Technical Report CS-TR-3981, UMIACS-TR-9904 To appear, *IJCAI-99*

SHOP: Simple Hierarchical Ordered Planner

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) is a domain-independent HTN planning system with the following characteristics.

- SHOP plans for tasks in the same order that they will later be executed. This avoids some goal-interaction issues that arise in other HTN planners, so that the planning algorithm is relatively simple.
- Since SHOP knows the complete world-state at each step of the planning process, it can use highly expressive domain representations. For example, it can do planning problems that require complex numeric computations.
- In our tests, SHOP was several orders of magnitude faster than Blackbox and several times faster than TLplan, even though SHOP is coded in Lisp and the other planners are coded in C.

1 Introduction

"Conventional wisdom" in AI planning holds that total-order forward search is a bad idea because it causes excessive backtracking. However, several groups of researchers have begun to argue that the opposite is true: that total-order forward-search allows planners to use a 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 [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 *et al*., 1991].
- TLplan [Bacchus and Kabanza, 1996, 1998] does a forward state-space search, using modal-logic axioms 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].
- Smith *et al.* [1997, 1998] developed an approach that combines HTN-style problem reduction with left-to-right backtracking to produce a search strategy similar to Prolog's. They used this approach 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 their approach head-to-head against domain-independent planning algorithms, because their implementations were domain-specific.

In order to test the performance of Smith *et al.*'s approach in a domain-independent setting, we have created a domain-independent formalization of the approach, and have implemented it in a planner called SHOP (Simple Hierarchical Ordered Planner). SHOP is available at <http://www.cs.umd.edu/projects/shop>, under the terms of the GNU General Public License. SHOP has the following characteristics:

1. SHOP plans for tasks in the same order that they will be executed. By avoiding some task-interaction issues, this makes SHOP 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.*, 1994]. It also makes it easier to prove soundness and completeness results.
2. Since SHOP always knows the complete world-state at each step of the planning process, it can use considerably more expressivity in its domain representations than most AI planners. For example, SHOP has the ability to do Horn-clause inferencing, numeric computations, and interactions with external agents and external information sources.
3. SHOP's expressive power can be used to create highly efficient domain representations. In our tests on blocks-world and logistics problems, SHOP was several orders of magnitude faster than Blackbox and several times faster than TLplan, even though SHOP

is coded in Lisp and the other planners are in C.

2 Formal Definitions

This section defines the syntax and semantics used in SHOP, as well as the SHOP planning algorithm. For brevity, the definitions below are for a somewhat simplified version of SHOP's syntax and semantics. Section 3 gives an informal overview of the additional features that appear in the full syntax and semantics. For a formal description of those features, see <www.cs.umd.edu/projects/shop/documentation.html>.

2.1 Syntax

We use the usual first-order-logic definitions of variable and constant symbols, function and predicate symbols, terms, atoms, conjuncts, most-general unifiers (mgu's), and Horn clauses; with the notation adapted for Lisp. For example, here are two Horn clauses, first in Prolog notation and then in our notation:

```
p(f(X) :- q(X,c), r(Y,d), s(d).
q(b,c).
```

```
(:- (p (f ?x)) ((q ?x c) (r ?y d) (s d)))
(:- (q b c) nil)
```

A *state* is a set of ground atoms, and an *axiom set* is a set of Horn clauses. If S and is a state and X is an axiom set, then $S \cup X$ *satisfies* a conjunct C if there is a substitution u (called a *satisfier*) such that $S \cup X$ entails C^u. u is a *most general satisfier* (or *mgs*) if there is no other satisfier v more general than u. In contrast to mgu's (which are unique modulo lexical renaming), there may be several distinct mgs's for C from S and X.

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 $t_1, t_2, \ldots, t_n$ (the task's *arguments*) are terms. The task is *primitive* if s is a *primitive* task symbol (a symbol whose first character is an exclamation point) and it is *compound* if s is a *compound* task symbol (a symbol whose first character is not a special character). A *task list* is a list of tasks.

