ELSEVIER

Available at www.ComputerScienceWeb.com POWERED BY SCIENCE DIRECT®

Artificial Intelligence 144 (2003) 157–211

Artificial Intelligence

www.elsevier.com/locate/artint

A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system [☆]

Thomas Eiter [a], Wolfgang Faber [a], Nicola Leone [b,*], Gerald Pfeifer [a], Axel Polleres [a]

[a] *Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9–11, A-1040 Wien, Austria*
[b] *Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy*

Received 10 January 2002

Abstract

In Part I of this series of papers, we have proposed a new logic-based planning language, called $\mathcal{K}$. This language facilitates the description of transitions between states of knowledge and it is well suited for planning under incomplete knowledge. Nonetheless, $\mathcal{K}$ also supports the representation of transitions between states of the world (i.e., states of complete knowledge) as a special case, proving to be very flexible. In the present Part II, we describe the DLV$^{\mathcal{K}}$ planning system, which implements $\mathcal{K}$ on top of the disjunctive logic programming system DLV. This novel planning system allows for solving hard planning problems, including secure planning under incomplete initial states (often called *conformant planning* in the literature), which cannot be solved at all by other logic-based planning systems such as traditional satisfiability planners. We present a detailed comparison of the DLV$^{\mathcal{K}}$ system to several state-of-the-art conformant planning systems, both at the level of system features and on benchmark problems. Our results indicate that, thanks to the power of knowledge-state problem encoding, the DLV$^{\mathcal{K}}$ system is competitive even with special purpose conformant planning systems, and it often supplies a more natural and simple representation of the planning problems.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Deductive planning system; Disjunctive logic programming; Answer sets; Knowledge-states; Incomplete information; Conformant planning; Secure planning

[☆] A preliminary system description has appeared in: Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01), Lecture Notes in Computer Science, Vol. 2173, Springer, 2001, pp. 429–433.

[*] Corresponding author.

E-mail addresses: eiter@kr.tuwien.ac.at (T. Eiter), faber@kr.tuwien.ac.at (W. Faber), leone@unical.it (N. Leone), pfeifer@dbai.tuwien.ac.at (G. Pfeifer), axel@kr.tuwien.ac.at (A. Polleres).

0004-3702/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0004-3702(02)00367-3

1. Introduction

The need for modeling the behavior of robots in a formal way led to the definition of logic-based languages for reasoning about actions and planning, such as [13,16–18,20,21, 26,33,45]. These languages allow for specifying planning problems of the form "Find a sequence of actions that leads from an initial state to a goal state".

A state is characterized by the truth values of a number of fluents, describing relevant properties of the domain of discourse. An action is applicable only if some preconditions (formulas over the fluents) hold in the current state; executing this action changes the current state by modifying the truth values of some fluents. Most of these languages are based on extensions of classical logics and describe transitions between *possible states of the world* where every fluent necessarily is either true or false. However, robots usually do not have a *complete view* of the world. Even if their knowledge is incomplete (a number of fluents may be unknown, e.g., whether a door in front of the robot is open), they must take decisions, execute actions, and reason on the basis of their (incomplete) information at hand. For example, if it is not known whether a door is open, the robot might do a sensing action, or decide to push back.

In [5,6], we have proposed a new language, $\mathcal{K}$ (where $\mathcal{K}$ should remind of states of *k*nowledge) for planning under incomplete knowledge. This language is very flexible, and is capable of modeling transitions between states of the world (i.e., states of complete knowledge) and reasoning about them as a particular case. Compared to similar planning languages, in particular Giunchiglia and Lifschitz' action language $\mathcal{C}$ [17,26,29], $\mathcal{K}$ is closer in spirit to answer set semantics [12] than to classical logics. It supports the explicit use of default negation, and thus exploiting the power of answer sets to deal with incomplete knowledge. In [6] we have defined the syntax and semantics of $\mathcal{K}$, discussed how it can be used for knowledge representation, plus we have analyzed the computational complexity of planning in $\mathcal{K}$.

In the present paper, which is Part II of this series of papers, we turn to the DLV$^{\mathcal{K}}$ planning system, which implements $\mathcal{K}$ on top of the DLV answer set programming system [7,9]. DLV$^{\mathcal{K}}$ is a powerful planning system, which is freely available at <URL:http://www.dbai.tuwien.ac.at/proj/dlv/> and ready-to-use for experiments. In comparison to similar logic-based planning systems like CCALC [30,31], CPlan [10,15], or CMBP [4] DLV$^{\mathcal{K}}$ has the following key features:

- *Explicit background knowledge*: The planning domain has a background (represented by a stratified Datalog program) which describes static predicates.
- *Type declarations*: The arguments of changeable predicates, called *fluents*, and action atoms are typed.
- *Strong and weak negation*: The DLV$^{\mathcal{K}}$ system provides two kinds of negation familiar from answer set semantics, namely weak (or default) negation "`not`" and strong (or classical) negation "¬", also denoted by "–". Weak negation allows for a simple and intuitive statement of inertia rules for fluents, or for the statement of default values for fluents in the domain.
- *Complete and incomplete states*: By default, states in DLV$^{\mathcal{K}}$ are consistent sets of ground literals, in which not every atom must appear, and thus represent *states*

of knowledge. However, by suitable constructs, DLV$^{\mathcal{K}}$ also allows for representing transitions between possible states of the world (which can be seen as states of complete knowledge).

- *Parallel/Sequential execution of actions*: Simultaneous execution of actions is possible, and in fact the default mode. All actions to be executed must qualify through an executability condition. Mutual exclusion of actions can be enforced in a sequential planning mode.
- *Secure* (*conformant*) *planning*: DLV$^{\mathcal{K}}$ is able to compute *secure plans* (often called *conformant plans* in the literature [19,42]). Informally, a plan is secure, if it is applicable starting at any legal initial state and enforces the goal, regardless of how the state evolves. Using this feature, we can also model *possible-worlds planning with an incomplete initial state*, where the initial world is only partially known, and we are looking for a plan reaching the desired goal from every possible world according to the initial state.

Main contributions. The main contributions of the present paper are the following:

(1) We reduce planning in $\mathcal{K}$ to answer set programming by means of an efficient transformation. Using this transformation, a planning problem in $\mathcal{K}$ is translated into an "equivalent" disjunctive logic program, which is then submitted to DLV for evaluation. The solutions of the original planning problem are obtained from the answer sets produced by DLV, which correspond to the optimistic plans. The use of disjunctive rules in the transformation, which we use for natural problem modeling, can be easily eliminated by using unstratified negation instead, and thus an adapted transformation can be implemented on systems such as Smodels [35].

(2) We discuss the issue of secure planning, alias *conformant planning* and its realization in the DLV$^{\mathcal{K}}$ system. Briefly, the system imposes a "security check" on optimistic plans in order to assess whether a plan is secure or not, which is transformed to a nested call to DLV itself. By the foundational results in [6], finding a secure plan is a Σ_3^P-hard[1] problem, and such a two-step approach for secure planning (that is, first find an optimistic plan and then check its security) is mandatory under polynomial reductions to answer set programming, since DLV can only solve problems with complexity in Σ_2^P with polynomial overhead.

(3) We compare DLV$^{\mathcal{K}}$ with the following state-of-the-art (conformant) planning systems: CCALC [30,31], CMBP [4], CPlan [10,15], GPT [3], and SGP [47].
In particular, we first provide an overview of these systems comparing their main features. We then consider a number of benchmark problems, namely problems in the blocksworld and "bomb in the toilet" domains, and discuss their encodings in the different systems from the viewpoint of knowledge representation. Having conducted extensive experimentation, we report the execution times of the systems on a number of

[1] We use the common notion where Σ_2^P describes the class of problems solvable in polynomial time by a nondeterministic Turing machine using an NP oracle, whereas Σ_3^P is the respectively problem class solvable polynomially by a nondeterministic Turing machine using a Σ_2^P oracle, and so on (cf. [36]).

planning-problem instances and compare the performance of the systems. As it turns out, thanks to the power of knowledge-state problem encodings DLV$^{\mathcal{K}}$ can compete even with special purpose conformant planning systems in the experiments, and it often supplies a more elegant and succinct representation of the planning problems. This may be taken as promising evidence for the potential usefulness of knowledge-state problem encodings for conformant planning.

To the best of our knowledge, DLV$^{\mathcal{K}}$ is the first declarative logic-programming based planning system which allows solving Σ_2^P-hard planning problems like planning under incomplete initial states.

The remainder of this paper is organized as follows: In the next section, we introduce the DLV$^{\mathcal{K}}$ planning system at the user and system architecture levels. After that, we turn to the technical realization of DLV$^{\mathcal{K}}$, and discuss in Sections 3 and 4 the transformation of DLV$^{\mathcal{K}}$ planning problems to answer set programs, where the former section is devoted to optimistic planning and the latter considers secure planning. After that, we compare the DLV$^{\mathcal{K}}$ planning system to a number of other planning systems. Section 6 discusses further related work and presents an outlook to ongoing and future work.

In order to alleviate reading, relevant definitions and notation from the foundational Part I [6] are provided in Appendix A of the present paper.

2. The planning system DLV$^{\mathcal{K}}$

In this section, we describe the DLV$^{\mathcal{K}}$ planning system, which provides an implementation of the language $\mathcal{K}$ as a front-end of the DLV system [7,9]. We first describe how planning problems are specified in DLV$^{\mathcal{K}}$, followed by the architecture of the system, and finally briefly the usage of DLV$^{\mathcal{K}}$. In order not to be abundant, we shall restrict ourselves to a short exposition in which we focus on the essential facts. Further information can be found in the foundational paper [6] or on the DLV$^{\mathcal{K}}$ web page <URL:http://www.dbai.tuwien.ac.at/proj/dlv/K/>.

2.1. Planning problems in DLV$^{\mathcal{K}}$

In this section, we describe how planning problems can be represented as "programs" in the DLV$^{\mathcal{K}}$ system. For this purpose, we shall consider an example in the well-known blocksworld domain. DLV$^{\mathcal{K}}$ programs are built using statements of the language $\mathcal{K}$, plus further optional control statements. We shall not exhaustively repeat all details of $\mathcal{K}$ here, and in particular we shall not formally define the semantics of $\mathcal{K}$. The details and the formal definition of the semantics of $\mathcal{K}$, which we include in abbreviated form in Appendix A, can be found in [6].

A *planning problem* is a pair $\mathcal{P} = \langle PD, q\rangle$ of a *planning domain* (informally, the world of discourse) *PD* and a query q, which specifies the goal. A planning problem is represented as a combination of a *background knowledge Π*, which is a stratified Datalog program (cf. Section 3.1), and a *program* of the following general form:

$$\begin{array}{ll}\texttt{fluents:} & F_D\\ \texttt{actions:} & A_D\\ \texttt{initially:} & I_R\\ \texttt{always:} & C_R\\ \texttt{goal:} & q\end{array}$$

where the sections `fluents` through `always` are optional and may be omitted. They consist of statements, described below, each of which is terminated by "`.`". Together with the background knowledge Π, they specify a $\mathcal{K}$ planning domain of form $PD = \langle \Pi, \langle D, R\rangle\rangle$ (see Appendix A), where the declarations D are given by F_D and $A_D$and the rules R by I_R and C_R.

The statements in F_D and A_D are fluent and action declarations, respectively, which type the fluents and actions with respect to the (static) background predicates. They have the form

$$p(X_1, \ldots, X_n)\ \texttt{requires}\ t_1, \ldots, t_m \tag{1}$$

where p is a fluent or action predicate of arity $n \geqslant 0$, and the t_i are classical literals, i.e., an atom α or its negation $\neg\alpha$ (also denoted $-\alpha$), over the predicates from the background knowledge, such that every variable X_i occurs in $t_1, \ldots, t_m$ (as common, upper case letters denote variables). Only ground instances of fluents and actions which are "supported" by some ground instance of a declaration, i.e., the `requires` part is true, need to be considered.

The `initially`-section specifies conditions that hold in an initial state (note that, in general, the initial state may not be unique). They have the form of causal rules, which are described next, without the `after` part.

The `always`-section specifies the dynamics of the planning domain in terms of causation rules of the form

$$\begin{aligned}\texttt{caused}\ f\ \ &\texttt{if}\ b_1, \ldots, b_k,\ \texttt{not}\ b_{k+1}, \ldots,\ \texttt{not}\ b_l\\ &\texttt{after}\ a_1, \ldots, a_m,\ \texttt{not}\ a_{m+1}, \ldots,\ \texttt{not}\ a_n\end{aligned} \tag{2}$$

where f is either a classical literal over a fluent or `false` (representing absurdity), the b_i's are classical literals over fluents and background predicates, and the a_j's are positive action atoms or classical literals over fluent and background predicates. Informally, the rule (2) states that f is true in the new state reached by executing (simultaneously) some actions, provided that the condition of the `after` part is true with respect to the old state and the actions executed on it, and the condition of the `if` part is true in the new state.

Both the `if`- and `after`-parts are optional. Specifically, both can be omitted together with the `caused`-keyword to represent simple facts.

The `always`-section also contains *executability conditions* for actions, i.e., expressions of the form

$$\texttt{executable}\ a\ \texttt{if}\ b_1, \ldots, b_k,\ \texttt{not}\ b_{k+1}, \ldots,\ \texttt{not}\ b_l \tag{3}$$

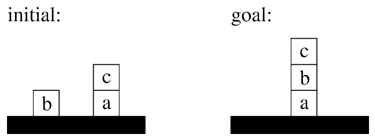


Fig. 1. Sussman's blocksworld planning problem.

```
fluents:     on(B, L) requires block(B), location(L).
             occupied(B) requires location(B).
actions:     move(B, L) requires block(B), location(L).
initially:   on(a, table). on(b, table). on(c, a).
always:      caused occupied(B) if on(B1, B), block(B).
             executable move(B, L) if B <> L.
             nonexecutable move(B, L) if occupied(B).
             nonexecutable move(B, L) if occupied(L).
             noConcurrency.
             caused on(B, L) after move(B, L).
             caused -on(B, L1) after move(B, L), on(B, L1), L <> L1.
             inertial on(B, L).
goal:        on(c, b), on(b, a), on(a, table) ? (3)
```

Fig. 2. DLV$^{\mathcal{K}}$ program for Sussman's problem in the blocksworld domain PD_{bw}.

where a is an action atom and $b_1, \ldots, b_l$ are classical literals. Informally, such a condition says that a (well-typed) action is eligible for execution in a state, if $b_1, \ldots, b_k$ are known to hold while $b_{k+1}, \ldots, b_l$ are not known to hold in that state.

The `goal`-section, finally, specifies the goal to be reached, and has the form

$$g_1, \ldots, g_m, \texttt{not}\ g_{m+1}, \ldots, \texttt{not}\ g_n\ ?\ (i) \qquad (4)$$

where $g_1, \ldots, g_n$ are ground fluent literals, $n \geqslant m \geqslant 0$, and $i \geqslant 0$ is the number of steps in which the plan must reach the goal.

All rules in I_R and C_R have to satisfy the *safety requirement* for default negated type literals,[2] i.e., each variable occurring in a default negated type literal has to occur in at least one non-negated type literal or dynamic literal. Note that this safety restriction does not apply to action and fluent literals whose variables are already range restricted by the respective declarations.

Example 2.1 (*Sussman's blocksworld planning problem*). An example of a DLV$^{\mathcal{K}}$ program is given in Fig. 2. It represents Sussman's famous planning problem in the blocksworld domain [44], depicted in Fig. 1, by which he showed anomalous behavior of STRIPS planning.

The blocksworld planning domain PD_{bw} involves distinguishable blocks and a table. Blocks and the table can serve as locations on which other blocks can be put (a block

[2] These are literals corresponding to predicates defined in the background knowledge.

can hold at most one other block, while the table can hold arbitrarily many blocks). The background knowledge Π_{bw} thus has predicates `block` and `location` defined as follows:

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

In the DLV$^{\mathcal{K}}$ program, two fluents are declared for representing states: `on(B,L)`, which states that some block `B` resides on some location `L`, and `occupied(L)`, which is true for a location `L`, if its capacity of holding blocks is exhausted. Furthermore, there is a single action predicate `move(B,L)`, which represents moving a block `B` to some location `L` (and implicitly removes that block from its previous location).

With this fluent and action repertoire, we can describe the initial state and the causal rules as well as executability conditions guarding state transitions. As for the initial state, the configuration of blocks shown on the left in Fig. 1 is expressed by the three facts `on(a,table)`, `on(b,table)`, and `on(c,a)`. Note that only positive facts are stated for `on`; nevertheless the initial state is unique because the fluent `on` is interpreted under the closed world assumption (CWA) [40], i.e., if `on(B,L)` does not hold, we assume that it is false.

The values of the fluent `occupied` in the initial state are not specified explicitly, rather they are obtained from a general rule that applies to all states, and thus is part of the `always`-section of the program (the first rule there). It says that a block `B` is occupied if something (`B1`) is on it. Note that the rule does not apply to `B` $=$ `table`, since the table is supposed to have unlimited capacity. Furthermore, `B1` must be a block, by the declaration of the fluent `on`.

Next we specify when an action `move(B,L)` is executable. The first condition states that this is possible if the block `B` and the target location `L` are distinct (a block cannot be moved onto itself). The two negative conditions `nonexecutable`... state that the move is not executable if either the block `B` or the target location `L` is occupied, respectively. These statements are shorthand macros for causation rules which interdict the execution of an action (see Appendix A.3). Thus, the `move` is executable, if the positive condition holds and both negative conditions fail.

In the standard blocksworld setting, only one block can be moved at a time. Another macro, `noConcurrency`, enforces this. This macro is convenient for computing *sequential plans*, i.e., plans under mutual exclusion of parallel actions.

The effects of a move action are defined by two dynamic rules. The first states that a moved block is on the target location after the move, and the second that a block is not on the location from which it was moved, provided it was moved to a different location.

The last statement in the `always`-section is an inertial statement for the fluent `on`, which is another macro (see Appendix A.3) informally expressing that the fluent should stay true, unless it explicitly becomes false in the new state.

To solve Sussman's problem, the query in the `goal`-section contains the configuration on the right side in Fig. 1, and furthermore, prescribes a plan length of 3 (which is feasible).

The semantics of planning domains is defined in terms of legal states and state transitions. Informally, a *state* is any consistent collection of ground fluent literals which respect the typing information. It is a *legal initial state*, if it satisfies all rules in the `initially`-section and the rules in the `always`-section with empty `after` part under answer set semantics (cf. Section 3.1) if causal rules are read as logic programming rules. A *state transition* is a triple $t = \langle s, A, s' \rangle$ where s, s' are states and A is a set of legal action instances in *PD*, i.e., action instances that respect the typing information. Such t is *legal*, if the action set A is *executable w.r.t.* s, i.e., each action `a` in A is the head of a clause (3) whose body is true, and s' satisfies all causal rules (2) from the `always`-section whose `after` part is true with respect to s and A under answer set semantics.

An *optimistic plan* for a goal $g_1, \ldots, g_m,$ `not` $g_{m+1}, \ldots,$ `not` g_n is now a sequence of action sets $\langle A_1, \ldots, A_i \rangle$, $i \geqslant 0$, such that a corresponding sequence $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ of legal state transitions $\langle s_{j-1}, A_j, s_j \rangle$ exists that leads from a legal initial state s_0 to a state s_i which establishes the goal, i.e., $\{g_1, \ldots, g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$. This sequence of legal state transitions is called *trajectory*, and a *solution* to a DLV$^{\mathcal{K}}$ planning problem is an optimistic plan of length i specified in the `goal`-section (cf. (4)).

