Schematic operators
Schematic operators

- Description of state variables and operators in terms of a given finite set of objects.
- Analogy: propositional logic vs. predicate logic
- Planners take input as schematic operators and translate them into (ground) operators. This is called grounding.
Schematic operators: example

Schematic operator \texttt{drive\_car\_from\_to}(x,y_1,y_2):

\[
\begin{align*}
  x & \in \{\text{car1, car2}\}, \\
  y_1 & \in \{\text{Freiburg, Strasbourg}\}, \\
  y_2 & \in \{\text{Freiburg, Strasbourg}\}
\end{align*}
\]

\[
\langle in(x, y_1), in(x, y_2) \land \neg in(x, y_1) \rangle
\]

corresponds to the operators

\[
\begin{align*}
  \langle in(\text{car1, Freiburg}), in(\text{car1, Strasbourg}) \land \neg in(\text{car1, Freiburg}) \rangle, \\
  \langle in(\text{car1, Strasbourg}), in(\text{car1, Freiburg}) \land \neg in(\text{car1, Strasbourg}) \rangle, \\
  \langle in(\text{car2, Freiburg}), in(\text{car2, Strasbourg}) \land \neg in(\text{car2, Freiburg}) \rangle, \\
  \langle in(\text{car2, Strasbourg}), in(\text{car2, Freiburg}) \land \neg in(\text{car2, Strasbourg}) \rangle,
\end{align*}
\]

plus four operators that are never applicable (inconsistent change set!) and can be ignored, like

\[
\langle in(\text{car1, Freiburg}), in(\text{car1, Freiburg}) \land \neg in(\text{car1, Freiburg}) \rangle.
\]
Schematic operators: quantification

Existential quantification (for formulae only)

Finite disjunctions $\phi(a_1) \lor \cdots \lor \phi(a_n)$ represented as
$$\exists x \in \{a_1, \ldots, a_n\} : \phi(x).$$

Universal quantification (for formulae and effects)

Finite conjunctions $\phi(a_1) \land \cdots \land \phi(a_n)$ represented as
$$\forall x \in \{a_1, \ldots, a_n\} : \phi(x).$$

Example

$$\exists x \in \{A, B, C\} : in(x, \text{Freiburg})$$ is a short-hand for
$$in(A, \text{Freiburg}) \lor in(B, \text{Freiburg}) \lor in(C, \text{Freiburg}).$$
used by almost all implemented systems for deterministic planning

supports a language comparable to what we have defined above (including schematic operators and quantification)

syntax inspired by the Lisp programming language: e.g. prefix notation for formulae

\[(\text{and} \ (\text{or} \ (\text{on} \ A \ B) \ (\text{on} \ A \ C)) \n\ (\text{or} \ (\text{on} \ B \ A) \ (\text{on} \ B \ C)) \n\ (\text{or} \ (\text{on} \ C \ A) \ (\text{on} \ A \ B)))\]
A domain file consists of

- `(define (domain DOMAINNAME)`
- a :requirements definition (use :strips :typing by default)
- definitions of types (each parameter has a type)
- definitions of predicates
- definitions of operators
Note: Unlike in the previous chapter, here we use a variant of the blocks world domain with an explicitly modeled gripper/hand.

(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (onTable ?x - block)
               (clear ?x - block)
               (handEmpty)
               (holding ?x - block)
  )
PDDL: operator definition

- (:action OPERATORNAME
- list of parameters: (?x - type1 ?y - type2 ?z - type3)
- precondition: a formula

  <schematic-state-var>
  (and <formula> ... <formula>)
  (or <formula> ... <formula>)
  (not <formula>)
  (forall (?x1 - type1 ... ?xn - typen) <formula>)
  (exists (?x1 - type1 ... ?xn - typen) <formula>)
**effect:**

\[ <\text{schematic-state-var}> \]
\[ (\text{not } <\text{schematic-state-var}>) \]
\[ (\text{and } <\text{effect}> \ldots <\text{effect}> ) \]
\[ (\text{when } <\text{formula}> <\text{effect}> ) \]
\[ (\text{forall } (?x_1 - \text{type1} \ldots ?x_n - \text{typen}) <\text{effect}> ) \]
(:action stack
  :parameters (?x - block ?y - block)
  :precondition (and (holding ?x) (clear ?y))
  :effect (and (not (holding ?x))
             (not (clear ?y))
             (clear ?x)
             (handempty)
             (on ?x ?y)))
A problem file consists of

- (define (problem PROBLEMNAME))
- declaration of which domain is needed for this problem
- definitions of objects belonging to each type
- definition of the initial state (list of state variables initially true)
- definition of goal states (a formula like operator precondition)
(define (problem example)
  (:domain BLOCKS)
  (:objects a b c d - block)
  (:init (clear a) (clear b) (clear c) (clear d)
           (ontable a) (ontable b) (ontable c)
           (ontable d) (handempty))
  (:goal (and (on d c) (on c b) (on b a)))
)
Fast Downward is the state-of-the-art planner, usable both for research and applications.

Main developers:
- Malte Helmert
- Gabi Röger
- Erez Karpas
- Jendrik Seipp
- Silvan Sievers
- Florian Pommerenening
Fast Downward is available at
http://www.fast-downward.org/

**Installation:**
Follow instructions at
http://www.fast-downward.org/
ObtainingAndRunningFastDownward

**Running:**
Follow instructions at
http://www.fast-downward.org/PlannerUsage
Example run of Fast Downward

# ./fast-downward.py --plan-file plan.txt \  
domain.pddl problem.pddl --search "astar(blind())"

[...]  
INFO Running search.  
[...]  
Solution found!  
[...]  
Plan length: 6 step(s).  
[...]  
Expanded 85 state(s).  
[...]  
Search time: 0s  
[...]
# Example plan found by Fast Downward

```plaintext
# cat plan.txt
(pick-up b)
(stack b a)
(pick-up c)
(stack c b)
(pick-up d)
(stack d c)
; cost = 6 (unit cost)
```
In case you are looking for a decent PDDL editor:

- Check out the PDDL editor in the cloud:
  http://editor.planning.domains/
- The website also includes a built-in solver:
  http://solver.planning.domains/
- ...and an API + domain repository:
  http://api.planning.domains/
Example: blocks world in PDDL

(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
    (ontable ?x - block)
    (clear ?x - block)
    (handempty)
    (holding ?x - block)
  )
(:action pick-up
  :parameters (?x - block)
  :precondition (and (clear ?x) (ontable ?x)
                   (handempty))
  :effect (and (not (ontable ?x))
               (not (clear ?x))
               (not (handempty))
               (holding ?x)))
(:action put-down
  :parameters (?x - block)
  :precondition (holding ?x)
  :effect (and (not (holding ?x))
    (clear ?x)
    (handempty)
    (ontable ?x)))
(:action stack
  :parameters (?x - block ?y - block)
  :precondition (and (holding ?x) (clear ?y))
  :effect (and (not (holding ?x))
              (not (clear ?y))
              (clear ?x)
              (handempty)
              (on ?x ?y)))
(:action unstack
  :parameters (?x - block ?y - block)
  :precondition (and (on ?x ?y) (clear ?x) (handempty))
  :effect (and (holding ?x)
               (clear ?y)
               (not (clear ?x))
               (not (handempty))
               (not (on ?x ?y))))
(define (problem example)
  (:domain BLOCKS)
  (:objects a b c d - block)
  (:init (clear a) (clear b) (clear c) (clear d)
          (ontable a) (ontable b) (ontable c)
          (ontable d) (handempty))
  (:goal (and (on d c) (on c b) (on b a)))
)