Module 9

Planning

Lesson 25

Planning algorithm - II

9.4.5 Partial-Order Planning

Total-Order vs. Partial-Order Planners

Any planner that maintains a partial solution as a totally ordered list of steps found so far is called a **total-order planner**, or a **linear planner**. Alternatively, if we only represent partial-order constraints on steps, then we have a **partial-order planner**, which is also called a **non-linear planner**. In this case, we specify a set of temporal constraints between pairs of steps of the form `S1 < S2` meaning that step S1 comes before, but not necessarily immediately before, step S2. We also show this temporal constraint in graph form as

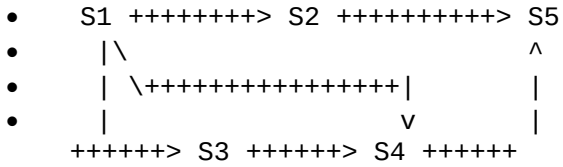
```
S1 +++++++++> S2
```

STRIPS is a total-order planner, as are situation-space progression and regression planners

Partial-order planners exhibit the property of least commitment because constraints ordering steps will only be inserted when necessary. On the other hand, situation-space progression planners make commitments about the order of steps as they try to find a solution and therefore may make mistakes from poor guesses about the right order of steps.

Representing a Partial-Order Plan

A partial-order plan will be represented as a graph that describes the temporal constraints between plan steps selected so far. That is, each node will represent a single step in the plan (i.e., an instance of one of the operators), and an arc will designate a temporal constraint between the two steps connected by the arc. For example,



graphically represents the temporal constraints S1 < S2, S1 < S3, S1 < S4, S2 < S5, S3 < S4, and S4 < S5. This partial-order plan implicitly represents the following three total-order plans, each of which is consistent with all of the given constraints:

[S1,S2,S3,S4,S5], [S1,S3,S2,S4,S5], and [S1,S3,S4,S2,S5].

9.5 Plan-Space Planning Algorithms

An alternative is to search through the space of *plans* rather than a space of *situations*. That is, we start with a simple, incomplete plan, which we call a **partial plan**. Then we consider ways of expanding the partial plan until we come up with a complete plan that

solves the problem. We use this approach when the ordering of sub-goals affects the solution.

Here one starts with a simple, incomplete plan, a partial plan, and we look at ways of expanding the partial plan until we come up with a complete plan that solves the problem. The operators for this search are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. Therefore the solution is the final plan.

Two types of operators are used:

- Refinement operators take a partial plan and add constraints to it. They eliminate some plans from the set and they never add new plans to it.
- A modification operator debugs incorrect plans that the planner may make, therefore we can worry about bugs later.

9.5.1 Representation of Plans

A **plan** is formally defined as a data structure consisting of the following 4 components:

1. A set of plan steps
2. A set of step ordering constraints
3. A set of variable binding constraints
4. A set of causal links

Example:

Plan(
STEPS:{S_1:*Op(*ACTION: *Start),*
S_2:*Op*(ACTION: *Finish,*
PRECOND: *Ontable*(c), *On*(b,c), *On*(a,b) },
ORDERINGS: {$S_1 < S_2$},
BINDINGS: {},
LINKS: {})

Key Difference Between Plan-Space Planning and Situation-Space Planning
In Situation-Space planners all operations, all variables, and all orderings must be fixed when each operator is applied. Plan-Space planners make commitments (i.e., what steps in what order) only as necessary. Hence, Plan-Space planners do least-commitment planning.

Start Node in Plan Space

The initial plan is created from the initial state description and the goal description by creating two "pseudo-steps:"

Start

```
P: none
     E: all positive literals defining the initial state
```

Finish

```
P: literals defining the conjunctive goal to be achieved
     E: none
```

and then creating the initial plan as: `Start ---------> Finish`

Searching Through Plan Space
There are two main reasons why a given plan may not be a solution:

Unsatisfied goal. That is, there is a goal or sub-goal that is not satisfied by the current plan steps.

