AUTOMATED PLANNING

Automated Planning is an important problem solving activity which consists in synthesizing a ***sequence of actions*** performed by an agent that leads from an initial state of the world to a given target state (**goal**)

AUTOMATED PLANNING

Given:

- an initial state
- a set of actions you can perform
- a state to achieve (**goal**)

Find:

a plan: a partially or totally ordered set of actions needed to achieve the goal from the initial state

Planning is:

- one application per se
- A common activity in many applications such as
  - diagnosis: test plans and actions to repair (reconfigure) a system
  - scheduling
  - robotics

AUTOMATED PLANNING: basics

An **automated planner** is an *intelligent agent* that operates in a certain domain described by:

1. ***a representation of the initial state***
2. ***a representation of a goal***
3. ***a formal description of the executable actions (also called operators)***

It dynamically summarizes the plan of actions needed to reach the goal from the initial state.

ACTION REPRESENTATION

- A planner relies on a formal description of the executable actions. It is called ***Domain Theory***.
- Each action is identified by a name and declaratively modeled through ***preconditions*** is ***postconditions***.
- Preconditions are the conditions which must hold to ensure that the action can be executed;
- Postconditions represent the effects of the action on the world.
- Often the Domain Theory consists of operators containing variables that define **classes of actions**. A different instantiation of the variables corresponds to a different action.

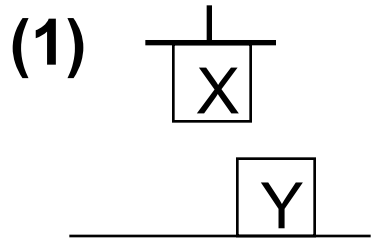
EXAMPLE: THE BLOCK WORLD

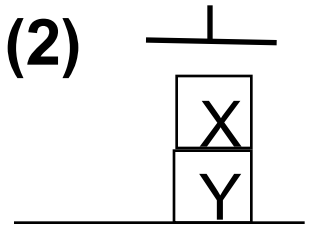
Actions:

STACK (X, Y)

IF: holding (X) and clear (Y)

THEN: handempty and clear (X) and on (X, Y);

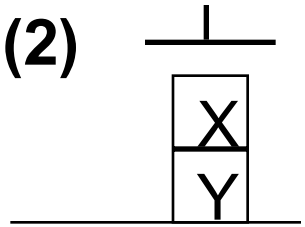


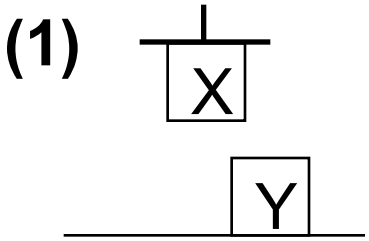


UNSTACK (X, Y)

IF: handempty and clear (X) and on (X, Y)

THEN: holding (X) and clear (Y);



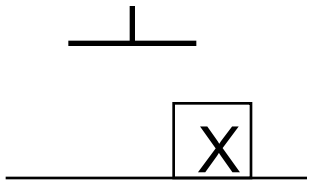


EXAMPLE: THE BLOCK WORLD

PICKUP (X)

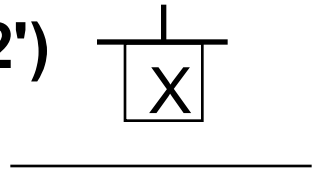
IF: ontable (X) and clear (X) and handempty

(1')



THEN: holding (X);

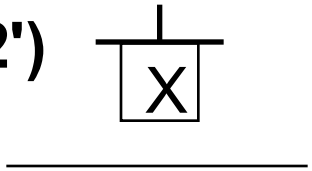
(2')



Putdown (X)

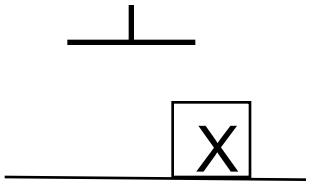
IF: holding (X)

(2')



THEN: ontable (X) and clear (X) and handempty.

(1')



PLANNING TECHNIQUES

- Deductive planning / planning as search/ linear planning in ***Foundations of AI***
- Nonlinear planning
  - Partial Order Planning (POP)
- Hierarchical planning
- Conditional planning
- Graph-based planning
- Planning for robotics paths
- Planning as emergent behavior: swarm intelligence

Search in the space of Plans

- Linear planners are search algorithms that explore the state space.
  - The plan is a linear sequence of actions to achieve the goals.
- Non-linear planners are search algorithms that generate a plan as a search problem in the space of plans.
  - In the search tree each node is a partial plan and operators are plan refinement operations.
- A non-linear generative planner assumes that the initial state is fully known **Closed World Assumption**: Everything that is not explicitly stated in the initial state is considered as false
- **Least Commitment planning**: never impose more restrictions than those that are strictly necessary.
  - Avoid making decisions when they are not required. This avoids many backtracking.

NON-LINEAR PLANNING

- A non-linear plan is represented as
  - a set of **actions** (instances of operators)
  - a (not exhaustive) set of **orderings** between actions
  - a set of "**causal links**" (Described later)

- Initial plan: it is an empty plan with two fake actions
  - **start**: No preconditions. Its effects match the initial state
  - **stop**: No effects. Its pre-conditions match the goal
  - Ordering: start <stop

NON-LINEAR PLANNING

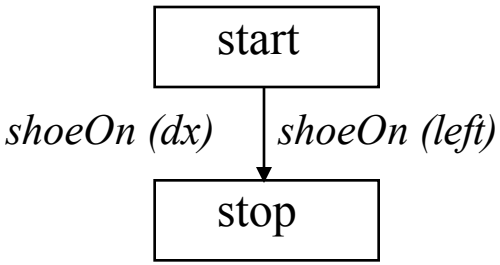
- At each step either the set of operators or the set of orderings or the set of causal links is increased until all goals are met.
  - Non required orderings are not posted
- A solution is a set of partially specified and partially ordered operators.
- To obtain a real plan the partial order should be linearized in one of the possible total orders (*linearization operation*).

Example: plan to wear shoes

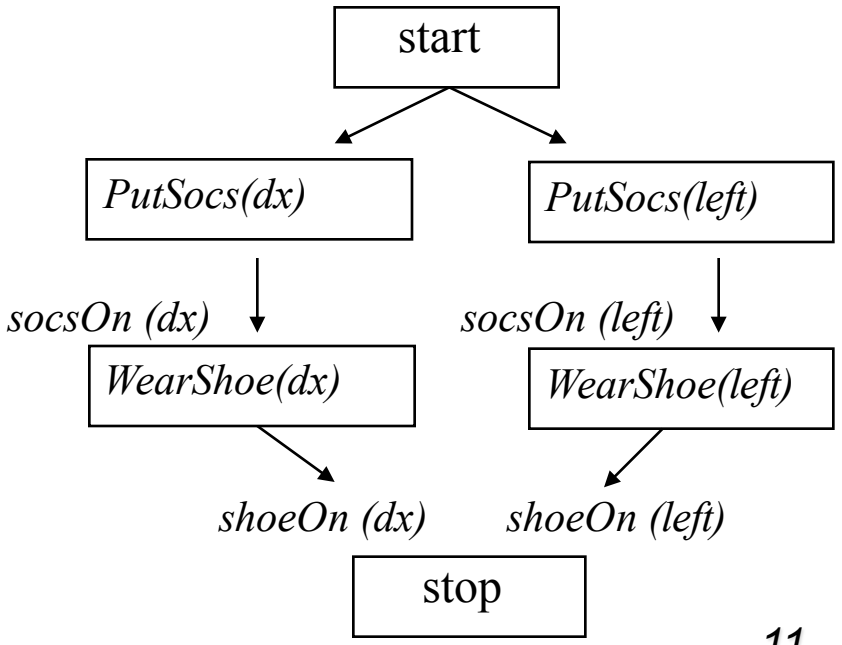
- Goal: shoeOn (dX), shoeOn (sX)

- actions
  - WearShoe(Foot)
    PRECOND: socsOn (Foot)
    EFFECT: shoeOn (Foot)
  - MettiCalza (Foot)
    PRECOND: ¬ socsOn (Foot)
    EFFECT: shoeOn (Foot)

Initial plan



Partial plans Finals



Partial Order Planning: intuitive algorithm

While (plan not complete) do
- select an action SN that has a precondition not satisfied;
- select an action S (new or already in the plan) that has C among its effects;
- add the order constraint S <SN;
- if S is a new action add the constraint Start <S <Stop;
- add the causal link <S, SN, C>;
- solve any threat on causal links

end

- In case of failure if choice points exist, the algorithm backtracks and it explores alternatives.

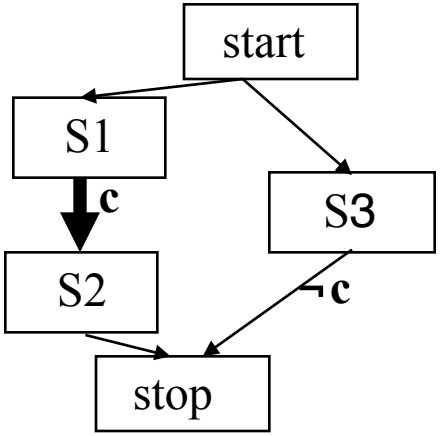
- A **causal link** is a triple that consists of two operators Si, Sj and a subgoal c . C should be precondition of Sj and effect of Si

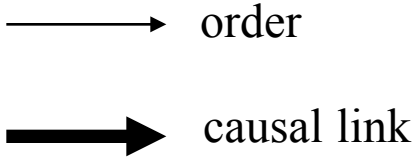
$$S_i \xrightarrow{c} S_j$$

- A causal link stores the causal relations between actions: it traces why a given operator has been introduced in the plan.
- Causal links help tackling the problem of *interacting goals*.

