Deliberative Agents that Plan

Vasant Honavar
Artificial Intelligence Research Laboratory
Informatics Graduate Program
Computer Science and Engineering Graduate Program
Bioinformatics and Genomics Graduate Program
Neuroscience Graduate Program

Center for Big Data Analytics and Discovery Informatics
Huck Institutes of the Life Sciences
Institute for Cyberscience
Clinical and Translational Sciences Institute
Northeast Big Data Hub
Pennsylvania State University

vhonavar@ist.psu.edu
http://faculty.ist.psu.edu/vhonavar
http://ailab.ist.psu.edu

Outline

- The Planning problem
- Planning with State-space search
- Partial-order planning
- Planning graphs
- Planning with propositional logic
- Analysis of planning approaches

Planning problem

- Classical planning environment: fully observable, deterministic, finite, static and discrete.
- Find a sequence of actions that achieves a given goal when executed from a given initial world state. That is, given
  - a set of action descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,
- compute a plan, which is
  - a sequence of action instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of subgoals to be achieved

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through plan space rather than state space (though first we will talk about state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Goal of Planning

- Choose actions to achieve a certain goal
- But isn't it exactly the same goal as for problem solving?
- Some difficulties with problem solving:
  - The successor function is a black box: it must be "applied" to a state to know which actions are possible in that state and what are the effects of each one

Have(Milk)

- Suppose that the goal is HAVE(MILK).
  - From some initial state where HAVE(MILK) is not satisfied, the successor function must be repeatedly applied to eventually generate a state where HAVE(MILK) is satisfied.
  - An explicit representation of the possible actions and their effects would help the problem solver select the relevant actions
  - Otherwise, in the real world an agent would be overwhelmed by irrelevant actions

Planning vs Problem Solving

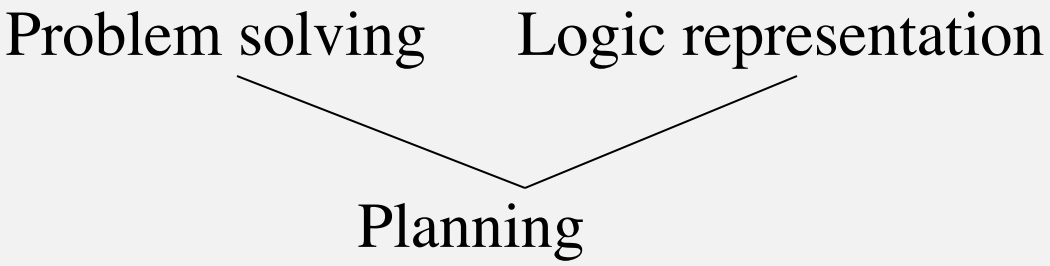
- Another difficulty with problem solving:
  - The goal test is another black-box function, states are domain-specific data structures, and heuristics must be supplied for each new problem
  - Suppose that the goal is HAVE(MILK)∧HAVE(BOOK)
  - Without an explicit representation of the goal, the problem solver cannot know that a state where HAVE(MILK) is already achieved is more promising than a state where neither HAVE(MILK) nor HAVE(BOOK) is achieved

Planning vs Problem Solving

- A third difficulty with problem solving:
  - The goal may consist of several nearly independent subgoals, but there is no way for the problem solver to know it
  - HAVE(MILK) and HAVE(BOOK) may be achieved by two nearly independent sequences of actions

Representations in Planning

- Planning opens up the black-boxes by using logic to represent:
  - Actions
  - States
  - Goals



Major approaches

- Situation calculus
- State space planning
- Partial order planning
- Planning graphs
- Planning with Propositional Logic
- Hierarchical decomposition (HTN planning)
- Reactive planning
- Note: This is an area in which AI is changing quickly.

Situation Calculus Planning

- Formulate planning problem in FOL
- Use theorem prover to find proof (aka plan)

Representing change

- Representing change in the world in logic can be tricky.
- One way is just to change the KB
  - Add and delete sentences from the KB to reflect changes
  - How do we remember the past, or reason about changes?
- Situation calculus is another way
  - A situation is a snapshot of the world at some instant in time
  - When the agent performs an action A in situation S1, the result is a new situation S2.

Situation calculus

- A situation is a snapshot of the world at an interval of time during which nothing changes
- Every true or false statement is made with respect to a particular situation.
  - Add situation variables to every predicate.
  - at(hunter,1,1) becomes at(hunter,1,1,s0): at(hunter,1,1) is true in situation (i.e., state) s0.
- Add a new function, result(a,s), that maps a situation s into a new situation as a result of performing action a. For example, result(forward, s) is a function that returns the successor state (situation) to s
- The action agent-walks-to-location-y could be represented by (∀x)(∀y)(∀s) (at(Agent,x,s) ^ ~onbox(s)) -> at(Agent,y,result(walk(y),s))

Situation calculus planning

- Initial state: a logical sentence about (situation) S0
  - At(Home, S0) ^ ~Have(Milk, S0) ^ ~ Have(Bananas, S0) ^ ~Have(Drill, S0)
- Goal state:
  - (∃s) At(Home,s) ^ Have(Milk,s) ^ Have(Bananas,s) ^ Have(Drill,s)
- Operators are descriptions of actions:
  - ∀(a,s) Have(Milk,Result(a,s)) <=> ((a=Buy(Milk) ^ At(Grocery,s)) V (Have(Milk, s) ^ a~=Drop(Milk)))
- Result(a,s) names the situation resulting from executing action a in situation s.
- Action sequences are also useful: Result'(l,s) is the result of executing the list of actions (l) starting in s:
  - (∀s) Result'([],s) = s
  - (∀a,p,s) Result'([a|p]s) = Result'(p,Result(a,s))

Situation calculus planning II

- A solution is thus a plan that when applied to the initial state yields a situation satisfying the goal query:
  - At(Home,Result'(p,S0))
  - ^ Have(Milk,Result'(p,S0))
  - ^ Have(Bananas,Result'(p,S0))
  - ^ Have(Drill,Result'(p,S0))
- Thus we would expect a plan (i.e., variable assignment through unification) such as:
  - p = [Go(Grocery), Buy(Milk), Buy(Bananas), Go(HardwareStore), Buy(Drill), Go(Home)]

Planning with propositional logic: SATPLAN

- Planning can be done by proving theorem in situation calculus
- Test the satisfiability of a logical sentence:
  - *initial state* $\wedge$ *all possible action descriptions* $\wedge$ *goal*
- Sentence contains propositions for every action occurrence.
  - A model will assign true to the actions that are part of the correct plan and false to the others
  - An assignment that corresponds to an incorrect plan will not be a model because of inconsistency with the assertion that the goal is true.
  - If the planning is unsolvable the sentence will be unsatisfiable

SC planning: analysis

- This is fine in theory, but remember that problem solving (search) is exponential in the worst case
- Also, resolution theorem proving only finds a proof (plan), not necessarily a good plan
- Another important issue: the Frame Problem

The Frame Problem

- In SC, need not only axioms to describe what changes in each situation, but also need axioms to describe what stays the same (can do this using successor-state axioms)
- Qualification problem: difficulty in specifying all the conditions that must hold in order for an action to work
- Ramification problem: difficulty in specifying all of the effects that will hold after an action is taken

So…

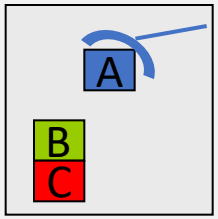
- we restrict the language and use a special-purpose algorithm (a planner) rather than general theorem prover

Planning language

- What is a good language?
- Must represent
  - States
  - Goals
  - Action.
- Must be
  - Expressive enough to describe a wide variety of problems.
  - Restrictive enough to allow efficient algorithms to operate.
- STRIPS (Stanford Research Institute Problem Solver), ADL…

Representing States

World states are represented as sets of facts.
We will also refer to facts as propositions.

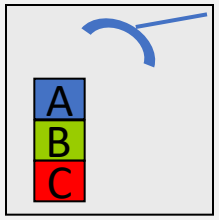


holding(A)
clear(B)
on(B,C)
onTable(C)

State 1

handEmpty
clear(A)
on(A,B)
on(B,C)
onTable(C)

State 2



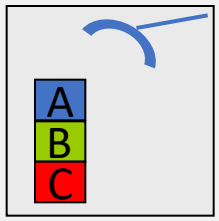
Closed World Assumption (CWA):

Fact not listed in a state are assumed to be false. Under CWA we are assuming the agent has full observability.

Representing Goals

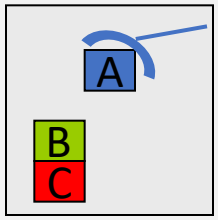
Goals are also represented as sets of facts.
For example { on(A,B) } is a goal in the blocks world.

A **goal state** is any state that contains all the goal facts.



handEmpty
clear(A)
on(A,B)
on(B,C)
onTable(C)

State 1

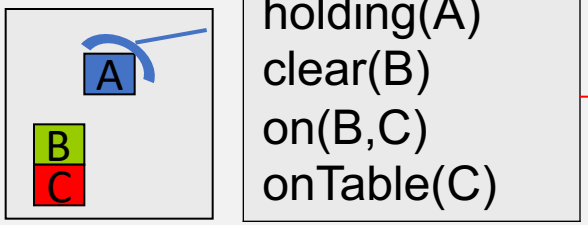


holding(A)
clear(B)
on(B,C)
onTable(C)

State 2

State 1 is a goal state for the goal { on(A,B) }.
State 2 is not a goal state for the goal { on(A,B) }.

Representing Action in STRIPS



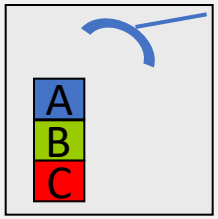
holding(A)
clear(B)
on(B,C)
onTable(C)

State 1

PutDown(A,B)

handEmpty
clear(A)
on(A,B)
on(B,C)
onTable(C)

State 2



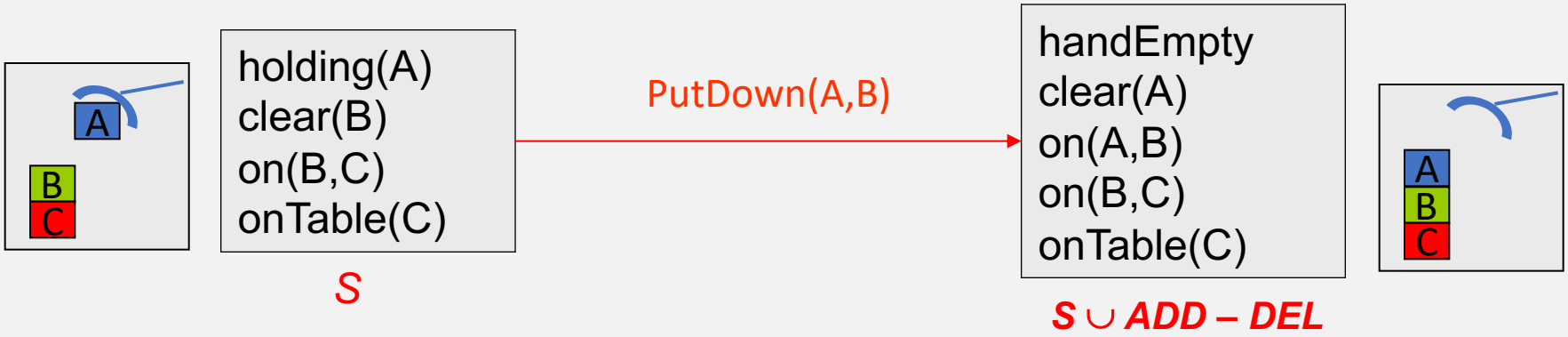
A STRIPS action definition specifies:

1) a set PRE of preconditions facts
2) a set ADD of add effect facts
3) a set DEL of delete effect facts

PutDown(A,B):

PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A) }
DEL: { holding(A), clear(B) }

Semantics of STRIPS Actions



- A STRIPS action is **applicable** (or allowed) in a state when its preconditions are contained in the state.
- Taking an action in a state ***S*** results in a new state ***S ∪ ADD – DEL*** (i.e. add the add effects and remove the delete effects)

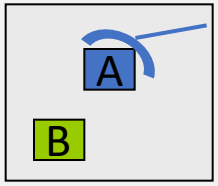
PutDown(A,B):

PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A)}
DEL: { holding(A), clear(B) }

A **STRIPS planning problem** specifies:
1) an initial state *S*
2) a goal *G*
3) a set of STRIPS actions

Objective: find a "short" action sequence reaching a goal state, or report that the goal is unachievable

Example Problem:



holding(A)
clear(B)
onTable(B)

Initial State

on(A,B)

Goal

PutDown(A,B):
PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A)}
DEL: { holding(A), clear(B) }

PutDown(B,A):
PRE: { holding(B), clear(A) }
ADD: { on(B,A), handEmpty, clear(B) }
DEL: { holding(B), clear(A) }

STRIPS Actions

Solution: (PutDown(A,B))

STRIPS Action Schemas

For convenience we typically specify problems via action schemas rather than writing out individual STRIPS actions.

Action Schema: (x and y are variables)

PutDown(x,y):
PRE: { holding(x), clear(y) }
ADD: { on(x,y), handEmpty, clear(x) }
DEL: { holding(x), clear(y) }

PutDown(B,A):
PRE: { holding(B), clear(A) }
ADD: { on(B,A), handEmpty, clear(B) }
DEL: { holding(B), clear(A) }

. . . .

PutDown(A,B):
PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A) }
DEL: { holding(A), clear(B) }

- Each way of replacing variables with objects from the initial state and goal yields a "ground" STRIPS action.
- Given a set of schemas, an initial state, and a goal, propositional planners compile schemas into ground actions and then ignore the existence of objects thereafter.

STRIPS Versus PDDL

- Your book refers to the PDDL language for defining planning problems rather than STRIPS
- The Planning Domain Description Language (PDDL) was defined by planning researchers as a standard language for defining planning problems
  - Includes STRIPS as special case along with more advanced features
  - Some simple additional features include: type specification for objects, negated preconditions, conditional add/del effects
  - Some more advanced features include allowing numeric variables and durative actions
- Most planners you can download take PDDL as input
  - Majority only support the simple PDDL features (essentially STRIPS)
  - PDDL syntax is easy to learn from examples packaged with planners, but a definition of the STRIPS fragment can be found at:

  http://eecs.oregonstate.edu/ipc-learn/documents/strips-pddl-subset.pdf

Properties of Planners

- A planner is **sound** if any action sequence it returns is a true solution
- A planner is **complete** if it outputs an action sequence or "no solution" for any input problem
- A planner is **optimal** if it always returns the shortest possible solution

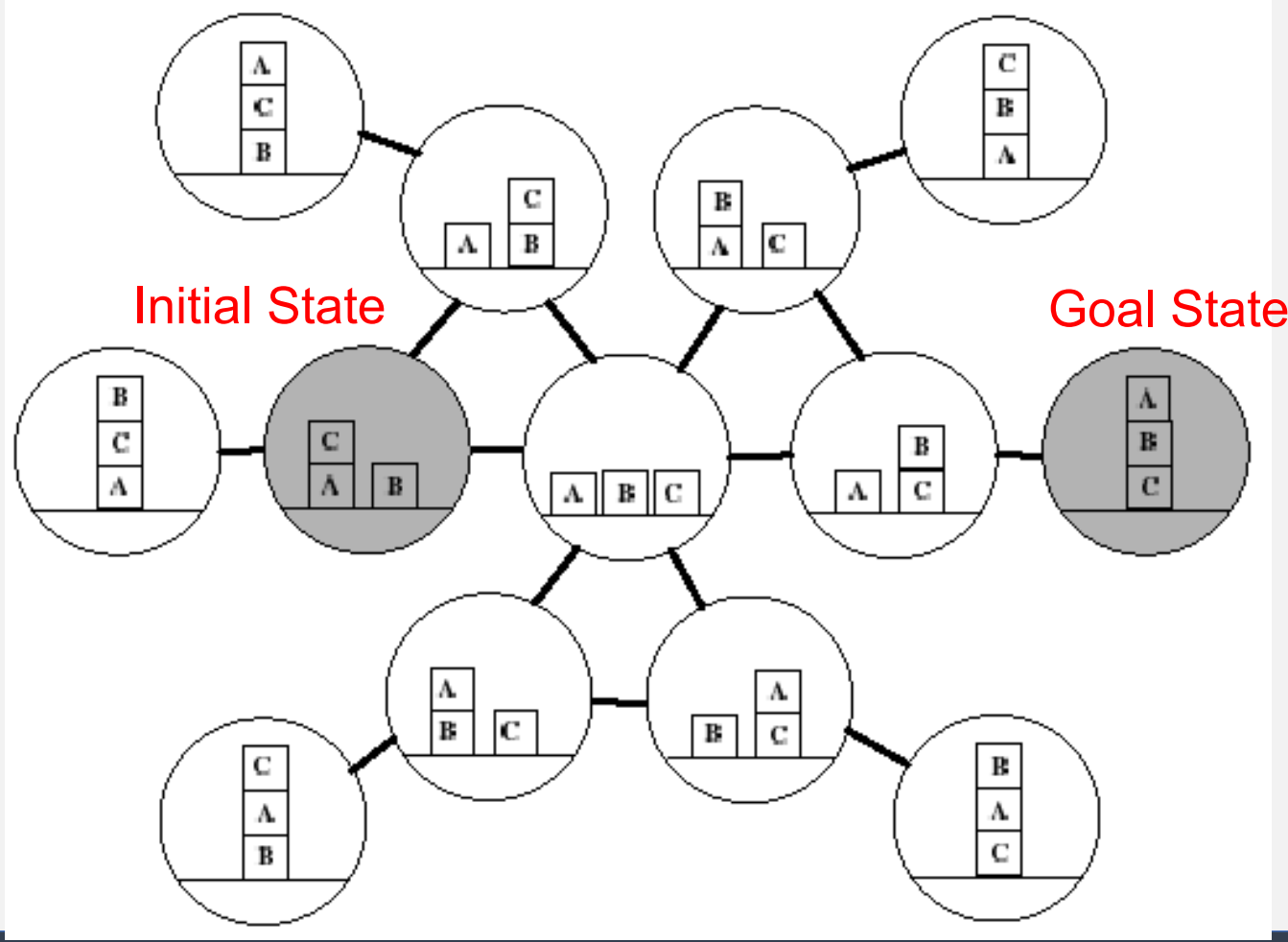
Is optimality an important requirement?
Is it a reasonable requirement?

Planning as Graph Search

- It is easy to view planning as a graph search problem
- Nodes/vertices = possible states
- Directed Arcs = STRIPS actions
- Solution: path from the initial state (i.e. vertex) to one state/vertices that satisfies the goal

Search Space: Blocks World

Graph is finite



Planning as Graph Search

- Planning is just finding a path in a graph
  - Why not just use standard graph algorithms for finding paths?
- **Answer:** graphs are exponentially large in the problem encoding size (i.e. size of STRIPS problems).
  - But, standard algorithms are poly-time in graph size
  - So standard algorithms would require exponential time

- Can we do better than this?

Satisficing vs. Optimality

- While finding a plan is hard in the worst case, for many planning domains, finding a plan is easy.
- However finding optimal solutions can still be hard in those domains.
  - For example, optimal planning in the blocks world is NP-complete.
- In practice it is often sufficient to find “good” solutions “quickly” although they may not be optimal.
  - This is often referred to as the “satisficing” objective.

