Planning under Uncertainty with Action Languages

A Logic Programming Approach

Axel Polleres, University of Innsbruck

axel.polleres@uibk.ac.at

joint work with Thomas Eiter, Wolfgang Faber, Gerald Pfeifer, TU Vienna

Nicola Leone, University of Calabria

universität innsbruck

TU WIEN TECHNISCHE UNIVERSITÄT WIEN VIENNA UNIVERSITY OF TECHNOLOGY

Overview

- Introduction
- Answer Set Programming in a nutshell
- Planning in Logic Programming (an adhoc solution).
- Planning & Knowledge Representation in Action Language $\mathcal{K}$,
  - Syntax & Semantics
  - Knowledge State vs. World State Encodings
- (short) Translations to LP, the $\texttt{DLV}^{\mathcal{K}}$ Planning System

Introduction

What is Planning?

- Start: initial situation (or state)
- Desired: reach a goal
- At disposal: actions

Problem: Find a suitable sequence of actions (a plan) whose execution brings about the goal.

Planning and AI

Planning is a challenging problem for AI since 1950's

McCarthy: Missionaries and Cannibals (1959)

- Logic-based approaches
- Heuristic search methods
- ad hoc approaches
- Graphplan
- Planning as Model-Checking
- ...
- Planning as Satisfiability (SAT Planning)
- Answer Set Planning (Planning using Answer Set Programming)

Answer Set Programming in a nutshell

Classical logic Programming extended with

- Disjunction
- Default negation
- Strong (classical) negation
- Integrity constraints
- No function symbols (finiteness)

Answer Set Programming: Idea

- Fundamental concept:
  Models, not proofs, **represent solutions**!
- Need techniques to **compute models** (not to compute proofs)

What is this good for?
Solve *search problems*

Compute, e.g., one/all solutions of the N-queens problem,

SAT, one/all routes to reach the airport; . . . compute plans!

ASP: Syntax

Rules:

$\mathtt{a_1} \vee \ldots \vee \mathtt{a_n}$ `:-` $\mathtt{b_1}, \ldots, \mathtt{b_k}$, `not` $\mathtt{b_{k+1}}, \ldots,$ `not` $\mathtt{b_l}$.

Constraints: `:-` $\mathtt{b_1}, \ldots, \mathtt{b_k}$, `not` $\mathtt{b_{k+1}}, \ldots, \mathtt{b_l}$.

- as and bs are atoms (p) or strongly negated atoms ($-p$)
- variables are allowed in arguments of atoms
- a program is a set of rules and constraints
- order of literals and rules does not matter

```
interested(X) v curious(X) :- attendsTalk(X).
attendsTalk(X) :- staff(X), not onVacation(X).
```

Answer Set Semantics 1/2

Let M be a (consistent) set of literals

A Rule

$\mathtt{a_1} \vee \ldots \vee \mathtt{a_n}$ `:-` $\mathtt{b_1}, \ldots, \mathtt{b_k}$, `not` $\mathtt{b_{k+1}}, \ldots,$ `not` $\mathtt{b_l}$.

is called satisfied wrt. M iff:

If $\mathtt{b_1}, \ldots, \mathtt{b_k} \in M$ and $\mathtt{b_{k+1}}, \ldots, \mathtt{b_l} \notin M$ then

at least one of $\mathtt{a_1}, \ldots, \mathtt{a_n} \in M$.

Answer Set Semantics 2/2

- P — logic program
- M — (consistent) set of literals
- *Reduct* P^M (Gelfond, Lifschitz)
  - for each $l \in M$ remove rules with $\texttt{not} \;\; l$ in the body
  - remove literals $\texttt{not} \;\; l$ from all other rules
- M is called answer set iff it is a minimal set such that all rules of P^M are satisfied wrt. M

Example 1 – Positive Program

```
interested(you) v curious(you) :- attendsTalk(you).
attendsTalk(you).
```

`M1 = {attendsTalk(you), curious(you)}` (Answer Set)

`M2 = {attendsTalk(you), interested(you)}` (Answer Set)

`M3 = {attendsTalk(you)}` (first rule not satisfied)

`M4 = {attendsTalk(you), interested(you), curious(you)}` (not minimal)

Example 2 – Constraints

Constraints "prohibit" Answer Sets:

```
interested(you) v curious(you) :- attendsTalk(you).
attendsTalk(you).
:- bored(you), interested(you).
bored(you).
```

Only one answer set:

```
M = {attendsTalk(you), curious(you), bored(you)}
```

Example 3 – Default Negation

```
interested(you) :- not sleepy(you).
```

```
M = {interested(you)}
```

Example 4 – Default Negation

```
interested(you) :- not sleepy(you).
sleepy(you).
M = {sleepy(you)}
```

Nonmonotonic Reasoning!

