Method Composition through Operator Pattern Identification

Maurício Cecílio Magnaguagno, Felipe Meneguzzi
School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
mauricio.magnaguagno@acad.pucrs.br, felipe.meneguzzi@pucrs.br

Introduction

- **Classical planning description (PDDL)**
  - Easier to describe
  - Harder to solve
- **Hierarchical task networks (HTN) and macros**
  - Harder to describe
  - Save time by focusing on certain actions/primitives

- We can start with PDDL and eventually jump to HTN

Introduction - Motivation

- Steps to convert classical domains to a Hierarchical Task Network (HTN):
  - Cluster operators into methods.
  - Convert goals into tasks that use such methods.
  - Repeat this process for every domain...
  - Notice sub-problems share method construction.
  - Modify old method to match new domain.

- Repetitive process for a human
- Can we automate such process?

Background

Classical Planning

- Initial state
- Goal state
- Use actions/operators
- Optimality is search/heuristic dependent
- Anarchical/flat description
  - Easier to make/maintain
  - Harder to solve

Hierarchical Planning

- Initial state
- Task list
- Use operators and methods
- Optimality is description dependent
- Hierarchical description
  - Harder to make/maintain
  - Easier to solve

Classical Planning - PDDL

```
(define (domain dependency)
  (:requirements :strips :typing :negative-preconditions)
  (:predicates (have ?a ?x) (got_money ?a) (happy ?a))

  (:action work
   :parameters (?a - agent)
   :precondition (not (got_money ?a))
   :effect (and (not (happy ?a)) (got_money ?a)) )

  (:action buy
   :parameters (?a - agent ?x - object)
   :precondition (and (got_money ?a) (not (have ?a ?x)))
   :effect (and (not (got_money ?a)) (have ?a ?x)) )

  (:action give
   :parameters (?a ?b - agent ?x - object)
   :precondition (and (have ?a ?x) (not (have ?b ?x)))
   :effect (and (not (have ?a ?x)) (have ?b ?x) (happy ?b)) ))
```

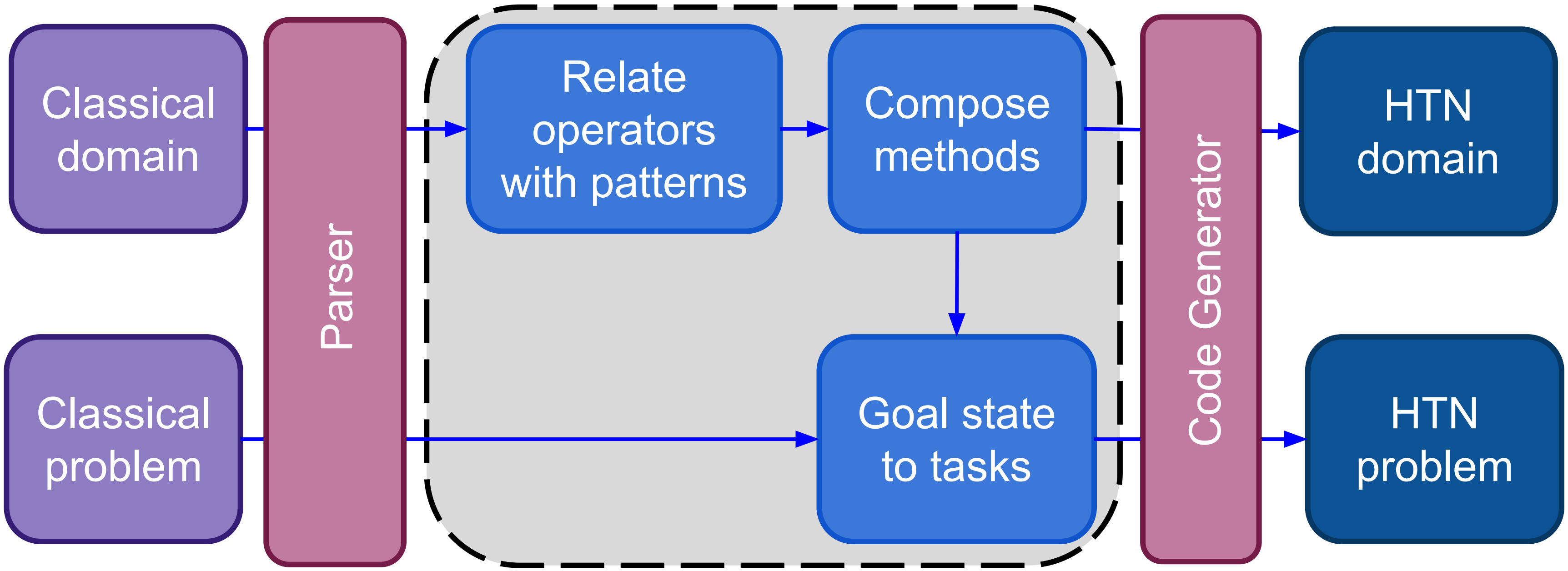
```
(define (problem pb1)
  (:domain dependency)
  (:objects
    ana bob - agent
    gift - object
  )
  (:init
    (got_money bob)
  )
  (:goal
    (happy bob)
  )
)
```

