

**CSC2542**

# **Domain-Customized Planning**

Sheila McIlraith

Department of Computer Science

University of Toronto

# Administrative Notes

The placement of this material doesn't follow the conceptual flow of the rest of the material I've presented, but this information may be useful to some of you for conception of your projects, so we're taking a brief sojourn from "Domain-Independent Planning" to review the basic techniques for domain-customized planning.

# Acknowledgements

Some of the slides used in this course are modifications of Dana Nau's lecture slides for the textbook *Automated Planning*, licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License:

<http://creativecommons.org/licenses/by-nc-sa/2.0/>

I would like to gratefully acknowledge the contributions of these researchers, and thank them for generously permitting me to use aspects of their presentation material.

# Outline

- Domain Control Knowledge
- Control Rules: TLPlan
- Procedural DCK: Hierarchical Task Networks
- Procedural DCK: Golog

# General Motivation

- Often, planning can be done much more efficiently if we have domain-specific information
- Example:
  - classical planning is EXPSPACE-complete
  - block stacking can be done in time  $O(n^3)$
- But we don't want to have to write a new domain-specific planning system for each problem!
- *Domain-configurable* planning algorithm
  - Domain-independent search engine
  - Input includes domain control knowledge for the domain

# What is Domain Control Knowledge (DCK)

- Domain specific constraints on the space of possible plans.
- Some might add that they serve to guide the planner towards more efficient search, but of course they all do this trivially by forcing or disallowing the occurrence of certain actions within a plan.
- Generally given by a domain expert at the time of domain encoding, but can also be learned automatically. (E.g., see DiscoPlan by Gereni et al.)
- Can we differentiate domain-control knowledge from temporally extended goals, state constraints or invariants? (Let's revisit this at the end of the talk.)

# Types of DCK

- Not all DCK is created equal. The language used for DCK as well as the way it is applied (often within a special-purpose planner or interpreter) distinguish the different approaches to DCK
- Here we distinguish *state-centric* from *action-centric* DCK
  - Control Rules (TLPlan [Bacchus & Kabanza, 00], TALPlan [Doherty et al, 00]) support **state-centric** DCK
  - HTN and Golog both support different forms of **action-centric** and **some state-centric** DCK

Note that one is representable in terms of the other. How?

# Advantages and Disadvantages

- + (Perhaps not surprisingly) well-crafted DCK can cause planners to outperform the best planners, today. It is an effective method of creating a planning system, when DCK exists and can be elicited.
- Creation of DCK can require arduous hand-coding by human expert
- + Often domain specific but problem independent
- DCK generally requires special-purpose machinery for processing, and thus can't easily exploit advances in planning (But see [Baier et al, ICAPS07] and [Fritz et al, KR08] for a possible way around this)
- +/- Some people feel that DCK is “cheating” in some way (silly)!



# Outline

- Domain Control Knowledge
- ➔ Control Rules: TLPlan
- Procedural DCK: Hierarchical Task Networks
- Procedural DCK: Golog

# Control Rules (TLPlan, TALPlan, and the like)

- Discussion here predominantly based on TLPlan [Bacchus & Kabanza 2000]
- Language for writing domain-specific pruning rules:
  - E.g., Linear Temporal Logic – a temporal modal logic
- Domain-configurable planning algorithm
  - Input is augmented by control rules

# Linear Temporal Logic (LTL)

Modal logic: formal logic plus *modal operators*

to express concepts that would be difficult to express within propositional or first-order logic

Linear Temporal Logic (LTL):

- (first-order) logic extended with modalities for time (and for “goal” here)
  - Purpose: to express a limited notion of time
    - An infinite sequence  $\langle 0, 1, 2, \dots \rangle$  of time instants
    - An infinite sequence  $M = \langle s_0, s_1, \dots \rangle$  of states of the world
  - Modal operators to refer to the states in which formulas are true:
    - $\bigcirc f$  - *next f* -  $f$  holds in the next state, e.g.,  $\bigcirc on(A,B)$
    - $\diamond f$  - *eventually f* -  $f$  either holds now or in some future state
    - $\square f$  - *always f* -  $f$  holds now and in all future states
    - $f_1 U f_2$  -  *$f_1$  until  $f_2$*  -  $f_2$  either holds now or in some future state, and  $f_1$  holds until then
  - Propositional constant symbols TRUE and FALSE

# Linear Temporal Logic (continued)

- Quantifiers cause problems with computability
  - Suppose  $f(x)$  is true for infinitely many values of  $x$
  - Problem evaluating truth of  $\forall x f(x)$  and  $\exists x f(x)$
- Bounded quantifiers
  - Let  $g(x)$  be such that  $\{x : g(x)\}$  is finite and easily computed
    - $\forall[x:g(x)] f(x)$ 
      - means  $\forall x (g(x) \Rightarrow f(x))$
      - expands into  $f(x_1) \wedge f(x_2) \wedge \dots \wedge f(x_n)$
    - $\exists[x:g(x)] f(x)$ 
      - means  $\exists x (g(x) \wedge f(x))$
      - expands into  $f(x_1) \vee f(x_2) \vee \dots \vee f(x_n)$

# Models for LTL

- A model is a triple  $(M, s_i, v)$ 
  - $M = \langle s_0, s_1, \dots \rangle$  is a sequence of states
  - $s_i$  is the  $i$ 'th state in  $M$ ,
  - $v$  is a *variable assignment* function
    - a substitution that maps all variables into objects in the domain of discourse
- Write  $(M, s_i, v) \models f$   
to mean that  $v(f)$  is true in  $s_i$
- Always require that
$$(M, s_i, v) \models \text{TRUE}$$
$$(M, s_i, v) \models \neg \text{FALSE}$$

# Examples

- Suppose  $M = \langle s_0, s_1, \dots \rangle$

$(M, s_0, v) \models \Box\Box on(A, B)$  means  $A$  is on  $B$  in  $s_2$

- Abbreviations:

$(M, s_0) \models \Box\Box on(A, B)$  no free variables, so  $v$  is irrelevant:

$M \models \Box\Box on(A, B)$  if we omit the state, it defaults to  $s_0$

- Equivalently,

$(M, s_2, v) \models on(A, B)$  same meaning w/o modal operators

$s_2 \models on(A, B)$  same thing in ordinary FOL

- $M \models \Box\neg holding(C)$

- in every state in  $M$ , we aren't holding  $C$

- $M \models \Box(on(B, C) \Rightarrow (on(B, C) U on(A, B)))$

- whenever we enter a state in which  $B$  is on  $C$ ,  $B$  remains on  $C$  until  $A$  is on  $B$ .

# Linear Temporal Logic (continued)

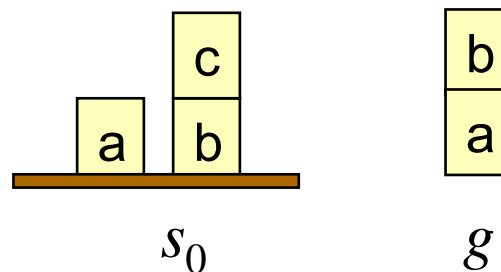
- Augment the models to include a set of *goal* states  $g$
- $\text{GOAL}(f)$  - says  $f$  is true in every  $s$  in  $g$ 
  - $((M, s_i, v), g) \models \text{GOAL}(f)$  iff  $(M, s_i, v) \models f$  for every  $s_i \in g$

# Blocks World - Example

- Blocks-world operators:

Operator	Preconditions and Deletes	Adds
$pickup(x)$	$ontable(x), clear(x), handempty.$	$holding(x).$
$putdown(x)$	$holding(x).$	$ontable(x), clear(x), handempty.$
$stack(x, y)$	$holding(x), clear(y).$	$on(x, y), clear(x), handempty.$
$unstack(x, y)$	$on(x, y), clear(x), handempty.$	$holding(x), clear(y).$

A planning problem:





# Supporting Axioms

- Want to define conditions under which a stack of blocks will never need to be moved
- If  $x$  is the top of a stack of blocks, then we want  $goodtower(x)$  to hold if
  - $x$  doesn't need to be anywhere else
  - None of the blocks below  $x$  need to be anywhere else
- Definitions to support this:
  - $goodtower(x) \Leftrightarrow clear(x) \wedge \neg GOAL(holding(x)) \wedge goodtowerbelow(x)$
  - $goodtowerbelow(x) \Leftrightarrow$ 
$$[ontable(x) \wedge \neg \exists [y:GOAL(on(x,y))]]$$
$$\vee \exists [y:on(x,y)] \{ \neg GOAL(ontable(x)) \wedge \neg GOAL(holding(y))$$
$$\wedge \neg GOAL(clear(y)) \wedge \forall [z:GOAL(on(x,z))] (z = y)$$
$$\wedge \forall [z:GOAL(on(z,y))] (z = x) \wedge$$
$$goodtowerbelow(y) \}$$
  - $badtower(x) \Leftrightarrow clear(x) \wedge \neg goodtower(x)$

# Blocks World Example (continued)

Three different control formulas:

(1) Every goodtower must always remain a goodtower:

$$\square \left( \forall [x:clear(x)] goodtower(x) \Rightarrow \bigcirc (clear(x) \vee \exists [y:on(y, x)] goodtower(y)) \right)$$

(2) Like (1), but also says never to put anything onto a badtower:

$$\square \left( \forall [x:clear(x)] goodtower(x) \Rightarrow \bigcirc (clear(x) \vee \exists [y:on(y, x)] goodtower(y)) \right. \\ \left. \wedge badtower(x) \Rightarrow \bigcirc (\neg \exists [y:on(y, x)]) \right)$$

(3) Like (2), but also says never to pick up a block from the table unless you can put it onto a goodtower:

$$\square \left( \forall [x:clear(x)] goodtower(x) \Rightarrow \bigcirc (clear(x) \vee \exists [y:on(y, x)] goodtower(y)) \right. \\ \left. \wedge badtower(x) \Rightarrow \bigcirc (\neg \exists [y:on(y, x)]) \right. \\ \left. \wedge (ontable(x) \wedge \exists [y:GOAL(on(x, y))] \neg goodtower(y)) \right. \\ \left. \Rightarrow \bigcirc (\neg holding(x)) \right)$$

