B.S. Abdur Rahman ™ Crescent Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956

CSD 3102- ARTIFICIAL INTELLIGENCE TECHNIQUES

MODULE 4
PLANNING GRAPHS

TOPIC: **Planning-Planning Graphs-Hierarchical Planning - Non- linear Planning - Conditional Planning-Reactive Planning - Knowledge based Planning-Using Temporal Logic – Execution Monitoring and Re-planning- Continuous Planning- Multi-agent Planning-Job shop Scheduling Problem**

MODULE IV PLANNING 9

Planning Problem – Simple Planning agent –Blocks world - Goal Stack Planning- Means Ends Analysis- Planning as a Statespace Search - Partial Order **Planning-Planning Graphs-Hierarchical Planning - Non- linear Planning - Conditional Planning-Reactive Planning - Knowledge based Planning-Using Temporal Logic – Execution Monitoring and Re-planning- Continuous Planning-Multi-agent Planning-Job shop Scheduling Problem**.

Graph Planning

- All of the heuristics we have suggested for total-order and partial-order planning can suffer from inaccuracies.
- P**lanning graph** can be used to give better heuristic estimates.
- Alternatively, we can extract a solution directly from the planning graph, using a specialized algorithm such as the one called GRAPHPLAN.

- A planning graph consists of a sequence of **levels** that correspond to time steps in the plan, where level 0 is the initial state.
- Each level contains a set of literals and a set of actions.
- Roughly speaking, the literals are all those that *could* be true at that time step, depending on the actions executed at preceding time steps.

- We start with state level *So,* which represents the problem's initial state.
- We follow that with action level A_o, in which we place all the actions whose preconditions are satisfied in the previous level.
- Each action is connected to its preconditions in S_o and its effects in S_1, in this case introducing new literals into ,S_1 that were not in *So.*

Persistence Actions

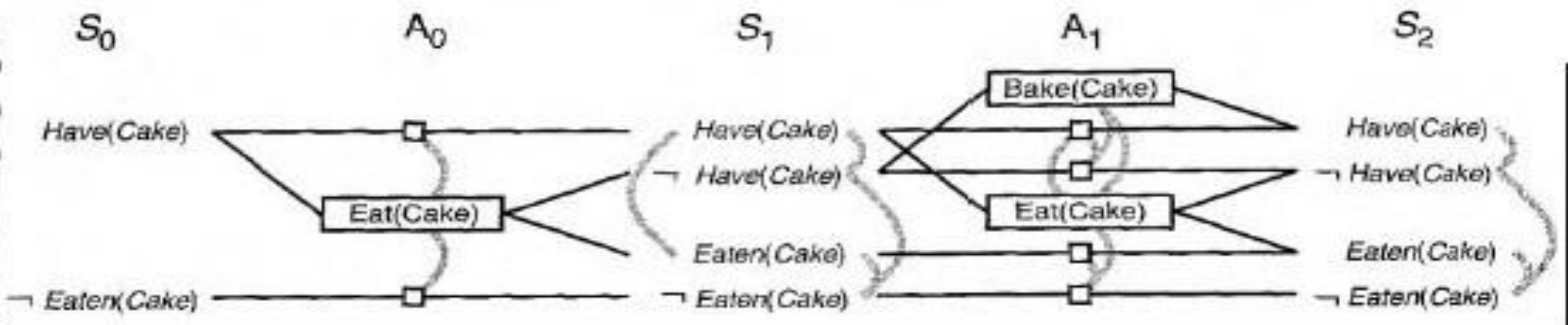
- The planning graph needs a way to represent inaction as well as action.
- That is, it needs the equivalent of the frame axioms in situation calculus that allow a literal to remain true from one situation to the next if no action alters it.
- In a planning graph this is done with a set of **persistence actions.**
- For every positive and negative literal C, we add to the problem a persistence action with precondition C and effect C.

Mutual Exclusion

- The gray lines in the above figure indicate **mutual exclusion** (or **mutex)** links. For example,
- *Eat(Cake)*MUTEX is mutually exclusive with the persistence of either *Have* (*Cake)* or ¬ *Eaten* (*Cake).*
- For example, *Have* (*Cake)* and *Eaten* (*Cake)*are **mutex**: depending on the choice of actions in *Ao,* one or the other, but not both, could be the result. In other words, S_l represents multiple states, just as regression state-space search does, and the mutex links are constraints that define the set of possible states. We continue in this way, alternating between state level *St* and action level A, until we reach a level where two consecutive levels are identical.

Init(Have(Cake))
Goal(Have(Cake) ∧ Eaten(Cake))
Action(Eat(Cake)
  PRECOND: *Have(Cake)*
  EFFECT: ¬ *Have(Cake) ∧ Eaten(Cake))*
Action(Bake(Cake)
  PRECOND: ¬ *Have(Cake)*
  EFFECT: *Have(Cake)*

The "have cake and eat cake too" problem.



The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

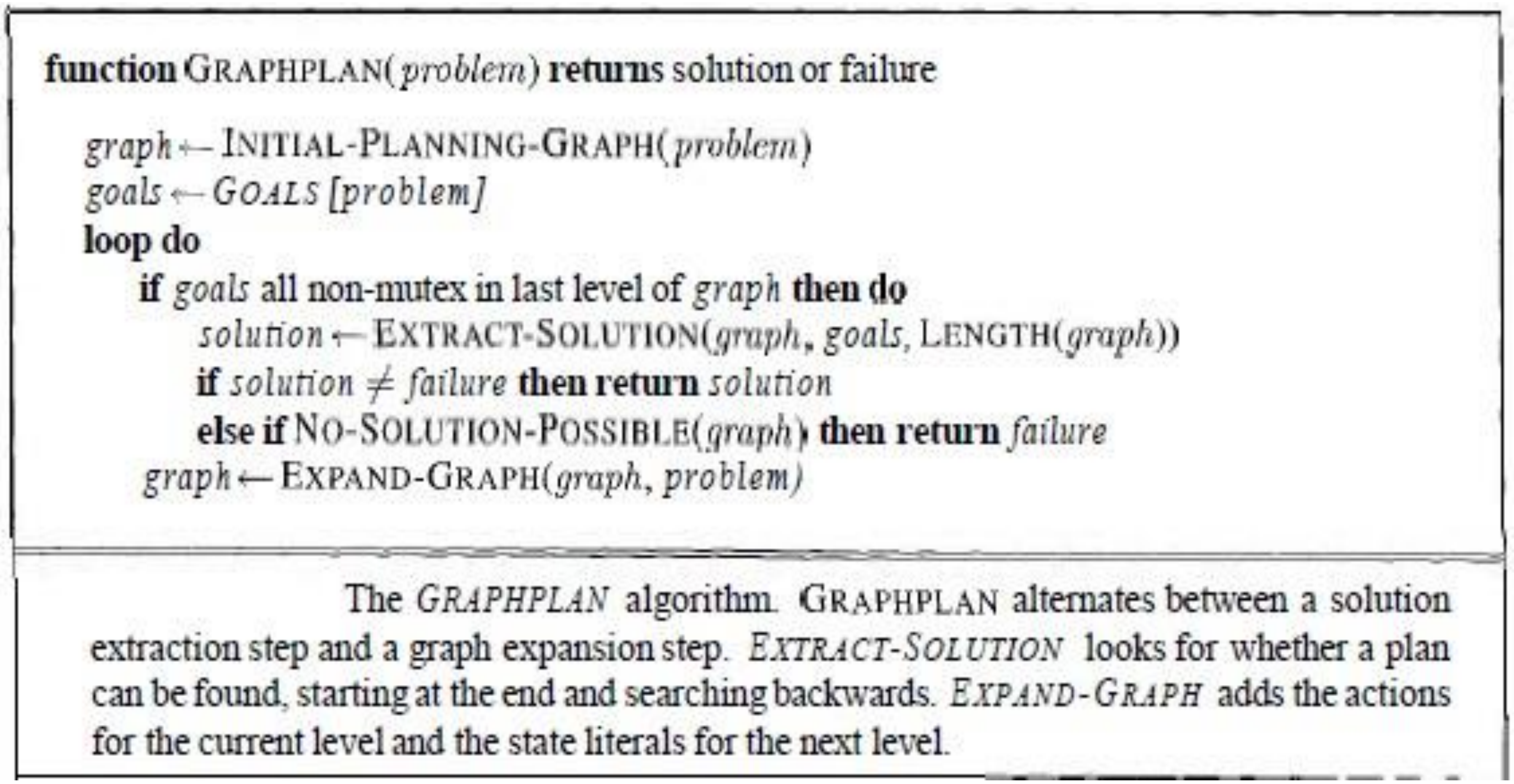
Leveled Off

- At this point, we say that the graph has **leveled off.**
- Every subsequent level will be identical, so further expansion is unnecessary.
- What we end up with is a structure where every A, level contains all the actions that are applicable in S, along with constraints saying which pairs of actions cannot both be executed.