Causal links and threats

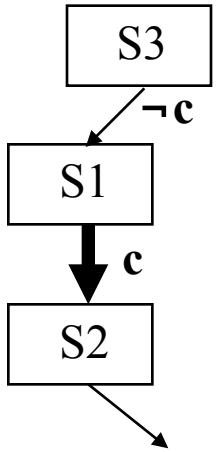
An action *S3* is a **threat** for a causal link <*S1, S2, c*> if it has an effect that negates *c* and no ordering constraint exists that prevents *S3* to be performed between *S1* and *S2*.

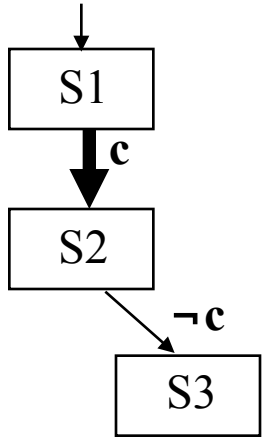




Possible solutions

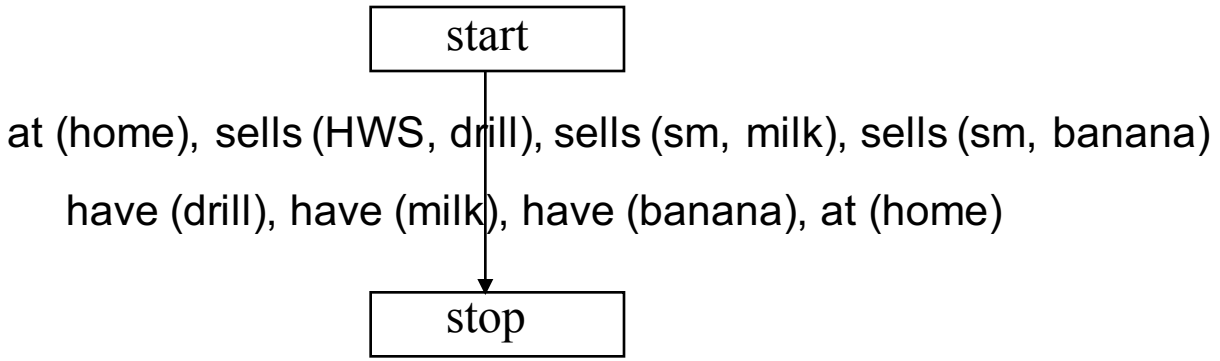
- Demotion: The constraint *S3 <S1 is imposed*
- Promotion: The constraint *S2 <S3 is imposed*





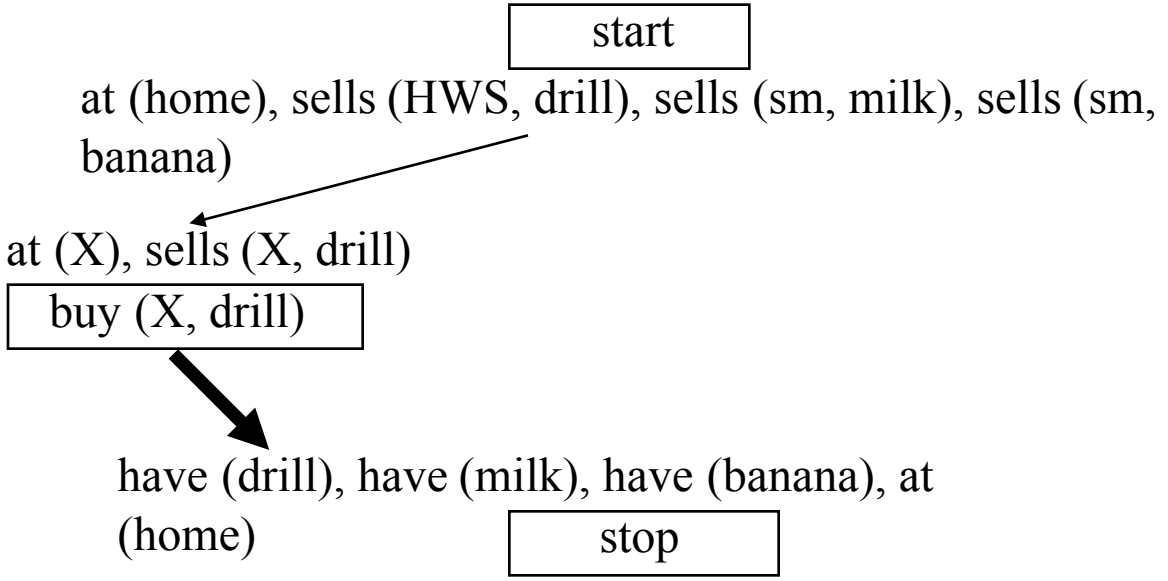
Example: Purchasing Schedule

- Initial state:
  at (home), sells (HWS, drill), sells (sm, milk), sells (sm, banana)
- Goal:
  at (home), have (drill), have (milk), have (banana)
- actions:
  Go (X, Y):
  PRECOND: at (X)
  EFFECT: at (Y), ¬ at (X)
  buy (S, Y):
  PRECOND: at (S), sells (S, Y)
  EFFECT: have (Y)
- initial plan (known *null*):



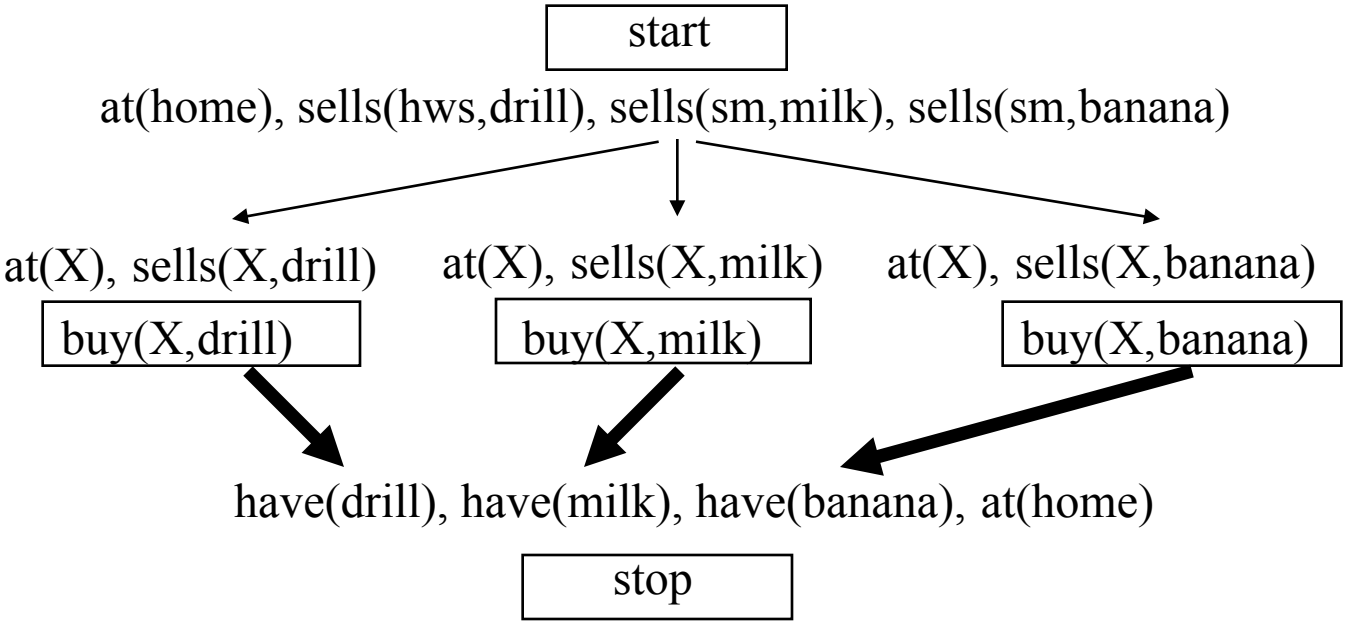
Example: Purchasing Schedule

- First step:
  - select a precondition (goal) to be fulfilled: *have (drill)*
  - select an action that has *have(drill)* as an effect: *buy (X, Y)*
  - Plan refinement:
    - Link variable Y with the term *drill*
    - Impose ordering constraints *Start <buy <Stop*
    - Insert the causal link *<buy (X, drill), stop, have (drill)>*



