Declarative Planning and Knowledge Representation in an Action Language

Thomas Eiter, Wolfgang Faber, Gerald Pfeifer, Axel Polleres

Institut für Informationssysteme, Technische Universität Wien, 1040 Wien, Austria

Email: {eiter,faber,gerald,axel}@kr.tuwien.ac.at

Abstract

This chapter introduces planning and knowledge representation in the declarative action language $\mathcal{K}$. Rooted in the area of Knowledge Representation & Reasoning, action languages like $\mathcal{K}$ allow to formalize complex planning problems involving non-determinism and incomplete knowledge in a very flexible manner. By giving an overview of existing planning languages and comparing these against our language, we aim on further promoting the applicability and usefulness of high-level action languages in the area of planning. As opposed to previously existing languages for modeling actions and change, $\mathcal{K}$ adopts a logic programming view where fluents representing the epistemic state of an agent might be true, false or undefined in each state. We will show that this view of knowledge states can be fruitfully applied to several well-known planning domains from the literature as well as novel planning domains. Remarkably, $\mathcal{K}$ often allows to model problems more concisely than previous action languages. All the examples given can be tested in an available implementation, the $\texttt{DLV}^{\mathcal{K}}$ planning system.

1 Introduction

While most existing planning systems rely on "classical" planning languages like STRIPS (Fikes & Nilsson, 1971) and PDDL (Ghallab et al., 1998; Fox & Long, 2003), the last few years have seen the development of action languages which provide expressive and flexible tools for describing the relation between fluents and actions. Action languages have received considerable attention in the Knowledge Representation & Reasoning community and their formal properties (complexity, etc.) have been studied in depth. Less efforts have been spent on how to use the constructs offered by these languages for problem solving.

In this chapter, we tackle this shortcoming and elaborate on knowledge representation & reasoning with action languages which are significantly different from the strict operator-based frameworks of STRIPS and PDDL.

To that end, we present the planning language $\mathcal{K}$ (Eiter, Faber, Leone, Pfeifer, & Polleres, 2004) via its realization in the DLV$^{\mathcal{K}}$ planning system (Eiter, Faber, Leone, Pfeifer, & Polleres, 2003a).[1] We discuss knowledge representation issues and provide both general guidelines for encoding action domains and detailed examples for illustration.

The language $\mathcal{K}$ significantly stands out from other action languages in that it offers proven concepts from logic programming to represent knowledge about the action domain. This includes the distinction between negation as failure (or default negation) and strong negation: In $\mathcal{K}$, it is possible to reason about states of knowledge, in which a fluent might be true, false or unknown, and states of the world, in which a fluent is either true or false. In this way, we can deal with uncertainty in the planning world at a qualitative level, in which default and plausibility principles might come into play when reasoning about the current or next

[1] `http://www.dbai.tuwien.ac.at/proj/dlv/K/`

state of the world, the effects of actions, etc. This allows different approaches to planning, including traditional planning (with information and knowledge treated in a classical way) and planning with default assumptions or forgetting.

2 States, Transitions, and Plans

Intuitively, a *planning problem* consists of the following task: Given an *initial state*, several *actions*, their *preconditions* and *effects*, find a sequence of actions (viz. a *plan*) to achieve a state in which a particular *goal* holds. In the following, we will describe and discuss these concepts in more detail.

2.1 Fluents and States

Fluents represent basic properties of the world which can change over time. They are comparable to first-order predicates or propositional assertions. *States* are collections (usually sets) of fluents, each of which is associated with a truth-value.

We distinguish between so called *world states* and *knowledge states*: The current state of the world wrt. a set of fluents $F = \{f_1, \ldots, f_n\}$ can be defined as a function $s : F \to \{true, false\}$, i.e., a set of literals which contains either f or $\neg f$ for any $f \in F$. From an agent's point of view, states can also be seen as partial functions s', i.e., consistent sets of fluent literals, where for a particular fluent $f \in F$ neither f nor $\neg f$ may hold. The state s' then only consists of the subset of s which is *known*; it is a *state of knowledge*.

Note that this view of the epistemic state of an agent differs from other approaches where incomplete knowledge states are defined as the set of all possible worlds an agent might be in (Son & Baral, 2001; Bonet & Geffner, 2000; Bertoli, Cimatti, Pistore, & Traverso, 2001). Such sets of (compatible) world states are often referred to as *belief states*. Knowledge states as described here can, to some extent, be viewed as assigning a value only to those fluents having the same value in all states of a corresponding belief state. When working with knowledge states ones usually does not consider any relationship to world states, though.

Both knowledge states and belief states can (to a certain degree) be modeled in the language $\mathcal{K}$ discussed in this text.

We remark that the terminology concerning knowledge and belief states is not always consistent in the literature. For example, Son and Baral use the term "states of knowledge" when they describe a set of reachable worlds in a Kripke structure (Son & Baral, 2001). This amounts to what we call "belief states" in our terminology. An in-depth discussion of the terms "knowledge" and "belief" can be found in (Hintikka, 1962).

A useful generalization is to allow not only Boolean fluents, but also *multi-valued fluents* (Giunchiglia, Lee, Lifschitz, & Turner, 2001) which take a certain value of a specific (finite) *domain* in each state. A state can then be seen as a set of functions which assigns to each fluent f a value of its domain D_f, Boolean fluents having the domain $\{true, false\}$. Such a multi-valued fluent f with finite domain $D_f = \{d_1, \ldots, d_n\}$ can be readily "emulated" by a set of Boolean fluents $f_{d_1}, \ldots f_{d_n}$ plus constraints which prohibit concurrent truth of two distinct f_{d_i}, f_{d_j}.

2.2 Actions, Transitions, and Plans

Actions represent dynamic momenta of the world, and their execution can change the state of the world (or knowledge). *Transitions* are atomic changes, represented by a previous state, a set of actions, and a resulting state. Implicitly, such a definition incurs the simplifying but commonly used abstraction that all actions have unique duration and the assumption that all effects materialize in the successor state (i.e., a discrete notion of time is employed). Given these assumptions, a *plan* is a sequence of n sets of actions, which is backed by *trajectories* (sequences of $n + 1$ states), such that interleaving these states and the sets of actions yields a chaining of transitions and the last state in the trajectory satisfies the goal.

In order to define the semantics of such transitions, the dynamic properties of fluents and actions are to be represented using an appropriate formalism. Key issues for such a formalism are how it deals with

- effects of actions,
- executability of actions (known as *qualification problem*),
- indirect effects, or interdependencies of fluents (the so-called *ramification problem*),
- the fact that usually fluents remain unchanged in a transition (known as the *frame problem* (McCarthy & Hayes, 1969; Russel & Norvig, 1995))

As we will see on the example of language $\mathcal{K}$, action languages provide an expressive means to deal with these issues.

3 Action Language $\mathcal{A}$ and Descendants

In the planning community, the development of formal languages is driven by a focus on special-purpose algorithms and systems, where ease of structural analysis of the problem description at hand is a main issue. On the other hand, expressive languages for formalizing actions and change in a more general context have emerged from the field of knowledge representation.

One of the first of these languages was $\mathcal{A}$ (Gelfond & Lifschitz, 1993) which essentially represents the propositional fragment of Pednault's ADL (Pednault,

1989) formalism, but offers a more "natural" logic-based language with constructs for the formalization of actions and change rather than a formal description of operators.

$\mathcal{A}$ has been extended in various ways, both syntactically and semantically, for example by constructs allowing to express ramifications, sensing actions, explicit inertia, action costs, and more.

In the sequel, we will describe the most important features of the language $\mathcal{A}$ and some important extensions thereof. In particular, we will focus on the language $\mathcal{K}$ (in a separate section), which we will use in the remainder of the paper.

Action Language $\mathcal{A}$

From the viewpoint of expressiveness $\mathcal{A}$ (Gelfond & Lifschitz, 1993) essentially represents the propositional fragment of Pednault's ADL, i.e., STRIPS enriched with conditional effects. Effects and preconditions are expressed by causation rules

a `causes` l `if` F.

where a is an action name, l is a fluent literal, and F is a conjunction of fluent literals. An action description D consists of a set of such propositions.

It should be noted that (Gelfond & Lifschitz, 1993) and (Gelfond & Lifschitz, 1998) provide differing semantics for $\mathcal{A}$. The semantics of (Gelfond & Lifschitz, 1998) is as follows: States are boolean valuations of fluents. Let $E(A, s)$ be the set of effects of action A wrt. the state s, i.e. all l of causation rules for A s.t. F is satisfied in s. Then $\langle s, A, s' \rangle$ is a valid transition if $E(A, s) \subseteq s' \subseteq E(A, s) \cup s$. Intuitively, the successor state s' must contain all action effects and can contain fluent values of s (and no other fluent values), i.e. s' contains all values of s which are not overridden by action effects. For each pair (s, A) there is at most one s'.

Example 1 Executability and effects of moving block b to block a in the well-known Blocks World example could be described in $\mathcal{A}$ as follows:

$\texttt{move}_{b,a}$ `causes f if` $\texttt{blocked}_a$.

$\texttt{move}_{b,a}$ `causes` $\neg$`f if` $\texttt{blocked}_a$.

$\texttt{move}_{b,a}$ `causes f if` $\texttt{blocked}_b$.

$\texttt{move}_{b,a}$ `causes` $\neg$`f if` $\texttt{blocked}_b$.

$\texttt{move}_{b,a}$ `causes` $\texttt{on}_{b,a}$.

$\texttt{move}_{b,a}$ `causes` $\neg\texttt{blocked}_c$ `if` $\texttt{on}_{b,c}$.

$\mathcal{A}$ does not provide any means for representing executability in an explicit way, but one can model the fact that for a state s, in which some condition holds, and an action A no consistent s' exists such that $\langle s, A, s' \rangle$ is a valid transition, rendering action A nonexecutable in such a state s. In the example above, the first four rules encode a nonexecutability condition for $\texttt{move}_{b,a}$ by enforcing inconsistency on the auxiliary fluent `f`. The last two rules encode an unconditional and a conditional action effect, respectively. $\diamond$

Extensions of $\mathcal{A}$

Language $\mathcal{AR}$ A further step in the development of action languages was the language $\mathcal{AR}$ (Giunchiglia, Kartha, & Lifschitz, 1997), which extends $\mathcal{A}$ by allowing to model indirect effects by introducing constraints

`always` F.

where F is a propositional formula. Valid states are those for which all constraints are satisfied. $\mathcal{AR}$ also allows for arbitrary propositional fluent formulae C and F in causal rules of the form

a `causes` C `if` F.

and is capable of modeling nondeterministic actions by statements

a `possibly changes` l `if` F.