Possible threat caused by a plan step that could cause the undoing of a needed goal if that step is done at the wrong time

So, define a set of **plan modification operators** that detect and fix these problems.

Example

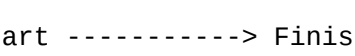
- Goal: Set the table, i.e., `on(Tablecloth) ^ out(Glasses) ^ out(Plates) ^ out(Silverware)`
- Initial State: `clear(Table)`
- Operators:

1. Lay-tablecloth
2.
```
P: clear(Table)
      E: on(Tablecloth), ~clear(Table)
```

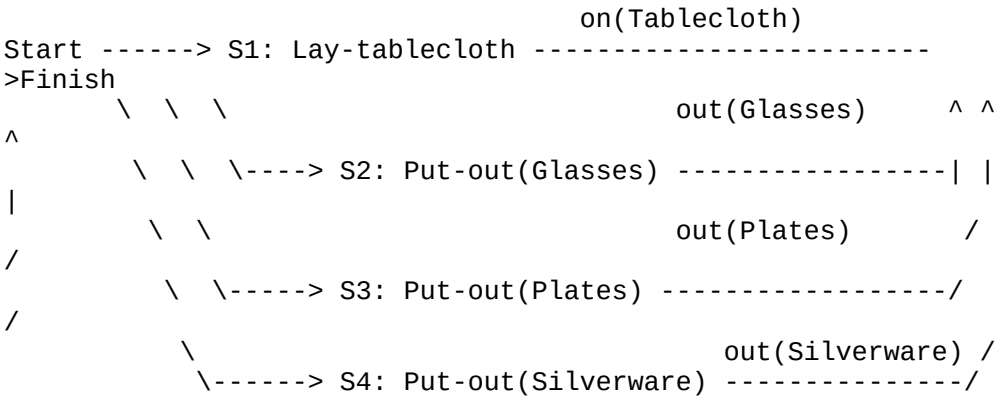
3. Put-out(x)
4.
```
P: none
      E: out(x), ~clear(Table)
```