HTN - JSHOP

```
(defdomain dependency (
  (:operator (!work ?a)
    ((agent ?a) (not (got_money ?a)))
    ((happy ?a))
    ((got_money ?a))
  )
  ...
  (:method (work_to_buy_to_give_gift_to ?a)
    do-nothing
    ((object ?gift) (have ?a ?gift) (happy ?a))
    ()

    somebody-have-gift
    ((object ?gift) (not (have ?a ?gift)) (agent ?b) (have ?b ?gift))
    ((!give ?b ?a ?gift))

    got-money
    ((object ?gift) (not (have ?a ?gift)) (agent ?b) (got_money ?b))
    ((!buy ?b ?gift) (!give ?b ?a ?gift))
    ...
  )
)
```

```
(defproblem pb1 dependency
  (;init
    (agent ana)
    (agent bob)
    (object gift)
    (got_money bob)
  )
  (;tasks
    (work_to_buy_to_give_gift_to bob)
  )
)
```

Domain Knowledge Construction



- Identifies subproblems based on PDDL operators only
- Requires no annotations
- Requires no examples (plan traces)

Classify Predicates

Algorithm 3 Classification of predicates into irrelevant, constant or mutable

```
function CLASSIFY_PREDICATES(predicates, operators)
    predicate_types ← Table
    eff ← EFFECTS(operators)
    pre ← PRECONDITIONS(operators)
    for each p ∈ predicates do
        if p ∈ eff
            predicate_types[p] ← mutable
        else if p ∈ pre
            predicate_types[p] ← constant
        else
            predicate_types[p] ← irrelevant
    return predicate_types
```

We partition to understand what is dynamic or static in the domain.

Identifying Operator Patterns

- **Swap operator pattern**
  - Zero or more actions swap the value of a predicate to satisfy certain condition
- **Dependency operator pattern**
  - Action/method precondition is satisfied by the effects of another action/method
- **Free variable operator pattern**
  - Action/method have free variable(s) to be decided at run-time

Swap Operator Pattern

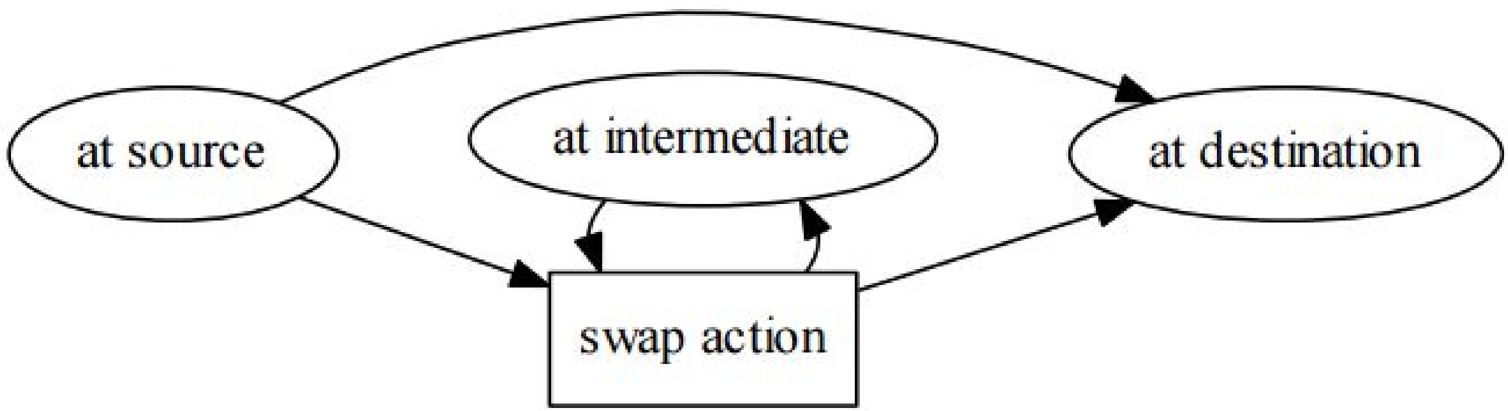
Preconditions	Effects
(**connected** ?location1 ?location2)*	
(**at** ?agent ?location1)	(not (**at** ?agent ?location1))
(not (**at** ?agent ?location2))**	(**at** ?agent ?location2)

