Using Temporal Logic to Control Search in a Forward Chaining Planner*

Fahiem Bacchus
Dept. Of Computer Science
University Of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
fbacchus@logos.uwaterloo.ca

Froduald Kabanza
Dept. De Math Et Informatique
Universite De Sherbrooke
Sherbrooke, Quebec
Canada, J1K 2R1
kabanza@dmi.usherb.ca

Abstract. Over the years increasingly sophisticated planning algorithms have been developed. These have made for more efficient planners, but unfortunately these planners still suffer from combinatorial explosion. Indeed, recent theoretical results demonstrate that such an explosion is inevitable. It has long been acknowledged that domain independent planners need domain dependent information to help them plan effectively. In this work we describe how natural domain information, of a "strategic" nature, can be expressed in a temporal logic, and then utilized to effectively control a forward-chaining planner. There are numerous advantages to our approach, including a declarative semantics for the search control knowledge; a high degree of modularity (the more search control knowledge utilized the more efficient search becomes); and an independence of this knowledge from the details of the planning algorithm. We have implemented our ideas in the TLPLAN system, and have been able to demonstrate its remarkable effectiveness in a wide range of planning domains.

1 Introduction

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.

Planners that explore the backward-chaining space or the space of partially ordered plans have an advantage over those that explore the forward-chaining space in that the latter spaces are generated in a "goal directed" manner. Hence, such planners are intrinsically goal directed: they need never consider actions that are not syntactically relevant to the goal because the spaces they explore do not include such actions. Partial-order planners have an additional advantage over simple backward chaining planners in that the objects in their search space are partially ordered plans. This allows these planners to delay ordering actions until they detect an interaction between them. Linear backward or forward-chaining planners, on the

*This research was supported by the Canadian Government through their IRIS project and NSERC programs.
Fahiem Bacchus is currently on sabbatical leave from the University of Waterloo, Canada.

other hand, might be forced into backtracking because they have prematurely committed to an ordering between the actions.

However, both backward-chaining and partial-order planners search in spaces in which knowledge of the state of the world is far less complete than in the forward-chaining space. For example, even if a backward-chaining planner starts with a completely described initial world and actions that preserve the completeness of this description, it will still have only incomplete knowledge of the world state at the various points of its search space. Partial order planners also suffer from this problem. The points of their search space are incomplete partially ordered plans, and at the various stages of an incomplete plan we have only limited knowledge of the state of the world. On the other hand, the points in the forward-chaining space are world descriptions. Such descriptions provide a lot of information about the world state, even if the description is incomplete. As we will demonstrate in this paper, such knowledge can be effectively utilized to control search in this space.

The choice between the various search spaces has been the subject of much recent inquiry [BW94, MDBP92], with current consensus seemingly converging on the space of partially ordered plans,[1] mainly because of its goal-directness and least commitment attitude towards action ordering. However, these studies have only investigated simple heuristic search over these spaces, where domain *independent* heuristics, like counting the number of unsatisfied sub-goals, are utilized. Domain independent heuristics cannot take advantage of structural features that might be present in a particular domain.

Theoretical work [ENS92, 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. More importantly, however, is that 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 partial order planner implemented by Soderland et al. [SBW90] cannot effectively generate plans for reconfiguring more than 5 blocks in the blocks world using domain independent heuristic search. Nevertheless, the blocks world does have sufficient structure to make it 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 merit of various planning algorithms and search spaces would also take into account how easy it is to exploit domain knowledge to control search in that space. The idea of search control is not new, e.g., it is a prominent part of the PRODIGY planing system [CBE$^+$92]. Our work, however, makes a number of new contributions to the notion of search control.

In particular, we demonstrate how search control information can be expressed in a first-order temporal logic, and we develop a method for utilizing this information to control search during planning. By using a logic we gain the advantage of providing a formal semantics for the search control information. Furthermore, we would claim that this semantics is quite natural and intuitive. This differentiates our mechanism for search control from classical state-based heuristics and from the control rules employed by the PRODIGY system. PRODIGY control rules are implemented as a rule-based system. Various rules are activated dependent on the properties of the node in the search space that is currently being expanded. These rules are activated in a particular order and have various effects. This means that any attempt to give a semantics to these rules would require an operational semantics that makes reference to way in which the rules are utilized.

By using the forward-chaining search space we end up searching in the space of world de-

[1] Although, see [VB94] for an refreshing counterpoint.

scriptions. This allows us to utilize search control knowledge that only makes reference to the properties of these worlds, i.e., to properties of the domain. Thus the search control knowledge used can be considered to be no different from the description of the domain actions: it is part of our knowledge of the dynamics of the domain. In contrast, PRODIGY control rules include things like binding selection rules that guide the planner in deciding how to instantiate actions. Such rules have to do with particular operations of the PRODIGY planning algorithm, and to compose such rules the user must not only possess domain knowledge but also knowledge of the planning algorithm.