Satisficing

- Still finding satisficing plans for arbitrary STRIPS problems is not easy.
  - Must still deal with the exponential size of the underlying state spaces
- Why might we be able to do better than generic graph algorithms?
- **Answer:** we have the compact and structured STRIPS description of problems
  - Try to leverage structure in these descriptions to intelligently search for solutions
- We will now consider several frameworks for doing this

STRIPS General language features

- Representation of states
  - Decompose world in logical conditions and represent state as conjunction of positive literals.
    - Propositional literals: Poor ∧ Unknown
    - FO-literals (grounded and function-free):
      - at(plane1, phl) ∧ at(plane2, bwi)
      - does *not* allow
        - at(X,Y), (not grounded)
        - at(father(fred),bwi), (function)
        - ¬at(plane1, phl. (negative literal)

STRIPS

- Closed world assumption
- Representation of goals
  - Partially specified state and represented as a conjunction of positive ground literals
  - A goal is satisfied if the state contains all literals in goal.
  - e.g.: at(paula, phl) ∧ at(john,bwi) satisfies the goal at(paula, phl)

General language features

- Representations of actions
  - Action = PRECOND + EFFECT
    - Action(Fly(p,from, to),
    - PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    - EFFECT: ¬AT(p,from) ∧ At(p,to))
  - = action schema (p, from, to need to be instantiated)
    - Action name and parameter list
    - Precondition (conj. of function-free literals)
    - Effect (conj of function-free literals and P is True and not P is false)
  - May split Add-list and delete-list in Effect

Example

- Paula flies from Philadelphia to Baltimore
  - Action(Fly(p,from,to)
  - PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
  - EFFECT: ¬At(p,from) ∧ At(p,to))
- We begin with
  - At(paula,phl) ∧ Plane(phl) ∧ Airport(phl) ∧ Airport(bwi)
- We take the action
  - Fly(paula, phl, bwi)
- We end with
  - ←At(paula,phl) ∧ At(paula, bwi)
- Note that we haven't said anything in the effect about what happened to the plane. Do we care?

Language semantics?

- How do actions affect states?
  - An action is applicable in any state that satisfies the precondition.
  - For FO action schema applicability involves a substitution θ for the variables in the PRECOND.
    - At(P1,JFK) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)
    - Satisfies : At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    - With θ ={p/P1,from/JFK,to/SFO}
    - Thus the action is applicable.

Language semantics?

- The result of executing action a in state s is the state s'
  - s' is same as s except
    - Any positive literal P in the effect of a is added to s'
    - Any negative literal ¬P is removed from s'
    - EFFECT: ¬AT(p,from) ∧ At(p,to):
    - At(P1,SFO) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)
  - STRIPS assumption: (avoids representational frame problem)
    - every literal NOT in the effect remains unchanged

Languages for Planning Problems

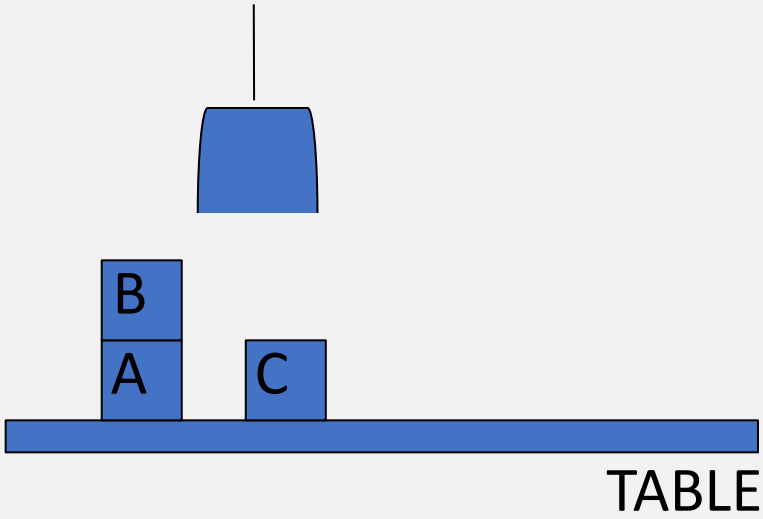
- STRIPS
  - Stanford Research Institute Problem Solver
  - Historically important
- ADL
  - Action Description Languages
  - Relaxed some of the restrictions that made STRIPS inadequate for real-world problems
- PDDL
  - Planning Domain Definition Language
  - Revised & enhanced for the needs of the International Planning Competition
  - Includes STRIPS and ADL

Planning Languages

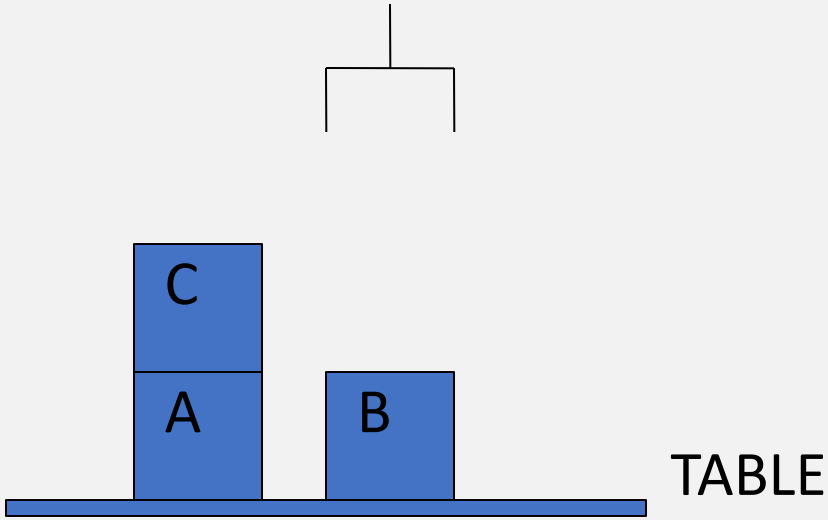
- STRIPS is simplest
  - Important limit: function-free literals
    - Allows for propositional representation
    - Function symbols lead to infinitely many states and actions
    - But poor expressivity
- Extension:Action Description language (ADL)
  - Allows negative literals
  - Allows quantified variables, conjunctions, disjunctions in goals
  - Open World assumption
    - Action(Fly(p:Plane, from: Airport, to: Airport),
    - PRECOND: At(p,from) $\wedge$ (from $\neq$ to)
    - EFFECT: $\neg$At(p,from) $\wedge$ At(p,to))

Blocks world

- The blocks world is a micro-world that consists of a table, a set of blocks and a robot hand.
- Some domain constraints:
  - Only one block can be on another block
  - Any number of blocks can be on the table
  - The hand can only hold one block
- Typical representation:
  - ontable(a)
  - ontable(c)
  - on(b,a)
  - handempty
  - clear(b)
  - clear(c)

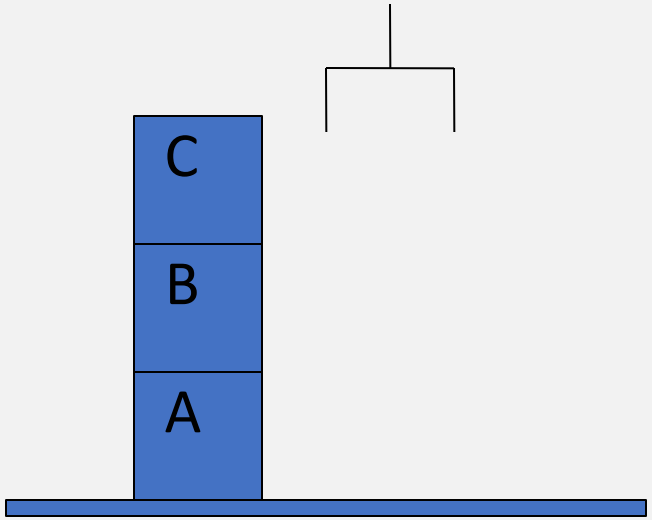


State Representation



Conjunction of propositions:
BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Goal Representation



Conjunction of propositions:
ON(A,TABLE), ON(B,A), ON(C,B)

The goal G is achieved in a state S if all the propositions in G are also in S

Action Representation

Unstack(x,y)

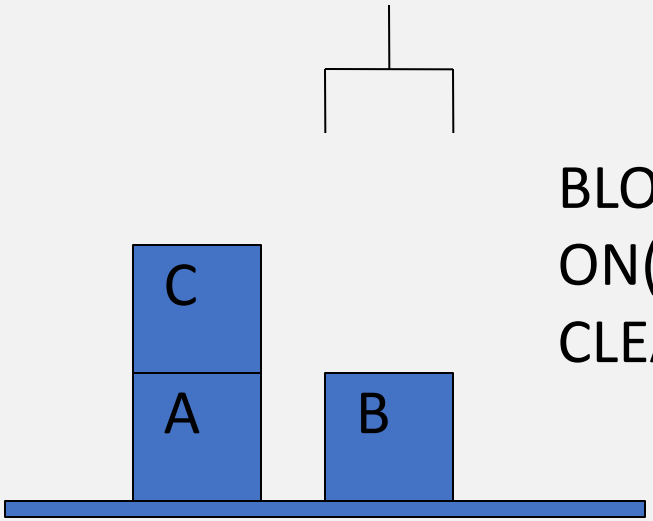
P = HANDEMPTY, BLOCK(x), BLOCK(y),
CLEAR(x), ON(x,y)

E = ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x),
¬ ON(x,y), CLEAR(y)

Effect: list of literals

Precondition: conjunction of propositions

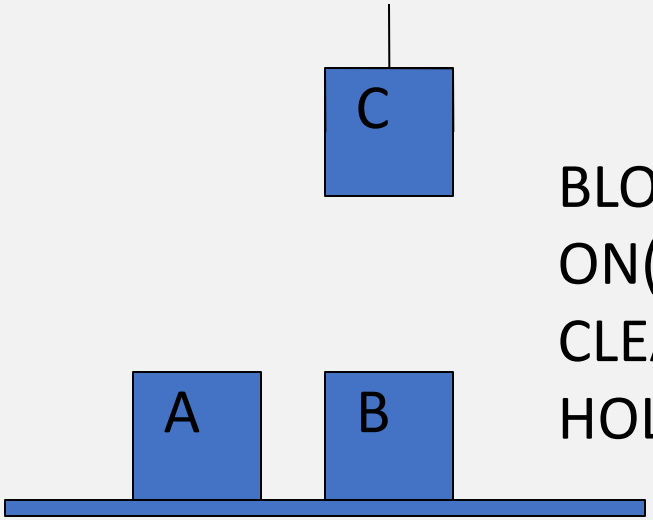
Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Unstack(C,A)
P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)

Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), ~~CLEAR(C), HANDEMPTY,~~
HOLDING(C), CLEAR(A)

Unstack(C,A)
P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)

Action Representation

Action(Unstack(x,y)
P: HANDEMPTY, BLOCK(x), BLOCK(y), CLEAR(x), ON(x,y)
E: ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ ON(x,y), CLEAR(y)

Action(Stack(x,y)
P: HOLDING(x), BLOCK(x), BLOCK(y), CLEAR(y)
E: ON(x,y), ¬CLEAR(y), ¬HOLDING(x), CLEAR(x), HANDEMPTY

Actions

Action(Pickup(x)
P: HANDEMPTY, BLOCK(x), CLEAR(x), ON(x,TABLE)
E: ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ON(x,TABLE)

Action(PutDown(x)
P: HOLDING(x)
E: ON(x,TABLE), ¬HOLDING(x), CLEAR(x), HANDEMPTY

Example: Spare tire problem

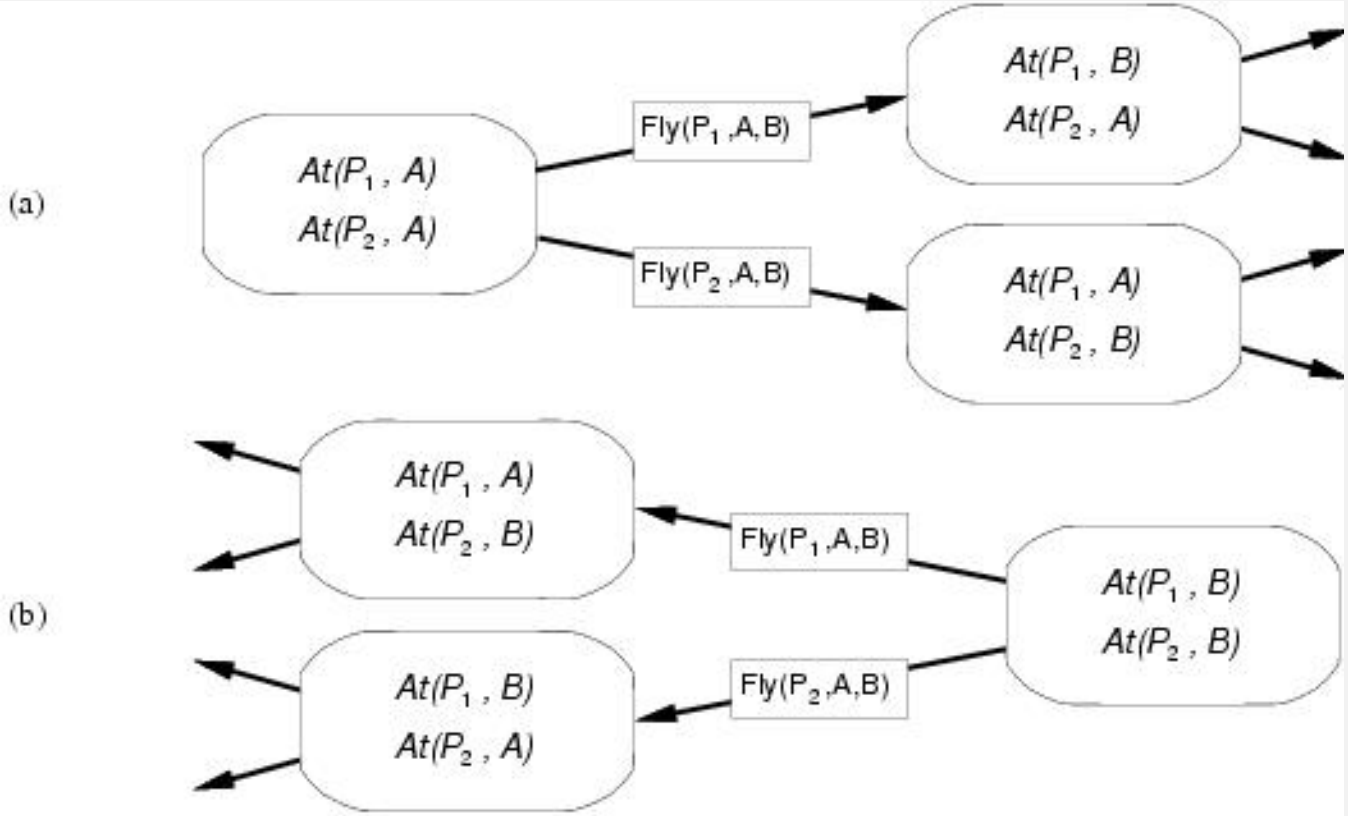
Init(At(Flat, Axle) ∧ At(Spare,trunk))
Goal(At(Spare,Axle))
Action(Remove(Spare,Trunk)
PRECOND: At(Spare,Trunk)
EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))
Action(Remove(Flat,Axle)
PRECOND: At(Flat,Axle)
EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))
Action(PutOn(Spare,Axle)
PRECOND: At(Spare,Groundp) ∧¬At(Flat,Axle)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

This example is ADL: negative literal in pre-condition

Planning with state-space search

- Search the space of states
- Progression planners
  - forward state-space search
  - Consider the effect of all possible actions in a given state
- Regression planners
  - backward state-space search
  - To achieve a goal, what must have been true in the previous state.

Progression and regression



State-Space Formulation

- Formulation as state-space search problem:
  - Initial state = initial state of the planning problem
    - Literals not appearing are false
  - Actions = those whose preconditions are satisfied
    - Add positive effects, delete negative
  - Goal test = does the state satisfy the goal?
  - Step cost = each action costs 1
  - Solution is a sequence of actions.

Progression Algorithm

- No functions, so the number of states is finite … any graph search that is complete is a complete planning algorithm.
  - E.g. A*
- Inefficient:
  - (1) irrelevant action problem
  - (2) good heuristic required for efficient search

Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?
- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.

Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors is as follows:
    - Any positive effects of A that appear in G are deleted.
    - Each precondition literal of A is added , unless it already appears.
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.

Heuristics for state-space search

- Neither progression or regression are very efficient without a good heuristic.
  - How many actions are needed to achieve the goal?
  - Exact solution is NP hard, find a good estimate
- Two approaches to find admissible heuristic:
  - The optimal solution to the relaxed problem.
    - Remove all preconditions from actions
  - The subgoal independence assumption:
    - The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.

Partial Order Planning

Search through State Space

The simplest way to build a planner is to cast the planning problem as search through the space of world states. Each node in the graph denotes a state of the world, and arcs connect worlds that can be reached by executing a single action. We would call it a **situation space** planner because it searches through the space of possible situations.

There are two types of planners:

Progression Planner: it searches forward from the initial situation to the goal situation.

Regression Planner: it searches backward from the goal situation to the initial situation.

Search through the Space of Plans

An alternative is to search through space of plans rather than space of situations i.e plan-space node denote plans

- Start with a simple partial plan
- Expand the plan until the complete plan is developed
- Operators in this step:
  - Adding a step
  - Imposing an ordering that puts one step after another
  - Instantiating a previously unbound variable
- The solution is the final plan

Representation of Plans

Consider a simple problem: Putting on a pair of shoes

Goal →RightShoeOn ^ LeftShoeOn

Four operators:

Op(Action:RightShoe, PreCond:RightSockOn, Effect:RightShoeON)

Op(Action:RightSock , Effect: RightSockOn)

Op(Action:LeftShoe, Precond:LeftSockOn, Effect:LeftShoeOn)

Op(Action:LeftSock,Effect:LeftSockOn)

Least Commitment –

One should make choices only about things that you currently care about ,leaving the others to be worked out later.

Partial Order Planner –

A planner that can represent plans in which some steps are ordered (before or after) w.r.t each other and other steps are unordered.

