# The Declarative Planning System DLV$^{\mathcal{K}}$: Progress and Extensions *

**Axel Polleres**
axel@kr.tuwien.ac.at
Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria

## Introduction

The knowledge based planning system DLV$^{\mathcal{K}}$ implements answer set planning on top of the DLV system (Eiter *et al.* 2000). It is developed at TU Wien and supports the declarative language $\mathcal{K}^c$ (Eiter *et al.* 2003b; 2002a; 2002b). The language $\mathcal{K}^c$ is syntactically similar to the action language $\mathcal{C}$ (Giunchiglia & Lifschitz 1998), but semantically closer to answer set programming (by including default negation, for example). $\mathcal{K}^c$ offers the following distinguishing features:

- *Handling of incomplete knowledge:* for a fluent $f$, neither $f$ nor its opposite $\neg f$ need to be known in any state.
- *Nondeterministic effects:* actions may have multiple possible outcomes.
- *Optimistic and secure (conformant) planning:* construction of a "credulous" plan or a "sceptical" plan, which works in all cases.
- *Parallel actions:* More than one action may be executed simultaneously.
- *Optimal cost planning:* In $\mathcal{K}^c$, one can assign a cost function to each action, where the total costs of the plan are minimized.

The prototype planning system DLV$^{\mathcal{K}}$ based on $\mathcal{K}^c$ is available at `http://www.dlvsystem.com/K/`.

We report here briefly on the language capabilities, system architecture, and ongoing progress of the system as well as extensions we are currently working on.

## Action Language $\mathcal{K}^c$ by example

We assume that the reader is familiar with action languages and the notion of actions, fluents, goals, and plans; see e.g. (Gelfond & Lifschitz 1998), for a background, and (Eiter *et al.* 2003b; 2002a) for the detailed syntax and semantics of our language $\mathcal{K}^c$.

For illustration, we shall use the following running example, for which a $\mathcal{K}^c$ encoding is shown in Figure 1:

**Bridge crossing.** Four men want to cross a river at night. It is bridged by a plank bridge, which can only hold up to two persons at a time. The men have a lamp, which must be used in crossing, as it is pitch-dark. The lamp must be brought back; no tricks (like throwing the lamp or halfway crosses) are allowed. The four guys need different times to cross the bridge, namely 1, 2, 5, and 10 minutes, respectively. Walking in two implies moving at the slower rate. What is the shortest time to get them all across? □

As exemplified in Figure 1 problems specified in our language $\mathcal{K}^c$ consist mainly of two parts: The *background knowledge* which can be viewed as the static knowledge invariant over the time is given as a normal logic program. In our case this includes the persons, their respective crossing times and some auxiliary predicates encoding the different sides of the bridge.

The *problem description* includes the declarations of actions and fluents involved where actions might have costs assigned. In our particular example this cost is just the maximum walk time of two persons crossing. Costs allow for a certain degree of dynamic behavior as they might depend on the point in time, when the action is executed. The actual action description is built up of so called *executability conditions*

```
executable a if F.
```

and *causation rules*

```
caused F if G after H.
```

where `a` is an action atom, `F` is a fluent literal or `false` (i.e. $\bot$, this is used to model constraints), `G` is a set of fluent literals and `H` is a set of fluent literals and actions. The language flexibly allows for modeling arbitrary action effects, static effects (ramifications), executability conditions of actions and state constraints. In the example these executability conditions and causation rules model the effects of crossing, that crossing is only executable when having the lamp, etc. Inertia of fluents is modeled by the macro `inertial F.` which is short for:

```
caused F if not -F after F.
```

Non-monotonic negation `not` in the `if` and `after` parts of causation rules interpreted under a minimal model

*This work was supported by FWF (Austrian Science Funds) under the projects P14781 and Z29-N04.

Background knowledge:

```
person(joe). person(jack). person(william). person(averell).
crossTime(joe, 1). crossTime(jack, 2). crossTime(will, 5). crossTime(averell, 10).
max(A, B, A) :- crossTime(_, A), crossTime(_, B), A >= B.
max(A, B, B) :- crossTime(_, A), crossTime(_, B), B > A.
side(here). side(across).
otherside(X, Y) :- side(X), side(Y), X != Y.
```

