:domain wumpus-a)
  (:objects
   sq-1-1 sq-1-2 sq-1-3
   sq-2-1 sq-2-2 sq-2-3
   the-gold
   the-arrow
   agent
   wumpus)

  (:init
   (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
   (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
   (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
   (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
   (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
   (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
   (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)

   (pit sq-1-2)

   (at the-gold sq-1-3)
   (at agent sq-1-1)
   (have agent the-arrow)
   (at wumpus sq-2-3))

  (:goal (and (have agent the-gold) (at agent sq-1-1)))
  )

Resulting plan:

(MOVE THE-GOLD SQ-1-3 SQ-2-3)
(MOVE THE-GOLD SQ-2-3 SQ-2-2)
(MOVE THE-GOLD SQ-2-2 SQ-2-1)
(MOVE THE-GOLD SQ-2-1 SQ-1-1)
(TAKE AGENT THE-GOLD SQ-1-1)
```

```
;
; Modelling the Wumpus World in PDDL: 2nd try...
; by: Patrik Haslum
; Source web page:
;   http://users.cecs.anu.edu.au/˜patrik/pddlman/wumpus.html
;

(define (domain wumpus-b)
  (:requirements :strips)
  (:predicates
   (adj ?square-1 ?square-2)
   (pit ?square)

   (at ?what ?square)
   (have ?who ?what)

   (takeable ?what)
   (is-gold ?what)
   (is-arrow ?what)

   (alive ?who)
   (dead ?who))

  (:action move
    :parameters (?who ?from ?to)
    :precondition (and (alive ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (alive ?who)
                       (takeable ?what)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?arrow ?victim ?where-victim)
    :precondition (and (alive ?who)
                       (have ?who ?arrow)
                       (is-arrow ?arrow)
                       (at ?who ?where)
                       (alive ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (alive ?victim))
                 (not (at ?victim ?where-victim))
                 (not (have ?who ?arrow)))
    )
)
```

```
(define (problem wumpus-b-1)
  (:domain wumpus-b)
  (:objects sq-1-1 sq-1-2 sq-1-3
            sq-2-1 sq-2-2 sq-2-3
            the-gold the-arrow
            agent wumpus)
  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
         (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
         (pit sq-1-2)
         (at the-gold sq-1-3)
         (is-gold the-gold)
         (takeable the-gold)
         (at agent sq-1-1)
         (alive agent)
         (have agent the-arrow)
         (is-arrow the-arrow)
         (takeable the-arrow)
         (at wumpus sq-2-3)
         (alive wumpus))
  (:goal (and (have agent the-gold)
              (at agent sq-1-1)
          ))
  )

Resulting plan:

(MOVE AGENT SQ-1-1 SQ-2-1)
(MOVE AGENT SQ-2-1 SQ-2-2)
(MOVE AGENT SQ-2-2 SQ-2-3)
(MOVE AGENT SQ-2-3 SQ-1-3)
(TAKE AGENT THE-GOLD SQ-1-3)
(MOVE AGENT SQ-1-3 SQ-2-3)
(MOVE AGENT SQ-2-3 SQ-2-2)
(MOVE AGENT SQ-2-2 SQ-2-1)
(MOVE AGENT SQ-2-1 SQ-1-1)
```

```
;
; Modelling the Wumpus World in PDDL: 3rd time’s a charm...
; by: Patrik Haslum
; Source web page:
;   http://users.cecs.anu.edu.au/˜patrik/pddlman/wumpus.html
;

(define (domain wumpus-c)
  (:requirements :strips)
  (:predicates
   (at ?what ?square)
   (adj ?square-1 ?square-2)
   (pit ?square)
   (wumpus-in ?square)
     ;; <-> (exists ?x (and (is-wumpus ?x) (at ?x ?square) (not (dead ?x))
   (have ?who ?what)
   (is-agent ?who)
   (is-wumpus ?who)
   (is-gold ?what)
   (is-arrow ?what)
   (dead ?who))

  (:action move-agent
    :parameters (?who ?from ?to)
    :precondition (and (is-agent ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to))
                       (not (wumpus-in ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to))
    )

  (:action take
    :parameters (?who ?what ?where)
    :precondition (and (is-agent ?who)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who ?where ?with-what ?victim ?where-victim)
    :precondition (and (is-agent ?who)
                       (have ?who ?with-what)
                       (is-arrow ?with-what)
                       (at ?who ?where)
                       (is-wumpus ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (dead ?victim)
                 (not (wumpus-in ?where-victim))
                 (not (have ?who ?with-what)))
    )
  (:action move-wumpus
    :parameters (?who ?from ?to)
    :precondition (and (is-wumpus ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                       (not (pit ?to))
                       (not (wumpus-in ?to)))
    :effect (and (not (at ?who ?from))
                 (at ?who ?to)
                 (not (wumpus-in ?from))
                 (wumpus-in ?to))
    )

)

(define (problem wumpus-c-1)
  (:domain wumpus-c)
  (:objects sq-1-1 sq-1-2 sq-1-3
            sq-2-1 sq-2-2 sq-2-3
            the-gold the-arrow
            agent wumpus)
  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
         (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
         (pit sq-1-2)
         (is-gold the-gold)
         (at the-gold sq-1-3)
         (is-agent agent)
         (at agent sq-1-1)
         (is-arrow the-arrow)
         (have agent the-arrow)
         (is-wumpus wumpus)
         (at wumpus sq-2-3)
         (wumpus-in sq-2-3))
  (:goal (and (have agent the-gold) (at agent sq-1-1)))
  )

