Planning under Incomplete Knowledge

Thomas Eiter, Wolfgang Faber, Nicola Leone*, Gerald Pfeifer, and Axel Polleres

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

Abstract. We propose a new logic-based planning language, called $\mathcal{K}$. Transitions between states of knowledge can be described in $\mathcal{K}$, and the language 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. A planning system supporting $\mathcal{K}$ is implemented on top of the disjunctive logic programming system DLV. This novel system allows for solving hard planning problems, including secure planning under incomplete initial states, which cannot be solved at all by other logic-based planning systems such as traditional satisfiability planners.

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 action planning, such as [24, 8, 15, 10, 34, 11, 19, 12, 14]. These languages allow us to specify planning problems of the form "find a sequence of actions that leads from a given initial state to a given 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 precondition (formula over the fluents) is true in the current state, and its execution 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 among *possible states of the world* (where every fluent must necessarily be either true or false). However, robots usually don't 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 this paper, we propose a new language, $\mathcal{K}$, for planning under incomplete knowledge. We name it $\mathcal{K}$ to emphasize that it describes transitions among *states of knowledge* rather than among states of the world. Nonetheless, the language is very flexible, and is capable of modeling transitions among states of the world (i.e., states of complete knowledge) and reason about them as a particular case (see below). Compared to similar planning languages, $\mathcal{K}$ is closer in spirit to answer set semantics [7] than to classical

* Please address correspondence to this author.

logics. It allows the use of default negation, exploiting the power of answer sets to deal with incomplete knowledge. We have implemented $\mathcal{K}$ on top of the `DLV` system [4, 5], and provide a powerful planning system (available on the web), which is ready-to-use for experiments.

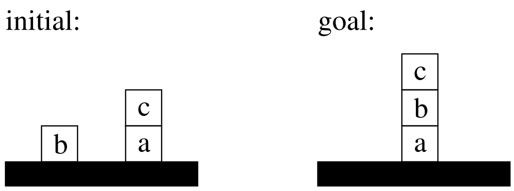

Fig. 1. A blocks world example.

Overview of $\mathcal{K}$

The main features of the language $\mathcal{K}$ (formally defined in Section 2) and our planning system are briefly summarized as follows. We occasionally refer to well-known planning problems in the blocks world, which require to turn one configuration of blocks into another (see Figure 1).

Type Declarations. They specify the ranges of the arguments of fluents and actions. For instance,

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

specifies the types for the arguments of action *move*. The literals after "`requires`" (`block(B)`, `location(L)`) must be positive literals of the static background knowledge, given by a normal (or-free) stratified logic program.

Causation Rules. A (causation) rule, which is main construct of $\mathcal{K}$, is syntactically similar to a rule of the language $\mathcal{C}$ [11, 19, 18] and has the form:

```
caused f if B after A.
```

Intuitively, the above rule says: if `B` is known to be true in the current state and `A` is known to be true in the previous state, then `f` is known to be true in the current state as well. Both the if part and the after part are allowed to be empty (which means that they are true).

Default Negation. `not` can appear in the bodies of the rules. It allows for natural modeling of inertial properties, default properties, and dealing with incomplete knowledge in general, like in logic programming with answer set semantics. Strong negation ("¬," written in programs as `-`) is allowed as well. Several shortcuts are defined for $\mathcal{K}$, e.g.

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

informally states that on(X,Y) holds at the current state if on(X,Y) held at the previous state unless −on(X,Y) is explicitly known to hold. Furthermore,

```
default -on(X,Y).
```

states that -on(X,Y) is assumed, unless on(X,Y) is known to hold (as it has been explicitly entailed by some causation rule).

Executability of Actions. This can be expressed in a direct way: For instance,

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

states that (block) X can be moved on (location) Y if both X and Y are clear and $X \neq Y$. Multiple executable statements for the same action are allowed. A statement with empty body

```
executable move(X,Y).
```

says that move is always executable, provided that the type restrictions on X and Y are respected (and no inconsistency arises from the execution). Execution of an action A under condition B is forbidden by

```
nonexecutable A if B.
```

In case of conflicts, nonexecutable A overrides executable A.

Initial State Constraints. In general, a rule expresses a state constraint that must be fulfilled in all states. An *initial state constraint* (which is preceded by the keyword initially:), must be satisfied only in the initial state. For example,

```
initially: caused false if block(B), not supported(B).
```

enforces the fluent supported to be true on every block at the initial state; the constraint is irrelevant for all subsequent states. Initial state constraints may profitably reduce computation effort: If we are guaranteed that the actions preserve a property, say P, then it is sufficient to check the validity of P only on the initial state for ensuring it holds in any state.

