In Defense of PDDL Axioms

Sylvie Thiébaux

Computer Sciences Laboratory, The Australian National University, Knowledge Representation & Reasoning Program, National ICT Australia, Canberra, ACT 0200, Australia

Jörg Hoffmann

Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany

Bernhard Nebel

Institut für Informatik, Albert-Ludwigs-Universität Freiburg, D-79110 Freiburg, Germany

Abstract

There is controversy as to whether explicit support for PDDL-like axioms and derived predicates is needed for planners to handle real-world domains effectively. Many researchers have deplored the lack of precise semantics for such axioms, while others have argued that it might be best to compile them away. We propose an adequate semantics for PDDL axioms and show that they are an essential feature by proving that it is impossible to compile them away if we restrict the growth of plans and domain descriptions to be polynomial. These results suggest that adding a reasonable implementation to handle axioms inside the planner is beneficial for the performance. Our experiments confirm this suggestion.

Key words: classical planning, PDDL, expressive power

1 Introduction

It is not uncommon for planners to support *derived* predicates, whose truth in the current state is inferred from that of some *basic* predicates via some *axioms* under

Email addresses: Sylvie.Thiebaux@anu.edu.au (Sylvie Thiébaux), hoffmann@mpi-sb.mpg.de (Jörg Hoffmann), nebel@informatik.uni-freiburg.de (Bernhard Nebel).

the closed world assumption. While basic predicates may appear as effects of actions, derived ones may only be used in preconditions, effect contexts and goals. Planners that support such constructs include the partial order planner UCPOP (Barrett et al., 1995), the HTN planner SHOP (Nau et al., 1999), and the heuristic search planner GPT (Bonet and Geffner, 2001; Bonet and Thiébaux, 2003), to cite but a few. The original version of PDDL (McDermott, 1998), the International Planning Competition language, also featured such axioms and derived predicates. However, these were never (yet) used in competition events, and did not survive PDDL2.1, the extension of the language to temporal planning (Fox and Long, 2003).

We believe that the lack of axioms impedes the ability to elegantly and concisely represent real-world domains. Such domains typically require checking complex conditions which are best built hierarchically, from elementary conditions on the state variables to increasingly abstract ones. Without axioms, preconditions and effect contexts quickly become unreadable, or postconditions are forced to include supervenient properties which are just logical consequences of the basic ones. Sometimes things get even worse as extra actions need to be introduced or action descriptions need to be propositionalised to fit a particular problem instance.

Moreover, axioms provide a natural way of capturing the effects of actions on common real world structures such as paths or flows, e.g. electricity flows, chemical flows, traffic flows, etc. [1] For instance, one benchmark used in the 2004 competition (see also below) is a deterministic version of the power supply restoration problem (PSR) described by Thiébaux and Cordier (2001). Given a network consisting of power sources, electric lines and switches, an important aspect of the problem is to determine which are the lines currently fed by the various sources, and how feeding is affected when opening or closing switches. Computing and updating "fed" following a switching operation requires traversing the possible paths of network. There is no intuitive way to do this in the body of a PDDL action, while a recursive axiomatisation of "fed" from the current positions (open or closed) of the switches is relatively straightforward (Bonet and Thiébaux, 2003).

The most common criticism of the original PDDL axioms was that their semantics was ill-specified. In particular, the organisers of the 2002 International Planning Competition objected that [2] the conditions under which the truth of the derived predicates could be uniquely determined were unclear. We remedy this by providing a clear semantics for PDDL axioms while remaining consistent with the original description by McDermott (1998). In particular, we identify conditions that are sufficient to ensure that the axioms have an unambiguous meaning, and explain how these conditions can efficiently be checked.

Another common view is that axioms are a non-essential language feature which

[1] In that respect, PDDL axioms offer advantages over the use of purely logical axioms as in the original version of STRIPS (Lifschitz, 1986).

[2] Personal communications.

it might be better to compile away than to deal with explicitly, compilation offering the advantage of enabling the use of more efficient, simple, standard planners without specific treatment (Gazen and Knoblock, 1997; Garagnani, 2000; Davidson and Garagnani, 2002). We bring new insight to this issue. We give evidence that axioms add significant expressive power to PDDL. We take "expressive power" to be a measure of how succinctly domains and plans can be expressed in a formalism and use the notion of compilability to analyse that (Nebel, 2000). As it turns out, axioms are an essential feature because it is impossible to compile them away. We prove that any compilation scheme involves either a worst-case exponential blow-up in the size of the domain description, or a worst-case exponential blow-up in the length of the shortest plans. Note that these are essential weaknesses. Exponentially sized domain descriptions are clearly prohibitive for efficient planning. As for exponentially sized plans in the compilation, from a plan execution point of view these don't hurt since one can back-translate the plan into a plan for the original task, and execute that shorter plan. However, exponentially longer plans will seriously impede, likely be prohibitive for, the performance of any automated planner trying to solve the compiled planning tasks.

If we allow for exponential growth, then compilations become possible. We specify one such transformation, which, unlike those previously published (Gazen and Knoblock, 1997; Garagnani, 2000; Davidson and Garagnani, 2002), works without restriction on the presence of negated derived predicates in the domain description. However, as said above our results suggest that it might be much more efficient to deal with axioms inside the planner than to compile them away. In fact, our experiments with FF (Hoffmann and Nebel, 2001) suggest that adding even a simple implementation of axioms to a planner clearly outperforms the original version of the planner run on the compilation.

Derived predicates and axioms were (in a slightly simpler version than what we consider in this paper) re-introduced into PDDL2.2 (Edelkamp and Hoffmann, 2004), the language for the 2004 International Planning Competition (IPC-4). The discussion that led to this decision, taken by the IPC-4 advisory committee in July 2003, was largely driven by the results we present in this paper (a short version of the paper appeared at IJCAI'03). IPC-4 featured two benchmark domains that were formulated with the help of derived predicates, namely the PSR problem mentioned above as well as a PROMELA domain dealing with the detection of errors in communication protocols. Both domains are treated in the experimental section of this paper.

The paper is structured as follows. Section 2 provides the syntax and semantics of axioms, Section 3 contains our results on expressive power, Section 4 discusses compilations, Section 5 describes our experiments, Section 6 concludes.

2 Syntax and Semantics

This paper remains in the sequential planning setting. We start from the syntax of PDDL2.1 level 1, i.e., PDDL with ADL actions as used in the 1998 and 2000 planning competitions (McDermott, 1998; Bacchus, 2000). Our syntax of PDDL with axioms, or PDDL$_{\mathcal{X}}$, is given in Figure 1. The PDDL2.2 syntax largely adopts this syntax.

`<axiom-def>` is the only addition to the original syntax. Let $\mathcal{B}$ and $\mathcal{D}$ be two sets of predicate symbols with $\mathcal{B} \cap \mathcal{D} = \emptyset$, called the set of basic and derived predicates, respectively. Symbols in $\mathcal{D}$ are not allowed to appear in the initial state description and in atomic effects of actions, but may appear in preconditions, effect contexts, and goals. The domain description features a set of axioms A. These have the form $(:\texttt{derived}\ (d\ ?\vec{x})\ (f\ ?\vec{x}))$, where $d \in \mathcal{D}$, and where f is a first-order formula built from predicate symbols in $\mathcal{B} \cup \mathcal{D}$ and whose free variables are those in the vector $\vec{x}$.

Intuitively, an axiom $(:\texttt{derived}\ (d\ ?\vec{x})\ (f\ ?\vec{x}))$ means that when $(f\ ?\vec{x})$ is true at the specified arguments in a given state, we should *derive* that $(d\ ?\vec{x})$ is true at those arguments in that same state. Unlike traditional implications, these derivations are not to be contraposed (the negation of f is not derived from the negation of d), and what cannot be derived as true is false (closed world assumption). Because of the closed world assumption, there is never any need to explicitly derive negative literals, so the constraint that the consequent of axioms be *positive* literals does not make us lose generality.

In our approach, as in previous work in planning (Barrett et al., 1995; McDermott, 1998; Bonet and Thiébaux, 2003), states are completely characterised by the values of the basic predicates, and derived predicates represent high-level properties of states whose truth value is uniquely determined given the basic ones. While this is a bit restrictive in postulating a strict distinction between basic and derived predicates, it typically suffices to formulate the problems that are of interest to the planning community. The approach has the advantage of being practical, in the sense that it only involves simple forms of reasoning in order to determine the successor states. A whole body of formalisms in the field of reasoning about action provide a more elaborate treatment of the ramification problem and allow a given predicate to be both explicitly changed by actions or inferred from the changes (Winslett, 1988; Brewka and Hertzberg, 1993; Sandewall, 1994; McCain and Turner, 1995; Lin, 1995; Gustafsson and Doherty, 1996).

In sum, axioms are essentially (function free) logic program statements (Lloyd, 1984). For example, from the basic predicate `on` and the predicate `holding` in Blocks World, we can define the predicate `clear`, as follows:

```
(:derived (clear ?x)
```

Fig. 1 Syntax of PDDL$_{\mathcal{X}}$

```
<domain>                  ::= (define (domain <name>)
                                      [<constant-def>]
                                      [<predicates-def>]
                                      [<axiom-def>*]
                                      <action-def>*)
<constants-def>           ::= (:constants <name>+)
<predicate-def>           ::= (:predicates <skeleton>+)
<skeleton>                ::= (<predicate> <variable>*)
<predicate>               ::= <name>
<variable>                ::= ?<name>
<axiom-def>               ::= (:derived <skeleton> <formula>)
<formula>                 ::= <atomic-formula>
<formula>                 ::= (not <formula>)
<formula>                 ::= (and <formula> <formula>+)
<formula>                 ::= (or <formula> <formula>+)
<formula>                 ::= (imply <formula> <formula>)
<formula>                 ::= (exists (<variable>+) <formula>)
<formula>                 ::= (forall (<variable>+) <formula>)
<atomic-formula>          ::= (<predicate> <term>*)
<ground-atomic-formula>   ::= (<predicate> <name>*)
<term>                    ::= <name>
<term>                    ::= <variable>
<action-def>              ::= (:action <name>
                               :parameters (<variable>*)
                               <action-def body>)
<action-def body>         ::= [:precondition <formula>]
                               :effect <eff-formula>
<eff-formula>             ::= <one-eff-formula>
<eff-formula>             ::= (and <one-eff-formula>
                                   <one-eff-formula>+)
<one-eff-formula>         ::= <atomic-effs>
<one-eff-formula>         ::= (when <formula> <atomic-effs>)
<one-eff-formula>         ::= (forall (<variable>+) <atomic-effs>)
<one-eff-formula>         ::= (forall (<variable>+)
                                 (when <formula> <atomic-effs>))
<atomic-effs>             ::= <literal>
<atomic-effs>             ::= (and <literal> <literal>+)
<literal>                 ::= <atomic-formula>
<literal>                 ::= (not <atomic-formula>)
<task>                    ::= (define (task <name>)
                                      (:domain <name>)
                                      <object declaration>
                                      <init>
                                      <goal>)
<object declaration>      ::= (:objects <name>*)
<init>                    ::= (:init <ground-atomic-formula>*)
<goal>                    ::= (:goal <formula>)
```

```
  (and (not (holding ?x))
       (forall (?y) (not (on ?y ?x))))))
```

Another classic is `above`, the transitive closure of `on`, e.g.:

```
(:derived (above ?x ?y)
  (or (on ?x ?y)
      (exists (?z) (and (on ?x ?z)
                        (above ?z ?y)))))
```

or equivalently:

```
(:derived (above ?x ?y)
  (or (on ?x ?y)
      (exists (?z) (and (above ?x ?z)
                        (above ?z ?y)))))
```

The formal semantics below will of course enforce that the result is not affected by the order in which atomic formulae appear in the antecedent.

In a planning context, it is natural and convenient to restrict attention to so-called *stratified* axiom sets. By disallowing negation "through recursion", stratified logic programs avoid unsafe use of negation and have an unambiguous, well-understood semantics (Apt et al., 1988). The idea behind stratification is that some derived predicates should first be defined in terms of the basic ones possibly using negation, or in terms of themselves (allowing for recursion) but *without* using negation. Next, more abstract predicates can be defined building on the former, possibly using their negation, or in terms of themselves but without negation, and so on. Thus, a stratified axiom set is partitionable into strata, in such a way that the negation normal form[3] (NNF) of the antecedent of an axiom defining a predicate belonging to a given stratum uses arbitrary occurrences of predicates belonging to strictly lower strata and *positive* occurrences of predicates belonging to the same stratum. Basic predicates may be used freely.

