ELSEVIER

Artificial Intelligence 116 (2000) 123–191

Artificial Intelligence

www.elsevier.com/locate/artint

# Using temporal logics to express search control knowledge for planning ☆

Fahiem Bacchus [a,*], Froduald Kabanza [b,1]

[a] *Department of Computer Science, University of Toronto, Toronto , Ontario, Canada, M5S 1A4*
[b] *Département de Mathématiques et Informatique, Université de Sherbrooke, Sherbrooke, Quebec, Canada, J1K 2R1*

Received 11 August 1998; received in revised form 7 June 1999

**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 complexity even in simple domains. Theoretical results demonstrate that planning is in the worst case intractable. Nevertheless, planning in particular domains can often be made tractable by utilizing additional domain structure. In fact, it has long been acknowledged that domain-independent planners need domain-dependent information to help them plan effectively. In this work we present an approach for representing and utilizing domain-specific control knowledge. In particular, we show how domain-dependent search control knowledge can be represented in a temporal logic, and then utilized to effectively control a forward-chaining planner. There are a number of advantages to our approach, including a declarative semantics for the search control knowledge; a high degree of modularity (new search control knowledge can be added without affecting previous control knowledge); 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. © 2000 Elsevier Science B.V. All rights reserved.

*Keywords:* Planning; Temporal Logic; Search control knowledge

☆ This research was supported by the Canadian Government through their IRIS project and NSERC programs. A preliminary version of the results of this paper was presented at the European Workshop on AI Planning, 1995, and appears in [3]. This work was done when Fahiem Bacchus was at the University of Waterloo.

* Corresponding author. Email: fbacchus@cs.toronto.edu.

[1] Email: kabanza@dmi.usherb.ca.

0004-3702/00/$ – see front matter © 2000 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(99)00071-5

## 1. Introduction

The classical planning problem, i.e., finding a finite sequence of actions that will transform a given initial state to a state that satisfies a given goal, is computationally difficult. In the traditional context, in which actions are represented using the STRIPS representation and the initial and goal states are specified as lists of literals, even restricted versions of the planning problem are known to be PSPACE-complete [20].

Although informative, these worst case hardness results do not mean that computing plans is impossible. As we will demonstrate many domains offer additional structure that can ease the difficult of planning.

There are a variety of mechanisms that can be used to exploit structure so as to make planning easier. Abstraction and the related use of hierarchical task network (HTN) planners have been studied in the literature and utilized in planning systems [36,48,55, 62], also mechanisms for search control have received much attention. Truly effective planners will probably utilize a number of mechanisms. Hence, it is important that each of these mechanisms be developed and understood. This paper makes a contribution to the development of mechanisms for search control.

Search control is useful since most planning algorithms employ search to find plans. Planning researchers have identified a variety of spaces in which this search can be performed. However, these spaces are all exponential in size, and blind search in any of them is ineffective. Hence, a key problem facing planning systems is that of guiding or controlling search.

The idea of search control is not new—the notion of search heuristics is one of the fundamental ideas in AI. Most planning implementations use heuristically guided search, and various sophisticated heuristics have been developed for guiding planning search [24, 30]. Knowledge-based systems for search control have also been developed. In particular, knowledge bases of forward-chaining rules have been used to guide search (these are in essence expert-systems for guiding search). The SOAR system was the first to utilize this approach [38], and a refined version is a prominent part of the PRODIGY system [59]. A similar rule-based approach to search control has also been incorporated into the UCPOP implementation [7], and a more procedural search control language has also been developed [43]. A key difference between the knowledge-based search control systems and various search heuristics is that knowledge-based systems generally rely on domain-dependent knowledge, while the search heuristics are generally domain-independent.

The work reported on here is a new approach to knowledge-based search control. In particular, we utilize domain-dependent search control knowledge, but we utilize a different knowledge representation and a different reasoning mechanism than previous approaches.

In previous work, search control has utilized the *current* state of the *planning algorithm* to provide advice as to what to do next. This advice has been computed either by evaluating domain-independent heuristics on the current state of the planner, or by using the current state to trigger a set of forward-chaining rules that ultimately generate the advice.

Our approach differs. First, the control it provides can in general depend on the entire sequence of predecessors of the current state not only on the current state. As we will demonstrate this facilitates more effective search control. And second, the search control

information we use does not make reference to the state of the planning algorithm, rather it only makes reference to properties of the planning domain. It is up to the planning algorithm to take advantage of this information, by mapping that information into properties of its own internal state. This means that although the control information we utilize is domain-dependent, the provider of this information need not know anything about the planning algorithm.

Obtaining domain-dependent search control information does of course impose a significant overhead when modeling a planning domain.[2] This overhead can only be justified by increased planning efficiency. In this paper we will give empirical evidence that such information can make a tremendous difference in planning efficiency. In fact, as we will show, it can often convert an intractable planning problem to a tractable one; i.e., it can often be the only way in which automatic planning is possible.

Our work makes an advance over previous mechanisms for search control in two crucial areas. First, it provides far greater improvements to planning efficiency that previous approaches. We can sometimes obtain polynomial time planners with relatively simply control knowledge. In our empirical tests, none of the other approaches have yielded speedups of this magnitude. And second, although our approach is of course more difficult to use than domain-independent search heuristics, it seems to be much easier to use than the previous rule-based mechanisms.[3] In sum, our approach offers a lower overhead mechanism that yields superior end results.

Our approach uses a first-order temporal logic to represent search control knowledge. By utilizing a logic we gain the advantage of providing a formal semantics for the search control knowledge, and open the door to more sophisticated off-line reasoning for generating and manipulating this knowledge. In other words, we have a declarative representation of the search control knowledge which facilitates a variety of uses. Through examples we will demonstrate that this logic allows us the express effective search control information, and furthermore that this information is quite natural and intuitive.[4]

Logics have been previously used in work on planning. In fact, perhaps the earliest work on planning was Green's approach that used the situation calculus [25]. Subsequent work on planning using logic has included Rosenschein's use of dynamic logic [46], and Biundo et al.'s use of temporal logic [10–12,53]. However, all of this work has viewed planning as a theorem proving problem. In this approach the initial state, the action effects, and the goal, are all encoded as logical formulas. Then, following Green, plans are generated by attempting to prove (constructively) that a plan exists. Planning as theorem proving has to date suffered from severe computational problems, and this approach has not yet yielded an effective planner.

Our approach uses logic in a completely different manner. In particular, we are not viewing planning as theorem proving. Instead we utilize traditional planning representations for actions and states, and we generate plans by search. Theorem provers also employ search to generate plans. However, their performance seems to be hampered by the fact that they

[2] We shall argue in Section 9 that this overhead is manageable.

[3] The more recently developed procedural search control mechanisms seem to be just as hard to use [43].

[4] In fact, it can be argued that this information is no different from our knowledge of actions; it is simply part of our store of domain knowledge. Hence, there is no reason why it should not be utilized in our planning systems.

must search in the space of proofs, a space that has no clear relation to the structure of plans. [5]

In our approach we use logic solely to express search control knowledge. We then show how this knowledge can be used to control search in a simple forward-chaining planner. We explain why such a planner is particularly effective at utilizing information expressed in the chosen temporal logic. We have implemented this combination of a simple forward-chaining planner and temporal logic search control in a system we call the TLPLAN system. The resulting system is a surprisingly effective and powerful planner. The planner is also very flexible, for example, it can plan with conditional actions expressed using the full ADL language [44], and can handle certain types of resource constraints. We will demonstrate its effectiveness empirically on a number of test domains.

Forward-chaining planners have fallen out of favor in the AI planning community. This is due to the fact that there are alternate spaces in which searching for plans is generally more effective. Partial order planners that search in the space of partially ordered plans have been shown to possess a number of advantages [9,41]. And more recently planners that search over GRAPHPLAN graphs [13] or over models of propositional theories representing the space of plans [32], have been shown to be quite effective. Nevertheless, as we will demonstrate, the combination of domain-specific search control information, expressed in the formalism we suggest, and a forward-chaining planner significantly outperforms competing planners in a range of test domains. It appears that forward-chaining planners, despite their disadvantages, are significantly easier to control, and hence the ultimate choice of planning technology may still be open to question. The point that forward-chaining planners are easier to control has also been argued by McDermott [40] and Bonet et al. [14]. They have both presented planning systems based on heuristically controlled forward-chaining search. They have methods for automatically generating heuristics, but there is still considerable work to be done before truly effective control information can be automatically extracted for a particular planning problem. As a result the performance of their systems is not yet competitive with the fastest domain-independent planning systems like BLACKBOX [33] or IPP [37] (check, e.g., the performance of the HSP planning system [14] at the recent AIPS'98 planning competition [1]). In this paper we utilize domain-specific search control knowledge, and present results that demonstrate that with this kind of knowledge our approach can reach a new level of performance in AI planning.

In the rest of the paper we will describe the temporal logic we use to express domain-dependent search control knowledge. Then we present an example showing how control information can be expressed in this logic. In Section 4 we show how a planning algorithm can be designed that utilizes this information, and in Section 6 we describe the TLPLAN system, a planner we have constructed based on these ideas. To show the effectiveness of our approach we present the results of a number of empirical studies in Section 7. There has been other work on domain-specific control for planning systems, and HTN planners also employ extensive domain-specific information. We compare our approach with these

[5] The most promising approaches to planning as theorem proving have utilized insights from non-theorem proving approaches to provide specialized guidance to the theorem proving search. For example, Stephan and Biundo [53] have utilized ideas from refinement planning to guide the theorem proving process.

works in Section 9. Finally, we close with some conclusions and a discussion of what we feel are some of the important research issues suggested by our work.

## 2. First-order linear temporal logic

We use as our language for expressing search control knowledge a first-order version of linear temporal logic (LTL) [19]. The language starts with a standard first-order language, $\mathcal{L}$, containing some collection of constant, function, and predicate symbols, along with a collection of variables. We also include in the language the propositional constants TRUE and FALSE, which are treated as atomic formulas. LTL adds to $\mathcal{L}$ the following temporal modalities: $\mathsf{U}$ (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. We will call the extension of $\mathcal{L}$ to include these temporal modalities $\mathcal{LT}$.

$\mathcal{LT}$ is interpreted over sequences of worlds, and the temporal modalities are used to assert properties of these sequences. In particular, the temporal modalities have the following intuitive interpretations: $\bigcirc f$ means that $f$ holds in the next world; $\Box f$ means that $f$ holds in the current world and in all future worlds; $\Diamond f$ means that $f$ either holds now or in some future world; and $f_1 \mathsf{U} f_2$ means that either now or in some future world $f_2$ holds and until that world $f_1$ holds. These intuitive semantics are, however, only approximations of the true semantics of these modalities. In particular, the formulas $f$, $f_1$, and $f_2$ can themselves contain temporal modalities so when we say, e.g., that $f$ holds in the next world we really mean that $f$ is true of the *sequence* of worlds that starts at the next world. The precise semantics are given below.

The formulas of $\mathcal{LT}$ are interpreted over models of the form $M = \langle w_0, w_1, \ldots \rangle$ where $M$ is a sequence of worlds. We will sometimes refer to this sequence of worlds as the timeline. Every world $w_i$ is a model for the base first-order language $\mathcal{L}$. Furthermore, we require that each $w_i$ share the same domain of discourse $D$. A constant domain of discourse across all worlds allows us to avoid the difficulties that can arise when quantifying into modal contexts [23].

We specify the semantics of the formulas of our language with the following set of interpretation rules. Let $w_i$ be the $i$th world in the timeline $M$, $V$ be a variable assignment function that maps the variables of $\mathcal{LT}$ to the domain $D$, and $f_1$ and $f_2$ be formulas of $\mathcal{LT}$.

- If $f_1$ is an *atomic* formula then $\langle M, w_i, V \rangle \models f_1$ iff $(w_i, V) \models f_1$. That is, atomic formulas are interpreted in the world $w_i$ under the variable assignment $V$ according to the standard interpretation rules for first-order logic.
- The logical connectives are handled in the standard manner.
- $\langle M, w_i, V \rangle \models \forall x.f_1$ iff $\langle M, w_i, V(x/d) \rangle \models f_1$ for all $d \in D$, where $V(x/d)$ is a variable assignment function identical to $V$ except that it maps $x$ to $d$.
- $\langle M, w_i, V \rangle \models f_1 \mathsf{U} f_2$ iff there exists $j \geqslant i$ such that $\langle M, w_j, V \rangle \models f_2$ and for all $k$, $i \leqslant k < j$ we have $\langle M, w_k, V \rangle \models f_1$: $f_1$ is true until $f_2$ is achieved.

- $\langle M, w_i, V\rangle \models \bigcirc f_1$ iff $\langle M, w_{i+1}, V\rangle \models f_1$: $f_1$ is true in the next state.
- $\langle M, w_i, V\rangle \models \Diamond f_1$ iff there exists $j \geqslant i$ such that $\langle M, w_j, V\rangle \models f_1$: $f_1$ is eventually true.
- $\langle M, w_i, V\rangle \models \Box f_1$ iff for all $j \geqslant i$ we have $\langle M, w_j, V\rangle \models f_1$: $f_1$ is true in all states from the current state on.

Finally, we say that the model $M$ satisfies a formula $f$ ($M \models f$) iff $\langle M, w_0, V\rangle \models f$ (i.e., the formula must be true in the initial world). It is not difficult to show that if $f$ has no free variables then the specific variable assignment function $V$ is irrelevant.

### 2.1. Discussion

One of the keys to understanding the semantics of temporal formulas is to realize that the temporal modalities move us along the timeline. That is, the formulas that are inside of a temporal modality are generally interpreted not at the current world, $w_i$, but at some world further along the sequence, $w_j$ with $j \geqslant i$. This can be seen from the semantic rules given above. The expressiveness of $\mathcal{LT}$ arises from its ability to nest temporal modalities and thus express complex properties of the timeline.

Another point worth making is that both the eventually and always modalities are in fact equivalent to until assertions. In particular, $\Diamond\phi \equiv \text{TRUE}\ \mathsf{U}\ \phi$. That is, since TRUE is "true" in all states we see that the until formula simply reduces to the requirement that $\phi$ eventually hold either now or in the future. Always is the dual of eventually: $\Box\phi \equiv \neg\Diamond\neg\phi$. That is, no state now or in the future can falsify $\phi$.

Finally, it should be noted that quantifiers require that the subformulas in their scope be interpreted under a modified variable assignment function (this is the standard manner in which quantifiers are interpreted). Since we can quantify into temporal contexts this means that variable can be "bound" in the current world $w_i$ and then "passed on" to constrain future worlds.

**Example 1.** Here are examples of what can be expressed in $\mathcal{LT}$.

- If $M \models \bigcirc\bigcirc on(A, B)$, then $A$ must be on $B$ in the third world of the timeline, $w_2$.
- If $M \models \Box\neg holding(C)$, then at no world in the timeline is it true that we are holding $C$.
- If $M \models \Box(on(B, C) \Rightarrow (on(B, C)\ \mathsf{U}\ on(A, B)))$, then whenever we enter a world in which $B$ is on $C$ it remains on $C$ until $A$ is on $B$, i.e., along this timeline $on(B, C)$ is preserved until we achieve $on(A, B)$.
- If $M \models \Box(\exists x.on(x, A) \Rightarrow \bigcirc\exists x.on(x, A))$, then whenever something is on $A$ there is something on $A$ in the next state. This is equivalent to saying that once something is on $A$ there will always be something on $A$. Note that in this example the scope of the quantifier does not extend into the "next" modal context. Hence, this formula would be true in a timeline in which there was a different object on $A$ in every world.
- If $M \models \forall x.ontable(x) \Rightarrow \Box ontable(x)$, then all objects that are on the table in the initial state remain on the table in all future states. In this example we are quantifying into a modal context, binding $x$ to the various objects that are on the table in the initial world and passing these bindings onto the future worlds.

We need two additions to our language $\mathcal{LT}$. The first extension introduces an additional modality, that of a "goal", while the second extension is a syntactic one.

### 2.2. The GOAL modality

We are going to use $\mathcal{LT}$ formulas to express domain-dependent strategies for search control. We are trying to control search so as to find a solution to a goal; hence, the strategies will generally need to take into account properties of the goal. In our empirical tests we have found that making reference to the current goal is essential in writing effective control strategies.

To facilitate this we augment our language with an additional modality, a *goal modality*. The intention of this modality is to be able to assert that certain formulas are true in every goal world. Syntactically we add the following formula formation rule to the rules we already have: if $f$ is a pure first-order formula containing no temporal or GOAL modalities, then GOAL($f$) is a formula of $\mathcal{LT}$. GOAL($f$) can thus subsequently appear as a subformula of a more complex $\mathcal{LT}$ formula. To give semantics to these formulas we augment the models of our language $\mathcal{LT}$ so that they become pairs of the form $(M, G)$, where $M$ is a timeline as described above, and $G$ is a set of worlds $w$ with domain $D$. Intuitively, $G$ is the set of *all* worlds that satisfy the agent's goal, i.e., the agent wants to modify its current world so as to reach a (any) world in $G$. Now we add the following semantic interpretation rule to the ones given above:

- $(\langle M, w_i, V\rangle, G) \models \text{GOAL}(f_1)$ iff for all $w \in G$ we have $(w, V) \models f_1$.[6]

Finally, if $f$ is a formula in the full language (i.e., the language $\mathcal{LT}$ with the goal modality added) containing no free variables, then we say that the model $(M, G)$ satisfies $f$, $(M, G) \models f$, iff $(\langle M, w_0\rangle, G) \models f$. From now on we will use $\mathcal{LT}$ to refer to the full language generated by the formation rules given above *including* the formation rule that allows use of the goal modality.

For example,

$$\forall x, y.on(x, y) \wedge clear(x) \wedge \text{GOAL}(on(x, y) \wedge clear(x)) \Rightarrow \bigcirc(on(x, y) \wedge clear(x))$$

is a syntactically legal formula in the augmented language. This formula is satisfied in a model $(M, G)$ iff for every pair of objects $x$ and $y$ such that (1) $on(x, y) \wedge clear(x)$ is true in $w_0$ and (2) $on(x, y) \wedge clear(x)$ is true in every $w \in G$, we have that $on(x, y) \wedge clear(x)$ is true in $w_1$ (the next world in the timeline $M$). On the other hand, $on(A, B) \wedge \text{GOAL}(\bigcirc(clear(B)))$ is not a well formed formula, as we cannot apply GOAL to a formula containing a temporal modality.