In addition, $\mathcal{AR}$ also allows for multi-valued fluents and non-inertial fluents.

The semantics relies on the principle of "minimal change": Let $Res_0(A, s)$ denote the set of states in which C holds for any causation rule for A s.t. F holds in s. Then, $\langle s, A, s'\rangle$ is a valid transition if the changes in $s' \in Res_0(A, s)$ are subset-minimal wrt. inertial fluents and nondeterministic action effects. It is important to note that `always` constraints do not give causal explanations and therefore not all indirect effects can be modeled (see Example 2 below).

Language $\mathcal{B}$ The language $\mathcal{B}$ (Gelfond & Lifschitz, 1998) extends the language $\mathcal{A}$ by so-called "static laws"

l `if` F.

where l is a fluent literal and F is a conjunction of fluent literals. As opposed to `always` in $\mathcal{AR}$ the semantics of static laws can give causal explanations.

The semantics of $\mathcal{B}$ is based on the principle of "minimal change" and causality. It incurs the operator $Cn_Z(s)$, which is defined on a set of static laws Z and a set of literals s, producing the smallest set of literals that contains s and satisfies Z. Then, $\langle s, A, s'\rangle$ is a valid transition if $s' = Cn_Z(E(A, s) \cup (s \cap s'))$, i.e. s' is stable when action effects $E(A, s)$ and unchanged fluents $s \cap s'$ are minimally extended to satisfy the static laws.

As an example, consider a simplified version of Lin's Suitcase (Lin, 1995):

Example 2 Assume we have a spring-loaded suitcase with two latches. Unlocking a latch turns its position to "up", and as an indirect effect the suitcase opens as soon as both latches are up. This can be modeled by the following $\mathcal{B}$ action description:

$unlock_1$ `causes` up_1.

$unlock_2$ `causes` up_2.

$open$ `if` up_1, up_2.

Consider an initial state $s = \{up_1, \neg up_2, \neg open\}$ and action $a = unlock_2$. For $s' = \{up_1, up_2, open\}$ we have $Cn_Z(E(A,s) \cup (s \cap s')) = Cn_Z(\{up_2\} \cup \{up_1\}) = \{up_1, up_2, open\} = s'$, and hence $\langle s, a, s' \rangle$ is a valid transition in $\mathcal{B}$. It can be verified that s' is the only valid successor state for s and a.

When we would replace the final static law by the $\mathcal{AR}$ constraint

`always` $up_1 \wedge up_2 \Rightarrow open$.

we obtain $Res_0(a, s) = \{s', s'', s'''\}$, where $s'' = \{\neg up_1, up_2, open\}$ and $s''' = \{\neg up_1, up_2, \neg open\}$ (i.e. $Res_0(a, s)$ contains all valid states in which up_2 holds). The changed fluents (wrt. s) for s' are $\{open, up_2\}$, for s'' $\{open, up_1, up_2\}$, and for s''' $\{up_1, up_2\}$, so by subset-minimality $\langle s, a, s' \rangle$ and $\langle s, a, s''' \rangle$ are valid transitions. In s''', $\neg up_1$ lacks a causal explanation (Why did it change its value wrt. s?), and hence $\langle s, a, s''' \rangle$ is intuitively not expected to be a valid transition. Note that both s' and s''' satisfy the criterion for "minimal change", but in the semantics of $\mathcal{AR}$ causal explanations among fluents are not considered. $\diamond$

Language $\mathcal{A}_K$ An extension of the action languages $\mathcal{AR}$ and $\mathcal{A}$ to formalize sensing actions was proposed by Son and Baral with language $\mathcal{A}_K$ (Son & Baral, 2001). $\mathcal{A}_K$ provides propositions of the form a `determines` f, which intuitively states that after executing action a, the value of fluent f is known. This concept of knowledge differs from what we referred to as knowledge states in Section 1 and which we will further discuss in the following.

Action Language $\mathcal{C}$

The most recent and evolved languages in this line of action languages are the languages $\mathcal{C}$ (Giunchiglia & Lifschitz, 1998) and its extension $\mathcal{C}$+ (Giunchiglia, Lee, Lifschitz, McCain, & Turner, 2004). $\mathcal{C}$ is similar to $\mathcal{B}$ in that it distinguishes between static and dynamic laws. It is in some ways more expressive than $\mathcal{B}$ and $\mathcal{AR}$ though, strictly speaking, not a superset of either.

$\mathcal{C}$ action descriptions consist of a set of causation laws c of the form

$$\texttt{caused}\ F\ \texttt{if}\ G\ \texttt{after}\ H. \tag{1}$$

where the `after`-part is optional: c is called *static* if it has no `after`-part and *dynamic* otherwise. These rules are more flexible than the previous approaches in that F and G are arbitrary propositional formulae over fluent literals and H is a propositional formula over fluent and action literals. Furthermore, constraints and qualifications can be expressed via $F = f \wedge \neg f$ which is written as

$$\texttt{caused}\ \perp\ \texttt{if}\ G\ \texttt{after}\ H.$$

These rules encode inconsistency similar to constraints in logic programming.

An action description D consists of static and dynamic causation laws. Its semantics is given by the following definition of *causally explained* transitions:

A transition $\langle s, a, s' \rangle$ is causally explained according to D if its resulting state s' is the only interpretation that satisfies all rules caused in this transition, where a formula F is caused if it is

- the head of a static law (1) from D such that $s' \models G$ or
- the head of a dynamic law (1) from D such that $s' \models G$ and $s \cup a \models H$

Note that this allows for nondeterministic actions and valid transitions $\langle s, a, s' \rangle$, $\langle s, a, s'' \rangle$ with $s' \neq s''$. The definition of causally explained transitions is closely related to causal theories as defined by (McCain & Turner, 1997) and the underlying concept of causal explanation (Lifschitz, 1997).

Remarkably, inertia (i.e., that a fluent remains unchanged unless explicitly stated otherwise) has to be explicitly encoded in $\mathcal{C}$; frame axioms are not implicit like in the previously discussed approaches. However, they can be conveniently expressed by the following macro:

$$\texttt{inertial}\ F. \quad \Leftrightarrow \quad \texttt{caused}\ F\ \texttt{if}\ F\ \texttt{after}\ F.$$

A further macro that allows for modeling qualifications of actions is

$$\texttt{nonexecutable}\ A\ \texttt{if}\ G. \quad \Leftrightarrow \quad \texttt{caused}\ \bot\ \texttt{after}\ A \wedge G.$$

$\mathcal{C}$ and $\mathcal{K}$ (which will be presented below) share several distinct features such as concurrent actions, the intuitive modeling of state constraints, action qualifications, inertia, non-determinism of actions, and incomplete initial knowledge.

Action Language $\mathcal{C}$+

A recent extension of $\mathcal{C}$ called $\mathcal{C}$+ allows for multi-valued, additive fluents which can be used to encode resources and allows for a more compact representation of several practical problems (Giunchiglia et al., 2001, 2004).

4 Action Language $\mathcal{K}$

We next give an overview of the language $\mathcal{K}$ as implemented in the $\text{DLV}^{\mathcal{K}}$ planning system, cf. Footnote 1 of Section 1. Details and the formal definition of the semantics of $\mathcal{K}$ can be found in (Eiter et al., 2004). Since we will use $\mathcal{K}$ throughout the rest of this chapter, we consider an example from the well-known Blocks World domain in detail.

The distinguishing feature of the language $\mathcal{K}$ wrt. to the action languages considered so far is the notion of incomplete states and the ability to reason about this incompleteness. In particular, a state may either contain a fluent f, its strong negation $-f$, or it may say nothing about f. Causal rules may contain default negated fluent literals `not f` which hold if either `–f` holds or nothing is said about `f` in the respective state. This is often referred to as negation as failure.

A $\mathcal{K}$ *planning problem* is a pair $\mathcal{P} = \langle PD, q \rangle$ of a *planning domain PD* (informally, the world of discourse) and a query q, which specifies the goal. A planning problem is represented as a combination of *background knowledge* Π, provided as a function-free logic program (possibly with negation) admitting exactly one answer set, and a *program* of the following general form:

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

where the first four sections 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 the form $PD = \langle \Pi, \langle D, R \rangle \rangle$, where the declarations D are given by F_D and A_D and the rules R by C_R and I_R.

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

$$p(X_1, \ldots, X_n) \; \texttt{requires} \; t_1, \ldots, t_m \qquad (2)$$

where p is a fluent or action predicate of arity $n \geq 0$, and the t_i are classical literals (i.e., an atom α or its strong negation $-\alpha$), 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 instances of fluents and actions which are "supported" by some ground instance of a declaration, where the `requires` part is true, need to be considered.

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

$$\begin{array}{l} \texttt{caused}\ f\ \texttt{if}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l \\ \qquad\qquad \texttt{after}\ a_1, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n \end{array} \tag{3}$$

where f is either a classical literal over a fluent or `false` (representing inconsistency), the b_is are classical literals over fluents and background predicates, and the a_js are positive action atoms or classical literals over fluents and background predicates. Informally, a rule of the form (3) states that f is true in the new state reached by (simultaneously) executing 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 facts.

The `always`-section also contains *executability conditions* for actions

$$\texttt{executable}\ a\ \texttt{if}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l \tag{4}$$

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

The `initially`-section specifies conditions that hold in any initial state (which is not unique in general). They have the form of causation rules, as described above, without the `after` part.

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

$$g_1, \ldots, g_m, \texttt{not}\ g_{m+1}, \ldots, \texttt{not}\ g_n \ ?\ (i) \tag{5}$$

where $g_1, \ldots, g_n$ are ground fluent literals, $n \geq m \geq 0$, and $i \geq 0$ 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 (i.e., literals corresponding to predicates from the background knowledge): each variable occurring in a default negated type literal has to occur in at least one positive type literal or dynamic literal. Note that this safety restriction does not apply to action and fluent literals whose variables are already safe due to their respective declarations.