Definition 1 *An axiom set A is stratified iff there exists a partition (stratification) of the set of derived predicates $\mathcal{D}$ into (non-empty) subsets $\{\mathcal{D}_i, 1 \leq i \leq n\}$ such that for every $d_i \in \mathcal{D}_i$ and every axiom* $(:\ \texttt{derived}\ (d_i\ ?\vec{x})\ (f\ ?\vec{x})) \in A$*:*

(1) if $d_j \in \mathcal{D}_j$ appears in NNF$(f\ ?\vec{x})$*, then $j \leq i$,*
(2) if $d_j \in \mathcal{D}_j$ appears negated in NNF$(f\ ?\vec{x})$*, then $j < i$.*

For instance, Figure 2 shows a blocks world domain description with 4 operators. The basic predicates are $\mathcal{B} = \{\texttt{on}, \texttt{ontable}\}$, and the derived ones are $\mathcal{D} = \{\texttt{above}, \texttt{holding}, \texttt{clear}, \texttt{handempty}\}$. The axiomatisation of `handempty` and that of `clear` use the negation of `holding`. The axiom set is stratified and a possible

[3] In a formula in NNF, negation occurs only in literals.

Fig. 2 Blocks World with Derived Predicates

```
(define (domain bw-axioms)
  (:requirements :strips)
  (:predicates (on-table ?x) (on ?x ?y) ;basic predicates
               (holding ?x) (above ?x ?y) (clear ?x) (handempty)) ; derived predicates

(:derived (holding ?x)
      (and (not (on-table ?x))(not (exists (?y) (on ?x ?y)))))

  (:derived (above ?x ?y)
      (or (on ?x ?y)
          (exists (?z) (and (on ?x ?z) (above ?z ?y)))))

  (:derived (clear ?x)
      (and (not (holding ?x))
           (not (exists (?y) (on ?y ?x)))))

  (:derived (handempty) (forall (?x) (not (holding ?x))))

  (:action pickup
    :parameters (?ob)
    :precondition (and (clear ?ob) (on-table ?ob) (handempty))
    :effect (not (on-table ?ob)))

  (:action putdown
    :parameters  (?ob)
    :precondition (holding ?ob)
    :effect (on-table ?ob))

  (:action stack
    :parameters  (?ob ?underob)
    :precondition (and (clear ?underob) (holding ?ob))
    :effect (on ?ob ?underob))

  (:action unstack
    :parameters  (?ob ?underob)
    :precondition (and (on ?ob ?underob) (clear ?ob) (handempty))
    :effect (not (on ?ob ?underob)))
)
```

stratification is $\mathcal{D}_1 = \{\texttt{above}, \texttt{holding}\}$, $\mathcal{D}_2 = \{\texttt{clear}, \texttt{handempty}\}$.

Note that any stratification $\{\mathcal{D}_i, 1 \leq i \leq n\}$ of $\mathcal{D}$ induces a stratification $\{A_i, 1 \leq i \leq n\}$ of A in the obvious way: $A_i = \{(\texttt{:derived}\ (d_i\ ?\vec{x})\ (f_i\ ?\vec{x})) \in A \mid d_i \in \mathcal{D}_i\}$. Note also that when no derived predicate occurs negated in the NNF of the antecedent of any axiom, a single stratum suffices. Several planning papers have considered this special case (Gazen and Knoblock, 1997; Garagnani, 2000; Davidson and Garagnani, 2002), in particular PDDL2.2 (Edelkamp and Hoffmann, 2004) restricts the use of axioms to this case.

Working through the successive strata, applying axioms in any order within each stratum until a fixed point is reached and then only proceeding to the next stratum, always leads to the same final fixed point independently of the chosen stratification (Apt et al., 1988, p. 116). It is this final fixed point which we take to be the meaning of the axiom set.

We now spell out the semantics formally. Since we have a finite domain and no functions, we identify the objects in the domain with the ground terms (constants) that denote them, and states with finite sets of ground *atoms*, i.e., ground atomic formulae. More precisely, a state is taken to be a set of ground *basic* atoms: the derived ones will be treated as elaborate descriptions of the basic state. In order to define the semantics, however, we first need to consider an extended notion of "state" consisting of a set S of basic atoms and an arbitrary set D of atoms in the derived vocabulary. The modeling conditions for an extended state $\langle S, D\rangle$ are just the ordinary ones of first order logic, as though there were no relationship between

S and D. Where $?\vec{x}$ denotes a vector of variables and $\vec{t}$ denotes a vector of ground terms, we define:

Definition 2

$\langle S, D\rangle \models (b\ \vec{t})$ *for* $b \in \mathcal{B}$ *iff* $(b\ \vec{t}) \in S$
$\langle S, D\rangle \models (d\ \vec{t})$ *for* $d \in \mathcal{D}$ *iff* $(d\ \vec{t}) \in D$
$\langle S, D\rangle \models (\texttt{not}\ f)$ *iff* $\langle S, D\rangle \not\models f$
$\langle S, D\rangle \models (\texttt{and}\ f_1\ f_2)$ *iff* $\langle S, D\rangle \models f_1$ *and* $\langle S, D\rangle \models f_2$
$\langle S, D\rangle \models (\texttt{or}\ f_1\ f_2)$ *iff* $\langle S, D\rangle \models f_1$ *or* $\langle S, D\rangle \models f_2$
$\langle S, D\rangle \models (\texttt{forall}\ (?\vec{x})\ (f\ ?\vec{x}))$ *iff* $\langle S, D\rangle \models (f\ \vec{t})$ *for all* $\vec{t}$
$\langle S, D\rangle \models (\texttt{exists}\ (?\vec{x})\ (f\ ?\vec{x}))$ *iff* $\langle S, D\rangle \models (f\ \vec{t})$ *for some* $\vec{t}$

Then, applying axiom $a \equiv (:\ \texttt{derived}\ (d\ ?\vec{x})\ (f\ ?\vec{x}))$ in an extended state $\langle S, D\rangle$ with derived atoms D, results in the set $[\![a]\!](S, D)$ of further derived atoms:

Definition 3 $\quad [\![a]\!](S, D) = \Big\{(d\ \vec{t}) \mid \langle S, D\rangle \models (f\ \vec{t}),\ \vec{t}\ \textit{is ground}\Big\}$

Given this, we associate stratum A_i with the function $[\![A]\!]_i$ which maps a given basic state S to the set of ground derived atoms derivable from S and from the axioms at strata A_i and lower. This function is recursively defined as the least fixed point attainable by applying the axioms in A_i starting from the extended state consisting of S and of the set of ground derived atoms returned at the previous stratum by the function $[\![A]\!]_{i-1}$. The stratified axiom set A denotes the function $[\![A]\!] = [\![A]\!]_n$:

Definition 4 *Let* $\{A_i, 1 \leq i \leq n\}$ *be an arbitrary stratification for a stratified axiom set* A*. For each state* S*, let:*

$$[\![A]\!]_0(S) = \emptyset, \quad \textit{and for all } 1 \leq i \leq n$$
$$[\![A]\!]_i(S) = \bigcap\Big\{D \mid \bigcup_{a \in A_i} [\![a]\!](S, D) \cup [\![A]\!]_{i-1}(S) \subseteq D\Big\}$$

Then $[\![A]\!](S)$ *is defined as* $[\![A]\!]_n(S)$.

Note that, in the definition of $[\![A]\!]_i(S)$, the set D itself is an argument of $[\![a]\!](S, D)$, forcing D to be closed under the applications of the axioms $a \in A_i$. The definition states that D is the smallest set ($\bigcap$) containing $[\![A]\!]_{i-1}(S)$ and closed under these axioms.

Finally, given a stratified axiom set A, we write $S \models_A f$ to indicate that a formula f composed of both basic and derived predicates holds in state S:

Definition 5 $\quad S \models_A f \quad \textit{iff} \quad \langle S, [\![A]\!](S)\rangle \models f$

This modeling relation is used when applying an action in state S to check preconditions and effect contexts, and to determine whether S satisfies the goal. This is the only change introduced by the axioms into the semantics of PDDL and completes our statement of the semantics. The rest carries over verbatim from (Bacchus, 2000).

Algorithm 1 Stratification

```
1.  function STRATIFY(D, A)
2.    R ← ORDER(D, A)
3.    if ∀i ∈ D R[i, i] ≠ 2 then
4.      return EXTRACT(D, R)
5.    else fail

6.  function ORDER(D, A)
7.    for each i ∈ D do
8.      for each j ∈ D do
9.        R[i, j] ← 0
10.   for each (: derived (j ?x⃗) (f ?x⃗)) ∈ A do
11.     for each i ∈ D do
12.       if i occurs negatively in NNF(f ?x⃗) then
13.         R[i, j] ← 2
14.       else if i occurs positively in NNF(f ?x⃗) then
15.           R[i, j] ← MAX(1, R[i, j])
16.   for each j ∈ D do
17.     for each i ∈ D do
18.       for each k ∈ D do
19.         if MIN(R[i, j], R[j, k]) > 0 then
20.           R[i, k] ← MAX(R[i, j], R[j, k], R[i, k])
21.   return R

22. function EXTRACT(D, R)
23.   stratification ← ∅, remaining ← D, level ← 1
24.   while remaining ≠ ∅ do
25.     stratum ← ∅
26.     for each j ∈ remaining do
27.       if ∀i ∈ remaining R[i, j] ≠ 2 then
28.         stratum ← stratum ∪ {j}
29.     remaining ← remaining \ stratum
30.     stratification ← stratification ∪ {(level, stratum)}
31.     level ← level + 1
32.   return stratification
```

Practically, given a domain description it must be tested if the axiom set is stratified. If so, a stratification needs to be computed. Both the test and the computation of the stratification can be done in polynomial time in the size of the domain description, using for instance Algorithm 1.

The computation done in Algorithm 1 is reminiscent of that of the transitive closure of a relation. The algorithm starts by calling the function ORDER which analyses the axioms to build a[4] $|\mathcal{D}| \times |\mathcal{D}|$ matrix R such that $R[i,j] = 2$ when it follows from the axioms that predicate i's stratum must be strictly lower than predicate j's stratum, $R[i,j] = 1$ when i's stratum must be lower than j's stratum but not necessarily strictly, and $R[i,j] = 0$ when there is no constraint between the two strata. R is initialised with 0 everywhere (lines 7-9). Then, R is filled with the values encoding the status (strict or not) of the base constraints, i.e., those obtained by direct examination of the axioms (lines 10-15). Finally, the consequences of the base constraints are computed, similarly as one would compute a transitive closure (lines 16-20). From the constraint that i's stratum should be lower than j's stratum and the constraint that j's stratum should be lower than k's stratum (i.e., $\text{MIN}(R[i,j], R[j,k]) > 0$, see line 19), follows the constraint that i's stratum should be lower than k's stratum. If either of the two former constraints are strict, (i.e. $R[i,j] = 2$ or $R[j,k] = 2$), the latter is strict too (i.e. $R[i,k] = 2$). It may also be the case that the latter constraint has already been discovered and proven strict during an earlier iteration. Therefore, the correct status of that constraint is computed by taking the maximum of $R[i,j]$, $R[j,k]$ and the previous $R[i,k]$ (line 20).

There exists a stratification iff the strict relation encoded in R is irreflexive, that is iff $R[i,i] \neq 2$ for all $i \in \mathcal{D}$ (line 3). In that case, the stratification corresponding to the smallest pre-order consistent with R (i.e. predicates are put in the lowest stratum consistent with R), is extracted from R using the function EXTRACT. EXTRACT iterates over the set of predicates $remaining$ to be allocated to strata, until this set is empty. At each iteration, the next $stratum$ is built (lines 25-28) by examining remaining predicates in turn, selecting those whose ancestors in R have all been allocated to previous strata. I.e., the current stratum consists of those remaining predicates j such that $R[i,j] \neq 2$ for all remaining predicates i. Then the selected predicates are removed from $remaining$, the current stratum is incorporated to the stratification, and the $level$ of the next stratum to build is increased (lines 29-31). This process terminates in less than $|\mathcal{D}|$ iterations, and the stratification is returned.

3 Axioms Add Significant Expressive Power

It is clear that axioms add something to the expressive power of PDDL. In order to determine how much power is added, we will use the *compilability approach* (Nebel, 2000), which is based on results from the area of knowledge compilation (Cadoli and Donini, 1997). Basically, what we want to determine is how succinctly

[4] By $|\cdot|$ we denote the cardinality of a set.

a planning task can be represented if we compile the axioms away. [5] Furthermore, we want to know how long the corresponding plans in the compiled planning task will become.

