

## Technical Report TR-ARP-01-03

Automated Reasoning Group  
Computer Sciences Laboratory  
Research School of Information Sciences and Engineering  
The Australian National University

February 2003

## IN DEFENSE OF PDDL AXIOMS

Sylvie Thiébaux  
Computer Sciences Laboratory  
The Australian National University  
Canberra, ACT 0200, Australia  
Sylvie.Thieboux@anu.edu.au

Jörg Hoffmann & Bernhard Nebel  
Institut für Informatik  
Universität Freiburg  
D-79110 Freiburg, Germany  
<last>@informatik.uni-freiburg.de

**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 they are a non-essential feature which is best compiled 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.

# 1 Motivation

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 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 in this family 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], 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 used in competition events, and did not survive PDDL2.1, the extension of the language to temporal planning [Fox and Long, 2002].

This is unfortunate, as 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—that is when we are lucky and extra actions do not need to be introduced or action descriptions customised 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. For instance, a serious benchmark contender for the 2004 competition is a deterministic version of the power supply restoration problem 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 and 2004 International Planning Competition objected that<sup>1</sup> 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 in [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 is better compiled away than explicitly dealt-with, 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 concisely 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—provided we require the domain descriptions to

---

<sup>1</sup>Personal communications

grow only polynomially and the plans to grow only polynomially in the size of the original plans and domain descriptions. Of course, if we allow for exponential growth, then compilations become possible and we specify one such transformation, which, unlike those previously published [Gazen and Knoblock, 1997; Garagnani, 2000; Davidson and Garagnani, 2002], works without restriction. However, the above mentioned 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 solving the compiled problem.

## 2 Syntax and Semantics

This paper remains in the sequential planning setting. We therefore start from the syntax of PDDL2.1 level 1, which is essentially that of the version of PDDL with ADL actions used in the 2000 planning competition [Bacchus, 2000]. For clarity, we omit types. Although we see axioms with conditions on numeric fluents, such as those featured in PDDL2.1 level 2, as very desirable, we do not consider them here for simplicity.

The syntax of PDDL with axioms is given in Figure 1. `<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 `(: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 `(: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 sum, axioms are essentially (function free) logic program statements [Lloyd, 1993]. 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)
  (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)))))
```

---

**Figure 1** Syntax of PDDL with axioms

---

```
<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>)
```

---

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.*, 1987]. 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 but *without* using negation. Next some 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<sup>2</sup> (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 (allowing for recursion). There is no restriction on the use of basic predicates.

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.*, 1987, p. 116]. It is this final fixed point which we take to be the meaning of the axiom set. Note 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 interesting special case [Gazen and Knoblock, 1997; Garagnani, 2000; Davidson and Garagnani, 2002].

We now spell out the semantics formally.

**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 axiom  $(: \text{derived } (d_i ?\vec{x}) (f ?\vec{x})) \in A$ :*

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

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:  $A_i = \{(: \text{derived } (d_i ?\vec{x}) (f_i ?\vec{x})) \in A \mid d_i \in \mathcal{D}_i\}$ .

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. 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

---

<sup>2</sup>In a formula in NNF, negation occurs only in literals.

$D$  of atoms in the derived vocabulary. The modeling conditions for extended states 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**

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

Then, applying axiom  $a \equiv (: \text{derived } (d ?\vec{x}) (f ?\vec{x}))$  in a state  $S$  augmented with derived atoms  $D$ , results in the set  $\llbracket a \rrbracket(S, D)$  of further derived atoms:

**Definition 3**  $\llbracket a \rrbracket(S, D) = \{(d \vec{t}) \mid \langle S, D \rangle \models (f \vec{t}), \vec{t} \text{ is ground}\}$

Given this, we associate stratum  $A_i$  with the function  $\llbracket A \rrbracket_i$  which maps a given basic state  $S$  to 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  $\llbracket A \rrbracket_{i-1}$ . The stratified axiom set  $A$  denotes the function  $\llbracket A \rrbracket = \llbracket A \rrbracket_n$ :

**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:

$$\begin{aligned} \llbracket A \rrbracket_0(S) &= \emptyset, \text{ and for all } 1 \leq i \leq n \\ \llbracket A \rrbracket_i(S) &= \bigcap \left\{ D \mid \bigcup_{a \in A_i} \llbracket a \rrbracket(S, D) \cup \llbracket A \rrbracket_{i-1}(S) \subseteq D \right\} \end{aligned}$$

Then  $\llbracket A \rrbracket(S)$  is defined as  $\llbracket A \rrbracket_n(S)$ .

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**  $S \models_A f \text{ iff } \langle S, \llbracket A \rrbracket(S) \rangle \models f$

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].

Checking that the axiom set in a given domain description is stratified and computing a stratification can be done in polynomial time in the size of the domain description, using for instance Algorithm 1. The computation is reminiscent of that of the transitive closure of a relation.

---

**Algorithm 1** Stratification

---

```
1. function STRATIFY( $\mathcal{D}, A$ )
2.    $R \leftarrow \text{ORDER}(\mathcal{D}, A)$ 
3.   if  $\forall i \in \mathcal{D} R[i, i] \neq 2$  then
4.     return EXTRACT( $\mathcal{D}, R$ )
5.   else fail

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

22. function EXTRACT( $\mathcal{D}, R$ )
23.   $\text{stratification} \leftarrow \emptyset$ ,  $\text{remaining} \leftarrow \mathcal{D}$ ,  $\text{level} \leftarrow 1$ 
24.  while  $\text{remaining} \neq \emptyset$  do
25.     $\text{stratum} \leftarrow \emptyset$ 
26.    for each  $j \in \text{remaining}$  do
27.      if  $\forall i \in \text{remaining} R[i, j] \neq 2$  then
28.         $\text{stratum} \leftarrow \text{stratum} \cup \{j\}$ 
29.     $\text{remaining} \leftarrow \text{remaining} \setminus \text{stratum}$ 
30.     $\text{stratification} \leftarrow \text{stratification} \cup \{(\text{level}, \text{stratum})\}$ 
31.     $\text{level} \leftarrow \text{level} + 1$ 
32.  return  $\text{stratification}$ 
```

---

The algorithm starts by calling the function ORDER which analyses the axioms to build a<sup>3</sup>  $|\mathcal{D}| \times |\mathcal{D}|$  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 initialized 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

---

<sup>3</sup>By  $|\cdot|$  we denote the cardinality of a set.

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

In the following, we take a  $\text{PDDL}_{\mathcal{X}}$  planning domain description to be a tuple  $\Delta = \langle \mathcal{C}, \mathcal{B}, \mathcal{D}, A, O \rangle$ , where  $\mathcal{C}$  is the set of constant symbols,  $\mathcal{B}$  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 on the appearance in atomic effects of the symbols in  $\mathcal{D}$ ). A  $\text{PDDL}_{\mathcal{X}}$  *planning instance* or *task* is a tuple  $\Pi = \langle \Delta, \mathcal{I}, \mathcal{G} \rangle$ , where  $\Delta$  is the domain description, 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 modeling relation in 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  $\langle \mathcal{C}, \mathcal{B}, \emptyset, \emptyset, O \rangle$ .

We now use *compilation schemes* [Nebel, 2000] to translate  $\text{PDDL}_{\mathcal{X}}$  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.<sup>4</sup>

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

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

and satisfies the following conditions:

---

<sup>4</sup>We use here a slightly simplified definition of compilability.

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

In addition, we measure the size of the corresponding plans in the target formalism.<sup>5</sup>

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

From a practical point of view, one can regard compilability preserving *plan size linearly* as an indication that the planning formalism we use as the target formalism is *at least as expressive* as the source formalism. Conversely, if a *super-linear* blowup of the plans in the target formalism is required, this is 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 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 analyze 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.

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

**Proof.** EXPTIME-hardness follows from EXPTIME-completeness of DATALOG entailment [Dantsin *et al.*, 2001]. 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 can be done in EXPTIME [Dantsin *et al.*, 2001]. ■

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.

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

**Proof.** 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

<sup>5</sup>The size of an instance, domain description, plan, etc. is denoted by  $\|\cdot\|$ .

clearly takes only polynomial space. So the entire verification can be carried out in polynomial space. ■

From these two statements 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. As argued, however, 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 changes nothing.

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 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*,  $\Sigma$  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 a 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 that never leaves 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 *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  $X/\text{poly}$  (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  $\text{EXPTIME} = \text{PSPACE}$ , there is no compilation scheme from  $\text{PDDL}_{\mathcal{X}}$  (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  $\text{PDDL}_{\mathcal{X}}$  planning domains

$\Delta_n$  that can be used to decide the LBATM problem up to size  $n$  by solving the 1-step planning problem for  $\langle \Delta_n, \mathcal{I}_I, g \rangle$ , where  $\mathcal{I}_I$  can be computed from  $I$  in polynomial time, and  $g$  is some constant predicate.

As constant symbols we use the set  $\mathcal{C}_n = \{0, 1, \dots, n, \#, U, E, A, L, R, S\}$ , where the numbers stand for tape cells, states, or elements of the input alphabet,  $\#$  stands for the blank symbol, the symbols  $U, E, A$  are used to denote, universal, existential, and accepting states, respectively, and  $L, R, S$  are used to denote head movement, i.e.,  $L$  for left,  $R$  for right, and  $S$  for stationary. The *basic predicates* are:

- $\text{cell}(i, s)$  describing that the  $i$ th tape cell of the input contains  $s$ , with  $i = 1$  denoting the leftmost cell,
- $\text{type}(q, t)$  describing the type  $t$  of state  $q$ , with  $q = 1$  denoting the initial state,
- $\text{trans}(q, s, q', s', m)$  describing one entry of the transition table corresponding to  $\delta(q, s) \ni \langle q', s', m \rangle$ ,
- $h$  is an auxiliary atom, which is always false in the initial state.

In addition, we use the following derived predicates:

- $\text{acc}(x_1, \dots, x_n, q, h)$  describing a configuration with tape contents  $x_1, \dots, x_n$ , state  $q$ , and head position  $h$ ,
- $g$  is the goal atom which is added by the only operator in our domain description.

Now we have for every state  $q$  the following rule accounting for accepting states:

```
(:derived (acc ?x1 ... ?xn q ?h)
  (type q A))
```

In addition, we have for every tuple  $\langle q, s, q', s', i \rangle$  the following rule to account for existential states:

```
(:derived (acc ?x1...?xi-1 s ?xi+1...?xn q i)
  (and
    (type q E)
    (trans q s q' s' L)
    (acc ?x1...?xi-1 s' ?xi+1...?xn q' i-1)))
```

There are similar rules for the the movements  $R$  and  $S$ . Furthermore, we have rules describing the semantics of universal states. For every tuple  $\langle q, s, i \rangle$  we have a rule as follows:

```
(:derived (acc ?x1...?xi-1 s ?xi+1...?xn q i)
  (and
    (type q U)
    (forall ?q' ?s'
      (and
        (or (not (trans q s ?q' ?s' L))
          (acc ?x1...?xi-1 ?s' ?xi+1...?xn
```

```

        ?q' i-1))
    (or (not (trans q s ?q' ?s' R))
        (acc ?x1...?xi-1 ?s' ?xi+1...?xn
            ?q' i+1))
    (or (not (trans q s ?q' ?s' S))
        (acc ?x1...?xi-1 ?s' ?xi+1...?xn
            ?q' i))))))

```

Finally, we have a rule describing how to generate the ultimate goal:

```

(:derived h
  (and
    (cell 1 ?x1)
    (cell 2 ?x2)
    ...
    (cell n xn)
    (acc x1 ... xn 1 0)))

```

Now the only operator we have is the following:

```

(:action a
  :parameters ()
  :precondition (h)
  :effect (g))

```

Let now  $I = \langle w, M \rangle$  be an LBATM instance of size  $n$ . Let  $\mathcal{I}_I$  be the state describing  $M$  and  $w$  using the derived predicates. It is then clear that the constructed  $\text{PDDL}_{\mathcal{X}}$  instance  $\Pi_n = \langle \Delta_n, \mathcal{I}_I, g \rangle$  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  $\text{PDDL}_{\mathcal{X}}$  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  $\Delta_n$  to a PDDL domain structure  $\Delta'_n$  can be used as the polynomial advice. The advice-taking Turing machine reads the instance, loads the advice  $\Delta'_n$ , 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  $\text{EXPTIME} = \text{PSPACE}$ . ■

## 4 Compilations with Exponential Results

While it is impossible to find a concise 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 each axiom into an operator having the axiom's antecedent as precondition and its consequent as effect. It also augments those of the original operators that affect the antecedent of an axiom with conditional effects deleting all ground instances of this axiom's consequent. This scheme gives the possibility to the planner

of inferring positive derived literals needed in a plan, but 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.<sup>6</sup> The latter scheme [Garagnani, 2000] is further restricted to pure DATALOG axioms, and although it more clever in keeping track of the ground derived atoms in need to be deleted, it still suffers from the same problems in the presence of negated derived predicates.

An interesting contrasting approach is that of Davidson and Garagnani [2002]. They propose to compile pure DATALOG axioms solely into conditional effects, which means that the resulting plans will have exactly the same length. However, as is implied by Theorem 3, their domain description suffers a super-polynomial growth.

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.

**Theorem 4** *There exists a polynomial time compilation scheme  $\mathbf{f} = \langle f_\delta, f_i, f_g \rangle$ , such that for every PDDL $_{\mathcal{X}}$  domain description  $\Delta = \langle \mathcal{C}, \mathcal{B}, \mathcal{D}, A, O \rangle$ :*

- $\|f_i(\Delta)\| = c_1$  for some constant  $c_1$ ,
- $\|f_g(\Delta)\| = c_2$  for some constant  $c_2$ ,
- $f_\delta(\Delta) = \langle \mathcal{C}, \mathcal{B}', \emptyset, \emptyset, O' \rangle$  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 main elements of the PDDL instances induced by  $\mathbf{f}$ .  $\mathbf{f}$  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  $(: \text{derived} (d_{i,j} \text{ ?}\vec{x}_{i,j}) (f_{i,j} \text{ ?}\vec{x}_{i,j}))$  for  $1 \leq j \leq |A_i|$ .  $\mathbf{f}$  encodes each stratum as an extra action  $\text{stratum}_i$  (see lines 5-15 in Figure 4) which applies all axioms  $a_{i,j}$  at this stratum in parallel, records that this was done ( $\text{done}_i$ ) and whether anything new ( $\text{new}$ ) was derived in doing so. Each  $a_{i,j}$  is encoded as a universally quantified and conditional effect of  $\text{stratum}_i$ —see lines 9-15. To ensure that the precedence between strata is respected,  $\text{stratum}_i$  is only applicable when the fixed point for the previous stratum has been reached (i.e. when  $\text{fixed}_{i-1}$ ) and the fixed point for the current stratum has not (i.e. when  $(\text{not} (\text{fixed}_i))$ )—see line 7.  $\mathbf{f}$  encodes the fixpoint computation at each stratum  $i$  using an extra action  $\text{fixpoint}_i$ , which is applicable after a round of one or more applications of  $\text{stratum}_i$  (i.e., when  $\text{done}_i$  is true), asserts that the fixed point has been reached (i.e.  $\text{fixed}_i$ ) whenever nothing new has been derived during this last round, and resets  $\text{new}$  and  $\text{done}_i$  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  $\text{fixed}_k$  to the precondition. Similarly, let

<sup>6</sup>A counter example is  $\mathcal{C} = \emptyset$ ,  $\mathcal{B} = \{a\}$ ,  $\mathcal{D} = \{b\}$ ,  $O = \{(: \text{action op} : \text{parameters} () : \text{effect} (a))\}$ ,  $A = \{(: \text{derived}(b)(a))\}$ ,  $\mathcal{I} = \emptyset$ ,  $\mathcal{G} = (\text{and } a (\text{not } b))$ . This instance is not solvable because establishing  $a$  also establishes  $b$  via the axiom, yet both schemes yield a compiled instance solvable by the plan  $(\text{op})$ . This remains true even if negation is compiled away as per the Gazen and Knoblock method [Gazen and Knoblock, 1997].

---

**Figure 2** PDDL instances induced by **f**


---

