Combining the Expressivity of UCPOP with the Efficiency of Graphplan

B. Cenk Gazen and Craig A. Knoblock

Information Sciences Institute and
Department of Computer Science
University of Southern California
Marina del Rey, CA 90292

Abstract. There has been a great deal of recent work on new approaches to efficiently generating plans in systems such as Graphplan and SATplan. However, these systems only provide an impoverished representation language compared to other planners, such as UCPOP, ADL, or Prodigy. This makes it difficult to represent planning problems using these new planners. This paper addresses this problem by providing a completely automated set of transformations for converting a UCPOP domain representation into a Graphplan representation. The set of transformations extends the Graphplan representation language to include disjunctions, negations, universal quantification, conditional effects, and axioms. We tested the resulting planner on the 18 test domains and 41 problems that come with the UCPOP 4.0 distribution. Graphplan with the new preprocessor is able to solve every problem in the test set and on the hard problems (i.e., those that require more than one second of CPU time) it can solve them significantly faster than UCPOP. While UCPOP was unable to solve 7 of the test problems within a search limit of 100,000 nodes (which requires 414 to 980 CPU seconds), Graphplan with the preprocessor solved them all in under 15 CPU seconds (including the preprocessing time).

1 Introduction

One of the important issues in planning is how to define domains and problems. There is a trade-off between the expressiveness and manageability of a formal domain definition language. On one hand, a practical language should be as high-level as possible so that the domain engineer can represent planning problems easily, accurately and naturally. On the other, the more complex the language, the harder it is for the planner to solve the problems.

Some of the planners that support a high-level language are ADL [1], Prodigy [2] and UCPOP [3]. UCPOP is a partial order planner that supports a very expressive domain definition language. The characteristic features of such a language are negations, conditional effects, disjunctive preconditions, universal and existential quantification, axioms, and facts. Given these constructs, it is usually possible to find a natural representation for a given domain.

Graphplan [4] is a graph algorithmic planner that runs much faster than traditional planners but supports a minimal language for defining domains and problems. In Graphplan, a domain is represented by a set of operators, each of which is defined by a list of parameters, a list of propositions as preconditions, and a list of add and delete effects. A problem is represented by a typed set of objects, a list of propositions as initial conditions, and a list of propositions as the goal. For most domains, this language is awkward to use, although once the domain is defined in this language, problems can be solved much faster than it is possible with UCPOP.

One approach to support a more expressive language is to extend the 'Planning Graph' of Graphplan [5]. Another is to develop a preprocessor that translates domains from an expressive representation language into a simpler one. Advantages of the second approach are that it is conceptually simple and that it is not necessarily specific to one planner. On the other hand, it cannot handle some language constructs as efficiently as a high-level planner can.

In this paper, we present a set of algorithms that transform a UCPOP domain and problem into an equivalent Graphplan domain and problem, although the same methods can be used with other fast planners, notably SATplan [6], that are based on simple representations. The goal is to make the best of both planners. UCPOP supports a rich set of domain definition language features, but is much slower compared to Graphplan, which only supports a minimal language.

The preprocessor takes as input a UCPOP domain and problem, and generates an equivalent pair in Graphplan's language by applying rewriting rules step by step. The result of each step is an equivalent representation of the domain where some of the language constructs have been replaced with simpler ones.

2 Rewriting UCPOP domains as Graphplan domains

In UCPOP, domains are defined by operators, axioms, facts, and safety constraints [7]. Problems are defined by a list of initial conditions and a goal expression. UCPOP operators are represented by a list of parameters, a precondition expression, and an effect expression. The last two are arbitrarily nested first-order logic expressions, although some semantically meaningless (in a planning problem) expressions are not allowed. The restriction follows from the fact that operators should have deterministic effects. The expressions can be formed using negations, conjunctions, disjunctions, implications, and quantification. When used in effect expressions, implications have different semantics and are called conditional effects. The semantics are different because the antecedent of a conditional effect is evaluated with respect to the set of propositions that hold before the operator is applied, whereas the consequent refers to the resulting set. Quantifications can be either universal or existential.