Finally, the language in which we express search control knowledge is richer than previous approaches. Hence, it can capture much more complex control knowledge. In particular, the control strategies are not restricted to considering only the current state, as are PRODIGY control rules and state-based heuristics. They can, e.g., consider past states and pass information forward into future states. All of these features make the control information employed by our system not only easier to express and understand, but also more effective, sometimes amazingly effective, as we will demonstrate in Section 4.

Using the forward-chaining search space is not, of course, a panacea. It does, however, seem to better support effective search control. We have already mentioned two reasons for this: we have access to more information about the world state in this space, and it allows us to express search control information that is independent of the planning algorithm. We have also found a third advantage during our experiments. In many domains, humans seem to possess strategies for achieving various kinds of goals. Such strategies seem to be most often expressed in a "forward-direction". This makes their use in controlling forward-chaining search straightforward, but exploiting them in the other search spaces not always so. Due to its support of effective search control, we have found that forward-chaining can in many domains yield planners that are more effective than those based on partial order planning or backwards-chaining regression.

The forward-chaining search space still suffers from the problem that it is not goal directed, and in many of our test domains we have found that some of the search control information we added was designed to recapture goal-directedness. Much of this kind of search control knowledge can be automatically inferred from the operator descriptions, using ideas like those of [Etz93]. Inferring and learning search control in the form we utilize is an area of research we are currently pursuing. One of the key advantages of using a logic to express search control knowledge is that it opens the door to reasoning with this knowledge to, e.g., generate further control knowledge or to verify and prove properties of the search control knowledge. But again this avenue is a topic for future research.

In the rest of the paper we will first describe the temporal logic we use to express domain strategies. We then describe how knowledge expressed in this language can be used to control forward chaining search. We have implemented our approach in a system we call TLPLAN, and we describe some of our empirical results with this system next. We close with a summary of our contributions and a description of some of the extensions to our approach we are currently working on.

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 works with standard STRIPS operators and world descriptions, and it takes advantage of the fact that these world descriptions support the efficient testing of various conditions. In particular, worlds described as lists of positive literals support the efficient evaluation of complex first-order formulas via model-checking [HV91]. Hence, we can express complex conditions as first-order formulas and evaluate their truth in the worlds generated by forward-chaining. Part of our TLPLAN 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 temporal control formulas, where they act to detect various conditions in the sequence of worlds explored by the planner.

To ensure that it is computationally effective to evaluate these first-order formulas and at the same time not limit ourselves to finite domains (e.g., we may want to use the integers in our domain axiomatization), we use *bounded* 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. Computational effectiveness is attained by requiring that in any world the set of satisfying instances of γ be finite.[3]

The formulas of LTL are interpreted over 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.

- $\langle M, s_i \rangle \models \bigcirc f_1$ iff $\langle M, s_{i+1} \rangle \models f_1$: f_1 is true 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 true.

- $\langle M, s_i \rangle \models \Box f_1$ iff for all $j \geq i$ we have $\langle M, s_j \rangle \models f_1$: f_1 is always true.

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 M. 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 $on(A, B)$. Quantification allows even greater expressiveness, 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.

We are going to use LTL formulas to express search control information (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

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

[3] That is, instead of allowing x to range over the entire domain, we only allow it to range over the elements satisfying γ. Thus, the underlying domain may be infinite, but any particular quantification over it is finite. Also we allow formulas of the form $\exists[x{:}\gamma]$ where ϕ is implicitly taken to be TRUE.

language $\mathcal{L}$ we add the following formula formation rule: if f is a formula of $\mathcal{L}$ then so is GOAL(f). If the agent's goal is expressed by the first order formula ϕ, then semantically this modality is a modality of entailment from ϕ and any state constraints. That is, GOAL(f) is true iff $\phi \wedge \psi \models f$, where ψ are the set of conditions true in every state, i.e., the set of state constraints.

However, testing if GOAL(f) is true given an arbitrary goal formula ϕ is intractable (in general, it requires theorem proving). Our implemented planning system TLPLAN allows the goal modality to be used only when goals are sets of positive literals, and it computes goal formulas by assuming that these literals describe a "goal world" using a closed world assumption. For example, if the goal is the set of literals $\{on(A, B), on(B, C)\}$ then these are assumed to be the only positive literals that are true in the goal world, every other atomic formula is false by the closed world assumption. For example $clear(A)$ is false in the goal world. Intuitively, it is not a necessary requirement of the goal. TLPLAN can then use its first-order formula evaluator over this goal world, so the formula GOAL$(\exists[y{:}on(A, y)])$ will also evaluate to true. If, however, we had as our goal $\{P(A)\}$ and the state constraint that in all states $P(A) \Rightarrow Q(A)$, then, in its current implementation, TLPLAN will incorrectly (according to the above semantics) conclude that GOAL$(Q(A))$ is false. That is, TLPLAN cannot currently handle state constraints over the goal world.

