Course "Automated Planning: Theory and Practice"
Chapter 02: Classical Planning and PDDL

Teacher: Marco Roveri - marco.roveri@unitn.it

M.S. Course: Artificial Intelligence Systems (LM)
A.A.: 2023-2024
Where: DISI, University of Trento
URL: https://bit.ly/3zOkGk8

Last updated: Sunday 17th September, 2023

Terms of Use and Copyright

USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2023-2024.

SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

STRIPS

History

In 1971 STRIPS (**S**tanford **R**esearch **I**nstitute **P**roblem **S**olver) was developed as an automated planner.
Later, the name STRIPS has been used to refer only to the formal language of the inputs. Fikes and Nilsson [2]

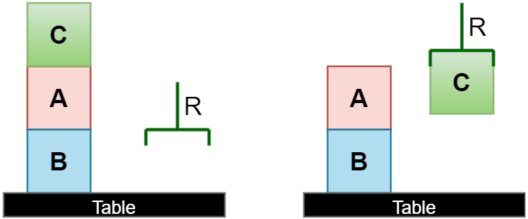
- State is database of ground literals
- If literal is not in database, assumed to be false
- Effects of actions represented using add and delete lists (insert and remove literals from database)
- No explicit representation of time
- No logical inference rules

Something more expressive 1

- Conditional Effects

```
(:action pickup
  :parameters (?X)
  :precondition (and (BLOCK ?X) (free) (clear ?X) (On ?X ?Y)
  :effect
   (and (holding ?X) (when (and (BLOCK ?Y)) (and (clear ?Y))) ; add
        (not (free)) (not (on ?X ?Y)))                        ; delete
```

- Quantified effects `(forall (?x) (when (and (in ?x ?y))...))`
- Disjunctive and negated preconditions `(or (conn ?x ?y) (not (in ?y ?x)))`



Something more expressive 2

- Functional effects `(increment ?x 10)`
- Disjunctive effects
- Probabilistic effects
- Duration (actions no more instantaneous)
- External events, agents, concurrent events, etc
- Inference operators

PDDL

Planning **D**omain **D**efinition **L**anguage : standard specification language for classical planning

Objects	Things in the world
Predicates	Properties of the objects
Initial state	The state of the world we start in
Goal specification	Things we want to be true
Actions	Ways of changing the state of the world

History

- 1998: PDDL McDermott [10].
- 2002: PDDL2.1, Levels 1-3 Fox and Long [3] (numeric fluents, plan-metrics, durative/continuous actions)
- 2004: PDDL2.2 Edelkamp [1] (timed initial literals)
- 2006: PDDL3 Fox and Long [4] (state-trajectory constraints and preferences)
- 2008: PDDL3.1 Helmert [9] (object-fluents, functions' range now could be not only integer/real, but it could also be any object-type)

PDDL: the structure

- PDDL separates domain and problem instances

Domain File

```
(define
  (domain dock-worker-robots)
  ... ; Semicolon is used to
      ;  start line comments
)
```

Problem Instance File

```
(define
  (problem dwr-problem-1)
  (:domain dock-worker-robots)
  ...
)
```

- A **domain file** for predicates, and actions
- A **problem file** for objects, initial states, and goal descriptions

Which one can we re-use?

Domain and Problem Definition

- Domain files declare their espressivity requirements

```
(:define (domain dock-worker-robots)
  (:requirements
     :strips ;; Standard level of expressivity
             ;; Tell planner which featurs are needed
             ;; There are more and we will see (some of) them
     ...)
  ;; Remaining domain information here
  ...
)
```

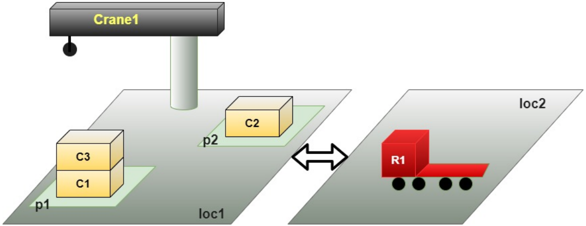
Warning

Many planners' parsers ignore (silently) espressivity specifications!!