Example 3 (Blocks World) Let us consider the Blocks World, one of the best-known scenarios in AI Planning. Here, the goal is to build stacks of blocks which are located on a table. The planning problem consists of an initial configuration of blocks and a (probably partly specified) goal configuration. The only action is *moving* a block x to a location l, i.e., onto the table or on top of another block which is clear, and we allow parallel moves. Figure 1 shows a simple instance.

A $\mathcal{K}$ encoding PD_{bw1} for this domain is shown in Figure 2. This encoding guarantees serializability, which means that parallel actions are non-interfering and could be executed in any sequential order; each parallel plan can be arbitrarily "unfolded" to a sequential plan.

We use three fluents: $\texttt{on}(\texttt{B}, \texttt{L})$ states that block `B` resides at location `L`, fluent $\texttt{blocked}(\texttt{B})$ indicates that the capacity of a block `B` to hold further blocks is exhausted, and fluent $\texttt{moved}(\texttt{B})$ holds directly after `B` was moved. There is a single action $\texttt{move}(\texttt{B}, \texttt{L})$, which represents moving a block `B` to some location `L` (and implicitly removes it from its previous location). Finally, we add background knowledge which defines the six blocks and the table as a location:

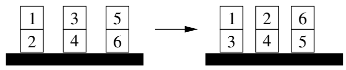

Figure 1: A Blocks World instance

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


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

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

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

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

          caused on(B,L) if not moved(B) after on(B,L).
```

Figure 2: $\mathcal{K}$ encoding for the Blocks World domain PD_{bw1}

The configurations of blocks shown in Figure 1 are expressed by extending PD_{bw1}, the program in Figure 2, as follows, yielding $\mathcal{P}_{bw1(l)}$:

$$
\begin{array}{ll}
\texttt{initially}: & \texttt{on}(1,2).\ \ \texttt{on}(2,\texttt{table}).\ \ \texttt{on}(3,4).\ \ \texttt{on}(4,\texttt{table}). \\
& \texttt{on}(5,6).\ \ \texttt{on}(6,\texttt{table}). \\
\texttt{goal}: & \texttt{on}(1,3),\ \ \texttt{on}(3,\texttt{table}),\ \ \texttt{on}(2,4),\ \ \texttt{on}(4,\texttt{table}), \\
& \texttt{on}(6,5),\ \ \texttt{on}(5,\texttt{table})\ \ ?\ \ (l)
\end{array}
$$

Here, l is a non-negative integer representing the plan length. Note that only positive knowledge is stated for `on` and `blocked`; this is because our modeling assumes that fluents are interpreted under the closed world assumption (CWA) (Reiter, 1978): If some fluent does not hold, we assume that it is false. Note that CWA is not a feature in the syntax or semantics of $\mathcal{K}$; it is just a modeling assumption in this example.

The values of the fluent `blocked` in the initial state are not specified explicitly; rather they are obtained from a general rule that applies to any state, and thus is part of the `always`-section: the first rule there says that a block `B` (but not the `table`) is blocked if another block is on it. Observe that the fluent `moved` can never hold in the initial state.

Next we specify when an action `move(B, L)` is executable. This is achieved by a combination of `executable` and `nonexecutable` statements defining defaults and exceptions, respectively. A `move` is executable, if the positive executability condition holds and all negative executability conditions fail. In our case, a block can be moved to any location except onto itself, with several exceptions: (i) blocks which are `blocked` cannot be moved; (ii) a block can not be moved to a `blocked` block; (iii) a block can not be moved on top of another block which is moved at the same time; (iv) two different blocks can not be moved to the same block at once; and (v) a block can not be moved to two different locations at once.

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 states that `moved(B)` holds directly after a block `B` has been moved.

The last rule is an explicit frame axiom for `on`. It states that blocks which have not been moved remain where they were before. Such frame axioms are not included for `blocked` and `moved`, because `blocked` follows as a ramification from `on`, and `moved` is supposed to hold only right after a respective `move` action occurred. $\diamond$

The semantics of a $\mathcal{K}$ planning domain *PD* is defined in terms of legal states and state transitions. Informally, a *state* is any consistent set 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 if causal rules are read as logic programming rules under the answer set semantics (Gelfond & Lifschitz, 1991). A *state transition* is a triple $\langle s, A, s' \rangle$ where s and s' are states and A is a set of legal action instances in *PD*, i.e., action instances that respect the typing information. A transition is *legal*, if the action set A is *executable wrt.* s, i.e., each action `a` in A is the head of a clause (4) whose body is true, and s' satisfies all causal rules (3) from the `always`-section whose `after` part is true with respect to s and A.

An *optimistic plan* for a goal $g_1, \ldots, g_m,$ `not` $g_{m+1}, \ldots,$ `not` g_n is a sequence of action sets $\langle A_1, \ldots, A_i \rangle$, $i \geq 0$, 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 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 $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$. T is called *trajectory*, and an optimistic plan of length i is a *solution* to the planning problem $\mathcal{P} = \langle PD, q \rangle$, q has the form (5).

Example 4 (Blocks World (cont'd)) *If we instantiate the plan length* l *by* 2 *in* $\mathcal{P}_{bw1(l)}$*, we get a plan which involves six* `move` *actions:*

$$P_2 = \langle\, \{\mathtt{move}(1,\mathtt{table}), \mathtt{move}(3,\mathtt{table}), \mathtt{move}(5,\mathtt{table})\},$$
$$\{\mathtt{move}(1,3), \mathtt{move}(2,4), \mathtt{move}(6,5)\}\,\rangle$$

By unfolding these steps, this plan gives rise to similar plans of length $l = 3, \ldots, 6$. For $l = 3$, we can also find the following plan comprising only five actions:

$$P_3 = \langle\, \{\mathtt{move}(3,\mathtt{table})\},\ \ \{\mathtt{move}(1,3), \mathtt{move}(5,\mathtt{table})\},\ \ \{\mathtt{move}(2,4), \mathtt{move}(6,5)\}\rangle$$

5 Knowledge Representation

We will now consider different aspects of knowledge representation in $\mathcal{K}$ and the DLV$^{\mathcal{K}}$ planning system. First, we discuss some particular constructs which facilitate expressing some commonly occurring concepts. Subsequently, we focus on the handling of incomplete knowledge and nondeterminism, differentiating various scenarios and suggesting techniques for modeling these, by providing examples. We then briefly cover an extension of $\mathcal{K}$ which allows to express action costs and compute optimal plans and conclude by giving some basic principles for knowledge representation in $\mathcal{K}$ as well as an overview of features and pitfalls.

5.1 Basic Features

Let us recall how the dynamic behavior was specified in the Blocks World program of Figure 2. The basic structures are causal rules and executability statements.

Direct Action Effects An important use of causal rules is the specification of direct action effects. If an action a has the effect that a fluent `f` holds, this can be expressed by `caused f after a`.

Qualification Problem The constructs `executable` and `nonexecutable` are used to express and solve the *qualification problem*, i.e., the problem of determining whether an action is executable in a particular state. By default an action does not qualify for execution. One can grant this qualification by specifying `executable` clauses (which can be as general as stating that the action is always executable). Dually, these qualifications can be narrowed down by specifying `nonexecutable` conditions. In our example, `move` is the only action. It is first made executable for all cases where its first and second arguments differ, and subsequently cases are excluded by using `nonexecutable` statements. Thus, $\mathcal{K}$ offers a flexible means for dealing with the qualification problem by offering constructs for specifying executability conditions and exceptions to them. Using $\mathcal{K}$, one can also create more complex hierarchies of exceptions by using auxiliary fluents and negation as failure, though no first-class syntactic constructs for doing so are provided in the language.

Ramification Problem Let us now turn to the *ramification problem*, i.e., the problem that some fluents may depend on other fluents rather than being directly affected by actions; sometimes this is also referred to as *indirect effects*. In $\mathcal{K}$, indirect effects are also dealt with by causal rules: If a fluent `f` causes another fluent `g`, this is expressed by `caused g if f.`, where the use of `if` indicates simultaneity. In Figure 2, the fluent `blocked` depends directly on the fluent `on` and indirectly on the effects of the action `move`.

Frame Problem The handling of the *frame problem*, i.e., the fact that fluents usually do not change their value, unless there is a direct or indirect cause for a change, leaves room for improvement. Indeed, in the program of Figure 2, we declare a fluent `moved` which indicates whether a block was just moved. Additionally, there is a causal law using this fluent which states that the fluent `on` should remain unchanged for fluents not affected by a move action. While not incorrect, this representation is not easily extensible. In particular, for each pair of actions and fluents at least one such statement should be included to describe unaffectedness conditions (Shanahan, 1997), whereas in general, one would rather like to express default assumption on fluents.

$\mathcal{K}$ directly supports inertia, that is the assumption that a fluent remains unchanged by default. Unlike in other languages, inertia is not implicitly assumed on all fluents; rather a fluent, say `f`, has to be declared inertial by `inertial f`.

What we have left open so far is how to express exceptions to the inertial default. To this end we consider the concept of strong negation, which we have briefly mentioned, but not used in an example so far. Concerning an inertial fluent `f`, the exception to its inertia is its strong negation `–f`. (Intuitively, strong negation `–f` says that we explicitly know that `f` does not hold, whereas `not f` states that we do not know that `f` holds and thus can implicitly assume that it does not.) Using this, `inertial f` can alternatively be written as `caused f if not –f after f`. Indeed, `inertial f` is implemented as such a macro in DLV$^{\mathcal{K}}$. Contrast this with the respective macro in the language $\mathcal{C}$, which is `caused f if f after f`: while in $\mathcal{K}$, `f` is assumed to hold in lack of any information to the contrary, $\mathcal{C}$ takes the view that `f` explains itself after it was true in the previous stage.

Note that in $\mathcal{K}$, inertia may also be defined on a truly negated fluent `–f` by the statement `inertial –f`, to which `f` acts as exception.

Coming back to the Blocks World domain, we can modify the program of Figure 2 by eliminating the fluent `moved`, replacing the pseudo-inertial rule by an `inertial` statement, and explicitly stating that a block is no longer on a particular location if it was just moved away. The resulting program PD_{bw2} is depicted in Figure 3. The planning problem $\mathcal{P}_{bw2(l)}$, obtained by replacing PD_{bw1} by PD_{bw2}

```
fluents: on(B,L) requires block(B), location(L).
         blocked(B) requires block(B).
actions: move(B,L) requires block(B), location(L).
always:  caused blocked(B) if on(B1,B).
         executable move(B,L) if B <> L.
         nonexecutable move(B,L) if blocked(B).
         nonexecutable move(B,L) if blocked(L).
         nonexecutable move(B,B1) if move(B1,L).
         nonexecutable move(B,L) if move(B1,L), B <> B1, block(L).
         nonexecutable move(B,L) if move(B,L1), L <> L1.