An *operator* is a expression (`:operator` $h\ D\ A$), where h (the *head*) is a primitive task, and D and A (the *deletions* and *additions*) are sets of atoms containing no variable symbols other than those in h. For example, here is an operator to put a block on the table:

```
(:operator (!putdown ?block)
  ((holding ?block))
  ((ontable ?block) (handempty)))
```

A *method* is an expression that has 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. For example, here is a pair of methods for clearing the top of a block:

```
(:method (make-clear ?y) ((clear ?y)) nil)
(:method (make-clear ?y)
  ((on ?x ?y))
  ((make-clear ?x)
   (!unstack ?x ?y) (!putdown ?x))
```

The first method says that if y is already clear we should do nothing; the second says that if another block x is on y, we should make x clear and then move x to the table.

2.2 Semantics

The intent of an operator o = (`:operator` $h\ D\ A\ c$) 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 o is *applicable* to t, and the list (h^u) is a *simple plan* for t. If we execute this plan in some state S, it produces the state $h^u(S) = o^u(S) = (S - D^u) \cup A^u$.

The intent of a method m = (`:method` $h\ C\ T$) is to specify that if the current state of the world satisfies C, then h can be accomplished by performing the tasks in T in the order given. More specifically, let S be a state, X be an axiom set, and t be a task atom. Suppose there is an mgu u that unifies t with h, and suppose $S \cup X$ satisfies C^u. Then m is *applicable* to t in $S \cup X$, and the result of applying m to t is the set of task lists $R = \{(T^u)^v : v$ is an mgs for C^u from $S\}$. Each task list r in R is a *simple reduction* of t by m in $S \cup X$.

A *plan* is a list of heads of ground operator instances. If p is a plan and S is a state, then $p(S)$ is the state produced by starting with S and executing the operator instances in the order that their heads appear in p.

A *planning problem* is a tuple $P = (S,T,D)$, where S is a state, T is a task list, and D is a set of axioms, operators, and methods. If (S,T,D) is a planning problem, then $\Pi(S,T,D)$, the set of all plans for T from S in D, is defined recursively as follows.

If T is empty, then Π(S,T,D) contains exactly one plan, namely the empty plan. Otherwise, let t be the first task atom in T, and R be the remaining task atoms. There are three cases. (1) If t is primitive and there is a simple plan p for t, then $\Pi(S,T,D) = \{\text{append}(p,q) : q \in \Pi(p(S),R,D)\}$. (2) If t is primitive and there is no simple plan for t, then $\Pi(S,T,D) = \emptyset$. (3) If t is compound, then $\Pi(S,T,D) = \cup\{\Pi(S,\text{append}(r,R),D) : r$ is a simple reduction of $t\}$.

2.3 Soundness and Completeness

The SHOP planning procedure is as follows:

procedure find-plan(S,T,D)
  return *seek-plan*(S,T,D,nil)
end *find-plan*
procedure seek-plan(S, T,D, p)
  if T = nil **then** return the list (p)
  t = the first task in T; R = the remaining tasks
  if t is primitive **then**
    if there is a simple plan q for t **then**
      return *seek-plan*($q(S),R,D$,append(p,q))
    else return FAIL
  else
    for every simple reduction r for t in S
      ans = *seek-plan*(S, append(r, R),D,p)
      if *ans* ≠ FAIL **then return** *ans*

end for
return FAIL
end if
end *seek-plan*

Since *find-plan* is a straightforward implementation of the definition of $\Pi(S,T,D)$, it is easy to show it is sound. For finite search spaces, *find-plan* is also complete. For infinite search spaces, it is incomplete for the same reason Prolog is incomplete: if the leftmost unexplored path is infinite, it will never return from that path. It is straightforward to make *find-plan* complete for infinite search spaces, by doing an iterative-deepening search of *find-plan*'s search space. Our implementation can do iterative deepening (at the user's option), but in practice we have found it more efficient not to use it.