Example 5 – Default Negation

```
interested(you) :- not sleepy(you).
sleepy(you) :- not interested(you).
```

```
M1 = {sleepy(you)}
```

```
M2 = {interested(you)}
```

Example 6 – Strong Negation

```
interested(you) :- not -interested(you).
-interested(you) :- not interested(you).
M1 = {-interested(you)}
M2 = {interested(you)}
```

Literals can be *true, false or unknown* in an answer set if the literal appears in positive and negative form!

Answer Set Planning

First attempt: adhoc encoding

Idea: Use expressiveness of Answer Set Semantics for guessing plans.

- NP (Σ_2^P=$\mathrm{NP}^{\mathrm{NP}}$) for normal (disjunctive) logic programs

Method: (by e.g. Subrahmanian & Zaniolo, Dimopoulos et al., Lifschitz . . . ...)

- Formulate planning problem as a logic program Π (describe trajectories)
- Compute any answer sets (i.e. models) of Π, which encode possible plans.

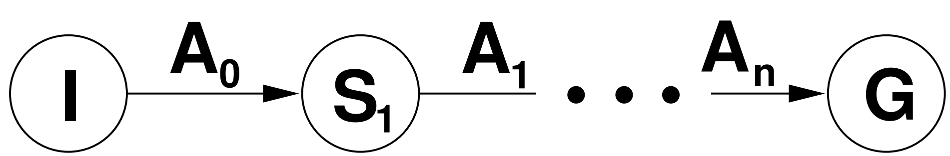
Advantage: Declarative problem solving

High-level View

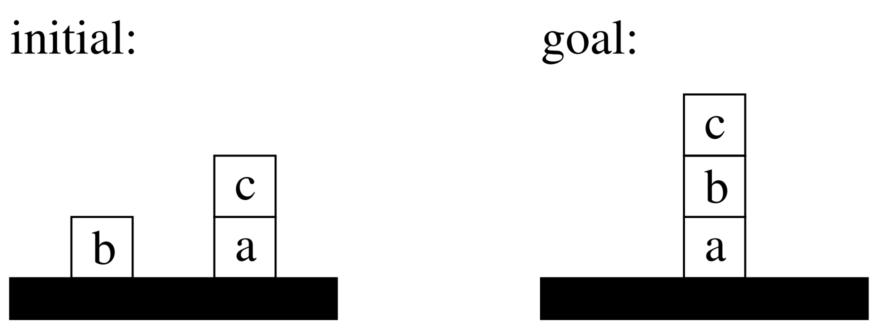
Input: Fluents (state variables) F
Actions (that usually modify fluents) A
Initial state I, goal state G
State constraints, action descriptions

Evolution: Discrete steps of time (stages), 0,1,. . . ,n

Output: A sequence of *action sets* (or single actions)
$\langle A_0, A_1, \ldots, A_n \rangle$ which transforms I into G



Example: Blocks World



Actions: Move a block from one location to another (`move(b,a), ...`)

Fluents: Predicates describing the state (`clear(b,0), on(c,a,0), ...`)

Example: Blocks World

Background Knowledge:

```
block(a).  block(b).  block(c).
location(X):- block(X).  location(table).
```

Use discrete time 0,1,2,. . .   Here: 3 Steps

```
act_time(0). act_time(1). act_time(2).          % 3 time stamps
```

Example: Blocks World

Background Knowledge:

```
block(a).  block(b).  block(c).
location(X):- block(X).  location(table).
```

Use discrete time 0,1,2,. . .   Here: 3 Steps

```
act_time(0). act_time(1). act_time(2).                    % 3 time stamps
% Initial state:
on(a, table, 0).  on(b, table, 0).  on(c, a, 0).
% Goal:
goal :- on(a, table, 3), on(b, a, 3), on(c, b, 3).
:- not goal.
```

Example: Blocks World

Background Knowledge:

```
block(a).  block(b).  block(c).
location(X) :- block(X).  location(table).
```

Use discrete time 0,1,2,...   Here: 3 Steps

```
act_time(0). act_time(1). act_time(2).                    % 3 time stamps
% Initial state:
on(a, table, 0).  on(b, table, 0).  on(c, a, 0).
% Goal:
goal :- on(a, table, 3), on(b, a, 3), on(c, b, 3).
:- not goal.
```

% The meat of the program ...

%Guess: At any action time T, move a block B to some location L or not

```
move(B, L, T) v -move(B, L, T) :- block(B), location(L),
                                  act_time(T), B != L.
```

% Effects of moving a block: The block is now at the new location...

$\mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T1})\,\mathtt{:-}\ \mathtt{move}(\mathtt{B},\mathtt{L},\mathtt{T}),\ \mathtt{T1} = \mathtt{T} + 1.$

% ... and it is no longer at the old location.

$-\mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T1})\,\mathtt{:-}\ \mathtt{move}(\mathtt{B},\mathtt{L1},\mathtt{T}),\mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T}),\ \mathtt{T1} = \mathtt{T} + 1,$
$\mathtt{L1} <> \mathtt{L}.$

% Inertia: Unless a block is *known* to be moved, assume it is still at old place.