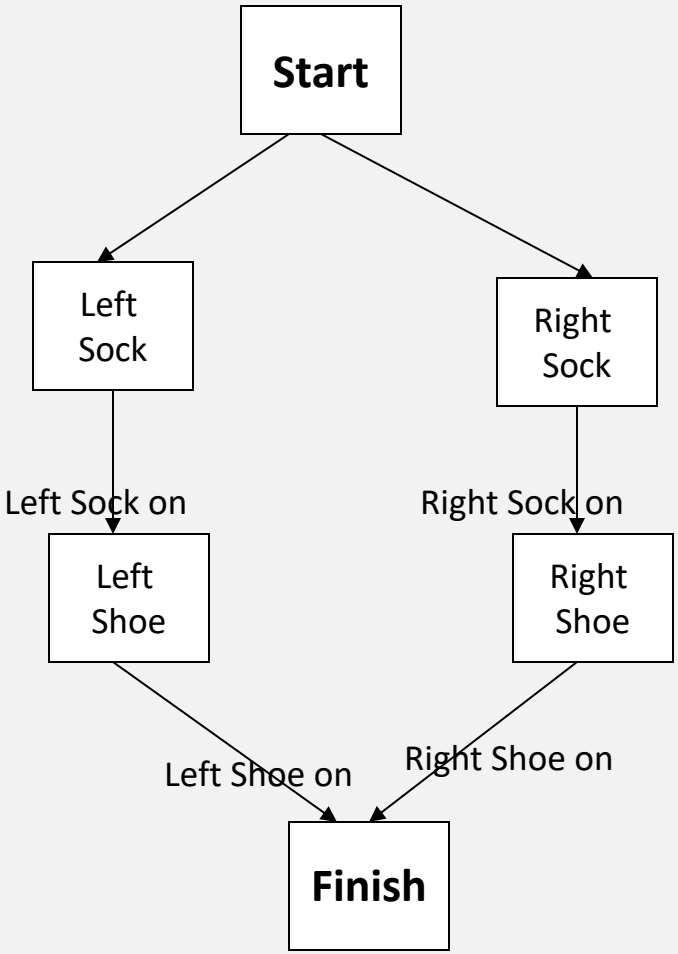
Total Order Planner—

Planner in which plans consist of a simple lists of steps

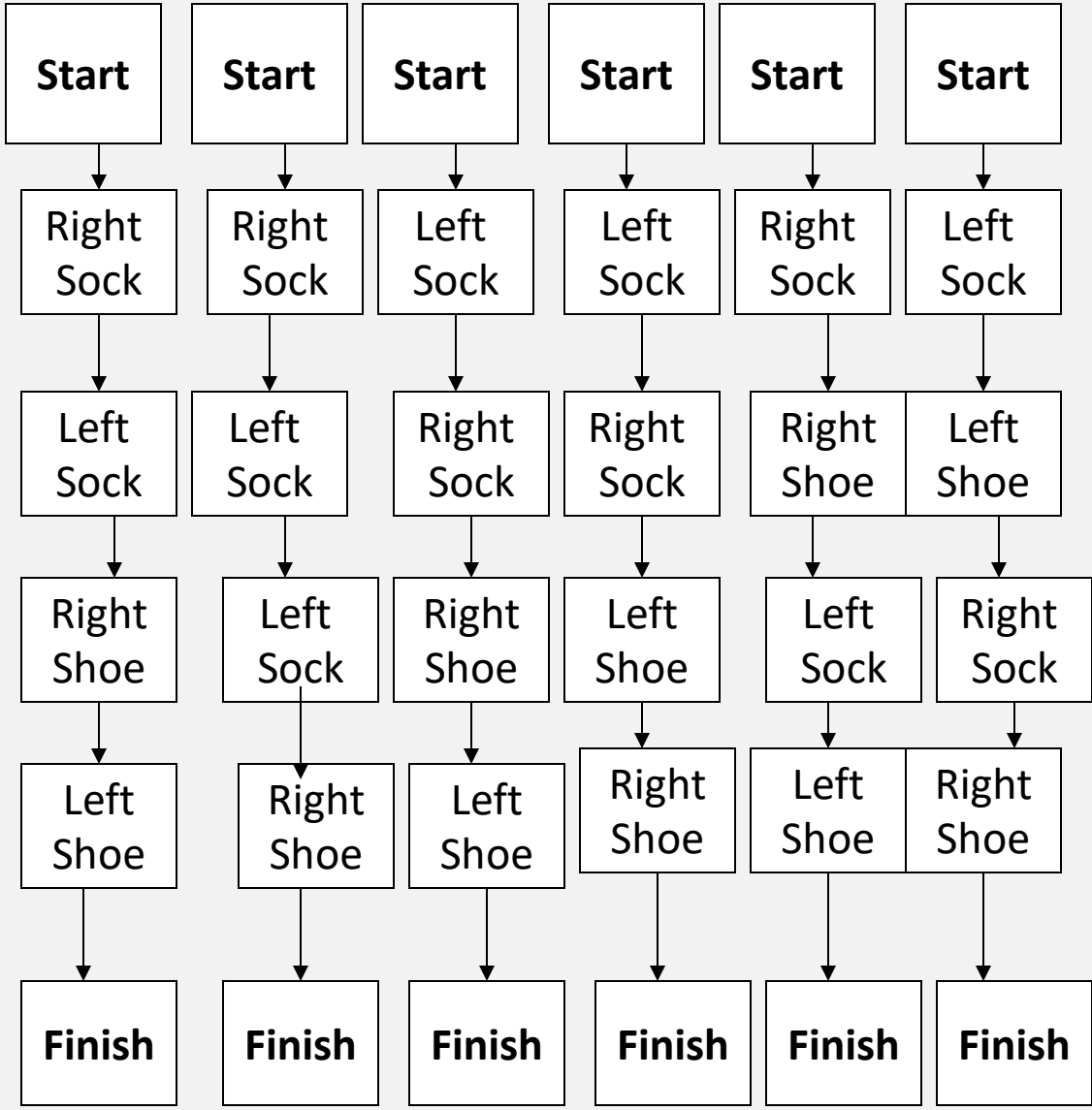
Linearization of P—

A totally ordered plan that is derived from a plan P by adding constraints

Partial Order Plans:



Total Order Plans:



Plans, Causal Links and Threats

Representing a Plan as a three tuple :-- <A,O,L>

- A → Set of Actions
- L → Set of Causal Links
- O → Set of Ordering constraints over A

If A ={A1,A2,A3} then O might be set{A1<A3,A2<A3}

These constraints specify a plan in which A3 is necessarily the last action but does not commit to a choice of which of the three actions comes first

As least commitment planners refine their plans, they must do constraint satisfaction to ensure consistency of O

Causal Links—

A Causal Links is written as $S_i \xrightarrow{c} S_j$ and read as " S_i achieves c for S_j ".Causal Links serve to record the purpose of steps in the plan:here a purpose of S_i is to achieve the precondition c of S_j

Threat –

Causal Links are used to detect when a newly introduced action interferes with past decision. Such an action is called a Threat.

Let At be a different action in A : we sat that A t *threatens* $A_p \xrightarrow{Q} A_c$ when the two criteria are met:

- O U$\{A_p<A_t<A_c\}$ is consistent
- A_t has ¬Q as an effect.

Resolving Threats

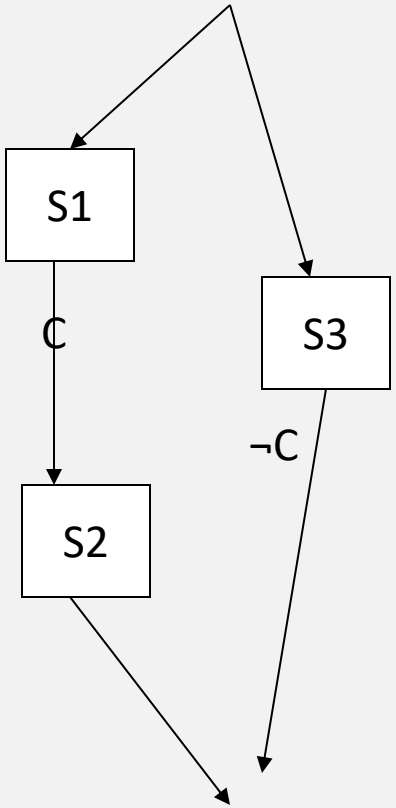
When a plan contains a threat , then there is a danger that the plan won't work as anticipated i.e. We have reached a dead-end in the search.

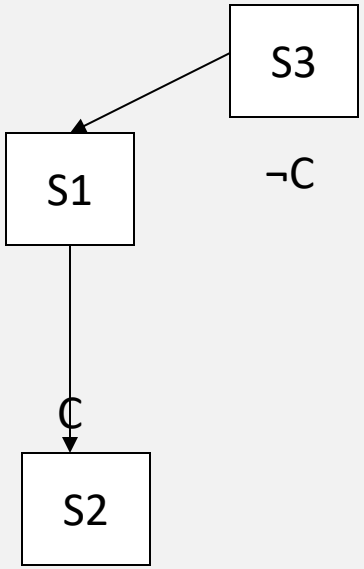
The causal link S1 $\xrightarrow{c}$ S2 is threatened by a new step S3 because one effect of S3 is to delete c.

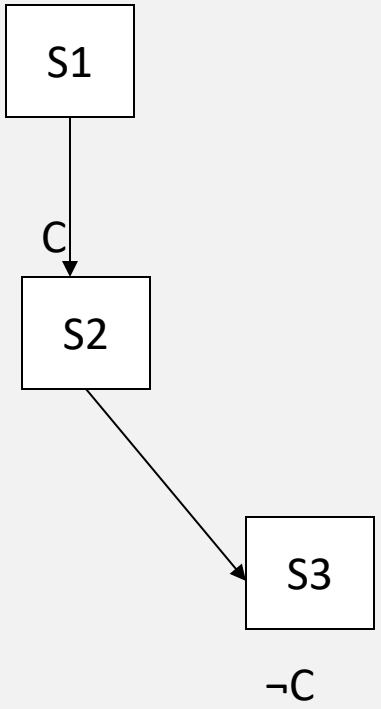
The way to resolve the Threat is to add ordering constraints to make sure that S3 does not intervene between S1 and S2

Demotion -- S3 is placed before S1 .

Promotion – S3 is placed after S2.


S1
C
S2
S3
¬C


S3
S1
¬C
C
S2


S1
C
S2
S3
¬C

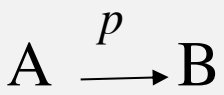
Partial Order Planning

- Plan from goals, back to initial state
- Search through partial plans

- Representation:
  - Operators given in declarative representation, rather than black box functions.
  - Plans represent only relevant commitments (e.g., relevant ordering of operators, not total ordering)

POP as a search problem

- States are (mostly unfinished) plans.
  - The empty plan contains only start and finish actions.
- Each plan has 4 components:
  - A set of actions (steps of the plan)
  - A set of ordering constraints: A $\prec$ B (A before B)
    - Cycles represent contradictions. A $\prec$ B and B $\prec$ A
  - A set of causal links between actions
    - A achieves *p* for B
    - Can't add an action C that conflicts with the causal link. (if the effect of C is ¬p and if C could come after A and before B).
    - eg: Right Sock $\xrightarrow{Right\ Sock\ On}$ Right Shoe



  - A set of open preconditions.
    - Planner tries to reduce this set to the empty set without introducing contradictions

Consistent Plan (POP)

- Consistent plan is a plan that has
  - No cycle in the ordering constraints
  - No conflicts with the causal links
- Solution
  - Is a consistent plan with no open preconditions
- To solve a conflict between a causal link $A \xrightarrow{p} B$ and an action C (that clobbers, threatens the causal link), we force C to occur outside the "protection interval" by adding
  - the constraint $C \prec A$ (demoting C) or
  - the constraint $B \prec C$ (promoting C)

Setting up the PoP

- Add dummy states
  - Start
    - Has no preconditions
    - Its effects are the literals of the initial state
  - Finish
    - Its preconditions are the literals of the goal state
    - Has no effects
- Initial Plan:
  - Actions: {Start, Finish}
  - Ordering constraints: {Start $\prec$ Finish}
  - Causal links: {}
  - Open Preconditions: {……}

Consistent Plan (POP)

- Consistent plan is a plan that has
  - No cycle in the ordering constraints
  - No conflicts with the causal links
- Solution
  - Is a consistent plan with no open preconditions
- To solve a conflict between a causal link $A \xrightarrow{p} B$ and an action C (that clobbers, threatens the causal link), we force C to occur outside the "protection interval" by adding
  - the constraint $C \prec A$ (demoting C) or
  - the constraint $B \prec C$ (promoting C)

POP as a Search Problem

- The successor function arbitrarily picks one open precondition *p* on an action B
- For every possible consistent action A that achieves *p*
  - It generates a successor plan adding the causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$
  - If A was not in the plan, it adds $Start \prec A$ and $A \prec Finish$
  - It resolves all conflicts between
    - the new causal link and all existing actions
    - between A and all existing causal links
  - Then it adds the successor states for combination of resolved conflicts
- It repeats until no open precondition exists

Process summary

- Operators on partial plans
  - Add link from existing plan to open precondition.
  - Add a step to fulfill an open condition.
  - Order one step w.r.t another to remove possible conflicts
- Gradually move from incomplete/vague plans to complete/correct plans
- Backtrack if an open condition is unachievable or if a conflict is irresolvable.

Given: Initial and Goal State

Start

At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Banana)

Have(Milk) At(Home) Have(Banana) Have(Drill)

Finish

Given: Plan Operators (Actions)

At(HWS)
Go(SM)
At(SM)

At(SM), Sells(SM,Banana)
Buy(Banana)
Have(Ban)

At(Home)
Go(HWS)
At(HWS)

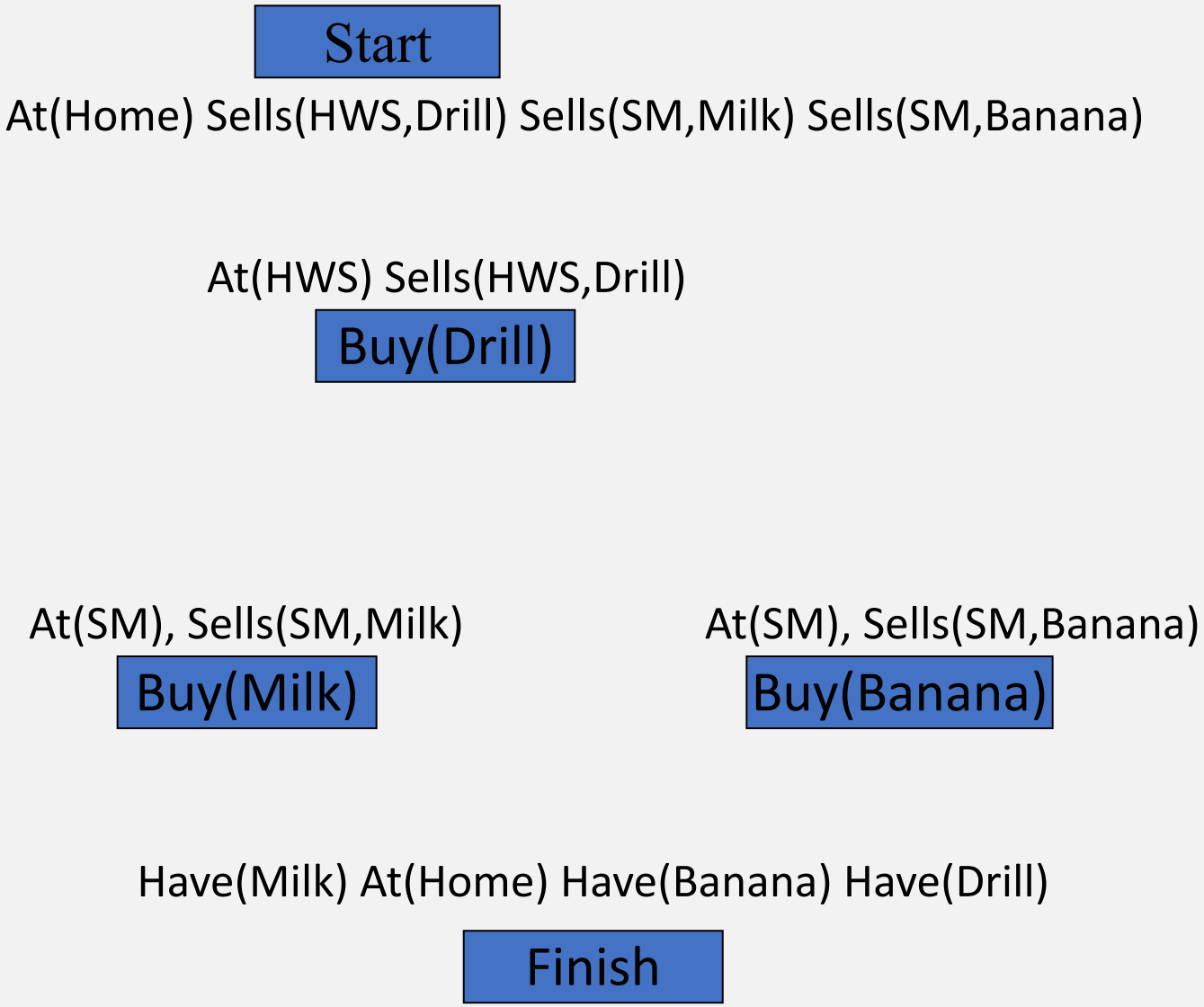
At(HWS) Sells(HWS,Drill)
Buy(Drill)
Have(Drill)

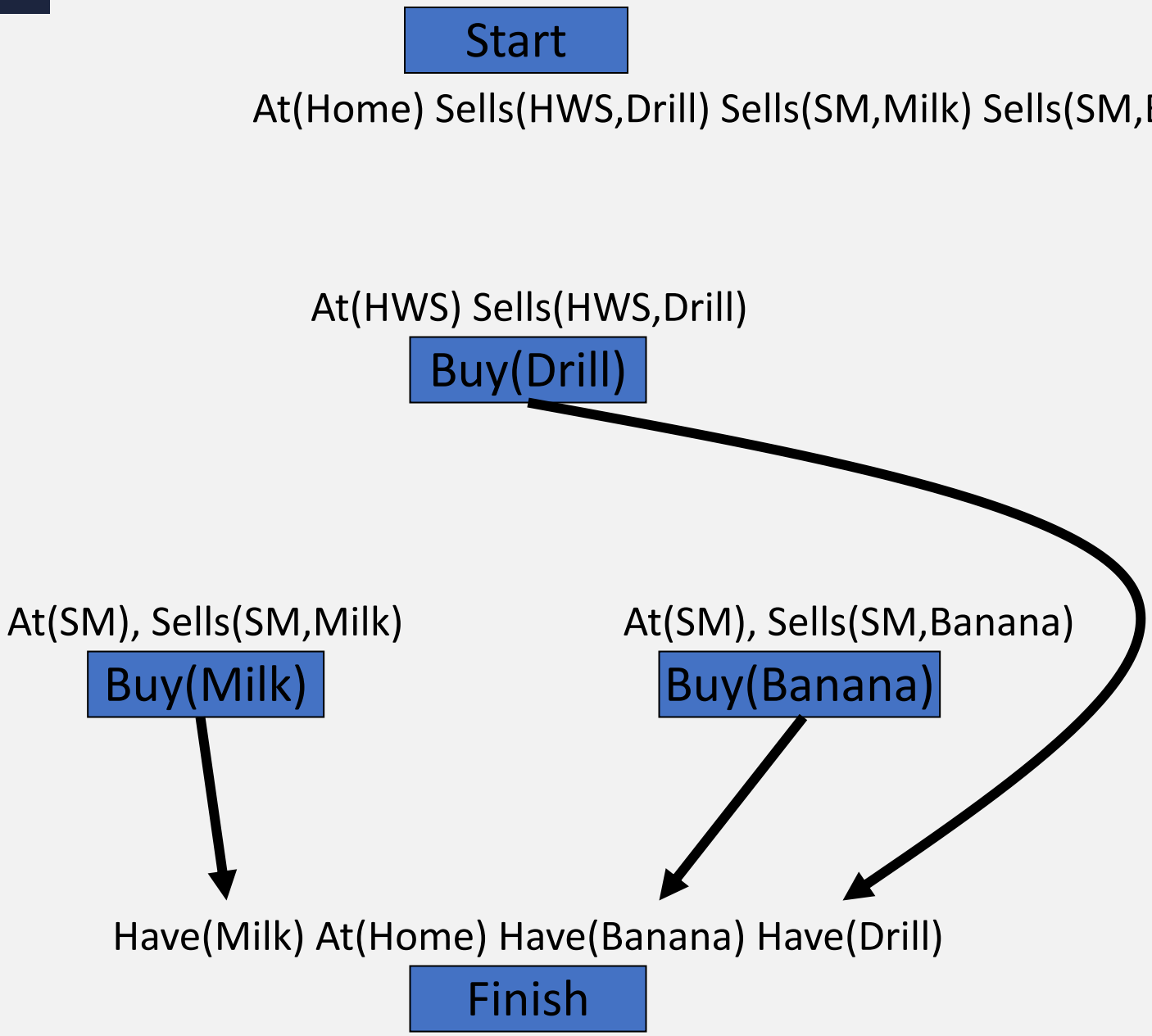
At(SM)
Go(Home)
At(Home)