Resulting plan:

(MOVE-AGENT AGENT SQ-1-1 SQ-2-1)
(MOVE-AGENT AGENT SQ-2-1 SQ-2-2)
(SHOOT AGENT SQ-2-2 THE-ARROW WUMPUS SQ-2-3)
(MOVE-AGENT AGENT SQ-2-2 SQ-2-3)
(MOVE-AGENT AGENT SQ-2-3 SQ-1-3)
(TAKE AGENT THE-GOLD SQ-1-3)
(MOVE-AGENT AGENT SQ-1-3 SQ-2-3)
(MOVE-AGENT AGENT SQ-2-3 SQ-2-2)
(MOVE-AGENT AGENT SQ-2-2 SQ-2-1)
(MOVE-AGENT AGENT SQ-2-1 SQ-1-1)
```

```
;
; Modelling the Wumpus World in PDDL: using ADL...
; by: Patrik Haslum
; Source web page:
;   http://users.cecs.anu.edu.au/~patrik/pddlman/wumpus.html
;

(define (domain wumpus-adl)
  (:requirements :adl :typing)

  ;; object types
  (:types agent wumpus gold arrow square)

  (:predicates
   (adj ?square-1 ?square-2 - square)
   (pit ?square - square)
   (at ?what ?square)
   (have ?who ?what)
   (alive ?who))

  (:action move
    :parameters (?who - agent ?from - square ?to - square)
    :precondition (and (alive ?who)
                       (at ?who ?from)
                       (adj ?from ?to)
                        )
    :effect (and (not (at ?who ?from))
                 (at ?who ?to)

                 (when (pit ?to)
                   (and (not (alive ?who))))

                 (when (exists (?w - wumpus) (and (at ?w ?to) (alive ?w)))
                   (and (not (alive ?who)))))
    )

  (:action take
    :parameters (?who - agent ?where - square ?what)
    :precondition (and (alive ?who)
                       (at ?who ?where)
                       (at ?what ?where))
    :effect (and (have ?who ?what)
                 (not (at ?what ?where)))
    )

  (:action shoot
    :parameters (?who - agent ?where - square ?with-arrow - arrow
                 ?victim - wumpus ?where-victim - square)
    :precondition (and (alive ?who)
                       (have ?who ?with-arrow)
                       (at ?who ?where)
                       (alive ?victim)
                       (at ?victim ?where-victim)
                       (adj ?where ?where-victim))
    :effect (and (not (alive ?victim))
                 (not (have ?who ?with-arrow)))
    )
)
```

```
(define (problem wumpus-adl-1)
  (:domain wumpus-adl)

  (:objects
   sq-1-1 sq-1-2 sq-1-3 sq-2-1 sq-2-2 sq-2-3 - square
   the-gold - gold
   the-arrow - arrow
   agent-1 - agent
   wumpus-1 - wumpus)

  (:init (adj sq-1-1 sq-1-2) (adj sq-1-2 sq-1-1)
         (adj sq-1-2 sq-1-3) (adj sq-1-3 sq-1-2)
         (adj sq-2-1 sq-2-2) (adj sq-2-2 sq-2-1)
         (adj sq-2-2 sq-2-3) (adj sq-2-3 sq-2-2)
         (adj sq-1-1 sq-2-1) (adj sq-2-1 sq-1-1)
         (adj sq-1-2 sq-2-2) (adj sq-2-2 sq-1-2)
         (adj sq-1-3 sq-2-3) (adj sq-2-3 sq-1-3)
         (pit sq-1-2)
         (at the-gold sq-1-3)
         (at agent-1 sq-1-1)
         (alive agent-1)
         (have agent-1 the-arrow)
         (at wumpus-1 sq-2-3)
         (alive wumpus-1))

  (:goal (and (have agent-1 the-gold) (at agent-1 sq-1-1) (alive agent-1)))
  )
```

Comments on planning

- It is a synthesis task.
- Classical planning is based on the assumptions of a deterministic and static environment.
- Theorem proving and situation calculus are not widely used nowadays for planning (see below).
- Algorithms to solve planning problems include:
  - forward chaining: heuristic search in state space
  - Graphplan: mutual exclusion reasoning using plan graphs
  - Partial order planning (POP): goal directed search in plan space
  - Satifiability based planning: convert problem into logic

Comments on planning (cont'd)

- **Non-classical planners include:**
  - **probabilistic planners**
  - **contingency planners (a.k.a. conditional planners)**
  - **decision-theoretic planners**
  - **temporal planners**
  - **resource based planners**

Comments on planning (cont'd)

- **In addition to plan generation algorithms we also need algorithms for**
  - **Carrying out the plan**
  - **Monitoring the execution**
    (because the plan might not work as expected; or the world might change)
    (need to maintain the consistency between the world and the program's internal model of the world)
  - **Recovering from plan failures**
  - **Acting on new opportunities that arise during execution**
  - **Learning from experience**
    (save and generalize good plans)