- Searching for a Solution in Plan Space

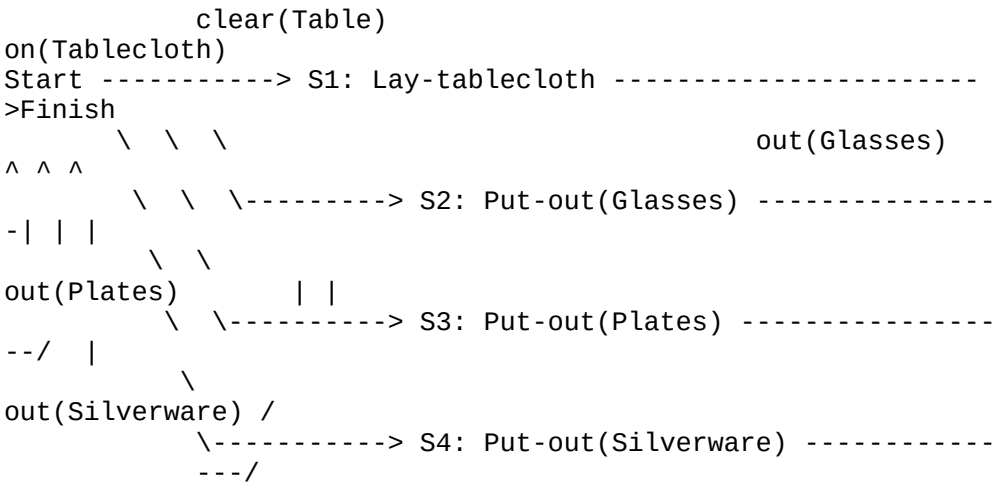
1. Initial Plan



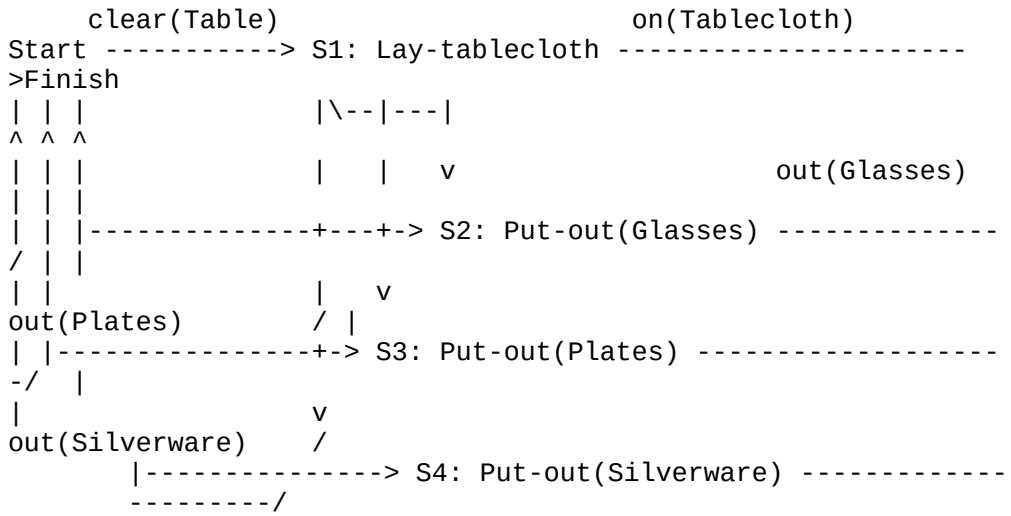
2. Solve 4 unsolved goals in `Finish` by adding 4 new steps with the minimal temporal constraints possible:



3. Solve unsolved subgoal `clear(Table)` which is a precondition of step `S1`:



4. Fix threats caused by steps `S2, S3`, and `S4` on the link from `Start` to `S1`. That is, `clear(Table)` is a necessary precondition of `S1` that is created by step `Start`. But `S2` causes `clear(Table)` to be deleted (negated), so if `S2` came *before* `S1`, `clear(Table)` wouldn't be true and step `S1` couldn't be performed. Therefore, add a temporal constraint that forces `S2` to come anytime *after* `S1`. That is, add constraint `S1 < S2`. Similarly, add `S1 < S3`, and `S1 < S4`, resulting in the new plan:



5. No threats and no unsolved goals in this plan, so it is a complete plan (i.e., a solution to the planning problem). Any total ordering of the steps implied by this partial-order plan is a solution plan. Here, there are six possible plans, where the first step is S1, and the steps S2, S3, and S4 follow in any order. (Don't include the pseudo-steps Start and Finish.)

Interleaving vs. Non-Interleaving of Sub-Plan Steps

Given a conjunctive goal, `G1 ^ G2`, if the steps that solve `G1` must either all come before or all come after the steps that solve `G2`, then the planner is called a **non-interleaving planner**. Otherwise, the planner allows interleaving of sub-plan steps. This constraint is different from the issue of partial-order vs. total-order planners. STRIPS is a non-interleaving planner because of its use of a stack to order goals to be achieved.

Partial-Order Planner (POP) Algorithm

```
function pop(initial-state, conjunctive-goal, operators)
  // non-deterministic algorithm
  plan = make-initial-plan(initial-state, conjunctive-goal);
  loop:
    begin
      if solution?(plan) then return plan;
      (S-need, c) = select-subgoal(plan) ; // choose an unsolved goal
      choose-operator(plan, operators, S-need, c);
          // select an operator to solve that goal and revise plan
      resolve-threats(plan);  // fix any threats created
    end
end

function solution?(plan)
```

```
  if causal-links-establishing-all-preconditions-of-all-steps(plan)
     and all-threats-resolved(plan)
     and all-temporal-ordering-constraints-consistent(plan)
     and all-variable-bindings-consistent(plan)
  then return true;
  else return false;
end

function select-subgoal(plan)
  pick a plan step S-need from steps(plan) with a precondition c
     that has not been achieved;
  return (S-need, c);
end

procedure choose-operator(plan, operators, S-need, c)
  // solve "open precondition" of some step
  choose a step S-add by either
    Step Addition: adding a new step from operators that
       has c in its Add-list
    or Simple Establishment: picking an existing step in Steps(plan)
       that has c in its Add-list;
  if no such step then return fail;
  add causal link "S-add --->c S-need" to Links(plan);
  add temporal ordering constraint "S-add < S-need" to Orderings(plan);
  if S-add is a newly added step then
    begin
    add S-add to Steps(plan);
    add "Start < S-add" and "S-add < Finish" to Orderings(plan);
    end
end

procedure resolve-threats(plan)
  foreach S-threat that threatens link "Si --->c Sj" in Links(plan)
   begin     // "declobber" threat
     choose either
       Demotion: add "S-threat < Si" to Orderings(plan)
       or Promotion: add "Sj < S-threat" to Orderings(plan);
     if not(consistent(plan)) then return fail;
   end
end
```