Note that our syntax allows GOAL formulas to be nested inside of temporal modalities (but not vice versa). For example,

$$\Box(\exists x, y.ontable(x) \wedge on(x, y) \wedge \text{GOAL}(on(x, y)))$$

is a syntactically legal formula. It says that in every world in the timeline there must exist a pair of blocks $x$ and $y$ such that $x$ is on the table and $y$ is on $x$, and such that it is true

[6] Remember that each $w$ is a first-order model for $\mathcal{L}$ and $V$ is a variable assignment. Hence $(w, V) \models f_1$ can be decided by the standard interpretation rules for first-order formulas.

in every goal world that $x$ is on $y$. Note that for any particular instantiation of $x$ and $y$, GOAL($on(x, y)$) will be either be true in every world in the timeline or false in every world of the timeline: the semantics of GOAL makes GOAL formulas independent of the timeline. However, the set of instantiations of $x$ and $y$ for which we require GOAL($on(x, y)$) to be true might change in every world of the timeline due to the outermost always modality. In particular, the formula will be true even if a completely different pair of blocks satisfies

$$\exists x, y.ontable(x) \wedge on(x, y) \wedge \text{GOAL}(on(x, y))$$

at each world $w_i$ of the timeline.

It can be noted that if we assert GOAL($\phi$) (i.e., that $\phi$ is our goal), then we will also have GOAL($\psi$) for any $\psi$ logically entailed by $\phi$. (Clearly if $\psi$ must be true in any world in which $\phi$ is true, then $\psi$ must be true in all $w \in G$ as well.)

### *2.3. Bounded quantification*

In Section 4.2 we will demonstrate one method by which information expressed in our temporal logic can be used computationally. To facilitate such usage, we eschew standard quantification and use *bounded* quantification. Hence, it is convenient at this point to introduce some addition syntax. For now we will take bounded quantification to be a purely syntactic extension. Later we will see that some additional restrictions are required to achieve computational effectiveness.

**Definition 2.1.** Let $\phi$ be any formula. Let $\gamma$ be any *atomic* formula or any atomic formula inside of a GOAL modality. The bounded quantifiers are defined as follows:

(1) $\forall[x{:}\gamma(x)]\phi \stackrel{\triangle}{=} \forall x.\gamma(x) \Rightarrow \phi$.

(2) $\exists[x{:}\gamma(x)]\phi \stackrel{\triangle}{=} \exists x.\gamma(x) \wedge \phi$.

(3) For convenience we further define: $\exists[x{:}\gamma(x)] \stackrel{\triangle}{=} \exists x.\gamma(x)$.

It is easiest to think about bounded quantifiers semantically: $\forall[x{:}\gamma(x)]\phi$ holds iff $\phi$ is true for all $x$ such that $\gamma(x)$ holds, and $\exists[x{:}\gamma(x)]\phi$ holds iff $\phi$ is true for some $x$ such that $\gamma(x)$ holds. That is, the quantifier bound $\gamma(x)$ simply serves to limit the range over which the quantified variable ranges. Without further restrictions bounded quantification is just as expressive as standard quantification: simply take $\gamma(x)$ to be the propositional constant TRUE.

We can also use atomic GOAL formulas as quantifier bounds. By the above definition, $\forall[x{:}\text{GOAL}(\gamma(x))]\phi$ is an abbreviation for $\forall x.\text{GOAL}(\gamma(x)) \Rightarrow \phi$, which can be seen to have the simple semantic meaning of asserting that $\phi$ holds for every $x$ such that $\gamma(x)$ is true in every goal world.

#### *2.3.1. Two uses of the language*

We have defined a formal language that possess a declarative semantics. It is possible to use this language as a logic, i.e., to perform inference from collections of sentences, by defining a standard notion of entailment. Let $f_1$ and $f_2$ be two formulas of the full language $\mathcal{LT}$, then we can define $f_1 \models f_2$ iff for all models $(M, G)$ such that $(M, G) \models f_1$ we have that $(M, G) \models f_2$.

We will not explore the use of $\mathcal{LT}$ as a logic in this paper (except for a few words in about the possibilities in Section 10). Rather we will explore its use as a means for declaratively specifying search control information, and we will utilize its formal semantics to verify the correctness of the algorithms that utilize the control knowledge.

## 3. An extended example

In this section we demonstrate how $\mathcal{LT}$ can be used to express domain-specific information. This information can be viewed as simply being additional knowledge of the dynamics of the domain. Traditionally, the planning task has been undertaken using very simple knowledge of the domain dynamics. In particular, all that is usually specified in a planning problem is information about the primitive actions: when they can be applied and what their effects are. Given this knowledge the planner is expected to be able to construct plans. Our experience with AI planners indicates that this problem is difficult, both from the point of view of the theoretical worst case behaviour, and from the point of view of practical empirical experience.[7]

Part of the motivation for this work is our opinion that successful planning systems will have access to addition useful knowledge about the dynamics of the domain, knowledge that goes beyond a simple specification of the primitive actions. Some of this knowledge can come from the designer of the planning system, but in the long term we would expect that much of this knowledge would be learned or computed by the system itself. For example, human agents often use experimentation and exploration to gather additional knowledge in dynamical domains, and we would expect that eventually an autonomous planning system would have to do the same.

For now, however, since our work on automatically generating such knowledge is preliminary, we will explore how to utilize such knowledge given that has been provided by the designer of the planning system. In this section we will give an extended example, using the familiar blocks world, that serves to demonstrate that there is often considerable additional knowledge available to the designer. And we will advance the argument that our temporal logic $\mathcal{LT}$ serves as a useful and flexible means for representing this knowledge. In the next section we will discuss how this knowledge can be put to computational use to reduce search during planning.

**Blocks world.** Consider the standard blocks world, which we describe using the four STRIPS operators given in Table 1. Despite its age the blocks world is still a hard domain for even the most sophisticated domain-independent AI planners. Our experiments indicate (see Section 7) that generating plans to reconfigure 11–12 blocks seems to be the limit of current planners.

[7] It can be noted that the AI planning systems that have had the most practical impact have been HTN-style planners. HTN (hierarchical task network) planners utilize domain knowledge (in the form of task decomposition schema) that goes well beyond the simple knowledge of action effects utilized by classical planners [17,62]. We discuss this point further in Section 9.

Table 1
Blocks world operators

| 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*$(y)$. |

Nevertheless, the blocks world does have a special structure that makes planning in this domain easy [26,35]. And it is easy to write additional information about the dynamics of the domain, information that could potentially be put to use during planning.

The most basic idea in the blocks world is that towers can be built from the bottom up. That is, once we have built a good prefix of a tower we need never dismantle that prefix in order to finish our task.

For example, consider the planning problem shown in Fig. 1. To solve this problem it is clear that we need not unstack $B$ from $C$. This tower of blocks is what can be called a good tower, i.e., a tower that need not be dismantled in order to achieve the goal.

More generally, we can write a straightforward first-order formula that for any single world describes when a clear block sits on top of a good tower, i.e., a tower of blocks that does not need to be dismantled.

$$\begin{aligned}
&\mathit{goodtower}(x) \stackrel{\triangle}{=} \mathit{clear}(x) \wedge \neg\mathrm{GOAL}(\mathit{holding}(x)) \wedge \mathit{goodtowerbelow}(x),\\
&\mathit{goodtowerbelow}(x) \stackrel{\triangle}{=} (\mathit{ontable}(x) \wedge \neg\exists[y{:}\mathrm{GOAL}(\mathit{on}(x, y))])\\
&\quad \vee \exists[y{:}\mathit{on}(x, y)]\neg\mathrm{GOAL}(\mathit{ontable}(x)) \wedge \neg\mathrm{GOAL}(\mathit{holding}(y)) \wedge \neg\mathrm{GOAL}(\mathit{clear}(y))\\
&\quad \wedge \forall[z{:}\mathrm{GOAL}(\mathit{on}(x, z))]z = y \wedge \forall[z{:}\mathrm{GOAL}(\mathit{on}(z, y))]z = x\\
&\quad \wedge \mathit{goodtowerbelow}(y).
\end{aligned}$$

A block $x$ satisfies the predicate *goodtower*$(x)$ if it is on top of a tower, i.e., it is clear, it is not required that the robot be holding it, and the tower below it does not violate any goal conditions. The various tests for the violation of a goal condition in the tower below are given in the definition of *goodtowerbelow*. If $x$ is on the table, the goal cannot 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 the robot be required to hold $y$, 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.

We can represent our insight that good towers can be preserved using an $\mathcal{LT}$ formula. A plan for reconfiguring a collection of blocks is a sequence of actions for manipulating those blocks. As these actions are executed the world passes through a sequence of states, the states brought about by the actions. Any "good" plan to reconfigure blocks should never dismantle or destroy a good tower, i.e., it should not generate a sequence of states in which a good tower is destroyed. If a good tower is destroyed it would eventually have to be reassembled, and there will be a better plan that preserved the good tower. Formulas of

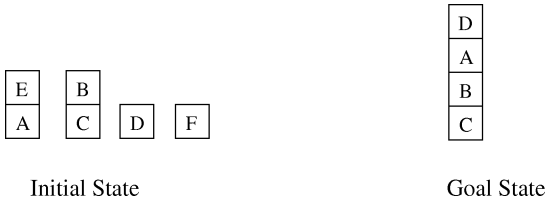


Fig. 1. A blocks world example.

$\mathcal{LT}$ specify properties of sequences of states, so we can write the following formula that characterizes those state sequences that do not destroy good towers.

$$\Box\big(\forall[x{:}\mathit{clear}(x)]\mathit{goodtower}(x) \Rightarrow \bigcirc(\mathit{clear}(x) \vee \exists[y{:}\mathit{on}(y,x)]\mathit{goodtower}(y))\big). \quad (1)$$

If a plan generates a state sequence that wastefully destroys good towers, then that state sequence will fail to satisfy this formula.

In the example given in Fig. 1, the state transitions caused by unstacking $B$ from $C$ or by stacking any block except $A$ on $B$ will violate this formula.

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 formula also rules out state sequences that wastefully dismantle towers of irrelevant blocks. The singleton tower $F$ in our example satisfies our definition of a good tower.

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:

$$\mathit{badtower}(x) \stackrel{\triangle}{=} \mathit{clear}(x) \wedge \neg\mathit{goodtower}(x).$$

And we can augment our characterization of good state sequences by ruling out those which grow bad towers using the formula:

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

This formula rules out sequences that place additional blocks onto a bad tower. Furthermore, by conjoining the new control with the previous one, the formula continues to rule out sequences that dismantle good towers. With this formula a good sequence can only pickup blocks on top of bad towers. This is what we want, as bad towers must be disassembled. In our example, the tower of blocks under $E$ is a bad tower. Hence, any action that stacks a block on $E$ will cause a state transition that violates the second conjunction of formula (2).

However, formula (2) does not rule out all useless actions. In particular, by our definitions a single block on the table that is not intended to be on the table is also a bad tower. In our example, block $D$ is such a singleton bad tower. $D$ is intended to be on block $A$ but is currently on the table. Formula (2) still permits us to pick up blocks that are on top of bad towers, however, there is no point in picking up $D$ until we have stacked $A$

on $B$. In general, there is no point in picking up singleton bad tower blocks unless their final position is ready. Adding this insight we arrive at our final characterization of good state sequences for the blocks world:

$$\begin{aligned}\Box\big(&\forall[x{:}\mathit{clear}(x)]\mathit{goodtower}(x) \Rightarrow \bigcirc(\mathit{clear}(x) \vee \exists[y{:}\mathit{on}(y,x)]\mathit{goodtower}(y))\\ &\wedge \mathit{badtower}(x) \Rightarrow \bigcirc(\neg\exists[y{:}\mathit{on}(y,x)])\\ &\wedge (\mathit{ontable}(x) \wedge \exists[y{:}\mathrm{GOAL}(\mathit{on}(x,y))]\neg\mathit{goodtower}(y))\\ &\quad\Rightarrow \bigcirc(\neg\mathit{holding}(x))\big). \end{aligned} \tag{3}$$

Although we have provided some intuitions as to how an $\mathcal{LT}$ formula like formula (3) can be used to rule out bad state sequences, there are a number of details that remain to be fleshed out. We will do this in the next two sections.

## 4. Finding good plans

In the previous sections we have provided a formal logic $\mathcal{LT}$ that can be used to assert various properties of a timeline, and we have given some examples showing that timelines violating the asserted properties are not worth exploring. To put these logical ideas to computational use we need to be more concrete about the data structures that will be used to represent the timeline models $(M, G)$, the manner in which these models are constructed, and how we can determine whether or not formulas of $\mathcal{LT}$ are satisfied or falsified by these models. In the next two sections we will provide these details.

We will utilize an extended version of the standard STRIPS database representation of the individual worlds of a timeline. In this representation each individual world is represented as a complete list of the ground atomic formulas that hold in that world. The closed world assumption is employed, so that every ground atomic formula not in a world's database is falsified by the world. Every planning problem provides a complete STRIPS database specification of the initial world, and actions generate new worlds by specifying a complete set of database updates that must be applied to the current world. Thus, every sequence of actions applied to the initial world generates a sequence of complete STRIPS databases.

STRIPS databases are essentially identical to traditional relational databases, and can be viewed as being finite first-order models [57]. First-order formulas can be evaluated against such models, [8] and in the next section we will provide the details of the evaluation algorithm. This algorithm allows us to determine whether or not an individual world satisfies or falsifies any first-order formula.

In general we need to deal with formulas of $\mathcal{LT}$ which go beyond standard first-order formulas by their inclusion of temporal and GOAL modalities. We will treat the GOAL modality formulas in the next section. In this section we will show how to deal with the temporal modalities. The method we will present builds on the ability to evaluate atemporal formulas on individual worlds.

[8] The fact that evaluating database queries in relational databases is essentially the same as evaluating logical formulas against finite models is a central theme in database theory [31].

Our approach to using $\mathcal{LT}$ formulas to guide planning search involves testing whether or not a candidate plan falsifies the formula. We will first formalize the notion of a plan satisfying an $\mathcal{LT}$ formula, then we will describe a mechanism for checking a plan prefix to see if all of its extensions necessarily falsify a given $\mathcal{LT}$ formula, and finally we describe a planning algorithm that can be constructed from this mechanism.

### *4.1. Checking plans*

Actions map worlds to new worlds. Hence a plan, which we take to be a finite sequence of actions, generates a finite sequence of worlds: the worlds that would arise as the plan is executed. Since each of these worlds is a standard STRIPS database, a plan in fact produces a finite sequence of first-order models—almost a suitable model for $\mathcal{LT}$.

The only difficulty is that models of $\mathcal{LT}$ are infinite sequences of first-order models. Intuitively, our plan is intended to control the agent for some finite time, up until the time the agent completes the execution of the plan. [9]

In classical planning it is assumed that the agent executing the plan is the only source of change. Since this paper addresses the issue of search control in the context of classical planning, we adopt the same assumption. This means that once execution of the plan is completed the world remains unchanged, or to use the phrase common in the verification literature, the world idles [39]. We can model this formally in the following manner:

**Definition 4.1.** Let plan $P$ be the finite sequence of actions $\langle a_1, \ldots, a_n \rangle$. Let $S = \langle w_0, \ldots, w_n \rangle$ be the sequence of worlds such that $w_0$ is the initial world and $w_i = a_i(w_{i-1})$. $S$ is the sequence of worlds visited by the plan. Then the *$\mathcal{LT}$ model corresponding to $P$ and $w_0$* is defined to be $\langle w_0, \ldots, w_n, w_n, \ldots \rangle$, i.e., $S$ extended to an infinite sequence by infinitely replicating the final world. In the verification literature this is know as *idling* the final world.

Therefore, every finite sequence of actions we generate corresponds to a *unique* model in which the final state is idling. Thus, given any formula of $\mathcal{LT}$ a given plan will either falsify or satisfy it.

**Definition 4.2.** Let $P$ be a plan and $w_0$ be the initial world. We say that $(P, w_0)$ satisfies (or falsifies) a formula $\phi \in \mathcal{LT}$ just in case the model corresponding to $P$ and $w_0$ satisfies (or falsifies) $\phi$.

Given a formula like the blocks world control formula (formula (3) above) designed to characterize good sequences of blocks world transformations, we can then check any plan to see if it is a good plan. That is, given an $\mathcal{LT}$ control formula $\phi$, we say that a plan $P$ is a good plan if $(P, w_0)$ satisfies $\phi$. Unfortunately, although it can be tractable to check

[9] Work on reactive plans [6] and policies [15,18,54] has concerned itself with on-going interactions between the agent and its environment. However, there are still many applications where we only want the agent to accomplish a task that has a finite horizon, in which case plans that are finite sequences of actions can generally suffice.

whether or not $P$ satisfies an arbitrary formula $\phi$ [10] knowing this does not directly help us when searching for a plan.

When we are searching for a plan we need to be able to test partially constructed plans, as these are the objects over which we are searching. Furthermore, we need to be able to determine if a good plan could possible arise from any further expansion of a partial plan. With such a test we will be able to mark partial plans as dead ends, pruning them from the search space; thus avoiding having to search any of their successors.

One of the contributions of this paper is a method for doing this in the space of partial plans searched by a forward-chaining planner.

### *4.2. Checking plan prefixes*

A forward-chaining planner, searches in the space of world states. In particular, it examines all executable sequences of actions that emanate from the initial world, keeping track of the worlds that arise as the actions are executed. Such sequences, besides being plans in their own right, are prefixes of all the plans that could result from their expansion.

We have developed an incremental mechanism for checking whether or not a plan prefix, generated by forward-chaining, could lead to a plan that satisfies an arbitrary $\mathcal{LT}$ formula. Our method is subject to the restriction that all quantifiers in the formula must range over finite sets, i.e., the quantifier bounds in the formula must specify finite sets. Clearly this restriction is satisfied when as in our application worlds are specified by finite STRIPS databases and the quantifier bounds are atomic formulas involving described predicates. [11] The key to our method is the progression algorithm given in Table 2. This algorithm takes as input an $\mathcal{LT}$ formula and produces a new formula as output. As can be seen from clauses (8) and (9), the algorithm handles quantification by expanding all instances. This is where our assumption about bounded quantification comes into play; the algorithm must iterate over all instances of the quantifier bounds. [12] It can be noted that the algorithm relies on an ability to evaluate atemporal formulas against individuals worlds (case (1)), the next section will provide the algorithm for this test.