```

1. (: predicates ; all predicates in  $\mathcal{B} \cup \mathcal{D}$ 
2.     (done1) ... (donen)
3.     (fixed0) ... (fixedn)
4.     (new))
   for each  $i \in \{1, \dots, n\}$ 
5. (: action stratumi
6.   : parameters ()
7.   : precondition (and (fixedi-1) (not (fixedi)))
8.   : effect (and (donei)
9.             (forall ( $\vec{x}_{i,1}$ )
10.                (when (and (fi,1 ? $\vec{x}_{i,1}$ ) (not (di,1 ? $\vec{x}_{i,1}$ )))
11.                    (and (di,1 ? $\vec{x}_{i,1}$ ) (new))))))
12.             ...
13.             (forall ( $\vec{x}_{i,|A_i|}$ )
14.                (when (and (fi,|A_i| ? $\vec{x}_{i,|A_i|}$ ) (not (di,|A_i| ? $\vec{x}_{i,|A_i|}$ )))
15.                    (and (di,|A_i| ? $\vec{x}_{i,|A_i|}$ ) (new))))))
16. (: action fixpointi
17.   : parameters ()
18.   : precondition (donei)
19.   : effect (and (when (not(new)) (fixedi)
20.                (not(new))
21.                (not (donei))))
   for each  $o \in O$ 
22. (: action NAME( $o$ )
23.   : parameters PARAMETERS( $o$ )
24.   : precondition (and PRECONDITION( $o$ ) (fixedk))
25.   : effect (and EFFECT( $o$ )
26.            (not (fixedm)) ... (not (fixedn))
27.            (not (donem)) ... (not (donen))
28.            (forall  $\vec{x}_{m,1}$  (not (dm,1 ? $\vec{x}_{m,1}$ )))
29.            ...
30.            (forall  $\vec{x}_{n,|A_n|}$  (not (dn,|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  $\mathcal{I} \cup (\text{fixed}_0)$ )
32. (: goal (and  $\mathcal{G}$  (fixedk)))
   Where  $k = \max(\{i \mid \text{some } d_{i,j} \text{ occurs in } \mathcal{G}\} \cup \{0\})$ 

```

---

$1 \leq m \leq n + 1$  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,  $\text{fixed}_0$  holds initially, and the goal requires  $\text{fixed}_k$  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—see lines 31-32.

The fact that **f** preserves domain description size polynomially, and the bounds given in theorem 4, follow directly from the construction. Let  $\Delta = \langle \mathcal{C}, \mathcal{B}, \mathcal{D}, A, O \rangle$  be a PDDL $_{\mathcal{X}}$  instance. We have  $f_i(\Delta) = (\text{fixed}_0)$  and so  $\|f_i(\Delta)\|$  is a constant.  $f_g(\Delta) = (\text{fixed}_k)$  for some  $0 \leq k \leq n$

and so  $\|f_g(\Delta)\|$  is a constant.  $f_\delta(\Delta)$  is the PDDL domain description  $\langle \mathcal{C}, \mathcal{B}', \emptyset, \emptyset, O' \rangle$ , where  $\mathcal{B}' = \mathcal{B} \cup \mathcal{D} \cup \{(\text{fixed}_i) \mid 0 \leq i \leq n\} \cup \{(\text{done}_i) \mid 1 \leq i \leq n\} \cup \{(\text{new})\}$  and  $O' = \{\text{stratum}_i \mid 1 \leq i \leq n\} \cup \{\text{fixpoint}_i \mid 1 \leq i \leq n\} \cup \{o' \mid o' \text{ is the augmentation of some } o \in O \text{ via } \mathbf{f}\}$ . 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  $\text{fixpoint}_i$  action description is bounded by a constant, that of each  $\text{stratum}_i$  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  $\|O\|$ .

Now, let  $\Pi = \langle \Delta, \mathcal{I}, \mathcal{G} \rangle$  be a PDDL $\mathcal{X}$  instance, and  $F(\Pi)$  be the PDDL instance obtained by compilation of  $\Pi$  via  $\mathbf{f}$ . We must prove that there is a plan  $P$  for  $\Pi$  iff there is a plan  $P'$  for  $F(\Pi)$ .

We start with the left to right direction. For  $1 \leq h \leq n$ , let  $q_h$  be the sequence:

$$\underbrace{\text{stratum}_h \dots \text{stratum}_h}_{|\overline{\mathcal{D}}_h|} \text{fixpoint}_h$$

where  $\overline{\mathcal{D}}_h$  denotes the set of all instances of predicates in  $\mathcal{D}_h$ , and let  $q'_h$  be the sequence:

$$q_h \text{stratum}_h \text{fixpoint}_h$$

Furthermore, for  $1 \leq m \leq n$  and  $1 \leq k \leq n$ , let  $R_{m,k}$  be the set of sequences  $r_m \dots r_k$ , where for all  $m \leq h \leq k$   $r_h \in \{q_h, q'_h\}$ . Note that if  $k < m$ , then  $R_{m,k}$  only contains the empty sequence. We are going to show that:

**Proposition 1** *Let  $P = P_1 \dots P_l$  be a plan for  $\Pi$ , and for each  $1 \leq j \leq l$ , let  $P'_j$  be the augmentation of  $P_j$  via  $\mathbf{f}$ . Then there exists  $\{m_j \mid 0 \leq j \leq l\}$ ,  $\{k_j \mid 0 \leq j \leq l\}$  and  $\{Q_{m_j, k_j} \in R_{m_j, k_j} \mid 0 \leq j \leq l\}$  such that  $P' = Q_{m_0, k_0} P'_1 Q_{m_1, k_1} P'_2 \dots Q_{m_{l-1}, k_{l-1}} P'_l Q_{m_l, k_l}$  is a plan for  $F(\Pi)$ .*

To prove this, we first need the following intermediate results.

**Proposition 2** *Let  $\Sigma$  be a state at some point of the execution of the compiled plan such that  $\Sigma = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_{h-1}(S)$  with  $S \subseteq \overline{\mathcal{B}}$ . Then, one of the two sequences in  $r_h$  is executable in  $\Sigma$  and leads to the state  $\Sigma' = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h\} \cup \llbracket A \rrbracket_h(S)$ .*

To show this, first note that  $\Sigma \models (\text{and}(\text{fixed}_{h-1}(\text{not}(\text{fixed}_h))))$ , the precondition of  $\text{stratum}_h$ , and since  $\text{stratum}_h$  does not modify any of the  $\text{fixed}_i$ , the prefix of  $q_h$  consisting of the  $|\overline{\mathcal{D}}_h|$  applications of  $\text{stratum}_h$  is executable in  $\Sigma$ . From then on, there are two cases. If no ground atom in  $\mathcal{D}_h$  is derivable from  $S$ , then  $\llbracket A \rrbracket_h(S) = \llbracket A \rrbracket_{h-1}(S)$ , and so the execution of this prefix results in the state  $\Sigma_1 = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_h(S) \cup \{(\text{done}_h)\}$ .  $\text{fixpoint}_h$  is applicable in  $\Sigma_1$  because  $(\text{done}_h)$  is true, and this application results in the desired outcome  $\Sigma' = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h\} \cup \llbracket A \rrbracket_h(S)$ . So, in the first case ( $\llbracket A \rrbracket_h(S) = \llbracket A \rrbracket_{h-1}(S)$ ),  $q_h$  is executable in  $\Sigma$  and yield the desired result. If, on the other hand,  $\llbracket A \rrbracket_h(S) \supset \llbracket A \rrbracket_{h-1}(S)$ , we show that the execution of the mentioned prefix of  $q_h$  results in the state  $\Sigma_2 = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_h(S) \cup \{(\text{done}_h), (\text{new})\}$ . That  $\Sigma_2$  is included in  $S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_h(S) \cup \{(\text{done}_h), (\text{new})\}$  is trivial: since  $\llbracket A \rrbracket_h(S)$  is a fixed point, it is closed under application of the axioms in  $A_h$ ; therefore no execution of  $\text{stratum}_h$  can lead out of it and the state reached after each iteration of  $\text{stratum}_h$  remains within  $S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_h(S) \cup$

$\{(done_h), (new)\}$ . The converse inclusion is proved by induction on the length of derivations using the axioms. That is, for any  $p \geq 0$  there is some  $p'$  such that all propositions in  $\llbracket A \rrbracket_h \setminus \llbracket A \rrbracket_{h-1}$  derivable with up to  $p$  applications of axioms in  $A_h$  are in the state obtained at the  $p'$ -th iteration of  $stratum_h$ . Each axiom application adds one more  $(d \vec{t})$ , and in every case, by the induction hypothesis, there is some iteration of  $stratum_h$  at which the effect contexts corresponding to the axiom's antecedent have been added and at which the action is therefore applicable to extend the state by adding  $(d \vec{t})$  to the state. Obviously  $|\overline{\mathcal{D}}_h|$  is an upper bound on the number of times  $stratum_h$  can extend the state, so convergence to a fixed point within  $|\overline{\mathcal{D}}_h|$  iterations is guaranteed. Finally, in the state  $S \cup \{(fixed_i) \mid 0 \leq i \leq h-1\} \cup \llbracket A \rrbracket_h(S) \cup \{(done_h), (new)\}$  the action  $fixpoint_h$  is applicable, and since  $(new)$  is true this simply removes  $(new)$  and  $(done)$ . The next application of  $stratum_h$  does not set  $(new)$  again because the state is already saturated under the axioms in  $A_h$ , so one more application of  $fixpoint_h$  adds  $(fixed_h)$  as required. That is, to sum up,  $q'_h$  is executable in  $\Sigma$  and leads to the desired state  $\Sigma'$ .

**Proposition 3** *Let  $\Sigma$  be a state at some point of the execution of the compiled plan such that  $\Sigma = S \cup \{(fixed_i) \mid 0 \leq i \leq m-1\} \cup \llbracket A \rrbracket_{m-1}(S)$  with  $S \subseteq \overline{B}$ . Then, one of the sequences  $Q_{m,k} \in R_{m,k}$  is executable in  $\Sigma$  and leads to a state  $\Sigma' = S \cup \{(fixed_i) \mid 0 \leq i \leq \sup(m-1, k)\} \cup \llbracket A \rrbracket_{\sup(m-1, k)}(S)$ .*

If  $k < m$  then  $R_{m,k}$  only contain the empty sequence, which makes the proposition true. The case  $k \geq m$  is easily proved by induction on the length of  $Q$  (i.e., the number of  $r$  subsequences in it) using  $k = m$  for the base case and Proposition 2 for the induction step.