Plan Modification Operations

The above algorithm uses four basic plan modification operations to revise a plan, two for solving a goal and two for fixing a threat:

- **Establishment -- "Solve an Open Precondition" (i.e., unsolved goal)**
  If a precondition `p` of a step `S` does not have a causal link to it, then it is not yet solved. This is called an **open precondition**. Two ways to solve:
  - **Simple Establishment**
    Find an existing step `T` prior to `S` in which `p` is necessarily true (i.e., it's in the Effects list of `T`). Then add causal link from `T` to `S`.

- **Step Addition**
  Add a new plan step `T` that contains in its Effects list `p`. Then add causal link from `T` to `S`.

- **Declobbering -- Threat Removal**
  A threat is a relationship between a step `S3` and a causal link `S1 --->p S2`, where `p` is a precondition in step `S2`, that has the following form:

```
   -------> S1 --------->p S2
   |
   |
 -------> S3 ~p
```

That is, step `S3` has effect `~p` and from the temporal links could possibly occur in-between steps `S1` and `S2`, which have a causal link between them. If this occurred, then `S3` would "clobber" the goal `p` "produced" by `S1` before it can be "consumed" by `S2`. Fix by ensuring that `S3` cannot occur in the "protection interval" in between `S1` and `S2` by doing either of the following:

- **Promotion**
  Force threatening step to come *after* the causal link. I.e., add temporal link S2 < S3.

  Demotion
  Force threatening step to come *before* the causal link. I.e., add temporal link S3 < S1.

9.5.2 Simple Sock/Shoe Example

In the following example, we will show how the planning algorithm derives a solution to a problem that involves putting on a pair of shoes. In this problem scenario, Pat is walking around his house in his bare feet. He wants to put some shoes on to go outside. *Note: There are no threats in this example and therefore is no mention of checking for threats though it a necessary step*

To correctly represent this problem, we must break down the problem into simpler, more atomic states that the planner can recognize and work with. We first define the Start operator and the Finish operator to create the minimal partial order plan. As mentioned before, we must simplify and break down the situation into smaller, more appropriate states. The Start operator is represented by the effects: ~LeftSockOn, ~LeftShoeOn, ~RightSockOn, and ~RightShoeOn. The Finish operator has the preconditions that we wish to meet: LeftShoeOn and RightShoeOn. Before we derive a plan that will allow Pat to reach his goal (i.e. satisfying the condition of having both his left and right shoe on) from the initial state of having nothing on, we need to define some operators to get us

there. Here are the operators that we will use and the possible state (already mentioned above).