PDDL Objects: Types

- In PDDL constants (can) have types, defined in the domain

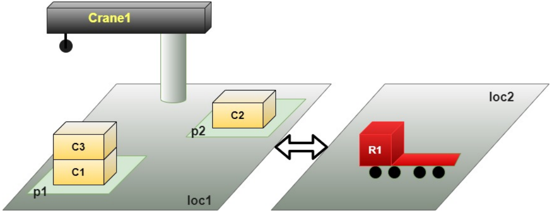
```
(define (domain dock-worker-robots)
  (:requirements :strips :typing)
  (:types
     location   ; there are several locations in the harbor
     pile       ; piles attached to a location, holds a pallet plus
     robot      ; a stack of containers, robots holds at most 1
     crane      ; container, only one robot per location, crane
     container) ; belongs to a location to pick up a container
)
```



PDDL Objects: Type hierarchies

- Many planners support type hierarchies
  - Convenient, but often not used in domain examples
  - Predefined "topmost super-type": `object`

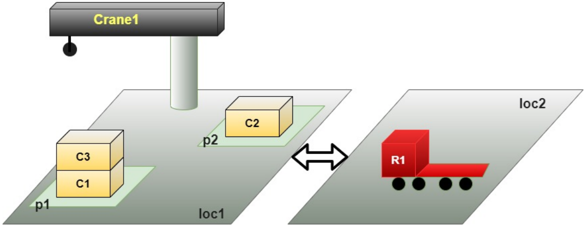
```
(define (domain dock-worker-robots)
  (:requirements :strips :typing)
  (:types
    movable - object
    container robot - movable ; container and robots are movable objects
    ...)
)
```



PDDL Objects: Object Definition

- Instance specific constants called objects

```
(define (problem dwr-problem-1)
  (:domain dock-worker-robots)
  (:objects r1                   - robot
            loc1 loc2            - location
            k1                   - crane
            p1 p2                - pile
            c1 c2 c3 pallet - container)
)
```

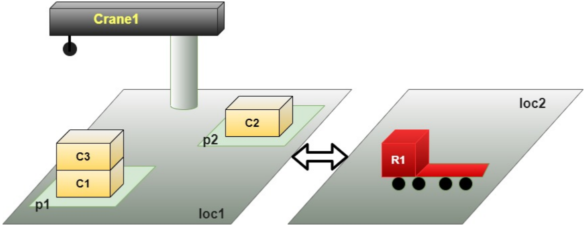


PDDL Objects: PDDL Constants

- Some instance object exist in all problem instances

Defined once - used in *all* problem instance files, as well as in the domain definition file

```
(define (domain dock-worker-robots)
  (:requirements :strips :typing)
  (:types ...)
  (:constants
    crane1 - crane
   )
)
```



PDDL Predicates

- In PDDL: Lisp-like *syntax* for predicates, atoms, ...

Variables are prefixed with "?"

```
(define (domain dock-worker-robots)
 (:requirements :strips :typing)
 (:types ...)
 (:predicates
  (adjacent ?l1  ?l2  - location)           ; can move from ?l1 to ?l2
  (attached ?p - pile ?l - location)        ; pile ?p is attached to location ?l
  (belong   ?k - crane ?l - location)       ; crane ?k belongs to location ?l
  (at       ?r - robot ?l - location)       ; robot ?r is at location ?l
  (occupied ?l - location)                  ; there is a robot at location ?l
  (loaded   ?r - robot ?c - container)      ; robot ?r is loaded with container ?c
  (unloaded ?r)                             ; robot ?r is empty
  (holding  ?k - crane ?c - container)      ; crane ?k holds container ?c
  (empty    ?k - crane ?c)                  ; crane ?k does not hold anything
  (in       ?c - container ?p - pile)       ; container ?c is somewhere in pile ?p
  (top      ?c - container ?p - pile)       ; container ?c is on top of pile ?p
  (on       ?c1 - container ?c2 - container) ; container ?c1 is on top of container ?c2
 )
)
```

Predicates: Modeling Issues

- Single or multiple predicates?