Conditions for Mutex relation

- A mutex relation holds between two *actions* at a given level if any of the following three conditions
- holds:
  - ***Inconsistent effects****:* one action negates an effect of the other. For example *Eat(Cake)* and the persistence of *Have(Cake)* have inconsistent effects because they disagree on the effect *Have* (*Cake).*
  - ***Interference****:* one of the effects of one action is the negation of a precondition of the other. For example *Eat(Cake)* interferes with the persistence of *Have(Cake)* by negating its precondition.
  - ***Competing needs****:* one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, *Bake(Cake)* and *Eat (Cake)* are mutex because they compete on the value of the *Have(Cake)* precondition.

The GRAPHPLAN algorithm



```
function GRAPHPLAN(problem) returns solution or failure

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← GOALS [problem]
  loop do
      if goals all non-mutex in last level of graph then do
          solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
          if solution ≠ failure then return solution
          else if NO-SOLUTION-POSSIBLE(graph) then return failure
      graph ← EXPAND-GRAPH(graph, problem)
```

The *GRAPHPLAN* algorithm. GRAPHPLAN alternates between a solution extraction step and a graph expansion step. *EXTRACT-SOLUTION* looks for whether a plan can be found, starting at the end and searching backwards. *EXPAND-GRAPH* adds the actions for the current level and the state literals for the next level.

The GRAPHPLAN algorithm

•The GRAPHPLAN algorithm has two main steps, which alternate within a loop.

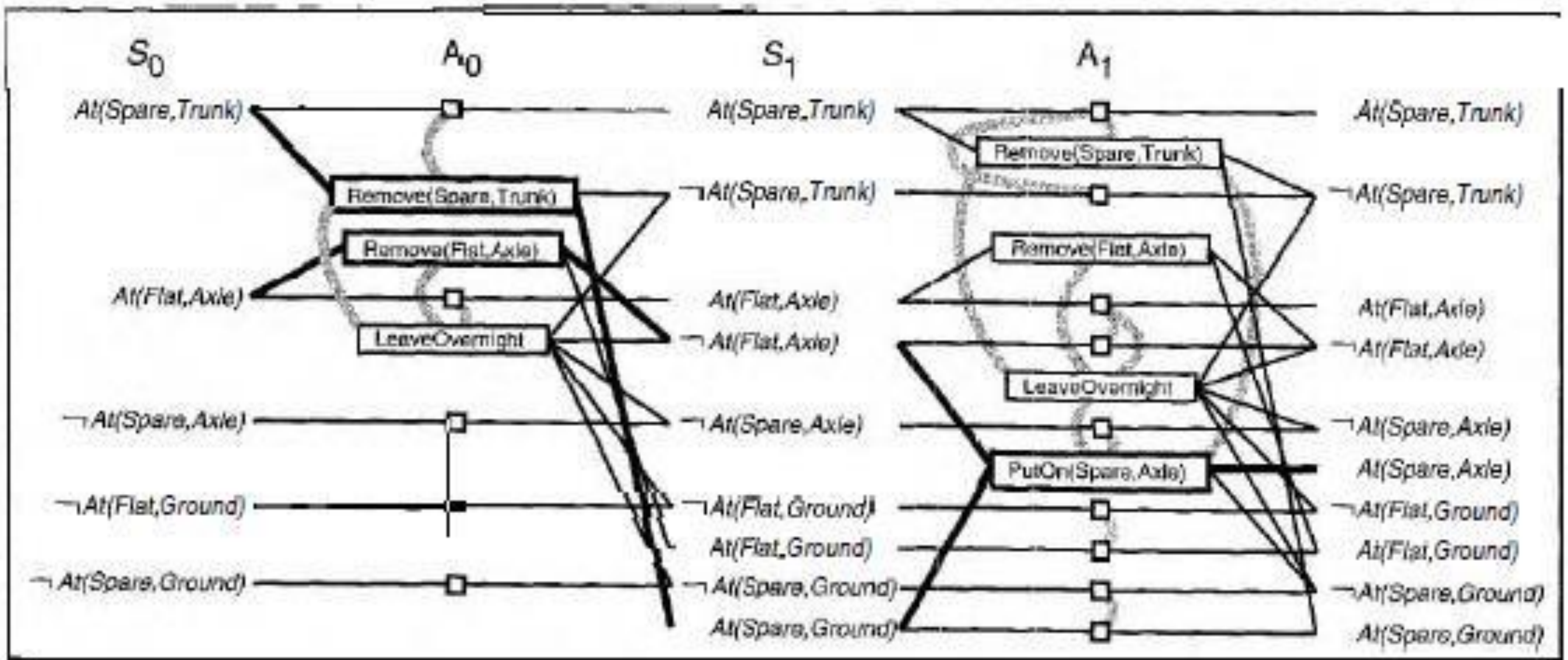
•First, it checks whether all the goal literals are present in the current level with no mutex links between any pair of them.

•If this is the case, then a solution ***might*** exist within the current graph, so the algorithm tries to extract that solution.

•Otherwise, it expands the graph by adding the actions for the current level and the state literals for the next level.

•The process continues until either a solution is found or it is learned that no solution exists.

Example: Spare Tire problem



The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

Graph Planning : Spare Tire problem

- The first line of GRAPHPLAN initializes the planning graph to a one-level (So) graph consisting of the five literals from the initial state.
- The goal literal *At(Spare, Axle)* is not present in So, so we need not call EXTRACT-SOLUTION-we are certain that there is no solution yet.
- Instead, EXPAND-GRAPH adds the three actions whose preconditions exist at level So (i.e., all the actions except *PutOn(Spare, Axle).*

- Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:
  - ***Inconsistent effects:*** *Remove (Spare, Trunk) is mutex with LeaveOvernight because one has the effect At (Spare, Ground) and the other has its negation.*
  - ***Inteference:*** *Remove(Flat, Axle) is mutex with LeaveOvernight because one has the precondition At(Flat, Axle) and the other has its negation as an effect.*
  - ***Competing needs:*** *Put On (Spare, Axle) is mutex with Remove (Flat, Axle) because one has At (Flat, Axle) as a precondition and the other has its negation.*
  - ***Inconsistent support:*** *At(Spare, Axle) is mutex with At(Flat, Axle) in S2 because the only way of achieving At(Spare, Axle) is by PutOn(Spare, Axle), and that is mutex with the persistence action that is the only way of achieving At(Flat, Axle). Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.*

HIERARCHIAL TASK NETWORK PLANNING

- Planning method based on **hierarchical task networks** or HTNs. The approach we take combines ideas from both partial-order and the area known as "HTN planning."

- Plans are refined by applying **action decompositions.** Each action decomposition reduces a high- level action to a partially ordered set of lower-level actions.