An axiom is a rule that allows the planner to deduce a proposition from the current set of valid propositions. UCPOP supports a restricted form of axioms, where an axiom can only deduce a single proposition. A fact is an arbitrary piece of code that is executed during planning. In UCPOP, they can only appear in

the preconditions of operators. Safety constraints are conditions that the planner must maintain throughout the plan.

Although both are based on the STRIPS representation [8], Graphplan has a very restrictive language as compared to UCPOP. A Graphplan domain definition consists of a list of operators. Each operator has a list of parameters, a list of preconditions, a list of 'add' effects and a list of 'delete' effects. The last three lists are assumed to be conjunctions. The precondition list contains propositions that need to hold before the operator can be applied. The application of an operator results in the propositions in the 'add' list to be added to the current set of valid propositions, whereas those propositions in the 'delete' list are removed from the same set. The restrictiveness of Graphplan's language comes from the fact that both preconditions and effects are lists and not arbitrary expressions.

Interestingly, this lack of expressive power in the domain definition language does not place an inherent limitation on the types of problems that Graphplan can solve. The preprocessor we have developed can automatically transform UCPOP domains into Graphplan domains. The current implementation handles all the UCPOP features except facts and safety constraints. We are working on extending the preprocessor to support facts, but safety constraints are better supported explicitly by a planner than with preprocessing.

In our preprocessing approach, we make two assumptions about the domains. First, we assume that objects are not created dynamically. This means that a list of all objects in the domain is available to the preprocessor. Second, we assume that all objects have types. However, this is not a limiting assumption as it is always possible to assign the same type (e.g., object) to every object in the domain. It is also possible to have multi-typed objects. For example, a plane can be typed both as a vehicle and a flying-object. If type information is available, our preprocessor can generate a smaller domain, which in turn makes Graphplan run more efficiently.

The preprocessor is structured as six layers (Figure 1). Each layer processes some specific language construct and generates output that is input to the next layer. The top layer accepts the domain definition from the user while the bottom layer creates the domain in Graphplan's language.

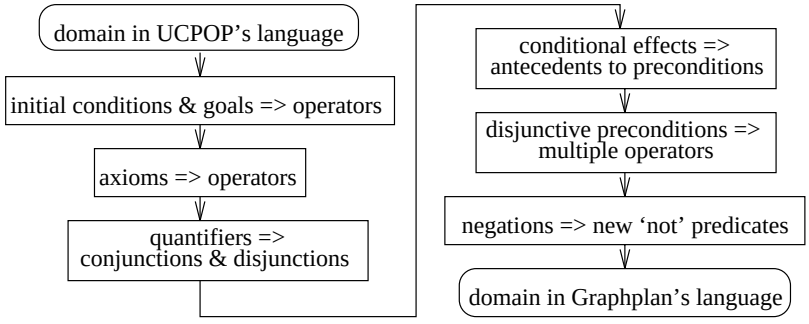

Fig. 1. Preprocessing Layers

2.1 Initial Conditions and Goals

Since initial conditions and goals can contain high-level language constructs, the first step is to convert them into operators. In general, this makes it unnecessary to make special case versions of the algorithms to process the initial conditions and goals.

