Planning with Complex Actions

Sheila McIlraith and Ronald Fadel

Knowledge Systems Laboratory
Department of Computer Science
Stanford University

Primitive vs. Complex Actions

Primitive Actions

- Non-decomposable actions
- Characterized in terms of preconditions and (conditional) effects

Complex Actions

- Compositions of primitive (or complex) actions
- Compositions generally in terms of procedural programming language constructs (e.g., if-then-else, sequence, etc.)

E.g.,

move(obj,orig,dest) ≅ pickup(obj,orig)**;** putdown(obj,dest)

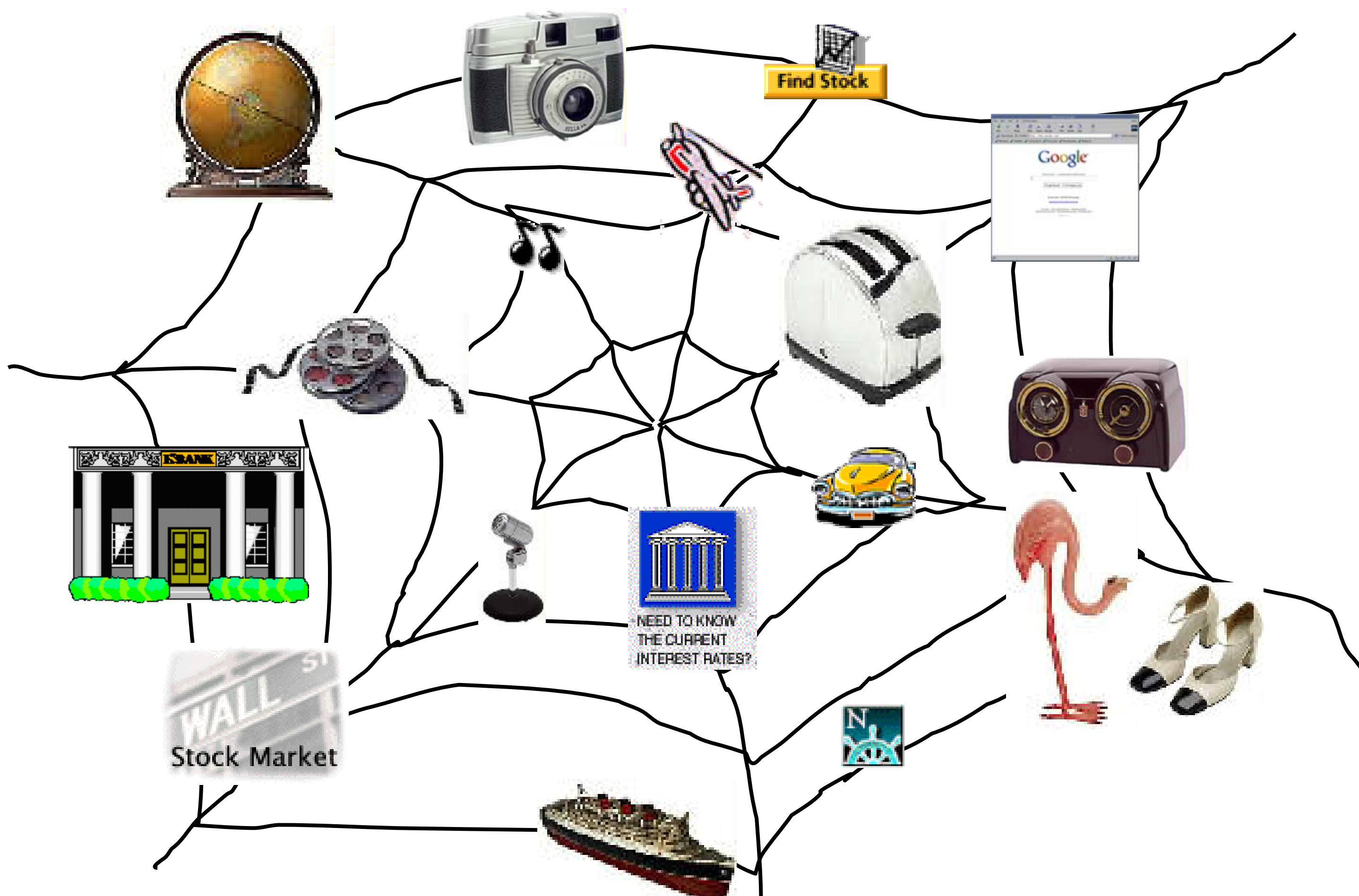
gotoAirpt(loc) ≅ **if** loc=Univ **then**
shuttle(Univ,PA)**;** train(PA,MB)**;**shuttle(MB,SFO)
else taxi(loc,SFO) **endif**

Motivation

1. Automated Composition of Web Services.

Automated Composition of Web Services

Web Services are Web-accessible programs and devices.



Automated Composition of Web Services

✓ Web Services are Web-accessible programs and devices.

➔ Web services can be conceived as primitive and complex actions.