The progression algorithm also does Boolean simplification on its intermediate results at various stages. That is, it applies the following transformation rules:

(1) $[\text{FALSE} \wedge \phi | \phi \wedge \text{FALSE}] \mapsto \text{FALSE}$,
(2) $[\text{TRUE} \wedge \phi | \phi \wedge \text{TRUE}] \mapsto \phi$,
(3) $\neg\text{TRUE} \mapsto \text{FALSE}$,
(4) $\neg\text{FALSE} \mapsto \text{TRUE}$.

[10] In particular, it is tractable to check whether or not a plan satisfies various formulas when we have that
(1) there is a bound on the size of the sets over which any quantification can range,
(2) there is a bound on the depth of quantifier nesting in the formulas, and
(3) it is tractable to test at any world $w_i$ visited by the plan whether or not $w_i$ satisfies any ground *atomic* formula.

[11] As defined in Section 5 described predicates are those predicates all of whose positive instances appear explicitly in the STRIPS database.

[12] This is very much like the technique of expanding the universal base used in the UCPOP planner [61], and in a similar manner UCPOP must assume that the universal base is finite.

Table 2
The progression algorithm

**Inputs:** An $\mathcal{LT}$ formula $f$ and a world $w$.

**Output:** A new $\mathcal{LT}$ formula $f^+$ representing the progression of $f$ through the world $w$.

**Algorithm** *Progress*$(f, w)$

**Case**

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

These transformations allow the algorithm to occasionally short circuit some of its recursive calls. For example, if the first conjunct of an $\wedge$ connective progresses to FALSE, there is no need to progress the remaining conjuncts.

The key property of the algorithm is characterized by the following theorem:

**Theorem 4.3.** *Let $M = \langle w_0, w_1, \ldots\rangle$ be any $\mathcal{LT}$ model. Then, we have for any $\mathcal{LT}$ formula $f$ in which all quantification is bounded, $\langle M, w_i\rangle \models f$ if and only if $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.*

**Proof.** We prove this theorem by induction on the complexity $f$.

- When $f$ is an atemporal formula then $\langle M, w_i\rangle \models f$ iff $w_i \models f$. Line (1) of the algorithm applies, so $\mathit{Progress}(f, w_i) =$ TRUE or FALSE dependent on whether or not $w_i \models f$. Every world satisfies TRUE and none satisfy FALSE. Hence, $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$ iff $\langle M, w_i\rangle \models f$ as required.
- When $f$ is of the form $f_1 \wedge f_2$, then $\langle M, w_i\rangle \models f$ iff $\langle M, w_i\rangle \models f_1$ and $\langle M, w_i\rangle \models f_2$, iff (by induction) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f_1, w_i)$ and $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f_2, w_i)$, iff $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f_1, w_i) \wedge \mathit{Progress}(f_2, w_i)$, iff (by line (2) of the algorithm) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.
- When $f$ is of the form $\neg f_1$. This case is similar to the previous one.
- When $f$ is of the form $\bigcirc f_1$, then $\langle M, w_i\rangle \models f$ iff (by the semantics of $\bigcirc$) $\langle M, w_{i+1}\rangle \models f_1$, iff (by line (4) of the algorithm) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.
- When $f$ is of the form $f_1 \mathsf{U} f_2$, then $\langle M, w_i\rangle \models f$ iff (by the semantics of $\mathsf{U}$) there exists $w_j$ $(j \geqslant i)$ such that $\langle M, w_j\rangle \models f_2$ and for all $k$ $(i \leqslant k < j)$ $\langle M, w_k\rangle \models f_1$, iff $\langle M, w_i\rangle \models f_2$ ($f_2$ is satisfied immediately) or $\langle M, w_i\rangle \models f_1$ and $\langle M, w_{i+1}\rangle \models f$

(the current state satisfies $f_1$ and the next state satisfies the entire formula $f$), iff (by induction) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f_2, w_i) \vee (\mathit{Progress}(f_1, w_1) \wedge f)$, iff (by line (5)) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.

- When $f$ is of the form $\Box f_1$ or $\Diamond f_1$. Both cases are similar to previous ones.
- When $f = \forall[x{:}\gamma(x)]f_1$, then $\langle M, w_i\rangle \models f$ iff $\langle M, w_i\rangle \models f_1(x/c)$ for all $c$ such that $w \models \gamma(x/c)$, iff (by induction) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f_1(x, c), w_i)$ for all such $c$, iff (since by assumption $\gamma(x)$ is only satisfied by a finite number of objects in $w_i$) $\langle M, w_{i+1}\rangle$ satisfies the conjunction of the formulas $\mathit{Progress}(f_1(x/c), w_i)$ for all such $c$ (if there are not a finite number of such $c$ the resulting conjunction would be infinite and not a valid formula of $\mathcal{LT}$), iff (by line (8)) $\langle M, w_{i+1}\rangle \models \mathit{Progress}(f, w_i)$.
- When $f = \exists[x{:}\gamma(x)]f_1$. This case is similar to the previous one. □

Say that we wish to check plan prefixes to determine whether or not they could satisfy an $\mathcal{LT}$ formula $\phi_0$ starting in the initial world $w_0$. By Theorem 4.3, any plan starting from $w_0$ will satisfy $\phi_0$ if and only if the subsequent sequence of worlds it visits satisfies $\phi_1 = \mathit{Progress}(\phi_0, w_0)$. If $\phi_1$ is the formula FALSE, then we know that no plan starting in the world $w_0$ can possibly satisfy $\phi_1$, as no model can satisfy FALSE. Similarly, if $\phi_1$ is the formula TRUE then every plan starting in the world $w_0$ will satisfy $\phi$, as every model satisfies TRUE. Otherwise, we will have to check the subsequent sequences against the progressed formula $\phi_1$. The progression through $w_0$ serves to check the null plan, which is a prefix of every plan.

Now suppose we apply action $a$ in $w_0$ generating the successor world $w_1$. If we compute $\phi_2 = \mathit{Progress}(\phi_1, w_1)$, then we know that any plan starting with the sequence of worlds $\langle w_0, w_1\rangle$ (i.e., any plan starting with the action $a$) will satisfy $\phi_0$ if and only if the sequence of worlds it visits after $w_1$ satisfies $\phi_2$. Once again if $\phi_2$ is FALSE then no such plan can satisfy $\phi$, and if $\phi_2$ is TRUE then every such plan satisfies $\phi$. Otherwise, we will have to continue to check all extensions of the action sequence $\langle a\rangle$ against the formula $\phi_2$.

It is not difficult to see that this process can be iterated to yield a mechanism that given any plan prefix (a sequence of actions) continually updates the original formula $\phi_0$ to a new formula $\phi_i$ that characterizes the property that must be satisfied by the subsequent actions in order that the entire action sequence satisfy $\phi_0$. If at any stage the progressed formula becomes one of TRUE or FALSE, we can stop, as we then have a definite answer about any possible extension of the current plan prefix. The above reasoning yields:

**Observation 4.4.** *Let $\langle w_0, w_1, \ldots, w_n\rangle$ be a finite sequence of worlds* (*generated by some finite sequence of actions $\langle a_1, \ldots, a_n\rangle$ applied to $w_0$ the initial world*). *Let $\phi_0$ be a formula of $\mathcal{LT}$ labeling $w_0$, and let $\phi_i$ be the output of Progress$(\phi_{i-1}, w_{i-1})$* (*i.e., the result of iteratively progressing $\phi_0$ through the sequence of worlds $w_0, \ldots, w_{i-1}$*). *If $\phi_n$ is the formula* FALSE, *then no sequence of worlds starting with $\langle w_0, \ldots, w_n\rangle$ can satisfy $\phi_0$.*

This shows that the progression algorithm is *sound*. That is, if it rules out a plan prefix $\langle a_1, \ldots, a_n\rangle$ then we are guaranteed that there is no extension of $\langle a_1, \ldots, a_n\rangle$ that could satisfy the original $\mathcal{LT}$ formula $\phi_0$.

Progression is a model checking algorithm: it operates by progressing an $\mathcal{LT}$ formula over a particular finite sequence of worlds (a finite prefix of a timeline); it does not reason

about timelines in general. Although progression often has the ability to give us an early answer to our question, it cannot always give us a definite answer. That is, progression is not *complete*.

In the verification literature the class of *safety* formulas has been defined [39]. These formulas are a subset of the set of linear temporal logic (LTL) formulas [13] that have the property that every violation of a safety formula occurs after a finite period of time. More precisely, $\phi_0$ is a safety formula if whenever $M \not\models \phi_0$ (i.e., a timeline $M$ falsifies $\phi_0$) there is some finite prefix of $M$ such that all extensions of this prefix falsify $\phi_0$. More generally, we might have a formula $\phi_0$ that is the conjunction of a safety formula and a liveness (non-safety) formula. In this case, some prefixes will falsify the safety component of $\phi_0$, and some infinite timelines will falsify the liveness component of $\phi_0$. [14]

The progression algorithm has, to a certain extent, the ability to model check the safety component of the initial $\mathcal{LT}$ formula $\phi_0$. In particular, the progression algorithms has the ability to detect some of the finite prefixes that falsify the safety component of $\phi_0$. The algorithm is not, however, complete, so there may be plan prefixes all of whose extensions falsify $\phi_0$ that are not ruled out by progression (i.e., $\phi_0$ might not progress to FALSE on these prefixes).

There are two components to this incompleteness. First, progression cannot check the liveness component of formulas. For example, it cannot check liveness requirements like the achievement of eventualities in the formula. Given just the current world it does not have sufficient information to determine whether or not these eventualities will be achieved in the future. So it could be that all the extensions of a particular plan prefix fail to satisfy the liveness requirements of $\phi_0$. This cannot be detected by the progression algorithm. For example, one of the actions in the prefix might use up an unrenewable resource that is needed to satisfy one of the liveness requirements): progression will not be able to detect this. [15]

The second source of incompleteness arises from the fact that progression does not employ theorem proving. For example, if we have the $\mathcal{LT}$ formula $\Diamond\phi$ where $\phi$ is unsatisfiable, then progressing this formula will never discover this. When we apply progression we obtain $\mathit{Progress}(\phi, w) \vee \Diamond\phi$. The algorithm may be able to reduce $\mathit{Progress}(\phi, w)$ to FALSE, but then it would still be left with the original formula $\Diamond\phi$. It will not reduce this formula further. We know that that no world in the future can ever satisfy an unsatisfiable formula, so from the semantics of $\Diamond$ we can see that in fact no plan can satisfy this formula. In general, detecting if $\phi$ is unsatisfiable requires a complete theorem prover. The advantage of giving up this component of completeness is computational efficiency. Ignoring quantification, the progression algorithm has complexity linear in the size of the formula (assuming that the tests in line (1) can be performed in time linear in the length of the formula $\phi$). [16] While the complexity of validity checking for quantifier free $\mathcal{LT}$ (i.e., propositional linear temporal logic) is known to be PSPACE-complete [51].

[13] $\mathcal{LT}$ is a first-order version of LTL with an added GOAL modality.

[14] Some syntactic characterizations of safety formulas exists, but in general testing if a propositional LTL formula is a safety formula is a PSPACE-complete problem [50].

[15] Unless the user can specify a safety formula prohibiting conditions that make liveness requirements impossible to achieve.

[16] We will return to the issue of quantification and the tests in line (1) later.

It should be noted however, that progression does have the ability to detect unsatisfiable formulas when they are not buried inside of an eventuality. For example, if we have the formula $\Box\phi$ where $\phi$ is atemporal and unsatisfiable, then the progression of this formula through any world $w$ will be FALSE. The progression of $\phi$ will (due to case (2) of the algorithm) return FALSE $\wedge\ \Box\phi$ which will be simplified to FALSE. In this case progression is model checking the atemporal formula $\phi$ against the model $w$ and determining it to be falsified by $w$. This is not the same as proving that $\phi$ is falsified by every world, a process that requires a validity checker. Model checking a formula against a particular world is a much more tractable computation than checking its validity (see [27] for a further discussion of this issue).

**Example 2.** Say that we progress the formula $\Box on(A, B)$ through the world $w$ in which $on(A, B)$ is true. This will result in the formula TRUE $\wedge\ \Box on(A, B)$, which will be reduced to $\Box on(A, B)$. On the other hand, says that $w$ falsifies $on(A, B)$, then the progressed formula would be FALSE $\wedge\ \Box on(A, B)$, which will be reduced to FALSE. This example shows that $\Box$ formulas generate a test on the current world and propagate the test to the next world.

As another example say that we progress the formula $\Box(on(A, B) \Rightarrow \bigcirc clear(A))$ through the world $w$ in which $on(A, B)$ is true. The result will be the formula $\Box(on(A, B) \Rightarrow \bigcirc clear(A)) \wedge clear(A)$. That is, the always test will be propagated to the next world, and in addition the next world will be required to satisfy $clear(A)$ since $on(A, B)$ is currently true. On the other hand, if $w$ falsified $on(A, B)$ the progressed formula would simply be $\Box(on(A, B) \Rightarrow \bigcirc clear(A))$. That is, we would simply propagate the constraint without any additional requirements on the next world.

It is possible to add to the progression algorithm an "idling" checking algorithm so that we can receive a definite answer to the question of whether or not a plan prefix satisfies an $\mathcal{LT}$ formula in the sense of Definition 4.2, see [2] for details. However, for the purposes of search control this is not necessary. In particular, the plan prefixes we are checking are not the final plan; all that we want to know is if they could possibility lead to a good final plan. For this purpose the partial information returned by progression is sufficient.

### *4.3. The planning algorithm*

The progression algorithm admits the planning algorithm shown in Table 3. This algorithm is used in the TLPLAN system described in Section 6.

The algorithm is described non-deterministically, search will have to be performed to explore the correct choice of action $a$ to apply at each world $w$. This algorithm is essentially a simple forward-chaining planner (a progressive world-state planner by the terminology of [61]). The only difference is that every world is labeled with an $\mathcal{LT}$ formula $f$, with the initial world being labeled with a user supplied formula expressing a control strategy for this domain. When we expand a world $w$ we progress its formula $f$ through $w$ using the progression 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 4.3 shows that none of the

Table 3
The planning algorithm

**Inputs:** A world $w$, an $\mathcal{LT}$ formula $f$ a goal $G$, a set of domain actions $A$, and the current plan prefix $P$. To start planning we call TLPLAN($w_0$,$f_0$,$G$,$A$,$\langle\rangle$), where $w_0$ is the initial world, $f_0$ is the initial $\mathcal{LT}$ control formula, and the current plan prefix is empty.

**Output:** A plan (a sequence of actions) that will transform $w_0$ into a world that satisfies $G$.

**Algorithm** TLPLAN($w$, $f$, $G$, $A$, $P$)

(1) **if** $w$ satisfies $G$ then **return** $P$.

(2) Let $f^+ = \mathit{Progress}(f, w)$.

(3) **if** $f^+$ is FALSE **return** failure.

(4) **choose** an action $a$ from the set of actions $A$ whose preconditions are satisfied in $w$.

(5) **if** no such action exists **return** failure.

(6) Let $w^+$ be the world that arises from applying $a$ to $w$.

(7) **return** TLPLAN($w^+$, $f^+$, $G$, $A$, $P + a$).

sequences of worlds emanating from $w$ can satisfy our $\mathcal{LT}$ formula. Hence, we can immediately mark $w$ as a dead-end in the search space and avoid exploring any of its successors.

## 5. Evaluating atemporal formulas in individual worlds

The previous section showed how we can checks plan prefixes to determine whether or not they (and thus all of their extensions) falsify an initial $\mathcal{LT}$ control formula. The process was proved to be sound but incomplete. [17] The progression algorithm makes two assumptions:

(1) each of the plan prefixes generated during search consist of sequences of first-order models, and
(2) for any formula $\phi$ of $\mathcal{LT}$ containing no temporal modalities and any of the models in this sequence $w$ we can determine whether $w$ satisfies $\phi$ (case (1) of the algorithm).

In this section we will show that these assumptions are satisfied in the planning system we construct. First, as mentioned in the previous section, we represent each state in the plan as a STRIPS database (with some extensions described below) and once the closed world assumption is employed such databases are formally first-order models. [18] Furthermore, our actions are modeled as performing database updates (this also follows the STRIPS

[17] Note that incompleteness does not pose a fundamental difficulty in our approach. Incompleteness means that we fail to prune away some of the invalid plan prefixes. The real issue, however, is whether or not we can prune away a sufficient number of prefixes to make search more computationally feasible. In Section 7 we will provide extensive evidence that we can.

[18] The pure STRIPS database is a finite first-order model. However the evaluator we describe below also has facilities to range variables over any finite set of integers and to evaluate numeric predicates and functions. This implies that our system is actually implicitly dealing with infinite models. In particular, it is checking formulas over a first-order model determined by the STRIPS database conjoined with the integers.

model). Thus each action maps a database to a new database, i.e., a first-order model to a new first-order model. Hence, assumption (1) is trivially satisfied—each plan prefix consists of a sequence of first-order models.

To satisfy assumption (2) we simply need to specify an algorithm for evaluating atemporal $\mathcal{LT}$ formulas in these models (STRIPS databases). The formula evaluator algorithm is specified below. Once these two assumptions are satisfied we have that the progression algorithm is sound (in the sense of Observation 4.4).

The planning algorithm specified in Table 3 searches for plans that transforms the initial world to a world satisfying the goal. It searches for this plan in the space of action sequences emanating from the initial world and eliminating from that search space some set of plan prefixes. We have the guarantee that any plan prefix eliminated from the search space has no extension satisfying the initial $\mathcal{LT}$ formula.

The planning algorithm is trivially sound. That is, if a plan is found then that plan does in fact correctly transform the initial state to a state satisfying the goal.[19] The planning algorithm will be complete, i.e., it will return a plan if one exists, when

(1) the underlying search algorithm is complete, and

(2) whenever a plan exists a plan that does not falsify the initial $\mathcal{LT}$ formula exists.[20]

The formula evaluator checks the truth of formulas in individual worlds. Each world is represented as an extended version of a STRIPS database. In particular, there are a distinguished set of predicates called the *described* predicates. Each world has a database containing all positive ground instances of the described predicates that hold in the world. The closed world assumption is employed to derive the negations of ground atomic facts.