Example 2.2 (*Sussman's problem* (*continued*)). A well-known solution to Sussman's problem consists of first moving block `c` onto the table, then moving `b` on top of `a`, and finally moving `c` on top of `b`.

In the DLV$^{\mathcal{K}}$ setting, this amounts to the optimistic plan

$$\big\langle \{\mathtt{move}(\mathtt{c}, \mathtt{table})\}, \{\mathtt{move}(\mathtt{b}, \mathtt{a})\}, \{\mathtt{move}(\mathtt{c}, \mathtt{b})\} \big\rangle.$$

We omit the description of the (unique) trajectory for this plan at this point; it will be given in Section 2.3.

2.1.1. Secure planning

DLV$^{\mathcal{K}}$ has a special statement "`securePlan.`" which may be specified before the `goal`-section. It instructs the system to compute only secure plans, which are special optimistic plans. Note that `securePlan` is *not* a macro, and is, by complexity arguments, not expressible as a macro which can be expanded efficiently.

Informally, an optimistic plan $\langle A_1, \ldots, A_n \rangle$ is *secure*, if it is applicable under any evolution of the system: starting from any legal initial state s_0, the first action set A_1 ($i \geqslant 1$) can always be done (i.e., some legal transition $\langle s_0, A_1, s_1 \rangle$ exists), and for every such possible state s_1, the next action set A_2 can be done etc., and eventually, after having performed all actions, the goal is always established. Secure plans are often called *conformant plans* in the literature, and are considered in scenarios with incomplete information about initial states or nondeterministic action effects.

Example 2.3 (*Blocksworld with incomplete initial state*). Let us consider a different planning problem in the blocksworld, illustrated in Fig. 3. Here, a further block `d` is present, whose exact location is unknown, but we know that it is not on top of `c`.

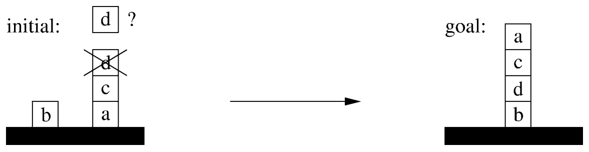


Fig. 3. A blocksworld planning problem with incomplete initial state.

The background knowledge Π_{bw} and the DLV$^{\mathcal{K}}$ program for Sussman's problem from above are modified as follows.

For introducing block `d` to the planning domain, we add the fact `block(d)` to Π_{bw} and the fact `-on(d,c)`. to the `initially`-section of the DLV$^{\mathcal{K}}$ program.

Let us first consider the necessary extensions for handling cases in which the initial state description is incorrect (e.g., when completing the partial initial state description, incorrect initial states can arise). The following conditions should hold for each block: (i) It is on top of a unique location, (ii) it does not have more than one block on top of it, and (iii) it is supported by the table (i.e., it is either on the table or on a stack of blocks which is on the table) [27].

It is straightforward to incorporate conditions (i) and (ii) into the `initially`-section:

```
initially:  forbidden on(B, L), on(B, L1), L <> L1.
            forbidden on(B1, B), on(B2, B), block(B), B1 <> B2.
```

Here, `forbidden` is a macro (cf. Section A.3) which amounts to a constraint.

For condition (iii), we introduce a fluent `supported`, which should be true for any block in a legal initial state:

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

We then describe `supported` and include a constraint that each block must be supported.

```
initially:  caused supported(B) if on(B, table).
            caused supported(B) if on(B, B1), supported(B1).
            forbidden not supported(B).
```

Now we modify the `goal`-section to

```
goal:       on(a, c), on(c, d), on(d, b), on(b, table) ? (4)
```

and, finally, to obtain a plan that works under any possible location of block `d` in the beginning, we use the `total f` macro of DLV$^{\mathcal{K}}$ (defined in Appendix A.3), which generates the two alternatives for the value of a fluent `f`:

```
initially:  total on(d, Y).
```

In this way, all completions of `on` which satisfy the initial state constraints lead to legal initial states; in fact, there are two such states, corresponding to `on(d,b)` and `on(d,table)`.

```
fluents:    on(B,L) requires block(B), location(L).
            occupied(B) requires location(B).
            supported(B) requires block(B).
actions:    move(B,L) requires block(B), location(L).
initially:  on(a,table). on(b,table). on(c,a). -on(d,c).
            total on(d,Y).
            forbidden on(B,L), on(B,L1), L <> L1.
            forbidden 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).
            forbidden not supported(B).
always:     caused occupied(B) if on(B1,B), block(B).
            executable move(B,L) if B <> L.
            nonexecutable move(B,L) if occupied(B).
            nonexecutable move(B,L) if occupied(L).
            noConcurrency.
            caused on(B,L) after move(B,L).
            caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
            inertial on(B,L).
goal:       on(a,c), on(c,d), on(d,b), on(b,table) ? (4)
```

Fig. 4. DLV$^{\mathcal{K}}$ program for a variant of Sussman's problem in an incomplete world.

The rewritten DLV$^{\mathcal{K}}$ program is depicted in Fig. 4, and using this program we are able to compute the following solution, which is a secure plan:

$$\big\langle \{\texttt{move(d,table)}\}, \{\texttt{move(d,b)}\}, \{\texttt{move(c,d)}\}, \{\texttt{move(a,c)}\} \big\rangle$$

2.1.2. Knowledge-state vs. world-state planning

Knowledge state planning in $\mathcal{K}$ offers some features which are not available in other planning languages. Recall that a knowledge state is a set of consistent fluent literals, which describes the current knowledge about the planning world. The negation as failure construct allows for expressing defeasible rules and default conclusions, by which a more natural modeling of rational planning agents which have to deal with incomplete information becomes possible at a qualitative level. In fact, a knowledge state describes more accurately the belief set of an agent about the world, which is formed by using strict and defeasible causal laws. This is in particular relevant if we are interested in "reasonable" plans for achieving a goal. However, our framework is limited to an elementary level, and does not directly allow for the representation of disjunctive knowledge.

A useful feature of knowledge-state planning is that it may allow for an elegant encoding of conformant planning problems with a world-state model in which the values of certain fluents remain open. In particular, this applies if world states are projected to subsets of fluents of interest. This supports *forgetting* information and, to some extent, *focusing* by restricting attention to those fluents whose value may have an influence on the evolution of the world depending on the actions that are taken. The advantages of a knowledge-

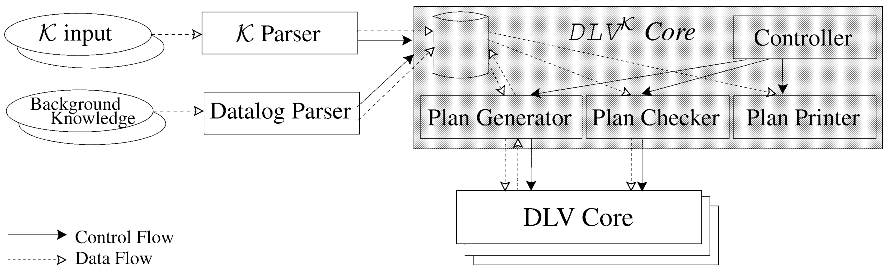


Fig. 5. DLV$^{\mathcal{K}}$ system architecture.

state encoding over a world-state encoding of the well-known “bomb in toilet” problem [34] are discussed in Appendix C. For further discussion of knowledge-state planning, see [6].

2.2. System architecture

The architecture of the DLV$^{\mathcal{K}}$ system is outlined in Fig. 5. It accepts files containing DLV$^{\mathcal{K}}$ input and background knowledge stored as plain Datalog files. Then, by means of suitable transformations from $\mathcal{K}$ to disjunctive logic programming that we will describe in Section 3, it uses the classic DLV core to solve the corresponding planning problem.

DLV$^{\mathcal{K}}$ comes with two parsers: The first accepts DLV$^{\mathcal{K}}$ files, that is, files with a filename extension of `.plan` that constitute a DLV$^{\mathcal{K}}$ program, while the second parser accepts optional background knowledge specified as stratified Datalog. Both parsers are able to read their input from an arbitrary number of files, and both convert this input to an internal representation and store it in a common database.

The actual DLV$^{\mathcal{K}}$ front-end consists of four main modules, the Controller, the Plan Generator, the Plan Checker, and the Plan Printer. The *Controller* manages the other three modules; it performs user interactions (where appropriate), and controls the execution of the entire front-end.

To that 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 as described in Section 3.1) according to the transformation $lp(\mathcal{P})$ provided in Section 3.2. The Controller then invokes the DLV kernel to solve the corresponding problem. The resulting answer sets (if any) are fed back to the Controller, which extracts the solutions to the original planning problem from these answer sets, transforms them back to the original planning domain, and saves them into the common database.

The Controller then optionally (if the user specified the `securePlan` command or invoked a secure check interactively) invokes the *Plan Checker*. Similarly to the Plan Generator, the Checker uses the original problem description together with the optimistic plan computed by the Generator to generate a disjunctive logic program that solves the problem of verifying whether this (optimistic) plan is in fact also a secure plan as intuitively introduced in Section 2.1.1 (details and the actual transformation employed by the Plan Checker will be provided in Section 4).

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

2.3. Using DLV$^{\mathcal{K}}$

DLV$^{\mathcal{K}}$ is a command-line oriented system, which is realized as front-end to the DLV logic programming system. It accepts two types of input files: (i) DLV$^{\mathcal{K}}$ files, which carry the filename extension `.plan` and contain DLV$^{\mathcal{K}}$ code as described in Section 2.1; (ii) optional background knowledge in the form of a stratified Datalog program, which is kept in files without any filename extension.

The planning front-end itself is invoked by means of the `-FP` family of command-line options: `-FP`, `-FPopt` and `-FPsec`, followed by any number of DLV$^{\mathcal{K}}$ files and files containing background knowledge.

- `-FP` invokes the DLV$^{\mathcal{K}}$ system in interactive mode, where an optimistic plan is computed and the user is then prompted whether to perform a security check for that plan and whether to compute another (optimistic) plan, respectively.
- `-FPopt` computes all optimistic plans in batch mode, without user intervention, while
- `-FPsec` computes all secure plans (applying by default secure check $\mathcal{SC}_1$, as defined in Section 4) in batch mode.

In all these cases, by means of the command-line option -n $= x$ the number of plans computed and printed can be limited to at most x; by default *all* possible plans are computed.

Further DLV$^{\mathcal{K}}$ command-line options which affect the security checking will be introduced at the end of Section 4.

As an example, assume that the DLV$^{\mathcal{K}}$ program for Sussman's blocksworld planning problem from Fig. 2 in Section 2.1 resides in a file `blocksworld.plan`, while the background knowledge about `block`s and `location`s is saved in a file `background`. Invoking

```
dlv -FP blocksworld.plan background
```

results in the following output:

```
DLV [build DEV/Dec 17 2001 gcc 2.95.3 (release)]

STATE 0: occupied(a), on(a,table), on(b,table), on(c,a)
ACTIONS: move(c,table)
STATE 1: on(a,table), on(b,table), on(c,table), -on(c,a)
ACTIONS: move(b,a)
STATE 2: occupied(a), on(a,table), on(b,a), on(c,table),
         -on(b,table)
ACTIONS: move(c,b)
```

```
STATE 3: on(a,table), on(b,a), on(c,b), -on(c,table),
         occupied(a), occupied(b)
PLAN:    move(c,table); move(b,a); move(c,b)

Check whether that plan is secure (y/n)? y
The plan is secure.

Search for other plans (y/n)? y
```

This describes a successful trajectory $\langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \langle s_2, A_3, s_3\rangle\rangle$ where $s_0, \ldots, s_3$ correspond to the lines starting with `STATE 0`, ..., `STATE 3` in the output above, and A_1, A_2, and A_3 correspond to the three `ACTIONS` lines; the entire plan is again printed at the end.

Now, let us consider the program from Fig. 4, that is, the variant of the Sussman problem with an incomplete initial state. Let us assume that we have added the fact `block(d)` to the background knowledge and modified the file blocksworld.plan accordingly. Again invoking DLV$^{\mathcal{K}}$ as above will produce the following output:

```
DLV [build DEV/Dec 17 2001 gcc 2.95.3 (release)]

STATE 0: occupied(a), on(a,table), on(b,table), on(c,a),
         -on(d,c), supported(a), supported(b),
         supported(c), -on(d,a), on(d,table), -on(d,b),
         -on(d,d), supported(d)
ACTIONS: move(c,table)
STATE 1: on(a,table), on(b,table), on(c,table),
         on(d,table), -on(c,a)
ACTIONS: move(d,b)
STATE 2: on(a,table), on(b,table), on(c,table), on(d,b),
         -on(d,table), occupied(b)
ACTIONS: move(c,d)
STATE 3: on(a,table), on(b,table), on(c,d), on(d,b),
         -on(c,table), occupied(b), occupied(d)
ACTIONS: move(a,c)
STATE 4: on(a,c), on(b,table), on(c,d), on(d,b),
         -on(a,table), occupied(b), occupied(c),
         occupied(d)
PLAN:    move(c,table); move(d,b); move(c,d); move(a,c)

Check whether that plan is secure (y/n)? y
The plan is NOT secure.

Search for other plans (y/n)? y
```

The first plan we arrive at is not secure, so we answer the question whether to search for other plans positively, and indeed find a secure plan (observe that initial states (`STATE 0`) are larger here because of the `total` statement for `on(d,Y)`):

```
STATE 0: occupied(a), on(a,table), on(b,table), on(c,a),
         -on(d,c), supported(a), supported(b),
         supported(c), -on(d,a), on(d,table), -on(d,b),
         -on(d,d), supported(d)
ACTIONS: move(d,c)
STATE 1: on(a,table), on(b,table), on(c,a), on(d,c),
         -on(d,table), occupied(a), occupied(c)
ACTIONS: move(d,b)
STATE 2: on(a,table), on(b,table), on(c,a), on(d,b),
         -on(d,c), occupied(a), occupied(b)
ACTIONS: move(c,d)
STATE 3: on(a,table), on(b,table), on(c,d), on(d,b),
         -on(c,a), occupied(b), occupied(d)
ACTIONS: move(a,c)
STATE 4: on(a,c), on(b,table), on(c,d), on(d,b),
         -on(a,table), occupied(b), occupied(c),
         occupied(d)
PLAN:    move(d,c); move(d,b); move(c,d); move(a,c)

Check whether that plan is secure (y/n)? y
The plan is secure.

Search for other plans (y/n)?
```

While looking for further secure plans, we encounter several optimistic plans, none of which is secure, so we change our strategy and invoke DLV$^{\mathcal{K}}$ with the `-FPsec` option instead of using the interactive mode enabled with `-FP`. This yield the following result:

```
DLV [build DEV/Dec 17 2001 gcc 2.95.3 (release)]
PLAN: move(d,c); move(d,b); move(c,d); move(a,c)
PLAN: move(d,table); move(d,b); move(c,d); move(a,c)
```

Indeed, while there are many optimistic plans, there is only just a single further secure plan in addition to the one we already found. Note that, as secure plans usually have many different trajectories, DLV$^{\mathcal{K}}$ only prints the plans themselves, omitting the information on states.

3. Transforming optimistic planning to answer set programming

In this section, we discuss how planning problems in DLV$^{\mathcal{K}}$ are transformed into answer set programs. We consider here optimistic planning, and deal with secure planing in the next section. As a preliminary, we first recall some concepts of (disjunctive) logic programming.

3.1. Disjunctive logic programming

We consider extended disjunctive logic programs with two kinds of negation like in the $\mathcal{K}$ language, i.e., weak negation "`not`" and strong negation "$\neg$", as introduced in [12] over a function-free first-order language. Strings starting with uppercase (respectively, lowercase) letters denote variables (respectively, constants). A *positive* (respectively *negative*) *classical literal* l is either an atom a or a negated atom $\neg a$, respectively; its *complement*, denoted $\neg l$, is $\neg a$ and a, respectively. A *positive* (respectively, *negative*) *failure* (*NAF*) *literal* ℓ is of the form l or $\texttt{not}\ l$, where l is a classical literal. Unless stated otherwise, by *literal* we mean a classical literal.

A *disjunctive rule* (*rule*, for short) R is a formula

$$a_1 \vee \ldots \vee a_n \texttt{ :- } b_1, \ldots, b_k,\ \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m, \tag{5}$$

where all a_i and b_j are classical literals and $n \geqslant 0$, $m \geqslant k \geqslant 0$. The part to the left (respectively, right) of "`:-`" is the *head* (respectively, *body*) of R, where "`:-`" is omitted if $m = 0$. We let $H(R) = \{a_1, \ldots, a_n\}$ be the set of head literals and $B(R) = B^+(R) \cup B^-(R)$ the set of body literals, where $B^+(R) = \{b_1, \ldots, b_k\}$ and $B^-(R) = \{b_{k+1}, \ldots, b_m\}$. A *constraint* is a rule with empty head ($n = 0$).

A *disjunctive logic program* (*DLP*) P (simply, *program*) is a finite set of rules. It is *positive*, if it is `not`-free (i.e., $\forall r \in P\colon\ B^-(r) = \emptyset$), and *normal*, if it is $\vee$-free (i.e., $\forall R \in P\colon\ |H(R)| \leqslant 1$). A normal program is also called a *Datalog program*. As usual, a term (atom, rule etc.) is *ground*, if no variables appear in it. A ground program is also called *propositional*.

Answer sets of DLPs are defined as consistent answer sets for EDLPs as in [12,25]. That is, for any program P, let U_P be its Herbrand universe and B_P be the Herbrand base of P over U_P (if no constant appears in P, an arbitrary constant is added to U_P). Let $\mathit{ground}(P) = \bigcup_{R \in P} \mathit{ground}(R)$ denote the grounding of P, where $\mathit{ground}(R)$ is the set of all ground instances of R.

Then, an *interpretation* is any set $I \subseteq B_P$ of ground literals. An *answer set* of a positive ground program P is any consistent interpretation I, i.e., $I \cap \{\neg l \mid l \in I\} = \emptyset$, such that I is the least (w.r.t. set inclusion) set closed under the rules of P, i.e., $B(R) \subseteq I$ implies $H(R) \cap I \neq \emptyset$ for every $R \in P$.[3] An interpretation I is an answer set of an arbitrary ground program P, if it is an answer set of the *Gelfond–Lifschitz reduct* P^I, i.e., the program obtained from P by deleting

[3] We only consider *consistent answer sets*, while in [12,25] also the (inconsistent) set B_P may be an answer set. Technically, we assume that negative classical literals $\neg a$ are viewed as new atoms $-a$, and constraints $\texttt{:-}\, a, -a$ are implicitly added. This is the standard way how true negation is implemented in systems like DLV or Smodels.

- all rules $R \in P$ such that $B^-(R) \cap I \neq \emptyset$, and
- all negative body literals from the remaining rules.

The answer sets of a non-ground program P are those of its ground instantiation $\mathit{ground}(P)$. We shall denote by $\mathcal{AS}(P)$ the set of all answer sets of any program P.