3 predicates with similar meaning

```
(define (domain dock-worker-robots)
 (:requirements :strips :typing)
 (:types ...)
 (:predicates
  (belong   ?k - crane ?l - location)         ; crane ?k belongs to location ?l
  (at       ?r - robot ?l - location)         ; robot ?r is at location ?l
  (occupied ?l - location)                    ; there is a robot at location ?l
  ...
```

- Could use type hierarchies instead - *supported in most planners*

```
(define (domain dock-worker-robots)
 (:requirements :strips :typing)
 (:types robot crane container pile - thing
         location)
 (:predicates
  (at       ?t - thing ?l - location)        ; thing ?t is at location ?l
  ...
```

Predicates: Modeling Issues (cont.)

- Domains *often* contain "duplicate information"
  - A location is occupied $\leftrightarrow$ there is some robot at the location

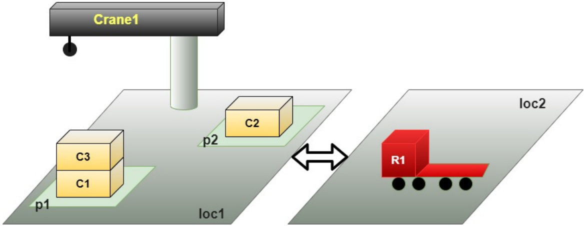
```
(define (domain dock-worker-robots)
 (:requirements :strips :typing)
 (:types ...)
 (:predicates
  (at       ?r - robot ?l - location)        ; robot ?r is at location ?l
  (occupied ?l - location)                   ; there is a robot at location ?l
  ...
```

  - Strictly speaking `occupied` is redundant!
    - Still necessary in some planners to help heuristics
    - There is no support for quantification: `(exists (?r) (at ?r ?l))`
    - $\Longrightarrow$ have to write `(occupied ?l)` instead
    - $\Longrightarrow$ have to provide this information and update it in actions (see later)

PDDL: Initial State

- Set (list) of true atoms!

```
(define (problem dwr-problem-1)
 (:domain dock-worker-robots)
 (:objects ...)
 (:init
    (attached p1 loc1) (in c1 p1) (on c1 pallet) (in c3 p1) (on c3 c1) (top c3 p1)
    (attached p2 loc1) (in c2 p2) (on c2 pallet) (top c2 p2)
    (at r1 loc2) (unloaded r1) (occupied loc2)
    (adjacent loc1 loc2) (adjacent loc2 loc1)  )
)
```



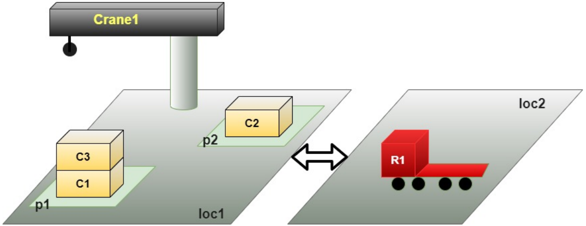
PDDL: Goal States

- The `:strips` level supports only positive conjunctive goals
  - Examples: "Container 1 and 3 should be in pile 2", *we do not care about their order, or any other fact*!

```
(define (problem dwr-problem-1)
  (:domain dock-worker-robots)
  (:objects ...) (:init ...)
  (:goal
     (and (in c1 p2) (in c3 p2))
)
```

Written as a formula (`and ...`), not as a set!

Other PDDL levels support for "or", "forall", "exists", ...



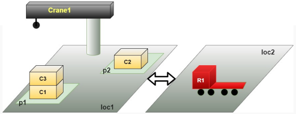
PDDL: Goal States (cont.)