# How TLPlan Works

- Nondeterministic forward state-space search
- Input includes a current state  $s_0$  and a control formula  $f_0$  for  $s_0$
- If  $f_0$  contains no temporal operators then we can tell immediately whether  $s_0$  satisfies  $f_0$ 
  - If it doesn't then this path is unsatisfactory, so backtrack
- If  $f_0$  contains temporal operators, then the only way  $s_0$  satisfies  $f_0$  is if  $s_0$  is part of a sequence  $M = \langle s_0, s_1, \dots \rangle$  that satisfies  $f_0$
- To tell this, need to look at the next state  $s_1$ 
  - $s_1$  may be any state  $\gamma(s_0, a)$  such that  $a$  is applicable to  $s_0$
- From  $s_0$  and  $f_0$ , compute a control formula  $f_1$  for  $s_1$ 
  - $f_1$  is a formula that *must* be true in  $s_1$  in order for  $f_0$  to be true in  $s_0$
  - Call TLPlan recursively on  $s_1$  and  $f_1$

## Procedure Progress ( $f, s$ )

### Case

1.  $f$  contains no temporal operators:

$$f^+ := \text{TRUE if } s \models f, \text{ FALSE otherwise.}$$

2.  $f = f_1 \wedge f_2$ :  $f^+ := \text{Progress}(f_1, s) \wedge \text{Progress}(f_2, s)$

3.  $f = \neg f_1$ :  $f^+ := \neg \text{Progress}(f_1, s)$

4.  $f = \bigcirc f_1$ :  $f^+ := f_1$

5.  $f = f_1 \cup f_2$ :  $f^+ := \text{Progress}(f_2, s) \vee (\text{Progress}(f_1, s) \wedge f)$

6.  $f = \diamond f_1$ :  $f^+ := \text{Progress}(f_1, s) \vee f$

7.  $f = \square f_1$ :  $f^+ := \text{Progress}(f_1, s) \wedge f$

8.  $f = \forall[x:g(x)] f_1$ :  $f^+ := \bigwedge \{ \text{Progress}(\theta f_1, s) : s \models g(c) \}$

9.  $f = \exists[x:g(x)] f_1$ :  $f^+ := \bigvee \{ \text{Progress}(\theta f_1, s) : s \models g(c) \}$

where  $\theta = \{x \leftarrow c\}$

### Boolean simplification rules:

1.  $[\text{FALSE} \wedge \phi \mid \phi \wedge \text{FALSE}] \mapsto \text{FALSE}$ ,

3.  $\neg \text{TRUE} \mapsto \text{FALSE}$ ,

2.  $[\text{TRUE} \wedge \phi \mid \phi \wedge \text{TRUE}] \mapsto \phi$ ,

4.  $\neg \text{FALSE} \mapsto \text{TRUE}$ .

# Examples

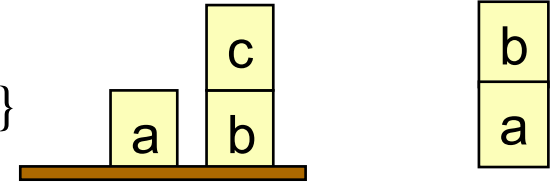
- Suppose  $f = \Box on(a,b)$ 
  - $f^+ = \text{Progress}(on(a,b), s) \wedge \Box on(a,b)$
  - If  $on(a,b)$  is true in  $s$  then
    - $f^+ = \text{TRUE} \wedge \Box on(a,b)$
    - simplifies to  $\Box on(a,b)$
  - If  $on(a,b)$  is false in  $s$  then
    - $f^+ = \text{FALSE} \wedge \Box on(a,b)$
    - simplifies to  $\text{FALSE}$
- Summary:
  - $\Box$  generates a test on the current state
  - If the test succeeds,  $\Box$  propagates it to the next state

# Examples (continued)

- Suppose  $f = \Box(\text{on}(a,b) \Rightarrow \text{Oclear}(a))$ 
  - $f^+ = \text{Progress}[\Box(\text{on}(a,b) \Rightarrow \text{Oclear}(a)), s]$
  - $= \text{Progress}[\text{on}(a,b) \Rightarrow \text{Oclear}(a), s] \wedge \Box(\text{on}(a,b) \Rightarrow \text{Oclear}(a))$
  - If  $\text{on}(a,b)$  is true in  $s$ , then
    - $f^+ = \text{clear}(a) \wedge \Box(\text{on}(a,b) \Rightarrow \text{Oclear}(a))$ 
      - Since  $\text{on}(a,b)$  is true in  $s$ ,  $s^+$  must satisfy  $\text{clear}(a)$
      - The “always” constraint is propagated to  $s^+$
  - If  $\text{on}(a,b)$  is false in  $s$ , then
    - $f^+ = \Box(\text{on}(a,b) \Rightarrow \text{Oclear}(a))$ 
      - The “always” constraint is propagated to  $s^+$

# Example

- $s = \{ontable(a), ontable(b), clear(a), clear(c), on(c,b)\}$
- $g = \{on(b, a)\} \square$
- $f = \square \forall [x:clear(x)] \{ (ontable(x) \wedge \neg \exists [y:GOAL(on(x,y))]) \Rightarrow O_{\neg}holding(x) \}$ 
  - never pick up a block  $x$  if  $x$  is not required to be on another block  $y$
- $f^+ = Progress(f,s) \wedge f$
- $Progress(f,s)$ 
  - =  $Progress(\forall [x:clear(x)] \{ (ontable(x) \wedge \neg \exists [y:GOAL(on(x,y))]) \Rightarrow O_{\neg}holding(x) \}, s)$
  - =  $Progress((ontable(a) \wedge \neg \exists [y:GOAL(on(a,y))]) \Rightarrow O_{\neg}holding(a)), s)$
  - $\wedge Progress((ontable(b) \wedge \neg \exists [y:GOAL(on(b,y))]) \Rightarrow O_{\neg}holding(b)), s)$
  - =  $\neg holding(a) \wedge TRUE$
- $f^+ = \neg holding(a) \wedge TRUE \wedge f$ 
  - =  $\neg holding(a) \wedge$
  - $\square \forall [x:clear(x)] \{ (ontable(x) \wedge \neg \exists [y:GOAL(on(x,y))]) \Rightarrow O_{\neg}holding(x) \}$



# Pseudocode for TLPlan

- Nondeterministic forward search
  - Input includes a control formula  $f$  for the current state  $s$
  - When we expand a state  $s$ , we progress its formula  $f$  through  $s$
  - If the progressed formula is false,  $s$  is a dead-end
  - Otherwise the progressed formula is the control formula for  $s$ 's children

```
Procedure TLPlan ( $s, f, g, \pi$ )  
   $f^+ \leftarrow$  Progress ( $f, s$ )  
  if  $f^+ = \text{FALSE}$  then return failure  
  if  $s$  satisfies  $g$  then return  $\pi$   
   $A \leftarrow$  {actions applicable to  $s$ }  
  if  $A = \text{empty}$  then return failure  
  nondeterministically choose  $a \in A$   
   $s^+ \leftarrow \gamma(s, a)$   
  return TLPlan ( $s^+, f^+, g, \pi.a$ )
```



# Performance of Planners at IPC

- 2000 International Planning Competition
  - TALplanner: same kind of algorithm, different temporal logic
    - received the top award for a “hand-tailored” (i.e., domain-configurable) planner
- TLPlan won the same award in the 2002 International Planning Competition
- Both of them:
  - Ran several orders of magnitude faster than the “fully automated” (i.e., domain-independent) planners
    - especially on large problems
  - Solved problems on which the domain-independent planners ran out of time/memory.

# Beyond TLPlan: HPlan-P

- One disadvantage to TLPlan is that it is a forward search planner, providing no guidance towards achievement of the goal. Its strong performance is largely based on
  - the strength of the pruning,
  - the fact that it does not ground all actions prior to planning.
- In 2007, Baier et al. developed an extension to TLPlan that added heuristic search. This was made possible by a clever compilation scheme that compiles LTL formulae into nondeterministic finite state automata, whose accepting conditions are equivalent to satisfaction of the formula. This heuristic search was used for both preference-based planning as well as planning with so-called temporally extended goals.

# Outline

- Domain Control Knowledge
- Control Rules: TLPlan
- ➔ Procedural DCK: Hierarchical Task Networks
- Procedural DCK: Golog

# HTN Motivation

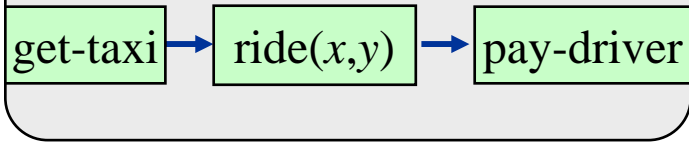
- We may already have an idea how to go about solving problems in a planning domain
- Example: travel to a destination that's far away:
  - Domain-independent planner:
    - many combinations of vehicles and routes
  - Experienced human: small number of “recipes”  
e.g., flying:
    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination
- How to enable planning systems to make use of such recipes?

# Two Approaches

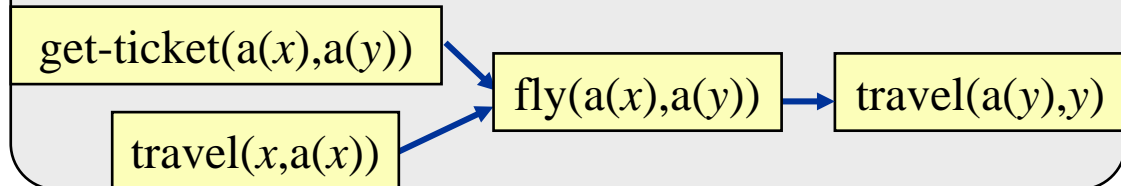
- Write rules to prune every action that *doesn't* fit the recipe
  - Control Rules  
(e.g., TLPlan, TALPlan)
- Describe the actions (and subtasks) that do fit the recipe
  - Procedural DCK  
(e.g, Golog, Hierarchical Task Network (HTN) planning)

*Task:* travel(x,y)

*Method:* taxi-travel(x,y)