3.2. Transformation $lp(\mathcal{P})$

The main building-block underlying the realization of the DLV$^{\mathcal{K}}$ system is the translation of a DLV$^{\mathcal{K}}$ planning problem $\mathcal{P}$, given by a background knowledge Π and a DLV$^{\mathcal{K}}$ program as in Section 2.1, into a logic program $lp(\mathcal{P})$, whose answer sets represent the optimistic plans of $\mathcal{P}$. For the sake of our translation, we extend fluent and action literals by a timestamp parameter T such that an answer set AS of the translated program $lp(\mathcal{P})$ corresponds to a successful trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$ of $\mathcal{P}$ in the following sense:

- The fluent literals in $\mathcal{AS}$ having timestamp 0 represent a (legal) initial state s_0 of T.
- The fluent literals in $\mathcal{AS}$ having timestamp $i > 0$ represent the state s_i obtained after executing i many action sets (i.e., they represent the evolution after i steps).
- The action literals in $\mathcal{AS}$ having timestamp i represent the actions in A_{i+1} (i.e., those actions which are executed at step $i + 1$).

Moreover, trajectories encoded in the answer sets of $lp(\mathcal{P})$ are guaranteed to establish the goal of the planning problem, and the underlying sequence of action sets is therefore an optimistic plan.

In the following, we incrementally describe a transformation from a planning problem $\mathcal{P}$ to a logic program $lp(\mathcal{P})$. We will illustrate this transformation on the blocksworld planning problem from Section 2.1.

In what follows, let σ^{act}, σ^{fl}, and σ^{typ} be the sets of action, fluent and type names, respectively, and let $\mathcal{L}_{act}$, $\mathcal{L}^+_{act}$, $\mathcal{L}_{fl}$, and $\mathcal{L}_{typ}$ be the set of all action, positive action, fluent, and type literals, respectively. Furthermore, $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$ and $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup \mathcal{L}^+_{act}$ (*dyn* stands for *dynamic literals*).

Step 0 (*Macro expansion*). In a preliminary step, replace all macros in the DLV$^{\mathcal{K}}$-program by their definitions (cf. Appendix A.3).

Example. In the encoding of Sussman's problem, among others the macros

```
always: nonexecutable move(B,L) if occupied(B).
        inertial on(B,L).
```

are replaced by

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

Step 1 (*Background knowledge*). The background knowledge Π is already given as a logic program; all the rules in Π can be directly included in $lp(\mathcal{P})$, without further modification.

Step 2 (*Auxiliary predicates*). To represent steps, we add the following facts to $lp(\mathcal{P})$

$$\texttt{time}(0).,\ldots,\texttt{time}(i).$$

$$\texttt{next}(0,1).,\ldots,\texttt{next}(i-1,i).$$

where i is the plan length of the query $q = G?(i) \in \mathcal{P}$ at hand.

The predicate `time` denotes all possible timestamps and the predicate `next` describes a successor relation over the timestamps in our program.

Note that we refrain from using built-in predicates of a particular logic programming engine here. In the DLV$^{\mathcal{K}}$ implementation, all above auxiliary predicates are efficiently handled in a preprocessing step.

Example. For Sussman's problem, where $q = \texttt{on}(\texttt{c},\texttt{b}),\ \texttt{on}(\texttt{b},\texttt{a}),\ \texttt{on}(\texttt{a},\texttt{table})?\,(3)$, we add the following facts:

```
time(0).   time(1).   time(2).   time(3).
next(0,1).   next(1,2).   next(2,3).
```

Step 3 (*Causation rules*). For each causation rule r:`caused` H `if` B `after` A in C_R, we include a rule r' into $lp(\mathcal{P})$ as follows:

$$\mathrm{h}(r') = \begin{cases} \emptyset, & \text{if } H = \texttt{false}, \\ f(\bar{t}, T_1), & \text{if } H = f(\bar{t}),\ f \in \sigma^{fl}, \end{cases}$$

where T_1 is a new variable. To the body of r', we add the following literals:

- each default type literal in r, i.e., $(\texttt{not})l \in A \cup B$ where $l \in \mathcal{L}_{typ}$;
- $(\texttt{not})\,b(\bar{t}, T_1)$, where $(\texttt{not})\,b(\bar{t}) \in B$ and $b(\bar{t}) \in \mathcal{L}_{fl}$;
- $(\texttt{not})\,b(\bar{t}, T_0)$, where $(\texttt{not})\,b(\bar{t}) \in A$ and $b(\bar{t}) \in \mathcal{L}_{dyn}$, where T_0 is a new variable;
- for timing, we add
  – $\texttt{time}(T_1)$, if A is empty;
  – $\texttt{next}(T_0,T_1)$, otherwise.
- To respect typing declarations and to establish standard safety of r', we add for any action/fluent literal in H and default negated fluents/actions literals typing information from the corresponding action/fluent declarations. That is, if $H = (\neg)p(\bar{t})$ respectively $\texttt{not}\ (\neg)p(\bar{t}) \in A \cup B$ such that $p(\bar{t}) \in \mathcal{L}_{dyn}$,

  $$p(\overline{Y})\ \texttt{requires}\ t_1(\overline{Y_1}),\ldots,t_m(\overline{Y_m})$$

  is an action/fluent declaration (standardized apart), and θ is a substitution such that $\theta(\overline{Y}) = \bar{t}$, then we add $\theta(t_1(\overline{Y_1})),\ldots,\theta(t_m(\overline{Y_m}))$ to the body for r'. If p has multiple action/fluent declarations, each of them is considered separately, which gives rise to multiple typed versions of r'.

Example. In our encoding of Sussman's problem, the statement

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

leads to the following rule in $lp(\mathcal{P})$:

```
occupied(B,T1):- on(B1,B,T1), block(B), location(B), time(T1).
```

Here, the timing atom `time(T1)` is added, and the type information `location(B)` for the fluent `occupied(B)` in the H-part of the statement.

Step 4 (*Executability conditions*). For each executability condition e of the form `executable` $a(\bar{t})$ `if` B in C_R, we introduce a rule e' in $lp(\mathcal{P})$ as follows:

$$\mathrm{h}(e') = a(\bar{t}, T_0) \vee \neg a(\bar{t}, T_0),$$

where T_0 is a new variable. To the body of e', we add the following literals:

- Each default type literal in e, i.e., (`not`)$l \in B$ where $l \in \mathcal{L}_{typ}$;
- (`not`) $b(\bar{t}, T_0)$, where (`not`) $b(\bar{t}) \in B$ and $l \in \mathcal{L}_{dyn}$;
- `next(`T_0,T_1`)` where T_1 is a new variable;
- for typing and safety, type information literals for $a(\bar{t})$ and every default literal `not` $(\neg)p(\vec{t}) \in B$ such that $p(\vec{t}) \in \mathcal{L}_{dyn}$, similar as in Step 3 (which may lead to multiple rules e').

Example. In our running example, the condition

```
executable move(B,L) if B <> L.
```

introduces in $lp(\mathcal{P})$ the rule

```
move(B,L,T0) v -move(B,L,T0):- B <> L, block(B),
                                location(L), next(T0,T1).
```

Here, type information `block(B)`, `location(L)` is added for `move(B,L)`.

Step 5 (*Initial state constraints*). Initial state constraints in $I_{R'}$ are transformed like static causation rules r in Step 3 (i.e., A is empty), but we use the constant 0 instead of the variable T_1 and omit the literal `time(0)`.

Example. The facts in I_R:

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

become:

```
on(a,table,0). on(b,table,0). on(c,a,0).
```

Step 6 (*Goal query*). Finally, the query q:

$$\texttt{goal}:\quad g_1(\overline{t_1}),\ldots,g_m(\overline{t_m}),\texttt{not}\ g_{m+1}(\overline{t_{m+1}}),\ldots,\texttt{not}\ g_n(\overline{t_n})\ \ ?\ \ (i).$$

is translated to:

$$\begin{array}{ll}\texttt{goal\_reached}\texttt{:-} & g_1(\overline{t_1},i),\ldots,g_m(\overline{t_m},i),\ \texttt{not}\ g_{m+1}(\overline{t_{m+1}},i),\ldots,\texttt{not}\ g_n(\overline{t_n},i).\\ \texttt{:-} & \texttt{not}\ \texttt{goal\_reached}.\end{array}$$

where `goal_reached` is a new predicate symbol.

Example. $q = \texttt{on}(\texttt{c},\texttt{b}),\ \texttt{on}(\texttt{b},\texttt{a}),\ \texttt{on}(\texttt{a},\texttt{table})\ ?\ (3)$, the goal for Sussman's problem, leads to the following rules in $lp(\mathcal{P})$:

$$\begin{array}{ll}\texttt{goal\_reached}\texttt{:-} & \texttt{on}(\texttt{c},\texttt{b},3),\ \texttt{on}(\texttt{b},\texttt{a},3),\ \texttt{on}(\texttt{a},\texttt{table},3).\\ \texttt{:-} & \texttt{not}\ \texttt{goal\_reached}.\end{array}$$

The complete transformation of Sussman's blocksworld problem, $\mathcal{P}_{bw}$, after expansion of all macros (see Appendix A.3), is shown in Fig. 6.

As the reader can easily verify, the above transformation employs disjunction only in Step 4 for translating executability conditions. Furthermore, negated action atoms $\neg a(\bar{t},T)$ occur only in the heads of the rules of $lp(\mathcal{P})$. Thus, the program is head-cycle-free, which is profitably exploited by the DLV engine underlying our implementation. The disjunction, which informally encodes a guess of whether the action $a(\bar{t})$ is executed or not at time

```
block(a). block(b). block(c). location(table).
location(B) :- block(B).
time(0). time(1). time(2). time(3). next(0,1). next(1,2). next(2,3).
on(a,table,0)   :- block(a), location(table).
on(b,table,0)   :- block(b), location(table).
on(c,a,0)       :- block(c), location(a).
move(B,L,T0) v -move(B,L,T0) :- B <> L, block(B),
                                location(L), next(T0,T1).
                :- move(B,L,T0), occupied(B,T0), next(T0,T1).
                :- move(B,L,T0), occupied(L,T0), next(T0,T1).
occupied(B,T1)  :- on(B1,B,T1), block(B), location(B), time(T1).
on(B,L,T1)      :- on(B,L,T0), not -on(B,L,T1), block(B), location(L),
                   next(T0,T1).
on(B,L,T1)      :- move(B,L,T0), block(B), location(L), next(T0,T1).
-on(B,L1,T1)    :- move(B,L,T0), on(B,L1), L <> L1, block(B),
                   location(L1), next(T0,T1).
                :- move(t1,t2,T0), move(t1',t2',T0).    for t1,t1' ∈ {a,b,c}
                                             t2,t2' ∈ {a,b,c,table}, (t1,t2) ≠ (t1',t2')
goal_reached    :- on(c,b,3), on(b,a,3), on(a,table,3).
                :- not goal_reached.
```

Fig. 6. Transformation of Sussman's planning problem $\mathcal{P}_{bw}$ from Section 2.1.

T_0, may equivalently be replaced by `v`-free guessing rules. The adapted transformation can then be used on engines for computing answer sets of normal programs, such as Smodels.

The following result formally states the desired correspondence between the solutions of a $\text{DLV}^{\mathcal{K}}$ planning problem $\mathcal{P}$ and the answer sets of the logic program $lp(\mathcal{P})$ obtained by following the procedure described above.

Theorem 3.1 (Answer set correspondence). *Let $\mathcal{P}$ be a planning problem, given by a background knowledge Π and a $\text{DLV}^{\mathcal{K}}$-program, and let $lp(\mathcal{P})$ the logic program generated by Steps* 0–6 *above. Define, for any consistent set of literals S, the sets $A_j^S = \{a(t) \mid a(t, j-1) \in S,\ a \in \sigma^{act}\}$ and $s_j^S = \{f(t) \mid f(t, j) \in S,\ f(t) \in \mathcal{L}_{fl}\}$, for all $j \geqslant 0$. Then,*

(i) *for each optimistic plan $P = \langle A_1, \ldots, A_i\rangle$ of $\mathcal{P}$ and witnessing trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$, there exists some answer set S of $lp(\mathcal{P})$ such that $A_j = A_j^S$ for all $j = 1, \ldots, i$ and $s_j = s_j^S$, for all $j = 0, \ldots, i$;*
(ii) *for each answer set S of $lp(\mathcal{P})$, the sequence $P = \langle A_1, \ldots, A_i\rangle$ is a solution of $\mathcal{P}$, i.e., an optimistic plan, witnessed by the trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$, where $A_j = A_j^S$ and $s_k = s_k^S$ for all $j = 1, \ldots, i$ and $k = 0, \ldots, i$.*

Proof. The proof is based on the well-known notion of *splitting* of a logic program as defined in [28]. We define the splitting sequence $U = \langle U_{BG}, U_0, \ldots, U_i, U_G\rangle = \langle BG, BG \cup S_0, \ldots, BG \cup S_0 \cup \cdots \cup S_i, BG \cup S_0 \cup \cdots \cup S_i \cup G\rangle$ of the program $P' = ground(lp(\mathcal{P}))$ as follows:

- BG is the set of type literals and `time` and `next` literals occurring in P';
- S_j, $0 \leqslant j \leqslant i$, is the set of literals in P' of the form $f(\bar{t}, j)$, where $f \in \sigma^{fl}$, and of the form $a(\bar{t}, j-1)$, where $a \in \sigma^{act}$;
- $G = \{$`goal_reached`$\}$.

By the *Splitting Sequence Theorem* of [28], P' (and thus $lp(\mathcal{P})$) has some (consistent) answer set S iff $S = X_{BG} \cup X_0 \cup \cdots \cup X_i \cup X_G$ for some solution $X = \langle X_{BG}, X_0, \ldots, X_i, X_G\rangle$ of P' w.r.t. U. We note the following facts.

- $P_{BG} = b_{U_{BG}}(P')$ (as defined in [28], intuitively the program corresponding to U_{BG}) consists of the background program and of the facts defining `time` and `next`.
- $P_0 = e_{U_{BG}}(b_{U_0}(P') \setminus b_{U_{BG}}(P'), X_{BG})$ (as defined in [28], intuitively the program corresponding to U_0) consists of rules and constraints which are translations of initial state constraints and static rules (i.e., causation rules with empty `after`), in which the argument of `time` and the last argument of the head predicates has been instantiated with 0.
- $P_j = e_{U_{j-1}}(b_{U_j}(P') \setminus b_{U_{j-1}}(P'), X_{BG} \cup X_0 \cup \cdots \cup X_{j-1})$, for $1 \leqslant j \leqslant i$ (intuitively, the program corresponding to U_j), consists of rules and constraints which are translations of causation rules and executability conditions in the `always`-section, in which the argument of `time` and the second argument of `next` is instantiated with j (thus, the

last argument in head predicates of causation rules and executability conditions is j and $j-1$, respectively).

- $P_G = e_{U_i}(b_{U_G}(P') \setminus b_{U_i}(P'),\ X_{BG} \cup X_0 \cup \cdots \cup X_i)$ (intuitively the program corresponding to U_G) consists of the rule and the constraint which were generated by Step 6.
- $X_{BG} \cup X_0 \cup \cdots \cup X_i \cup X_G$ is a consistent set iff each of the sets $X_{BG}, X_0, \ldots, X_i, X_G$ is consistent, since no literal in any of the sets $BG, S_0, \ldots, S_i, G$ exists such that its complement is contained in any other of these sets.

We now prove (i) and (ii) of the theorem.

(i) We show that for each optimistic plan a corresponding answer set S of $lp(\mathcal{P})$ as described exists. By the Splitting Sequence Theorem, we must prove that a respective solution $X = \langle X_{BG}, X_0, \ldots, X_i, X_G \rangle$ of $lp(\mathcal{P})$ exists:

X_{BG}: As s_0 is a legal initial state, the background knowledge has a consistent answer set. Thus, by definition of P_{BG}, it clearly has a consistent answer set X_{BG}.

X_0: s_0 in the witnessing trajectory must be a legal initial state, so s_0 satisfies all rules in the `initially`-section and the rules in the `always`-section with empty `after` part, under answer set semantics if causal rules are read as logic programming rules. These rules are essentially identical (modulo the `time` literals and the timestamp arguments) to P_0, so X_0 exists and $s_0 = s_0^{S,\mathcal{P}}$.

X_j: For $1 \leqslant j \leqslant i$, $\langle s_{j-1}, A_j, s_j \rangle$ must be a legal transition. We proceed inductively. A_j has to be an executable action set w.r.t. s_{j-1}, so each action $a \in A_j$ must occur in the head of an executability condition whose body is true w.r.t. s_{j-1}. There must be a corresponding clause in P_j constructed by Step 4 of the translation such that, the body is true w.r.t. X_{j-1}. If we choose X_j such that $A_j^{S,\mathcal{P}} = A_j$, then all these rules are satisfied. Each rule in P_j which has an action literal l_a in the head such that l_a is not in A_j either does not have a true body in X_{j-1}, or we include its negation $\neg l_a$ into X_j.

Furthermore, s_j satisfies all causal rules from the `always`-section whose `after` part is true w.r.t. s_{j-1} and A_j under answer set semantics. From the correspondence of causal rules from the `always`-section and rules in P_j, we may thus conclude that P_j has an answer set X_j s.t. $s_j = s_j^{S,\mathcal{P}}$ and $A_j = A_j^{S,\mathcal{P}}$, as seen above.

X_G: s_i satisfies the goal of $\mathcal{P}$. Let $g_1, \ldots, g_m,$ `not` $g_{m+1}, \ldots,$ `not` $g_n?(i)$ be the goal of $\mathcal{P}$. Then $\{g_1, \ldots, g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$ hold. Since $s_i = s_i^{X_i,\mathcal{P}}$, the body of the rule generated in Step 6 is true and therefore $X_G = \{$`goal_reached`$\}$ exists.

In total, we have shown that for each optimistic plan of $\mathcal{P}$ a corresponding answer set S of $lp(\mathcal{P})$ exists, which contains literals representing a witnessing trajectory.

(ii) We must prove that for each answer set S of $lp(\mathcal{P})$, a corresponding optimistic plan of $\mathcal{P}$ exists. By the Splitting Sequence Theorem, a solution $X = \langle X_{BG},\ X_0, \ldots, X_i, X_G \rangle$

exists for $lp(\mathcal{P})$. Since X_0 is an answer set for the program corresponding to initial state constraints and static rules, a legal initial state $s_0 = s_0^{S,\mathcal{P}}$ must exist as well.

For X_j, $1 \leqslant j \leqslant i$, we proceed inductively. All rules corresponding to executability conditions w.r.t. time $j-1$ must be satisfied, so for every literal l_a in X_j which corresponds to a positive action literal, a rule in P_j whose body is true w.r.t. X_{j-1} and in which l_a is the only true head literal, must exist. By construction, an executability condition whose body is true w.r.t. s_{j-1} and whose head is the corresponding action literal, must exist in $\mathcal{P}$, so an executable action set $A_j = A_j^{S,\mathcal{P}}$ exists w.r.t. s_{j-1}.

All rules in P_j corresponding to causal rules from the `always`-section of $\mathcal{P}$ must be satisfied by X_j and X_{j-1} (for literals translated from `after`-parts). So for each causal rule in $\mathcal{P}$, either its `after`-part is false w.r.t. A_j and s_{j-1}, or the causal rule is satisfied by the state $s_j = s_j^{S,\mathcal{P}}$.

Finally, since X_G exists, `goal_reached` must be true. Hence, the body of the rule generated in Step 6 must be true, and therefore s_i must establish the goal of $\mathcal{P}$.

In total, we have shown that for each answer set S of $lp(\mathcal{P})$ an optimistic plan of $\mathcal{P}$ exists, such that the witnessing trajectory can be constructed from S as described. □

4. Secure planning

The translation in the previous section results in logic programs where the projections of the answer sets on the positive actions correspond to optimistic plans.