- *Some* planners supports conjunction of positive/negative literals
  - Examples:
    - "Container 1 and 3 should be in pile 2"
    - "Container 2 should *not* be in pile 4"
    - `(:requirements :negative-preconditions ...)`

```
(define (problem dwr-problem-2)
 (:domain dock-worker-robots)
 (:objects ...) (:init ...)
 (:goal (and (in c1 p2) (in c3 p2) (not (in c2 p4)))
)
```

  - Buggy support in some planners!
    - Can be worked around
    - Define `(outside ?c - container ?p - pile)` predicate as inverse of `in`
    - Ensure actions update this predicate

```
(define (problem dwr-problem-2)
 (:domain dock-worker-robots)
 (:objects ...) (:init ...)
 (:goal (and (in c1 p2) (in c3 p2) (outside c2 p4))
)
```



PDDL: Operators

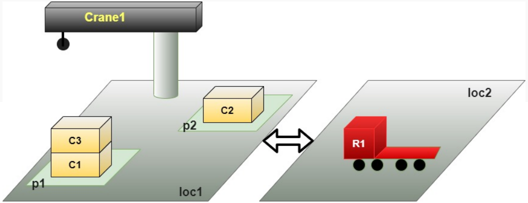
Typed parameters $\Longrightarrow$ can be instantiated with the intended objects

- Operators in PDDL are called (improperly) actions

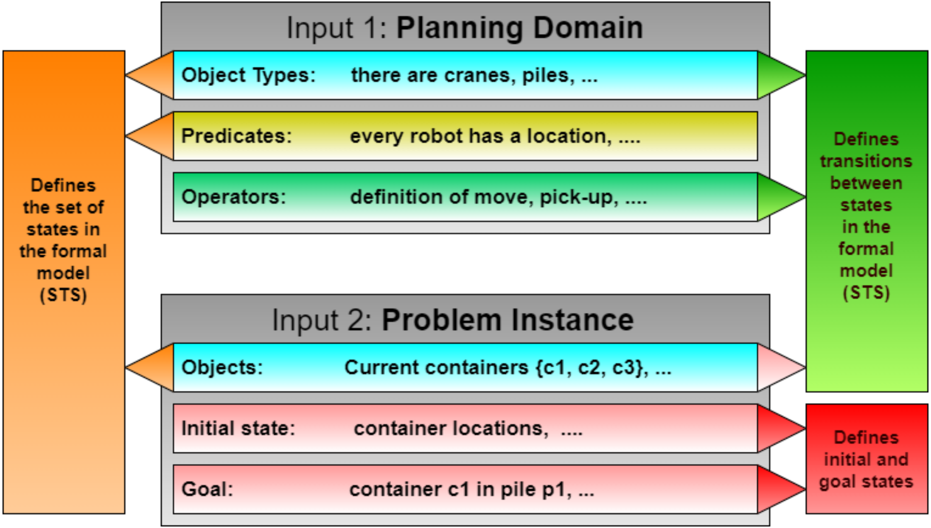
```
(define (domain dock-worker-robots)
 (:requirements ...) (:types ...) (:constants ...) (:predicates ...)
 (:action move
   :parameters (?r - robot ?from - location ?to - location)
    :precondition (and (adjacent ?from ?to)
                       (at ?r ?from)
                       (not (occupied ?to)))
    :effects (and (at ?r ?to) (not (occupied ?from))
                       (occupied ?to)
                       (not (at ?r ?from)))
   )

)
```

Again, written as logical conjunctions, not a set



From PDDL/strips to STS



Expressivity Requirements

```
:strips  :typing  :disjunctive-preconditions :equality
:existential-preconditions :universal-preconditions :quantified-preconditions
:conditional-effects
:action-expansions :foreach-expansions :dag-expansions
:domain-axioms :subgoal-through-axioms
:safety-constraints
:expression-evaluation
:fluents
:open-world
:true-negation
:adl
:ucpop
:numeric-fluents
:negative-preconditions
:durative-actions :durative-inequalities :continuous-effects
```

Expressivity Requirements (cont.)