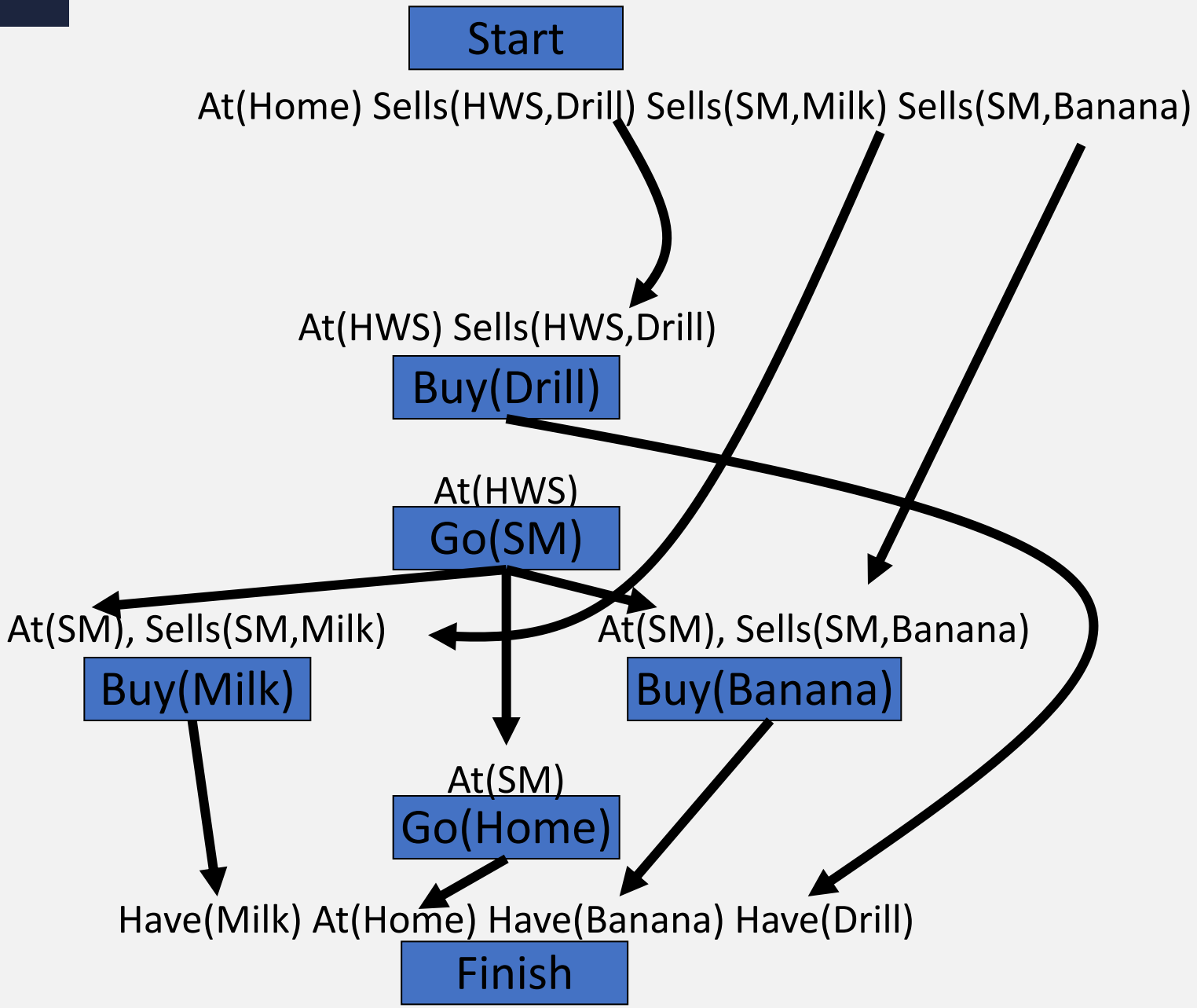
At(SM), Sells(SM,Milk)
Buy(Milk)
Have(Milk)

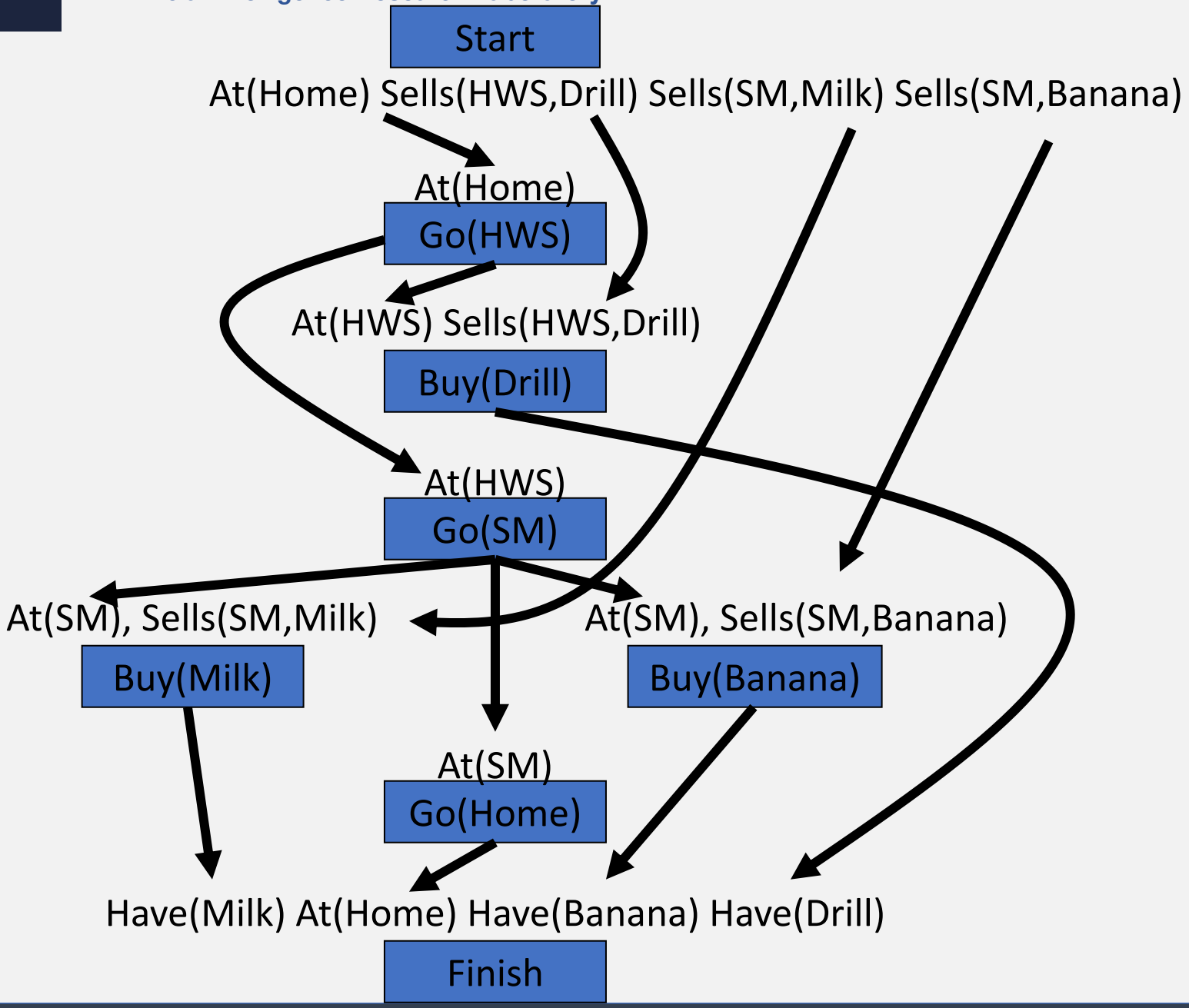
What is a solution?

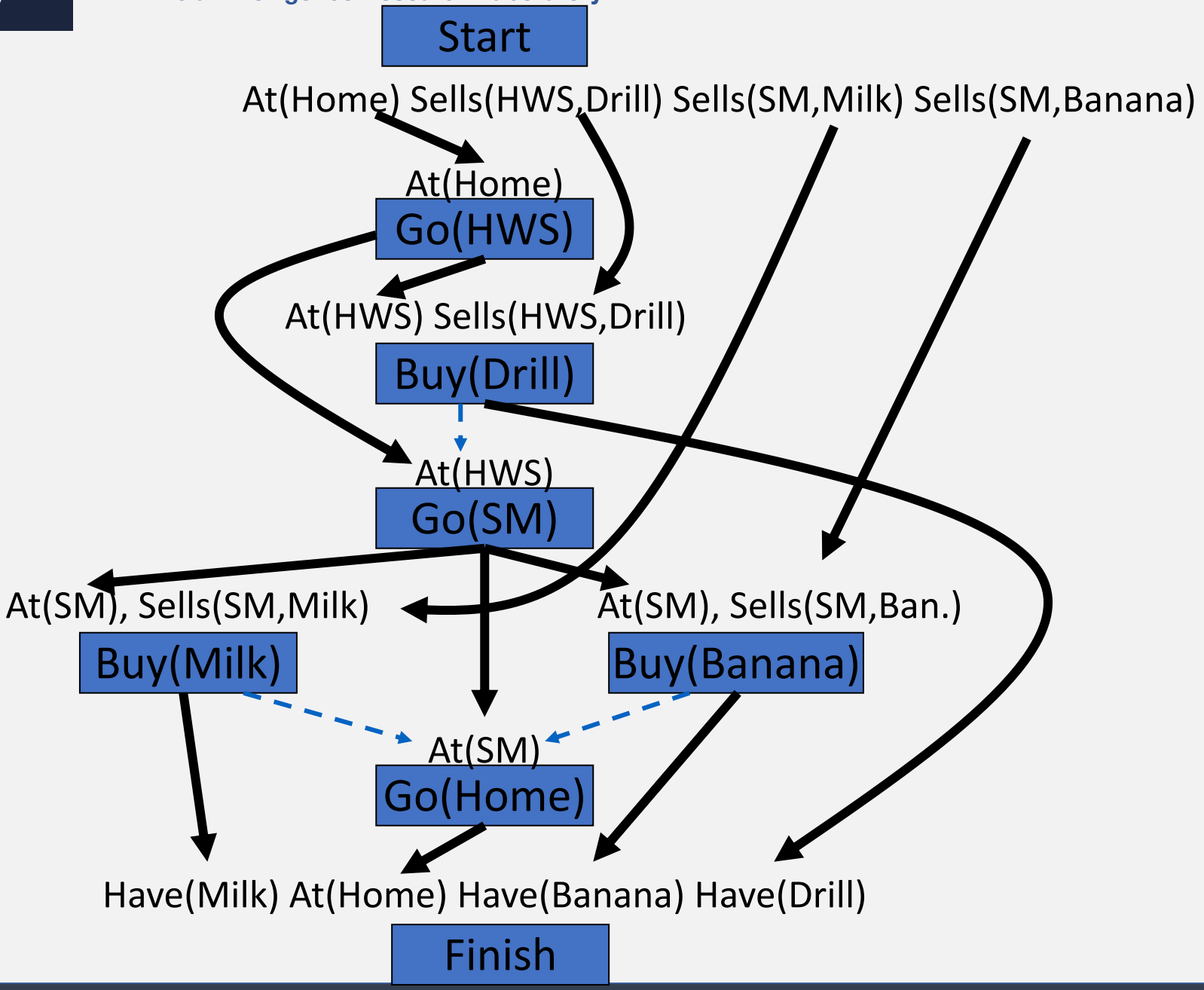
Partial Order Plan <Actions,Orderings,Links>



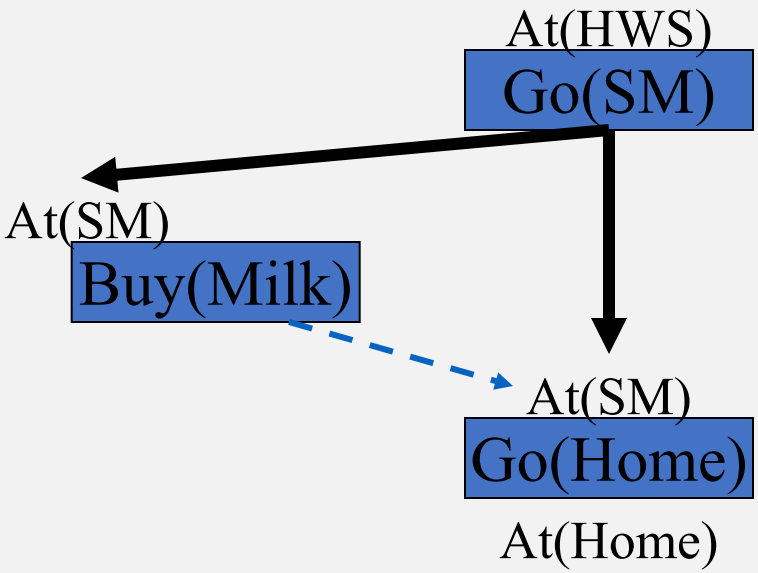

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Banana)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Banana)
Buy(Banana)
Have(Milk) At(Home) Have(Banana) Have(Drill)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Banana)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Banana)
Buy(Banana)
At(SM)
Go(Home)
Have(Milk) At(Home) Have(Banana) Have(Drill)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Banana)
At(Home)
Go(HWS)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Banana)
Buy(Banana)
At(SM)
Go(Home)
Have(Milk) At(Home) Have(Banana) Have(Drill)
Finish

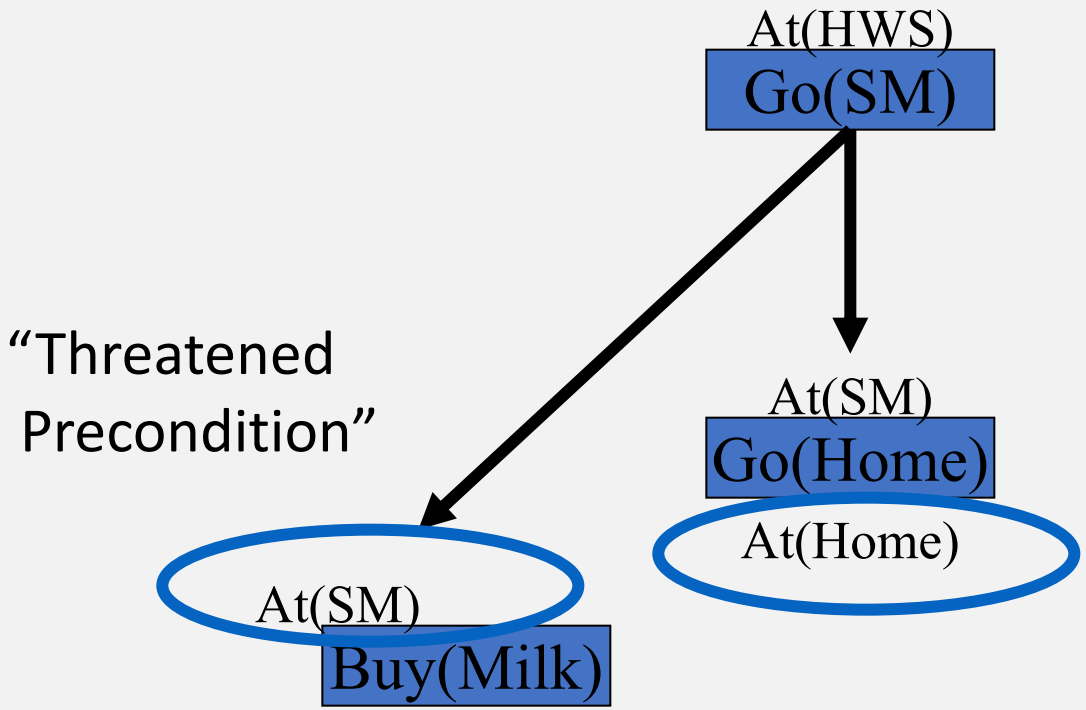

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Banana)
At(Home)
Go(HWS)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Ban.)
Buy(Milk)
Buy(Banana)
At(SM)
Go(Home)
Have(Milk) At(Home) Have(Banana) Have(Drill)
Finish

Why is an ordering needed?



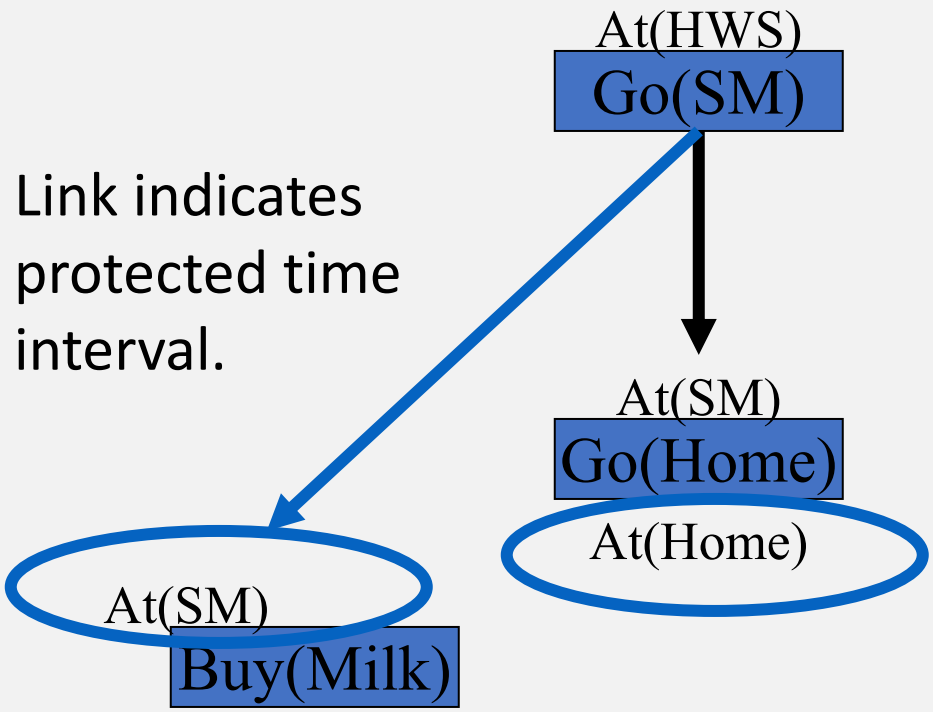
Why is an ordering needed?

Suppose the other order is allowed, what happens?

