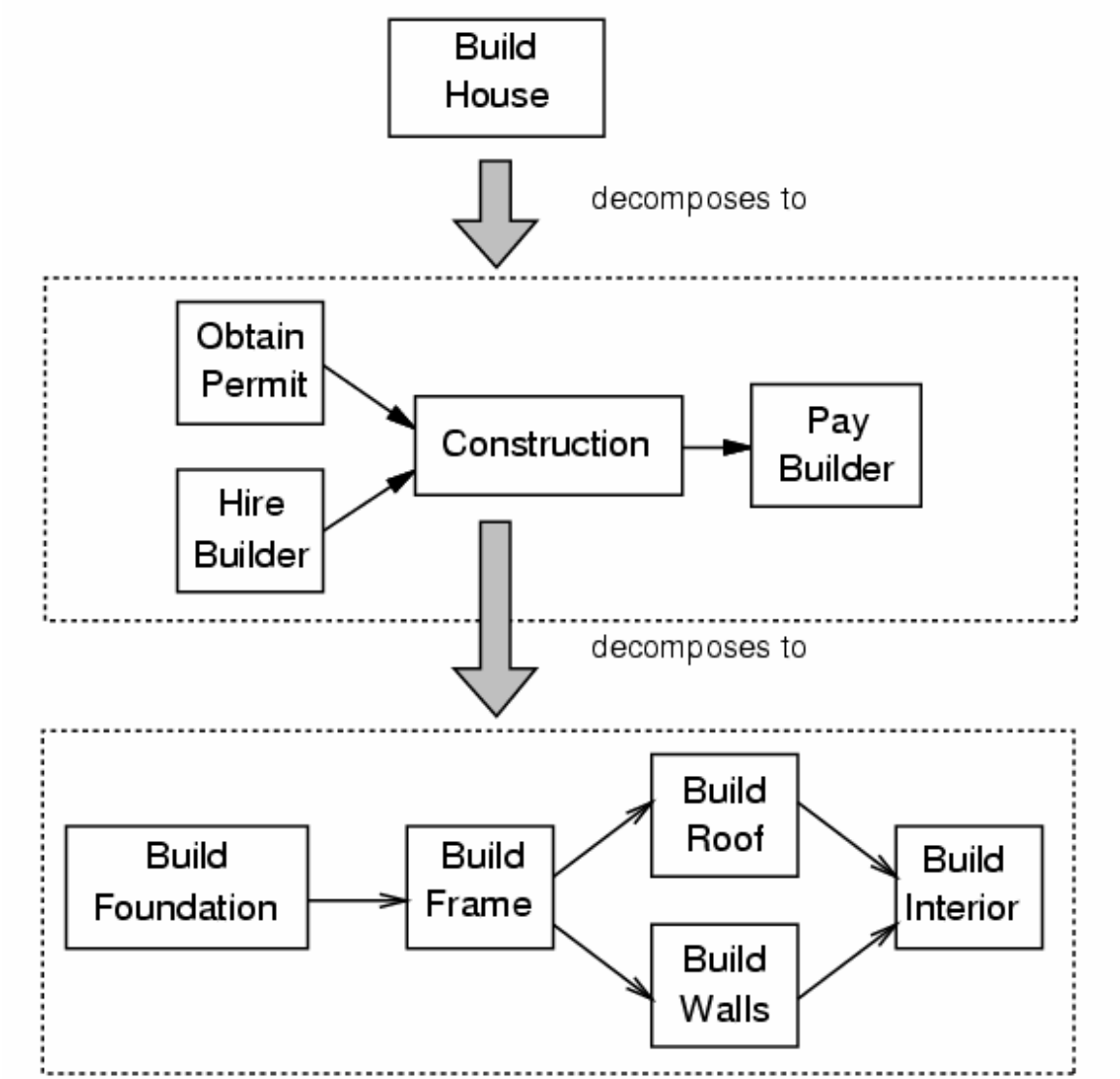
Hierarchical Task Network (HTN) Planning

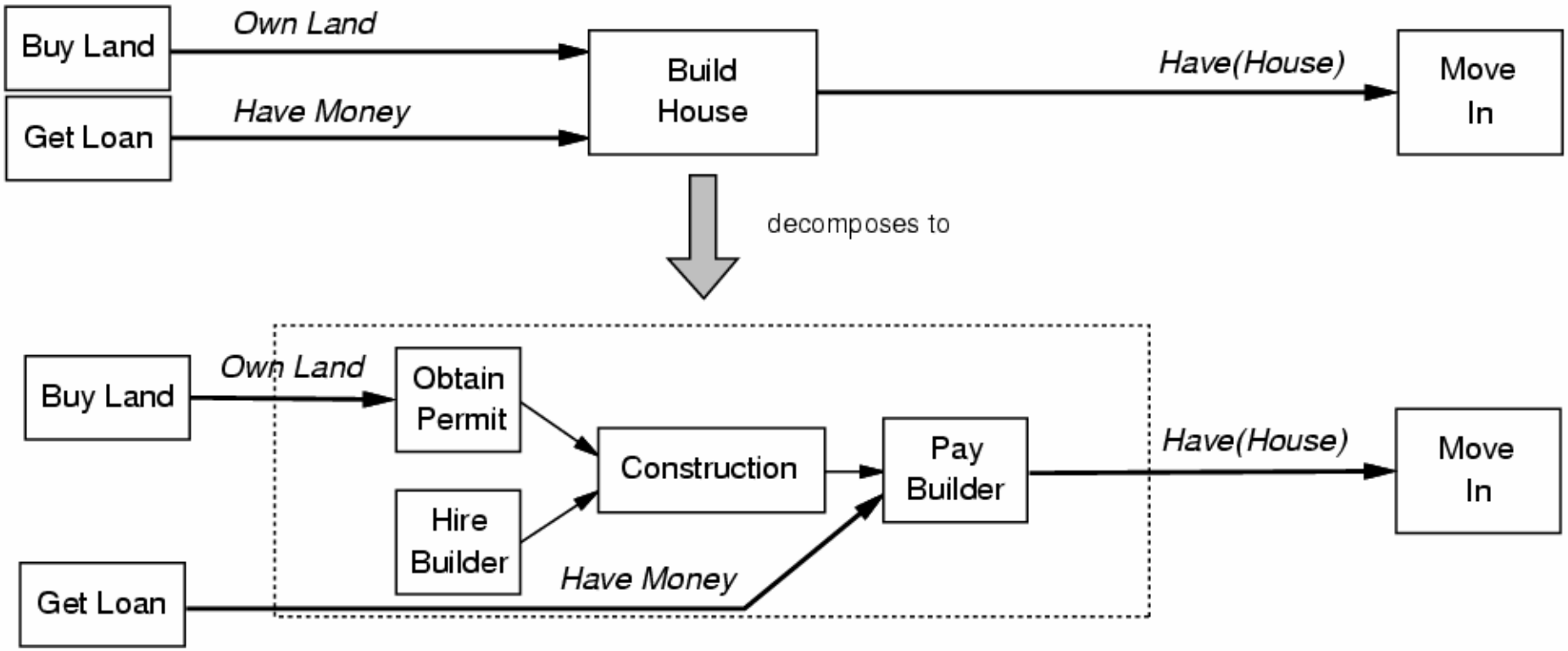
José Luis Ambite*

[* Based in part on presentations by Dana Nau and Rao Kambhampati]

Hierarchical Decomposition



Task Reduction



Hierarchical Planning Brief History

- Originally developed about 25 years ago
  - NOAH [Sacerdoti, IJCAI 1977]
  - NONLIN [Tate, IJCAI 1977]
- Knowledge-based → Scalable
  - Task Hierarchy is a form of domain-specific knowledge
- Practical, applied to real world problems
- Lack of theoretical understanding until early 1990's [Erol et al, 1994] [Yang 1990] [Kambhampati 1992]
  - Formal semantics, sound/complete algorithm, complexity analysis [Erol et al, 1994]

Deployed, Practical Planners

- SIPE, SIPE-2 [Wilkins, 85-]
  - http://www.ai.sri.com/~sipe/
- NONLIN/O-Plan/I-X [Tate et. al., 77-]
  - http://www.aiai.ed.ac.uk/~oplan/
  - http://www.aiai.ed.ac.uk/project/ix/
- Applications:
  - Logistics
    - Military operations planning: Air campaign planning, Non-Combatant Evacuation Operations
    - Crisis Response: Oil Spill Response
  - Production line scheduling
  - Construction planning: Space platform building, house construction
  - Space applications: mission sequencing, satellite control
  - Software Development: Unix administrator's script writing

Deployed, Practical Planners

Many features:

- Hierarchical decomposition
- Resources
- Time
- Complex conditions
- Axioms
- Procedural attachments
- Scheduling
- Planning and Execution
- Knowledge acquisition tools
- Mixed-initiative