Parallel Execution of Actions. Simultaneous execution of a number of actions is allowed in $\mathcal{K}$. This can be prohibited by the statement

```
noConcurrency.
```

which enforces the execution of at most one action at a time.

Handling of Complete and Incomplete Knowledge. The language allows one also to represent transitions between possible states of the world (which can be seen as states of complete knowledge). First of all, we can easily check that the knowledge on a fluent, say `f`, is complete, using a rule

```
caused false if not f, not -f.
```

Moreover, we can "totalize" the knowledge of a fluent by declaring

```
total f.
```

which means that, unless a truth value for `f` can be derived, the cases where `f` resp. `-f` is true will be both considered.

Goals and Plans. A goal is a conjunction of ground literals, and a plan for a goal is a sequence of (in general, sets of) actions whose execution leads from an initial state to a state where all literals in the goal are true. In $\mathcal{K}$, the goal is followed by a question mark and by the number of allowed steps in a plan. For instance,

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

asks to find a plan of length 3 for the goal of Figure 1.

Secure Plans. A key feature of $\mathcal{K}$ is the command

```
securePlan.
```

by which we ask the system to compute only *secure* plans (a secure plan is often called *conformant* in the literature). Informally, a plan is secure if it is applicable starting at any legal initial state, and enforces, regardless of how the state evolves, the goal. Using this feature, we can also model *possible-worlds planning with an incomplete initial state*, where the initial world is known only partially, and we look for a plan reaching the desired goal from every possible world according to the initial state.

Contribution of the work

The main contributions of the paper are the following:

- We introduce the planning language $\mathcal{K}$ and provide a declarative model theoretic semantics for it.
- We illustrate the knowledge modeling features of the language by encoding some classical planning problems in $\mathcal{K}$.
- We analyze the computational complexity of language $\mathcal{K}$ in the propositional case. Deciding the existence of an optimistic plan achieving the goal in a fixed number of steps is NP-complete. If the plan should be secure, the problem is obviously harder, because it allows us to encode also planning under incomplete initial states as in [1]. The problem is then Σ_2^P-complete in general, but only mildly harder than NP if concurrent actions are not allowed. On the other hand, deciding existence of a secure plan of variable (arbitrary) length is NEXPTIME-complete, and thus not polynomially reducible to planning in STRIPS-like systems [6] which are PSPACE-complete [2].

An implementation of $\mathcal{K}$ exists as a frontend to the DLV system [4]. This frontend applies a $\mathcal{K}$ evaluator on top of the DLV system, using an efficient translation from $\mathcal{K}$ to disjunctive logic programming.

DLV (including the $\mathcal{K}$ frontend) can be freely retrieved from `<URL:http://www.dbai.tuwien.ac.at/proj/dlv/>`. To the best of our knowledge, this is the first declarative LP-based planning system which allows to solve Σ_2^P-hard planning problems, like planning under incomplete initial states.

For space limitations, we omit some technical material here. Further details can be obtained from the web page mentioned above.

2 Language $\mathcal{K}$

2.1 Syntax

Actions, Fluents and Types. Let σ^{act}, σ^{fl}, and σ^{typ} be disjoint sets of action, fluent and type names, respectively. These names are effectively predicate symbols with associated arity (≥ 0). Here, $\sigma^{fl} \cup \sigma^{act}$ are used to describe *dynamic knowledge*, whereas $\sigma^{typ}$is used to describe *static background knowledge*.

Furthermore, let σ^{con} and σ^{var} be the disjoint sets of constant and variable symbols, respectively.[1]

Given $p \in \sigma^{act}$ (resp. σ^{fl}, σ^{typ}), an action (resp. fluent, type) atom is defined as $p(t_1, \ldots, t_n)$, where n is the arity of p and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$. An action (resp. fluent, type) literal is an action (resp. fluent, type) atom, which is possibly preceded by the true negation symbol "$\neg$". A literal (or any other syntactic object) is *ground* if it does not contain variables.

For any literal l, let $\neg.l$ denote its complement, i.e. $\neg l$ if l is an atom and a if $l = \neg a$. Similarly, for a set L of literals, $\neg.L = \{\neg.l \mid l \in L\}$. A set L of literals is *consistent*, if $L \cap \neg.L = \emptyset$. Furthermore, L^+ (resp., L^-) denotes the set of positive (resp., negative) literals in L.