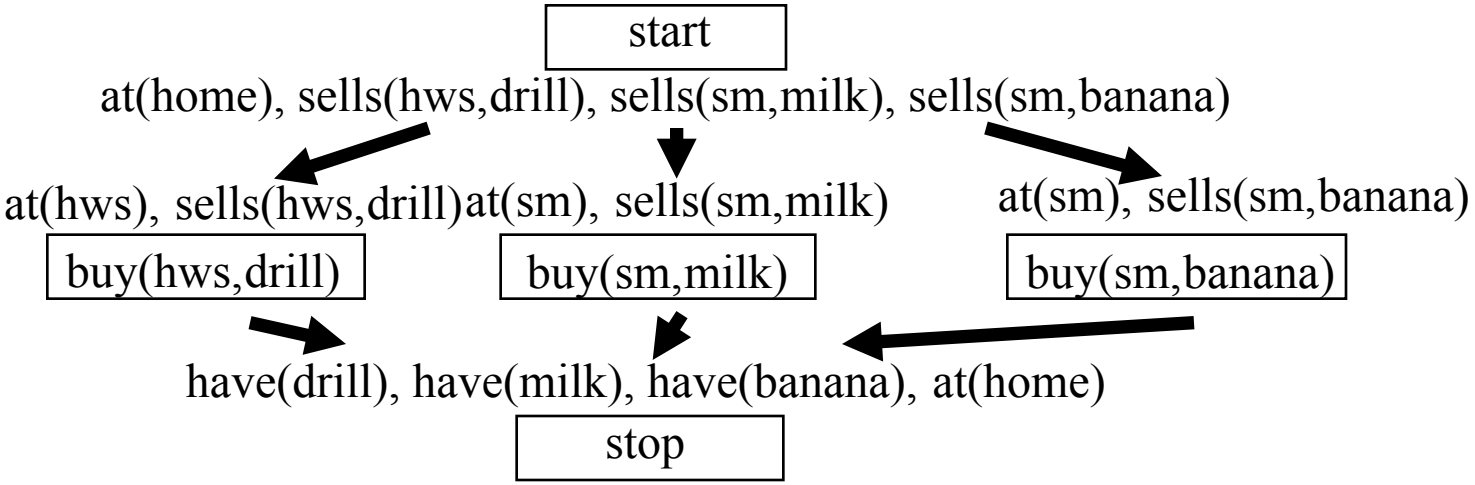
Example: Purchasing Schedule

- Same procedure for
  - *have (milk)*
  - *have (banana)*

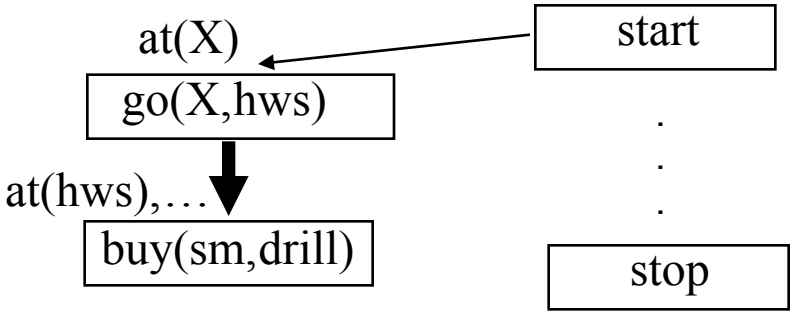


Example: Purchasing Schedule

- Select *sells(X,drill),* true in the initial state imposing *X=hws*. The same happens for *sells(X,milk)* and *sells(X,banana)* with *X=sm*. Add causal links.

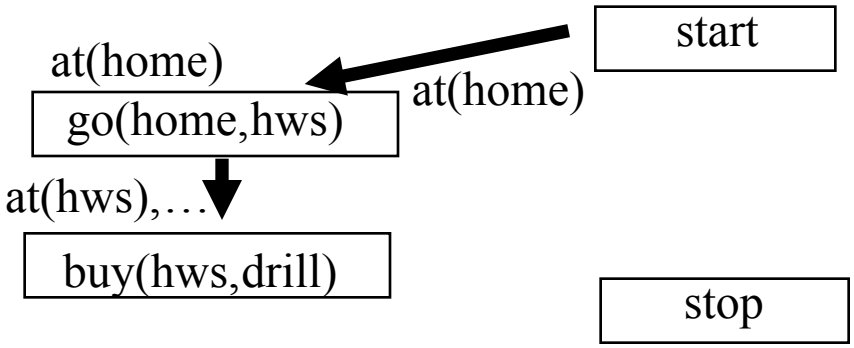


- Select
  - *at(hws)* precondition of *buy(hws,drill)*
  - Add *go(X,hws)* in the plan along with orderings and causal links.

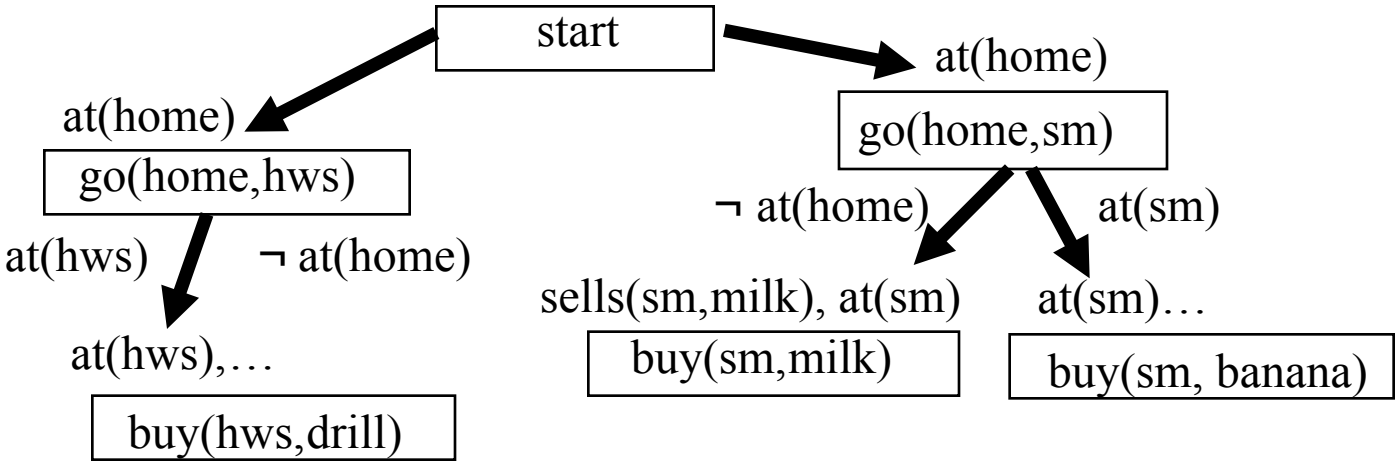


Example: Purchasing Schedule

- Select *at(X)* as precondition of *go(X, hws), true in Start* with *X* = *home* and protect the causal link.

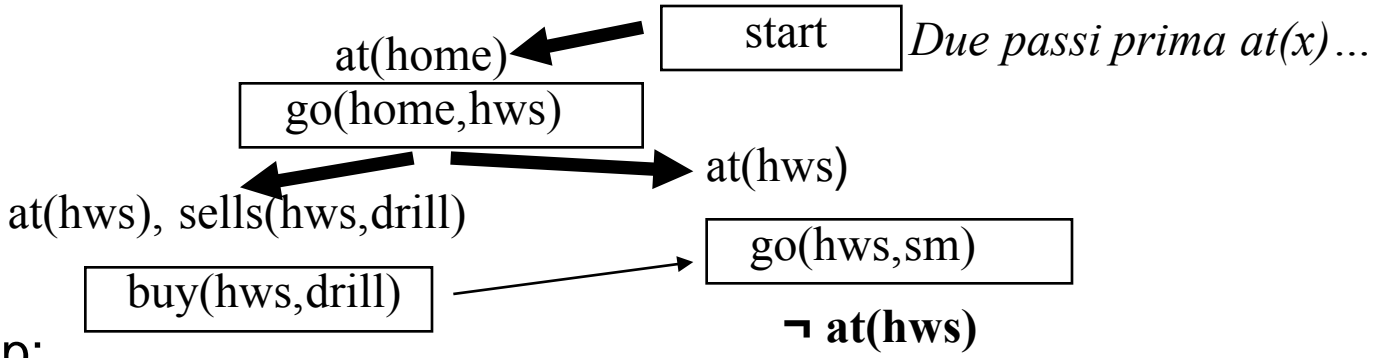


- Same procedure for *at (sm)*

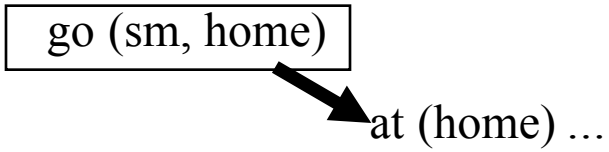


Example: Purchasing Schedule

- Solve the conflict between actions *go(home, hws)* and *go(home, sm)*
  - If the agent performs *go(home, hws)* it cannot be *at(home)* to perform *go(home, sm)* and viceversa
  - imposing ordering constraints does not work
  - backtracking on the solution step of *at(X)* (Precondition of *go(X,sm))*
  - use *go(home, hws)* instead of *Start* with *X = home to satify at(X) with X= hws*
  - So we have *buy(hws, drill) < go(hws, sm). This way at (hws)* is protected by the causal link between *go (home, HWS) and buy (HWS, drill)* **(promotion)**