         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).
```

Figure 3: Alternative DLV$^{\mathcal{K}}$ program for the Blocks World domain PD_{bw2}

in $\mathcal{P}_{bw1(l)}$, has the same plans as $\mathcal{P}_{bw1(l)}$.

Negation and Closed World Assumption We point out that the only negative information in this encoding is the exception for the inertia of on. Indeed, the encoding focuses on the relevant information. Any state reachable by a legal transition only consists of positive fluents on(B, L) and blocked(L) describing a "relevant clipping" of knowledge. We do not care which blocks are currently unblocked or wherever a block is *not* located, and indeed $\mathcal{K}$ does not require to completely specify truth values for all fluents, as in this example the fluents are interpreted under a *closed world assumption* (CWA), meaning that fluents which are not explicitly caused are considered false. Note that the CWA is a modeling decision (like a programming technique), and indeed the next sections will show examples where the CWA is not applicable. Also note that one could reify the CWA by including a rule `caused -on(X,Y) if not on(X,Y).`; doing so eliminates the computational benefits of CWA, however.

5.2 Planning with Incomplete Knowledge

Let us now focus on domains with inherent nondeterminism and incomplete knowledge. In this context incomplete knowledge is a lack of knowledge in the problem specification rather than incompleteness resulting from model abstraction, focusing onto the relevant part of the specification. E.g. in the Blocks World domain we did not represent some knowledge which was irrelevant for the problem at hand, resulting in incomplete states; the planning domain was sufficiently specified, though, and did not admit nondeterminism.

The forms of incompleteness we will consider now are of a more fundamental nature, as relevant knowledge is missing, usually resulting in nondeterminism. In particular, we will consider three main sources of nondeterminism:

1. incomplete initial states;
2. nondeterministic actions;
3. nondeterministic evolutions.

We will exemplify each of them in some domain encoding below. Source 1 deals with scenarios where some aspects of the initial state are unknown. This entails a comparatively light form of nondeterminism, since it is confined to a single point in time. The Square domain will serve as an example for such a setting. Source 2 refers to actions with multiple alternative outcomes, where the knowledge about action effects is incomplete. This form of nondeterminism potentially affects all points in time. In the Paint example we will tackle such a problem. Finally, for source 3 the environment itself can change nondeterministically. Affected fluents may change values without actions causing this change, meaning that there are dynamics which are not under the agent's control. The Ring domain comprises such evolutions. Summarizing, these three sources are uncertainties on the initial state, the action effects, and the world evolution, respectively. Since those uncertainties are not associated with probabilities and thus are not quantified in our framework, we refer to them as *qualitative uncertainties*. Indeed this is a common setting, as probabilities are often hard to obtain or simply unknown.

In the context of nondeterministic planning problems, *optimistic* plans can establish the goal in *some* nondeterministic evolutions, while so-called *secure* or *conformant* plans (Goldman & Boddy, 1996; Smith & Weld, 1998) establish the goal for *all* possible evolutions, i.e., the plan is executable from every initial state and eventually establishes the goal in any possible evolution. $\mathcal{K}$ and the $\texttt{DLV}^{\mathcal{K}}$ system allow to specify such domains, as demonstrated below, and support conformant plan generation. For details we refer to (Eiter et al., 2004).

5.2.1 Square

The Square domain is about self-location of a robot which moves in a wall-bounded $n \times n$ grid. The robot can move in four directions (up, down, left, right) and its initial position is unknown. Moving towards a wall has no effect, and the robot stays in its position. The problem of finding a conformant plan for reaching the corner position $(0, 0)$ is referred to as SQUARE(n) in the literature (Bonet & Geffner, 2000; Parr & Russel, 1995). SQUARE(4) with one of the possible initial states – the robot is at position $(2, 1)$ – is illustrated in Figure 4.

A $\mathcal{K}$ encoding for this problem is as follows, where Π_{square} consists of facts $\mathtt{index(0)}.\ldots\mathtt{index(n-1)}.$ and $\mathtt{next(0,1)}.\ldots\mathtt{next(n-2,n-1)}.$

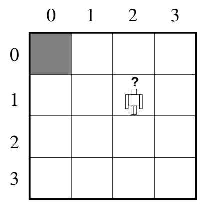

Figure 4: SQUARE(4)

```
fluents :    atX(P) requires index(P).  atY(P) requires index(P).
             anywhere.

actions :    up. down. left. right.

always :     executable up.  executable right.
             executable left.  executable down.
             nonexecutable up if down.
             nonexecutable left if right.
             inertial atX(X).  inertial atY(Y).

             caused atY(Y) after atY(Y1), next(Y,Y1), up.
             caused atY(Y1) after atY(Y), next(Y,Y1), down.
             caused atX(X) after atX(X1), next(X,X1), left.
             caused atX(X1) after atX(X), next(X,X1), right.
             caused -atX(X) if atX(X1), X1 <> X after atX(X).
             caused -atY(Y) if atY(Y1), Y1 <> Y after atY(Y).
initially :  total atX(X). total atY(Y).
             forbidden atX(X), atX(X1), X <> X1.
             forbidden atY(Y), atY(Y1), Y <> Y1.
             caused anywhere if atX(X), atY(Y).
             forbidden not anywhere.

goal :       atX(0), atY(0)?(n)
```

Fluents `atX` and `atY` represent the current position of the robot in the grid and are inertial. Another fluent, `anywhere`, is used to ensure the validity of the initial state. Four actions move one step `up`, `down`, `left` or `right`, respectively. They are concurrently executable, giving the possibility to move diagonally in one step. Just concurrent execution of $\{\texttt{up}, \texttt{down}\}$ and $\{\texttt{left}, \texttt{right}\}$ is not admitted. The effects of the respective move actions are changes in the horizontal or vertical coordinates and an invalidation of the previous horizontal or vertical coordinates, overriding inertia.

For the initial state, we use new language constructs: `total f` is a macro representing the two causal rules `caused f if not –f` and `caused –f if not f`. It gives rise to nondeterminism in that both states containing `f` and `–f`, respectively, are considered. In the example, `total atX(X)` and `total atY(Y)` give rise to 2^{2n} possible initial states, corresponding to all possible assignments of $\{\texttt{atX(i)}, \texttt{–atX(i)}\}$ and $\{\texttt{atY(i)}, \texttt{–atY(i)}\}$ for $0 \leq \texttt{i} < n$. These statements thus create many illegal states, e.g. one containing $\texttt{atX}(0), \ldots, \texttt{atX}(\texttt{n}-1), \texttt{atY}(0), \ldots,$ $\texttt{atY}(\texttt{n}-1)$, and one containing $-\texttt{atX}(0), \ldots, -\texttt{atX}(\texttt{n}-1), -\texttt{atY}(0), \ldots, -\texttt{atY}(\texttt{n}-1)$.

We therefore also use the macro `forbidden`, which renders states where the specified condition holds illegal. In our example, we express that `atX` and `atY` hold for at most one horizontal and vertical position, respectively. The fluent `anywhere` is used to avoid states in which only `–atX` or only `–atY` holds, respectively, in that the case where `anywhere` is not caused is `forbidden`. These conditions narrow the number of legal initial states down to the actual n^2 possible initial positions.

For the problem depicted in Figure 4, the following optimistic plan works if the initial position of the robot is as in Figure 4 (or anywhere closer the upper left), but not if the initial position of the robot is further down or right, so it is not secure:

$P_1 = \langle\, \{\texttt{left}, \texttt{up}\}, \{\texttt{left}\} \rangle$

The following, on the other hand, is a three-step secure plan for SQUARE(4):

$P_2 = \langle \{\mathtt{left}, \mathtt{up}\}, \{\mathtt{left}, \mathtt{up}\}, \{\mathtt{left}, \mathtt{up}\}\rangle$

Note that in this domain the only source of uncertainty is the initial state. All actions are always executable and effects are deterministic. The actions do not "gain" any knowledge, so a representation exploiting knowledge states is not beneficial. Since the exact initial position is not known, knowledge of the position at each step is necessary in order to determine the action effects. Encoding all possible initial world states seems to be the only option for representing this problem.

5.2.2 Paint

Consider the following scenario: A house is to be painted. Several colors for painting are available, and several painters, e.g. `joe` and `jack`. Assume `joe` suffers from a red-green color-blindness known as "Daltonism". When we tell him to paint the house `red`, we do not know whether it will be `red` or `green` when he is done. Therefore, we have incomplete knowledge about the action effect, resulting in a nondeterministic action effect. However, even in this case some facts are known, e.g. that the house is definitely not blue.

Basic Encoding Let us first consider a simple planning problem in which the house is initially colored `blue` and we want it colored `green` after one time unit. In the background knowledge we define predicates $\mathtt{c}(\mathtt{x})$ for colors `x`, $\mathtt{painter}(\mathtt{p})$ for persons `p` to paint, and a predicate $\mathtt{conf}(\mathtt{c1}, \mathtt{c2}, \mathtt{p})$ if painter `p` confuses the colors `c1` and `c2`; the latter is symmetric on colors. Furthermore, we use a predicate $\mathtt{confusedBy}(\mathtt{p}, \mathtt{c})$ for painters `p` which confuse a color `c`; this is conveniently expressed by a logic programming rule representing projections.

