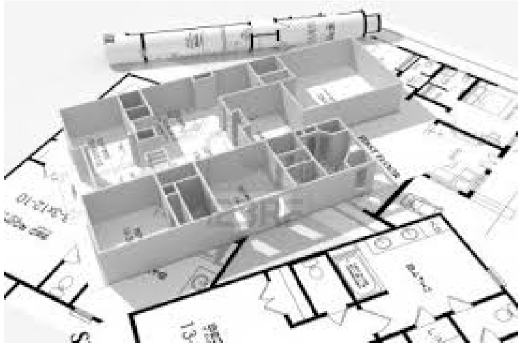
Planning & Knowledge

DV2557

Dr. Prashant Goswami
Senior Lecturer, BTH (DIDA)
prashantgos.github.io/

prashant.goswami@bth.se



KNOWLEDGE REPRESENTATION

Toy vs. Real World problems

- So far the logic problems we have faced are ”toy” problems.
- It is often quite easy to find a consistent vocabulary and representation for such limited worlds.
- When dealing with real world problems, some more issues arise:
    - Deal with actions
    - How to represent time
    - Physical Objects vs. Mental Objects
    - Beliefs
    - …

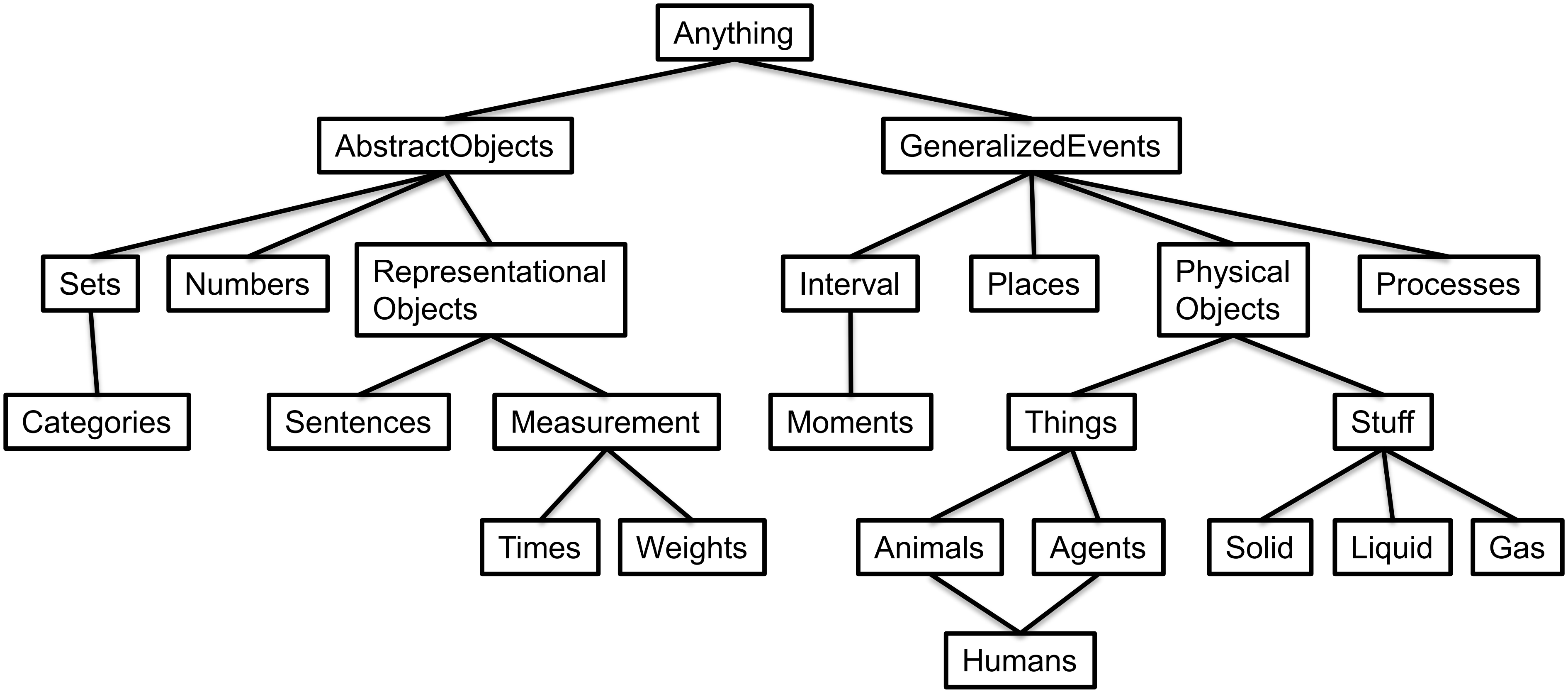
Fortunately…

- … a lot of engineers and philosophers have spent lot of time thinking about this.
- The field is called *ontological engineering*.
- It describes formal, generalized ways of describing the world around us, on different levels of detail.
- It involves describing abstract concepts such as actions, time, physical objects and beliefs.

Upper Ontology

- The *upper ontology* is the general framework for describing a world.
- Very abstract concepts are at the top, and the lower you get in the graph the more specialized the concepts become.
- An ontology for a problem can be seen as an instance of the upper ontology.