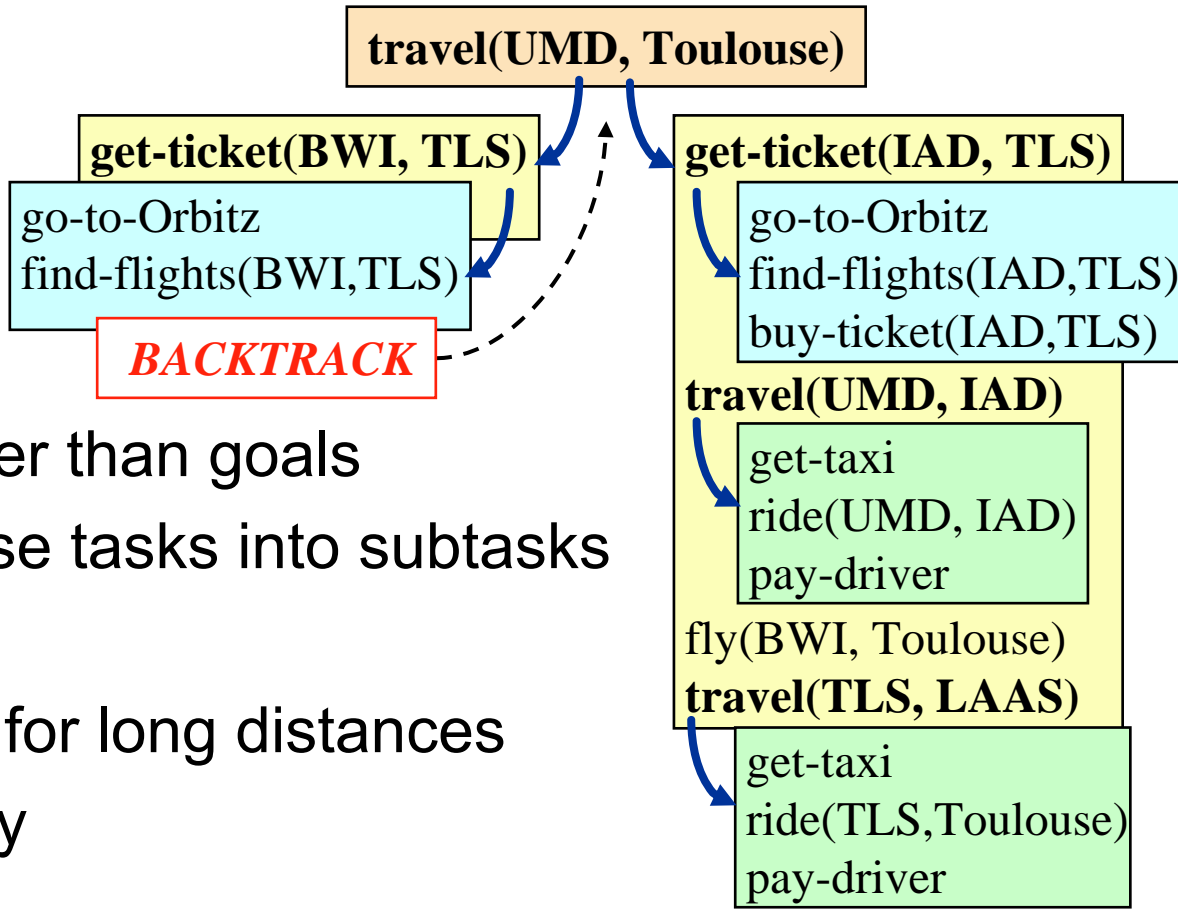
*Method:* air-travel(x,y)



# HTN Planning

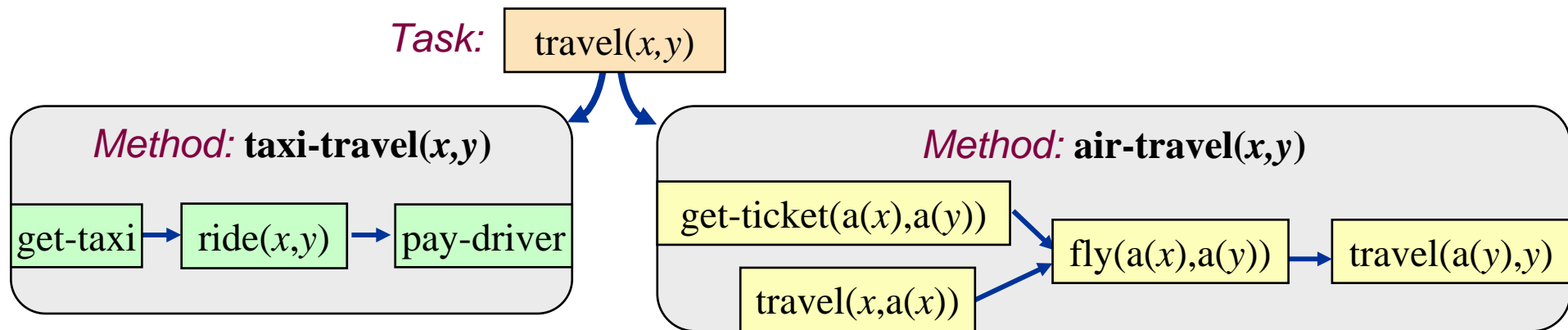
Problem reduction:

- *Tasks* (activities) rather than goals
- *Methods* to decompose tasks into subtasks
- Enforce constraints
  - E.g., taxi not good for long distances
- Backtrack if necessary



# HTN Planning

- HTN planners may be domain-specific
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    - domain description, initial state, initial task network



# Simple Task Network (STN) Planning

- A special case of HTN planning
- States and operators
  - The same as in classical planning
- *Task*: an expression of the form  $t(u_1, \dots, u_n)$ 
  - $t$  is a *task symbol*, and each  $u_i$  is a term
  - Two kinds of task symbols (and tasks):
    - *primitive*: tasks that we know how to execute directly
      - task symbol is an operator name
    - *nonprimitive*: tasks that must be decomposed into subtasks
      - use *methods* (next slide)

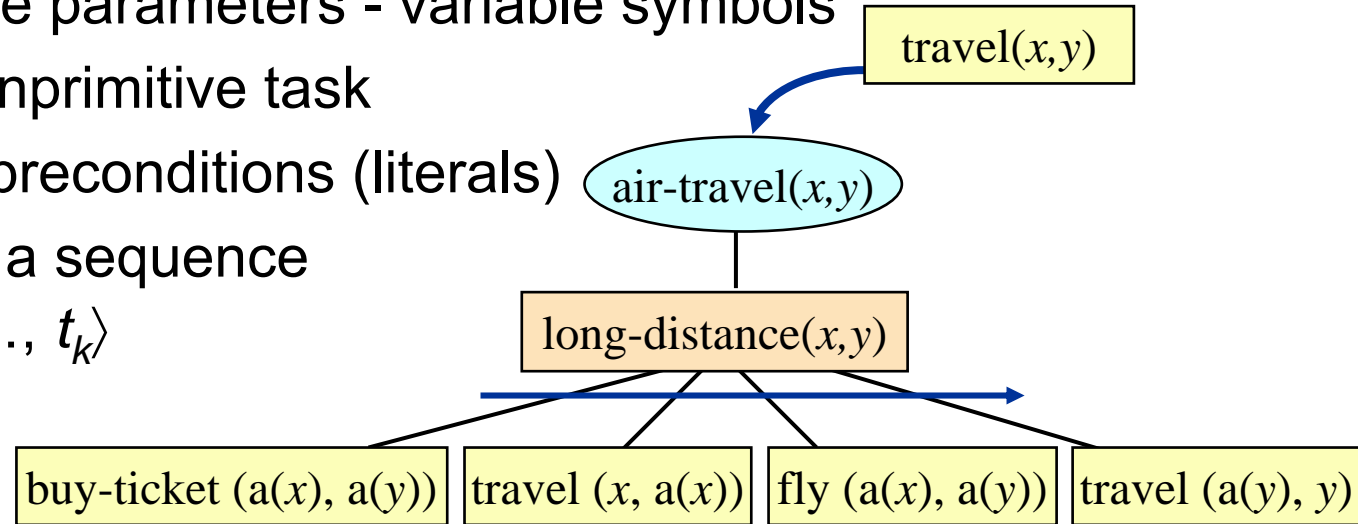


# Methods

- **Totally ordered method:** a 4-tuple

$$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m))$$

- $\text{name}(m)$ : an expression of the form  $n(x_1, \dots, x_n)$ 
  - $x_1, \dots, x_n$  are parameters - variable symbols
- $\text{task}(m)$ : a nonprimitive task
- $\text{precond}(m)$ : preconditions (literals)
- $\text{subtasks}(m)$ : a sequence of tasks  $\langle t_1, \dots, t_k \rangle$



$\text{air-travel}(x,y)$

*task:*  $\text{travel}(x,y)$

*precond:*  $\text{long-distance}(x,y)$

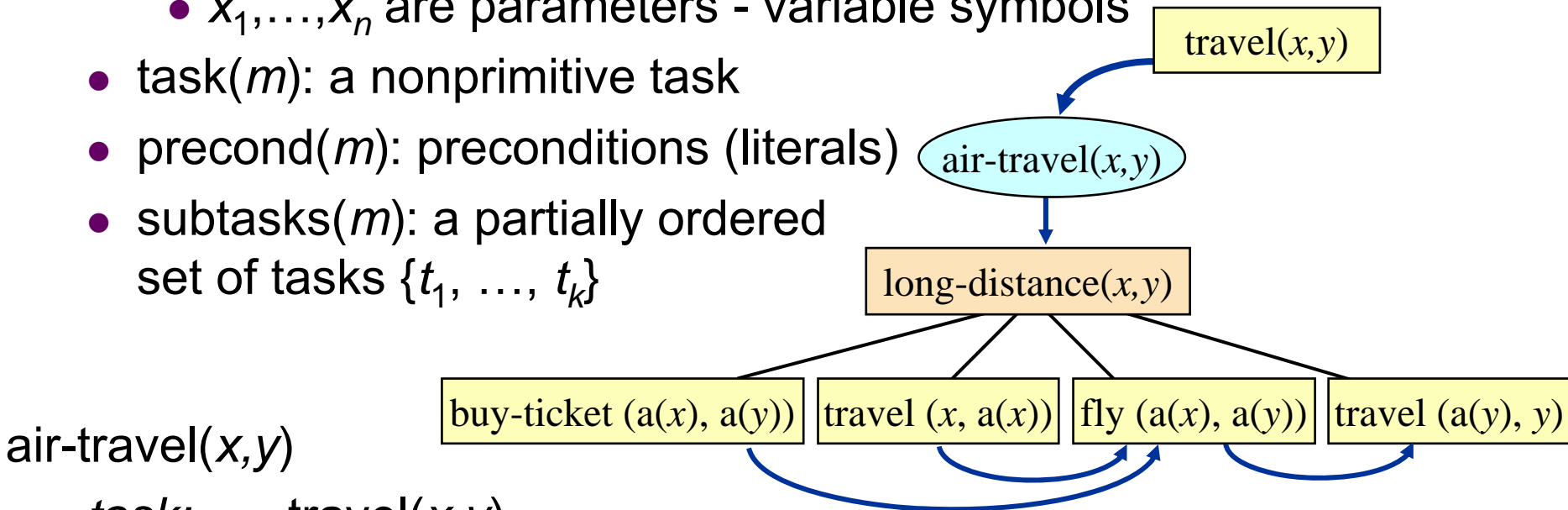
*subtasks:*  $\langle \text{buy-ticket}(a(x), a(y)), \text{travel}(x, a(x)), \text{fly}(a(x), a(y)), \text{travel}(a(y), y) \rangle$