- The process continues until only **primitive actions** remain in the plan. Typically, the primitive actions will be actions that the agent can execute automatically

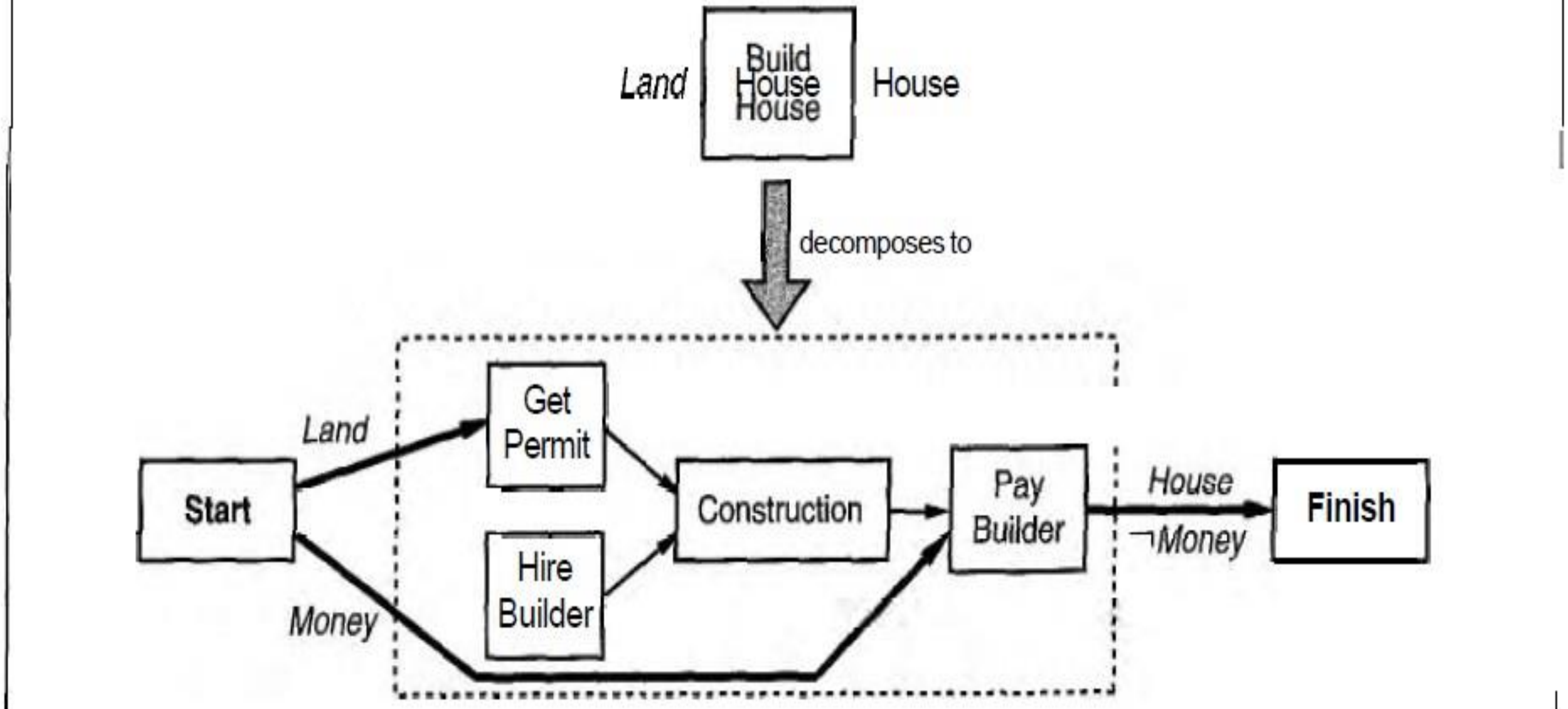


Figure 12.5 One possible decomposition for the *BuildHouse* action.

Action(*BuyLand*, PRECOND:*Money*, EFFECT:*Land* A ¬ *Money*)
Action(*GetLoan*, PRECOND: *GoodCredit*, EFFECT:*Money* A *Mortgage*)
Action(*BuildHouse*, PRECOND:*Land*, EFFECT:*House*)

$Action(BuyLand, \textsc{Precond}{:}Money, \textsc{Effect}{:}Land \land \neg Money)$
$Action(GetLoan, \textsc{Precond}{:}GoodCredit, \textsc{Effect}{:}Money \land Mortgage)$
$Action(BuildHouse, \textsc{Precond}{:}Land, \textsc{Effect}{:}House)$

$Action(GetPermit, \textsc{Precond}{:}Land, \textsc{Effect}{:}Permit)$
$Action(HireBuilder, \textsc{Effect}{:}Contract)$
$Action(Construction, \textsc{Precond}{:}Permit \land Contract,$
$\quad \textsc{Effect}{:}HouseBuilt \land \neg Permit)$
$Action(PayBuilder, \textsc{Precond}{:}Money \land HouseBuilt,$
$\quad \textsc{Effect}{:}\neg Money \land House \land \neg Contract)$

$Decompose(BuildHouse,$
$\quad Plan(\textsc{Steps}{:}\ \{S_1 : GetPermit,\ S_2 : HireBuilder,$
$\qquad\qquad S_3 : Construction,\ S_4 : PayBuilder\}$
$\qquad \textsc{Orderings}{:}\ \{Start \prec S_1 \prec S_3 \prec S_4 \prec Finish,\quad Start \prec S_2 \prec S_3\},$
$\qquad \textsc{Links}{:}\ \{Start \xrightarrow{Land} S_1,\ Start \xrightarrow{Money} S_4,$
$\qquad\qquad S_1 \xrightarrow{Permit} S_3,\ S_2 \xrightarrow{Contract} S_3,\ S_3 \xrightarrow{HouseBuilt} S_4,$
$\qquad\qquad S_4 \xrightarrow{House} Finish,\ S_4 \xrightarrow{\neg Money} Finish\}))$

Figure 12.6 Action descriptions for the house-building problem and a detailed decomposition for the *BuildHouse* action. The descriptions adopt a simplified view of money and an optimistic view of builders.

Non-Linear Planning in AI

- This planning technique involves creating a goal stack and incorporating it into the search space of all potential sub-goal sequences.
- It addresses interactions between goals by utilizing the interleaving method.
- Non-Linear Planning in AI is an essential problem-solving approach that deals with complex situations where the solution is not easily predicted.
- "It involves finding a sequence of actions that transforms the system's current state to the desired goal state while also considering non-linear constraints, such as resource limitations or temporal ordering".
- Non-Linear Planning is commonly used in robotics, scheduling, and logistics to solve challenging problems.

Non Linear Planning - Algorithm

1. Select a goal, denoted as g, from the set of goals
2. If the current state does not match the goal g, then select an operator o whose add-list matches the goal g
3. Push the operator o onto the OpenStack and add the preconditions of o to the set of goals
4. Continue this process while all preconditions of the operator on top of the OpenStack are satisfied in the current state
5. Pop the operator o from the top of the OpenStack, apply it to the current state, and add it to the plan

Advantages of Non-Linear Planning in AI

- **Flexibility:**
  Non-Linear Planning can handle complex problem spaces that involve multiple, interdependent sub-goals with non-linear dependencies.
- **Optimality:**
  Non-Linear Planning algorithms often use heuristic search techniques to find the optimal solution.
- **Resource utilization:**
  Non-Linear Planning considers resource limitations and temporal ordering constraints when generating a plan.
- **Reusability:**
  Non-Linear Planning in AI can reuse previously generated plans for similar problems, thus reducing the computational overhead required for solving similar problems and improving the efficiency of the planning process.
- **Integration:**
  Non-Linear Planning can be integrated with other AI techniques, such as machine learning and decision-making algorithms, to create more robust and adaptable problem-solving systems.

