Pre-processing planning domains containing Language Axioms[*]

Marina Davidson and **Max Garagnani**
Department of Computing, The Open University,
Milton Keynes, MK7 6AA, U.K.
{m.davidson, m.garagnani}@open.ac.uk

Abstract

We present an automatic procedure for pre-processing planning problems containing *language* axioms, a specific type of domain axioms. The axioms considered are assumed to be in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow c$. The pre-processing approach described consists of *encoding* the language axioms directly inside the given operators, contrary to other (not always correct) existing approaches in which axioms are converted into additional operators. A distinction is made between two different types of axiom sets, namely, *recursive* and *non-recursive* sets, which are treated differently. While a simple replacement algorithm is appropriate for the non-recursive case, a more sophisticated procedure is proposed for pre-processing sets of recursive language axioms. Subject to some assumptions, the method presented can be generalised and used with *any* set of axioms of the type considered. Preliminary experimental results seem to suggest that the approach identified is sound and may lead to improvements in planning performance.

1 Motivation

One of the important issues in AI planning is how to define domains and problems in order to find a compromise between the expressiveness of a domain-definition language and the planners' ability to handle it. More expressive languages are necessary in order to model the *real world* and allow a natural, easy and correct encoding of realistically *large* problems. However, expressive languages require more complex (and, in general, slower) planning machinery. The underlying motivation of this work is to allow more expressive domain-definition languages while simultaneously (1) retaining good performance and (2) avoiding (whenever possible) the redevelopment of existing planning technology, which is continuously required for bringing and maintaining up-to-date 'old' planners that cannot handle new, expressive languages. To achieve this, we propose the adoption of a *pre-processing* approach. The main idea behind pre-processed planning is to build automatic tools that translate *domains* (and problems) from expressive representation languages into simpler ones, for which fast and efficient planners already exist. This approach allows preserving requirements (1) and (2); in addition, it offers the advantages of modularity, simplicity and non-specificity (a pre-processing tool can be plugged into *any* planning system that accepts as input the chosen 'target' domain-definition language).

In this paper, we focus on a specific feature of domain definition languages, namely, that of *domain axioms*. According to the description given in the original Planning Domain Description Language (PDDL) document, "axioms are logical formulas that assert relationships among propositions that hold within situations" (McDermott, Knoblock et al. 1998). A domain axiom *a* has the format *context*(*a*) $\rightarrow$ *implies*(*a*), where *context*(*a*) is a well-formed formula containing predicates of the language (possibly containing existential and/or universal quantification) and *implies(a)* is a one-predicate expression which is to be considered '*True'* whenever *context*(*a*) is *True*.

[*] This work has been partially supported by the EPSRC, grant no. GR/R53432/01.

For example, consider a '*Towns & Roads*' (TR) domain (see Figure1), in which towns (nodes) are connected by *one-way* roads (oriented arcs – the *distance* between towns is not relevant for the example) such that the operators of the domain allow *new roads* to be built (or existing road to be 'destroyed') at any time.

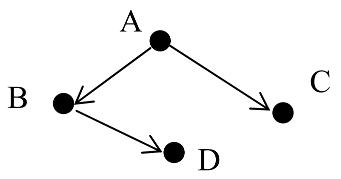

Figure 1. Example of initial state for a 'Towns & Roads' (TR) domain.

To avoid complexity, the domain should be modelled using the minimum number of predicates sufficient to describe unambiguously the current world state. The simplest domain-definition language consists of only one (*non-static*) predicate – say, *road(x,y)* – indicating that there is a one-way road going from town *x* to town *y*: $L_{in} = \{ \text{road}(x,y) \}$.

Suppose to be given a planning problem in which the goal '*g*' consists of having town A connected with town D in both directions. If we use the very 'primitive' language L_{in} to encode '*g*', we obtain a conjunction of two five-term disjunctive expressions, in which every term represents one of the possible connections between towns A and D:

$$\begin{aligned} g = {} & [(\text{road(A,D)}) \vee (\text{road(A,B)} \wedge \text{road(B,D)}) \vee (\text{road(A,B)} \wedge \text{road(B,C)} \wedge \text{road(C,D)}) \\ & \vee (\text{road(A,C)} \wedge \text{road(C,D)}) \vee (\text{road(A,C)} \wedge \text{road(C,B)} \wedge \text{road(B,D)})] \\ \wedge {} & [(\text{road(D,A)}) \vee (\text{road(D,B)} \wedge \text{road(B,A)}) \vee (\text{road(D,B)} \wedge \text{road(B,C)} \wedge \text{road(C,A)}) \\ & \vee (\text{road(D,C)} \wedge \text{road(C,A)}) \vee (\text{road(D,C)} \wedge \text{road(C,B)} \wedge \text{road(B,A)})] \end{aligned}$$

In order to avoid an exponential growth in the number of terms needed to represent goals (and preconditions), we can extend L_{in} and introduce a second predicate, '*connected(x,y)*', indicating the existence of a (not necessarily *direct*) one-way connection going from town *x* to town *y*. Using this more expressive, extended language $L_{out} = \{\text{road}(x,y), \text{connected}(x,y)\}$, the original goal '*g*' can be re-stated simply as $g' = \text{connected(A,D)} \wedge \text{connected(D,A)}$. However, the introduction of a new predicate requires the system to take into account the implicit *relationship* existing between the different terms of the language L_{out}. In particular, the two following axioms express the predicate *connected*( ) in terms of *road*( ) predicates:

(a_1) $\text{road}(x,y) \rightarrow \text{connected}(x,y)$
(a_2) $\text{road}(x,y) \wedge \text{connected}(y,z) \rightarrow \text{connected}(x,z)$

(where all the variables are implicitly universally quantified). The two '*language*' axioms[1] a_1 and a_2 are *necessary* and *sufficient* to determine the truth of any *connected*( ) predicate on the basis of the truth of the *road*( ) terms present in the current state.[2] They are part of the domain description and are given as input to the planner, which can use them to find the appropriate plan solution for an 'abstract' goal such as *g'*.

The above example illustrates how the adoption of axioms (and, in general, of more expressive domain definition languages) allows a clear and direct specification of goals and preconditions and a simpler and more natural problem description. This is necessary if we want the modern AI planning technology to become accessible to non planning-experts users who need to model complex, real-world domains. In addition, a 'user-friendly' representation is also required in

[1] The exact meaning of the term '*language axiom*' is explained in more details in Section 2.
[2] With reference to Figure 1, it is easy to see that the terms 'connected(A,B)', 'connected(A,C)' and 'connected(A,D)' follow immediately from the application of (a_1) and (a_2).

order to reduce errors in the domain encoding, which are much more likely when long and complex expressions are used. However, the introduction of axioms in the problem requires the planner's ability to handle such feature; in the next section we argue that this ability is currently offered only by a very limited set of systems.

The rest of the paper is organised as follows: Section 2 defines the term '*language axiom*' and compares it with other types of domain axioms; a further distinction is also made – namely, between '*recursive'* and '*non-recursive*' language axioms. In Section 3 we propose a simple algorithm for pre-processing planning problems containing non-recursive sets of axioms, whereas Section 4 describes the more complex algorithm for the recursive case. Section 5 illustrates the functioning of the algorithm with an experimental example in Blocks-World (BW). Finally, preliminary results, limitations and future directions are discussed in the last section of the paper.

2 Related work

According to the description language used in UCPOP (Penberthy, Barrett et al. 1995), "Domain axioms provide a convenient way to structure domains so that action effects can be short and sweet". Penberthy and colleagues suggest partitioning the predicates of the language into two sets: '*primitive*' and '*derived*'. Actions can only affect primitive predicates, while axioms are restricted to assert *derived* predicates by defining them in terms of a *context( )* which may include both primitive *and* derived terms. The original PDDL specification also requires that action definitions do not "have effects which mention predicates that occur in the *implies( )* field of an axiom. The intention is that action definitions mention 'primitive' predicate like *On( )* [in the BW domain]**,** and that all changes in truth value of 'derived' predicates – like *Above( )* **–** occur through axioms" (McDermott, Knoblock et al. 1998).

In this work, we are concerned with a specific type of domain axioms, which we refer to as '*language*' axioms. The main reason for introducing language axioms in a domain is to have a more expressive language that can model more closely and easily the real-world and offer the flexibility of higher-level, natural languages. The definition of language axiom – given below – reflects the above distinction between primitive and derived predicates.

Let us denote with the generic term L_{in} the 'inner' domain definition language consisting only of primitive predicates, i.e. *all and only* the predicates that are necessary and sufficient to specify *unambiguously* any possible state of the domain.[3] In the previous TR domain, L_{in} was simply {*road(x,y)*}. L_{out} will be the 'global', extended domain definition language consisting of *all* the predicates (primitive *and* derived), such that the *truth*-value of any derived predicate can be deduced from the truth of primitive predicates by applying the corresponding (language) axioms. In the previous example, L_{out} consisted of {*road(x,y), connected(x,y)*}. Finally, $L_{ext} = L_{out} \setminus L_{in}$ will be the 'external portion' of a domain language, consisting only of its derived predicates (in the example, the set {*connected*(*x*,*y*)}).

Then, a *Language Axiom* can be defined as an expression in the form

$$p_1 \wedge \ldots . \wedge p_n \rightarrow c \qquad (1)$$

such that $p_i \in L_{out}$, $i=\{1,..n\}$ (each p_i is a – derived or primitive – predicate containing typed variables) and $c \in L_{ext}$ is a single *derived* predicate (containing variables). All variables are implicitly *universally quantified*. Following the literature, we also require that *no action contain a derived predicate in its effects*.

Language axioms can be divided into two separate types: *recursive* and *non-recursive* (or simple) axioms. Given a set A of language axioms in the form of (1), A is said to be *recursive iff* there exists a subset of axioms $X \subseteq A$, $X=\{a_1, a_2, \ldots, a_m\}$ such that, for $i=\{1,\ldots m\text{-}1\}$, the predicate

[3] Clearly, this choice is determined by which aspects of the domain is important to model in a given context.

contained in *implies*(a_i) appears in *context*(a_{i+1}), and the predicate in *implies*(a_m) appears in *context*(a_1). The basic case of recursion consists of a single axiom in the form $p_1 \wedge \ldots \wedge p_k \wedge c \rightarrow c$ (e.g., with reference to the previous TR domain, X= {a_2}). The reasons for this distinction will become clearer in the following sections.