$\mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T1})\,\mathtt{:-}\ \mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T}),\mathtt{not} - \mathtt{on}(\mathtt{B},\mathtt{L},\mathtt{T1}),\ \mathtt{T1} = \mathtt{T} + 1.$

% Constraints

$\mathtt{:-}\ \mathtt{move}(\mathtt{B},\mathtt{L},\mathtt{T}),\mathtt{on}(\_,\mathtt{B},\mathtt{T}).$ % A block can only be moved if it's clear.

$\mathtt{:-}\ \mathtt{move}(\mathtt{B},\mathtt{B1},\mathtt{T}),\mathtt{on}(\_,\mathtt{B1},\mathtt{T}),\mathtt{block}(\mathtt{B1}).$ % No move onto an occupied block

$\mathtt{:-}\ \mathtt{move}(\mathtt{B},\_,\mathtt{T}),\mathtt{move}(\mathtt{B1},\_,\mathtt{T}),\mathtt{B} \mathrel{!=} \mathtt{B1}.$ % Move only one block at each step

$\mathtt{:-}\ \mathtt{move}(\_,\mathtt{L},\mathtt{T}),\mathtt{move}(\_,\mathtt{L1},\mathtt{T}),\mathtt{L} \mathrel{!=} \mathtt{L1}.$ % No move to different locations

Disadvantages of the Method

- Ad hoc encoding
- Semantics is implicit in the encoding
- Inflexible (changes, etc)
- …

Disadvantages of the Method

- Ad hoc encoding
- Semantics is implicit in the encoding
- Inflexible (changes, etc)
- ...

Better: Provide genuine action / planning language.

- Syntax and first class citizen semantics
- Compile to logic engines (e.g., to Answer Set solvers like `DLV`, smodels)

Planning & Knowledge Representation in Action Language $\mathcal{K}$

Planning Languages

Early attempts

- Deductive Planning (Green, 1969)
- Situation Calculus (McCarthy & Hayes, 1969)
- STRIPS (Fikes et al., 1971) + descendants (e.g. PDDL)
- Temporal Logic (McDermott, 1982)
- Event Calculus (Kowalski & Sergot, 1986; Eshghi 1988)
- Fluent Calculus (Thielscher, 2000)
- ...
- Action Languages, e.g. $\mathcal{A}$, $\mathcal{AR}$, $\mathcal{A}_K$, $\mathcal{C}$, $\mathcal{K}$,...

Action Language $\mathcal{K}$

A relative of action languages $\mathcal{A}$ (Gelfond & Lifschitz, 1993) and $\mathcal{C}$ (Giunchiglia & Lifschitz, 1998)

Divide predicates in

- state predicates, further divided in
  - rigid predicates (constants)
  - fluent predicates (variables)
- action predicates (variables)

Formulate axioms about transitions rather than operators like in “classical” planning languages.

Main Features of $\mathcal{K}$

- Incomplete states (“knowledge states”)
- Default (nonmonotonic) negation and strong (classical) negation
- Typed fluents and actions
- Initial state constraints
- Conditional executability
- Causation rules
- Inertia
- Nondeterministic action effects

$\mathcal{K}$ Planning Domains and Problems

Background Knowledge Π: A logic program Π with a single model,
(answer set) defining type information and static knowledge.

$\mathcal{K}$ Action Description AD:

fluents: D_F	% fluent defs
actions: D_A	% action type defs
always: C_R	% causation rules + exec. cond's
initially: C_I	% initial state constraints

$\mathcal{K}$ Planning Domain: $\langle \Pi, AD \rangle$

$\mathcal{K}$ Planning Problem: additional goal

goal: $G?(i)$ ground literal(s) G; plan length $i \geq 0$.

Blocksworld – $\mathcal{K}$ Representation

Background knowledge

$$\Pi = \{\texttt{block}(\texttt{a}).\ \texttt{block}(\texttt{b}).\ \texttt{block}(\texttt{c}).$$
$$\texttt{location}(\texttt{table}).$$
$$\texttt{location}(\texttt{L})\ \texttt{:-}\ \texttt{block}(\texttt{L}).\}$$

Note: Syntactic and/or other restrictions might help ensure that Π has a single model (answer set).