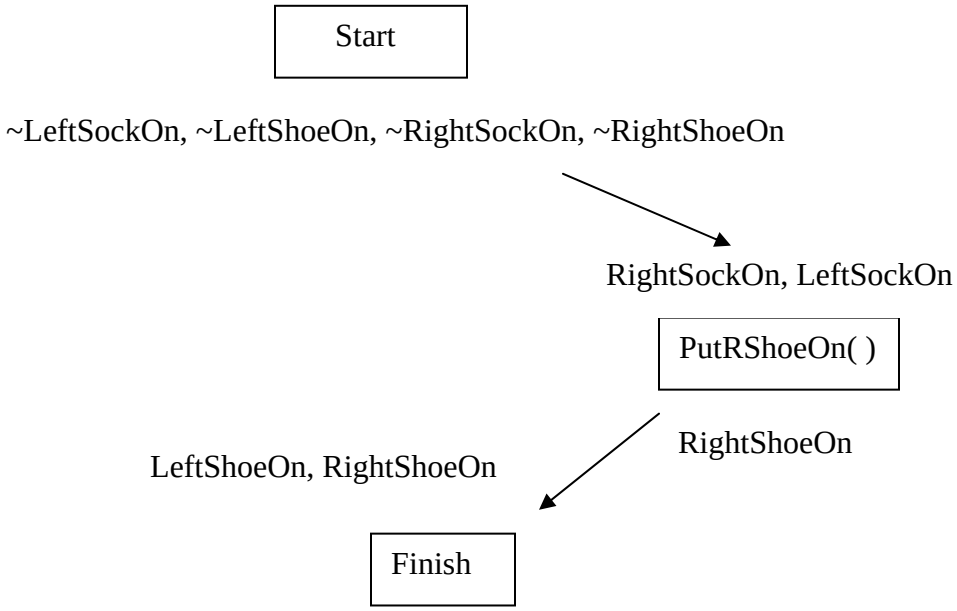
Operators
Op(ACTION: PutLeftSockOn() PRECOND:~LeftSockOn EFFECT: LeftSockOn)
Op(ACTION: PutRightSockOn() PRECOND:~RightSockOn EFFECT: RightSockOn)
Op(ACTION: PutLeftShoeOn() PRECOND:LeftSockOn EFFECT: LeftShoeOn)
Op(ACTION: PutRightShoeOn() PRECOND:RightShoeOn EFFECT: RightShoeOn)

States
LeftSockOn, LeftShoeOn, RightSockOn, RightShoeOn

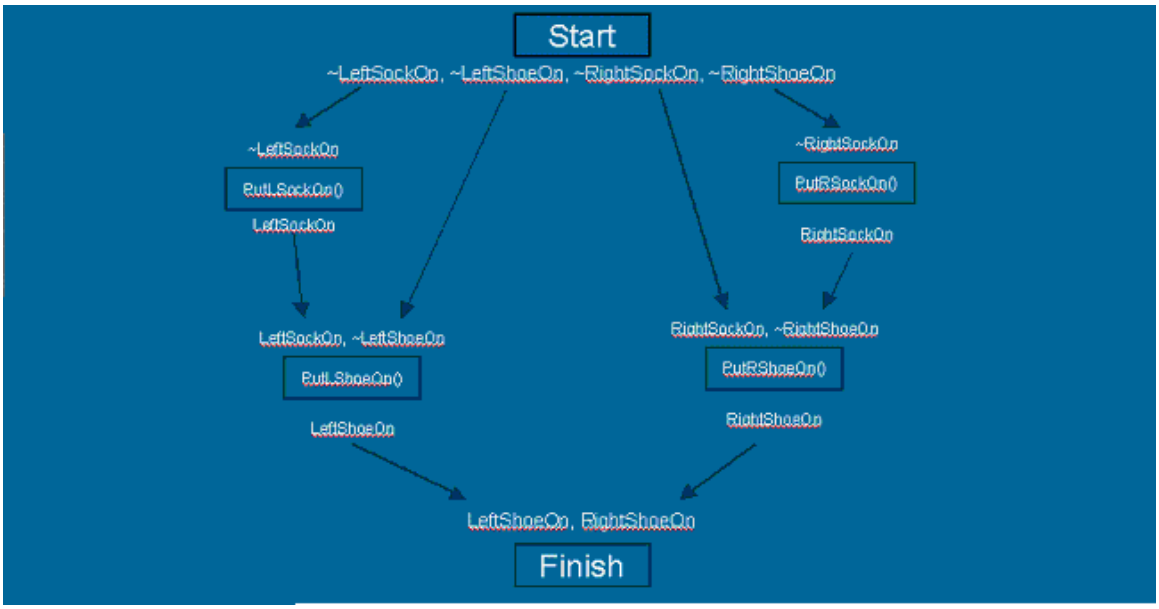
Creating A Plan

From the states listed above, we first create a minimal partial order plan. We can represent bare feet (Start operator) by saying that Pat is not wearing any socks or shoes and shoes on (Finish operator) with the two shoe on states. Here is the minimal partial order plan.