In addition to the described predicates a world might also include a set of described functions. These also are specified by a database, a database storing the value the described function has given various arguments.

Actions map worlds to worlds, and their effects are ultimately specified as updates to the *described* predicates functions. This is the standard operational semantics for STRIPS actions, and in fact these semantics are also applicable to ADL actions.[21]

Building on the database of described predicates we add defined predicates and functions. These are predicates and functions whose value is defined by a first-order formula. And we also add computed predicates, functions and generators. These are mainly numeric predicates and functions that rely on computations performed by the underlying hardware. Thus the evaluator can evaluate complex atemporal formulas that involve symbols not appearing in the underlying database of described predicates.

The formula evaluator is given in Tables 4–6.

The lowest level of the recursive algorithm is *EvalTerm* (Table 6) which is used to convert complex first-order terms (containing functions and variables) into constants. Variables are easy, we simply look up their value in the current set of variable bindings.

[19] Actions have a precise representation and a precise operational semantics (discussed below). The plan returned will be correct under specific interpretation of the action effects.

[20] The first condition is a standard one for any algorithm that employs search. The second condition means that it is up to the user to specify sensible control knowledge, i.e., control knowledge that only eliminates redundant plans.

[21] ADL actions can have more complex first-order preconditions along with conditional add/deletes. However, the set of add/deletes each action generates is always a set of ground atomic facts.

Table 4
The formula evaluator

**Inputs:** An atemporal $\mathcal{LT}$ formula $f$, a world $w$, and a set of variable bindings $V$.

**Output:** TRUE or FALSE dependent on whether or not $(w, V) \models f$.

**Algorithm** *Eval*$(f, w, V)$

**Case**

(1) $f = P(t_1, \ldots, t_n)$ (an atomic formula)

**return** (*EvalAtomic*$(P($*EvalTerm*$(t_1, w, V), \ldots,$ *EvalTerm*$(t_n, w, V))), w)$

(2) $f = f_1 \wedge f_2$: **if not** *Eval*$(f_1, w, V)$ **then return** (FALSE)

**else return** (*Eval*$(f_1, w, V)$)

(3) $f = \neg f_1$: **return** (**not** *Eval*$(f_1, w, V)$)

(3.1) Similar processing for the other boolean connectives.

(4) $f = \forall[x{:}\gamma(x)]f_1$:

generator := make-generator$(\gamma(x), w, V)$

*tval* := TRUE

**while** $(c :=$ generator.next$() \wedge$ *tval*$)$

*tval* := *tval* $\wedge$ *Eval*$(f_1, w, V \cup \{x = c\})$

**return** (*tval*)

(5) $f = \exists[x{:}\gamma]f_1$:

generator := make-generator$(\gamma(x), w, V)$

*tval* := FALSE

**while** $(c :=$ generator.next$() \vee \neg$*tval*$)$

*tval* := *tval* $\vee$ *Eval*$(f_1, w, V \cup \{x = c\})$

**return** (*tval*)

(6) $f = (x := t)$:

$V(x) :=$ *EvalTerm*$(t, w, V)$

**return** (TRUE)

It is not hard to see that as long as the top level formula passed to *Eval* contains no free variables (i.e., it is a sentence), the set of bindings will have a value for every variable by the time that variable must be evaluated. [22]

*EvalTerm* allows for three types of functions: computed, described and defined functions. Computed functions can invoke arbitrary computations on a collection of constant arguments (the arguments to the function are evaluated prior to being passed as arguments). The value of the function can depend on the current world or the function may be independent of the world. For example, it is possible to declare all of the standard arithmetic functions to be computed functions. Then when the evaluator encounters a term like $t_1 + t_2$ it first recursively evaluates $t_1$ and $t_2$ and then invokes the standard addition function to compute their sum.

[22] The quantifier clauses in *Eval* will set the variable values prior to their use.

Table 5
Evaluating atomic formulas

**Inputs:** An ground atomic formula $P(c_1, \ldots, c_n)$ and a world $w$.

**Output:** TRUE or FALSE dependent on whether or not $w \models P(c_1, \ldots, c_n)$.

**Algorithm** *EvalAtomic*$(P(c_1, \ldots, c_n), w)$

**Case**

(1) $P$ is a described predicate:

**return** (*lookup*$(P(c_1, \ldots, c_n), w)$)

(2) $P$ is defined by a computed predicate:

**return** $(P(c_1, \ldots, c_n, w))$.

(3) $P$ is defined by the formula $\phi$:

Let $x_1, \ldots, x_n$ be the arguments of $\phi$.

**return** $(Eval(\phi, w, V \cup \{x_1 = c_1, \ldots, x_n = c_n\}))$.

Table 6
Evaluating terms

**Inputs:** A term $t$, a world $w$, and a set of variable bindings $V$.

**Output:** A constant that is the value of $t$ in the world $w$.

**Algorithm** *EvalTerm*$(t, w, V)$

**Case**

(1) $t = x$ where $x$ is a variable:

**return** $(V(x))$ (i.e., return $x$'s binding)

(2) $t = c$ where $c$ is a constant:

**return** $(c)$.

(3) $t = f(t_1, \ldots, t_k)$ where $f$ is a described function:

**return** $(lookup(f(EvalTerm(t_1, w, V), \ldots, EvalTerm(t_k, w, V)), w))$.

(4) $t = f(t_1, \ldots, t_k)$ where $f$ is a computed function:

**return** $(f(EvalTerm(t_1, w, V), \ldots, EvalTerm(t_k, w, V), w))$.

(5) $t = f(t_1, \ldots, t_k)$ where $f$ is defined by the formula $\phi$:

For $i = 1, \ldots, k$, let $c_i = EvalTerm(t_i, w, V)$, $x_i$ be the arguments for $\phi$,

and $V' = V \cup \{f = ?, x_1 = c_1, \ldots, x_k = c_k\}$

$Eval(\phi, w, V')$

**return** $(V'(f))$

Every world contains a database of values for each described function, and these functions can be evaluated by simple database lookup. The user must ensure that these function values are specified in the initial state and that the action descriptions properly

update these values. For example, in the blocks world we could specify a function *below* such that *below*($x$) is equal to the object that is below of block $x$ in the current world (using the convention that the table is below itself). The initial state would specify the initial values for *below*, and the actions *stack* and *putdown* would have to update these function values. Updating function values by an action is accomplished by utilizing the ADL representation of actions that allows for the specification of updates to function values [44].

Defined functions are functions whose value is defined by a formula. Evaluating such functions requires a recursive call to the top level of the formula evaluator. Hence, we describe the rest of the evaluator prior to describing defined functions.

The next level up from terms is the evaluation of atomic formulas (ground atomic formulas since all terms are evaluated prior to evaluating the formula). The evaluator allows for described, computed, and defined predicates. Described predicates are the standard type. Each world maintains a database of all positive instances of such predicates, and the truth of any ground instance can be determined by a database lookup. As is standard the initial state must specify all positive instances of the described predicates and the actions must specify correct adds and deletes to keep the database up to date.

Computed predicates, like computed functions, can be used to invoke an arbitrary computation (which in this case must return true or false). In this way we can include, e.g., arithmetic predicates in our formulas. For example, $weight(A) > weight(B)$, would be a legitimate formula given that *weight* has been declared to a function. The formula evaluator would first evaluate the terms $weight(A)$ and $weight(B)$ prior to invoking the standard numeric comparison function to compare the two values.

Finally, the most interesting type of predicate are the defined predicates. Like the defined functions these predicates are defined by first-order formulas. The predicate *goodtower* (defined in Section 3) is an example of a defined predicate. Defined predicates can be evaluated by simply recursively invoking the formula evaluator on the formula that defines the predicate (with appropriate modifications to the set of bindings). The key feature is that this mechanism allows us to write and evaluate recursively defined predicates. For example, we can define *above* to be the transitive closure of *on*:

$$above(x, y) \stackrel{\triangle}{=} on(x, y) \vee \exists[z{:}on(z, y)]above(x, z).$$ [23]

At the top level the evaluator simply decomposes a formula into an appropriate set of atomic predicate queries. The decomposition is determined by the semantics of the formula connectives.

Quantifiers are treated in a special manner. As previously mentioned our implementation utilizes bounded quantification. The formula specifying the quantifier bound is restricted: it can only be an atomic formula involving a described predicate, a goal atomic formula involving a described predicate, or a special computed function. Inside of the evaluator this is implemented by using each quantifier bound to construct a generator of instances over that bound. The function make-generator does this, and every time we send the returned generator a "next" message it returns the next value for the variable. When a

[23] Of course the user has to write their recursively defined predicates in such a manner that the recursion terminates. The short circuiting of booleans and quantifiers (e.g., not evaluating the remaining disjunctions of a $\vee$ once one of the disjunctions evaluates to true) is essential to this process.

described predicate is used as a quantifier bound, a generator over its instances is easy to construct given the world's database: the generator simply returns the positive instances of that predicate contained in the database one by one. The implementation also allows for computed generators which invoke arbitrary computations to return the sequence of variable bindings.[24] There is considerable generality in the implementation. $N$-ary predicates can be used as generators. Such generators will bind tuples of variables; e.g., when evaluating the formula "$\forall[x, y{:}on(x, y)]\ldots$" a generator of all pairs $(x, y)$ such that $on(x, y)$ holds in the current world will be constructed. The generators will also automatically take into account previously bound variables. For example, when evaluating "$\forall[x{:}clear(x)]\exists[y{:}on(x, y)]\ldots$", the outer generator will successively bind $x$ to each clear block and the inner generator will bind $y$ to the single block that is below the block currently bound to $x$.

The last clause of the formula evaluator algorithm is used to deal with defined functions. We will discuss defined functions in Section 6.1.1.

### 5.1. *Evaluating* GOAL *formulas*

As mentioned in Section 2.2 the language utilized to express control formulas (and action preconditions in ADL formulas) includes a GOAL modality. In practice the user specifies the goal as some first-order formula $\Phi$. This generally means that if we can transform the initial state to *any* world satisfying $\Phi$ we have found a solution to our problem. Hence, formally, the set of goal worlds $G$ used to interpret GOAL formulas (see Section 2.2) should be taken to be the set of all first-order models satisfying $\Phi$.

By the semantics for GOAL given in Section 2.2 and the above interpretation of the set of goal worlds, we have that GOAL($\phi$) is true iff $\Phi \models \phi$. When the temporal control formula includes a goal modality (most control formulas do), then at every world when we progress the control formula through that world we may have to invoke the evaluator to determine the truth GOAL formulas, and hence the truth of $\Phi \models \phi$ for various $\phi$. To be of use in speeding up search we must be able to efficiently evaluate GOAL formulas. In general, checking entailment (i.e., checking $\Phi \models \phi$) is not efficient.

When GOAL formulas are used in the control formulas (or as preconditions of ADL actions) we must enforce some restrictions in our implementation to ensure that they can be evaluated efficiently. In particular, if GOAL formulas are to be used we require that the goal, $\Phi$, be specified as a list of ground atomic facts involving only described predicates, $\{\alpha_1, \ldots, \alpha_k\}$, and we restrict the GOAL formulas that appear in the domain specification to be of the form GOAL($\alpha$) where $\alpha$ is an atomic formula involving a described predicate. Under these restrictions we can evaluate GOAL formulas efficiently with a simple lookup operation. Any set of ground atomic formulas has a model that falsifies every atomic formula not in the set. Hence, under these restrictions GOAL($\alpha$) will be true if and only if $\alpha \in \Phi$. We can also efficiently utilize GOAL formulas in bounded quantification: all instances in the quantifier range must be instances that explicitly appearin $\Phi$.

[24] This is often useful when we want a quantified variable to range over a finite set of integers.

**Example 3.** Let the goal be the set of ground atomic facts $\{ontable(A), clear(A)\}$.

- GOAL($ontable(A)$) will evaluate to true.
- GOAL($ontable(C)$) will evaluate to false.
- $\forall[x{:}ontable(x)]$GOAL($ontable(x)$) will evaluate to true iff all the blocks on the table in the current world are equal to $A$. The quantifier is evaluated in the current world $w$, and $x$ is successively bound to every instance satisfying *ontable* in $w$. If $x$ is bound to $A$, i.e., if $ontable(A)$ is true in $w$, then GOAL($ontable(x)$) will evaluate to true. It will evaluate to false for every other binding. Hence, the formula will be true if there are no blocks on the table in $w$ or if the only block on the table is $A$.
- $\forall[x{:}$GOAL($ontable(x)$)$]ontable(x)$, in this case the quantifier is evaluated in the goal world, and the only binding for $x$ satisfying the bound is $\{x = A\}$. Hence, this formula will evaluate to true in a world $w$ iff $A$ is on the table in $w$. There may be any number of other blocks on the table in $w$.

### 5.2. Correctness of the evaluator

Since the evaluator breaks down formulas according to their standard first-order semantics, it is not difficult to see that if it evaluates atomic formulas and quantifier bounds correctly it will immediately follow that it evaluates all formulas correctly. First we deal with the quantifier bounds:

(1) If the quantifier bound is GOAL($P(\vec{x})$), then by the previous restrictions $P$ must be a described predicate. Furthermore, a binding $\vec{c}$ for the sequence of variables $\vec{x}$ satisfies the quantifier bound if and only if $P(\vec{x}/\vec{c})$ is in the list of ground atomic facts that specifies the goal. Hence iterating over this list of facts will correctly evaluate the quantifier bound.

(2) If the quantifier bound is $P(\vec{x})$ where $P$ is a described predicate, then a binding $\vec{c}$ for the variables $\vec{x}$ satisfies the quantifier bound if and only if $P(\vec{x}/\vec{c})$ is in the world's database of positive $P$ instances. Iterating over the world's database correctly evaluates the quantifier bound.

(3) If the quantifier bound is a computed generator then whatever function the user supplies it must generate some sequence of bindings given the current world and current set of bindings. We take this sequence to be the definition of the set of satisfying instances of the quantifier bound. Thus by definition computed generators are correctly evaluated. [25]

Atomic formulas require using *EvalTerm* to evaluate the terms they contain:

(1) If the term is a variable, then its value is its the current binding which will have been set by the generator for the quantifier bound. It has been shown above that the generators operate set these bindings correctly.

(2) Constants are their own value, thus *EvalTerm* correctly evaluates such terms.

[25] The system cannot ensure the user supplied generator function implements what the user intended. Rather all it can do is provide a specific semantics as to how the output of that function will be used, and ensure that it correctly implements those semantics. This is the same approach as that taken by, e.g., programming languages compilers. The language specifies a specific semantics for every language construct and the compiler is correct if it correctly maps programs to this specified semantics. Whether or not the program implements what the user intended is a separate matter.

(3) If the term is a described function, then the STRIPS database contains all of the values of that function and a simple lookup will correctly evaluate such terms.
(4) If the term is a computed or defined function then we take the value returned to define the function. (The operational semantics of defined functions is described in the next section.) Hence, such terms are evaluated correctly by definition.

Finally, we have the atomic formulas.

(1) If the atomic formula involves a described predicate, then the STRIPS database contains all positive instances of the predicate. Such predicates can be evaluated by a simple database lookup procedure.
(2) If the atomic formula involves a defined predicate then its evaluation is can be shown to be correct by induction (with the base case being the atomic predicates that are not defined predicates). [26]
(3) If the atomic formula is a computed predicate, then again we take the value returned by the computation to define the predicate.

## 6. The TLPLAN system

We have constructed a planning system called the TLPLAN system that utilizes the planning algorithm shown in Table 3. In this section we describe the system and supply some final details about the design of the system.

TLPLAN is a very simple system, as the diagram of its components shown in Fig. 2 demonstrates. The distinct components of the system are:

- A search engine which implements a range of search algorithms.
- A goal tester that is called by the search engine to determine if it has reached a goal world. The goal tester in turn calls the formula evaluator to implement this test.
- A state expander that is called by the search engine to find all the successors of a world. The state expander in turn calls the formula evaluator to determine the actions that are applicable at a world. It also calls the formula progressor to determine the formula label of these new worlds.
- A formula progressor which implements the progression algorithm shown in Table 2. The progression algorithm uses the formula evaluator to realize line 1 of the algorithm.
- A formula evaluator which implements the algorithm shown in Tables 4–6.

Forward-chaining planners like TLPLAN are inherently simple. Nevertheless, is it worth pointing out that all of the functionality needed in such a planner can be implemented using the evaluator. As Fig. 2 shows, this is the design used by TLPLAN. A properly designed formula evaluator also provides considerable additional flexibility and expressiveness to the system, and understanding its operation provides insights into the worst case complexity of the planner's basic operations.

[26] There is a subtlety when the defined predicate is recursive. In this case we need fixpoints to give a precise semantics to the predicate. It goes beyond the scope of this paper to supply such semantics, but many approaches to this problem have been developed by those concerned with providing semantics to database queries (which can be recursive), e.g., see [58].

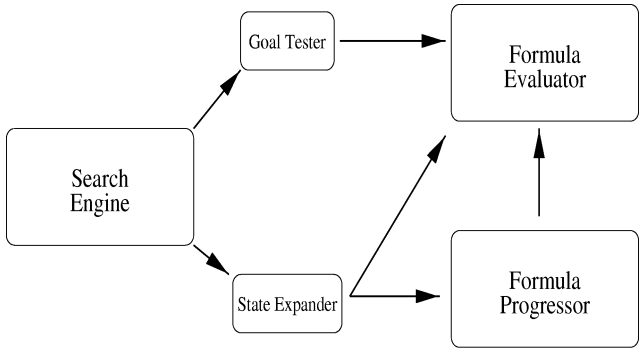


Fig. 2. The TLPLAN system.

### 6.1. Utilizing the evaluator's operational semantics

It was shown in the previous section that the formula evaluator correctly determines the truth of any atemporal $\mathcal{LT}$ formula. However, there is another way to view of the evaluator: the evaluator can be viewed as being an interpreter for a language, a language whose syntactic structure is that of the atemporal component of $\mathcal{LT}$. When viewed as an interpreter the evaluator has an operational semantics [28] that is precisely specified by the algorithm given in Tables 4–6.