As we will show, it is impossible to compile away axioms, provided we require that the domain description and the plan length both grow only polynomially. There is, of course, the question of what the main source of expressive power in this case is and how one could get around the problem. One way of answering this question would be to vary systematically the expressivity of the axiom language and of the operator language and then determine compilability between every pair of planning formalisms. This has been done, e.g., for variants of propositional STRIPS, where no axioms were allowed (Nebel, 2000). In the present case, however, such an analysis would certainly be much too extensive and we are, moreover, mainly interested in answering the question of how much expressivity axioms add to the existing planning formalism PDDL. For this reason, we will only consider what restrictions on the axiom language lead to.

In detail, we will consider the following three variants of axiom languages:

(1) the full axiom language as defined in Section 2;
(2) the axiom language restricted to function-free logic-programming rules without negation (which is also called DATALOG);
(3) the axiom language restricted to non-recursive DATALOG.

These variations address the following questions:

(1)→(2): Is the capability of using arbitrary quantification and Boolean connectors important? In particular, is negation in axioms significant?
(2)→(3): What role does recursion in axioms play?

In the following, we take a PDDL$_{\mathcal{X}}$ planning domain description to be a tuple $\Delta = \langle \mathcal{B}, \mathcal{D}, A, O \rangle$, where $\mathcal{B}$ is the set of basic predicates, $\mathcal{D}$ is the set of derived predicates, A is a stratified axiom set as in Definition 1, and O is a set of action descriptions (with the mentioned restriction that atomic effects cannot contain predicates in $\mathcal{D}$). A PDDL$_{\mathcal{X}}$ *planning instance* or *task* is a tuple $\Pi = \langle \Delta, \mathcal{C}, \mathcal{I}, \mathcal{G} \rangle$, where Δ is the domain description, $\mathcal{C}$ is the set of constant symbols, and $\mathcal{I}$ and $\mathcal{G}$ are the initial state (a set of ground basic atoms) and goal descriptions (a formula), respectively. The result of applying an action in a (basic) state and what constitutes a valid plan (sequence of actions) for a given planning task are defined in the usual way (Bacchus, 2000), except that the modelling relation of Definition 5 is used in place of the usual one. By a PDDL domain description and planning instances we mean those without any axioms and derived predicates, i.e., a PDDL domain description has the form $\langle \mathcal{B}, \emptyset, \emptyset, O \rangle$.

[5] Similar techniques have been used to measure the relative succinctness of logical representation formalisms by Cadoli et al. (1996) and Gogic et al. (1995).

We use *compilation schemes* (Nebel, 2000) to translate PDDL$_{\mathcal{X}}$ domain descriptions to PDDL domain descriptions. Such schemes are functions that translate domain descriptions between planning formalisms without any restriction on their computational resources but the constraint that the target domain should be only polynomially larger than the original. [6]

Definition 6 *A* compilation scheme from $\mathcal{X}$ to $\mathcal{Y}$ *is a tuple of functions* $\mathbf{f} = \langle f_\delta, f_c, f_i, f_g \rangle$ *that induces a function* F *from* $\mathcal{X}$*-instances* $\Pi = \langle \Delta, \mathcal{C}, \mathcal{I}, \mathcal{G} \rangle$ *to* $\mathcal{Y}$*-instances* $F(\Pi)$ *as follows:*

$$F(\Pi) = \langle f_\delta(\Delta), \mathcal{C} \cup f_c(\Delta), \mathcal{I} \cup f_i(\mathcal{C}, \Delta), \mathcal{G} \wedge f_g(\mathcal{C}, \Delta) \rangle$$

and satisfies the following conditions:

(1) there exists a plan for Π *iff there exists a plan for* $F(\Pi)$*,*
(2) and the size of the results of f_δ*,* f_c*,* f_i*, and* f_g *is polynomial in the size of their argument* Δ*.*

In addition, we measure the size of the corresponding plans in the target formalism. [7]

Definition 7 *If a compilation scheme* $\mathbf{f}$ *has the property that for every plan* P *solving an instance* Π*, there exists a plan* P' *solving* $F(\Pi)$ *such that* $||P'|| \leq ||P|| + k$ *for a positive integer* k*, we say that the compilation scheme* $\mathbf{f}$ preserves plan size exactly. *If we can guarantee that* $||P'|| \leq c \times ||P|| + k$ *for positive integer constants* c *and* k*, then we say* $\mathbf{f}$ preserves plan size linearly*, and if* $||P'|| \leq p(||P||, ||\Pi||)$ *for some polynomial* p*, then we say* $\mathbf{f}$ preserves plan size polynomially.

From a practical point of view, one can regard compilability preserving *plan size exactly* as an indication that the planning formalism we use as the target formalism is *at least as expressive* as the source formalism. In other words, the additional language features in the source formalism can be regarded as *syntactic sugar*. If a linear blowup is required, then the compilation process might already be a problem since planning algorithms are usually exponential in the length of the plan. If a linear growth is not sufficient and a polynomial blowup measured in the size of the domain description and the original plan length is required, it might be already infeasible to use the compilation technique. This may be taken as an indication that the source formalism *is more expressive* than the target formalism—since it indicates that a planning algorithm for the target formalism would be forced to generate significantly longer plans for compiled instances, making it probably infeasible to

[6] We use here a slightly modified definition of compilability, which incorporates the set of constant symbols $\mathcal{C}$. Furthermore, we have simplified the definition by not allowing general transformations of the initial and goal state but simple extensions of the respective state descriptions.

[7] The size of an instance, domain description, plan, etc. is denoted by $|| \cdot ||$.

solve such instances. If plans are required to grow even *super-polynomially*, then the increase of expressive power must be dramatic. Incidentally, such exponential growth of plan size is necessary to compile axioms away.

In order to investigate the compilability between PDDL and PDDL$_\mathcal{X}$, we will analyse restricted planning problems such as the *1-step planning problem* and the *polynomial step planning problem*. The former is the problem of whether there exists a 1-step plan to solve a planning task, the latter is the problem whether there exists a plan polynomially sized (for some fixed polynomial) in the representation of the domain description. Furthermore, we will often restrict the form of the right-hand side of axioms to a particularly simple form, namely, conjunctions of atoms, where all variables not appearing on the left-hand side of the axiom are implicitly existentially quantified. Such axioms are syntactically identical to DATALOG programs and for this reason we will call such axioms to be in DATALOG form. If the axioms contain negation, we say that they are in DATALOG$^\neg$ form. It is now a well-known fact from database theory that first-order queries can be rewritten into DATALOG$^\neg$ programs, which are linear in the size of the original formula (Abiteboul et al., 1995). For this reason, we can concentrate in the following on stratified axioms in DATALOG$^\neg$ form as the most expressive axiom language.

Theorem 1 *The 1-step planning problem for* PDDL$_\mathcal{X}$ *is EXPTIME-complete, even if all axioms are in pure* DATALOG *form.*

Proof. EXPTIME-hardness follows from EXPTIME-completeness of pure DATALOG entailment (Dantsin et al., 2001, Theorem 4.5). EXPTIME membership follows because the evaluation of the precondition and the goal formula can be done in PSPACE (Vardi, 1982) and the evaluation of axioms in stratified DATALOG$^\neg$ can be done in EXPTIME, which follows from EXPTIME-completeness of entailment in stratified DATALOG$^\neg$ programs (Dantsin et al., 2001, Theorem 5.1). ■

If we now consider PDDL planning tasks, it turns out that the planning problem is considerably easier, even if we allow for polynomial length plans. While in general STRIPS planning (and hence PDDL planning) is EXPSPACE-hard when variables are permitted (Erol et al., 1995), the restriction to plans of polynomially many steps leads to PSPACE-completeness. [8]

Theorem 2 *The polynomial step planning problem for* PDDL *is PSPACE-complete.*

Proof. PSPACE-hardness follows from the fact that the evaluation of quantified formulas (such as precondition and goal formulas) is already PSPACE-hard (Vardi,

[8] One should note that this result is unrelated to Bylander's PSPACE-completeness result for *propositional* planning. While Bylander's result is about arbitrarily long plans for operators that do not contain object variables, our result is about polynomially long plans for operators that contain variables.

1982).

Membership in PSPACE can be shown as follows. In order to solve the polynomial step planning problem, we can guess a polynomially sized plan and guess instantiations of the free variables in all operators. This can clearly be done using only polynomial space.

Now we can verify that the guessed plan solves the problem using only polynomial space. By iterating over each ground atom, we check that the goals are satisfied, checking recursively that the operators in the plan were executable and that the right atoms were generated (or deleted). This clearly takes only polynomial space. So the entire verification can be carried out in polynomial space. ■

From these two theorems it follows immediately that it is very unlikely that there exists a *polynomial time* compilation scheme from $\text{PDDL}_{\mathcal{X}}$ to PDDL preserving plan size polynomially. Otherwise, it would be possible to solve all problems requiring exponential time in polynomial space, which is considered as quite unlikely. However, as argued by Nebel (2000), if we want to make claims about *expressiveness*, then we should not take the computational resources of the compilation scheme into account but allow for computationally unconstrained transformations. Interestingly, even allowing for such unconstrained compilation schemes we get a similar result.

In order to prove this, we will use an idea similar to the one Kautz and Selman (1992) used to prove that approximations of logical theories of a certain size are not very likely to exist. In order to do so, we will describe all *linearly bounded alternating Turing machine acceptance problem instances* up to a certain size by one fixed $\text{PDDL}_{\mathcal{X}}$ domain description.

An *alternating Turing machine* (ATM) M is a tuple $\langle Q, \Sigma, \Gamma, \#, \delta, q_0, U, A\rangle$, where Q is a finite set of *states*, Σ is the *input alphabet*, $\Gamma \supset \Sigma$ is the *tape alphabet*, $\# \in \Gamma - \Sigma$ is the *blank symbol*, $\delta : (Q \times \Gamma) \rightarrow 2^{(Q\times\Gamma\times\{L,R,S\})}$ is the *transition function*, $q_0 \in Q$ is the *initial state*, $U \subseteq Q$ is the set of *universal states*, and $A \subseteq Q$ is the set of *accepting states*. All non-accepting, non-universal states are called *existential*. Such a machine is in an *accepting configuration* if

- the state is an accepting state,
- the state is an existential state and there exists a successor configuration that is an accepting configuration, or
- the state is a universal state and all successor configurations are accepting configurations.

A *linearly bounded ATM* (or LBATM) is an ATM whose tape head is not allowed to leave the space occupied by the input string. The *LBATM acceptance problem* is now the problem of deciding for a given LBATM and a given string, whether the string is accepted. This problem is EXPTIME-complete (Chandra et al., 1981).

In addition to the LBATM problem we need the notion of *advice-taking Turing machines* and of *non-uniform complexity classes* to prove our claim. An *advice-taking Turing machine* is a Turing machine with an *advice oracle*, which is a (not necessarily computable) function $a(\cdot)$ from positive integers to bit strings. On input w the machine loads the bit string $a(||w||)$ and then continues as usual. Note that the oracle derives its bit string only from the length of the input and not from the contents of the input. An advice is said to be *polynomial* if the oracle string is polynomially bounded by the instance size. Further, if X is a complexity class defined in terms of resource-bounded machines, e.g., P, NP or PSPACE, then X/poly (also called *non-uniform X*) is the class of problems that can be decided on machines with the same resource bounds and polynomial advice.

Theorem 3 *Unless EXPTIME = PSPACE, there is no compilation scheme from* PDDL$_\mathcal{X}$ *(even restricted to pure* DATALOG *axioms) to* PDDL *preserving plan size polynomially.*

Proof. Consider LBATM instances $I = \langle w, M\rangle$, with $w \in \{0,1\}^*$ and M being a LBATM with $\Sigma = \{0,1\}$. We measure the size of these instances by taking the maximum of the length of w and the number of states in M. As a next step, we specify a family of PDDL$_\mathcal{X}$ planning domains Δ_n. These are constructed in a way such that they can be used to decide the LBATM problem up to size n by solving the 1-step planning problem for $\langle \Delta_n, \mathcal{C}, \mathcal{I}_I, g\rangle$, where $\mathcal{C} = \{\mathtt{0}, \mathtt{1}, \#, \mathtt{U}, \mathtt{E}, \mathtt{A}, \mathtt{L}, \mathtt{R}, \mathtt{S}\}$, $\mathcal{I}_I$ describes the LBATM instance and can be computed from I in polynomial time, and g is some constant predicate. The numbers in $\mathcal{C}$ stand for elements of the input alphabet, $\#$ stands for the blank symbol, the symbols $\mathtt{U}, \mathtt{E}, \mathtt{A}$ are used to denote, universal, existential, and accepting states, respectively, and $\mathtt{L}, \mathtt{R}, \mathtt{S}$ are used to denote head movement, i.e., L for left, R for right, and S for stationary.