**Proposition 4** *Let  $\Sigma$  be a state at some point of the execution of the compiled plan such that  $\Sigma = S \cup \{(fixed_i) \mid 0 \leq i \leq m-1\} \cup \llbracket A \rrbracket_{m-1}(S)$  with  $S \subseteq \overline{B}$ . Let  $P_j \in O$  be an action applicable in  $S$ , let  $P'_j \in O'$  be the augmentation of  $P_j$  via  $\mathbf{f}$ , let  $k$  be the highest stratum of any derived predicate appearing in the precondition of  $P_j$  (or 0 if there is none), and let  $m_j$  be the lowest stratum such that some predicate in the antecedent of an axiom in  $A_{m_j}$  is modified in the effect of  $P_j$  (or  $n+1$  if there is none). Then, there exists a sequence  $Q_{m,k} \in R_{m,k}$  such that  $Q_{m,k}P'_j$  is executable in  $\Sigma$  and leads to a state  $\Sigma' = S' \cup \{(fixed_i) \mid 0 \leq i \leq \inf(m_j-1, \sup(m-1, k))\} \cup \llbracket A \rrbracket_{\inf(m_j-1, \sup(m-1, k))}(S')$ , with  $S' \subseteq \overline{B}$ .*

To show this, note that in virtue of Proposition 3, there exists a  $Q_{m,k} \in R_{m,k}$  which is executable in  $\Sigma = S \cup \{(fixed_i) \mid 0 \leq i \leq m-1\} \cup \llbracket A \rrbracket_{m-1}(S)$ , and results in the state  $\Sigma'' = S \cup \{(fixed_i) \mid 0 \leq i \leq \sup(m-1, k)\} \cup \llbracket A \rrbracket_{\sup(m-1, k)}(S)$ . Now consider the precondition  $f$  of  $P_j$ . Because  $P_j$  is applicable in  $S$ ,  $S \models_A f$ , and then, because  $k$  is the highest stratum of any derived predicate in  $f$ , the precondition  $(\text{and } f \text{ (fixed}_k))$  of  $P'_j$  is verified in  $\Sigma''$ , i.e.  $\Sigma'' \models (\text{and } f \text{ (fixed}_k))$ . So  $P'_j$  is applicable in  $\Sigma''$  and from the fact that the effects of  $P'_j$  only affect  $S$  and delete the  $fixed_i$  and atoms in  $\mathcal{D}_i$  for  $m_j \leq i \leq n$ , this application results in a state  $\Sigma' = S' \cup \{(fixed_i) \mid 0 \leq i \leq \inf(m_j-1, \sup(m-1, k))\} \cup \llbracket A \rrbracket_{\inf(m_j-1, \sup(m-1, k))}(S)$ , with  $S' \subseteq \overline{B}$ . Because no antecedent of any axiom in  $A_i$  for  $0 \leq i \leq m_j$  is changed by  $P'_j$ ,  $\llbracket A \rrbracket_{\inf(m_j-1, \sup(m-1, k))}(S) = \llbracket A \rrbracket_{\inf(m_j-1, \sup(m-1, k))}(S')$  and the proposition follows.

Now, we come back to proving Proposition 1. From Proposition 4 and the fact that the state at the start of the compiled plan is  $\Sigma_0 = \mathcal{I} \cup fixed_0 = \mathcal{I} \cup fixed_0 \cup \llbracket A \rrbracket_0(\mathcal{I})$ , it follows by induction that there exists values of the  $m_j$ s,  $0 \leq j \leq l-1$ , and  $k_j$ s,  $0 \leq j \leq l-1$ , as well as sequences

$Q_{m_j, k_j} \in R_{m_j, k_j}$  such that all actions in the compiled subplan  $Q_{m_0, k_0} P'_1 Q_{m_1, k_1} P'_2 \dots Q_{m_{l-1}, k_{l-1}} P'_l$  are applicable and the subplan leads to a state  $\Sigma = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq \inf(m_l - 1, \text{sup}(m_{l-1}, k_{l-1}))\} \cup \llbracket A \rrbracket_{m_{l-1}}(S)$  where  $m_l$  is the lowest stratum such that some predicate in the antecedent of an  $A_{m_l}$  is modified in the effect of  $P_l$  (or  $n + 1$  if there is none). If we take  $k_l$  to be the highest stratum of any predicate appearing in  $\mathcal{G}$  (or 0 if there is none), then there exists a sequence  $Q_{m_l, k_l} \in R_{m_l, k_l}$  applicable in  $\Sigma$  by proposition 3. We show that the compiled plan  $Q_{m_0, k_0} P'_1 Q_{m_1, k_1} P'_2 \dots Q_{m_{l-1}, k_{l-1}} P'_l Q_{m_l, k_l}$  satisfies the compiled goal (and  $\mathcal{G}(\text{fixed}_{k_l})$ ): since  $P$  is a plan for  $\Pi$ , the state  $S$  to which  $P$  leads is such that that  $S \models_A \mathcal{G}$ ; by proposition 3, the compiled plan  $Q_{m_0, k_0} P'_1 Q_{m_1, k_1} P'_2 \dots Q_{m_{l-1}, k_{l-1}} P'_l Q_{m_l, k_l}$  ends in a state  $\Sigma' = S \cup \{(\text{fixed}_i) \mid 0 \leq i \leq \text{sup}(m_{l-1}, k_{l-1})\} \cup \llbracket A \rrbracket_{\text{sup}(m_{l-1}, k_{l-1})}(S)$ , and then, because  $k_l$  is the highest stratum of any predicate in  $\mathcal{G}$ ,  $\Sigma' \models$  (and  $\mathcal{G}(\text{fixed}_{k_l})$ ).

Therefore, we have shown that if there exists a plan  $P$  for a PDDL $_{\mathcal{X}}$  instance  $\Pi$ , then there exists a compiled plan  $P'$  for the PDDL instance  $F(\Pi)$ . Conversely, we are going to show that:

**Proposition 5** *let  $P'$  be a plan for  $F(\Pi)$  and let  $P'_1 \dots P'_l$  be the subsequence of  $P'$  obtained by stripping all stratum and fixpoint actions. The sequence  $P = P_1 \dots P_l$ , where for all  $1 \leq j \leq l$   $P'_j$  is the augmentation of  $P_j$ , is a plan for  $\Pi$ .*

We need the following intermediate result, which states that any state of the compiled plan in which  $\text{fixed}_k$  is true contains the right values for the derived ground atoms at stratum  $k$  and below:

**Proposition 6** *Let  $\Sigma$  be a state at some point of the execution of the compiled plan. If  $\Sigma \models (\text{fixed}_k)$  where  $0 \leq k \leq n$ , then  $\Sigma = S \cup \llbracket A \rrbracket_k(S) \cup D$ , with  $\Sigma \cap \overline{\mathcal{B}} = S$  and  $D \cap \bigcup_{i=1}^k \overline{\mathcal{D}}_i = \emptyset$ .*

The proof is by induction on  $k$ . The case  $k = 0$  holds trivially. Let us assume that the proposition hold for some  $k$ , and let  $\Sigma$  such that  $\Sigma \models (\text{fixed}_{k+1})$ . The only way for  $(\text{fixed}_{k+1})$  to hold in  $\Sigma$  is for a  $\text{fixpoint}_{k+1}$  action to have established it before  $\Sigma$  and no augmented action to have deleted it inbetween (these are the only actions that can delete it). For the  $\text{fixpoint}_{k+1}$  action to be applicable, it must be the case that a  $\text{stratum}_{k+1}$  action has been applied before (because  $(\text{done}_{k+1})$  needs to be true and this is the only way this can happen), and in virtue of the preconditions of  $\text{stratum}$  and  $\text{fixpoint}$  actions nothing but an augmented action could have happened inbetween. Furthermore, this  $\text{stratum}_{k+1}$  action must not have set  $(\text{new})$  to true, and for it to be applicable,  $(\text{fixed}_k)$  must have held. Therefore, by the induction hypothesis, the state  $\Sigma'$  in which  $\text{stratum}_{k+1}$  was applied is such that  $\Sigma' = S' \cup \llbracket A \rrbracket_k(S') \cup D'$ , with  $\Sigma' \cap \overline{\mathcal{B}} = S'$  and  $D' \cap \bigcup_{i=1}^k \overline{\mathcal{D}}_i = \emptyset$ . Because nothing new was derived by  $\text{stratum}_{k+1}$ , it must be the case that  $\llbracket A \rrbracket_{k+1}(S') \subseteq D'$ . We also have  $(D' \setminus \llbracket A \rrbracket_{k+1}(S')) \cap \bigcup_{i=1}^{k+1} \overline{\mathcal{D}}_i = \emptyset$ : because the only actions that can add anything in  $\mathcal{D}_{k+1}$  to the state are those in  $\text{stratum}_{k+1}$ , which has  $(\text{fixed}_k)$  as a precondition, and by the induction hypothesis the only atoms in  $\mathcal{D}_k \cap \Sigma'$  are those in  $\llbracket A \rrbracket_k(S')$ , it must be the case that  $D' \cap \mathcal{D}_{k+1} \subseteq \llbracket A \rrbracket_{k+1}(S')$ . It follows that  $\Sigma'$  can be rewritten as  $S' \cup \llbracket A \rrbracket_k \cup D'' \cup (\llbracket A \rrbracket_{k+1} \setminus \llbracket A \rrbracket_k) = S' \cup \llbracket A \rrbracket_{k+1} \cup D''$  with  $\Sigma' \cap \overline{\mathcal{B}} = S'$  and  $D'' \cap \bigcup_{i=1}^{k+1} \overline{\mathcal{D}}_i = \emptyset$ , which completes the proof of Proposition 6.

Coming back to the proof of Proposition 5, we note that each augmented action  $P'_j$  has precondition (and  $f_j(\text{fixed}_{k_j})$ ), where  $f_j$  is the precondition of  $P_j$  and  $k_j$  is the highest stratum of any derived predicate appearing in  $f_j$  (or 0 if there is none). Consequently each  $P'_j$  is applied in a state  $\Sigma_j$  in which  $(\text{fixed}_{k_j})$  holds, and therefore, by Proposition 6,  $\Sigma_j = S_j \cup \llbracket A \rrbracket_{k_j}(S_j) \cup D$  with

$\Sigma_j \cap \overline{\mathcal{B}} = S_j$  and  $D \cap \bigcup_{i=1}^{k_j} \mathcal{D}_i = \emptyset$ . Furthermore  $\Sigma_j \models f_j$ , and since  $f_j$  only contains predicates in  $\mathcal{B}$  and  $\bigcup_{i=1}^{k_j} \mathcal{D}_i$ ,  $S_j \cup \llbracket A \rrbracket_{k_j}(S_j) \models f_j$ , which implies  $S_j \models_A f_j$ . Therefore, each  $P_j$  in the plan  $P$  is applicable. The same argument suffices to prove that  $P$  satisfies the goal  $\mathcal{G}$ , starting from the fact that the state  $\Sigma$  in which  $P'$  ends satisfies (and  $\mathcal{G}(\text{fixed}_k)$ ), where  $k$  is the highest stratum of any derived predicate appearing in  $\mathcal{G}$ . ■

As stated in Proposition 5, a plan  $P$  for a planning task  $\Pi$  can be recovered from a plan  $P'$  for the compiled planning task  $F(\Pi)$ , by simply stripping all occurrences of `stratum` and `fixpoint` actions. In the worst case of course, there is no polynomial  $p$  such that  $\|P'\| \leq p(\|P\|, \|\Pi\|)$ . Indeed, the worst-case is obtained 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 `stratumi` actions. Even if the planner is able to interleave as few `fixpointi` actions as possible with the `stratumi` 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  $|\mathcal{C}|$ .