Disadvantages of Non-Linear Planning in AI

- **Complexity:**
  Non-Linear Planning in AI involves dealing with complex and interconnected sub-goals. This complexity can make the planning process more challenging, increasing the computational overhead and time required to generate a plan.
- **Scalability:**
  Non-Linear Planning may need to be more scalable to large problem spaces. As the number of sub-goals and dependencies increases, the search space can become too large to explore efficiently.
- **Uncertainty:**
  Non-Linear Planning assumes a deterministic world in which actions have predictable outcomes. However, real-world problems often involve uncertainty, and the outcomes of actions may be difficult to predict accurately.
- **Domain expertise:**
  Non-Linear Planning relies on domain expertise to generate effective plans. With sufficient domain knowledge, defining appropriate sub-goals, identifying relevant actions, and specifying their dependencies may be more accessible.

Conditional Planning

UNCERTAINTY

•The agent might not know what the initial state is.

•The agent might not know the outcome of its actions.

•The plans will have branches rather than being straight line plans, includes *conditional steps*

•**if** $< test >$ **then** $plan_A$ **else** $plan_B$

•*Full observability*: The agent knows what state it currently is, does not have to execute an *observation action*

•Simply get plans ready for all possible contingencies

Modeling Uncertainty

- Actions sometimes fail → disjunctive effects Example: moving left sometimes fails
- *Action*(*Left*, PRECOND: *AtR*, EFFECT: *AtL* ∨ *AtR*)
- *Conditional effects*: effects are conditioned on secondary preconditions

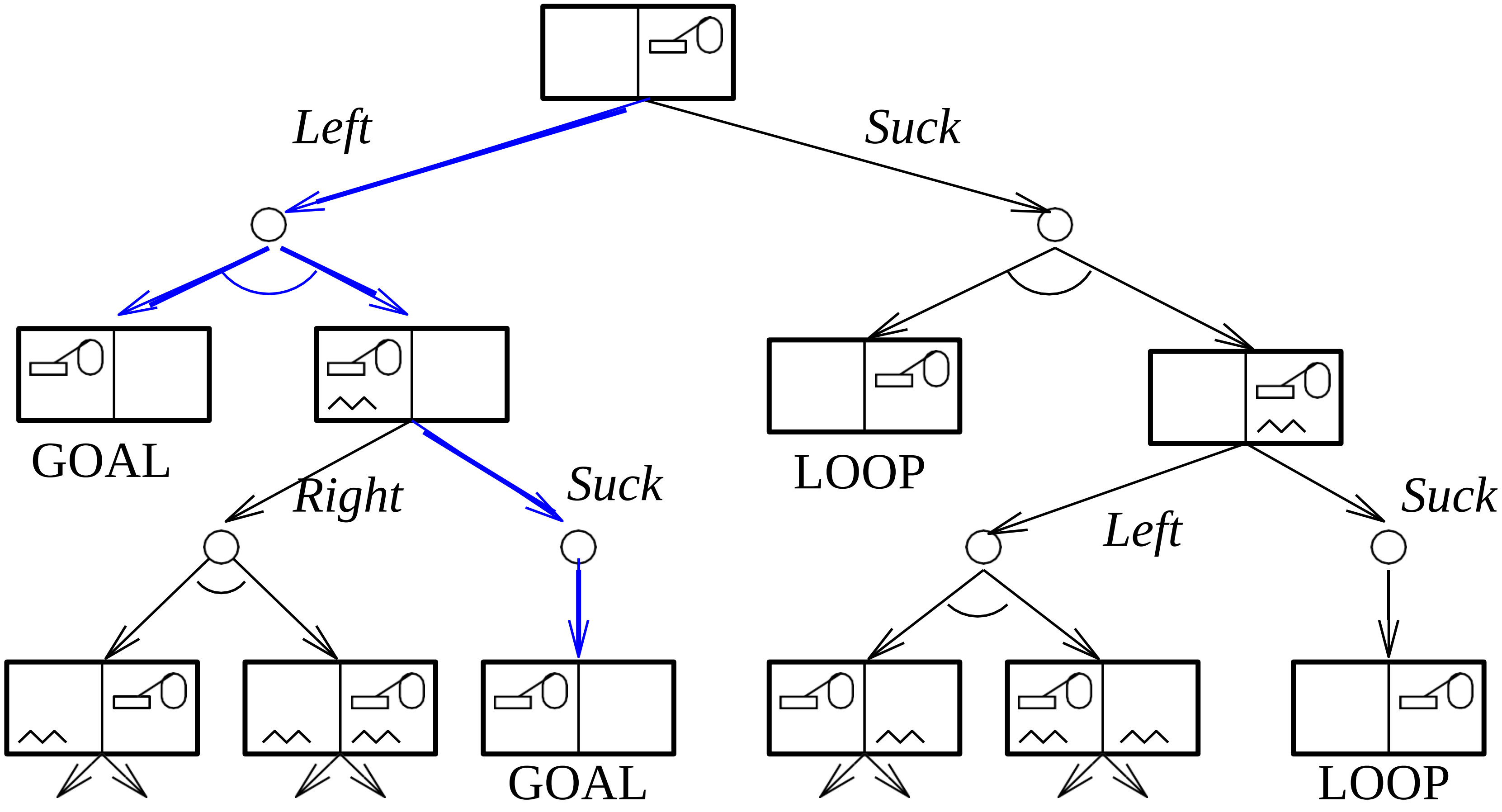
  Action(*Suck*, PRECOND: ;,

  EFFECT: (**when** *AtL: CleanL*) ∧ (**when** *AtR: CleanR*))
- Actions may have both disjunctive and conditional effects:
- Moving sometimes dumps dirt on the destination square only when that square is clean
- *Action*(*Left*, PRECOND: *AtR*;, EFFECT: *AtL* ∨ (*AtL* ∧ **when** *CleanL:* ¬ *CleanL*))

The vacuum world example

- Double Murphy world
  - the vacuum cleaner sometimes deposits dirt when it moves to a clean destination square
  - sometimes deposits dirt if SUCK is applied to a clean square
- The agent is playing a game against nature

Perform and-or search



The conditional plan

In the “double-Murphy” vacuum world, the plan is:

```
[
*Left*,
**if** *AtL* $\wedge$ *CleanL* $\wedge$ *CleanR*
**then** [ ]
**else** *Suck*
]
```

Reactive Planning

- Reactive planning is suitable for highly dynamic and unpredictable environments.
- Rather than following a pre-defined plan, the AI agent continuously reacts to changes in the environment in real-time.
- This approach doesn’t rely on creating a full plan ahead of time but focuses on immediate responses to the current situation.
- **Example**: A robot avoiding obstacles in an unknown environment or video game AI adapting to player actions.

Key Features:

- **Real-time Adaptation**: AI reacts dynamically to changes in the environment.
- **No Pre-computed Plan**: Focuses on immediate actions rather than long-term planning.
- **Challenges**: May lack long-term strategy or foresight, focusing only on immediate responses.

Knowledge based Planning

- Knowledge-based planning (KBP) can improve the efficiency of planning by automating the process and reducing operator dependence.
- KBP systems use past plan data to generate optimized plans, resulting in improved planning quality and consistency.
- These systems will reduce planning time and improve plan quality.

Knowledge based Planning using Temporal Logic

- Temporal logic is a subfield of mathematical logic that deals with reasoning about time and the temporal relationships between events.
- In artificial intelligence, temporal logic is used as a formal language to describe and reason about the temporal behavior of systems and processes.
- Temporal logic extends classical propositional and first-order logic with constructs for specifying temporal relationships, such as “before,” “after,” “during,” and “until.”

Execution Monitoring and Re-planning

- **An execution monitoring** agent checks its percepts to see whether everything is going according to plan.
- Two kinds of execution monitoring:
  - **Action monitoring,** whereby the agent checks the environment to verify that the next action will work.
  - **Plan monitoring,** in which the agent verifies the entire remaining plan.
- A **replanning** agent knows what to do when something unexpected happens: call a planner again to come up with a new plan to reach the goal.