Preconditions	Effects
(**trade** ?item1 ?item2)*	
(**have** ?agent ?item1)	(not (**have** ?agent ?item1))
(not (**have** ?agent ?item2))**	(**have** ?agent ?item2)

*Constant predicate

**Optional precondition, mutually exclusive (can only be at one place at a time)

Swap Operator Pattern



Swap Operator Pattern

- Method
  - Base case
  - One recursive case for each operator that swaps the same predicate
- Cache
  - Visit operator
  - Unvisit operator

```
(:operator (!!visit_predicate ?object ?current)
  ()
  ()
  ( (visited_predicate ?object ?current) )
)
```

```
(:operator (!!unvisit_predicate ?object ?current)
  ()
  ( (visited_predicate ?object ?current) )
  ()
)
```

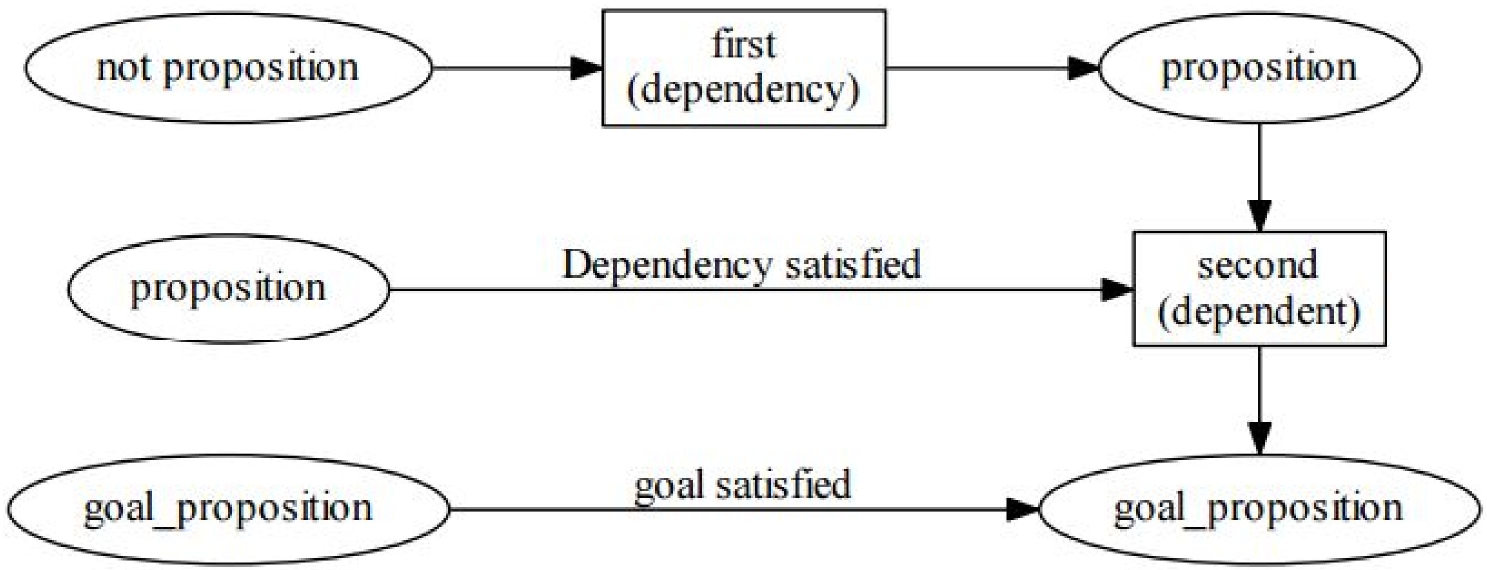
```
(:method (swap_predicate ?object ?goal)

base
( (predicate ?object ?goal) )
()

using_operator
(
        (constraint ?current ?intermed)
        (swap_predicate ?object ?current)
        (not (predicate ?object ?goal))
        (not (visited_predicate ?object ?intermed))
)
(
        (!operator ?object ?current ?intermed)
        (!!visit_predicate ?object ?current)
        (swap_predicate ?object ?goal)
        (!!unvisit_predicate ?object ?current)
)
)
```