Here we would like to note the main difference between language axioms and other types of domain axioms, such as *heuristic* axioms (Kautz and Selman 1998). Heuristic axioms (which include state constraints, optimality heuristics and simplifying heuristics) are powerful means of restricting the search space and speeding up the planning process, but are not aimed at extending the expressiveness of the domain definition language. Some type of heuristic axioms (e.g. state constraints) can also be treated as rules in the form $p_1 \wedge \ldots . \wedge p_n \rightarrow c$, but, unlike language axioms, in a state constraint all predicates of *context*( ) and *imply*( ) belong to the language L_{in}.

Heuristic axioms have been widely discussed in a number of works (e.g., (Ginsberg and Smith 1988), (Gerevini and Schubert 1998), (Fox and Long 1998), (Scholz 2000)), but they do not fall within the scope of this paper, whose focus is restricted to the problem of planning in presence of language axioms.

One possible approach to support a more expressive language containing axioms is to extend or modify the planner's algorithm itself. Although in theory it should be possible to extend any of the modern forward state-space search algorithms to support domain axioms[4] (e.g. FF (Hoffmann 2000), GRT (Refanidis and Vlahavas 1999), TLPlan (Bacchus and Kabanza 2000)), since such extension is not of particular academic interest, the current potential users of AI planning technology are left with a very limited choice in terms of 'ready-to-use' systems. Indeed, to the best of our knowledge, of all of the modern planners only some HTN planners (namely, SHOP (Nau, Cao et al. 1999) and SIPE-2 (Wilkins, Myers et al. 1995)) can support language axioms directly. In addition to these, Prodigy (Veloso, Carbonell et al. 1995) and UCPOP (Penberthy and Weld 1992) also support axioms, but are much slower than other systems which adopt a restricted language (e.g. Graphplan (Blum and Furst 1997) and descendants) in finding a plan solution. Finally, extending SAT-based planners (Kautz and Selman 1999) (Ernst, Millstein et al. 1997) to deal with language axioms is not as trivial as it would seem, since a language axiom's implication sign ('$\rightarrow$') does not correspond exactly to a *classical* propositional logic 'implication', and the encoding of the domain axioms inside the CNF is not straightforward. Extending backward-chaining planning systems to deal with domain axioms is, on the other hand, even more difficult.

Since, in general, solving concise problems expressed in a sophisticated language seems harder than dealing with 'long' encodings adopting a simple, STRIPS-like formalism, an alternative approach which has been tried in the past is that of *pre-processed planning* (e.g. (Gazen and Knoblock 1997) (Garagnani 2000)). 'Pre-processing' means translating a domain written in an expressive language into an *equivalent* one using a simpler language and for which more efficient planners exist. In particular, given a planning problem containing domain axioms, an appropriate pre-processing module should be able to transform such problem into an equivalent one containing no axioms. The advantages of this approach are that it is conceptually simple, it is modular, and it is not necessarily specific to one planner. Gazen and Knoblock (Gazen and Knoblock 1997) proposed the algorithm for the conversion of a UCPOP domain representation into a Graphplan equivalent encoding, although the same algorithm can be used for other fast planners. Their method involves the transformation of domain axioms in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow c$ into equivalent 'deduce' operators (P,A,D) = ([$p_1, p_2, \ldots p_n$], [c], [ ]), which are then employed by the planner in the usual way during its search for a plan. However, the planning problems produced by Gazen and Knoblock's pre-processing algorithm lead to inefficient planning and are not always *equivalent* to the original problem (see (Garagnani 2000)). Garagnani

[4] During the search, forward planners have access to the complete world state and can easily deduce *all* the consequences currently implied by the axioms.

proposed a revised version of such algorithm, in which extra assertions ('deduction facts') were added to the state to keep track of any use of the axioms made during the plan. This allowed to correctly update the state using appropriate (pre-compiled) conditional effects. Although the work by Garagnani (see also (Garagnani 1998)) was one of the first attempts to transform domain axioms into conditional effects, such effects were still added to extra 'deduce' operators created according to Gazen and Knoblock's approach. In contrast, in the method presented here the conditional effects are automatically encoded inside the 'standard' operators given in the domain definition, a procedure which, so far, has been carried out only 'by hand'. Consider, for example, the ADL version of the BW '*Puton(b,l)*' action schema, reported in Figure 2.

Precond: $b \neq l,\ b \neq TABLE,\ \forall z\ \neg On(z,b),\ l = TABLE \vee \forall z\ \neg On(z,l)$
Add: *On(b,l)*
Above(b,l)
Above(b,z) for all z such that Above(l,z)
Delete: *On(b,z) for all z such that* $z \neq l$
Above(b,z) for all z such that $z \neq l \wedge \neg Above(l,z)$.

Figure 2. ADL version of Blocks-World '*Puton(b,l)*' action schema (adapted from (Pednault 1989)).