E.g.,

Primitive Action/Service:

getWeather

preconditions: know(cityname(y))

effects: know(weather(y))

Complex Action/Service:

buyAirTicket(**x**) $\cong$ locateFlight(**x**);**if** available(**x**) $\wedge$ okPrice(**x**) **then** provideCustInfo(**x**); payTicket(**x**) **endif**

Automated Composition of Web Services

- ✓ Web Services are Web-accessible programs and devices.
- ✓ Web services can be conceived as primitive and complex actions.
- ➔ Web service composition can be conceived as program composition, or, in a restricted form, as *planning with complex actions.*

Motivation

1. Automated Composition of Web Services.

2. Improving the Efficiency of Planning

   (by representing useful plan segments as complex actions
   and planning with these segments as primitive building blocks).

Planning with Complex Actions

Objective:

Use operator-based planning formalisms (e.g., STRIPS, SitCalc w/ deductive plan synthesis, TALPlan, FF, SModels, etc.) to plan using complex actions as the *primitive building blocks* of a plan.

Challenge:

Complex actions are not represented as operators.
How do we represent them and how do we plan with them?

Approach:

- compile complex actions in T into new primitive actions in new theory, T'
- plan in new theory, T'
- extract primitive action plan in T from "complex action plan" in T'

Planning with Complex Actions

Related to:

- ABStrips [Sacerdoti, 74]
- macro-operators [Fikes,Hart,Nilsson, 72], [Korf, 87]
- HTN planning [Erol, Hendler, Nau, 94]
- Encoding planning constraints in p.o. planning [Baioletti et al., 98]

Contrasted to:

- traditional use of Golog [de Giacomo, et al, 00] [Levesque et al, 97]
- TALPlan [Doherty, Kvarnstrom, 00]
- other planning with domain constraints (see AIPS competition)

All of which use complex actions as advice to constrain search. They do not use complex actions as the *building blocks* of a plan.

Contributions

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation technique (sit calc & PDDL)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Evaluation of Efficiency of Planning

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

We appeal to the situation calculus to formalize our ideas.

Sorts:

Actions

e.g., *a*, *locateFlight(**x**)*

Situations

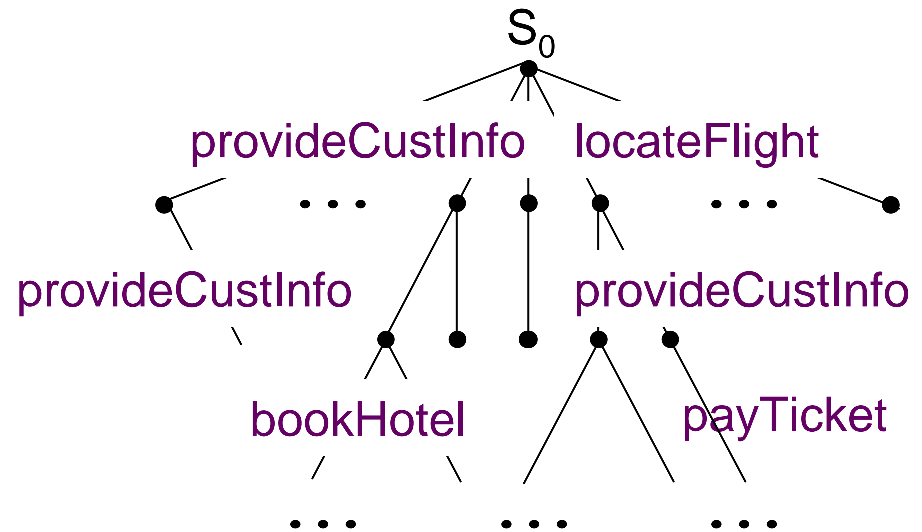
e.g., s, S_0,

*do(locateFlight(**x**),s)*

Fluents

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

Other



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

A situation calculus theory D comprises the following axioms:

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

- domain independent foundational axioms, Σ
- 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]

• programming language instructions set = situation calculus actions

• constructs for assembling actions

sequence $(\delta_1 ; \delta_2)$
test of truth $?\phi$
nondeterministic choice between actions $(\delta_1 \mid \delta_2)$
nondeterministic choice of arguments $\pi\, x.\delta$
nondeterministic iteration δ^*
conditional **if** ϕ **then** δ_1 **else** δ_2
loop **while** ϕ **do** δ

• complex actions originally characterized are macros that reduce to formulae in situation calculus language, in some later work, complex actions formulated as first-class objects.

"Big Do"

E.g., Let δ be a complex action such as

locateFlight(x);if available(**x**) $\wedge$ okPrice(**x**)
then provideCustInfo(**x**); payTicket(**x**)
endif

$Do(\delta, s, s')$ is an abbreviation

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

Each abbreviation is a formula in the situation calculus.

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

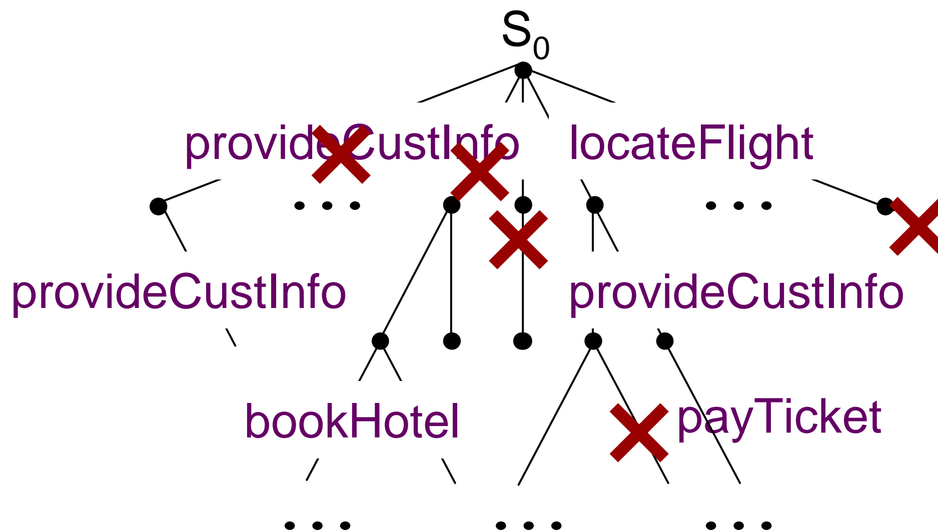
$Do([\delta_1 ; \delta_2], s, s') \cong (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')$