Dependency Operator Pattern

Preconditions	Effects
(**connected** ?location1 ?location2)	
(**at** ?agent ?location1)	(not (**at** ?agent ?location1))
(not (**at** ?agent ?location2))	(**at** ?agent ?location2)

Preconditions	Effects
(**at** ?agent ?location)	
(**dropped** ?item ?location)	(not (**dropped** ?item ?location))
	(**have** ?agent ?item)

Dependency Operator Pattern



Dependency Operator Pattern

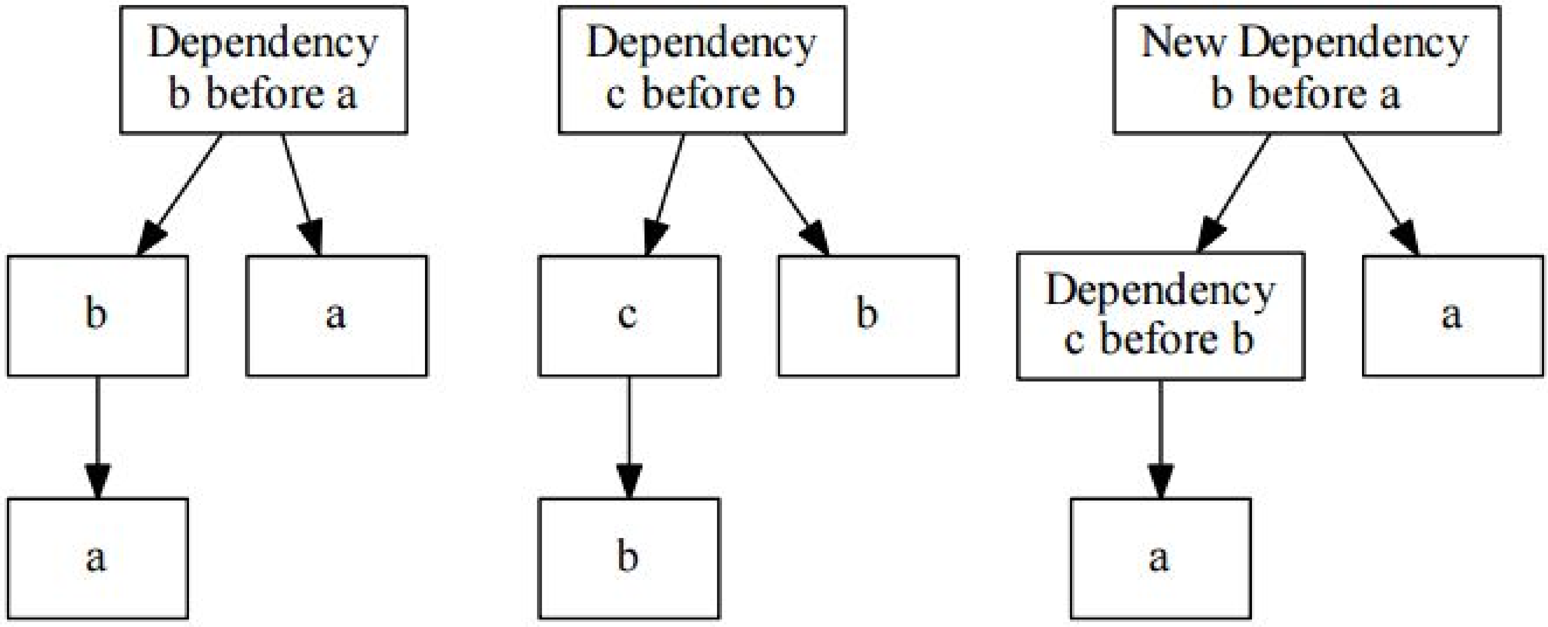
- Method
  - Goal satisfied
  - Satisfied
  - Unsatisfied

```
(:method (dependency_first_before_second ?param)
  goal_satisfied
  ( (goal_predicate) )
  ()
)
```