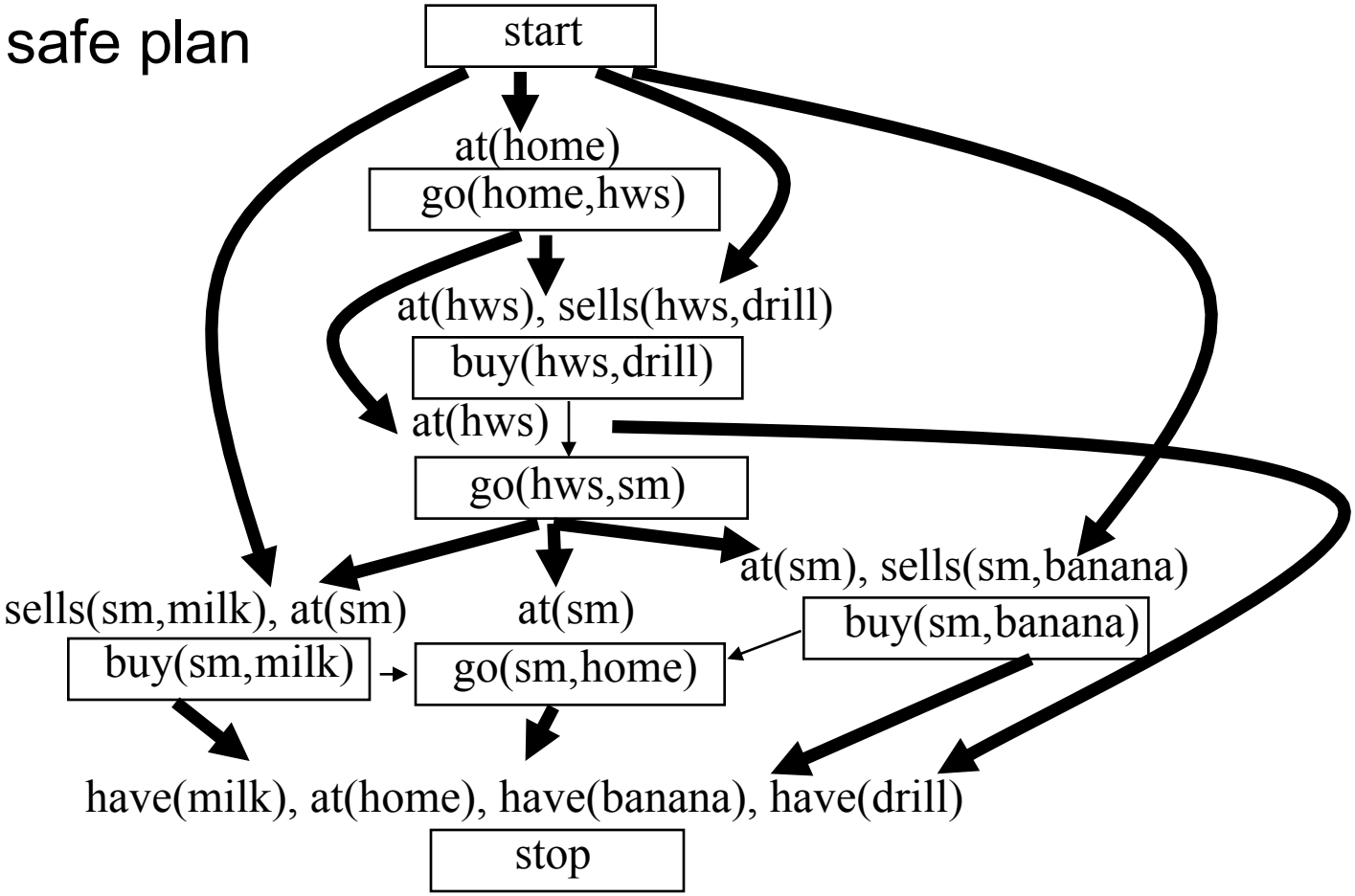


- Last step:
  - solve *at(home)* of *stop*: the only way is to put the action *go(home)* before *stop*



Example: Purchasing Schedule

- Complete and safe plan



A final plan is obtained by ordering all actions:

1) go (home, hws) 2) buy (hws, drill) 3) Go (hws, sm)
4) buy (sm, milk) 5) buy (sm, banana) 6) go (sm, home)

Partial Order Planning Algorithm (POP)

```
function POP (initialGoal Operators) returns plan
    plan: = INITIAL_PLAN (start, stop, initialGoal)
    loop
        if  SOLUTION (plan) then return plan;
        SN, C: = SELECT_SUBGOAL (plan);
        CHOOSE_OPERATOR (plan, operators, SN, C);
        RESOLVE_THREATS (plan)
    end

function SELECT_SUBGOAL (plan)
    select SN from STEPS (plan) with unsolved precondition C;
    return SN, C
```

Partial Order Planning Algorithm (POP)

procedure *CHOOSE_OPERATOR (plan, ops, SN, C)*
    pick an *S* with effect C from *ops* or from *STEPS (plan)*;
    if S does not exist then fail;
    add the causal link <*S, SN, C*>
    add the ordering constraint *S* <*SN*
    if *S* is a new action added to the plan
    then add *S* to *STEPS(plan)*
                add the constraint *Start* <*S* <*Stop*
procedure *SOLVE_THREAT (plan)*
    for each action *S* that threats a causal link between *Si* and *Sj*,
    choose either
        demotion: add the constraint *S* < *Si*
        promotion: add the constraint *Sj* <*S*
        if *NOT_CONSISTENT (plan)* then fail

Modal Truth Criterion (MTC)

Promotion and *demotion* alone are not enough to ensure the completeness of the planner. A planner is complete if it always finds a solution if a solution exists.

The *Modal Truth Criterion* is a construction process that guarantees planner completeness.

A Partial Order Planning algorithm interleaves goal achievement steps with threat protection steps.

The MTC provides five plan refinement methods (one for the open goal achievement and 4 for threat protection) that ensure the completeness of the planner.

Modal Truth Criterion (MTC)

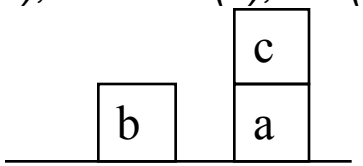
1. ***Establishment***, open goal achievement by means of: (1) a new action to be inserted in the plan, (2) an ordering constraint with Action already in the plan or simply (3) of a variable assignment.
2. ***Promotion***, ordering constraint that imposes the threatening action before the first of the causal link;
3. ***Demotion***, ordering constraint that imposes the threatening action after the second of the causal link;
4. ***White knight***, insert a new operator or use one already in the plan between to S1 and S3 such that it establishes the precondition of S3 threatened by S1.
5. ***Separation***, insert *non codesignation constraints* between the variables of the negative effect and the threatened precondition so to avoid unification.
   This is useful when variables have not yet been instantiated.

Ex. Given the causal link $pickups\ (X) \xrightarrow{holding\ (X)} stack\ (X, b)$

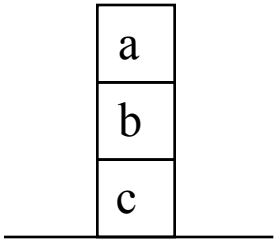
any threat imposed by stack(Y,c) can be solved by imposing X ≠ Y

Example: Sussmann Anomaly

Initial state ([*Yes*]):
clear (b), clear (c), on (c, a), ontable (a), on (ab), on (b, c).
ontable (b), handempty.



Goal ([*g*]):



actions

pickup (X)
PRECOND: *ontable (X), clear (X), handempty*
POSTCOND: *holding (X) not ontable (X), not clear (X), not handempty*

putdown (X)
PRECOND: *holding (X)*
POSTCOND: *ontable (X), clear (X), handempty*

stack (X, Y)
PRECOND: *holding (X), clear (Y)*
POSTCOND: *not holding (X), not clear (Y), handempty, on (X, Y), clear (X)*

unstack (X, Y)
PRECOND: *handempty, on (X, Y), clear (X)*
POSTCOND: *holding (X), clear (Y) ,. not handempty, not on (X, Y), not clear (X),*

Example: Sussmann Anomaly

initial plan: start stop

Orderings: [start <stop]
List of Causal links: [].
Goal agenda:[on(a, b), on(b, c)]

Establishment: chose two actions (without orders) that meet the goals
stack (a, b)
PRECOND: *holding (a), clear (b)*
POSTCOND: *handempty, on (a, b), not clear (b), not holding (a)*

stack (b, c)
PRECOND: *holding (b), clear (c)*
POSTCOND: *handempty, on (b, c), not clear (c), not holding (b)*

Orderings: [start <stop, start < stack (a, b), start <stack (b, c), stack (b, c) <stop]
List of Causal links: [<stack (a, b), stop, on (a, b)>, <stack (b, c), stop, on (b, c)>].
Goal agenda: [holding (a), clear (b), holding (b), clear (c)]

The actions are not ordered. At the moment we only know that both of them will occur.

Example: Sussmann Anomaly

Some of the preconditions on the agenda (*clear (b) and clear (c))* are already met in the initial state: just add the causal link.

Orderings: [see. above]
List of causal link: [..., <start, stack (a, b), clear (b)>, <start, stack (b, c), clear (c)>].
Goal Agenda: [holding (a), holding (b)]

Now we proceed with the establishment of: holding(a), holding (b)

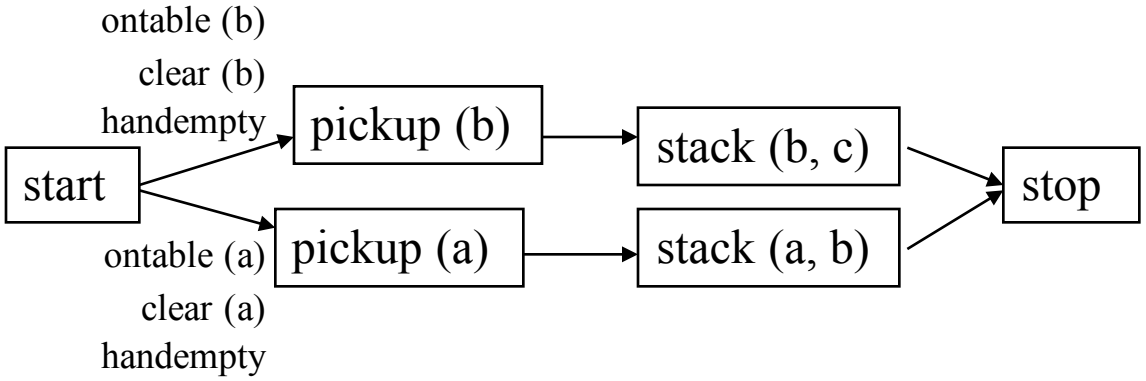
pickup (a)
PRECOND: *ontable (a), clear (a), handempty*
POSTCOND: *not ontable (a), not clear (a), holding (a), not handempty,*

pickup (b)
PRECOND: *ontable (b), clear (b), handempty*
POSTCOND: *not ontable (b), not clear (b), holding (b), not handempty*

List of orders : [..., pickup (a) < stack (a, b), pickup (b) < stack (b, c)]
Causal links: [...,<pickup(a), stack(a,b), holding(a)>, <pickup(b), stack(b,c), holding (b)>]
Goal Agenda: [ontable (a), clear (a), ontable(b), clear(b), handempty]

Example: Sussmann Anomaly

Partial current plan:



ontable(b), ontable(a) clear(b) are met in the initial state.