3 Example Planning Domain

To illustrate how SHOP works, we now describe a simple transportation-planning domain. Table 1 defines the domain, Table 2 shows a specific problem in that domain, and Table 3 shows plans found by SHOP on several problems in that 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.

As mentioned earlier, SHOP incorporates several extensions to the syntax and semantics described in this paper. To illustrate those extensions, the transportation-planning domain uses most of them. In particular:

1. Axioms' tails and methods' preconditions can include negations (which are evaluated using the closed-world assumption) and calls to the Lisp evaluator. For example, Axiom A1 of Table 1 says that the taxi fare is \$1.50 plus \$1 for each mile traveled; and Method M1's precondition says that to pay the driver, we need sufficient money for the fare.
2. If a method's precondition is satisfied, then its entire tail is passed to the Lisp evaluator. Lisp's *quote*, *backquote*, and *comma* constructs can be used to prevent evaluation (see Method M2) or to do conditional evaluation (see Method M1, which does subtraction to create `set-cash`'s second argument).
1. Axioms can have multiple tails, to be used in an "if-then-else" fashion. For example, the axiom "(:- *head tail1 tail2 tail3*)" says *head* is true if *tail1* is true, or if *tail1* is false but *tail2* is true, or if *tail1* and *tail2* are false but *tail3* 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 says that walking distance is ≤ 3 miles in good weather, and ≤ 1 mile otherwise.

Table 1: A simple transportation-planning domain. The item numbers are for reference in the text.

No.	Item
A1	(:- (have-taxi-fare ?distance) ((have-cash ?m) (eval (>= ?m (+ 1.5 ?distance))))
A2	(:- (walking-distance ?u ?v) ((weather-is 'good) (distance ?u ?v ?w) (eval (<= ?w 3))) ((distance ?u ?v ?w) (eval (<= ?w 0.5))))
M1	(:method (pay-driver ?fare) ((have-cash ?m) (eval (>= ?m ?fare))) `((!set-cash ?m ,(- ?m ?fare))))
M2	(:method (travel-to ?q) ((at ?p) (walking-distance ?p ?q)) '((!walk ?p ?q)))
M3	(: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)))
O1	(:operator (!hail ?vehicle ?location) () ((at ?vehicle ?location)))
O2	(:operator (!wait-for ?bus ?location) () ((at ?bus ?location)))
O3	(:operator (!ride ?vehicle ?a ?b) ((at ?a) (at ?vehicle ?a)) ((at ?b) (at ?vehicle ?b)))
O4	(:operator (!set-cash ?old ?new) ((have-cash ?old)) ((have-cash ?new)))
O5	(:operator (!walk ?here ?there) ((at ?here)) ((at ?there)))

Table 2: An example transportation-planning problem.

Initial state:	1. ((at downtown) 2. (weather-is 'good) 3. (have-cash 12) 4. (distance downtown park 2) nil) 5. (distance downtown uptown 8) 6. (distance downtown suburb 12) 7. (at-taxi-stand taxi1 downtown) 8. (at-taxi-stand taxi2 downtown) 9. (bus-route bus1 downtown park) 10. (bus-route bus2 downtown uptown) 11. (bus-route bus3 downtown suburb))
Task list:	((travel-to suburb))

Table 3: Plans produced by SHOP on problems in the planning domain of Table 1. .In each problem, the distances and bus routes are the same as in lines 4–11 of Table 2. In each problem, SHOP found all possible plans in less than 0.01 seconds.

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

Go to 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	`((!HAIL TAXI1 DOWNTOWN)`
	`(!RIDE TAXI1 DOWNTOWN UPTOWN)`
	`(!SET-CASH 12 2.5))`

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

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

Go to suburb, good weather, have $80	`((!HAIL-TAXI DOWNTOWN)`
	`(!RIDE-TAXI DOWNTOWN SUBURB)`
	`(!SET-CASH 80 66.5))`

4. If the first element of a method's precondition or an axiom's tail is `:first`, SHOP's theorem prover returns after finding the first satisfier (just as Prolog would do), rather than looking for all satisfiers. As an example, in Method M3 this is used to tell SHOP that it should only consider hailing the first taxi at the taxi stand, rather than hailing all of them.
5. A method can have multiple pairs of preconditions and tails, to be used in an "if-then-else" fashion. For example, "(`:method` *head pre1 tail1 pre2 tail2*)" says that the reduction of *head* is *tail1* if *pre1* is true, or *tail2* if *pre1* is false and *pre2* is true. Method M3 uses this to specify that we won't consider bus travel unless we don't have enough money for taxi travel.
6. Operators have numeric costs (the default cost is 1), and the cost of a plan is the sum of its operator costs. The transportation domain does not illustrate this.