The set of all action (resp. fluent, type) literals is denoted as $\mathcal{L}_{act}$ (resp. $\mathcal{L}_{fl}$, $\mathcal{L}_{typ}$). Furthermore, let then $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$; $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup \mathcal{L}^+_{act}$ (dyn stands for *dynamic literals*); and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup \mathcal{L}^+_{act}$.[2]

Action/Fluent declarations. All actions and fluents have to be declared using an *action* (resp. *fluent*) *declaration* of the form:

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

where $p \in \mathcal{L}^+_{act}$ (resp. $p \in \mathcal{L}^+_{fl}$), $X_1, \ldots, X_n \in \sigma^{var}$, $t_1, \ldots, t_m \in \mathcal{L}_{typ}$, n is the arity of p, and all X_i occur also in $t_1, \ldots t_n$ and $m \geq 0$. If $m = 0$, the `requires` part may be skipped.

[1] Following logic programming conventions, in this paper constant and variable symbols are denoted as strings starting with a lower or upper case character, respectively.

[2] Note that in this definition only positive action literals are allowed.

Causation Rules. Causation rules are used to define static and dynamic dependencies. *Causation rules* (*rules*, for short) are of the form

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

where $f \in \mathcal{L} \cup \{\texttt{false}\}$, $b_1,\ldots,b_l \in \mathcal{L}_{fl,typ}$, $a_1,\ldots,a_n \in \mathcal{L}$, $l \geq k \geq 0$, and $n \geq m \geq 0$. Rules where $n = 0$ are referred to as *static rules*, all other rules as *dynamic rules*. When $l = 0$, the `if` part can be omitted; likewise, if $n = 0$, the `after` part can be skipped. If both $l = n = 0$, also `caused` is optional. Given a causation rule r, let $h(r) = \{f\}, post^{+}(r) = \{b_1,\ldots,b_k\}, post^{-}(r) = \{b_{k+1},\ldots,b_l\}, pre^{+}(r) = \{a_1,\ldots,a_m\}, pre^{-}(r) = \{a_{m+1},\ldots,a_n\}, lit(r) = \{f, b_1,\ldots,b_l, a_1,\ldots,a_n\}$.

Initial State Constraints. While the scope of general static rules is over all knowledge states, it is often useful to specify rules only for the initial states. *Initial state constraints* are static rules of the form (2) preceded by the keyword `initially`. For an initial state constraint ic. $h(ic)$, $post^{+}(ic)$, $post^{-}(ic)$, $pre^{+}(ic)$, and $pre^{-}(ic)$ are defined as for its rule part.

Conditional Executability. $\mathcal{K}$ allows STRIPS-style [6] conditional execution of actions. A difference is that $\mathcal{K}$ allows several alternative executability conditions for an action which are beyond the repertoire of standard STRIPS. An *executability condition* is an expression of the form

$$\texttt{executable}\ a\ \texttt{if}\ b_1,\ldots,b_m,\texttt{not}\ b_{m+1},\ldots,\texttt{not}\ b_n \tag{3}$$

where $a \in \mathcal{L}^{+}_{act}$ and $b_1,\ldots,b_n \in \mathcal{L}$, and $n \geq m \geq 0$. If $n = 0$, the `if` part is usually skipped, expressing unconditional executability. Given an executability condition e, let $h(e) = \{a\}$, $post^{+}(e) = post^{-}(e) = \emptyset, pre^{+}(e) = \{b_1,\ldots,b_m\}$, $pre^{-}(e) = \{b_{m+1},\ldots,b_n\}$, and $lit(e) = \{a, b_1,\ldots,b_n\}$.

Safety restriction. In $\mathcal{K}$, all rules (including initial state constraints) and executability conditions have to satisfy the following syntactic restriction, which is similar to the notion of safety in logic programs [35]. All variables in a default negated type literal must also occur in some literal which is not a default negated type literal.

Thus, safety is required only for variables appearing in default negated type literals, while it is not required at all for variables appearing in fluent and action literals. The reason is that the range of the latter variables is implicitly restricted by the respective type declaration.

Action Descriptions, Planning Domains, Planning Problems. An *action description* is a pair $\langle D, R\rangle$ where D is a finite set of action and fluent declarations and R is a finite set of safe executability conditions, safe causation rules, and safe initial state constraints.