The *basic predicates* of the PDDL$_\mathcal{X}$ instance we are constructing are:

- $(\mathtt{cell}_i\ ?s)$ describing that the ith tape cell of the input contains $?s$, with $i = 0$ denoting the leftmost cell,
- $(\mathtt{type}_q\ ?t)$ describing the type $?t$ of state q, with $q = 0$ denoting the initial state,
- $(\mathtt{trans}_{q,q'}\ ?s\ ?s'\ ?m)$ describing one entry of the transition table corresponding to $\delta(q, ?s) \ni \langle q', ?s', ?m\rangle$, and
- $(\mathtt{notrans}_{q,q'}\ ?s\ ?s'\ ?m)$ describing that the transition table does not contain an entry $\langle q', ?s', ?m\rangle$ at $\delta(q, ?s)$.

Using these basic predicates, every LBATM instance with input size less or equal n can be described.

In addition, we use the following derived predicates:

- $(\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{n-1})$ describing an accepting configuration with tape contents $?x_0 \ldots ?x_{n-1}$, state q $(0 \leq q \leq n-1)$, and head position i $(0 \leq i \leq n-1)$,
- $(\mathtt{ok}_{q,i,q',s',m}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$ describing that the successor con-

figuration resulting from $\langle q', s', m\rangle$ for given state q and input $?s'$ is either an accepting configuration or it is not reachable (i.e., $\langle q', s', m\rangle \notin \delta(q, ?s)$),
- $(\mathtt{g})$ is the goal atom which is added by the only operator in our domain description.

Now we have for every state q and head position i the following rule accounting for accepting states:

$(:\ \mathtt{derived}\ (\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{n-1})\ (\mathtt{type}_q\ \mathtt{A}))$

Additionally, for every tuple $\langle q, q', i\rangle$ we have the following rules:

For $1 \leq i \leq n-1$, we have left-movement rules:
$(:\ \mathtt{derived}\ (\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad(\mathtt{and}$
$\quad\quad(\mathtt{type}_q\ \mathtt{E})$
$\quad\quad(\mathtt{trans}_{q,q'}\ ?s\ ?s'\ \mathtt{L})$
$\quad\quad(\mathtt{acc}_{q',i-1}\ ?x_0 \ldots ?x_{i-1}\ ?s'\ ?x_{i+1} \ldots ?x_{n-1})))$

For $0 \leq i \leq n-2$, we have right-movement rules:
$(:\ \mathtt{derived}\ (\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad(\mathtt{and}$
$\quad\quad(\mathtt{type}_q\ \mathtt{E})$
$\quad\quad(\mathtt{trans}_{q,q'}\ ?s\ ?s'\ \mathtt{R})$
$\quad\quad(\mathtt{acc}_{q',i+1}\ ?x_0 \ldots ?x_{i-1}\ ?s'\ ?x_{i+1} \ldots ?x_{n-1})))$

For $0 \leq i \leq n-1$, we have stay rules:
$(:\ \mathtt{derived}\ (\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad(\mathtt{and}$
$\quad\quad(\mathtt{type}_q\ \mathtt{E})$
$\quad\quad(\mathtt{trans}_{q,q'}\ ?s\ ?s'\ \mathtt{S})$
$\quad\quad(\mathtt{acc}_{q',i}\ ?x_0 \ldots ?x_{i-1}\ ?s'\ ?x_{i+1} \ldots ?x_{n-1})))$

In words, we consider a configuration with an existential state as accepting if there exists a successor configuration that is accepting.

The semantics of universal configurations is described with the following rules for every tuple $\langle q, i\rangle$:

$(:\ \mathtt{derived}\ (\mathtt{acc}_{q,i}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad(\mathtt{and}$
$\quad\quad(\mathtt{type}_q\ \mathtt{U})$
$\quad\quad(\mathtt{and}$
$\quad\quad\quad(\mathtt{ok}_{q,i,0,0,L}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad\quad\quad(\mathtt{ok}_{q,i,0,1,L}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad\quad\quad(\mathtt{ok}_{q,i,0,0,R}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\quad\quad\quad(\mathtt{ok}_{q,i,0,1,R}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$

$(\texttt{ok}_{q,i,0,0,S}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,0,1,S}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$\vdots$
$(\texttt{ok}_{q,i,n-1,0,L}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,n-1,1,L}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,n-1,0,R}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,n-1,1,R}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,n-1,0,S}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
$(\texttt{ok}_{q,i,n-1,1,S}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1}))))$

For each tuple $\langle q, i, q', s' \rangle$, we have now the following set of rules:

$(:\ \texttt{derived}\ (\texttt{ok}_{q,i,q',s',m}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
  $(\texttt{notrans}_{q,q'}\ ?s\ s'\ m))$
$(:\ \texttt{derived}\ (\texttt{ok}_{q,i,q',s',\texttt{L}}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
  $(\texttt{and}$
    $(\texttt{trans}_{q,q'}\ ?s\ s'\ \texttt{L})$
    $(\texttt{acc}_{q',i-1}\ ?x_0 \ldots ?x_{i-1}\ s'\ ?x_{i+1} \ldots ?x_{n-1})))$
$(:\ \texttt{derived}\ (\texttt{ok}_{q,i,q',s',\texttt{R}}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
  $(\texttt{and}$
    $(\texttt{trans}_{q,q'}\ ?s\ s'\ \texttt{R})$
    $(\texttt{acc}_{q',i+1}\ ?x_0 \ldots ?x_{i-1}\ s'\ ?x_{i+1} \ldots ?x_{n-1})))$
$(:\ \texttt{derived}\ (\texttt{ok}_{q,i,q',s',\texttt{S}}\ ?x_0 \ldots ?x_{i-1}\ ?s\ ?x_{i+1} \ldots ?x_{n-1})$
  $(\texttt{and}$
    $(\texttt{trans}_{q,q'}\ ?s\ s'\ \texttt{S})$
    $(\texttt{acc}_{q',i}\ ?x_0 \ldots ?x_{i-1}\ s'\ ?x_{i+1} \ldots ?x_{n-1})))$

Now the only operator we need is the following:

$(:\ \texttt{action a}$
  $:\ \texttt{parameters}\ (?x_0 \ldots ?x_{n-1})$
  $:\ \texttt{precondition}$
    $(\texttt{and}$
      $(\texttt{cell}_0\ ?x_0)$
      $(\texttt{cell}_1\ ?x_1)$
      $\vdots$
      $(\texttt{cell}_{n-1}\ ?x_{n-1})$
      $(\texttt{acc}_{0,0}\ ?x_0 \ldots ?x_{n-1}))$
  $:\ \texttt{effect}\ (\texttt{g}))$

Let $I = \langle w, M \rangle$ be an LBATM instance of size n. Let $\mathcal{I}_I$ be the initial planning state describing M and w using the basic predicates. It is then clear that the constructed PDDL$_\mathcal{X}$ instance $\Pi_n = \langle \Delta_n, \mathcal{C}, \mathcal{I}_I, g \rangle$ has a successful 1-step plan if and only if w is accepted by M.

Let us now assume that there exists a compilation scheme from PDDL$_{\mathcal{X}}$ to PDDL preserving plan size polynomially. Such a scheme could be used to derive a *polynomial advice* for an advice-taking Turing machine in the following way. Let I be an LBATM instance of size n, then the compilation of Δ_n to a PDDL domain structure Δ'_n can be used as the polynomial advice. The advice-taking Turing machines reads the instance, loads the advice Δ'_n, computes $\mathcal{I}_I$ and then decides polynomial-step PDDL plan existence, which can be done in PSPACE because of Theorem 2. This, however, implies, that all of EXPTIME can be decided in PSPACE/poly, which by the results of Karp and Lipton (1982) implies that EXPTIME = PSPACE. ■

This result strongly suggests that compilation approaches that try to compile general axioms away are most probably doomed to failure. Either the domain descriptions or the plans will be blown up exponentially, which means that current automated planning techniques will most probably fail on such compiled instances. As a matter of fact, our experiments described in Section 5 seem to confirm this.

Furthermore, the result also indicates that not all of the expressive power of the axiom language is necessary in order to get such a strong result. Simple DATALOG (without negation) suffices to achieve the result.

However, if we restrict our axiom sets to be *non-recursive, stratified* DATALOG$^{\neg}$, things change. A set of axioms is called non-recursive if it has a stratification such that each predicate occurs in its defining stratum only on the left-hand side of axioms. As it is well known, entailment for such kind of axiom sets is PSPACE-complete (Dantsin et al., 2001, Theorem 5.3). Hence, the arguments in the proofs of Theorem 1 and Theorem 3 do not work any longer. Moreover, it is also obvious how to construct a polynomial-time compilation scheme that preserves plan length polynomially in the size of the original plan and the domain description.

It is much less clear, however, whether a compilation scheme preserving plan size linearly or exactly would be possible. The reason is that simply replacing derived predicates by their definitions could result in an exponentially large formula. using results from database theory, however, lead to the desired compilation scheme.

Theorem 4 *There exists a polynomial-time compilation scheme from* PDDL$_{\mathcal{X}}$ *to* PDDL *preserving plan size exactlty, provided the axioms are in* DATALOG$^{\neg}$ *form and non-recursive.*

Proof. We can translate each derived predicate that is used in a precondition or goal formula in polynomial time into a first-order query with only a linear increase in size.

This follows from the statement in the proof of the complexity of evaluating relational algebra programs (Vardi, 1982, Theorem 9) that relational algebra programs can be translated into first-order queries that are only linearly sized in the original

program. A construction how to do that can be found in a paper by Vorobyov and Voronkov (1997).

This construction needs equality, which we do not have. However, one can easily introduce an extensional equality relation by extending the initial state with atoms $(\texttt{EQ}\ \ x\ \ y)$ for all constants x and y. ■

In other words, it is recursion that makes axioms so powerful. If we disallow recursion, then axioms are simply *syntactic sugar* and can be compiled away.

Another variation that would be worthwhile to analyse is propositional PDDL and PDDL$_{\mathcal{X}}$. We will not do that here, but just note that there exists an obvious compilation scheme that preserves plan length polynomially. Furthermore, it seems very unlikely that there exists a compilation scheme that preserves plan length linearly. The reason is that evaluating propositional, stratified DATALOG$^{\neg}$ (even without recursion) is PTIME-complete (Dantsin et al., 2001, Theorem 5.5), while evaluating Boolean expressions is in LOGSPACE (Lynch, 1977).

4 Compilations with Exponential Results

While it is impossible to find a succinct equivalent PDDL planning instance that guarantees short plans, it is possible to come up with a poly-size instance which may have exponentially longer plans in the worst case. Such compilation schemes have been described by e.g. Gazen and Knoblock (1997) and Garagnani (2000) under severe restrictions on the use of negated derived predicates.

Specifically, the former scheme (Gazen and Knoblock, 1997) translates an axiom $(\texttt{:derived}\ (d\ ?\vec{x})\ (f\ ?\vec{x}))$ into an extra "axiom" operator with parameters $(?\vec{x})$ having the axiom's antecedent $(f\ ?\vec{x})$ as precondition and its consequent $(d\ ?\vec{x})$ as effect. It also augments those of the (original) operators which affect any predicate present in the axiom's antecedent with the effect $(\texttt{forall}\ ?\vec{\texttt{x}}\ (\texttt{not}(\texttt{d}\ ?\vec{\texttt{x}})))$ deleting all ground instances of the consequent. This scheme, when appropriately generalised to a set of axioms by repeatedly examining the impact of axioms on the set of operators until a fix point is reached — this generalisation is not discussed in (Gazen and Knoblock, 1997) — gives the *possibility* to the planner of inferring *positive* derived literals needed in a plan. However, it does not force the planner to establish the actual truth of any of the derived predicates. For this reason, serious problems arise if negated derived predicates appear anywhere in the planning task. For instance consider the PDDL$_{\mathcal{X}}$ task $\mathcal{B} = \{\texttt{a}\}$, $\mathcal{D} = \{\texttt{b}\}$, $O = \{(\texttt{:action op :parameters ()}\ \ \texttt{:effect}\ (\texttt{a}))\}$, $A = \{(\texttt{:derived(b)(a)})\}$, $\mathcal{C} = \emptyset$, $\mathcal{I} = \emptyset$, $\mathcal{G} = (\texttt{and}\ (\texttt{a})\ (\texttt{not b}))$. This task is not solvable because establishing a also establishes b via the axiom. Yet, the scheme yields a compiled task solvable by the plan $(\texttt{op})$, because nothing forces the planner to execute the axiom

action after `op` to derive that `b` actually holds. This remains true even if negation is compiled away as per the method of Gazen and Knoblock (1997).