$\mathcal{K}^c$ problem description:

```
actions :   crossTogether(X, Y) requires person(X), person(Y), X != Y
                                costs S where crossTime(X, SX), crossTime(Y, SY), max(SX, SY, S).
            cross(X) requires person(X) costs SX where crossTime(X, SX).
            takeLamp(X) requires person(X).

fluents :   at(X, S) requires person(X), side(S).
            hasLamp(X) requires person(X).

always :    executable crossTogether(X, Y) if hasLamp(X), at(X, S), at(Y, S).
            executable crossTogether(X, Y) if hasLamp(Y), at(X, S), at(Y, S).
            executable cross(X) if hasLamp(X).
            executable takeLamp(X) if hasLamp(Y), at(X, S), at(Y, S), X != Y.

            caused at(X, S1) after crossTogether(X, Y), at(X, S), otherside(S, S1).
            caused at(Y, S1) after crossTogether(X, Y), at(Y, S), otherside(S, S1).
            caused -at(X, S) after crossTogether(X, Y), at(X, S).
            caused -at(Y, S) after crossTogether(X, Y), at(Y, S).

            caused at(X, S1) after cross(X), at(X, S), otherside(S, S1).
            caused -at(X, S) after cross(X), at(X, S).

            caused hasLamp(X) after takeLamp(X).
            caused -hasLamp(X) after takeLamp(Y), hasLamp(X).

            inertial at(X, S).
            inertial hasLamp(X).

noConcurrency.

initially : caused at(X, here). caused hasLamp(joe).

goal :      at(joe, across), at(jack, across), at(william, across), at(averell, across)? (7)
```

Figure 1: $\mathcal{K}^c$ encoding of the Bridge crossing problem

semantics of transitions (based on the answer set semantics (Gelfond & Lifschitz 1991)) allows for modeling several legal initial states and several legal transitions to subsequent states from one state wrt. to some executed actions. This can be used to model reasoning under incomplete knowledge and with nondeterministic action effects. We refer to (Eiter *et al.* 2003b; 2003a) for a detailed discussion. By default, $\mathcal{K}^c$ allows simultaneous execution of actions. This is prohibited by the keyword `noConcurrency`. which enforces the execution of at most one action at a time.

The two final lines in our example model initial and goal state, i.e. that initially all persons are at side `here` of the bridge and all should finally be brought `across`. The goal is given as a set of ground literals in our language and includes the plan length (7, in our example).

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

We have implemented an experimental prototype system, DLV$^{\mathcal{K}}$, for solving $\mathcal{K}^c$ planning problems; a detailed description is given in (Eiter *et al.* 2003a). The current system is capable of optimal and admissible [1] planning for planning problems specified in $\mathcal{K}^c$.

For nondeterministic planning domains the system allows for computing *optimistic plans* and *secure plans*: We define plans as sequences of sets of actions $P = \langle A_1, \ldots, A_n \rangle$ where actions in the same action set are executed in parallel: $P$ is called *optimistic plan* if there is an execution possibly reaching the goal. If $P$ reaches the goal under all contingencies, we call $P$ *secure* (i.e. conformant, in the sense of (Goldman & Boddy 1996)).

DLV$^{\mathcal{K}}$ has been realized as a frontend to the DLV disjunctive logic programming system (Eiter *et al.* 2000; Leone *et al.* 2002). First, the planning problem at hand is tranformed to a disjunctive logic program where answer sets correspond to optimistic plans. Details about the transformation can be found in (Eiter *et al.* 2003a; 2002b). Then, the DLV kernel is invoked to produce answer sets. For *optimistic planning* the (optimal, if action costs are defined) answer sets are then simply

[1] We call a plan *admissible* if it does not exceed a given cost bound

translated back into suitable output and printed.

Assume that the above background knowledge and planning problem description are given in files `crossing.bk` and `crossing.plan`, respectively. The execution of the command:

```
$ dlv -FP crossing.bk crossing.plan
```

computes for instance the following plan:

```
PLAN: crossTogether(jack,joe):2;
cross(joe):1; takeLamp(averell);
crossTogether(averell,william):10;
takeLamp(jack); cross(jack):2;
crossTogether(jack,joe):2 COST: 17
```

which is indeed an optimal seven step plan for this problem, where the plan cost (i.e. the shortest time for getting them all across) is 17. The costs of single actions are also displayed.

In case of *secure planning*, our system has to check security of the plans computed, i.e. whether the plans work for all possible executions. In normal (non-optimal) planning, this is simply done by checking each answer set returned right from the DLV kernel before transforming it back to user output. On the other hand, for optimal secure planning the candidate answer set generation of the DLV kernel itself has to be "intercepted": Optimal answer set generation wrt. so called weak constraints (Buccafurri, Leone, & Rullo 2000) used in our translation is a built-in feature of DLV: The kernel proceeds computing candidate answer sets, returning an answer set with optimal costs, by running through all candidates. Here, in order to generate *optimal secure plans*, the planning frontend interrupts computation, allowing only answer sets which represent secure plans to be considered as candidates, as we want to find the optimal among the secure plans only.