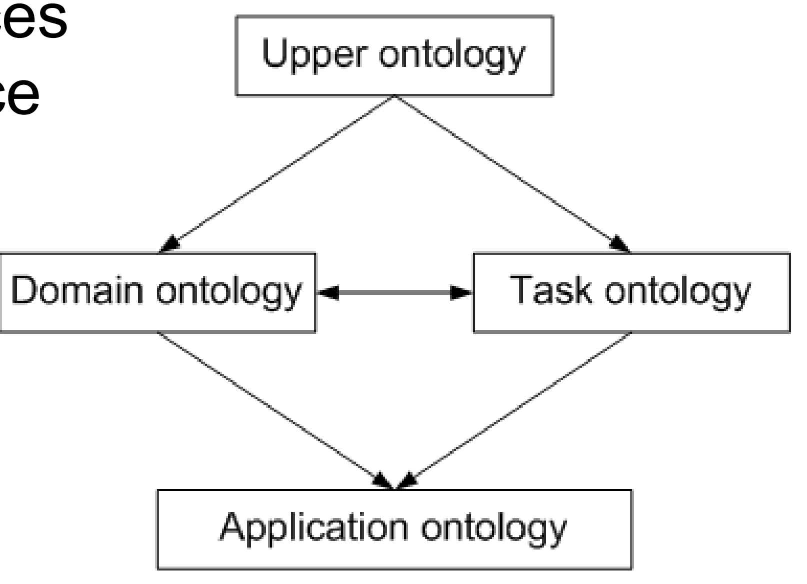
Upper Ontology



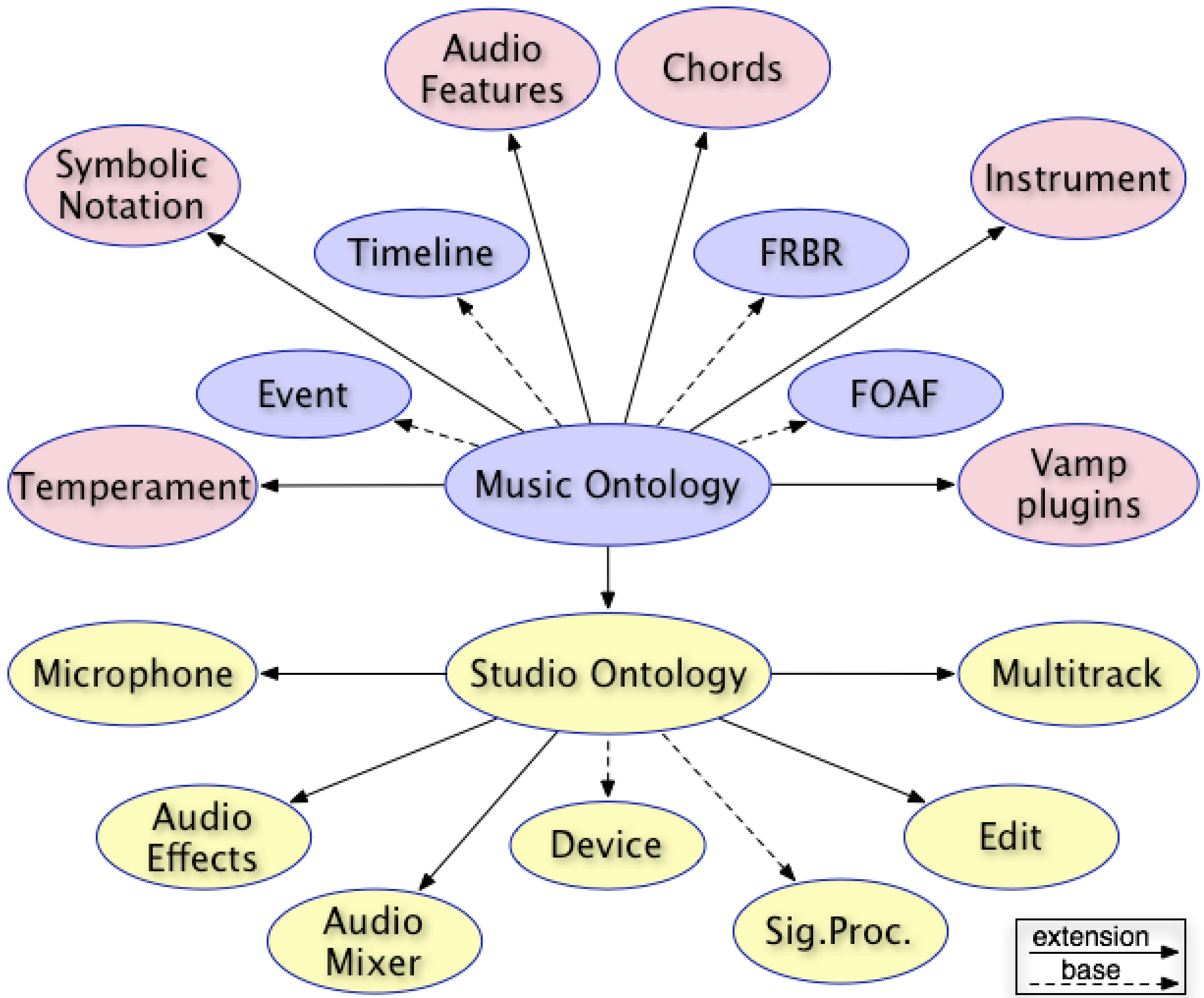
- Each concepts is a more specialized version of the upper one.
- This is one version of upper ontology, there are others…

Upper Ontology

- The two major differences between an upper ontology and a special-purpose ontology are:
  1. A general-purpose ontology shall be applicable in (almost) any special-purpose ontology.
  2. Different areas of knowledge must be unified. Sentences describing time and space must handle seconds, years, meters, mm, …



Application Ontology



Categories and Objects

- The basketball object b_1 is a member of the category *Basketballs*, formally written as:
  - $b_1 \in Basketballs$ ”b_1 is an element of…”
- *Basketballs* are in turn a subcategory of *Balls*:
  - Basketballs $\subset$ Balls ”… is a subset of...”
- This is important, since we can infer that every basketball object is round if *Balls* are round.
- We must however be able to handle exceptions:
  - Most, but not all, tomatoes are red…

Categories and Properties

- All categories can have properties, which are <u>inherited</u> by members and subcategories.
  - $x \in Balls \Rightarrow Round(x)$
  - $x \in Basketballs \Rightarrow Orange(x)$
  - … If:
  - $b_1 \in Basketballs$
  - … then we can infer that:
  - $Round(b_1) \wedge Orange(b_1)$
  - … it can also be used to recognize objects:
  - $Round(x) \wedge Orange(x) \wedge x \in Balls \Rightarrow x \in Basketballs$

Disjoint Categories

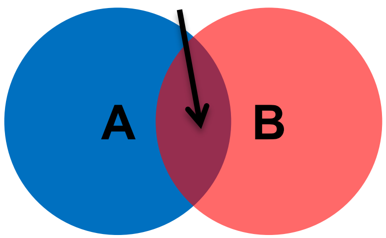
- Sometimes we want to state relations between categories at the same level.
- Example:
  - *Males* ∈ *Humans*
  - *Females* ∈ *Humans*
  - … both are subcategories of the same category, but they have no members in common. We call these *disjoint* categories.
  - *Disjoint*({*Males, Females*})

Disjoint Categories

- Disjoint does not explicitly state that a human must be female if it is not male.
- A category where an object must belong to one of the categories are called an *exhaustive decomposition*:
    - *ExhaustiveDecomposition*({*Americans*, *Canadians*, *Mexicans*}, *NorthAmericans*)
- An *exhaustive decomposition* must not be *disjoint*. Some people have dual citizenship.
- A *disjoint exhaustive decomposition*, like males and females, is called a *partition*:
    - *Partition*({*Males*, *Females*}, *Humans*)

