Control Strategies in Planning*

Fahiem Bacchus
Dept. Of Computer Science
University Of Waterloo
Waterloo, Ontario
Canada, N2L 3G1

Froduald Kabanza
Dept. De Math Et Informatique
Universite De Sherbrooke
Sherbrooke, Quebec
Canada, J1K 2R1

1 Introduction

Over the years increasing sophisticated planning algorithms have been developed. These have made for more efficient planners. However, current state of the art planners still suffer from severe complexity problems, problems that can surface even in domains in which good plans are easy to generate, like the blocks world.

Planners generally employ search to find plans, and planning research has identified a number of different spaces in which search can be performed. Of these, three of the most common are (1) the forward-chaining search space, (2) the backward-chaining search space, and (3) the space of partially ordered plans. The forward-chaining space is generated by applying all applicable actions to every state starting with the initial state; the backward-chaining space by regressing the goal conditions back through actions that achieve at least one of the subgoals; and the space of partially ordered plans by applying a collection of plan modification operators to an initial "dummy" plan.

Backward-chaining and partial-order planning both have a significant advantage over forward-chaining in that they are goal directed: they never consider actions that are not syntactically relevant to the goal. Partial-order planning has an additional advantage over backward-chaining in that it explores partially ordered plans. This means that the search algorithm can detect at every point in its search space whether or not various actions interact, and impose an ordering between them only if they do. Linear backward or forward chaining planners might have to backtrack over an exponential number of improper orderings.

However, both backward-chaining and partial-order planners search in spaces in which knowledge of the state of the world is incomplete. For example, computing whether or not a literal holds at a particular point in a partially-ordered plan is only tractable in certain restricted cases [Cha87]. In the forward-chaining space, on the other hand, for most planners, the points are complete world descriptions.[1] Hence, there is not the same difficulty in determining if various conditions hold after a sequence of forward-chained actions.

The choice between the various search spaces has been the subject of much inquiry, with current consensus seemingly converging on the space of partial plans [BW94, MDBP92], mainly because of its goal-directness and its ability to dynamically detect interactions between actions. However, these studies only indicate that "blind" search in the space of partially ordered plans is superior to "blind" search in other spaces, where by "blind" search we mean search that uses only simple domain *independent* heuristics for search control, like counting the number of unsatisfied goal conditions.

In fact, it is clear that domain independent search strategies in any of these planning spaces are bound to fail. Theoretical work [ENS92, Byl92, Sel94] indicates that for the traditional STRIPS actions used by almost all current planners, finding a plan is, in general, intractable. This means that no domain independent planning algorithm can succeed except in very simple (and probably artificial) domains. There may be many domains where it is feasible to find plans, but where domain structure must be exploited to do so. This can be verified empirically, e.g., the SNLP implementation of Soderland et al. [SBW90] cannot generate plans to reconfiguring more than 5 blocks in the blocks world, even though it is easy to generate good plans in this domain [GN92].

One way of exploiting domain structure during planning is to use domain information to control search. Hence, a more practical evaluation of the relative merits of various planning algorithms and search spaces would take into account how easy it is to exploit domain knowledge to control search in that space. The idea of search control is, of course, not new, e.g., it is a prominent part of the PRODIGY system [CBE+92]. However we have been investigating a new approach to specifying and utilizing control knowledge that draws on the work of researchers in program and system verification (see, e.g., [CG87]). Specifically, we have built a planner, TLPLAN

*This research was supported by the Canadian Government through their IRIS project and NSERC programs. Please note that this abstract has also been submitted to a AAAI spring symposium workshop.

[1] This is true for planners that utilize a complete described initial state and the STRIPS assumption. It is also true for planners that model uncertainty as "branching" probabilistic actions, e.g., the BURIDAN planner [KHW94]. In such planners each action maps a completely described world to a *set* of worlds, each of which is assigned some probability, but each world in this set is completely described.

that takes as its inputs not only the standard initial and goal state descriptions along with a specification of a set of actions, but also a domain control strategy expressed as a formula of a first-order temporal logic. It utilizes this formula to control its search of the forward-chaining search space by incrementally evaluating the formula on the sequences of worlds generated by forward-chaining. We have experimented with a number of domains and have demonstrated that our method for expressing and utilizing search control information is both natural and very effective, sometimes amazingly effective.

