Planning with Action Languages

Wolfgang Faber

Institute for Information Systems

Vienna University of Technology

wf@wfaber.com

http://www.wfaber.com

Overview

- What is planning?
- Formalising planning
- Tour of Language $\mathcal{K}$
- Special features of $\mathcal{K}$
- Conclusions

What is Planning?

- Starting from a situation
- Try to reach a goal
- By executing some actions

Given the initial situation, a goal, and some actions, choose the actions such that the goal is achieved.

Planning – Example

- I am at the university at Linz at 12:00
- And have to give a talk at TU Vienna at 16:00
- I can exit the university, take the tram to the train station, enter the train to Vienna, exit the train at Westbahnhof, enter U3 to Simmering, exit at Stephansplatz, enter U1 to Reumannplatz, exit at Taubstummengasse and enter TU Vienna to achieve the goal.

Obviously, there could also be other plans (e.g. going by car if one is available to me), taking a taxi from Westbahnhof to the university, etc.

Note also that the plan need not always work (the train could have a delay, the car could break down etc).

Planning – How to formalize?

We can identify 2 main elements:

- Static statements “I am at the train station in Linz.”
- Action statements “Enter the train to Vienna.”

Planning – How to formalize?

Observations:

- In general, action statements need a *precondition* (which is a static statement), e.g. I can board a train in Linz only if I am currently in Linz.
- Action statements usually have *effects* (which are static statements)
  e.g. exiting the train at Westbahnhof has the effect that I am at Westbahnhof.
- Often, static statements remain true even if they are not affected by actions
  e.g. if I am at the Linz station and do an action “drink coffee”, I am still at Linz afterwards.
  Such static statements are called *inertial*.

Planning

Input: Fluents (static statements), e.g. “I am in Linz.” “It is 15:05.”

Initial state I, described by fluents

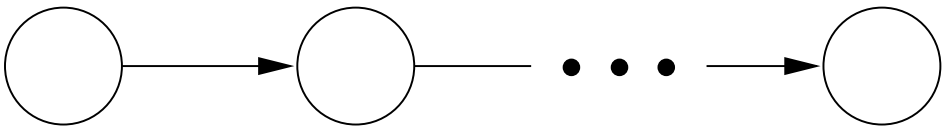
Goal (desired) state G, described by fluents

Actions, e.g. “Enter the train to Vienna.”

Description of the action effects and inertial fluents

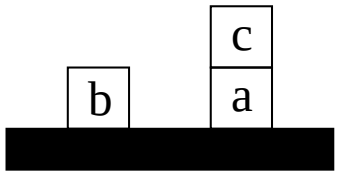
Problem: Find a sequence of action sets $\langle A_0, A_1, \ldots, A_n \rangle$

transforming the initial state into the goal state

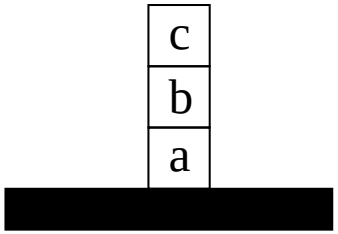


Example: Blocksworld Planning

initial:



goal:



Fluents: on(B,L) *(block B is on top of location L)*

Initial state: on(b, table), on(c, a), on(a, table).

Goal: on(c,b), on(b, a), on(a, table).

Actions: move(B,L) *(move block B to location L).*

System Description: source and destination locations must be clear, no concurrent actions, ...

$\mathcal{K}$– a declarative (logic-based) planning language

- high expressiveness
- nonmonotonic negation, strong negation
- causation rules
- inertia
- conditional executability
- initial state constraints
- representation of **incomplete knowledge** (conformant plans)
- Implementation of planning with $\mathcal{K}$ on top of DLV

Language $\mathcal{K}$: Background Knowledge

We assume that an Answer Set Program (the *background knowledge*) exists, which admits exactly one answer set which is computable in polynomial time.