Orderings: [see. above]

Causal links: [..., <start, pickup(a), ontable (a)>, <start, pickup (b), ontable (b)>,
<start, pickup (b), clear (b)>]

Goal Agenda: *[handempty, clear(a)]*

stack(a, b) threats <*start, pickup(b), clear (b)*> as *not clear(b) is one of its effects* and nothing prevents it to precede *pickup (b)* and invalidate its precondition *clear (b)*.

impose (**promotion**) *pickup (b)* < *stack (a, b)*

Example: Sussmann Anomaly

handempty is true in the initial state.

The new causal link <*start, pickup (b), handempty*> is threatened by *pickup(a)*:

↓

Impose (promotion) *pickup (b) < pickup(a)*

The precondition *handempty* of *pickup(a)* cannot be satisfied from the start because *pickup(b) preceding pickup(a) would negate it*.

↓

we apply the **white knight** using the plan action *stack(b,c)* that reinforces *handempty*:

pickup(b) < stack (b, c) <pickup(a)

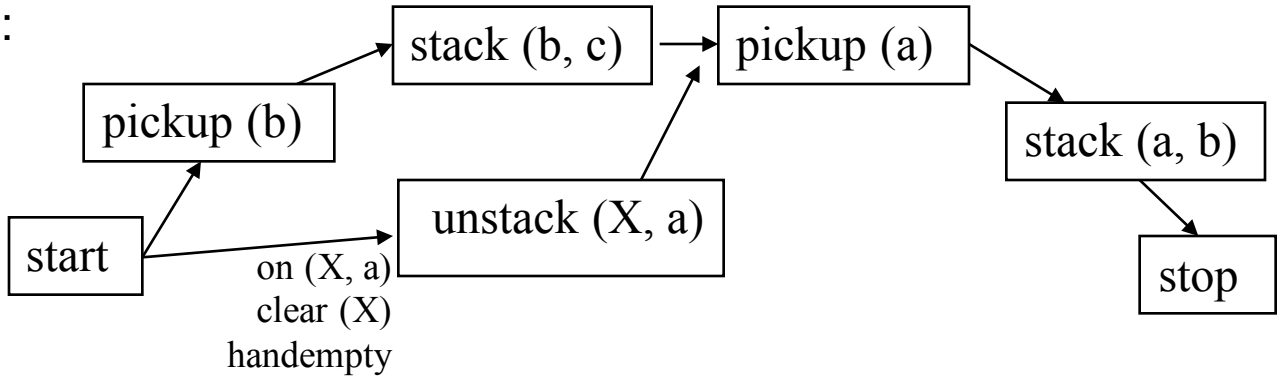
Orderings: [..., pickup(b)< stack (a,b), pickup(b) < pickup(a), stack (b,c) <pickup (a)]
Causal Links: [..., <start, pickup(b), handempty>, <stack (b,c), pickup(a), handempty>].
Goal Agenda: [clear(a)]

Example: Sussmann Anomaly

To satisfy *clear (a)* we can add the action:

unstack (X, a)
PRECOND: *handempty, on (X, a), clear (X)*
POSTCOND: *handempty not, clear (a), holding (X) not on (X, a)*

current partial plan:



Orderings: [..., start < unstack (X, a), unstack (X, a) < stop, unstack (X, a) <pickup (a)]
the causal link list: [..., <unstack(X, a), pickup(a), clear (a)>
Agenda of goals: [On (X, a), clear (X), handempty]

The three preconditions are met on the agenda in the initial state by imposing *X* = *c* in the move *unstack (X, a).*

Example: Sussmann Anomaly

unstack (c, a) is a threat to the causal link <*start, pickup(b), handempty*>

Add a new move **white knight**

putdown (c)
PRECOND: *holding (c)*
POSTCOND: *ontable (c), clear (c), handempty, not holding (c)*

ordering as well:
unstack (c, a) <putdown (c) <pickup (b)

We then used two types of white knights, one using an action which is already in the plan, and one introducing a new action.

Now all the preconditions are met so we can linearize the plan (add the ordering constraints) and possibly instantiating not yet instantiated variables, to get a concrete plan.

1) unstack (c, a) 2) putdown (c) 3) pickup (b)
4) stack (b, c) 5) pickups (a) 6) stack (a, b)

CLOSING REMARKS

- It is always preferable to apply the promotion and demotion before white knight (in particular with the addition of new actions).
- Unfortunately non-linear planners can generate very inefficient plans, even if correct.
- Planning is semi-decidable: if there is a plan that solves a problem the planner finds it, but if there is not, the planner can work indefinitely.
- In domains of increasing complexity it is hard to use a correct and complete planner, which is efficient and domain independent. Often we use ad-hoc methods.

Planning in practice

- Many applications in complex domains:
  - Planners for the Internet (search for information,
  - Automatic creation of Unix commands

  http://www.cs.washington.edu/research/projects/softbots/www/softbots.html
  - Management of space missions

  http://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/
  - Robotics

  http://www.robocup.org/
  - Graphplan developed by Carnegie Mellon University

  http://www.cs.cmu.edu/~avrim/graphplan.html
  - Industrial production plans
  - Logistics

- Classical algorithms have efficiency problems in case of complex domains

- There are techniques that make the algorithms more efficient

↓

HIERARCHICAL PLANNING

HIERARCHICAL PLANNING

Hierarchical planners are search algorithms that manage the creation of complex plans at **different levels of abstraction**, by considering the simplest details only after finding a solution for the most difficult ones.

- We need a language that enables operators at different levels of abstraction:
  - *Values of criticality assigned to the preconditions*
  - *Atomic operators vs. Marco operators*

All operators are again defined by PRECONDITIONS and EFFECTS.

- Popular hierarchical planning algorithms:
  - *STRIPS-Like*
  - *Partial-Order*

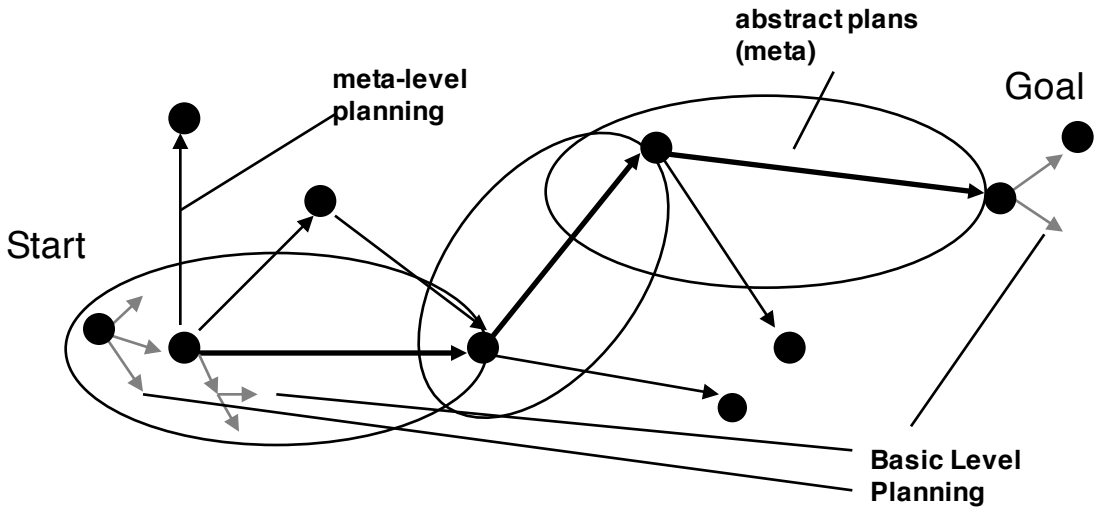
Given a goal, the hierarchical planner performs a ***meta-level search*** to generate a ***meta-level plan*** which leads from a state that is very close to the initial one to a state which is very close to the goal.

HIERARCHICAL PLANNING

The plan is then completed with a lower level search, taking account of details omitted at the previous level.

So a hierarchical algorithm must be able to:

- organize high (*meta-level*)
- expand abstract plans into concrete plans
  - planning abstract parts in terms of more specific actions (planning *basic level*)
  - expanding already prebuilt plans



ABSTRIPS

Hierarchical planner who enhances the Strips definition of actions with a **criticality value** (proportional to the complexity of its achievement) to each precondition.

The planning algorithm proceeds at different levels of abstraction spaces. At each level, lower level preconditions are ignored.

ABSTRIPS fully explores the space of a certain level of abstraction before moving on to a more detailed level: **length search**

At every level of abstraction, it generates a complete plan

Application examples:

- construction of a building,
- organization of a trip,
- draft a top-down program.

ABSTRIPS: Methodology of solution

1. A threshold value is fixed.
2. All the preconditions whose criticality value is less than the threshold value are considered true.
3. STRIPS(*) finds a plan that meets all the preconditions whise value is greater or equal to the threshold value.
4. It then uses the full plan pattern obtained as a guide and lowers the value of the threshold.
5. It extends the plan with operators that meet the preconditions whose level of criticality is higher than or equal to the new threshold value.
6. It lowers the threshold value until all the preconditions of the original rules have been considered.

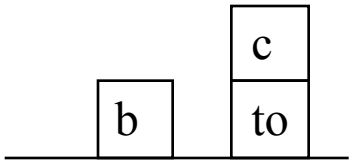
It is important to give good values critical preconditions !!!

(*) At every level we can use a different planner,
not necessarily STRIPS.