This operator has been 'hand-tailored' so that its effects include *all* the possible *consequences* of the language axioms (b_1) and (b_2):

(b_1) on(x,y) $\rightarrow$ above(x,y)
(b_2) on(x,y) $\wedge$ above(y,z) $\rightarrow$ above(x,z)

To the best of our knowledge, there is no automatic pre-processing system which, given these axioms and the 'classical' STRIPS version of the 'Put-on' schema, is able to generate the compiled ADL version with the appropriate conditional effects. In section 4 we describe an algorithm to carry out precisely this task. Interestingly, preliminary experimental results – some of which are reported in Sections 3 and 5 – suggest that this type of pre-processing can produce interesting results with respect to other pre-processing methods in terms of efficiency.

In general, the pre-processing approach seems to allow an efficient solution of problems formulated in an expressive language through the use of independently developed, fast planners. In this work, we have adopted this approach for the solution of planning problems containing language axioms.

3 Pre-processing non-*recursive* language axioms

If the given set of axioms is non-recursive, then each derived predicate can actually be 'reduced' to a disjunction of conjunctions of primitive terms. In other words, each predicate of L_{ext} can be expressed as a disjunction of conjunctions containing only predicates belonging to L_{in}. Hence, the algorithm to pre-process the given problem will simply consist of replacing each occurrence of a derived predicate '*c*' in the goal or preconditions with the disjunction of all the *contexts* of the axioms having '*c*' as their *implies*( ) field, and iterating this process until *all* derived predicates have disappeared from the problem. Notice that the termination of this process is guaranteed by the requirement of non-recursion in the axiom set.

For example, consider the following language axioms s_1 and s_2:

(empty-fridge ?x) $\wedge$ (shop-closed) $\rightarrow$ (very-hungry ?x) (s_1)
(empty-fridge ?x) $\wedge$ (no-money ?x) $\rightarrow$ (very-hungry ?x) (s_2)

"Party" problem (number of obj.)	Method (1): Language Axioms given as Input		Method (2): Gazen & Knoblock' s algorithm		Method (3): Iterative replacement			
	PRO DIGY	UC POP	FF	IPP	PRO DIGY	UC POP	FF	IPP
Party (1)	0.9/6	0.1/5	0.0/6	0.0/6	0.7/5	0.0/5	0.0/5	0.0/5
Party (2)	2.1/11	0.2/9	0.0/11	NS	1.6/9	0.0/9	0.0/9	0.0/9
Party (3)	3.8/16	0.2/13	0.0/16	NS	2.9/13	0.0/13	0.0/13	0.0/13
Party (4)	6.1/21	0.3/17	0.0/21	NS	4.6/17	0.1/17	0.0/17	0.0/17
Party (5)	8.8/26	0.3/21	0.0/26	NS*	6.5/21	0.1/21	0.0/21	0.0/21

Table 1: CPU-time(seconds)/plan-length for IPP, Prodigy, UCPOP and FF to solve 'Party' domain's problems containing language axioms s_1 and s_2, where NS - problem is proved unsolvable, NS* - problem was not solved after several hours. For IPP the number of parallel steps in the last column is 5 for all cases. All experiments were run on a P-III 550 MHz Linux machine with 1 GB of RAM.

An operator containing the term "(very-hungry ?x)" as precondition would simply have it replaced with "((empty-fridge ?x) $\wedge$ (shop-closed)) $\vee$ ((empty-fridge ?x) $\wedge$ (no-money ?x))". The same procedure can be applied to any goal expression.

In spite of the simplicity of the algorithm, its application produced some interesting results. Table 1 reported below shows the results obtained using the four different planners to solve problems in this trivial domain (here called 'Party'), using an increasing number of objects. In the experiment, three different approaches for dealing with language axioms have been used: (1) the language axioms were simply given as input to the systems; (2) Gazen and Knoblock's pre-processing algorithm, and (3) the Iterative replacement procedure described above, which shows better performances for all the problems considered across the three methods and no unsolved instances.

4 Pre-processing *recursive* Language Axioms

In order to describe the algorithm for pre-processing sets of recursive language axioms, let us illustrate with an example how such axioms can be encoded as *conditional effects* inside a given operator. Suppose to have two language axioms A1 and A2 and a (ground) operator instance *Op* = (P, A, D), where:

A1) $P(x,y) \rightarrow Q(x,y)$
A2) $P(x,y) \wedge Q(y,z) \rightarrow Q(x,z)$

Op(a,b,c): (P = [P(a,c), C(a), C(b)], A=[P(a,b)], D=[P(a,c)])

In order to identify the effects that the two axioms will have on the state at the moment of the application of *Op*, it is necessary to consider all the terms in the ADD and DEL lists of the operator. Let us begin with the conditional effects produced by the ADD list:

Step 1 :

- The term P(a,b) – which is being added – unifies with P(x,y) in axiom A1. After the unification, A1 will be P(a,b) $\rightarrow$ Q(a,b). Since P(a,b) is being added, then Q(a,b) should be added as well. We can substitute this 'reasoning' step by appending a new addition (in this case unconditional) to the operator *Op*:

ADD Q(a,b) (ε_1)