Initially we have two preconditions to achieve; RightShoeOn and LeftShoeOn. Let's start with the condition of having our right shoe on. We must choose an operator that will result in this condition. To meet this condition we need to the operator 'PutRightShoeOn()'. We add the operator and create a causal link between it and the Finish operator. However, adding this operator results a new condition (i.e. precondition of PutRightShoeOn()) of having the right sock on.



At this point we still have two conditions to meet: having our left shoe on and having our right sock on. We continue by selecting one of these two preconditions and trying to achieve it. Let's pick the precondition of having our right sock on. To satisfy this condition, we must add another step, operator 'PutRightSockOn()'. The effects of this operator will satisfy the precondition of having our right sock on. At this point, we have achieved the 'RightSockOn' state. Since the precondition of the 'PutRightSockOn()' operator is one of the effects of the Start operator, we can simply draw a causal link between the two operators. These two steps can be repeated for Pat's left shoe. The plan is complete when all preconditions are resolved.

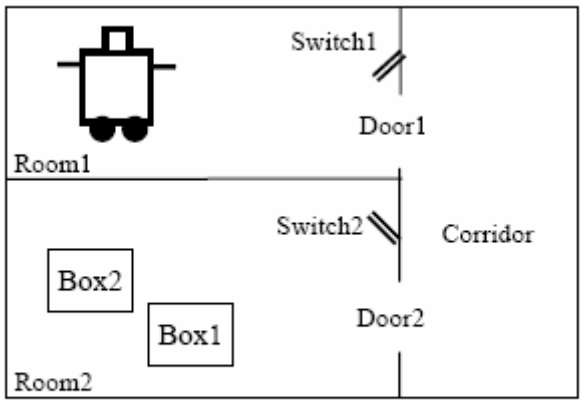


The Partial Order Planning algorithm can be described as a form of regression planning that use the principle of least commitment. It starts with a minimal partial order plan that consists of a Start operator (initial state) and a Finish operator (goal state). It then chooses a precondition that has not been resolved and chooses an operator whose effect matches the precondition. It then checks if any threats were created by the addition of the operator and if one is detected, resolves it either by demoting the operator, promoting the operator, or backtracking (removing the operator). It continues to choose operators until a solution is found (i.e. all preconditions are resolved).

Solutions created by the Partial Order Planning algorithm are very flexible. They may be executed in many ways. They can represent many different total order plans (partial order plans can be converted to total order plans using a process called linearization). Lastly they can more efficiently if steps are executed simultaneously.

Questions

1. Consider the world of Shakey the robot, as shown below.



NOTE: The intended interpretation of the switches drawn is that Switch1 is in the off position and Switch2 is in the on position.

Shakey has the following six actions available:

• *Go(x,y)*, which moves Shakey from *x* to *y*. It requires Shakey to be at *x* and that *x* and *y* are locations in the same room. By convention a door between two rooms is in both of them, and the corridor counts as a room.

• *Push(b,x,y)*, which allows Shakey to push a box *b* from location *x* to location *y*. Both Shakey and the box must be at the same location before this action can be used.

• *ClimbUp(b,x)*, which allows Shakey to climb onto box *b* at location *x*. Both Shakey and the box must be at the same location before this action can be used. Also Shakey must be on the *Floor*.

• *ClimbDown(b,x)*, which allows Shakey to climb down from a box *b* at location *x*. Shakey must be on the box and the box must be in location *x* before this action can be used.

• *TurnOn(s,x)*, which allows Shakey to turn on switch *s* which is located at location *x*. Shakey must be on top of a box at the switch's location before this action can be used.