The latter scheme (Garagnani, 2000), is further restricted to DATALOG axioms, and suffers from the same worst-case plan length and from the same problems in the presence of negated derived predicates. However, whereas the Gazen Knoblock scheme deletes at once *all* ground instances of a derived predicate as soon as *one* ground instance of that predicate is put at risk by the performance of an action, Garagnani's scheme keeps track of which ground derived atoms really need to be deleted. New predicates are introduced to record the instances of applications of the axiom operators. The ground instances of these predicates can be used to identify which antecedents have led to which consequences at given steps in the plan, so that if one of these antecedents is deleted, all and only its consequences can be identified and removed from the state. These removals are implemented as conditional effects in the original operators.

An interesting contrasting approach is that of Davidson and Garagnani (2002). They propose to compile DATALOG axioms solely into conditional effects of existing operators, which means that the resulting plans will have exactly the same length. However, as is implied by Theorem 3, the domain description suffers a super-polynomial growth. More precisely, non-recursive axioms are compiled away using backward chaining, that is by substituting, in preconditions, their definition for the derived predicates until none remains. Recursive axioms are compiled away by using forward chaining to find the consequences of predicates in effect descriptions and asserting these consequences as additional conditional effects.

We now specify a generally applicable compilation scheme producing poly-size instances, which we will use as a baseline in our performance evaluation. In contrast to the schemes mentioned above, it complies with the stratified semantics specified in Section 2 while dealing with negated occurrences of derived predicates anywhere in the planning task. Figure 3 shows the domain description that results from applying this scheme to the Blocks World domain in Figure 2, and is meant to help the reader understand what follows.

Theorem 5 *There exists a polynomial time compilation scheme* $\mathbf{f} = \langle f_\delta, f_c, f_i, f_g \rangle$*, such that for every* PDDL$_\mathcal{X}$ *domain description* $\Delta = \langle \mathcal{B}, \mathcal{D}, A, O \rangle$*:*

- $||f_c(\Delta)|| = 0$*,*
- $||f_i(\mathcal{C}, \Delta)|| = c_1$ *for some constant* c_1*,*
- $||f_g(\mathcal{C}, \Delta)|| = c_2$ *for some constant* c_2*,*
- $f_\delta(\Delta) = \langle \mathcal{B}', \emptyset, \emptyset, O' \rangle$ *is a* PDDL *domain with* $|\mathcal{B}'| \leq |\mathcal{B}| + 3\ |\mathcal{D}| + 2$ *and with* $||O'|| \leq p(||O||, ||A||)$ *for some polynomial* p*.*

Proof. Figure 4 shows the PDDL instances induced by **f**. **f** computes a stratification $\{A_i, 1 \leq i \leq n\}$ of the set of axioms A, as explained in Section 2, where in stratum i, each axiom $a_{i,j}$ is of the form $(:\ \texttt{derived}\ (d_{i,j}\ ?\vec{x}_{i,j})\ (f_{i,j}\ ?\vec{x}_{i,j}))$ for $1 \leq j \leq |A_i|$. **f**

Fig. 3 Blocks World with Derived Predicates Compiled Away

```
(define (domain blocksworld-compiled)
  (:requirements :strips)
  (:predicates (on-table ?x) (on ?x ?y) (holding ?x) (above ?x ?y) (clear ?x) (handempty)
               (fixed-0) (fixed-1) (fixed-2) (done-1) (done-2) (new))

  (:action stratum-1
     :precondition (and (fixed-0) (not (fixed-1)))
     :effect (and (done-1)
                  (forall (?x)
                     (when (and (not (holding ?x))
                                (not (on-table ?x))
                                (not (exists (?y) (on ?x ?y))))
                           (and (holding ?x)
                                (new))))
                  (forall (?x ?y)
                     (when (and (not (above ?x ?y))
                                (or (on ?x ?y)
                                    (exists (?z) (and (on ?x ?z) (above ?z ?y)))))
                           (and (above ?x ?y)
                                (new))))))
  (:action fixpoint-1
     :parameters ()
     :precondition (done-1)
     :effect (and (when (not (new)) (fixed-1))
                  (not (new))
                  (not (done-1))))

  (:action axiom-2
     :precondition (and (fixed-1) (not (fixed-2)))
     :effect (and (done-2)
                  (forall (?x)
                  (when (and (not (clear ?x))
                             (not (holding ?x))
                             (not (exists (?y) (on ?y ?x))))
                        (and (clear ?x)
                             (new))))
                  (when (and (not (handempty))
                             (forall (?x) (not (holding ?x))))
                        (and (handempty)
                             (new)))))

  (:action fixpoint-2
     :parameters ()
     :precondition (done-2)
     :effect (and (when (not (new)) (fixed-2))
                  (not (new))
                  (not (done-2))))

  (:action pickup
    :parameters (?ob)
    :precondition (and (fixed-2) (clear ?ob) (on-table ?ob) (handempty))
    :effect (and (not (on-table ?ob))
                 (not (fixed-1)) (not (fixed-2))
                 (not (done-1)) (not (done-2))
                 (forall (?x) (not (holding ?x)))
                 (forall (?x ?y) (not (above ?x ?y)))
                 (forall (?x) (not (clear ?x)))
                 (not (handempty))))

  (:action putdown
    :parameters  (?ob)
    :precondition (and (fixed-1)(holding ?ob))
    :effect (and (on-table ?ob)
                 (not (fixed-1)) (not (fixed-2))
                 (not (done-1)) (not (done-2))
                 (forall (?x) (not (holding ?x)))
                 (forall (?x ?y) (not (above ?x ?y)))
                 (forall (?x) (not (clear ?x)))
                 (not (handempty))))

  (:action stack
    :parameters  (?ob ?underob)
    :precondition (and (fixed-2) (clear ?underob) (holding ?ob))
    :effect (and (on ?ob ?underob)
                 (not (fixed-1)) (not (fixed-2))
                 (not (done-1)) (not (done-2))
                 (forall (?x) (not (holding ?x)))
                 (forall (?x ?y) (not (above ?x ?y)))
                 (forall (?x) (not (clear ?x)))
                 (not (handempty))))

  (:action unstack
    :parameters  (?ob ?underob)
    :precondition (and (fixed-2)(on ?ob ?underob) (clear ?ob) (handempty))
    :effect (and (not (on ?ob ?underob))
                 (not (fixed-1)) (not (fixed-2))
                 (not (done-1)) (not (done-2))
                 (forall (?x) (not (holding ?x)))
                 (forall (?x ?y) (not (above ?x ?y)))
                 (forall (?x) (not (clear ?x)))
                 (not (handempty))))
)
```

Fig. 4 PDDL instances induced by **f**

1. (: predicates ; all predicates in $\mathcal{B} \cup \mathcal{D}$
2. $(\texttt{done}_1) \ldots (\texttt{done}_n)$
3. $(\texttt{fixed}_0) \ldots (\texttt{fixed}_n)$
4. (new))

for each $i \in \{1, \ldots, n\}$

5. (: action $\texttt{stratum}_i$
6. : parameters ()
7. : precondition (and $(\texttt{fixed}_{i-1})$ (not $(\texttt{fixed}_i)$))
8. : effect (and $(\texttt{done}_i)$
9. (forall $(?\vec{x}_{i,1})$
10. (when (and $(f_{i,1}\ ?\vec{x}_{i,1})$ (not $(d_{i,1}\ ?\vec{x}_{i,1})$))
11. (and $(d_{i,1}\ ?\vec{x}_{i,1})$ (new))))
12. ...
13. (forall $(?\vec{x}_{i,|A_i|})$
14. (when (and $(f_{i,|A_i|}\ ?\vec{x}_{i,|A_i|})$ (not $(d_{i,|A_i|}\ ?\vec{x}_{i,|A_i|})$))
15. (and $(d_{i,|A_i|}\ ?\vec{x}_{i,|A_i|})$ (new))))))
16. (: action $\texttt{fixpoint}_i$
17. : parameters ()
18. : precondition $(\texttt{done}_i)$
19. : effect (and (when (not(new)) $(\texttt{fixed}_i)$)
20. (not(new))
21. (not $(\texttt{done}_i)$))))

for each $o \in O$

22. (: action NAME(o)
23. : parameters PARAMETERS(o)
24. : precondition (and PRECONDITION(o) $(\texttt{fixed}_k)$)
25. : effect (and EFFECT(o)
26. (not $(\texttt{fixed}_m)$) ... (not $(\texttt{fixed}_n)$)
27. (not $(\texttt{done}_m)$) ... (not $(\texttt{done}_n)$)
28. (forall $(?\vec{x}_{m,1})$ (not $(d_{m,1}\ ?\vec{x}_{m,1})$))
29. ...
30. (forall $(?\vec{x}_{n,|A_n|})$ (not $(d_{n,|A_n|}\ ?\vec{x}_{n,|A_n|})$)))))

Where $k = \max(\{i \mid \text{some } d_{i,j} \text{ occurs in PRECONDITION}(o)\} \cup \{0\})$ and $m = \min(\{i \mid \text{a predicate in some } f_{i,j} \text{ is modified in EFFECT}(o)\} \cup \{n{+}1\})$

31. (: init $\mathcal{I} \cup (\texttt{fixed}_0)$)
32. (: goal (and $\mathcal{G}$ $(\texttt{fixed}_k)$))

Where $k = \max(\{i \mid \text{some } d_{i,j} \text{ occurs in } \mathcal{G}\} \cup \{0\})$

encodes each stratum as an extra action $\texttt{stratum}_i$ (see lines 5-15 in Figure 4) which applies all axioms $a_{i,j}$ at this stratum in parallel, records that this was done ($\texttt{done}_i$) and whether anything new (new) was derived in doing so. Each $a_{i,j}$ is encoded as a universally quantified and conditional effect of $\texttt{stratum}_i$—see lines 9-15. To ensure that the precedence between strata is respected, $\texttt{stratum}_i$ is only applicable

when the fixed point for the previous stratum has been reached (i.e. when $\texttt{fixed}_{i-1}$) and the fixed point for the current stratum has not (i.e. when $(\texttt{not}\ (\texttt{fixed}_i))$)—see line 7. **f** encodes the fixpoint computation at each stratum i using an extra action $\texttt{fixpoint}_i$, which is applicable after a round of one or more applications of $\texttt{stratum}_i$ (i.e., when $\texttt{done}_i$ is true), asserts that the fixed point has been reached (i.e. $\texttt{fixed}_i$) whenever nothing new has been derived during this last round, and resets new and $\texttt{done}_i$ for the next round—see lines 16-21. Next, the precondition and effect of each action description $o \in O$ are augmented as follows (see lines 22-30). Let $0 \leq k \leq n$ be the highest stratum of any derived predicate appearing in the precondition of o, or 0 if there is no such predicate. Before applying o, we must make sure that the fixed point for that stratum has been computed by adding $\texttt{fixed}_k$ to the precondition. Similarly, let $1 \leq m \leq n+1$ be the lowest stratum such that some predicate in the antecedent of some axiom in A_m is modified in the effect of o, or $n+1$ if there is none. After applying o, we may need to re-compute the fixed points for the strata above m, that is, the effect must reset fixed, done, and the value of all derived propositions, at strata m and above. Finally, $\texttt{fixed}_0$ holds initially, and the goal requires $\texttt{fixed}_k$ to be true, where $0 \leq k \leq n$ is the highest stratum of any derived predicate appearing in $\mathcal{G}$ or 0 if there is no such predicate [9] —see lines 31-32.

The fact that **f** preserves domain description size polynomially, and the bounds given in theorem 5, follow directly from the construction. Let $\Delta = \langle \mathcal{B}, \mathcal{D}, A, O \rangle$ be a PDDL$_\mathcal{X}$ instance. We have $f_i(\mathcal{C}, \Delta) = (\texttt{fixed}_0)$ and so $||f_i(\mathcal{C}, \Delta)||$ is a constant. $f_g(\mathcal{C}, \Delta) = (\texttt{fixed}_k)$ for some $0 \leq k \leq n$ and so $||f_g(\mathcal{C}, \Delta)||$ is a constant. $f_\delta(\Delta)$ is the PDDL domain description $\langle \mathcal{B}', \emptyset, \emptyset, O' \rangle$, where $\mathcal{B}' = \mathcal{B} \cup \mathcal{D} \cup \{(\texttt{fixed}_i) \mid 0 \leq i \leq n\} \cup \{(\texttt{done}_i) \mid 1 \leq i \leq n\} \cup \{(\texttt{new})\}$ and $O' = \{\texttt{stratum}_i \mid 1 \leq i \leq n\} \cup \{\texttt{fixpoint}_i \mid 1 \leq i \leq n\} \cup \{o' \mid o'$ is the augmentation of some $o \in O$ via **f**$\}$. So $|\mathcal{B}'| = |\mathcal{B}| + |\mathcal{D}| + 2n + 2$ and since $|\mathcal{D}| \leq n$, then $|\mathcal{B}'| \leq |\mathcal{B}| + 3\,|\mathcal{D}| + 2$. Obviously, the size of each $\texttt{fixpoint}_i$ action description is bounded by a constant, that of each $\texttt{stratum}_i$ is linear in $||A_i||$, and the size of each other action description is only augmented by a quantity linear in $||A||$. Therefore, the total size of O' is bounded by some polynomial in $||A||$ and $||O||$.