3 Expressing Search Control Information

Any LTL formula specifies a property of a sequence of states: it is satisfied by some sequences and falsified by others. 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 worlds 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 world visited by that plan using standard model checking techniques developed in the program verification area (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 the LTL formula $\Box(on(B, A) \Rightarrow on(B, A) \,\mathsf{U}\, on(C, B))$ and check its truth against candidate plans, rejecting any plans that violate this condition.

What we need is an incremental way of checking our control strategies against the partial plans generated as we search for a correct plan. If one of our partial plans violates our control strategy we can reject it, thus pruning all of its extensions from the search space.[5] We have developed a mechanism for doing incremental checking of an LTL formula. The key to this method is the progression algorithm given in Table 1. In the algorithm quantified formulas are progressed by progressing all of their instances. This algorithm is characterized by the following theorem:

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

The progression algorithm admits the following implementation strategy, used in TLPLAN.

[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 "do nothing" action. This corresponds to infinitely replicating the final world in the sequence of worlds visited by the plan.

[5] Drummond and Bresina in [DB90] explored a similar idea of using temporally extended behavioral constraints to guide forward chaining search. Their approach to expressing and using such information was however less formal and less systematic than the approach we present here.

Inputs: An LTL formula f and a world w (generated by forward-chaining).
Output: A new formula f^+, also expressed as an LTL formula, representing the progression of f through the world w.

Algorithm *Progress*(f,w)
1. **Case**
2. $f = \phi \in \mathcal{L}$ (i.e., ϕ contains no temporal modalities):
   $f^+ :=$ TRUE if $w \models f$, FALSE otherwise.
3 $f = f_1 \wedge f_2$: $f^+ := \mathit{Progress}(f_1, w) \wedge \mathit{Progress}(f_2, w)$
4. $f = \neg f_1$: $f^+ := \neg\mathit{Progress}(f_1, w)$
5. $f = \bigcirc f_1$: $f^+ := f_1$
6. $f = f_1 \,\mathsf{U}\, f_2$: $f^+ := \mathit{Progress}(f_2, w) \vee (\mathit{Progress}(f_1, w) \wedge f)$
7. $f = \Diamond f_1$: $f^+ := \mathit{Progress}(f_1, w) \vee f$
8. $f = \Box f_1$: $f^+ := \mathit{Progress}(f_1, w) \wedge f$
9. $f = \forall[x{:}\gamma]\, f_1$: $f^+ := \bigwedge_{\{c:w\models\gamma(c)\}} \mathit{Progress}(f_1(x/c), w)$
10. $f = \exists[x{:}\gamma]\, f_1$: $f^+ := \bigvee_{\{c:w\models\gamma(c)\}} \mathit{Progress}(f_1(x/c), w)$

Table 1: The progression algorithm.

Every world generated during our search of the forward-chaining space is labeled with an LTL formula f, with the initial world being labeled with a user supplied LTL control formula that expresses a control strategy for this domain. When we expand a world w we progress its formula f through w using the given algorithm, 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). If f progresses to FALSE, (i.e., f^+ is FALSE), then Theorem 3.1 shows that none of the 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.

The complexity of evaluating Clause 2 of the progression algorithm depends on the form of the world descriptions. It requires us to test an atemporal formula in the current world. Hence, its complexity depends on two things, the complexity of the atemporal formula and the form of the world description. If the worlds are incompletely described and represented simply as a first-order formula that characterizes some of its properties, then this clause will require theorem proving to evaluate. If the world is described as a set of atomic formulas and a collection of horn clauses, and if the formula is quantifier free then evaluating the formula in the world might still be tractable. The efficiency of this step is important however, as we must use this algorithm at every world expanded during plan search.

In our current implementation of TLPLAN we use worlds that are described as sets of positive literals, and we employ a closed world assumption: every positive literal not in this set is assumed to be false. In this way we need make no restrictions on the atemporal formulas that can appear in our LTL control formula. In particular, we can use arbitrary first-order formulas in our LTL control. With worlds described by (assumed to be) complete sets of positive literals, we can employ model-checking instead of theorem proving to determine the truth of these formulas in any world, and thus evaluate clause 2 efficiently.

4 Empirical Results

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 2: Blocks World operators.