• *TurnOff(s,x)*, which allows Shakey to turn off switch *s* which is located at location *x*. Shakey must be on top of a box at the switch's location before this action can be used.

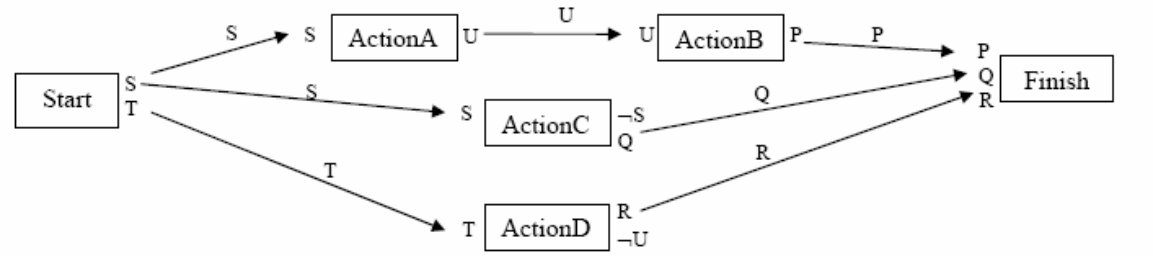
Using STRIPS syntax, define the six actions from above. In your action definitions, use only the following predicates: *Box(b)* to mean that *b* is a box, *In(x,r)* to mean that location *x* is in room *r*, *At(x,y)* to mean that the object *x* is at location *y*, *ShakeyOn(x)* to mean that Shakey is on the object *x*, *Switch(s)* to mean that *s* is a switch, and *SwitchOn(s)* to mean

that the switch *s* is on. You may also use the constants *Shakey* and *Floor* in the action definitions.

2. In the above problem, using STRIPS, define the initial state depicted on the previous page. Use only the predicates from part (a) and the constants *Box1*, *Box2*, *Switch1*, *Switch2*, *Floor*,*Shakey*, *Room1*, *Room2*, *Corridor*, L_{Door1}, L_{Door2}, $L_{ShakeyStart}$, $L_{Switch1}$, $L_{Box1Start}$, $L_{Box2Start}$,$L_{Switch2}$. The L_x constants are intended to represent the locations of *x*,

3. Provide a totally ordered plan for Shakey to turn off *Switch2* using the actions and the initial state defined in (2) and (3).

4. Consider the inconsistent partially ordered plan below. Identify the conflicts in this plan and show all ways of resolving them that follow the principle of least commitment. For each solution, draw the new partially ordered plan, and list all of its linearizations.



Solutions

1. STRIPS description of the operators are:

Action(Go(x, y),
Pre: At(Shakey, x) ∧ In(x, r) ∧ In(y, r) ∧ ShakeyOn(Floor)
Eff: At(Shakey,y) ∧ −At(Shakey, x))
Action(Push(b, x, y),
Pre: At(Shakey, x) ∧ At(b, x) ∧ Box(b) ∧ ShakeyOn(Floor) ∧ In(x, r) ∧ In(y, r)
Eff: At(Shakey, y) ∧ At(b, y) ∧ −At(Shakey, x) ∧ −At(b, x))
Action(ClimbUp(b, x),
Pre: At(Shakey, x) ∧ At(b, x) ∧ Box(b) ∧ ShakeyOn(Floor)
Eff: −ShakeyOn(Floor) ∧ ShakeyOn(b))
Action(ClimbDown(b, x),
Pre: ShakeyOn(b) ∧ Box(b) ∧ At(b, x) ∧ At(Shakey, x)
Eff: −ShakeyOn(b) ∧ ShakeyOn(Floor))
Action(TurnOn(s, x),
Pre: At(Shakey, x) ∧ At(s, x) ∧ Switch(s) ∧ Box(b) ∧ At(b,x) ∧ ShakeyOn(b)
Eff: SwitchOn(s))
Action(TurnOff(s, x),
Pre: At(Shakey, x) ∧ At(s, x) ∧ Switch(s) ∧ Box(b) ∧ At(b,x) ∧ ShakeyOn(b) ∧ SwitchOn(s)
Eff: −SwitchOn(s))