- `:strips` allows to use add/delete effects
- `:typing` allows to use types
- `:disjunctive-preconditions` allows to use disjunctions in preconditions
  - `(or (walls-built ?x) (windows-fitted ?x))`
- `:equality` allows to check whether two objects are the same
  - `(not (= ?x ?y))`
- `:existential-preconditions` allows to use `exists` in goals and preconditions
  - `(exists (?c - crane) (crane-is-free ?c))`
- `:universal-preconditions` allows to use `forall` in goals and preconditions
  - `(forall (?c - crane) (crane-is-free ?c))`
- `:quantified-preconditions` equivalent to
  - `(:requirements :existential-preconditions :universal-preconditions)`

Expressivity Requirements (cont.)

- **:conditional-effects** Allows for the usage of **when** in expressing action effects. Essentially saying if something is true, then apply this effect too.

```
(when
   ;Antecedent
   (and (has-hot-chocolate ?p ?c) (has-marshmallows ?c))
   ;Consequence
   (and (person-is-happy ?p))
)
```

- **:domain-axioms** Allows to define axioms
  - `(:derived (clear ?x) (and (not (holding ?x)) (forall (?y) (not (on ?y ?x)))))`

Expressivity Requirements (cont.)

- `:action-expansions` Allows for usage of action expansions. This allows for the definition of variant condition and effects of actions. Essentially, we could define a `move` action to describe movement of say a person, but include different expansions to describe movement by plane, train, car or foot.
- `:foreach-expansions` allows to use `foreach` in action expansion
- `:dag-expansions` equivalent to
  - `(:requirements :action-expansions :foreach-expansions)`
- `:subgoal-through-axioms` allows to use axioms as sub-goals
- `:safety-constraints` allows to define predicates that must be valid at the end of the execution of a plan
- `:expression-evaluation` allows to use `eval` in axioms
  - `(eval (im-not-true ?a) (im-true ?b))`
- `:fluents` allows to use `(fluent t)` in axioms, his scope changed in PDDL2.1, and it is no longer clear its use!

Expressivity Requirements (cont.)

- `:numeric-fluents` allows to use functions that represents numeric values
  - `(:functions (battery-amount ?r - rover))`
- `:negative-preconditions` allows to use `not` in preconditions
- `:open-world` relaxes the closed-world assumption! It is no longer true that not specified predicates are assumed to be false!
- `:true-negation` Don't treat negation as failure, treat it how it is in first order logic. This requirement implies the existence of the `:open-world` requirement.
- `:adl`, `:ucpop` implies other requirements (see `https://planning.wiki/ref/pddl/requirements`)
- `:durative-actions`, `:durative-inequalities`, `:continuous-effects` are extensions to deal with durative actions (temporal planning, and hybrid planning aka PDDL+)

Expressivity Requirements (cont.)

Warning

Many planners' parsers ignore (silently) espressivity specifications!!

Modeling Tips

- Modeling properties in a first-order predicate representation has some advantages

color_of(chair, silver)	**Yes**	**Each atom is "separate"**
color_of(chair, red)	**-**	
color_of(chair, green)	**-**	**Good: can easily model 0 colors**
color_of(chair, blue)	**Yes**	**Good: Can easily model multiple colors**
color_of(chair, yellow)	**-**	

Modeling Tips (cont.)

- Let us model a "drive" operator for a truck
  - Natural parameters: the truck and the destination

```
(:action drive
  (:parameters (?t - truck ?dest - location)
  (:precondition ...)
  (:effect ...) )
```

  - Natural precondition:
    - There must be a path between the current location and the destination
    - Assume we have a predicate `(path-between ?from ?to - location)`
  - How do we continue?
    - `(:precondition (path-between ...something... ?dest)` ???
    - Cannot talk about the location of the truck - could have 0 or many locations
    - Can only test whether a truck is at some specific location: `(at ?t ?location)`

Modeling Tips (cont.)