In particular, when the evaluator is given a formula to evaluate it will perform a precise sequence of computations determined by the syntactic structure of the formula and the properties of the world against which the formula is being evaluated. For example, say we evaluate the formula $on(A, B) \wedge on(B, C)$ in a world $w$ in which $on(A, B)$ holds and $on(B, C)$ does not. The evaluator will perform the following computations:

(1) It will evaluate $on(A, B)$ in $w$. This will evaluate to TRUE.
(2) It will then evaluate $on(B, C)$ in $w$. This will evaluate to FALSE.
(3) It will return FALSE.

On the other hand if we evaluate the formula $on(B, C) \wedge on(A, B)$ the evaluator will never perform the computation of evaluating $on(A, B)$ in $w$.

This behaviour stems from the fact that the evaluator utilizes early termination of boolean connectives and quantifiers. From the point of view of correctness early termination makes no difference; the evaluator returns the same value no matter how the formula is written. However, early termination can be a very useful control flow mechanism when we add additional computed predicates.

Computed predicates and functions invoke a user defined computation in order to return a value to the evaluator. Great economy in the implementation can be achieved by taking advantage of this fact. In particular, much of the implementation is realized by simply supplying an additional set of computed predicates and functions. These predicates and functions return specific values, but they are mainly designed to invoke useful computation when they are evaluated.

Printing is a good example. The system defines a computed "*print*" predicate (taking an arbitrary number of arguments). This predicate always returns true as its truth value, thus

any formula containing *print* can always be rewritten by replacing all instances of *print* with TRUE. However, when the evaluator evaluates the *print* predicate the computation it invokes generates I/O as a side effect. Such predicates have a trivial declarative semantics (usually they are equivalent to the propositional constant TRUE). Their effects are determined by the computation they invoke when evaluated and the evaluator's operational semantics (which determines under what conditions they will be invoked).

For example, in the world $w$ used in the example above the formula

$$on(A, B) \wedge print(\text{“}on(A, B) \text{ holds!”}) \wedge on(B, C) \wedge print(\text{“}on(B, C) \text{ holds!”})$$

will evaluate to FALSE just as before, but its evaluation will print out the string “$on(A, B)$ holds!” as a side effect, while the formula

$$on(B, C) \wedge print(\text{“}on(B, C) \text{ holds!”}) \wedge on(A, B) \wedge print(\text{“}on(A, B) \text{ holds!”})$$

will also return FALSE but will not generate any I/O since its evaluation will terminate prior to the first print statement being evaluated.

This example shows that viewing the evaluator as an interpreter makes formula evaluation more syntax-dependent, but in no way affects the correctness of the final value it returns.[27] Despite this drawback, by utilizing the evaluator as an interpreter we can implement many of the remaining components of the planner quite easily. We discuss these components next.

*6.1.1. Defined functions*

The operational semantics of the evaluator provides a mechanism that allows the user to specify defined functions. Such functions are handled by the last clause of the formula evaluator algorithm (Table 4).

Consider a function $depth(x)$ that returns the depth of a block $x$, where clear blocks have *depth* zero. Such a function can be computed by evaluating the following formula:

$$\begin{aligned} &depth(x) \stackrel{\triangle}{=} \\ &\quad clear(x) \Rightarrow depth := 0 \\ &\quad \wedge \exists[y{:}on(y, x)] \Rightarrow depth := 1 + depth(y). \end{aligned}$$

That is, the depth of $x$ is zero if $x$ is clear, otherwise there must exist another block $y$ that is on top of $x$ and then the depth of $x$ is one more than the depth of $y$.

Formulas defining functions utilize the computed assignment predicate “:=”. This predicate is handled by the last clause of the evaluator's algorithm. Assignment always returns TRUE and as a side effect it sets the binding of a variable to be equal to the value of the term on the left hand side.

Defined functions use the convention of assigning values to the function's name as a shorthand for setting the return value. The value of the function is the last value assigned to its name. Internally, defined functions are handled by adding the function

[27] It should also be noted that if we are thinking of the evaluator as an interpreter it should not be too surprising that, e.g., the order of the conjunctions in a formula make a difference. The order of statements makes a difference in most programming languages.

name as a new unassigned variable to the set of variable bindings (see clause (5) of the *EvalTerm* algorithm Table 6). We then evaluate the formula defining the function using this augmented set of variable bindings. When the evaluator encounters an assignment "predicate" like *depth* := 0 it modifies the binding of that variable. Thus, after the evaluator has processed the defining formula, the function's name variable has been set to a value, and that value is returned as the function's value. This simple mechanism adds considerable flexibility when defining a planning domain. [28]

It should also be noted that this formula defining *depth* has been written so that its interpretation by the evaluator will yield the correct value for *depth*. In particular, the consequent of each of the implications (i.e., the assignments) will only be evaluated when the antecedent evaluates to TRUE.

*6.1.2. The progression algorithm*

As shown in Table 2 case one of the progression algorithm needs to evaluate whether or not various subformulas holds in the current world. This is accomplished by calling the formula evaluator. A useful illustration of the working of the progression algorithm and the formula evaluator is provided by the following example.

**Example 4.** Consider a control formula from the blocks world:

$$\Box\big(\forall[x{:}\mathit{clear}(x)]\mathit{ontable}(x) \wedge \neg\exists[y{:}\mathrm{GOAL}(\mathit{on}(x, y))] \Rightarrow \bigcirc(\neg\mathit{holding}(x))\big). \tag{4}$$

This formula asserts that a good plan will never pickup a block $x$ from the table if that block is not required to be on another block $y$. Say that we wish to progress this formula through a world $w$ in which the $\{\mathit{ontable}(a),\ \mathit{ontable}(b)\}$, and $\{\mathit{clear}(a),\ \mathit{clear}(b)\}$, are the set of *ontable* and *clear* facts that hold in $w$. Further, say that the goal is specified by the set $\{\mathit{on}(b, a)\}$. On encountering the $\Box$ modality the progressor will compute the progression of

$$\forall[x{:}\mathit{clear}(x)]\mathit{ontable}(x) \wedge \neg\exists[y{:}\mathrm{GOAL}(\mathit{on}(x, y))] \Rightarrow \bigcirc(\neg\mathit{holding}(x)), \tag{5}$$

and then return the conjunction of the result and the original formula (4) (case (7) of Table 2).

To progress the subformula the evaluator will be called to make a generator of the instances of $\mathit{clear}(x)$ that hold in $w$, and for each of these instances the progressor will progress the subformula

$$\mathit{ontable}(x) \wedge \neg\exists[y{:}\mathrm{GOAL}(\mathit{on}(x, y))] \Rightarrow \bigcirc(\neg\mathit{holding}(x)). \tag{6}$$

The first call to this generator will return $\{x = a\}$. Using this binding subsequent calls to the evaluator will return true for $\mathit{ontable}(x)$, and then true for $\neg\exists[y{:}\mathrm{GOAL}(\mathit{on}(x, y))]$, as there are no instantiations for $y$ that satisfy $\mathit{on}(a, y)$ in the goal world. This terminates the progression of the antecedent of the implication. Since the antecedent is true the progressor is forced to progress the consequent of the implication $\bigcirc(\neg\mathit{holding}(x))$. The end result of the first instantiation for $x$ is the progressed formula $\neg\mathit{holding}(a)$.

[28] Our implementation extends this mechanism to allow defined functions (and predicates) to have local variables that can be assigned to. Local variables are not essential, but they can speed up certain computations.

The next call to the top level generator returns the binding $\{x = b\}$. Under this new binding $ontable(x)$ evaluates to true but $\neg\exists[y{:}\text{GOAL}(on(x, y))]$ evaluates to false, as the binding $\{y = a\}$ satisfies the existential. Thus the conjunction evaluates to false, and the entire implication then progresses to true.

The final result is the formula

$$\neg holding(a) \wedge$$
$$\Box\big(\forall[x{:}clear(x)]ontable(x) \wedge \neg\exists[y{:}\text{GOAL}(on(x, y))] \Rightarrow \bigcirc(\neg holding(x))\big),$$

which says that in the subsequent state we should not be holding $a$ (remember that the progressed formula is used to label all of the successor worlds of $w$).

*6.1.3. Implementing the operators*

Actions are specified as either STRIPS or ADL operators. When we instantiate the parameters of the operators we obtain an action instance with instantiated precondition, add, and delete clauses. An action can be applied to the current world if its instantiated precondition is satisfied in the world.

It is easy to use the formula evaluator to determine if an action precondition is satisfied in the current world. However, we can go further than this. By utilizing the evaluator as an interpreter and adding some appropriate computed predicates, we can use the formula evaluator to fully implement the operators.

This process is best illustrated by an example. Consider the STRIPS operator *unstack* specified in Table 1. Its precondition list is $\{on(x, y), clear(x), handempty\}$, its add list is $\{holding(x), clear(y)\}$, and its delete list is $\{on(x, y), clear(x), handempty\}$. We can represent this operator as a formula. When this formula is evaluated in the current world it will, as a side effect of some of its computed predicates, correctly construct all of the successor worlds that could be generated by various executable instances of the operator. The formula for *unstack* is

$$handempty \wedge$$
$$\forall[x{:}clear(x)]\forall[y{:}on(x, y)]\text{MakeNewWorld}$$
$$\wedge \text{Del}(on(x, y)) \wedge \text{Del}(clear(x)) \wedge \text{Del}(handempty)$$
$$\wedge \text{Add}(holding(x)) \wedge \text{Add}(clear(y)).$$

When this formula is evaluated in the current world the first thing that is done by the evaluator is to test if *handempty* is true. If it is not then no instance of *unstack* is applicable and no further computation is necessary. Then the quantified subformulas are evaluated. The variables $x$ and $y$ will be instantiated to objects that satisfy the preconditions of the operator; i.e., by representing the operator's parameters as quantified variables we can use the standard processing of quantifiers to find all executable actions. The new computed-predicates we need are "MakeNewWorld", "Add", and "Del". "MakeNewWorld", generates a new copy of the current world, and "Add" and "Del" modify the databases that describe the instances of the various predicates that hold in that

copy.[29] (All of these predicates evaluate to TRUE in every world.) That is, by "evaluating" these predicates the world generated by applying the current action (given by the current bindings of $x$ and $y$) is computed. It is not difficult to see that any STRIPS operator can be translated into a formula of this form.[30]

Using the same mechanism it is also easy to handle ADL operators (in their full generality). ADL operators can take arbitrary first-order formulas as their preconditions, and have conditional add and delete lists. Furthermore, these operators can update function values. Every ADL operator is converted into a formula with the following form

$$\begin{aligned}
&\forall \vec{x}.\phi(\vec{x}) \\
&\quad \Rightarrow \text{MakeNewWorld} \\
&\quad \wedge \forall \vec{y}_1.\psi_1(\vec{x},\vec{y}_1) \Rightarrow \text{Add}(\ell_1(\vec{x},\vec{y}_1)) \\
&\quad\ \vdots \\
&\quad \wedge \forall \vec{y}_n.\psi_n(\vec{x},\vec{y}_n) \Rightarrow \text{Del}(\ell_n(\vec{x},\vec{y}_n)).
\end{aligned}$$

The formula $\phi(\vec{x})$ is the precondition of the operator. It contains a vector of free variables $\vec{x}$. Ever instantiation of $\vec{x}$ that makes $\phi(\vec{x})$ true in the current world specifies a single executable action. For every action all of the conditional updates $\psi_i(\vec{x},\vec{y}_i)$ are activated. Each of these conditional updates can potentially add or delete many instances of a predicate $\ell_i$. That is, for a fixed instantiation of $\vec{x}$ there may be many instantiations of $\vec{y}_i$ that satisfy the conditional update formula $\psi_i(\vec{x},\vec{y}_i)$. The action instance will add or delete an instance of the predicate $\ell_i$ for every distinct instantiation of $\vec{y}_i$ that satisfies $\psi_i(\vec{x},\vec{y}_i)$ (in the current world).

Function updates are handled in a uniform manner using equality as the predicate. That is, a term like $\text{Add}(f(c) = y)$ will update the function $f$ so that its value on $c$ is equal to $y$ (i.e., the current binding of $y$). Since functions have unique values, the add of a function value automatically deletes the old value.

The actual ADL and STRIPS operators are specified using a slightly more standard syntax and then translated to the above form. Once in this form we can make direct use of the formula evaluator to apply these operators to the current world.

### 6.2. Testing goal achievement

As discussed above the goal is usually specified as a list of ground atomic facts $\{\ell_1,\ldots,\ell_k\}$. To test if a world $w$ satisfies the goal we evaluate the conjunction $\ell_1 \wedge \cdots \wedge \ell_k$ in $w$. Thus the evaluator is used directly to test for goal achievement.

As we will point out below checking arbitrary formulas against a world is efficient. So we could in principle give the planner goals, $\Phi$, expressed as complex first-order formulas.

[29] Add and Del are syntactically unusual in that they actually take atomic formulas as arguments. If we wanted to be pedantic we would say that they are computed modalities not computed predicates.

[30] An interesting point is that since we convert operator into formulas universally quantified by the operator parameters, there is never any need to do unification. In particular, the unification algorithm plays no role in the TLPLAN system.

The planner can perform search and at every world evaluate the formula $\Phi$ in the current world to see if the goal has been achieved. This would produce a planner capable of generating plans for achieving, e.g., disjunctive or quantified goals, and in fact TLPLAN can be configured to accept an arbitrary first-order formula as its goal.

The only problem with general goals of this form is that if $\phi$ is an arbitrary formula, then checking if GOAL($\phi$) holds for various $\phi$ (i.e., checking if $\Phi \models \phi$) becomes hard (it requires theorem proving). For this reason TLPLAN does not accept formulas as goals when the domain utilizes GOAL formulas.

### *6.3. Complexity of the planner's components*

The domain specifications accepted by TLPLAN are sufficiently general so that it is quite possible to write specifications which cause the planner's basic operations to be intractable. Nevertheless, we have found that in practice the planner is very efficient in its basic operations. Since the formula evaluator is at the heart of the system, we start by examining its complexity.

#### *6.3.1. Evaluating formulas*

Evaluating a formula is usually very efficient. In particular, if $\phi$ is a quantifier free formula in which no computed or defined functions or predicates appear, then evaluating $\phi$ has complexity linear in the length of $\phi$. The basic set of described functions and predicates in $\phi$ can be evaluated in near constant time, [31] as can the boolean connectives.

When $\phi$ contains computed predicates or functions nothing can be said in general about the performance of the evaluator, since such predicates and functions can invoke arbitrary computations. In our test domains have we found computed predicates and functions to be very useful, but have never found a need to define ones that were particularly expensive to compute.

Once we allow $\phi$ to contain quantifiers, formula evaluation becomes PSPACE-complete. This is easily shown by reduction to the quantified boolean formula problem, which is known to be PSPACE-complete [29]. A quantified boolean formula is a formula of the form

$$Q_1 x_1 . Q_2 x_2 \ldots Q_k x_k (F(x_1, x_2, \ldots, x_j)),$$

where each $Q_i$ is either a universal or existential quantifier, each $x_i$ is a boolean variable, and $F$ is a boolean expression involving the $x_i$. The problem is to determine whether or not this formula is true. For example, $\forall x . \exists y . x \vee y$ is a true formula, as no matter what value $x$ takes there exists a value for $y$ (namely TRUE) that makes the formula $x \vee y$ true. On the other hand, $\forall x, y . x \vee y$ is false, as the values $x = \text{FALSE}$ and $y = \text{FALSE}$ make $x \vee y$ false.

Consider a world $w$ in which we have two predicates, a type predicate *Bool*, and a "truth" predicate $T$. The only positive instances of these two predicates true in $w$ are *Bool*(TRUE), *Bool*(FALSE), and $T$(TRUE). We can convert any quantified boolean formula

[31] Using indexing or hashing techniques we can perform these database lookups in near constant time. In the actual implementation, however, we found that a simpler albeit log-time binary tree representation of the world databases gives excellent performance.

$\phi$ to a first-order formula by replacing each universal (existential) quantifier $\forall x$ ($\exists x$) in $\phi$ by the bounded quantification $\forall[x{:}Bool(x)]$ ($\exists[x{:}Bool(x)]$). Similarly in $\phi$'s boolean expression $F(x_1, \ldots, x_n)$ we replace every variable $x_i$ by the atomic formula $T(x_i)$. For example, the quantified boolean formula $\forall x.\exists y.x \vee y$ becomes the formula

$$\forall[x{:}Bool(x)]\forall[y{:}Bool(y)]T(x) \vee T(y).$$

It is not difficult to see that the converted formula evaluates to true in the world $w$ if and only if the original quantified boolean formula was true. This shows that evaluating quantified formulas is PSPACE-hard. That the algorithm is in PSPACE is also an easy observation: although we may need to test many different sets of bindings for the quantified variables, at any stage the algorithm need store only one set of bindings.

This observation indicates that we can easily write quantified formulas that would be intractable for the formula evaluator. However, in practice things are not as bad. Let $N$ be the total number of objects in the domain, and let the deepest level of quantifier nesting in the formula $\phi$ be $k$. Then at worst, evaluating $\phi$ will take time $O(N^k)$. The PSPACE result holds because we can write formulas with $k$ nested quantifiers in length $O(k)$. Every increase in quantifier nesting adds to the size of the exponent. In practice we have rarely found a need to nest quantifier more than 3 deep, in which case evaluating these formulas remains polynomial in complexity, $O(N^3)$ in fact. The formula evaluator has not been a performance bottleneck in any of our test domains.

There is one area, however, where we must be careful about evaluating quantified formulas. As mentioned above, we determine the set of actions that can be executed in the current world by evaluating a formula in which the operator's parameters are converted into quantified variables. The way in which we convert the operator description into a quantified formula can make a significant difference in the planner's performance. This is best illustrated by an example.