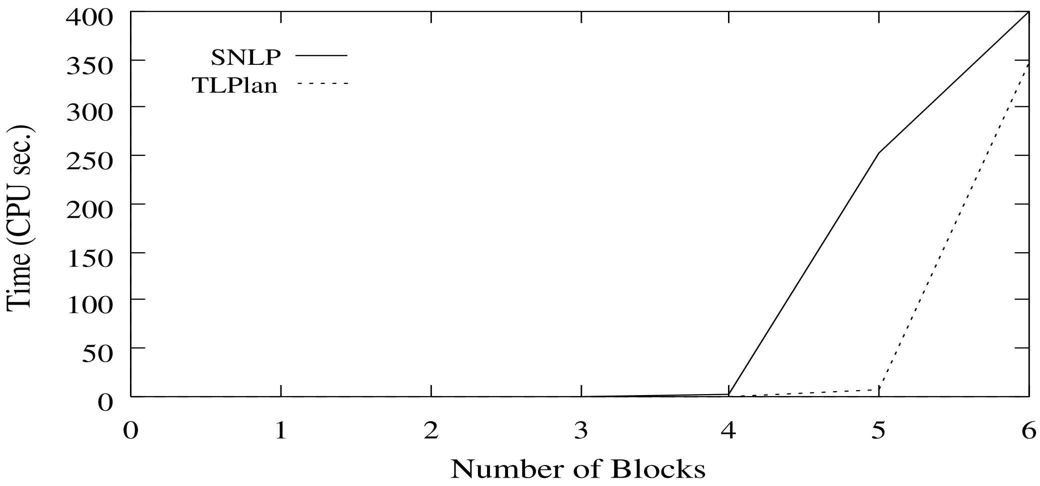

Figure 1: Performance of blind search in the blocks world

Blocks World. Our first empirical results come from the blocks world, which we describe using the four operators given in table 2. If we run our planner with the vacuous search control formula $\Box$TRUE[6], and exploring candidate plans in a depth-first manner, rejecting plans with state cycles, we obtain the performance given in Figure 1. Each data point represents the average time required to solve 5 randomly generated blocks world problems (in CPU seconds on a SUN-1000), where the initial state and the goal state were independently randomly generated. The same problems were also run using SNLP, a partial order planner [MR91, SBW90], using domain independent heuristic search. The graph demonstrates that both of these planners hit a computational wall at or before 6 blocks.[7] SNLP failed to solve 4 of the six block problems posed; the times shown on the graph include the time taken by the runs that failed.[8]

This shows that domain independent heuristic search does not work well in this domain, even for the sophisticated SNLP algorithm. Domain independent heuristics have difficult exploiting the special structure of blocks world.

Nevertheless, the blocks world does have a special structure that makes planning in this domain easy [GN92], and it is easy to come up with effective control strategies. 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 a good tower, i.e., a good base that need not be

[6]This formula is satisfied by every sequence, and hence it provides no pruning of the search space.

[7]We can note that with 5 blocks and a *holding* predicate there are only 866 different configurations of the blocks world. This number jumps to 7057 when we have 6 blocks, and to 65990 when we have 7 blocks.

[8]That is, SNLP exceeded the resource bounds we set (on the number of nodes in the search tree). Note that by including these times in the data we are making SNLP's performance seem better than it really is: SNLP would have taken strictly more time to solve these problems than the numbers indicate. The same comment applies to the tests described below.

disassembled.

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

A block x satisfies the predicate *goodtower*(x) if it is on top of a tower, i.e., it is clear, and the tower below it 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 cannot require that it be on another block y nor can it require that the robot be holding x. On the other hand, if x is on another block y, then x should not be required to be on the table, nor should the robot be required to hold it, 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 this first-order definition of a predicate (rewritten in Lisp syntax) as input. And we can then use this predicate in an LTL control formula where during the operation of the progression algorithm (Table 1) its first-order definition will be evaluated in the current world for various instantiations of its "parameter" x. Hence, we can use a strategy of preserving good towers by setting our LTL control formula to

$$\Box\Big(\forall[x{:}\textit{clear}(x)]\, \textit{goodtower}(x) \Rightarrow \bigcirc \textit{goodtowerabove}(x)\Big), \tag{1}$$

where the predicate *goodtowerabove* is defined in a manner that is symmetric to *goodtowerbelow*. In any world the formula will prune all successor worlds in which a good tower x is destroyed, either by picking up x or by stacking a new block y on x that results in a violation of a goal condition. Note also that by our definition of *goodtower*, a tower will be a good tower if none of its blocks are mentioned in the goal: such a tower of irrelevant blocks cannot violate any goal conditions. Hence, this control rule also stops the planner from considering actions that unstack towers of irrelevant blocks.