Action Description

AD:

```
fluents :    on(B, L) requires block(B), location(L).
             occupied(B) requires location(B).
actions :    move(B, L) requires block(B), location(L).
always :     executable move(B, L) if not occupied(B),
                                     not occupied(L), B <> L.
             inertial on(B, L).
             caused on(B, L) after move(B, L).
             caused −on(B, L1) after move(B, L), on(B, L1), L <> L1.
             caused occupied(B) if on(B1, B), block(B).
             noConcurrency.
initially :  on(a, table). on(b, table). on(c, a).
```

AD – Fluent and Action Type Defs

```
fluents:   on(B, L) requires block(B), location(L).
           occupied(B) requires location(B).
actions:   move(B, L) requires block(B), location(L).
always:    executable move(B, L) if not occupied(B),
                                    not occupied(L), B <> L.
           inertial on(B, L).
           caused on(B, L) after move(B, L).
           caused -on(B, L1) after move(B, L), on(B, L1), L <> L1.
           caused occupied(B) if on(B1, B), block(B).
           noConcurrency.
initially: on(a, table). on(b, table). on(c, a).
```

AD – Action Executability

```
fluents:   on(B,L) requires block(B), location(L).
           occupied(B) requires location(B).
actions:   move(B,L) requires block(B), location(L).
always:    executable move(B,L) if not occupied(B),
                                   not occupied(L), B <> L.
           inertial on(B,L).
           caused on(B,L) after move(B,L).
           caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
           caused occupied(B) if on(B1,B), block(B).
           noConcurrency.
initially: on(a,table). on(b,table). on(c,a).
```

$\mathcal{AD}$ – Transition Rules (Causality)

```
fluents :    on(B, L) requires block(B), location(L).
             occupied(B) requires location(B).
actions :    move(B, L) requires block(B), location(L).
always :     executable move(B, L) if not occupied(B),
                                      not occupied(L), B <> L.
             inertial on(B, L).
             caused on(B, L) after move(B, L).
             caused -on(B, L1) after move(B, L), on(B, L1), L <> L1.
             caused occupied(B) if on(B1, B), block(B).
             noConcurrency.
initially :  on(a, table). on(b, table). on(c, a).
```

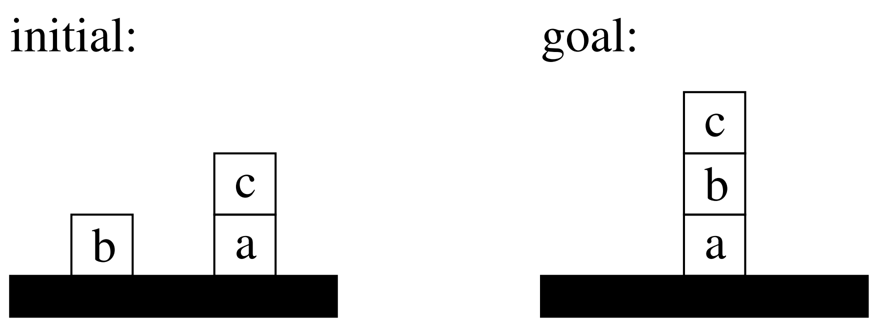
Remark: Causation Rules describe valid transitions rather than operator descriptions in languages like STRIPS or PDDL!

AD – Initial State Constraints

```
fluents :   on(B,L) requires block(B), location(L).
            occupied(B) requires location(B).
actions :   move(B,L) requires block(B), location(L).
always :    executable move(B,L) if not occupied(B),
                                    not occupied(L), B <> L.
            inertial on(B,L).
            caused on(B,L) after move(B,L).
            caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
            caused occupied(B) if on(B1,B), block(B).
            noConcurrency.
initially : on(a,table). on(b,table). on(c,a).
```

Goal

$\texttt{goal}: \ \texttt{on}(\texttt{c}, \texttt{b}), \ \texttt{on}(\texttt{b}, \texttt{a}), \ \texttt{on}(\texttt{a}, \texttt{table})?(3)$



Intuitively: Feasible plan is

$\texttt{move}(\texttt{c}, \texttt{table}); \ \texttt{move}(\texttt{b}, \texttt{a}); \ \texttt{move}(\texttt{c}, \texttt{b})$

Features 1/3:

- Qualification Problem:

  Overriding by exceptions to executability

  $\texttt{executable act if } < \texttt{cond1} >$

  $\texttt{nonexecutable act if } < \texttt{cond2} >$

  Example:

  $\texttt{executable move}(\mathrm{B}, \mathrm{L}) \texttt{ if } \mathrm{B} != \mathrm{L}.$

  $\texttt{nonexecutable move}(\mathrm{B}, \mathrm{L}) \texttt{ if occupied}(\mathrm{B}).$

  $\texttt{nonexecutable move}(\mathrm{B}, \mathrm{L}) \texttt{ if occupied}(\mathrm{L}).$

Features 2/3

- Frame Problem/Inertia:

  `inertial on(B, L).`

  short for

  `caused on(B, L) if not -on(B, L) after on(B, L).`

- Uncertainty: in general by unstratified negation.