2. The initial state is:

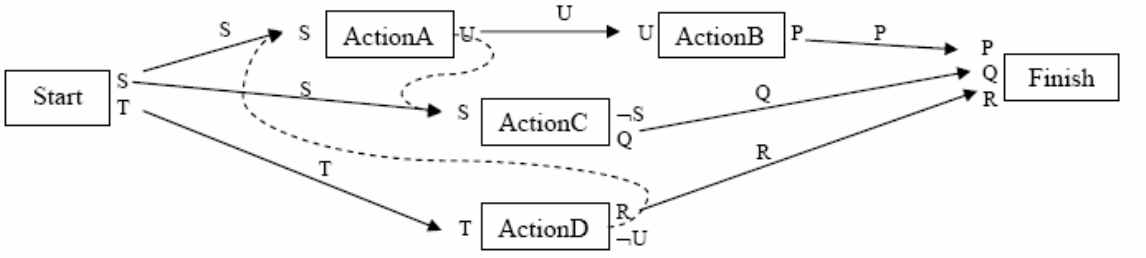
$Box(Box1) \wedge Box(Box2) \wedge Switch(Switch1) \wedge Switch(Switch2) \wedge$
$SwitchOn(Switch2) \wedge ShakeyOn(Floor) \wedge In(L_{ShakeyStart}, Room1) \wedge In(L_{Door1}, Room1) \wedge$
$In(L_{Door1}, Corridor) \wedge In(L_{Door2}, Room2) \wedge In(L_{Door2}, Corridor) \wedge In(L_{Switch1}, Room1) \wedge$
$In(L_{Box1Start}, Room2) \wedge In(L_{Box2Start}, Room2) \wedge In(L_{Switch2}, Room2) \wedge At(Shakey, L_{ShakeyStart}) \wedge$
$At(Box1, L_{Box1Start}) \wedge At(Box2, L_{Box2Start}) \wedge At(Switch1, L_{Switch1}) \wedge At(Switch2, L_{Switch2})$

3. The plan is

$Go(L_{ShakeyStart}, L_{Door1})$
$Go(L_{Door1}, L_{Door2})$
$Go(L_{Door2}, L_{Box1Start})$
$Push(Box1, L_{Box1Start}, L_{Switch2})$
$ClimbUp(Box1, L_{Switch2})$
$TurnOff(Switch2, L_{Switch2})$

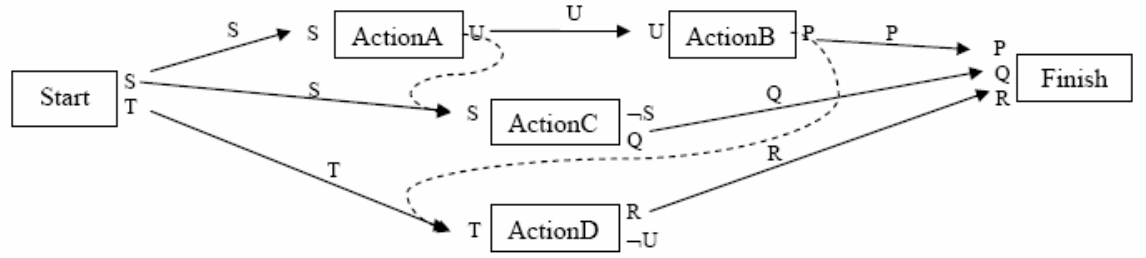
4. Conflicts:
• ActionC cannot come between Start and ActionA.
• ActionD cannot come between ActionA and ActionB.

Resolution 1:



Linearizations of Resolution 1: (ActionA abbreviated as A, and so on…)
Start, D, A, B, C, Finish
Start, D, A, C, B, Finish

Resolution 2:



Linearizations of Resolution 2: (ActionA abbreviated as A, and so on…)
Start, A, B, D, C, Finish

Start, A, B, C, D, Finish
Start, A, C, B, D, Finish