- General approach: iterate and test!

```
(:precondition
   (forall (?from - location)
           (implies
              (at ?t ?from)
              (path-between ?from ?dest))))
```

Warning!!
Many planners do not support forall, implies, ...

- Trick
  - Add an additional parameter to the operator

```
(:action drive :parameters (?t truck ?from - location ?dest - location)
   :precondition ...
   :effect ...
)
```

  - Constrain the variable in the precondition
    - `:precondition (and (at ?t ?from) (path-between ?from ?dest))`
    - Can only apply to those instances of the operator where `?from` is the current location of the truck!

Modeling Tips (cont.)

These parameters are "extraneous" in the sense that they do not add choice: We can choose truck and dest (given some constraints); from is uniquely determined by state + other parameters!

- Example
  - Initially: `(at truck5 home)`
  - Action:

```
(:action drive :parameters (?t - truck ?from - location ?to - location)
   :precondition (and (at ?t from) (path-between ?from ?to))
   :effect ...
)
```

- Which actions are executable?
  - `(drive truck5 work home)` - no, precondition false: not `(at truck5 work)`
  - `(drive truck5 work work)` - no, precondition false
  - `(drive truck5 work store)` - no, precondition false
  - `(drive truck5 home store)` - precondition true, can be applied
  - `(drive truck5 home work)` - precondition true, can be applied

With quantification, we could have changed the precondition:

`(exists (?from - location) (and (at ?t ?from) (path-between ?from ?dest)))`

$\Longrightarrow$ No need for a new parameter - in this case

Modeling Tips (cont.)

- What about the *effects*?
  - Same natural parameters: the truck and the destination

```
(:action drive :parameters (?t - truck ?to - location)
   :precondition ..
   :effect ...
)
```

  - Natural effects:
    - The truck ends up at the destination: `(at ?t ?to)`
    - The truck *is no longer* where it started: `(not (at ?t ...???... work)`
  - How to you find out where the truck was before the action?
    - Using additional parameters still works: `(not (at ?t ?from))`
    - The value of `?from` is constrained in the preconditions - before
    - The value is used in the effect state - later

Alternative Representations

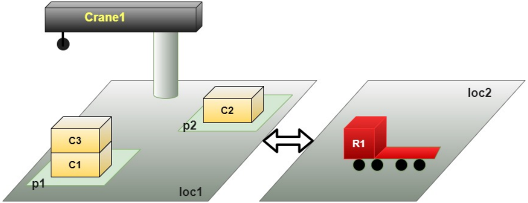
Three wide classes of logic-based representations (general classes, containing many languages)		
Propositional (Boolean propositions)	**First-Order** (Boolean predicates)	**State-variable-based** (non-Boolean functions)
atHome, atWork	at(truck, location)	loc(truck) = location
Language: PDDL :strips (if you avoid objects), ...	*Language:* PDDL :strips PDDL :adl, ADL, ...	

Read Chapter 2 of Ghallab et al. [6] for other perspectives on representations!

Classical and State-Var Representation

- Classical Planning with classical representation
  - A state defines the values of logical atoms (Boolean)
    - `adjacent(location, location)` - can you go directly from one location to another?
    - `loaded(robot, container)` - is the robot loaded with the given container?

Flexible (e.g. color example)



May be *wasteful*: A container can never be on many robots, which never happens!

Can be convenient, space efficient $\implies$ often used internally

Seems more powerful, but it is equivalent!

- Alternative: Classical with state-variable representation
  - A state defines the values of arbitrary state variables
    - `boolean adjacent(location, location)` ;; still Boolean
    - `container carriedby(robot)` ;; *which* container is on the robot?

Classical and State-Var Representation (cont.)

- Alternative: Classical with state-variable representation
  - A state defines the values of arbitrary state variables
    - `boolean adjacent(location, location)` ;; still Boolean
    - `container carriedby(robot)` ;; *which* container is on the robot?

No! What if a robot is not carrying a container?