$Do(\pi x.\delta(x), s, s') \cong (\exists x).Do(\delta(x), s, s')$

…

Golog

E.g., locateFlight(**x**);**if** available(**x**) $\wedge$ okPrice(**x**) **then**
provideCustInfo(**x**); payTicket(**x**) **endif**



From Formulae to Action Sequences

Instantiate δ (e.g.,)using theorem proving (deductive synthesis).

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

"*When the complex action δ is executed starting in S_0 it legally terminates in situation s.*"

The action theory *D* imposes constraints on the evaluation of the program.

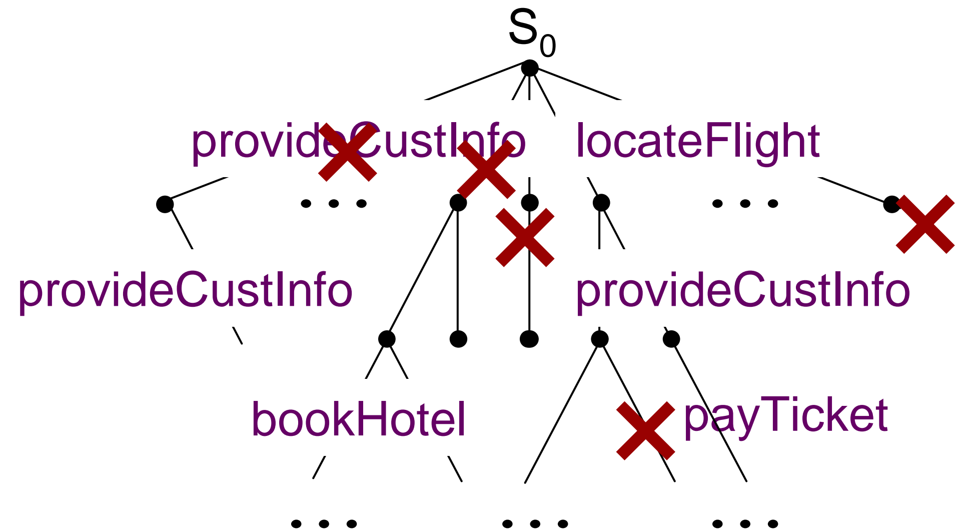
Theorem proving returns a binding for the situation variable s.
E.g., $s = do(a_3, do(a_2, do(a_1, S_0)))$

The binding is a sequence of services that realizes the complex action.

Golog

E.g., locateFlight(**x**);**if** available(**x**) $\wedge$ okPrice(**x**) **then**
provideCustInfo(**x**); payTicket(**x**) **endif**

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



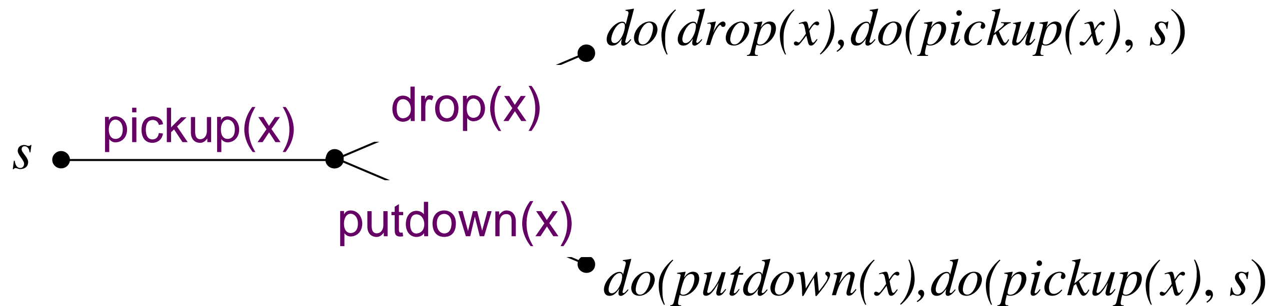
Contributions

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation technique (sit calc & PDDL)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Evaluation of Efficiency of Planning

Simple Running Example

Complex action δ is

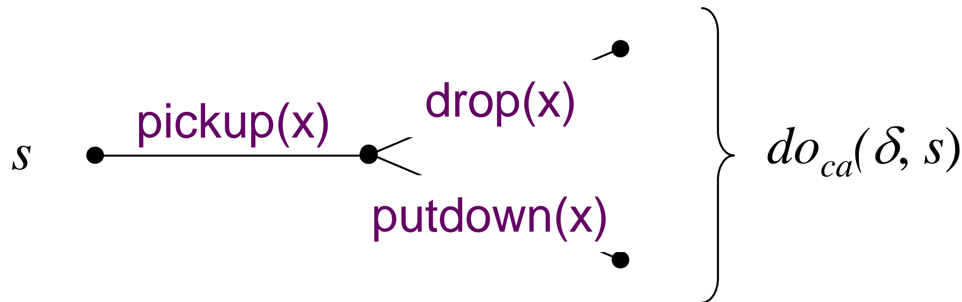
pickup(x); **if** hot(x) **then** drop(x) **else** putdown(x) **endif**