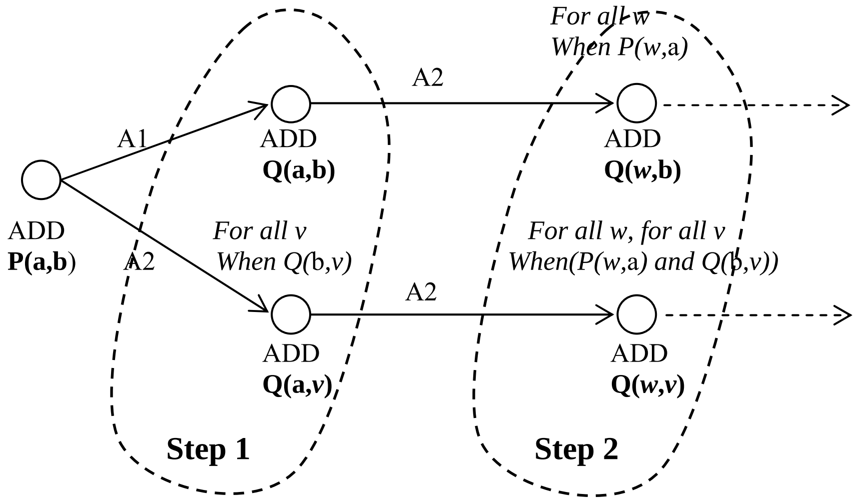

Figure 3: Generation of new conditional ADD effects for the operator Op (see text).

- The same term P(a.b) unifies with P(x,y) in A2. After the unification, axiom A2 will be P(a,b) $\wedge$ Q(b,$\boldsymbol{v}$) $\rightarrow$ Q(a,$\boldsymbol{v}$), where '$\boldsymbol{v}$' is free to vary on the appropriate domain. Since P(a,b) is being added, then *for each* predicate Q(b,$\boldsymbol{v}$) currently holding, Q(a,$\boldsymbol{v}$) should also hold. This can be realised by appending the following conditional addition to *Op*:

For all *v*, When Q(b,*v*) ADD Q(a,*v*) (ε_2)

At this point, we have encoded in the operator the *immediate* effects of its initial addition, P(a,b). This can be thought of as having 'produced' the first two nodes to the immediate right of the 'root' node P(a,b) in Figure 3. However, since these new effects can re-combine with the axioms and produce further consequences, we need to consider the effects of the newly added terms Q(a,b) and Q(a,v) (subject to the conditions specified in (1) and (2)):

Step 2

- Q(a,b) unifies with Q(y, z) in axiom A2. After the unification, A2 will be P($\boldsymbol{w}$,a) $\wedge$ Q(a,b) $\rightarrow$ Q($\boldsymbol{w}$,b) (for any $\boldsymbol{w}$ of the appropriate type). Since Q(a,b) is being added, then for all $\boldsymbol{w}$ such that P($\boldsymbol{w}$,a), Q($\boldsymbol{w}$,b) should be added as well. Hence:

For all *w*, When P(*w*,a) ADD Q(*w*,b) (ε_3)

- Q(a,v) unifies with Q(y, z) in A2. After the unification, axiom A2 will be P($\boldsymbol{w}$,a) $\wedge$ Q(a,v) $\rightarrow$ Q($\boldsymbol{w}$,v) for any $\boldsymbol{w}$ of the appropriate type (and for any 'v' such that Q(b,v) holds, as required by (ε_2)). Then for all $\boldsymbol{w}$ such that P($\boldsymbol{w}$,a), we should add Q($\boldsymbol{w}$, v) (subject to Q(b,v)). In summary:

For all w, For all v, When (P(w,a) $\wedge$ Q(b,v)) ADD Q(w,v) (ε_4)

Step 2 should then be repeated on the last set of effects added (the two rightmost nodes in Figure 3), and so on.

Although in general this algorithm will produce an infinite number of new conditional effects, in practice the specific characteristics of the domains are such that the procedure will often terminate. For example, when the main preconditions of the operator being pre-processed *contradict* any of the necessary conditions for the application of a new effect, then that branch of the graph can be pruned. To see this, assume, for example, that the two axioms A1 and A2 are the two BW axioms (b_1) and (b_2) considered in Section 2. If the predicates P( ), Q( ) and C( ) are interpreted, respectively, as *On*( ), *Above*( ) and *Clear*( ), then the operator instance *Op* is simply the step 'Put-on(a,b,c)', in which the block 'a', currently on 'c', is put on 'b'. Now, the precondition Clear(a) contradicts one of the conditions required by the two nodes created in Step 2, namely, *For all w, When P(w,*a*)…* . In fact, if 'a' is clear, there cannot be a block '*w*' such that On(*w*,a) – i.e., such that *P(w*,a*).* Therefore, in this case, both branches of the graph would be terminated, and the algorithm would produce a *finite* list of conditional additions (those produced by Step 1).