- Must define a new type `container-or-none`!
  - Containing a new value '`none`'
  - `container-or-none carriedby(robot)`

Objects revisited

- Let us consider again the "drive" operator for a truck
  - Natural parameters: the truck and the destination

```
(:action drive
 (:parameters (?t - truck ?dest - location)
 (:precondition ...)
 (:effect ...) )
```

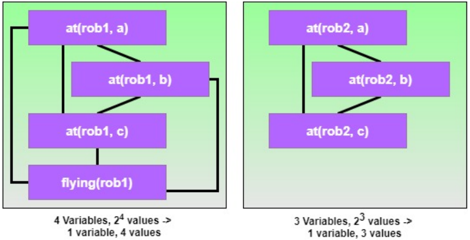
  - Natural precondition:
    - There must be a path between the current location and the destination
    - Should use the predicate `(path-between ?from ?to - location)`

  - State variable representation $\Longrightarrow$ can express the location of the truck
    `(:precondition (path-between (location-of ?t) ?to))`

  - No STS changes are required!

State Variables Internally

- Many planners convert to state variables internally
  - Basic idea:
    - Make a graph where each ground atom is a node



    - Find out (somehow!) that certain pairs of ground atoms cannot occur in the same state (mutually exclusive) - add edges
    - Each clique (all nodes connected in pairs) can become a new state variable

rob1loc	{atA, atB, atC, flying}
rob2loc	{atA, atB, atC}

Extended Example

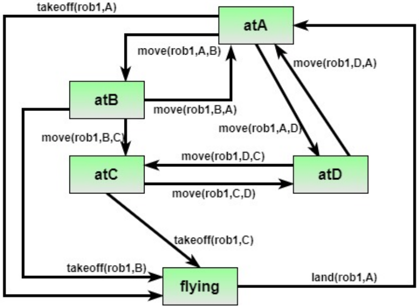
- Let us extend the previous example...

rob1loc	{atA, atB, atC, flying}
rob2loc	{atA, atB, atC}

- Assume there are only roads between some locations:
  - `move(rob1,a,b)` and `move(rob1,b,a)`
  - `move(rob1,b,c)` - but not `move(rob1,c,b)` *e.g. too steep in that direction*
  - `move(rob1,c,d)` and `move(rob1,d,c)`
  - `move(rob1,d,a)` and `move(rob1,a,d)`
- And you can take off anywhere, but only land at A
  - `takeoff(rob1, a),...,takeoff(rob1,d)`
  - `land(rob1,a)`

Domain Transition Graph

- With state variables domain transition graphs
  - For each state variable
    - Add a node for each value
    - Add an edge for each action changing the value



Useful form of *domain analysis* (as we will see later)

References I

[1] Stefan Edelkamp. Pddl2. 2: The language for the classical part of the 4th international planning competition. 01 2004. 37

[2] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5. URL `https://doi.org/10.1016/0004-3702(71)90010-5`. 34

[3] Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20: 61–124, 2003. doi: 10.1613/jair.1129. URL `https://doi.org/10.1613/jair.1129`. 37

[4] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.*, 27:235–297, 2006. doi: 10.1613/jair.2044. URL `https://doi.org/10.1613/jair.2044`. 37

[5] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL `https://doi.org/10.2200/S00513ED1V01Y201306AIM022`.

[6] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6. 5, 63

[7] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL `http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB`.

References II

[8] C. Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 219–240. William Kaufmann, 1969. URL `http://ijcai.org/Proceedings/69/Papers/023.pdf`. 4

[9] Malte Helmert. Changes in PDDL 3.1. Unpublished summary from the IPC-2008 website. `https://ipc08.icaps-conference.org/deterministic/index.html`, 2008. 37

[10] Drew V. McDermott. The 1998 AI planning systems competition. *AI Mag.*, 21(2):35–55, 2000. doi: 10.1609/aimag.v21i2.1506. URL `https://doi.org/10.1609/aimag.v21i2.1506`. 37

[11] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pages 256–264. UNESCO (Paris), 1959. 3