A *planning domain* is a pair $PD = \langle \Pi, AD \rangle$, where Π is a normal stratified datalog program (referred to as *background knowledge*) that is safe (in the standard LP sense), and AD is an action description. PD is *positive*, if no default negation occurs in AD.

A *query* is of the form

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

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

A *planning problem* $\langle PD, q \rangle$ is a pair of a planning domain PD and a query q.

2.2 Semantics

First we will define the legal instantiations of a planning problem. This is similar to the grounding of a logic program—the difference is that only correctly typed fluent and action literals are generated.

Instantiation. Let substitutions and their application to syntactic objects be defined as usual (assignments of constants to variables). We first define the notion of legal action (resp. fluent) instantiations:

Let $PD = \langle \Pi, \langle D, R \rangle \rangle$ be a planning domain, and let M be the (unique) answer set of Π. Then, $\theta(p(X_1, \ldots, X_n))$ is a *legal action* (resp. *fluent*) *instantiation* for an action (resp. fluent) declaration $d \in D$ of the form (1), if θ is a substitution defined over all $X_1, \ldots, X_n$ 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 instantiations.

Based on the above definition of action and fluents instantiations, we define the instantiation of a planning domain, $PD{\downarrow}$, as follows:

$$PD{\downarrow} = \bigcup_{r \in R} \bigcup_{\theta \in \Theta_r} \{\theta(r)\},$$

where Θ_r is the set of all substitutions θ defined over all variables in r, such that $lit(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD}$ and $(post^+(\theta(r)) \cup pre^+(\theta(r))) \cap \mathcal{L}_{typ} \subseteq M$ hold. In other words, actions and fluents must agree with their declarations and positive type literals must agree with the background knowledge.

$PD{\downarrow}$ has a ground action description, in which all fluent and action literals agree with their declarations and all type literals agree with the background knowledge.

States and Transitions. In analogy to the definition of stable models and answer sets [7], we will first define the semantics for positive (i.e., default negation-free) planning problems. Subsequently we define a reduction from general planning problems to positive ones.

A consistent set of ground fluent literals is called *state*. A tuple $t = \langle s, A, s' \rangle$ where s, s' are states and A is a set of action atoms is called a *state transition*.

In what follows, let PD be a planning domain, whose instantiation is $PD{\downarrow} = \langle \Pi, \langle D, R \rangle \rangle$, and M is the unique answer set of Π.

A state s_0 is called *legal initial state* for a positive *PD* iff for each initial state constraint $c \in R$, $h(c)$ is in s_0 if $post^+(c) \subseteq s_0 \cup M$ holds and s_0 is minimal under this condition.

For a positive *PD* and a state s, a set $A \subseteq \mathcal{L}_{act}^+$ is called *executable action set* w.r.t. s iff for each $a \in A$ there exists an executability condition $e \in R$ such that $h(e) = \{a\}$, $pre^+(e) \cap \mathcal{L}_{fl,typ} \subseteq s \cup M$, and $pre^+(e) \cap \mathcal{L}_{act}^+ \subseteq A$.[3]

For a positive *PD* and a state transition $t = \langle s, A, s' \rangle$ is called *legal state transition* 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. $s \cup A$, i.e. for every causation rule $r \in R$, if (i) $post^+(r) \cap \mathcal{L}_{fl} \subseteq s'$, (ii) $pre^+(r) \cap \mathcal{L}_{fl} \subseteq s$, and (iii) $pre^+(r) \cap \mathcal{L}_{act} \subseteq A$ all hold, then $h(r) \neq \{\texttt{false}\}$ and $h(r) \subseteq s'$. Note that we need not consider type literals, as they have already been dealt with in the instantiation step.

The above definitions are now generalized to a PD containing default negation by defining a reduction to a positive planning domain.

For an arbitrary *PD* and a state transition $t = \langle s, A, s' \rangle$, the *reduction* $PD^t = \langle \Pi, \langle D, R^t \rangle \rangle$ is a planning domain where R^t is obtained from R by deleting those $r \in R$, for which either $post^-(r) \cap (s' \cup M) \neq \emptyset$ or $pre^-(r) \cap (s \cup A \cup M) \neq \emptyset$ holds, and by deleting all `not` L ($L \in \mathcal{L}$) from the remaining $r \in R$. Note that PD^t is positive and ground.

For an arbitrary *PD*, a state s_0 is called *legal initial state* iff s_0 is a legal initial state for PD^t with $t = \langle \emptyset, \emptyset, s_0 \rangle$.