Although the transportation-planning domain is easy to represent in SHOP, we believe that most other AI planners would not have sufficient expressive power to represent it fully, because of the numeric computations that need to be done as part of the planning process.

4 Experiments

We have tested SHOP against two other planners: Blackbox [Kautz and Selman, 1998], which was one of the two fastest planners in the *AIPS-98* planning competition [McDermott, 1998]; and TLplan [Bacchus and Kabanza, 1998], which outperformed Blackbox by several orders of magnitude in Bacchus and Kabanza's tests.

Table 4: Performance of Blackbox, TLplan, and SHOP on blocks-world problems in the Blackbox software distribution.

Problem	CPU time (seconds)			No. of actions in plan		
	Black-box	TLplan	SHOP	Black-box	TLplan	SHOP
bw-simple	.00	.00	.00	2	2	2
bw-sussman	.06	.10	.10	6	6	6
bw-reversal4	.08	.10	.20	8	12	8
bw-12step	.74	.20	.60	12	12	12
bw-large-a	2.67	.50	.40	12	20	12
bw-large-b	1915.30	1.00	.90	18	28	18
bw-large-c	*	.26	.20	*	44	28
bw-large-d	*	.55	.36	*	58	36

* Blackbox ran out of memory on bw-large-c and bw-large-d after about six minutes and 40 minutes, respectively.

4.1 Blocks-World Planning

To run SHOP in the blocks world, we encoded the blocks-world planning algorithm of [Gupta and Nau, 1992] as a set of axioms, operators, and methods. We tested SHOP, TLplan and Blackbox on the blocks-world problems in the Blackbox software distribution. We ran SHOP and TLplan on a 167-MHz Sun Ultra, and Blackbox on a 143-MHz Sun Ultra. Both machines had 64 MB of RAM. The results are shown in Table 4.

Blackbox did worst: its time requirements increased far more quickly with problem size than SHOP's and TLplan's. This was to be expected, because SHOP and TLplan are guaranteed to run in low-order polynomial time on blocks-world problems, whereas Blackbox does an exponential-time search. Blackbox could not solve the two largest problems at all, because it ran out of memory.

On the larger problems, TLplan took more time than SHOP, and found longer plans. We should run more tests to establish statistical significance, but the results clearly are algorithmically significant: TLplan found some non-optimal plans that the blocks-world algorithm that encoded into SHOP's methods and operators [Gupta and Nau, 1992] had been designed to avoid.

4.2 Logistics Problems

To run SHOP in the logistics domain, we encoded the following procedure into methods, operators, and axioms.

- First remove from the current world-state all "useless objects" that will not contribute to the plan. These include packages not mentioned in the goal, and empty trucks and airplanes in the same city with other trucks and airplanes. Then do the following steps repeatedly until every package is at its final destination:
  1. If there is a truck or airplane at the same location as some packages that need to be picked up or dropped off, then pick them up or drop them off.

Table 5: Performance of Blackbox, TLplan, and SHOP on logistics problems from the Blackbox software distribution.