Sipe-2 Planner

Drawings

APPLICATION
- PROFILE
- DOMAIN
- PLAN
- DRAWINGS
- NODE

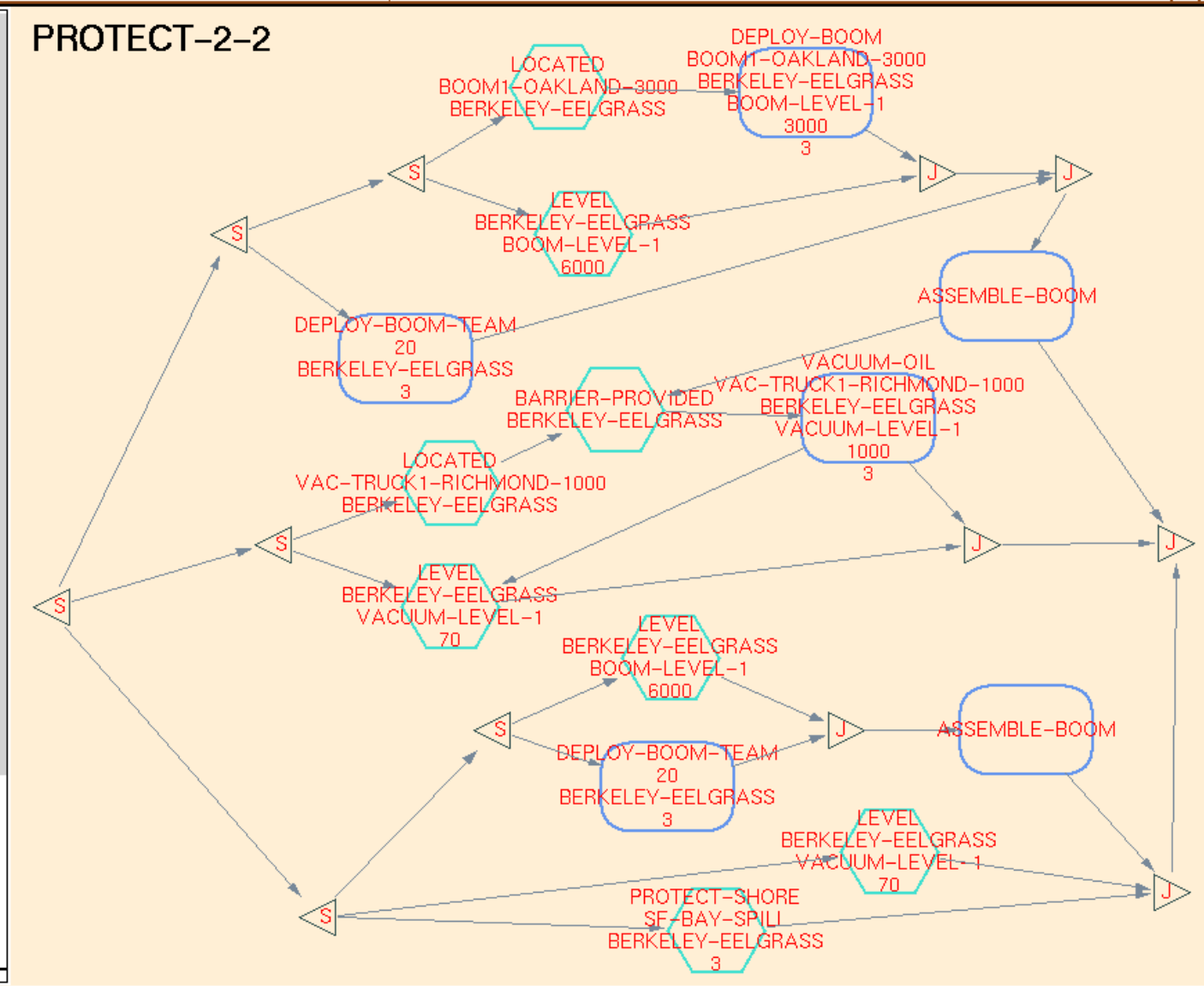
DRAW:
- operator
- problem
- plan
- plan-to-node
- world
- objects

SELECT
DESTROY
DESTROY-multiple

NEW VIEW
RENAME
REDRAW
REPARSE

BACKUP
REVERT
RESCALE
HARDCOPY

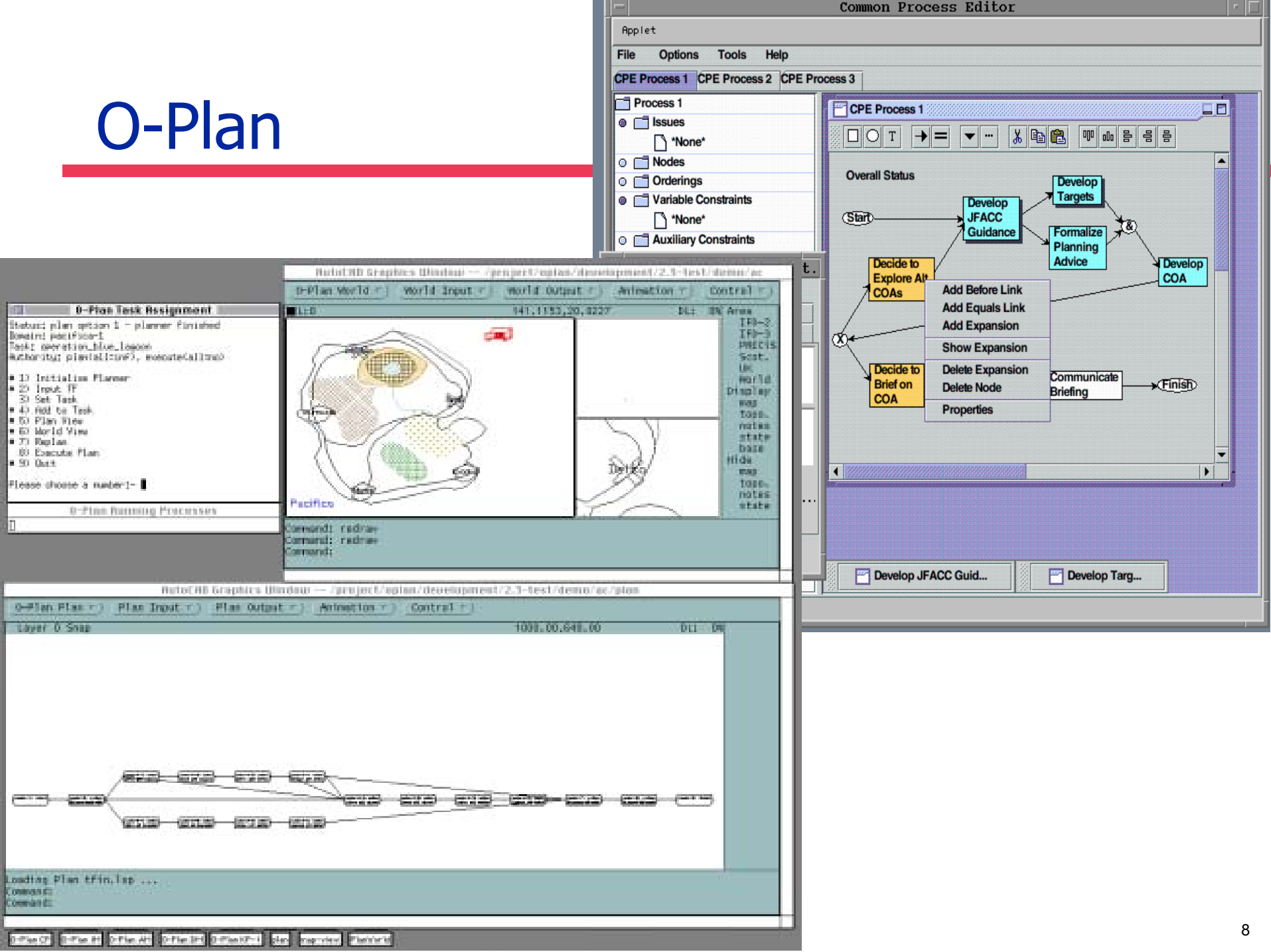
GRAPH:
- select
- create
- destroy
- input
- output




```
Command: :Drawings Mode
Command:
Command:
```

/socap2/project/aic/aic-export/sipe/beta/lisp/oil-spill.graph PROTECT-2-2 Sipe-2 -- Unpublished (C) 1986-1994, S

O-Plan


Common Process Editor
Applet
File
Options
Tools
Help
CPE Process 1
CPE Process 2
CPE Process 3
Process 1
Issues
None
Nodes
Orderings
Variable Constraints
None
Auxiliary Constraints
CPE Process 1
Overall Status
Start
Develop JFACC Guidance
Develop Targets
Formalize Planning Advice
&
Develop COA
Decide to Explore Alt COAs
Add Before Link
Add Equals Link
Add Expansion
Show Expansion
Delete Expansion
Delete Node
Properties
X
Decide to Brief on COA
Communicate Briefing
Finish
Develop JFACC Guid...
Develop Targ...
O-Plan Task Assignment
O-Plan Running Processes
Pacifics