A is an *executable action set* in *PD* w.r.t. a state s iff A is executable w.r.t. s in PD^t with $t = \langle s, A, \emptyset \rangle$.

A transition $t = \langle s, A, s' \rangle$ is a *legal state transition* in *PD* iff it is a legal transition w.r.t PD^t. A sequence of state transitions $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{n-1}, A_n, s_n \rangle \rangle$, $n \geq 0$, is a *legal transition sequence* for *PD*, if s_0 is a legal initial state of *PD* and all $\langle s_{i-1}, A_i, s_i \rangle$, $1 \leq i \leq n$, are legal state transitions of *PD*. In particular, $T = \langle \rangle$ is empty if $n = 0$.

We say that an arbitrary planning domain *PD* is *proper* if, given a state s and an action sequence A, the existence of a legal state transition $\langle s, A, s' \rangle$ is polynomially decidable (i.e., we can check efficiently the existence of a successor state s'). A planning problem $\langle PD, q \rangle$ is *proper* if the underlying planning domain *PD* is proper.

2.3 Plans

Given a planning problem $PP = \langle PD, q \rangle$, where q has form (4), a sequence of action sets $\langle A_1, \ldots, A_i \rangle$, $i \geq 0$, is an *optimistic plan* for PP, if a legal transition sequence $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ in *PD* exists such that T establishes the goal, i.e., $\{g_1, \ldots g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$.

However, the existence of an optimistic plan does not guarantee that executing the plan, due to incomplete information and possible alternative transitions, will always lead to the goal. In case of incomplete initial state specification, we must be sure that

[3] This is useful to model dependent actions, i.e. actions which depend on the execution of other actions.

the plan is executable and the goal is reached whatever is the legal initial state consistent with the specification.

An optimistic plan $\langle A_1, \ldots, A_n \rangle$ for PP as previously is a *secure plan*, if for every legal initial state s_0 and legal transition sequence $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$ such that $0 \leq j \leq n$, 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.

A plan $\langle A_1, \ldots, A_n \rangle$ is called *sequential* (or *non-concurrent*) if $|A_j| \leq 1$, for all $1 \leq j \leq n$.

3 The Planning System $\text{DLV}^{\mathcal{K}}$

We have implemented a fully operational prototype supporting the $\mathcal{K}$ language as a frontend on top of the `DLV` system [4]. This frontend is invoked by the command-line option `-FP` of `DLV`. It reads $\mathcal{K}$ files, that is, files as described in the following subsection whose names carry the extension `.plan`, and optionally also background knowledge in the form of stratified datalog and transforms these into the core language of `DLV`. The frontend then invokes the `DLV` kernel and translates possible solutions back into output appropriate for the planning user.

3.1 Programs in $\text{DLV}^{\mathcal{K}}$

A $\mathcal{K}$ program, as implemented in $\text{DLV}^{\mathcal{K}}$, consists of various (optional) sections that start with a keyword followed by a colon. The overall structure of a $\mathcal{K}$ program is as follows:

```
fluents:    <fluent declarations>
actions:    <action declarations>
always:     <rules>
initially:  <init rules>
            [noConcurrency.]    [securePlan.]
goal:       <query>?(i)
```

where <*fluent declarations*> and <*action declarations*> are sequences of declarations as defined in Section 2.1, and <*rules*> (resp., <*init rules*>) is a sequence of causation rules and executability conditions which apply to any (resp., any initial) state.

By default, $\text{DLV}^{\mathcal{K}}$ will look for plans allowing concurrent actions (that is, plans that may contain transitions $\langle s, A, s' \rangle$ with $|A| > 1$). By specifying `noConcurrency.` the user can ask for sequential plans. In the presence of `securePlan.` or the command-line option `-FPsec`, $\text{DLV}^{\mathcal{K}}$ will only compute secure plans, as opposed to the default situation where all (optimistic) plans are computed and the user interactively decides whether to check their safety.[4]

[4] In the current implementation, some syntactic conditions, ensuring proper planning domains, are required to allow the security check of the plans. Optimistic plans are computed under no restriction.

3.2 Language Enhancements

In planning it is often useful to declare some fluents as inertial, which means that these fluents keep their truth values in a state transition, unless explicitly affected by an action. In the AI literature this has been studied intensively and is referred to as the *frame problem* [24, 31].

To allow for an easy representation of this kind of situation, we have enhanced the language by a shortcut `inertial` f`.` which is equivalent to the rule `caused` f `if` **not** $\neg.f$ `after` f`.` where f is a fluent literal.