Consider the formula that encodes the *unstack* operator (previously given in Section 6.1.3):

$$\begin{aligned}
&\mathit{handempty} \wedge \\
&\quad \forall[x{:}\mathit{clear}(x)]\forall[y{:}\mathit{on}(x, y)]\text{MakeNewWorld} \\
&\qquad \wedge \text{Del}(\mathit{on}, x, y) \wedge \text{Del}(\mathit{clear}, x) \wedge \text{Del}(\mathit{handempty}) \\
&\qquad \wedge \text{Add}(\mathit{holding}, x) \wedge \text{Add}(\mathit{clear}, y).
\end{aligned}$$

An alternate encoding of this operator would be the formula

$$\begin{aligned}
&\forall[x{:}\mathit{clear}(x)]\forall[y{:}\mathit{on}(x, y)]\mathit{handempty} \\
&\quad \Rightarrow \text{MakeNewWorld} \\
&\quad \wedge \text{Del}(\mathit{on}, x, y) \wedge \text{Del}(\mathit{clear}, x) \wedge \text{Del}(\mathit{handempty}) \\
&\quad \wedge \text{Add}(\mathit{holding}, x) \wedge \text{Add}(\mathit{clear}, y).
\end{aligned}$$

This formula is logically equivalent, yet far less efficient. In worlds where *handempty* is false, the evaluator can immediately recognize that no instance of *unstack* is applicable

when using the first formula. When using the second formula, however, the evaluator must iterate over every pair of objects $x$, $y$ such that *clear*$(x)$ and *on*$(x, y)$. For every iteration, the evaluation of *handempty* will fail to produce an applicable instance of *unstack*. Thus the first formula evaluates in constant time, while the second takes $O(N^2)$ where $N$ is the number of blocks in the domain.

Since the action formulas must be evaluated at every world in the forward-chaining search such differences can have a significant impact on the planner's efficiency. The issues involved in choosing which of the logically equivalent formulas to generate when converting an action into a formula (e.g., how to choose the ordering of two adjacent universal quantifiers) are essentially the same as the issues that arise in the area of query optimization in databases. And needless to say there is a considerable body of work in this area that could be applied to this problem. Our implementation employs some simple heuristics along these lines when it converts operators into formulas.

The final issue that arises when examining the complexity of the formula evaluator is that of defined predicates. As mentioned above, defined predicates invoke the evaluator on the formula that defines the predicate. This formula can be recursive. This means that a single predicate instance in a formula may end up invoking considerable additional computation as the evaluator recourses over its definition. Again it is easy to see that there can be no a-priori bound on the complexity of this process. However, as in the previous cases we have not found this to be a particular problem in our test domains.

*6.3.2. State expansion and goal testing*

As described above, state expansion (i.e., finding and applying the set of actions that can be applied to the current world) and testing for goal achievement both involve utilizing the formula evaluator. Hence, the complexity of these two components is determined by the complexity of the formula evaluator.

*6.3.3. Progressing formulas*

The process of progressing formulas is another area where expensive computations might be invoked. As can be seen from Table 2, the progression algorithm is generally quite efficient. In particular, except for quantification the process is essentially linear in the size of the input formula.[32] With quantification, however, it is possible to specify a short formula that takes a long time to progress.

The difficulty with progression lies not so much with progressing a formula once, but rather with the repeated progression of a formula through a sequence of worlds. During planning when we explore a sequence of states $w_0, \ldots, w_k$ we have to progress the original temporal control formula $k$ times, one through every world $w_i$. The formula might grow in length with each progression, and if we are not careful this can lead to excessive space and time requirements. For example, consider the progression of the temporal formula

$$P(a) \mathsf{U} \big(P(b) \mathsf{U} (P(c) \mathsf{U} Q(a))\big)$$

[32] Progression also invokes the evaluator on atemporal subformulas, and, as noted above, this also has the potential to be intractable.

through a world $w$ in which $P(a)$, $P(b)$, and $P(c)$ all hold, but $Q(a)$ does not. The progression algorithm yields the new formula

$$P(c) \mathbin{\mathsf{U}} Q(a)$$
$$\vee P(b) \mathbin{\mathsf{U}} \big(P(c) \mathbin{\mathsf{U}} Q(a)\big)$$
$$\vee P(a) \mathbin{\mathsf{U}} \big(P(b) \mathbin{\mathsf{U}} (P(c) \mathbin{\mathsf{U}} Q(a))\big).$$

Formulas of this form progress to formulas that have grown quadratically in size. Furthermore, the formula grows even longer as we progress it through multiple worlds.

The key to an efficient implementation of the progression algorithm is to realize that the progressed formula has many subformulas in common with the original formula. Hence, considerable efficiency can be gained by sharing these subformulas. In fact, in the above example, if we share substructures the progressed formula only requires us to store two new top level "$\vee$" connectives. Structure sharing is a well known technique in automated theorem provers, and we have employed similar techniques in our implementation. In addition to space efficiency structure sharing also yields computational efficiency. Progression distributes over the logical connectives (e.g., $\mathit{Progress}(\phi \wedge \psi) = \mathit{Progress}(\phi) \wedge \mathit{Progress}(\psi)$). Hence, once we have computed the progression of subformula that progression can be spliced in where ever the subformula appears, i.e., we need only compute the progression of a subformula once. In the above example, if we have to progress the new formula one more time we only need to progress the subformula "$P(c) \mathbin{\mathsf{U}} Q(a)$" once, even though it appears three times in the formula.

With these structure sharing techniques it is quantification that has the main impact on the efficiency of progression in practice. Consider the formula

$$\Box\forall[x{:}\mathit{object}(x)]\Diamond P(x).$$

If we progress this formula through a world $w$ in which no object satisfies $P$ and $\mathit{object}(a)$, $\mathit{object}(b)$, and $\mathit{object}(c)$ all hold, then we get the new formula

$$\Box\forall[x{:}\mathit{object}(x)]\Diamond P(x)$$
$$\wedge \Diamond P(a) \wedge \Diamond P(b) \wedge \Diamond P(c).$$

Since the progression algorithm deals with quantifiers by expanding each of the particular instances, we see that the progression of the formula grows in length by a factor determined by the number of objects satisfying *object* that currently fail to satisfy $P$. When there are $k$ nested quantifiers the progressed formula can be of length $\mathrm{O}(N^k)$, where $N$ is the number of objects in the domain. This behaviour is analogous to the behaviour of the formula evaluator in the face of nested quantification. However, as in that case, we have rarely found a need to nest quantifiers more than 3 deep in our temporal control formulas.

Furthermore, many natural control formulas do not continue to grow continually length. Consider, for example, the control formula specified for the blocks world (formula (3)). This formula when progressed through any world will generate a collection of conditions that must hold in the next world. In particular, there will be a collection of good towers that must be preserved, a collection of bad towers that nothing can be placed on top of,

and a collection of blocks that cannot be held in the next state. These conditions are all checked and *discharged* in the next world. Thus, the length of the control formula grows and shrinks, but never grows monotonically as we progress it through a sequence of worlds.

In summary, TLPLAN allows for very expressive domain specifications. It is sufficiently expressive that it is quite possible to express domains in which the basic operations of the planner become intractable. In practice, however, we have have found the planner's expressiveness to be a boon not a bane. It allows for a easy specification of domains and the potential of intractability of the basic operations has not been a major issue so far. Note that the tractability of planning in any domain is a separate issue from the tractability of the planner's basic operations. That is, although tractability of the basic operations is a necessary condition for tractable planning, it is by no means sufficient. Our empirical results (Section 7), however, do show that with the right control knowledge TLPLAN can plan very effectively in many test domains.

## 7. Empirical results

We have implemented a range of test domains to determine how easy it is specify control information in our formalism and how effective that information is in controlling planning search.

In our empirical tests we ran TLPLAN on a Pentium Pro 200 MHz machine with 128 MB of RAM. This amount of memory was more than sufficient for TLPLAN in all of the tests. We also ran various tests using the BLACKBOX [33], IPP [37], SATPLAN [32], PRODIGY [59], and UCPOP [7] systems.

BLACKBOX and SATPLAN are similar systems both of which encode planning problems as satisfiability problems. SATPLAN uses a different encoding that can be more efficient, while BLACKBOX employs various simplification steps interleaved with its generation of the encodings. IPP is based on the GRAPHPLAN [13] algorithm, but has been optimized in various ways and extended to handle a subset of the ADL language. BLACKBOX and IPP are both state of the art planning systems. They were the best performers in the AIPS'98 planning competition [1] and are both coded in C (as is TLPLAN). However both of these systems have tremendous appetites for memory, and so we ran them on a SUN Ultra 2 296 MHz machine with 256 MB of RAM. This still was not sufficient memory for these systems, but we were careful in recording the CPU time used so as not to count the time taken by swapping. Furthermore, fairly clear trends were already established by time the problems became large enough to start excessive thrashing. It should be noted however that BLACKBOX's and IPP's high memory consumption is not something that should be ignored. Space can be as much of a limiting resources as time, and in some cases more so.

The older systems UCPOP and PRODIGY are coded in lisp, and so we ran them on a 196 MHz SUN Ultra 2 that had support for lisp. However, the performance difference between these systems and the others was so great that recoding in C and running of the faster machine would not have helped much.

### 7.1. Blocks world

TLPLAN's performance with the three different control formulas, (formulas (1)–(3) given in Section 3), using depth-first search is shown in Fig. 3. Each $(x, y)$ data point represents the average time $y$ taken to solve 10 randomly generated blocks world problems involving $x$ blocks. In particular, for each value of $x$ we generated a random initial configuration of blocks and asked the planner to transform this configuration to a randomly generated goal configuration.

The graph also shows the time taken by the planner when it employed breadth-first search using the final control strategy, and the time taken by blind breadth-first search. (Blind breath-first outperforms blind depth-first search in this domain.) The data shows that control information acts incrementally, as we add more clauses to the control formula the planner is able to search more efficiently by pruning more paths from the search space. It also shows just how effective the search control can be—TLPLAN is able to solve 100 blocks problems in about 8 minutes when using depth-first search and Control strategy 3 (compare this with the performance of other state of the art planners shown in Fig. 5).

The data generated by the breadth-first search represent the time to find optimal plans.[33] The data shows that control strategies can be a considerable aid in solving the optimization problem as well. Using Control 3 TLPLAN is able to generate optimal plans for 18 block problems in reasonable time (42 seconds on average), while without control optimal 6 block problems are about the limit (7 blocks take more than 1000 seconds). Nevertheless, generating optimal plans in the blocks world is known to be NP-hard [26], and even the control strategies are insufficient for generating optimal solutions to all of the 19 blocks problems.

In the blocks world depth-first search can always find a solution,[34] but the solution may be very long. Fig. 4 shows the length of the plan found by the planner using the different control strategies. (Control 2 generates identical plans to Control 3, but takes longer.) The data also shows that the plans generated by Control 3 are quite high quality plans (measuring quality by plan length). They are only slightly longer then the optimal length plans. In fact, it can be show that the plans generated by Control 3 (and Control 2) are no longer than 2 times the length of the optimal plan.[35] Furthermore, TLPLAN is able to generate plans using these strategies without backtracking. Hence, these control strategies yield a polynomial time blocks world planner with a reasonable plan quality guarantee.

The blocks world remains a very difficult domain for current domain-independent planners. Fig. 5 shows how a range of the other planning systems perform in the blocks world.

[33] A control strategy could eliminate optimal plans from the search space: if no optimal plan satisfies the control strategy the strategy will stop the planner from finding an optimal plans. However, it is easy to show that no optimal plan is eliminated by these blocks world control strategies.

[34] In the blocks world every state is reachable from every other state, so any cycle-free depth-first path must eventually reach the goal.

[35] Control 3 encodes a strategy very similar to the reactive strategy given by Selman in [49], and he proves that this reactive strategy never exceeds the optimal by more than a factor of 2.

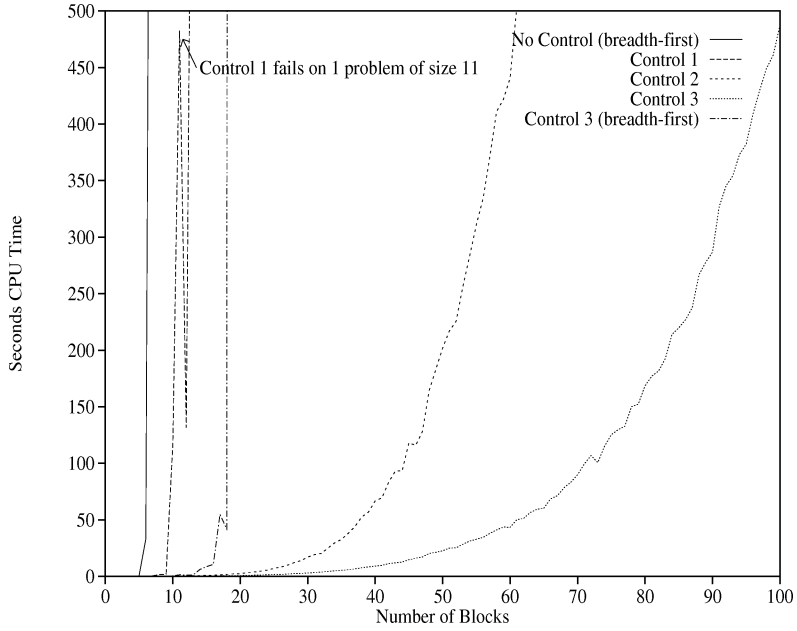


Fig. 3. Performance of TLPLAN search control in the blocks world.

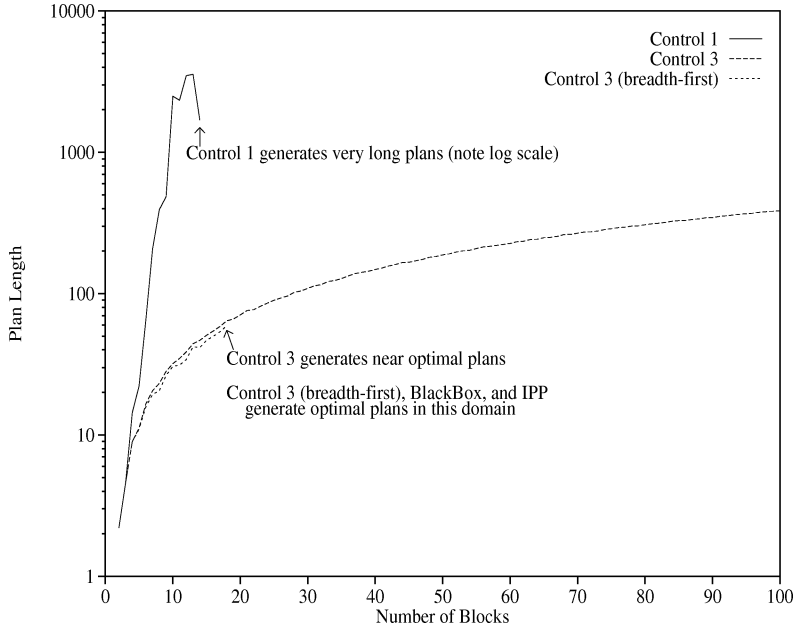


Fig. 4. Length of plans generated by TLPLAN in the blocks world.

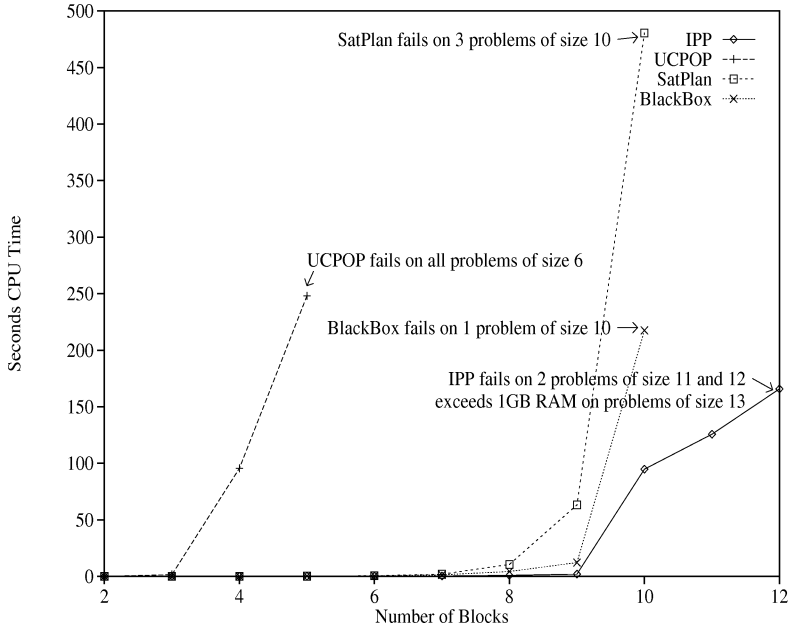


Fig. 5. Performance of other planners in the blocks world.

### *7.2. Briefcase world*

The briefcase world is a very simple domain invented by Pednault to illustrate his ADL action representation [44]. In this domain we have a briefcase that can be moved between different locations. Objects can be put in and taken out of the briefcase, and when they are in the briefcase they are moved with it. There is a simple and intuitive search control formula that can be written for this domain. What is most interesting however, is that the ideas in this search control appear almost unchanged in another popular test domain, the logistics domain (see below). We have found that there are many "meta-level" strategies that are applicable across different domains under slightly different concrete realizations.

The operators in this domain are given in Table 7.

The operators are given in TLPLAN's input language, which is basically first-order logic written in a lisp syntax. Two types of operators are accepted, standard STRIPS operators and ADL operators. Each consists of a sequence of clauses. The first clause is the name of the operator, and it may contain variables (e.g., ?x in the "take-out" operator). [36]

Taking something out of the briefcase can be specified as a simple STRIPS-style operator with a precondition, an add and a delete list. Each of these is a list of atomic predicates. The other two operators, "move-briefcase" and "put-in" are specified as ADL-style operators.

[36] Variables in TLPLAN are always prefixed with "?".

