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
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 $\text{DLV}^{\mathcal{K}}$ 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:
\[ a_1 \lor \ldots \lor a_n :\!-\! b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_l. \]

Constraints:
\[ :\!-\! b_1, \ldots, b_k, \neg b_{k+1}, \ldots, b_l. \]

- \( a \)s and \( b \)s 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

\[ \text{interested}(X) \lor \text{curious}(X) :\!-\! \text{attendsTalk}(X). \]
\[ \text{attendsTalk}(X) :\!-\! \text{staff}(X), \neg \text{onVacation}(X). \]
Let $M$ be a (consistent) set of literals

A Rule

$$a_1 \lor \ldots \lor a_n : \neg b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_l$$

is called satisfied wrt. $M$ iff:

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

at least one of $a_1, \ldots, a_n \in M$. 
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 not $l$ in the body
  - remove literals 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 \( (\Sigma_2^P = NP^{NP}) \) for normal (disjunctive) logic programs

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

- Formulate planning problem as a logic program \( \Pi \) (describe trajectories)
- Compute any answer sets (i.e. models) of \( \Pi \), 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), $0, 1, \ldots, n$

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).
% 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...
\[
on(B, L, T1) :- \text{move}(B, L, T), \; T1 = T + 1.
\]

% ... and it is no longer at the old location.
\[
-\text{on}(B, L, T1) :- \text{move}(B, L1, T), \text{on}(B, L, T), \; T1 = T + 1,
\]
\[
\quad \quad \quad L1 \leftrightarrow L.
\]

% Inertia: Unless a block is \textit{known} to be moved, assume it is still at old place.
\[
on(B, L, T1) :- \text{on}(B, L, T), \text{not} - \text{on}(B, L, T1), \; T1 = T + 1.
\]

% Constraints
\[
:- \text{move}(B, L, T), \text{on}(\_), B, T). \quad \quad \% \text{A block can only be moved if it’s clear.}
\]
\[
:- \text{move}(B, B1, T), \text{on}(\_), B1, T), \text{block}(B1). \quad \% \text{No move onto an occupied block}
\]
\[
:- \text{move}(B, \_), T), \text{move}(B1, \_), T), B \neq B1. \% \text{Move only one block at each step}
\]
\[
:- \text{move}(\_, L, T), \text{move}(\_, L1, T), L \neq L1. \quad \% \text{No move to different locations}
\]
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. $A$, $AR$, $AK$, $C$, $K$, ...
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
Planning Domains and Problems

Background Knowledge $\Pi$: A logic program $\Pi$ 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 $i \geq 0$. 
Background knowledge

\[ \Pi = \{ \text{block}(a). \text{block}(b). \text{block}(c). \]
\[ \quad \text{location}(\text{table}). \]
\[ \quad \text{location}(L) : - \text{block}(L). \} \]

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

\(AD:\)

**fluents:**
- \(\text{on}(B,L)\) requires \(\text{block}(B)\), \(\text{location}(L)\).
- \(\text{occupied}(B)\) requires \(\text{location}(B)\).

**actions:**
- \(\text{move}(B,L)\) requires \(\text{block}(B)\), \(\text{location}(L)\).

**always:**
- executable \(\text{move}(B,L)\) if not \(\text{occupied}(B)\),
  - not \(\text{occupied}(L)\), \(B <> L\).

- inertial \(\text{on}(B,L)\).
  - caused \(\text{on}(B,L)\) after \(\text{move}(B,L)\).
  - caused \(- \text{on}(B,L1)\) after \(\text{move}(B,L)\), \(\text{on}(B,L1)\), \(L <> L1\).

- caused \(\text{occupied}(B)\) if \(\text{on}(B1,B)\), \(\text{block}(B)\).

**noConcurrency.**

**initially:**
- \(\text{on}(a,\text{table})\).
- \(\text{on}(b,\text{table})\).
- \(\text{on}(c,a)\).
**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**
- 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).
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).
noConcurrent.
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: on(c, b), on(b, a), on(a, table)?(3)

intuitively: Feasible plan is

move(c, table); move(b, a); move(c, b)
Qualification Problem:

Overriding by exceptions to executability

executable act if \(< \text{cond1}\>\)
nonexecutable act if \(< \text{cond2}\>\)

Example:

executable move(B,L) if B \neq L.
nonexecutable move(B,L) if occupied(B).
nonexecutable move(B,L) if occupied(L).
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).
Ramifications/State Axioms:

Simple by causal rules:

caused supported(B1) if on(B1, table).
caused supported(B1) if on(B1, B2), supported(B2).