Although, it is possible that our method could be adapted to control search in other planning spaces, we have found that the complete world descriptions generated by forward-chaining make it significantly easier to express natural domain strategies. There are two reasons for this. First, complete world descriptions support the efficient evaluation of complex first-order formulas via model-checking [HV91]. This allows us to determine the truth of complex conditions, expressed as first-order formulas, in the worlds generated by forward-chaining. Part of our implementation is a first-order formula evaluator, and TLPLAN allows the user to define predicates by first-order formulas. These predicates can in turn be used in the temporal control formula, where they act to detect various conditions in the sequence of worlds explored by the planner. And second, many domain strategies seem to be most naturally expressed in a "forward-direction." This makes their application to controlling forward-chaining obvious, but their application to the other search spaces less than obvious.

In the rest of the abstract we briefly describe the temporal logic we use to express control strategies. And then we give as an example of our approach a description of its application to the blocks world domain. We are exploring a number of extensions of our approach, but will not have space to discuss these in this abstract.

2 First-order Linear Temporal Logic

We use as our language for expressing strategic knowledge a first-order version of linear temporal logic (LTL) [Eme90]. The language starts with a standard first-order language, $\mathcal{L}$, containing some collection of constant, function, and predicate symbols. LTL adds to $\mathcal{L}$ the following temporal modalities: U (until), $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next). The standard formula formation rules for first-order logic are augmented by the following rules: if f_1 and f_2 are formulas then so are $f_1 \mathsf{U} f_2$, $\Box f_1$, $\Diamond f_1$, and $\bigcirc f_1$. Note that the first-order and temporal formula formation rules can be applied in any order, so, e.g., quantifiers can scope temporal modalities allowing *quantifying into* modal contexts.

Our planner takes advantage of the complete world descriptions generated by forward chaining to evaluate first-order formula using model checking. To allow this to be computationally effective and at the same time not limit ourselves to finite domains (e.g., we may want to have access to the integers in our axiomatization), we use *bounded quantification* instead of standard quantification. In particular, instead of the quantifiers $\forall x$ or $\exists x$, we have $\forall[x{:}\gamma]$ and $\exists[x{:}\gamma]$, where γ is an atomic formula[2] whose free variables include x. It is easiest to think about bounded quantifiers semantically: $\forall[x{:}\gamma]\,\phi$ for some formula ϕ holds iff ϕ is true for all x such that $\gamma(x)$ holds, and $\exists[x{:}\gamma]\,\phi$ holds iff ϕ is true for some x such that $\gamma(x)$ holds. We require that in any world the set of satisfying instances of γ be finite.

Semantically, formulas of LTL are interpreted by models of the form $M = \langle s_0, s_1, \ldots\rangle$, i.e., a sequence of states. Every state s_i is a model (a first-order interpretation) for the base language $\mathcal{L}$. In addition to the standard rules for the first-order connectives and quantifiers, we have that for a state s_i in a model M and formulas f_1 and f_2:

- $\langle M, s_i\rangle \models f_1 \mathsf{U} f_2$ iff there exists $j \geq i$ such that $\langle M, s_j\rangle \models f_2$ and for all k, $i \leq k < j$ we have $\langle M, s_k\rangle \models f_1$: f_1 is true until f_2 is achieved.[3]
- $\langle M, s_i\rangle \models \bigcirc f_1$ iff $\langle M, s_{i+1}\rangle \models f_1$: f_1 is satisfied in the next state.
- $\langle M, s_i\rangle \models \Diamond f_1$ iff there exists $j \geq i$ such that $\langle M, s_j\rangle \models f_1$: f_1 is eventually satisfied.
- $\langle M, s_i\rangle \models \Box f_1$ iff for all $j \geq i$ we have $\langle M, s_j\rangle \models f_1$: f_1 is always satisfied.

Finally, we say that the model M satisfies a formula f if $\langle M, s_0\rangle \models f$.

First-order LTL allows us to express various claims about the sequence of states S. For example, $\bigcirc\bigcirc on(A,B)$ asserts that in state s_2 we have that A is on B. Similarly, $\Box\neg holding(C)$, asserts that we are never in a state where we are holding C, and $\Box(on(B,C) \Rightarrow (on(B,C) \mathsf{U}\ on(A,B)))$ asserts that whenever we enter a state in which B is on C it remains on C until A is on B, i.e., $on(B,C)$ is preserved until we achieve $on(A,B)$. With quantification we can express even more, e.g., $\forall[x{:}clear(x)]\,\bigcirc clear(x)$ asserts that every object that is clear in the current state remains clear in the next state. This is an example of quantifying into a modal context.