Given: initial condition expression i; goal expression g; list of operators l_o

let o_{init} be a new operator with precondition (init-problem) and
  effect (and i (not (init-problem)))
let o_{goal} be a new operator with precondition g and effect (goal-achieved)
add o_{init} and o_{goal} to l_o
set the new initial condition to be (init-problem)
set the new goal to be (goal-achieved)

For example, the following problem definition:

```
(problem blocks :inits (forall (block ?x) (clear ?x))
                :goal (or (on a table) (exists (block ?x) (clear ?x))))
```

would be transformed into:

```
(problem blocks :inits (init-problem)
                :goal (goal-achieved))
(operator init-operator :precondition (init-problem)
                        :effect (and (forall (block ?x) (clear ?x)) (not (init-problem))))
(operator goal-operator :precondition (or (on a table) (exists (block ?x) (clear ?x)))
                        :effect (goal-achieved))
```

Creating these kinds of operators is a standard planning technique. However, it introduces two extraneous steps when introduced in preprocessing, and these steps need to be removed from the final plan or simply ignored. On the other hand, Graphplan's efficiency is not affected in any significant way as the only operator that is applicable at the first step is the 'init-operator'. After its application, the second level of propositions of the planning graph is exactly the same as it would be in the first level if the initial conditions were stated directly. A 'symmetric' argument holds for the 'goal-operator'.

2.2 Axioms

UCPOP restricts axioms to asserting a single proposition. Such axioms can be easily converted into 'deduce' operators. Since any deduced proposition may lose its validity after each step, it is necessary to find the operators that modify the propositions from which the axiom is derived, and to add an effect which negates the deduced proposition to these operators. This forces the axiom to be re-evaluated in a latter step if the deduced proposition is needed for another operator.

Given: axiom a; list of operators l_o

let o be a new operator
set the precondition of o to context(a) where
  context(a) is an expression that must be true before the axiom can be applied
set the effect of o to implies(a) where
  implies(a) is the single deduced proposition, with predicate p of n arguments
for each op in l_o
  if effect(op) contains any predicate in context(a)
    add (forall (v_1 v_2 ... v_n) (not (p v_1 v_2 ... v_n))) to effect(op)
add o to l_o

For example,

```
(axiom is-clear :context (or (eq ?x Table) (not (exists (obj ?b) (on ?b ?x))))
                :implies (clear ?x))
(operator put-on :parameters (obj ?x) (obj ?y) (obj ?d)
                 :precondition (and (on ?x ?d) (clear ?x) (clear ?y))
                 :effect (and (on ?x ?y) (not (on ?x ?d))))
```

would be transformed into:

```
(operator deduce-is-clear :parameters (obj ?x)
                          :precondition (or (eq ?x Table)
                                            (not (exists (obj ?b) (on ?b ?x))))
                          :effect (clear ?x))
(operator put-on :parameters (obj ?x) (obj ?y) (obj ?d)
                 :precondition (and (on ?x ?d) (clear ?x) (clear ?y))
                 :effect (and (forall (?v1) (not (clear ?v1)))
                              (and (on ?x ?y) (not (on ?x ?d)))))
```

Having axioms in a language makes the operator definitions cleaner and less error-prone by allowing the deducible effects that are repeated in many operators to be stated in a single axiom. In a way, preprocessing undoes that, and although the resulting domain is not as compact, its correctness is preserved. Since a conservative approach is followed in invalidating the deduced propositions, in the worst case an axiom may need to be asserted after each step. For example, the 'put-on' operator above does not have to assert (not (clear ?x)) because ?x is still clear after this action, but because it does, the axiom needs to be applied to re-assert (clear ?x) if another action requires (clear ?x) later.

2.3 Quantifiers

In this layer, universal quantifiers are expanded into conjunctions and existential quantifiers into disjunctions. Because dynamic creation of objects is not allowed and all the objects are explicitly declared, the expansion is straightforward. The preprocessor rewrites the quantified expression as a conjunction or disjunction of expressions. Each of these expressions is generated by instantiating the quantified variable with each of the objects that the variable can denote.