Situation Resulting from Performing δ in s

Introduce the term $do_{ca}(\delta, s)$ to denote the situation resulting from performing δ in s.

pickup(x); **if** hot(x) **then** drop(x) **else** putdown(x) **endif**



$do_{ca}(\delta, s)$

The interpretation of $do_{ca}(\delta, s)$ is constrained by the following axioms which we add to D,

$$Do(\delta, S_0, do_{ca}(\delta, s)) \vee (\neg\exists s'.Do(\delta, S_0, s') \wedge \neg executable(do_{ca}(\delta, s))$$

Recall

$executable(s)$ is defined as $\forall a,s^*.\ do(a,s^*) \subseteq s \rightarrow Poss(a,s^*)$

Preconditions for Complex Action, δ

Introduce the term $Poss_{ca}(\delta, s)$ to denote the preconditions for δ.

Its definition is captured tidily by the inductive definition of Do. I.e.,

$$Poss_{ca}(\delta, s) \equiv \exists s'.Do(\delta, S_0, s')$$

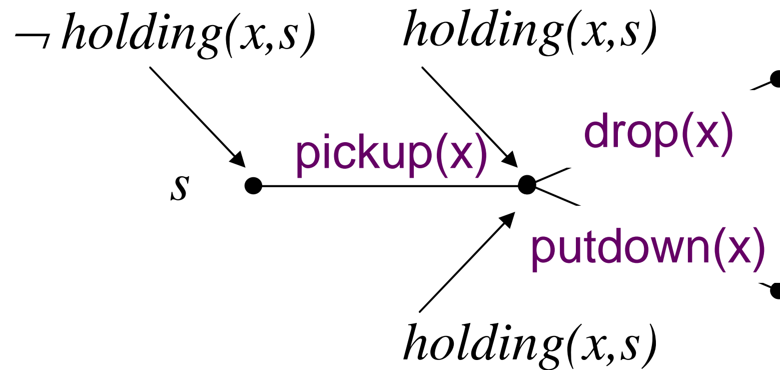
E.g.,

$$Poss_{ca}(a_1;a_2, s) \equiv \exists s'. Poss(a_1,s) \wedge s' = do(a_1,s) \wedge Poss(a_2,s')$$

Preconditions for Complex Action, δ

$Poss_{ca}(\delta, s) \equiv \exists s'.Do(\delta, S_0, s')$

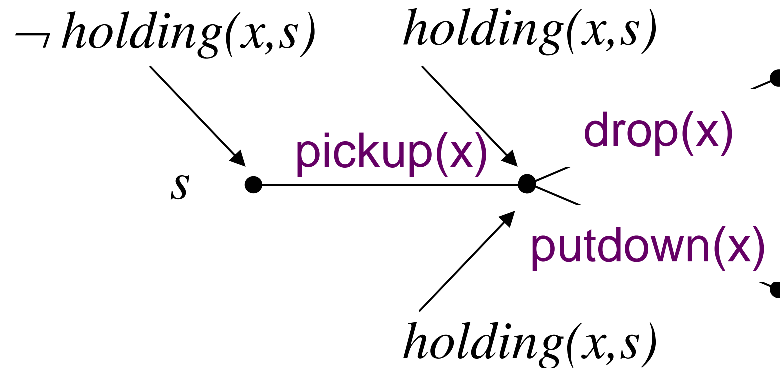
E.g.,



Preconditions for Complex Action, δ

$$Poss_{ca}(\delta, s) \equiv \exists s'.Do(\delta, S_0, s')$$

E.g.,

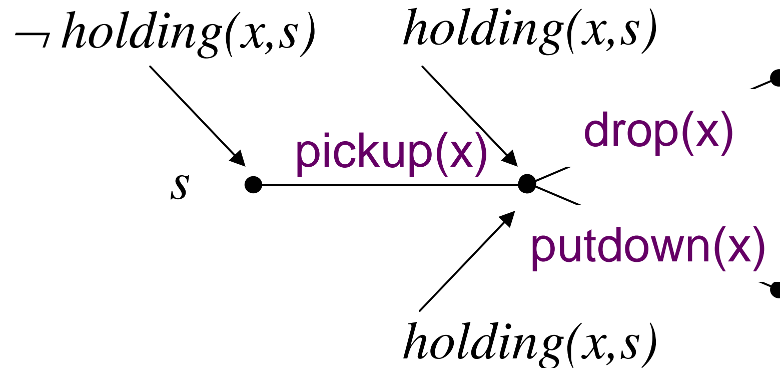


$$Poss_{ca}(\delta, s) \equiv Poss(pickup(x),s) \wedge s'' = do(pickup(x),s) \wedge$$
$$((hot(x,s'') \wedge Poss(drop(x),s'') \wedge s' = do(drop(x),s''))$$
$$\vee (\neg hot(x,s'') \wedge Poss(putdown(x),s'') \wedge s' = do(putdown(x),s'')))$$