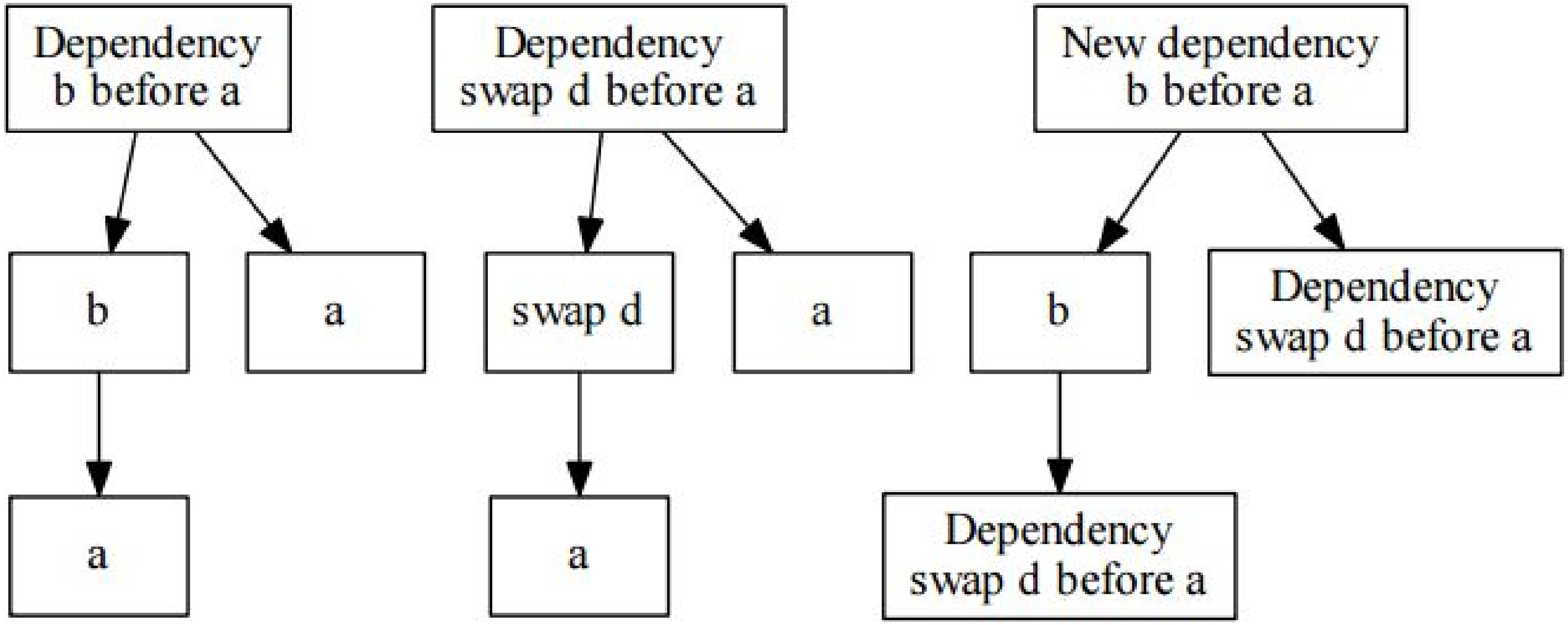
```
(:method (dependency_first_before_second ?param)
  satisfied
  ( (predicate ?param) )
  ( (!second ?param) )
)
```

```
(:method (dependency_first_before_second ?param)
  unsatisfied
  ( (not (predicate ?param)) )
  ( (!first ?param) (!second ?param) )
)
```

Dependency Injection



Dependency Injection



Free Variable Operator Pattern

- High-level pattern (beyond operators)
- Find value of variable at run-time
- Value is related to goals
  - (happy bob)
  - Who works?
  - What is the gift?

