Tech. Report CS TR 4157, Computer Science Department, University of Maryland, June 2000

SHOP and M-SHOP: Planning with Ordered Task Decomposition

Dana Nau   Yue Cao   Amnon Lotem   Héctor Muñoz-Avila

Department of Computer Science, and Institute for Systems Research
University of Maryland, College Park, MD 20742
U.S.A.

Abstract

SHOP (Simple Hierarchical Ordered Planner) and M-SHOP (Multi-task-list SHOP) are HTN planning algorithms with the following characteristics.

- SHOP and M-SHOP plan for tasks in the same order that they will later be executed. This avoids some task-interaction issues that arise in other HTN planners, making the planning algorithms relatively simple. This also makes it easy to prove soundness and completeness results.
- Since SHOP and M-SHOP know the complete world-state at each step of the planning process, they can use highly expressive domain representations. For example, they can do planning problems that require Horn-clause inferencing, complex numeric computations, and calls to external programs.
- In our tests, SHOP and M-SHOP were several orders of magnitude faster than Blackbox, IPP, and UMCP, and were several times as fast as TLplan.
- The approach is powerful enough to be used in complex real-world planning problems. For example, we are using a Java implementation of SHOP as part of the HICAP plan-authoring system for Noncombatant Evacuation Operations (NEOs).

In this paper we describe SHOP and M-SHOP, present soundness and completeness results for them, and compare them experimentally to Blackbox, IPP, TLplan, and UMCP. The results suggest that planners that generate totally ordered plans starting from the initial state can "scale up" to complex planning problems better than planners that use partially ordered plans.

1 Introduction

For many years, a widespread assumptions among AI planning researchers has been that total-order forward search is a bad idea because it causes excessive backtracking. Thus, AI planning systems have incorporated a number of clever and sophisticated ways to manipulate sets of actions that are partially ordered rather than totally ordered. Examples include causal-link planners such as UCPOP [Penberthy and Weld, 1992], planning-graph planners such as IPP [Penberthy and Weld, 1992], satisfiability planners such as SatPlan [Kautz and Selman, 1996] and Blackbox [Kautz and Selman, 1999], and HTN planners such as SIPE [Wilkins, 1988], O-Plan [Tate, 1994], and UMCP [Erol *et al.*, 1994a; Erol *et al.*, 1996].

However, recently several groups of researchers have begun to argue that total-order forward search also has a significant advantage: that it allows planners to use more expressive domain representations, which can be used to encode domain knowledge to make the planners highly efficient. More specifically:

- Prodigy [Veloso and Blythe, 1994; Fink and Veloso, 1995] does a forward state-space search that is guided by a means-end analysis made by backward chaining on the goals. Veloso and Blythe [Veloso and Blythe, 1994] showed that causal link commitments can affect the performance of partial-order planners when the goals have a property called *linkability*. In their experiments, Prodigy ran many times faster than SNLP [McAllester and Rosenblitt, 1991].

- TLplan [Bacchus and Kabanza, 1996; Bacchus and Kabanza, 2000] does a forward state-space search. It uses axioms written in modal logic to prune unpromising search paths. In Bacchus and Kabanza's tests, TLplan ran several orders of magnitude faster than Blackbox [Kautz and Selman, 1998], IPP [Koehler *et al.*, 1997], SatPlan [Kautz and Selman, 1996], Prodigy [Veloso and Blythe, 1994], and UCPOP [Penberthy and Weld, 1992].
- HSP [Bonet and Geffner, 1999] performs a heuristic-search to solve STRIPS-style planning problems. The search is performed in the forward direction, starting from the initial state. In each state, the value of the heuristic function is re-evaluated, to guide the search. HSP achieved very good performance results in the AIPS-98 competition. The FF planner [Hoffmann, 2000], which uses a similar approach to solve planning problems specified in ADL, was one of the best performers at the AIPS-2000 planning competition.
- Smith *et al.* [Smith *et al.*, 1997; Smith *et al.*, 1998] developed an approach called *ordered task decomposition*, which combines HTN-style problem reduction with left-to-right backtracking in a manner somewhat similar to Prolog's search strategy. They used ordered task decomposition successfully in domain-specific planners for several practical applications, including manufacturing planning [Smith *et al.*, 1997] and the game of bridge [Smith *et al.*, 1998]. They argued for the advantages of their approach by analyzing the reasons for its success in real-world applications [Nau *et al.*, 1998]. However, they could not compare ordered task decomposition head-to-head against domain-independent planning algorithms, because their implementations were domain-specific.

In order to test the performance of ordered task decomposition in a domain-independent setting, we have created a domain-independent formalization of the approach, and have implemented two different planners based on it: SHOP (Simple Hierarchical Ordered Planner) and M-SHOP (Multi-task-list SHOP). These planners have the following characteristics:

1. As ordered-task-decomposition planners, SHOP and M-SHOP require each HTN method to specify a linear ordering for the subtasks, and they take advantage of this restriction by decomposing tasks in the same order that they will be executed.
2. M-SHOP generalizes the SHOP planning algorithm by allowing the initial task specification to be unordered, and by automatically maintaining protection conditions and lists of subtasks for those tasks. In some planning domains, this allows the domain representation to be formulated more easily.
3. SHOP and M-SHOP avoid some of the task-interaction issues that occur in partial-order HTN planning systems, which generally need to have several different types of protection conditions [Erol *et al.*, 1994a] in order to handle partial orderings among tasks and subtasks. As a result, SHOP and M-SHOP are much simpler than HTN planners such as such as NONLIN [Tate, 1977], SIPE-2 [Wilkins, 1990], O-PLAN [Currie and Tate, 1991], and UMCP [Erol *et al.*, 1994b].
4. Since SHOP and M-SHOP always know the complete world-state at each step of the planning process, they can use considerably more expressive power in their domain representations than is available in most AI planners. For example, to evaluate the preconditions of HTN methods, SHOP and M-SHOP can use Horn-clause inferencing, numeric computations, and calls to arbitrary external programs.
5. SHOP and M-SHOP are sound and complete, provided that the precondition-evaluation algorithm is also sound and complete. For example, since Horn-clause inferencing is sound and complete, SHOP and M-SHOP are sound and complete if the preconditions are restricted to include only Horn clauses. Soundness and completess results can also be developed in the presence of certain kinds of calls to external programs, as described in [Dix *et al.*, 2000].

6. The expressive power of SHOP and M-SHOP can be used to create domain representations that encode highly efficient planning procedures. In our tests on blocks-world and logistics problems, SHOP and M-SHOP were several orders of magnitude faster than Blackbox, IPP, and UMCP, and generally about an order of magnitude faster than TLplan. This occurred even though SHOP and M-SHOP is coded in a slower language than most of the other planners.[1]
7. Lisp implementations of SHOP and M-SHOP are available as freeware at <http://www.cs.umd.edu/projects/shop>, under the terms of the GNU General Public License.
8. The approach is powerful enough for use in complex real-world planning problems. For example, in a joint effort with researchers at the US Naval Research Laboratory, we are developing a Java implementation of SHOP, for use as part of HICAP [Munoz-Avila *et al.*, 1999] plan-authoring system for noncombatant evacuation operations (NEOs). We intend to make JSHOP (the Java implementation of SHOP) available at the SHOP web site once it is complete.

In Sections 2 and 3, we describe the SHOP and M-SHOP algorithms, give some examples to illustrate their expressive power, and present soundness and completeness results for them. In Section 4 we describe experimental tests comparing them to four other planning systems: Blackbox, IPP, TLplan, and UMCP. In Section 5 we analyze the results and present our conclusions.

Table 1. Examples of SHOP notation for various logical expressions.

Type of object	An example in Prolog notation	The same example in SHOP notation
An atom	`on(block2,X)`	`(on block2 ?x)`
A conjunct	`ontable(block1),clear(block1)`	`((ontable block1) (clear block1))`
A substitution	`{block1/B, f(X)/Y}`	`((?b . block1) (?y . (f ?x)))`
Horn clause	`p(f(X)) :- q(X,c), r(g(Y),d), s(d).`	`(:- (p (f ?x)) ((q ?x c) (r (g ?y) d) (s d)))`
Horn clause	`q(b,c).`	`(:- (q b c) nil)`

2 SHOP

2.1 Definitions

This section defines the syntax and semantics used in the SHOP planning algorithm. For brevity, the definitions below are for a somewhat simplified version of SHOP's syntax and semantics, omitting some of the features available in the actual SHOP implementation. Section 2.3 gives an informal summary of those additional features, and a formal description of them is available at <http://www.cs.umd.edu/projects/shop/>.

Logical symbols. SHOP uses the usual first-order-logic definitions of the following entities, but with the notation adapted for use in Lisp:

- Constant symbols, function symbols, and predicate symbols. For these, SHOP uses Lisp symbols that do not begin with question marks, such as `block1`, `above`, or `make-clear`.
- Variable symbols. For these, SHOP uses Lisp symbols that begin with question marks, such as `?x` or `?foobar`. As usual, an expression is *ground* if it contains no variable symbols.

[1] SHOP, M-SHOP, and UMCP are in LISP; Blackbox, IPP, and TLplan are in C.

- Terms, atoms, ground atoms, conjuncts of atoms, Horn clauses, substitutions, and most-general unifiers (mgu's). SHOP uses the obvious Lisp notation for these, as illustrated in Table 1.

Logical inference. A *state* is a list of ground atoms, interpreted as a set (thus two states are equal if they contain the same atoms, regardless of the order in which those atoms appear). An *axiom set* is a set of Horn clauses. If S is a state, then S *satisfies* a conjunct C if there is a substitution u (called a *satisfier*) such that $S \cup X$ entails C^u (where C^u is the result of applying the substitution u to C). The satisfier u is a *most general satisfier* (mgs) if there is no other satisfier v for C that is more general than u. In contrast to mgu's (which are well known to be unique modulo lexical renaming), there may be several distinct mgs's for C.

Tasks. In addition to the logical symbols described earlier, SHOP has two other symbols: *primitive task symbols* (which are represented as Lisp symbols beginning with exclamation points), and *non-primitive task symbols* (which are represented as Lisp symbols that do not begin with exclamation points). A *task* is a list of the form

$(s\ t_1\ t_2\ \ldots\ t_n)$

where s (the task's *name*) is a task symbol, and t_1, t_2, …, t_n (the task's *arguments*) are terms. The task is *primitive* or *non-primitive* depending on whether s is primitive or non-primitive, respectively. A *task list* is a list of tasks. A task list is *primitive* if all of its tasks are primitive; otherwise it is *non-primitive.*

Operators. An *operator* is a expression o of the form

(:operator h D A c)

where h (the *head* of o) is a primitive task, D and A (o's *delete list* and *add list*, respectively) are lists of atoms containing no variable symbols other than those in h, and c (the *cost* of h) is a number. The number c may be omitted from the expression, in which case c is taken to be 1.

The intent of an operator is to specify that h can be accomplished by modifying the current state of the world to remove every atom in D and add every atom in A. More specifically, if t is a primitive task and there is an mgu u for t and h such that h^u is ground, then the operator instance o^u is *applicable* to t, and its head h^u is a *simple plan* for t. If S is a state, then the state produced by executing o^u (or equivalently, h^u) in S is the state

$\text{result}(S,h^u) = \text{result}(S,o^u) = (S - D^u) \cup A^u$.

Plans. A *plan* is a list of heads of ground operator instances. If $P = (p_1\ p_2\ \ldots\ p_n)$ is a plan and S is a state, then the *result* of applying p to S is the state

$\text{result}(S,p) = \text{result}(\text{result}(\ldots(\text{result}(S,p_1),p_2),\ldots),p_n)$.

Note that this definition differs from the corresponding one that would occur in STRIPS-style planning, because we don't need to worry about whether each operator is executable in the state to which it is applied.

Example 1. Suppose that the state S, the operators o and o', the subsitutions u and u', and the plan P are as shown below:

```
S = ((on a b) (ontable b) (clear a) (handempty));

o = (:operator (!unstack ?x ?y)
        ((clear ?x) (on ?x ?y) (handempty))
        ((holding ?x) (clear ?y)))

o' =   (:operator (!putdown ?block)
        ((holding ?block))
        ((ontable ?block) (clear ?block) (handempty))).
```

u = `((?x . a) (?y . b));`

u' = `((?block . b));`

P = `((!unstack a b) (!putdown b)).`

Then

head$(o)^u$ = `(!unstack a b);`

head$(o')^{u'}$ = `(!putdown b);`

result(S,o^u) = `((ontable b) (clear b) (holding a));`

result(S,P) = result(result(S,o^u),$(o')^{u'}$)
= `((ontable b) (clear b) (ontable a) (clear a) (handempty)).`

Methods. A *method* is an expression of the form

`(:method` h C `'`T`)`

where h (the method's *head*) is a compound task, C (the method's *precondition*) is a conjunct, and T (the method's *tail*) is a task list (the purpose of the quotation mark before T is explained in Section 2.2.1). The intent of a method m = `(:method` h C `'`T`)` is to specify that if the current state of the world satisfies C, then the task h can be accomplished by performing the tasks in T in the order given. More specifically, let t be a task atom and S be a state. Suppose that u is a unifier for h and t, and that v is a satisfier for C^u in S. Then the method instance $(m^u)^v$ is *applicable* to t in S, and the result of applying it to t is the task list $r = (T^u)^v$. The task list r is a *simple reduction* of t by m in S.