Note:

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

transition-based semantics

cause if $\text{Cond1}$ after $\text{Cond2}$

A ... set of actions (executable)

- $\text{Cond2}$ is evaluated in $s_0$, might include actions.
- $\text{fl}$ and $\text{Cond1}$ are evaluated in $s_1$

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

$\text{fl} :- \text{Cond1}$

Remark: several answer sets $\Leftrightarrow$ several possible succ. states
Incomplete states

Usual assumption for action language: total states
State $s$ is total $\iff$ For each fluent $f$, either $f$ or $\neg f$ must be in $s$.

Total states correspond to total (w.r.t. strong negation!) interpretations
Incomplete state: values of some fluents are unknown.

Handle incompleteness using principles from nonmonotonic Logic Programming

Note: Differs from Kripke-style $A_K$ (Baral & Son, 2001)
Plans

Trajectories:

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

- \( s_0 \) 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: \( P = \text{move}(c, \text{table}); \text{move}(b, a); \text{move}(c, b) \)
Planning under uncertainty in $\mathcal{K}$

- Incomplete states

- Use of default principles: Express “Unknown”
  
  executable check_door if not open, not — open.
  forbidden not on(B,L), not — on(B,L).

- “Totalize” fluents (case distinction)
  
  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

\[ \Pi = \{ \text{package}(1). \ \text{package}(2). \ \ldots \ \text{package}(p). \} \]

**fluents:**  \( \text{armed}(P) \) requires \( \text{package}(P) \).

\( \text{unsafe} \).

**actions:**  \( \text{dunk}(P) \) requires \( \text{package}(P) \).

\( \text{executable} \ \text{dunk}(P) \).

\( \text{inertial} \ \text{armed}(P) \).

\( \text{caused} \ - \ \text{armed}(P) \) after \( \text{dunk}(P) \).

\( \text{caused} \ \text{unsafe} \) if \( \text{armed}(P) \).

**initially:**  total \( \text{armed}(P) \).  % 3 lines: one package is armed

\( \text{forbidden} \ \text{armed}(P), \ \text{armed}(P1), \ P <> P1. \)

\( \text{forbidden} \ \text{not} \ \text{unsafe} \).

**goal:**  \( \text{not} \ \text{unsafe} \ ? \ (p) \)

Encodes ALL possible world states!
Example: Bomb in the Toilet

$$\Pi = \{\text{package}(1), \text{package}(2), \ldots \text{package}(p)\}$$

**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($P_1$), $P \leftrightarrow P_1$.
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"

⇒ "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 ⇒ need no security check
Similar to “unknown” fluents in the initial state, we can use knowledge state encodings to “forget” about certain facts by overriding inertia. Example: non-deterministic “clogging” in Bomb in the toilet.

\[
\text{total clogged(T) after flush(T)}
\]
\[
\text{inertial } \neg \text{clogged(T)}. \]

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

Alternative:

\[
\text{inertial } \neg \text{clogged(T) after not flush(T)}. \]

“The toilet stays unclogged \textbf{unless} it has been flushed”
Avoiding totalization is not always possible: example SQUARE
[Bonet-Geffner, 2000]:

fluent: atX(P) requires index(P). atY(P) requires index(P). anywhere.
action: up. down. left. right.
always: executable up. executable right. executable left. executable down.
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)? ($n$)
Translations to Disjunctive Datalog - The $\text{DLV}^K$ Planning System
Translation

caused on(B,L) after move(B,L).
⇒ on(B,L,T₁) :- move(B,L,T), T₁ = T + 1.

inertial on(B,L).
(= caused on(B,L) if not ¬on(B,L) after on(B,L).)
⇒ on(B,L,T₁) :- on(B,L,T), not ¬on(B,L,T₁), T₁ = T + 1.

initially caused on(a,table).
⇒ on(a,table,0).

executable move(B,L) if B != L.
⇒ move(B,L,T) ∨ ¬move(B,L,T) :- B != L, block(B), location(L),
  actiontime(T).

goal: on(c,b), on(b,a), on(a,table)? (3)
⇒ goal:- on(a,table,3), on(b,a,3), on(c,b,3).
  :- not goal.
  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 $\Pi_{\text{check}}(p)$, or an action of $p$ is not executable.