## 5 Planning: With or Without Axioms?

The absence of a polynomial time compilation scheme preserving plan size linearly 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  $\text{FF}_{\mathcal{X}}$ —and compared results obtained by  $\text{FF}_{\mathcal{X}}$  on  $\text{PDDL}_{\mathcal{X}}$  instances with those obtained by FF on the PDDL instances produced via compilation with **f**.

$\text{FF}_{\mathcal{X}}$  transforms each axiom  $(: \text{derived}(d \ ?\vec{x})(f \ ?\vec{x}))$  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.

One domain we chose for our experiments is good old 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. 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 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 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`.

Both domain description versions and their compilation via **f** are given in the appendix. For each of those versions, we considered 3 types of planning tasks:

**strict:** A PDDL $\chi$  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. Note that expressing the resulting goal using `on` and `ontable` would require exponential space, highlighting once more the utility of derived predicates.

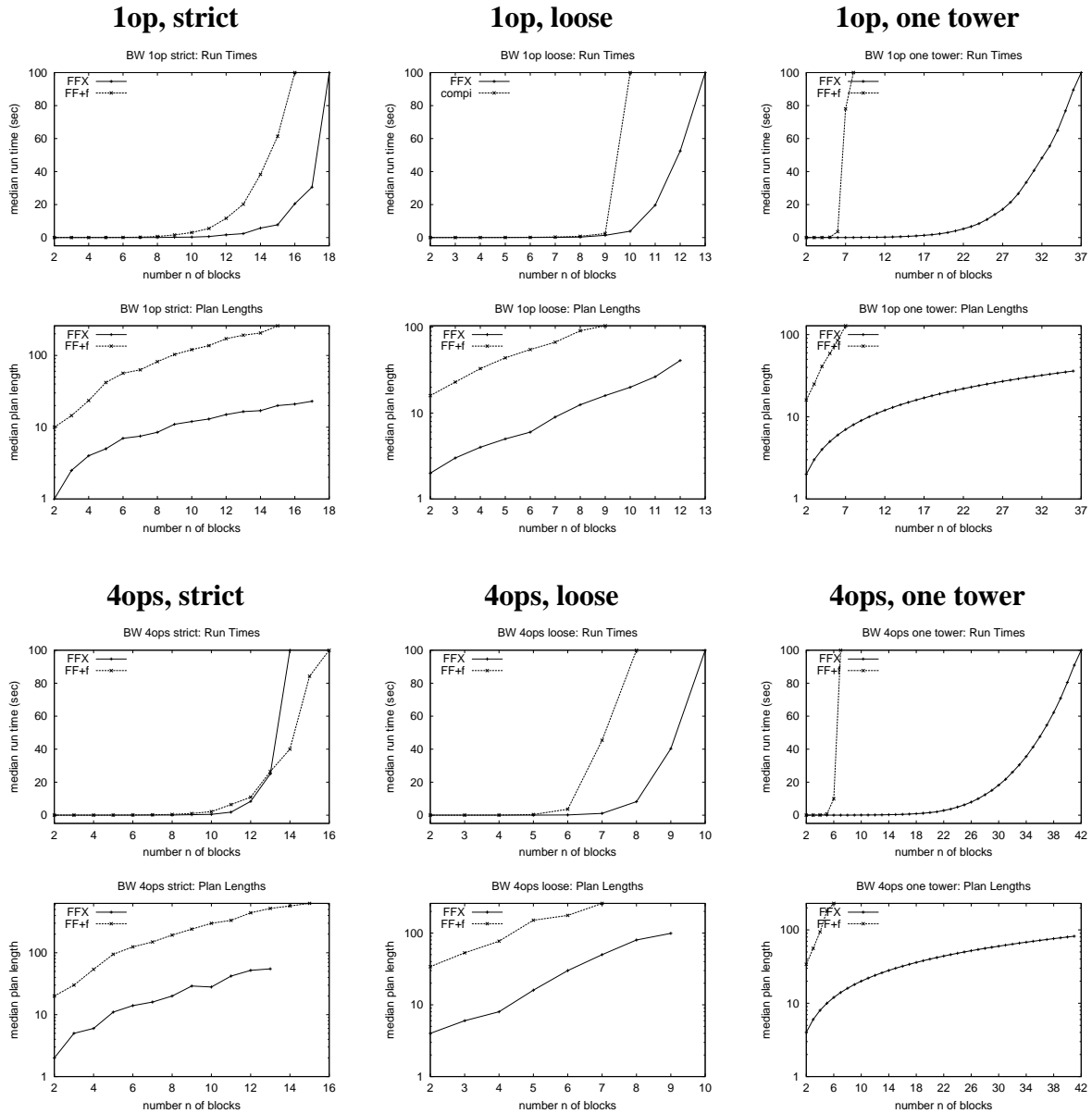
**loose:** A PDDL $\chi$  task is built from a single BW state by taking it to be the initial state and asking that any block which is on the table initially end up above all those that were initially not.

**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, which are all identical up to a permutation of the blocks, a single instance suffices per value of  $n$ . Figure 3 shows the median run-time and median plan length obtained by FF $\chi$  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 $\chi$  shows a significant improvement over that of FF+**f**. For **one tower** tasks, the improvement is dramatic, as FF $\chi$  finds the optimal plans whose length is linear in  $n$ . With the **strict** and **loose** tasks in contrast, the plans found by FF $\chi$  are only an order of magnitude larger 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 behavior in the above experiments is significantly worse than reported in the literature [Hoffmann and Nebel, 2001].

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. The domain description requires a number of complex, recursive, derived predicates to axiomatize the power flow, see [Bonet and Thiébaux, 2003] and the appendix. We considered a version of the benchmark in which the locations of the faults and the current network configuration are completely known, and the goal is to resupply all resuppliable lines. For each number  $n = 1$  to 7 feeders, we generated 100 random networks with a maximum of 3 switches per feeder and with 30% faulty lines. We also considered the networks of increasing difficulty described in [Bertoli *et al.*, 2002; Bonet and Thiébaux, 2003]: **basic**, **small-rural**, **simple**, **random**, and **simplified-rural**. The left-hand part of Figure 4 compares the median run times and plan length for FF $\chi$  and FF+**f** as a function of  $n$  on the random instances, while the right-hand one reports run times and plan length on the known instances. A dash (-) indicates that the instance could not be solved within 5000 secs. Again the improvement in performance resulting from handling axioms explicitly is undeniable. In these experiments, the plan length does not vary much with  $n$ : with our parameters for the random instances generation, it is clustered around 5 actions for

**Figure 3** Experimental Results for BW

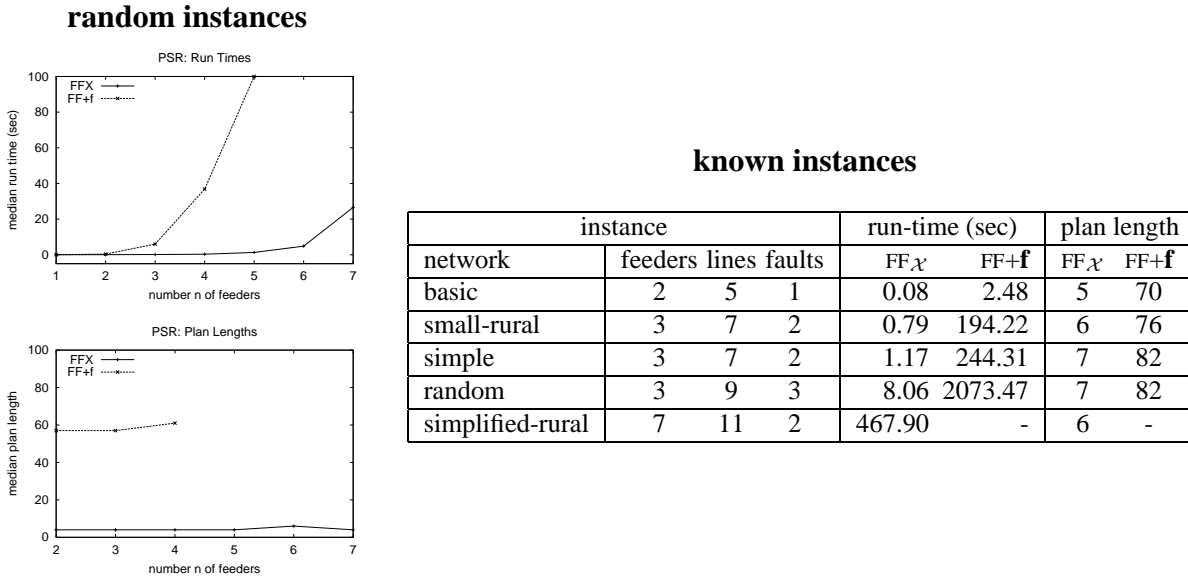


PDDL $\mathcal{X}$  instances, and around 50 for the compiled instances (the known instances exhibit similar figures). Yet this makes all the difference between what is solvable in reasonable time and what is not.

Although the domains in these experiments are by no means chosen to show off the worst-case for the compilation scheme, they nevertheless illustrate its drawbacks. The difference of performance we observe is due to the facts that compilation increases the branching factor, increases the plan length, and obscures the computation of the heuristic.

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

**Figure 4** Experimental Results for PSR



their definition for them wherever they occur until no occurrence remains. We did not experiment with compiling recursive derived predicate 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 report. Instead, we considered two treatments of the recursive predicates: one using axioms and running  $FF\chi$  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  $FF\chi$ , resp. by  $FF+f$  in Figure 3. To be precise, the run-times are slightly larger than those in Figure 3 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 axiomatized 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 constructs 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 above domains whose descriptions involve negated derived predicates.

## 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 compilation scheme which is more generally applicable than those previously published (and also more effective in conjunction with forward heuristic search planners like FF). Future work includes more extensive empirical studies involving a more elaborate treatment of axioms within FF and planners of different types, as well as the extension of derived predicates and axioms to the context of the numerical and temporal language features recently introduced with PDDL 2.1.

## Acknowledgements

Thanks to Blai Bonet, Marina Davidson, Stefan Edelkamp, Maria Fox, John Lloyd, and John Slaney for fruitful discussions which helped to improve this paper. Blai Bonet and John Slaney also contributed to the PDDL encoding of PSR given in the appendix.

## References

- [Apt *et al.*, 1987] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1987.
- [Bacchus, 2000] F. Bacchus. Subset of PDDL for the AIPS2000 planning competition. [www.cs.toronto.edu/aips2000](http://www.cs.toronto.edu/aips2000), 2000.
- [Barrett *et al.*, 1995] A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, Y. Sun, and D. Weld. UCPOP User’s Manual. Technical Report 93-09-06d, The University of Washington, Computer Science Department, 1995.
- [Bertoli *et al.*, 2002] P. Bertoli, A. Cimatti, J. Slaney, and S. Thiébaux. Solving power supply restoration problems with planning via symbolic model checking. In *Proc. ECAI*, pages 576–580, 2002.
- [Bonet and Geffner, 2001] B. Bonet and H. Geffner. GPT: a tool for planning with uncertainty and partial information. In *Proc. IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, pages 82–87, 2001.
- [Bonet and Thiébaux, 2003] B. Bonet and S. Thiébaux. GPT Meets PSR. In *Proc. ICAPS*, to appear, 2003.

- [Cadoli and Donini, 1997] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3,4):137–150, 1997.
- [Chandra *et al.*, 1981] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
- [Dantsin *et al.*, 2001] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
- [Davidson and Garagnani, 2002] M. Davidson and M. Garagnani. Pre-processing planning domains containing language axioms. In *Proc. UK PlansIG workshop*, 2002.
- [Fox and Long, 2002] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. [www.dur.ac.uk/d.p.long/IPC/pddl.html](http://www.dur.ac.uk/d.p.long/IPC/pddl.html), 2002.
- [Garagnani, 2000] M. Garagnani. A correct algorithm for efficient planning with preprocessed domain axioms. In *Research and Development in Intelligent Systems XVII*. Springer, 2000.
- [Gazen and Knoblock, 1997] C. Gazen and C. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proc. ECP*, 1997.
- [Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
- [Karp and Lipton, 1982] R. M. Karp and R. J. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28:191–210, 1982.
- [Kautz and Selman, 1992] H. A. Kautz and B. Selman. Forming concepts for fast inference. In *Proc. AAAI*, 1992.
- [Lloyd, 1993] J. Lloyd. *Foundations of Logic Programming*. Springer, 1993.
- [McDermott, 1998] D. McDermott. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
- [Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: simple hierarchical ordered planner. In *Proc. IJCAI*, 1999.
- [Nebel, 2000] B. Nebel. On the compilability and expressive power of propositional planning formalisms. *JAIR*, 12:271–315, 2000.
- [Slaney and Thiébaux, 2001] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.
- [Thiébaux and Cordier, 2001] S. Thiébaux and M.-O. Cordier. Supply restoration in power distribution systems — a benchmark for planning under uncertainty. In *ECP*, pages 85–95, 2001.
- [Vardi, 1982] M. Y. Vardi. The complexity of relational query languages. In *Proc. STOC*, 1982.

# A Domain Descriptions Used in Experiments

## A.1 BW 1op, with Axioms

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

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

  (:derived (clear ?x)
    (or (= ?x table) (forall (?y) (not (on ?y ?x)))))

  (:action move
    :parameters (?x ?y ?z)
    :precondition (and (not (= ?x table))
                      (not (= ?z ?y))
                      (not (= ?x ?z))
                      (clear ?x)
                      (clear ?z)
                      (on ?x ?y))
    :effect (and (on ?x ?z)
                 (not (on ?x ?y))))
)
```