Example 2. Let m be the following method for moving a block from the top of a stack to the table:

```
m = (:method (move-block-to-table ?x)
     ((on ?x ?y) (clear ?x))
     '((!unstack ?x ?y) (!putdown ?x))
```

Let S be the state given in Example 1, and let t be the task atom

t = `(move-block-to-table a).`

Let u and v be the following substitutions:

u = `((?x a));`

v = `((?y b)).`

Then

```
(m^u)^v = (:method (move-block-to-table a)
          ((on a b) (clear a))
          '((!unstack a b) (!putdown a));
```

r = `((!unstack a b) (!putdown a));`

so r is the same as the plan P of Example 1.

Domains and problems. A *domain representation* is a set of axioms, operators, and methods. A *planning problem* is a triple (S,T,D), where S is a state, $T = (t_1\ t_2\ \dots\ t_k)$ is a task list, and D is a domain representation. Suppose (S,T,D) is a planning problem and $P = (p_1\ p_2\ \dots\ p_n)$ is a plan. Then we say that P *solves* (S,T,D), or equivalently, that P *achieves* T from S in D (we will omit the phrase "in D" if the identity of D is obvious) if any of the following is true:

- **Case 1.** T and P are both empty, (i.e., $k = 0$ and $n = 0$);

- **Case 2.** t_1 is a primitive task, p_1 is a simple plan for t_1, and $(p_2 \ldots p_n)$ achieves $(t_2 \ldots t_k)$ from result(S,p_1);
- **Case 3.** t_1 is a compound task, and there is a simple reduction $(r_1 \ldots r_j)$ of t_1 in S such that P achieves $(r_1 \ldots r_j\, t_2 \ldots t_k)$ from S.

The planning problem (S,T,D) is *solvable* if there is a plan that solves it.

Example 3. Let S, t, m, o, o',P be as in Examples 1 and 2. Let T be the task list (t), and let D be the domain representation $\{m,o,o'\}$. Then P solves (S,T,D).

2.2 Algorithm, Soundness, and Completeness

The SHOP planning algorithm is shown below:

```
procedure SHOP($S,T,D$)
1.  if $T$ = nil then return nil endif
2.  $t$ = the first task in $T$
3.  $U$ = the remaining tasks in $T$
4.  if $t$ is primitive and there is a simple plan for $t$ then
5.      nondeterministically choose a simple plan $p$ for $t$
6.      $P$ = SHOP(result($S,p$),$U,D$)
7.      if $P$ = FAIL then return FAIL endif
8.      return cons($p,P$)
9.  else if $t$ is non-primitive and there is a simple reduction of $t$ in $S$ then
10.     nondeterministically choose any simple reduction $R$ of $t$ in $S$
11.     return SHOP($S$,append($R,U$),$D$)
12. else
13.     return FAIL
14. endif
end SHOP
```

Since the algorithm is a straightforward implementation of the definition of the solution to a planning problem, it is easy to prove that the algorithm is both sound and complete.

Theorem 1 (Soundness of SHOP). Suppose one of the nondeterminstic traces of SHOP(S,T,D) returns a plan P. Then P solves the planning problem (S,T,D).

Proof. The proof is by induction on n, where n is the number of times SHOP is called.

Base case (n = 1). In this case SHOP does not call itself recursively, so it must return at step 1. Thus T = `nil` and P = `nil`, so from Case 1 of the definition of "achieves," P achieves T in S.

Induction step. Let $n>1$, and suppose that the theorem is true for every $m<n$. There are two cases:

Case 1. SHOP returns P at step 6. Let t, U, and p be as computed in steps 2–5 of SHOP. Then $U = (t_1\, t_2 \ldots t_i)$ for some i. Let $(p_1\, p_2 \ldots p_j)$ be the plan returned by the recursive call to SHOP(result(S,p),U,D) in step 6. From the induction assumption it follows that $(p_1\, p_2 \ldots p_j)$ achieves $(t_1\, t_2 \ldots t_i)$ in the state result(S,p). But t is a primitive task and p is a simple plan for t in S. Thus from Case 2 of the definition of "achieves," the plan $(p\, p_1\, p_2 \ldots p_j)$ achieves the task list $T = (t\, t_1\, t_2 \ldots t_i)$ in S.

Case 2. SHOP returns P at step 11. Let t and U be as computed in steps 2 and 3 of SHOP. Then $U = (t_1\, t_2 \ldots t_i)$ for some i. Let R be the simple reduction of t chosen in step 10 of SHOP. We know that $R = (r_1\, r_2 \ldots r_j)$ for some j and $P=(p_1\, p_2 \ldots p_k)$ for some k. From the induction assumption we know that $(p_1\, p_2 \ldots p_k)$ achieves $(r_1\, r_2 \ldots r_j\, t_1\, t_2 \ldots t_i)$ in S. Thus from Case 3 of the definition of "achieves," P achieves $T = (t\, t_1\, t_2 \ldots t_i)$ in S. ■

Theorem 2 (Completeness of SHOP) Suppose the planning problem (S,T,D) is solvable. Then at least one of the nondeterministic traces of SHOP(S,T,D) returns a plan.

Proof. For every plan P that solves (S,T,D), let P's *solution depth* be the length of P plus the total number of simple reductions needed to produce P from T. Let the *minimum solution depth* of (S,T,D) be the smallest solution depth of any plan that solves it. The proof is by induction on n, where n is the minimum solution depth of T.

Base step ($n = 0$). In this case, $T =$ `nil` and $P =$ `nil`, so SHOP returns P at step 1.

Induction step. Let $n>1$, and suppose that the theorem is true for every $m<n$. There are two cases:

Case 1. $T=(t_1\ t_2\ \ldots\ t_k)$ for some k, and t_1 is primitive. Then there must be at least one simple plan p for t_1 for which the minimum solution depth of (result(S,p),$(t_2\ \ldots\ t_k)$,D) is $n–1$, for otherwise the minimum solution depth of (S,T,D) could not be n. At step 6, one of the nondeterministic traces of SHOP recursively invokes SHOP(result(S,p),$(t_2\ \ldots\ t_k)$,D). From the induction assumption, this recursive invocation of SHOP returns a plan $(p_1\ p_2\ \ldots\ p_k)$. Thus at step 6, SHOP returns $(p\ p_1\ p_2\ \ldots\ p_k)$.

Case 2. $T=(t_1\ t_2\ \ldots\ t_k)$ for some k, and t_1 is non-primitive. Then there must be at least one simple reduction $R = (r_1\ r_2\ \ldots\ r_j)$ for t_1 such that the minimum solution depth of $(S,(r_1\ r_2\ \ldots\ r_j\ t_2\ \ldots\ t_k),D)$ is $n–1$, for otherwise the minimum solution depth for (S,T,D) could not be n. At step 11, one of the nondeterministic traces of SHOP recursively invokes SHOP$(S,(r_1\ r_2\ \ldots\ r_j\ t_2\ \ldots\ t_k),D)$. From the induction assumption, this recursive invocation of shop returns a plan $(p_1\ p_2\ \ldots\ p_k)$. Thus at step 6, SHOP returns $(p_1\ p_2\ \ldots\ p_k)$. ■

2.3 Extensions to SHOP

The implementation of SHOP includes several extensions to the syntax and semantics described in Section 2.1. Formal definitions of the extensions are available at <http://www.cs.umd.edu/projects/shop>. For brevity, we will not define the extensions formally here. However, we will illustrate each of them, in the context of a simple transportation-planning domain.

The scenario for the domain is that we want to travel from one location to another in a city. There are three possible modes of transportation: taxi, bus, and foot. Taxi travel involves hailing the taxi, riding to the destination, and paying the driver \$1.50 plus \$1.00 for each mile traveled. Bus travel involves hailing the bus, paying the driver \$1.00, and riding to the destination. Foot travel just involves walking, but the maximum feasible walking distance depends on the weather. Thus, different plans are possible depending on what the layout of the city is, where we start, where we want to go, how much money we have, and what the weather is like.

2.3.1 List of Extensions

Although the above domain is quite simple, most other AI planning systems do not have sufficient expressive power to fully represent it, because of the numeric computations that need to be done as part of the planning process. In contrast, the extended version of SHOP can represent it quite easily, as shown in Table 2. Below we describe each of the extensions to SHOP, referring to Table 2 for examples.

1. The tail of an axiom or the precondition of a method may include atoms of the form `(eval e)` where e is an expression to be evaluated by the Lisp evaluator. The atom is taken to be false or true depending on whether the evaluator returns `nil` or a non-`nil` value, respectively. For example, axiom A_1 of Table 2 uses this to specify that the taxi fare is \$1.50 plus \$1 for each mile traveled, and method M_1 uses this to specify that a precondition for paying the driver is that we must have enough money for the fare.

Table 2. A domain representation for SHOP for the transportation-planning domain.

	Axiom	**Meaning**

A_1 =	(:- (have-taxi-fare ?dist)
	    ((have-cash ?m)
	     (eval (>= ?m (+ 1.5 ?dist)))))
		To have enough money for a taxi, we need at least \$1.50 + \$1 for each mile to be traveled.

A_2 =	(:- (walking-distance ?u ?v)
	    ((weather-is 'good)
	     (distance ?u ?v ?w)
	     (eval (<= ?w 3)))
	    ((distance ?u ?v ?w)
	     (eval (<= ?w 0.5))))
		We are within walking distance of our destination if the weather is good and the distance is ≤ 3 miles, or if the weather is bad and the distance is ≤ 1/2 mile.

M_1 =	(:method (pay-driver ?fare)
	    ((have-cash ?m)
	     (eval (>= ?m ?fare)))
	    `((!set-cash ?m ,(- ?m ?fare))))
		If we have enough money to pay the taxi driver, then we can pay the driver by subtracting the taxi fare from our cash-on-hand.

M_2 =	(:method (travel-to ?q)
	    ((at ?p)
	     (walking-distance ?p ?q))
	    '((!walk ?p ?q)))
		If *q* is within walking distance, then one way to travel there is to walk there directly.

M_3 =	(:method (travel-to ?y)
	    (:first
	     (at ?x)
	     (at-taxi-stand ?t ?x)
	     (distance ?x ?y ?d)
	     (have-taxi-fare ?d))
	    `((!hail ?t ?x)
	      (!ride ?t ?x ?y)
	      (pay-driver ,(+ 1.50 ?d)))
	    ((at ?x)
	      (bus-route ?bus ?x ?y))
	    '((!wait-for ?bus ?x)
	     (pay-driver 1.00)
	     (!ride ?bus ?x ?y)))
		If we are at a taxi stand and we have enough money to pay the taxi fare, then we can travel to *y* by hailing the first taxi at the taxi stand, riding in it to *y*, and paying the driver the required fare. Otherwise, if we are on a bus route, then we can travel to *y* by waiting for a bus, paying the driver \$1, and riding the bus to *y*.

O_1 =	(:operator (!hail ?vehicle ?location)
	    ()
	    ((at ?vehicle ?location)))
		This is the operator for hailing a vehicle. It brings the vehicle to our current location.

O_2 =	(:operator (!wait-for ?bus ?location)
	    ()
	    ((at ?bus ?location)))
		This is the operator for waiting for a bus. It brings the bus to our current location.

O_3 =	(:operator (!ride ?vehicle ?a ?b)
	    ((at ?a) (at ?vehicle ?a))
	    ((at ?b) (at ?vehicle ?b)))
		This is the operator for riding a vehicle to a location. It puts both us and the vehicle at that location.

O_4 =	(:operator (!set-cash ?old ?new)
	    ((have-cash ?old))
	    ((have-cash ?new)))
		This is the operator for changing how much cash we have left.

O_5 =	(:operator (!walk ?here ?there)
	    ((at ?here))
	    ((at ?there)))
		This is the operator for walking to a location. It puts us at that location.

2. Axioms can have multiple tails, to be used in an "if-then-else" fashion. The axiom

   `(:-` $h\ t_1\ t_2\ t_3\ \ldots\ t_n$`)`

   says that

   - h is true if t_1 is true (in which case $t_2, \ldots, t_n$ will not be evaluated);
   - otherwise, h is true if t_1 is false but t_2 is true (in which case $t_3, \ldots, t_n$ will not be evaluated);
   - otherwise, h is true if t_1 and t_2 are false but t_3 is true (in which case $t_4, \ldots, t_n$ will not be evaluated);
   - …;
   - otherwise, h is true if $t_1, t_2, \ldots, t_{n-1}$ are false but t_n is true.

   This gives expressivity similar to a restricted version of Prolog's "cut," but in a way that is easier to understand. For example, axiom A2 uses this along with the `eval` construct described above, to say that walking distance is ≤ 3 miles in good weather, and ≤ 1 mile otherwise.

3. If a method's precondition is satisfied, then the tail is passed to the Lisp evaluator to obtain the reduction T described in Section 2.1. The syntax described in Section 2.1 uses Lisp's "quote" construct to prevent evaluation; for example, method M_2 uses this to invoke the `!walk` operator. However, Lisp's "backquote" and "comma" constructs can be used instead, to do partial evaluation. For example, Method M_1 uses this to compute how much money we will have after paying the fare, and pass this value as an argument to the `!set-cash` operator.

4. A method can have multiple pairs of preconditions and tails, to be used in an "if-then-else" fashion. The method

   `(:method` $h\ p_1\ t_1\ p_2\ t_2\ p_3\ t_3\ \ldots\ p_n\ t_n$`)`

   says that h reduces to t_1 if p_1 is true, or to t_2 if p_1 is false and p_2 is true, or to t_3 if p_1 and p_2 are false and p_3 is true, …, or to t_n if $p_1,\ldots,p_{n-1}$ are false and p_n is true. For example, method M3 uses this to specify that we won't consider bus travel unless we don't have enough money for taxi travel.