  `total loaded(gun).`

  short for

  `caused loaded(gun) if not -loaded(gun).`

  `caused -loaded(gun) if not loaded(gun).`

Features 3/3

- Ramifications/State Axioms:

  Simple by causal rules:

  $\texttt{caused supported}(\texttt{B1})\ \texttt{if on}(\texttt{B1}, \texttt{table}).$

  $\texttt{caused supported}(\texttt{B1})\ \texttt{if on}(\texttt{B1}, \texttt{B2}),\ \texttt{supported}(\texttt{B2}).$

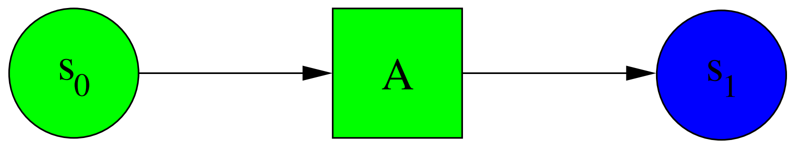
  Note:

  - transitive closure naturally expressed (LP-flavored semantics of $\mathcal{K}$)

Semantics of $\mathcal{K}$ – Principles

transition-based semantics

caused fl if Cond1 after Cond2



A . . . set of actions (executable)

- Cond2 is evaluated in s_0, might include actions.
- fl and Cond1 are evaluated in s_1

Define new state s_1 by a non-monotonic logic program of rules!

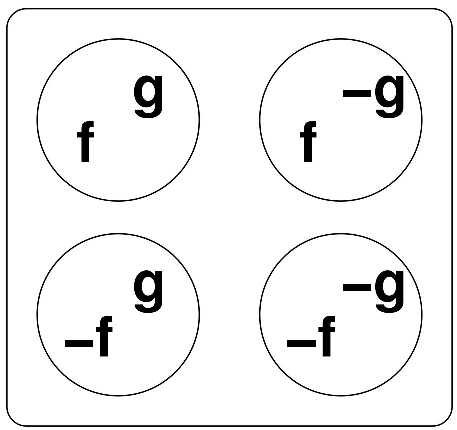
fl :- Cond1

Remark: several answer sets ⇔ several possible succ. states

Incomplete states

Usual assumption for action language: total states

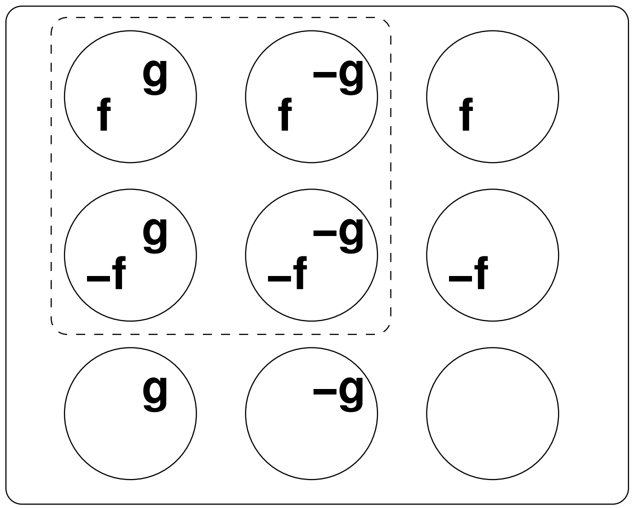
State s is total $\Leftrightarrow$ For each fluent f, either f or $\neg f$ must be in s.



set of total states for the two atoms f and g

Total states correspond to total (w.r.t. strong negation!) interpretations

Incomplete state: values of some fluents are unknown.



consistent partial interpretations for the two atoms f and g

Handle incompleteness using principles from nonmonotonic Logic Programming

Note: Differs from Kripke-style $\mathcal{A}_K$ (Baral & Son, 2001)

Plans

Trajectories:

$$T = \langle s_0, A_1, s_1, \ldots, s_{n-1}, A_n, s_n \rangle, \qquad n \geq 0$$

- s_0 is initial state
- each s_{i+1} is reached by "legal" transition s_i, A_{i+1}, s_{i+1}

"Optimistic" Plan:

- project T where s_n satisfies the goal to $A_1, A_2, \ldots, A_n$

Example: $P = \mathtt{move}(\mathtt{c}, \mathtt{table});\ \mathtt{move}(\mathtt{b}, \mathtt{a});\ \mathtt{move}(\mathtt{c}, \mathtt{b})$

Planning under uncertainty in $\mathcal{K}$

- Incomplete states
- Use of default principles: Express “Unknown”

  $\texttt{executable check\_door if not open, not } -\texttt{open.}$

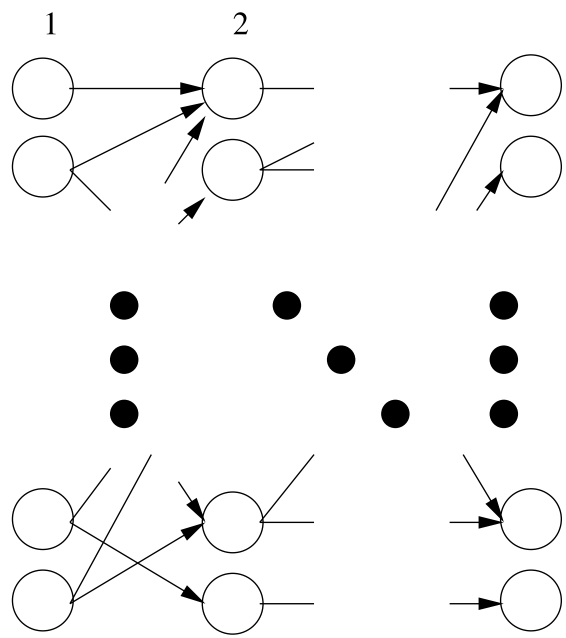
  $\texttt{forbidden not on(B, L), not } -\texttt{on(B, L).}$