We are going to use LTL formulas to express search control information, or domain strategies. Search control generally needs to take into account properties of the goal, and we have found a need to make reference to requirements of the goal in our LTL formulas. To accomplish this we augment the base language $\mathcal{L}$ with a *goal modality*. In particular, to the base language $\mathcal{L}$ we add the following formula formation rule: if f is a formula of $\mathcal{L}$ then so is $\textsc{goal}(f)$. This modality can be used whenever the goal is given as a list of literals to be achieved (most planners take goals specified in this manner). Our planner uses this list of literals as if it were a complete world description, and evaluates the formula $\textsc{goal}(f)$ by evaluating f in the goal world. Of course, the goal is generally only a partial specification of the world, so in treating the goal as a complete description the goal modality takes on the semantics of "provable requirement of the goal." For example, if the goal is the set of literals

[2] We also allow γ to be an atomic formula within the scope of a GOAL modality (see below).

[3] Note that, since we only test k strictly less than j, as is standard, any state s_i that satisfies f_2 satisfies $f_1 \mathsf{U} f_2$ for any f_1.

$\{on(A,B), on(B,C)\}$ then GOAL$(on(A,B) \wedge on(B,C))$ will evaluate to true, GOAL$(clear(A))$ will evaluate to false—although $clear(A)$ does not contradict the goal it is not a necessary/provable requirement of the goal.

3 Using LTL to Express Search Control Information

Any LTL formula specifies a property of a sequence of states. In planning, we are dealing with sequences of executable actions, but to each such sequence there corresponds a sequence of worlds: the worlds we pass through as we execute the actions. These sequences act as models for the language $\mathcal{L}$. Hence, we can check the truth of an LTL formula given a plan, by checking its truth in the sequence of worlds visited by that plan using standard model checking techniques (see, e.g., [CG87]).[4] Hence, if we have a domain strategy for the goal $\{on(B,A), on(C,B)\}$ like "if we achieve $on(B,A)$ then preserve it until $on(C,B)$ is achieved", we could express this information as an LTL formula and check its truth against candidate plans.

In order to use our LTL formula to control search we have developed an algorithm for incrementally evaluating an LTL formula. Specifically, our planner labels each world generated in our search of the forward-chaining space with an LTL formula f, with the initial world being labeled with the original LTL control formula. When we expand a world w we *progress* its formula f through w generating a new formula f^+. This new formula becomes the label of all of w's successor worlds (the worlds generated by applying all applicable actions to w). A formula f and its progression f^+ computed by our progression algorithms (which we will not present due to space limitations) are related by the following theorem:

Theorem 3.1 *Let $M = \langle s_0, s_1, \ldots \rangle$ be any LTL model, and let s_i be the i-th state in the sequence of states M. Then, we have for any LTL formula f, $\langle M, s_i \rangle \models f$ if and only if $\langle M, s_{i+1} \rangle \models f^+$.*

If f progresses to FALSE, (i.e., f^+ is FALSE), then this theorem shows that no sequences of worlds emanating from w can satisfy our LTL formula. Hence, we can mark w as a dead-end in the search space and prune all of its successors.