HTN Planning

- Capture hierarchical structure of the planning domain
- Planning domain contains non-primitive actions and schemas for reducing them
- Reduction schemas:
  - given by the designer
  - express preferred ways to accomplish a task

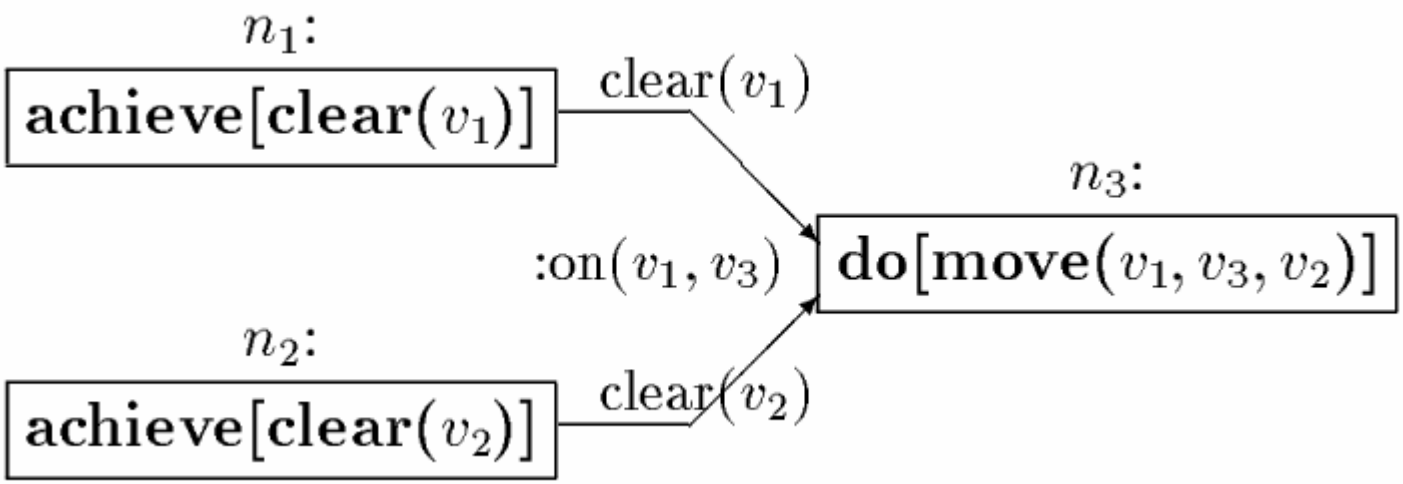
HTN Formalization (1)

- State: list of ground atoms
- Tasks:
  - Primitive tasks: do[f(x_1, …, x_n)]
  - Non-primitive tasks:
    - Goal task: achieve(l) (l is a literal)
    - Compound task: perform[t(x_1, …, x_n)]
- Operator:
  - [operator f(x_1, …, x_n) (pre: l_1, …, l_n) (post: l'_1, …, l'_n)]
- Method: (α, d)
  - α is a non-primitive task and d is a task network
- Plan: sequence of ground primitive tasks (operators)

HTN Formalization (2)

- Task network: $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$
  - n_i = node label
  - α_i = task
  - ϕ = formula that includes
    - Binding constraints: $(v = v')$ or $(v \neq v')$
    - Ordering constraints: $(n < n')$
    - State constraints:
      - (n, l, n'): interval preservation constraint (causal link)
      - (l, n): l must be true in state immediately before n
      - (n, l): l must be true in state immediately after n

Task Network Example



$$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])(n_3 : do[move(v_1, v_3, v_2)])$$
$$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3) \wedge (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3)$$
$$\wedge \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

HTN Planning Algorithm (intuition)

Problem reduction:

- Decompose tasks into subtasks
- Handle constraints
- Resolve interactions
- If necessary, backtrack and try other decompositions

Basic HTN Procedure

1. Input a planning problem P
2. If P contains only primitive tasks, then resolve the conflicts and return the result. If the conflicts cannot be resolved, return failure
3. Choose a non-primitive task t in P
4. Choose an expansion for t
5. Replace t with the expansion
6. Find interactions among tasks in P and suggest ways to handle them. Choose one.
7. Go to 2

procedure UMCP:

1. Input a planning problem $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$
2. if d is primitive, then
   If $comp(d, I, \mathcal{D}) \neq \emptyset$, return a member of it.
   Otherwise return FAILURE.
3. Pick a non-primitive task node $(n : \alpha)$ in d.
4. Nondeterministically choose a method m for α.
5. Set $d := reduce(d, n, m)$.
6. Set $\Gamma := \tau(d, I, \mathcal{D})$.
7. Nondeterministically set $d :=$ some element of Γ.
8. Go to step 2.

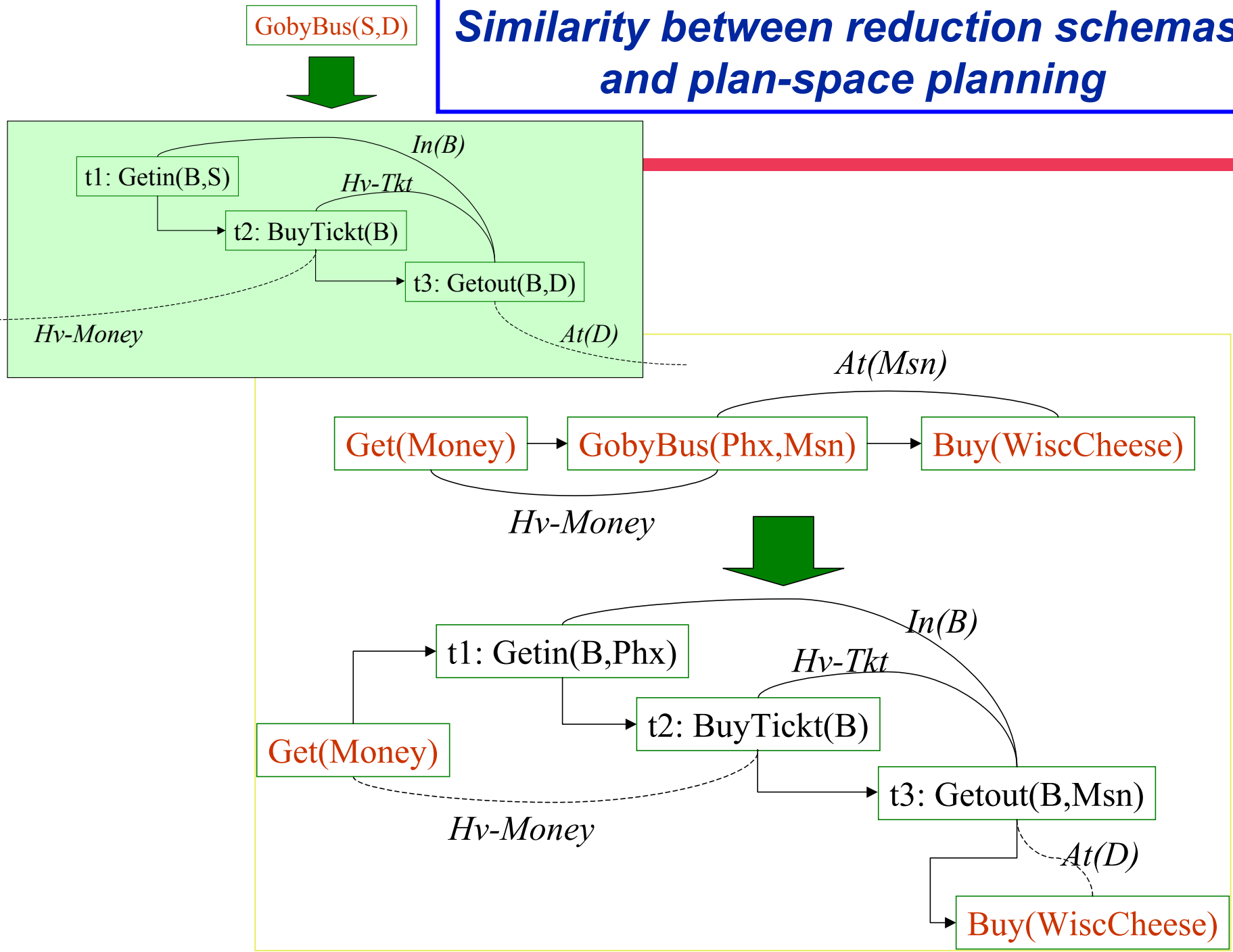
ground total ordering satisfying constraints

m = (α', d') s.t. mgu(α, α') = θ

τ= "critics" to resolve conflicts

Reduce(d, n, m) = task network obtained from d by replacing (n : α)θ with d'θ (and modifying the constraint formula)

Similarity between reduction schemas and plan-space planning



Algorithm Refine-Plan-PO($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */

Parameters: `sol`: the routine for picking solution candidates from the candidate set of the partial plan **`pick-open`**: the routine for picking open conditions. **`pre-order`**: the routine which adds orderings to the plan to make conflict resolution tractable. **`conflict-resolve`**:the routine which resolves conflicts with auxiliary constraints.