What about towers that are not good towers? Clearly they violate some goal condition. Hence, there is no point in stacking more blocks on top of them as eventually we must disassemble these towers. We can define:

$$\textit{badtower}(x) \triangleq \textit{clear}(x) \wedge \neg\textit{goodtower}(x)$$

And we can augment our control strategy to prevent growing bad towers, by using the formula:

$$\begin{aligned}
\Box\Big(&\forall[x{:}clear(x)]\, \textit{goodtower}(x) \Rightarrow \bigcirc\textit{goodtowerabove}(x)\\
&\wedge \textit{badtower}(x) \Rightarrow \bigcirc(\neg\exists[y{:}\textit{on}(y,x)]\,)\Big)
\end{aligned} \tag{2}$$

This control formula stops the placement of additional blocks onto a bad tower. With this control formula only blocks on top of bad towers can be picked up. This is what we want, as bad towers must be disassembled. However, a single block on the table that is not intended to be on the table is also a bad tower, and there is no point in picking up such a block unless its final position is ready. Adding this insight we arrive at our final control strategy for the blocks world:

$$\begin{aligned}
\Box\Big(&\forall[x{:}clear(x)]\, \textit{goodtower}(x) \Rightarrow \bigcirc\textit{goodtowerabove}(x)\\
&\wedge \textit{badtower}(x) \Rightarrow \bigcirc(\neg\exists[y{:}\textit{on}(y,x)]\,)\\
&\wedge (\textit{ontable}(x) \wedge \exists[y{:}\text{GOAL}(\textit{on}(x,y))]\, \neg\textit{goodtower}(y))\\
&\qquad \Rightarrow \bigcirc(\neg\textit{holding}(x))\Big)
\end{aligned} \tag{3}$$

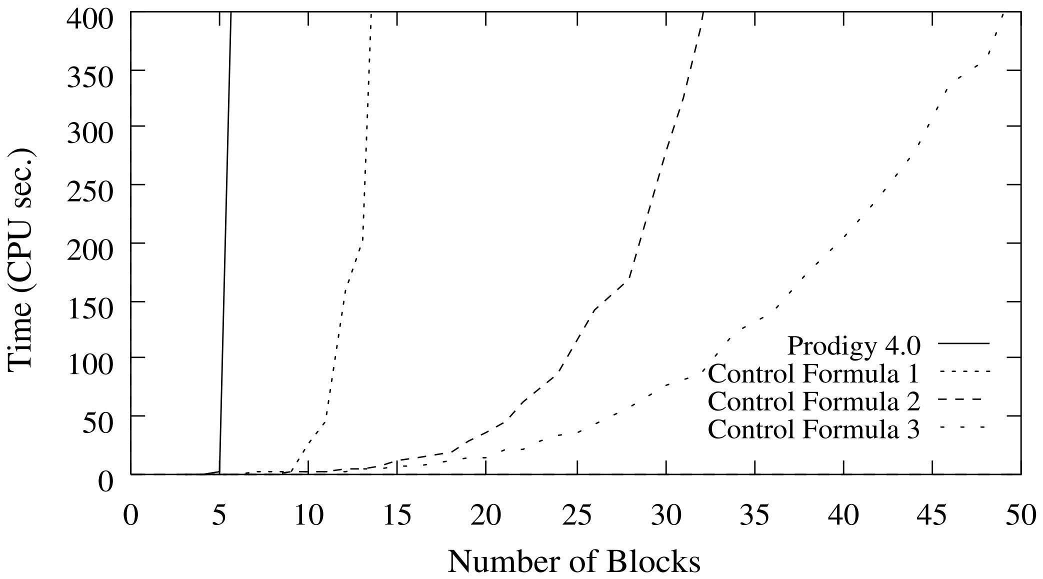

Figure 2: Performance of search control in the blocks world

The performance of our planner with these three different control formulas is shown in Figure 2. As in Figure 1 each data point represents the average time taken to solve 5 randomly generated blocks world problems. This figure also shows the performance of the PRODIGY planner on these problems. This planner, like TLPLAN, was run with a collection of hand written search control rules. In this domain our method proved to be more effective than that of PRODIGY. In fact, it is not difficult to show that the final control rule yields an $O(n^2)$ blocks world planner (where n is the number of blocks in the world). In particular, the planner can find a near optimal plan (at most twice the length of the optimal) using depth-first search without ever having to backtrack. Furthermore, there is always an optimal plan admitted by this control formula. Hence, if we employ breadth-first search our planner will find an optimal plan. However, finding an optimal plan is known to be NP-hard [GN92].

PRODIGY employed 11 different control rules some of which have similar intuitive content to our control formulas, but with others that required an understanding the PRODIGY planning algorithm. We would claim that our final control formula is easily understood by anyone familiar with the blocks world. PRODIGY does end-means analysis, so at every node in its search space it has available a world description. It would be possible to use our control formulas on this world description. However, most of the search performed by PRODIGY is a goal regressive search to find an action applicable to the current world. It is this part of the search that seems to be hard to control.