Given: a list of objects l_{ob} and a quantified expression $(q\ (t\ v)\ e)$ where $q \in \{forall, exists\}$; t is a type; v is the quantified variable; e is an expression
for each $ob \in l_{ob}$ s.t. type$(ob) = t$
  $e_i =$ instantiate v with ob in e
if existential(q)
  replace $(q\ (t\ v)\ e)$ with (or $e_1\ e_2\ \ldots e_n$)
else // universal(q)
  replace $(q\ (t\ v)\ e)$ with (and $e_1\ e_2\ \ldots e_n$)

Given the object definition '(block a) (block b) (block c)', the preprocessor will expand the expressions on the left into those on the right:

(forall (block ?x) (clear ?x))	(and (clear a) (clear b) (clear c))
(exists (block ?x) (clear ?x))	(or (clear a) (clear b) (clear c))

The expansion of quantifiers over all the objects is where the static domain assumption is necessary. Since it is assumed that a list of all the objects in the domain is available to the preprocessor, the expansion can be done easily. Our second assumption that all objects are typed is also useful at this layer, because the expansion is done only over the objects that have the corresponding type. In the worst case, where a quantified variable can range over all the objects in the domain (i.e., all objects have the same type), the expansion contains as many terms as there are objects. After the expansion, a domain in which only a few objects belong to each type will be more compact than an equivalent domain where all the objects have the same type. In practice, expansion of quantified expressions proved to be an acceptable preprocessing technique because most quantified variables can be restricted to range over a small number of objects.

2.4 Conditional Effects

Conditional effects are translated into simpler expressions by moving the antecedents into the preconditions. By definition, the consequent of a conditional effect is asserted only when the antecedent holds before the operator is applied. An equivalent way to represent such an operator is to use two operators: one with the antecedent in the preconditions and the consequent in the effects, and another with the negation of the antecedent in the preconditions and the consequent removed from the effects (Figure 2).

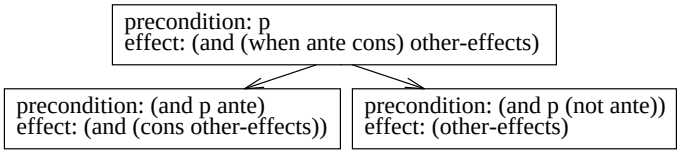

Fig. 2. Transforming Conditional Effects

More generally, an operator can have multiple conditional effects, which may appear either as separate conjuncts or as nested conditional effects. An example

of the former is (and (when p q) (when r s)), and an example of the latter is (when p (when q r)). Both cases are handled by applying the expansion of Figure 2 recursively to the resulting operators until all the conditional effects are eliminated.

This expansion generates an exponential number (in terms of the number of conditional effects) of operators, but in some cases it can avoid the problem by partially evaluating the antecedent and replacing the operator with a simplified one which does not have the conditional effect. Depending on the outcome of the evaluation, the consequent can be added to or removed from the effects.

Given: an operator o with precondition p and effect (and $e_1\ e_2\ \ldots\ e_n$)
if for some i, e_i is a conditional effect (when $a_i\ c_i$)
  if can-partially-evaluate(a_i)
    if a_i evaluates to true
      replace e_i with c_i in the effect
    else // a_i evaluates to false
      remove e_i from the effect
    recursively apply algorithm to o
  else // partial evaluation is not applicable
    recursively apply algorithm to the operator with precondition
      (and $p\ a_i$) and effect (and $e_1\ e_2\ \ldots\ e_{i-1}\ c_i\ e_{i+1}\ \ldots\ e_n$)
    recursively apply algorithm to the operator with precondition
      (and p (not a_i)) and effect (and $e_1\ e_2\ \ldots\ e_{i-1}\ e_{i+1}\ \ldots\ e_n$)
else // no conditional effects in the effect
  add o to the set of operators

Here are example operators:

```
(operator move-briefcase
    :parameters (location ?x) (location ?y)
    :precondition (at ?x)
    :effect (and (not (at ?x)) (at ?y)
                 (when (money-in) (and (not (money-at ?x)) (money-at ?y)))))
(operator deduce-table-clear
    :effect (and (when (eq a table) (clear a)) // after expansion of (forall (obj ?x) ...)
                 (when (eq b table) (clear b)) // over obj = {a, b, table}
                 (when (eq table table) (clear table))))
```