- We want to unify ?p3
- Discover value at run-time based on the preconditions of the original method

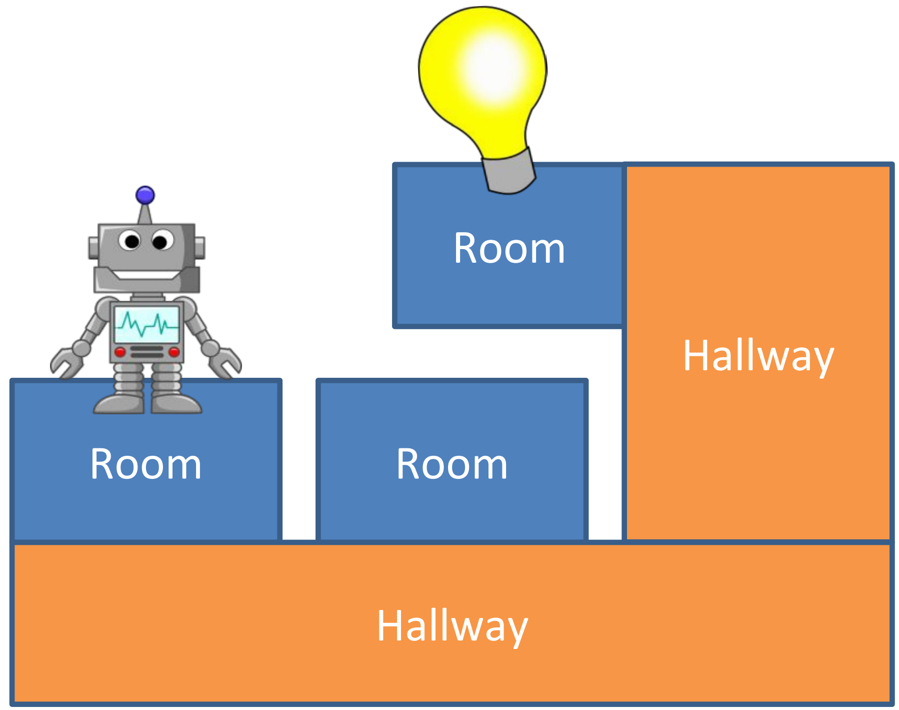
```
(:method (apply_op ?p1 ?p2 ?p3)
  apply_op_with_3_parameters
  ((precond1 ?p1 ?p2) (precond2 ?p2 ?p3))
  ((!op ?p1 ?p2 ?p3))
)

(:method (unify_apply_op ?p1 ?p2)
  unify_parameter_p3
  ((precond2 ?p2 ?p3))
  ((apply_op ?p1 ?p2 ?p3))
)
```

Composing methods and tasks

- Classify operators
- Add methods to domain
- Relate goals to operator effects
- Find methods that contain such operators (and maintain such effects)
- Replace variables of tasks using goal state predicates
- Ground tasks or create a free-variable methods and tasks to ground at run-time
- Add tasks to task list

Use Case: Rescue Robot Domain

- Operators
  - Enter
  - Exit
  - Move
  - Report



This domain was created by Kartik Talamadupula and Subbarao Kambhampati.

Use Case: Rescue Robot Domain

Preconditions	Effects
(**connected** ?l1 ?l2)	
(hallway/room ?l1)	
(hallway/room ?l2)	
(**at** ?agent ?l1)	(not (**at** ?agent ?l1))
(not (**at** ?agent ?l2))	(**at** ?agent ?l2)