Example (Algorithm Strips-Like)

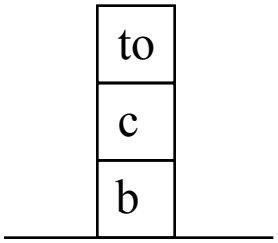
- Initial state:

clear (b)
clear (c)
on (c, a)
handempty
ontable (a)
ontable (b)



- Goal:

on (c, b) ∧ on (a, c)



We specify a hierarchy of goals and preconditions assigning criticality values (that reflect the degree of difficulty in satisfying them):

on **(3)**
ontable, clear, holding **(2)**
handempty **(1)**

Example (Algorithm Strips-Like)

We use the STRIPS rules:

pickup (X)
PRECOND: *ontable (X), clear (X), handempty*
DELETE: *ontable (X), clear (X), handempty*
ADD: *holding (X)*

putdown (X)
PRECOND: *holding (X)*
DELETE: *holding (X)*
ADD: *ontable (X), clear (X), handempty*

stack (X, Y)
PRECOND: *holding (X), clear (Y)*
DELETE: *holding (X), clear (Y)*
ADD: *handempty, on (X, Y), clear (X)*

unstack (X, Y)
PRECOND: *handempty, on (X, Y), clear (X)*
DELETE: *handempty, on (X, Y), clear (X)*
ADD: *holding (X), clear (Y)*

Example (Algorithm Strips-Like)

- **First level** of abstraction: we consider only the preconditions with criticality value 3 and you get the following goal stack:

on (c, b)
on (a, c)
on (c, b) ∧ on (a, c)

- The action *stack (c, b)* is added to the plan to meet *on (c, b)*.
- Since its preconditions are all less critical than 3 they are considered to be met.

A new state is generated by simulating the execution of action *stack (c, b)*.

state description	**goal stack**
clear (b) clear (c) ontable (a) ontable (b) **on (c, b)** handempty on (c, a)	on (a, c) on (c, b) ∧ on (a, c)

Note: in the description of the state only effects with criticality greater than or equal to 3 are added/deleted. There may be inconsistencies, but this is fine!!

Example (Algorithm Strips-Like)

- The action *stack (a, c)* is added following the same process for *stack (c, b)*. The complete plan at Level 1 is:

  1. ***stack (a, c)***
  2. ***stack (c, b)***

- **Second level** of abstraction: we restart from initial state and in the goal stack we insert the initial goals, the actions computed at level 1 and their preconditions (along with a goal ordering):

holding (c)
clear (b)
holding (c) ∧ clear (b)
stack (c, b)
holding (a)
clear (c)
holding (a) ∧ clear (c)
stack (a, c)
on (c, b) ∧ on (a, c)

Example (Algorithm Strips-Like)

- The condition *holding(c)* is satisfiable with the action *unstack(c, X)* and the stack becomes

clear (c)
on (c, X)
clear (c) and on (c, X)
unstack (c, X)
clear (b)
holding (c) ∧ clear (b)
stack (c, b)
holding (a)
clear (c)
holding (a) ∧ clear (c)
stack (a, c)
on (c, b) ∧ on (a, c)

- the preconditions of *unstack (c, X)* to be considered at level 2 (*clear(c)* and *on(c, X)*) are all met in the initial state with the substitution *X / a*.

Example (Algorithm Strips-Like)

Executing the action *unstack (c, a)* on the initial state results in:

state description	goal stack
clear (b) clear (a) ontable (a) ontable (b) handempty holding (c)	clear (b) holding (c) ∧ clear (b) **stack (c, b)** holding (a) clear (c) holding (a) ∧ clear (c) **stack (a, c)** on (c, b) ∧ on (a, c)

And so on until you get the full plan of level 2:

1. ***unstack (c, a)***
2. ***stack (c, b)***
3. ***pickup (a)***
4. ***stack (a, c)***

Example (Algorithm Strips-Like)

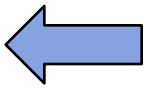
- **Third level** of abstraction: we have the following goal stack

handempty ∧ clear (c) ∧ on (c, a)
unstack (c, a)
holding (c) ∧ clear (b)
stack (c, b)
handempty ∧ clear (a) ∧ ontable (a)
pickup (a)
holding (a) ∧ clear (c)
stack (a, c)
on (c, b) ∧ on (a, c)

Only the precondition *handempty* still needs to be considered.
The search for a solution to level 3 in this case is simply a check: the solution at level 2 is also correct for level 3 and the search stops.

Note: A level 1 the valid plan
1. ***stack (a, c)***
2. ***stack (c, b)***

would fail causing backtracking.



Homework: check it!!

HIERARCHICAL PLANNING
MACRO-OPERATORS

Two types of operators:

- ***Atomic*** operators
- ***Macro*** operators

- **Atomic** operators represent elementary actions typically defined as STRIPS rules.
  - Atomic operators can be directly executed by an agent

- **Macro** operators in turn represent a set of elementary actions: they are decomposable into atomic operators
  - Macro operators before execution should be further decomposed
  - Their decomposition can be *precompiled* or from *plan*.

HIERARCHICAL PLANNING
MACRO-OPERATORS

Precompiled decomposition: the description of the macro operator also contains the DECOMPOSITION - the sequence of basic operators to be executed at run-time.

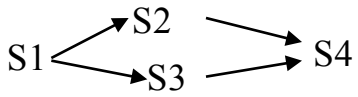
ACTION: ***Cook (X)***

PRECOND: *have (X), have (pot), have (salt) in (water, pot)*

EFFECT: *cooked*(X)

DECOMPOSITION: *S1: boil (water), S2: put (salt, water), S3: put (X, pot), S4: boilinWater (X)*

ORDER: S1 <S2, S1 < S3, S2 < S4, S3 < S4



Planned decomposition: the planner must perform a low-level search for synthesising the atomic action plan that implements the macro action.

HIERARCHICAL PLANNING
MACRO-OPERATORS

The planning algorithm can be either linear or non-linear

A hierarchical non-linear algorithm is similar to POP where at each step one can choose between:

- Reach an open goal with an operator (including macro operators)
- expand a macro step of the plan (the decomposition method can be precompiled or schedule).

function **HD_POP (initialGoal, methods, operators) returns plan**

```
plan: = INITIAL_PLAN (start, stop, initialGoal)
loop
  if SOLUTION (plan) then return plan;
  choose between
    • SN, C: = SELECT_SUBGOAL (plan);
      CHOOSE_OPERATOR (plan, operators, SN, C);
    • SnonPrim: = SELECT_MACRO_STEP (plan)
      CHOOSE_DECOMPOSITION (SnonPrim, methods, plan)
  SOLVE_THREATS (plan)
end
```

Example (POP-like Algorithm)

Initial state Goal

have (pot), have (pan), have (oil), have (onion),made (pasta, tomato)

have (pasta), have (tomato), have (salt), have (water)

ATOMIC ACTIONS

ACTION ***makePasta (X)***

PRECOND: *have (pasta), cooked(pasta), Sauce (X)*

EFFECT: *made(pasta, X)*

ACTION ***boil (X)***

PRECOND: *have (X), have (pot), in (X, pot), plain (X)*

EFFECT: *boiled (X)*

ACTION ***boilinWater (X)***

PRECOND: *have (X), have (pot), in (water, pot), boiled (water), in (salt, water), in (X, water)*

EFFECT: *cooked (X)*

ACTION ***cookSauce (X)***

PRECOND: *have (X), have (pan), in (onion, pan), fried (onion), in (X, oil)*

EFFECT: *sauce (X)*

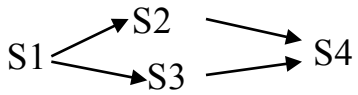
Example (POP-like Algorithm)

ACTION ***fry (X)***
PRECOND: *have (X), have (pan), have (oil), in (oil pan), in (X, pan), plain (oil)*
EFFECT: *fried (X)*

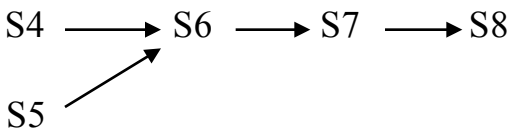
ACTION: ***put (X, Y)***
PRECOND: *have (X), have (Y)*
EFFECT: *in (X, Y),¬ plain (Y)*

ACTIONS MACRO

ACTION: ***Cook (X)***
PRECOND: *have (X), have (pot), have (salt) in (water, pot)*
EFFECT: *cooked*(X)
DECOMPOSITION: *S1: boil (water), S2: put (salt, water), S3: put (X, pot), S4: boilinWater (X)*
ORDER: S1 < S2, S1 < S3, S2 < S4, S3 < S4

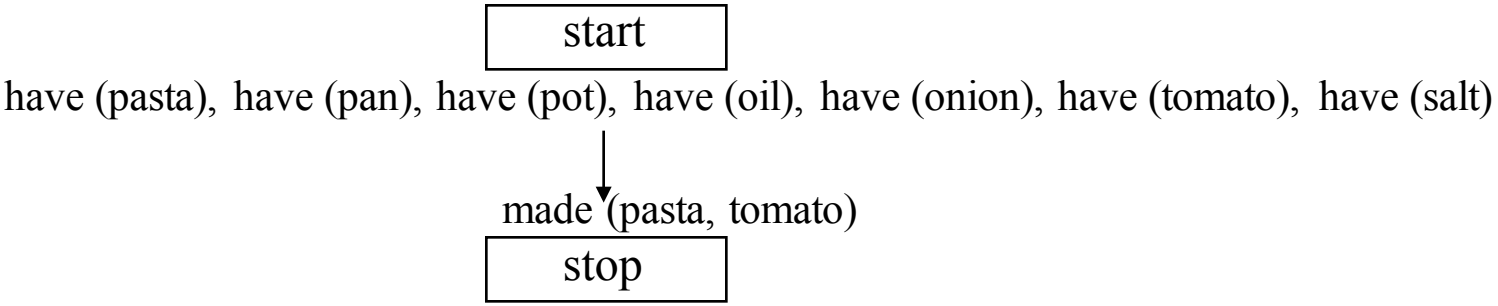


ACTION: ***MakeSauce (X)***
PRECOND: *have (X), have (oil), have (onion), have (pan)*
EFFECT: *sauce (X)*
DECOM: *S4: put (oil pan), S5: put (onion, pan), S6: fry (onion), S7: put (X, oil), S8: cookSauce (X)*
ORDER: S4 < S6, S5 < S6, S6 < S7, S7 < S8