Bounded Blocks World. We have also implemented a bounded blocks world in which the table has a limited amount of space. Dealing with resource constraints of this kind is fairly easy for a forward-chaining planner, but much more difficult for partial order planners.[9] A strategy capable of generating good plans in polynomial time can be written for this domain also, but it is more complex than our strategy for the unconstrained blocks world. But even very simple strategies can be amazingly effective. With no search control at all TLPLAN took an average of 470 CPU seconds to solve 10 different randomly generated six block problems when the table had space for 3 blocks. When we changed the strategy to a simple "trigger" rule (trigger rules are rules that take advantage of fortuitous situations but do not attempt to achieve these situations) the average dropped to 0.48 seconds! The particular rule we added was: whenever a block is clear and its final position is ready, move it there immediately. This rule is easily expressed as an LTL formula. Figure 3 plots the average time taken by TLPLAN to solve 10

[9] Resource constraints are beyond the capabilities of SNLP, but see [YC94] for a partial order planner with some ability to deal with resource constraints.

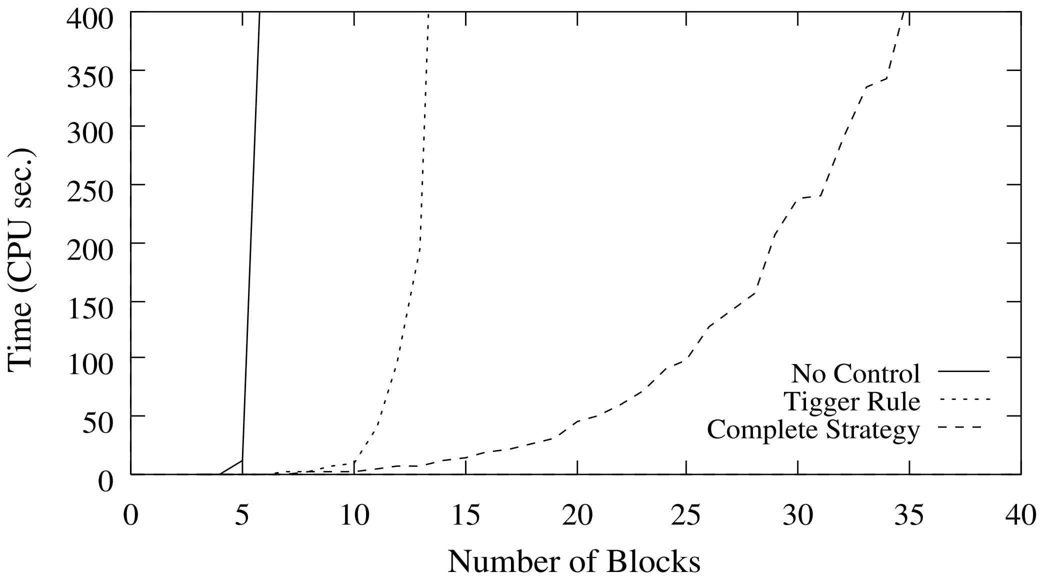

Figure 3: Performance of TLPLAN in the bounded blocks World

random n block reconfiguration problems in the bounded blocks world where the table has space for 3 blocks. The graph shows TLPLAN's performance using no control knowledge, the simple trigger rule given above, and a complete backtrack-free strategy.

Schedule World. Another domain we have tested is the PRODIGY scheduling domain. This domain has a large number of actions and the branching factor of the forward chaining space is large. Nevertheless, we were able to write a conjunction of natural control formulas that allowed TLPLAN to generate good schedules without backtracking. In this domain the task is to schedule a collection of objects on various machines to achieve various machining effects. One example of the control formulas we used is, $\forall[x{:}\mathit{object}(x)]\,\Box(\mathit{polish}(x)\wedge\bigcirc\neg\mathit{polish}(x)) \Rightarrow \bigcirc\Box\neg\mathit{polish}(x)$, where *polish* is true of an object x if x is being polished in the current world. This formula prohibits action sequences where an object is polished twice (basically the transition from being polished to when polishing stops prohibits future polishing). Another use of the control formulas was to detect impossible goals. For example the formula

$$\forall[x{:}\mathit{object}(x)]\,\forall[s{:}\mathrm{GOAL}(\mathit{shape}(x,s))]\; s = \mathit{cylindrical} \vee \Box\mathit{shape}(x,s).$$

In the domain the only shape we can create are cylinders. This formula, when progressed through the initial state, checks every object for which the goal mentions a desired shape to ensure that the desired shape is cylindrical. If it is not then the object must have that shape in the current state (i.e., in the initial state) and in all subsequent states. Hence, when we pose a goal such as $\mathit{shape}(A, \mathit{cone})$, the planner will immediately detect the impossibility of achieving this goal unless A starts off being cone shaped. The performance of TLPLAN and SNLP (using domain independent heuristics) is shown in Figure 4.[10] The data points in the figure show the average time taken to solve 10 random problems. The x-axis plots the number of new features the goal requires (i.e., the number of machining operations that must be scheduled).[11] In the experiments the number of objects are just sufficient to allow goals involving that number of new features (in this domain we can only add a limited number of new features to each object). TLPLAN was able to solve all of the problems at each data point. However, SNLP failed to solve 4 of the 3 new feature problems.