And how they would be rewritten:

```
(operator move-briefcase-1 :parameters (location ?x) (location ?y)
                           :precondition (and (at ?x) (money-in))
                           :effect (and (not (at ?x)) (at ?y)
                                        (not (money-at ?x)) (money-at ?y)))
(operator move-briefcase-2 :parameters (location ?x) (location ?y)
                           :precondition (and (at ?x) (not (money-in)))
                           :effect (and (not (at ?x)) (at ?y)))
(operator deduce-table-clear :effect (clear Table)))
```

An expansion of the 'deduce-table-clear' operator would result in 2^3 operators, but with partial evaluation only one operator is created. The partial evaluation technique requires the preprocessor to determine the static predicates of the

domain. As a first approximation, this can be done by finding those predicates that do not appear in the effects of any operator.

In some cases, although an antecedent contains only static predicates, the arguments of the predicates are parameters of the operator. Since the parameters are not bound during preprocessing, the truth value of the antecedent cannot be determined. Our solution is to expand the operators by instantiating over those parameters that appear in the antecedent. The antecedents in the following operator cannot be evaluated because ?boxy is not bound.

```
(operator push-box
      :parameters (?boxx ?boxy ?roomx)
      :precondition ...
      :effect ... (when (neq box1 ?boxy) ...) (when (neq box2 ?boxy) ...) ...)
```

The preprocessor instantiates ?boxy and creates multiple operators, so that partial evaluation is possible. One of the new operators is:

```
(operator push-box-box1
      :parameters (?boxx ?roomx)
      :precondition ...
      :effect ... (when (neq box1 box1) ...) (when (neq box2 box1) ...) ...)
```

At worst, this technique generates $(\#\text{ of objects})^{(\#\text{ of parameters})}$ operators, but this number is certainly much smaller than $2^{\#\text{ of objects}}$. In fact, Graphplan needs to do the expansion internally (for uninstantiated parameters) to build the planning graph, so it can still work efficiently.

However, partial evaluation is not always possible, in which case the number of operators created from a single operator is 2^n, where n is the number of conditional effects of that operator. Although n is generally a small number, it can get large when conditional effects are combined with universal quantifiers:

```
(forall (block ?x) (when (not (painted ?x)) (color ?x blue)))
```

The previous preprocessing layer would generate as many conditional effects as there are blocks, and removing them would create $2^{\#\text{of blocks}}$ operators. Although this is a significant theoretical limitation of the preprocessing approach, the problem does not occur often. In fact, in all the test domains, all occurrences of universally quantified conditional expressions could be expanded by using partial evaluation.

2.5 Disjunctions

A disjunction in the precondition of an operator is eliminated by creating multiple operators such that each new operator has one of the disjuncts in its precondition and the exact same effect as the original operator (Figure 3).

Given: an operator o with precondition (or $p_1\ p_2\ \ldots\ p_n$) and effect e

for each p in $\{p_1\ p_2\ \ldots\ p_n\}$

  let o_i be a new operator with precondition p and effect e

  add o_i to the set of operators

remove o from the set of operators

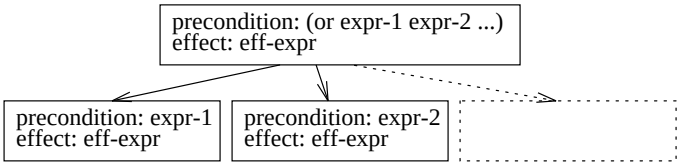

Fig. 3. Transforming Disjunctions

In the example below, the first operator, 'move', will be transformed into the next two operators, 'move-1' and 'move-2'.

```
(operator move :parameters (location ?x) (location ?y)
               :precondition (or (and (at ?x) (adj ?x ?y)) (and (at ?x) (adj ?y ?x)))
               :effect (and (at ?y) (not (at ?x))))
(operator move-1 :parameters (location ?x) (location ?y)
                 :precondition (and (at ?x) (adj ?x ?y))
                 :effect (and (at ?y) (not (at ?x))))
(operator move-2 :parameters (location ?x) (location ?y)
                 :precondition (and (at ?x) (adj ?y ?x))
                 :effect (and (at ?y) (not (at ?x))))
```