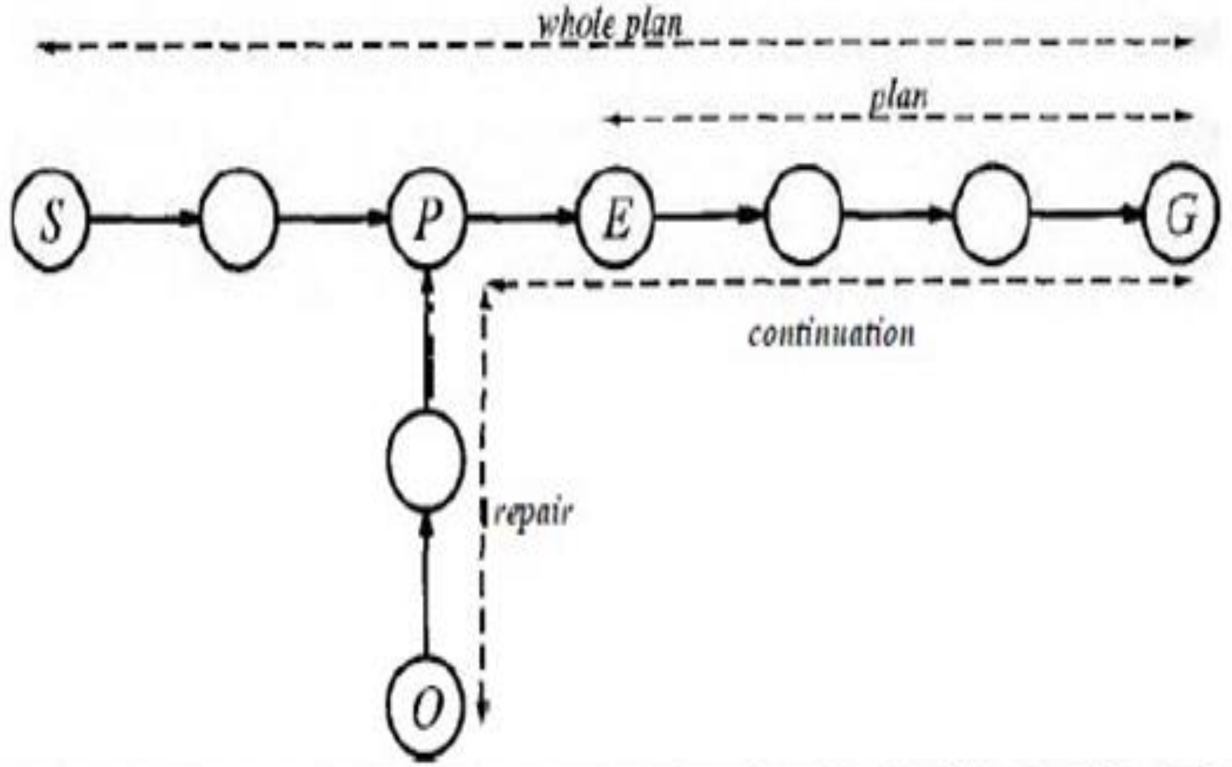
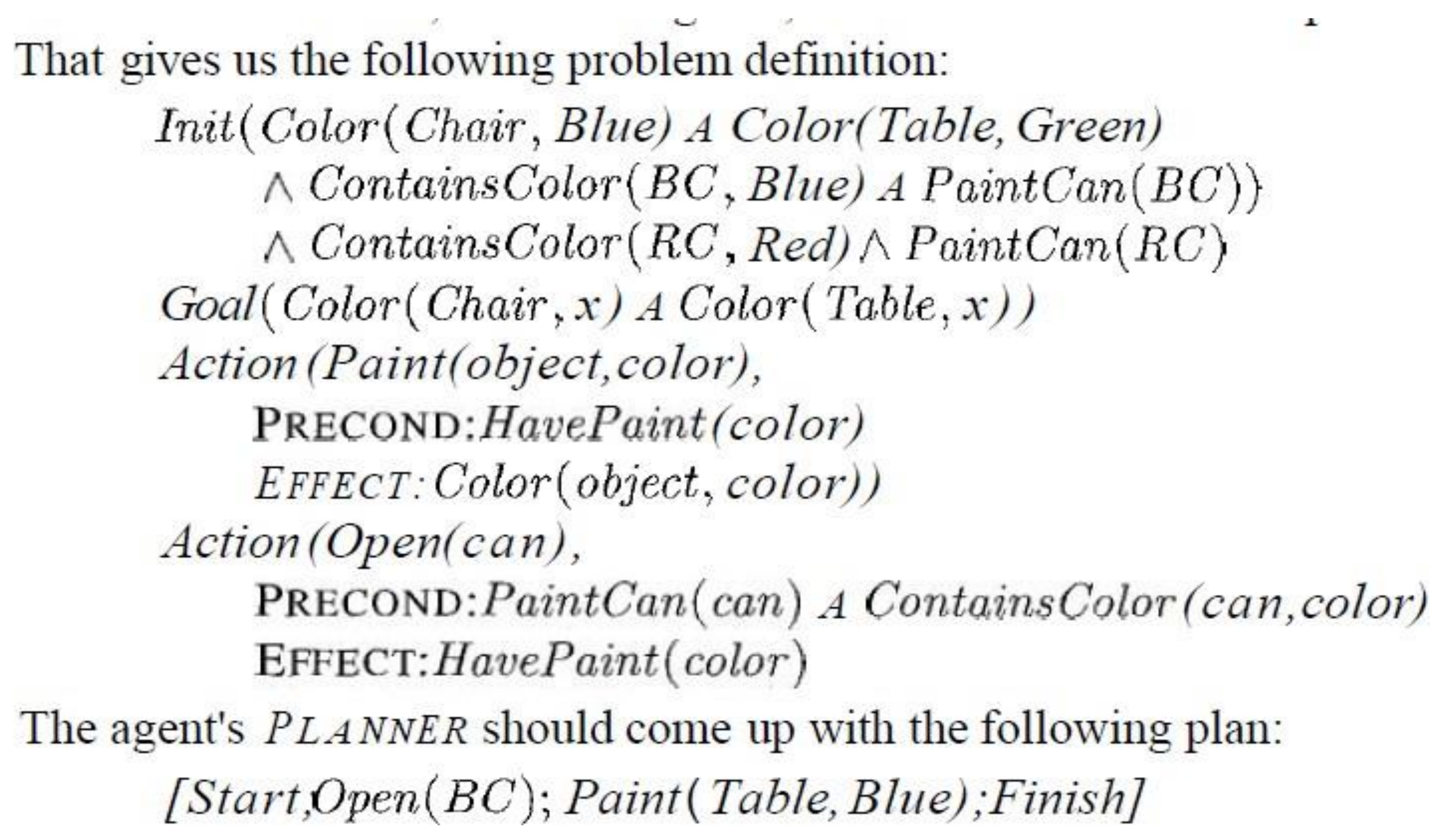


Figure 12.14 Before execution, the planner comes up with a plan, here called *whole-plan,* to get from S to G. The agent executes the plan until the point marked E. Before executing the remaining *plan,* it checks preconditions as usual and finds that it is actually in state O rather than state E. It then calls its planning algorithm to come up with *repair,* which is a plan to get from O to some point **P** on the original *whole-plan.* The new *plan* now becomes the concatenation of *repair* and *continuation* (the resumption of the original *whole-plan).*

Example problem of achieving a chair and table of matching color, this time via replanning. We'll assume a fully observable environment. In the initial state the chair is blue, the table is green, and there is a can of blue paint and a can of red paint. That gives us the following problem definition:



That gives us the following problem definition:

$Init(Color(Chair, Blue) \wedge Color(Table, Green)$
$\quad \wedge\, ContainsColor(BC, Blue) \wedge PaintCan(BC))$
$\quad \wedge\, ContainsColor(RC, Red) \wedge PaintCan(RC))$
$Goal(Color(Chair, x) \wedge Color(Table, x))$
$Action(Paint(object, color),$
$\quad$ PRECOND: $HavePaint(color)$
$\quad$ EFFECT: $Color(object, color))$
$Action(Open(can),$
$\quad$ PRECOND: $PaintCan(can) \wedge ContainsColor(can, color)$
$\quad$ EFFECT: $HavePaint(color)$

The agent's *PLANNER* should come up with the following plan:

$[Start, Open(BC); Paint(Table, Blue); Finish]$

- Now the agent is ready to execute the plan.
- Assume that all goes well as the agent opens the blue paint and applies it to the table.
- The agents from previous sections would declare victory at this point, having completed the steps in the plan.
- But the execution monitoring agent must first check the precondition of the *Finish* step, which says that the two pieces must have the same color.
- Suppose the agent perceives that they do not have the same color, because it missed a spot of green on the table.
- **The agent then needs to figure out a position in *whole-plan* to aim for and a repair action sequence to get there.**

Continuous Planning

- In this planning, the planner at first achieves the goal and then only can stop.
- A continuous planner is designed to persist over a lifetime.
- It can handle any unfavorable circumstances in the environment.