It remains to demonstrate the correctness of the compilation scheme, i.e., that there is a plan for a PDDL$_\mathcal{X}$ instance Π iff there is a plan for the PDDL instance $F(\Pi)$ defined via **f**. The direction from right to left follows from the following observation:

(I) Let Σ be a reachable state in $F(\Pi)$, such that Σ contains $\texttt{fixed}_k$ for $0 \leq k \leq n$. Let S be the basic atoms in Σ (ground instances of predicates in $\mathcal{B}$). Let D be the derived atoms in Σ up to stratum k (ground instances of predicates $d_{i,j} \in \mathcal{D}$ with $1 \leq i \leq k$). Then $D = [\![A]\!]_k(S)$.

[9] The resulting f_g does not strictly obey our simplified definition of compilation schemes, which forbids it to look at $\mathcal{G}$. The attentive reader may observe that using $\texttt{fixed}_n$ in place of $\texttt{fixed}_k$ sets everything right, at the cost of some efficiency.

In short, when $\texttt{fixed}_k$ is contained in a state in the compiled task, then the derived predicates are correctly computed up to stratum k. With this, and the $\texttt{fixed}_k$ conditions introduced by **f** as shown in Figure 4, a plan P for Π can be constructed from a plan P' for $F(\Pi)$ simply by skipping all $\texttt{stratum}_i$ and $\texttt{fixpoint}_i$ actions in P'. All action preconditions in P, and the goal, will be satisfied due to the semantics of the $\models_A$ relation. The other direction of the proof follows from this second observation:

(II) Let Σ be a reachable state in $F(\Pi)$, such that Σ contains $\texttt{fixed}_k$ for $0 \leq k \leq n$. Then, for any k', $0 \leq k < k' \leq n$, there is a sequence of $\texttt{stratum}_i$ and $\texttt{fixpoint}_i$ actions ($k+1 \leq i \leq k'$) leading to a state Σ' that contains $\texttt{fixed}_{k'}$.

With this, and observation (I), a plan P' for $F(\Pi)$ can be constructed from a plan P for Π by inserting an appropriate sequence of axiom fixpoint computations ($\texttt{stratum}_i$ and $\texttt{fixpoint}_i$ actions) in front of each action in P, and at the end of P. One just needs to achieve, in order to satisfy an action precondition or the goal, $\texttt{fixed}_k$ beforehand, where k is the highest stratum that the condition refers to.

The observations (I) and (II) follow directly from the construction of **f**. Observation (I) can be seen by induction over k. For $k = 0$ there is nothing to show. For any reachable state Σ' that contains $\texttt{fixed}_{k+1}$, there is a state Σ such that: Σ contains $\texttt{fixed}_k$; Σ contains no (ground instances of) derived predicates at stratum $k+1$; Σ' is reached from Σ by applying $\texttt{stratum}_{k+1}$ actions until a fixpoint occurs (where the $\texttt{stratum}_{k+1}$ actions are interleaved with $\texttt{fixpoint}_{k+1}$ actions, and, possibly, augmented actions from O that can not affect the truth of derived predicates at strata $j \leq k+1$). Observation (II) can be seen as follows. From a state Σ that contains $\texttt{fixed}_k$, one can get to a state Σ' that contains $\texttt{fixed}_{k+1}$ by: applying the $\texttt{stratum}_{k+1}$ action $|\overline{\mathcal{D}_{k+1}}|$ times, where $\overline{\mathcal{D}_{k+1}}$ denotes the ground instances of derived predicates in stratum $k+1$ (after that, all possible atoms at this stratum are derived); applying $\texttt{fixpoint}_{k+1}$ (to delete `new`); applying $\texttt{stratum}_{k+1}$ once (to re-achieve $\texttt{done}_{k+1}$); applying $\texttt{fixpoint}_{k+1}$ again (to achieve $\texttt{fixed}_{k+1}$). ■

Plans P for PDDL$_\mathcal{X}$ tasks Π and plans P' for compiled PDDL tasks $F(\Pi)$ correspond to each other modulo removal/insertion of `stratum` and `fixpoint` actions. The number of such actions is worst-case exponential, i.e., there is no polynomial p such that $||P'|| \leq p(||P||, ||\Pi||)$ for all possible tasks Π and plans P. The worst-case occurs when, initially and after each action from P, all derived predicates need to be (re)computed and only one proposition is ever derived per application of $\texttt{stratum}_i$ actions. Even if the planner is able to interleave as few $\texttt{fixpoint}_i$ actions as possible with the $\texttt{stratum}_i$ actions, this still leads to a plan of length $||P'|| = ||P|| + (||P|| + 1)(\sum_{i=1}^{n}(|\overline{\mathcal{D}_i}| + 3)) = ||P|| + (||P|| + 1)(3n + |\overline{\mathcal{D}}|)$. Observe that $|\overline{\mathcal{D}}|$ is not polynomially bounded in $|\mathcal{D}|$ and $|\mathcal{C}|$ ($|\overline{\mathcal{D}}|$ is exponential in the arity of the derived predicates).

5 Planning: With or Without Axioms?

The absence of a compilation scheme preserving plan size polynomially not only indicates that axioms bring (much needed) expressive power, but it also suggests that extending a planner to explicitly deal with axioms may lead to much better performance than using a compilation scheme with the original version of the planner. To confirm this hypothesis, we extended the FF planner (Hoffmann and Nebel, 2001) with a straightforward implementation of axioms—we call this extension FF$_{\mathcal{X}}$—and compared results obtained by FF$_{\mathcal{X}}$ on PDDL$_{\mathcal{X}}$ instances with those obtained by FF+**f**, i.e. by FF on the PDDL instances produced via compilation with **f**.

FF$_{\mathcal{X}}$ transforms each axiom $(:\texttt{derived}(d\ ?\vec{x})(f\ ?\vec{x}))$ into an operator with parameters $(?\vec{x})$, precondition $(f\ ?\vec{x})$ and effect $(d\ ?\vec{x})$, with a flag set to distinguish it from a "normal" operator. During the relaxed planning process that FF performs to obtain its heuristic function, the axiom actions are treated as normal actions and can be chosen for inclusion in a relaxed plan. However, the heuristic value only counts the number of *normal* actions in the relaxed plan. During the forward search FF performs, only normal actions are considered; after each application of such an action, the axiom actions are applied so as to obtain the successive fixed points associated with the stratification computed by Algorithm 1.

Note that FF$_{\mathcal{X}}$ differs from FF+**f** in the two ways regarding the special case treatment of axiom actions in the heuristic function, and their special case treatment in generating the search space. It is theoretically possible to obtain two versions of FF that are halfway between FF$_{\mathcal{X}}$ and FF+**f**, by dropping either of these two special case treatments. We did not try this because the resulting planning methods do not make much sense intuitively, due to mismatches between the search spaces and the used heuristic functions. If one treats axiom actions in the search space like FF$_{\mathcal{X}}$ does, thereby avoiding state transitions that are only needed to take care of the values of derived predicates, then it makes no sense to count axioms as normal actions in the relaxed plan. The number of actions in the relaxed plan is supposed to estimate the number of necessary state transitions (until the goal can be reached), so there is no justification for including the axiom actions in the count. Doing so would correspond to an unnecessary over-estimation. The other way round, say one treats the semantics of axioms by introducing additional state transitions, as FF+**f** does. Then it makes no sense to not take account of these additional transitions (axiom actions) in the relaxed plans. This would lead to an unnecessary under-estimation. In our experiments, we therefore focused only on FF$_{\mathcal{X}}$, in which both special case treatments are switched on, and FF+**f**, in which they are both switched off.

One domain we chose for our experiments is Blocks World (BW). In contrast to most other common benchmarks, in BW there is a natural distinction between basic and derived predicates; in particular BW with 4 operators is the only common benchmark domain we are aware of where the stratification of the axioms requires more than one stratum. [10] We experimented with two versions of BW:

1op: The version with a single `move` operator. `on` is the only basic predicate, the table being treated as a (special kind of) block. There is a single stratum consisting of the `clear` and `above` derived predicates. Note that `above` is only used in goal descriptions.

4ops: The version we used earlier as an example with the 4 operators `pickup`, `putdown`, `stack` and `unstack`. The basic predicates are `on` and `ontable`, and the derived ones are `above` and `holding` (stratum 1), as well as `clear` and `handempty` (stratum 2) whose axiomatisations use the negation of `holding`.

For each of those versions, we considered 3 types of planning tasks:

strict: A PDDL$_{\mathcal{X}}$ task is built from a given pair of BW states as follows. The first state is taken to be the initial one, and the second is converted into an incompletely specified goal description by writing "`above`" whereas one would normally have written "`on`" and omitting the mention of those blocks that would normally have been on the table. [11]

loose: A PDDL$_{\mathcal{X}}$ task is built from a single BW state by taking it to be the initial state and asking that any block which is on the table initially end up above all those that were initially not.

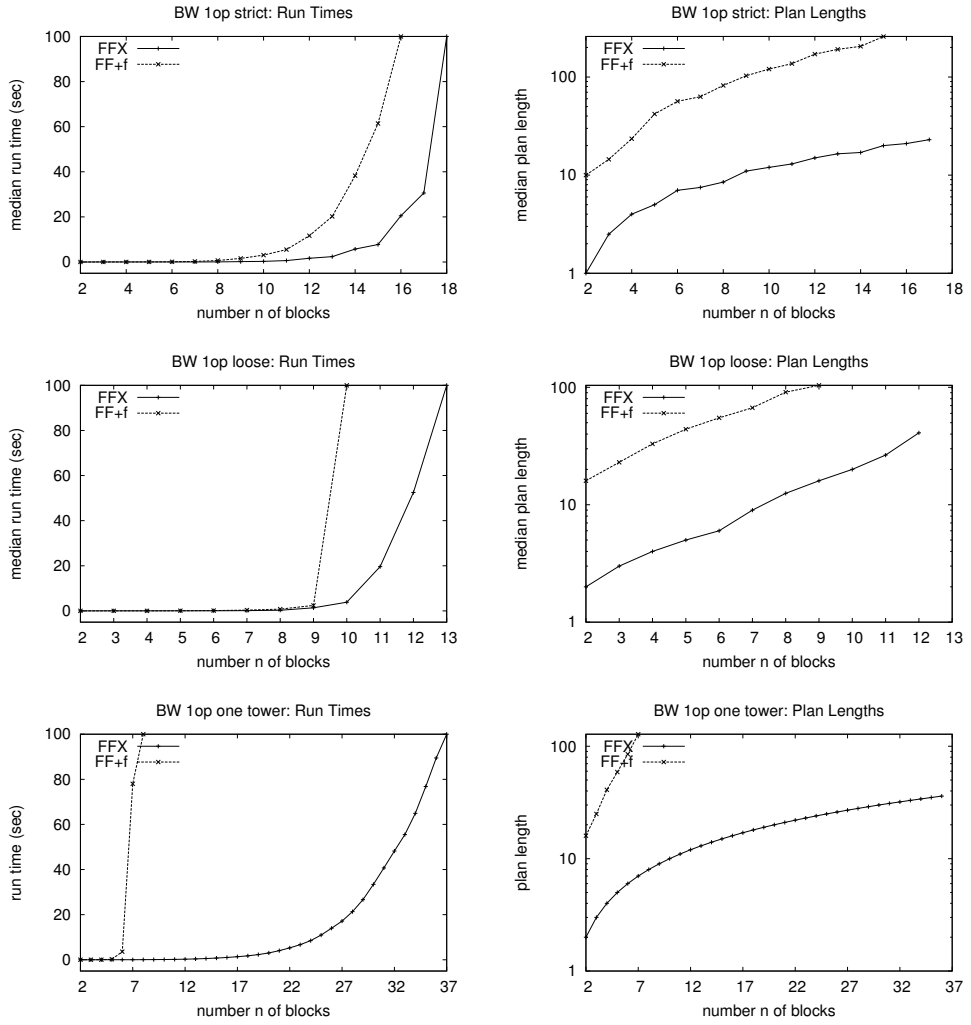
one tower: This is the special case of those **loose** tasks for which the initial state has only one tower. In their **1op** versions, those tasks are one of the examples considered by Davidson and Garagnani (2002).