The line of reasoning behind the pre-processing of the DEL list of the operator is similar to the ADD case just described, although slightly more complex. For reasons of space, we report here only the pseudo-code of the procedure:

Algorithm δ (DEL list)

- Let *Op* be the operator which we are currently considering, and **A** the set of language axioms;

```
For each term y of the delete list of Op, do:
   W := $\emptyset$        ;; set W will contain the 1st new set of conditional deletions.
   For each axiom $a_i \in A$,
     if y unifies with any predicate $p_{ij} \in context(a_i)$
     then  Add a new conditional deletion to Op (and to the set W) in the form:
           "$\forall$ <free variables>, When ($p_{i1} \wedge \ldots \wedge p_{i,j-1} \wedge p_{i,j+1} \wedge \ldots \wedge d_i$) $\wedge$
             $\wedge$ ($\nexists$ applicable axiom $a_k$ such that $implies(a_k)$ unifies with $c_i$) DEL $c_i$"
   End-For;

   While W $\neq \emptyset$ do
      Z := $\emptyset$    ;; set Z will contain the most recent set of conditional deletions.
      For each conditional deletion $w \in W$,
        For each axiom $a_i \in A$,
          if  RHS(w) unifies with a derived predicate $d_i \in context(a_i)$
          then  Add a new conditional deletion to Op (and to the set Z) in the form:
                "$\forall$ <free variables>, When ($p_{i1} \wedge \ldots \wedge p_{i,n} \wedge$ LHS(w) ) $\wedge$
                  $\wedge$ ($\nexists$ applicable axiom $a_l$ such that $implies(a_l)$ unifies with $c_i$) DEL $c_i$"
        End-for;
      End-for;
      Re-assign set W := Z;
   End-while;
End-For.
```

The notations LHS(*w*) and RHS(*w*) indicate, respectively, the *condition* (left-hand-side of *w*) for the application of the conditional effect *w* and the term (right-hand-side of *w*) to be deleted when *w* is applied. With respect to the ADD list procedure, an extra '*When*' condition is required in order to prevent the incorrect deletion of terms which are currently being implied by other axioms. This check is implemented by requiring that, for each axiom a_k such that *implies*(a_k) unifies with the predicate c_i currently considered for deletion, *context*(a_k) is *False*.

As for the ADD list case, this algorithm can potentially generate an infinite number of conditional effects. However, once again, the specific characteristics of the domain may lead to a finite number of iterations. This is the case when the BW operator 'Put-on( )' is pre-processed with the two languages axioms (b_1) and (b_2), as demonstrated by the example described in the following section.

It should be pointed out that this algorithm relies on an important assumption, namely, that the *context*( ) field of each of the axioms being pre-processed contains *at most one* derived predicate. This assumption – which was not needed for the non-recursive case – is necessary here to avoid the possibility of a positive *interaction effect* between pairs (or tuples, in general) of derived terms. However, given a constant '*k*' such that the number of derived terms in the axiom contexts is always less than *k*, it is possible to derive a similar algorithm that can deal with all the possible interactions of up to *k* derived terms.

5 An example in Blocks-World

Given the two axioms A1 and A2 and the operator *Op* of the example of the previous section, the new, pre-processed operator *Op'* produced by the algorithm described is as follows:

Action *Op'*(a,b,c):
Main Precond.: (P a c) (C a) (C b) **Add:** (P a b) **Del.:** (P a c)
Conditional effects:
$\varnothing \Rightarrow$ **Add** (Q a b) (ε_1)
$\forall$?v, *When* (Q b ?v) $\Rightarrow$ **Add** (Q a ?v) (ε_2)
$\forall$?w, *When* (P ?w a) $\Rightarrow$ **Add** (Q ?w b) (ε_3)
$\forall$?w,?v, *When* (P ?w a) $\wedge$ (Q b ?v) $\Rightarrow$ **Add** (Q ?w ?v) (ε_4)
$\forall$?w,?v, *When* (P ?w a) $\wedge$ (P ?v ?w) $\Rightarrow$ **Add** (Q ?v b)
….
When ($\nexists$?w (P a ?w)(Q ?w c)) $\Rightarrow$ **Del** (Q a c)
$\forall$?v, *When* (Q c ?v) $\wedge$ (not (P a ?v)) $\wedge$ ($\nexists$?w ((P a ?w) $\wedge$ (Q ?w ?v))) $\Rightarrow$ **Del** (Q a ?v)
$\forall$?v, *When* (P ?v a) $\wedge$ …. … $\wedge$ ($\nexists$?w ((P a ?w) $\wedge$ (Q ?w c))) $\Rightarrow$ **Del** (Q ?v c)
….

In absence of specific information concerning the inherent properties of the domain, both branches (add and delete lists) of the algorithm would continue to produce conditional effects *ad infinitum*, since each new effect contains the predicate Q( ) which appears in the recursive axiom A2 and no explicit contradiction exists between the main preconditions and the conditions of the new effects. Notice that, unlike Blocks-World, there are domains having no inherent constraints that can terminate the algorithm. As an example, consider the Towns & Roads domain introduced in Section 1. Here, the predicates P( ) and Q( ) would be interpreted, respectively, as *road*( ) and *connected*( ), while the step *Op(a,b,c)* would consist of 'destroying' the road going from 'a' to 'c' to build a new one from 'a' to 'b'. In this case, it would *not* be possible to produce – without more specific information regarding, for example, the total number of towns and roads present in the initial state – a finite, general, axiom-free version of the given domain.[5]

In order to have a finite result and be able to use it in our experiments, we have applied the algorithm to the BW domain with one operator, *syntactically* equivalent to the above example.