## A.2 BW 1op, Compiled Version

```
(define (domain bw-compiled)
  (:requirements :strips)
  (:predicates (on ?x ?y)
    (clear ?x)
    (above ?x ?y)
    (fixed-0)
    (fixed-1)
    (done-1)
    (new))

  (:action axiom-1
    :parameters ()
    :precondition (and (fixed-0) (not (fixed-1)))
    :effect (and (done-1)
                 (forall (?x)
                   (when (and (not (clear ?x))
                              (or (= ?x table)
                                  (forall (?y) (not (on ?y ?x)))))
                     (and (clear ?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 fixpoint1
    :parameters ()
    :precondition (done-1)
    :effect (and (when (not (new)) (fixed-1))
                 (new))))
```

```

                (not (new))
                (not (done-1))))

(:action move
 :parameters (?x ?y ?z)
 :precondition (and (not (= ?x table))
                   (not (= ?z ?y))
                   (not (= ?x ?z))
                   (clear ?x)
                   (clear ?z)
                   (on ?x ?y)
                   (fixed-1))
 :effect (and (on ?x ?z)
              (not (on ?x ?y))
              (not (fixed-1))
              (forall (?x1) (not (clear ?x1)))
              (forall (?x1 ?y1)
                (not (above ?x1 ?y1))))
              (not (done-1))))
)