- “Totalize” fluents (case distinction)

  $\texttt{total on(B, L).}$

- “secure” plans (= conformant plans)
  Always reach the goal by the plan, no matter what happens
- “optimal” plans
  action cost declarations, minimize overall cost (not covered here)

Special Plans – Secure Plans



Trajectories: spanning paths, labels form plans

- Optimistic Plans: at least one trajectory to goal
- Secure Plans: *each* trajectory from *each* initial state to goal
  no “dead-end” trajectories

Example: Bomb in the Toilet

$\Pi = \{\texttt{package}(1).\quad \texttt{package}(2).\quad \ldots \quad \texttt{package}(p).\}$

```
fluents :    armed(P) requires package(P).
             unsafe.
actions :    dunk(P) requires package(P).
always :     executable dunk(P).
             inertial armed(P).
             caused -armed(P) after dunk(P).
             caused unsafe if armed(P).
initially :  total armed(P).                 % 3 lines: one package is armed
             forbidden armed(P), armed(P1), P <> P1.
             forbidden not unsafe.
goal :       not unsafe ? (p)
```

Encodes ALL possible world states!

Example: Bomb in the Toilet

$\Pi = \{$package(1). package(2). ... package(p).$\}$

```
fluents :    armed(P) requires package(P).
             unsafe.
actions :    dunk(P) requires package(P).
always :     executable dunk(P).
             inertial armed(P).
             caused -armed(P) after dunk(P).
             caused unsafe if armed(P).
initially :  total armed(P).                 % 3 lines: one package is armed
             forbidden armed(P), armed(P1), P <> P1.
             forbidden not unsafe.
goal :       not unsafe ? (p)
```

Encodes ALL possible world states!

Variant: At least one package is armed. What changes?

Avoiding Totalization

Change view: "unsafe = known that one package is armed"

$\Rightarrow$ "unsafe = not known that all packages are unarmed".

```
fluents:    armed(P) requires package(P).
            unsafe.
actions:    dunk(P) requires package(P).
always:     executable dunk(P).
            inertial -armed(P).
            caused -armed(P) after dunk(P).
            caused unsafe if not -armed(P).
initially:    % tabula rasa, nothing is known
goal:       not unsafe ? (p)
```

Encodes only what is KNOWN! - Knowledge States

Single initial state, only det. actions $\Rightarrow$ need no security check

Action Language $\mathcal{K}$- “Forgetting”

Similar to “unknown” fluents in the initial state, we can use knowledge state encodings to “forget” about certain facts by overriding intertia. Example: non-deterministic “clogging” in Bomb in the toilet.

```
total clogged(T) after flush(T)
inertial -clogged(T).
```

“The toilet might be clogged or unclogged after being flushed”. “It stays unclogged normally.”

Alternative:

```
inertial -clogged(T) after not flush(T).
```

“The toilet stays unclogged **unless** it has been flushed”

Action Language $\mathcal{K}$- Another example

Avoiding totalization is not always possible: example SQUARE [Bonet-Geffner, 2000]:

```
fluents:  atX(P) requires index(P). atY(P) requires index(P). anywhere.
actions:  up. down. left. right.
always:   executable up.   executable right. executable left. executable d
          nonexecutable up if down. nonexecutable left if right.
          inertial atX(X). inertial atY(Y).

          caused atX(X) after atX(X1), next(X,X1), left.
          caused atX(X1) after atX(X), next(X,X1), right.
          caused -atX(X) if atX(X1), X1 != X after atX(X).

          [...]
initially:  total atX(X). total atY(Y).
            forbidden atX(X), atX(X1), X != X1.
            forbidden atY(Y), atY(Y1), Y != Y1.
            caused anywhere if atX(X), atY(Y).
            forbidden not anywhere.

goal: atX(0),atY(0)? ($n$)
```

We can not make use of “unknown” in case of conditional effects.