Table 7
The briefcase world operators

```
(def-adl-operator (MOVE-BRIEFCASE ?to)
 (pre
    (?from) (at briefcase ?from)
    (?to)   (location ?to)
   (not (= ?from ?to)))
 (add (at briefcase ?to))
 (del (at briefcase ?from))
 (forall (?z) (in-briefcase ?z)
   (and
      (add (at ?z ?to))
      (del (at ?z ?from)))))
```

```
(def-adl-operator (PUT-IN ?x)
  (pre
     (?loc)  (at briefcase ?loc)
     (?x)    (at ?x ?loc)
    (and (not (= briefcase ?x))
         (not (in-briefcase ?x))))
  (add (in-briefcase ?x)))

(def-strips-operator (TAKE-OUT ?x)
  (pre (in-briefcase ?x))
  (del (in-briefcase ?x)))
```

ADL operators are specified by using a precondition clause that acts exactly like a quantifier clause. In particular, all of the variables in the clause must be specified first (as with all quantifiers). All quantification is bounded quantification, so we must specify a quantifier bound for each of these variables. For example, for the "put-in" operator, `?loc` ranges over all of the locations the briefcase is at (there is in fact only one such location), while `?x` ranges over all objects that are at that location. Note that `?x` is scoped by `?loc`. Thus for each binding of `?loc` we compute a distinct range of bindings for `?x`. After the variable bindings the precondition can include an additional formula that can test the variable bindings and any other required features of the current world. Note that this formula can be an arbitrary first-order formula, it may include disjunction, other quantifiers, etc. [37] For example, the operator "put-in" includes the formula `(and (not (= briefcase ?x)) (not (in-briefcase ?x)))`. Each binding of the precondition variables that satisfies the precondition formula generates a unique instance of the operator (an operator instance is also called an action). The bindings of the variables appearing in the operator name are then used to give each action a unique name. [38]

Subsequent to the precondition formula come a sequence of clauses. These clauses are all scoped by the precondition's variables (and thus may access their bindings) and they are each individually evaluated by the formula evaluator (see Section 6). During evaluation any "add" or "del" clause always evaluates to TRUE and has the side-effect of modifying the new world. The current state of the world as well as the manner in which the evaluator works (i.e., its rules for the early termination of formula evaluation) precisely specifies the set of adds and deletes generated by this instance of the operator.

Thus, in the "move-briefcase" action, we add the briefcase's new location and delete its old location. [39] Then a universal quantifier is used to successively bind `?z` to all objects that are currently in the briefcase. For each such binding the body of the universal is

[37] Other planning systems that accept ADL specified actions, e.g., UCPOP and IPP, accept only a restricted subset of the ADL specification. For example, disjunctive preconditions are usually not allowed.

[38] Every operator instance need not have a unique name. Sometimes it is useful to treat different instances as being the same action.

[39] TLPLAN internally reorders the adds and deletes so that all deletes are executed prior to any adds.

Table 8
The briefcase world control strategy

```
(always
 (and
  (forall (?l) (at briefcase ?l)
    (forall (?x) (at ?x ?l)
      (implies (not (= ?x briefcase))
        (and
          ;; 1.
          (implies (goal (at ?x ?l))
                   (until (at briefcase ?l) (not (in-briefcase ?x))))
          ;; 1.
          (implies (not (goal (at ?x ?l)))
                   (until (at briefcase ?l) (in-briefcase ?x)))
          ;; 2.
          (implies (and (in-briefcase ?x) (not (goal (at ?x ?l))))
                   (next (in-briefcase ?x)))
          ;; 3.
          (implies (and (goal (at ?x ?l)) (not (in-briefcase ?x)))
                   (next (not (in-briefcase ?x))))))))

  (forall (?l) (location ?l)
          ;; 4.
          (implies
            (and
             ;;If we are not at location ?l
             (not (at briefcase ?l))
             ;;and we don't need to deliver something in the
               briefcase to ?l
             (not (exists (?x) (in-briefcase ?x) (goal (at ?x ?l))))
             ;;and we don't need to pickup something from that location
             (not (exists (?x) (at ?x ?l)
                          (or
                           (exists (?gl) (goal (at ?x ?gl))
                                   (not (= ?gl ?l)))
                           (goal (in-briefcase ?x)))))
             ;;and we don't need to move briefcase there
             (not (goal (at briefcase ?l))))
            ;;Then don't go there
            (next (not (at briefcase ?l)))))))
```

evaluated. The body of the universal is a conjunction, so we evaluate the first add. All terms in the add clause are evaluated, and in this case the variables `?z` and `?to` evaluate to the objects they are currently bound to (see Table 6). This results in a ground atomic fact being added to the world database.[40] The add clause always evaluates to TRUE, so the evaluator moves on to evaluate the second conjunct, the delete. The end result is that we update the locations of the briefcase and all the objects in the briefcase.

[40] All of the predicates and functions that appear inside of an add or a delete must be described symbols, as only these can be directly updated.

Table 9
Performance of TLPLAN and IPP in the briefcase world

| Problem name | TLPLAN time CPU seconds | IPP time CPU seconds |
|---|---|---|
| getpaid | 0.002 | 0.01 |
| getpaid3 | 0.004 | 0.02 |
| ex3a | 0.009 | 0.05 |
| ex3b | 0.005 | 0.01 |
| ex4a | 0.020 | 0.52 |
| ex4b | 0.009 | 0.04 |
| ex4c | 0.009 | 0.05 |
| ex4d | 0.021 | 0.51 |
| ex4f | 0.022 | 0.36 |
| ex4g | 0.022 | 0.20 |
| ex4h | 0.029 | 0.15 |
| ex4i | 0.020 | 0.09 |
| ex4j | 0.026 | 0.11 |
| ex5 | 0.046 | 0.91 |
| ex5a | 0.029 | 37.50 |
| ex5b | 0.030 | 22.23 |
| ex5c | 0.039 | 7.85 |
| ex5d | 0.043 | 15.46 |
| ex5e | 0.042 | 03.95 |
| ex5max | 0.045 | 0.92 |
| ex10 | 0.174 | >1669.9 |
| ex12a | 0.029 | 571.52 |
| ex12b | 0.030 | 691.47 |
| ex12c | 0.046 | 21.85 |
| ex12d | 0.026 | 14.85 |
| ex13a | 0.051 | 1057.65 |
| ex13b | 0.051 | 1887.27 |
| t1 | 0.002 | 0.01 |
| t2 | 0.004 | 0.01 |
| t3 | 0.008 | 0.04 |
| t4 | 0.021 | 0.52 |
| t5 | 0.029 | 35.32 |
| t6 | 0.047 | 3094.26 |
| t7 | 0.070 | >11053.00 |
| t8 | 0.098 | >7178.20 |
| t9 | 0.131 | >7639.00 |
| t10 | 0.186 | >3288.50 |

Search control formulas for this domain are easy to write. They embody the following obvious ideas:

(1) Don't move the briefcase from its current location if there is an object that needs to be taken out or put into the briefcase.
(2) Don't take an object out of the briefcase if the briefcase is not at the object's goal location.
(3) Don't put objects that don't need to be moved into the briefcase.
(4) Don't move the briefcase to an irrelevant location. In this domain a location is irrelevant if there is no object to be picked up there, there is no object in the briefcase that needs to be dropped off there, and it is not a goal to move the briefcase to that location.

The control formula given in Table 9 realizes these rules. We give the formula exactly as it is input to the planner. The planner can take as control input any formula of $\mathcal{LT}$. The only differences are that

(1) we use a prefix lisp syntax, and
(2) all of the logical symbols and modalities are given text names, e.g., the universal quantifier "$\forall$" is specified by `forall`.

The performance of TLPLAN using this control rule is demonstrated in Table 9. The table shows the planning time in seconds required by TLPLAN and by IPP to solve a suite of problems taken from the IPP distribution. (Briefcase world requires ADL actions, so cannot be solved with the current version of BLACKBOX; UCPOP can handle ADL actions but its performance is far worse than IPP.) The suite of problems includes the standard "getpaid" problem (the briefcase, a dictionary, and a paycheque are at home with the paycheque in the briefcase, and we want the take the dictionary to the office along with the briefcase, but leave the paycheque at home), "ti" problems that involve picking up $i$ objects at $i$ different locations and taking them home, and "exi" problems that involve permuting the locations of $i$ objects.

TLPLAN is faster on all of these problems. In fact, none of them is difficult for TLPLAN. However, IPP was unable to solve a number of the larger problems. The entries with values $> n$ for some $n$ indicate that IPP was aborted after that many seconds of CPU time without having found a plan.

### *7.3. Logistics world*

A popular test domain is the logistics world. In this domain we have two types of vehicles: trucks and airplanes. Trucks can be used to transport goods within a city, and airplanes can be used to transport goods between two airports. The problems in this domain typically start off with a collection of objects at various locations in various cities, and the goal is to redistribute these objects to their new locations. If the object's new location is in the same city it can be transported solely by truck. If its new location is in a different city it might have to be transported by truck to the city's airport, and then by plane to the new city, and then by truck to its final location within the new city.

The operators in this domain are given in Table 10. We have encoded this domain using ADL operators, simply because we find them to be easier to write than STRIPS operators. However, these operators can written as standard STRIPS operators as they

Table 10
The logistics world operators

```
(def-defined-predicate (vehicle ?vehicle)
  (or
   (truck ?vehicle)
   (airplane ?vehicle)))

(def-adl-operator (load ?obj ?vehicle ?loc)
  (pre
   (?obj ?loc) (at ?obj ?loc)
   (?vehicle) (at ?vehicle ?loc)
   (and
    (vehicle ?vehicle) (object ?obj)))
  (add
   (in ?obj ?vehicle))
  (del
   (at ?obj ?loc)))

(def-adl-operator (unload ?obj ?vehicle ?loc)
  (pre
   (?obj ?vehicle) (in ?obj ?vehicle)
   (?loc) (at ?vehicle ?loc))
  (add
   (at ?obj ?loc))
  (del
   (in ?obj ?vehicle)))

(def-adl-operator (drive-truck ?truck ?from ?to)
  ;;; We only allow trucks to move around in the same city.
  (pre
   (?truck) (truck ?truck)
   (?from) (at ?truck ?from)
   (?city) (loc-at ?from ?city)
   (?to) (loc-at ?to ?city)
   (not (= ?from ?to)))
  (add
   (at ?truck ?to))
  (del
   (at ?truck ?from)))

(def-adl-operator (fly-airplane ?plane ?from ?to)
  ;;; Airplanes may only fly from airport to airport.
  (pre
   (?plane) (airplane ?plane)
   (?from) (at ?plane ?from)
   (?to) (airport ?to)
   (not (= ?from ?to)))
  (add
   (at ?plane ?to))
  (del
   (at ?plane ?from)))
```

have simple preconditions, modify no function values, and have no conditional effects. We have also compressed the `load` and `unload` operators into a single case by using a defined predicate that tells us that an object is a vehicle when it is either a truck or an airplane. The standard STRIPS encoding would have four actions `load-truck`, `unload-truck`, `load-plan`, and `unload-plane`. It should be apparent that the search space is identical (e.g., whenever an instance of one of our `load` operators is executable with `?vehicle` bound to a truck, an equivalent instance of a `load-truck` operator will be executable). The other planners we ran were supplied with the standard STRIPS encoding.

A control strategy very similar to the briefcase world is applicable in the logistics world (and in fact in many domains that involve transporting goods the same meta-level principles appear). In particular, we can write a control strategy that embodies the following ideas:

(1) Don't move a vehicle if there is an object at the current location that needs to be loaded into it. Similarly, don't move a vehicle if there is an object in it that needs to be unloaded at the current location.
(2) Don't move a vehicle to a location unless, (1) the location is a where we want the vehicle to be in the goal, (2) there is an object at that location that needs to be picked up by this kind of vehicle, or (3) there is an object in the vehicle that needs to be unloaded at that location.
(3) Don't load an object into a vehicle unless it needs to be moved by that type of vehicle.
(4) Don't unload an object from a vehicle unless it needs to be unloaded at that location.

There are two types of vehicles, each used for a distinct purpose. So it is helpful to define a collection of auxiliary predicates. [41]

```
(def-defined-predicate (in-wrong-city ?obj ?curr-loc ?goal-loc)
  ;;TRUE IFF an object in ?curr-loc with goal location
  ;;?goal-loc is in right city. (loc-at ?loc ?city) is true
  ;;if ?loc is located in city ?city.
  ;;
  (exists (?city) (loc-at ?curr-loc ?city)
          (not (loc-at ?goal-loc ?city))))

(def-defined-predicate (need-to-move-by-truck ?obj ?curr-loc)
  ;;We need to move an object located at curr-loc by truck iff
  ;;the object is in the wrong city and is not at an airport
  ;;or the object is in the right city but not at the right
  ;;location.
  ;;
  ;;Note if there is no goal location we don't need to move
  ;;by truck.
  ;;
  (exists (?goal-loc) (goal (at ?obj ?goal-loc))
          (if-then-else
```

[41] The logical connective `(if-then-else f1 f2 f3)` is simply short hand for `(and (implies f1 f2)(implies (not f1) f3))`.

```
            (in-wrong-city ?obj ?curr-loc ?goal-loc)
              (not (airport ?curr-loc))
              ;;in right city
              (not (= ?curr-loc ?goal-loc)))))

(def-defined-predicate (need-to-unload-from-truck
 ?obj ?curr-loc)
  ;;We need to unload an object from a truck at the current
  ;;location iff, ?curr-loc is the goal location of the object,
  ;;or the object is in the wrong city and the
  ;;current-location is an airport.
  (exists (?goal-loc) (goal (at ?obj ?goal-loc))
          (or
           (= ?curr-loc ?goal-loc)
           (and (in-wrong-city ?obj ?curr-loc ?goal-loc)
                (airport ?curr-loc)))))

(def-defined-predicate (need-to-move-by-airplane
 ?obj ?curr-loc)
  ;;We need to move an object at curr-loc by airplane iff
  ;;the object is in the wrong city.
  ;;
  (exists (?goal-loc) (goal (at ?obj ?goal-loc))
          (in-wrong-city ?obj ?curr-loc ?goal-loc)))

(def-defined-predicate (need-to-unload-from-airplane
 ?obj ?curr-loc)
  ;;We need to unload an object from an airplane at the
  ;;current location iff, ?curr-loc is in the right city.
  (exists (?goal-loc) (goal (at ?obj ?goal-loc))
          (not (in-wrong-city ?obj ?curr-loc ?goal-loc))))
```

With these predicates we can define the following control strategy that realizes the above rules.

```
(always
 (and

  (forall (?x ?loc) (at ?x ?loc)
   (and
    (implies (vehicle ?x)
     (and
      ;;; don't move a vehicle if there is an object that
      ;;; needs to be moved by it, or if there is an object that
      ;;; needs to be unloaded from it at the current location.
      (implies
       (exists (?obj) (object ?obj)
        (or
         (and
```

```
      (at ?obj ?loc)
      (implies (truck ?x)
               (need-to-move-by-truck ?obj ?loc))
      (implies (airplane ?x)
               (need-to-move-by-airplane ?obj ?loc)))
     (and
      (in ?obj ?x)
      (implies (truck ?x)
               (need-to-unload-from-truck ?obj ?loc))
      (implies
       (airplane ?x)
       (need-to-unload-from-airplane ?obj ?loc)))))
   (next (at ?x ?loc)))
  ;;;Similarly when we move a vehicle one of these
  ;;;conditions should be meet.
  (next
   (exists (?newloc) (at ?x ?newloc)
   ;;at the next location of the vehicle
    (or
     ;;either we didn't move it.
     (= ?newloc ?loc)
     ;;or the location was a goal location for the vehicle
     (goal (at ?x ?newloc))
     ;;or there is an object such that
     (exists (?obj) (object ?obj)
      (or
       ;;the object is at the new location and needs a
       ;;pickup.
       (and
        (at ?obj ?newloc)
        (implies (truck ?x)
                 (need-to-move-by-truck ?obj ?newloc))
        (implies
        (airplane ?x)
        (need-to-move-by-airplane ?obj ?newloc)))
       ;;or the object is in the vehicle and needs to be
       ;;unloaded
       (and
        (in ?obj ?x)
        (implies (truck ?x)
                 (need-to-unload-from-truck ?obj ?newloc))
        (implies
         (airplane ?x)
         (need-to-unload-from-airplane ?obj ?newloc)))))
     )))))

(implies (object ?x)
 (and
  ;;;don't load into a vehicle unless we need to move by
```

```
      ;;;that type of vehicle.
      (forall (?truck) (truck ?truck)
              (implies (not (need-to-move-by-truck ?x ?loc))
                       (next (not (in ?x ?truck)))))
      (forall (?plane) (airplane ?plane)
              (implies (not (need-to-move-by-airplane ?x ?loc))
                       (next (not (in ?x ?plane))))))))

;;;Finally, don't unload objects unless we need to.
  (forall (?obj ?vehicle) (in ?obj ?vehicle)
   (exists (?loc) (at ?vehicle ?loc)
    (implies
     (or
      (and (truck ?vehicle)
           (not (need-to-unload-from-truck ?obj ?loc)))
      (and (airplane ?vehicle)
           (not (need-to-unload-from-airplane ?obj ?loc))))
     (next (in ?obj ?vehicle)))))
 ))
```

With this control rule we obtain the performance shown in Fig. 6. The data shows the planner solving problems where there are $n$ objects to be moved (plotted on the $x$-axis). In the initial state we place 3 objects in each city (and thus we have $\lfloor n/3 \rfloor$ different cities), one truck per city, two locations per city (a post-office and an airport), and $\lfloor n/10 \rfloor$ airplanes.

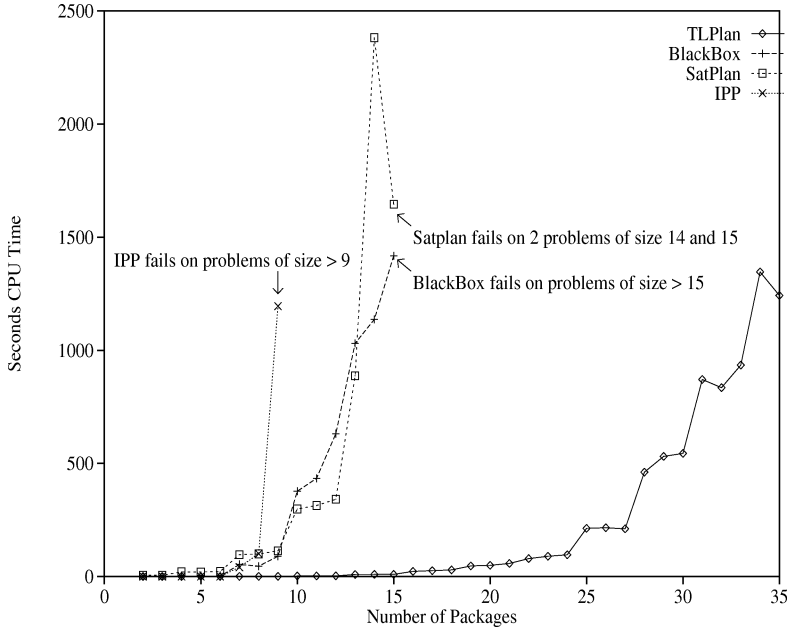


Fig. 6. Performance of various planners in the logistics world.