4 Empirical Results from the Blocks World

We demonstrate our approach using the blocks world. In our case we use four operators in our axiomatization (Table 1). If we run our planner with the vacuous search control formula $\Box$TRUE, which admits every sequence of worlds and thus provides no search control, we obtain the performance given in Figure 1 using blind depth-first search that checks for cycles. Each data point represents 5 randomly generated blocks world problems, where the initial state and the goal state were independently randomly generated. The graph shows a plot of the average time taken to solve all 5 problems (in CPU seconds on a SUN-1000). The same problems were also run using the SNLP and PRODIGY4.0 systems. The graph demonstrates that these planners hit a computational wall at or before 6 blocks. Furthermore, within the time bounds imposed only blind depth-first search in the forward-chaining space was able to solve all of the problems. The SNLP system failed to solve 4 of 6 block problems, while the PRODIGY system failed to solve 2 of the 6 block problems. The times shown in the graph include the times taken by the failed runs. The PRODIGY system was the only system that was run with domain dependent search control (i.e., it used a collection of control rules specific to the blocks world), and this shows up in its performance. In fact, of the 4 six block problems it was able to solve, it was able to solve them quite quickly. But its failure on the other two (compare with our results for controlled TLPLAN below) indicates that its search space is not as easy to control. Note that for the blocks world with a holding predicate there are only 866 points in the forward-chaining space (i.e., 866 different configurations of the world) for 5 blocks, 7057 points for 6 blocks and 65990 for 7 blocks. TLPLAN can exhaustively search the 6 blocks space in about 20 minutes of CPU time, but the search spaces explored by SNLP and PRODIGY, are much larger.

Despite the size of the search space, it is easy to come up with strategies in the blocks world. A basic one is that towers in the blocks world can be build from the bottom up. That is, if we have built a good base we need never disassemble that base to achieve the goal. We can write a first-order formula that defines when a block x is on top of a good tower, i.e., a good base that need not be disassembled.

$$
\begin{aligned}
&\texttt{goodtower} \equiv clear(x) \wedge \texttt{goodtowerbelow}(x)\\
&\texttt{goodtowerbelow}(x) \equiv\\
&\quad (ontable(x) \wedge \neg\text{GOAL}(\exists[y{:}on(x,y)]))\\
&\quad \vee \exists[y{:}on(x,y)]\, \neg\text{GOAL}(ontable(x)) \wedge \neg\text{GOAL}(clear(y))\\
&\quad\quad \wedge \forall[z{:}\text{GOAL}(on(x,z)] \Rightarrow z = y)\\
&\quad\quad\quad \wedge \forall[z{:}\text{GOAL}(on(z,y)] \Rightarrow z = x)\\
&\quad\quad \wedge \texttt{goodtowerbelow}(y)
\end{aligned}
$$

A block x satisfies the predicate **goodtower**(x) if it is on top of a tower, i.e., it is clear, and it and the tower below it are good, i.e., the tower below does not violate any goal conditions. The various tests for the violation of a goal condition are given in the definition of **goodtowerbelow**. If x is on the table, the goal can not require that it be on another block y. On the other hand, if x is on another block y, then x should not be required to be on the table, nor should y be required to be clear, any block that is required to be below x should be y, any block that is required to be on y should be x, and finally the tower below y cannot violate any goal conditions.

Our planner can take as input a first-order definition for a predicate like the above (written in Lisp syntax) and it can evaluate the truth of this defined predicate in

[4] LTL formulas actually require an infinite sequence of worlds as their model. In the context of standard planning languages, where a plan consists of a finite sequence of actions, we can terminate every finite sequence of actions with an infinitely replicated "idle" action. This corresponds to infinitely replicating the final world in the sequence of worlds visited by the plan.

Operator	Preconditions and Deletes	Adds
pickup(x)	*ontable*(x), *clear*(x), *handempty*.	*holding*(x).
putdown(x)	*holding*(x).	*ontable*(x), *clear*(x), *handempty*.
stack(x, y)	*holding*(x), *clear*(y).	*on*(x, y), *clear*(x), *handempty*.
unstack(x, y)	*on*(x, y), *clear*(x), *handempty*.	*holding*(x), *clear*(y).

Table 1: Blocks World operators.