Fixing plan flaws continually

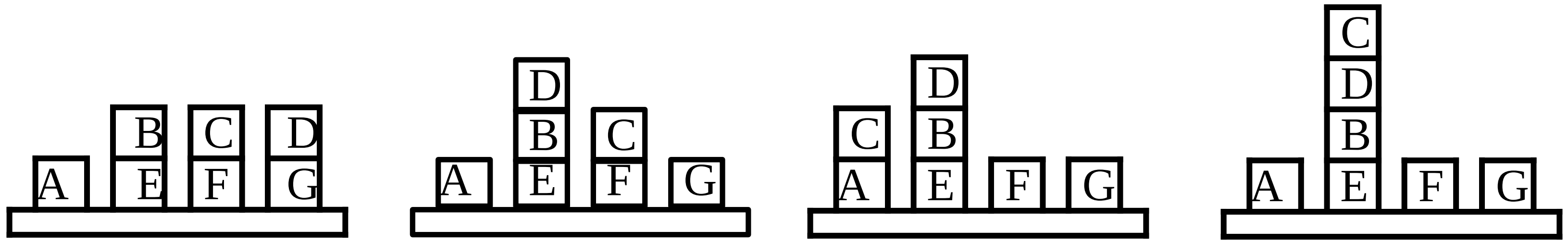
- *Missing goal*: adding new goals
- *Open precondition*: close using causal links (POP)
- *Causal conflict*: resolve threats (POP)
- *Unsupported link*: remove causal links supporting conditions tha
  are no longer true
- *Redundant action*: remove actions that supply no causal links
- *Unexecuted action*: return an action that can be executed
- *Unnecessary historical goal*: if the current goal set has been achieved, remove them and allow for new goals

Continuous planning algorithm

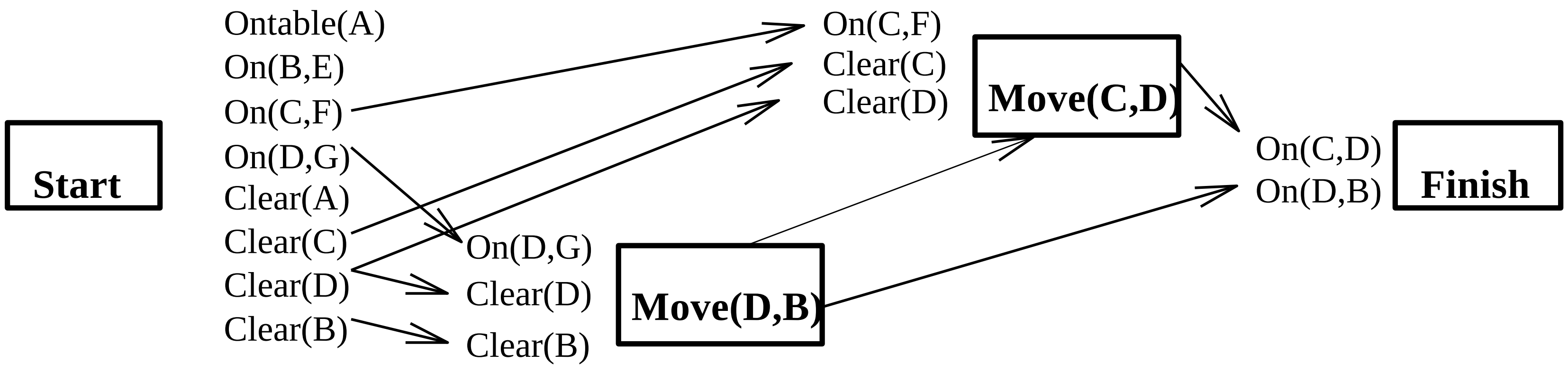
function CONTINUOUS-POP-AGENT(*percept*) **returns** an action

action NoOp (the default)
EFFECTS[*Start*] = UPDATE(EFFECTS [*Start*], *percept*) REMOVE-FLAW(*plan*) // possibly updating action **return** *action*