0. Termination Check: If `sol`($\mathcal{P}$) returns a ground operator sequence that **`solves`** the problem, return it and terminate.

1.1 Goal Selection: Using the **`pick-open`** function, pick an open prerequisite $\langle C, t \rangle$ (where C is a precondition of step t) from $\mathcal{P}$ to work on. *Not a backtrack point.*

1.2. Goal Establishment: Non-deterministically select a new or existing establisher step t' for $\langle C, t \rangle$. Introduce enough constraints into the plan such that (i) t' will have an effect C, and (ii) C will persist until t. *Backtrack point; all establishers need to be considered.*

1.3. Bookkeeping: (Optional) Add auxiliary constraints noting the establishment decisions, to ensure that these decisions are not violated by latter refinements. This in turn reduces the redundancy in the search space. Bookkeeping strategies used by most existing planners can be modeled in terms of addition of interval preservation constraints.

2. Tractability Refinements: (Optional) These refinements help in making the plan handling and consistency check tractable. Use either one or both:

2.a. Pre-Ordering: Use some given static ordering mechanism, `pre-order`, to impose additional orderings between steps of the partial plans generated by the establishment refinement. *Backtrack point; all interaction orderings need to be considered.*

2.b. Conflict Resolution: Add orderings and bindings to resolve conflicts between the steps of the plan, and the plan's auxiliary constraints. *Backtrack point; all possible conflict resolution constraints need to considered.*

3. Consistency Check: (Optional) If the partial plan is inconsistent (i.e., has no safe ground linearizations), prune it.

4. Recursive Invocation: Call `Refine-Plan-PO` on the the refined partial plan (if it is not pruned).

Algorithm Refine-Plan-Htn($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */
Parameters: `pick-task`: the routine for picking non-primitive tasks for reduction plan to make conflict resolution tractable. auxiliary constraints.

0′. Termination Check: If all tasks of $\mathcal{P}$ are primitive, and if there is a safe ground linearization of $\mathcal{P}$ that `solves` the problem, return it and terminate.

1.1′. Task Selection: Using the `pick-task` function, pick an unreduced task $t \in T$ from $\mathcal{P}$ to work on. *Not a backtrack point.*

1.2′. Task Reduction: Non-deterministically select a reduction schema $S : \mathcal{P}'$ for reducing t. Replace t in $\mathcal{P}$ with $\mathcal{P}'$ (This involves removing t from $\mathcal{P}$, merging the step, binding, ordering, symbol table and auxiliary constraints fields of $\mathcal{P}'$ with those of $\mathcal{P}$, and modifying the ordering and auxiliary constraints in $\mathcal{P}$ which refer to t so that they refer to elements of $\mathcal{P}'$.

Backtrack point; all reduction possibilities need to be considered

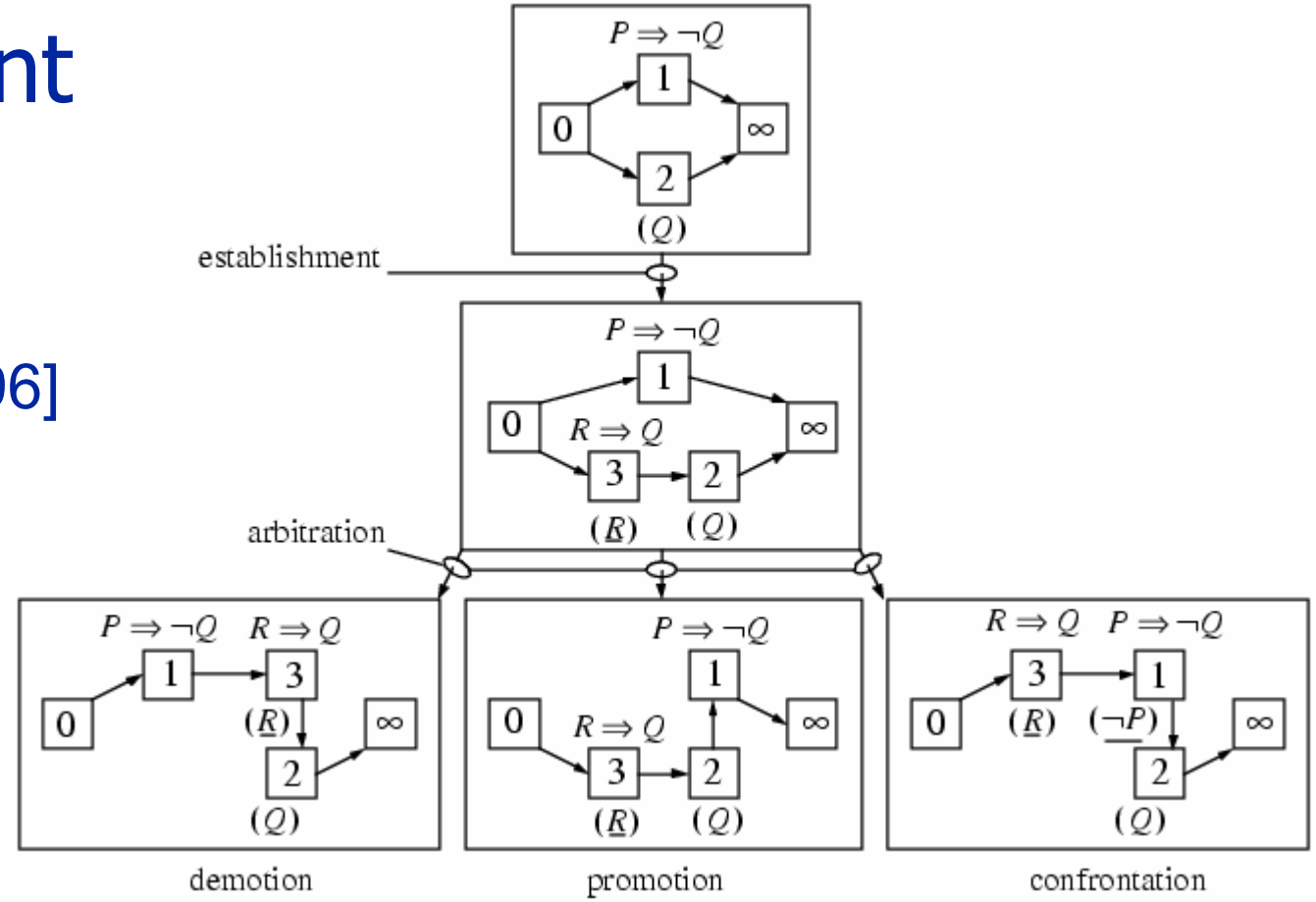
1.3′. Bookkeeping: Same as in `Refine-Plan-PO`

2′. Tractability Refinements: Same as in `Refine-Plan-PO`

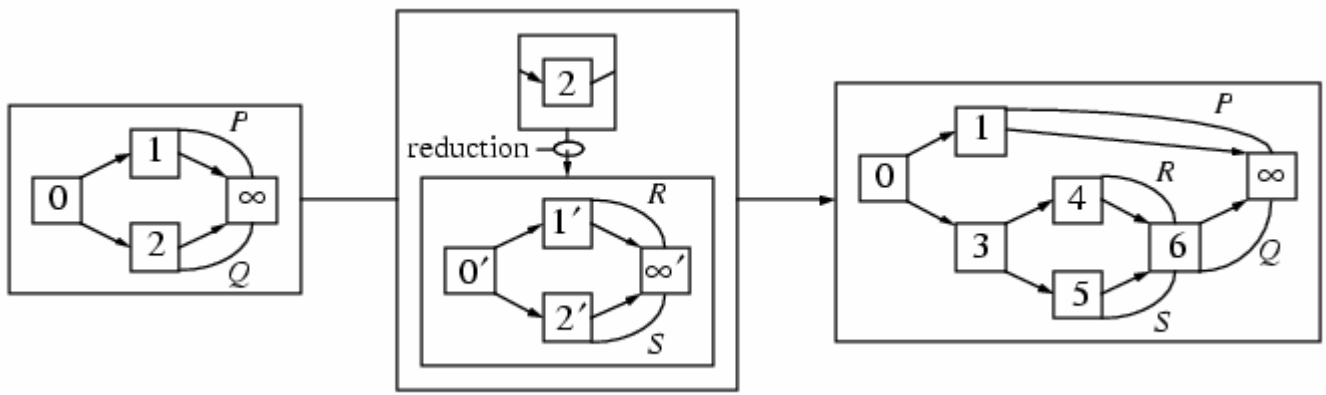
4′. Recursive Invocation: Call `Refine-Plan-HTN` on the the refined partial plan (if it is not pruned).