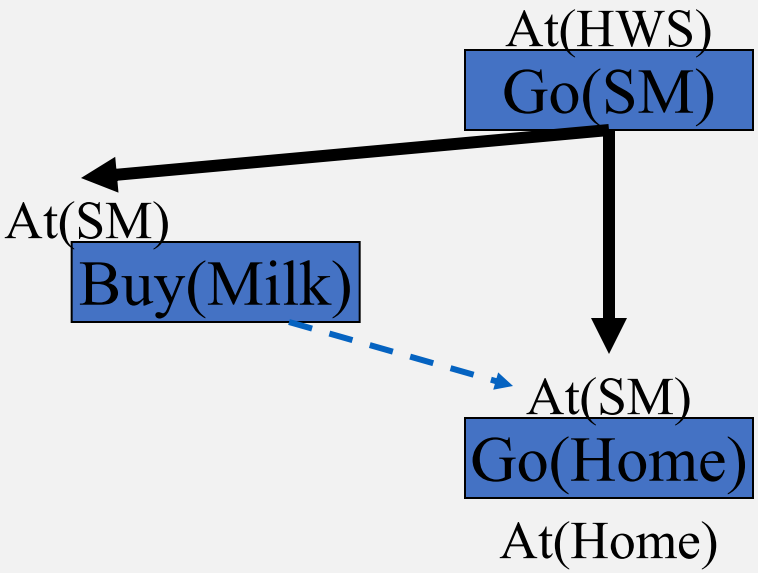


Why is an ordering needed?

Suppose the other order is allowed, what happens?



Ordering Resolves Threat

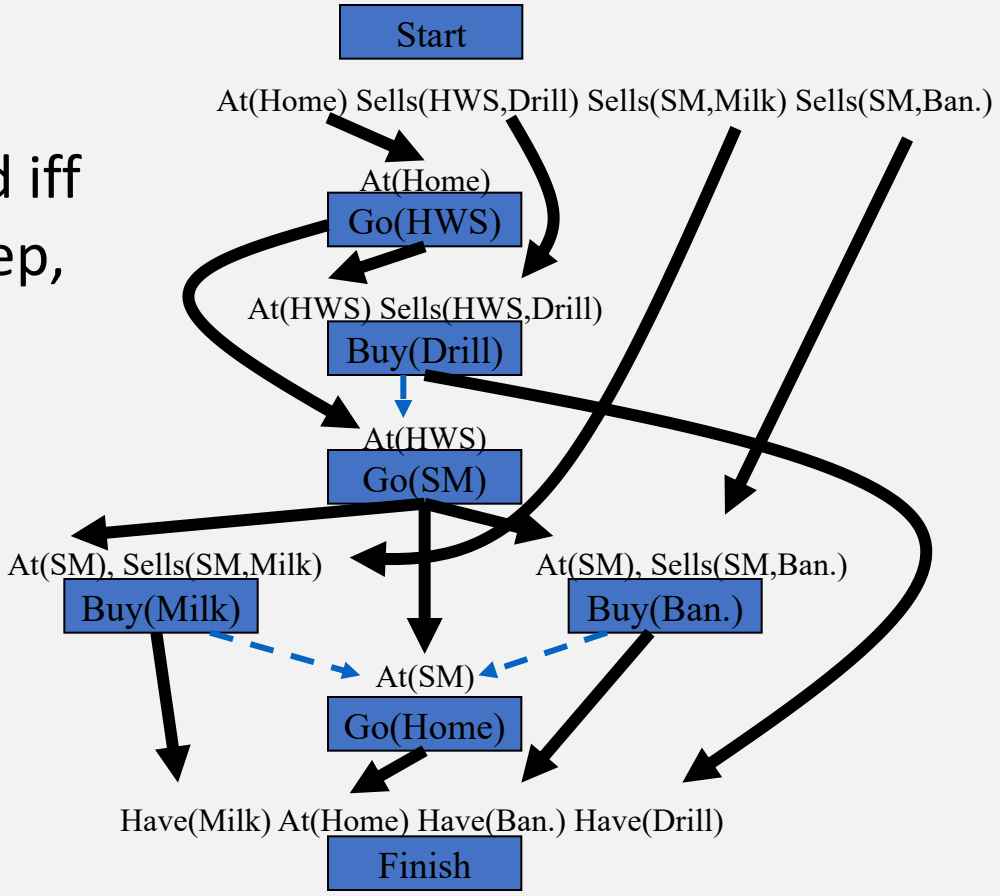


A Solution: Complete and Consistent Plan

- Complete Plan
  IFF every precondition of every step is achieved
  A step's precondition is achieved iff
  - its the effect of a preceding step,
  - no possibly intervening step undoes it.

- Consistent Plan

  IFF there is no contradiction in the ordering constraints (I.e., never $s_i < s_j$ and $s_j < s_i$.)



POP(<A,O,L>, agenda, actions)

- <A,O,L>, A partial plan to expand
- Agenda: A queue of open conditions still to be satisfied: <p, a_{need} >
- Actions: A set of actions that may be introduced to meet needs.
- a_{add}: an action that produces the needed condition p for a_{need}
- A_{threat} : an action that might threaten a causal link from $a_{producer}$ to $a_{consumer}$

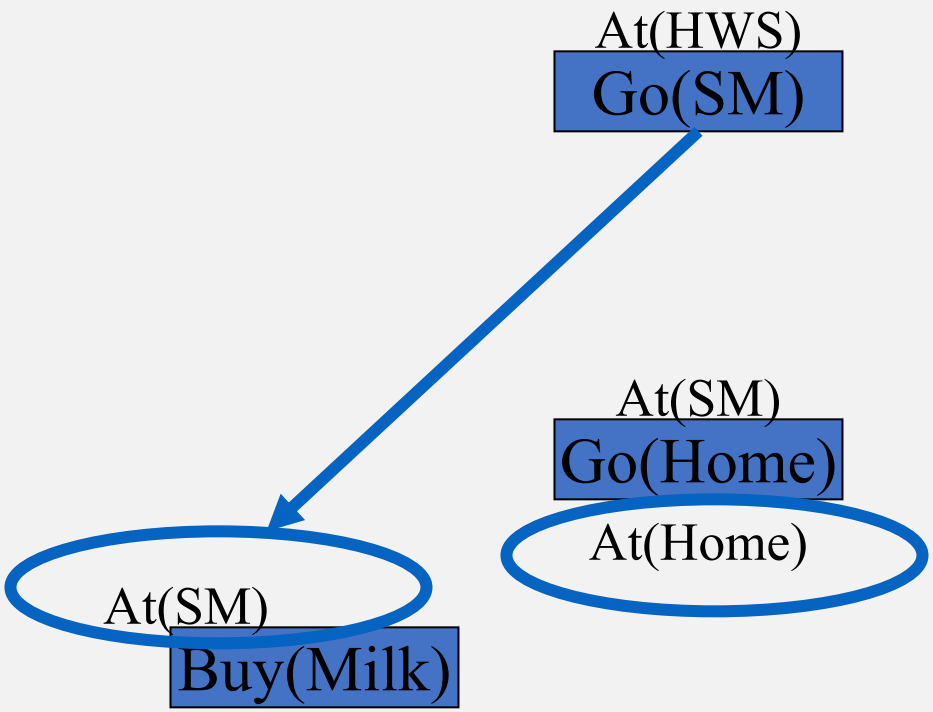
POP(<A,O,L>, agenda, actions)

1. **Termination**: If agenda is empty, return plan <A,O,L>.
2. **Goal Selection**: select and remove open condition <p, a_{need} > from agenda.
3. **Action Selection**: Choose new or existing action a_{add} that can precede a_{need} and whose effects include p.
   Link and order actions.
4. **Update Agenda**: If a_{add} is new, add its preconditions to agenda.
5. **Threat Detection**: For every action a_{threat} that might threaten some causal link from $a_{produce}$ to $a_{consume}$, choose a consistent ordering:
   a) Demotion: Add $a_{threat} < a_{produce}$
   b) Promotion: Add $a_{consume} < a_{threat}$
6. **Recurse:** on modified plan and agenda