# Methods (Continued)

- Partially ordered method: a 4-tuple

$$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m))$$

- $\text{name}(m)$ : an expression of the form  $n(x_1, \dots, x_n)$ 
  - $x_1, \dots, x_n$  are parameters - variable symbols
- $\text{task}(m)$ : a nonprimitive task
- $\text{precond}(m)$ : preconditions (literals)
- $\text{subtasks}(m)$ : a partially ordered set of tasks  $\{t_1, \dots, t_k\}$




air- $\text{travel}(x,y)$

*task*:  $\text{travel}(x,y)$

*precond*:  $\text{long-distance}(x,y)$

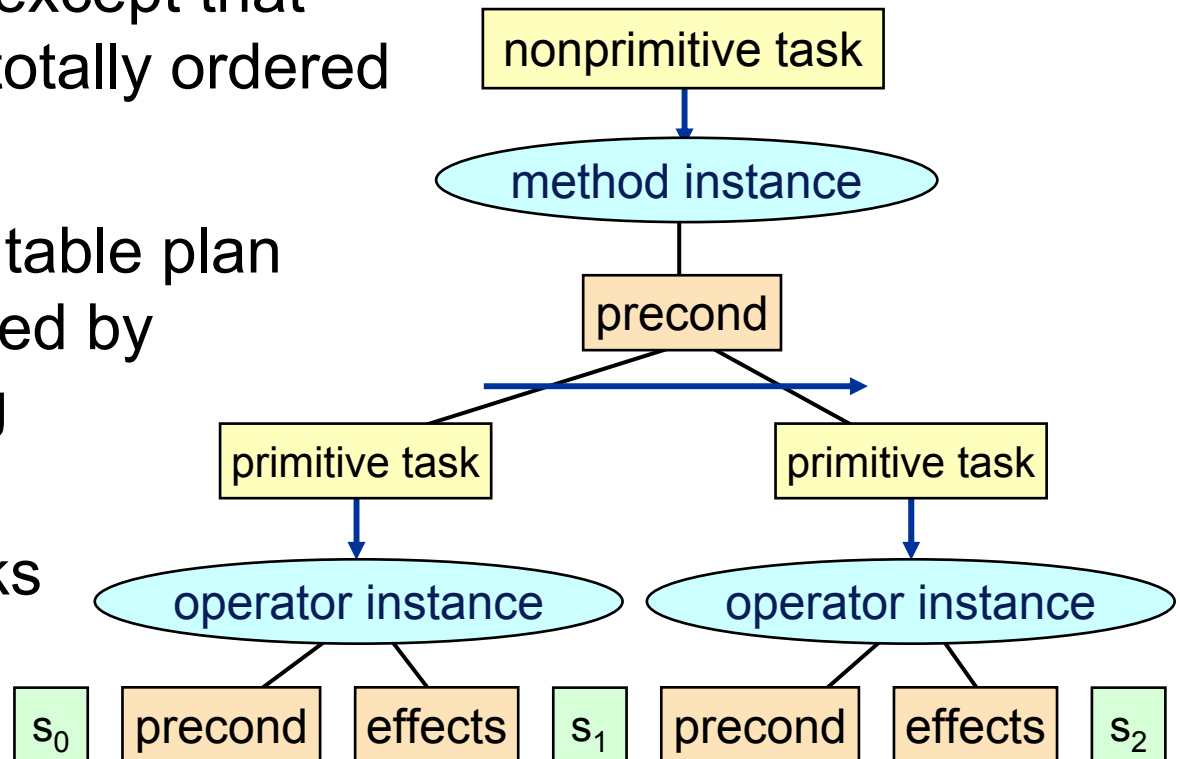
*network*:  $u_1 = \text{buy-ticket}(a(x), a(y)), u_2 = \text{travel}(x, a(x)),$   
 $u_3 = \text{fly}(a(x), a(y)), u_4 = \text{travel}(a(y), y),$   
 $\{(u_1, u_3), (u_2, u_3), (u_3, u_4)\}$

# Domains, Problems, Solutions

- STN planning domain: methods, operators
- STN planning problem: methods, operators, initial state, task list ~goal 
- Total-order STN planning domain and planning problem:
  - Same as above except that all methods are totally ordered

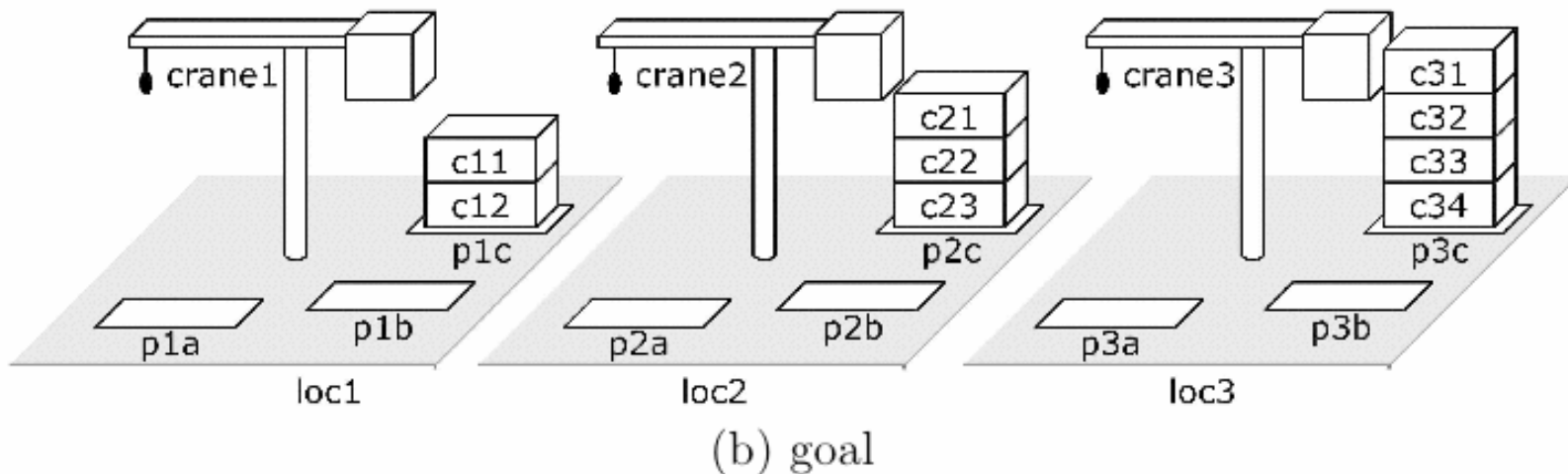
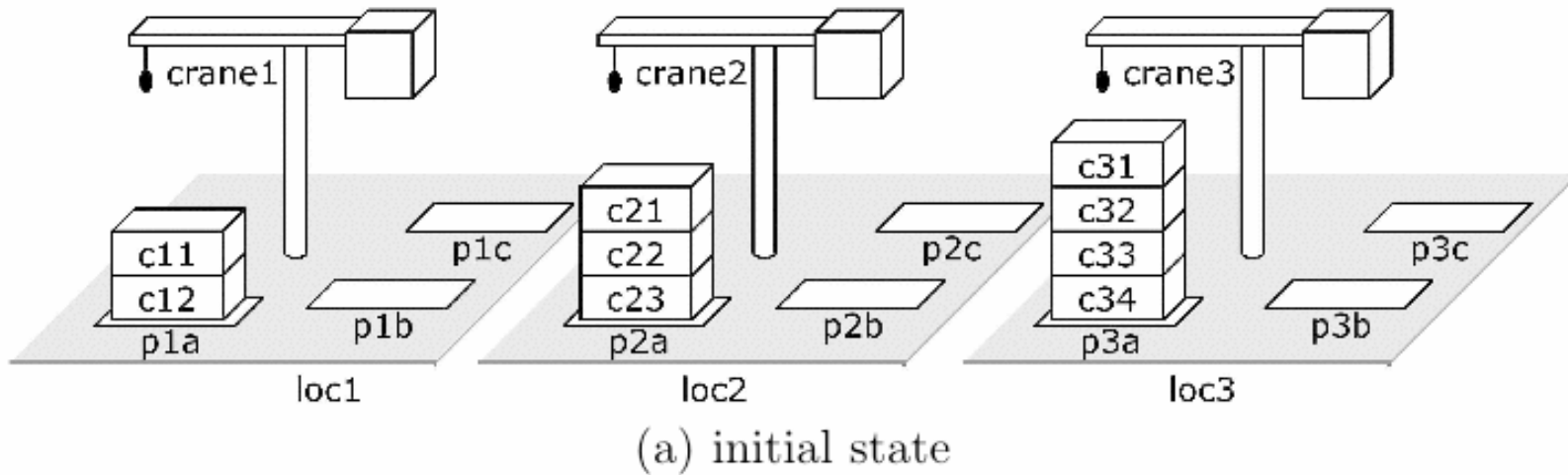
- Solution: any executable plan that can be generated by recursively applying

- methods to nonprimitive tasks
- operators to primitive tasks



# Example

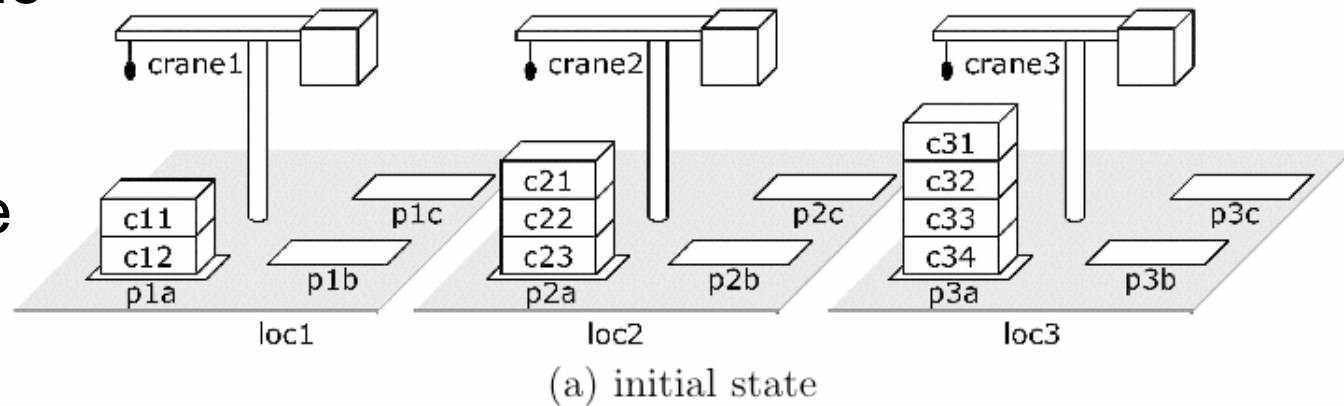
- Suppose we want to move three stacks of containers in a way that preserves the order of the containers