As we have already mentioned above, the DLV$^{\mathcal{K}}$ system also provides the functionality of checking whether a given optimistic plan is secure for certain planning domains. Thus, a secure plan may be found in two steps as follows: (i) Find an optimistic plan P, and (ii) check whether P is secure. The test (ii) informally amounts to testing the following three conditions:

(1) the actions of P are executable in the respective stages of the execution;
(2) at any stage, executing the respective actions of P always leads to some legal successor state; and
(3) the goal is true in every possible state reached if all steps of the plan are successfully executed.

In arbitrary planning domains, the security check is Π_2^P-complete [6],[4] and thus, by widely believed complexity hypotheses, it is not polynomially reducible to a SAT solver or other computational logic system with expressiveness bounded by NP or co-NP. However, as shown in [6], a polynomial reduction is possible for the class of proper propositional planning problems, where a planning problem $\mathcal{P}$ is *proper*, if the underlying planning domain *PD* is proper, i.e., given any state s and any set of actions A, deciding whether some legal state transition $\langle s, A, s' \rangle$ exists is possible in polynomial time.

[4] I.e., co-Σ_2^P-complete (cf. [36]).

In the DLV$^{\mathcal{K}}$ system, we have focused on proper propositional planning domains, and we have implemented security checking by a polynomial reduction to logic programs with complexity in co-NP.

Note that for a proper planning domain *PD*, there is an algorithm $\mathcal{A}_{PD}$ which, given an arbitrary state s and a set of actions A, decides in polynomial time whether some legal transition $\langle s, A, s'\rangle$ exists. The existence of such an algorithm $\mathcal{A}_{PD}$, given *PD*, is difficult to decide, even in the propositional case, and $\mathcal{A}_{PD}$ is not efficiently constructible under widely accepted complexity beliefs. We thus looked for suitable semantic properties of planning domains which can be ensured by syntactic conditions and enable a simple (or even trivial) check for the existence of a legal transition $\langle s, A, s'\rangle$, which uniformly works for a class of accepted planning domains.

4.1. `false`-*committed domains and security check* $\mathcal{SC}_1$

One such condition is when, informally, the existence of a legal transition $\langle s, A, s'\rangle$ can only be blocked by a causal rule with head `false` or by an (implicit) consistency constraint `:-f, -f`. That is, if such constraints are disregarded, some legal transition $\langle s, A, s'\rangle$ always exists, otherwise, if some constraint is violated in any such s', then no legal transition $\langle s, A, s'\rangle$ exists. This condition can be ensured by a syntactic condition which employs stratification on the causation rules.

With this in mind, we develop a security check $\mathcal{SC}_1$, which, given an optimistic plan $P = \langle A_1, \dots, A_n\rangle$ of length $n \geqslant 0$ for a planning problem $\mathcal{P}$, rewrites the logic program $lp(\mathcal{P})$ in Section 3 to a logic program $\Pi_1(\mathcal{P}, P)$ and returns "yes" if this program has no answer set, and "no" otherwise.

The modifications are as follows:

• In order to check condition (1) mentioned at the beginning of this section, the rules resulting from executability conditions are removed from $lp(\mathcal{P})$. Instead, for an executability condition of the form

$$\texttt{executable}\, a(\overline{X})\ \texttt{if}\ B_1(\overline{Y_1}), \dots, B_m(\overline{Y_m}), \texttt{not}\ B_{m+1}(\overline{Y_{m+1}}), \dots, \texttt{not}\ B_n(\overline{Y_n})$$

in $\mathcal{P}$, we generate the following rule for each $a(\bar{c}) \in A_{j+1}$ $(j = 0, \dots, n-1)$ of P, where σ is the substitution mapping the variables $\overline{X}$ to $\bar{c}$:

$$\sigma\big(a(\overline{X}, j)\ \texttt{:-}\ B_1(\overline{Y_1}, j), \dots, B_m(\overline{Y_m}, j),\\ \texttt{not}\ B_{m+1}(\overline{Y_{m+1}}, j), \dots, \texttt{not}\ B_n(\overline{Y_n}, j), \texttt{next}(j, j+1)\big).$$

This enforces that whenever an action $a(\bar{c})$ in the plan P is executable in the respective state j, then $a(\bar{c}, j)$ will be derived by $\Pi_1(\mathcal{P}, P)$; no further actions will be derived. To guarantee that the actions of the plan P are always executable, we add a rule

$$\texttt{notex :- not}\ a(\bar{c}, j).$$

for each $a(\bar{c}) \in A_{j+1}$. Here, `notex` is a new auxiliary predicate which intuitively expresses that the plan P cannot be properly executed; its truth allows building a witness for the insecurity of P.

• Concerning condition (2), in any situation where a causal rule with head `false` is violated or a fluent inconsistency arises, an answer set witnessing the insecurity of P should be generated. To this end, the transformation is modified as follows:

Each constraint `:-` *Body*, `time`(T_1). in $lp(\mathcal{P})$ derived from a causal rule of the form `caused false...`, is rewritten to

`notex :-` *Body*, `time`$(T_1), T_1 > 0$.

`:-` $\sigma(Body)$.

where σ is a substitution mapping T_1 to 0. Observe that violation of constraints referring to an initial state does not generate a counterexample.

Each constraint `:-` *Body*, `next`(T_0, T_1). in $lp(\mathcal{P})$ which has been derived from a causal rule `caused false...`, is rewritten to

`notex :-` *Body*, `next`(T_0, T_1).

And for each fluent $f(\overline{X})$, the (implicit) consistency constraint (discussed in Footnote 2) is transformed to a rule for non-initial states

`notex :-` $f(\overline{X}, T_1), -f(\overline{X}, T_1)$, `time`$(T_1), T_1 > 0$.

while those for the initial state remain unchanged:

`:-` $f(\overline{X}, 0), -f(\overline{X}, 0)$.

Constraint violations (explicit or implicit) in non-initial states therefore lead to a witnessing answer set containing `notex`.

• Finally, for condition (3), the goal constraint `:- not goal_reached.` is modified to

`:- goal_reached, not notex.`

We can read the rewritten goal constraint as follows: The constraint is satisfied, and thus the plan P is not secure, if (i) either `notex` is true, which means that some action in P cannot be executed or a constraint is violated when executing the actions in P, or (ii) if `goal_reached` is false, which means that after successfully executing all actions in P, the goal is not established.

Before we can state the informal conditions, under which the security check $\mathcal{SC}_1$ works, more precisely, we need some auxiliary concepts.

Definition 4.1 (*Constraint-free, constraint- & executability-condition-free shadow*). For any planning domain $PD = \langle \Pi, \langle D, R\rangle\rangle$, let $cfs(PD)$ denote the planning domain which results from PD by dropping all causal constraints with head `false` and interpreting negative fluents as new (positive) fluents, and call it the *constraint-free shadow of PD*. Furthermore, let $cefs(PD)$ denote the planning domain derived from $cfs(PD)$ by omitting all executability conditions and adding `executable a.` for each legal action instance `a`, and call it the *constraint- and executability–condition-free shadow of PD*.

Definition 4.2 (`false`-*committed planning domains*). We call a planning domain *PD* `false`-*committed*, if the following conditions hold:

(i) If s is a legal state in *PD* and A is an action set which is executable in s w.r.t. *PD*, then either (i.1) every legal transition $\langle s, A, s'\rangle$ in *cfs*(*PD*) is also a legal transition in *PD*, or (i.2) no $\langle s, A, s''\rangle$ is a legal transition in *PD*, for all states s'' in *PD*.
(ii) For any state s and action set A in *cefs*(*PD*), there exists some legal transition $\langle s, A, s'\rangle$ in *cefs*(*PD*).

Example 4.1 (*Blocksworld with incomplete initial state* (*continued*)). Let us reconsider the blocksworld planning problem of Example 2.3. It is easily seen that our formulation of the respective planning domain, PD_{bwi}, is `false`-committed. Indeed, it contains a single occurrence of default negation `not` , via the statement `inertial on(B,L).`, which is not critical for the existence of a successor state in *cefs*(*PD*), so condition (ii) is guaranteed. As for condition (i), for each state s and action set A, which is executable in s, there is a single legal transition $\langle s, A, s'\rangle$ in $cfs(PD_{bwi})$, and thus one of the cases (i.1) and (i.2) must apply.

Consider first the optimistic plan P which is secure, as we have seen:

$$\big\langle\{\mathtt{move}(\mathtt{d},\mathtt{table})\},\{\mathtt{move}(\mathtt{d},\mathtt{b})\},\{\mathtt{move}(\mathtt{c},\mathtt{d})\},\{\mathtt{move}(\mathtt{a},\mathtt{c})\}\big\rangle.$$

Indeed, an attempt to build an answer set S of $\Pi_1(\mathcal{P}_{bwi}, P)$ fails: starting from any initial state, the actions in A_i are always executable and no constraint is violated, thus `notex` cannot be included in S. To satisfy the rewritten goal constraint `:- goal_reached,not notex.` thus `goal_reached` must not be included in S. However, as easily seen, the atoms `on(a,c,2)`, `on(c,d,2)`, `on(d,b,2)`, `on(b,table,2)` must be included in S. This, however, means that `goal_reached` has to be included in S, which is a contradiction. Thus, no answer set S exists, which means that the plan P is secure.

Let us now modify the number of steps in the goal to $i = 2$, and consider the optimistic plan P

$$\big\langle\{\mathtt{move}(\mathtt{c},\mathtt{d})\},\{\mathtt{move}(\mathtt{a},\mathtt{c})\}\big\rangle$$

In this case we can build an answer set of $\Pi_1(\mathcal{P}_{bwi}, P)$ starting from an initial state in which block `d` is on the table, by including at each stage the literals that are enforced. Then, both actions in the plan can be executed, and we end up in a state in which the goal is not satisfied. Both `goal_reached` and `notex` cannot be derived, and thus the constraint `:-goal_reached, not notex.` in $\Pi_1(\mathcal{P}_{bwi}, P)$ is satisfied, admitting an answer set which witnesses the insecurity of P. Hence, the check outputs “no”, i.e., the plan is not secure.

To show that the security check $\mathcal{SC}_1$ works properly for all `false`-committed planning domains, we need the notions of soundness and completeness for security checks.

Definition 4.3 (*Security check*). A *security check* for a class of planning domains $\mathcal{PD}$ is any algorithm which takes as input a planning problem $\mathcal{P}$ in a planning domain from the class $\mathcal{PD}$ and an optimistic plan P for $\mathcal{P}$, and outputs “yes” or “no”. A security check is *sound*, if it reports “yes” only if P is a secure plan for $\mathcal{P}$, and is *complete* if it reports “yes” in case P is a secure plan for $\mathcal{P}$.

In other words, for a sound security check only “yes” can be trusted, while for a complete security check “no” can be trusted.

Theorem 4.1. *The security check $\mathcal{SC}_1$ is sound and complete for the class of* `false`*-committed planning domains.*

Proof. We outline the proof, but omit the details. Let $P = \langle A_1, \ldots, A_n\rangle$ be an optimistic plan for a planning problem $\mathcal{P}$ in a `false`-committed planning domain *PD*.

(Soundness) Suppose that P is not secure. This means that an initial state s_0 and a trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ in *PD* (where $0 \leqslant j \leqslant n$) exist, such that one of the conditions (1)–(3) for plan security stated at the beginning of this section is violated. We then can build an answer set S of the program $\Pi_1(\mathcal{P}, P)$, in which, starting from s_0, respective literals are included which correspond to the legal transitions in T as in $lp(\mathcal{P})$. We consider the three cases:

Suppose first that condition (1) is violated, i.e., some action $a(\bar{c})$ in the action set A_j of P is not executable. Then, no rule with head $a(\bar{c}, j)$ fires, and thus we may add `notex` to S, as it can be derived from the rule `notex :- not` $a(\bar{c}, j)$. By (ii) of `false`-committedness for *PD*, we can add literals for the stages $j+1, \ldots, n$ modeling transitions in *cefs*(*PD*) to S such that we obtain an answer set of $\Pi_1(\mathcal{P}, P)$.

Suppose next that condition (2) is violated, i.e., no successor state exists. By (ii) of `false`-committedness for *PD*, we can add literals to S modeling a legal transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$ in *cfs*(*PD*), and by (i) of `false`-committedness for *PD*, `notex` will be derived, as in *PD* some rule with head `false` fires or opposite fluent literals `f`, `- -f` are in S. Using (ii) again, we can add literals for the remaining stages $j+2, \ldots, n$ modeling transitions in *cefs*(*PD*), such that we obtain an answer set S of $\Pi_1(\mathcal{P}, P)$.

Suppose finally that condition (3) is violated. That is, $j = n$ and the goal is not satisfied by s_n. Then, the rule with head `goal_reached` is not applicable, the modified goal constraint is satisfied, and an answer set S exists. Note that this also includes the case $n = 0$.

In any of these three cases, an answer set S of $\Pi_1(\mathcal{P}, P)$ exists, and $\mathcal{SC}_1(\mathcal{P}, P)$ outputs “no”.

(Completeness) Suppose $\mathcal{SC}_1(\mathcal{P}, P)$ outputs “no”, i.e., $\Pi_1(\mathcal{P}, P)$ has some answer set S. Then, either `notex` $\in S$ or `goal_reached` $\notin S$ must hold. In the former case, `notex` must be derived either (a) from some rule r : `notex :- not` $a(\bar{c}, j)$`.`, or (b) from some rule `notex :- ... time(j).` corresponding to a rewritten constraint with head `false` or a consistency constraint for strong negation. Let r be such that j is minimal. Then S encodes with respect to the stages $0, \ldots, j-1$ a trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$, such that A_{j+1} is not executable in s_j w.r.t. *PD*. In case (a), we immediately obtain that condition (1) of security is violated and hence that P is not secure. In case (b), a trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-2}, A_{j-1}, s_{j-1}\rangle\rangle$ in *PD* exists such that executing A_j in s_{j-1} w.r.t. *cfs*(*PD*) as encoded in S leads to a state s_j which violates some constraint of *PD* with head `false` or contains opposite literals. By item (i) of `false`-committedness for *PD*, we can conclude that no legal transition $\langle s_{j-1}, A_j, s_j\rangle$ exists in *PD*, which violates condition (2) of security. On the other hand, if `goal_reached` $\notin S$ while `notex` $\notin S$, then S encodes a trajectory

$T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$ w.r.t. *PD* such that in the final state s_n the goal is false, i.e., condition (3) of security is violated. That is, in all cases P is not secure. □

Now that we have introduced the class of `false`-committed planning domains, we look for syntactic conditions on planning domains which can be efficiently checked and guarantee `false`-committedness. One such condition can be obtained by imposing stratification on causation rules as follows: For any causation rule r of the form (A.2) let $lp(r)$ be the corresponding logic programming rule f `:-`$b_1, \ldots, b_k,$ `not` $b_{k+1}, \ldots,$ `not` b_l which emerges by skipping the `after`-part.

Definition 4.4 (*Stratified planning domain*). A planning domain $PD = \langle \Pi, \langle D, R\rangle\rangle$ is *stratified*, if the logic program Π_{PD} consisting of all rules $lp(r)$, where $r \in C_R$ has $\mathsf{h}(r) \neq$ `false` and a nonempty `if`-part, is stratified in the usual sense (and strongly negated atoms are treated as new atoms).

For example, the blocksworld planning domain PD_{bwi} described above is stratified.

It is easy to see that stratified planning domains are `false`-committed. Indeed, since any stratified logic program is guaranteed to have an answer set, item (ii) of Definition 4.2 holds. Furthermore, for each legal state s and action set A which is executable in s w.r.t. *PD*, there exists a single candidate state s' for a legal transition $\langle s, A, s'\rangle$ in $cfs(PD)$, which is computed by evaluating a subset of the rules of Π_{PD}; this transition is not legal in *PD* if s' violates some constraint in C_R with head `false` or introduces inconsistency. Note that stratified planning domains *PD* are proper.

Corollary 4.2. *The security check $\mathcal{SC}_1$ is sound and complete for the class of stratified planning domains.*

A possible extension of Corollary 4.2 allows for limited usage of unstratified causation rules. For example, pairs

```
inertial f.
inertial -f.
```

of positive and negative inertia rules for the same ground fluent `f`, which amount to the rules

r_f^+: `caused f if not -f after f.`
r_f^-: `caused -f if not f after -f.`

violate stratification. Nevertheless, pairwise inertia for a fluent `f` can be allowed safely, if each of the two rules together with the remainder of the planning domain is stratified. That is, we check for stratification of the two subdomains that result from the planning domain *PD* by omitting the positive and negative inertia rules for f, denoted by PD^{-f} and PD^{+f}, respectively. If both PD^{-f} and PD^{+f} are stratified, then $\mathcal{SC}_1$ is sound and complete for *PD*. This holds because in any state s, only one of the rules r_f^+ and r_f^- can be active with respect to s.

We can further extend this to multiple pairs of ground inertia rules, where combinations for positive and negative inertia rules have to be checked. We go one step further and extend it to mux-stratified planning domains, which we define next.

Two causation rules r_0, r_1 in *PD* are a *mutually exclusive pair* (*mux-pair*), if their `after`-parts are not simultaneously satisfiable in any state s and for any executable action set A w.r.t. s in *PD*.

Definition 4.5 (*Mux-stratified planning domains*). Let *PD* be a planning domain and $E = \{(r_{i,0}, r_{i,1}) \mid 1 \leqslant i \leqslant n\}$, $n \geqslant 0$, a set of mux-pairs in *PD*. Then, *PD* is called *mux-stratified* w.r.t. E, if each planning domain *PD*$'$ that results from *PD* by removing one of the rules $r_{i,0}$ and $r_{i,1}$ for all $i = 1, \ldots, n$ is stratified.

Notice that E does not necessarily contain all mux-pairs occurring in *PD*; we may even choose $E = \emptyset$, where mux-stratified coincides with stratified planning domain.

Note that E induces a bipartite graph G_E, whose vertices are the rules occurring in E and whose edges are the pairs in E. The removal sets for building *PD*$'$ which need to be considered are given by the maximal independent sets of G_E. There may be exponentially many such sets, and thus the cost for (simple) mux-stratification testing grows fast.

We now establish the following result.

Theorem 4.3. *Every planning domain PD which is mux-stratified w.r.t. some set of mux-pairs E is* `false`*-committed.*

Proof. Consider any state s and executable action set A w.r.t. s in *PD*. Denote by *active*(s, A, PD) the set of all ground rules in *ground*(Π_{PD}) which correspond to instances r' of causation rules in *PD* such that the `after`-part of r' is true w.r.t. s, A and the answer set M of the background knowledge.

Then, we claim that *active*(s, A, PD) is stratified, i.e., its (ground) dependency graph does not contain a negative cycle. Indeed, towards a contradiction assume that the (ground) dependency graph of *active*(s, A, PD) contains a negative cycle C. Then, C involves only rules which correspond to instances of causation rules not occurring in E, and rules which correspond to instances of causation rules from $r_{1,i_1}, \ldots, r_{n,i_n}$, where $E = \{(r_{i,0}, r_{i,1}) \mid 1 \leqslant i \leqslant n\}$ and $i_j \in \{0, 1\}$, for all $j = 1, \ldots, n$. (A rule R is involved in all edges $l_1 \to l_2$ of the dependency graph, where $l_1 \in H(R)$ and $l_2 \in B(R)$.) This means that C is also present in the ground dependency graph of $\Pi_{PD'}$ for some *PD*$'$ which results from *PD* by removing the causation rules $r_{1,1-1_j}, r_{1,1-2_j}, \ldots, r_{n,1-n_j}$. Consequently, the (non-ground) dependency graph of $\Pi_{PD'}$ contains a negative cycle. This, however, contradicts that *PD* is mux-stratified w.r.t. E; the claim is proved.