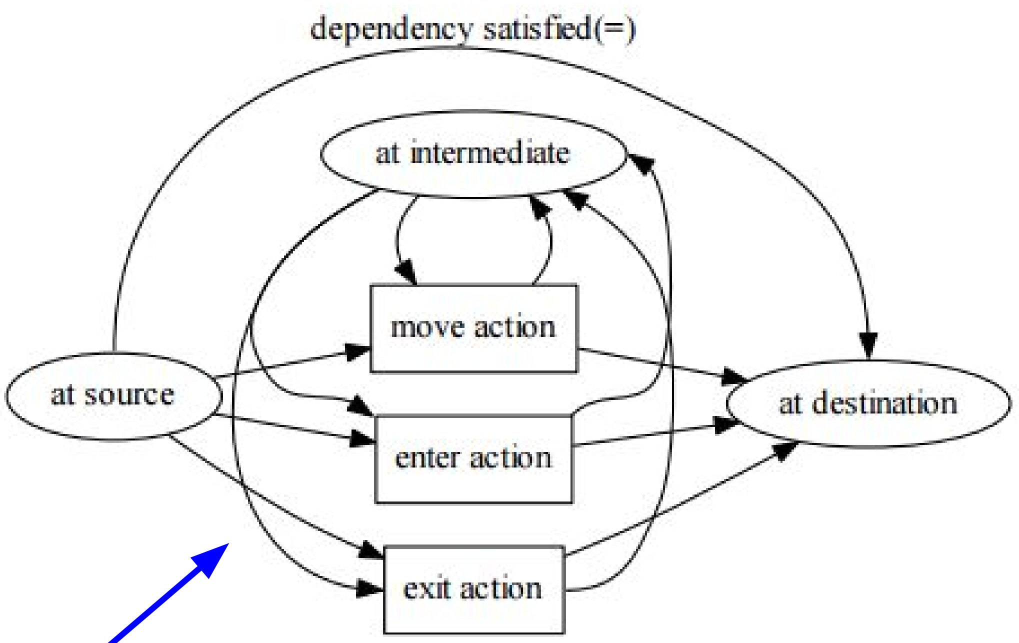
Move
Enter
Exit

Preconditions	Effects
(**in** ?l1 ?b)	
(**at** ?agent ?l1)	
(not (**reported** ?agent ?b))	(**reported** ?agent ?b)

Report

Use Case: Rescue Robot Domain

- Patterns
  - Swaps:
    - Enter
    - Exit
    - Move
  - Dependencies
    - Enter ⇒ Report
    - Exit ⇒ Report
    - Move ⇒ Report
- Methods
  - Swap_at = Enter|Exit|Move
  - Swap_at_before_Report = Swap_at ⇒ Report



Brute-force Fallback

- If no permutation of tasks obtain the goal state we fallback to a modified version of the method described in *Complexity Results for HTN planning*

- We mark actions to avoid infinite loops (each action can be used N times)

In this transformation, the HTN representation uses the same constants and predicates used in the STRIPS representation. For each STRIPS operator o, we declare a primitive task f with the same effects and preconditions as o. We also use a dummy primitive task f_d with no effects or preconditions. We declare a single compound task symbol t. For each primitive task f, we construct a method of the form

$$\boxed{perform[t]} \Longrightarrow \boxed{\boxed{do[f]} \longrightarrow \boxed{perform[t]}}$$

We declare one last method $\boxed{perform[t]} \Rightarrow \boxed{do[f_d]}$. Note that t can be expanded to any sequence of actions ending with f_d, provided that the preconditions of each action are satisfied. The input task network has the form $[(n : perform[t]), (n, G_1) \wedge \ldots \wedge (n, G_m)]$ where $G_1, \ldots, G_m$ are the STRIPS-style goals we want to achieve.

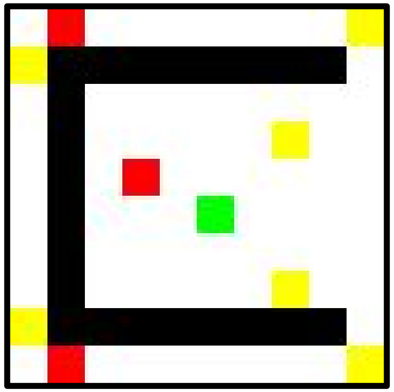
Experimentation - Rescue Robot Robby

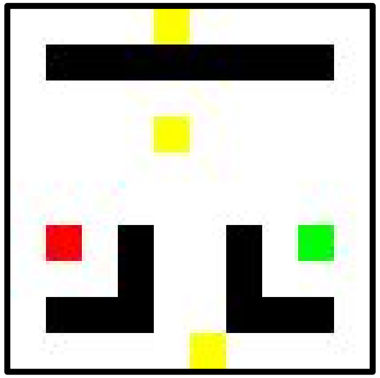
Robby Problem	Classical planner	HTN Brute force	HTN Patterns + Brute force
pb1	0.000	0.008	0.021
pb2	0.000	2.594	0.025
pb3	0.001	Time-out (> 100s)	0.072
pb4	0.000	4.399	0.031
pb5	0.001	20.812	0.062
pb6	0.000	Time-out (> 100s)	0.046