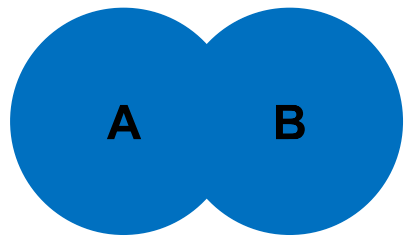
Notes on Set Theory

Intersection



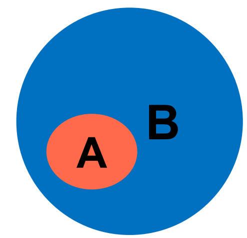
$A \cap B$

Union

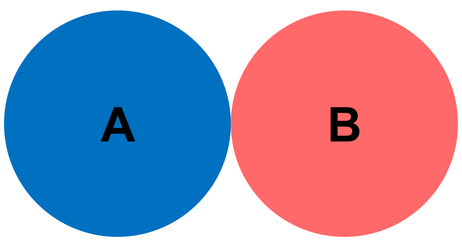


$A \cup B$

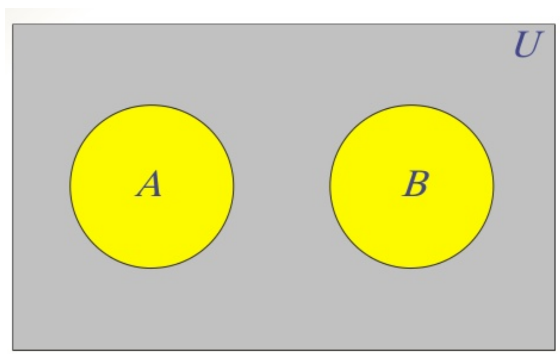
Subset



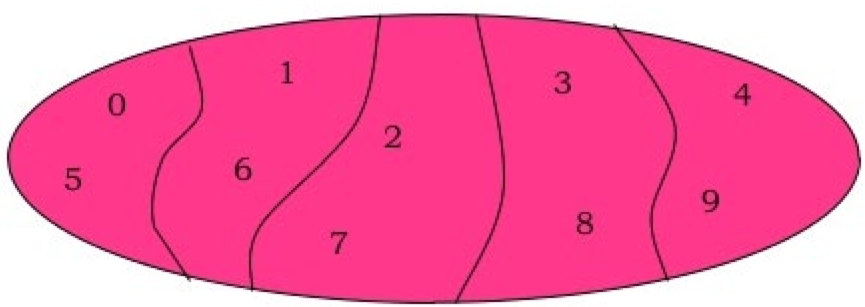
$A \subseteq B$



Intersection(A,B) = { }



Disjoint



Partition

Physical Composition

- We also have to deal with objects being part of other objects:
  - … such as Sweden being part of Europe:
  - *PartOf*(*Sweden*, *Europe*)
  - … Sweden is also part of Scandinavia and Scandinavia is part of Europe. This is called a transitive relationship:
  - *PartOf*(*Sweden*, *Scandinavia*) ∧ *PartOf*(*Scandinavia*, *Europe*) ⇒ *PartOf*(*Sweden*, *Europe*)
- An object consisting of parts is called a *composite object*.

Objects and Stuff

- Most objects are either primitive- or composite objects (objects made up of primitive objects).
- Sometimes we run into a problem where we cannot divide something into distinct objects (*individualization*).
- We call this *stuff*.
- A very good example of *stuff* is butter.

Objects and Stuff

- The distinction is between:
  - Count nouns: holes, theorems, apples
  - Mass nouns: butter, water, energy
- We can divide an apple to get two halves of apple.
- If we divide butter we still have butter.
  - Unless we make a category TwoKilosOfButter, which we can divide into two KiloOfButter. But then it's not stuff any more.

Intrinsic and Extrinsic properties

- We make a distinction between intrinsic and extrinsic properties.
- IP belong to the substance rather than the object:
    - Density, boiling point, flavor, color, …
- EP belong to objects:
    - Weight, length, shape, …
- EP changes if we divide objects, IP does not.
- Things with only IP are substances and belong to the *Stuff* category.
- Things with at least one EP are objects and belong to the *Thing* category.

Measurements

- Quantitative measurements are usually expressed with *unit functions*:
  - *Length*(*a*) = *Centimeters*(3.81)
  - *Length*(*a*) = *Inches*(1.5) = *Centimeters*(3.81)
- Conversions are written as:
  - *Centimeters*(2.54 * *d*) = *Inches*(*d*)
- Measures can describe objects:
  - *Diameter*($basketball_1$) = *Inches*(9.5)
  - $d \in days \Rightarrow Duration(d) = Hours(24)$
  - *Price*($basketball_1$) = $(19)

Actions and Situations

- In many problems the world is not static. Actions happen that change the world, like moving the player in the Wumpus world.
- A world state is called a situation.
    - Initial state is called S_0
    - Executing action a in S_0 is expressed as:
      Result(a, S_0)
    - Returns a result that is a new situation, S_1.

Actions and Situations

- Actions are logical terms that can have parameters:
  - *Forward*, *Turn*(*Right*)
- *Fluents* are functions and predicates that can vary between situations.
  - Location of the player:
    $At([1,1], S_0)$
  - Not holding the gold at the start of a game:
    $\neg Holding(g, S_0)$

Actions and Situations

- Deciding the resulting situation after a sequence of actions is called the *projection* task.

- Finding a sequence of actions that leads to a situation is called the *planning* task.
    - … which we will soon dig further into.

Describing Actions

- To define an action we need:
  - The action to execute
  - Preconditions (called *possibility axiom*)
    $Preconditions \Rightarrow Poss(a,s)$
  - The result (called *effect axiom*)
    $Poss(a,s) \Rightarrow$ changes resulting from a
  - $Poss(a,s)$ means it is possible to do action a in situation s.
- Examples:
  - $At(player,x,s) \wedge Adjacent(x,x1) \Rightarrow Poss(Go(x,x1),s)$
  - $Gold(g) \wedge At(player,x,s) \wedge At(g,x,s) \Rightarrow Poss(Grab(g),s)$
  - … results in:
  - $Poss(Go(x,x1),s) \Rightarrow At(player,x1, Result(Go(x,y),s))$
  - $Poss(Grab(g),s) \Rightarrow Holding(g, Result(Grab(g),s))$

Problem

- The result:
  - $Poss(Go(x,y),s) \Rightarrow At(player,x1, Result(Go(x,y),s))$
- … states that the fluent position is changed so the x coordinate of the player is updated.
- … and that the new situation is the result from the *Go(x,y)* action in situation *s*.
- It says what has changed, but <u>not</u> what stays the same (y coordinate) in the fluent!
  - Frame problem

Frame problem

- One solution to this is to write rules for how things change (and not change).
- This will however lead to a large number of rules.
- The easiest, and most common way, is to assume that if something is not mentioned in the result it stays that same.

Generalized Events

- A *generalized event* occurs over some time, and can include subevents:
  - *SubEvent*(*BattleOfBritain*, *WorldWarII*)
  - *SubEvent*(*WorldWarII*, *TwentiethCentury*)
- We can also state the length of an event:
  - *Duration*(*Period*(*WorldWarII*)) > *Years*(*5*)
  - … *Period(e)* is the smallest interval enclosing the event *e*.