For each combination $\{\mathbf{1op}, \mathbf{4ops}\} \times \{\mathbf{strict}, \mathbf{loose}\}$, we generated 30 random instances of each size n (number of blocks), using the random BW states generator provided by Slaney and Thiébaux (2001). For **one tower** tasks of a given size,

[10] Moreover, as Hoffmann (2002) shows, BW with 4 operators is one of the more interesting benchmark domains for heuristic planners such as FF (because relaxed-plan based heuristics exhibit more complex behaviour in this domain than in many others such as, e.g., *Logistics* or *Grid*).

[11] Note that expressing the resulting goal using only `on` and `ontable` would be very awkward. Without axioms to derive the value of "`above`", one would need to distinguish all the different cases when a block can be above another one. Without the use of quantifiers, this requires exponential space for the exponentially many different cases. With quantifiers, the size of the needed formula is still in the order of (number of goal "`above`" facts) $*$ (number of blocks in the task).

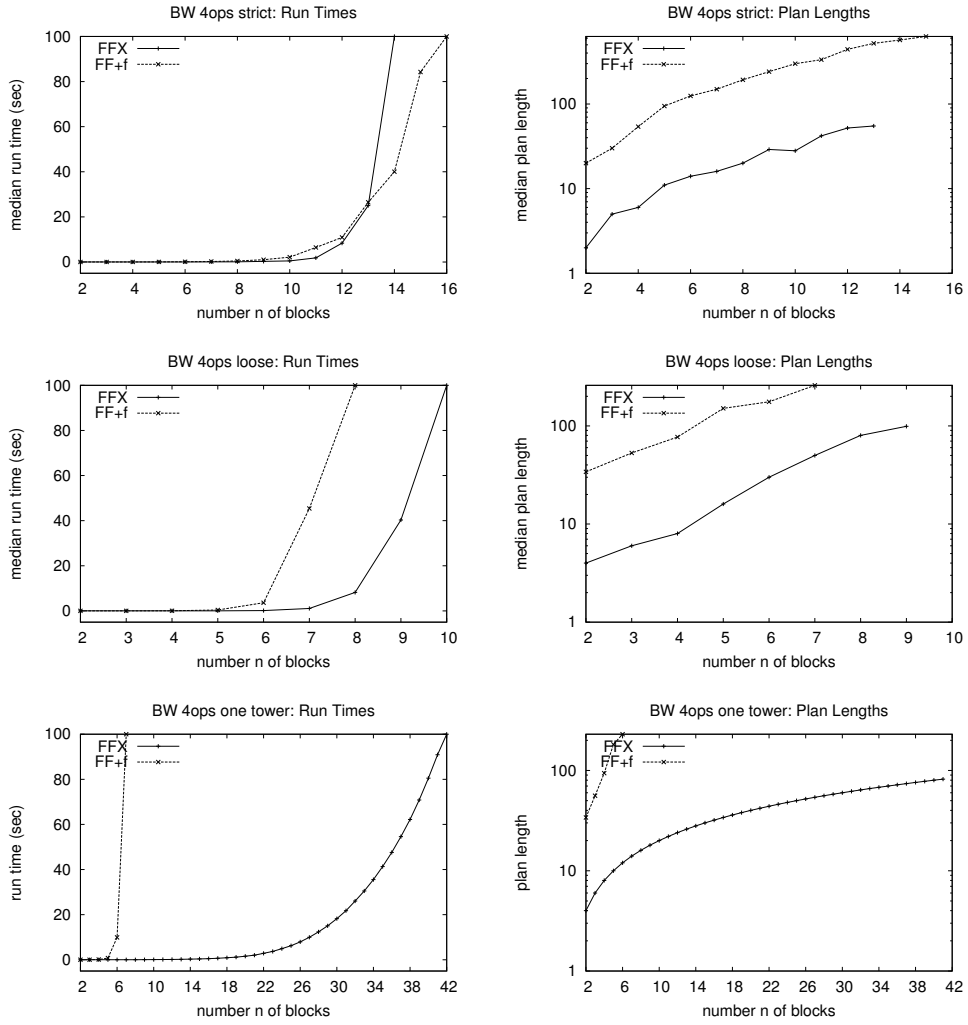
Fig. 5 Experimental results for BW with 1 operator



which are all identical up to a permutation of the blocks, a single instance suffices per value of n. Figures 5 and 6 show the median run-time and median plan length obtained by FF$_{\mathcal{X}}$ and FF+**f** and as a function of n for each of the 6 combinations. In all cases but **4ops strict**, the median run-time of FF$_{\mathcal{X}}$ shows a significant improvement over that of FF+**f**. For **one tower** tasks, the improvement is dramatic, as FF$_{\mathcal{X}}$ finds the optimal plans whose length is linear in n. With the **strict** and **loose** tasks in contrast, the plans found by FF$_{\mathcal{X}}$ are only an order of magnitude smaller than those found by FF+**f**. Note that FF's goal-ordering techniques were not used in either versions of the program. Although extending these techniques to deal with axioms is relatively straightforward, we have not invested any time yet in doing so. Goal ordering has been shown to greatly improve the performance of FF on BW, and due to the lack of it, FF's behaviour in the above experiments is significantly worse than reported in the literature (Hoffmann and Nebel, 2001).

In our BW experiments, we also considered the possibility of compiling the *non-recursive* derived predicates away as suggested by Davidson and Garagnani (2002),

Fig. 6 Experimental results for BW with 4 operators



by simply substituting their definition for them wherever they occur until no occurrence remains. We did not experiment with compiling recursive derived predicates as per their method because this requires significant implementation effort and the authors were not able to provide us with an implementation at the time of writing this paper. Instead, we considered two treatments of the recursive predicates: one using axioms and running FF$_{\mathcal{X}}$ and the other using compilation via **f** and running the original FF. In **1op** domains, the run-times obtained with the former, resp. the latter, treatment are similar to those obtained by FF$_{\mathcal{X}}$, resp. by FF+**f** in Figure 5. To be precise, the run-times for both planners are slightly larger than those in Figure 5 for **1op loose** and **1 op one tower**, and slightly lower for **1op strict**. On the other hand, in **4ops** domains, both variants (i.e. regardless of whether the recursive predicates were axiomatised or compiled away) were unable to cope with problems larger than $n = 4$. This is due to the fact that substituting for the non-recursive derived predicates easily results in operator descriptions with quite complex ADL constructs. These make FF's pre-processing infeasible, as it compiles the ADL con-

structs away following Gazen & Knoblock [2001] (instantiating the operators, and expanding all quantifiers in the formulae), and needs to create and simplify thousands of first-order formulae even in comparatively small planning tasks. In the **1op** case, preconditions are kept manageable because `clear` is the only derived predicate and its definition in terms of `on` is relatively simple, while the **4ops** case suffices to make preconditions too challenging.

We did not experiment with the other published compilation schemes (Gazen and Knoblock, 1997; Garagnani, 2000), as they are not applicable to the domains we considered whose descriptions involve negated derived predicates.

5.2 Power Supply Restoration

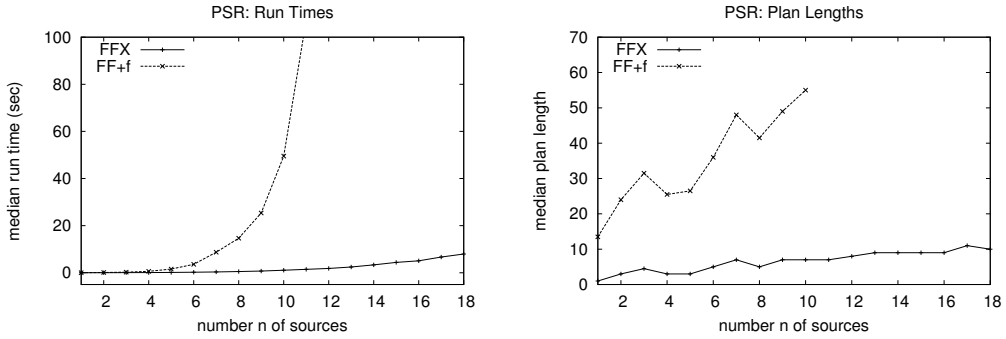
Another domain we ran experiments on is the challenging Power Supply Restoration (PSR) benchmark (Thiébaux and Cordier, 2001), which is derived from a real-world problem in the area of power distribution. This problem deals with reconfiguring a faulty power distribution system to resupply customers affected by the faults. A power distribution system is viewed as a network of electric lines connected by switches and fed via a number of power sources. When a power source feeds a faulty line, the circuit-breaker fitted to this source opens to protect the rest of the network from overloads. This leaves all the lines fed by the source without power. The problem consists in planning a sequence of switching operations (opening or closing switches and circuit-breakers) bringing the network into a configuration where non-faulty lines are resupplied. The domain description requires a number of complex, recursive, derived predicates to axiomatise the power flow, see e.g. (Bonet and Thiébaux, 2003). We considered the version of the benchmark featured in the 4th International Planning Competition, IPC-4, in which the locations of the faults and the current network configuration are completely known, and the goal is to resupply a given set of lines. That version of the problem is somewhat easier to solve than the version used in the experiments reported in Thiébaux et al. (2003), in which the planner must additionally infer which lines are resuppliable using extra derived predicates.

We used John Slaney's RANDOMNET program [12] to generate, for each number $n = 1$ to 18 sources, 30 random networks with with 30% faulty lines and a maximum of 3 switches initially fed by each given source. We also considered the networks of increasing difficulty described in (Bertoli et al., 2002; Bonet and Thiébaux, 2003): **basic**, **small-rural**, **simple**, **random**, **simplified-rural**, and **rural**. The upper part of Figure 7 compares the median run times and plan length for FF$_{\mathcal{X}}$ and FF+**f** as a function of n on the random instances, while the lower part reports run times and plan length on the known instances. Again the improvement in performance resulting from handling axioms explicitly is undeniable. In contrast to BW, the plan

[12] `http://rsise.anu.edu.au/~jks/software/randomnet.tar.gz`

Fig. 7 Experimental results for PSR

random instances



known instances

instance				run-time (sec)		plan length	
network	feeders	lines	faults	FF$_{\mathcal{X}}$	FF+**f**	FF$_{\mathcal{X}}$	FF+**f**
basic	2	5	1	0.02	0.08	5	36
small-rural	3	7	2	0.06	0.41	6	44
simple	3	7	2	0.07	0.53	7	50
random	3	9	3	0.20	1.65	7	61
simplified-rural	7	11	2	0.68	7.81	6	44
rural	7	26	2	9.67	469.11	6	76

length in PSR increases only slowly with n: with our parameters for the random instances generation, it is around 5-10 actions for PDDL$_{\mathcal{X}}$ instances, and around 30-60 for the compiled instances (the known instances exhibit similar figures). Yet this makes all the difference between what is solvable in reasonable time and what is not.

As was said earlier, PSR was also used as a benchmark in the IPC-4. The domain versions and instance generation used were the same as we use here, i.e., IPC-4 featured the version with derived predicates and the version where they are compiled away using **f**. Similarly to what we observed above for FF$_{\mathcal{X}}$ and FF+**f**, the competing systems showed clearly better behaviour in the domain version using explicit axioms, than the domain version where the axioms were compiled away via **f**. Several planners were able to solve the entire suite of scaling instances using explicit axioms. For just one of these planners, the designers chose to also run the planner on the compiled instances, with the result that only some of the smallest instances could be solved. The only other planner that was run on the compiled instances

could solve instances up to about half of the largest size. [13]

Another interesting lesson we learnt when trying to design a STRIPS version of PSR for the competition is that compiling both derived predicates and ADL constructs away is impractical. All of the 20 or so combinations of compilation schemes we tried either led to domain descriptions of unmanageable size which could not reasonably be stored on disk, or to plans of unmanageable length which no existing domain-independent planner could generate. The problem is that compiling derived predicates away generates complex conditional effects for which there is no compilation scheme preserving plan length linearly Nebel (2000). Compilations into "simple-ADL", i.e., STRIPS + conditional effects (used by Fahiem Bacchus in IPC-2), on the other hand, turned out to be feasible.