Problem	CPU time (seconds)			No. of actions in plan		
	Blackbox	TLplan	SHOP	Blackbox	TLplan	SHOP
log001	0.58	0.23	0.24	25	25	25
log002	95.98	0.3	0.19	31	27	24
log003	99.00	0.29	0.29	28	27	27
log004	130.75	1.09	0.46	71	51	51
log005	231.94	0.95	0.66	69	42	42
log006	321.27	1.48	0.71	82	51	50
log007	264.05	3.61	1.66	96	70	66
log008	317.42	4.79	1.67	110	70	66
log009	1609.46	2.73	1.66	121	70	66
log010	84.05	1.95	0.5	71	41	39
log011	137.93	1.54	0.54	68	46	44
log012	136.30	1.53	0.41	49	38	34
log013	165.84	4.54	1.32	85	66	63
log014	77.75	6	2.23	89	73	69
log015	424.37	3.9	1.37	91	63	62
log016	926.97	0.92	0.67	85	39	39
log017	758.47	0.84	0.48	83	43	43
log018	152.35	5.08	1.19	105	46	46
log019	149.22	2.12	0.64	78	45	43
log020	538.22	6.86	1.47	113	89	84
log021	190.49	4.04	1.16	87	59	58
log022	846.84	4.19	1.72	111	75	70
log023	173.97	3.45	1.43	85	62	58
log024	74.83	3.23	1.38	87	64	60
log025	74.00	2.68	1.01	84	57	53
log026	233.41	2.35	0.95	80	55	49
log027	145.16	3.36	1.28	97	70	65
log028	867.40	7.26	2.47	118	74	71
log029	89.52	3.42	1.67	84	45	44
log030	495.37	2.26	0.74	92	51	47

Table 6: Performance of TLplan and SHOP on logistics problems in the TLplan software distribution. According to Fahiem Bacchus, these problems originally came from the *AIPS-98* planning competition [McDermott, 1998].

Problem	CPU time (seconds)		No. of actions in plan	
	TLplan	SHOP	TLplan	SHOP
log-x-1	0.83	0.33	26	26
log-x-2	3.2	2.18	33	33
log-x-3	34.89	4.84	55	55
log-x-4	95.36	8.87	59	59
log-x-5	0.66	0.64	22	24
log-x-6	156.33	20.28	72	72
log-x-7	10.13	3.47	34	36
log-x-8	180.64	9.55	41	41
log-x-9	398.61	16.81	85	76
log-x-10	317.04	37.27	105	103
log-x-11	10.38	0.46	31	30
log-x-12	528.16	20.32	41	43
log-x-13	2026.3	49.49	67	69
log-x-14	1577.6	17.05	94	90
log-x-15	49.04	2.55	94	90

2. Else if there is a package p in some city c, and p's destination is a city other than c, then choose any airplane, and let d be the city that it is in. Use the truck in city d to collect all packages in city d that need to be moved. Bring to the airport all packages that need to go to the airport, and load them onto the airplane. Then fly the airplane to city c.
3. Else if there is an airplane with at least one package on board, then fly it to the destination of one of the packages on board.
4. Else if there are one or more packages that need to be picked up, then drive a truck to the location of any one of them.
5. Else if there is a truck that is carrying one or more packages, then drive it to the final destination of one of the packages in the truck.

We ran SHOP and TLplan on logistics problems in the Blackbox and TLplan distributions, on a 167-MHz Sun Ultra with 64 MB of RAM. Because of Blackbox's memory requirements, we did not run it ourselves. Instead, we used published data for Blackbox on a machine that is faster than ours and has 8 GB of RAM.[1] Tables 5 and 6 show the results.

Again Blackbox did worst and SHOP did best. Blackbox was several orders of magnitude slower than both SHOP and TLplan, and it found significantly larger plans. SHOP and TLplan found plans of comparable size, but on most of the problems SHOP ran several times faster than TLplan (more than an order of magnitude faster on the more difficult problems).

5 Discussion and Conclusions

It did not surprise us that SHOP did so much better than Blackbox, for SHOP's methods and axioms contained sophisticated domain knowledge that could not be represented in Blackbox's operators. However, it did surprise that SHOP did so much better than TLplan. Here, we think, are the primary reasons why it did so:

1. Although TLplan's modal-logic representation capabilities are quite sophisticated, their use (at least in the examples we have seen) has been limited to writing pruning heuristics rather than actual planning algorithms. SHOP's use of HTN methods makes it easy to write efficient planning algorithms, as we did for both the blocks world and the logistics domain.
2. TLplan's planning algorithm is basically a state-space search, whereas SHOP uses HTN-style problem reduction. Problem reduction can be much more efficient than state-space search (even by an exponential amount in some cases [Korf, 1987; Yang *et al.*, 1992]).