For reasoning under incomplete knowledge we introduce `total` f`.` which is a shortcut for `caused` f `if` **not** $\neg f$`.` `caused` $\neg f$ `if` **not** f`.` where f is a positive fluent literal. Such `total` statements can be defined in the `always:` and `initially:` sections of a DLV$^{\mathcal{K}}$ program, where in the latter case, only the initial state is completed.

Finally, it may be convenient to explicitly forbid executing an action under specific circumstances. To this end, we introduce `nonexecutable` a `if` B`.` where a is an action atom, as a shortcut for `caused false after` a, B`.`

4 Knowledge Representation in $\mathcal{K}$

4.1 A simple Blocks World instance

Now we are ready to give a short blocks world example. Referring to Figure 1, we want to turn the initial configuration of blocks into the goal state[5] in three steps, where only one move is allowed in each step (i.e., concurrent moves are not permitted).

First of all, the static background knowledge Π consists of the following rules and actions:

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

This program describes the relevant objects in our planning domain.

The action description for the blocks world needs one action `move` and two fluents `on`, and `occupied`. We first assume that the knowledge on the initial state is complete (we know the location of all blocks) and correctly specified. We will then show how to deal with incorrect or incomplete initial state specifications.

```
fluents:  on(B,L) requires block(B), location(L).
          occupied(B) requires location(B).
actions:  move(B,L) requires block(B), location(L).
always:   executable move(B,L) if not occupied(B),
                                  not occupied(L), B <> L.
          inertial on(B,L).
```

[5] This problem illustrates the well known Sussman anomaly [33].

```
          caused occupied(B) if on(B1,B), block(B).
          caused on(B,L) after move(B,L).
          caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
initially: on(a,table). on(b,table). on(c,a).
          noConcurrency.
goal:     on(c,b),on(b,a),on(a,table)? (3)
```

Intuitively, the `executable` rule says that a block `B` can be moved on location `L` $\neq$ `B` if both `B` and `L` are clear (note that the table is always clear, as it is not a block). The causation rules for `on` and `-on` specify the effect of a move. It is worthwhile noting that the totality of these fluents is not enforced. Both `on(x,y)` and `-on(x,y)` may happen to be not true at a given instant of time.

Actually, the rule for `-on` could be replaced by a rule stating: wherever a block is, it is not anywhere else (`caused -on(B,L1) if on(B,L), L <> L1.`). This rule would give us a sharper description of the state making fluent `on` total at every instant of time. Nevertheless, the extra knowledge derived for `-on` from this rule is useless for our goal, as `-on` does not appear in the body of any rule, and `-on(x,y)` is used only to override the inertial property `on(x,y)` after moving `x` from `y`. Thus, we refrain from using the more general rule which would cause a computational overhead (as more inferences are to be done during the computation) without providing relevant benefits.

The execution of this program on DLV$^{\mathcal{K}}$ computes the following:

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

Here, the additional argument in a `move` atom represents the instant of time when the action is executed. Thus, the above plan requires to first move `c` on the table, then to move `b` on top of `a`, and, finally, to move `c` on `b`. It is easy to see that this sequence of actions leads to the desired goal.

4.2 Checking correctness and completeness of the initial state

In the previous example, the knowledge on the block locations in the initial state is complete and correctly specified with respect to domain laws. To ensure that an arbitrary given (partial) knowledge state is not flawed, we should check it properly. We should here verify that every block: (i) is on top of a unique location, (ii) does not have more than one block on top of it, and (iii) is supported by the table (i.e., it is either on the table or on a stack of blocks which is on the table) [20]. To this end, we add the declaration:

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

And we add the following rules in the `initially` section:

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

The resulting $\text{DLV}^{\mathcal{K}}$ program does not compute any plan if the initial state is either incomplete (in the sense that not all block locations are known) or incorrectly specified. Note that, under `noConcurrency`, the action `move` preserves the properties (i),(ii), (iii) above; thus, we do not need to check these properties in all states, if concurrent actions are forbidden.

4.3 Reasoning under incomplete knowledge

Suppose now that there is a further block `d` in Figure 1. The exact location if `d` is unknown, but we know that it is not on top of `c`.