5. If the first element of a method's precondition or an axiom's tail is `:first`, SHOP's theorem prover will not look for all satisfiers, but instead will return after finding the first satisfier (just as Prolog would do). For example, method M3 uses this to tell SHOP that it should only consider hailing the first taxi at the taxi stand, rather than hailing all of them.

6. Axioms' tails and methods' preconditions can include negated atoms, which are evaluated using the closed-world assumption. The negation of an atom a is denoted by the expression `(not` a`)`. This extension is not illustrated in Table 2, but axiom A_2 could equivalently have been written as the following pair of axioms:

   ```
   (:- (walking-distance ?u ?v)
       ((weather-is 'good) (distance ?u ?v ?w) (eval (<= ?w 3))))
   (:- (walking-distance ?u ?v)
       ((not (weather-is 'good)) (distance ?u ?v ?w) (eval (<= ?w 0.5))))
   ```

In the most general case, it would not be possible to prove soundness and completeness for all of the extensions described above. However, with appropriate restrictions it is still possible to prove soundness and completeness. Below are two examples.

- If we allow negations in the tails of Horn clauses as in Item 6 above, then it is unclear what it means for the precondition of a method to be satisfied by a state, because there is more than one possible semantics for what logical entailment might mean [Subrahmanian, 1999]. However, if we restrict the set of Horn-clause axioms to be a stratified logic program, then the two major semantics for logical entailment agree with each other [Baral and Subrahmanian, 1993], and in this case SHOP will still be sound and complete.

- In Item 1 above, it is unclear what it would mean for a call to the Lisp evaluator to be sound and complete. However, the main reasons for allowing calls to the Lisp evaluator are (i) to provide a way to do numeric computations, and (ii) to allow queries to external information sources. If we do not allow calls to the Lisp evaluator but instead define ways to perform numeric computations and query external databases, it is possible to do this in a way that is both sound and complete. This is described further in [Dix *et al.*, 2000].

Table 3. Planning problems and their solutions.

Problem	Solutions
Go to the park, good weather, no cash	`1. ((!WALK DOWNTOWN PARK))`
Go to the park, bad weather, no cash	None (can't afford a taxi or bus, and it's too far to walk).

Go to the park, good weather, have $12	`1. ((!WALK DOWNTOWN PARK))`
	`2. ((!HAIL TAXI1 DOWNTOWN)`
	`(!RIDE TAXI1 DOWNTOWN PARK)`
	`(!SET-CASH 12 8.5))`

Go to park, good weather, have $80	`1. ((!WALK DOWNTOWN PARK))`
	`2. ((!HAIL TAXI1 DOWNTOWN)`
	`(!RIDE TAXI1 DOWNTOWN PARK)`
	`(!SET-CASH 80 76.5))`

Go uptown, good weather, no cash	None (can't afford a taxi or bus, and it's too far to walk).

Go uptown, good weather, have $12	`1. ((!HAIL TAXI1 DOWNTOWN)`
	`(!RIDE TAXI1 DOWNTOWN UPTOWN)`
	`(!SET-CASH 12 2.5))`

Go uptown, good weather, have $80	`1. ((!HAIL TAXI1 DOWNTOWN)`
	`(!RIDE TAXI1 DOWNTOWN UPTOWN)`
	`(!SET-CASH 80 70.5))`

Go to the suburb, good weather, have $12	`1. ((!WAIT-FOR BUS3 DOWNTOWN)`
	`(!SET-CASH 12 11.0)`
	`(!RIDE BUS3 DOWNTOWN SUBURB))`

1.1.2 Planning Problems and Solutions

Recall that a planning problem consists of an initial state, an initial task list, and a domain representation. Here is a planning problem in the transportation-planning domain:

- Initial state:

```
((at downtown)
 (weather-is 'good)
 (have-cash 12)
 (distance downtown park 2) nil)
 (distance downtown uptown 8)
 (distance downtown suburb 12)
 (at-taxi-stand taxi1 downtown)
 (at-taxi-stand taxi2 downtown)
 (bus-route bus1 downtown park)
```

```
        (bus-route bus2 downtown uptown)
        (bus-route bus3 downtown suburb))
```

- Task list: `((travel-to suburb))`
- Domain representation: see Table 2.

There are two solutions for the above problem:

1. `((!walk downtown park));`
2. `((!hail taxi1 downtown) (!ride taxi1 downtown park) (!set-cash 12 8.5))`.

Table 3 gives several other planning problems in the transportation-planning domain, along with their solutions. In each problem, the distances and bus routes are the same as above.

3 M-SHOP

3.1 Motivation

The SHOP planner assumes a total order between the tasks of the planning problem and between the subtasks of each method. While a total order between the subtasks of each method is quite natural in many domains, it often is undesirable to define a total order between the tasks of the planning problem.

As an example, consider a subset of the well known "logistics domain" [Veloso, 1992], simplified to exclude the notions of cities, airplanes and airports. The result is a simple planning domain that we will call the *simplified logistics domain*, in which packages have to be transported between locations (in the same city) using trucks. For a STRIPS-style representation of a problem in this domain, one could use atoms of the form

`(obj-at` p_i l_j`)`

to specify that object p_i is at location l_j. The initial state would then be a list of atoms $\{a_1,a_2,\ldots,a_i\}$ giving the initial locations of the packages, and the goal formula would be a similar set of atoms $\{b_1,b_2,\ldots,b_j\}$ giving the desired locations for the packages, as in the following example:

initial state: `((truck t1) (obj-at p1 L1) (obj-at p2 L2) (truck-at t1 L3));`
goal: `((obj-at p1 L4) (obj-at p2 L4)).`

There would be three STRIPS operators: one for loading a package onto the truck, one for moving the truck to a different location, and one for unloading a package from the truck. Most STRIPS-style planners would use these operators to develop a plan such as the one in Figure 1(a), which interleaves operations for p1 with operations for p2.

To represent this problem in SHOP, a naïve approach would be to let the initial state and task list be

S = `((truck t1) (obj-at p1 L1) (obj-at p2 L2) (truck-at t1 L3));`
T = `((achieve (obj-at p1 L4)) (achieve (obj-at p2 L4)));`

and to write methods and operators for the "achieve" task that correspond directly to the STRIPS operators. However, if we do this, then we have changed the semantics of the problem. Since a task list is an ordered sequence of tasks, what we are telling SHOP is that we want it to finish moving `p1` before it starts moving p2, as illustrated in Figure 1a.

One solution to this problem would be to change T to something like

T = `((achieve (obj-at p1 L4) (obj-at p2 L4)))`

to tell SHOP that we want it to achieve both of the conditions at the same time. In order to write methods, operator, and axioms to accomplish this kind of goal, we need to address the following issues:

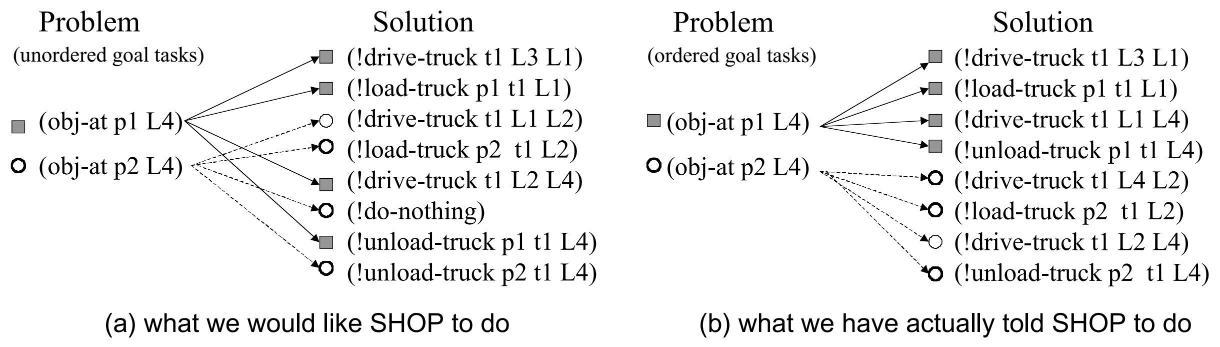

Figure 1. Example solutions for a planning problem in the simplified logistics domain.

- **Multiple agendas.** M-SHOP will need to keep track of multiple task lists. For example, there might be a separate task list for each package *p* that needs to be moved somewhere, telling what operations need to be done on *p*. For the package `p1` discussed earlier, the task list might be something like

```
((achieve (truck-at t1 L1))
 (!load-truck p1 t1 L1)
 (achieve (truck-at t1 L4))
 (!unload-truck p1 t1 L4)).
```

Similarly, the task list for p2 might be something like

```
((achieve (truck-at t1 L2))
 (!load-truck p2 t1 L2)
 (achieve (truck-at t1 L2))
 (!unload-truck p2 t1 L4)).
```

In order for M-SHOP to interleave the operations on different task lists (as shown in Figure 1a), M-SHOP needs to perform the following steps repeatedly (with some additional details as explained later):

```
select the first task t at the beginning of one of the task lists, and remove it from the task list
if t is primitive, then append it to the plan being constructed
otherwise, compute a reduction ($t_1$ ... $t_k$) of t, and insert it at the beginning of the task list
```

- **Deleted-condition interactions.** When SHOP interleaves items from multiple task lists, deleted-condition interactions may occur. Continuing the above example, suppose that SHOP first selects the task

```
(achieve (truck-at t1 L1))
```

Once SHOP has achieved this task, the next two tasks that are eligible to be selected are

```
(!load-truck p1 t1 L1);
(achieve (truck-at t1 L2)).
```

If SHOP now selects the second one of these tasks, it will delete the condition `(truck-at t1 L1)` needed to perform the first one. To prevent this from happening, M-SHOP will need to protect the condition `(truck-at t1 L1)` until this condition is no longer needed. As explained later, we accomplish this by allowing the add lists and delete lists of M-SHOP's operators to include the construct

```
(:protect a)
```

where *a* is any logical atom. For example, the construct

```
(:protect (truck-at t1 L1))
```

might occur in add list for the operator that moves the truck `t1` to location `L1`, and in the delete list for the operator that loads `p1` onto `t1`.

- **Specifying that some tasks cannot be interleaved.** In some cases, we may want to prevent M-SHOP to immediately perform the next task on the current task list, without even considering the possibility of interleaving tasks from other task lists. To indicate this, we allow tasks in M-SHOP to be flagged with the keyword `:immediate`. Continuing the above example, instead of attaching a `:protect` flag to the location of the truck, we could instead tell M-SHOP to load and unload the package `p1` immediately after moving the truck, as follows:

```
((achieve (truck-at t1 L1))
 (:immediate !load-truck p1 t1 L1)
 (achieve (truck-at t1 L4))
 (:immediate !unload-truck p1 t1 L4)).
```

Table 4 gives a domain representation for SHOP for the simplified logistics domain. To maintain multiple agendas, the domain representation stores the agenda information as part of the current state of the world. To handle protected conditions, the domain representation uses a predicate called `wait-on-location` to implement a counter for each possible atom that might need to be protected. It increments the atom's counter whenever an atom needs to be protected, and decrements the counter when the protection request is released. The domain representation considers the atom to be protected whenever the counter is nonzero.

The details of these methods, operators, and axioms are rather specific to the simplified logistics domain. However, several of the underlying ideas—e.g., how to maintain multiple agendas, and how to implement protected conditions via counters—are general principles that can be useful on a variety of planning problems. For example, in our domain representations for SHOP for the blocks world and the (unsimplified) logistics problem (see Section 4.1), we again needed to implement agendas and protected conditions. Because of the general need for such constructs, we have developed a modified version of SHOP called M-SHOP that incorporates them directly into the planning algorithm, so that they do not need to be implemented as part of the domain representation. This allows the domain representations for M-SHOP to be much simpler than those for SHOP. As an example, Table 5 shows the domain representation for M-SHOP for the simplified logistics domain.

The definition of M-SHOP and the proofs for its soundness and completeness are presented in the following sections.

Table 4. A domain representation for SHOP for the simplified logistics domain.

Axioms, operators, and methods	**Descriptions of what they mean**

`(:- (different ?x ?y)`
`    ((not (same ?x ?y))))`
 	
`(:- (same ?x ?x) nil)`
	Axioms to tell whether two objects are the same or different.

`(:operator (!load ?p ?t ?l)`
`  ((at ?p ?l))`
`  ((at ?p ?t)))`
	Operator to load package *p* onto truck *t*.