Checking plan security is done by rewriting the translated logic program wrt. the candidate answer set/plan in order to verify whether the plan is secure. The rewritten "check program" is tested by a separate invocation of the DLV kernel. To avoid duplicate secure checking of a plan, the checked plans are cached.

Our example does not include nondeterminism, so secure checking is not necessary here.

For further details on the system architecture, performance and experimental results we refer to (Eiter *et al.* 2003a; 2002b).

## Theoretical Background of the Translation

In (Eiter *et al.* 2003b; 2002b) we have thoroughly analyzed the computational complexity of optimal optimistic and secure planning. Results obtained there give a fundamental basis for the polynomial translations of these problems to disjunctive logic programs. We assume that the reader is familiar with the basic notions of complexity theory, such as NP and the Polynomial Hierarchy (PH), cf. (Papadimitriou 1994) and references therein.

As for **optimistic planning** we have shown in (Eiter *et al.* 2003b) that optimistic planning with given (polynomial) plan length is NP-complete (propositional case) and finding such a plan is NPMV-complete. Head cycle free disjunctive logic programs evaluated under the answer set semantics cover exactly this class of problems and the respective polynomial translation is given in (Eiter *et al.* 2003a) and implemented in the $\text{DLV}^{\mathcal{K}}$ system outlined above.

When considering **optimistic optimal planning** the complexity of finding such optimal plans to $\text{F}\Delta_2^P$. When extending logic programs with weak constraints (Buccafurri, Leone, & Rullo 2000), computing the respective optimal answer sets wrt. these constraints covers exactly this complexity class. The respective translation is given in (Eiter *et al.* 2002b). As weak constraints are a built-in feature of DLV, we could also implement this translation in our prototype system. The bridge crossing problem belongs to the class of problems solvable in this framework.

**Secure planning** is $\Sigma_3^P$-complete even for given plan length which makes a direct translation to answer set programming infeasible. This is why we decided for the interleaved plan checking approach outlined in the previous section.

**Checking plan security** is $\Pi_2^P$-complete for given plan length, but we have identified subclasses where "cheaper" checks can be applied (for details, see (Eiter *et al.* 2003b; 2003a)):

Check 1 is applicable to programs which are `false`-committed (defined in (Eiter *et al.* 2003a)). This condition is e.g. guaranteed for the class of $\mathcal{K}$ domains which are stratified, when viewed as a logic program.

Check 2 is applicable for domains where the existence of a legal transition (i.e., executability of some action leading to a consistent successor state) is always guaranteed. In (Eiter *et al.* 2002a) we have shown that using these two checks implemented in our system we can solve relevant conformant planning problems.

Both checks carry over to planning with action costs in a straightforward way, and optimal resp. admissible secure plans can be similarly computed by answer set programming. A generalization of these secure checks via a translation to full disjunctive logic programs is under investigation. The methods investigated so far for planning under uncertainty cover only optimistic and secure planning. Furthermore, the possibility of computing conditional plans under full/partial observability by means of translations to logic programs as well could be an interesting direction for further research.

## Extensions in Progress

### Multivalued fluents in $\mathcal{K}^c$

Taking a closer look to the example in Figure 1 one might criticize some deficiencies in the way fluents are represented: Fluent literals in $\mathcal{K}^c$ can only represent boolean predicates. However, in planning we often have to face state variables which do not only take boolean

values but a range of values from a (finite) domain. For the example above, a person can always be at at most one location (`here`, `across`) or exactly one person has the lamp. In order to express such settings, multivalued fluents like in the action language $\mathcal{C}+$ (Giunchiglia *et al.* 2001; Lee & Lifschitz 2001; Giunchiglia *et al.* 2003) would be desirable. Intuitively, we would preferably write something like:

```
caused hasLamp:=jack after takeLamp(jack).
```

instead of two causation rules:

```
caused hasLamp(jack) after takeLamp(jack).
caused -hasLamp(joe) after takeLamp(jack).
```

as effect of `takeLamp(jack)` in a state where joe has the lamp. In $\mathcal{K}^c$ we often use classical negation only to "override" inertia, if fluents are concerned which can only take a distinct value at each point of time. Using this notation, our example in Figure 1 could be represented more compact and comprehensive.