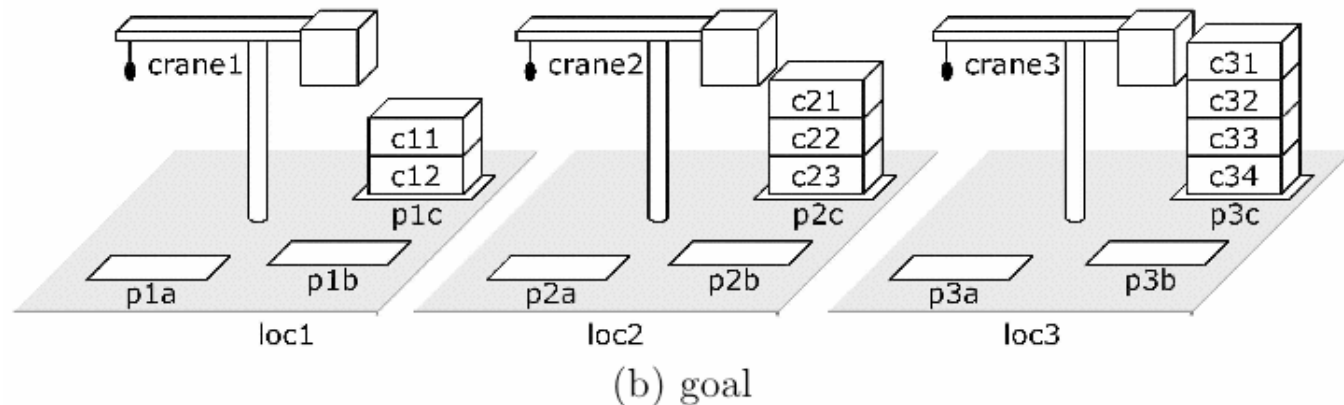
# Example (continued)

- A way to move each stack:

- first move the containers from  $p$  to an intermediate pile  $r$



- then move them from  $r$  to  $q$



# Partial-Order Formulation

take-and-put( $c, k, l_1, l_2, p_1, p_2, x_1, x_2$ ):

task: move-topmost-container( $p_1, p_2$ )

precond: top( $c, p_1$ ), on( $c, x_1$ ), ; true if  $p_1$  is not empty  
attached( $p_1, l_1$ ), belong( $k, l_1$ ), ; bind  $l_1$  and  $k$   
attached( $p_2, l_2$ ), top( $x_2, p_2$ ) ; bind  $l_2$  and  $x_2$

subtasks:  $\langle$ take( $k, l_1, c, x_1, p_1$ ), put( $k, l_2, c, x_2, p_2$ ) $\rangle$

recursive-move( $p, q, c, x$ ):

task: move-stack( $p, q$ )

precond: top( $c, p$ ), on( $c, x$ ) ; true if  $p$  is not empty

subtasks:  $\langle$ move-topmost-container( $p, q$ ), move-stack( $p, q$ ) $\rangle$   
;; the second subtask recursively moves the rest of the stack

do-nothing( $p, q$ )

task: move-stack( $p, q$ )

precond: top( $pallet, p$ ) ; true if  $p$  is empty

subtasks:  $\langle$  ; no subtasks, because we are done

move-each-twice()

task: move-all-stacks()

precond: ; no preconditions

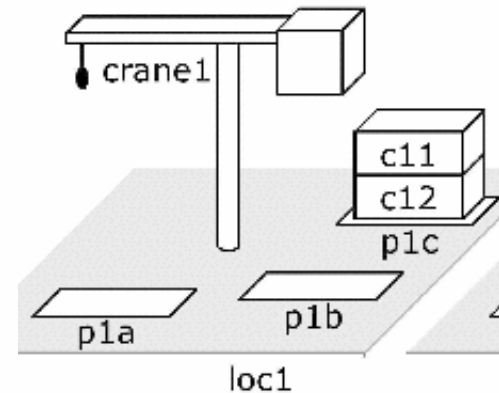
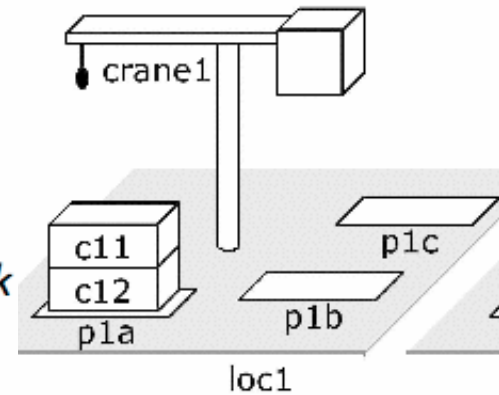
network: ; move each stack twice:

$u_1 = \text{move-stack}(p1a, p1b)$ ,  $u_2 = \text{move-stack}(p1b, p1c)$ ,

$u_3 = \text{move-stack}(p2a, p2b)$ ,  $u_4 = \text{move-stack}(p2b, p2c)$ ,

$u_5 = \text{move-stack}(p3a, p3b)$ ,  $u_6 = \text{move-stack}(p3b, p3c)$ ,

$\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$



take-and-put( $c, k, l_1, l_2, p_1, p_2, x_1, x_2$ ):

task: move-topmost-container( $p_1, p_2$ )

precond: top( $c, p_1$ ), on( $c, x_1$ ), ; true if  $p_1$  is not empty  
attached( $p_1, l_1$ ), belong( $k, l_1$ ), ; bind  $l_1$  and  $k$   
attached( $p_2, l_2$ ), top( $x_2, p_2$ ) ; bind  $l_2$  and  $x_2$

subtasks:  $\langle$ take( $k, l_1, c, x_1, p_1$ ), put( $k, l_2, c, x_2, p_2$ ) $\rangle$

recursive-move( $p, q, c, x$ ):

task: move-stack( $p, q$ )

precond: top( $c, p$ ), on( $c, x$ ) ; true if  $p$  is not empty

subtasks:  $\langle$ move-topmost-container( $p, q$ ), move-stack( $p, q$ ) $\rangle$   
;; the second subtask recursively moves the rest of the stack

do-nothing( $p, q$ )

task: move-stack( $p, q$ )

precond: top( $pallet, p$ ) ; true if  $p$  is empty

subtasks:  $\langle$  ; no subtasks, because we are done

move-each-twice()

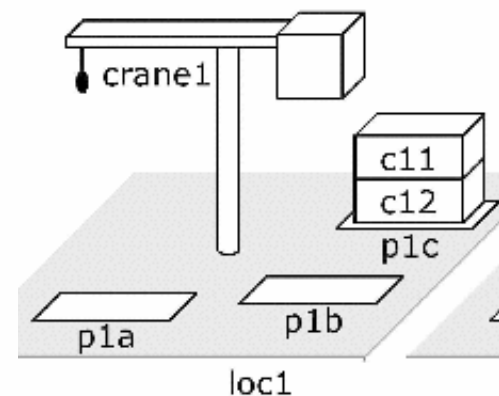
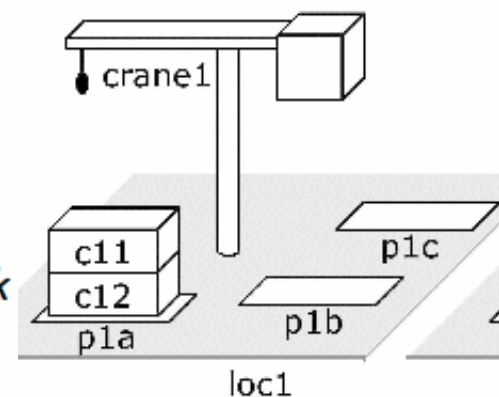
task: move-all-stacks()

precond: ; no preconditions

subtasks: ; move each stack twice:

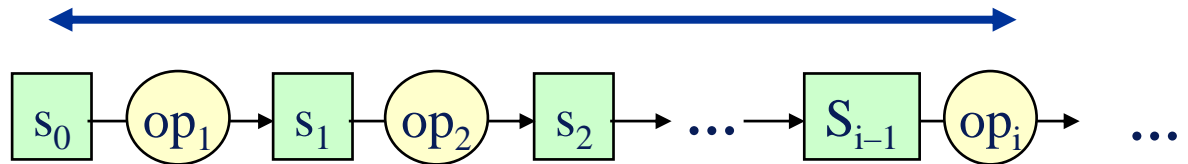
$\langle$ move-stack( $p1a, p1b$ ), move-stack( $p1b, p1c$ ),  
move-stack( $p2a, p2b$ ), move-stack( $p2b, p2c$ ),  
move-stack( $p3a, p3b$ ), move-stack( $p3b, p3c$ ) $\rangle$

# Total-Order Formulation

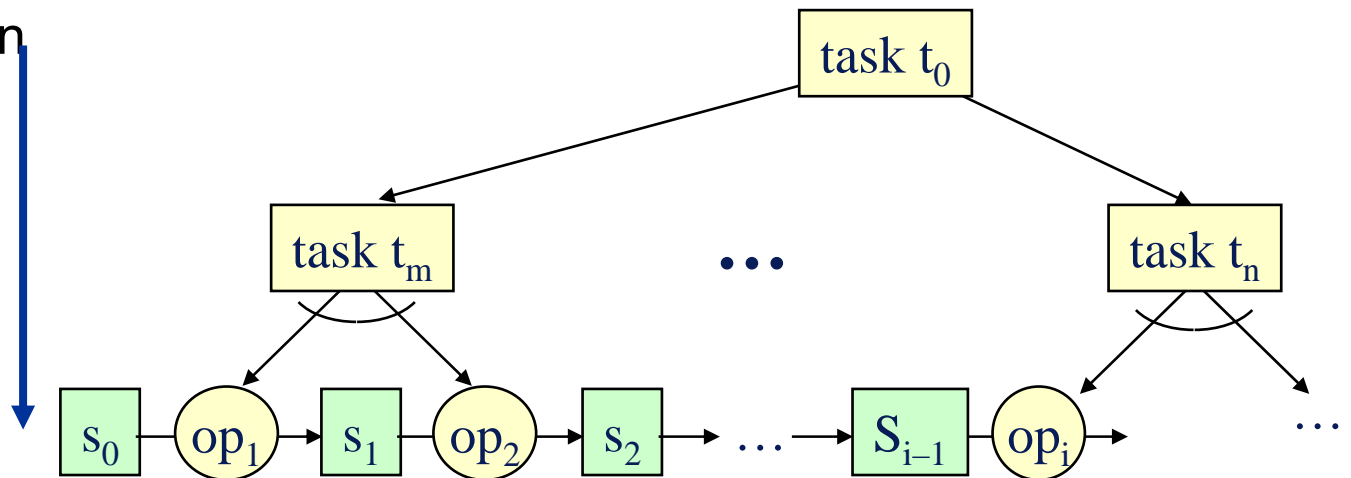


# Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward



- In HTN planning, there are *two* choices to make about direction:
  - forward or backward
  - up or down