```
c(blue).  c(red).  c(green).
painter(joe).  painter(jack).
conf(red, green, joe).
conf(C1, C2, A) :- conf(C2, C1, A).
confusedBy(C, A) :- conf(C, C1, A).
```

An encoding PD_{paint} of the Paint domain is shown in Figure 5. We only use one fluent, `col`, which describes the color of the house, and one action `paint`, expressing that a painter is asked to paint the house in a particular color. We declare this action to be unconditionally executable, and the macro `noConcurrency` forces actions to be executed sequentially.

We next define action effects. If the painter does not confuse the color he is asked to paint with, the action has the deterministic effect of the house being in the requested color. If, however, the painter is asked to paint the house in a color he might confuse with another color, we model a nondeterministic effect.

This is achieved in way which is reminiscent of the causal rules making up the `total` macro presented in Section 5.2.1. A pair of causal rules `caused f if not g` and `caused g if not f` gives rise to two alternative successor states, one containing `f` and one containing `g`. In our example, `f` and `g` are the fluents `col(C1)` and `col(C2)` where `C1` and `C2` are the confusable colors. We want these alternative states to occur exactly after a suitable action is performed, so we add `after paint(C1, A)` to each of the rules.

Finally, `col` is declared `inertial`. Concerning exceptions to inertia, the situation is different than in the Blocks World or Square domains, because of the nondeterministic action effect. The rule

```
caused -col(C1) after paint(C2, A), col(C1), C1 <> C2.
```

would be incorrect if `conf(C1, C2, A)` holds. We could use the rule

```
fluents :  col(C) requires c(C).
actions :  paint(C, A) requires c(C), painter(A).
always :   executable paint(C, A).
           noConcurrency.
           caused col(C) after paint(C, A), not confusedBy(C, A).
           caused col(C1) if not col(C2), conf(C1, C2, A) after paint(C1, A).
           caused col(C2) if not col(C1), conf(C1, C2, A) after paint(C1, A).
           inertial col(C).
           caused -col(C1) if col(C), C <> C1.
initially : col(blue).
goal :     col(green)? (1)
```

Figure 5: An encoding of the painting domain (PD_{paint})

```
caused -col(C1) if col(C2), C1 <> C2 after col(C1).
```

expressing that -col(C1) holds if the color of the house has really changed. Alternatively, as in Figure 5, we can state that col should be true for only one color, explicitly deriving -col for all other colors. In that case, negative inertia for -col can be safely ignored.

It should be noted that the knowledge states reachable from the initial state in PD_{paint} are in one-to-one correspondence with the actual world states.

For this planning problem, the following three optimistic plans exist:

$P_1 = \langle \{\texttt{paint(green, joe)}\}\rangle$
$P_2 = \langle \{\texttt{paint(red, joe)}\}\rangle$
$P_3 = \langle \{\texttt{paint(green, jack)}\}\rangle$

The color-blind painter joe can be told to paint red or green, and we can hope that he will choose green, but he might also choose red. On the other hand, jack, who is not color-blind, will paint the house green for sure, and therefore only the latter plan P_3 is secure.

Forgetting In some cases we can avoid nondeterminism by employing a knowledge state view. In the Paint domain, nondeterminism arises from the fact that the exact color of the house after a paint action is not known in some cases, and two possible world states need to be considered nondeterministically.

However, the language $\mathcal{K}$ also allows for a knowledge-oriented representation. We modify the domain by modeling only definitely known information and omit the two rules responsible for the nondeterministic choice in PD_{paint}. We thus no longer cause the house to be of some color after a color-blind painter has been asked to paint the house in a color he might confuse. However, we still need to block inertia, or the house would retain its color. One way to achieve this is to encode the negative information about the house color, which is known even if no positive information is available. In the particular example, we know that the house will not have a color which the painter does not confuse with the asked-for color. For example, we know $-\texttt{col}(\texttt{blue})$ after $\texttt{paint}(\texttt{green}, \texttt{joe})$, and this can be expressed by the general causal rule

$$\texttt{caused}\ -\texttt{col}(\texttt{C})\ \texttt{after}\ \texttt{paint}(\texttt{C1}, \texttt{A}),\ \texttt{conf}(\texttt{C1}, \texttt{C2}, \texttt{A}),\ \texttt{col}(\texttt{C}), \texttt{C} <> \texttt{C1}, \texttt{C} <> \texttt{C2}.$$

Note that executing $\texttt{paint}(\texttt{green}, \texttt{joe})$ in the modified domain, PD_{kpaint}, encodes *forgetting* parts of the knowledge about col, and that the knowledge states reachable from the initial state no longer correspond one-to-one with the actual world states. By applying forgetting techniques, we have managed to transform the nondeterministic domain PD_{paint} to a deterministic one. Indeed, the only secure plan when using PD_{paint} is the single optimistic plan (which is also trivially secure) when using PD_{kpaint}. In our experience, problems formulated by such knowledge-oriented encodings are usually much easier to solve (cf. benchmarks in (Eiter et al., 2003a)), but it is probably not always possible to find a deterministic knowledge-oriented encoding for a nondeterministic domain, by complexity results presented in (Eiter et al., 2003a).

Forgetting cannot be emulated directly by formalisms which adopt a world state view. There, leaving fluents open necessarily amounts to a disjunction over all possible world states (as argued in (Lin & Reiter, 1994)), whereas we can explicitly distinguish between such a totalization and a true forgetting approach.

Conditional Inertia The encoding PD_{kpaint} has a minor problem, though: It will not work correctly if the painter confuses all available colors, because inertia is not overridden by the added rule in this case. Indeed, if we remove $\texttt{c}(\texttt{blue})$ from the background knowledge of the example above, and the house is already $\texttt{green}$ in the initial state, i.e. $\texttt{initially}: \ \texttt{col}(\texttt{blue}).$ is replaced by $\texttt{initially}: \ \texttt{col}(\texttt{green}).$, we get the following (optimistic and secure) plans:

$$P_a = \langle \{\texttt{paint}(\texttt{green}, \texttt{joe})\}\rangle$$
$$P_b = \langle \{\texttt{paint}(\texttt{red}, \texttt{joe})\}\rangle$$
$$P_c = \langle \{\texttt{paint}(\texttt{green}, \texttt{jack})\}\rangle$$
$$P_d = \langle \emptyset \rangle$$

of which P_a and P_b are wrong, as they do not necessarily establish the goal.

As already mentioned, the reason for this fault is that no exception to inertia is provided when $\texttt{paint}(\texttt{green}, \texttt{joe})$ or $\texttt{paint}(\texttt{red}, \texttt{joe})$ are executed, and so $\texttt{col}(\texttt{green})$ continues to hold, even if it should not. The $\texttt{inertia}$ macro requires negative knowledge about the inertial fluent to be derived. In situations as the one above, however, there is no cause for such a negative knowledge.

One approach to solve such a scenario is to create an additional way for providing exceptions to inertia, by adding explicit conditions under which inertia

```
fluents:   unknowncolor.    col(C) requires c(C).
actions:   paint(C,A) requires c(C), painter(A).
always:    executable paint(C,A).
           noConcurrency.
           caused col(C) after paint(C,A), not confusedBy(C,A).
           caused unknowncolor after paint(C,A), confusedBy(C,A).
           caused -col(C) after paint(C1,A), conf(C1,C2,A),
                  col(C), C <> C1, C <> C2.
           inertial col(C) if not unknowncolor.
           caused -col(C1) if col(C), C <> C1.
initially: col(blue).
goal:      col(green)? (1)
```

Figure 6: A conditional inertia encoding of the painting domain ($PD_{cipaint}$)

applies. We refer to this concept as *conditional inertia*. In $\mathcal{K}$, we simply extend the `inertial` macro by allowing `if` and `after` conditions, just as for standard causal rules.

In the Paint domain, we modify PD_{kpaint} by introducing a new auxiliary fluent `unknowncolor`, which explicitly represents the fact that the color of the house is not known. This fluent holds after a painter has been asked to paint with a color he confuses and inertia is not applied in that case. The modified domain $PD_{cipaint}$ is given in Figure 6. The planning problem involving $PD_{cipaint}$ correctly yields only P_c and P_d as (optimistic and secure) plans.

It turns out that conditional inertia is a versatile concept, which can be used to encode many domains involving nondeterministic action effects by a deterministic knowledge oriented encoding.

5.2.3 Ring

Imagine a robot moving in a ring of n rooms which are all connected. There are two actions `fwd` and `back` to move to the previous and next room, respectively. Each room has a window and the robot can `close` and `lock` any window, where locking is only possible if the window is closed. The goal is to lock all windows. However, gusts of wind (which are obviously not under the control of the robot) may change the state of a window from being closed to being open and vice versa. The robot therefore cannot be sure that a window remains closed after he has closed it. In the initial situation the position of the robot is unknown and all windows are open. This domain has been described in (Cimatti & Roveri, 1999) and is referred to as RING(n). Figure 7 shows an instance with eight rooms.

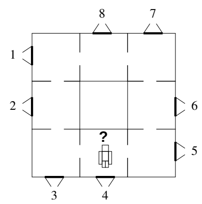

Figure 7: RING(8)

The background knowledge models the room layout:

```
next(r1,r2)....next(r7,r8). next(r8,r1).
room(R):-next(R,R1).
```

Let us first consider what kind of knowledge is crucial in this domain. The robot does not have knowledge about its position, but it also has no means of gaining knowledge in this respect (a similar situation as in the Square domain). Concerning the closed-state of a window, the robot knows that a window is closed immediately after having closed it. The robot also knows that a closed window stays closed after being locking, but nothing else is known about the closed-state of a window.

According to this analysis, we present an encoding in Figure 8, which uses a world view for `position` and a knowledge view for `closed`. We use fluents `closed` and `locked` to encode whether the window in a room is closed or locked, respectively. The robot's position is expressed using `position`. Fluent `unlocked` should hold whenever some windows are not locked, and `anywhere` is an auxiliary fluent used for determining legal initial states.

The actions `fwd` and `back` represent forward and backward moves by the robot, `close` and `lock` are robot actions for closing and locking the window in the current room. Executability of `fwd`, `back`, and `close` is always given, while for `lock` the window at the current position must be `closed`. `unlocked` holds whenever some window is not known to be locked. The actions `close` and `lock` cause the window at the current position to be `closed` and `locked`, respectively, immediately after the respective action, and `fwd` and `back` cause the respective position changes.

Fluents `locked` and `position` are inertial. The exception for `position` inertia occurs whenever the robot moves to another position, while for `locked` no exception can occur. The fluent `closed` is not inertial until the respective window is locked, accounting for the lack and gain of knowledge we have discussed above by means of forgetting via conditional inertia.

Finally, the initial state is described. As discussed, a knowledge approach is not feasible for positional information, so we use the nondeterministic macro `total` together with appropriate restricting rules to form all initial states containing exactly one instance of `position`, similar to the Square encoding. We also represent the knowledge about all windows being open initially. Finally, the goal is reached whenever `unlocked` does not hold after l steps.