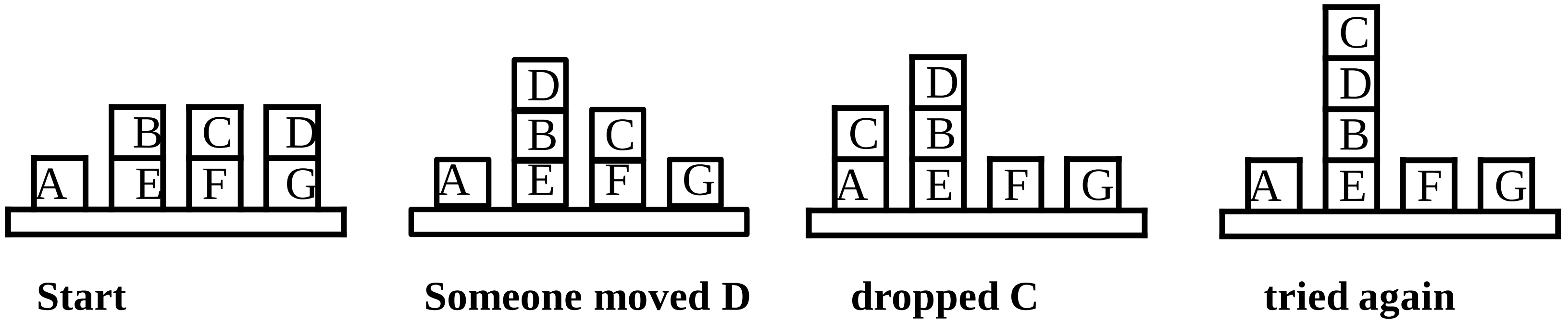
Example - start

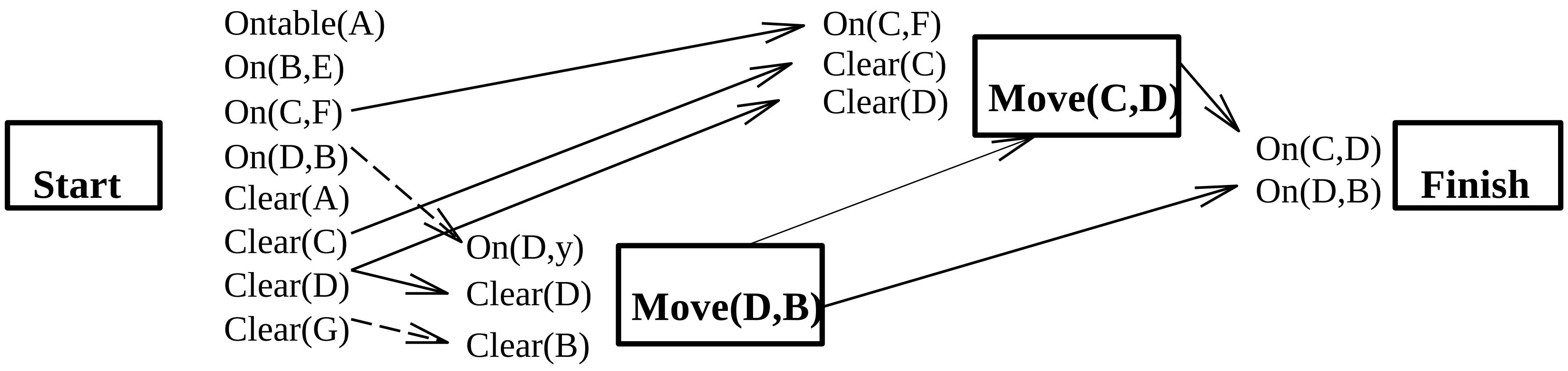


Start **Someone moved D** **dropped C** **tried again**

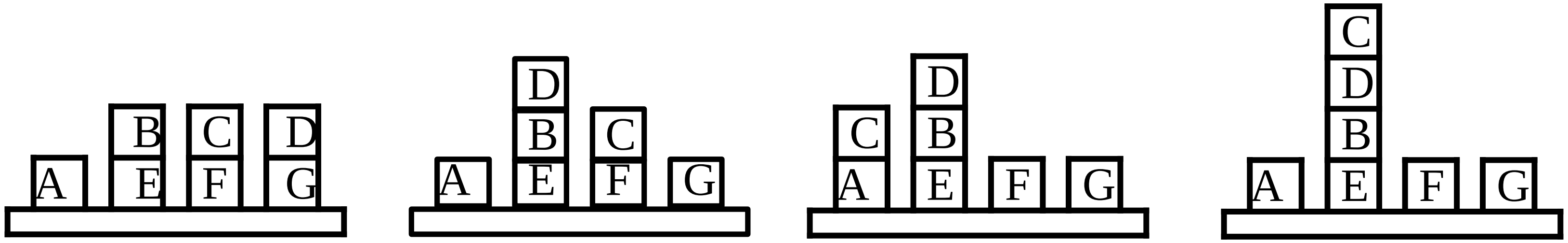


Example - after D is moved onto B

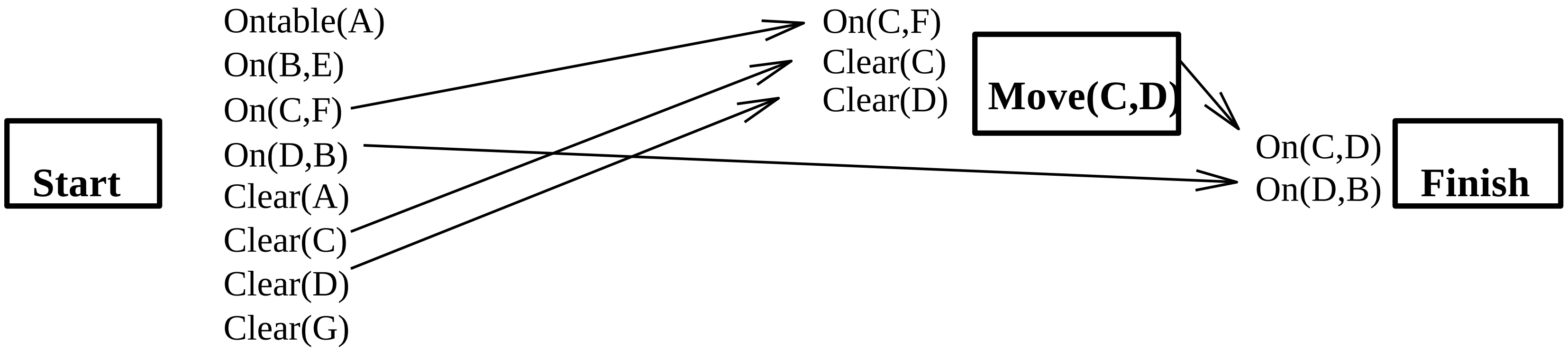




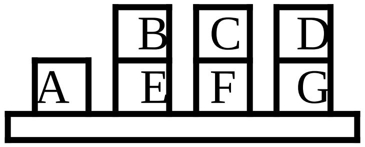
Example - Move(D,B) was redundant



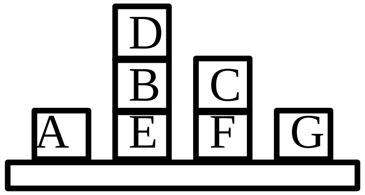
Start **Someone moved D** **dropped C** **tried again**