[10] We were unable to obtain the PRODIGY control rules for this domain and for the Tyre world. Hence, we omitted PRODIGY from this comparison.

[11] In this test we did not pose any impossible goals; so did not take advantage of the TLPLAN control formulas designed to detect impossible goals.

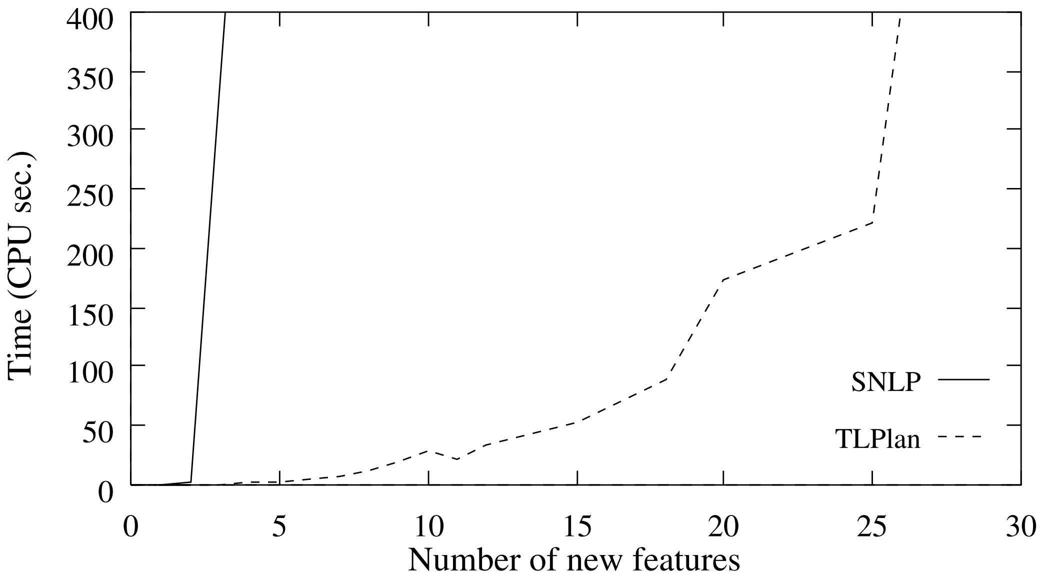

Figure 4: Performance in the Schedule world.

Tyre World. The flat tyre domain due to Russell (see [SBW90]) is the final domain we will discuss in this paper. In this domain the general task is to change a flat tire with a sequence of actions involving jacking the wheel up, loosening and tightening nuts, etc. The branching factor in this domain is large in both the forward and the backward directions. A set of TLPLAN control formulas was written that included a collection of natural strategies like not putting away useful tools (e.g., the wrench and the jack) until finished with them; not removing wheels not required to be changed by the goal; keeping the hub jacked up until the wheel has been changed, etc. All of these strategies were easily expressed in the LTL language. In this domain some subgoals, like having an intact tire, entail a number of other subgoals. So randomly selecting some number of subgoals does not yield a reasonable experiment. Instead, for every n we constructed a goal that involved more and more of the subgoals required to change a tire. (For larger n the goals required changing more than a single tire). We tested SNLP and TLPLAN on these goals. For SNLP, we could vary the difficult of the goal by permuting the ordering of the subgoals. So for SNLP we computed the average run time over a collection of randomly generated permutations of the subgoals. However, TLPLAN is *completely independent* of subgoal ordering. Hence, for TLPLAN this meant that for each n we had only a single problem. The performance of TLPLAN, and SNLP in this experiment is shown in Figure 5. The problems shown in the figure were small enough that SNLP was able to solve all of the random permutations of the subgoals over which their time average was computed. Another test we performed in this domain was to duplicate the test described by Barrett and Weld [BW94]. This problem was originally described by Russell and involved replacing a flat tire and eventually putting away the old tire and all of the tools. SNLP is above to solve this problem quickly under "trivially serializable" subgoal orderings, but can take a very long time under "non-serializable" orderings.[12] Testing a set of 10 random permutations that were trivially serializable we found that SNLP took an average of 30 CPU seconds to solve this problem on our machine. On the non-serializable orderings SNLP took over 2000 CPU seconds without finding a solution for 8 of the 10 problems. TLPLAN took around 1.3 CPU seconds on all of these orderings. TLPLAN is independent of the subgoal ordering; the variance in its performance over the different orderings was due only to dynamic changes in the environment at the time the run was executed.

It is important to note that we are not using our experiments to claim that TLPLAN is a superior planner to SNLP (or PRODIGY). After all, in the experiments described above, TLPLAN

[12] See [BW94] for a full explanation of this terminology. Basically, trivially serializable orderings are easy orders for goal-directed planners, while non-serializable orderings are hard.