Unlike universally quantified conditional effects, we are not faced with an exponential blow-up problem for existentially quantified preconditions. This is because the number of operators generated for such an operator is proportional only to the number of disjuncts in the precondition of the operator.

2.6 Negations

Graphplan supports negated propositions in the effects through the use of 'delete' lists. However, negations are not allowed in the preconditions. As suggested in the Graphplan package, it is possible to work around this restriction by introducing a new predicate not-p when it is necessary to use the negation of p in the preconditions. Of course, the effects of all the operators need to be modified to keep p and not-p consistent. The 'normal' operators and the 'init-operator' require different processing. When 'normal' operators assert p, the preprocessor modifies the effects to also delete not-p. Similarly, when they assert the negation of p, that effect is changed to assert 'not-p' and delete p. The algorithm below processes the 'normal' operators.

Given: a list of operators l_o

for each p that appears as (not p) in some precondition(o) s.t. $o \in l_o$
  create a new predicate not-p
  for each $o \in l_o$ s.t. o is not the 'init-operator'
    if (p ...) $\in$ effect(o), add (del (not-p ...)) to effect(o)
    if (not (p ...)) $\in$ effect(o)
      replace it with (not-p ...)
      add (del (p ...)) to effect(o)
    if (not (p ...)) $\in$ precondition(o), replace it with (not-p ...)
replace all other negated effects (not (p ...)) with (del (p ...))

The next algorithm modifies the 'init-operator' to initialize the world such that it is consistent and 'closed' for all the predicates. By default, the initial conditions are always consistent and closed for predicates that do not have negated predicates introduced by the algorithm above. However, this is not true for those that do have negated predicates. When the truth of some proposition is asserted by two predicates p and not-p, the initial conditions must contain either one or the other. Since the initial conditions do not contain propositions with the latter predicate initially, the algorithm must add such propositions to the effect expression of the 'init-operator'.

Given: the init-operator o_{init}; a list of predicates that have negated predicates l_p

for each p in l_p
  for each $inst$ $(p\ arg_1\ arg_2\ \ldots\ arg_n)$ of p
    if $inst \notin$ effect(o_{init}) // effect(o_{init}) are the initial conditions
      add (not-$p\ arg_1\ arg_2\ \ldots\ arg_n$) to effect($o_{init}$)

For example, assuming 'at' appears as (not (at ...)) in some precondition, the first domain below would be transformed into the second:

```
(problem p :objects (location bank) (location home) (location office)
           :inits (at office)
(operator go :parameters (location ?x) (location ?y)
             :precondition (at ?x)
             :effect (and (not (at ?x)) (at ?y)))
```

```
(problem p :objects (location bank) (location home) (location office)
           :inits (not-at home) (not-at bank) (at office)
(operator go :parameters (location ?x) (location ?y)
             :precondition (at ?x)
             :effect (not-at ?x) (del (at ?x)) (del (not-at ?y)) (at ?y))
```

This transformation can result in a huge set of initial conditions because the number of possible instantiations of a single predicate is proportional to $(\#\text{ of objects})^{(\#\text{ of parameters})}$. In practice, this is not a problem because the number of parameters of any predicate is usually a small number.

3 Results

We ran the preprocessor+Graphplan pair on all the problems that come with the UCPOP 4.0 package. The preprocessor was implemented in about 600 lines of Lisp code. Both the preprocessor and UCPOP were run in Lucid Common Lisp on a Sun Ultra I workstation. Graphplan was originally written in C and its executable was run on the same machine.