Choose is nondeterministic Select is deterministic

To remove threats…

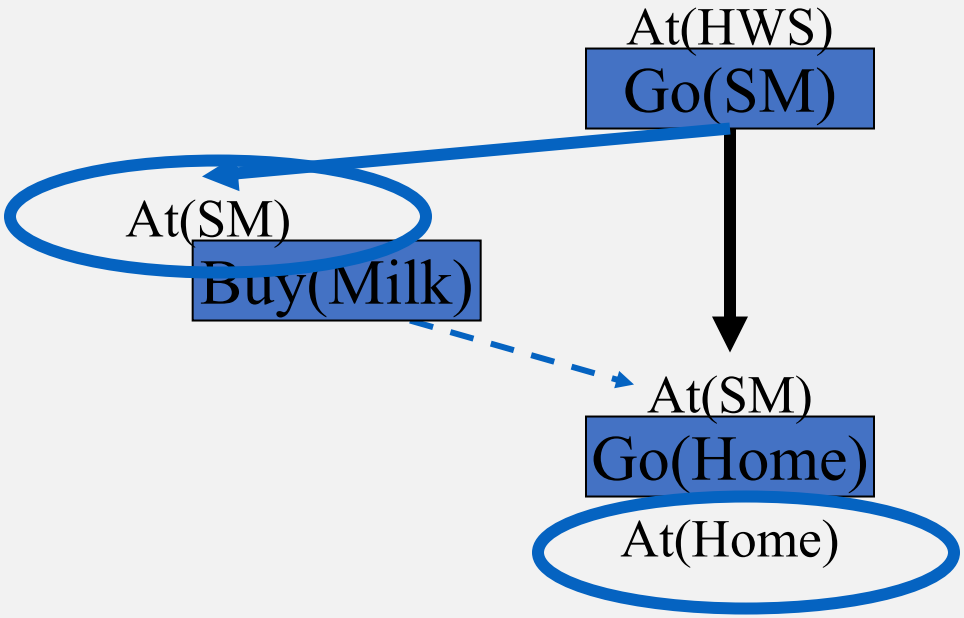
promote the threat…



To remove threats…

promote the threat…

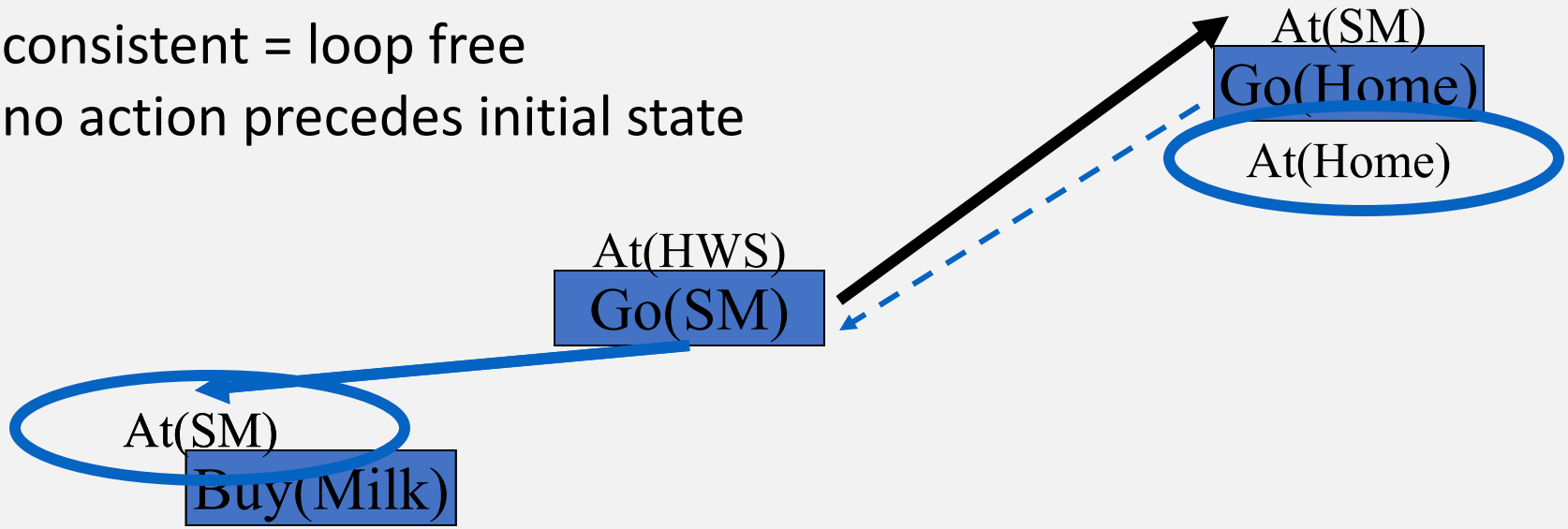
demote the threat…

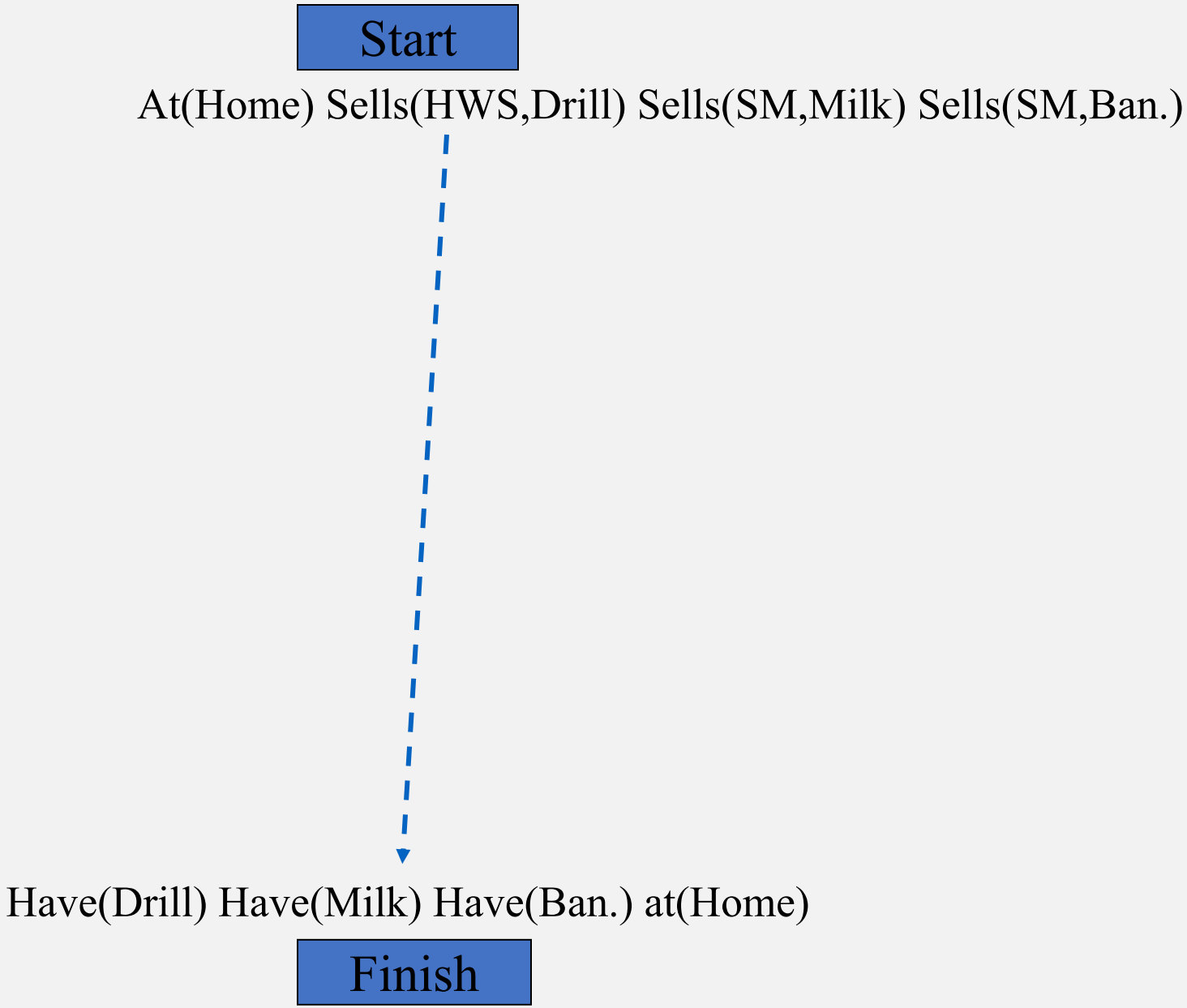


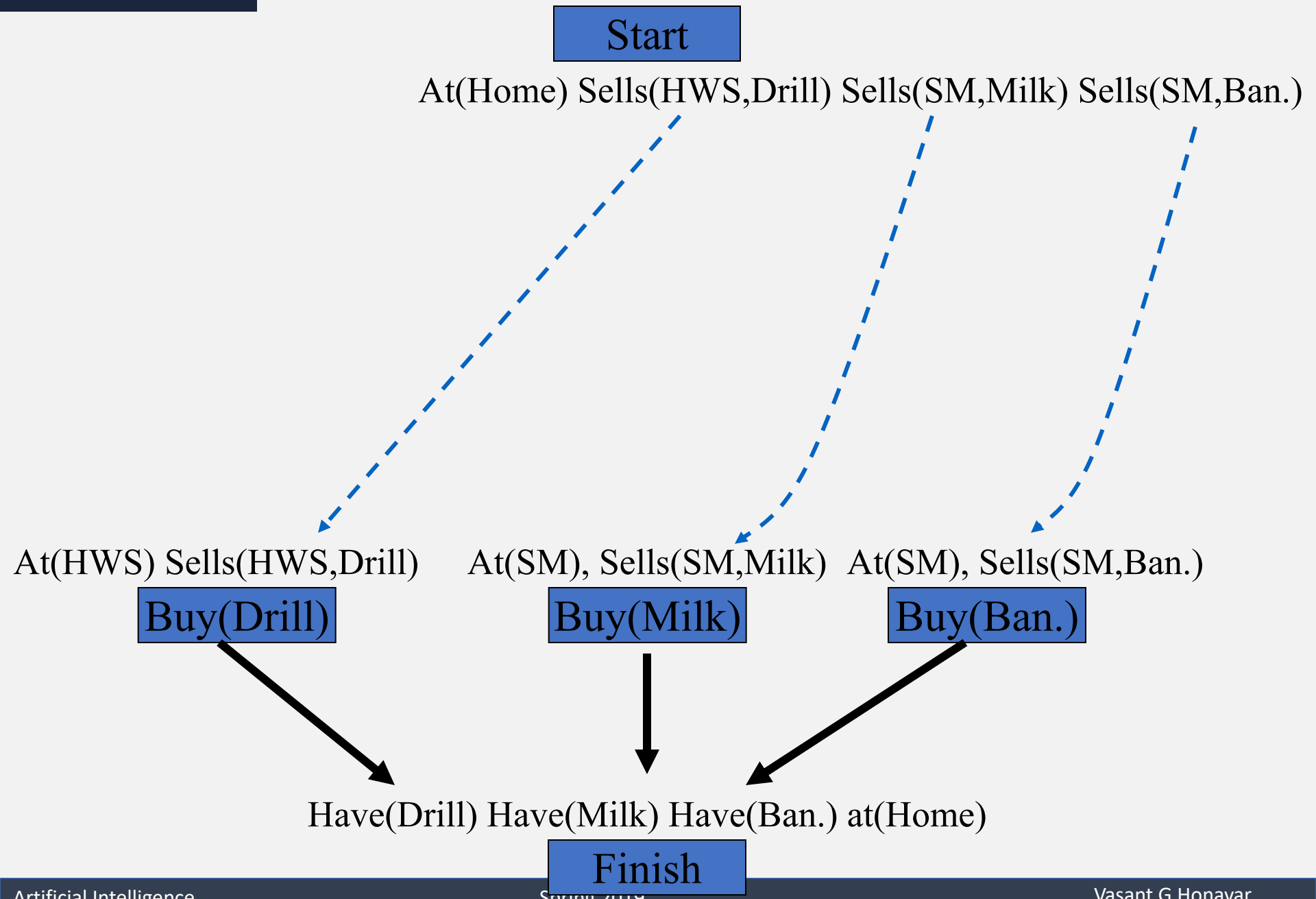
To remove threats…

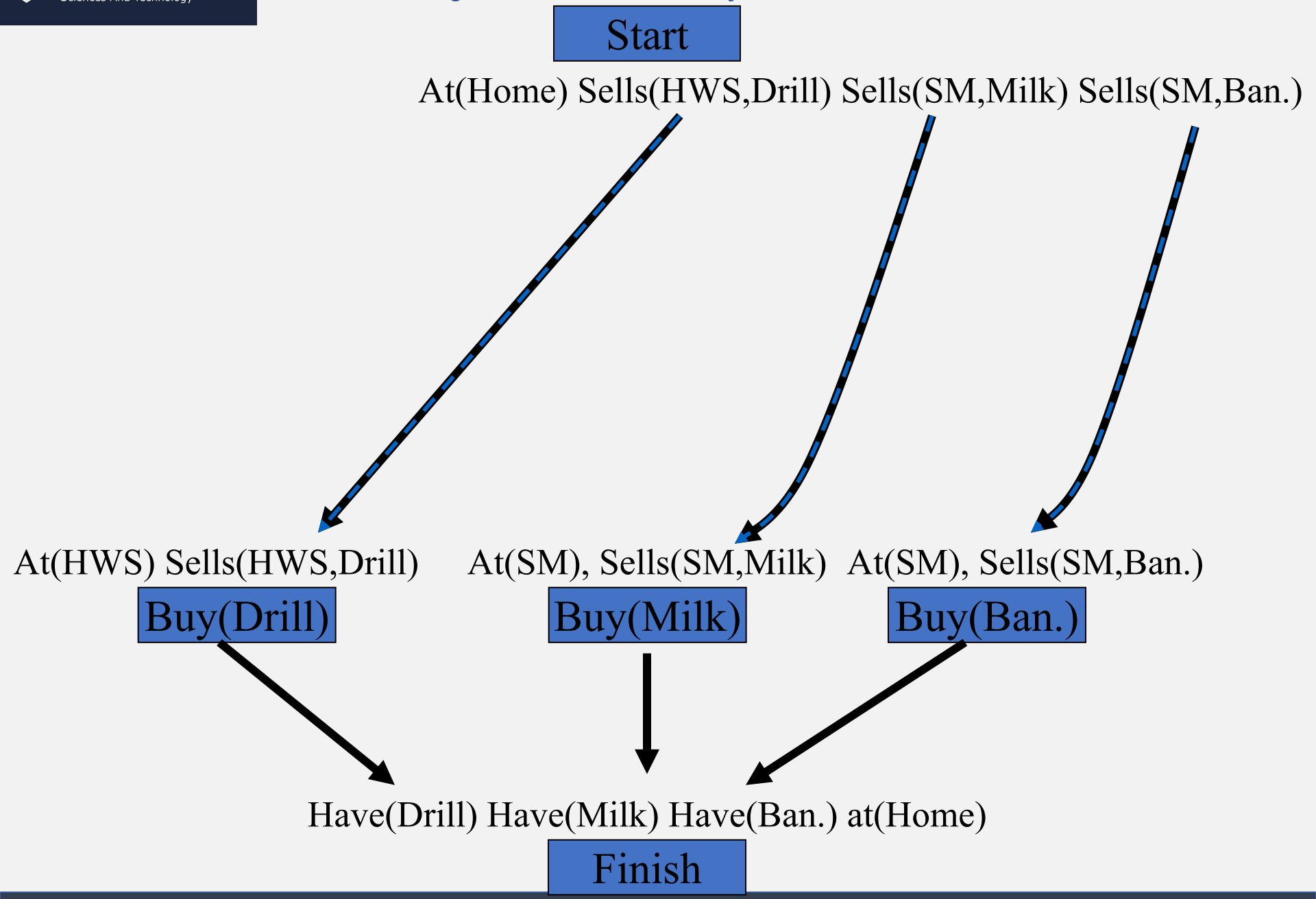
promote the threat… demote the threat…

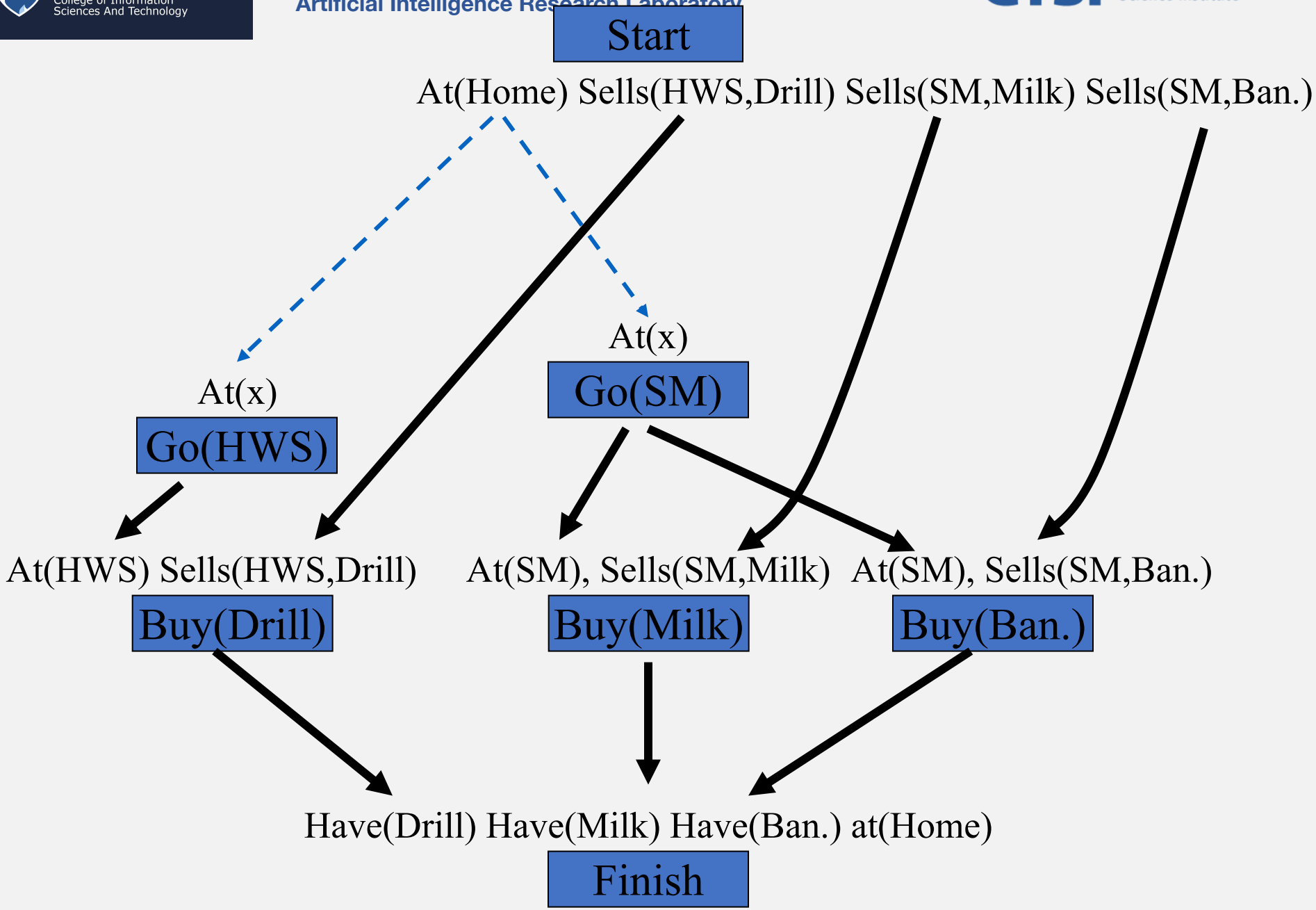
- But allow demotion/promotion only if schedulable
  - consistent = loop free
  - no action precedes initial state

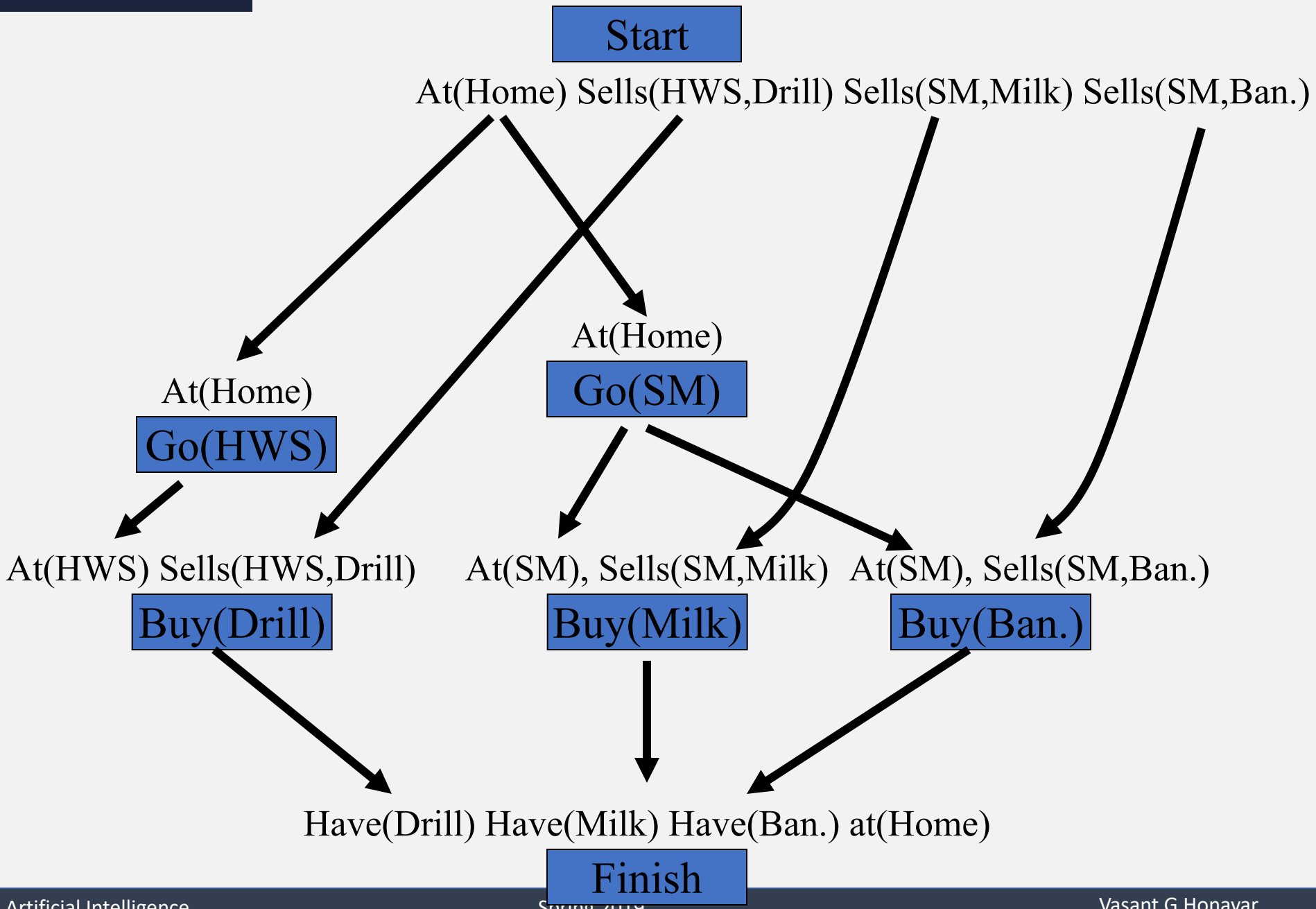


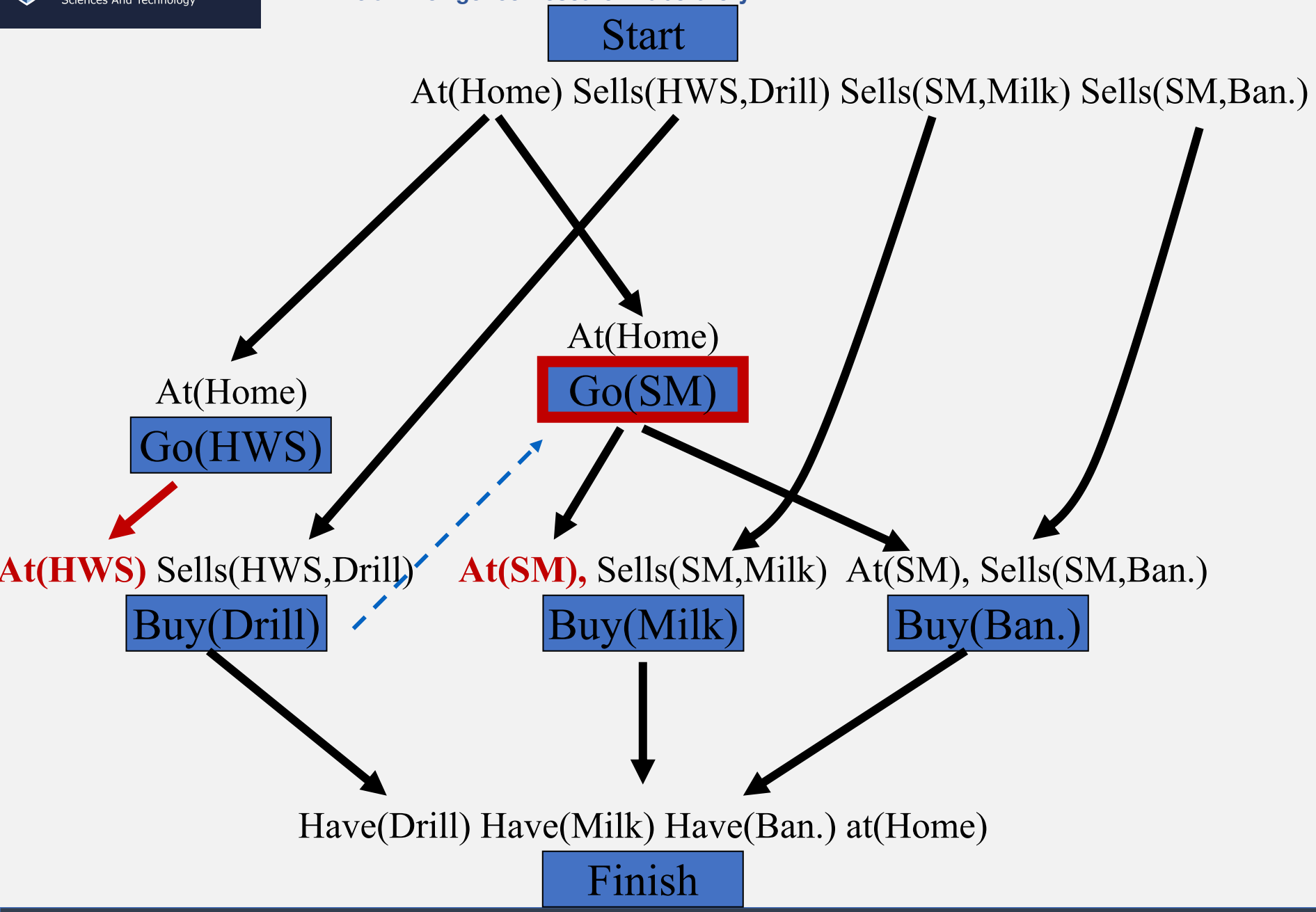

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

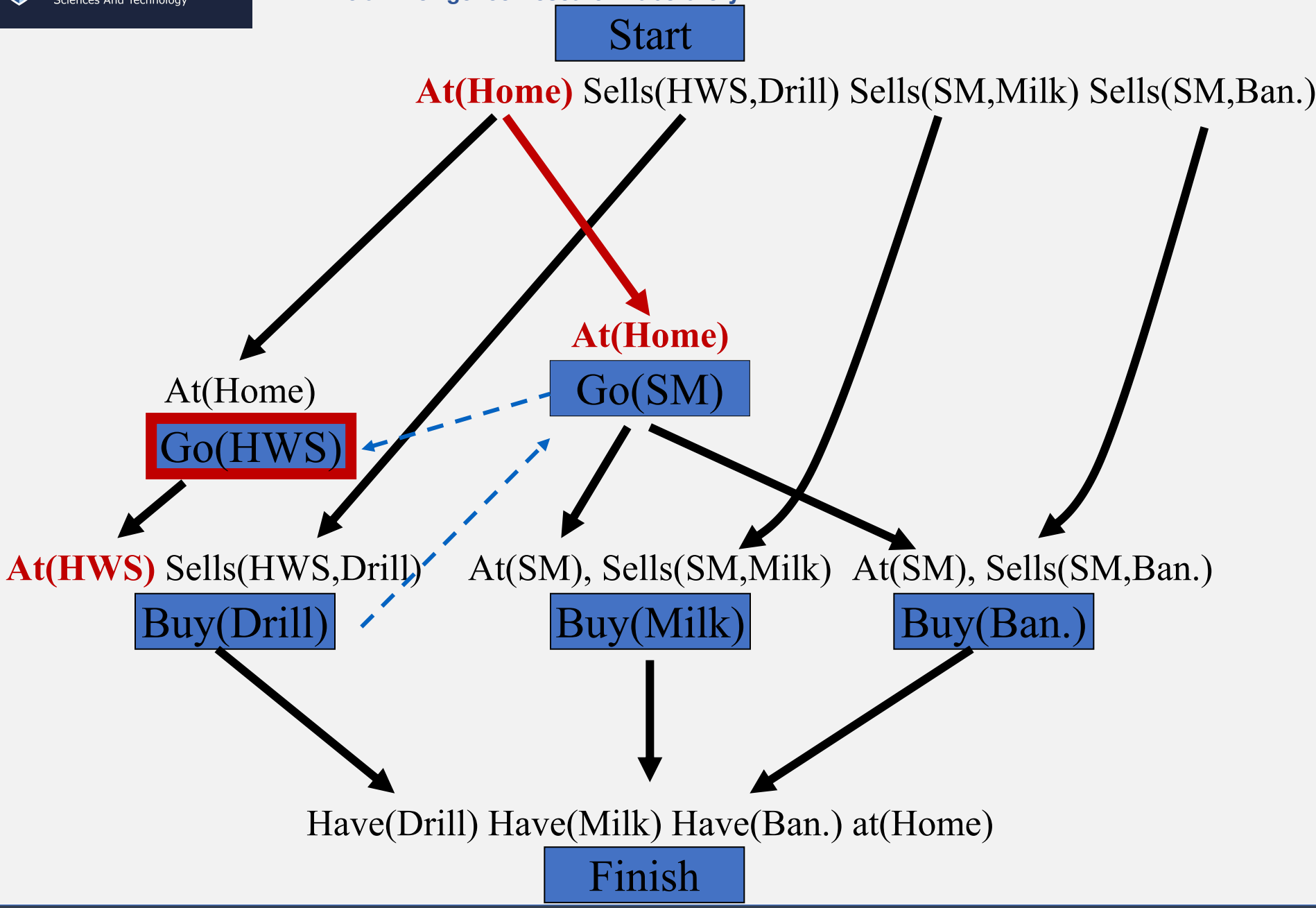

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(HWS) Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Ban.)
Buy(Drill)
Buy(Milk)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