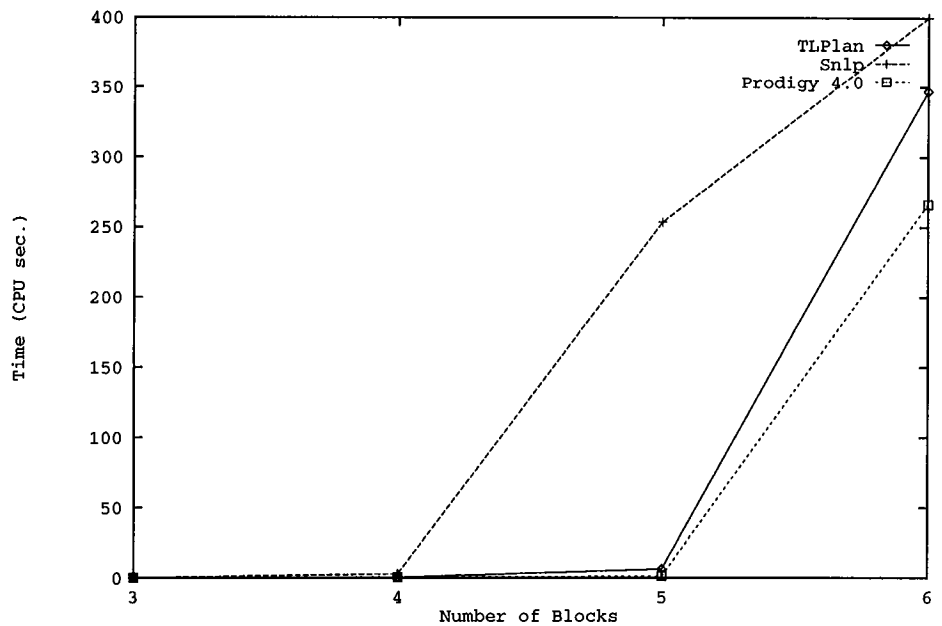

Figure 1: Blind Search in the Blocks World

the current world for any block. Using this we can write the following LTL control formula

$$\Box(\forall[x{:}clear(x)]\,\mathtt{goodtower}(x) \Rightarrow \bigcirc\mathtt{goodtowerabove}(x)). \quad (1)$$

This formula says that whenever we have a good tower, in the the next state this tower must be preserved. **Goodtowerabove** is defined in a manner symmetric to **goodtowerbelow**. In particular, it is falsified if we are holding x or if we stack another block y on x that violates a goal condition. Thus, the planner can prune those successor worlds that fail these conditions.

What about towers that are not good towers? Clearly they violate some goal condition. So there is no point in stacking more blocks on top of them as eventually we must disassemble them. So at the same time as preserving good towers we can define a bad tower predicate as

$$\mathtt{badtower}(x) \equiv clear(x) \wedge \neg\mathtt{goodtower}(x).$$

And we can augment our control strategy to prevent growing bad towers.

$$\Box(\forall[x{:}clear(x)]\,\mathtt{goodtower}(x) \Rightarrow \bigcirc\mathtt{goodtowerabove}(x) \wedge \mathtt{badtower}(x) \Rightarrow \bigcirc(\neg\exists[y{:}on(y,x)])) \quad (2)$$

This control formula allows only blocks on top of bad towers to be picked up. This is what we want, as such tower must be disassembled. However, a single block on the table that is not intended to be on the table is also a bad tower. In this case we do not want such blocks to be picked up except when the block they are intended to be on is on top of a good tower (i.e., when their final position is ready). Without this additional control the planner will continually attempt to pick up such blocks only to find that it must return them to the table. This causes a number of one-step backtracks as the planner detects a state cycle in the forward-chaining space. Although it does not affect the quality of the plan we construct (as we backtrack from these steps) it does slow down the planner. Hence our final control for the blocks world becomes

$$\Box(\forall[x{:}clear(x)]\,\mathtt{goodtower}(x) \Rightarrow \bigcirc goodtowerabove(x) \wedge \mathtt{badtower}(x) \Rightarrow \bigcirc(\neg\exists[y{:}on(y,x)]) \wedge (ontable(x) \wedge \exists[y{:}\mathrm{GOAL}(on(x,y))]\,\neg\mathtt{goodtower}(y)) \Rightarrow \bigcirc(\neg holding(x))). \quad (3)$$

The performance our our planner with these three different control formulas is shown in Figure 2. As in Figure 1 each data point represents the average time to solve 5 randomly generated blocks world problems of various sizes. We observe that our final control formula for the blocks world allows our planner to find plans that are most a factor of 2 longer than the optimal using backtrack free depth-first search taking time quadratic in the