We look for a plan that works on every possible initial state (i.e., no matter if `on(d,b)` or `on(d,table)` holds), and reaches the goal `on(a,c), on(c,d), on(d,b), on(b,table)` in four steps. We modify the program Π and the goal q by adding (i) `-on(d,c)` and `total on(X,Y)` in the `initially` section, and (ii) the command `securePlan`. The execution of this program on $\text{DLV}^{\mathcal{K}}$ computes the following:

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

The plan is clearly valid on all possible initial legal states. Since the effects of all actions are determined, this plan is also secure.

Note that an optimistic 2-step plan exists: `move(c,d,0), move(a,c,1)`, as `c` could initially be on `b`. However, this plan is not secure.

5 Complexity of $\mathcal{K}$

We briefly report some results on the computational complexity of planning in $\mathcal{K}$ for the ground (propositional) case (see [1, 2] and references therein for related results). In particular, we consider deciding existence of a (secure) optimistic plan for a planning program $\langle PD, q\rangle$ and checking whether a given optimistic plan for it is secure.

An optimistic plan can be generated nondeterministically, by guessing the transitions $\langle s_{i-1}, A_i, s_i\rangle$ subsequently, starting from some (nondeterministically generated) legal initial state. Since this requires only polynomial workspace and NPSPACE = PSPACE, the problem is in PSPACE. On the other hand, STRIPS, which is PSPACE-complete [2], can be easily reduced to $\mathcal{K}$. If the number of steps in q is fixed, the complexity decreases because altogether there is fixed number of guesses, which have polynomial size.

Theorem 1. *Deciding whether a given proper ground planning problem $\langle PD, q\rangle$ has an optimistic plan is* PSPACE*-complete, and* NP*-complete if the number of steps in q is fixed.*

Deciding the existence of a secure plan appears to be harder, since it allows us to encode also planning under incomplete initial states. Already recognizing a secure plan is difficult.

Theorem 2. *Given an optimistic plan P and a proper ground planning problem $\langle PD, q\rangle$, deciding whether P is secure is* coNP*-complete. Hardness holds even if the number of steps in q is fixed.*

When looking for a secure plan, these complexities combine, even if the number of steps is bounded.

Theorem 3. *Deciding whether a given proper ground planning problem $\langle PD, q\rangle$ has a secure plan is* NEXPTIME*-complete in general and Σ_2^P-complete, if the number of steps in q is fixed.*

Intuitively, a secure plan can only be built if we know all states reachable after steps $A_1, \ldots, A_i$ so far when the next step A_{i+1} is generated. This requires exponential work space in general. Note that NEXPTIME strictly contains PSPACE, and thus this problem can not be efficiently translated to traditional STRIPS planning. The Σ_2^P-completeness result implies that even if short secure plans can not be efficiently generated by using systems which allow to solve only problems in NP, such as Blackbox [16], CCALC [21], smodels [25], or satisfiability checkers. The hardness relies on the fact that parallel actions are possible. Note that Baral et al. [1] report the related result that deciding in language $\mathcal{A}$ [8] the existence of an, in our terminology, secure sequential plan of polynomially bounded length is Σ_2^P-complete.

Under no concurrency, the complexity remains unaffected in the general case, but is much lower complexity if the plan length is fixed.

Theorem 4. *Deciding whether a given proper ground planning problem $\langle PD, q\rangle$ has a secure sequential plan is* NEXPTIME*-complete in general and D^P-complete, if the number of steps in q is fixed. (D^P is the conjunction of* NP *and* coNP*.)*

Informally, the complexity drops to D^P since in the sequential case only polynomially many candidates for secure plans of fixed length exist, which can be generated in polynomial time. A secure plan exists if (i) some legal initial state exists (which is in NP), and (ii) some of the (polynomially many) candidates has the property that it works on every evolution of every initial state. This check is in coNP for a single candidate and thus, as easily seen, also for all simultaneously. Hence, the problem is in D^P. The hardness for D^P implies, however, that the problem is still not efficiently representable in systems with expressiveness limited to NP.

In arbitrary rather than proper planning domains, security checking as in Theorem 2 is Π_2^P-complete, and deciding the existence of a secure plan of fixed length as in Theorems 3 and 4 is Σ_3^P-complete and Π_2^P-complete, respectively. Intuitively, this is due to an additional nested consistency check. In all other cases, the complexity is the same.

6 Related Work and Conclusion

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 [17], UDTPOP [26], Conformant Graphplan [32], CNLP [27] and CASSANDRA [28] fall in this class. In particular, Buridan, UDTPOP, and Conformant Graphplan can solve secure planning (also called conformant planning) like $\texttt{DLV}^{\mathcal{K}}$. 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 [30] a technique for encoding conditional planning problems in terms of 2-QBF formulas is proposed. The work in [29] 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 [23], 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.