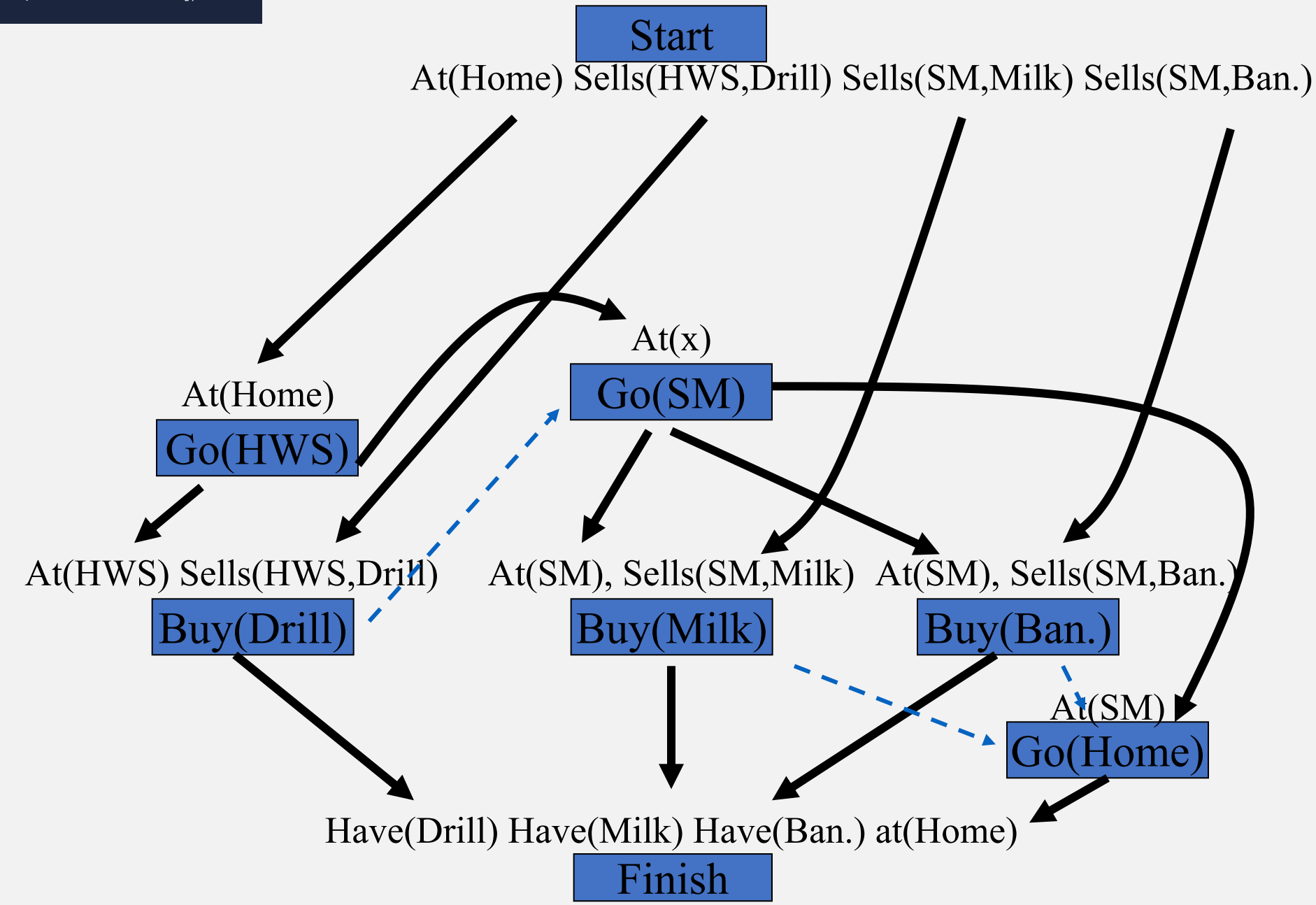

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(HWS) Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Ban.)
Buy(Drill)
Buy(Milk)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

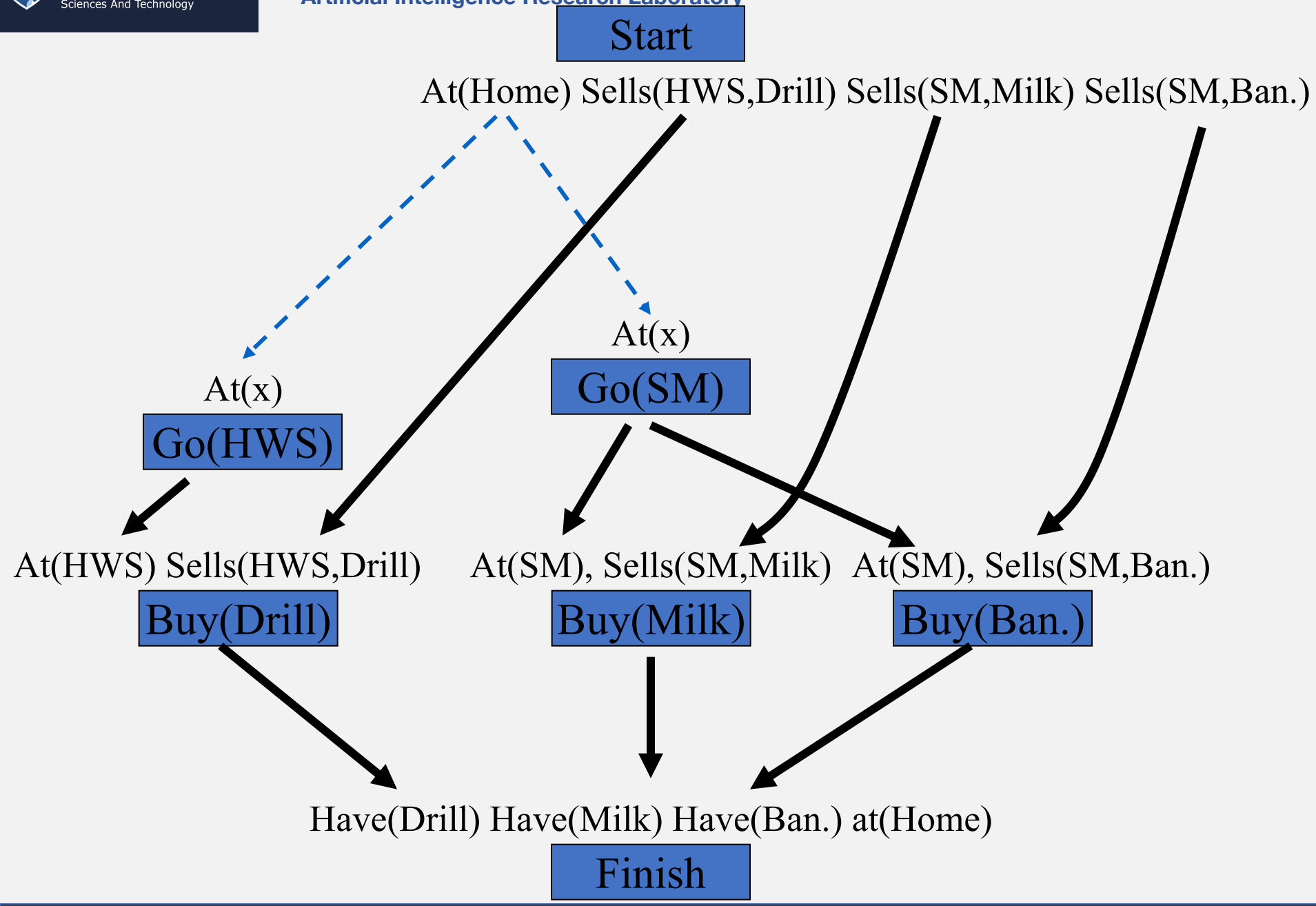

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(x)
Go(HWS)
At(x)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

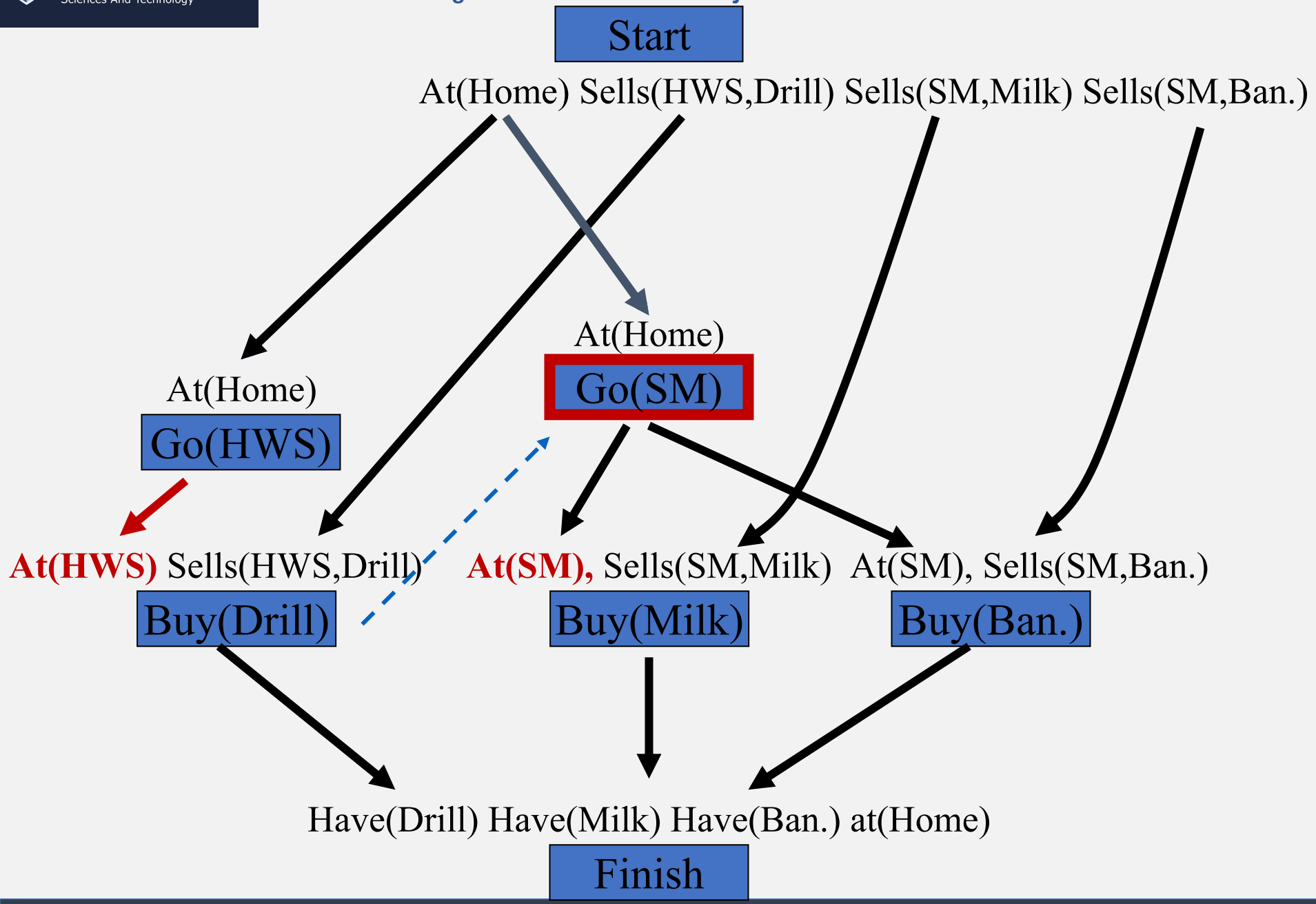

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

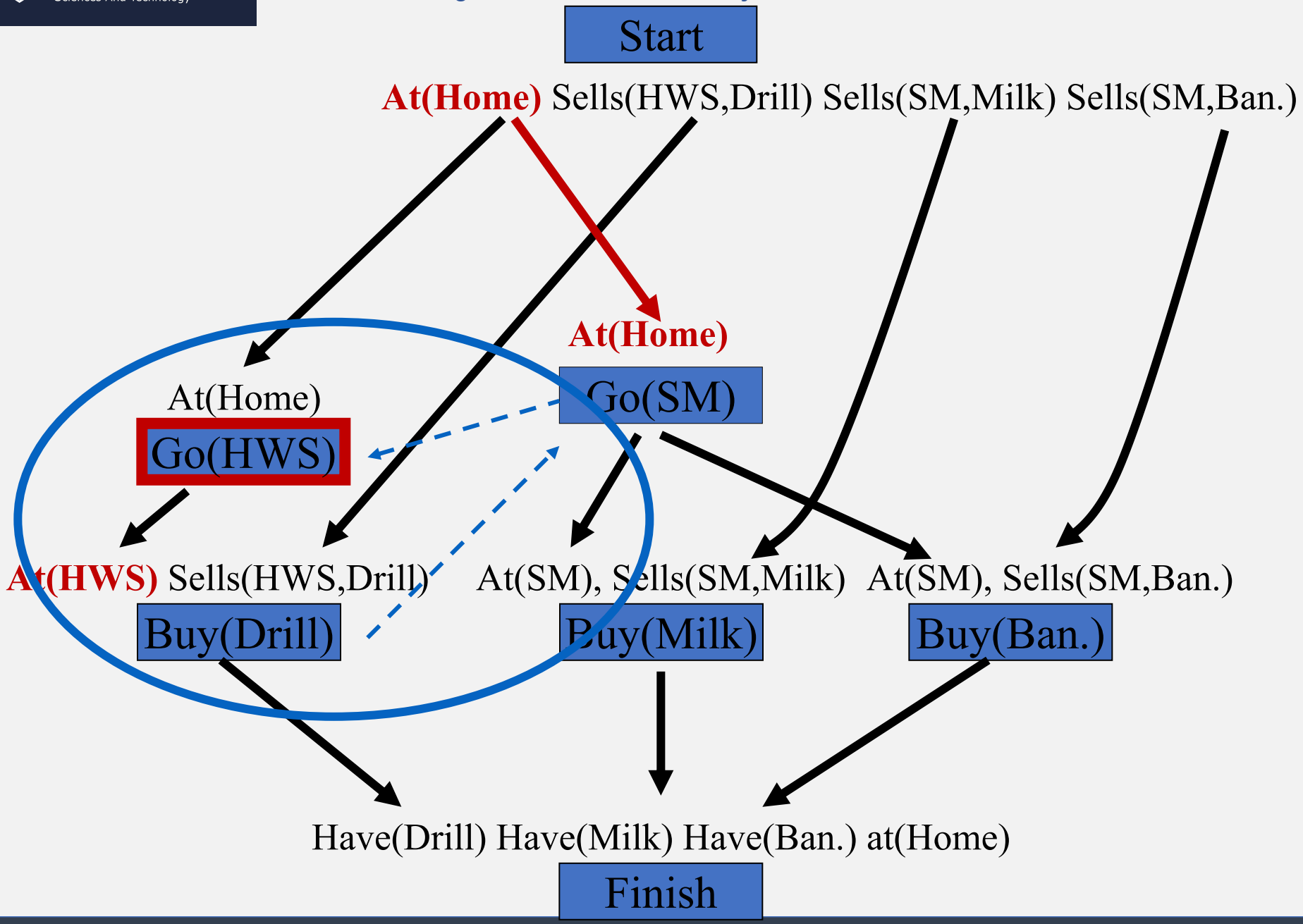

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

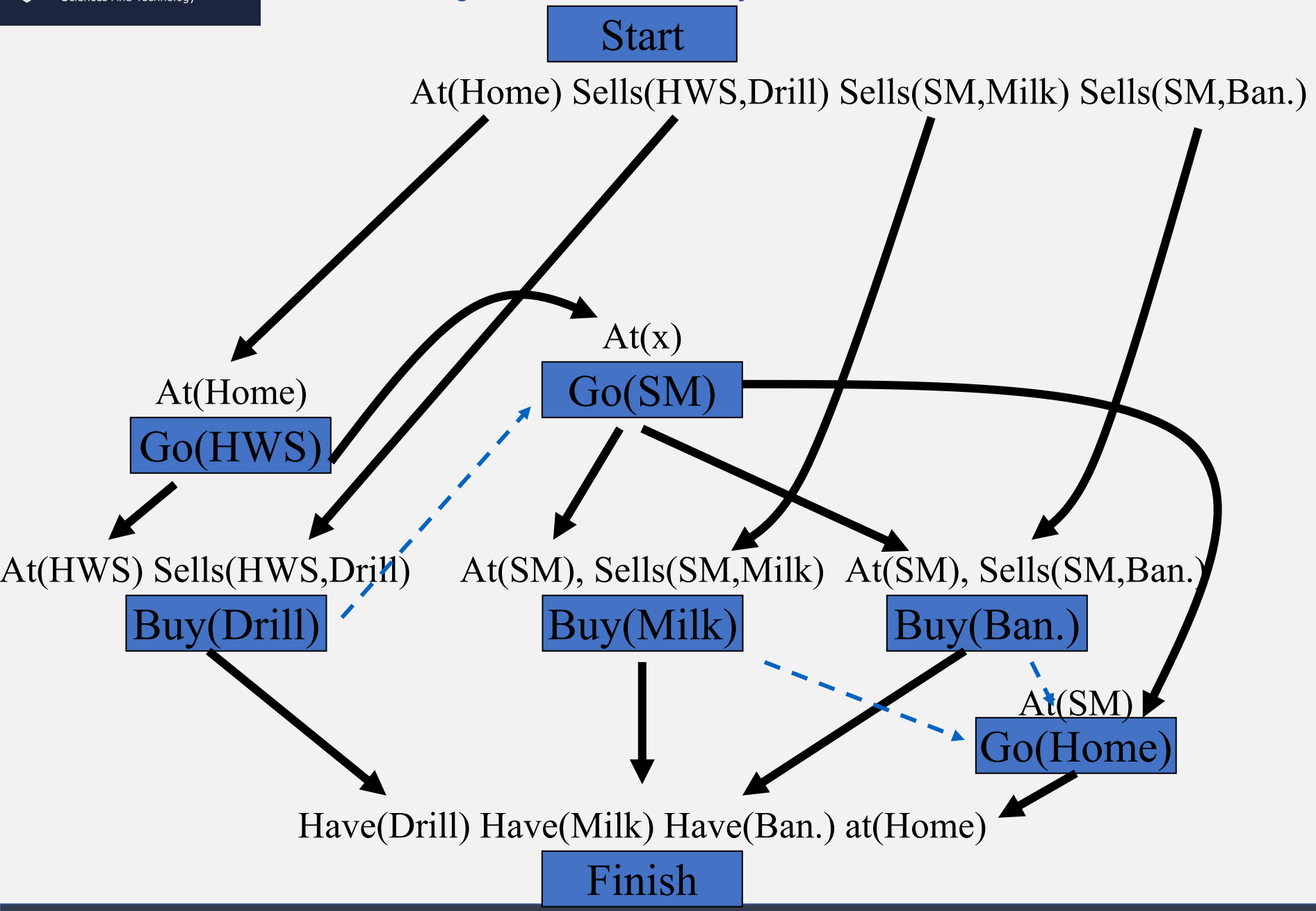

Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(x)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
At(SM)
Go(Home)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(x)
Go(HWS)
At(x)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(Home)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish


Start
At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)
At(Home)
Go(HWS)
At(x)
Go(SM)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(SM), Sells(SM,Milk)
Buy(Milk)
At(SM), Sells(SM,Ban.)
Buy(Ban.)
At(SM)
Go(Home)
Have(Drill) Have(Milk) Have(Ban.) at(Home)
Finish

POP Algorithm (1)

- Backtrack when fails to resolve a threat or find an operator
- Causal links
  - Recognize when to abandon a doomed plan without wasting time expanding irrelevant part of the plan
  - allow early pruning of inconsistent combination of actions
- When actions include variables, we need to find appropriate substitutions
  - Typically we try to delay commitments to instantiating a variable until we have no other choice (least commitment)
- POP is sound, complete, and systematic (no repetition)

POP Algorithm (2)

- Decomposes the problem (advantage)
- But does not represent states explicitly: it is hard to design heuristic to estimate distance from goal
  - Example: Number of open preconditions – those that match the effects of the start node. Not perfect (same problems as before)
- A heuristic can be used to choose which plan to refine (which precondition to pick-up):
  - Choose the most-constrained precondition, the one satisfied by the least number of actions. Like in CSPs!
  - When no action satisfies a precondition, backtrack!
  - When only one action satisfies a precondition, pick up the precondiction.