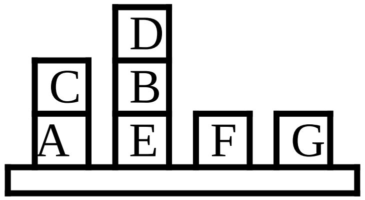
Example - Move(C,D) was executed



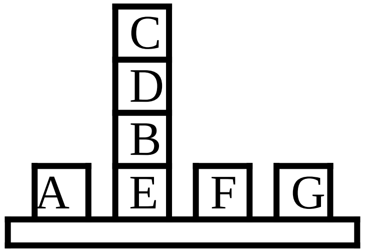
Start



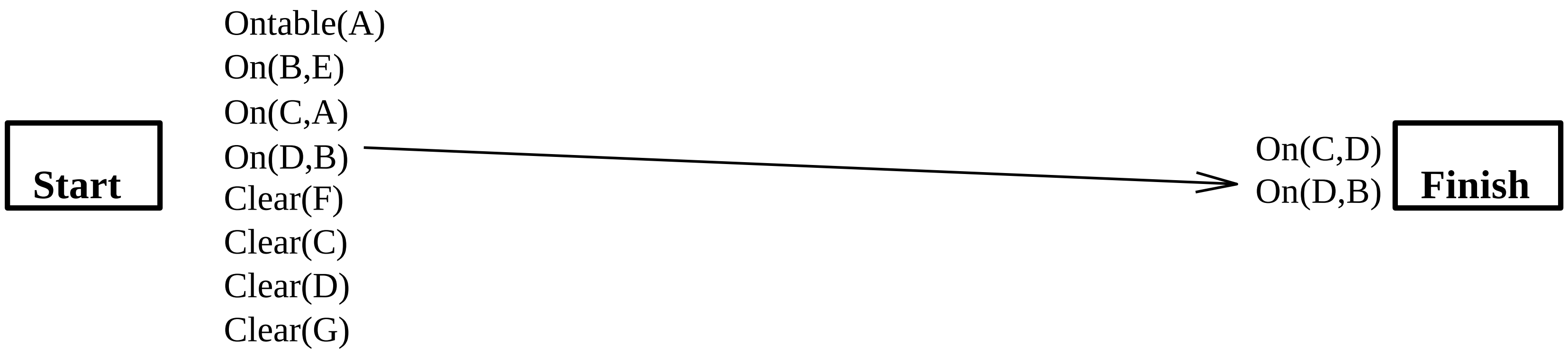
Someone moved D



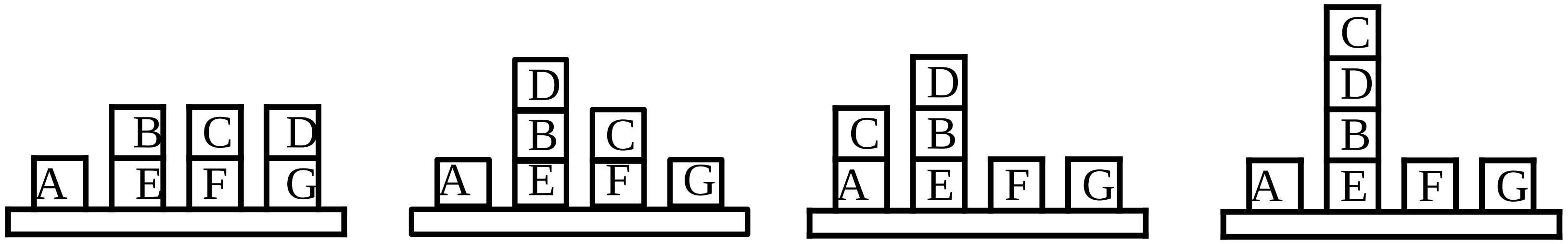
dropped C



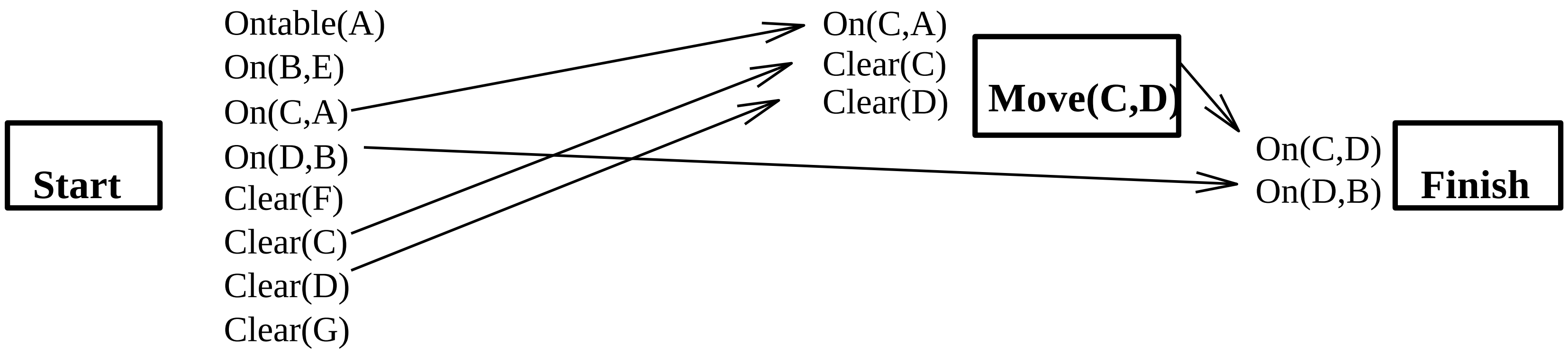
tried again



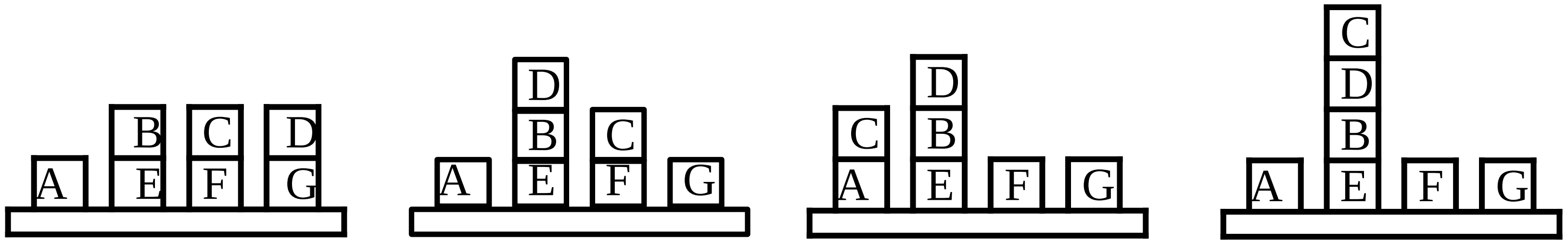
Example - put Move(C,D) back in



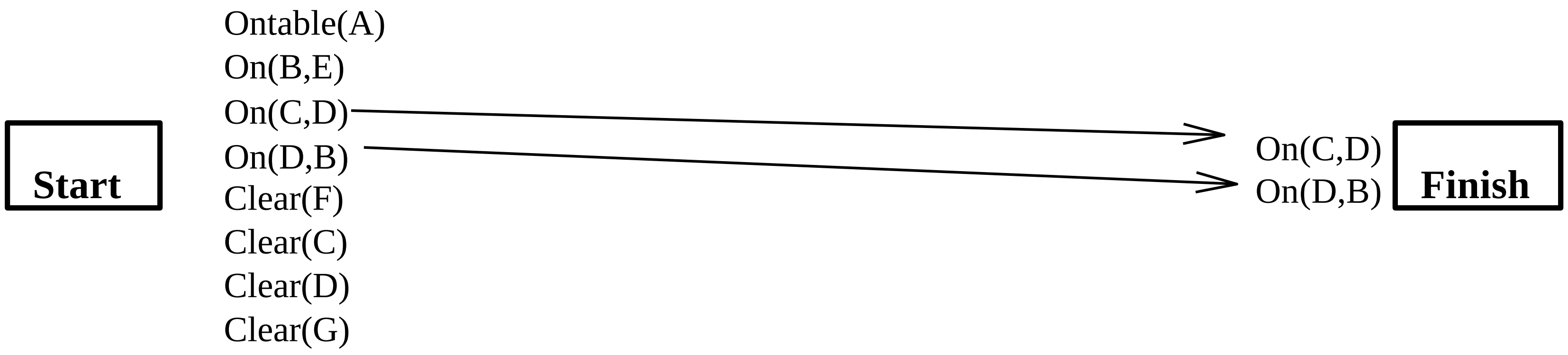
Start **Someone moved D** **dropped C** **tried again**



Example - plan complete



Start **Someone moved D** **dropped C** **tried again**



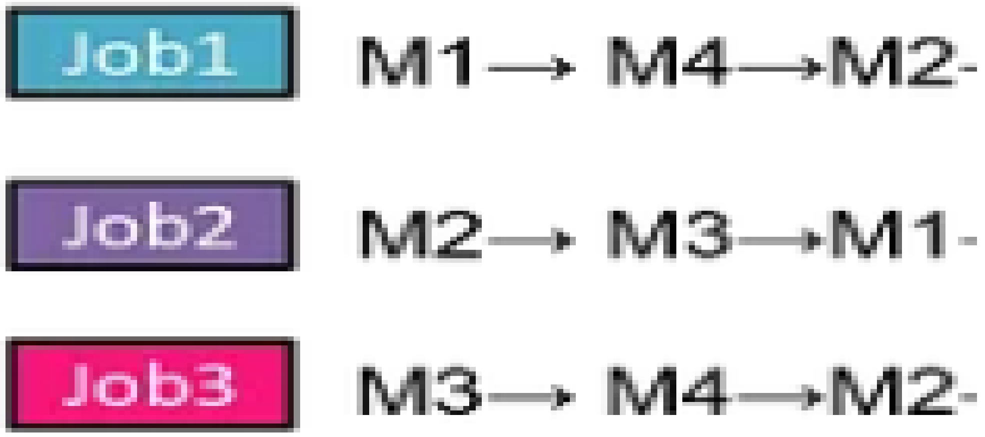
Multiagent planning

- *Cooperation:* Joint goals and plans
- *Multibody planning:* Synchronization, joint actions, concurrent actions
- *Coordination mechanisms:* convention, social laws, emergent behavior, communication, plan recognition, joint intention
- *Competition:* agents with conflicting utility functions
- This may lead to a poor performance because dealing with other agents is not the same as dealing with the nature. It is necessary when there are other agents in the environment with which to cooperate, compete or coordinate.

JOB SHOP SCHEDULING PROBLEM-(Flipped Classroom)

- **Job Shop Scheduling Problem** (JSSP), which aims to schedule several jobs over some machines in which each job has a unique machine route, for finding optimal sequences over machines.
- Assume there are 3 jobs (J1, J2, J3) and 4 Machines (M1, M2, M3, M4)
- Generate a plan estimating the duration of the job or processes and how it can handle the resources or machines.

- Generate a plan estimating the duration of the job or processes and how it can handle the resources or machines.
  Assume the Machine sequence of job execution given below while planning the activity.



Job1 M1 → M4 → M2

Job2 M2 → M3 → M1

Job3 M3 → M4 → M2