Generalized Events

- We can also use *In* to state where an event took place:
  - *In*(*Sydney*, *Australia*)
- And *Location(e)* for the smallest place enclosing the event *e*:
  - $\exists w\ w \in \mathit{CivilWars} \land \mathit{SubEvent}(w, \mathit{1640s}) \land \mathit{In}(\mathit{Location}(w), \mathit{England})$
  - … a civil war occured in England in the 1640s.

Intervals

- An interval is the time between start and end of an event.
  - $Interval(i) \Rightarrow Duration(i) = (Time(End(i)) - Time(Start(i)))$
  - $Duration(Minute) = Seconds(60)$
- We can also describe relative times:
  - $Before(i,j) \Leftrightarrow Time(End(i)) < Time(Start(j))$
  - $After(j,i) \Leftrightarrow Before(i,j)$
  - $During(i,j) \Leftrightarrow Time(Start(j)) \leq Time(Start(i)) \wedge Time(End(i)) \leq Time(End(j))$
  - $Overlap(i,j) \Leftrightarrow \exists k\ During(k,i) \wedge During(k,j)$

Now we know how to describe real world problems, let's move into…

PLANNING

Planning

- Planning is the process of finding a sequence of actions to go from a situation s_1 to a new situation s_2.
- In theory, we can search through all possible combinations of actions and find a solution.
- In practice, most real world planning problems are too large…

Planning

- An efficient planner needs to
  - Be able to work forwards or backwards depending on the problem.
  - Have an efficient heuristic to limit the search space.
  - Be able to do *problem decomposition* – divide a problem into subproblems that can be solved in parallel
  - Be able to compose subplans from decomposition to a full plan.

Language

- A planner for full FOL language will be extremely complex.
- Therefore we need a reduced language that:
  - can describe a wide variety of problems.
  - allow the use of efficient planning algorithms.
- The most widespread language is STRIPS, and variations of it.

STRIPS

- States:
  - A state is represented by a conjunction of positive literals:
  - *Rich* ∧ *Famous* can describe a state.
  - … we can also use first-order literals:
  - $At(Plane_1, Melbourne) \wedge At(Plane_2, Sydney)$
  - … but not functions.
  - Closed-world assumption is used. Any conditions not mentioned are assumed to be *false*.

STRIPS

- Goals:
    - … a goal is a specified state, represented as a conjunction of positive ground literals:
    - *Rich* ∧ *Famous*
    - $At(Plane_1,Tahiti)$
    - … a state *s* satisfies a goal *g* if it contains all literals in *g* (and possible others):
    - *Rich* ∧ *Famous* ∧ *Miserable* satisfies the goal *Rich* ∧ *Famous*.

STRIPS

- Actions:
  - Actions are represented with preconditions and effects, for example:

    $$\begin{aligned}&\mathit{Action}(\mathit{Fly}(p,\mathit{from},\mathit{to}),\\ &\quad \text{PRECOND: } \mathit{At}(p,\mathit{from}) \wedge \mathit{Plane}(p) \wedge \\ &\qquad\qquad\quad \mathit{Airport}(\mathit{from}) \wedge \mathit{Airport}(\mathit{to})\\ &\quad \text{EFFECT: } \neg \mathit{At}(p,\mathit{from}) \wedge \mathit{At}(p,\mathit{to}))\end{aligned}$$

  - This is often called an *action schema*.
  - Effect is sometimes divided into *add list* (positive literals) and *delete list* (negative literals).

STRIPS

- An action is *applicable* in any state that satisfies the precondition.
- The state s_2 is a *result* from executing action *a* in state s_1.
  - = same as s_1, but:
  - All positive effects added. If already in s_1, they are ignored.
  - All negative effects removed. If not in s_1, they are ignored.
  - STRIPS assumption: Every literal not mentioned in effect remains unchanged - avoids the frame problem.
- A *solution* is an action sequence leading from start state to goal state.

Example problem: Change a flat tire

```
Init(At(Flat,Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle))
Action(LeaveOvernight,
  PRECOND:
  EFFECT: ¬At(Spare,Ground) ∧ ¬At(Spare,Axle) ∧ ¬At(Spare,Trunk)
        ∧ ¬At(Flat,Ground) ∧ ¬Flat(Axle))
```

Forward State-Space Search

- Also called *progression planning*.

- Start at the initial state.
- See which actions are applicable.
- Each action generates a new state.
- See which actions are applicable in the new states.
- …

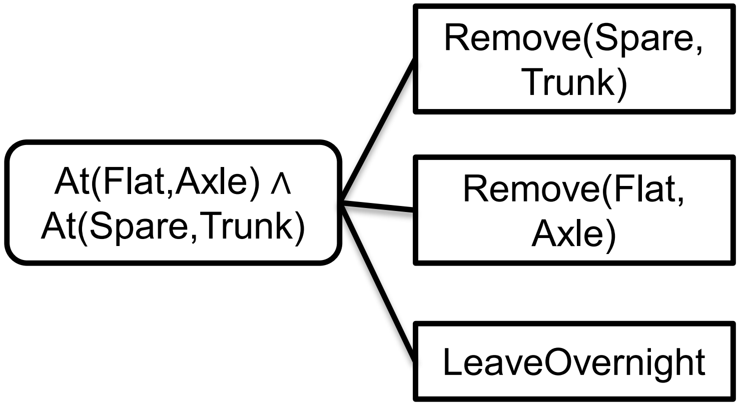
- Literals not mentioned are assumed to be *false*.

Forward State-Space Search

At(Flat,Axle) ∧
At(Spare,Trunk)

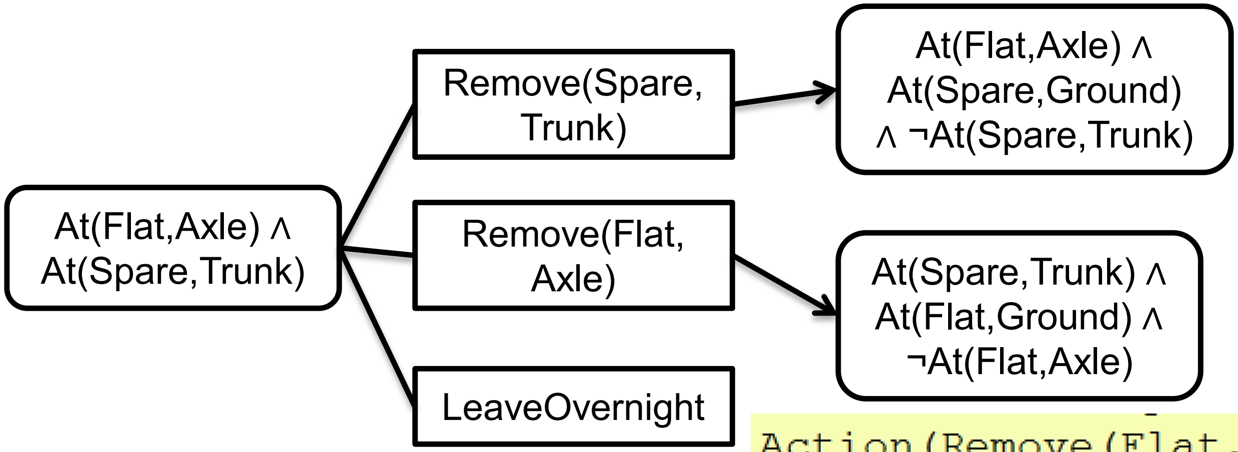
```
Init(At(Flat,Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
Action(LeaveOvernight,
  PRECOND:
  EFFECT: ¬At(Spare,Ground) ∧ ¬At(Spare,Axle) ∧
¬At(Spare,Trunk)
          ∧ ¬At(Flat,Ground) ∧ ¬Flat(Axle))
```