Refinement Planning

[Kambhampati 96]

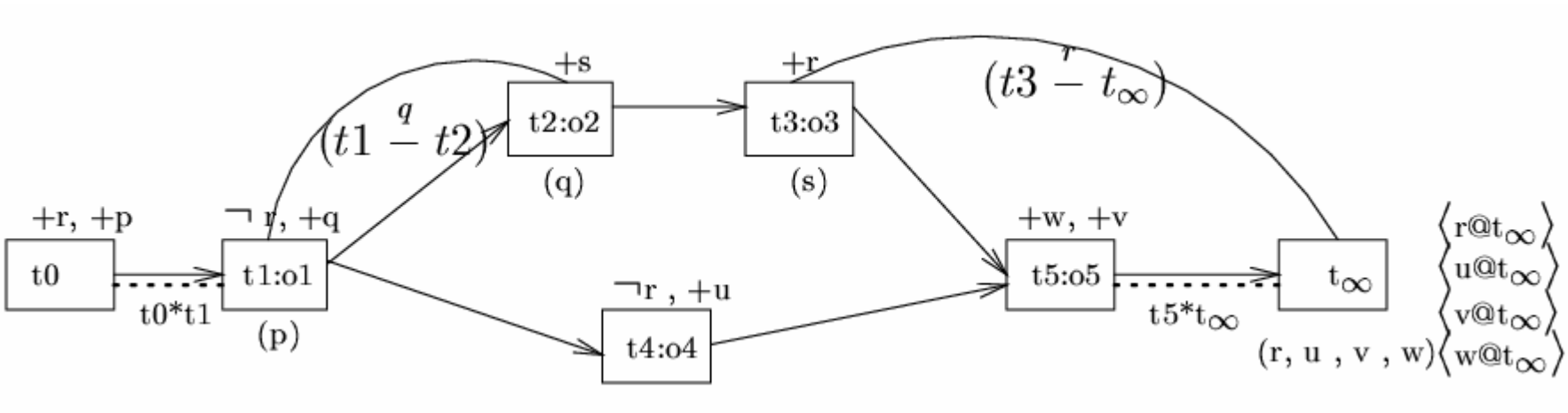


Task reduction



Refinement Planning

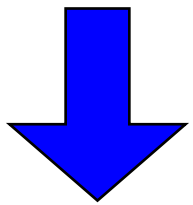
- Unified framework for state-space, plan-space, and HTN planning



[Kambhampati et al, 96]

Expressiveness of STRIPS vs HTN planning

- Solutions to STRIPS problems are regular sets: $(a_1 | a_2 | \ldots a_n)^*$
- Solutions to HTN problems can be arbitrary context-free sets: $a_1^n a_2^n \ldots a_n^n$

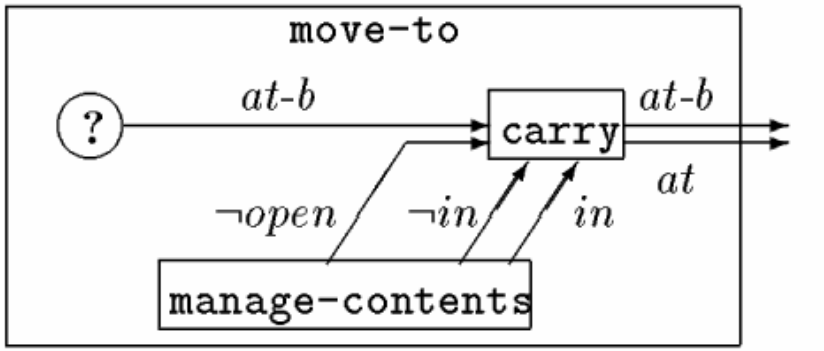


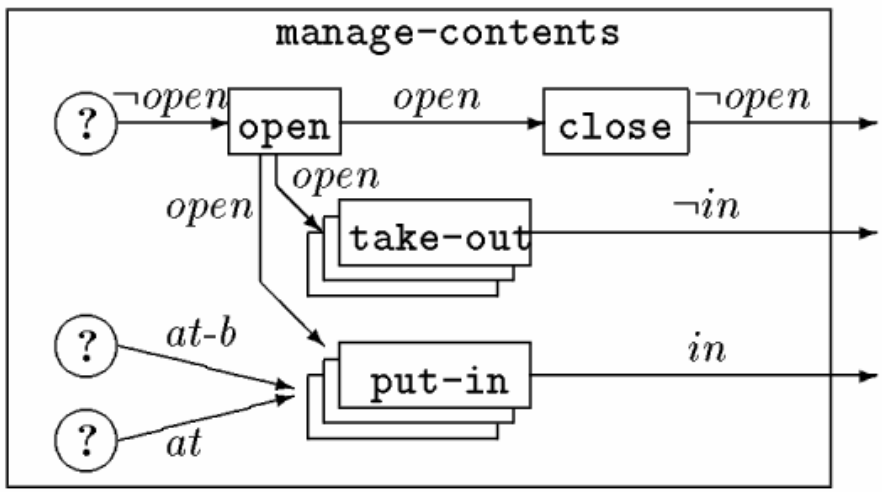
HTN's are more expressive than STRIPS

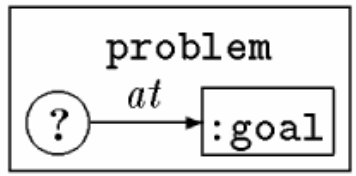
Task Decomposition via Plan Parsing

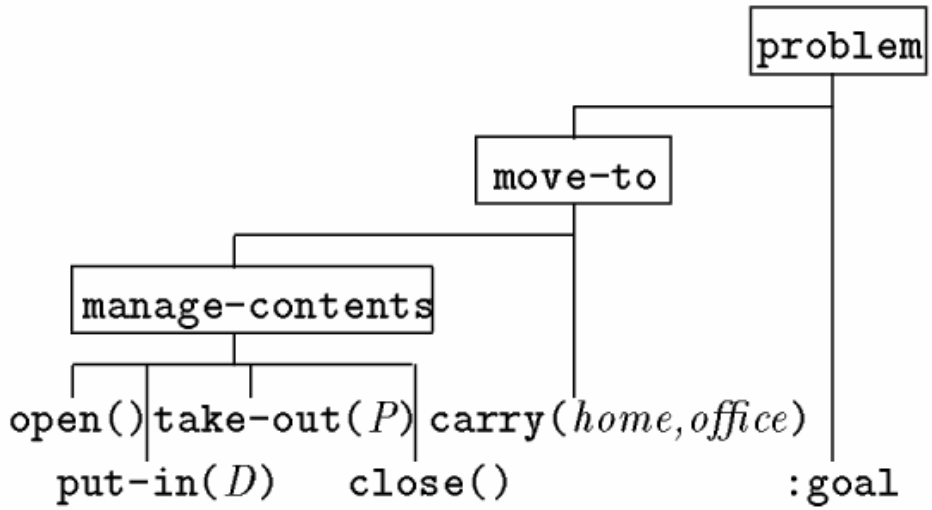
- Task decomposition hierarchy can be seen as a context-free grammar
- Prune plans that do not conform to the grammar in a Partial-Order planner [Barret & Weld, AAAI94]