Work similar to ours has been independently reported in [9]. In that paper, the author presents a SAT-based procedure for computing secure plans over planning domains specified in the action language $\mathcal{C}$ [11, 19, 18]. The main differences between our paper and [9] are (i) the different action languages used for specifying planning domains ($\mathcal{C}$ vs $\mathcal{K}$; the former is closer to classical logic, while the latter is more "logic programming oriented" by the use default negation); (ii) the different computational engines underlying the two systems (a SAT Checker vs a DLP system), which imply completely different translation techniques for the implementation. Other recent studies on secure planning are reported in [3]. Experimental evaluations and a comparison of the performance of the various systems for secure planning are on the agenda for future work.

Acknowledgments. The authors were supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

This work has greatly benefited from interesting discussions with and comments of Michael Gelfond, Vladimir Lifschitz, Riccardo Rosati, and Hudson Turner.

References

1. C. Baral, V. Kreinovich, and R. Trejo. Computational complexity of planning and approximate planning in presence of incompleteness. *IJCAI '99*, pp. 948–953.
2. T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *AIJ*, 69:165–204, 1994.
3. A. Cimatti, M. Roveri. Conformant Planning via Model Checking. *Proc. ECP'99*, LNCS, 1999.
4. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. *KR'98*, pp. 406–417.
5. W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. *LPNMR'99*, pp. 177–191.
6. R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *AIJ*, 2(3-4):189–208, 1971.
7. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.
8. M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *JLP*, 17:301–321, 1993.

9. E. Giunchiglia. Planning as Satisfiability with Expressive Action Languages: Concurrency, Constraints and Nondeterminism. *KR '00*.
10. E. Giunchiglia, G. Neelakantan Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *AIJ*, 95:409–443, 1997.
11. E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. *AAAI '98*, pp. 623–630.
12. E. Giunchiglia and V. Lifschitz. Action languages, temporal action logics and the situation calculus. *IJCAI'99 Workshop on Nonmonotonic Reasoning, Action, and Change*.
13. S. Hanks and D. McDermott. Nonmonotonic Logic and Temporal Projection. *AIJ*, 33(3):379–412, 1987.
14. L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing, concurrency, and exogenous events: Logical framework and implementation. *KR'2000*.
15. G. Neelakantan Kartha and V. Lifschitz. Actions with indirect effects. *KR '94*, pp. 341–350.
16. H. Kautz and B. Selman. Unifying SAT-based and Graph-based Planning. *IJCAI '99*, pp. 318–325.
17. N. Kushmerick, S. Hanks and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2), pp. 239–286, 1996.
18. V. Lifschitz and H. Turner. Representing transition systems by logic programs. *LPNMR'99*, pp. 92–106.
19. V. Lifschitz. Action Languages, Answer Sets and Planning. *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999.
20. V. Lifschitz. Answer set planning. *ICLP '99*, pp. 23–37.
21. N. McCain. Using the Causal Calculator with the $\mathcal{C}$ Input Language. Unpublished draft, 1999.
22. J. McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.
23. N. McCain, H. Turner. Fast Satisfiability Planning with Causal Theories. *KR'98*.
24. J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*.
25. I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *AMAI*, 25(3-4):241–273, 1999.
26. M. Peot. Decision-theoretic planning. PhD thesis, Dept. of Engineering and Economic Systems, Stanford University, 1998.
27. M. Peot and D. Smith. Conditional nonlinear planning. In *Proc. of AIPS'92*, pp. 189–197, 1992.
28. L. Pryor and G. Collins. Planning for contingencies: a decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.
29. R. Reiter. Open world planning in the situation calculus. To appear in *Proc. of AAAI 2000*.
30. J. Rintanen. Constructing Conditional Plans by a Theorem-Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
31. S.J. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice-Hall, Inc., 1995.
32. D. Smith and D. Weld. Conformant Graphplan. In *Proc. of AAAI'98*, 1998.
33. G.J. Sussman. The Virtuous Nature of Bugs. *Readings in Planning*, chapter 3. Morgan Kaufmann, 1990. Originally written 1974.
34. H. Turner. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach, *JLP*, 31:245–298, 1997.
35. J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1989.
36. M. Veloso. *Nonlinear problem solving using intelligent causal-commitment*. Technical Report CMU-CS-89-210, Carnegie Mellon University, 1989.