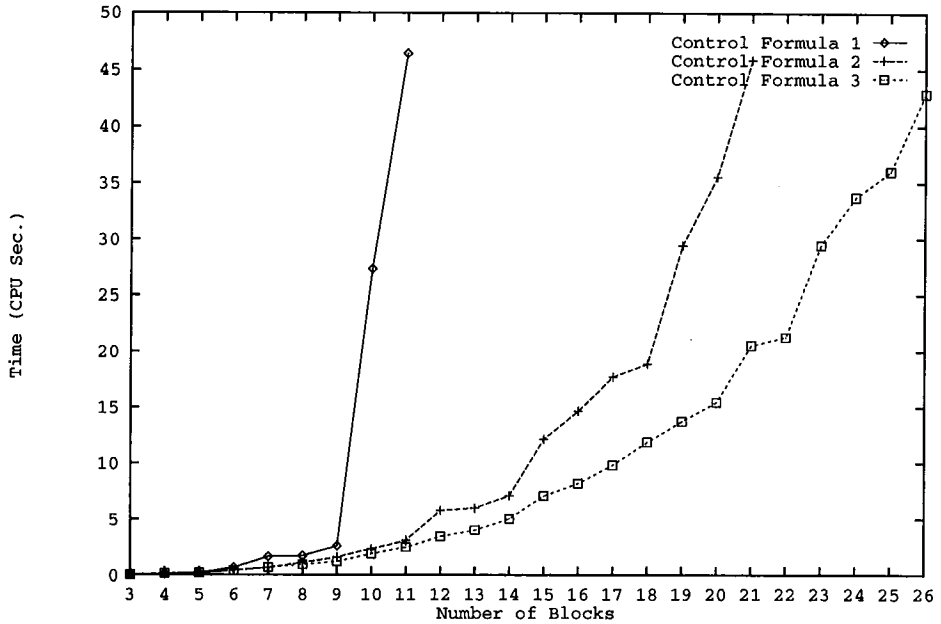

Figure 2: Controlled Search in the Blocks World

number of blocks. If it uses breadth-first search it can find an optimal plan, but this task is NP-Hard [GN92].

5 Conclusions

We have experimented with a number of other domains including the blocks world with limited space on the table, the flat tyre domain, and the PRODIGY scheduling domain. In all of these domains we have found that it is *easy* to outperform planners like SNLP and PRODIGY using very simple and obvious control strategies. It is more difficult to write complete strategies, i.e., strategies that yield plans in polynomial time, although this was possible in a number of these domains also. Some important points are

1. Just as we can axiomatize "static" knowledge about domains like $\exists x.holding(x) \Rightarrow \neg handempty$, we have and can axiomatize "dynamic" strategic knowledge.
2. This strategic knowledge can be expressed in a declarative representation and utilized to guide problem solving. The declarative nature of this representation is one of the ways our approaches differs from classical state-based heuristics.
3. Forward chaining search seems to be a very natural fit with most forms of strategic knowledge, and in conjunction with this kind of knowledge it can be used to construct very efficient planners for various domains.

References

[BW94] A. Barrett and D.S. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[Byl92] T. Bylander. Complexity results for planning. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 274–279, 1992.

[CBE+92] J.G. Carbonell, J. Blythe, O. Etzioni, Y. Gill, R. Joseph, D. Khan, C. Knoblock, S. Minton, A. Pérez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and turorial. Technical Report CMU–CS–92–150, School of Computer Science, Carnegie Mellon University, 1992.

[CG87] E. M. Clarke and O. Grümberg. Research on automatic verification of finite-state concurrent systems. In Joe F. Traub, Nils J. Nilsson, and Barbara J. Grozf, editors, *Annual Review of Computing Science.* Annual Reviews Inc., 1987.

[Cha87] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 16, pages 997–1072. MIT, 1990.

[ENS92] K. Erol, D.S. Nau, and V.S. Subrahmanian. On the complexity of domain-independent planning. In *Proceedings of the AAAI National Conference*, pages 381–386, 1992.

[GN92] N. Gupta and D.S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254, 1992.

[HV91] J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: a manifesto. In *Pro-*

ceedings of the International Conference on Principles of Knowledge Representation and Reasoning, pages 325–334, 1991.

[KHW94] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the AAAI National Conference*, pages 1073–0178, 1994.

[MDBP92] S. Minton, M. Drummond, J. Bresina, and A. Phillips. Total order vs. partial order planning: Factors influencing performance. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 83–82, 1992.

[SBW90] S. Soderland, T. Barrett, and D. Weld. The SNLP planner implementation. Contact bug-snlp@cs.washington.edu, 1990.

[Sel94] B. Selman. Near-optimal plans, tractability and reactivity. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 521–529, 1994.