Preconditions for Complex Action, δ

$$Poss_{ca}(\delta, s) \equiv \exists s'.Do(\delta, S_0, s')$$

E.g.,



$$Poss_{ca}(\delta, s) \equiv \exists s',s''.\ Poss(pickup(x),s) \wedge s'' = do(pickup(x),s) \wedge ((hot(x,s'') \wedge Poss(drop(x),s'') \wedge s' = do(drop(x),s'')) \vee (\neg hot(x,s'') \wedge Poss(putdown(x),s'') \wedge s' = do(putdown(x),s'')))$$

$$= \exists s',s''.\ \neg holding(x,s) \wedge s'' = do(pickup(x),s) \wedge ((hot(x,s'') \wedge holding(x,s'') \wedge s' = do(drop(x),s'')) \vee (\neg hot(x,s'') \wedge holding(x,s'') \wedge s' = do(putdown(x),s'')))$$

Make it Markovian: Action Precondition Axioms

Action Precondition Axioms, D_{caap}, one for every δ

$$Poss_{ca}(\delta, s) \equiv R^s[\exists s'.Do(\delta, S_0, s')]$$

where R^s is a regression rewriting operator relative to s, using D_{SS}

E.g. (continued),

$$\begin{aligned} Poss_{ca}(\delta, s) \equiv R^s[\exists s',s''.\ & Poss(pickup(x),s) \wedge s'' = do(pickup(x),s) \wedge \\ & ((hot(x,s'') \wedge Poss(drop(x),s'') \wedge s' = do(drop(x),s'')) \\ & \vee (\neg hot(x,s'') \wedge Poss(putdown(x),s'') \wedge s' = do(putdown(x),s'')))] \end{aligned}$$

$$= \neg holding(x,s)$$

Intermediate Successor State Axioms for δ

(Pseudo/Intermediate) successor state axioms can be defined as follows

$$Poss_{ca}(\delta, s) \rightarrow [F(x, do_{ca}(\delta, s)) \equiv \Phi^*(x, \delta, s)]$$

where $\Phi^*(x,s) = \exists s'.Do(\delta, S_0, s') \wedge F(x,s') \wedge s' = F(do_{ca}(\delta, s))$

E.g., Complex action δ is

pickup(x); **if** hot(x) **then** drop(x) **else** putdown(x) **endif**

$$\begin{aligned} Poss_{ca}(\delta, s) \rightarrow [broken(x, do_{ca}(\delta, s)) \equiv \\ \exists s', s''.\ Poss(pickup(x),s) \wedge s'' = do(pickup(x),s) \wedge \\ ((hot(x,s'') \wedge Poss(drop(x),s'') \wedge s' = do(drop(x),s'')) \\ \vee (\neg hot(x,s'') \wedge Poss(putdown(x),s'') \wedge s' = do(putdown(x),s''))) \\ \wedge broken(x,s') \wedge do_{ca}(\delta, s) = s'] \end{aligned}$$

Make it Markovian: Successor State Axioms

Successor State Axioms, D_{cass}, one for every fluent-δ pair

$$Poss_{ca}(\delta, s) \rightarrow [F(x, do_{ca}(\delta, s)) \equiv R^s[\Phi^*(x, \delta, s)]$$

where R^s is a regression rewriting operator relative to s, using D_{SS}.

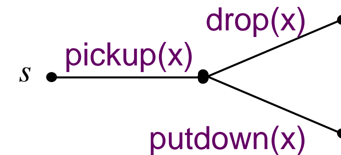
E.g., Complex action δ is

pickup(x); **if** hot(x) **then** drop(x) **else** putdown(x) **endif**

$$Poss_{ca}(\delta, s) \rightarrow [broken(x, do_{ca}(\delta, s)) \equiv$$
$$R^s[\exists s', s''.\ Poss(pickup(x),s) \wedge s'' = do(pickup(x),s) \wedge$$
$$((hot(x,s'') \wedge Poss(drop(x),s'') \wedge s' = do(drop(x),s''))$$
$$\vee (\neg hot(x,s'') \wedge Poss(putdown(x),s'') \wedge s' = do(putdown(x),s'')))$$
$$\wedge\ broken(x,s') \wedge do_{ca}(\delta, s) = s']$$

Make it Markovian: Successor State Axioms

$$Poss_{ca}(\delta, s) \rightarrow [F(x, do_{ca}(\delta, s)) \equiv R^s[\Phi^*(x, \delta, s)]$$



E.g. (continued), Complex action δ is

pickup(x); **if** hot(x) **then** drop(x) **else** putdown(x) **endif**

$$Poss_{ca}(\delta, s) \rightarrow [broken(x, do_{ca}(\delta, s)) \equiv \neg holding(x,s) \wedge$$
$$((hot(x,s) \wedge do_{ca}(\delta, s) = do(drop(x),do(pickup(x),s))$$
$$\vee$$
$$(\neg hot(x,s) \wedge broken(x,s) \wedge do_{ca}(\delta, s) = do(putdown(x),do(pickup(x),s)))]$$

Note that the successor state axiom embeds the trajectories upon which the truth of the fluent F is predicate. Critical for nondeterministic actions!