Forward State-Space Search



Forward State-Space Search

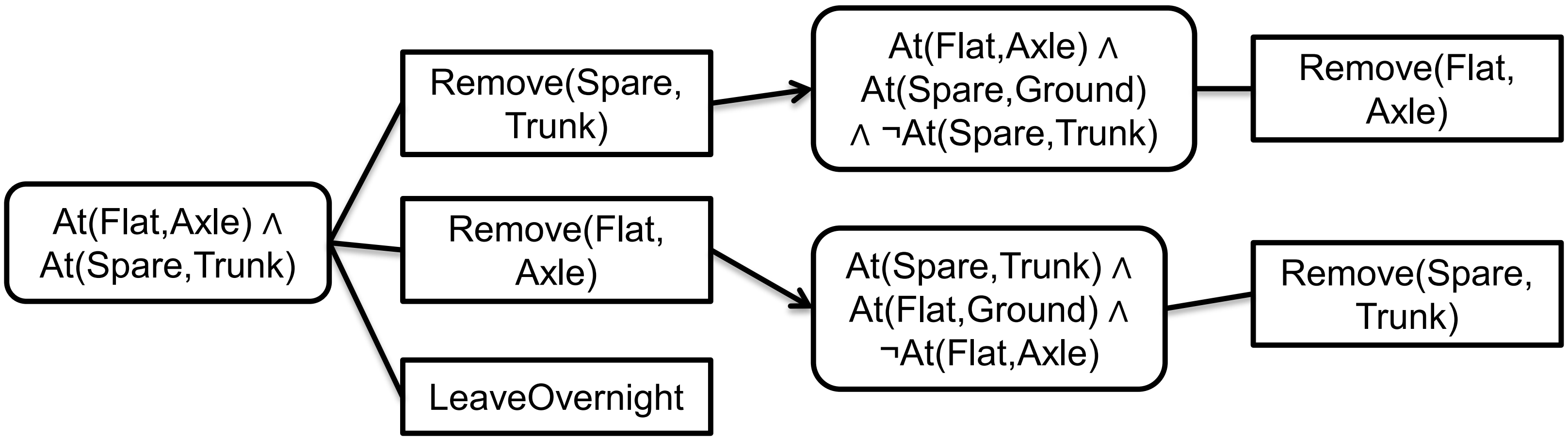
```
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
```



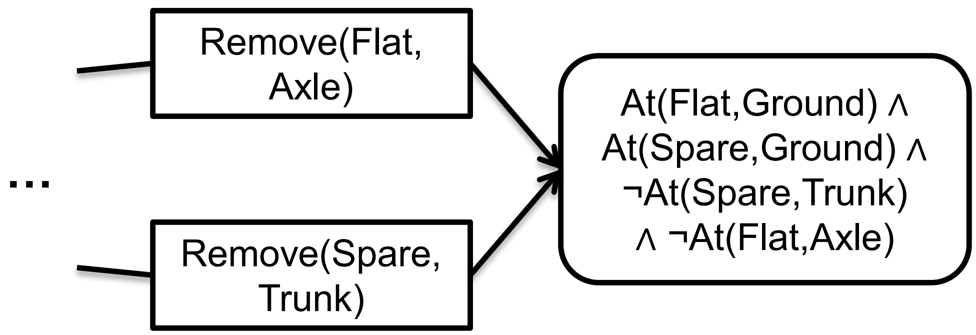

```
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
```

LeaveOvernight is a dead end, since we cannot do any actions after it. All literals are false.

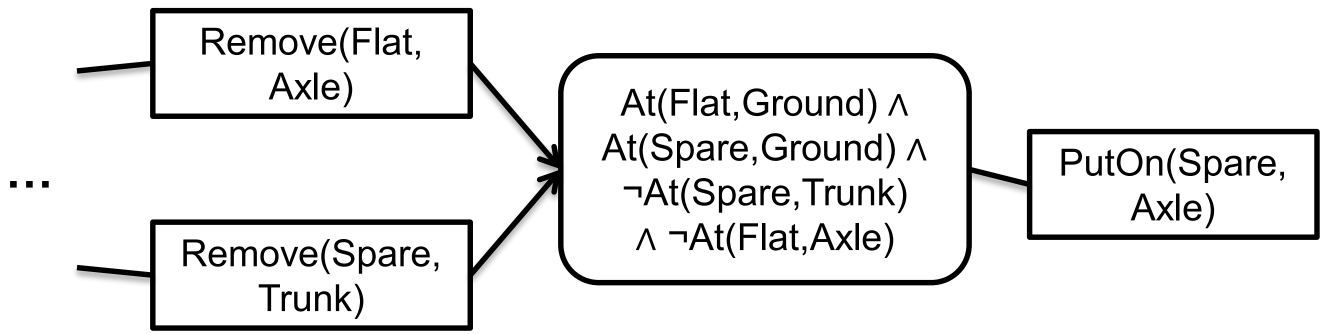
Forward State-Space Search



Forward State-Space Search



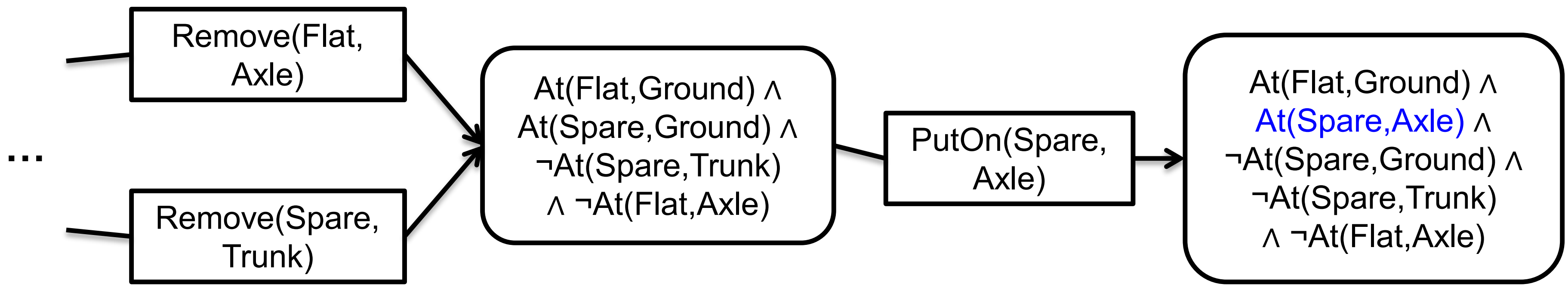
Forward State-Space Search




```
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
```

Forward State-Space Search

`Goal(At(Spare,Axle))`



Satisfies goal
At(Spare,Axle).