Our results support the contention that total-order forward search, combined with HTN-style problem reduction, can "scale up" to complex planning problems better than partial-order action-based planning. Our

[1] We got the Blackbox performance data from Table 11 of [Bacchus and Kabanza 1998]. 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.

results also illustrate the impact that planning applications can have on planning theory: SHOP is a domain-independent formalization and implementation that evolved from our previous domain-specific work on manufacturing planning and computer bridge.

Our ongoing and future work is as follows:

- We are doing additional experiments and analyses in order to get a better understanding of the efficiency issues discussed above.
- SHOP appears to be powerful enough to be of use in complex applications such as noncombatant evacuation operation planning [Muñoz-Avila *et al*., 1999]. To make it easier to embed SHOP in such applications, we are creating an implementation of SHOP in Java.
- It is straightforward to prove soundness and completeness using the definitions in Section 2, but it is more difficult to prove soundness and completeness in the presence of some of the extensions discussed in Section 3 (such as the calls to the Lisp evaluator). We have begun working with others who have experience in these issues, to put this aspect of SHOP on a more solid formal footing.
- We are developing a general way to handle some partial-order-planning operations while preserving SHOP's expressivity and left-to-right control strategy. We intend to describe this in a forthcoming paper.

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 NSF DMI-9713718.

References

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

[Bacchus and Kabanza, 1998] F. Bacchus and K. Kabanza. Using temporal logic to express search control knowledge for planning. <ftp://logos.uwaterloo.ca/pub/bacchus/BKTlplan.ps>. Submitted to *Artificial Intelligence*, 1998.

[Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Erol *et al*., 1994] K. Erol, K, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for Hierarchical Task-Network planning. *Proc. 2nd Int'l Conf. AI Planning Systems (AIPS-94)*, 249-254, 1994.

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

[Gupta and Nau, 1992] N. Gupta and D. Nau. On the complexity of blocks-world planning. *Artificial Intelligenc*e 56(2-3), 223–254, 1992.

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

[Koehler *et al*., 1997] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *ECP-97*, 273–285, 1997.

[Korf, 1987] R. Korf. Planning as search: A quantitative approach. *Artificial Intelligence* 33:65–88, 1987.

[McAllester *et al*., 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. AAAI-91*, 1991.

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

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

[Muñoz-Avila *et al*., 1999] H. Muñoz-Avila, D. Aha, J. Ballas, L. Breslow, and D. Nau. Using guidelines to constrain interactive case-based HTN planning. Tech. Report AIC-99-004, Naval Center for Applied Research on AI, Naval Research Lab., Washington, DC, 1999.

[Penberthy and Weld, 1992] J. S. Penberthy and D. Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. KR-92*, 1992.

[Sacerdoti, 1977] E. Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, 1977.

[Smith *et al*., 1997] S. J. Smith, K. Hebbar, D. Nau, and I. Minis. Integrating electrical and mechanical design and process planning. In Martti Mantyla, Susan Finger and Tetsuo Tomiyama (ed.), *Knowledge Intensive CAD, Volume 2*, pp. 269-288, 1997.

[Smith *et al*., 1998] S. J. Smith, D. Nau, and T. Throop. Computer bridge: a big win for AI planning. *AI Magazine* 19(2), 93–105, 1998.

[Tate, 1977] A. Tate. Generating project networks. In *Proc. IJCAI-77*, 888–893, 1977.

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

[Wilkins, 1990] D. Wilkins. Can AI planners solve practical problems?. *Computational Intelligence* 6 (4): 232–246, 1990.

[Yang *et al*., 1992] Q. Yang, D. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence* 8(2):648–676, February 1992.