Effect Axioms for Complex Actions

Effect Axioms, D_{caef}, up to one positive and one negative for every fluent-δ pair

$$Poss_{ca}(\delta, s) \wedge \varepsilon^{+}(x,s) \rightarrow F(x, do_{ca}(\delta, s))$$
$$Poss_{ca}(\delta, s) \wedge \varepsilon^{-}(x,s) \rightarrow \neg F(x, do_{ca}(\delta, s))$$

Proposition:

$$D \cup D_{caap} \cup D_{caSS} \models D_{caef}$$

Contributions

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation technique (sit calc & PDDL)

  PDDL + complex action description $\rightarrow$ new PDDL theory

  (semantically justified by formal definitions)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Evaluation of Efficiency of Planning

Contributions

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation technique (sit calc & PDDL)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Evaluation of Efficiency of Planning

Planning Methodology

Given:

- primitive action theory T_A with primitive actions A,
- set of complex actions Δ_A constructed from primitive actions A

1. COMPILE$[T_A, \Delta_A] \rightarrow T_A'$ (A' is new set of primitive actions)
2. PLAN$[T_A', \text{goal}] \rightarrow \text{plan}[A']$
3. REWRITE$[\text{plan}[A']] \rightarrow \text{plan}[A, \Delta_A]$
4. EXPAND$[\text{plan}[A, \Delta_A], T_A] \rightarrow \text{plan}[A]$

Observe: Steps 3,4 not necessary for Web Service composition

Planning Methodology: Situation Calculus

1. COMPILE$[T_A, \Delta_A] \rightarrow T_A'$ (A' is new set of primitive actions)
   <see details in paper>

2. PLAN$[T_A'$,goal$] \rightarrow$ plan$[A']$
   Deductive plan synthesis
   $T_A' \vdash \exists s.Goal(s)$

3. REWRITE[plan$[A']] \rightarrow$ plan$[A, \Delta_A]$
   <trivial, see details in paper>

2. EXPAND[plan$[A, \Delta_A], T_A] \rightarrow$ plan$[A]$
   - Rewrite plan: $a_1, \delta_1, a_2, a_3, \delta_2, \ldots$ as a Golog sequence, δ_G
     E.g., δ_G is $a_1; \delta_1; a_2; a_3; \delta_2 \ldots$
   - Generate expansion using Golog interpreter
     $T_A \vdash \exists s.Do(\delta_G, s, s')$
   - Rewrite sequence of actions from binding for s'

Planning Methodology: Situation Calculus

Theorem (informally stated):
Planning in T_A' is sound and complete with respect to planning in T_A. I.e., every plan our approach finds is also a plan in the original primitive action theory, and vice versa.

Planning Methodology: ADL-PDDL Planners

1. COMPILE$[T_A, \Delta_A] \rightarrow T_A'$ (A' is new set of primitive actions)
   <PDDL translator, based on theory in this paper>

2. PLAN$[T_A'$,goal$] \rightarrow$ plan$[A']$
   - Generate a plan in T_A' using an ADL-PDDL planner

3. REWRITE[plan$[A']] \rightarrow$ plan$[A, \Delta_A]$
   <trivial, see details in paper>

4. EXPAND[plan$[A, \Delta_A], T_A] \rightarrow$ plan$[A]$
   <straightforward>

Recall: Motivation

1. Automated Composition of Web Services.

2. Improving the Efficiency of Planning

   (by representing useful plan segments as complex actions
   and planning with these segments as primitive building blocks).

Contributions

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation technique (sit calc & PDDL)

  PDDL + complex action description $\rightarrow$ new PDDL theory

  (semantically justified by formal definitions)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Evaluation of Efficiency of Planning

Efficiency of Complex Action Planning

Compiling:
- offline & amortized over multiple plans (polynomial in no. of primitive actions).

Expansion:
- linear in the length of the plan & not always relevant.

Planning dominates:

i) Complex action operators have more complex preconditions and effects than primitive actions.

ii) the size of the search space will change

Search Space:
Consider breadth-first search, finite horizon planner. n ground actions, shortest successful path is of lenth l, so the search space is $O(n^l)$

Efficiency of Complex Action Planning

Compiling can be done offline and amortized over multiple plans, but compiling a complex action δ is polynomial in no. of primitive actions.

Expansion is linear in the length of the plan.

Planning dominates

i) Complex action operators have more complex preconditions and effects than primitive actions (linear slow down)

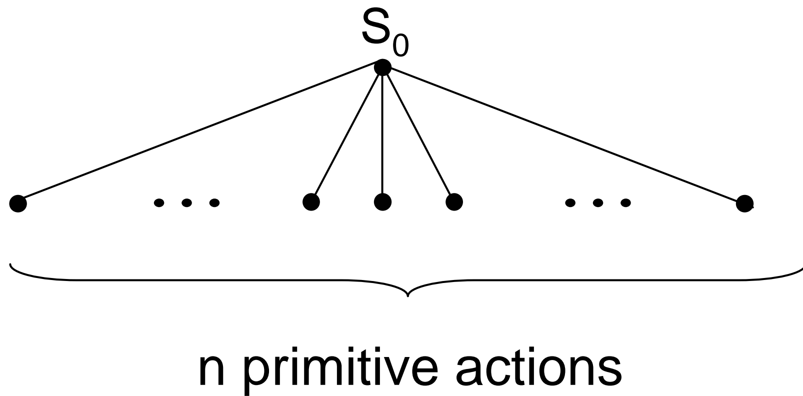
ii) the size of the search space will be change (critical)