Done!

Backward State-Space Search

- Also called *regression planning*.

- Start at the goal state.
- See which actions that lead to the preconditions of the goal state.
- Generate new states from the actions.
- …

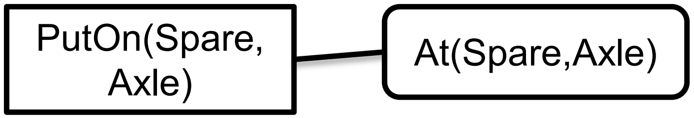
- Literals not mentioned are assumed to be *false*.

Backward State-Space Search

```
Init(At(Flat,Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
Action(LeaveOvernight,
  PRECOND:
  EFFECT: ¬At(Spare,Ground) ∧ ¬At(Spare,Axle) ∧
¬At(Spare,Trunk)
          ∧ ¬At(Flat,Ground) ∧ ¬Flat(Axle))
```

At(Spare,Axle)

Backward State-Space Search

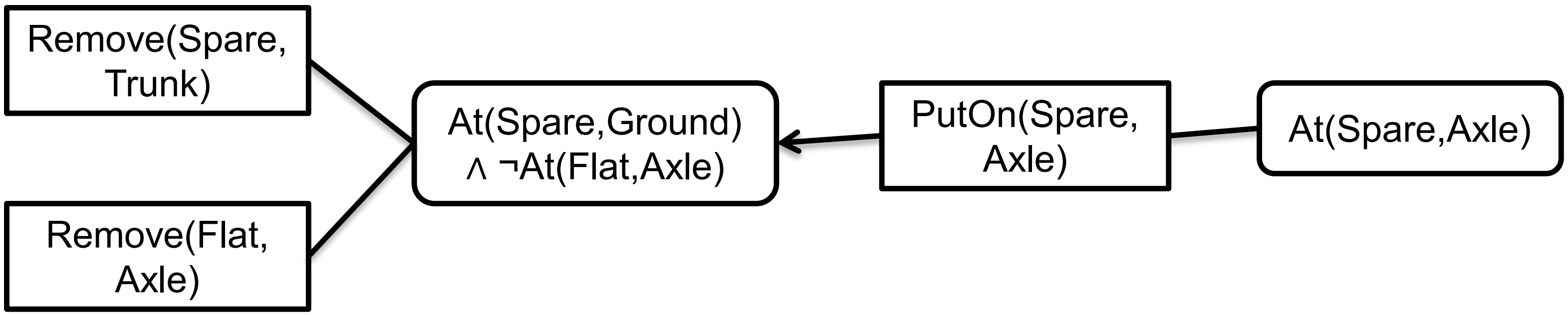



```
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
```

Backward State-Space Search

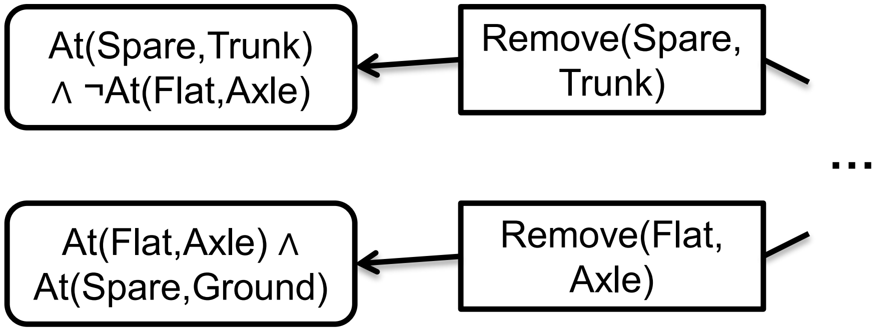

```
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
```

Backward State-Space Search



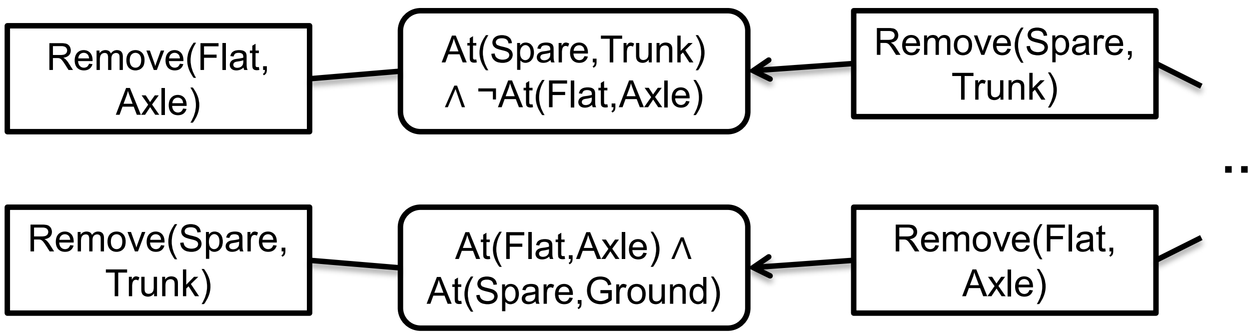

```
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
```

Backward State-Space Search



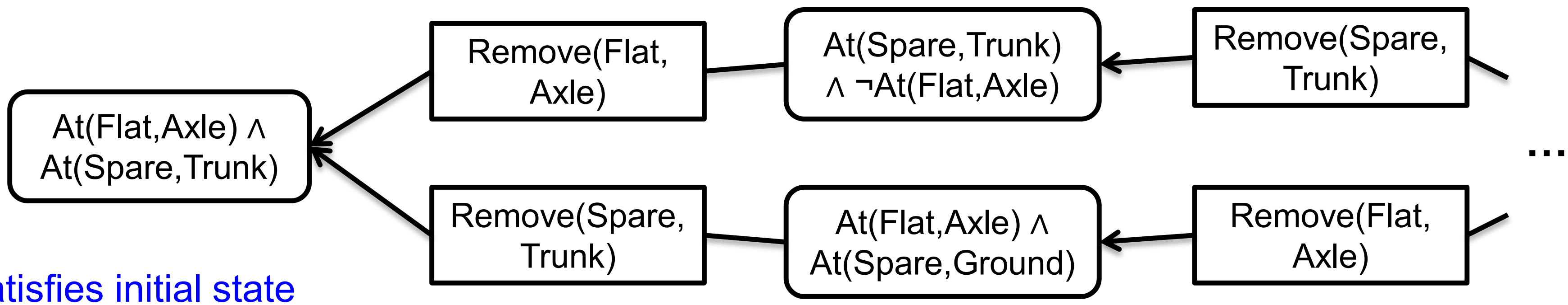

```
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
```

Backward State-Space Search




```
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
```

Backward State-Space Search



Satisfies initial state
At(Flat,Axle) ∧ At(Spare,Trunk).