5.3 Promela

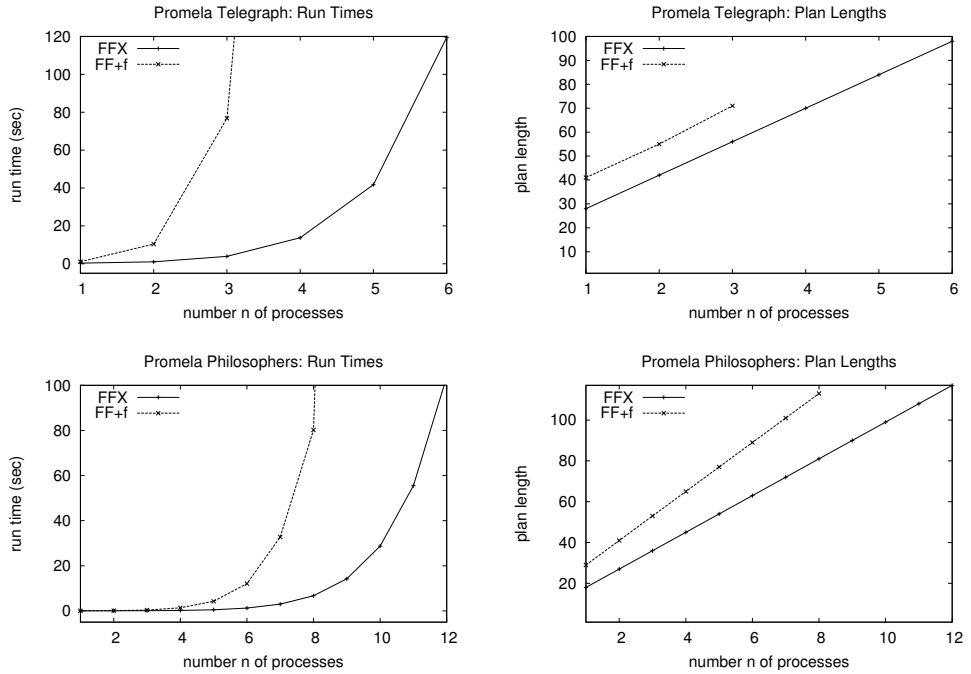
The last domain we experimented with is Stefan Edelkamp's PROMELA domain Edelkamp (2003). PROMELA, which is the input language of the model checker SPIN Holzmann (1997), is designed to ease the specification of asynchronous communication protocols which SPIN checks for errors. Given a suitable PDDL description of PROMELA and the automatic translation of a PROMELA description into a planning task, the planner must generate an error trail similar to the counter example SPIN would return in case of error.

The domain versions we considered are the IPC-4 domain featuring (non-recursive) derived predicates and no fluents, and its compilation via **f**. We experimented with the two types of PROMELA planning tasks used in the competition: dining **philosophers** protocol tasks, and optical **telegraph** protocol tasks. The instances used were the same as in the competition: for each number n of processes, we solved the (single) corresponding competition instance and recorded the time and plan length for FF and FF$_{\mathcal{X}}$.

The results are shown in Figure 8. They illustrate the best case for the compilation scheme, namely a small linear increase in plan length – recall that none of the derived predicates in PROMELA are recursive. The run-time figures show that, here, even this best case materialises into an important increase in run-time. In the IPC-4, derived predicates were not compiled away using **f**. Instead, changes to derived predicates were incorporated manually and quite cleverly into the effects of normal actions, similarly as we normally do with `clear`, `holding` and `handempty` in blocks world. Yet, the competing planners still showed much better performance with the derived predicates version. In **philosophers**, several planners were able to

[13] The competing teams were allowed to choose which domain versions they wanted to run their planner on. So the lack of participation in the compiled suite strongly indicates that nobody was able to obtain good results there.

Fig. 8 Experimental results for PROMELA



solve the entire example suite of that domain version, whereas in the version with compiled derived predicates the best planner (one of the planners that solved the entire suite with explicit axioms) only scaled up to middle-sized instances. In **telegraph** the observations aren't that easy to interpret, but still the only planner that could solve a large fraction of the example instances competed in the suite with explicit axioms.

Although the domains in our experiments/in the IPC-4 were by no means chosen to show off the worst-case for our compilation scheme, they nevertheless illustrate its drawbacks. The difference of performance we observe for FF is due to the facts that compilation increases the branching factor, increases the plan length, and degrades the informativity of the heuristic function.

6 Conclusion

As reflected by recent endeavours in the international planning competitions, there is a growing (and, in our opinion, desirable) trend towards more realistic planning languages and benchmark domains. In that context, it is crucial to determine which additional language features are particularly relevant. The main contribution of this paper is to give theoretical and empirical evidence of the fact that axioms *are* important, from both an expressivity and efficiency perspective. In addition, we have provided a clear formal semantics for PDDL axioms, identified a general and easily testable criterion for axiom sets to have an unambiguous meaning, and given a com-

pilation scheme which is more generally applicable than those previously published (and also more effective in conjunction with forward heuristic search planners like FF).

Axioms have long been an integral part of action formalisms in the field of reasoning about action and change where, much beyond the inference of derived predicates considered here, they form the basis for elegant solutions to the frame and ramification problems, see e.g. (McCain and Turner, 1995). It is our hope that the adoption of PDDL axioms will eventually encourage the planning community to make greater use of these formalisms.

Acknowledgements

Thanks to Blai Bonet, Marina Davidson, Stefan Edelkamp, Maria Fox, John Lloyd, and especially John Slaney for feedback which helped to improve and correct this paper. Blai Bonet and John Slaney also contributed to the PDDL encoding of PSR used in our experiments and the competition. This paper is an extended and revised version of a shorter paper (Thiébaux et al., 2003) published by the same authors at IJCAI'03.

The first author would like to acknowledge National ICT Australia (NICTA) and the Australian Research Council (ARC) for their support. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the ARC.

The second and third author would like to acknowledge DFG for its support. The second author had been supported by funds from DFG as part of the project *HEU-PLAN II* (Ne 623/4-2).

References

Abiteboul, S., Hall, R., Vianou, V., 1995. Foundations of Databases. Addison Wesley, Reading, MA.

Aiello, L. C., Doyle, J., Shapiro, S. (Eds.), 1996. Principles of Knowledge Representation and Reasoning: Proceedings of the 5th International Conference (KR-96). Morgan Kaufmann, Cambridge, MA.

Apt, K., Blair, H., Walker, A., 1988. Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, pp. 89–148.

Bacchus, F., 2000. Subset of PDDL for the AIPS2000 Planning Competition. The AIPS-00 Planning Competition Committee.

Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Sun, Y., Weld, D., 1995. UCPOP user's manual. Tech. Rep. 93-09-06d, The University of Washington, Computer Science Department.

Bertoli, P., Cimatti, A., Slaney, J., Thiébaux, S., 2002. Solving power supply restoration problems with planning via symbolic model checking. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02). Wiley, Lyon, France, pp. 576–80.

Bonet, B., Geffner, H., 2001. GPT: a tool for planning with uncertainty and partial information. In: Proceedings IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. pp. 82–87.

Bonet, B., Thiébaux, S., 2003. GPT meets PSR. In: Giunchiglia, E., Muscettola, N., Nau, D. (Eds.), Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-03). Trento, Italy, pp. 102–111.

Brewka, G., Hertzberg, J., 1993. How to do things with worlds: On formalizing actions and plans. Journal Logic and Computation 3 (5), 517–532.

Cadoli, M., Donini, F. M., 1997. A survey on knowledge compilation. AI Communications 10 (3,4), 137–150.

Cadoli, M., Donini, F. M., Liberatore, P., Schaerf, M., 1996. Comparing space efficiency of propositional knowledge representation formalism. In: Aiello et al. (1996), pp. 364–373.

Chandra, A., Kozen, D., Stockmeyer, L., 1981. Alternation. Journal of the Association for Computing Machinery 28 (1), 114–133.

Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A., 2001. Complexity and expressive power of logic programming. ACM Computing Surveys 33 (3), 374–425.

Davidson, M., Garagnani, M., 2002. Pre-processing planning domains containing language axioms. In: Proceedings of the 21st Workshop of the UK Planning and Scheduling SIG (PlanSIG-02). pp. 23–34.

Edelkamp, S., 2003. Promela planning. In: Model Checking Software, 10th International SPIN Workshop. Springer-Verlag, Berlin, pp. 197–212.

Edelkamp, S., Hoffmann, J., 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. Rep. 195, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany.

Erol, K., Nau, D. S., Subrahmanian, V. S., 1995. Complexity, decidability and undecidability results for domain-independent planning. Artificial Intelligence 76 (1–2), 75–88.

Fox, M., Long, D., 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research 20, 61–124.

Garagnani, M., 2000. A correct algorithm for efficient planning with preprocessed domain axioms. In: Research and Development in Intelligent Systems XVII. Springer-Verlag, Berlin, pp. 363–374.

Gazen, B. C., Knoblock, C., 1997. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In: Steel, S., Alami, R. (Eds.), Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97). Vol. 1348 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Toulouse, France, pp. 221–233.

Gogic, G., Kautz, H. A., Papadimitriou, C. H., Selman, B., 1995. The comparative linguistics of knowledge representation. In: IJCAI-95 (1995), pp. 862–869.

Gustafsson, J., Doherty, P., 1996. Embracing occlusion in specifying the indirect effects of actions. In: Aiello et al. (1996), pp. 87–98.

Hoffmann, J., 2002. Local search topology in planning benchmarks: A theoretical analysis. In: Ghallab, M., Hertzberg, J., Traverso, P. (Eds.), Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02). Morgan Kaufmann, Toulouse, France, pp. 92–100.

Hoffmann, J., Nebel, B., 2001. The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302.

Holzmann, G., 1997. The model checker Spin. IEEE Transactions on Software Engineering 23 (5), 279–295.

IJCAI-95, 1995. Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95). Montreal, Canada.

Karp, R. M., Lipton, R. J., 1982. Turing machines that take advice. L' Ensignement Mathématique 28, 191–210.

Kautz, H. A., Selman, B., 1992. Forming concepts for fast inference. In: Proceedings of the 10th National Conference of the American Association for Artificial Intelligence (AAAI-92). MIT Press, San Jose, CA, pp. 786–793.

Lifschitz, V., 1986. On the semantics of STRIPS. In: Georgeff, M. P., Lansky, A. (Eds.), Reasoning about Actions and Plans: Proceedings of the 1986 Workshop. Morgan Kaufmann, Timberline, OR, pp. 1–9.

Lin, F., 1995. Embracing causality in specifying the indirect effects of actions. In: IJCAI-95 (1995), pp. 1985–1993.

Lloyd, J. W., 1984. Foundations of Logic Programming. Springer-Verlag, Berlin, Heidelberg, New York.

Lynch, N. A., 1977. Log space recognition and translation of parenthesis languages. Journal of the Association for Computing Machinery 24, 583–590.

McCain, N., Turner, H., 1995. A causal theory of ramifications and qualifications. In: IJCAI-95 (1995), pp. 1978–1984.

McDermott, D., 1998. PDDL – the planning domain definition language. Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nau, D., Cao, Y., Lotem, A., Munoz-Avila, H., 1999. SHOP: simple hierarchical ordered planner. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99). Stockholm, Sweden, pp. 968–975.

Nebel, B., 2000. On the compilability and expressive power of propositional planning formalisms. Journal of Artificial Intelligence Research 12, 271–315.

Sandewall, E., 1994. Features and Fluents. Oxford University Press, Oxford, UK.

Slaney, J., Thiébaux, S., 2001. Blocks world revisited. Artificial Intelligence 125, 119–153.

Thiébaux, S., Cordier, M., Sep. 2001. Supply restoration in power distribution systems — a benchmark for planning under uncertainty. In: Cesta, A., Borrajo, D. (Eds.), Recent Advances in AI Planning. 6th European Conference on Planning (ECP'01). Springer-Verlag, Toledo, Spain, pp. 85–95.

Thiébaux, S., Hoffmann, J., Nebel, B., 2003. In defence of PDDL axioms. In: Gottlob, G. (Ed.), Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03). Acapulco, Mexico, pp. 961–966.

Vardi, M. Y., 1982. The complexity of relational query languages. In: Proceedings of the 14th ACM Symposium on the Theory of Computation. pp. 137–146.

Vorobyov, S., Voronkov, A., 1997. Complexity of nonrecursive programs with complex values. Tech. Rep. MPI-I-97-2-010, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Winslett, M. S., 1988. Reasoning about action using a possible models approach. In: Proceedings of the 7th National Conference of the American Association for Artificial Intelligence (AAAI-88). Saint Paul, MI, pp. 89–93.