Task Decomposition via Plan Parsing



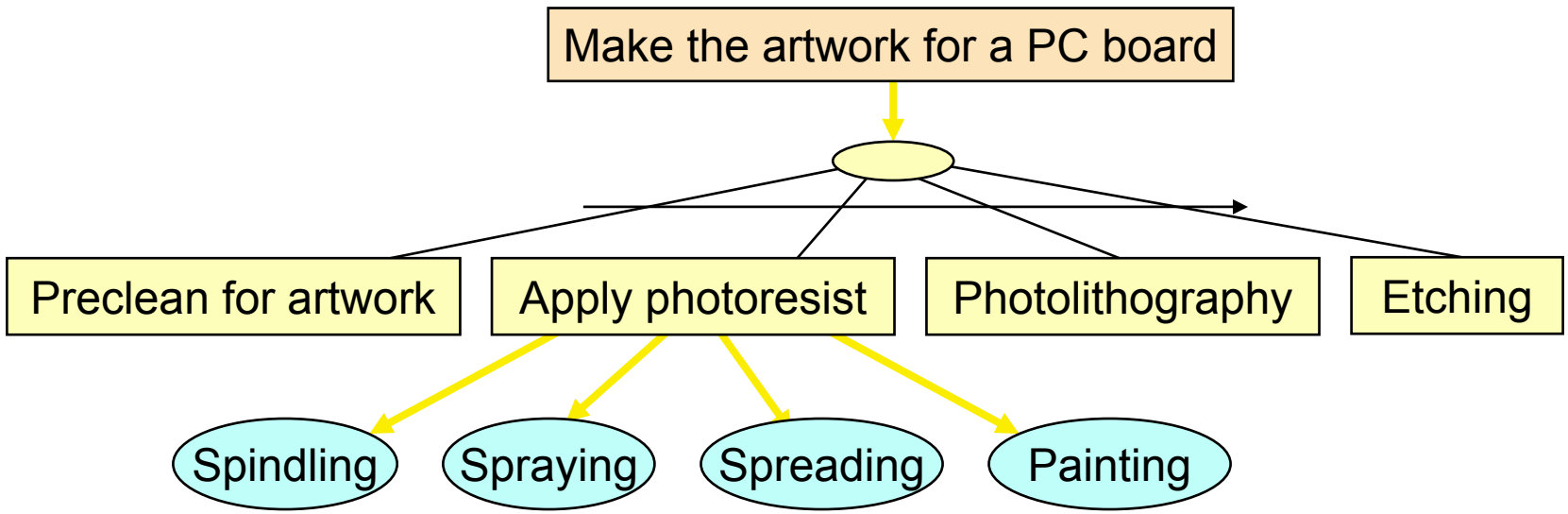






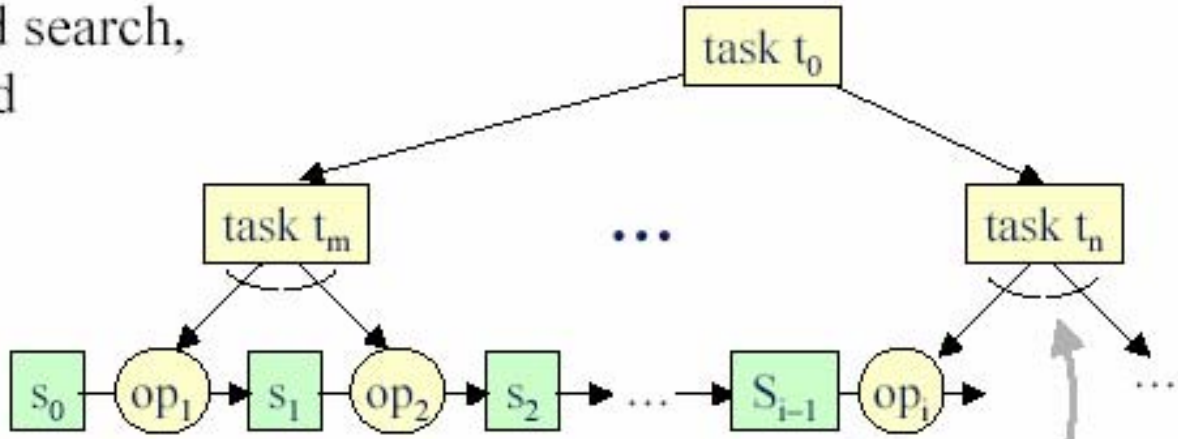
Ordered Task Decomposition

- Adaptation of HTN planning
- Subtasks of each method to be totally ordered
- Decompose these tasks left-to-right
  - The same order that they'll later be executed



Combines Advantages of Both Forward and Backward Search

- Like a backward search, it's goal-directed
  - Goals correspond to tasks

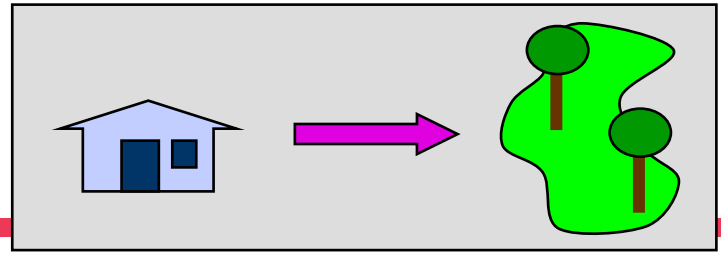


- Like a forward search, it generates actions in the same order in which they'll be executed
- Whenever we want to plan the next task
  - we've already planned everything that comes before it
  - Thus, we know the current state of the world

SHOP (Simple Hierarchical Ordered Planner)

- Domain-independent algorithm for Ordered Task Decomposition
  - Sound/complete
- Input:
  - State: a set of ground atoms
  - Task List: a linear list of tasks
  - Domain: methods, operators, axioms
- Output: one or more plans, it can return:
  - the first plan it finds
  - all possible plans
  - a least-cost plan
  - all least-cost plans

Simple Example

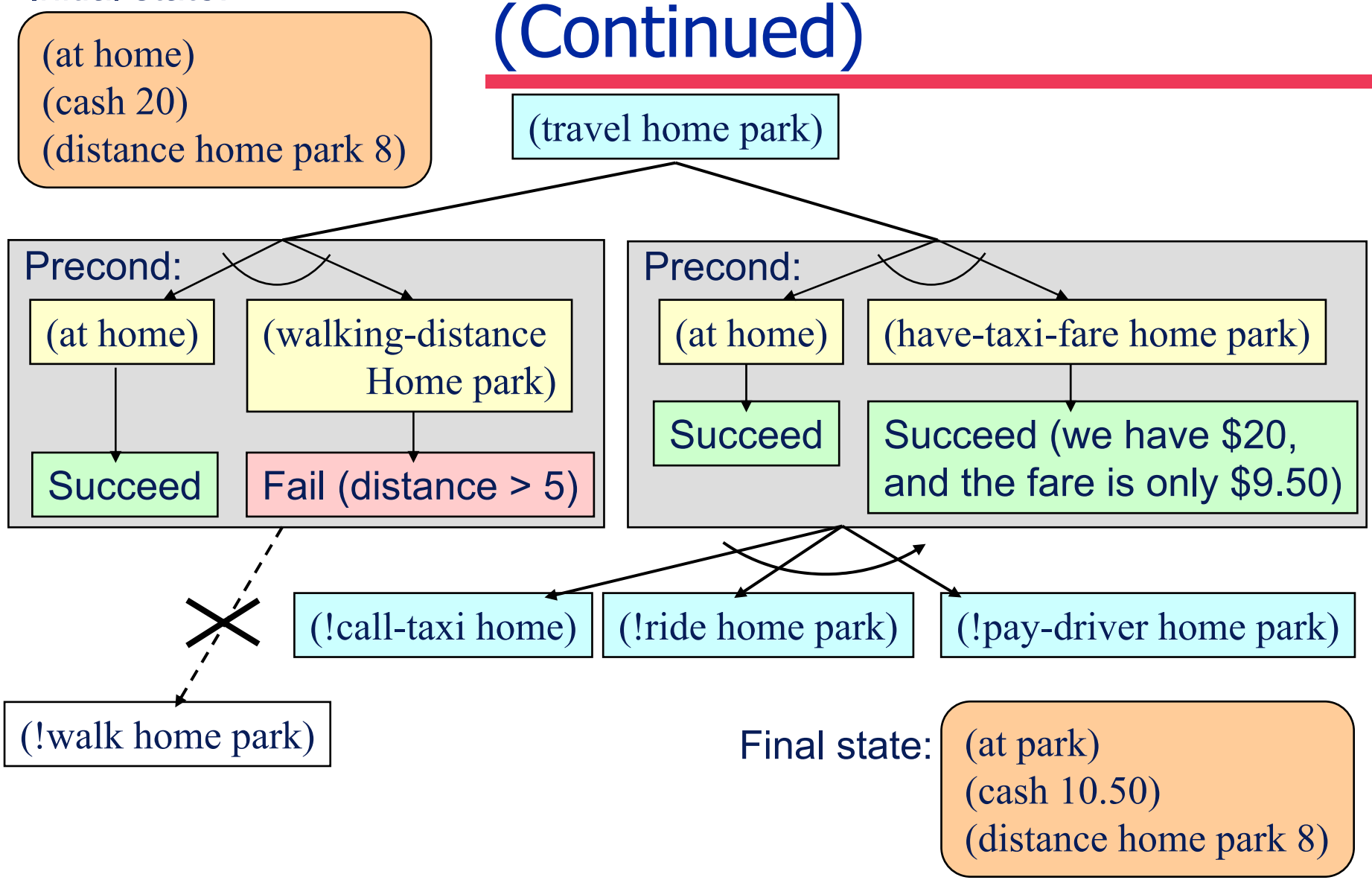


- **Initial task list:** ((travel home park))
- **Initial state:** ((at home) (cash 20) (distance home park 8))
- **Methods** (task, preconditions, subtasks):
  - (:method (travel ?x ?y)
    ((at x) (walking-distance ?x ?y)) ' ((!walk ?x ?y)) 1)
  - (:method (travel ?x ?y)
    ((at ?x) (have-taxi-fare ?x ?y))
    ' ((!call-taxi ?x) (!ride ?x ?y) (!pay-driver ?x ?y)) 1)