Table 1 shows a comparison of the running times. UCPOP was able to find a solution faster than the preprocessor+Graphplan pair in only 13 problems among 41, but even that figure is misleading because all of those 13 problems are 'trivial' in the sense that the solutions are found in under one second. In fact, for 9 of them, both planners find solutions in less than 0.1 seconds. Moreover, UCPOP is much slower on harder problems even in domains where it did better than Graphplan on trivial problems of the same domain.

Two domains that caused trouble were the 'office-world' and the 'strips-world'. For the 'office-world' domain, there was not much we could do because in that domain objects are created dynamically and our static-world assumption does not hold. The 'strips-world' domain presented a number problems. First the domain contained a fact, which the preprocessor was not ready to handle. Fortunately, the fact was not a necessary part of the domain and the planning problems could be stated without using facts. The resulting simpler domain was still not solvable by UCPOP, but it was also not easy to preprocess because of the universally quantified conditional effects. Here is a typical operator from the 'strips-world' domain:

```
(operator push-box
    :parameters (?boxx ?boxy ?roomx)
    :precondition ...
    :effect (and (forall (?1) (and (when (neq ?1 ?boxx) (not (next-to robot ?1)))
                                   (when (neq ?1 ?boxy) (not (next-to ?boxx ?1)))
                                   (when (and (neq ?1 robot) (neq ?1 ?boxy))
                                              (not (next-to ?1 ?boxx)))))
                 (next-to ?boxy ?boxx) (next-to ?boxx ?boxy)  (next-to robot ?boxx)))
```

The effects assert that all 'next-to' propositions (with the exception of three) that refer to the robot or to either of the parameters ?boxx or ?boxy do not hold after this operator is applied. The three exceptions are those that appear in the last line. Expanding this operator without partial evaluation would generate 2^{36} operators because there are 12 objects that ?1 can denote and there are 3 conditional effects for each instantiation. Fortunately, all the antecedents involve static predicates, so partial evaluation is possible. Both problems of this domain are solved in about 10 seconds, whereas UCPOP fails after almost 1000 seconds with a search limit of 100,000 nodes. In this particular domain, preprocessing time dominates the planning time because a lot of partial evaluation needs to be done by the preprocessor, but the resulting domain is relatively simple for Graphplan.

Finally, it is also important to compare the quality of solutions generated by the planners. Since Graphplan always finds the plan with the least number of steps, its solutions are guaranteed to have equal or fewer steps than UCPOP's. In the set of problems we have experimented with, Graphplan's solutions are exactly the same as UCPOP's except in 'uget-paid' where UCPOP adds an extra step. The quality of Graphplan's solutions does not really show up because most of the test problems are simple and UCPOP fails to return a solution for the more difficult ones.

4 Discussion

Automated translation from a high-level domain definition language into a simpler one makes it possible to use simple but fast planners in complicated domains. Graphplan is one example of such a planner, but the same approach will also work with SATplan, which in some domains can perform an order of magnitude better than Graphplan. In fact, a similar rewriting approach [6] is followed to

