

The DLV^K Planning System: Progress Report ^{*}

Thomas Eiter¹, Wolfgang Faber¹,
Nicola Leone², Gerald Pfeifer¹, and Axel Polleres¹

¹ Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
{eiter,faber,axel}@kr.tuwien.ac.at, pfeifer@dbai.tuwien.ac.at

² Department of Mathematics, University of Calabria, 87030 Rende, Italy
leone@unical.it

1 Introduction

The knowledge based planning system DLV^K implements answer set planning on top of the DLV system [1]. It is developed at TU Wien and supports the declarative language \mathcal{K} [2,3] and its extension \mathcal{K}^c [5]. The language \mathcal{K} is syntactically similar to the action language \mathcal{C} [7], but semantically closer to answer set programming (by including default negation, for example). \mathcal{K} and \mathcal{K}^c offer 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 an arbitrary cost function to each action, where the total costs of the plan are minimized.

An operational prototype of DLV^K is available at

<URL:<http://www.dlvsystem.com/K/>>.

We report here briefly the architecture and new features of the system which we have accomplished in the last year.

2 System architecture

The architecture of DLV^K is outlined in Figure 1. The input of the system consists of domain descriptions (DLV^K files) and optional static background knowledge specified by a disjunctive logic program having a unique answer set. Input is read from an arbitrary number of files, converted to an internal representation and stored in a common database.

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

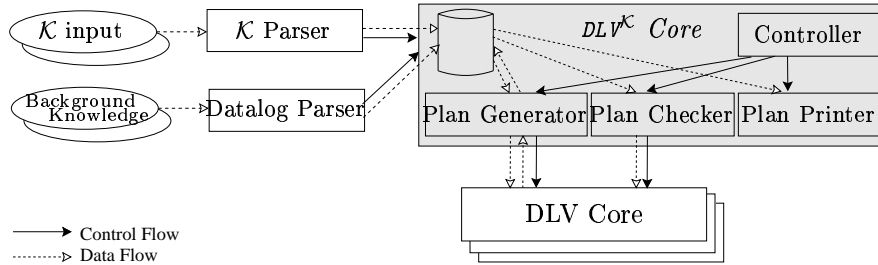


Fig. 1. DLV^K System Architecture

The actual DLV^K front-end consists of four main modules: The *Controller* manages the other three modules; it performs user interactions (where appropriate), and controls the execution of the entire front-end. The Controller first invokes the *Plan Generator*, which translates the planning problem at hand into a suitable program in the core language of DLV (disjunctive logic programming under the answer set semantics). The Controller then invokes the DLV kernel to solve the corresponding problem. The resulting answer sets (if any exist) are fed back to the Controller, which extracts the solutions to the original problem from these answer sets and transforms them back to the original planning domain.

If the user specified that secure/conformant planning should be performed, the Controller then invokes the *Plan Checker*. Similarly to the Plan Generator, the Checker uses the original problem description and, in addition, the optimistic plan computed by the Generator to generate a disjunctive logic program which verifies whether this plan is in fact also secure.

The *Plan Printer*, finally, translates the solutions found by the Generator (and optionally verified by the Checker) back into user output and prints it.

Details about the transformations mentioned above can be found in [4].

3 DLV^K Example: Blocks World

We assume that the reader is familiar with action languages and the notion of actions, fluents, goals, and plans; see e.g. [6], for a background, and [3, 5] for the detailed syntax and semantics of our language \mathcal{K}^c .



Fig. 2. A blocks world example.

To give a flavor of $DLV^{\mathcal{K}}$, we show the encoding of a planning problem in the well-known blocks world. The problem is turning the configuration of blocks on the left hand side of Figure 2 into the one on the right hand side. The static background knowledge of our blocks world is the following logic program:

```
block(1). block(2). block(3). block(4). block(5). block(6).
location(table). location(B) :- block(B).
```

Referring to Figure 2, we want to turn the initial configuration of blocks into the goal state in three steps, concurrent moves allowed.

The \mathcal{K} domain description uses an action `move` and the fluents `on` and `blocked`. In this simple example, we assume that in the initial state the locations of all blocks are known. The domain is described by the following $DLV^{\mathcal{K}}$ program:

```
fluents:   on(B,L) requires block(B), location(L).
           blocked(B) requires block(B).

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

always:    executable move(B,L) if B <> L.
           nonexecutable move(B,L) if blocked(B).
           nonexecutable move(B,L) if blocked(L).
           nonexecutable move(B,L) if move(B1,L), B <> B1, block(L).
           nonexecutable move(B,L) if move(B,L1), L <> L1.
           nonexecutable move(B,B1) if move(B1,L).
           caused on(B,L) after move(B,L).
           caused -on(B,L) after move(B,L1), on(B,L), L <> L1.
           inertial on(B,L).
           caused blocked(B) if on(_,B).

initially: on(1,2). on(2,table). on(3,4). on(4,table). on(5,6). on(6,table)
goal:      on(1,3), on(3,table), on(2,4), on(4,table), on(6,5), on(5,table)
           ? (3)
```

Actions may have assigned costs, which should be minimized. In our case, each move has cost 1, resulting in plans, where a minimum number of moves is executed to achieve the plan.

The rules following the declarations of actions and fluents describe the transitions and constraints on the initial states of the domain. Finally, the `goal`: section defines the goal to be reached and the plan length.

4 Usage of the System

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

```
$ dlv -FP blocks.bk blocks.plan -n=1
```

computes the following result:

```
PLAN: move(3,table), move(5,table); move(1,3), move(6,5); move(2,4)
COST: 5
```

where parallel actions are separated by “,” and plan steps are separated by “;”.

The command-line option `-n=1` tells DLV to compute only one plan. Option `-FP` is used to invoke the planning front-end in DLV. Alternatively, one can use the options `-FPopt` and `-FPsec`, which explicitly enforce optimistic and secure planning, respectively. For planning problems with a unique initial state and a deterministic domain, these two options do not make a difference, as optimistic and secure plans coincide in this case. But for problems with multiple initial states or nondeterministic effects, the latter computes only secure (conformant) plans. With `-FP` the user can interactively decide whether to check plan security.

Checking plan security is Π^2_P -complete in general, but we have identified subclasses where “cheaper” checks can be applied (for details, see [3, 4]). By means of the command-line options `-check=1` and `-check=2` a particular method to check plan security can be chosen, where 1 is the default. Check 1 is applicable to programs which are `false-committed` (defined in [4]). 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 [5] we have shown that using these two checks we can solve relevant conformant planning problems. More general secure checks are on the agenda. As mentioned above, the $DLV^{\mathcal{K}}$ system has been recently extended by the possibility to assign costs to actions, in order to generate cost optimal plans [5]. Apart from optimal plans the system can also compute “cost bound” plans, i.e., all plans which do not exceed a certain cost limit, by means of the command line option `-costbound=N`. In our example, by `-costbound=6` we could also compute plans involving 6 moves.

Performance and experimental results are reported in [4]. However, the results there do not include optimal planning. For example, we performed some promising experiments for parallel blocksworld under action costs. Using the encoding above, the instances P1–P4 from [4] can be solved in less than one second.

References

1. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In: *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.
2. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. *CL2000*, pp. 807–821, London, UK, July 2000. Springer Verlag.
3. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. Technical Report INFSYS RR-1843-01-11, TU Wien, December 2001.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the $DLV^{\mathcal{K}}$ System. Technical Report INFSYS RR-1843-01-12, TU Wien, December 2001.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer set planning under action costs. Unpublished manuscript, available from the authors.
6. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.
7. E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *AAAI '98*, pp. 623–630, 1998.