Example (POP-like Algorithm)

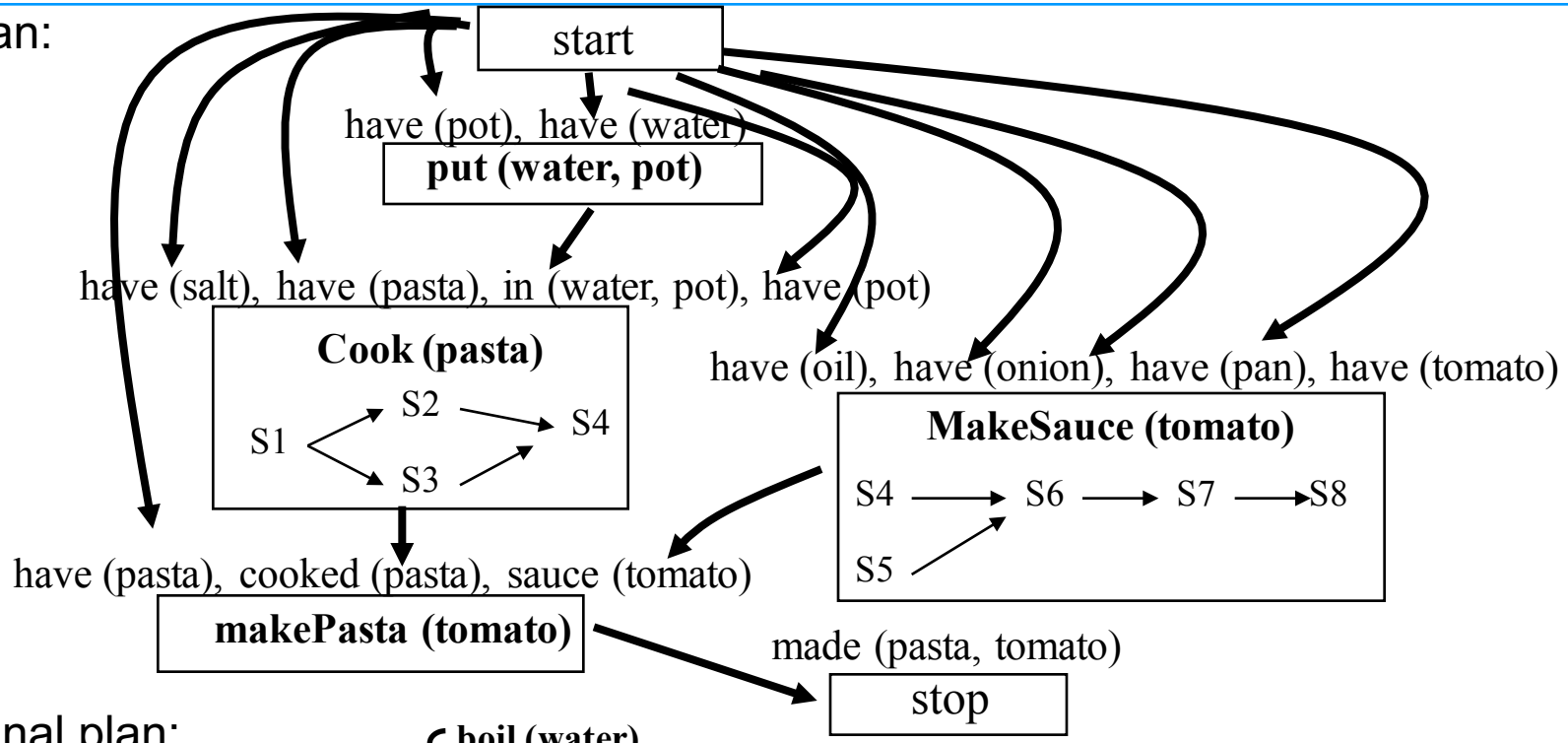
- Initial empty plan



- At every step you can choose between
  - Reach an open goal with an operator (including macro operators)
  - Expand a macro step of the plan (the decomposition method can be precompiled or schedule).

Example (POP-like Algorithm)

Complete plan:



A possible final plan:

put (water, pot)
Cook (pasta)
MakeSauce (tomato)
makePasta (tomato)

- **boil (water)**
- **put (salt, pot)**
- **put (pasta, pot)**
- **boilinWater (pasta)**
- **put (oil pan)**
- **put (onion, pan)**
- **fry (onion)**
- **put (tomato, pan)**
- **cookSauce (tomato)**

DECOMPOSITION

To ensure the decomposition is safe, some properties must be guaranteed:

Constraints on the decomposition

- If the A macro action has the effect X and is expanded with the plan P
  - X must be the effect of at least one of the actions in which A is decomposed and it should be protected until the end of the plan P
  - each precondition of the actions in P must be guaranteed by the previous actions in P or it must be a precondition of A
  - the P action must not threat any causal link when P is substituted for A in the plan

Only under these conditions you can replace the macro action A with the plan P

DECOMPOSITION

Replacing A with P

- When replacing A with P, the orderings and causal links should be added

  – **Orderings**

    - for each B such that B <A then B <first(P) is imposed (first action of P)
    - for each B such that A <B then last (P) <B is imposed (last action of P)

  – **Causal links**

    - if <S, A, C> is a causal link in the initial plan, then it must be replaced by a set of causal links <S, Si, C> where Si are the actions of P that have C as a precondition and no other step of A before it has a C as a precondition
    - if <A, S, C> is a causal link in the initial plan, then it must be replaced by a set of causal links <Si, S, C> where Si are the actions that have C as an effect and no other step of P after it has the effect C

EXECUTION

Generative planners build plans that are then executed by an **executing agent**.

Possible problems encountered during execution:

- An action should be executed and its preconditions are not satisfied
  - incomplete or incorrect knowledge
  - unexpected conditions
  - the world changes indipendently from the planner

- Action effects are not the one expected
  - Errors of the executing agent
  - Non deterministic/unpredictable effects

While executing, the agent should *perceive* the changes in the world and *Act* accordingly

EXECUTION

- Some planners run under the hypothesis of **Open World Assumption** as opposed to the *Closed World Assumption*: they consider that the information that is not explicitly stated in a state is not false, but **unknown**
- The unknown information can be retrieved via **sensing actions** added to the plan.
- ***Sensing actions*** are modeled as causal actions.
- The preconditions are the conditions that must be true to perform a certain observation, while postconditions are the result of the observation.
- Two possible approaches:
  - Conditional Planning
  - Integration between Planning and Execution

CONDITIONAL PLANNING

A **conditional planner** is a search algorithm that generates various alternative plans for each source of uncertainty of the plan.

A **conditional plan** it is therefore constituted by:

- causal actions
- Sensing actions for retrieving unknown information
- Several alternative partial plans of which only one will be executed depending on the results of the observations.

Example: Conditional Planning

actions:

remove (X, Y)
PRECOND: *on (X, Y),* ***¬ intact (X)***
EFFECT: *¬ on (X, Y), off (X), clearHub (Y)*

Puton (X, Y)
PRECOND: *off (X), clearHub (Y)*
EFFECT: *on (X, Y), ¬ off (X), ¬ clearHub (Y)*

inflate (X)
PRECOND: ***intact (X)****, Flat (X)*
EFFECT: *inflated (X), ¬ flat (X)*

Sensing action*:*

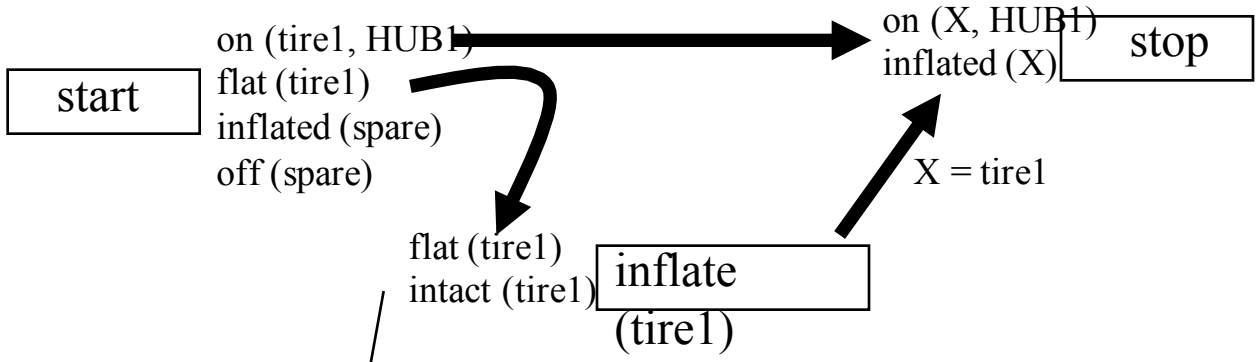
checkTire (X)
PRECOND: tire (X)
EFFECT: knowsWhether (intact (X))