- TFD\* goes *down* and *forward*

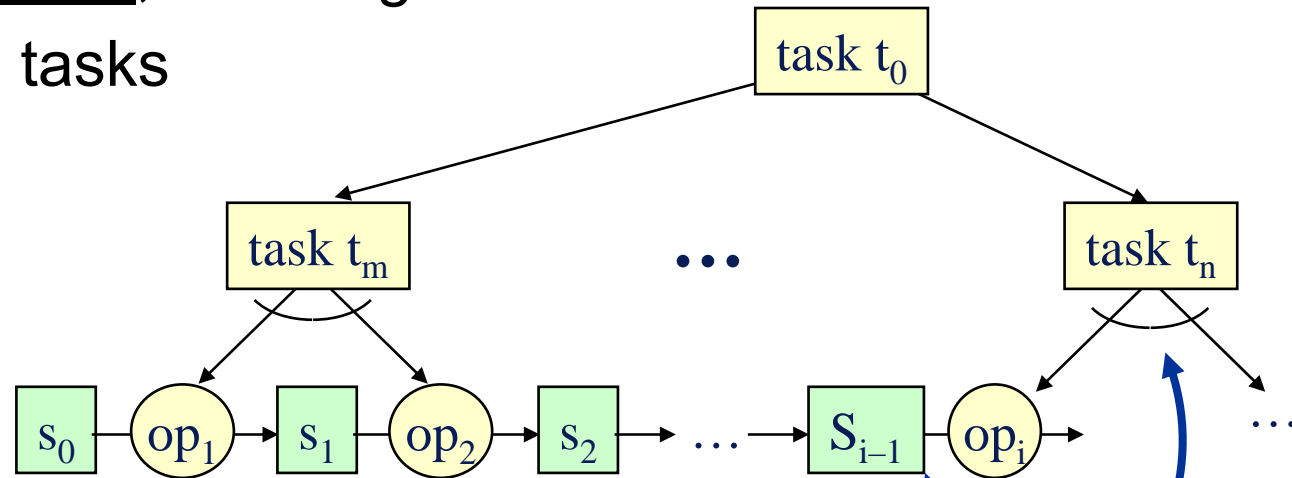
\* TFD = Total Order STN Planning



# Comparison to Forward & Backward Search

Like a backward search, TFD is goal-directed

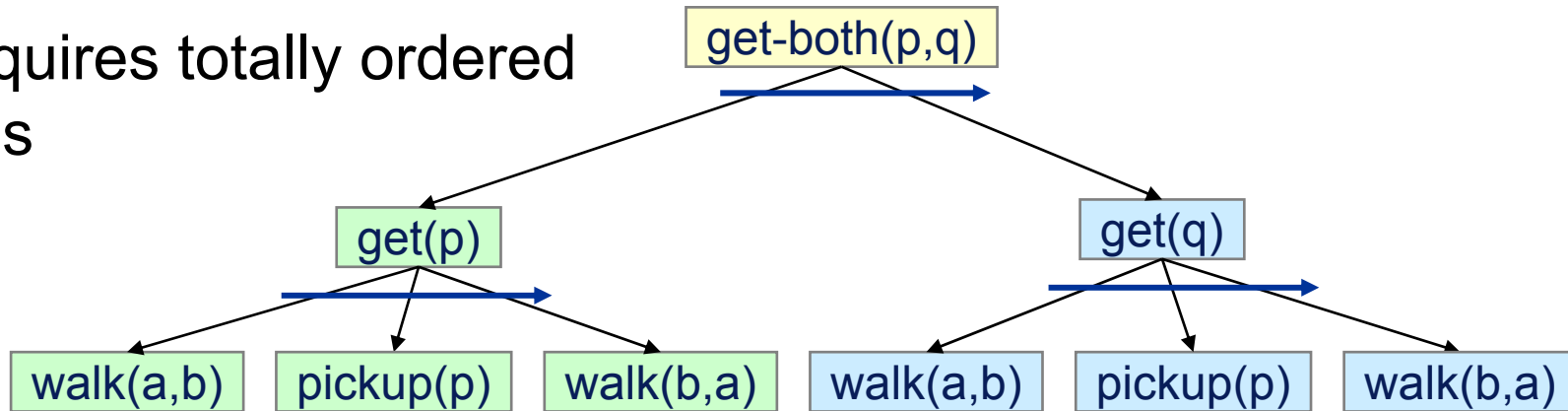
- Goals are the tasks



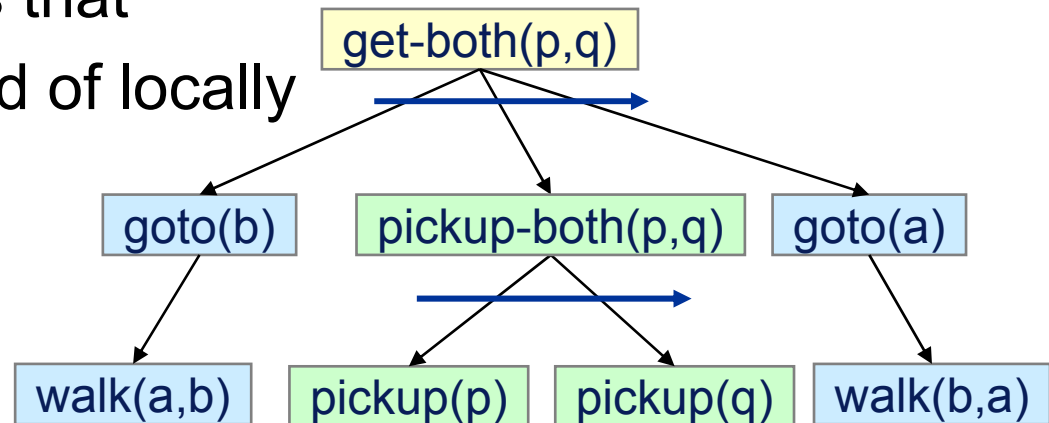
- Like a forward search, it generates actions in the same order in which they'll be executed.
- Whenever we want to plan the next task
  - we've already planned everything that comes before it
  - Thus, we know the current state of the world

# Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

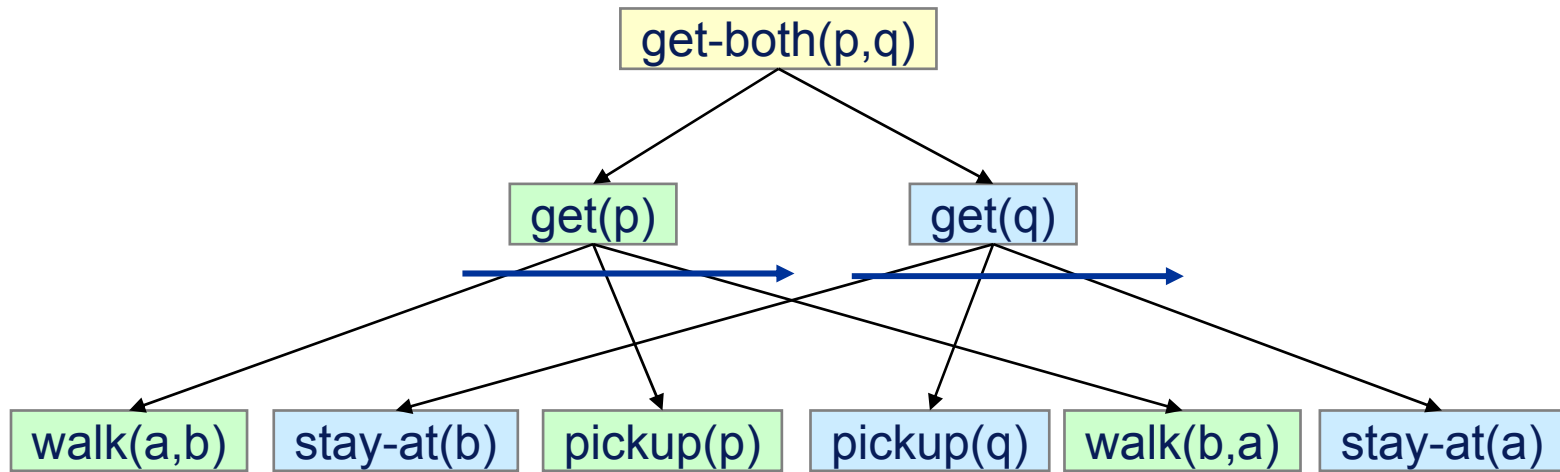


- Can't interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally



# Partially Ordered Methods

- With partially ordered methods, the subtasks can be interleaved



- Fits many planning domains better
- Requires a more complicated planning algorithm

# Comparison to Classical Planning

STN planning is strictly more expressive than classical planning

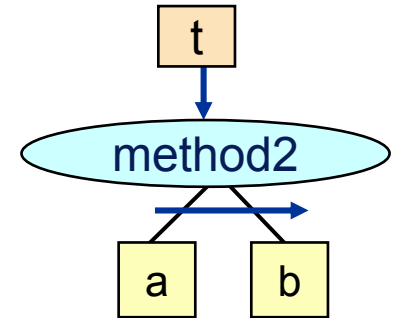
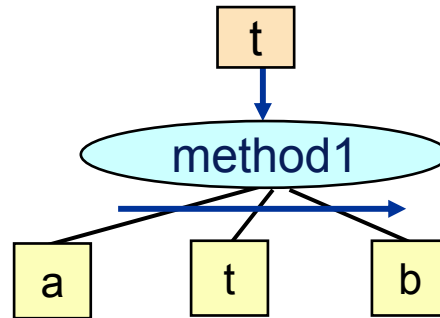
- Any classical planning problem can be translated into an ordered-task-planning problem in polynomial time
- Several ways to do this. One is roughly as follows:
  - For each goal or precondition  $e$ , create a task  $t_e$
  - For each operator  $o$  and effect  $e$ , create a method  $m_{o,e}$ 
    - Task:  $t_e$
    - Subtasks:  $t_{c_1}, t_{c_2}, \dots, t_{c_n}, o$ , where  $c_1, c_2, \dots, c_n$  are the preconditions of  $o$
    - Partial-ordering constraints: each  $t_{c_i}$  precedes  $o$
- *Etc.*
  - *E.g., how to handle deleted-condition interactions ...*