Optional cost; default is 1

- **Axioms:**
  - (:- (walking-dist ?x ?y) ((distance ?x ?y ?d) (eval (<= ?d 5))))
  - (:- (have-taxi-fare ?x ?y)
    ((have-cash ?c) (distance ?x ?y ?d) (eval (>= ?c (+ 1.50 ?d)))))
- **Primitive operators** (task, delete list, add list)
  - (:operator (!walk ?x ?y) ((at ?x)) ((at ?y)))
  - …

Simple Example (Continued)



The SHOP Algorithm

procedure SHOP (state *S*, task-list *T*, domain *D*)
1. **if** *T* = nil **then return** nil
2. t_1 = the first task in *T*
3. *U* = the remaining tasks in *T*
4. **if** *t* is primitive & an operator instance *o* matches t_1 **then**
5. P = SHOP ($o(S)$, *U*, *D*)
6. **if** *P* = FAIL **then return** FAIL
7. **return** cons(*o*,*P*)
8. **else if** *t* is non-primitive
   & a method instance *m* matches t_1 in *S*
   & *m*'s preconditions can be inferred from *S* **then**
9. **return** SHOP (*S*, append ($m(t_1)$, *U*), *D*)
10. **else**
11. **return** FAIL
12. **end if**

end SHOP

state S; task list T=(**t_1** ,t_2,…)
operator instance **o**

state o(S); task list T=(t_2, …)

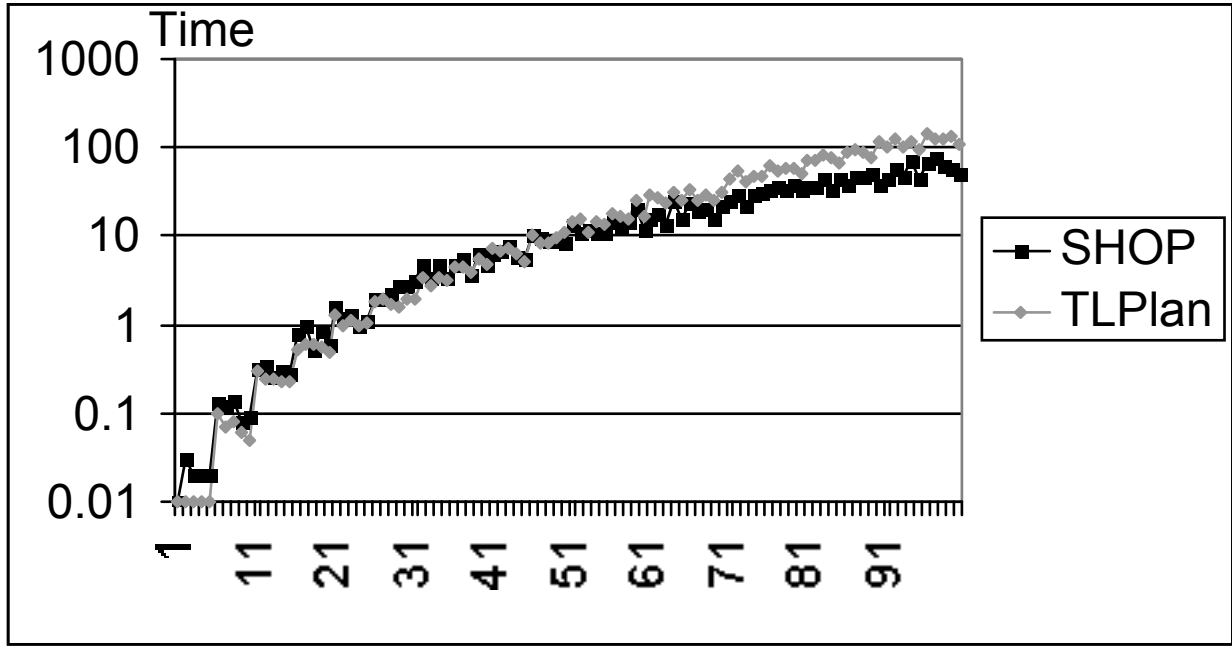
nondeterministic choice among all methods *m* whose preconditions can be inferred from *S*

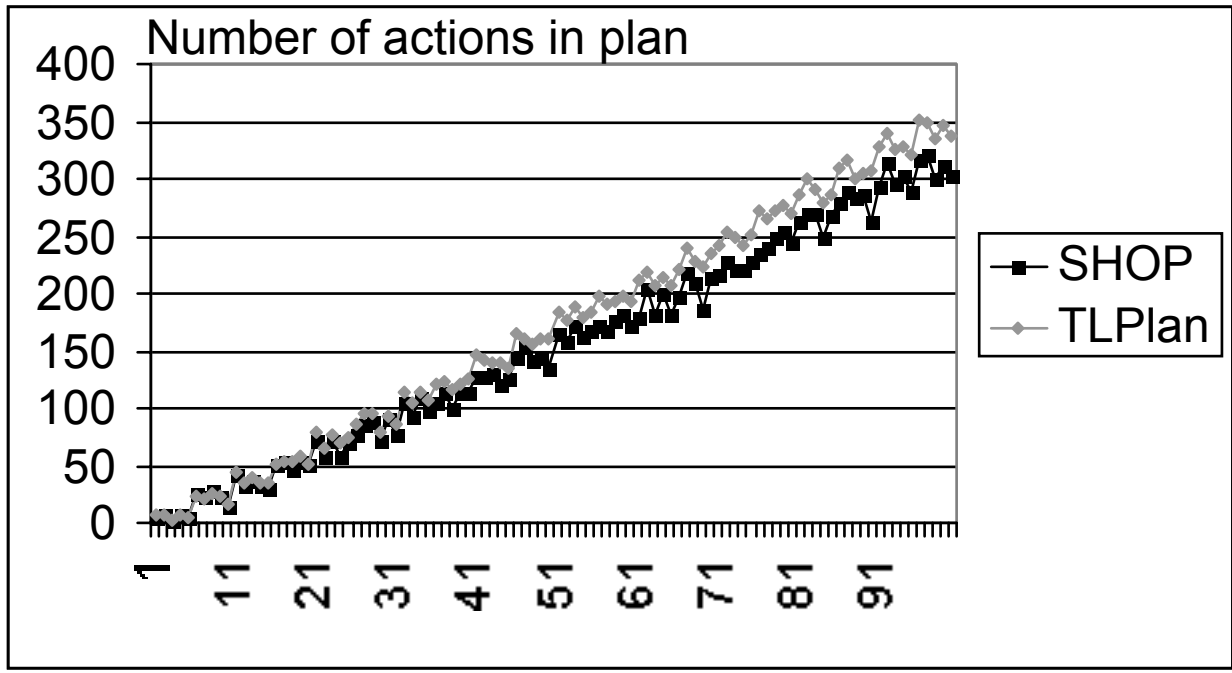
task list T=(**t_1** ,t_2,…)
method instance **m**

task list T=(**u_1,…,u_k** ,t_2,…)

Blocks World

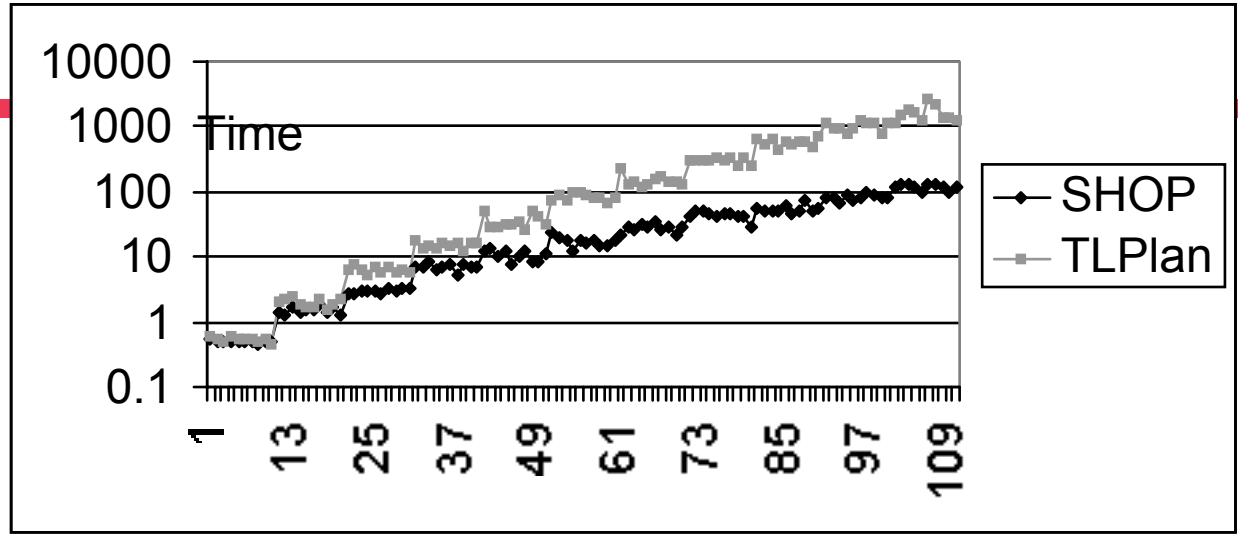
- 100 randomly generated problems
- 167-MHz Sun Ultra with 64 MB of RAM
- Blackbox and IPP could not solve any of these problems
- TLplan's running time was only slightly worse than SHOP's
  - TLplan's pruning rules [Bacchus *et al.*, 2000] have expressive power similar to SHOP's
  - Using its pruning rules, they encoded a block-stacking algorithm similar to ours