```

### A.3 BW 4ops, with Axioms

```

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

  (:derived (holding ?ob)
            (and (not (on-table ?ob))(not (exists (?b) (on ?ob ?b)))))

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

  (:derived (clear ?ob)
            (and (not (holding ?ob))
                (not (exists (?b) (on ?b ?ob)))))

  (:derived (arm-empty) (forall (?ob) (not (holding ?ob))))

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

```


```

## A.4 BW 4ops, Compiled Version

```

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

  (:action axiom-1
    :precondition (and (fixed-0) (not (fixed-1)))
    :effect (and (done-1)
                (forall (?ob)
                  (when (and (not (holding ?ob))
                             (not (on-table ?ob))
                             (not (exists (?b) (on ?ob ?b))))
                    (and (holding ?ob)
                        (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 fixpoint1
    :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 (?ob)
                  (when (and (not (clear ?ob))
                             (not (holding ?ob))
                             (not (exists (?b) (on ?b ?ob))))
                    (and (clear ?ob)
                        (new))))
                (when (and (not (arm-empty))
                          (forall (?ob) (not (holding ?ob))))
                    (and (arm-empty)
                        (new))))))

  (:action fixpoint2
    :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) (arm-empty))
: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 (arm-empty)))

(: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 (arm-empty))))

(: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 (arm-empty))))

(:action unstack
 :parameters (?ob ?underob)
 :precondition (and (fixed-2)(on ?ob ?underob) (clear ?ob) (arm-empty))
 :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 (arm-empty))))
)

```

## A.5 PSR, with Axioms

```

(define (domain psr-axioms)
  (:requirements :adl)
  (:types DEVICE SIDE LINE)
  (:predicates ; basic predicates
    (ext ?l - LINE ?x - DEVICE ?s - SIDE)
    (con ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
    (breaker ?x - DEVICE)
    (closed ?x - DEVICE)
    (faulty ?l - LINE)
    ; derived predicates
    (upstream ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
    (affected ?x - DEVICE)
    (safe-path ?x - DEVICE ?sx - SIDE ?l - LINE)
    (closed-path ?x - DEVICE ?sx - SIDE ?l - LINE)
    (feedable ?l - LINE)
    (fed ?l - LINE)
  )
; (upstream ?x ?sx ?y ?sy) iff side ?sx of device ?x

```

```

; is upstream of side ?sy of device ?y
(:derived (upstream ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
  (and (closed ?x)
    (or (breaker ?x)
      (exists (?z - DEVICE)
        (or (and (con ?z sidel ?x ?sx)
          (upstream ?z side2 ?x ?sx))
          (and (con ?z side2 ?x ?sx)
            (upstream ?z sidel ?x ?sx))))))
    (or (and (= ?sx sidel) (con ?x side2 ?y ?sy))
      (and (= ?sx side2) (con ?x sidel ?y ?sy))
      (exists (?z - DEVICE)
        (and (closed ?z)
          (or (and (con ?z sidel ?y ?sy)
            (upstream ?x ?sx ?z side2))
            (and (con ?z side2 ?y ?sy)
              (upstream ?x ?sx ?z sidel))))))))))

; (affected ?x) iff ?x is downstream of a faulty line
(:derived (affected ?x - DEVICE)
  (and (breaker ?x)
    (exists (?l - LINE)
      (and (faulty ?l)
        (or (and (closed ?x) (exists (?sx - SIDE) (ext ?l ?x ?sx)))
          (exists (?y - DEVICE)
            (and (closed ?y)
              (or (and (ext ?l ?y sidel)
                (exists (?sx - SIDE)
                  (upstream ?x ?sx ?y side2)))
                (and (ext ?l ?y side2)
                  (exists (?sx - SIDE)
                    (upstream ?x ?sx ?y sidel))))))))))))))

; (safe-path ?x ?sx ?l) = true iff there is a path made of non-faulty lines
; from ?x to ?l
(:derived (safe-path ?x - DEVICE ?sx - SIDE ?l - LINE)
  (or (and (= ?sx sidel)
    (or (ext ?l ?x side2)
      (exists (?l2 - LINE)
        (and (ext ?l2 ?x side2)
          (not (faulty ?l2))
          (exists (?z - DEVICE ?sz - SIDE)
            (and (ext ?l2 ?z ?sz)
              (safe-path ?z ?sz ?l)))))))
    (and (= ?sx side2)
      (or (ext ?l ?x sidel)
        (exists (?l2 - LINE)
          (and (ext ?l2 ?x sidel)
            (not (faulty ?l2))
            (exists (?z - DEVICE ?sz - SIDE)
              (and (ext ?l2 ?z ?sz)
                (safe-path ?z ?sz ?l))))))))))

; (closed-path ?x ?sx ?l) = true iff there is a path made of non-faulty
; lines and *closed* switches downstream from ?x to ?l
(:derived (closed-path ?x - DEVICE ?sx - SIDE ?l - LINE)
  (and (closed ?x)
    (not (faulty ?l))
    (or (and (= ?sx sidel)
      (or (ext ?l ?x side2)
        (exists (?l2 - LINE)
          (and (not (faulty ?l2))
            (ext ?l2 ?x side2)
            (exists (?z - DEVICE)
              (and (closed ?z)
                (exists (?sz - SIDE)
                  (and (ext ?l2 ?z ?sz)
                    (safe-path ?z ?sz ?l))))))))))))))

```

```

(closed-path ?z ?sz ?l)))))))))
  (and (= ?sx side2)
    (or (ext ?l ?x side1)
      (exists (?l2 - LINE)
        (and (not (faulty ?l2))
          (ext ?l2 ?x side1)
          (exists (?z - DEVICE)
            (and (closed ?z)
              (exists (?sz - SIDE)
                (and (ext ?l2 ?z ?sz)
                  (closed-path ?z ?sz ?l)))))))))))))
; (feedable ?l) = true iff there is breaker ?x and side ?sx such that
; (safe-path ?x ?sx ?l)
(:derived (feedable ?l - LINE)
  (and (not (faulty ?l))
    (exists (?x - DEVICE)
      (and (breaker ?x) (exists (?sx - SIDE) (safe-path ?x ?sx ?l))))))
)

; (fed ?l) = true iff there is a breaker ?x and side ?sx such that
; (closed-path ?x ?sx ?l)
(:derived (fed ?l - LINE)
  (exists (?x - DEVICE)
    (and (breaker ?x)
      (closed ?x)
      (exists (?sx - SIDE) (closed-path ?x ?sx ?l))))
)

(:action open
  :parameters (?x - DEVICE)
  :precondition (and (not (= ?x earth))
    (closed ?x)
    (not (exists (?b - DEVICE) (affected ?b))))
  :effect (not (closed ?x)))

(:action close
  :parameters (?x - DEVICE)
  :precondition (and (not (= ?x earth))
    (not (closed ?x))
    (not (exists (?b - DEVICE) (affected ?b))))
  :effect (closed ?x))

(:action wait
  :parameters ()
  :precondition (exists (?b - DEVICE) (affected ?b))
  :effect (forall (?b - DEVICE) (when (affected ?b) (not (closed ?b)))))
)

```

## A.6 PSR, Compiled Version

Note that although a single stratum suffices, we used two strata in order to improve the performance of the compilation scheme. The first stratum consists of `upstream` and `affected`, and the second consists of `safe - path`, `closed - path`, `fed`, and `feedable`.