Search Space:

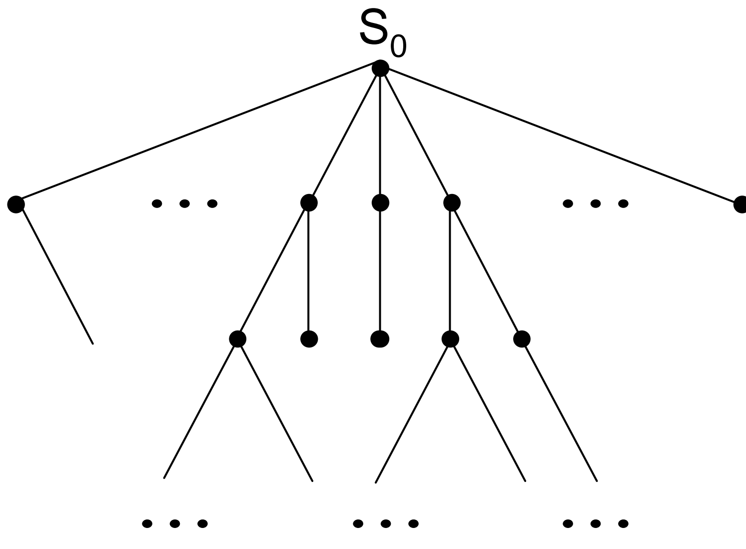
Consider breadth-first search, finite horizon planner. n ground actions, shortest successful path is of lenth l, so the search space is $O(n^l)$

Search Space

Consider the search space of a breadth-first search, finite horizon planner with n primitive actions.

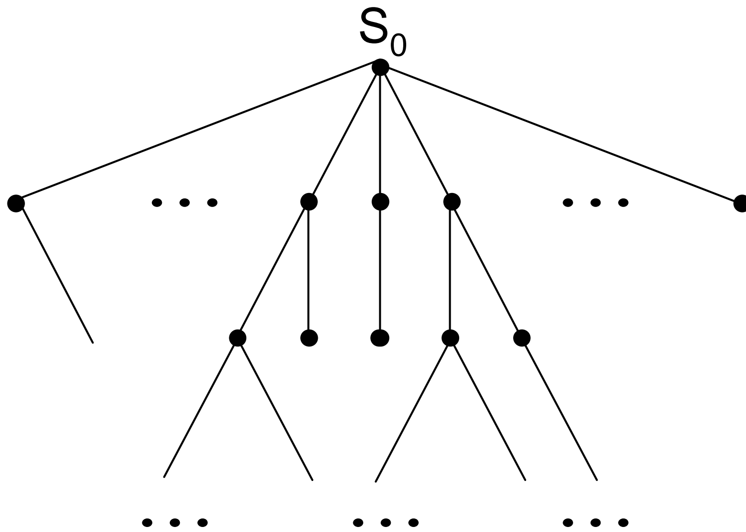


Search Space (cont.)



shortest plan of length l

Search Space (cont.)