`(:operator (!unload ?p ?t ?l)`
`  ((at ?p ?t))`
`  ((at ?p ?l)))`
	Operator to unload package *p* from truck *t*.

```
(:operator (!move ?t ?o ?d)
  ((at-truck ?t ?o))
  ((at-truck ?t ?d)))
```

Operator to move truck *t* from location *o* to location *d.*

```
(:operator (!assert ?g)
  ()
  ?g
  0)
```

Operator to insert a fact *g* into the current state. Since this is just a bookkeeping operator, its cost is 0.

```
(:operator (!remove ?g)
  (?g)
  ()
  0)
```

Operator to remove a fact *g* from the current state. Since this is just a bookkeeping operator, its cost is 0.

```
(:method (transport ?p ?o ?d)
  ((package ?p)
   (location ?o)
   (location ?d)
   (truck ?t))
 '((!assert ((tasks (at-truck ?t ?o)
                    (load ?p ?t ?o)
                    (at-truck ?t ?d)
                    (unload ?p ?t ?d))))))
```

To transport package *p* from location *o* to location *d*, create an agenda of subtasks to be accomplished, and insert this agenda into the current state.

```
(:method (do-plan)
  ()
  '((initialize) (do-plan1)))
```

This is the method that the user should call to begin the planning process. It initializes all the protection counters, then starts generating the plan.

```
(:method (initialize)
  (:first (location ?d)
          (not (wait-on-location ?d ?x)))
 '((!assert ((wait-on-location ?d 0)))
   (initialize))
  ()
 '())
```

If there is a `location` predicate in the current state, then initialize its predicate's protection counter to 0, and call the `initialize` method recursively to look for another `location` predicate.

```
(:method (do-plan1)
   ((tasks ?sometasks . ?moretasks))
   '(?sometasks (do-plan1))
   ()
   '())
```

Take the first task off of one of the agendas and execute it, then call the `do-plan1` method recursively to do the next task

```
(:method (at-truck ?t ?d)
  ((tasks (at-truck ?t ?d) . ?moretasks)
   (truck ?t)
   (location ?d)
   (at-truck ?t ?d)
   (wait-on-location ?d ?x))
 `((!remove (tasks (at-truck ?t ?d) . ?moretasks))
   (!assert ((tasks . ?moretasks)))
   (!remove (wait-on-location ?d ?x))
   (!assert ((wait-on-location ?d ,(+ ?x 1)))))
  ((tasks (at-truck ?t ?d) . ?moretasks)
   (truck ?t)
   (location ?d)
   (at-truck ?t ?o)
   (wait-on-location ?d ?x)
   (wait-on-location ?o 0))
 `((!remove (tasks (at-truck ?t ?d) . ?moretasks))
   (!assert ((tasks . ?moretasks)))
   (!move ?t ?o ?d)
   (!remove (wait-on-location ?d ?x))
   (!assert ((wait-on-location ?d ,(+ ?x 1))))))
```

If the first item of some agenda is to move truck *t* to location *d*, then move *t* to *d*, and remove that task from whichever agenda it is in.

```
(:method (load ?p ?t ?o)
  ((tasks (load ?p ?t ?o) . ?moretasks)
```

If the first item of some agenda is a "load" task, then do it, and remove that

```
   (wait-on-location ?o ?x))
 `((!remove (tasks (load ?p ?t ?o) . ?moretasks))
   (!assert ((tasks . ?moretasks)))
   (!load ?p ?t ?o)
   (!remove (wait-on-location ?o ?x))
   (!assert ((wait-on-location ?o ,(- ?x 1))))))
```

task from whichever agenda it is in.

```
(:method (unload ?p ?t ?o)
  ((tasks (unload ?p ?t ?o) . ?moretasks)
   (wait-on-location ?o ?x))
 `((!remove (tasks (unload ?p ?t ?o) . ?moretasks))
   (!assert ((tasks . ?moretasks)))
   (!unload ?p ?t ?o)
   (!remove (wait-on-location ?o ?x))
   (!assert ((wait-on-location ?o ,(- ?x 1))))))
```

If the first item of some agenda is an “unload” task, then do it, and remove that task from whichever agenda it is in.

Table 5. A domain representation for M-SHOP for the simplified logistics domain.

Axioms, operators, and methods	**Descriptions of what they mean**

`(:method (obj-at ?obj ?loc-goal)`
`  ((obj-at ?obj ?loc-now) (truck ?truck))`
` '((delivery ?truck ?obj ?loc-now ?loc-goal))`
	To get *obj* to its goal location, use the *delivery* task.

`(:method (delivery ?truck ?obj ?loc-from ?loc-to)`
`  ((same ?loc-from ?loc-to))`
` '((!do-nothing))`
`  ((in-city ?loc-from ?city)`
`   (truck ?truck ?city))`
` '((truck-at ?truck ?loc-from)`
`   (!load-truck ?obj ?truck ?loc-from)`
`   (truck-at ?truck ?loc-to)`
`   (!unload-truck ?obj ?truck ?loc-to)))`
	If an object's goal location is the same as its source location, then don't do anything (the `!do-nothing` operator). Otherwise, create subtasks of having a truck at the source location, loading the package, having the truck at the destination, and unloading the truck.

`(:method (truck-at ?truck ?loc-to)`
`  ((truck-at ?truck ?loc-from)`
`   (different ?loc-from ?loc-to))`
` '((!drive-truck ?truck ?loc-from ?loc-to))`
`  ()`
` '(!do-nothing))`
	If the truck is not yet at the desired location, then drive the truck to that location from its current location. If the truck is already in the desired location, then don't do anything (the `!do-nothing` operator).

`(:operator (!load-truck ?obj ?truck ?loc)`
`  ((obj-at ?obj ?loc)`
`   (:protection (truck-at ?truck ?loc)))`
`  ((in-truck ?obj ?truck))`
	Operator to load package *p* onto truck *t*.

`(:operator (!unload-truck ?obj ?truck ?loc)`
`  ((in-truck ?obj ?truck)`
`   (:protection (truck-at ?truck ?loc)))`
`  ((obj-at ?obj ?loc))`
	Operator to unload package *p* form truck *t*.

`(:operator (!drive-truck ?truck ?loc-from ?loc-to)`
`  ((truck-at ?truck ?loc-from))`
`  ((truck-at ?truck ?loc-to)`
`   (:protection (truck-at ?truck ?loc-to))))`
	Operator to move truck *t* from location *o* to location *d*.

`(:operator (!do-nothing)`
`  ()`
`  ()`
`  0)`
	The "no-op" operator used in methods m_2 and m_3. The cost of this operator is 0.

`(:- (different ?x ?y)`
`    ((not (same ?x ?y))))`
`(:- (same ?x ?x) nil)`
	Axioms to tell whether two objects are the same or different.

3.2 Definitions

To define M-SHOP, we will use the same definitions that we gave for SHOP in Section 2.1, but with the modifications and additions described below.

A *protection request* and a *protection cancellation* both are expressions of the form:

(:protection *l*)

where *l* is a logical atom.

For M-SHOP, the SHOP definition of an operator is modified as follows (all modifications are underlined). An *operator* is a list having either of the following forms:

(:operator *h D A c*)

where

- h (the operator's *head*) is a primitive task atom;
- D (the operator's *delete list*) is a list of logical atoms and protection cancellations that contain no variable symbols other than those in h;
- A (the operator's *add list*) is a list of logical atoms and protection requests that contain no variable symbols other than those in h.
- c (the operator's *cost*) is a number. The number c may be omitted from the expression, in which case c is taken to be 1.

In the above definition, let A_P be the set of all protection requests in A, and D_P be the set of all protection cancellations in D. Then the “real” additions and deletions in A and D, respectively, are

$A_E = A – A_P$;
$D_E = D – D_P$.

As an example, see the `!drive-truck` and `!load-truck` operators in Table 5. The `!drive-truck` operator adds a protection request for whatever location the truck drives to, and the `!load-truck` operator cancels this request.

A *protection list* is a list L of ground atoms, intended to indicate how many protection requests have been issued but not yet cancelled. Note that L is a list rather than a set: thus for each atom a, if there have been n more protection requests than protection cancellations for a, then there will be n copies of a in L.

For M-SHOP, the SHOP definition of an *applicable* operator is modified as follows (all the modifications are underlined). Let t be a primitive task atom, and o be a planning operator whose head, delete list, add list, and cost are h, D, A, and c, respectively. Suppose there is an mgu u for t and h such that h^u is ground and no atom in L is also in D_E^u. Then o^u is *applicable* to t under L, and its head h is a *simple plan* for t under L. If S is a state, then the state and the protection list produced by executing o^u (or equivalently, h^u) in S under L are

result(S,h^u) = result(S,o^u) = $(S - D_E^u) \cup A_E^u$;
protect(L,h^u) = protect(L,o^u) = append(A_P^u,remove(D_P^u,L)),

where “append” is the Lisp `append` function, and “remove(D_P^u,L)” is the list produced by taking L and removing one copy of each atom in D_P^u.

A *task list* is a list of tasks, just as in SHOP. An *immediate task list* is a list of the form

`(:immediate` t_1 t_2 … t_k`)`

where each t_i is a task. The purpose of the `:immediate` keyword will be explained shortly.

For M-SHOP, a *method* is the same as in SHOP, except that each tail may be either a task list or an immediate task list. A *simple reduction* is the same as in SHOP. If t is a task and S is a state, then a *composite reduction* of t in S is defined as recursively follows. First, every simple reduction of t in S is also a composite reduction of t in S. Second, suppose that $r = (r_1 \ldots r_k)$ is a composite reduction of t in S, and that $q = (q_1 \ldots q_j)$ is a simple reduction of r_1 in S. Then $(q_1 \ldots q_j\, r_2 \ldots r_k)$ is a composite reduction of t in S.

A multi-task list is a list of task lists M = `(`T_1 T_2 … T_1`)` in which at most one of the task lists is an immediate task list. For example, in the simplified logistics domain, the expression

```
(((obj-at p1 L4)) ((obj-at p2 L4)))
```

is a multi-task list comprised of the two task lists `((obj-at p1 L4))` and `((obj-at p2 L4))`. Although each task list T_i in M is an ordered sequence (just as in SHOP), M itself is unordered. For example, if two of the task lists in M are

$T = (t_1 \dots t_{i-1}\ t_i\ t_{i+1} \dots t_m)$,

$U = (u_1 \dots u_{j-1}\ u_j\ u_{j+1} \dots u_n)$,

then this means that the task t_i should be performed after the task t_{i-1} and before the task t_{i+1}, but no particular ordering is required between the tasks t_i and u_j. Thus, for example, it would be permissible for M-SHOP to interleave operators for subtasks of t_i with operators for subtasks of u_j. However, there is one exception: if some task list T in M is an immediate task list, then contains an immediate task list, then M-SHOP should perform the first task in the immediate task list before performing any other task. For example, if two of the task lists in M are

$T =$ `(:immediate` $t_1\ t_2 \dots t_m$`)`,

$U = (u_1\ u_2 \dots u_n)$,

then this means that the task t_1 should be performed before both t_2 and u_1. The following paragraph states these things mathematically.

A *multi-task-list planning problem* (or, for short, a *multi-planning problem*) is a 4-tuple (S,M,L,D), where S is a state, M is a multi-task list, L is a protection list, and D is a domain representation. Suppose that (S,M,L,D) is a multi-planning problem, where M is the multi-task list $(T_1\ T_2 \dots T_m)$. If $P = (p_1\ p_2 \dots p_n)$ is a plan, then we say that P *solves* (S,M,L,D), or equivalently, that P *achieves* M from S in D (we will omit the phrase "in D" if the identity of D is obvious) in any of the following cases:

- **Case 1.** P is empty (i.e., $n = 0$) and every task list in M is empty.
- **Case 2.** M contains an immediate task list $T_i =$ `(:immediate` $t_1\ t_2 \dots t_m$`)`, and there is a composite reduction $R = (r_1 \dots r_j)$ for t_1 in S such that r_1 is a primitive task, p_1 is a simple plan for r_1 under L, and $(p_2 \dots p_n)$ solves the multi-planning problem (S',M',L',D), where

  $S' = \text{result}(S,p_1)$;

  $M' = (T_1 \dots T_{i-1}\ (r_2 \dots r_j\ t_2 \dots t_k)\ T_{i+1} \dots T_m)$;

  $L' = \text{protect}(L,p_1)$.
- **Case 2.** M contains no immediate task list, and there is a task list $T_i = (t_1\ t_1 \dots t_k)$ in M and a composite reduction $R = (r_1 \dots r_j)$ for t_1 in S such that r_1 is a primitive task, p_1 is a simple plan for r_1 under L, and $(p_2 \dots p_n)$ solves the multi-planning problem (S',M',L',D), where

  $S' = \text{result}(S,p_1)$;

  $M' = (T_1 \dots T_{i-1}\ (r_2 \dots r_j\ t_2 \dots t_k)\ T_{i+1} \dots T_m)$;

  $L' = \text{protect}(L,p_1)$.

The biggest difference between this definition and the corresponding definition of Section 2.1 is our use of the composite reduction R in cases 2 and 3. The reason for this is to ensure that all of the simple reductions in R are applicable in the state S. If we had stated the definition using a simple reduction as we did in Section 2.1, this would have made it possible to construct situations where some the reductions were applicable in states prior to S, but not in S itself.

The multi-planning problem (S,M,D) is *solvable* if there is a plan that solves it.

3.3 Algorithm

The M-SHOP planning algorithm is as follows. The subroutine REDUCE(*S,t,D*) finds a composite reduction of the task *t*.

```
procedure M-SHOP($S,M,L,D$)
1.  if every task list in $M$ is empty  then return nil endif
2.  let $T_1,T_2,\ldots,T_n$ be the task lists in $M$
3.  nondeterministically choose a nonempty task list $T_i$
4.  $t$ = the first task in $T_i$
5.  $T'$ = the remaining tasks in $T_i$
6.  $R$ = REDUCE($S,t,D$)
7.  if $R$ = FAIL then return FAIL endif
8.  $r_1$ = the first element of $R$          (comment: note that $r_1$ is a primitive task)
9.  $R'$ = the remaining elements of $R$
10. nondeterministically choose a simple plan $p$ for $r_1$ under $L$
11. $M' = (T_1, T_2, \ldots, T_{j-1}, \text{append}(R',T'), T_{j+1}, \ldots, T_n)$
12. $P$ = M-SHOP(result($S,p$), $M'$, protect($L,p$), $D$)
13. return cons($p,P$)
end M-SHOP
procedure REDUCE($S,t,D$)
1.  if $t$ is primitive then return ($t$)
2.  else if there is no simple reduction of $t$ in $S$ then  return FAIL
3.  else
4.      nondeterministically choose any simple reduction ($r_1 \ldots r_k$) of $t$ in $S$
5.      $R_1$ = REDUCE($S,r_1,D$)
6.      if $R_1$ = FAIL then return FAIL endif
7.      return append($R_1$, ($r_2 \ldots r_k$))
8.   endif
end REDUCE
```

Since the algorithm is a straightforward implementation of the definition of the solution to a multi-planning problem, it is easy to prove that the algorithm is both sound and complete.