Table 1. Comparison of UCPOP and Preprocessor+Graphplan

DOMAIN	PROBLEM	UCPOP (s)	Preprocessor (s)	Pre.+GP (s)
blocks-world-domain	suss.-anomaly	0.04	-	0.04
	tower-invert3	0.06	-	0.05
	tower-invert4	0.43	-	0.18
road-operators	road-test	0.02	-	0.01
hanoi-domain	hanoi-3	80.13	-	0.13
	hanoi-4	423 (NS)	-	1.54
ferry-domain	test-ferry	0.49	-	0.03
molgen-domain	rat-insulin	0.83	0.01	0.28
robot-domain	r-test1	0.02	0.02	0.04
	r-test2	9.76	0.01	0.07
monkey-domain	monkey-test1	0.14	-	0.07
	monkey-test2	0.82	-	0.17
	monkey-test3	253.87	-	0.51
briefcase-world	get-paid	0.01	-	0.02
	get-paid2	0.05	0.01	0.05
	get-paid3	0.25	0.01	0.13
	get-paid4	0.13	-	0.12
init-flat-tire	fixit	524 (NS)	0.01	0.24
	fix1	0.01	0.01	0.04
	fix2	0.02	-	0.02
	fix3	0.66	0.01	0.07
	fix4	0.03	0.01	0.02
	fix5	0.01	0.01	0.02
ho-world	ho-demo	0.02	0.03	0.05
fridge-domain	fixa	0.42	0.02	0.54
	fixb	408 (NS)	0.02	1.45
mcd-blocksworld	mcd-suss.-ano.	0.07	0.01	0.08
	mcd-tower-invert	414 (NS)	0.02	1.02
mcd-bw-axiom	mcd-sussman	0.03	-	0.03
	mcd-tower	0.07	-	0.05
uni-bw	uget-paid	0.01	0.03	0.06
	uget-paid2	0.06	0.02	0.08
	uget-paid3	1.46	0.03	0.32
	uget-paid4	0.13	0.03	0.33
sched-world-domain2	sched-test1a	0.01	0.13	0.15
	sched-test2a	0.01	0.25	0.28
prodigy-bw	prodigy-sussman	0.36	-	0.04
	prodigy-p22	695 (NS)	0.01	1.51
strips-world	move-boxes	980 (NS)	9.29	10.21
	move-boxes-1	961 (NS)	9.30	13.31
init-flat-tire2	fixit2	18.15	0.01	0.19
AVERAGE*		10.84	0.47	0.82

NS : No solution within the set search limit, which was 100,000 nodes.
- : '0', i.e., less than the minimum value the timer could show.
* : Average only for the solved problems.

encode planning problems into SAT. The input language for the SAT encoding algorithms is similar to the output from our preprocessor. By combining the two, SATplan can be used to solve problems defined in the language of UCPOP.

From a theoretical point of view, the expansion of conditional effects is a serious limitation of the preprocessing approach because the number of operators generated by the transformation is exponential in terms of the number of conditional effects. However, preprocessing is still a practical technique in that, for most problems, the exponential blow-up can be avoided by partial evaluation or object typing or both. Also, it is possible to avoid the problem altogether by using a planner, such as IPP [5], that can handle conditional effects efficiently.

One interesting improvement to the preprocessor is to eliminate some of the operators from a domain by looking at the problem at hand. This can be done similar to the way Graphplan builds planning graphs except by starting from the goals instead of from the initial conditions. When some operators are eliminated, it might also be possible to determine that more predicates are in fact static, thus making room for more partial evaluation than is possible initially. Although in trivial domains this optimization will not help much, in larger domains such an approach can reduce the set of operators for many problems.

Another extension is preprocessing facts. The same backward chaining algorithm that is used to find the relevant operators can also be used to determine the facts that are relevant to the problem being solved. The relevant facts can then be added to the initial conditions as normal propositions.

References

1. Edwin P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st. Int. Conf. On Principles of Knowledge Representation and Reasoning*, 1989.
2. Steven Minton, Craig A. Knoblock, D. Koukka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. Prodigy 2.0: The manual and the tutorial. Technical report, Department of Computer Science, Carnegie Mellon University, 1989.
3. Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4), 1994.
4. Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artifical Intelligence*, 90(1-2):281–300, 1997.
5. Jana Koehler, Bernhard Nebel, Jörg Hoffman, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, Toulouse, France, 1997.
6. Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI-96*, Portland, OR, 1996.
7. A. Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, Ying Sun, and Daniel Weld. UCPOP user's manual, version 4.0. Technical report, Department of Computer Science and Engineering, University of Washington, 1995.
8. Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.