Experimentation - Goldminers

Goldminers	Classical planner	HTN Brute force	HTN Patterns + Brute force
pb1	Time-out (> 100s)	Time-out (> 100s)	6.270
pb2	Time-out (> 100s)	Time-out (> 100s)	3.668

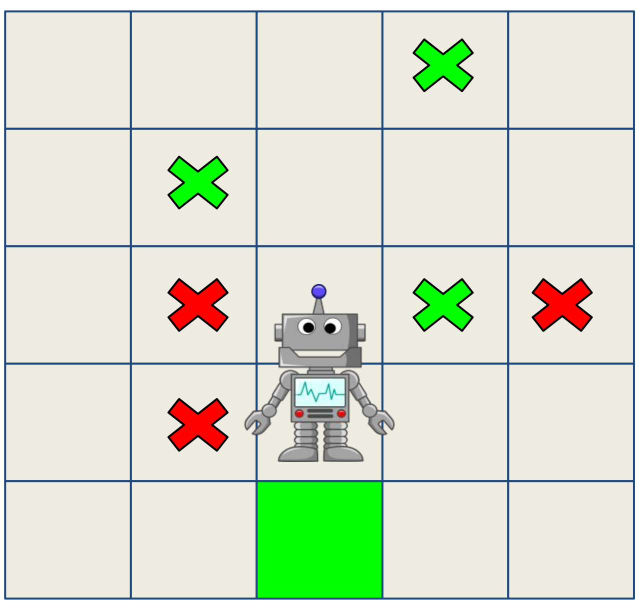
- Obstacle
- Deposit
- Gold
- Agent

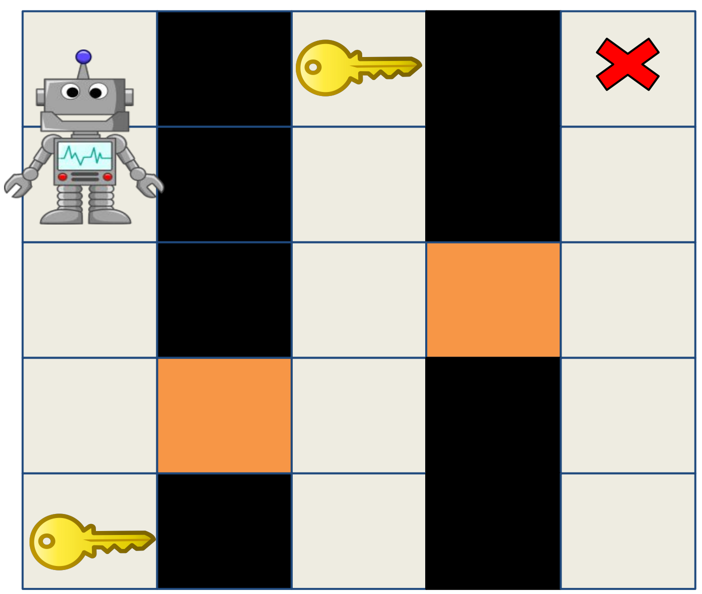




Experimentation - Almost

- Domains from IPC 2014
- ChildSnack
  - Fails to see where to start decomposing: moving tray to the kitchen
- FloorTile
  - Fails to see when to use paint-up: first row
- Grid
  - Fails to see multiple journeys are required to reach goal position: multiple keys





Conclusions and Future Work

- It is possible to automatically obtain an HTN description from a classic description without examples/annotations
    - At least for some domains
- May be used to increase domain knowledge on systems that can achieve speed-up when such knowledge is available
- Erol et al. brute-force conversion
- Lotinac and Jonsson, invariance analysis
- Shivashankar et al. GoDeL

- Improve the efficiency and quality of the resulting HTN domain knowledge
- Selectively choose methods for decompositions rather than performing blind search