```

(define (domain psr-compiled)
  (:requirements :adl)
  (:types DEVICE SIDE LINE)
  (:predicates (ext ?l - LINE ?x - DEVICE ?s - SIDE)
    (breaker ?x - DEVICE)
    (closed ?x - DEVICE)
    (faulty ?l - LINE)
    (con ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)

```

```

(upstream ?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
(affected ?x - DEVICE)
(safe-path ?x - DEVICE ?sx - SIDE ?l - LINE)
(closed-path ?x - DEVICE ?sx - SIDE ?l - LINE)
(feedable ?l - LINE)
(fed ?l - LINE)
(fixed-0)
(fixed-1)
(fixed-2)
(done-1)
(done-2)
(new))

(:action axiom-1
 :parameters ()
 :precondition (and (fixed-0) (not (fixed-1)))
 :effect
  (and (done-1)
    (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
      (when (and (not (upstream ?x ?sx ?y ?sy))
        (closed ?x)
        (or (breaker ?x)
          (exists (?z - DEVICE)
            (or (and (con ?z side1 ?x ?sx)
              (upstream ?z side2 ?x ?sx))
              (and (con ?z side2 ?x ?sx)
                (upstream ?z side1 ?x ?sx))))))
          (or (and (= ?sx side1) (con ?x side2 ?y ?sy))
            (and (= ?sx side2) (con ?x side1 ?y ?sy))
            (exists (?z - DEVICE)
              (and (closed ?z)
                (or (and (con ?z side1 ?y ?sy)
                  (upstream ?x ?sx ?z side2))
                  (and (con ?z side2 ?y ?sy)
                    (upstream ?x ?sx ?z side1))))))))
        (and (upstream ?x ?sx ?y ?sy)
          (new))))
      (forall (?x - DEVICE)
        (when (and (not (affected ?x))
          (breaker ?x)
          (exists (?l - LINE)
            (and (faulty ?l)
              (or (and (closed ?x)
                (exists (?sx - SIDE) (ext ?l ?x ?sx)))
                (exists (?y - DEVICE)
                  (and (closed ?y)
                    (or (and (ext ?l ?y side1)
                      (exists (?sx - SIDE)
                        (upstream ?x ?sx ?y side2)))
                    (and (ext ?l ?y side2)
                      (exists (?sx - SIDE)
                        (upstream ?x ?sx ?y side1))))))))))
          (and (affected ?x)
            (new))))))

(:action fixpoint1
 :parameters ()
 :precondition (done-1)
 :effect (and (when (not (new)) (fixed-1))
  (not (new))
  (not (done-1))))

(:action axiom-2
 :parameters ()
 :precondition (and (fixed-1) (not (fixed-2)))
 :effect

```

```

(and (done-2)
  (forall (?x - DEVICE ?sx - SIDE ?l - LINE)
    (when (and (not (safe-path ?x ?sx ?l))
              (or (and (= ?sx side1)
                      (or (ext ?l ?x side2)
                          (exists (?l2 - LINE)
                            (and (ext ?l2 ?x side2)
                                  (not (faulty ?l2))
                                  (exists (?z - DEVICE ?sz - SIDE)
                                    (and (ext ?l2 ?z ?sz)
                                          (safe-path ?z ?sz ?l))))))))
              (and (= ?sx side2)
                    (or (ext ?l ?x side1)
                        (exists (?l2 - LINE)
                          (and (ext ?l2 ?x side1)
                                (not (faulty ?l2))
                                (exists (?z - DEVICE ?sz - SIDE)
                                  (and (ext ?l2 ?z ?sz)
                                        (safe-path ?z ?sz ?l))))))))))
      (and (safe-path ?x ?sx ?l)
            (new))))
  (forall (?x - DEVICE ?sx - SIDE ?l - LINE)
    (when (and (not (closed-path ?x ?sx ?l))
              (and (closed ?x)
                   (not (faulty ?l))
                   (or (and (= ?sx side1)
                           (or (ext ?l ?x side2)
                               (exists (?l2 - LINE)
                                 (and (not (faulty ?l2))
                                       (ext ?l2 ?x side2)
                                       (exists (?z - DEVICE)
                                         (and (closed ?z)
                                               (exists (?sz - SIDE)
                                                 (and (ext ?l2 ?z ?sz)
                                                       (closed-path ?z ?sz ?l))))))))))
                   (and (= ?sx side2)
                         (or (ext ?l ?x side1)
                             (exists (?l2 - LINE)
                               (and (not (faulty ?l2))
                                     (ext ?l2 ?x side1)
                                     (exists (?z - DEVICE)
                                       (and (closed ?z)
                                             (exists (?sz - SIDE)
                                               (and (ext ?l2 ?z ?sz)
                                                     (closed-path ?z ?sz ?l))))))))))))))
      (and (closed-path ?x ?sx ?l)
            (new))))
  (forall (?l - LINE)
    (when (and (not (feedable ?l))
              (not (faulty ?l))
              (exists (?x - DEVICE)
                (and (breaker ?x) (exists (?sx - SIDE) (safe-path ?x ?sx ?l))))))
      (and (feedable ?l)
            (new))))
  (forall (?l - LINE)
    (when (and (not (fed ?l))
              (exists (?x - DEVICE)
                (and (breaker ?x)
                    (closed ?x)
                    (exists (?sx - SIDE) (closed-path ?x ?sx ?l))))))
      (and (fed ?l)
            (new))))))

(:action fixpoint2
 :parameters ()
 :precondition (done-2)
 :effect (and (when (not (new)) (fixed-2))

```

```

        (not (new))
        (not (done-2))))

(:action open
 :parameters (?c - DEVICE)
 :precondition (and (not (= ?c earth))
                   (closed ?c)
                   (not (exists (?b - DEVICE) (affected ?b)))
                   (fixed-1))
 :effect (and (not (closed ?c))
              (not (fixed-1)) (not(fixed-2))
              (not(done-1)) (not(done-2))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
                (not (upstream ?x ?sx ?y ?sy)))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?l - LINE)
                (and (not (safe-path ?x ?sx ?l))
                     (not (closed-path ?x ?sx ?l))))
              (forall (?x - DEVICE) (not (affected ?x)))
              (forall (?l - LINE)
                (and (not (feedable ?l))
                     (not (fed ?l))))))

(:action close
 :parameters (?c - DEVICE)
 :precondition (and (not (= ?c earth))
                   (not (closed ?c))
                   (not (exists (?b - DEVICE) (affected ?b)))
                   (fixed-1))
 :effect (and (closed ?c)
              (not(fixed-1)) (not(fixed-2))
              (not(done-1)) (not(done-2))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
                (not (upstream ?x ?sx ?y ?sy)))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?l - LINE)
                (and (not (safe-path ?x ?sx ?l))
                     (not (closed-path ?x ?sx ?l))))
              (forall (?x - DEVICE) (not (affected ?x)))
              (forall (?l - LINE)
                (and (not (feedable ?l))
                     (not (fed ?l))))))

(:action wait
 :parameters ()
 :precondition (and (exists (?b - DEVICE) (affected ?b)) (fixed-1))
 :effect (and (forall (?b - DEVICE) (when (affected ?b) (not (closed ?b))))
              (not(fixed-1)) (not(fixed-2))
              (not(done-1)) (not(done-2))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?sy - SIDE)
                (not (upstream ?x ?sx ?y ?sy)))
              (forall (?x - DEVICE ?sx - SIDE ?y - DEVICE ?l - LINE)
                (and (not (safe-path ?x ?sx ?l))
                     (not (closed-path ?x ?sx ?l))))
              (forall (?x - DEVICE) (not (affected ?x)))
              (forall (?l - LINE)
                (and (not (feedable ?l))
                     (not (fed ?l))))))
)

```