We therefore extend the notion of fluent declarations in $\mathcal{K}^c$ from (Eiter *et al.* 2003b) in order to allow for *multivalued fluent declarations* is of the form:

$$p(X_1, \ldots, X_n) : \mathit{range}\ \texttt{requires}\ t_1, \ldots, t_m \quad (1)$$

where $p$ is the fluent name, $X_1, \ldots, X_n$ are variables and $n \geq 0$ is the arity of $p$. $t_1, \ldots, t_m$ refer to background predicates, $m \geq 0$, every $X_i$ occurs in $t_1, \ldots, t_m$, and *range* is a unary predicate from the background knowledge.

Furthermore, causation rules may contain *multivalued fluent literals* of the form $\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}) := \texttt{V}$ in this extended version, where `V` is a constant or a variable.

The semantics can be defined by viewing multivalued fluent literals as macros for regular boolean fluent literals, where we execute the following preprocessing steps: (i) Each multivalued fluent declaration

```
fluent :   fl(X) : type requires...
```

will be rewritten to a boolean fluent declaration:

```
fluent :   fl(X, Val) requires type(Val)...
```

(ii) Causation rules with multivalued fluent literals can be naively rewritten as follows: Each causation rule with a multivalued fluent literal in the head

```
caused fl(X) := val ...
```

will be transformed to:

```
caused fl(X, val) ...
caused -fl(X, Val1) if val != Val1, type(Val1) ...
```

where `type` is the range predicate from the fluent declaration of `fl`. A (possibly default negated) multivalued fluent literal $\texttt{fl}(\overline{\texttt{X}}) := \texttt{Val}$ in the `if` (resp. `after`) part of a causation rule or executability condition can then be simply rewritten to its boolean counterpart $\texttt{fl}(\overline{\texttt{X}}, \texttt{Val})$.

However, in this macro translation, we do not yet make use of an important representational advantage of $\mathcal{K}^c$: As stated above, classical negation is only used to "override" the old fluent value here, and the subsequent state, i.e. we do not necessarily want to carry over all negative values of `-fl` to the next state. In $\mathcal{K}^c$ states are defined as consistent sets of fluent literals (i.e. the current knowledge of the planning agent about the world) and not as a mapping from fluents to truth values like in other approaches. So the naive transformation from above can be further simplified for $\mathcal{K}^c$ under certain circumstances.

## Dynamic action costs

A further extension of $\mathcal{K}^c$ are fully dynamic action costs. Plan costs in $\mathcal{K}^c$ are defined as the sum of the single action costs of all actions in the plan, where plans are sequences of (sets of) actions. At present, $\mathcal{K}^c$ only allows for action costs with very restricted means of dynamic cost contributions: Action costs may only depend on background knowledge facts or on the time when the action occurs, but not on dynamic fluent values.

However, the current restriction has a practical reason: Whereas in deterministic planning with complete knowledge each plan corresponds to a unique sequence of states, whenever nondeterminism comes into play, a plan might have several possible "trajectories" [2]. The current restriction guarantees unique costs of plans as different intermediate states, i.e. different fluent values can not influence action costs per definition.

Nevertheless, dynamic costs are an important issue: For instance, in our example we have no possibility to express that a person gets tired when crossing several times. This could for instance be expressed as follows:

```
fluents :   fatigue(X) : integer requires person(X).
actions :   cross(X) requires person(X) costs C
                where crossTime(X, SX),
                fatigue(X) := K, C = SX + K.
  ⋮
always :    caused fatigue(X) := K1 after
                cross(X), fatigue(K), K1 = K + 1.
            inertial fatigue(X) := K.
```

expressing that a person fatigues from subsequent crosses, so the cost of crossing is dependent on the new fluent `fatigue`, which increases by one on every cross. Things get even more involved if fatigue is nondeterministic, i.e. a person might fatigue or not from crossing. This could be modeled:

```
always :    caused fatigue(X) := K1 if
                not fatigue(X) := K after
                cross(X), fatigue(X) := K, K1 = K + 1.
            caused fatigue(X) := K if
                not fatigue(X) := K1 after
                fatigue(X) := K, K1 != K.
```

where the unstratified negation in these two causation rules models nondeterminism. In a setting with dynamic costs we have to extend our definition of plan costs from (Eiter *et al.* 2002b) wrt. trajectories [3]:

[2] We use the term trajectory for a sequence of state transitions.