Since the ground program *active*(s, A, PD) is stratified, it is easily seen that conditions (i) and (ii) of `false`-committedness hold for s. Since s was arbitrary, it follows that *PD* is `false`-committed. □

By combining Theorems 4.1 and 4.3, we obtain the following corollary.

Corollary 4.4. *The security check $\mathcal{SC}_1$ is sound and complete for the class of mux-stratified planning domains, and in particular if E consists of opposite ground* `inertial`*-rules.*

The DLV$^{\mathcal{K}}$ system provides limited support for testing mux-stratification, which currently works for the set E consisting of all opposite ground inertia rules; an extension to larger classes is planned for future DLV$^{\mathcal{K}}$ releases. Notice, however, that deciding whether a given pair (r_0, r_1) is a mux-pair in a given planning domain is intractable in general.

A generalization of the result in Corollary 4.4 to sets E of non-ground opposite inertial rules fails. The reason is that in this case, multiple transition candidates $\langle s, A, s'\rangle$ exist in *cfs*(PD) in general, which correspond to multiple answer sets of the program *active*(s, A, PD). However, some of them might not be legal in *PD*, and condition (i) of `false`-committedness may be violated. Preliminary results suggest that under further restrictions, like excluding constraints and causation rules with opposite unifiable heads, $\mathcal{SC}_1$ may be applied. We leave this for further work.

4.2. Serial planning domains and security check $\mathcal{SC}_2$

Besides $\mathcal{SC}_1$, the DLV$^{\mathcal{K}}$ system provides an alternative security check $\mathcal{SC}_2$ for handling other classes of proper planning domains, and the system design easily allows the incorporation of further security checks.

The check $\mathcal{SC}_2$ is obtained by a slight modification of the program clauses in $\Pi_1(PD, P)$, resulting in a program $\Pi_2(PD, P)$ as follows: the head `notex` of each rule which stems from a causal rule r such that $\mathrm{h}(r) =$ `false` and the `if`-part is not empty, is shifted to the negative body, i.e.,

```
notex :- Body.
```

is rewritten to

```
:- Body, not notex.
```

Informally, this shift means that the violation of a constraint on the successor state s' is tolerated, and we eliminate s' as a counterexample to the security of the plan.

We will see that this check works for the following class of planning domains.

Definition 4.6 (*Serial planning domains*). A planning domain *PD* is *serial*, if it has the following properties:

(i) if s is a state in *PD* and A is executable in s w.r.t. *PD*, then some legal transition $\langle s, A, s'\rangle$ is guaranteed to exist, and, moreover,
(ii) for any state s and set of actions A in *cefs*(PD), some legal transition $\langle s, A, s'\rangle$ exists w.r.t. *cefs*(PD).

Obviously, serial planning domains *PD* are proper, as the check $\mathcal{A}_{PD}$ for telling whether a legal transition exists for s and executable A is trivial (just always return "yes"). The following can be observed:

Theorem 4.5. *The security check $\mathcal{SC}_2$ is sound and complete for serial planning domains.*

The proof of this result is similar to the proof of Theorem 4.1, and we thus omit it.

A syntactical restriction guaranteeing seriality are stratified planning domains *PD* which contain no rules r such that $\mathtt{h}(r) = \mathtt{false}$ and employ no strong negation. The serial property is preserved if we we also allow arbitrary totalization statements and limited use of strong negation, e.g., either all occurrences of a fluent are strongly negated or none is. Note that such planning domains are not `false`-committed in general.

The security check $\mathcal{SC}_2$ also works for generalizations of serial planning domains. For example, we may safely add rules r of the form `caused false after` B. Furthermore, $\mathcal{SC}_2$ may also be profitably combined with $\mathcal{SC}_1$ in order to enlarge classes for which security checking is supported.

4.3. Incomplete security checking

We may combine (fast) security checks which are sound and security checks which are complete to obtain checks which return the correct answer if possible, and leave the answer open otherwise. This is similar to the use of incomplete constraint solvers in constraint programming, which return either "yes", "no", or "unknown" if queried about satisfiability of a constraint; the obvious requirement is that the answer returned does not contradict the correct result.

Suppose that we have a suite of security checks $\mathcal{SC}_1, \ldots, \mathcal{SC}_n$, where $\mathcal{SC}_1, \ldots, \mathcal{SC}_j$, for some $j \leqslant n$, are known to be sound for a class of planning domains $\mathcal{PD}$ and $\mathcal{SC}_k, \ldots, \mathcal{SC}_n$, for some $k \leqslant n$, are known to be complete for $\mathcal{PD}$. Then, we can combine them to the following test $\mathcal{T}$:

$$\mathcal{T}(\mathcal{P}, P) = \begin{cases} \text{"yes"}, & \text{if } \mathcal{SC}_i(\mathcal{P}, P) = \text{"yes", for some } i \in \{1, \ldots, j\}; \\ \text{"no"}, & \text{if } \mathcal{SC}_i(\mathcal{P}, P) = \text{"no", for some } i \in \{k, \ldots, n\}; \\ \text{"unknown"}, & \text{otherwise.} \end{cases}$$

Observe that in the "yes" case of $\mathcal{T}$, $\mathcal{SC}_i(\mathcal{P}, P) =$ "yes" must hold for all $i \in \{k, \ldots, n\}$, and symmetrically in the "no" case that $\mathcal{SC}_i(\mathcal{P}, P) =$ "no", for all $i \in \{1, \ldots, j\}$; this can be used for checking integrity of the sound respectively complete security checks involved.

Note that we can always use a dummy complete security check which reports "yes" on every input. By merging the "unknown" case into the "no" case, we thus can combine sound security checks $\mathcal{SC}_1, \ldots, \mathcal{SC}_j$ to another, more powerful sound security check $\mathcal{SC}$ for the class $\mathcal{PD}$. In particular, if $\mathcal{SC}_1, \ldots, \mathcal{SC}_j$ are known to exhaust all secure plans, then $\mathcal{SC}$ is a sound and complete security check for $\mathcal{PD}$.

To account for the results in this section, in addition to the command-line options `-FP`, `-FPopt`, and `-FPsec` that we have seen in Section 2.3, DLV$^{\mathcal{K}}$ provides three further options controlling the security checking: `-FPcheck=`n where $n \in \{1, 2\}$ (which correspond to $\mathcal{SC}_1$ and $\mathcal{SC}_2$ in the current implementation) selects a security check, while `-FPsoundcheck=`n and `-FPcompletecheck=`n where $n \in \{1, 2\}$, as above, can be used to specify a security check known to be sound and complete, respectively, for the input domain. The incorporation of further built-in security checks and support for user-defined security checks is planned for the future.

5. Comparison and experiments

In the following, we will compare DLV$^{\mathcal{K}}$ with several state-of-the-art conformant planning systems, and report about experimental results about the performance of the system. The results presented here are mainly intended to give a momentary view on the state of the current implementation of DLV$^{\mathcal{K}}$ and its capabilities. To that end, we present extensive benchmark results, and also compare the expressive power and flexibility of the various systems.

5.1. Overview of compared systems

5.1.1. CCALC

The *Causal Calculator* (*CCALC*) is a model checker for the languages of causal theories [30]. It translates programs in the action language $\mathcal{C}$ into the language of causal theories which are in turn transformed into SAT problems using literal completion as described in [31]. This approach is based on Satisfiability Planning [22], where planning problems are reduced to SAT problems which are then solved by means of an efficient SAT solver like *SATO* [48] or *relsat* [1].

Though its input language allows nondeterminism in the initial state and also nondeterministic action effects, CCALC as such is not capable of conformant planning and only computes "optimistic plans" (according to DLV$^{\mathcal{K}}$ terminology). Plan length is fixed, and both sequential and concurrent planning are supported.

CCALC is written in Prolog. For our tests, we used version 1.90 of CCALC which we obtained from <URL:http://www.cs.utexas.edu/users/tag/cc/> and a trial version of SICStus Prolog 3.8.6; we tested the system with SATO 3.2.1 and relsat 1.1.2. On the instances SATO could solve it was significantly faster than relsat; relsat was used only for the instances SATO could not solve in our experiments.

5.1.2. CMBP

The *Conformant Model Based Planner* [4] is based on the model checking paradigm as well and relies on symbolic techniques such as BDDs. CMBP only allows sequential planning. Its input language is an extension of $\mathcal{AR}$ [16]. Unlike action languages such as $\mathcal{C}$ or $\mathcal{K}$, this language only supports propositional actions. Nondeterminism is allowed in the initial state and for action effects. The length of computed plans is always minimal, but the user has to declare an upper bound using command-line option `-pl`. If `-pl` is set equal to the minimal plan length for the specific problem, this can be used to fix the plan length in advance. We used this method to be comparable with DLV$^{\mathcal{K}}$ which currently can only deal with fixed plan length.

For our tests, we used CMBP 1.0, available at <URL:http://sra.itc.it/people/roveri/cmbp/>.

5.1.3. CPlan

Introduced in [10,15], *CPlan* is a conformant planner based on CCALC and the $\mathcal{C}$ action language [17,26,29]. This language is similar to $\mathcal{K}$ in many respects, but close to classical logic, while $\mathcal{K}$ is more "logic programming oriented" by the use default negation (see [6] for further discussion). CPlan uses CCALC only to generate a SAT instance and replaces the optional SAT-solvers used by CCALC with an own procedure that extracts conformant plans from these SAT instances. CPlan implements full conformant planning and supports the computation of both minimal length plans as well as plans of fixed length, by incrementing plan length from a given lower bound until a plan is found or a given upper bound is reached. We set the upper and lower bound equal to the minimal plan length of the specific problems for our experiments to be comparable with DLV$^{\mathcal{K}}$. Sequential and concurrent planning are possible; nondeterminism is allowed in the initial state as well as for action effects.

For our tests, we used CPlan 1.3.0, which is available at <URL:http://frege.mrg.dist.unige.it/∼otto/cplan.html>, together with CCALC 1.90 to produce the input for CPlan.

5.1.4. GPT

The *General Planning Tool* [3] employs heuristic search techniques like A^* to search the belief space. Its input language is a subset of *PDDL*. Nondeterminism is allowed in the initial state as well as for action effects. GPT only supports sequential planning and calculates plans of minimal length.

We used version GPT 1.14 obtained from <URL:http://www.cs.ucla.edu/∼bonet/software/>.

5.1.5. SGP

In addition to conformant planning, *Sensory Graphplan* (*SGP*, [47]) can also deal with sensing actions. SGP is an extension of the Graphplan algorithm [2]. Its input language is an extension of PDDL [14]. Nondeterminism is allowed only in the initial state. The program always calculates plans of minimal length.[5] SGP does not support sequential planning, but computes concurrent plans automatically recognizing mutually exclusive actions. That means, minimal length plans in terms of SGP are not plans with a minimal number of actions but with a minimal number of steps needed. At each step an arbitrary number of parallel actions are allowed, as long as the preconditions or effects are not mutually exclusive which is automatically detected by the algorithm.

SGP is written in LISP and available at <URL:http://www.cs.washington.edu/ai/sgp.html>. For our tests, we used a trial version of Allegro Common Lisp 6.0.

5.1.6. Specific system features

We would also like to point out further specific features of some of these special purpose planning systems:

- SGP automatically recognizes mutually exclusive actions in concurrent plans. It is possible to encode concurrent plans in DLV$^{\mathcal{K}}$ by explicitly describing the mutually exclusive actions, as done in our encodings of the “bomb in the toilet” benchmark problems for multiple toilets (see Section 5.2.2). However, the language $\mathcal{K}$ is more

[5] SGP comprises the functionality of another system by Smith and Weld called CGP (Conformant Graphplan, [42]), but is slower in general. As CGP is no longer maintained and not available online, we nevertheless decided to choose SGP for our experiments.

Table 1
Overview of system features

	DLV$^{\mathcal{K}}$	CCALC	CPlan	CMBP	SGP	GPT
Input language	$\mathcal{K}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{AR}$	PDDL	PDDL
Sequential plans	yes	yes	yes	yes	no	yes
Concurrent plans	yes	yes	yes	no	yes	no
Optimistic plans	yes	yes	no	no	no	no
Conformant plans	yes	no	yes	yes	yes	yes
Minimal plan length	no	no	yes	yes	yes	yes
Fixed plan length	yes	yes	yes	no[a]	no	no

[a] An upper bound can be specified, but computed plans are always minimal.

complex than PDDL, which makes automatic recognition of possible conflicts of actions much harder in our framework. On the other hand, our notions of executability and nonexecutability allow more flexible encodings of parallel actions than SGP.

- GPT and SGP always compute minimal plans, which is not possible in the current version of DLV$^{\mathcal{K}}$.
- CMBP and CPlan optionally compute minimal plans, where the user may specify upper and/or lower bounds for the plan length.

Table 1 provides a comparison of DLV$^{\mathcal{K}}$ and all the systems introduced above. Note that CCALC is not capable of conformant planning, and thus we cannot use it on the respective benchmark problems. On the other hand, CPlan showed slow performance on the deterministic planning benchmarks that we considered. Therefore, we considered these two systems in combination (CCALC for deterministic planning benchmarks and CPlan for conformant planning benchmarks).

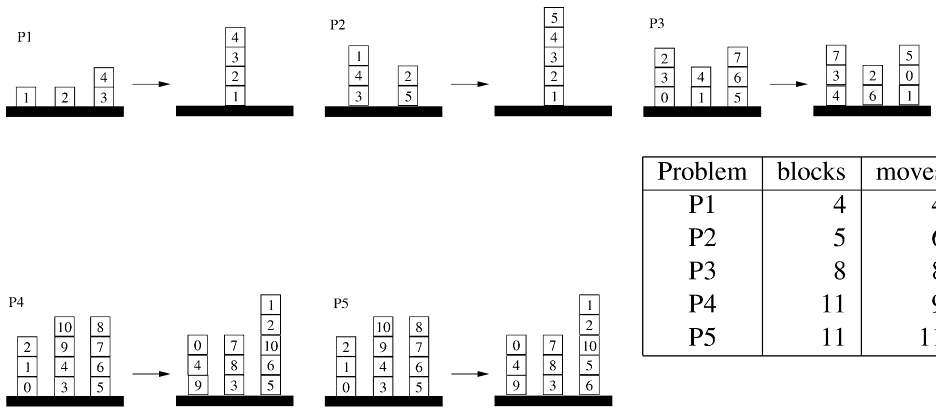
5.2. Benchmark problems and encodings

5.2.1. Blocksworld

For benchmarking we have chosen some blocks world instances to illustrate the performance of DLV$^{\mathcal{K}}$ on deterministic domains. Problems P1–P4 are due to [8], and problem P5 is a slight modification of P4, which needs two moves more. The initial configurations and the respective goal configurations of P1–P5 together with the minimum number of moves (steps) needed to solve these problems are shown in Fig. 7.

5.2.2. Bomb in the toilet

To show the capabilities of DLV$^{\mathcal{K}}$ on planning under incomplete information, and in particular conformant planning, we have chosen the well-known "bomb in the toilet" problem [34] and variations thereof, where we employ a naming convention due to [4]. The respective planning domain comprises actions with nondeterministic effects, the initial state is incomplete and, in more elaborated versions, several actions are available that can be done in parallel.



Problem	blocks	moves
P1	4	4
P2	5	6
P3	8	8
P4	11	9
P5	11	11

Fig. 7. Blocksworld planning instances.

BT(p)—Bomb in the toilet with p packages. The basic scenario of the "bomb in the toilet" problem is as follows. We have been alarmed that there is a bomb (exactly one) in a lavatory. There are p suspicious packages which could contain the bomb. There is one toilet bowl, and it is possible to dunk a package into it. If the dunked package contained the bomb, then the bomb is disarmed and a safe state is reached. The obvious goal is to reach a safe state via a secure plan.

BTC(p)—Bomb in the toilet with certain clogging. In a slightly more elaborated version, dunking a package clogs the toilet, making further dunking impossible. The toilet can be unclogged by flushing it. The toilet is assumed to be unclogged initially. Note that this domain still comprises only deterministic action effects.

BTUC(p)—Bomb in the toilet with uncertain clogging. In a further elaboration of the domain, dunking a package has a nondeterministic effect on the status of toilet, which is either clogged or not clogged afterwards.

BMTC(p,t), BMTUC(p,t)—Bomb in the toilet with multiple toilets. Yet another elaboration is that several toilet bowls ($t \geqslant 1$, rather than just one) are available in the lavatory.

5.2.3. Encodings used

As far as possible, we used the original encodings which come along with the distributions of the respective systems.

CCALC/CPlan. CCALC is not capable of conformant planning, while CPlan proved very slow on deterministic domains. Thus, for the blocksworld problems P1–P5 we used the $\mathcal{C}$ encoding provided by Esra Erdem with pure CCALC [8], while we used CPlan for the "bomb in the toilet" problems, with slight modifications of the $\mathcal{C}$ encodings provided with the current CPlan distribution.

CMBP. For CMBP, we used the “bomb in the toilet” encodings which are included in the distribution. BMTUC(p, t) is not included, but only a trivial modification of BMTC(p, t) is needed to obtain an encoding for BMTUC. Because only propositional actions are allowed in the input language of CMBP, an encoding of blocks world where many different moves are possible is quite large. As no encoding is included in the examples, a (straightforward) encoding of P1 which we used for comparison can be found in Section B.1 of Appendix B.

GPT. The distribution of GPT provides encodings for various “bomb in the toilet” problems; BMTUC(p, t) was not included, but the respective extension of BTUC(p) is trivial. For blocksworld, we used an adapted version of the SGP encoding, as the PDDL dialects of the two systems slightly differ. The encoding for P1 can be found in the appendix.

SGP. For SGP we used the blocks world and bomb in toilet encodings coming with the distribution. BTUC(p) and BMTUC(p, t) cannot be encoded in SGP which only allows nondeterminism in the initial state.

SGP generates concurrent plans, so we did not compare the sequential versions of BT(p) and BMTC(p, t). Furthermore, for the blocks world problems, this means that the minimal plan lengths differ from the $\mathcal{K}$ encodings, and we provide them in an extra column of Table 2. Note that the number of actions in the plans computed by SGP is not necessarily minimal. For example, for P3 a plan with 4 steps and 9 moves exists whereas SGP finds a plan with 4 steps and 12 moves.

DLV$^{\mathcal{K}}$. We have tested for the “bomb in the toilet” problems two different encodings in DLV$^{\mathcal{K}}$, developed in [6]. The first one, labeled ws in the results, mimics world-state planning, in which the different completions of the states (“totalizations”) to world-states are considered. The second one, labeled ks, uses the power of knowledge-state planning provided by DLV$^{\mathcal{K}}$; it does not complete the states right away, but leaves the value of unknown fluents open in accordance with the real knowledge of the planning agent about the state of affairs. In both encodings, we first consider concurrent actions and then one action at time.

Since the blocksworld problems are not conformant planning instances, we use optimistic planning for them. For the knowledge-state encoding of the “bomb in the toilet” problems, the applicability of the security check $\mathcal{SC}_1$ is straightforward even for BTUC(p) and BMTUC(p, t), as the domains are mux-stratified w.r.t. the inertia rules for `clogged` and `-clogged`, as these fluents do not occur in any bodies of other causation rules. Furthermore, thanks to the knowledge-state representation, the domains are deterministic and have unique initial states, so the security check is trivial and negligible for timing.