Translations to Disjunctive Datalog - The $\texttt{DLV}^{\mathcal{K}}$ Planning System

Translation

```
caused on(B,L) after move(B,L).
```

$\Rightarrow$ $\mathtt{on(B,L,T_1)\ {:}{-}\ move(B,L,T),\ T1 = T+1.}$

```
inertial on(B,L).
(= caused on(B,L) if not -on(B,L) after on(B,L).)
```

$\Rightarrow$ $\mathtt{on(B,L,T_1)\ {:}{-}\ on(B,L,T),\ not\ -on(B,L,T_1),\ T1 = T+1.}$

```
initially caused on(a,table).
```

$\Rightarrow$ $\mathtt{on(a,table,0).}$

```
executable move(B,L) if B != L.
```

$\Rightarrow$ $\mathtt{move(B,L,T) \lor -move(B,L,T)\ {:}{-}\ B\ {!}{=}\ L, block(B), location(L),}$
$\mathtt{actiontime(T).}$

```
goal: on(c,b), on(b,a), on(a,table)?  (3)
```

$\Rightarrow$ $\mathtt{goal\ {:}{-}\ on(a,table,3), on(b,a,3), on(c,b,3).}$
$:-\ \mathit{not\ goal}.$
$\mathtt{actiontime(0).actiontime(1).actiontime(2).}$

Projection of answer sets to positive action literals yields optimistic plans.

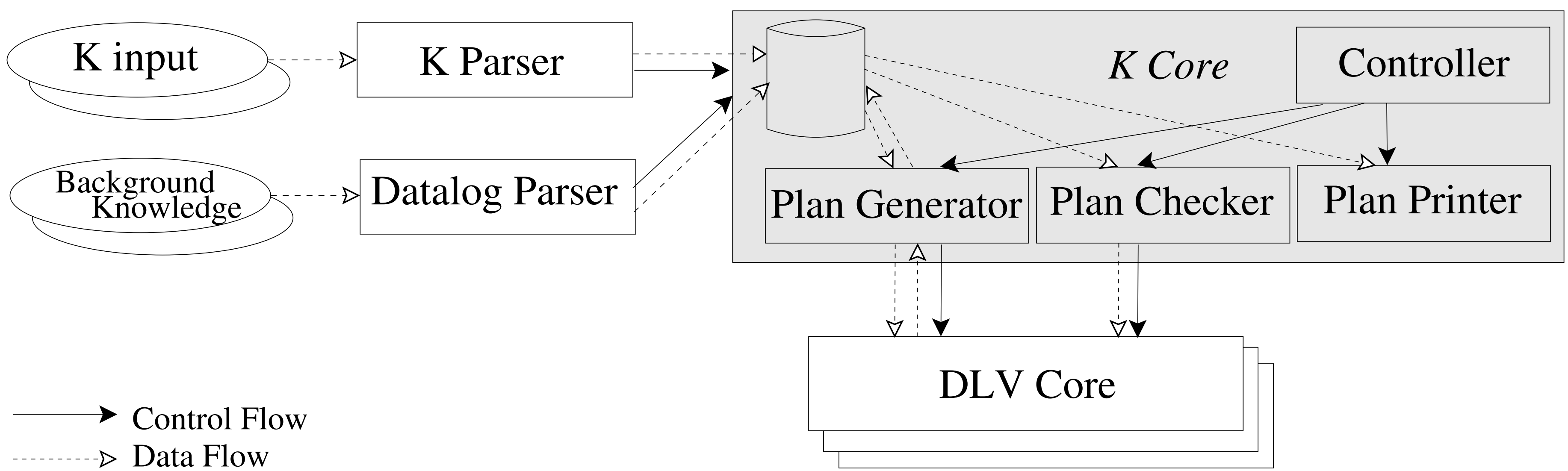
Actions/Fluents are time-stamped.

Translation - Secure check

- Secure planning: Check plan by rewriting this program wrt. plan.
- Hard-code an optimistic plan p inside the translated program and try to find an answer set where the goal does not hold $\Pi_{check}(p)$, or an action of p is not executable.
- Algorithm for secure planning:
  - compute optimistic plans (i.e. answer sets)
  - Create Π_{check} for optimistic plan) found, if has no answer set, the plan is secure (i.e. conformant).
  - use caching.
- Integrated encodings: cf. [Eiter, Polleres, 2004], [Polleres, 2003]

DLV$^{\mathcal{K}}$ System

Architecture:



Some - old - benchmarks

BMTC(p, t)	steps	DLV^K us	DLV^K ks	CPLAN	SGP
BMTC(2, 2)	1	0.02s	0.01s	1.41s	0.95s
BMTC(3, 2)	3	0.04s	0.02s	1.50s	3.40s
BMTC(4, 2)	3	0.11s	0.03s	1.72s	7.17s
BMTC(5, 2)	5	2.79s	0.04s	3.37s	-
BMTC(6, 2)	5	37.04s	0.07s	13.04s	-
BMTC(7, 2)	7	-	0.52s	71.50s	-
BMTC(8, 2)	7	-	10.66s	-	-
BMTC(9, 2)	9	-	206.27s	-	-
BMTC(10, 2)	9	-	-	-	-
BMTC(2, 3)	1	0.02s	0.02s	1.62s	1.15s
BMTC(3, 3)	1	0.02s	0.02s	2.31s	1.76s
BMTC(4, 3)	3	0.08s	0.03s	4.81s	15.01s
BMTC(5, 3)	3	0.35s	0.03s	13.55s	76.28s
BMTC(6, 3)	3	17.81s	0.06s	43.34s	592.41s
BMTC(7, 3)	5	223.31s	0.13s	210.71s	-
BMTC(8, 3)	5	-	0.74s	417.62s	-
BMTC(9, 3)	5	-	5.90s	-	-
BMTC(10, 3)	7	-	389.08s	-	-
BMTC(2, 4)	1	0.02s	0.02s	2.89s	1.52s
BMTC(3, 4)	1	0.02s	0.02s	9.19s	2.34s
BMTC(4, 4)	1	0.03s	0.02s	37.55s	3.71s
BMTC(5, 4)	3	0.18s	0.04s	158.74s	372.74s
BMTC(6, 4)	3	5.29s	0.05s	571.77s	-
BMTC(7, 4)	3	61.73s	0.09s	-	-
BMTC(8, 4)	3	668.74s	0.41s	-	-
BMTC(9, 4)	5	-	1.06s	-	-
BMTC(10, 4)	5	-	12.14s	-	-