# Comparison to Classical Planning (cont.)

- Some STN planning problems are *not* expressible in classical planning

- Example:

- Two STN methods:
  - No arguments
  - No preconditions



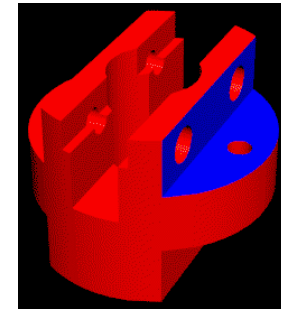
- Two operators, a and b
  - Again, no arguments and no preconditions
- Initial state is empty, initial task is t
- Set of solutions is  $\{a^n b^n \mid n > 0\}$
- No classical planning problem has this set of solutions
  - The state-transition system is a finite-state automaton
  - No finite-state automaton can recognize  $\{a^n b^n \mid n > 0\}$
- Can even express undecidable problems using STNs

# Increasing Expressivity Further

- Knowing the current state makes it easy to do things that would be difficult otherwise
  - States can be arbitrary data structures

Us:	East declarer, West dummy
Opponents:	defenders, South & North
Contract:	East – 3NT
On lead:	West at trick 3

East:	★KJ74
West:	★A2
Out:	★QT9865



- Preconditions and effects can include
  - logical inferences (e.g., Horn clauses)
  - complex numeric computations
  - interactions with other software packages
- e.g., SHOP and SHOP2:

<http://www.cs.umd.edu/projects/shop>

*method* travel-by-foot

precond:  $distance(x, y) \leq 2$

task:  $travel(a, x, y)$

subtasks:  $walk(a, x, y)$

*method* travel-by-taxi

task:  $travel(a, x, y)$

precond:  $cash(a) \geq 1.5 + 0.5 \times distance(x, y)$

subtasks:  $\langle call-taxi(a, x), ride(a, x, y), pay-driver(a, x, y) \rangle$

*operator* walk

precond:  $location(a) = x$

effects:  $location(a) \leftarrow y$

*operator* call-taxi( $a, x$ )

effects:  $location(taxi) \leftarrow x$

*operator* ride( $a, x$ )

precond:  $location(taxi) = x, location(a) = x$

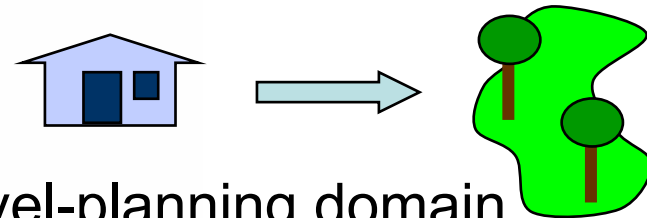
effects:  $location(taxi) \leftarrow y, location(a) \leftarrow y$

*operator* pay-driver( $a, x, y$ )

precond:  $cash(a) \geq 1.5 + 0.5 \times distance(x, y)$

effects:  $cash(a) \leftarrow cash(a) - 1.5 - 0.5 \times distance(x, y)$

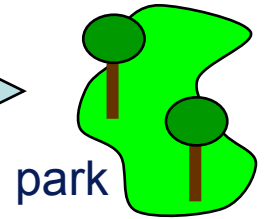
## Example



- Simple travel-planning domain
  - Go from one location to another
  - State-variable formulation

# Planning Problem:

I am at home, I have \$20,  
I want to go to a park 8 miles away



Initial task: `travel(me,home,park)`

`travel-by-foot`

`travel-by-taxi`

Precond:  $\text{distance}(\text{home},\text{park}) \leq 2$

Precond:  $\text{cash}(\text{me}) \geq 1.50 + 0.50 * \text{distance}(\text{home},\text{park})$

Precondition fails

Precondition succeeds

Decomposition into subtasks

Initial state

$s_0$

`call-taxi(me,home)`

Precond: ...  
Effects: ...

$s_1$

`ride(me,home,park)`

Precond: ...  
Effects: ...

$s_2$

`pay-driver(me,home,park)`

Precond: ...  
Effects: ...

Final state

$s_3$

$s_0 = \{\text{location}(\text{me})=\text{home}, \text{cash}(\text{me})=20, \text{distance}(\text{home},\text{park})=8\}$

$s_1 = \{\text{location}(\text{me})=\text{home}, \text{location}(\text{taxi})=\text{home}, \text{cash}(\text{me})=20, \text{distance}(\text{home},\text{park})=8\}$

$s_2 = \{\text{location}(\text{me})=\text{park}, \text{location}(\text{taxi})=\text{park}, \text{cash}(\text{me})=20, \text{distance}(\text{home},\text{park})=8\}$

$s_3 = \{\text{location}(\text{me})=\text{park}, \text{location}(\text{taxi})=\text{park}, \text{cash}(\text{me})=14.50, \text{distance}(\text{home},\text{park})=8\}$



# SHOP2

- SHOP2: implementation of PFD-like algorithm + generalizations
  - Won one of the top four awards at IPC 2002
  - Freeware, open source
  - Implementations in Lisp and Java available online

# HTN Planning

- HTN planning is even more general
  - Can have constraints associated with tasks and methods
    - Things that must be true before, during, or afterwards
- See GNT for further details

# SHOP & SHOP2 vs. TLPlan & TALplanner

- These planners have equivalent expressive power
- They know the current state at each point during the planning process, and use this to prune actions
  - Makes it easy to call external subroutines, do numeric computations, etc.
- Main difference: how the DCK is expressed and the pruning realized
  - SHOP and SHOP2: the methods say what *can* be done
    - Don't do anything unless a method says to do it
  - TLPlan and TALplanner: rules say what *cannot* be done
    - Try everything that the control rules don't prohibit
- Which approach is more convenient depends on the problem domain

# Domain-Configurable vs. Classical Planners

Disadvantage:

- writing DCK can be more complicated than just writing classical operators
- can't easily exploit advances in planning technology

Advantage:

- can encode “recipes” as collections of methods and operators
  - Express things that can't be expressed in classical planning
  - Specify standard ways of solving problems
    - Otherwise, the planning system would have to derive these again and again from “first principles,” every time it solves a problem
- Can speed up planning by many orders of magnitude

# Outline

- Domain Control Knowledge
- Control Rules: TLPlan
- Procedural DCK: Hierarchical Task Networks
- ➔ Procedural DCK: Golog

# Golog & ConGolog [Levesque et al, 97]

- Golog & ConGolog\* are agent programming languages based on the situation calculus .
- A Golog program can also be viewed as
  - an agent program
  - a plan sketch or plan skeleton, and/or
  - procedural DCK
- Important Feature: programs **non-determinism** (which enables search)

E.g.,

**if** *in(car,driveway)* **then** *walk* **else** *drive*

**while** ( $\exists$  *block*) *ontable(block)* **do** *remove\_a\_block* **endwhile**

**proc** *remove\_a\_block* (**pick**(*x*).*block(x)*) *pickup(x); putaway(x)*]

\*For simplicity we will henceforth only describe Golog. ConGolog extends Golog with constructs to deal with concurrency, interrupts, etc.

# Golog “Planning”

Analogy to planning follows (but the Golog implementation is more than a planner)

## Plan Domain and Plan Instance Description

- Plan Domain (preconditions, effects, etc.) described in situation calculus
- Initial State: formula in the situation calculus
- Goal:  $\delta$  - Golog program to be realized (much like the task in HTN)

## Plan Generation:

- Golog interpreter that effectively performs deductive plan synthesis following [Green, IJCAI-09]  
$$D \models \exists s'. Do(\delta, S_0, s')$$
- Golog interpreter is 20 lines of Prolog code!
- We discuss recent advances at the end (e.g., [Fritz et al., KR08])

# Situation Calculus [Reiter, 01] [McCarthy, 68] etc.

We appeal to the “Reiter axiomatization” of the situation calculus.

## Sorts:

Actions

e.g.,  $a$ ,  $bookTaxi(x)$

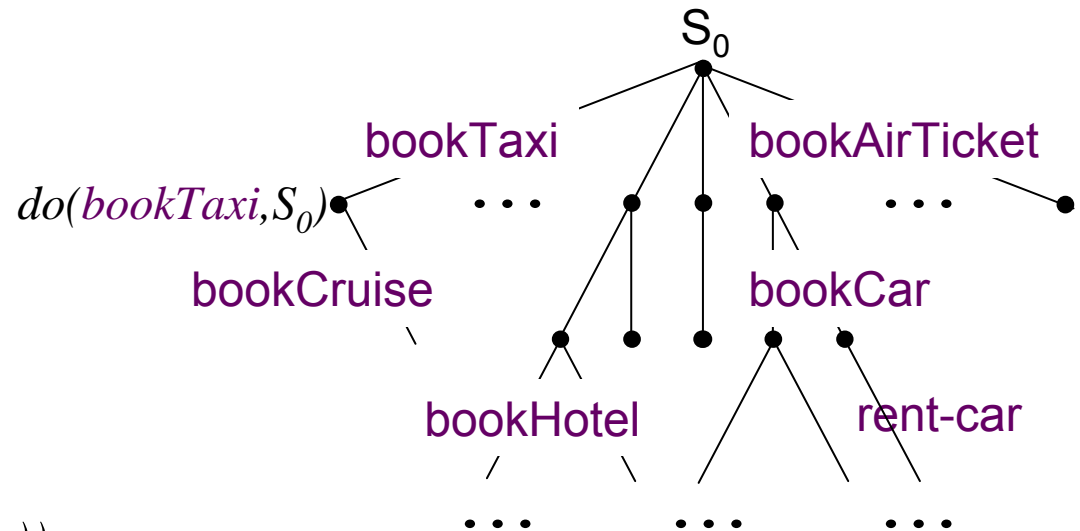
Situations

e.g.,  $s$ ,  $S_0$ ,

$do(bookTaxi(x),s)$

Fluents

e.g.,  $ownTicket(x, do(a,s))$





# Situation Calculus [Reiter, 01] [McCarthy, 68] etc.

A situation calculus theory  $D$  comprises the following axioms:

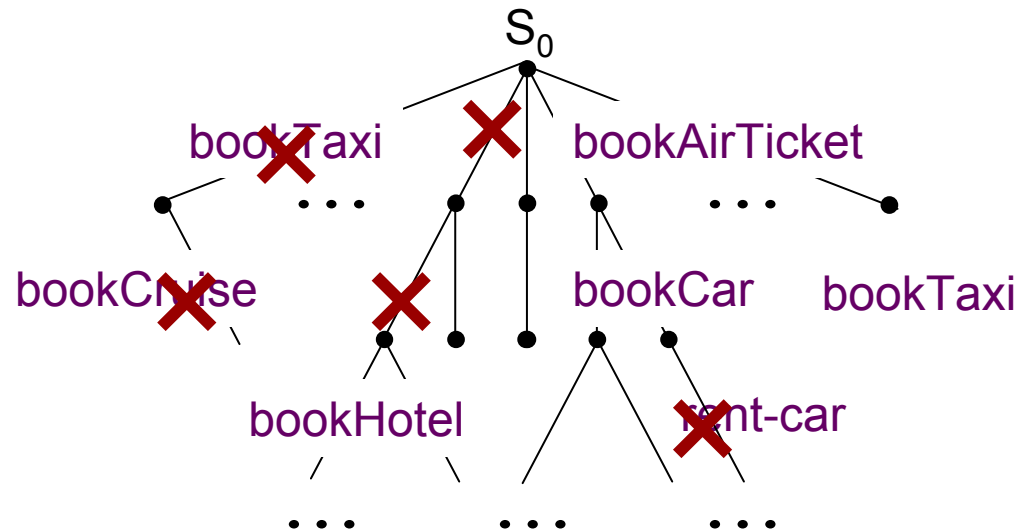
$$D = \Sigma \cup D_{una} \cup D_{S0} \cup D_{ap} \cup D_{SS}$$

- domain independent foundational axioms,  $\Sigma$
- unique names assumptions for actions,  $D_{una}$
- axioms describing the initial situation,  $D_{S0}$
- action precondition axioms,  $D_{ap}$ ,  $Poss(a,s) \equiv \Pi(\mathbf{x},s)$   
e.g.,  $Poss(pickup(x),s) \equiv \neg holding(x,s)$
- successor state axioms,  $D_{SS}$ ,  $F(x,s) \equiv \Phi(\mathbf{x},s)$   
e.g.,  $holding(x,do(a,s)) \equiv a = pickup(x) \vee$   
 $(holding(x,s) \wedge (a \neq putdown(x) \vee a \neq drop(x)))$

# Golog [Levesque et al. 97, De Giacomo et al. 00, etc]

procedural constructs:

- sequence
- if-then-else
- nondeterministic choice
  - actions
  - arguments
- while-do
- ...



E.g., *bookAirTicket(x); if far then bookCar(x) else bookTaxi(y)*

# “Big Do” over Complex Actions

$Do(\delta, s, s')$  is an abbreviation. It holds whenever  $s'$  is a terminating situation following the execution of complex action  $\delta$  in  $s$ .

Each abbreviation is a **formula** in the situation calculus.

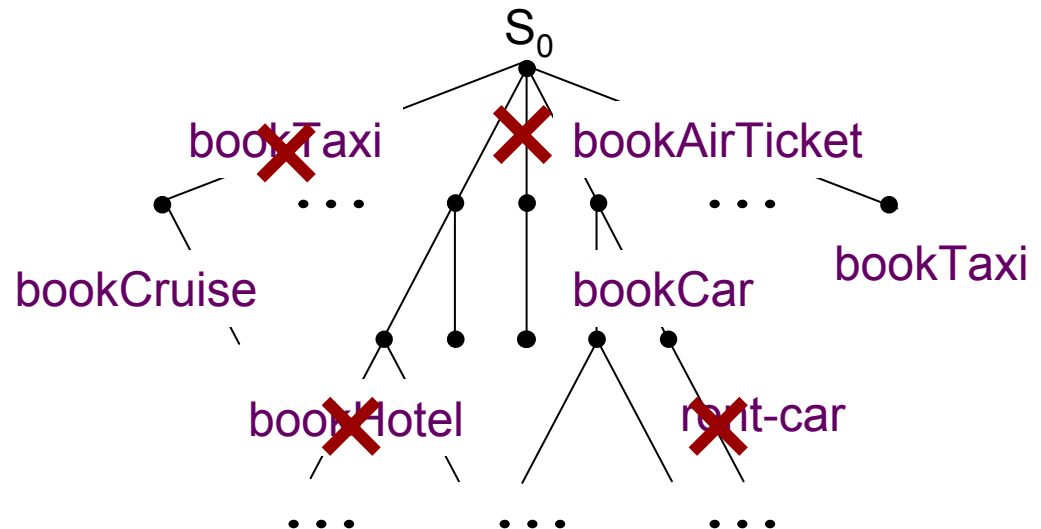
$$Do(a, s, s') \cong Poss(a[s], s) \wedge s' = do(a[s], s)$$

$$Do([a_1; a_2], s, s') \cong (\exists s^*). (Do(a_1, s, s^*) \wedge Do(a_2, s^*, s'))$$

...

E.g., Let  $\delta$  be *bookAirTicket(x); if far then bookCar(x) else bookTaxi(y)*

$$D \models \exists s'. Do(\delta, S_0, s')$$



# Golog Complex Actions, cont.

## 1. Primitive Actions

$$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a[s], s) \wedge s' = do(a[s], s).$$

## 2. Test Actions

$$Do(\phi, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s.$$

## 3. Sequence

$$Do([\delta_1; \delta_2], s, s') \stackrel{\text{def}}{=} (\exists s^*). (Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')).$$

# Complex Actions, cont.

## 4. Nondeterministic choice of two actions

$$Do((\delta_1 \mid \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

## 5. Nondeterministic choice of two arguments

$$Do((\pi x) \delta(x), s, s') \stackrel{def}{=} (\exists x) Do(\delta(x), s, s').$$

## 6. Nondeterministic Iterations

$$Do(\delta^*, s, s') \stackrel{def}{=} (\forall P). \{ (\forall s_1) P(s_1, s_1) \wedge (\forall s_1, s_2, s_3) [P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)] \} \supset P(s, s').$$

# Complex Actions, cont.

Conditional and loops definition in GOLOG

**if**  $\phi$  **then**  $\delta_1$  **else**  $\delta_2$  **endIf**  $\stackrel{def}{=} [\phi? ; \delta_1] \mid [\neg\phi? ; \delta_2],$

**while**  $\phi$  **do**  $\delta$  **endWhile**  $\stackrel{def}{=} [[\phi? ; \delta]^* ; \neg\phi?].$

Procedures difficult to define in GOLOG

- No easy way of macro expansion on recursive procedure calls to itself

# Golog in a Nutshell

- Golog programs are instantiated using a theorem prover
- User supplies, axioms, successor state axioms, initial situation condition of domain, and Golog program describing agent behaviour
- Execution of program gives:

$$Axioms \models (\exists s) Do(program, S_0, s)$$

$$do(a_n, \dots do(a_2, do(a_1, S_0)) \dots)$$

# Golog Example: Elevator Controller

## Primitive Actions

- *Up*( $n$ ): move the elevator to a floor  $n$
- *Down*( $n$ ): move the elevator down to a floor  $n$
- *Turnoff*: turn off call button  $n$
- *Open*: open elevator door
- *Close*: close the elevator door
- Fluents
  - *CurrentFloor*( $s$ ) =  $n$ , in situation  $s$ , the elevator is at floor  $n$
  - *On*( $n, s$ ), in situation  $s$  call button  $n$  is on
  - *NextFloor*( $n, s$ ) = in situation  $s$  the next floor ( $n$ )



# Example, cont.

- Primitive Action Preconditions

$$Poss(up(n), s) \equiv current\_floor(s) < n.$$

$$Poss(down(n), s) \equiv current\_floor(s) > n.$$

$$Poss(open, s) \equiv true.$$

$$Poss(close, s) \equiv true.$$

$$Poss(turnoff(n), s) \equiv on(n, s).$$

- Successor State Axiom

$$Poss(a, s) \supset [on(m, do(a, s)) \equiv on(m, s) \wedge a \neq turnoff(m)].$$

# Example, cont.

- One of the possible fluents

$$\begin{aligned} \text{next\_floor}(n, s) \equiv & \text{on}(n, s) \wedge \\ & (\forall m). \text{on}(m, s) \supset |m - \text{current\_floor}(s)| \geq |n - \text{current\_floor}(s)|. \end{aligned}$$

## Elevator GOLOG Procedures

**proc** *serve*(*n*) *go\_floor*(*n*) ; *turnoff*(*n*) ; *open* ; *close* **endProc**.

**proc** *go\_floor*(*n*) (*current\_floor* = *n*)? | *up*(*n*) | *down*(*n*) **endProc**.

**proc** *serve\_a\_floor* ( $\pi$  *n*)[*next\_floor*(*n*)? ; *serve*(*n*)] **endProc**.

**proc** *control* [**while** ( $\exists n$ )*on*(*n*) **do** *serve\_a\_floor* **endWhile**] ; *park* **endProc**.

**proc** *park* **if** *current\_floor* = 0 **then** *open* **else** *down*(0) ; *open* **endIf** **endProc**.

# Example, cont.

Theorem proving task

$$\text{Axioms} \models (\exists s) \text{Do}(\text{control}, S_0, s)$$

Successful Execution of GOLOG program

$$s = \text{do}(\text{open}, \text{do}(\text{down}(0), \text{do}(\text{close}, \text{do}(\text{open}, \text{do}(\text{turnoff}(5), \\ \text{do}(\text{up}(5), \text{do}(\text{close}, \text{do}(\text{open}, \text{do}(\text{turnoff}(3), \text{do}(\text{down}(3), S_0)))))))))))))$$

Returns the following to elevator hardware control system

$$[\text{down}(3), \text{turnoff}(3), \text{open}, \text{close}, \text{up}(5), \text{turnoff}(5), \text{open}, \text{close}, \text{down}(0), \text{open}]$$

# The Golog Interpreter

Many different Golog interpreters for different versions of Golog, e.g.,

- ConGolog
- IndiGolog
- ccGolog
- DTGolog
- ...

All are available online and easy to use!

The *vanilla* Golog interpreter is 20 lines of Prolog Code....

# The Golog Interpreter

```
/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions. */
```

```
holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- \+ holds(some(V,P),S). /* Negation */
holds(-P,S) :- isAtom(P), \+ holds(P,S). /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
```

# The Golog Interpreter

```
do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(¬P) : E2),S,S1).
do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(¬P),S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).
```

```
/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by
New. */
```

....

# Discussion

Limitations of the Golog interpreter (particularly as a planner):

- The search is “dumb” (i.e., uninformed)
- Attempts to improve search:
  1. use FF planner in the nondeterministic parts [Nebel et al.07]
  2. Desire: Want to use heuristic search  
[Baier et al, ICAPS07][Fritz et al, KR08]: Compile a Congolog program into a PDDL domain
    - Now can exploit any state of the art planner

Other Merits of the Baier/Fritz et al. compilation

- HTN can be described as a ConGolog program.  
→ Compiler can also be used to compile HTN!

Other recent advances

- Incorporating preferences into Golog and HTN [Sohrabi, Baier et al.]