The world-state encodings of BT(p), BTC(p), and BMTC(p, t) are stratified, so the security check $\mathcal{SC}_1$ is guaranteed to be sound and complete for these problems by Corollary 4.2. In the case of BTUC(p) and BMTUC(p, t) in the world-state programs, the macro `total` violates stratification. However, both BTUC(p) and BMTUC(p, t) are `false`-committed domains, and thus the security check $\mathcal{SC}_1$ is sound and complete for these problems by Theorem 4.5. Indeed, the respective programs have no cycle with an odd number of negative arcs in their dependency graphs (cf. BMTUC(p, t) in

Appendix C.1; BTUC(p,t) and BMTUC(p,t) have the same dependency graph, since the only difference is that some fluents get an additional argument [6]), so by well-known results at least one answer set is guaranteed, and thus condition (ii) of `false`-committedness holds. Furthermore, the only constraints are those resulting from expanding the `nonexecutable` statements. Since these constraints refer only to actions, either all s' in a transition $\langle s, A, s'\rangle$ satisfy them or no s' does. Therefore, condition (i) of `false`-committedness is enforced as well. The world-state encodings of "bomb in the toilet" are not deterministic, so the security check is responsible for a considerable portion of the timings.

5.3. Benchmark results and discussion

In this section, we compare the various systems in terms of representation capabilities and run-time benchmarks.

5.3.1. Test environment

All tests were performed on a Pentium III 733 MHz machine with 256 MB of main memory running SuSE Linux 6.4. The results for the blocks world problems are summarized in Table 2. Tables 3–9 show the results for the various "bomb in the toilet" problems. The minimal plan length is reported in the second column of each table. Note that for CCALC the results include 1.23 s startup time for SICStus Prolog, while for SGP 0.27 s startup time is included. Run-times longer than 1200 CPU seconds were omitted, which is indicated by a dash in the tables.

5.3.2. Representation

From the viewpoint of expressiveness, the language $\mathcal{K}$ often allows a more compact and readable encoding than $\mathcal{AR}$ or *PDDL* dialects: CMBP allows only propositional actions (see Appendix B.1 for a blocks world encoding in $\mathcal{AR}$), whereas languages like $\mathcal{C}$ and $\mathcal{K}$ allow a much more elegant encoding of complex actions. PDDL dialects as used by GPT or SGP, on the other hand, do not allow expressing ramifications which makes the encoding of

Table 2
Experimental results for blocksworld problems P1–P5

Problem	Steps	Blocks	DLV$^{\mathcal{K}}$	CCALC	CMBP	GPT	SGP	
							steps/actions[a]	time
P1	4	4	0.04 s	1.73 s	0.18 s	1.13 s	3/4	9.69 s
P2	6	5	0.11 s	2.18 s	7.95 s	2.52 s	5/7	43.85 s
P3	8	8	8.81 s	5.42 s	–	–	4/12	248.45 s
P4	9	11	8.91 s	15.83 s	–	–	–	–
P5	11	11	21.14 s	350.43 s[b]	–	–	–	–

[a] As SGP supports only concurrent planning, the number of steps and number of actions for the solutions found are displayed in an extra column. Note that the number of actions is not necessarily minimal.

[b] With CCALC and SATO no solution for P5 could be found, the timing for P5 was generated using relsat, which is significantly slower on the other problem instances.

Table 3
Experimental results for BT(p) with concurrent dunks

BT(p)	Steps	DLV$^{\mathcal{K}}$		CPlan	SGP
		ws	ks		
BT(2)	1	0.01 s	0.01 s	1.38 s	0.69 s
BT(3)	1	0.02 s	0.01 s	1.38 s	0.80 s
BT(4)	1	0.01 s	0.01 s	1.39 s	0.95 s
BT(5)	1	0.02 s	0.01 s	1.42 s	1.21 s
BT(6)	1	0.02 s	0.01 s	1.47 s	1.55 s
BT(7)	1	0.02 s	0.01 s	1.56 s	2.00 s
BT(8)	1	0.02 s	0.01 s	1.79 s	2.56 s
BT(9)	1	0.01 s	0.02 s	2.29 s	3.32 s
BT(10)	1	0.02 s	0.02 s	3.41 s	4.27 s
BT(11)	1	0.02 s	0.02 s	6.04 s	5.34 s
BT(12)	1	0.02 s	0.02 s	11.98 s	6.66 s
BT(13)	1	0.03 s	0.02 s	25.28 s	8.16 s
BT(14)	1	0.03 s	0.01 s	57.71 s	9.98 s
BT(15)	1	0.03 s	0.01 s	127.75 s	12.11 s
BT(16)	1	0.03 s	0.01 s	294.44 s	14.57 s
BT(17)	1	0.03 s	0.02 s	678.19 s	17.43 s
BT(18)	1	0.03 s	0.02 s	–	20.74 s
BT(19)	1	0.03 s	0.02 s	–	24.47 s
BT(20)	1	0.04 s	0.02 s	–	28.78 s

Table 4
Experimental results for BT(p) sequential

T(p)	Steps	DLV$^{\mathcal{K}}$		CPlan	CMBP	GPT
		ws	ks			
BT(2)	2	0.02 s	0.02 s	1.37 s	0.03 s	0.56 s
BT(3)	3	0.03 s	0.02 s	1.39 s	0.04 s	0.55 s
BT(4)	4	0.11 s	0.02 s	1.39 s	0.04 s	0.61 s
BT(5)	5	1.50 s	0.03 s	1.45 s	0.04 s	0.61 s
BT(6)	6	28.78 s	0.03 s	1.81 s	0.04 s	0.63 s
BT(7)	7	593.15 s	0.03 s	5.12 s	0.05 s	0.67 s
BT(8)	8	–	0.05 s	65.85 s	0.06 s	0.68 s
BT(9)	9	–	0.06 s	–	0.07 s	0.78 s
BT(10)	10	–	0.08 s	–	0.10 s	0.95 s
BT(11)	11	–	0.10 s	–	0.19 s	1.27 s
BT(12)	12	–	0.13 s	–	0.39 s	2.12 s
BT(13)	13	–	0.16 s	–	0.82 s	3.89 s
BT(14)	14	–	0.21 s	–	1.76 s	8.87 s
BT(15)	15	–	0.28 s	–	4.00 s	19.13 s
BT(16)	16	–	0.35 s	–	8.82 s	42.17 s
BT(17)	17	–	0.47 s	–	19.03 s	93.69 s
BT(18)	18	–	0.61 s	–	38.95 s	208.00 s
BT(19)	19	–	0.78 s	–	91.89 s	496.95 s
BT(20)	20	–	0.98 s	–	199.63 s	546.43 s

Table 5
Experimental results for BTC(p)

BTC(p)	Steps	DLV$^{\mathcal{K}}$		CPlan	CMBP	GPT	SGP
		ws	ks				
BTC(2)	3	0.02 s	0.01 s	1.37 s	0.04 s	0.59 s	0.92 s
BTC(3)	5	0.08 s	0.02 s	1.39 s	0.04 s	0.60 s	3.30 s
BTC(4)	7	1.56 s	0.02 s	1.39 s	0.05 s	0.60 s	191.60 s
BTC(5)	9	36.28 s	0.03 s	2.36 s	0.05 s	0.62 s	–
BTC(6)	11	–	0.04 s	28.95 s	0.06 s	0.66 s	–
BTC(7)	13	–	0.06 s	178.97 s	0.07 s	0.68 s	–
BTC(8)	15	–	0.08 s	–	0.12 s	0.74 s	–
BTC(9)	17	–	0.11 s	–	0.21 s	0.81 s	–
BTC(10)	19	–	0.14 s	–	0.39 s	1.04 s	–
BTC(11)	21	–	0.20 s	–	0.81 s	1.48 s	–
BTC(12)	23	–	0.26 s	–	1.72 s	2.51 s	–
BTC(13)	25	–	0.34 s	–	3.79 s	4.68 s	–
BTC(14)	27	–	0.45 s	–	8.82 s	10.84 s	–
BTC(15)	29	–	0.58 s	–	16.92 s	23.31 s	–
BTC(16)	31	–	0.74 s	–	42.92 s	51.40 s	–
BTC(17)	33	–	0.94 s	–	92.03 s	114.21 s	–
BTC(18)	35	–	1.17 s	–	197.85 s	273.25 s	–
BTC(19)	38	–	1.46 s	–	–	374.00 s	–
BTC(20)	39	–	1.80 s	–	–	–	–

Table 6
Experimental results for BTUC(p)

BTUC(p)	Steps	DLV$^{\mathcal{K}}$		CPlan	CMBP	GPT
		ws	ks			
BTUC(2)	3	0.03 s	0.02 s	1.35 s	0.03 s	0.59 s
BTUC(3)	5	0.61 s	0.02 s	1.45 s	0.04 s	0.60 s
BTUC(4)	7	87.54 s	0.03 s	1.93 s	0.04 s	0.61 s
BTUC(5)	9	–	0.03 s	2.48 s	0.06 s	0.66 s
BTUC(6)	11	–	0.04 s	–	0.06 s	0.65 s
BTUC(7)	13	–	0.05 s	51.72 s	0.07 s	0.74 s
BTUC(8)	15	–	0.08 s	–	0.12 s	0.75 s
BTUC(9)	17	–	0.10 s	–	0.20 s	0.88 s
BTUC(10)	19	–	0.14 s	–	0.39 s	1.18 s
BTUC(11)	21	–	0.19 s	–	0.80 s	1.81 s
BTUC(12)	23	–	0.25 s	–	1.72 s	3.18 s
BTUC(13)	25	–	0.33 s	–	3.79 s	6.42 s
BTUC(14)	27	–	0.43 s	–	8.81 s	14.43 s
BTUC(15)	29	–	0.55 s	–	16.94 s	32.25 s
BTUC(16)	31	–	0.71 s	–	42.93 s	71.10 s
BTUC(17)	33	–	0.90 s	–	92.02 s	159.53 s
BTUC(18)	35	–	1.15 s	–	197.84 s	368.12 s
BTUC(19)	38	–	1.41 s	–	–	–
BTUC(20)	39	–	1.74 s	–	–	–

Table 7
Experimental results for BMTC(p) with concurrent dunks

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

action effects less readable and elaboration tolerant (see Appendix B.2 for a GPT encoding of blocksworld).

Similar remarks apply also to the “bomb in the toilet” problems, where $\mathcal{K}$ allows for very compact and at the same time intuitive encodings.

5.3.3. Performance

The running times on blocksworld instances in Table 2 show that DLV$^{\mathcal{K}}$ is significantly faster than the other systems if there are many action instances.

Under the world-state encodings of the different “bomb in the toilet” instances, DLV$^{\mathcal{K}}$ is not competitive except for BT(p) with concurrent dunks, where plan length is always 1, and BMTC(p). This indicates that DLV$^{\mathcal{K}}$’s performance is quite sensitive to (increasing) plan length, especially for sequential planning. Still, DLV$^{\mathcal{K}}$ outperforms SGP, a special purpose planning system, on all comparable instances, and also CPlan (which is the system most comparable to DLV$^{\mathcal{K}}$ in terms of expressiveness and similar in nature) seems to be within reach.

Table 8
Experimental results for BMTC(p) sequential

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

Under the knowledge-state encodings, DLV$^{\mathcal{K}}$ outperforms its competitors in many of the chosen examples. The sensitivity to increasing plan length/search space can, however, also partly be observed here, where execution times seem to grow drastically from one instance to the next. This can be partly explained by the general heuristics of the underlying DLV system, which might not scale up well in some cases. For instance, DLV as a general purpose problem solver does not include special heuristics towards plan search. In particular, during the answer set generation process, no distinction is made between actions and fluents, which might be useful for planning tasks to control the generation of answer sets respectively plans; this may be part of further investigations.

5.3.4. *Effect of concurrent actions and default negation*

Once we also consider concurrent actions (which are not supported by GPT and CMBP), DLV$^{\mathcal{K}}$ performs better than CPlan on some larger instances of BMTC(p,t) and BMTUC(p,t) (see Tables 7 and 9).

Using the expressive power of default negation to express unknown fluents with the knowledge-state encodings of "bomb in the toilet" in $\mathcal{K}$ pays off well: DLV$^{\mathcal{K}}$ outperforms all

Table 9
Experimental results for BMTUC(p) with concurrent dunks

BMTUC(p, t)	Steps	DLV$^{\mathcal{K}}$		CPlan
		ws	ks	
BMTUC(2, 2)	1	0.02 s	0.02 s	1.40 s
BMTUC(3, 2)	3	0.11 s	0.03 s	2.06 s
BMTUC(4, 2)	3	7.39 s	0.03 s	3.54 s
BMTUC(5, 2)	5	–	0.04 s	8.18 s
BMTUC(6, 2)	5	–	0.07 s	787.58 s
BMTUC(7, 2)	7	–	0.80 s	–
BMTUC(8, 2)	7	–	23.57 s	–
BMTUC(9, 2)	9	–	818.23 s	–
BMTUC(10, 2)	9	–	–	–
BMTUC(2, 3)	1	0.02 s	0.02 s	1.55 s
BMTUC(3, 3)	1	0.02 s	0.02 s	10.27 s
BMTUC(4, 3)	3	0.28 s	0.03 s	41.03 s
BMTUC(5, 3)	3	34.09 s	0.03 s	181.45 s
BMTUC(6, 3)	3	–	0.05 s	600.66 s
BMTUC(7, 3)	5	–	0.10 s	–
BMTUC(8, 3)	5	–	0.74 s	–
BMTUC(9, 3)	5	–	9.55 s	–
BMTUC(10, 3)	7	–	693.99 s	–
BMTUC(2, 4)	1	0.02 s	0.02 s	2.54 s
BMTUC(3, 4)	1	0.02 s	0.02 s	119.18 s
BMTUC(4, 4)	1	0.03 s	0.02 s	582.84 s
BMTUC(5, 4)	3	0.84 s	0.04 s	–
BMTUC(6, 4)	3	748.90 s	0.05 s	–
BMTUC(7, 4)	3	–	0.08 s	–
BMTUC(8, 4)	3	–	0.55 s	–
BMTUC(9, 4)	5	–	0.98 s	–
BMTUC(10, 4)	5	–	17.89 s	–

other systems, including the special purpose conformant planners GPT and CMBP, except on sequential BMTC(p, t) and BMTUC(p, t) with more than two toilets (see Tables 8 and 10), where CMBP is fastest.

5.3.5. Summary of experimental results

Overall, the results indicate that DLV$^{\mathcal{K}}$ is competitive with state of the art conformant planners, especially when exploiting the $\mathcal{K}$ language features in terms of knowledge-state problem encodings. Recall, however, that some of the systems compute minimal plans, which is (currently) not supported by DLV$^{\mathcal{K}}$. The comparison of DLV$^{\mathcal{K}}$ to CCALC/CPlan is particularly relevant, since these systems are closest in spirit to DLV$^{\mathcal{K}}$. As we can see, the advanced features of knowledge-state encoding lead to significant performance improvements.

Table 10
Experimental results for BMTUC(p) sequential

BMTUC(p,t)	Steps	DLV$^{\mathcal{K}}$		CPlan	CMBP	GPT
		ws	ks			
BMTUC(2, 2)	2	0.02 s	0.02 s	1.39 s	0.04 s	0.78 s
BMTUC(3, 2)	4	0.52 s	0.02 s	1.96 s	0.04 s	0.80 s
BMTUC(4, 2)	6	264.20 s	0.04 s	3.37 s	0.05 s	0.81 s
BMTUC(5, 2)	8	–	0.05 s	361.64 s	0.06 s	0.85 s
BMTUC(6, 2)	10	–	0.07 s	–	0.08 s	0.92 s
BMTUC(7, 2)	12	–	0.10 s	–	0.12 s	1.04 s
BMTUC(8, 2)	14	–	0.14 s	–	0.23 s	1.34 s
BMTUC(9, 2)	16	–	0.21 s	–	0.47 s	2.00 s
BMTUC(10, 2)	18	–	0.27 s	–	0.96 s	3.71 s
BMTUC(2, 3)	2	0.02 s	0.02 s	1.49 s	0.04 s	0.79 s
BMTUC(3, 3)	3	0.04 s	0.03 s	6.47 s	0.05 s	0.81 s
BMTUC(4, 3)	5	71.03 s	0.04 s	22.07 s	0.06 s	0.86 s
BMTUC(5, 3)	7	–	0.05 s	150.72 s	0.09 s	0.98 s
BMTUC(6, 3)	9	–	0.08 s	–	0.14 s	1.19 s
BMTUC(7, 3)	11	–	0.21 s	–	0.29 s	1.74 s
BMTUC(8, 3)	13	–	13.39 s	–	0.61 s	3.15 s
BMTUC(9, 3)	15	–	–	–	1.45 s	6.69 s
BMTUC(10, 3)	17	–	–	–	3.31 s	15.57 s
BMTUC(2, 4)	2	0.01 s	0.02 s	1.93 s	0.04 s	0.79 s
BMTUC(3, 4)	3	0.78 s	0.02 s	41.70 s	0.05 s	0.86 s
BMTUC(4, 4)	4	5.81 s	0.04 s	182.92 s	0.07 s	0.97 s
BMTUC(5, 4)	6	–	0.06 s	837.33 s	0.12 s	1.33 s
BMTUC(6, 4)	8	–	0.09 s	–	0.23 s	2.23 s
BMTUC(7, 4)	10	–	0.13 s	–	0.51 s	4.79 s
BMTUC(8, 4)	12	–	0.42 s	–	1.13 s	11.37 s
BMTUC(9, 4)	14	–	64.02 s	–	2.94 s	28.07 s
BMTUC(10, 4)	16	–	–	–	6.37 s	68.26 s

6. Further related work and conclusion

We have discussed the relation of DLV$^{\mathcal{K}}$ to a number of planning systems in Section 5 already, and complement this by briefly addressing further approaches and systems here.

6.1. *Further related work*

The idea to employ declarative logic programming systems for planning finds its roots in the seminal paper Subrahmanian and Zaniolo [43], which carried out the idea of satisfiability planning [22] to the framework of declarative logic programming.

Planning under incomplete knowledge has been widely investigated in the AI literature. Most works extend algorithms/systems for classical planning, rather than using deduction techniques for solving planning tasks as proposed in this paper. The systems Buridan [23], UDTPOP [37], Conformant Graphplan [42], CNLP [38] and CASSANDRA [39] fall in this class. In particular, Buridan, UDTPOP, and Conformant Graphplan can solve secure planning (also called conformant planning) problems, like DLV$^{\mathcal{K}}$. On the other hand, the systems CNLP and CASSANDRA deal with conditional planning (where the sequence of actions to be executed depends on dynamic conditions).

More recent works propose the use of automated reasoning techniques for planning under incomplete knowledge. In [41] a technique for encoding conditional planning problems in terms of 2-QBF formulas is proposed. The work in [11] proposes a technique based on regression for solving secure planning problems in the framework of the Situation Calculus, and presents a Prolog implementation of such a technique. In [31], sufficient syntactic conditions ensuring security of every (optimistic) plan are singled out. While sharing their logic-based nature, our work presented in this paper differs considerably from such proposals, since it is based on a different formalism.