```
fluents:     closed(R) requires room(R).
             locked(R) requires room(R).
             position(R) requires room(R).
             unlocked. anywhere.
actions:     fwd. back. close. lock.
always:      executable fwd. executable back. executable close.
             executable lock if position(R), closed(R).
             caused unlocked if not locked(W).
             caused closed(R) after close, position(R).
             caused locked(R) after lock, position(R).
             caused position(R1) after fwd, position(R), next(R,R1).
             caused position(R1) after back, position(R), next(R1,R).
             inertial locked(R). inertial position(R).
             inertial closed(R) if locked(R).
             caused -position(R) after fwd, position(R).
             caused -position(R) after back, position(R).
             noConcurrency.
initially:   total position(R).
             forbidden position(R), position(R1), R <> R1.
             caused anywhere if position(R).
             forbidden not anywhere.
             caused -closed(R).
goal:        not unlocked?(l)
```

Figure 8: Ring domain (PD_{ring})

The secure plans of this domain for RING(2) and plan-length 5 are

$$P_f = \langle \{\texttt{close}\}, \{\texttt{lock}\}, \{\texttt{fwd}\}, \{\texttt{close}\}, \{\texttt{lock}\}\rangle$$
$$P_b = \langle\ \{\texttt{close}\}, \{\texttt{lock}\}, \{\texttt{back}\}, \{\texttt{close}\}, \{\texttt{lock}\}\rangle$$

and for RING(n) and plan-length $3n - 1$ two analogous plans exist.

It is possible to easily switch from a knowledge view to a world view on `closed` by adding a causal rule

```
total closed(R).
```

to the `always`-section, creating nondeterminism for each step in which some `closed` state is not known.

5.3 Action Costs

In (Eiter, Faber, Leone, Pfeifer, & Polleres, 2003b) we have defined an extension of the language $\mathcal{K}$ called $\mathcal{K}^c$ which allows to assign costs to actions. For instance, in $\mathcal{K}^c$ one can assign a cost of 1 (representing e.g. energy resources consumed by the action) to each `move` action by modifying the declaration of `move` in PD_{bw2} of Figure 3 to read

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

The plans for a $\mathcal{K}^c$ planning problem are defined as those plans which minimize the sum of the respective costs of all actions in the plan. For the Blocks World planning problem from Section 4 and plan length 3, we obtain two plans with five actions, but none of the plans with six actions considered originally.

$$P_a = \langle\{\texttt{move(3,table)}\}, \{\texttt{move(1,3)}, \texttt{move(5,table)}\}, \{\texttt{move(2,4)}, \texttt{move(6,5)}\}\rangle$$
$$P_b = \langle\{\texttt{move(3,table)}, \texttt{move(5,table)}\}, \{\texttt{move(1,3)}\}, \{\texttt{move(2,4)}, \texttt{move(6,5)}\}\rangle$$

Cost statements may contain integer arithmetics supported by the underlying DLV system. They may also contain the designated constant `time`, allowing for dynamic cost assignment: `time` will evaluate to the time-step in which the particular action instance occurs. This provides a flexible framework for performing qualitative optimization planning.

Using this machinery, it is possible to solve several generic problems (Eiter et al., 2003b): finding (α) plans with minimal cost for a given number of steps (*cheapest plan*), (β) plans with minimal time steps (*shortest plan*), (γ) plans which are the shortest among the cheapest, and (δ) plans which are the cheapest among the shortest.

One might think that assigning costs to fluents in a similar manner would be useful as well. However, this would trigger semantic issues, since plans may have more than one supporting trajectory, i.e., sequences of states serving as a witness for the viability of the plan. These different trajectories could then have different fluent costs assigned, and one would have to apply some sort of aggregation (maximum, arithmetic mean,...).

5.4 Features and Pitfalls

After having presented multiple aspects of knowledge representation in $\mathcal{K}$ by means of several examples, we now summarize and discuss the features (and pitfalls) of encoding domains in this language in more detail.

5.4.1 Knowledge States

We have seen that default negation and the concepts of $\mathcal{K}$ provide a flexible tool for knowledge representation in the field of planning, but using negation as failure also involves some subtleties via the full freedom of normal logic programs to describe state constraints. In analogy to the term "Planning as Satisfiability" (coined by Kautz and Selman) our approach may well be conceived as "Planning as Answer Set Programming" or even "Answer Set Programming as Planning", to some extent.

$\mathcal{K}$ and $\mathcal{K}^c$ provide more than classical action languages where transitions are defined between completely defined world states or sets of such states (i.e., belief states). In fact, the knowledge state view implicit to the semantics of $\mathcal{K}$ requires the user to know about basic principles of logic programming and especially how to deal with non-monotonic (default) negation.

In this context we can state two major modeling principles:

Representation Principle 1: Exploit Closed World Assumption.

Representation Principle 2: Forget unnecessary information rather than keep complete state information.

Both of these principles should also be viewed in the light of "elaboration tolerance" in the sense of (McCarthy, 1999). Flexible frameworks such as $\mathcal{K}$ leave much of the responsibility how far domain- and problem-specific knowledge is exploited up to the user.

Knowledge state encodings somehow relieve the user from encoding every possible constraint on legal states of a particular domain by simply leaving "irrelevant" information open. We have discussed the applicability of the knowledge state view vs. the world state view and the concept of forgetting about fluents with illustrative examples in the Paint and Ring domains.

In order to design planning domains in $\mathcal{K}$, one has to be aware of the inherent non-monotonicity of the knowledge state view. Informally, a transition $\langle s, A, s' \rangle$ in $\mathcal{K}$ can be viewed as a transition between (answer sets of) normal logic programs where causation rules of the form

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

form a logic program $\Pi_{s'}$ consisting of all rules r

$$f\ \texttt{:-}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l.$$

such that $\{a_1, \ldots, a_m\} \in s \cup A$ and $\{a_{m+1}, \ldots, a_n\} \cap (s \cup A) = \emptyset$. $\Pi_{s'}$ then has all legal successor states for s and A as its answer sets.

Another example shows the strength of this logic programming view in planning: Modeling transitive closure in $\mathcal{K}$ is more concise and, in our opinion, more natural than in similar formalisms.

5.4.2 Transitive Closure

Expressing transitive closure in language $\mathcal{K}$ is straightforward because of its logic programming-based semantics. Let us assume there is a fluent `on(B,L)` which represents whether a block `B` resides on location `L` in the Blocks World.

Now, we want to define causation rules for a fluent `above(B,L)` which states that block `B` resides somewhere above location `L`. This can be modeled by static rules as follows:

```
caused above(B,L) if on(B,L).
caused above(B,L) if on(B,B1), above(B1,L).
```

In $\mathcal{K}$ these two rules sufficiently describe the values of fluent `above`, while in the action language $\mathcal{C}$ we would need to explicitly add negative information on `above`.

5.4.3 "Hidden" Default Negation in Macros

As we have already seen in the previous examples, default negation (`not`) allows a great degree of freedom and flexibility in the encoding of planning domains. However, default negation and nondeterminism might sometimes not be obvious when dealing with $\mathcal{K}$ macros. For instance, `inertial` statements can interfere with other rules using default negation. Consider for instance the rules

```
caused -f if not f.
inertial f.
```

in a state $s = \{\mathtt{f}\}$, with the empty action set $A = \emptyset$. Here, there are two legal transitions $\langle s, A, \{\mathtt{f}\}\rangle$ and $\langle s, A, \{\mathtt{-f}\}\rangle$.

Indeed, both statements encode default reasoning, and after a state containing `f` these defaults are in conflict. Since no further priority information is available, this gives rise to two alternatives. Priorities can be added in different ways; a simple method to prefer the alternative in which `-f` is selected, follows (Lukaszewicz, 1990): We introduce a fluent $\mathtt{-f_a}$, add a rule

$$\mathtt{caused\ -f_a\ if\ not\ f.}$$

and replace the original inertial rule with conditional inertia

$$\mathtt{inertial\ f\ if\ not\ -f_a.}$$

More sophisticated incorporation of priorities and preferences in general is a subject for further research. In particular, it might be possible to employ defeasible logic (Antoniou, Billington, Governatori, & Maher, 2001) or logic programs with preferences (Brewka & Eiter, 1999) for representing such concepts.

6 Comparison to STRIPS, ADL, and PDDL

In this section, we briefly compare $\mathcal{K}$ to STRIPS, ADL, and PDDL; a detailed comparison of $\mathcal{K}$ to many action languages can be found in (Eiter et al., 2004; Polleres, 2003).

As for STRIPS (Fikes & Nilsson, 1971), it is not hard to see that this formalism can be embedded into $\mathcal{K}$, as discussed in (Eiter et al., 2004). The same is possible for ADL (Pednault, 1989), since an extension by conditional effects is straightforward. We remind that propositional ADL has the same expressiveness as action language $\mathcal{A}$.

PDDL (Ghallab et al., 1998) emerged as a de-facto standard modeling language for classical planning, fostered by the variety of planning tools and algorithms that have been developed in the last decade. PDDL significantly differs from STRIPS and ADL; it stands for a modular family of languages rather than a single language, defined by so called *requirements*. Any planning system accepting PDDL might or might not implement these requirements. STRIPS and ADL amount to particular fragments of PDDL, which as discussed are expressible in $\mathcal{K}$.

PDDL version 1.2 comprises a number of requirements including value ranges comparable to typing in $\mathcal{K}$, domain axioms, disjunctive preconditions of actions, and quantified preconditions, which can be emulated in $\mathcal{K}$ like further ones. Evaluation of arithmetic expressions in PDDL can, to some extent, also be emulated in $\mathcal{K}$ within the restrictions of DLV$^{\mathcal{K}}$ integer arithmetics.

Noticeable for the concern of $\mathcal{K}$ are the PDDL requirements `:open-world` and `:true-negation`, by which the user can flexibly decide whether CWA should be applied or not for a particular fluent. These requirements can be easily realized in $\mathcal{K}$, given the logic programming flavored semantics of $\mathcal{K}$ and the totalization construct.

However, other requirements such as compound tasks, which are definable with `:action-expansions` in PDDL, are beyond the scope of $\mathcal{K}$. The techniques of Dix et al. (2002) to encode Hierarchical Task Network (HTN) Planning in Answer Set Programs might serve as a starting point for providing similar capabilities.

Actions are first-class citizens in PDDL and syntactically tightly coupled with their preconditions and effects. Here, preconditions can be modeled as formulae over fluents and effects can be modeled as conjunctions of fluent literals. Note that disjunctive effects are prohibited since in its basic form, PDDL does not deal with nondeterminism. For instance, the move action of Example 3 could be written as a PDDL operator as follows:

```
(:action move
  (:parameters ?b - block ?from ?to - loc)
  (:preconditions (and (not (blocked ?b))
                       (not (blocked ?to))
                       (on ?b ?from)
                       (not (= from? to?))))
  (:effect        (and (not (blocked ?from))
                       (on ?b ?to)
                       (when (not (= ?to table))
                   (blocked ?to)))))
```

This description, however, does not contain information about constraints on parallel move actions. An important note here is that the majority of PDDL planners only deal with sequential planning and do not consider parallel actions. Since operator preconditions are not allowed to include action predicates, constraints on parallel actions can not be expressed directly, as with the (`non`)`executable` statements in $\mathcal{K}$. Still, some PDDL based planners deal with parallel actions by automatically determining pairs of "mutex" actions: they automatically detect actions with interfering preconditions/effects and do not allow these to occur in parallel. In a formalism like PDDL, with only conjunctive effects, these "mutex" action pairs can easily be determined. For instance, the Graphplan (Blum & Furst, 1997) algorithm and its descendants make use of this to compute parallel plans for PDDL domain descriptions.

However, mutex detection is not enough for the example above. In order to state under which conditions parallel moves are allowed, one would need to add state constraints which prohibit states where one block has two locations at once. Such state constraints can be expressed in PDDL using the `:safety-constraints` and `:domain-axioms` requirements. Prohibiting that a block resides at two different locations at once can be formulated as follows:

```
(:safety forall(?b - block ?l1 ?l2 - loc)
          (or (= ?l1 ?l2)
              (not (and (on ?b ?l1) (on ?b ?l1)))))
```

Our $\mathcal{K}$ formulation to avoid such states, by directly forbidding respective actions to occur in parallel, is somewhat orthogonal to this. However, $\mathcal{K}$ also allows for expressing domain axioms and constraints as in PDDL by the use of static causation rules and the `forbidden` statement.

Action languages like $\mathcal{K}$ offer a more flexible description of transitions than the operator-based framework of PDDL. On the other hand, automatic determination of mutex pairs can in $\mathcal{K}$ perhaps not be as easily achieved as in the Graphplan algorithm. We consider the more flexible handling of concurrent actions in $\mathcal{K}$ as a language feature.

PDDL has evolved to version 2.1 (Fox & Long, 2003) recently, which adds additional levels introducing for instance durative actions, continuous and/or conditional effects, etc. This is currently not expressible in $\mathcal{K}$ (or $\mathcal{K}^c$) in a straightforward way. Interestingly, the requirements `:open-world` and `:true-negation` from version 1.2 have been dropped; this may be explained by the lack of broad support by current planning systems adhering to PDDL. Incomplete knowledge and nondeterminism hence are not addressed in this version of PDDL. Thus, for declarative planning in such settings, one has currently to resort to other formalisms and systems, such as $\mathcal{K}$ and DLV$^{\mathcal{K}}$. Noticeably, this shortcoming of PDDL has been realized and steps towards extensions for incomplete initial state specifications and nondeterminism have been made in the last "Workshop on PDDL" held at the "International Conference on Automated Planning and Scheduling (ICAPS'03)". For instance, the language NPDDL (Bertoli et al., 2003) accepted by the MBP planner (Bertoli et al., 2001) includes such extensions.

From a general modeling perspective, we feel that action languages like $\mathcal{K}$ are more versatile for describing actions and transitions than PDDL; they allow to express relationships among actions and fluents in a rule based language with natural reading, rather than in an operator-centric syntax. However, one has to bear the different objectives of these languages in mind. PDDL originally has been designed as a domain specification language for the International Planning Competition (IPC) based on ADL, and is conceived as a generic language representing the features of various special-purpose planners. Extensions to it are made very cautiously to maintain a widely accepted standard. Furthermore, the strict setting of an operator-based PDDL syntax is advantageous for a structural analysis of planning domains, and provides a better handle for optimizations and tailoring search heuristics, which is more of a concern for PDDL-based systems than natural problem representation.

7 Summary and Perspectives

In this chapter, we have considered a logic-based approach to planning based on action languages, which have been developed in the area of Knowledge Representation and Reasoning. Various such languages have been proposed in the literature, offering different capabilities and expressiveness. Compared to familiar planning formalisms like STRIPS or PDDL, which have an operator-centric view, action languages take a broader perspective in describing the planning world in terms of a theory, in which action execution and fluent values can be more flexibly interrelated than in an action-precondition-effect setting. At the same time, action languages have a clear formal semantics with a logical underpinning, which is supportive to considering reasoning tasks on actions theories and also provides a basis for implementations by exploiting efficient reductions to solvers for related logic formalisms.

Advanced action languages, such as $\mathcal{C}$ or $\mathcal{K}$, allow to deal with features like non-determinism, qualifications, ramifications, concurrent action execution, and incomplete information about states. The language $\mathcal{K}$ in particular, which we have discussed in more detail, is semantically based on logic programming and provides constructs from there such as negation by default, which allow for a flexible and natural modeling of incomplete information and non-determinism in planning domains. Exploiting these constructs, frame axioms, non-deterministic action effects, and other concepts can be modeled easily. By defining suitable macros for such concepts, one can allow for a more natural-language like, intuitive description of planning domains.

As a distinguishing feature with respect to similar languages, $\mathcal{K}$ supports a knowledge state view of state descriptions, where the values of fluents also might be unknown, rather than a classical world view, where each fluent must either be true or false. This view can be fruitfully exploited to handle indeterminism and nondeterminism in planning domains. In particular, we have identified three main sources of these, all of which are beyond the modeling capabilities of classical planning languages: Incomplete initial knowledge, nondeterministic action effects, and nondeterministic evolutions of the environment by uncontrollable, exogenous events.

We have exemplified the modeling of all these forms of non-determinism in $\mathcal{K}$ by illustrative examples. As shown on them, we may achieve a beneficial modeling of the domain of discourse by adopting the knowledge state view, where only a relevant "clipping" of the world is modeled. As discussed, conditional formulations of frame axioms can be used in our language to model "forgetting" about particular fluent values.

A fully operational implementation of $\mathcal{K}$, the DLV$^{\mathcal{K}}$ system, is available at

`http://www.dbai.tuwien.ac.at/proj/dlv/K/`

along with the examples in this text and many more, some of which are rather intricate and show further capabilities of the language, e.g. computing optimal plans. The reader is encouraged to browse them and to experiment with the system, setting up also new domains.

As shown by the results on using action languages as a host for solving planning problems so far, this is an interesting direction towards semantically rich and expressive formalisms for declarative planning. With the advent of solvers like DLV (Eiter et al., 2000) or SMODELS (Niemelä & Simons, 1997), to which these formalisms can be mapped in the spirit of satisfiability planning (Kautz & Selman, 1992), implementations have become available (Eiter et al., 2003a; Ferraris & Giunchiglia, 2000; McCain, 1999) which make experimentation and practical problem solving possible. The strength of these systems is at this time their modeling power rather than efficiency and scalability; improvements on these issues remain subject for future research. Nevertheless, $\text{DLV}^{\mathcal{K}}$ performs surprisingly well already in its current implementation. Compared with other planning systems tailored for conformant planning, $\text{DLV}^{\mathcal{K}}$ outperforms several of them as shown in (Eiter et al., 2003a; Cimatti, Roveri, & Bertoli, 2003), particularly when using knowledge state encodings.

For future development of planning systems based on action languages, we see different perspectives. On the computational side, the current systems do not employ sophisticated, goal-oriented heuristics for pruning the search space. Rather, the search is guided by built-in heuristics of the underlying logic solvers, which are geared towards problem solving in general and thus not always work best on the particular input to which planning problems are mapped. Hence, there is high potential for improvements. It remains to research more efficient mappings of action languages to logic solvers, which employ for the purpose of planning suitable heuristics to control the search at the level of the mapping, in reconciliation with the heuristics employed by the underlying logic solver. The experimentation with different heuristics for answer set solvers like DLV and SMODELS is still under research, and input from planning applications may guide the development of heuristics beneficial in practice.

Another perspective is further extension of action languages and resultant planning frameworks to increase expressivity. While $\text{DLV}^{\mathcal{K}}$ implements secure plans (Eiter et al., 2003a; Polleres, 2003), it currently does not support sensing actions and conditional plans. Sensing actions may be emulated to some extent by a suitable encoding of the action domain, but availability as first class citizens in the language would be desirable. Conditional plans allow for respecting any contingency by branching on conditions over the current state (Warren, 1976; Peot & Smith, 1992), and thus are more general than secure plans. However, their size can be exponential in general, and thus their generation is provably intractable. Feasible restrictions must be identified in order to apply our logic-programming approach to this kind of planning; (Son et al., 2001, 2004) present some results in this direction. An extension in a different dimension, towards a very general formalism for planning with uncertainty, is by probabilistic knowledge, such that both qualitative *and* quantitative uncertainty can be orthogonally combined within one language; (Eiter & Lukasiewicz, 2003) presents an approach for $\mathcal{C}$, which can be readily adapted to $\mathcal{K}$.

A further interesting perspective for planning via action languages emerges from their rooting in Knowledge Representation and Reasoning, which by their logic-based underpinnings are amenable to problems studied in this area such as Diagnosis, Belief Revision, or Knowledge Base Update. Methods which have been developed for accomplishing these tasks may be applied in order to reason about plan failures and for developing suitable recovery strategies, cf. (Giacomo, Reiter, & Soutchanski, 1998). (Dix et al., 2003) is an initial step of using $\texttt{DLV}^{\mathcal{K}}$ to this end in an execution monitoring framework. An agent might be situated in a dynamic environment, in which changes happen which are not reflected appropriately in the domain theory. Here, methods and techniques from belief revision and knowledge base update might be applied in order to revise the action theory of the planning domain. The logic-based setting of action languages eases this, while this would be much more involved for traditional planning approaches.

Finally, since most action languages have been conceived for reasoning about actions and change in general, implementations may allow for expressing a broader range of problems beyond traditional planning, like the CCALC system (McCain, 1999) implementing $\mathcal{C}$. Also $\texttt{DLV}^{\mathcal{K}}$ can, by the nature of its implementation, be adapted to accept more general than traditional planning goals (e.g., that in addition to the goal, certain conditions never hold along an execution). This holds potential for providing planning systems which can easily handle extended goals whereas classical planning systems cannot.

Acknowledgments

This work was supported by FWF (Austrian Science Funds) under the projects P14781, P16536-N04, and Z29-INF and the European Commission under projects IST-2001-37004 WASP and IST-2001-33570 COLOGNET.

References

Antoniou, G., Billington, D., Governatori, G., & Maher, M. J. (2001). Representation Results for Defeasible Logic. *ACM Transactions on Computational Logic*, *2*(2), 255–287.

Bertoli, P., Cimatti, A., Lago, U. D., & Pistore, M. (2003, June 9–13). Extending PDDL to Nondeterminism, Limited Sensing and Iterative Conditional Plans. In *Proceedings of ICAPS'03 Workshop on PDDL.* Trento, Italy.

Bertoli, P., Cimatti, A., Pistore, M., & Traverso, P. (2001, August). MBP: A Model Based Planner. In A. Cimatti, H. Geffner, E. Giunchiglia, & J. Rintanen (Eds.), *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information.*

Blum, A. L., & Furst, M. L. (1997). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, *90*, 281–300.

Bonet, B., & Geffner, H. (2000, April 14–17). Planning with Incomplete Information as Heuristic Search in Belief Space. In S. Chien, S. Kambhampati, & C. A. Knoblock (Eds.), *Proceedings of the Fifth International Conference* *on Artificial Intelligence Planning and Scheduling (AIPS'00)* (pp. 52–61). Breckenridge, Colorado, USA.

Brewka, G., & Eiter, T. (1999). Preferred Answer Sets for Extended Logic Programs. *Artificial Intelligence*, *109*(1-2), 297–356.

Cimatti, A., & Roveri, M. (1999, September 8–10). Conformant Planning via Model Checking. In S. Biundo & M. Fox (Eds.), *Proceedings of the Fifth European Conference on Planning (ECP'99)* (Vol. 1809, pp. 21–34). Durham, UK.

Cimatti, A., Roveri, M., & Bertoli, P. (2003). Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*. (Accepted for publication)

Dix, J., Eiter, T., Fink, M., Polleres, A., & Zhang, Y. (2003, September 15–18). Monitoring Agents using Declarative Planning. In *Proceedings of the 26th German Conference on Artificial Intelligence (KI2003)* (pp. 646–660). Springer.

Dix, J., Kuter, U., & Nau, D. (2002, February). *HTN Planning in Answer Set Programming* (Tech. Rep. No. CS-TR-4332 (UMIACS-TR-2002-14)). Dept. of CS, University of Maryland, College Park, MD 20752.

Eiter, T., Faber, W., Leone, N., & Pfeifer, G. (2000). Declarative Problem-Solving Using the DLV System. In J. Minker (Ed.), *Logic-Based Artificial Intelligence* (pp. 79–103). Kluwer Academic Publishers.

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

Eiter, T., Faber, W., Leone, N., Pfeifer, G., & Polleres, A. (2003b). Answer Set Planning under Action Costs. *Journal of Artificial Intelligence Research*, *19*, 25–71.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., & Polleres, A. (2004, April). A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, *5*(2).

Eiter, T., & Lukasiewicz, T. (2003). Probabilistic Reasoning about Actions in Nonmonotonic Causal Theories. In C. Meek & U. Kjærulff (Eds.), *Proceedings Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-2003), August 7-10, 2003, Acapulco, Mexico* (pp. 192–199). San Francisco, CA: Morgan Kaufmann Publishers.

Ferraris, P., & Giunchiglia, E. (2000). Planning as Satisfiability in Nondeterministic Domains. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA* (pp. 748–753). AAAI Press / The MIT Press.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *2*(3-4), 189–208.

Fox, M., & Long, D. (2003, April). *PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains.* (Manuscript, `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`)

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

Gelfond, M., & Lifschitz, V. (1993). Representing Action and Change by Logic Programs. *Journal of Logic Programming*, *17*, 301–321.

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

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998, October). *PDDL — The Planning Domain Definition language* (Tech. Rep.). Yale Center for Computational

Vision and Control. (Available at `http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`)

Giacomo, G. D., Reiter, R., & Soutchanski, M. (1998). Execution monitoring of high-level robot plans. In A. G. Cohn, L. Schubert, & S. C. Shapiro (Eds.), *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)* (pp. 453–464). Morgan Kaufmann Publishers.

Giunchiglia, E., Kartha, G. N., & Lifschitz, V. (1997). Representing Action: Indeterminacy and Ramifications. *Artificial Intelligence*, *95*, 409–443.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., & Turner, H. (2004). Nonmonotonic Causal Theories. *Artificial Intelligence*, *153*(1-2), 49–104.

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

Giunchiglia, E., & Lifschitz, V. (1998). An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)* (pp. 623–630).

Goldman, R., & Boddy, M. (1996). Expressive Planning and Explicit Knowledge. In *Proceedings AIPS-96* (pp. 110–117). AAAI Press.

Hintikka, J. (1962). *Knowledge and Belief.* Cornell University Press Ithaca, NY.

Kautz, H., & Selman, B. (1992). Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)* (pp. 359–363).

Lifschitz, V. (1997). On the Logic of Causal Explanation. *Artificial Intelligence*, *96*, 451–465.

Lin, F. (1995). Embracing Causality in Specifying the Indirect Effects of Actions. In C. S. Mellish (Ed.), *Proceedings of the 14th International Joint*

Conference on Artificial Intelligence (IJCAI '95) (pp. 1985–1993). Morgan Kaufmann Publishers.

Lin, F., & Reiter, R. (1994). Forget It! In R. Greiner & D. Subramanian (Eds.), *Working Notes, AAAI Fall Symposium on Relevance* (pp. 154–159). American Association for Artificial Intelligence.

Lukaszewicz, W. (1990). *Non-Monotonic Reasoning, Formalization of Commonsense Reasoning*. Ellis Horwood Limited.

McCain, N. (1999). *The Causal Calculator Homepage.* (`http://www.cs.utexas.edu/users/tag/cc/`)

McCain, N., & Turner, H. (1997). Causal Theories of Actions and Change. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)* (pp. 460–465).

McCarthy, J. (1990). *Formalization of Common Sense, papers by John McCarthy edited by V. Lifschitz.* Ablex.

McCarthy, J. (1999). *Elaboration Tolerance.* (Available at `http://www-formal.stanford.edu/jmc/elaboration.html`.)

McCarthy, J., & Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 4* (pp. 463–502). Edinburgh University Press. (reprinted in (McCarthy, 1990))

Niemelä, I., & Simons, P. (1997, July). Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In J. Dix, U. Furbach, & A. Nerode (Eds.), *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)* (Vol. 1265, pp. 420–429). Dagstuhl, Germany: Springer Verlag.

Parr, R., & Russel, S. (1995). Approximating Optimal Policies for Partially

Observable Stochastic Domains. In C. S. Mellish (Ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI '95)* (pp. 1088–1094). Morgan Kaufmann Publishers.

Pednault, E. P. D. (1989, May 15–1). Exploring the Middle Ground between STRIPS and the Situation Calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)* (pp. 324–332). Toronto, Canada: Morgan Kaufmann Publishers, Inc.

Peot, M. A., & Smith, D. E. (1992). Conditional Nonlinear Planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (pp. 189–197). AAAI Press.

Polleres, A. (2003). *Advances in Answer Set Planning*. Doctoral dissertation, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich.

Reiter, R. (1978). On Closed World Data Bases. In H. Gallaire & J. Minker (Eds.), *Logic and Data Bases* (pp. 55–76). New York: Plenum Press.

Russel, S. J., & Norvig, P. (1995). *Artificial Intelligence, A Modern Approach*. Prentice-Hall, Inc.

Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

Smith, D. E., & Weld, D. S. (1998, July). Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, (AAAI'98)* (pp. 889–896). AAAI Press / The MIT Press.

Son, T. C., & Baral, C. (2001). Formalizing Sensing Actions – A Transition Function Based Approach. *Artificial Intelligence*, *125*(1–2), 19–91.

Son, T. C., Baral, C., & McIlraith, S. (2001, September 17–19). Planning with Different Forms of Domain-Dependent Control Knowledge – An Answer

Set Programming Approach. In T. Eiter, W. Faber, & M. Truszczyński (Eds.), *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings* (pp. 226–239). Springer Verlag.

Son, T. C., Tu, P. H., & Baral, C. (2004, January). Planning with Sensing Actions and Incomplete Information Using Logic Programming. In V. Lifschitz & I. Niemelä (Eds.), *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)* (pp. 261–274). Fort Lauderdale, Florida, USA: Springer.

Warren, D. H. D. (1976, July 12–14). Generating Conditional Plans and Programs. In *Proceedings of the Summer Conference on Artificial Intelligence and Simulation of Behaviour* (pp. 344–354). Edinburgh, UK.