Algorithm for secure planning:
- compute optimistic plans (i.e. answer sets)
- Create $\Pi_{\text{check}}$ for optimistic plan) found, if has no answer set, the plan is secure (i.e. conformant).
- use caching.

DLV$^K$ System

Architecture:

- K input
- Background Knowledge
- K Parser
- Datalog Parser
- Plan Generator
- Plan Checker
- Plan Printer
- DLV Core
- K Core
- Controller

Control Flow: →
Data Flow: ——>
### Table 8.6: Experimental results for BMTC($p$), conc. duks

<table>
<thead>
<tr>
<th>BMTC($p$, $t$)</th>
<th>steps</th>
<th>DLV5 $w$</th>
<th>CPLAN $k$</th>
<th>SGP $k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMTC(3, 2)</td>
<td>1</td>
<td>0.02s</td>
<td>0.01s</td>
<td>1.41s</td>
</tr>
<tr>
<td>BMTC(3, 2)</td>
<td>3</td>
<td>0.02s</td>
<td>0.01s</td>
<td>1.50s</td>
</tr>
<tr>
<td>BMTC(4, 2)</td>
<td>3</td>
<td>0.11s</td>
<td>0.03s</td>
<td>1.72s</td>
</tr>
<tr>
<td>BMTC(5, 2)</td>
<td>5</td>
<td>2.70s</td>
<td>0.03s</td>
<td>3.37s</td>
</tr>
<tr>
<td>BMTC(6, 2)</td>
<td>5</td>
<td>37.04s</td>
<td>0.07s</td>
<td>13.04s</td>
</tr>
<tr>
<td>BMTC(7, 2)</td>
<td>7</td>
<td>-</td>
<td>0.52s</td>
<td>71.50s</td>
</tr>
<tr>
<td>BMTC(8, 2)</td>
<td>7</td>
<td>-</td>
<td>10.66s</td>
<td>-</td>
</tr>
<tr>
<td>BMTC(9, 2)</td>
<td>9</td>
<td>-</td>
<td>306.27s</td>
<td>-</td>
</tr>
<tr>
<td>BMTC(10, 2)</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 8.7: Experimental results for BMTC($p$) sequential

<table>
<thead>
<tr>
<th>BMTC($p$, $t$)</th>
<th>steps</th>
<th>DLV5 $w$</th>
<th>CPLAN $k$</th>
<th>CMAC $k$</th>
<th>GPT $k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMTC(2, 3)</td>
<td>2</td>
<td>0.02s</td>
<td>0.02s</td>
<td>1.41s</td>
<td>0.04s</td>
</tr>
<tr>
<td>BMTC(3, 2)</td>
<td>4</td>
<td>0.07s</td>
<td>0.02s</td>
<td>1.50s</td>
<td>0.05s</td>
</tr>
<tr>
<td>BMTC(4, 2)</td>
<td>6</td>
<td>2.47s</td>
<td>0.04s</td>
<td>1.64s</td>
<td>0.06s</td>
</tr>
<tr>
<td>BMTC(5, 2)</td>
<td>8</td>
<td>208.52s</td>
<td>0.05s</td>
<td>8.06s</td>
<td>0.06s</td>
</tr>
<tr>
<td>BMTC(6, 2)</td>
<td>10</td>
<td>-</td>
<td>0.07s</td>
<td>32.77s</td>
<td>0.06s</td>
</tr>
<tr>
<td>BMTC(7, 2)</td>
<td>12</td>
<td>-</td>
<td>0.10s</td>
<td>12.46s</td>
<td>0.06s</td>
</tr>
<tr>
<td>BMTC(8, 2)</td>
<td>14</td>
<td>-</td>
<td>0.13s</td>
<td>-</td>
<td>0.31s</td>
</tr>
<tr>
<td>BMTC(9, 2)</td>
<td>16</td>
<td>-</td>
<td>0.20s</td>
<td>-</td>
<td>0.48s</td>
</tr>
<tr>
<td>BMTC(10, 2)</td>
<td>18</td>
<td>-</td>
<td>0.28s</td>
<td>-</td>
<td>0.56s</td>
</tr>
</tbody>
</table>

Table 8.6: Experimental results for BMTC($p$), conc. duks

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


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