Planning graphs

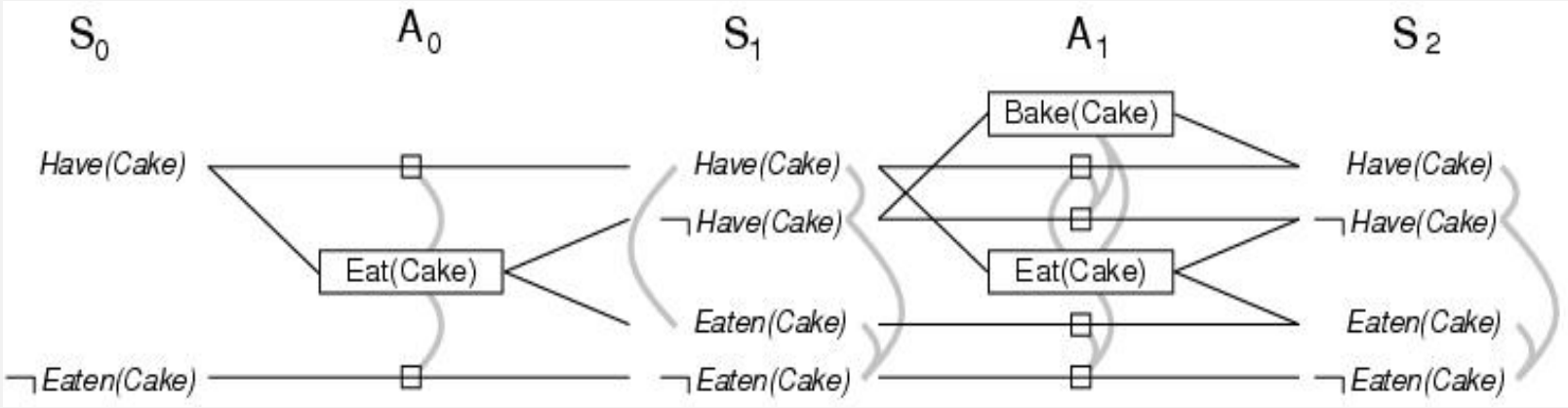
- Used to achieve better heuristic estimates.
  - A solution can also directly extracted using GRAPHPLAN algorithm
- Consists of a sequence of levels that correspond to time steps in the plan.
  - Level 0 is the initial state.
  - Each level consists of a set of literals and a set of actions.
    - Literals = all those that could be true at that time step, depending upon the actions executed at the preceding time step.
    - Actions = all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.

Planning graphs

- "Could"?
  - Records only a restricted subset of possible negative interactions among actions.
- They work only for propositional problems.
- Example:

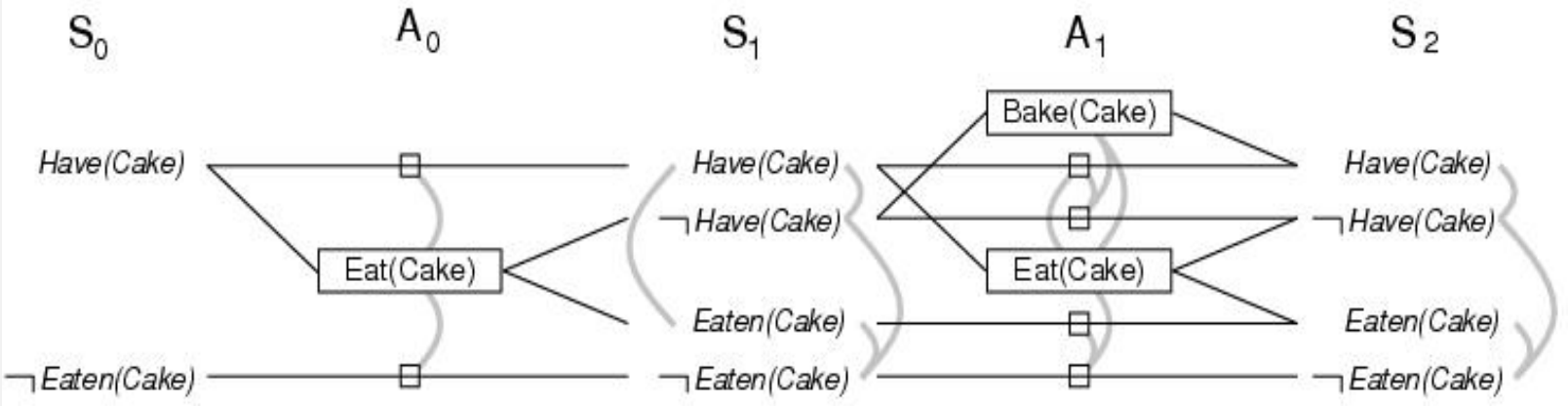
Init(Have(Cake))
Goal(Have(Cake) ∧ Eaten(Cake))
Action(Eat(Cake), PRECOND: Have(Cake)
EFFECT: ¬Have(Cake) ∧ Eaten(Cake))
Action(Bake(Cake), PRECOND: ¬ Have(Cake)
EFFECT: Have(Cake))

Cake example



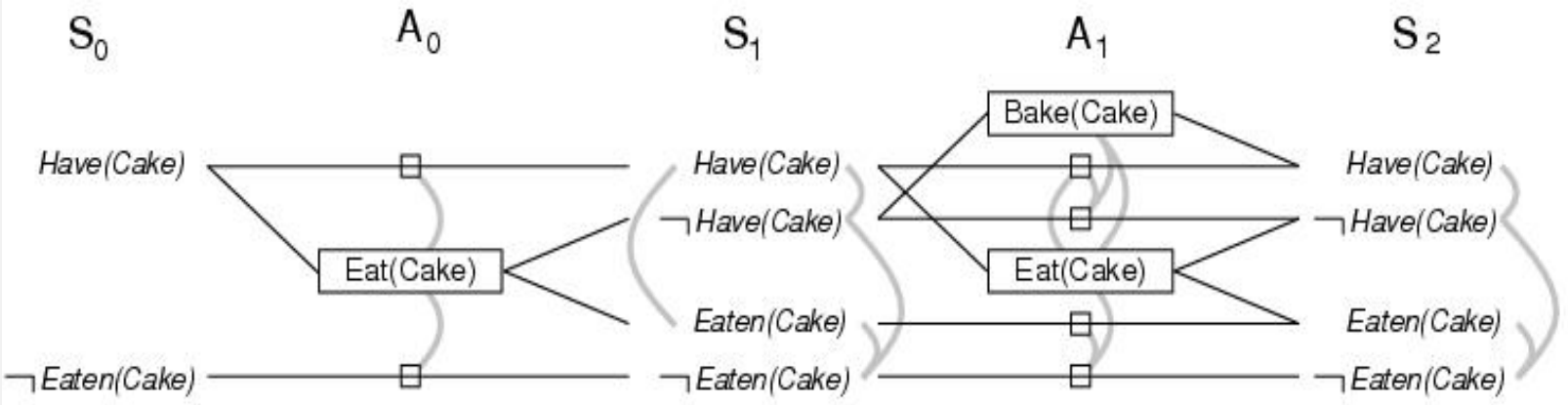
- Start at level S0 and determine action level A0 and next level S1.
  - A0 >> all actions whose preconditions are satisfied in the previous level.
  - Connect precond and effect of actions S0 --> S1
  - Inaction is represented by persistence actions.
- Level A0 contains the actions that could occur
  - Conflicts between actions are represented by mutex links

Cake example



- Level S1 contains all literals that could result from picking any subset of actions in A0
  - Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  - S1 defines multiple states and the mutex links are the constraints that define this set of states.
- Continue until two consecutive levels are identical: leveled off
  - Or contain the same amount of literals (explanation follows later)

Cake example



- A mutex relation holds between two actions when:
  - Inconsistent effects: one action negates the effect of another.
  - Interference: one of the effects of one action is the negation of a precondition of the other.
  - Competing needs: one of the preconditions of one action is mutually exclusive with the precondition of the other.
- A mutex relation holds between two literals when (inconsistent support):
  - If one is the negation of the other OR
  - if each possible action pair that could achieve the literals is mutex.

The GRAPHPLAN Algorithm

- How to extract a solution directly from the PG

function GRAPHPLAN(problem) **return** 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**

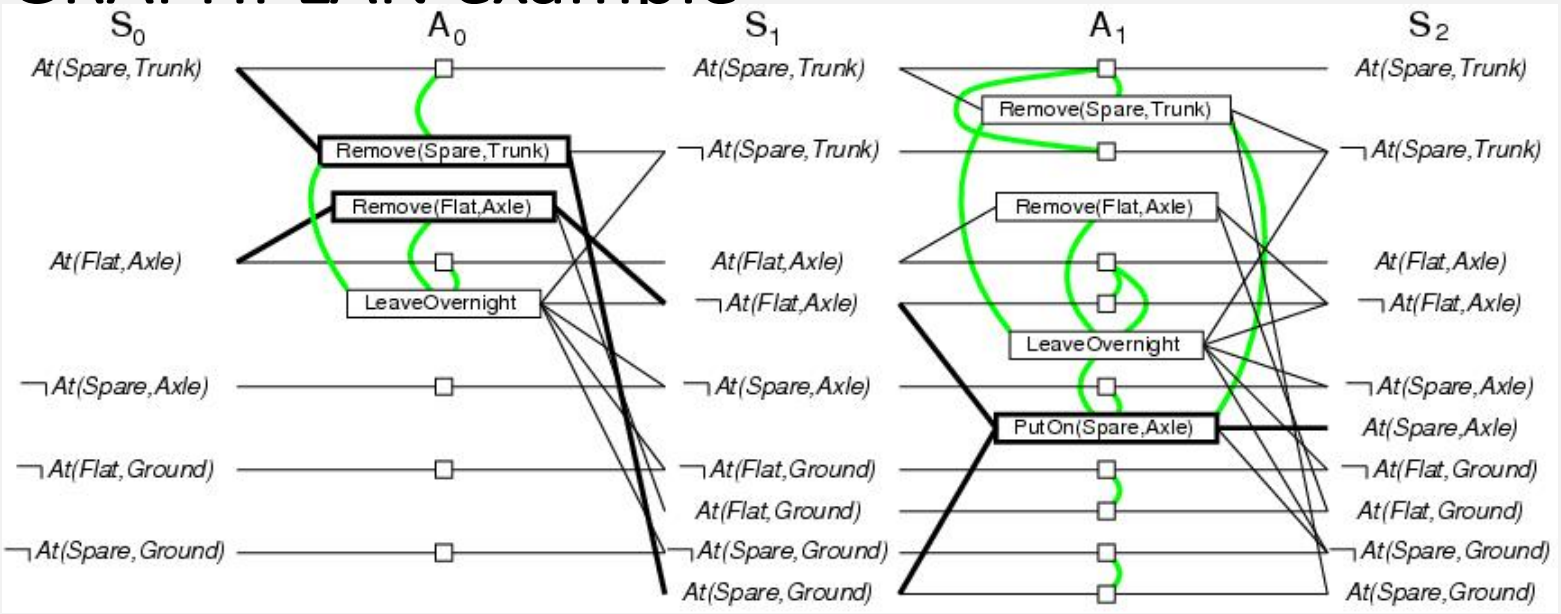
solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))

if solution ≠ failure **then return** solution

else if NO-SOLUTION-POSSIBLE(graph) **then return** failure

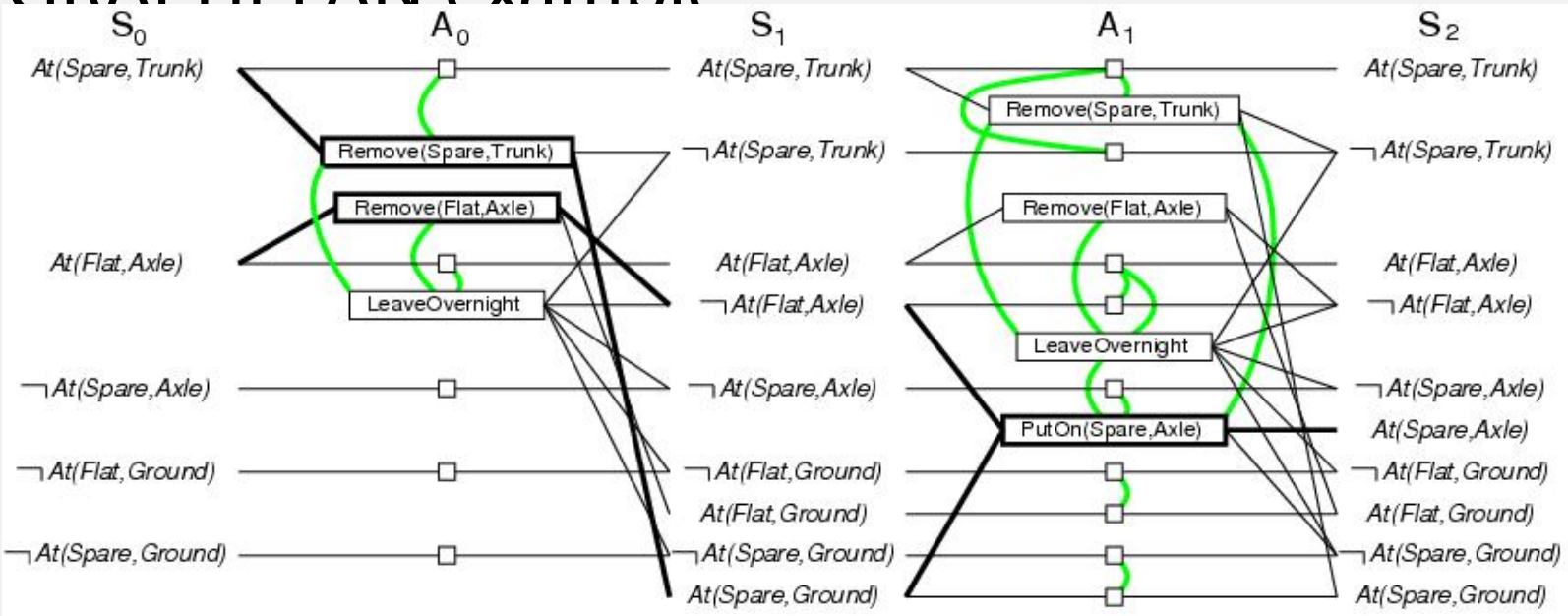
graph ← EXPAND-GRAPH(graph, problem)

GRAPHPLAN example



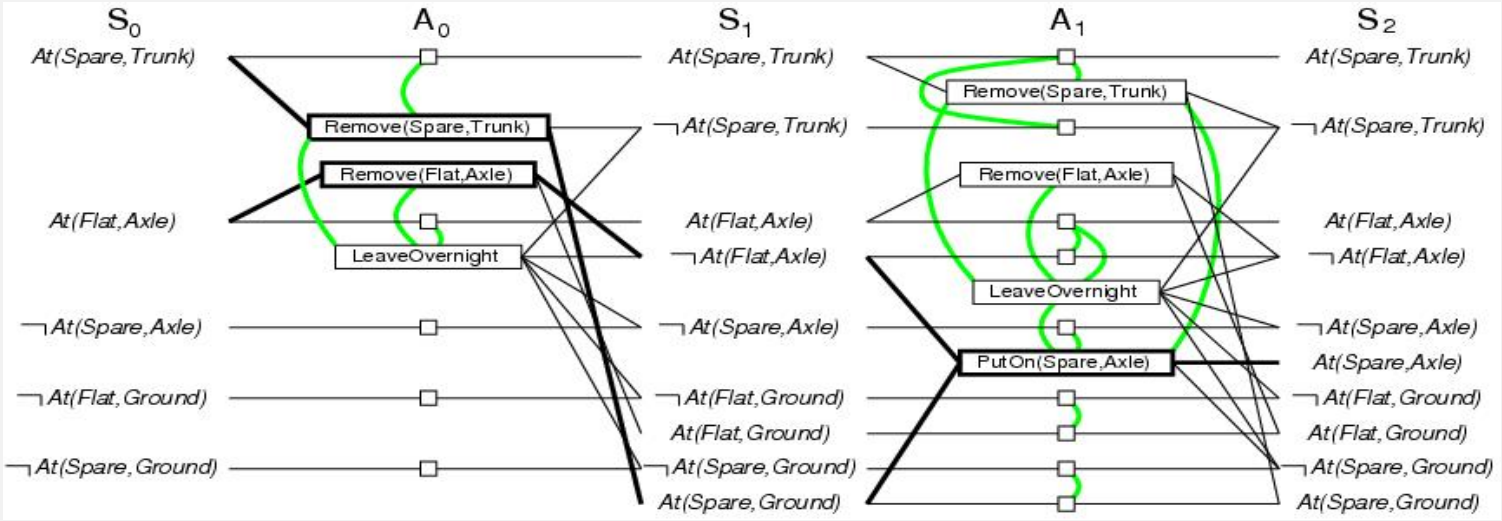
- Initially the plan consist of 5 literals from the initial state and the CWA literals (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si

GRAPHPLAN example



- EXPAND-GRAPH also looks for mutex relations
  - Inconsistent effects
    - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and not At(Spare, Ground)
  - Interference
    - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and not At(Flat,Axle) as EFFECT
  - Competing needs
    - E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and not At(Flat, Axle)
  - Inconsistent support
    - E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

GRAPHPLAN example



- In S2, the goal literals exist and are not mutex with any other
  - Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
  - Initial state = last level of PG and goal goals of planning problem
  - Actions = select any set of non-conflicting actions that cover the goals in the state
  - Goal = reach level S0 such that all goals are satisfied
  - Cost = 1 for each action.

GRAPHPLAN characteristics

- Will terminate.
  - PG are monotonically increasing or decreasing:
    - Literals increase monotonically
    - Actions increase monotonically
    - Mutexes decrease monotonically
  - Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off .
- A solution is guaranteed not to exist when
  - The graph levels off with all goals present & non-mutex, and
  - EXTRACTSOLUTION fails to find solution

PG and heuristic estimation

- PG's provide information about the problem
  - A literal that does not appear in the final level of the graph cannot be achieved by any plan.
    - Useful for backward search (cost = inf).
  - Level of appearance can be used as cost estimate of achieving any goal literals = level cost.
  - Small problem: several actions can occur
    - Restrict to one action using serial PG (add mutex links between every pair of actions, except persistence actions).
  - Cost of a conjunction of goals? Max-level, sum-level and set-level heuristics.
- PG is a relaxed problem.