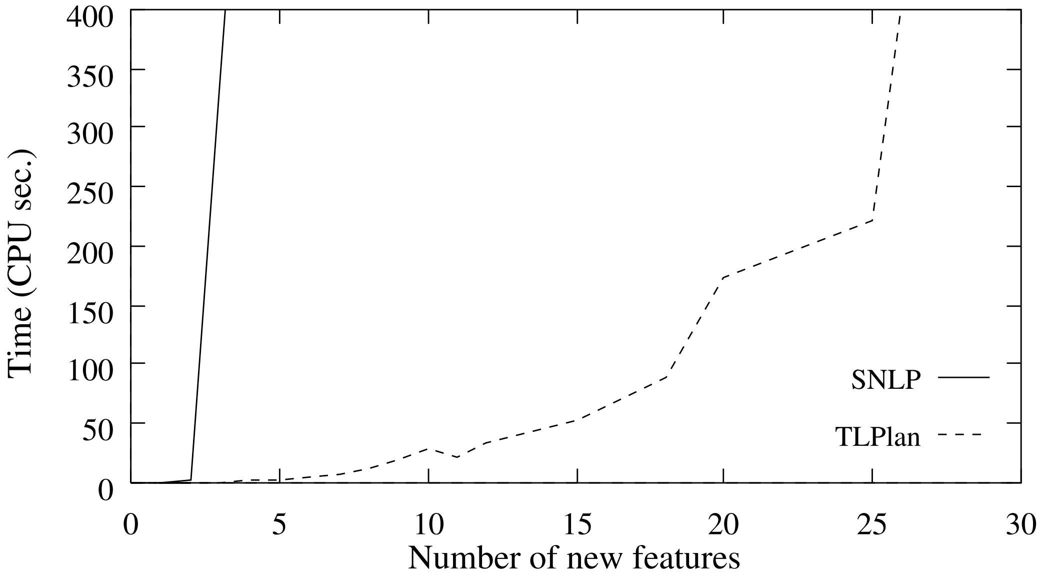

Figure 5: Performance in the Tyre world

is being run with extensive control knowledge while SNLP was not. Hence, it should be expected to outperform SNLP. What we are claiming is that (1) domain independent heuristic search in real domains is totally inadequate, (2) planning can be effective in these domains with the addition of natural domain dependent search control knowledge, and (3) the TLPLAN approach to search control knowledge in particular is an effective way to utilize such knowledge. These points are borne out by the data we have presented. We have only incomplete evidence, given from the blocks world, that our approach to specifying search control knowledge is superior to PRODIGY's. However, we are currently engaging in a more systematic comparison. What is very clear at this point however, is that our LTL control formulas are much easier to understand, and their behavior easier to predict, than PRODIGY control rules.

In conclusion, we have demonstrated that search control knowledge for forward chaining planners can be expressed in a logical formalism and utilized effectively to produce efficient planners for a number of domains. We are currently working on extending our planner so that it can plan for temporally extended goals (e.g., maintenance goals) and quantified goals. TLPLAN is not yet capable of generating plans that satisfy all such goals. This is due to its handling of eventuality goals like $\Diamond p$. If we pose the goal q, then our current implementation will search for a plan whose final state satisfies q. The temporal formula $\Diamond p$ will be used only as a search control formula, and since it involves only an eventuality that can be postponed indefinitely, this will have no pruning effect. Hence, our implementation could return a plan that achieves q in the final state but never passes through a state satisfying p. A more sophisticated implementation is required to ensure that eventualities are eventually satisfied and not postponed indefinitely. In addition, we are working on applying our planner to some practical problem domains. Finally, as mentioned above, we are working on automatically generating search control knowledge from operator descriptions along the lines of [Etz93], and on a more systematic comparison with the PRODIGY system.

Acknowledgments: thanks to Adam Grove for extensive discussions; David McAllister for insights into inductive definitions; John McCarthy for the idea of trigger rules; Manuela Veloso and Jim Blythe for help with PRODIGY; and Hector Levesque, Nir Friedman and Jeff Siskind, David Etherington, Oren Ertzioni, and Dan Weld for various useful insights and comments.

References

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

[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.

[DB90] M. Drummond and J. Bresina. Anytime synthetic projection: Maximazing probability of goal satisfaction. In *Proceedings of the AAAI National Conference*, pages 138–144, 1990.

[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.

[Etz93] Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–302, 1993.

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

[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.

[MR91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the AAAI National Conference*, pages 634–639, 1991.

[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.

[VB94] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, 1994.

[YC94] Q. Yang and A. Chan. Delaying variable binding committments in planning. In *Proceedings of the Second International Conference on AI Planning Systems*, 1994.