Done!

Heuristics

- The change tire problem is very simple compared to most real world problems.
- For more complex problems, a heuristic is needed to guide the search.
- One such heuristic is the *relaxed problem* approach.
- It means that we should select the state with least number of positive literals.
- Because it is assumed that the more positive literals a state has, the farther it is from the goal.

Partial-Order Planning

- Forward and Backward State-Space Search are *totally ordered* plan searchers.
- It means that they work in a linear fashion, and cannot take advantage of problem decomposition.
- Partial-Order Planning can do this, by working independently on subgoals to create subplans which can be combined to a full plan.

Partial-Order Planning

- POP requires some more information about a problem.
- Ordering constraints:
  - $A \prec B$ Action A must be executed before B
  - $B \succ A$ Action B must be executed after A
- Causal links:
  - $A \xrightarrow{p} B$ A *achieves* p for B, meaning that A satisfies the precondition p for B. This also means that we are not allowed to add a new action between A and B that is in conflict with the link, i.e. has the effect ¬p.

Partial-Order Planning

- Open preconditions:
  - A precondition is open if it is not solved by some action in the plan. POP works by reducing the number of open preconditions, until all are solved.
- Consistent plan:
  - The goal of the planner is to create a *consistent plan*, which means a plan with no causal link conflicts, no cycles in ordering constraints and no open preconditions in the set.
- Start and Finish state:
  - The planner starts with a *Start* state with the initial state as effect, and a *Finish* state with the goal as precondition.

Let's go back to our example

```
Init(At(Flat,Axle) ∧ At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
Action(LeaveOvernight,
  PRECOND:
  EFFECT: ¬At(Spare,Ground) ∧ ¬At(Spare,Axle) ∧ ¬At(Spare,Trunk)
          ∧ ¬At(Flat,Ground) ∧ ¬Flat(Axle))
```

Partial-Order Planning

Partial-Order Planning

Pick one open precondition
to solve.

At(Spare,Trunk)

Start

At(Flat,Axle)

At(Spare,Axle) Finish

Partial-Order Planning

At(Spare,Trunk) Remove(Spare,Trunk)

Remove(Spare,Trunk) has the open precondition as its pre-condition.

Start

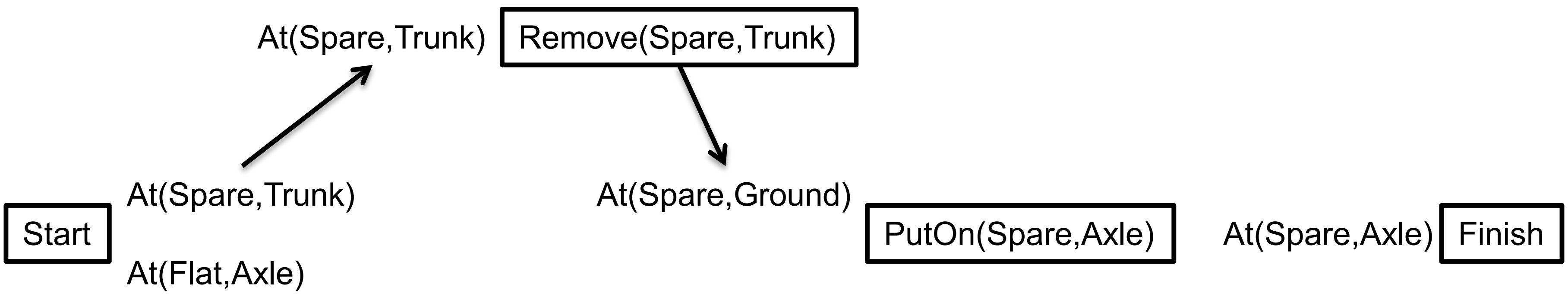
At(Spare,Trunk)

At(Flat,Axle)

At(Spare,Axle) Finish

```
Action(Remove(Spare,Trunk),
  PRECOND: At(Spare,Trunk)
  EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
```

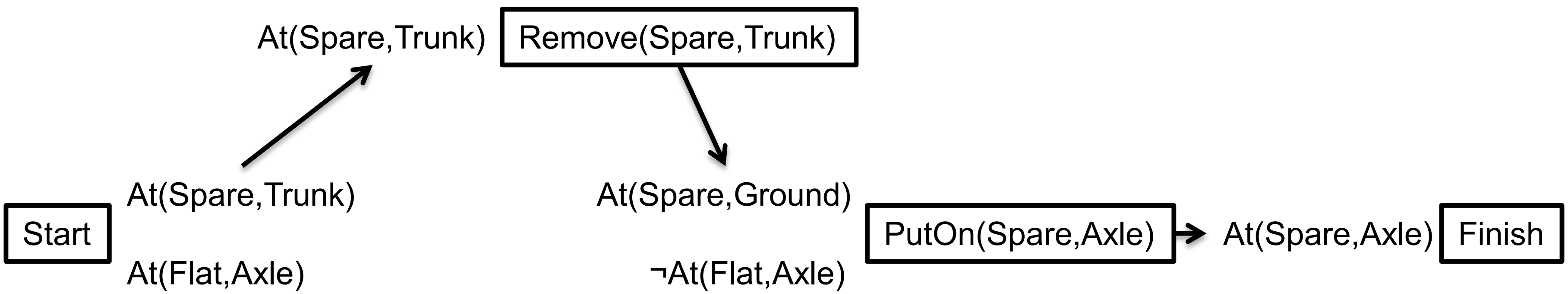
Partial-Order Planning



Remove(Spare,Trunk) has the effect At(Spare,Ground). The only matching action is PutOn(Spare, Axle).

```
Action(PutOn(Spare,Axle),
  PRECOND: At(Spare,Ground) ∧ ¬At(Flat,Axle)
  EFFECT: ¬At(Spare,Ground) ∧ At(Spare,Axle)
```