Theorem 3 (Soundness of M-SHOP). Suppose one of the nondeterminstic traces of M-SHOP(*S,M,D*) returns a plan *P*. Then *P* solves the multi-planning problem (*S,M,D*).

Proof. The proof is analogous to the proof of Theorem 1. The details are left to the reader. ■

Theorem 4 (Completeness of SHOP) Suppose the multi-planning problem (*S,M,D*) is solvable. Then at least one of the nondeterministic traces of M-SHOP(*S,M,D*) returns a plan.

Proof. The proof is analogous to the proof of Theorem 2. The details are left to the reader. ■

Table 6 shows the first steps of a possible nondeterministic trace of the M-SHOP algorithm on a problem in the simplified logistics domain, using the domain representation of Table 5. In this problem two packages have to be transported from L1 and L2 to L4, using a single truck t1. The multiple task list initially includes two task lists, one for each of the packages.

The selection of the second task list of *M* at step 10 leads to a failure in the next steps. This is because the only first primitive task which can be generated from decomposing '`(obj-at p2 L4)`' is '(!drive-truck t1 L1 L2)', and this action violates the protected predicate '(at-truck t1 L1)'. A plan is returned by M-SHOP returns a plan for this planning problem only when at step 8, the first task list of *M* is selected.

Table 6. M-SHOP's first steps in solving a problem in the simplified logistics domain.

Step	Actions	Result
0	The initial planning problem	`M = (((obj-at p1 L4)) ((obj-at p2 L4)))` `S = I = ((truck-at p1 L3)(obj-at p1 L1)(obj-at p2 L2)`
1	In line 3 of M-SHOP, nondeterministically choose a task T_i to be the task list that will generate the next action of the plan. In line 6 of M-SHOP, call REDUCE to recursively decompose the first task *t* of T_i.	`M =(((obj-at p1 L4)) ((obj-at p2 L4)))` `Ti = ((obj-at p1 L4))` `t = (obj-at p1 L4)`
2	In line 4 of the 1st call to REDUCE, choose a reduction for *t*, and call REDUCE recursively to decompose r_1 - the first task of the reduction.	`t = (obj-at p1 L4)` `reduction = ((delivery t1 p1 L1 L4))` `r1 = (delivery t1 p1 L1 L4)`
3	In line 4 of the 2nd call to REDUCE, choose a reduction for *t*, and call REDUCE recursively.	`t = (delivery t1 p1 L1 L4)` `reduction =((truck-at t1 L1) (!load-truck p1 t1 L1) (truck-at t1 L4) (!unload-truck p1 t1 L4))` `r1 = (truck-at t1 L1)`
4	In line 4 of the 3rd call to REDUCE, choose a reduction for *t*, and call REDUCE recursively.	`t = (truck-at t1 L1)` `reduction = ((!drive-truck t1 L3 L1))` `r1 = (!drive-truck t1 L3 L1)`
5	In line 1 of the 4th call to REDUCE, *t* is primitive. Return (*t*) as the reduction.	`t = (!drive-truck t1 L3 L1)`
6	In line 7 of the 3rd call to REDUCE, append the returned reduction *R* to (r_2 … r_k).	`R = ((!drive-truck t1 L3 L1))` `(r2 … rk) = ()` `append(R,(r2 … rk)) = ((!drive-truck t1 L3 L1))`
7	In line 7 of the 2nd call to REDUCE, append the returned reduction *R* to (r_2 … r_k).	`R = ((!drive-truck t1 L3 L1))` `append(R,(r2 … rk)) = ((!drive-truck t1 L3 L1)) (!load-truck p1 t1 L1) (truck-at t1 L4) (!unload-truck p1 t1 L4))`
8	In line 7 of the 1st call to REDUCE, append the returned reduction *R* to (r_2 … r_k).	`R = ((!drive-truck t1 L3 L1)) (!load-truck p1 t1 L1) (truck-at t1 L4) (!unload-truck p1 t1 L4))` `append(R,(r2 … rk)) = ((!drive-truck t1 L3 L1)) (!load-truck p1 t1 L1) (truck-at t1 L4) (!unload-truck p1 t1 L4))`

9	In line 10 of M-SHOP, choose a simple plan *p* for the first task of the reduction. In Line 12 of M-SHOP, call M-SHOP recursively, with the arguments result(*S*,*p*), M’, and protect(*L*,*p*).	`p = (!drive-truck t1 L3 L1)`
		`result(S,p) = ((truck-at p1 L1)(obj-at p1 L1)`
		`(obj-at p2 L2))`
		`M′ = (((!load-truck p1 t1 L1)`
		`(truck-at t1 L4)`
		`(!unload-truck p1 t1 L4))`
		`((obj-at p2 L4)))`
		`protect(L,p) = ((truck-at p1 L1))`

10	In line 3 of the 2nd call to M-SHOP, nondeterministically choose a task T_i to be the task list that will generate the next action of the plan. In line 14, call REDUCE to recursively decompose the first task *t* of T_i.	`M = (((!load-truck p1 t1 L1)`
		`(truck-at t1 L4)`
		`(!unload-truck p1 t1 L4))`
		`((obj-at p2 L4)))`
		T_i`= ((obj-at p2 L4))`
		`S = ((truck-at p1 L1)(obj-at p1 L1)(obj-at p2 L2)`
		`L = ((truck-at p1 L1))`

4 Experiments

We have tested SHOP and M-SHOP against four other planners: Blackbox, IPP, TLplan, and UMCP:

- Blackbox [Kautz and Selman, 1998] and IPP [Koehler *et al.*, 1997]. These were the two fastest planners in the *AIPS-98* planning competition [McDermott, 1998]. Both of them are action-based planners that make use of Graphplan-style planning graphs.
- TLplan [Bacchus and Kabanza, 2000]. This is an action-based planner that does a forward state-space search guided by statements written in modal logic. It outperformed Blackbox by several orders of magnitude in Bacchus and Kabanza’s tests.
- UMCP [Erol *et al.*, 1994b; Erol *et al.*, 1996]. This planner is an HTN planning system. However, unlike SHOP, it searches a space whose nodes are partially ordered plans rather than totally ordered plans.

For our tests, we used three different domains: logistics, UM Translog, and the blocks world.

4.1 Logistics Problems

The logistics domain [Veloso, 1992] is a popular test domain for action-based planners. In this domain, there are a number of cities, and within each city there may be a number of different locations. Various objects need to be transported from their initial locations to various goal locations. Trucks can be used to transport objects within the same city. Each city has an airport, and airplanes can be used to transport the objects from one airport to another. Since most action-based planners cannot do numeric computations, the problem explicitly ignores all numeric considerations (such as the geometric coordinates of each location or each city, the distance from one location or city to another, fuel consumption, and cost).

4.1.1 Domain Representation for SHOP

In action-based representations of logistics problems, there usually is a “goal formula” that is a list of atoms (g_1 g_2 … g_k), where each g_i is one of the goals. To represent a logistics problem in SHOP, we need to use a task list instead of a goal formula. A task list is a sequence of tasks to accomplish sequentially, but the goals in the goal formula need to be true simultaneously, so to represent them in a task list, we use a task list that contains a single task atom of the form `(goals` g_1 g_2 … g_k`)`.

In order to get SHOP to solve logistics problems, we need to give it a way to keep track of its progress toward accomplishing each goal g_i. To do this, we use a set of axioms, methods, and operators that maintain an "agenda" for each goal g_i that tells what tasks need to be done to accomplish g_i. The logistics domain-description for SHOP is a set of axioms, methods, and operators that basically implement the following algorithm.

First, the algorithm modifies the current state by removing any "useless objects" that will not contribute to the plan. These include packages not mentioned in the goals, and empty trucks and airplanes in the same city with other trucks and airplanes. Then for each goal g_i, the algorithm creates an initial "agenda" that consists solely of g_i and inserts this agenda into the current state. Once this has been done, the actual planning algorithm works as follows:

- If there are no more tasks to perform, end the planning.
- Otherwise, if we need to load or unload a truck, then perform that task, update the agendas for whatever packages have been loaded or unloaded, and apply the algorithm recursively.
- Otherwise, if we need to drive a truck to some location that the truck is already at, then remove that task from its agenda, and apply the algorithm recursively.
- Otherwise, if there is a city that has package needs to be delivered to another city and no airplane in that city, then fly an airplane to that city, update the package's agenda, and apply the algorithm recursively.
- Otherwise, if there is a city that has an airplane, and at least one package in that city needs to be delivered to another city, then fly the airplane to the destination city of the package, update the package's agenda, and apply the algorithm recursively.
- Otherwise, if there is any package on an airplane, then fly the airplane to the destination of that package and unload the package, update the package's agenda, and apply the algorithm recursively.
- Otherwise, if there is any location that has package(s) that need to be picked up, then drive truck to that location pick up the package(s), update their agendas, and apply the algorithm recursively.
- Otherwise, if there is any package in a truck that need to be delivered to some location in the same city, then drive the truck to that location, unload the package, update the package's agenda, and apply the algorithm recursively.

The domain representation also specifies that before flying any airplane, the planner should always collect all the packages in that city that need to be delivered to another city and load them onto the airplane.

4.1.2 Domain Representation for M-SHOP

The logistics domain representation for M-SHOP is much simpler than the one for SHOP. It does not need to maintain agendas inside the current state of the world, because M-SHOP automatically maintains agendas and protection intervals. Thus, instead of specifying an algorithm for transporting all of the packages, we just need to describe how a single package should be transported to its destination.

The logistics domain representation for M-SHOP distinguishes between two cases, one in which the package is transported within the same city, and one in which the package is transported between two different cities. For the task of transporting a package within the same city, it creates a task called the *in-city-delivery* task, which is handled using a method identical to the one for the *delivery* task in the simplified transportation domain (see Table 5). For the task of transporting a package between two cities, there is a method that decomposes this task into the following sequence of subtasks:

1) *in-city-delivery*, which is the task of transporting the package from its source location to an airport in the source city;

2) *air-delivery*, which is the task of transporting the package from the selected airport in the source city to an airport in the goal city;
3) *in-city-delivery*, which is the task of transporting the package from the selected airport in the goal-city to the goal location.

The method for the *air-delivery* task is similar to the method for the *in-city-delivery* task, but uses an airplane instead of a truck.

4.1.3 Experimental Results

We performed tests for the logistics transportation domain on a set of 110 randomly generated problems. The problems involved *N* packages to be delivered, for *N* = 10, 15, …, 60. For each value of *N* we generated 10 problems, for a total of 110 problems. In each problem, the number of cities was no larger than N/2; and each city contained three locations, one truck and *N*/5 or fewer airports. For each package, the original location and the destination location were randomly chosen and were guaranteed to be different from each other. We ran M-SHOP, SHOP, IPP, and TLplan on these problems, using a 167-MHz Sun Ultra with 64 MB of RAM. The results are shown in Figures 1 and 2. No results are shown for Blackbox and IPP, because Blackbox's memory requirements were too high for it to solve logistics problems successfully on our Sun, and IPP was unable to solve any of the problems within a one-hour time limit.

Published results are available data for Blackbox on a set of 30 logistics problems, on a machine that is faster than ours and has 8 GB of RAM.[2] Figures 3 and 4 compare the published results for Blackbox on these problems with the results that we got by running TLplan, M-SHOP, and SHOP on that set of problems using our Sun. No results are shown for IPP, because it was able to solve only four of the problems within a one-hour time limit.

Overall, the best CPU times were obtained with M-SHOP, followed by SHOP and then by TLplan. SHOP was generally about an order of magnitude faster than TLplan, and M-SHOP was about half an order of magnitude faster than SHOP. M-SHOP, SHOP and TLplan found plans of comparable size for both sets of problems.

[2] We got the Blackbox performance data from [Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. "Using Temporal Logics to Express Search Control Knowledge for Planning,." *Artificial Intelligence*, 116(1-2):123-191, January, 2000. . According to Fahiem Bacchus, the data came originally from the Blackbox distribution, and the machine was a Silicon Graphics with 8 GB of RAM, running at around 200 MHz.

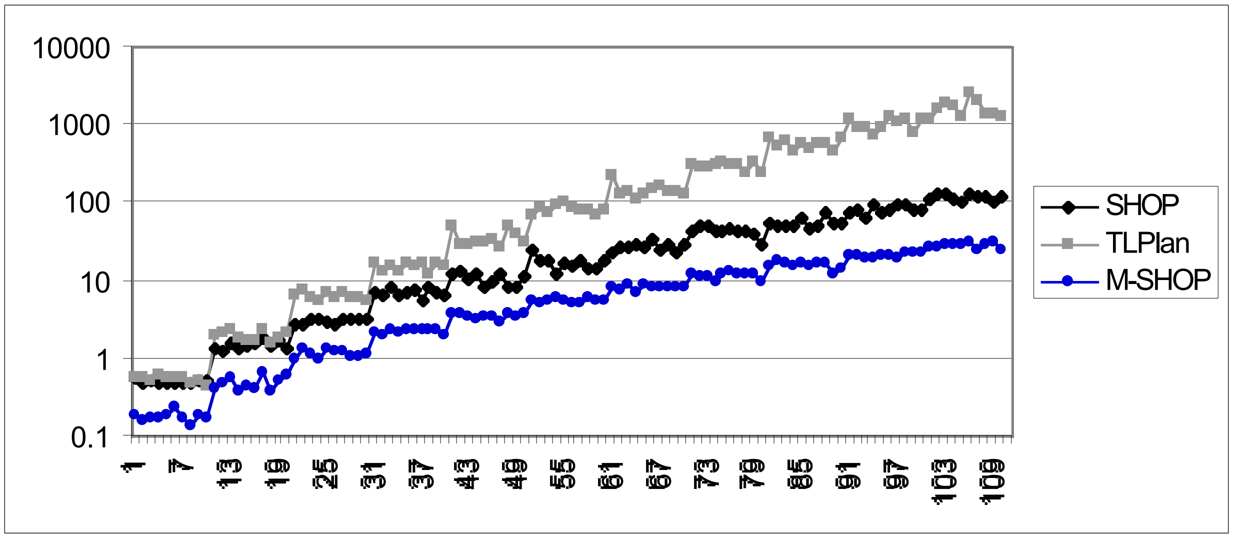

Figure 1. CPU times for each planner on the randomly generated logistics problems. The x-axis gives the problem number, and the y-axis displays the CPU time on a logarithmic scale. Results are not shown for Blackbox and IPP, because Blackbox needed too much memory to run on our computer, and IPP was unable to solve the problems within a one-hour time limit.