able 8.6: Experimental results for BMTC(p), conc. dunks

BMTC(p, t)	steps	DLV^K us	DLV^K ks	CPLAN	CMBP	GPT
BMTC(2, 2)	2	0.02s	0.02s	1.41s	0.04s	0.76s
BMTC(3, 2)	4	0.07s	0.02s	1.50s	0.05s	0.78s
BMTC(4, 2)	6	2.47s	0.04s	1.64s	0.06s	0.81s
BMTC(5, 2)	8	208.52s	0.05s	2.66s	0.06s	0.82s
BMTC(6, 2)	10	-	0.07s	32.77s	0.09s	0.86s
BMTC(7, 2)	12	-	0.10s	12.46s	0.12s	0.96s
BMTC(8, 2)	14	-	0.13s	-	0.23s	1.11s
BMTC(9, 2)	16	-	0.20s	-	0.48s	1.48s
BMTC(10, 2)	18	-	0.28s	-	0.96s	2.26s
BMTC(2, 3)	2	0.02s	0.02s	1.50s	0.04s	0.76s
BMTC(3, 3)	3	0.03s	0.02s	1.85s	0.04s	0.81s
BMTC(4, 3)	5	1.84s	0.03s	2.86s	0.06s	0.84s
BMTC(5, 3)	7	291.24s	0.06s	5.92s	0.09s	0.90s
BMTC(6, 3)	9	-	0.09s	14.50s	0.14s	0.99s
BMTC(7, 3)	11	-	0.25s	40.41s	0.30s	1.17s
BMTC(8, 3)	13	-	15.42s	-	0.62s	1.66s
BMTC(9, 3)	15	-	-	-	1.44s	2.79s
BMTC(10, 3)	17	-	-	-	3.31s	5.64s
BMTC(2, 4)	2	0.02s	0.02s	2.02s	0.04s	0.81s
BMTC(3, 4)	3	0.41s	0.02s	3.67s	0.05s	0.83s
BMTC(4, 4)	4	0.60s	0.03s	9.03s	0.07s	0.92s
BMTC(5, 4)	6	149.65s	0.06s	30.55s	0.13s	1.01s
BMTC(6, 4)	8	-	0.10s	-	0.23s	1.27s
BMTC(7, 4)	10	-	0.15s	199.73s	0.51s	1.85s
BMTC(8, 4)	12	-	0.47s	-	1.13s	3.34s
BMTC(9, 4)	14	-	67.07s	-	2.94s	7.18s
BMTC(10, 4)	16	-	-	-	6.38s	17.34s

Table 8.7: Experimental results for BMTC(p) sequential

Conclusions

- Expressive action language, based on principles of LP
- Competitive implementation with suitable encodings (without specialized heuristics (yet)).
- The idea is similar to SAT Planning or CPlan (Castellini, et al. 2001): Translate to a declarative formalism and use existing solvers (`dlv`, `smodels`, etc.).
- Improvements (Magic Sets, Heuristics?), etc.
- Further steps: Integrated encodings, Conditional Planning(?) Plan Repair.

References

- M. Gelfond, V. Lifschitz *"Action Languages"* Electronic Transactions on Artificial Intelligence 2(3-4), 1998.
- V. Lifschitz *"Answer Set Planning" Proc. of the 16th Int. Conf. on Logic Programming (ICLP'99), 1999.*
- *T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres* "A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity" *ACM TOCL. 5(2), 2004.*
- *T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres* "A Logic Programming Approach to Knowledge-State Planning: , II: the DLV$^{\mathcal{K}}$ System" *Artificial Intelligence 144(1-2), 2003. To appear.*
- *T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres* "Answer Set Planning under Action Costs" *JAIR 19 , 2003.*
- *T.Eiter, A.Polleres* "Towards Automated Integration of Guess and Check Programs in Answer Set Programming"*, Proc. LPNMR-7, 2004.*
- *A. Polleres* "Advances in Answer Set Planning"*, PhD thesis, 2003.*

`http://www.dlvsystem.com/K/`