initial state*: On (tire1, HUB1), flat (tire1), inflated (spare), off (spare)*

Goal*: On (X, HUB1), inflated (X)*

Example: Conditional Planning

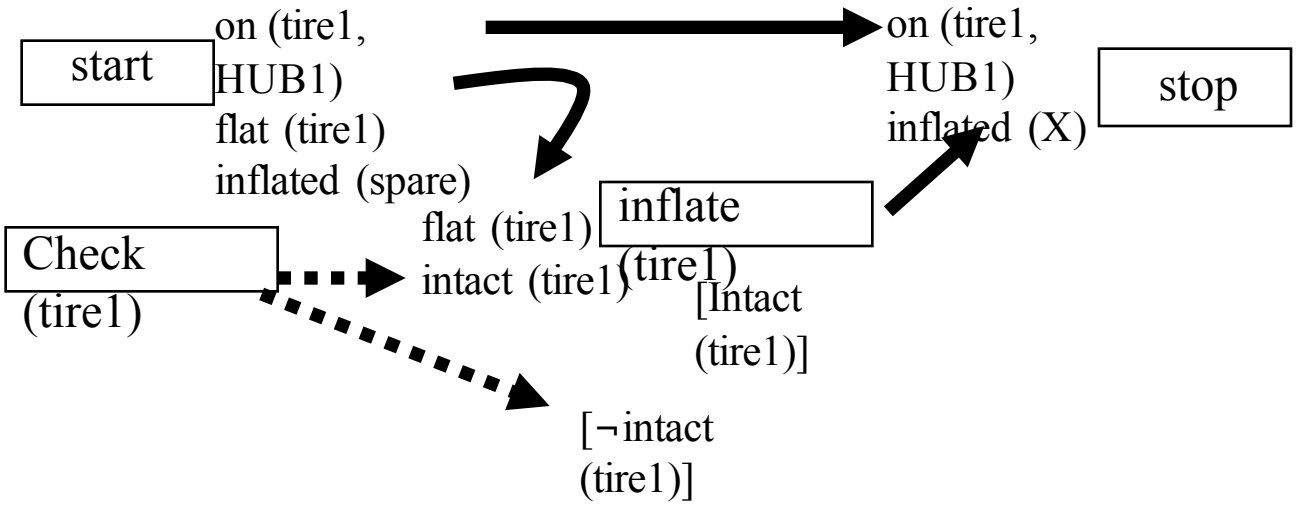
A traditional planner (without sensing actions) would produce the following plan:



Upon backtracking the plan fails again as we get *X* = *spare*: *remove (tire1, HUB1)* - *Puton (spare, HUB1)* as the precondition ¬ *intact (tire1)* cannot be achieved in any way.

Example: Conditional Planning

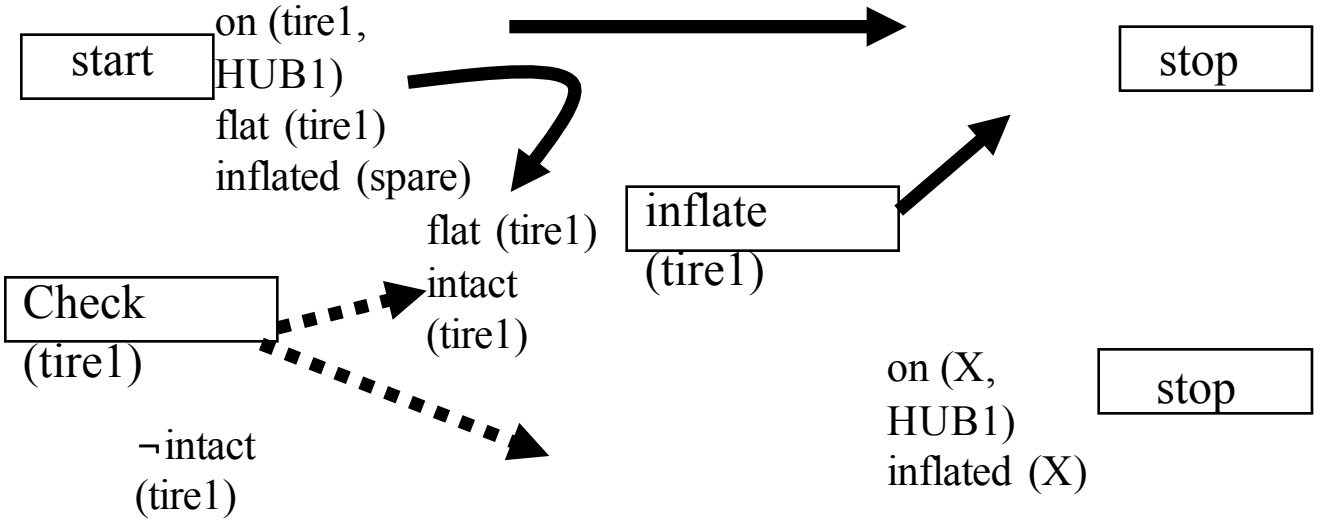
Using the sensing action one can build a **conditional plan**:



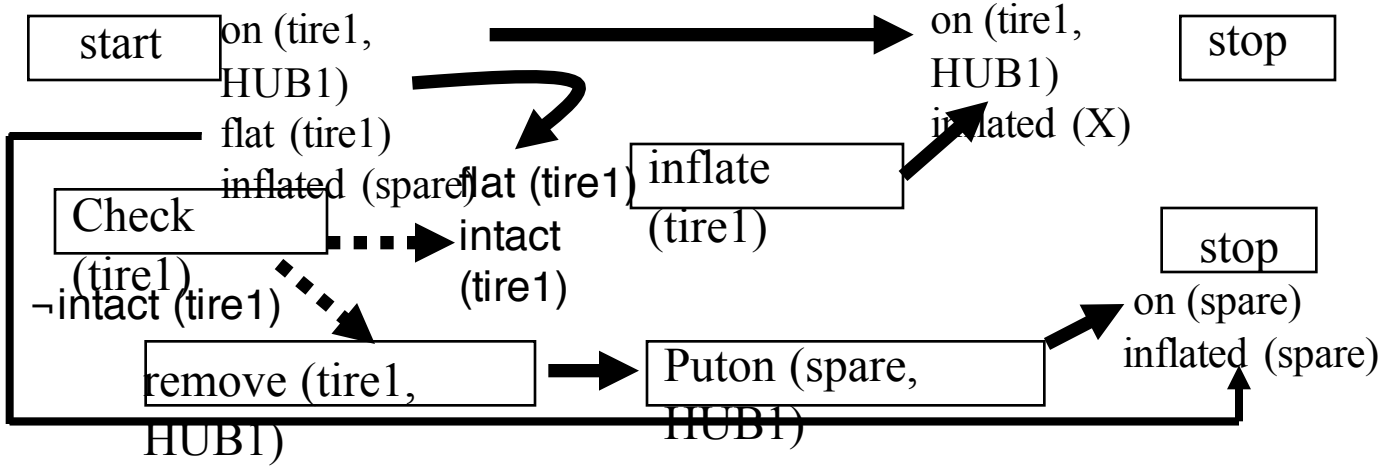
- *inflate(tire1)* is the correct plan in only one executing scenario: a **context** in which *intact (tire1)* is true in the initial state

- We should generate a copy of the goal for every other executing scenario and generate a corresponding plan for each of them.
- We have an exponential number of plans

Example: Conditional Planning



It constructs the plan **conditional**



Conditional Planning: limitations

- Combinatorial **explosion of the search tree** with high numbers of alternative contexts.
- A comprehensive plan which takes account every possible contingency might require a lot of **memory**.
- Not always all alternative contexts are known in advance
- Often conditional planners are associated with **probabilistic planners** that plan only for the most probable contexts.

Contingency planners

- Cassandra deterministic contingency planner
- Buridan builds plans that have a probability greater than a certain threshold
- **C.BURIDAN**
- **Algorithm https: //www.aaai.org/Papers/AIPS/1994/AIPS94-006.pdf**
- **Action representation Draper et al. 1994a**A probabilistic model of action for least-commitment planning with information gathering. In Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence, pp. 178-186 Seattle, WA Morgan Kauffman.

- Systems interleaving planning and execution
- IPEM
- SAGE
- XII

REACTIVE PLANNING (Brooks - 1986)

We have described a deliberative planning process where the plan is built before execution

Reactive planners are on-line algorithms, capable of interacting with the world to deal with the dynamicity and the non-determinism of the environment:

- They observe the world in the planning stage
- They acquire unknown information
- They monitor the implementation of actions and check the effects
- They interleave planning and execution

Pure reactive systems do not plan, they only react as triggers to world variations.

PURE REACTIVE SYSTEMS

They have access to a knowledge base that describes what actions must be carried out and under what circumstances. Choose one action at a time, without any lookahead activicy.

- A thermostat uses the simple rules:

1) If the temperature T of the room is K degrees above the threshold T0, turn the air conditioner on;
2) If the room temperature is below T0 K degrees, turn the air conditioner off.

Advantages:

- They are able to interact with the real system. They are robust in domains for which it is difficult to provide complete and accurate models.
- They do not use models, but perceive world changes. That's why they are also extremely fast in responding.

Downside:

Their performance in predictable domains that require reasoning and deliberate is quite low (eg. Chess) as they are not able to generate plans.

HYBRID SYSTEMS

Modern responsive planners are **hybrids** integrate a **generative approach** and a **reactive approach** in order to exploit the computational capacity of the first and the ability to interact with the system of the second thus addressing the problem of the execution.

A **hybrid planner:**

- generates a plan to achieve the goal
- checks the preconditions of the action that is about to run, and the effects of the action that just executed
- disassembles the effects of action (importance of action reversibility) and reschedules in case of failures
- corrects the plans if unforeseen external events occur.