6.2. Summary

In this paper, we have presented the DLV$^{\mathcal{K}}$ planning system, which implements the $\mathcal{K}$ action and planning language, introduced and discussed in the companion paper [6], on top of the DLV logic programming system. In the course of this, we have shown a transformation of planning problems in $\mathcal{K}$ into logic programming. In particular, we have given such a transformation for optimistic planning, which is planning in the traditional sense, and we have discussed how secure planning, i.e., conformant planning, can be realized for certain classes of planning problems via a transformation of security checking into logic programming. Our transformations use disjunctions in rule heads supported by DLV, but can be easily adapted to be disjunction-free, and thus become available for other logic programming systems such as Smodels [35]. Furthermore, we have compared our system on some standard benchmark problems to similar logic-based planning systems, namely CCALC [30,31], CPlan [10,15], CMBP [4]), GPT [3], and SGP [47]. We obtained promising performance results for secure planning exploiting the power of knowledge-state problem encodings, which are a distinguishing feature of the $\mathcal{K}$ planning language. As we believe, the results of the present paper show that knowledge-state encoding of planning problems has, besides it conceptual conciseness and natural appeal, potential also from a computational perspective.

6.3. Further and future work

Enhancing and further improving the DLV$^{\mathcal{K}}$ planning system is an ongoing effort. There are several issues which we address in our current and future research. One issue, discussed more in detail in the companion paper [6], is the development of a methodology for profitably using the knowledge-state planning approach.

Another issue concerns improvements and enhanced capabilities for secure planning. We have performed further experiments with a different approach of conformant answer set planning presented in [24]. In contrast to the plan security checking described here, that paper sketches an integrated encoding of conformant planning domains. In that approach, all answer sets correspond to secure plans and no further checking is necessary. These results seem to be very encouraging, but it is only possible to encode a rather restricted class of domains in DLV. In fact, since secure planning is Σ_3^P-complete [6], complexity arguments show that this method can not be efficiently extended to all planning domains.

On the other hand, security checking for all planning domains is in the class Π_2^P, and thus can be polynomially encoded to DLV. However, such a transformation remains to be designed in full generality. Besides these issues, also extended handling of incomplete security checking, as described in this paper, is part of our research, and we consider further built-in as well as support for user-defined security checks.

Finally, the use of the DLV engine as a computational backbone suggests to use its capabilities to enhance the DLV$^{\mathcal{K}}$ planning system by further features. In particular, by the use of weak constraints, it is possible to compute in DLV optimal answer sets of a logic program. This provides a computational basis for determining optimal plans of a planning problem, which are plans that minimize a given objective function, such as cost of actions, or execution time. To our knowledge, current logic-based planning systems do not offer comprehensive such capabilities. Enhancing the $\mathcal{K}$ language and the DLV$^{\mathcal{K}}$ system for optimal planning is on our agenda, and such features will be included in future DLV$^{\mathcal{K}}$ releases.

Acknowledgements

This work has greatly benefited from interesting discussions with and comments of Michael Gelfond, Vladimir Lifschitz, Riccardo Rosati, and Hudson Turner. Furthermore, we are grateful to Claudio Castellini, Alessandro Cimatti, Esra Erdem, Enrico Giunchiglia, David E. Smith, and Dan Weld for kindly supplying explanations, support, and comments on the systems that we used for comparison. Furthermore, we appreciate the review comments which helped to improve this paper.

This work was supported by FWF (Austrian Science Funds) under the projects P14781-INF and Z29-INF and the European Commission under projects FET-2001-37004 WASP and IST-2001-33570 INFOMIX.

Appendix A. Definition of language $\mathcal{K}$

This appendix contains, in shortened form, the definition of the language $\mathcal{K}$; see [6] for more details and examples.

A.1. Basic syntax

We assume σ^{act}, σ^{fl}, and σ^{typ} disjoint sets of action, fluent and type names, respectively, i.e., predicate symbols of arity $\geqslant 0$, and disjoint sets σ^{con} and σ^{var} of constant and variable symbols. Here, σ^{fl}, σ^{act} describe *dynamic knowledge* and σ^{typ} describes *static background knowledge*. An *action* (*respectively fluent*, *type*) *atom* is of form $p(t_1, \ldots, t_n)$, where $p \in \sigma^{act}$ (respectively, σ^{fl}, σ^{typ}) has arity n and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$. An action (respectively, fluent, type) literal l is an action (respectively, fluent, type) atom a or its negation $\neg a$, where "$\neg$" (alternatively, "–") is the true negation symbol. We define $\neg.l = a$ if $l = \neg a$ and $\neg.l = \neg a$ if $l = a$, where a is an atom. A set L of literals is *consistent*, if $L \cap \neg.L = \emptyset$. Furthermore, L^+ (respectively, L^-) is the set of positive (respectively, negative) literals in L. The set of all action (respectively, fluent, type) literals is denoted as $\mathcal{L}_{act}$ (respectively, $\mathcal{L}_{fl}$, $\mathcal{L}_{typ}$). Furthermore, $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$, $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup \mathcal{L}^{+}_{act}$, and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup \mathcal{L}^{+}_{act}$.

All actions and fluents must be declared using statements as follows.

Definition A.1 (*Action, fluent declaration*). An *action* (respectively, *fluent*) *declaration*, is of the form:

$$p(X_1, \ldots, X_n)\ \texttt{requires}\ t_1, \ldots, t_m \tag{A.1}$$

where $p \in \mathcal{L}^{+}_{act}$ (respectively, $p \in \mathcal{L}^{+}_{fl}$), $X_1, \ldots, X_n \in \sigma^{var}$ where $n \geqslant 0$ is the arity of p, $t_1, \ldots, t_m \in \mathcal{L}_{typ}$, $m \geqslant 0$, and every X_i occurs in $t_1, \ldots, t_m$.

If $m = 0$, the keyword `requires` may be omitted. Causation rules specify dependencies of fluents on other fluents and actions.

Definition A.2 (*Causation rule*). A *causation rule* (*rule*, for short) is an expression of the form

$$\begin{aligned}\texttt{caused}\ f\ \texttt{if}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l\\ \texttt{after}\ a_1, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n\end{aligned} \tag{A.2}$$

where $f \in \mathcal{L}_{fl} \cup \{\texttt{false}\}$, $b_1, \ldots, b_l \in \mathcal{L}_{fl,typ}$, $a_1, \ldots, a_n \mathcal{L}$, $l \geqslant k \geqslant 0$, and $n \geqslant m \geqslant 0$.

Rules where $n = 0$ are *static rules*, all others *dynamic rules*. When $l = 0$ (respectively, $n = 0$), "`if`" (respectively, "`after`") is omitted; if both $l = n = 0$,"`caused`" is optional.

We access parts of a causation rule r by $\mathsf{h}(r) = \{f\}$, $\mathsf{post}^{+}(r) = \{b_1, \ldots, b_k\}$, $\mathsf{post}^{-}(r) = \{b_{k+1}, \ldots, b_l\}$, $\mathsf{pre}^{+}(r) = \{a_1, \ldots, a_m\}$, $\mathsf{pre}^{-}(r) = \{a_{m+1}, \ldots, a_n\}$, and $\mathsf{lit}(r) = \{f, b_1, \ldots, b_l, a_1, \ldots, a_n\}$. Intuitively, $\mathsf{pre}^{+}(r)$ (respectively, $\mathsf{post}^{+}(r)$) accesses the state before (respectively, after) some action(s) happen.

Special static rules may be specified for the initial states.

Definition A.3 (*Initial state constraint*). An *initial state constraint* is a static rule of the form (A.2) preceded by "`initially`".

The language $\mathcal{K}$ allows conditional execution of actions, where several alternative executability conditions may be specified.

Definition A.4 (*Executability condition*). An *executability condition* e is an expression of the form

$$\texttt{executable}\ a\ \texttt{if}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l \tag{A.3}$$

where $a \in \mathcal{L}^{+}_{act}$ and $b_1, \ldots, b_l \in \mathcal{L}$, and $l \geqslant k \geqslant 0$.

If $l = 0$ (i.e., executability is unconditional), "`if`" is skipped. The parts of e are accessed by $\mathsf{h}(e) = \{a\}$, $\mathsf{pre}^{+}(e) = \{b_1, \ldots, b_k\}$, $\mathsf{pre}^{-}(e) = \{b_{k+1}, \ldots, b_l\}$, and $\mathsf{lit}(e) = \{a, b_1, \ldots, b_l\}$. Intuitively, $\mathsf{pre}^{-}(e)$ refers to the state at which some action's suitability is evaluated. Here, the state after action execution is not involved. For convenience, we define $\mathsf{post}^{+}(e) = \mathsf{post}^{-}(e) = \emptyset$.

All causal rules and executability conditions must satisfy the following condition, which is similar to safety in logic programs [46]: Each variable in a default-negated type literal must also occur in some literal which is not a default-negated type literal. No safety is requested for variables appearing in other literals. The reason is that variables appearing in fluent and action literals are implicitly safe by the respective type declarations.

Notation. For any causal rule, initial state constraint, and executability condition r and $\nu \in \{\mathsf{post}, \mathsf{pre}, \mathsf{b}\}$, we define $\nu(r) = \nu^+(r) \cup \nu^-(r)$, where $\mathsf{b}^s(r) = \mathsf{post}^s(r) \cup \mathsf{pre}^s(r)$.

A.1.1. Planning domains and planning problems

Definition A.5 (*Action description, planning domain*). An *action description* $\langle D, R\rangle$ consists of a finite set D of action and fluent declarations and a finite set R of safe causation rules, safe initial state constraints, and safe executability conditions. A $\mathcal{K}$ *planning domain* is a pair $PD = \langle \Pi, AD\rangle$, where Π is a stratified Datalog program (the *background knowledge*) which is safe (cf. [46]), and AD is an action description. We call PD *positive*, if no default negation occurs in AD.

Definition A.6 (*Planning problem*). A *planning problem* $\mathcal{P} = \langle PD, q\rangle$ is a pair of a planning domain PD and a *query* q, i.e.,

$$g_1, \ldots, g_m, \texttt{not}\ g_{m+1}, \ldots, \texttt{not}\ g_n\ ?\ (i) \tag{A.4}$$

where $g_1, \ldots, g_n \in \mathcal{L}_{fl}$ are variable-free, $n \geqslant m \geqslant 0$, and $i \geqslant 0$ denotes the plan length.

A.2. Semantics

We start with the preliminary definition of the typed instantiation of a planning domain. This is similar to the grounding of a logic program, with the difference being that only correctly typed fluent and action literals are generated.

Let $PD = \langle \Pi, \langle D, R\rangle\rangle$ be a planning domain, and let M be the (unique) answer set of Π [12]. Then, $\theta(p(X_1, \ldots, X_n))$ is a *legal action* (respectively, *fluent*) *instance* of an action (respectively, fluent) declaration $d \in D$ of the form (A.1), if θ is a substitution defined over $X_1, \ldots, X_n$ such that $\{\theta(t_1), \ldots, \theta(t_m)\} \subseteq M$. By $\mathcal{L}_{PD}$ we denote the set of all legal action and fluent instances. The instantiation of a planning domain respecting type information is as follows.

Definition A.7 (*Typed instantiation*). For any planning domain $PD = \langle \Pi, \langle D, R\rangle\rangle$, its *typed instantiation* is given by $PD{\downarrow} = \langle \Pi{\downarrow}, \langle D, R{\downarrow}\rangle\rangle$, where $\Pi{\downarrow}$ is the grounding of Π (over σ^{con}) and $R{\downarrow} = \{\theta(r) \mid r \in R,\ \theta \in \Theta_r\}$, where Θ_r is the set of all substitutions θ of the variables in r using σ^{con}, such that $\mathsf{lit}(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD} \cup (\neg.\mathcal{L}_{PD} \cap \mathcal{L}_{fl}^-)$.

In other words, in $PD{\downarrow}$ we replace Π and R by their ground versions, but keep of the latter only rules where the atoms of all fluent and action literals agree with their declarations. We say that a $PD = \langle \Pi, \langle D, R\rangle\rangle$ is *ground*, if Π and R are ground, and moreover that it is *well-typed*, if PD and $PD{\downarrow}$ coincide.

A.2.1. States and transitions

Definition A.8 (*State, state transition*). A *state* w.r.t. a planning domain PD is any consistent set $s \subseteq \mathcal{L}_{fl} \cap (\mathsf{lit}(PD) \cup \mathsf{lit}(PD)^-)$ of legal fluent instances and their negations. A *state transition* is any tuple $t = \langle s, A, s' \rangle$ where s, s' are states and $A \subseteq \mathcal{L}_{act} \cap \mathsf{lit}(PD)$ is a set of legal action instances in PD.

Observe that a state does not necessarily contain either f or $\neg f$ for each legal instance f of a fluent, and may even be empty ($s = \emptyset$). State transitions are not constrained; this will be done in the definition of *legal state transitions* below. We proceed in analogy to the definition of answer sets in [12], considering first positive (i.e., involving a positive planning domain) and then general planning problems.

In what follows, we assume that $PD = \langle \Pi, \langle D, R \rangle \rangle$ is a well-typed ground planning domain and that M is the unique answer set of Π. For any other PD, the respective concepts are defined through its typed grounding $PD\!\downarrow$.

Definition A.9 (*Legal initial state*). A state s_0 is a *legal initial state* for a positive PD, if s_0 is the least set (w.r.t. $\subseteq$) such that $\mathsf{post}(c) \subseteq s_0 \cup M$ implies $\mathsf{h}(c) \subseteq s_0$, for all initial state constraints and static rules $c \in R$.

For a positive PD and a state s, a set $A \subseteq \mathcal{L}^+_{act}$ is called *executable action set* w.r.t. s, if for each $a \in A$ there exists an executability condition $e \in R$ such that $\mathsf{h}(e) = \{a\}$, $\mathsf{pre}(e) \cap \mathcal{L}_{fl,typ} \subseteq s \cup M$, and $\mathsf{pre}(e) \cap \mathcal{L}^+_{act} \subseteq A$. Note that this definition allows for modeling dependent actions, i.e., actions which depend on the execution of other actions.

Definition A.10 (*Legal state transition*). Given a positive PD, a state transition $t = \langle s, A, s' \rangle$ is called *legal*, if A is an executable action set w.r.t. s and s' is the minimal consistent set that satisfies all causation rules w.r.t. $s \cup A \cup M$. That is, for every causation rule $r \in R$, if (i) $\mathsf{post}(r) \subseteq s' \cup M$, (ii) $\mathsf{pre}(r) \cap \mathcal{L}_{fl,typ} \subseteq s \cup M$, and (iii) $\mathsf{pre}(r) \cap \mathcal{L}_{act} \subseteq A$ all hold, then $\mathsf{h}(r) \neq \{\mathtt{false}\}$ and $\mathsf{h}(r) \subseteq s'$.

This is now extended to general a well-typed ground PD containing default negation using a Gelfond–Lifschitz type reduction to a positive planning domain [12].

Definition A.11 (*Reduction*). Let PD be a ground and well-typed planning domain, and let $t = \langle s, A, s' \rangle$ be a state transition. Then, the *reduction* $PD^t = \langle \Pi, \langle D, R^t \rangle \rangle$ of PD by t is the planning domain where R^t is obtained from R by deleting

(1) each $r \in R$, where either $\mathsf{post}^-(r) \cap (s' \cup M) \neq \emptyset$ or $\mathsf{pre}^-(r) \cap (s \cup A \cup M) \neq \emptyset$, and
(2) all default literals $\mathtt{not}\ L$ ($L \in \mathcal{L}$) from the remaining $r \in R$.

Note that PD^t is positive and ground. We extend further definitions as follows.

Definition A.12 (*Legal initial state, executable action set, legal state transition*). For any planning domain PD, a state s_0 is a *legal initial state*, if s_0 is a legal initial state for

$PD^{\langle\emptyset,\emptyset,s_0\rangle}$; a set A is an *executable action set* w.r.t. a state s, if A is executable w.r.t. s in $PD^{\langle s,A,\emptyset\rangle}$; and, a state transition $t = \langle s, A, s'\rangle$ is *legal*, if it is legal in PD^t.

A.2.2. Plans

Definition A.13 (*Trajectory*). A sequence of state transitions $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$, $n \geqslant 0$, is a *trajectory* for PD, if s_0 is a legal initial state of PD and all $\langle s_{i-1}, A_i, s_i\rangle$, $1 \leqslant i \leqslant n$, are legal state transitions of PD.

If $n = 0$, then $T = \langle\,\rangle$ is empty and has s_0 associated explicitly.

Definition A.14 (*Optimistic plan*). A sequence of action sets $\langle A_1, \ldots, A_i\rangle$, $i \geqslant 0$, is an *optimistic plan* for a planning problem $\mathcal{P} = \langle PD, q\rangle$, if a trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$ exists in PD which establishes the goal, i.e., $\{g_1, \ldots, g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$.

Optimistic plans amount to "plans", "valid plans", etc. as defined in the literature. The term "optimistic" should stress the credulous view in this definition, with respect to incomplete fluent information and nondeterministic action effects. In such cases, the execution of an optimistic plan P might fail to reach the goal. We thus resort to secure plans.

Definition A.15 (*Secure plans* (*alias conformant plans*)). An optimistic plan $\langle A_1, \ldots, A_n\rangle$ is a *secure plan*, if for every legal initial state s_0 and trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ such that $0 \leqslant j \leqslant n$, it holds that (i) if $j = n$ then T establishes the goal, and (ii) if $j < n$, then A_{j+1} is executable in s_j w.r.t. PD, i.e., some legal transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$ exists.

Note that plans admit in general the concurrent execution of actions. We call a plan $\langle A_1, \ldots, A_n\rangle$ *sequential* (or *non-concurrent*), if $|A_j| \leqslant 1$, for all $1 \leqslant j \leqslant n$.

A.3. Macros

$\mathcal{K}$ includes several macros as shorthands for frequently used concepts. Let $\mathtt{a} \in \mathcal{L}^+_{act}$ denote an action atom, $\mathtt{f} \in \mathcal{L}_{fl}$ a fluent literal, $\mathtt{B}$ a (possibly empty) sequence $b_1, \ldots, b_k$, `not` $b_{k+1}, \ldots,$ `not` b_l where each $b_i \in \mathcal{L}_{fl,typ}$, $i = 1, \ldots, l$, and $\mathtt{A}$ a (possibly empty) sequence $a_1, \ldots, a_m$, `not` $a_{m+1}, \ldots,$ `not` a_n where each $a_j \in \mathcal{L}$, $j = 1, \ldots, n$.

Inertia. To allow for an easy representation of fluent inertia, $\mathcal{K}$ provides

```
inertial f if B after A.
  ⇔  caused f if not ¬.f,B after f,A.
```

Defaults. A default value of a fluent can be expressed by the shortcut

```
default f.  ⇔  caused f if not ¬.f.
```

It is in effect unless some other causation rule provides evidence to the opposite value.

Totality. For reasoning under incomplete, but total knowledge $\mathcal{K}$ provides (`f` positive):

```
total f if B after A.
  ⇔  caused f if not -f, B after A.
     caused -f if not f, B after A.
```

State integrity. For integrity constraints that refer to the preceding state, $\mathcal{K}$ provides

```
forbidden B after A.  ⇔  caused false if B after A.
```

Nonexecutability. For specifying that some action is *not* executable, $\mathcal{K}$ provides

```
nonexecutable a if B.  ⇔  caused false after a, B.
```

By this definition, `nonexecutable` overrides `executable` in case of conflicts.

Non-concurrent plans. To exclude simultaneous execution of actions, $\mathcal{K}$ provides

noConcurrency. ⇔ caused false after a_1, a_2.

where a_1 and a_2 range over all possible actions such that $a_1, a_2 \in \mathcal{L}_{PD} \cap \mathcal{L}_{act}$ and $a_1 \neq a_2$.

In all macros, "`if B`" (respectively, "`after A`") can be omitted, if B (respectively, A) is empty.