In the blocksworld example, it is

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

It admits one answer set:

```
{block(a), block(b), block(c), location(table),
    location(a), location(b), location(c)}
```

Language $\mathcal{K}$: Type Declarations

Specify the ranges of the arguments of the fluents and actions, using the background knowledge.

For fluents `on` and `occupied`:

```
on(B,L) requires block(B), location(L).
occupied(B) requires location(B).
```

For action `move`:

```
move(B,L) requires block(B), location(L).
```

Language $\mathcal{K}$: Causation Rules

Causation rules are used to specify action effects and effects within states.

If some block `B1` is on another block `B`, then `B` is occupied in the same moment:

```
caused occupied(B) if block(B), on(B1,B).
```

Moving a block to a location causes the block to be on the location afterwards:

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

One causation rule can also have both `if` and `after` conditions.

Language $\mathcal{K}$: Causation Rules

Default negation (`not`)

```
caused clear(B) if not occupied(B).
```

Strong negation (-)

```
caused -on(B,L) after move(B,L1), L ≠ L1.
```

Difference between default negation and strong negation:

Default negation: `not` a holds if a cannot be proved

Strong negation: $-a$ holds only if $-a$ can be proved explicitly

Language $\mathcal{K}$: Constraints

Constraints are special causation rules:

```
forbidden a after b.
```

Any state satisfying `a after b` is illegal. Such constraints can also be written as

```
caused false if a after b.
```

Language $\mathcal{K}$: Conditional Executability

Executability conditions state preconditions which must hold for an action to be applicable in some state.

For example, `move` is applicable if neither the moved block nor the destination block are occupied and if the moved block and the destination block are not the same.

```
executable move(X,Y) if not occupied(X),
                        not occupied(Y), X≠Y.
```

An executability condition can have both `if` and `after` conditions.

Language $\mathcal{K}$: Initial State Constraints

It is sometimes useful to have causation rules which only apply to the initial state.

These rules cannot have an `after` condition, and they are grouped into a block preceded by `initially:`, while causation rules applying to all states are gruped into another block preceded by `always:`.

```
initially: forbidden block(B), not
           supported(B).
```

Language $\mathcal{K}$: Goal

A goal is a conjunction of fluents, plus a planlength.

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

A goal may also contain default negated fluents.

Language $\mathcal{K}$: Inertia

A fluent may be declared to be inertial:

```
inertial on(B,L).
```

An inertial fluent `a` continues to hold, unless `-a` is the consequence of a preceding action.

Language $\mathcal{K}$: Parallel vs. Sequential Plans

Can actions be executed in parallel or only one at a time?

In $\mathcal{K}$, parallel actions are allowed by default. Sequential plans can be enforced by the keyword:

```
noConcurrency.
```

Blocksworld in $\mathcal{K}$

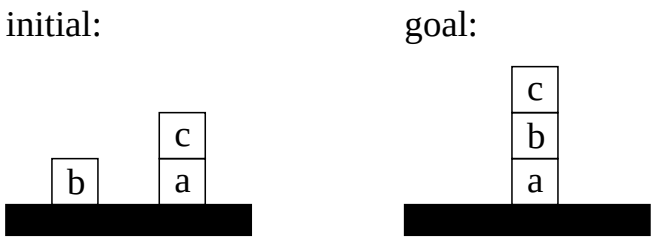
(Domain Description)

```
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 occupied(B) if on(B1,B), block(B).
         caused  on(B,L)  after move(B,L).
         caused -on(B,L1) after move(B,L), on(B,L1), L<>L1.

noConcurrency.
```

Blocksworld in $\mathcal{K}$ (cont'd)

(Planning Instance)

```
initially:  on(a,table). on(b,table). on(c,a).

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



Result of the execution

```
PLAN: move(c,table); move(b,a); move(c,b)
```

Special Features of $\mathcal{K}$

- Handling of complete and incomplete knowledge.
- Secure Plans.

For some state and some fluent `f`, neither `f` nor `-f` needs to hold. The fluent is then “unknown”.

Special Features of $\mathcal{K}$– Totality

If nothing is unknown about a fluent or in a state, it is called total.

Check totality for the fluent `on`:

```
forbidden not on(B,L), not -on(B,L).
```

“Totalize” a fluent:

```
total on(X,Y).
```

Special Features of $\mathcal{K}$– Security

There may be multiple initial states, or multiple successor states.

Secure Plans or **Conformant Plans**

A plan is secure, if it always reaches the goal and if its actions are always applicable.

In $\mathcal{K}$, this planning mode is activated by including the keyword

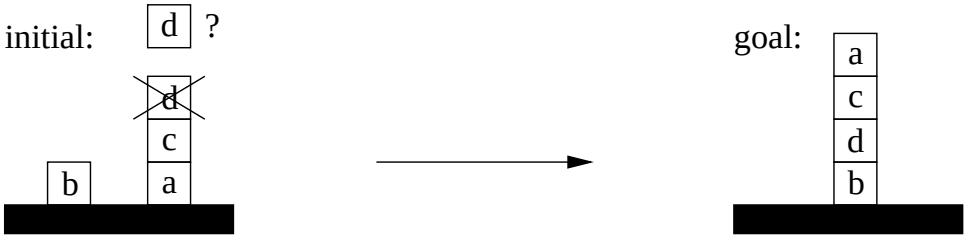
`securePlan.`

Checking correctness of the initial state

```
fluents:  ....
      supported(B) requires block(B).

initially:....
      caused false if on(B,L),  on(B,L1), L <> L1.
      caused false if on(B1,B), on(B2,B), block(B), B1<>B2.
      caused supported(B) if on(B,table).
      caused supported(B) if on(B,B1), supported(B1).
      caused false if not supported(B).
```

Reasoning under Incomplete Knowledge

The exact location of d is unknown, but we know that it is not on top of c.




```
initially:....
          -on(d,c).
          total on(X,Y).
goal:     on(a,c), on(c,d), on(d,b), on(b,table)? (4)
securePlan.
```

Result of the execution

```
 PLAN: move(d,table,0),move(d,b,1),move(c,d,2),move(a,c,3)
```

Complexity Results

Proper domain: given a state s and an action sequence A, the existence of a legal state transition $\langle s, a, s' \rangle$ is polynomially decidable.

Theorem Deciding whether a given proper ground planning problem $\langle PD, q \rangle$ has an optimistic plan is PSPACE-complete, and NP-complete if the number of steps in q is fixed.

Theorem Given an optimistic plan P and a given proper ground planning problem $\langle PD, q \rangle$, deciding whether P is secure is coNP-complete. Hardness holds even if the number of steps in q is fixed.

Complexity Results (cont'd)

Theorem Deciding whether a given proper ground planning problem $\langle PD, q\rangle$ has a secure plan is NEXPTIME-complete in general and Σ_2^P-complete, if the number of steps in q is fixed.

Theorem Deciding whether a given proper ground planning problem $\langle PD, q\rangle$ has a secure sequential plan is D^P-complete, if the number of steps in q is fixed. (D^P is the conjunction of NP and coNP.)

Remark: The complexity increases if the domain is not proper!

Implementation

- Rewriting to DLP programs.
- Prototype ($\mathsf{DLV}^{\mathcal{K}}$) implemented on top of DLV.
- Freely available in standard DLV distribution.
- Run DLV with option **-FP** (Front-end Planning).

Action Costs

Associate a cost of 1 to action `move` :

```
move(B,L) requires block(B), location(L) costs 1.
```

Time-dependent costs for action `move`:

```
move(B,L) requires block(B), location(L) costs time.
```

Other Action Languages

- $\mathcal{A}$
- $\mathcal{B}$
- $\mathcal{C}$
- $\mathcal{C}+$
- ...

Conclusion

- $\mathcal{K}$ — an expressive declarative planning language.
  - More “logic programming” oriented, than other similar languages.
  - Reasoning under Incomplete Knowledge.
- Complexity Analysis.
- Fully operational prototype available at
  `http://www.dlvsystem.com/K/`