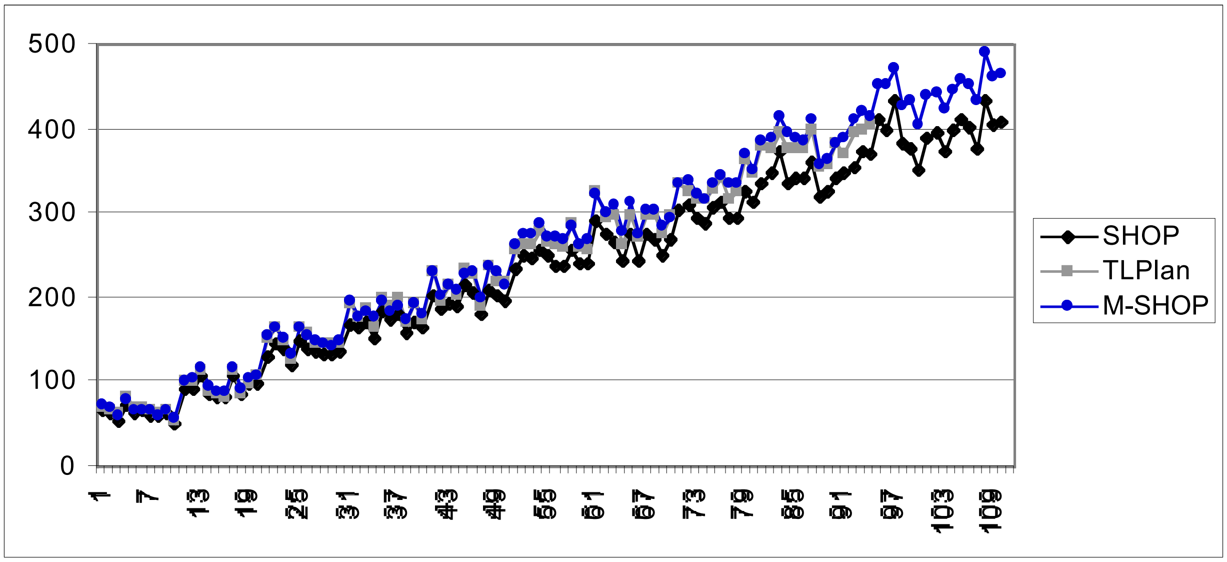

Figure 2. Number of actions in the plans found in Figure 1. The x-axis gives the problem number, and the y-axis gives the number of actions.

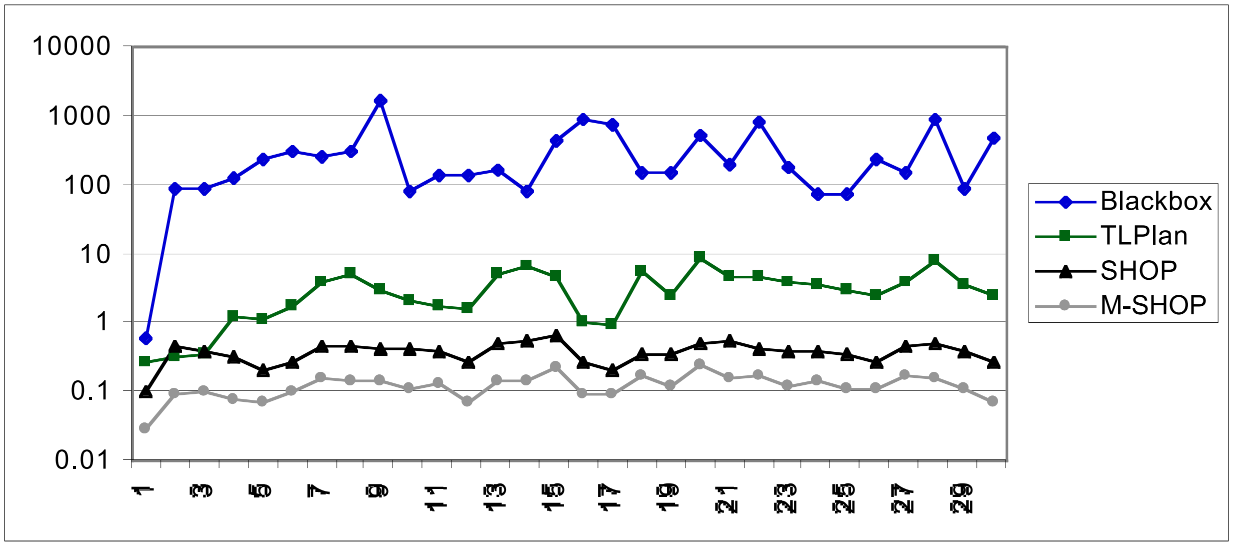

Figure 3. CPU times for each planner on the 30 logistics transportation problems in the TLplan distribution. The x-axis gives the problem number, and the y-axis displays the CPU time on a logarithmic scale. Results are not shown for IPP because it was unable to solve any of the problems within a one-hour time limit. The results for Blackbox are taken from [Bacchus and Kabanza, 2000].

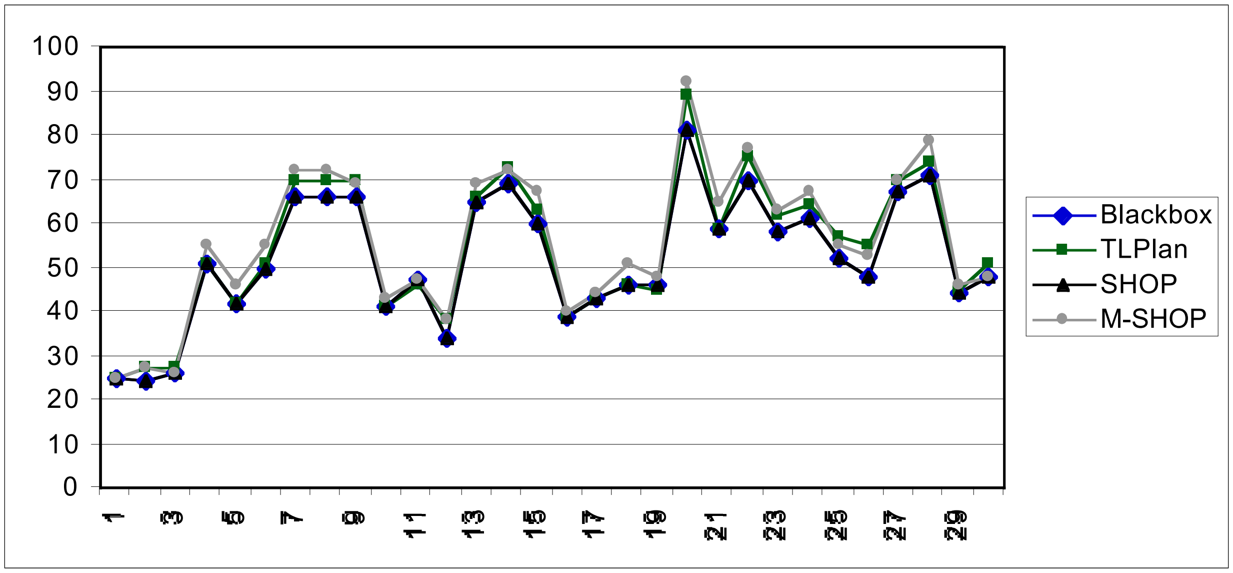

Figure 4. Number of actions in the plans found in Figure 3. The x-axis gives the problem number, and the y-axis gives the number of actions.

4.2 UM-Translog

UM Translog [Andrews *et al.*, 1995; Group, 1995] is a large HTN planning domain that was inspired by the well known CMU logistics domain [Veloso, 1992]. However, the UM Translog domain is about an order of magnitude larger than the logistics domain, as it deals with various types of packages, vehicles, and procedures for handling the packages.

4.2.1 Domain Representations for SHOP and M-SHOP

The UM-Translog domain representation for SHOP consists of 43 operators and 66 methods. Just as with the logistics domain, one of the basic ideas we used in the domain representation was to tell SHOP to maintain (as part of the current state) an "agenda" for each package that represents what tasks need to be done to that package. This agenda-manipulation is done as follows. Initially, SHOP starts with an empty plan. Then it checks to see if all of the agendas are empty. If they are, then the planning process finishes and the solution plan is returned. Otherwise, SHOP selects one of the agendas and remove the first task from it. If this task is an operator, then SHOP performs this operator, removes it from the agenda, and appends it to the solution plan. Otherwise, the task is replaced by its decomposition and the process continues recursively.

Below is a high-level description of the methods used for SHOP and M-SHOP for the UM Translog domain. SHOP also needs some additional methods to maintain the agendas as described above, but those methods are not needed in M-SHOP because the M-SHOP algorithm keeps track of this information automatically.

- Each top-level task is always of the form `(transport ?p ?o ?d)`, indicating that the package ?p needs to be to transported from location `?o` to location `?d`. If `?o` and `?d` are the same location, then the method for this task simply removes it. Otherwise, there is a method that decomposes it into three subtasks:

  9. `(pickup ?p)`: pick up the package.
  10. `(carry ?p ?o ?d)`: carry the package from `?o` to `?d`.
  11. `(deliver ?p)`: deliver the package.

- There is a method that decomposes the task (pickup ?p) into two subtasks:

  12. `(handle-insurance ?p)`: check whether the package needs insurance, and if it does, then collect the insurance for the package.
  13. `(handle-hazardous ?p)`: check whether the package is hazardous, and if so, get a permit for the package.

- There are methods that produce ten different possible decompositions for the task `(carry ?p ?o ?d)`, depending on the values of `?o` and `?d`.
- There is a method for the task `(deliver ?p)`, that simply executes an operator to deliver the package.

4.2.2 Experimental Results

We compared M-SHOP, SHOP and UMCP on UM Translog domain. It was not feasible for us to run Blackbox, IPP, and TLplan on the UM Translog domain, because UM Translog contains a number of HTN constructs that are difficult to translate into the action-based representations used by those planners (for a description of some of the difficulties involved, see [Lotem and Nau, 2000]).

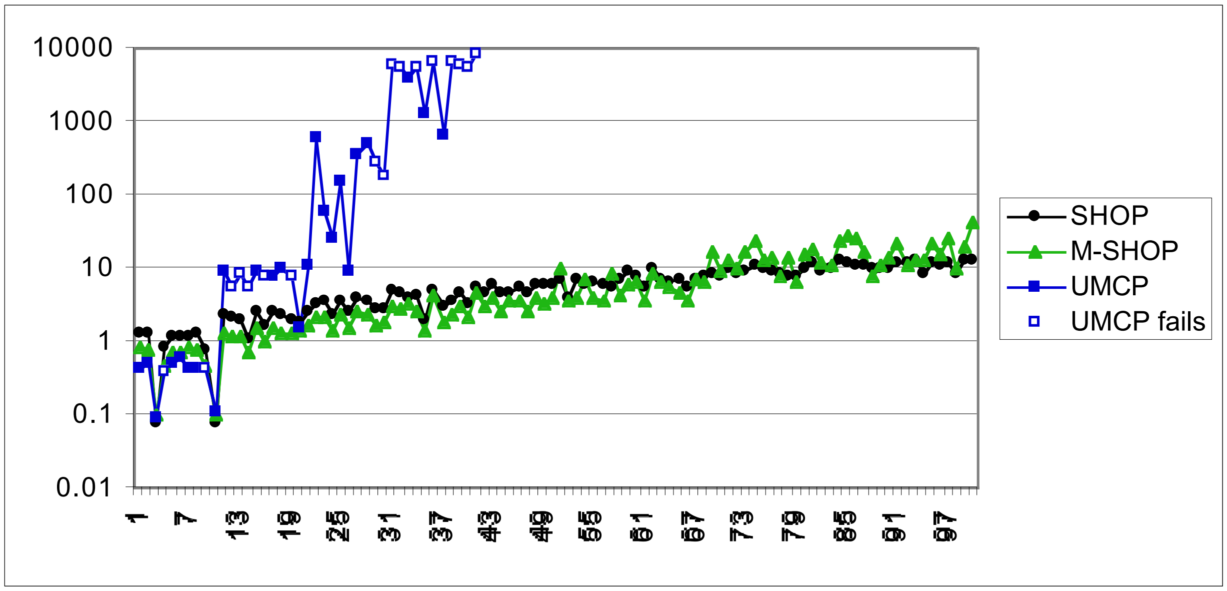

Figure 5. CPU times for each planner on the UM Translog problems. The x-axis gives the problem number, and the y-axis displays the CPU time on a logarithmic scale. Results are not shown for Blackbox, IPP, and TLplan because of the difficulty of translating the UM Translog domain into an action-based domain representation. The nodes labeled "UMCP fails" are cases where UMCP failed to find an answer (either it ran out of memory or time before finding an answer, or else it simply returned failure).