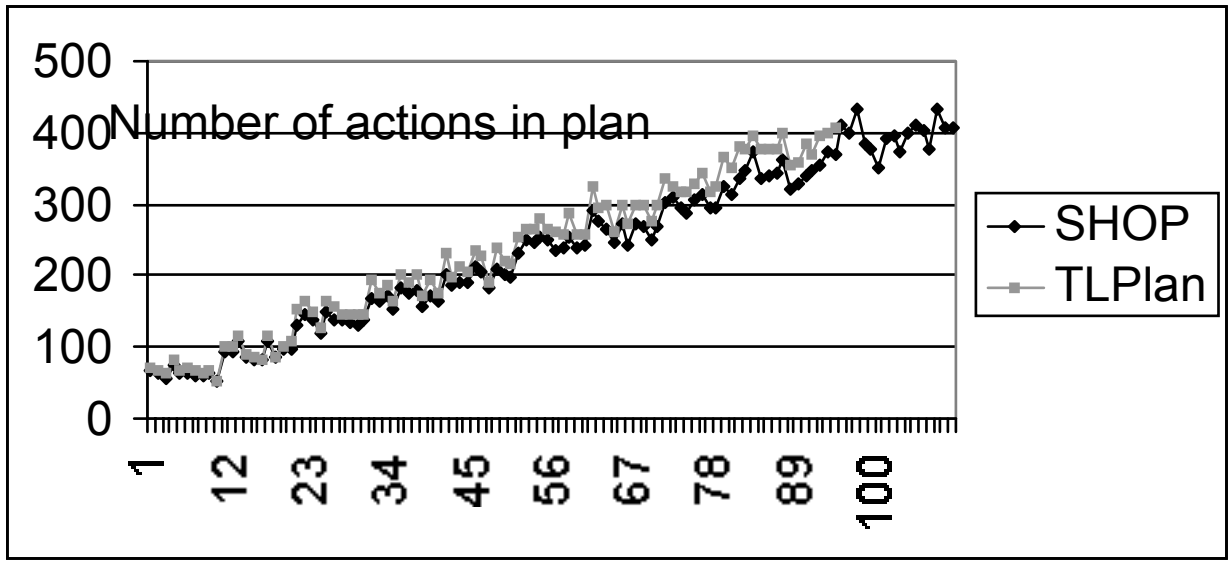




Logistics

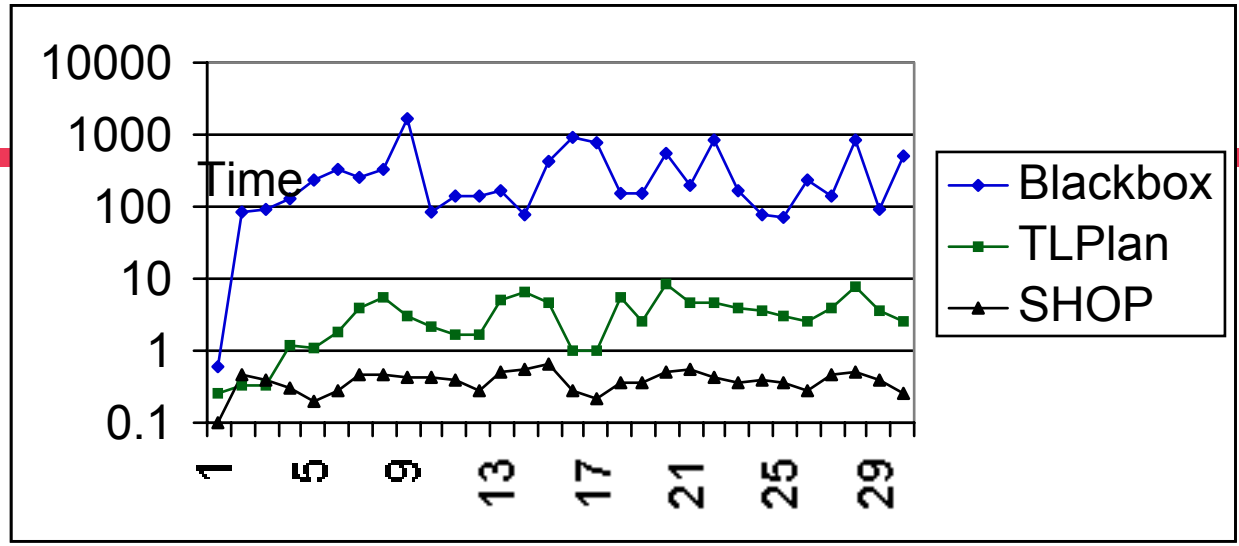
- 110 randomly generated problems
- Same machine as before
- As before, Blackbox and IPP could not solve any of these problems
- TLplan ran somewhat slower than SHOP (about an order of magnitude on large problems)

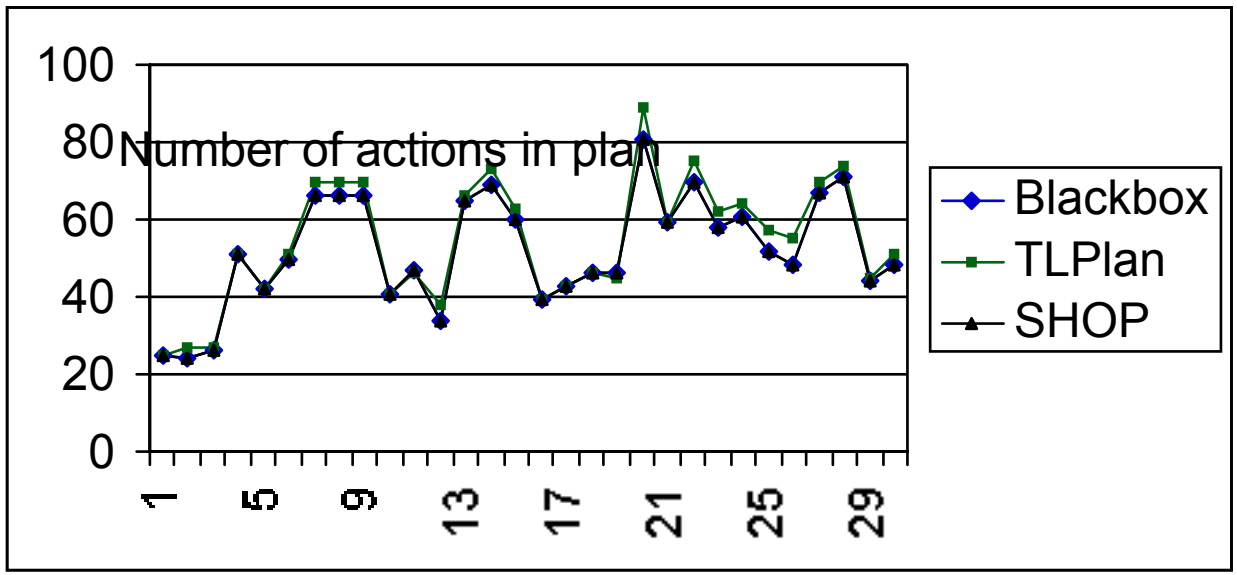




Logistics

- 30 problems from the Blackbox distribution
- SHOP and TLplan on the same machine as before
- Blackbox on a faster machine, with 8GB of RAM
- SHOP was about an order of magnitude faster than TLplan
- TLplan was about two orders of magnitude faster than Blackbox





SHOP demo

[Nau et. aL, 99]

Full procedural control: The SHOP way

Shop provides a "high-level" programming language in which the user can code his/her domain specific planner

-- Similarities to HTN planning
-- Not declarative (?)

The SHOP engine can be seen as an interpreter for this language

```
(:method (travel-to ?y)
    (:first (at ?x)
     (at-taxi-stand ?t ?x)
     (distance ?x ?y ?d)
     (have-taxi-fare ?d))
   `((!hail ?t ?x) (!ride ?t ?x ?y)
     (pay-driver ,(+ 1.50 ?d)))
    ((at ?x) (bus-route ?bus ?x ?y))
   `((!wait-for ?bus ?x)
     (pay-driver 1.00)
     (!ride ?bus ?x ?y)))
```

Travel by bus only if going by taxi doesn't work out

Blurs the domain-specific/domain-independent divide

How often does one have this level of knowledge about a domain?