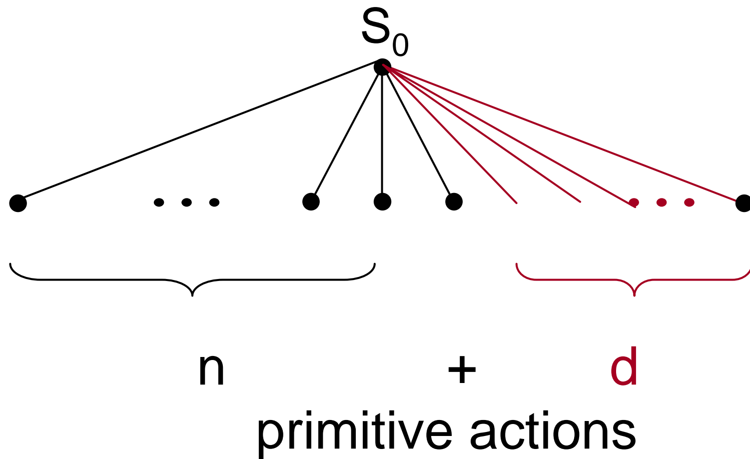
n primitive actions

shortest plan of length l

search space $O(n^l)$

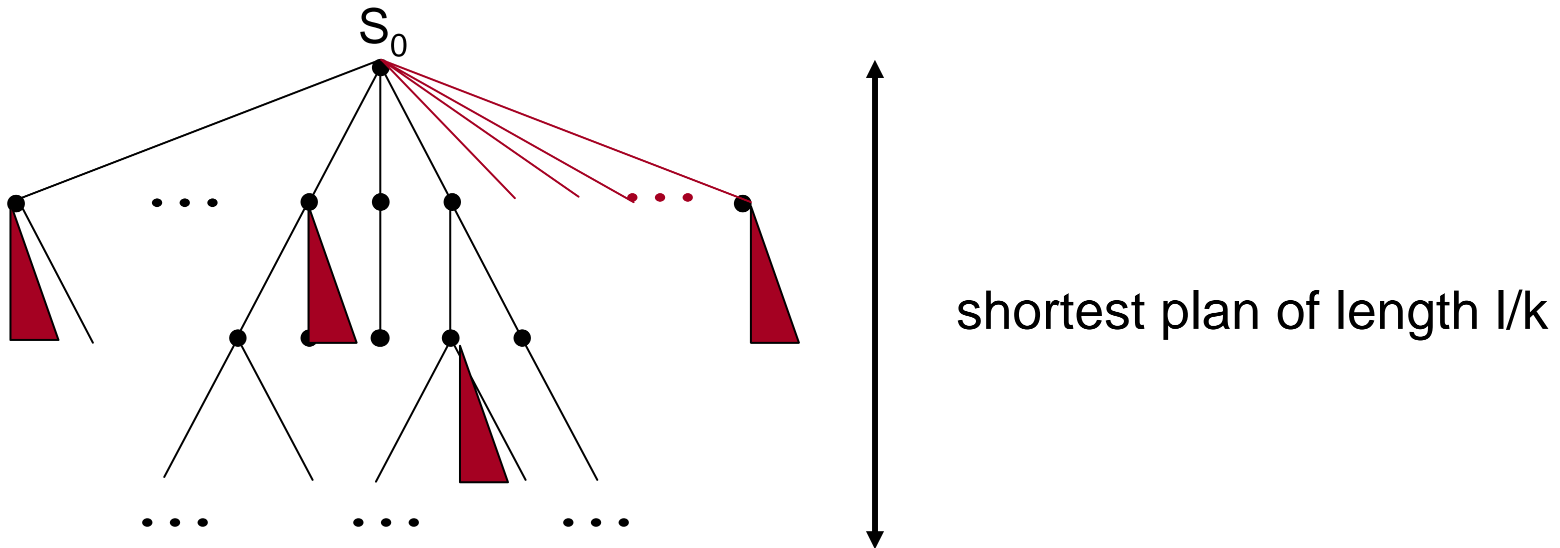
Search Space w/ “Complex Actions”

Add d new primitive actions representing the complex actions



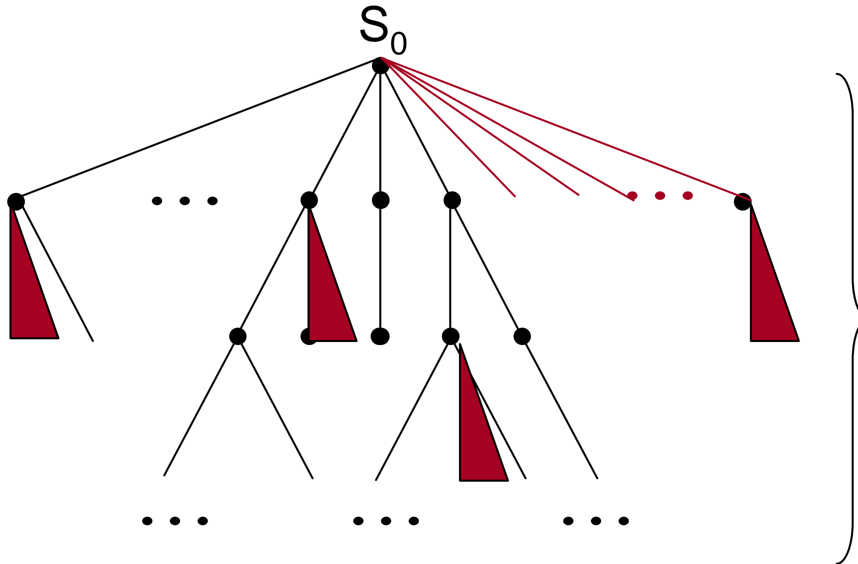
Search Space w/ “Complex Actions”

Assume the use of complex actions reduce plan length by 1/k



Search Space w/ "Complex Actions"

Assume the use of complex actions reduce plan length by 1/k



n+d primitive actions

shortest plan of length l/k

search space $O((n+d)^{(l/k)})$

Efficiency of Complex Action Planning

Planning with complex actions improves efficiency when the increase in the number of actions is compensated for by the decrease in the length of the plan. I.e., if $n^k-(n+d) > 0$

More generally, complex action planning improves efficiency when:

- $n^k-(n+d) > 0$
- complex actions are subplans in the shortest plan
- complex actions remove subgoal conflicts
  (reduces need for backtracking, enables fast hill-climbing techniques)

Illustrative Experiments

Tested the effectiveness of complex action planning on the AIPS briefcase domain, using FF [Hoffmann, Nebel, 01].

Experiments:
- increase no. of locations, packages
- with/without complex actions

New Complex Actions
- Move-object MO(locIniti, locObj, Obj, locFinal)
- LOC(loc-bc,loc)

	#loc:5,#pack:20	#loc:6,#pack:30	#loc:7,#pack:42
Briefcase	5549(1.39)	201,006(2261)	? (>40)
Briefcase+MO	859 (11.83)	2345 (201.47)	5195 (2211)
Briefcase+LOC	75(.08)	139 (.27)	260 (.85)

No. of nodes visited and time of run (seconds)*

* Experiments run on Sun Sparc v9, 2x 750MHz, 4GB memory

Summary: Planning with Complex Actions

Motivation:

1. Automated Web Service Composition
2. Improve the Efficiency of Planning

Contributions:

- Formal definition of successor situation, preconditions & effects of complex actions
- Automated compilation techniques (sit calc & PDDL)
- Planning Methodology
  - SitCalc+deductive synthesis (Soundness and Completeness)
  - Operator-based planner
- Illustrated improvement in the of efficiency of planning. Analysis bears out conditions underwhich this is true.