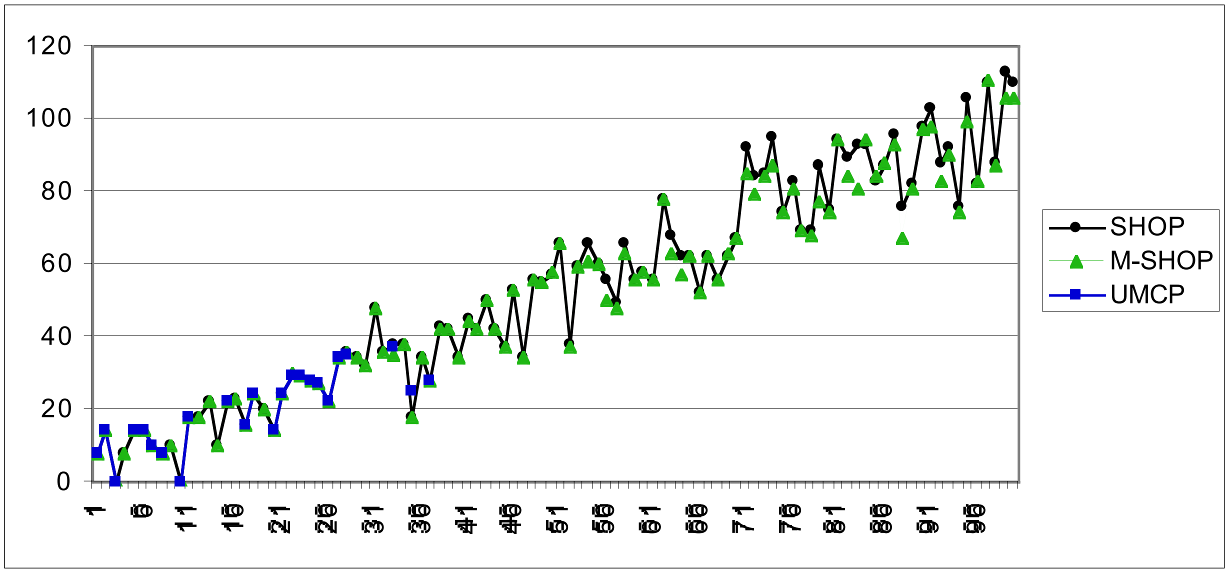

Figure 6. Number of actions in the plans found in Figure 5. The x-axis gives the problem number, and the y-axis gives the number of actions.

For this comparison, we randomly generated 100 problems in the UM Translog domain. The problems consisted of 10 one-goal problems, 10 two-goal problems and so forth. Each goal was to relocate a package. The original location and destination location of each package was randomly chosen and could possibly be the same as the final destination. Also, the type of the package was randomly chosen from *bulky*, *liquid*, *granular*, and *mail*. In addition, each problem involved five connected cities, fifteen locations, and eleven trucks in one location in the initial state.

We ran M-SHOP, SHOP and UMCP on these problems, using a 167-MHz Sun Ultra with 64 MB of RAM. The results are shown in Figures 5 and 6. SHOP and M-SHOP have comparable CPU times and number of actions, and the CPU time for UMCP is several orders of magnitude larger than SHOP and M-SHOP. UMCP failed on 17 of the first 41 problems (either it ran out memory, it failed to find an answer within our time limit, or it simply returned with failure), and for these problems, the CPU time shown in Figure 5 was the time when UMCP was stopped. By the time we reached problem 41, UMCP was taking excessive amounts of real time (often several days) on each problem due to swapping, so we did not attempt to run UMCP on the last 69 problems.

4.3 Blocks-World Planning

The blocks-world planning problem is widely known in the AI planning literature [Nilsson, 1980; Gupta and Nau, 1992], primarily because they appear to capture several of the relevant difficulties posed to planning systems.

4.3.1 Domain Representation for SHOP

To run SHOP in the blocks world, we created a domain description that encoded the blocks-world planning algorithm of [Gupta and Nau, 1992]. A copy of the encoding is available as part of the SHOP software distribution at <http://www.cs.umd.edu/projects/shop>. The algorithm is guaranteed to find near-optimal blocks-world plans in low-order polynomial time.

The algorithm requires the planner to be able to reason about a block's *position*, which is the entire stack of blocks underneath the block. If the logical atoms describing a block's position are not consistent with the atoms in the goal state, then the block will need to be moved at some point during the planning process. Otherwise, the block will not need to be moved at all. The algorithm is roughly as follows:

```
loop
    Current state ← initial state
    if the current state satisfies the goal conditions then exit
    else if there is a clear block b whose position is inconsistent with the goal conditions, such
            that b can be moved immediately to a position consistent with the goal conditions
    then
        move it
    else
        choose a clear block whose position whose position is inconsistent with the goal
            conditions (this is a nondeterministic choice)
        move the block to the table
    endif
repeat
```

To encode this algorithm as a SHOP domain representation, we again told SHOP to assert atoms into the current state of the world to represent what goals still need to be achieved. It was somewhat simpler to do this for the blocks world than it was for the logistics and UM Translog domains, because in the blocks world, each "agenda" consists just of a single goal (rather than a complex task that needs to be decomposed into subtasks).

4.3.2 Domain Representation for M-SHOP

For M-SHOP, we used two domain representations:

- The first one was the exactly the same domain representation that we used for SHOP; we did this as a check to see how much overhead would be introduced by the M-SHOP algorithm. This domain representation is labeled M-SHOP (S) in Figures 7 and 8.
- The second domain representation, which is labeled M-SHOP (M) in Figures 7 and 8, made use of M-SHOP's ability to maintain multiple agendas automatically. In this domain representation, there are three kinds of goals, each of which is handled separately as follows:
  1. `(on ?x ?y)`, i.e., the block `?x` needs to be on the block `?y`. If `?x` is on `?y` and `?y` doesn't need to be moved, then this condition is protected as the goal has been achieved. Otherwise, if `?x` is clear and `?y` is clear, and `?y` doesn't need to be moved, `?x` is moved onto `?y` and this condition is then protected as the goal has been achieved. Otherwise, a random selection is made for a block ?z that needs to be moved, it is moved onto the table and the process is continued recursively
  2. `(on-table ?x)`, i.e., the block `?x` needs to be on the table. If `?x` is already on the table, this condition is protected. Otherwise, if `?x` is clear, it is moved onto the table and this condition is then protected. Otherwise, a random selection is made for a block ?z that needs to be moved, it is moved onto the table and the process is continued recursively.
  3. `(clear ?x)`, i.e., the block `?x` needs to be clear. If `?x` is clear, this condition is protected. Otherwise, a random selection is made for a block ?z that needs to be moved, it is moved onto the table and the process is continued recursively.

4.3.3 Exper imental Results

100 problems in the blocks world were randomly generated in groups of 5 problems, each consisting of N= 5, 10, …, 100 blocks to be relocated. To build the initial and final states, we generated configurations of blocks as follows:

- First, put a block onto the table (thereby creating a new tower).
- For each block after the first one, if *t* is the number of existing towers, then there are *t*+1 different possible places to put the new block: on top of any of the existing towers, or on the table (thereby creating a new tower). Choose one of those locations at random, with an equal probability for each choice.

We ran M-SHOP (S), M-SHOP (M), SHOP, TLplan, and IPP on these problems. We did not run Blackbox on these problems because it needed more memory than we had available. As shown in Figure 7, SHOP and M-SHOP (S) did best (with virtually identical performance), TLplan was next, M-SHOP (M) was next after TLplan, and IPP was unable to solve any of the problems within a one-hour time limit. As shown in Figure 8, all of the planners (except for Blackbox and IPP) generated plans having similar numbers of actions, but the number of actions was slightly better for M-SHOP (S) and SHOP than for TLplan and M-SHOP (M).

The reason why M-SHOP (M) didn't do as well as SHOP and M-SHOP (S) is because at each point in the planning process, it was important to make a good choice of which goal to perform next. In SHOP and M-SHOP (S), our domain representation included a good way to select the next goal. However, in M-SHOP (M), the job of selecting the next goal was left up to M-SHOP's agenda-selection mechanism, and there was no way to tell it which choices were the better ones.

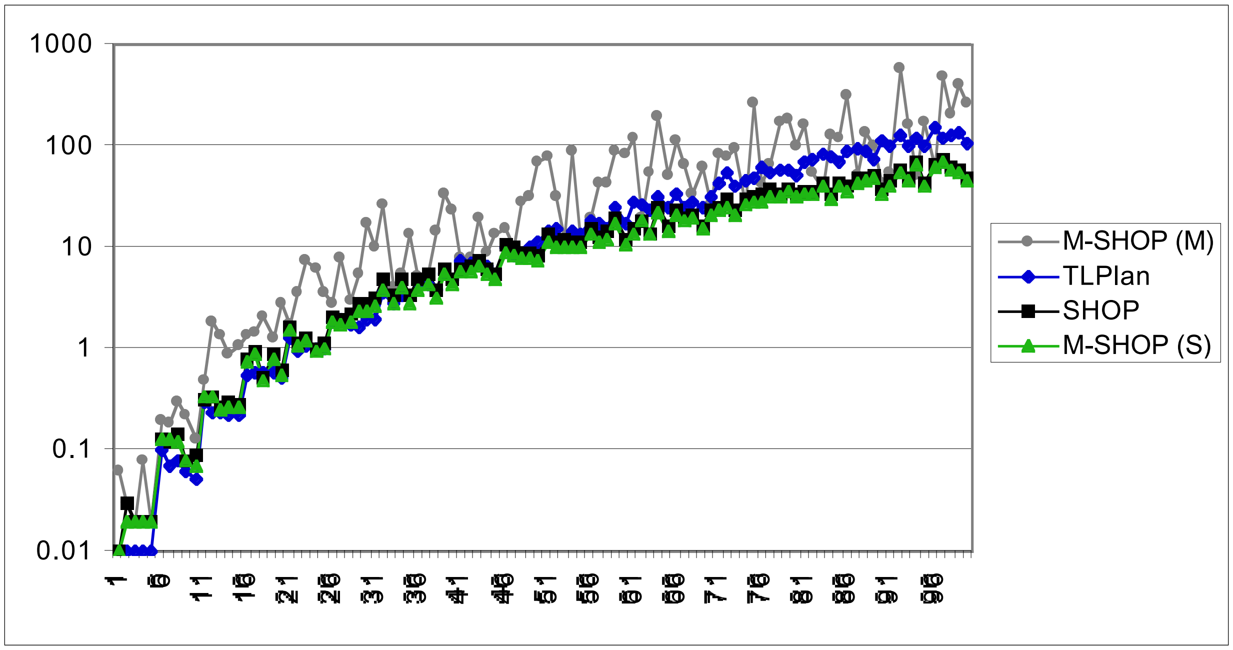

Figure 7. CPU times for each planner on the blocks-world problems. The x-axis gives the problem number, and the y-axis displays the CPU time on a logarithmic scale. Results are not shown for Blackbox because it required more memory than was available on our computer. Results are not shown for IPP because it was unable to solve any of the problems on our computer within a one-hour time limit. The values for SHOP and M-SHOP (S) are nearly identical.

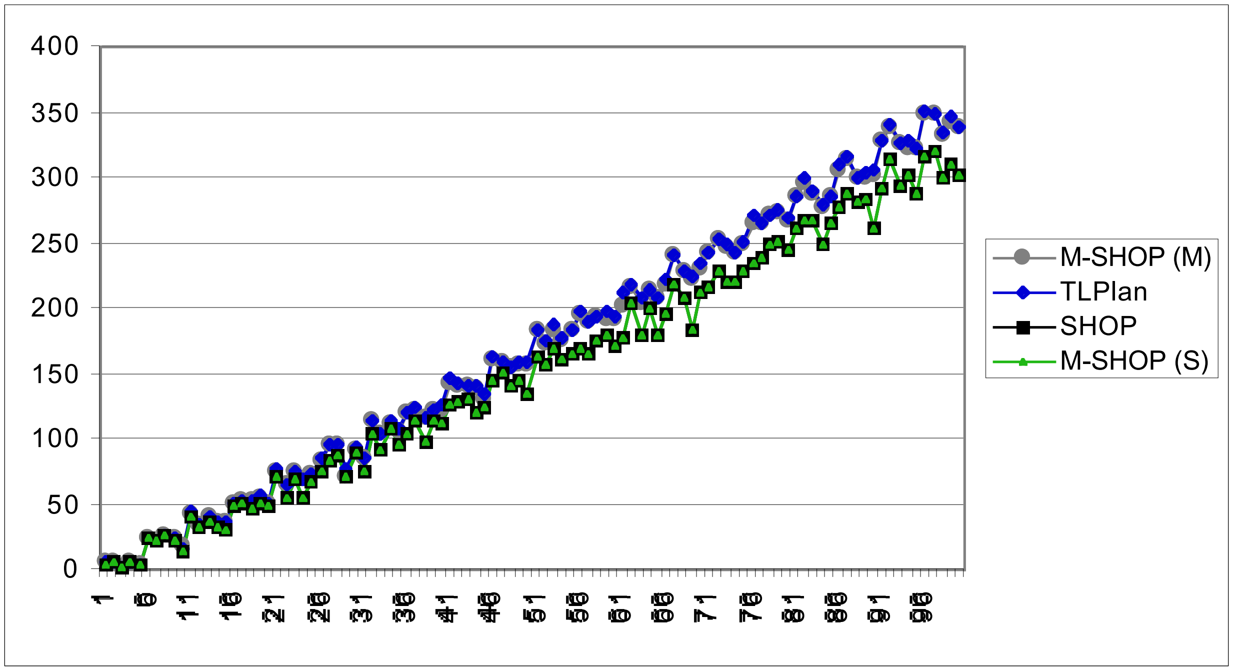

Figure 8. Number of actions in the plans found in Figure 7. The x-axis gives the problem number, and the y-axis gives the number of actions. The values for SHOP and M-SHOP (S) are identical, and the values for TLPlan and M-SHOP (M) are nearly identical.