Appendix B. Problem encodings for other systems

B.1. Blocksworld problem P1 for CMBP

```
DOMAIN blocks_P1
ACTIONS
act : { move_1_4, move_1_3, move_1_2, move_1_0, move_2_4, move_2_3,
        move_2_1, move_2_0, move_3_4, move_3_2, move_3_1, move_3_0,
        move_4_3, move_4_2, move_4_1, move_4_0 };
FLUENTS
        on_1 : 0..4; on_2 : 0..4; on_3 : 0..4; on_4 : 0..4;
        blocked_1 : boolean; blocked_2 : boolean;
        blocked_3 : boolean; blocked_4 : boolean;
INERTIAL on_1, blocked_1, on_2, blocked_2, on_3, blocked_3, on_4, blocked_4;
CAUSES act = move_1_4 FALSE IF blocked_1 | blocked_4;
CAUSES act = move_1_3 FALSE IF blocked_1 | blocked_3;
CAUSES act = move_1_2 FALSE IF blocked_1 | blocked_2;
CAUSES act = move_1_0 FALSE IF blocked_1;
CAUSES act = move_1_4 on_1 = 4 & blocked_4 IF 1;
CAUSES act = move_1_4 !blocked_2 IF on_1 = 2;
CAUSES act = move_1_4 !blocked_3 IF on_1 = 3;
CAUSES act = move_1_3 on_1 = 3 & blocked_3 IF 1;
CAUSES act = move_1_3 !blocked_2 IF on_1 = 2;
CAUSES act = move_1_3 !blocked_4 IF on_1 = 4;
CAUSES act = move_1_2 on_1 = 2 & blocked_2 IF 1;
CAUSES act = move_1_2 !blocked_3 IF on_1 = 3;
CAUSES act = move_1_2 !blocked_4 IF on_1 = 4;
CAUSES act = move_1_0 on_1 = 0 IF 1;
CAUSES act = move_1_0 !blocked_2 IF on_1 = 2;
CAUSES act = move_1_0 !blocked_3 IF on_1 = 3;
CAUSES act = move_1_0 !blocked_4 IF on_1 = 4;
CAUSES act = move_2_4 FALSE IF blocked_2 | blocked_4;
CAUSES act = move_2_3 FALSE IF blocked_2 | blocked_3;
CAUSES act = move_2_1 FALSE IF blocked_2 | blocked_1;
CAUSES act = move_2_0 FALSE IF blocked_2;
CAUSES act = move_2_4 on_2 = 4 & blocked_4 IF 1;
CAUSES act = move_2_4 !blocked_1 IF on_2 = 1;
CAUSES act = move_2_4 !blocked_3 IF on_2 = 3;
CAUSES act = move_2_3 on_2 = 3 & blocked_3 IF 1;
CAUSES act = move_2_3 !blocked_1 IF on_2 = 1;
CAUSES act = move_2_3 !blocked_4 IF on_2 = 4;
CAUSES act = move_2_1 on_2 = 1 & blocked_1 IF 1;
CAUSES act = move_2_1 !blocked_3 IF on_2 = 3;
CAUSES act = move_2_1 !blocked_4 IF on_2 = 4;
CAUSES act = move_2_0 on_2 = 0 IF 1;
CAUSES act = move_2_0 !blocked_1 IF on_2 = 1;
CAUSES act = move_2_0 !blocked_3 IF on_2 = 3;
CAUSES act = move_2_0 !blocked_4 IF on_2 = 4;
CAUSES act = move_3_4 FALSE IF blocked_3 | blocked_4;
CAUSES act = move_3_2 FALSE IF blocked_3 | blocked_2;
CAUSES act = move_3_1 FALSE IF blocked_3 | blocked_1;
CAUSES act = move_3_0 FALSE IF blocked_3;
CAUSES act = move_3_4 on_3 = 4 & blocked_4 IF 1;
CAUSES act = move_3_4 !blocked_1 IF on_3 = 1;
CAUSES act = move_3_4 !blocked_2 IF on_3 = 2;
CAUSES act = move_3_2 on_3 = 2 & blocked_2 IF 1;
CAUSES act = move_3_2 !blocked_1 IF on_3 = 1;
CAUSES act = move_3_2 !blocked_4 IF on_3 = 4;
CAUSES act = move_3_1 on_3 = 1 & blocked_1 IF 1;
CAUSES act = move_3_1 !blocked_2 IF on_3 = 2;
CAUSES act = move_3_1 !blocked_4 IF on_3 = 4;
CAUSES act = move_3_0 on_3 = 0 IF 1;
CAUSES act = move_3_0 !blocked_1 IF on_3 = 1;
CAUSES act = move_3_0 !blocked_2 IF on_3 = 2;
CAUSES act = move_3_0 !blocked_4 IF on_3 = 4;
CAUSES act = move_4_3 FALSE IF blocked_4 | blocked_3;
CAUSES act = move_4_2 FALSE IF blocked_4 | blocked_2;
CAUSES act = move_4_1 FALSE IF blocked_4 | blocked_1;
CAUSES act = move_4_0 FALSE IF blocked_4;
CAUSES act = move_4_3 on_4 = 3 & blocked_3 IF 1;
CAUSES act = move_4_3 !blocked_1 IF on_4 = 1;
CAUSES act = move_4_3 !blocked_2 IF on_4 = 2;
CAUSES act = move_4_2 on_4 = 2 & blocked_2 IF 1;
CAUSES act = move_4_2 !blocked_1 IF on_4 = 1;
CAUSES act = move_4_2 !blocked_3 IF on_4 = 3;
CAUSES act = move_4_1 on_4 = 1 & blocked_1 IF 1;
CAUSES act = move_4_1 !blocked_2 IF on_4 = 2;
CAUSES act = move_4_1 !blocked_3 IF on_4 = 3;
CAUSES act = move_4_0 on_4 = 0 IF 1;
CAUSES act = move_4_0 !blocked_1 IF on_4 = 1;
CAUSES act = move_4_0 !blocked_2 IF on_4 = 2;
CAUSES act = move_4_0 !blocked_3 IF on_4 = 3;
INITIALLY on_1 = 0 & on_2 = 0 & on_3 = 0 & on_4 = 3 &
          blocked_3 & !blocked_1 & !blocked_2 & !blocked_4;
CONFORMANT on_1 = 0 & on_2 = 1 & on_3 = 2 & on_4 = 3;
```

B.2. Blocks world problem P1 for GPT

```
(define (domain bw)
        (:model SEARCH)
        (:types BLOCK)
        (:functions (on BLOCK BLOCK )
        (clear BLOCK :boolean))
        (:objects table - BLOCK)
        (:action puton
                 :parameters ?X - BLOCK ?Y - BLOCK ?Z - BLOCK
                 :precondition (:and (= (on ?X) ?Z)
                                     (= (clear ?X) true)
                                     (:or (= (clear ?Y) true) (= ?Y table))
                                     (:not (= ?Y ?Z))
                                     (:not (= ?X ?Z))
                                     (:not (= ?X table)))
                 :effect
                                     (:set (on ?X) ?Y)
                                     (:set (clear ?Z) true)
                                     (:set (clear ?Y) false)))
(define (problem p1)
        (:domain bw)
        (:objects b0 b1 b2 b3 - BLOCK)
        (:init
               (:set (on b0) table)
               (:set (on b1) table)
               (:set (on b2) table)
               (:set (on b3) b2)
               (:set (clear b0) true)
               (:set (clear b1) true)
               (:set (clear b2) false)
               (:set (clear b3) true)
               (:set (clear table) false))
        (:goal (:and (= (on b3) b2)
               (= (on b2) b1)
               (= (on b1) b0)
               (= (on b0) table))))
```

Appendix C. DLV$^{\mathcal{K}}$ encodings of BMTUC(p, t)

C.1. World-state encoding

Background knowledge:

```
package(1).  package(2).  ...  package(p).
toilet(1).  toilet(2).  ...  toilet(t).
```

DLV$^{\mathcal{K}}$ program:

```
fluents:    clogged(T) requires toilet(T).
            armed(P) requires package(P).
            unsafe.
actions:    dunk(P,T) requires package(P), toilet(T).
            flush(T) requires toilet(T).
always:     inertial armed(P).
            inertial clogged(T).
            caused -clogged(T) after flush(T).
            caused -armed(P) after dunk(P,T).
            total clogged(T) after dunk(P,T).
            caused unsafe if armed(P).
            executable flush(T).
            executable dunk(P,T) if not clogged(T).
            nonexecutable dunk(P,T) if flush(T).
            nonexecutable dunk(P,T) if dunk(P1,T), P <> P1.
            nonexecutable dunk(P,T) if dunk(P,T1), T <> T1.
initially:  total armed(P).
            forbidden armed(P), armed(P1), P <> P1.
            forbidden not unsafe.
```

In this encoding, weak negation of the fluent `clogged` is a CWA representation of the negated fluent `-clogged`, which relieves us from storing negative information explicitly.

The possible world-states are encoded (1) via the `total`-statement for the fluent `armed` in the `initially` section, which generates all possible initial states, and (2) via the `total`-statement for the fluent `clogged` in the `always` section, which specifies the effect of dunking a package.

C.2. Knowledge-state encoding

Background knowledge:

```
package(1).  package(2).  ...  package(p).
toilet(1).  toilet(2).  ...  toilet(t).
```

DLV$^{\mathcal{K}}$ program:

```
fluents:    clogged(T) requires toilet(T).
            armed(P) requires package(P).
            dunked(T) requires toilet(T).
            unsafe.
actions:    dunk(P, T) requires package(P), toilet(T).
            flush(T) requires toilet(T).
always:     inertial -armed(P).
            inertial clogged(T) if not dunked(T).
            inertial -clogged(T) if not dunked(T).
            caused dunked(T) after dunk(P, T).
            caused -clogged(T) after flush(T).
            caused -armed(P) after dunk(P, T).
            caused unsafe if not -armed(P).
            executable flush(T).
            executable dunk(P, T) if -clogged(T).
            nonexecutable dunk(P, T) if flush(T).
            nonexecutable dunk(P, T) if dunk(P1, T), P <> P1.
            nonexecutable dunk(P, T) if dunk(P, T1), T <> T1.
initially: -clogged(T).
```

In this encoding, the fluents `armed` and `clogged` are treated as three-valued. Instead of encoding all possible initial world states by cases, we have a single initial state in which we only know that all toilets are not clogged, while the values of the fluents `armed` are open. We may gain, on the one hand, knowledge on fluent `armed` by executing an action `flush`, while on the other hand, we may lose ("forget") information on fluent `clogged`, if we know that something has been dunked into the respective toilet (for his projection, we use the auxiliary fluent `dunked`).

An advantage of this encoding is that optimistic and secure plans coincide on this encoding, since nondeterministic effects of action `dunk` are treated by "forgetting" the value of the respective fluent `clogged`. We point out that the "bomb in toilet problem" is per se computationally easy; so it seems that encodings based on world-states artificially bloat this problem, because of their lack of a natural statement about fluents being unknown in some state. For further discussion, we refer to [6].

References

[1] R. Bayardo, R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proc. AAAI-97, Providence, RI, 1997, pp. 203–208.

[2] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1997) 281–300.

[3] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: S. Chien, S. Kambhampati, C.A. Knoblock (Eds.), Proc. AIPS-00, Breckenridge, CO, 2000, pp. 52–61.

[4] A. Cimatti, M. Roveri, Conformant planning via symbolic model checking, J. Artificial Intelligence Res. 13 (2000) 305–338.

[5] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, Planning under incomplete knowledge, in: J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, P.J. Stuckey (Eds.), CL2000, in: Lecture Notes in AI (LNAI), Vol. 1861, Springer, London, 2000, pp. 807–821.

[6] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity, Technical Report INFSYS RR-1843-01-11, TU Wien, 2001.

[7] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, The KR system `dlv`: Progress report, comparisons and benchmarks, in: A.G. Cohn, L. Schubert, S.C. Shapiro (Eds.), Proc. KR-98, Morgan Kaufmann, San Mateo, CA, 1998, pp. 406–417.

[8] E. Erdem, Applications of logic programming to planning: Computational experiments, unpublished draft, 1999, http://www.cs.utexas.edu/users/esra/papers.html.

[9] W. Faber, N. Leone, G. Pfeifer, Pushing goal derivation in DLP computations, in: M. Gelfond, N. Leone, G. Pfeifer (Eds.), Proc. LPNMR-99, El Paso, TX, in: Lecture Notes in AI (LNAI), Vol. 1730, 1999, pp. 177–191.

[10] P. Ferraris, E. Giunchiglia, Planning as satisfiability in nondeterministic domains, in: Proc. AAAI-00, Austin, TX, AAAI Press/MIT Press, Cambridge, MA, 2000, pp. 748–753.

[11] A. Finzi, F. Pirri, R. Reiter, Open world planning in the situation calculus, in: Proc. AAAI-00, Austin, TX, AAAI Press/MIT Press, Cambridge, MA, 2000, pp. 754–760.

[12] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Comput. 9 (1991) 365–385.

[13] M. Gelfond, V. Lifschitz, Representing action and change by logic programs, J. Logic Programming 17 (1993) 301–321.

[14] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—The Planning Domain Definition language, Tech. Report, Yale Center for Computational Vision and Control, 1998, available at http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz.

[15] E. Giunchiglia, Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism, in: A.G. Cohn, F. Giunchiglia, B. Selman (Eds.), Proc. KR-2000, Morgan Kaufmann, San Mateo, CA, 2000, pp. 657–666.

[16] E. Giunchiglia, G.N. Kartha, V. Lifschitz, Representing action: Indeterminacy and ramifications, Artificial Intelligence 95 (1997) 409–443.

[17] E. Giunchiglia, V. Lifschitz, An action language based on causal explanation: Preliminary report, in: Proc. AAAI-98, Madison, WI, 1998, pp. 623–630.

[18] E. Giunchiglia, V. Lifschitz, Action languages, temporal action logics and the situation calculus, in: Working Notes of the IJCAI'99 Workshop on Nonmonotonic Reasoning, Action, and Change, Stockholm, Sweden, 1999.

[19] R. Goldman, M. Boddy, Expressive planning and explicit knowledge, in: Proc. AIPS-96, AAAI Press, 1996, pp. 110–117.

[20] L. Iocchi, D. Nardi, R. Rosati, Planning with sensing, concurrency, and exogenous events: Logical framework and implementation, in: A.G. Cohn, F. Giunchiglia, B. Selman (Eds.), Proc. KR-2000, Morgan Kaufmann, San Mateo, CA, 2000, pp. 678–689.

[21] G.N. Kartha, V. Lifschitz, Actions with indirect effects (preliminary report), in: Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94), Bonn, Germany, 1994, pp. 341–350.

[22] H. Kautz, B. Selman, Planning as satisfiability, in: Proc. ECAI-92, Vienna, Austria, 1992, pp. 359–363.

[23] N. Kushmerick, S. Hanks, D.S. Weld, An algorithm for probabilistic planning, Artificial Intelligence 76 (1–2) (1995) 239–286.

[24] N. Leone, R. Rosati, F. Scarcello, Enhancing answer set planning, in: A. Cimatti, H. Geffner, E. Giunchiglia, J. Rintanen (Eds.), IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information, Seattle, WA, 2001, pp. 33–42.

[25] V. Lifschitz, Foundations of logic programming, in: G. Brewka (Ed.), Principles of Knowledge Representation, CSLI Publications, Stanford, 1996, pp. 69–127.

[26] V. Lifschitz, Action languages, answer sets and planning, in: K. Apt, V.W. Marek, M. Truszczyński, D.S. Warren (Eds.), The Logic Programming Paradigm—A 25-Year Perspective, Springer, Berlin, 1999, pp. 357–373.
[27] V. Lifschitz, Answer set planning, in: D.D. Schreye (Ed.), Proc. ICLP-99, MIT Press, Las Cruces, NM, 1999, pp. 23–37.
[28] V. Lifschitz, H. Turner, Splitting a logic program, in: P. Van Hentenryck (Ed.), Proc. ICLP-94, MIT Press, Cambridge, MA, 1994, pp. 23–37.
[29] V. Lifschitz, H. Turner, Representing transition systems by logic programs, in: M. Gelfond, N. Leone, G. Pfeifer (Eds.), Proc. LPNMR-99, El Paso, TX, in: Lecture Notes in AI (LNAI), Vol. 1730, Springer, Berlin, 1999, pp. 92–106.
[30] N. McCain, H. Turner, Causal theories of actions and change, in: Proc. AAAI-97, Providence, RI, 1997, pp. 460–465.
[31] N. McCain, H. Turner, Satisfiability planning with causal theories, in: A.G. Cohn, L. Schubert, S.C. Shapiro (Eds.), Proc. KR-98, Morgan Kaufmann, San Mateo, CA, 1998, pp. 212–223.
[32] J. McCarthy, Formalization of Common Sense. Papers by John McCarthy edited by V. Lifschitz, Ablex, Norwood, NJ, 1990.
[33] J. McCarthy, P.J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer, D. Michie (Eds.), Machine Intelligence 4, Edinburgh University Press, Edinburgh, 1969, pp. 463–502, reprinted in [32].
[34] D. McDermott, A critique of pure reason, Comput. Intelligence 3 (1987) 151–237, cited in [4].
[35] I. Niemelä, Logic programming with stable model semantics as constraint programming paradigm, Ann. Math. Artificial Intelligence 25 (3–4) (1999) 241–273.
[36] C.H. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.
[37] M.A. Peot, Decision-theoretic planning, Ph.D. thesis, Stanford University, Stanford, CA, 1998.
[38] M.A. Peot, D.E. Smith, Conditional nonlinear planning, in: Proceedings of the First International Conference on Artificial Intelligence Planning Systems, AAAI Press, 1992, pp. 189–197.
[39] L. Pryor, G. Collins, Planning for contingencies: A decision-based approach, J. Artificial Intelligence Res. 4 (1996) 287–339.
[40] R. Reiter, On closed world data bases, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, New York, 1978, pp. 55–76.
[41] J. Rintanen, Constructing conditional plans by a theorem-prover, J. Artificial Intelligence Res. 10 (1999) 323–352.
[42] D.E. Smith, D.S. Weld, Conformant Graphplan, in: Proc. AAAI-98, Madison, WI, AAAI Press/MIT Press, Cambridge, MA, 1998, pp. 889–896.
[43] V. Subrahmanian, C. Zaniolo, Relating stable models and AI planning domains, in: L. Sterling (Ed.), Proceedings of the 12th International Conference on Logic Programming, Tokyo, Japan, MIT Press, Cambridge, MA, 1995, pp. 233–247.
[44] G.J. Sussman, The virtuous nature of bugs, in: J. Allen, J. Hendler, A. Tate (Eds.), Readings in Planning, Morgan Kaufmann, San Mateo, CA, 1990, Chapter 3, pp. 111–117, originally written 1974.
[45] H. Turner, Representing actions in logic programs and default theories: A situation calculus approach, J. Logic Programming 31 (1–3) (1997) 245–298.
[46] J.D. Ullman, Principles of Database and Knowledge Base Systems, Vol. 1, Computer Science Press, 1989.
[47] D.S. Weld, C.R. Anderson, D.E. Smith, Extending Graphplan to handle uncertainty & sensing actions, in: Proc. AAAI-98, Madison, WI, AAAI Press/MIT Press, Cambridge, MA, 1998, pp. 897–904.
[48] H. Zhang, SATO: An efficient propositional prover, in: Proceedings of the International Conference on Automated Deduction (CADE-1997), 1997, pp. 272–275.