[5] Although in all these examples the operator being processed was instantiated, the procedure would work identically even if 'a', 'b' and 'c' were variables.

The pre-processed *Put-on’* action reported below has been obtained simply by replacing predicate P( ) with On( ), Q( ) with Above( ) and C( ) with Clear( ):

Action *Put-on’*(a,b,c):
 Main Precond.: (On a c) (Clear a) (Clear b) **Add:** (On a b) **Del.:** (On a c)
 Conditional effects:
  ∅ ⇒ **Add** (Above a b)
  ∀ ?v, *When* (Above b ?v) ⇒ **Add** (Above a ?v)
  ~~∀ ?w, *When* (On ?w a) ⇒ **Add** (Above ?w b)~~
  ~~∀ ?w,?v, *When* (On ?w a) ∧ (Above b ?v) ⇒ **Add** (Above ?w ?v)~~
    When ~~(∄ ?w (On a ?w)(Above ?w c))~~ ⇒ **Del** (Above a c)
  ∀?v *When* (Above c ?v) ∧ ~~¬(On a ?v) ∧ (∄?w ((On a ?w)∧(Above ?w ?v)))~~ ⇒ **Del** (Above a ?v)
  ~~∀?v *When* (On ?v a) ∧ … … ⇒ **Del** (Above ?v c)~~

Given the inherent properties of BW, all the effects containing the condition “∀ ?w, *When* (On ?w a) …” can be removed and their branch terminated, as such condition cannot occur if Clear(a). In addition, since a block can only be on top of at most another block, parts of the conjunctive expressions of the two conditional deletions are always *True*. If all the indicated parts are removed, the resulting operator is sound and *semantically* equivalent to the ‘*Puton(b,l)’* operator reported at the end of Section 2, which had been produced manually (Pednault 1989).

In order to assess the effectiveness of this pre-processing approach, we used the operator resulting from the above procedure[6] with four different planners and compared the results with those obtained using Gazen and Knoblock’s algorithm and other systems (Prodigy and UCPOP) that can handle language axioms directly. The results are reported in Table 2, which seems to suggest a better performance of the proposed approach in all of the problem instances considered.

	Axioms given directly as input		Gazen & Knoblock’s		Our pre-processing algorithm			
Tower- invert* (no. of blocks)	PRO DIGY	UC POP	FF	IPP	PRO DIGY	UC POP	FF	IPP
B-World (3)	26/6	0.0/3	0.0/6	0.0/5	4.3/3	0.0/3	0.0/3	0.0/3
B-World (4)	NS*	0.1/4	0.2/11	0.0/7	10.7/4	0.1/4	0.0/4	0.0/4
B-World (5)	NS*	0.3/5	17/17	0.1/9	NS*	0.2/5	0.1/5	0.1/5
B-World (6)	NS*	2.3/10	NS	1.4/11	NS*	0.4/6	0.2/6	0.2/6
B-World (7)	NS*	NS	NS	170/13	NS*	1.7/7	0.6/7	0.3/7
B-World (8)	NS*	NS	NS	NS*	NS*	3.8/8	1.5/8	0.5/8
B-World (14)	NS*	NS	NS	NS*	NS*	NS	68/14	16/14
B-World (16)	NS*	NS	NS	NS*	NS*	NS	160/16	35/16
B-World (20)	NS*	NS	NS	NS*	NS*	NS	NS	164/20

Table 2: CPU-time(seconds)/plan-length for IPP, Prodigy, UCPOP, FF planners solving a variation of the BW Tower-invert problem with language axioms (b_1) and (b_2), in which the initial state consists of a tower of *n* blocks (Z, A_1, A_2, …A_{n-1}, with ‘*Z’* at the bottom) and the goal is given as a conjunction *above*(Z, A_1) ∧ *above*(Z, A_2) ∧… ∧ *above*(Z, A_{n-1}). This allows *(n–1)*! possible solutions of towers having *Z* on top and a permutation of the remaining (*n*-1) blocks underneath. NS – problem is not solvable, search limit exceed; NS* – problem was not solved after several hours. Experiments were run on a P-III 550 MHz Linux machine with 1 GB of RAM.

[6] The *Put-on’* operator shown here is actually a simplified version of the one used in the experiments, as a few extra preconditions are needed in order to deal with the possibility of the variables to be instantiated with the ‘Table’ object.

6 Discussion and further work

The work presented in this paper is currently in progress, and several issues and problems still have to be addressed. For example, one aspect that was not discussed here concerns the *order* in which the added conditional effects should be applied. Although in the specific case of the 'Put-on' operator the order between the effects happens to be irrelevant, in the general case it would seem logical to expect that the *application* of the conditional effects should follow the same 'causal' order in which they were created. The anomalous semantics of such an operator could, however, be avoided by adding further constraints on the conditional effects, or by 'splitting' the operator into several components and imposing an order among them using their preconditions.