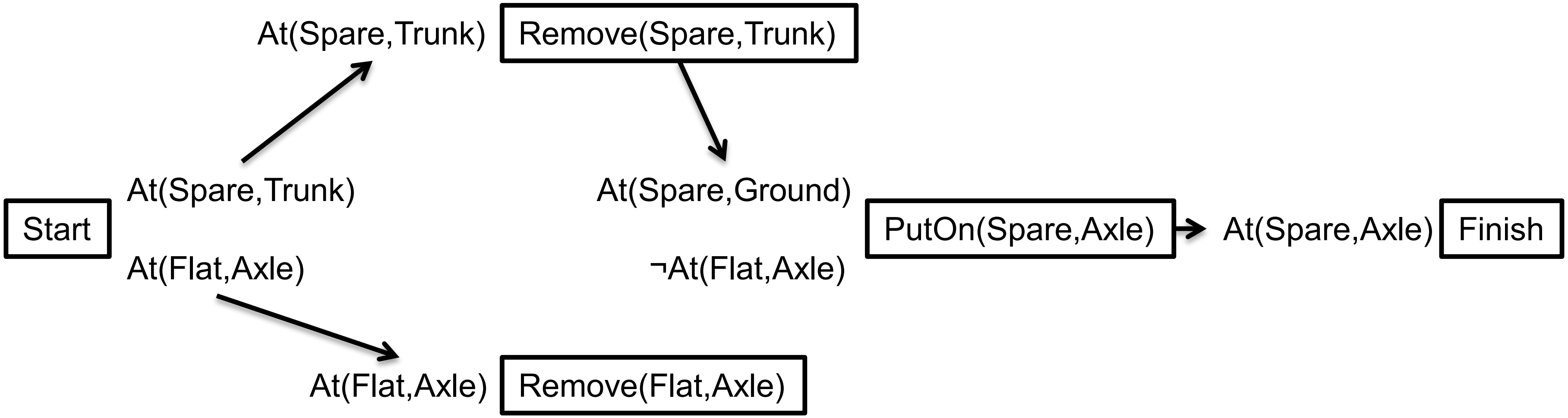
Partial-Order Planning



We have two more open preconditions to deal with.

PutOn(Spare,Axle) also matches the precondition at Finish.

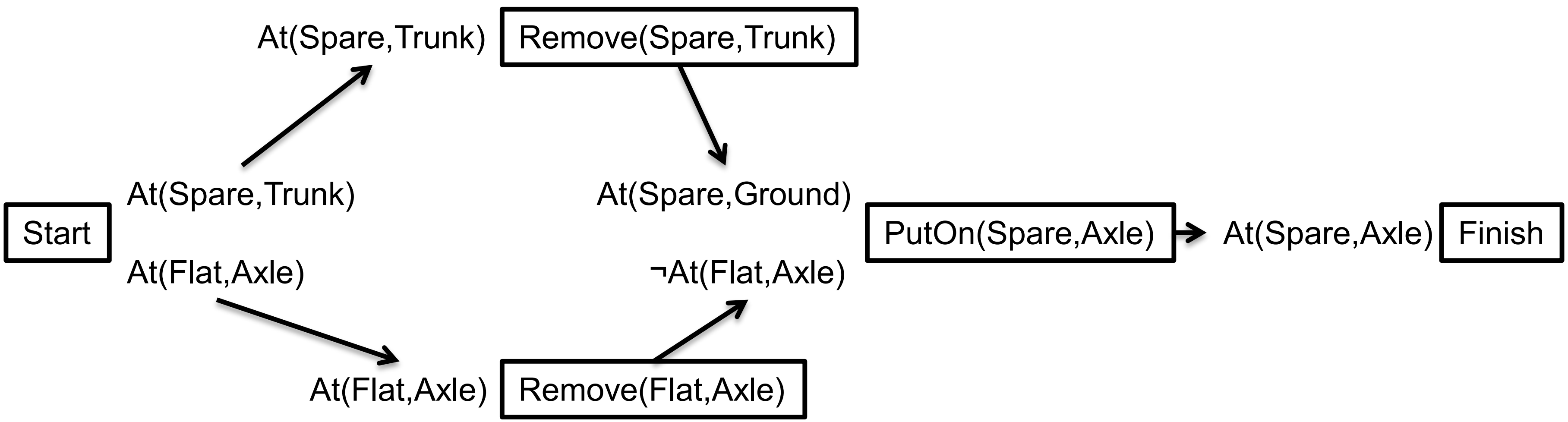
Partial-Order Planning



Remove(Flat,Axle) solves the open precondition at start.

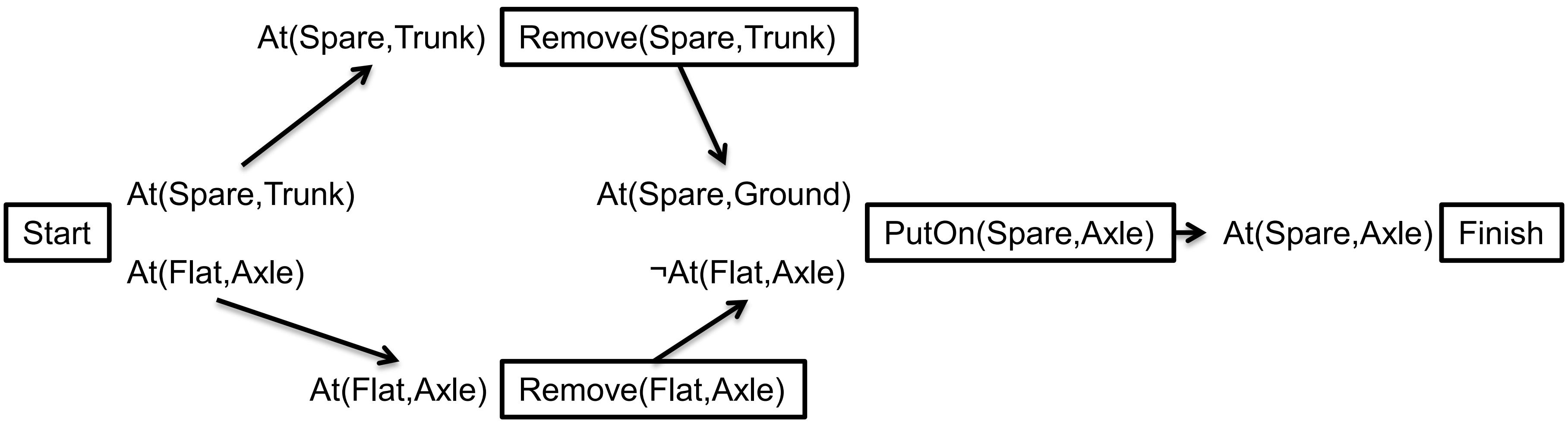
```
Action(Remove(Flat,Axle),
  PRECOND: At(Flat,Axle)
  EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
```

Partial-Order Planning



… and also satisfies the open precondition at PutOn(Spare, Axle).

Partial-Order Planning



Now we have a consistent plan.

Done!

Summary

- There are lots of other planning algorithms:
    - Planning Graphs
    - Graphplan
    - Planning with propositional logic
    - Conditional Planning
    - …
- The ones we have learned about are very common, and should give us an idea about how planners work.

That was all for this lecture

Acknowledgements

Dr. Johan Hagelbäck

Linnæus University

johan.hagelback@lnu.se

http://aiguy.org