[3] For the exact definitions of transitions, trajectories and plans we refer to (Eiter *et al.* 2003b)

**Definition 1** *Let $T = \langle t_1, \ldots, t_l \rangle$ be a trajectory, where $t_j = \langle s_{j-1}, A_j, s_j \rangle$ is a legal state transition from state $s_{j-1}$ to state $s_j$ executing a set of actions $A_j$ ($j = 1, \ldots, l$). Then, the cost of $T$ wrt. a $\mathcal{K}^c$ planning domain is defined as*

$$cost(T) = \sum_{j=1}^{l} \left( \sum_{a \in A_j} cost_j(a, s_{j-1}) \right).$$

*where $cost_j(a, s_{j-1})$ is the cost of action $a$ wrt. state $s_{j-1}$ at time $j$ according to the* `costs` *part of the resp. action declaration of $a$.*

Now that action costs depend on the state where they are executed we have to change the definition of plan costs accordingly:

**Definition 2** *Let $\mathcal{P}$ be a $\mathcal{K}^c$ planning problem. Then, for any plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$, where $A_i$ is the set of actions executed at time $i$, the cost of $p$ is defined as the maximum cost over all trajectories $T$ constituting a successful execution of $p$ (written $T \models p$), i.e.*

$$cost(p) = \max_{T \models p} cost(T).$$

An *optimal plan* is again a plan with minimal cost; an *admissible* plan wrt. cost $c$ is a plan with $cost(p) \leq c$. As for secure planning, we define optimal plans as plans with minimal costs among all secure plans.

This cautious definition of optimality could be in particular important for an estimation of worst case bounds in presence of uncertainty in critical applications.

As for implementation, our current approach for computing optimal plans outlined above is not feasible any longer wrt. this definition. In order to surmount this, a naive approach would be caching all plans and their maximal costs during answer set computation. However, as there might be an exponential number of plans, caching would be highly inefficient. A better strategy is again "intercepting" candidate answer set generation at the same point where checking plan security is performed now: In order to find the optimal plan, consider only those answer sets representing maximum cost trajectories by checking whether there is a more expensive trajectory for the answer set at hand. Possible improvements of this computation and analysis of complexity aspects are under investigation.

## Conclusions and Outlook

The presented work describes further progress in the research on modeling planning tasks in a declarative high-level language $\mathcal{K}^c$ and solving such tasks by efficient reductions of the respective problems to answer set programs. We have outlined the architecture of the running prototype DLV$^{\mathcal{K}}$ based on the DLV system. Furthermore, we have proposed two upcoming extensions we are currently working on in order to (i) allow for a more concise representation of problems and (ii) be able to expressed and solve a wider range of planning problems in our framework.

## References


Buccafurri, F.; Leone, N.; and Rullo, P. 2000. Enhancing Disjunctive Datalog by Constraints. *IEEE TKDE* 12(5):845–860.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 2000. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*. Kluwer. 79–103.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2002a. Answer Set Planning under Action Costs. *Proc. of the 8th European Conference on Artificial Intelligence (JELIA)*, LNCS 2424, 186–197.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2002b. Answer Set Planning under Action Costs. Tech. Report INFSYS RR-1843-02-13, TU Wien. Accepted for publication in Journal of Artificial Intelligence Research.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003a. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artificial Intelligence* 144(1–2):157–211.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003b. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM TOCL* To appear.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

Gelfond, M., and Lifschitz, V. 1998. Action Languages. *Electronic Transactions on Artificial Intelligence* 2(3-4):193–210.

Giunchiglia, E., and Lifschitz, V. 1998. An Action Language Based on Causal Explanation: Preliminary Report. In *AAAI '98*, 623–630.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; and Turner, H. 2001. Causal Laws and Multi-Valued Fluents. In *Working Notes of the Fourth Workshop on Nonmonotonic Reasoning, Action and Change.*

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2003. Nonmonotonic causal theories. *Artificial Intelligence* Special Issue on Logical Formalizations of Commonsense Reasoning. To appear.

Goldman, R., and Boddy, M. 1996. Expressive Planning and Explicit Knowledge. In *Proc. AIPS-96*, 110–117. AAAI Press.

Lee, J., and Lifschitz, V. 2001. Additive Fluents. *Proc. AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, 116–123. Stanford, CA: AAAI Press.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Koch, C.; Mateis, C.; Perri, S.; and Scarcello, F. 2002. The DLV System for Knowledge Representation and Reasoning. Technical Report INFSYS RR-1843-02-14, TU Wien, A-1040 Vienna, Austria.

Papadimitriou, C. H. 1994. *Computational Complexity.* Addison-Wesley.