With regard to the use of the algorithm proposed, a possible objection might be that a non-terminating procedure cannot possibly have many practical applications. On the contrary, we believe that an automatic tool which can produce – theoretically – the *correct* and *complete* list of conditional effects required for pre-processing a specific set of axioms inside an operator can be used in three important ways. First, in checking the *correctness* of hand-crafted operators (e.g. the ADL '*Puton*' schema proposed by Pednault, see Section 2); second, in actually supporting the *development* of operators and in understanding whether a set of language axioms *can* be finitely pre-processed; and third, in actually automating the pre-processing of planning problems containing language axioms in all those cases which result to be finite. However, further work is needed to investigate the possibility of guaranteeing the termination of the pre-processing procedure using alternative methods, for example on the basis of the specific instance of the problem which is being considered and which contains a finite number of objects. In addition, as mentioned earlier, the algorithm for the recursive case relies on the assumption that the number of derived predicates appearing in the *context*( ) field of the axioms is restricted to at most one. Part of the ongoing work is aimed at producing a generalised procedure which can deal with recursive language axioms having at most '*k*' (for a certain fixed *k*) derived predicates in the left-hand side.

In conclusion, we would like to point out what the main novel aspects of this work are. The idea behind the proposed approach is that the language axioms are encoded directly *inside* the existing operators, contrary to other existing pre-processing approaches in which the axioms are converted into *new* additional operators (a method which can lead to inefficient planning and, in some 'pathological' cases, to incorrect results). Although this idea was already present in the hand-crafted operator described in (Pednault 1989), to the best of our knowledge there exists currently *no general automatic procedure* which can produce the correct and complete pre-processed ADL version of an operator schema for a given set of language axioms. In this paper, an algorithm has been described to carry out this task under certain assumptions. According to the preliminary results reported, this approach seems to represent a promising direction in the area of pre-processed planning, although many issues still remain to be fully investigated.

References

Bacchus, F. and F. Kabanza (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* **116**:123-191.

Blum, A. and M. Furst (1997). Fast planning through planning graph analysis. *Artificial Intelligence* **90**: 281-300.

Ernst, M., T. Millstein, S. Weld (1997). Automatic SAT-compilation of planning problems. *Proceedings of the 15th International Joint Conference on AI (IJCAI-97)*, pp. 1169-1176.

Fox, M. and D. Long (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* **9**:367-421.

Garagnani, M. (2000). A correct algorithm for efficient planning with pre-processed domain axioms. In *Research and Development in Intelligent Systems XVII*, M.Bramer, A.Preece, F.Coenen (Eds), Springer-Verlag, pp.363-374.

Garagnani, M. (1998) Converting Inference Rules into Conditional Effects. *Proceedings of the 17th Workshop of the UK Planning and Scheduling SIG,* Huddersfield (UK), pp.203-205.

Gazen, B. and C. Knoblock (1997). Combining the expressivity of UCPOP with the efficiency of Graphplan. *Proceedings of 4th European Conference on Planning (ECP-97)*, Toulouse, France, pp.221-233.

Gerevini, A. and L. Schubert (1998). Inferring state constraints for domain-independent planning. *Proceedings of the 15th National Conference on AI*, Madison, WI, pp.905-912.

Ginsberg, M. and D. E. Smith (1988). "Reasoning about Action I: A Possible Worlds Approach." *Artificial intelligence* **35**: 165-195.

Hoffmann, J. (2000). A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*, Charlotte, North Carolina, USA.

Kautz, H. and B. Selman (1998). The role of domain-specific knowledge in the planning as satisfiability framework. *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Pittsburg, PA.

Kautz, H. and B. Selman (1999). Unifying SAT-based and Graph-based planning. *Proceedings of the 16th International Joint Conference on AI (IJCAI-99)*, Stockholm, Sweden, Morgan Kaufmann.

McDermott, D., C. Knoblock, M.Veloso, S.D.Weld, D.Wilkins (1998). PDDL - the planning domain definition language.Version 1.2. //www.cs.yale.edu/homes/dvm/

Nau, D., Y. Cao, A. Lotern, H. Munoz-Avila (1999). SHOP: Simple Hierarchical Ordered Planner. *Proceedings of IJCAI-99*, pp.968-973.

Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. *Proceedings of the 1st International Conference Principles of Knowledge Representation and Reasoning (KR-89)*, pp.324-332.

Penberthy, J. S., A. Barrett, M.Friedman, C.Kwok, K.Golden, Y.Sun, D.Weld (1995). UCPOP User's Manual, Version 4.0. Tech. report 93-09-06d. Seattle, USA, University of Washington.

Penberthy, J. S. and S. D. Weld (1992). UCPOP: A sound, complete, partial order planner for ADL. *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pp.108-114.

Refanidis, I. and I. Vlahavas (1999). A domain-independent heuristic for STRIPS worlds based on greedy regression tables. *Proceedings of the 5th European Conference on Planning (ECP-99)*, pp. 346-358.

Scholz, U. (2000). Extracting State Constraints from PDDL-like Planning Domains. *Proceedings of Workshop at AIPS2000 on "Analyzing and Exploiting Domain Konwledge for Efficient Planning"*, pp.43-48.

Veloso, M., J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe (1995). Integrating planning and learning: The PRODIGY architecture. *J. of Experimental and Theoretical AI* **7(1)**: 81-120.

Wilkins, D., K. Myers, J. Lowrance, L. Wesley (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI* **7(1)**:197-227.