5 Discussion and Conclusions

In this paper, we have described the idea of ordered task decomposition, and its implementation in the SHOP and M-SHOP planning systems. Since SHOP and M-SHOP plan for tasks in the same order in which those tasks will be executed, they always know the complete world-state at each step of the planning process. Thus, SHOP and M-SHOP can incorporate a high degree of expressivity in their domain representations, including Horn-clause inferencing, numeric computations, and calls to arbitrary external programs. Lisp implementations of SHOP and M-SHOP are available as freeware at <http://www.cs.umd.edu/projects/shop>, under the terms of the GNU General Public License. A Java implementation of SHOP is also being developed.

The expressive power of SHOP and M-SHOP can be used to create domain representations that encode highly efficient planning procedures. In our tests on blocks-world and logistics problems, SHOP and M-SHOP were several orders of magnitude faster than Blackbox, IPP, and UMCP, and were several times faster than TLplan, even though SHOP and M-SHOP are coded in Lisp and the other planners (except for UMCP) are in C.

Furthermore, the expressive power of the approach makes it powerful enough to be used in complex real-world planning problems. For example, in a joint effort with researchers at the US Naval Research Laboratory [Munoz-Avila *et al.*, 1999], we are using JSHOP (the Java implementation of SHOP) as part of HICAP, a plan-authoring system for noncombatant evacuation operations (NEOs). NEOs are military evacuation operations that require performing hundreds of subtasks and whose primary goal is to minimize loss of life. Formulating a NEO plan is a complex task because it involves considering a wide range of factors (e.g., military resources, political issues, meteorological predictions) and uncertainties (e.g., hostility levels and locations), and because flawed NEO plans could yield dire consequences.

M-SHOP generalizes the SHOP planning algorithm by allowing the initial task specification to be unordered, and by automatically maintaining protection conditions and lists of subtasks for those tasks. This gives M-SHOP an advantage over SHOP in domains where it is natural to specify how to carry out individual tasks without reference to other tasks that might need to be achieved. For example, in the logistics and UM-Translog domains, the M-SHOP methods describe how to transfer individual packages from one location to another, and when several packages need to be transferred, it is M-SHOP's responsibility to combine the plans for those packages into a single plan. The same problem domains can also be represented in SHOP, but at the price of introducing special-purpose methods, operators, and predicates that emulate the operation of M-SHOP. The trade-offs involved in using such special-purpose domain elements are as follows:

- In cases where it is natural to represent the subtasks for one task without reference to the subtasks for another task, it is better to let M-SHOP use its own automatic agenda-manipulation abilities. In the logistics domain and the UM Translog domain, this made the domain representations much simpler (and thus easier to debug), and it also resulted in more efficient planning: M-SHOP outperformed SHOP on the logistics domain, and performed similarly to SHOP on the UM Translog domain.
- In some problem domains, it may be necessary to reason about dependencies among the subtasks for different tasks. In such cases, we can get greater efficiency if we hand-code the agenda manipulations rather than depending on M-SHOP to do them for us, because this gives us the ability to reason about the agendas in a global manner. One example of this occurs in the blocks-world domain, in which the domain algorithm [Gupta and Nau, 1992] needs to be capable of giving higher priority to tasks involving blocks that can be moved directly to their final positions, and lower priority to tasks involving blocks that cannot be moved directly to their final positions. Here, the domain

representation that hand-coded the agenda manipulations performed significantly better than the one that depended on M-SHOP to do them.

- Even if we do not want to make use of M-SHOP's additional capabilities, there appears to be no performance penalty for using the M-SHOP algorithm rather than the SHOP algorithm. For example, our blocks-world domain representation for SHOP ran equally efficiently in both SHOP and M-SHOP.

It did not particularly surprise us that SHOP and M-SHOP did so much better than Blackbox, IPP, and UMCP, because SHOP and M-SHOP have so much more expressive power than those planners. However, it did surprise that SHOP and M-SHOP did so much better than TLplan. Like SHOP and M-SHOP, TLplan knows the current state of the world at each point in its planning process, and TLplan's modal-logic representation makes it possible to write some very sophisticated pruning axioms. Furthermore, TLplan is written in C, which is a faster language than Lisp. Thus, when we first began to test SHOP, we had not expected it to do as well against TLplan as it actually did.

In this regard, it is interesting to note that subsequent to the experiments reported in this paper, a planner called TALplanner [Doherty and Kvarnström, 1999] outperformed SHOP in the AIPS-2000 planning competition. TALplanner is a planning system that is based on TLplan, but it uses a different temporal-logic representation that incorporates substantial optimizations to the data structures. We suspect that these data-structure optimizations are the primary reason why TALplanner outperformed SHOP. For example, while the planning competition was in progress, we discovered that a simple change to the data structure SHOP uses to represent its world-states would speed SHOP up by about an order of magnitude on large problems.

Since SHOP, M-SHOP, TLplan, and TALplanner all are total-order forward-search planners, the results suggest that total-order forward search can "scale up" to complex planning problems better than partial-order planning. Our results also illustrate the impact that planning applications can have on planning theory, since the SHOP and M-SHOP algorithms evolved from our previous domain-specific work on manufacturing planning and computer bridge.

Our ongoing and future work is as follows:

- We are starting to make optimizations to SHOP's data structures, as described above. We believe that this will speed up SHOP by several orders of magnitude.
- For use in the evacuation-planning project mentioned above [Munoz-Avila *et al.*, 1999; Munoz-Avila *et al.*, 2000], we intend to extend SHOP to incorporate ways to reason about time, and reason about uncertainty, generate and evaluate contingency plans, and react to new information that comes in from external programs.
- We have begun integrating SHOP with the IMPACT [Eiter and Subrahmanian, 1999; Eiter *et al.*, 1999] multi-agent architecture, to provide planning in a multi-agent environment. We have developed the theoretical foundations for this integration [Dix *et al.*, 2000], and are beginning to develop an implementation.

Acknowledgements

This work was supported in part by the following grants and contracts: Army Research Laboratory DAAL01-97-K0135, Naval Research Laboratory N00173981G007, Air Force Research Laboratory F306029910013 and F30602-00-2-0505, and NSF DMI-9713718.

References

[Andrews *et al.*, 1995] S. Andrews, B. Kettler, K. Erol and J. Hendler. "UM Translog: A Planning Domain for the Development and Benchmarking of Planning Systems." Tech. Report CS-TR-3487, Dept. of Computer Science, University of Maryland, College Park, MD, 1995.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. "Using Temporal Logics to Express Search Control Knowledge for Planning,." *Artificial Intelligence*, 116(1-2):123-191, January, 2000.

[Bacchus and Kabanza, 1996] F. Bacchus and K. Kabanza. "Using temporal logic to control search in a forward chaining planner." In M. G. a. A. Milani, Ed., *New Directions in Planning.* IOS Press, 1996, pp. 141–153.

[Baral and Subrahmanian, 1993] C. Baral and V. S. Subrahmanian. "Dualities Between Alternative Semantics for Logic Programming and Non-Monotonic Reasoning." *Journal of Automated Reasoning*, 10:399-420, 1993.

[Bonet and Geffner, 1999] B. Bonet and H. Geffner. Planning as Heuristic Search: New Results. In *Proc. European Conference on Planning (ECP-99)*, 1999. Durham, UK: Springer-Verlag.

[Currie and Tate, 1991] K. Currie and A. Tate. "O-Plan: The Open Planning Architecture." *Artificial Intelligence*, 52(1):49-86, 1991.

[Dix *et al.*, 2000] J. Dix, H. Munoz and D. Nau. IMPACTing SHOP: Planning in a Multi-Agent Environment. In *CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA-00)*, 2000. London.

[Doherty and Kvarnström, 1999] P. Doherty and J. Kvarnström. TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner. In *Proceedings of the 6th Int'l Workshop on the Temporal Representation and Reasoning (TIME'99).*, 1999. Orlando, FL.

[Eiter and Subrahmanian, 1999] T. Eiter and V. S. Subrahmanian. "Heterogeneous Active Agents, II: Algorithms and Complexity." *Artificial Intelligence*, 108(1-2):257-307, 1999.

[Eiter *et al.*, 1999] T. Eiter, V. S. Subrahmanian and G. Pick. "Heterogeneous Active Agents, I: Semantics." *Artificial Intelligence*, 108(1-2):179-255, 1999.

[Erol *et al.*, 1996] K. Erol, J. Hendler and D. Nau. "Complexity Results for Hierarchical Task-Network Planning." *Annals of Mathematics and Artificial Intelligence*, 18:69-93, 1996.

[Erol *et al.*, 1994a] K. Erol, J. Hendler and D. S. Nau. "Semantics for Hierarchical Task-Network Planning." Tech. Report CS TR-3239, UMIACS TR-94-31, ISR-TR-95-9, University of Maryland, March, 1994a.

[Erol *et al.*, 1994b] K. Erol, J. Hendler and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *Proc. Second International Conf. on AI Planning Systems (AIPS-94)*, June, 1994b, pages 249-254.

[Fink and Veloso, 1995] E. Fink and M. Veloso. Formalizing the Prodigy planning algorithm. In *Proc. European Workshop in AI Planning (EWSP-95)*, 1995.

[Group, 1995] Parallel Understanding Systems Group. The UM Translog Planning Domain. <http://www.cs.umd.edu/projects/plus/UMT>, 1995.

[Gupta and Nau, 1992] N. Gupta and D. S. Nau. "On the Complexity of Blocks-World Planning." *Artificial Intelligence*, 56(2-3):223-254, August, 1992.

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

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference of the American Association for Artificial Intelligence*, 1996, pages 1194-1201.

[Kautz and Selman, 1998] H. Kautz and B. Selman. Blackbox: A SAT-technology planning system. <http://www.research.att.com/~kautz/blackbox>, 1998.

[Kautz and Selman, 1999] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 17th National Conference of the American Association for Artificial Intelligence (IJCAI-99)*, 1999, pages 318-325.

[Koehler *et al.*, 1997] J. Koehler, B. Nebel, J. Hoffman and Y. Dimopoulus. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, 1997. Toulouse, France.

[Lotem and Nau, 2000] A. Lotem and D. Nau. New Advances in GraphHTN: Identifying Independent Subproblems in Large HTN Domains. In *AIPS-2000*, 2000. To appear

[McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. AI*, July, 1991, pages 634-639.

[McDermott, 1998] D. McDermott. AIPS-98 Planning Competition Results. <http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html>, 1998.

[Munoz-Avila *et al.*, 1999] H. Munoz-Avila, D. Aha, L. Breslow and D. Nau. HICAP: an interactive case-based planning architecture and its application to noncombatant evacuation operations. In *IAAI-99*, 1999, pages 870-875.

[Munoz-Avila *et al.*, 2000] H. Munoz-Avila, D. W. Aha, L. A. Breslow, D. S. Nau and R. Weber. Integrating Conversational Case Retrieval with Generative Planning. In *EWCBR-2000*, 2000. Trento, Italy: Springer-Verlag.

[Nau *et al.*, 1998] D. S. Nau, S. J. J. Smith and K. Erol. Control Strategies in HTN Planning: Theory versus Practice. In *AAAI-98/IAAI-98 Proceedings*, 1998, pages 1127-1133.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[Penberthy and Weld, 1992] J. S. Penberthy and D. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proc. KR-92*, 1992.

[Smith *et al.*, 1997] S. J. Smith, K. Hebbar, D. Nau and I. Minis. "Integrating Electrical and Mechanical Design and Process Planning." In M. Mantyla, S. Finger and T. Tomiyama, Eds., *Knowledge Intensive CAD, Volume 2.* Chapman and Hall, 1997, pp. 269-288.

[Smith *et al.*, 1998] S. J. J. Smith, D. S. Nau and T. Throop. "Computer Bridge: A Big Win for AI Planning." *AI Magazine*, 19(2):93-105, June, 1998.

[Subrahmanian, 1999] V. S. Subrahmanian. "Nonmonotonic Logic Programming." *IEEE Transactions on Knowledge and Data Engineering*, 11(1):143-152, Jan/Feb, 1999.

[Tate, 1977] A. Tate. Generating Project Networks. In *Proc. IJCAI-77*, 1977, pages 888-893.

[Tate, 1994] A. Tate. Mixed Initiative Planning in O-Plan2. In *Proceedings of the ARPA/Rome Laboratory Knowledge-Based Planing and Scheduling Initiative*, 1994, pages 512-516. Tuscon, AR: Morgan Kaufmann.

[Veloso, 1992] M. Veloso. "Learning by Analogical Reasoning in General Problem Solving." Tech. Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Veloso and Blythe, 1994] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *AIPS-94*, 1994.

[Wilkins, 1990] D. Wilkins. "Can AI Planners Solve Practical Problems?" *Computational Intelligence*, 6(4):232-246, 1990.

[Wilkins, 1988] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.