Table 11
Performance of TLPLAN and BLACKBOX on logistics problems

| Problem | TLPLAN time | BLACKBOX time | TLPLAN length | BLACKBOX length |
|---|---|---|---|---|
| log001 | 0.260 | 0.575 | 25 | 25 |
| log002 | 0.281 | 95.977 | 27 | 31 |
| log003 | 0.245 | 98.998 | 27 | 28 |
| log004 | 1.371 | 130.748 | 51 | 71 |
| log005 | 1.105 | 231.938 | 42 | 69 |
| log006 | 1.918 | 321.272 | 51 | 82 |
| log007 | 5.547 | 264.046 | 70 | 96 |
| log008 | 6.844 | 317.422 | 70 | 110 |
| log009 | 3.792 | 1609.455 | 70 | 121 |
| log010 | 2.427 | 84.046 | 41 | 71 |
| log011 | 2.245 | 137.93 | 46 | 68 |
| log012 | 1.936 | 136.229 | 38 | 49 |
| log013 | 6.543 | 165.844 | 66 | 85 |
| log014 | 9.348 | 77.749 | 73 | 89 |
| log015 | 5.364 | 424.369 | 63 | 91 |
| log016 | 1.146 | 926.967 | 39 | 85 |
| log017 | 1.242 | 758.471 | 43 | 83 |
| log018 | 9.270 | 152.35 | 46 | 105 |
| log019 | 2.660 | 149.224 | 45 | 78 |
| log020 | 10.180 | 538.220 | 89 | 113 |
| log021 | 6.838 | 190.490 | 59 | 87 |
| log022 | 6.406 | 846.842 | 75 | 111 |
| log023 | 4.693 | 173.966 | 62 | 85 |
| log024 | 4.714 | 74.832 | 64 | 87 |
| log025 | 4.099 | 73.995 | 57 | 84 |
| log026 | 3.646 | 233.406 | 55 | 80 |
| log027 | 5.529 | 145.164 | 70 | 97 |
| log028 | 14.533 | 867.349 | 74 | 118 |
| log029 | 5.998 | 89.515 | 45 | 84 |
| log030 | 3.482 | 495.373 | 51 | 92 |

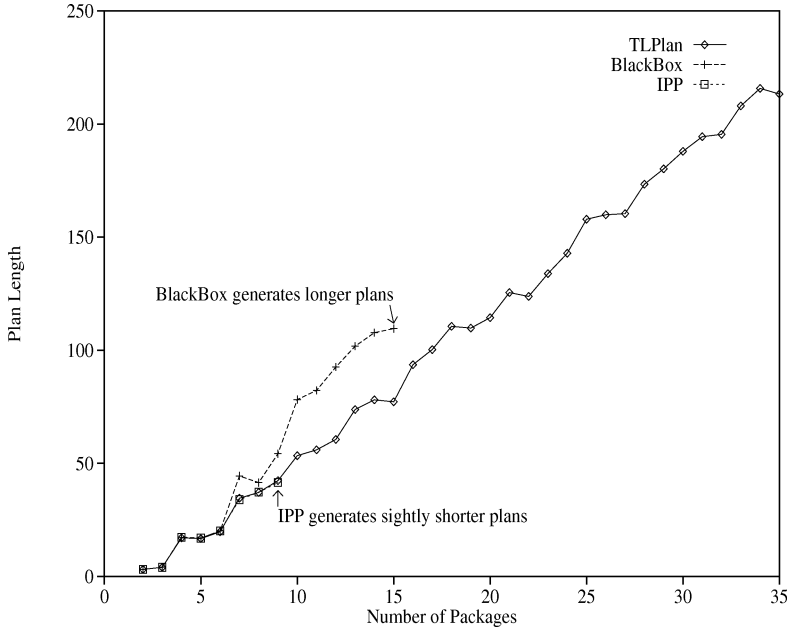


Fig. 7. Length of plans generated in the logistics world.

The final locations of the objects are chosen at random from the $2 \times \lfloor n/3 \rfloor$ different locations. Each data point shows the average time require to solve 10 random problems of that size.

Fig. 6 also shows how well the other planners perform in this domain. It can be seen that the control strategy gives TLPLAN a multiple order of magnitude improvement in performance over these planners.

BLACKBOX's performance was hindered by the fact that our machine only had 256 MB of RAM, although it was significantly slower than TLPLAN even on smaller problems prior to the onset of thrashing. The timing routines do a fairly good job of accounting just for the CPU time taken (i.e., the time taken by the program while waiting for a page to be swapped in is not counted), but to check whether or not this was a significant bias we ran a suite of 30 problems that come with the BLACKBOX distribution. BLACKBOX's solutions to these problems also come with the distribution, and were generated on a machine that is slightly faster than the machines used in our experiments (judging by the solution times for the smaller problems on which BLACKBOX was not thrashing) and more significantly had sufficient main memory to avoid thrashing. Table 11 shows TLPLAN's performance on this test suite (run on our 128 MB machine). (TLPLAN had no difficulty completing much larger problems than these while using less than 128 MB or RAM.) The results show that the memory bottleneck was not a significant factor in our experiments: TLPLAN with this control strategy remains significantly faster than BLACKBOX.

Finally, Fig. 7 compares the total number of actions in the plans generated by the three planners. Again we see that TLPLAN is generating very good plans, as good as the other two planners. These other planners both search for a plan incrementally, looking for shorter

plans first. Hence, we would expect them to be generating relatively short plans. TLPLAN employs no such strategy. It simply does a depth-first search. It is the control strategy that stops "stupid" moves from being included in the plan. Similar results can be seen in Table 11, where again TLPLAN is generating shorter plans than BLACKBOX. [42]

### *7.4. Tire world*

The tire world is another standard test domain due originally to Russell [47]. 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 the forward direction with 14 different operators. We wrote a control strategy for this domain that included the following ideas:

(1) Only fetch an object from a container if you need it. This rule involved defining predicates that determine, e.g., when one needs the wrench, the jack, the pump, etc.
(2) A number of rules to deal with the wheels and nuts:
   - Don't inflate a wheel unless it needs to be inflated.
   - Don't jack up a wheel unless it needs to be jacked up.
   - Keep correctly placed wheels on their current hubs, and don't place a wheel on an incorrect hub.
   - If a wheel needs to be removed from a hub, don't undo any of the removal steps.
   - Keep a hub jacked up until its wheel is on and the nuts are tight.
   - Execute the actions for putting wheels on hubs and removing them from hubs in a particular order.
(3) Only open containers that contain something you need.
(4) Don't put away any objects until you don't need them anymore.
(5) Keep containers open until you have removed everything you need and everything that needs to be stored there has been returned.

Each of these rules is fairly intuitive, and their encoding as formulas of $\mathcal{LT}$ is straightforward (albeit lengthy).

With this control strategy we obtain the performance shown in Fig. 8. We designed a suite of test problems that involved changing an increasing number of tires using one set of tools. The data shows the planner solving problems in which the goal has $n$ literals (plotted on the $x$-axis). As $n$ increases we need to increase the number of tires in order to generate $n$ different goal literals. Each data point shows the time required to solve the problem of that size. The final problem ($n = 74$ involved changing 15 tires). The data also shows the performance of the IPP and BLACKBOX planners on these problems.

Once again since we are generating plans using depth-first search we compare the length of the generated plans in Fig. 9. The data shows that TLPLAN once again is able to achieve exceptional performance once it is given an appropriate control strategy.

[42] BLACKBOX employs a stochastic search for a plan once it has constructed a GRAPHPLAN graph of $k$ time steps. Thus, even though a plan of $k$ time steps might exist it could fail to find it if its stochastic search runs out of time.

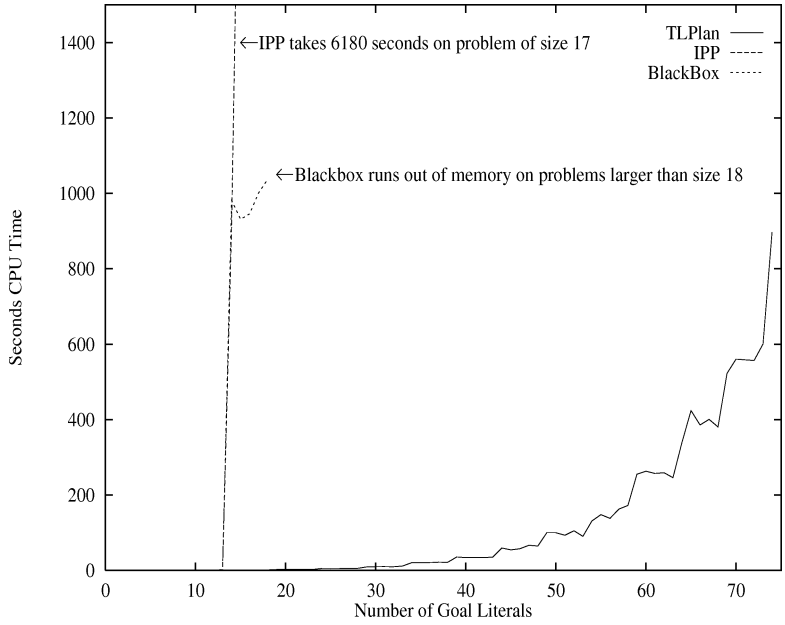


Fig. 8. Performance of TLPLAN and other planners in the tire world.

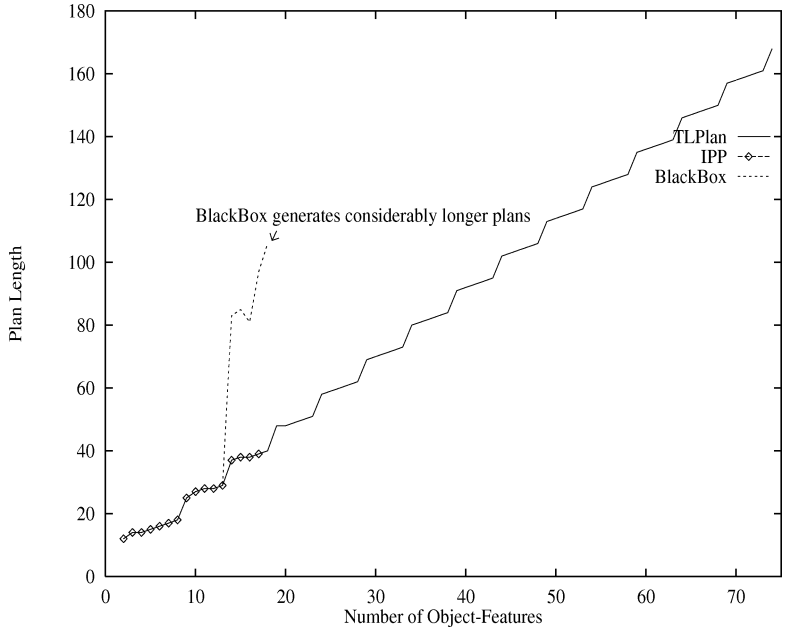


Fig. 9. Length of plans generated in the tire world.

### 7.5. Schedule world

The schedule world involves scheduling a set of objects on various machines in order to achieve various effects, e.g., shaping the object, painting it, polishing it, etc. Some of the operations undo the effects of other operations, and sometimes make other operations impossible. This domain has 8 operators, and when the number of objects climbs so does the branching factor in the forward direction.

It is worth while noting that in this domain the actions are inherently concurrent. Every machine can be run in parallel. This is not a problem for TLPLAN even though it explores linear sequences of actions. In particular, the sequence of worlds we explore can have whatever structure we choose, so a linear sequence of worlds need not correspond to a linear sequence of times in the domain being modeled. In this domain we added a time stamp to the world, the time stamp denotes the current time in the partial schedule. The actions generate new worlds by scheduling currently unscheduled objects on currently unscheduled machines (i.e., neither the object nor the machine can be marked as being scheduled in the current time step). When no further scheduling actions are possible, or desirable, there is an action that can increment the time stamp. This has the effect of making all of the objects and machines available for scheduling (in the new current time step). [43] In other words, TLPLAN explores a sequence of worlds in which there are a sequence of scheduling actions that schedule a concurrent set of operations, followed by a time step, followed by another sequence of scheduling actions that schedule the next set of concurrent operations. Other types of concurrent actions can be modeled in this manner, e.g., we have implemented a job-shop scheduling domain that solves the standard job-shop scheduling problems using TLPLAN.

The performance of TLPLAN is shown in Fig. 10. The data shows the planner solving problems where there are $n$ objects and $n$ randomly chosen properties involving those objects to be achieved (a single object might be randomly selected to require more than one property). $n$ forms the $x$-axis. Each data point represents the average time required to solve 10 random problems of that size. The graph also shows the performance of IPP in this domain. The domain requires ADL-actions so we were unable to run BLACKBOX in this test.

The control strategy used by TLPLAN included the following ideas:

(1) Never schedule an operation twice. This is a particularly simple scheduling domain in which there is never a need to perform an operation twice: there is always a better plan in which the operations are sequenced in such a manner that no needed effects are undone.
(2) All scheduled operations must achieve at least one unachieved goal.
(3) Once a goal condition has been achieved do not allow it to be destroyed. In this domain we never need to undo achieved goals.

[43] There are two common versions of this domain, a much simplified one that first appeared in the UCPOP distribution. The UCPOP version discarded all notion of time, it simply computes what operations need to be run on what objects. The original version that appeared in the PRODIGY distribution involves a nontrivial use of time. We coded the PRODIGY version for our tests, both in TLPLAN and IPP, and ran this version in our experiments.

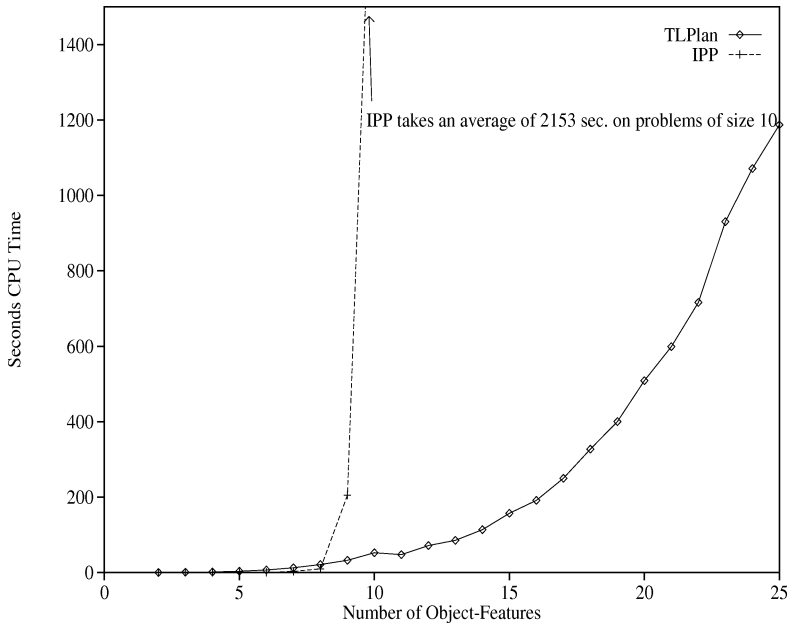


Fig. 10. Performance of TLPLAN and IPP in the schedule world.

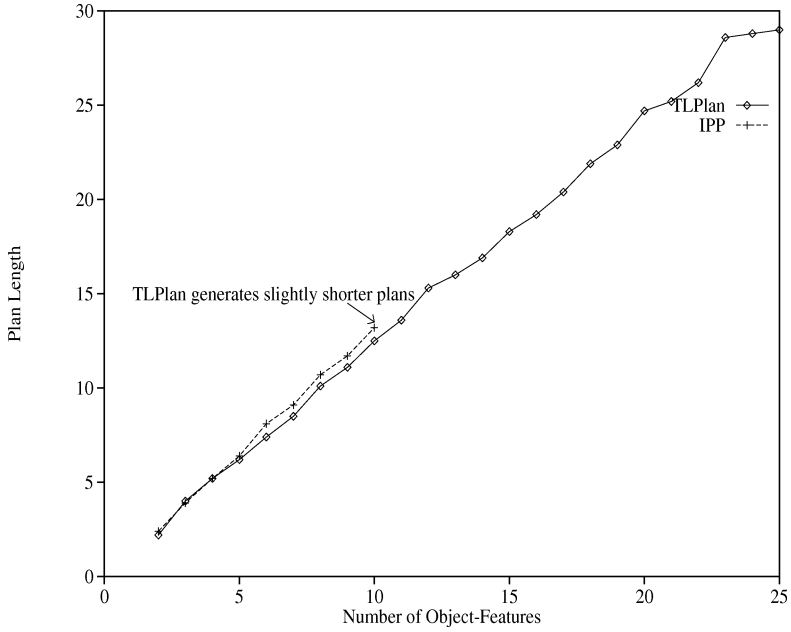


Fig. 11. Length of plans generated the schedule world.

(4) Some ordering constraints on goal achievements:
- Rolling (to make an object cylindrical) destroys a number of things that can never be reachieved after rolling (as the object's temperature goes up). So prohibit rolling if one of these conditions is a goal.
- Do any shaping operations prior to any surface conditioning operations, as these destroy the surface condition.
- Do any grinding or lathing prior to painting, as these destroy the paint.

Fig. 11 shows the average number of actions in the plans generated by the two different planners. Once again we see that the search control allows TLPLAN to construct good plans using depth-first search. In this case TLPLAN is able to generate slightly shorter plans than IPP. IPP employs a deterministic search for a plan on each $k$-step GRAPHPLAN graph, so it will find a plan that has shortest GRAPHPLAN parallel length. [44] However, the plan it finds might still include redundant actions (as long as the redundant actions can be executed in parallel). In a similar manner however the plans found by TLPLAN need not be of shortest parallel length. Plan length is at best a rough estimate of plan quality.

### *7.6. Bounded blocks world*

Another interesting problem is the bounded blocks world in which the table has a limited amount of space. It is easy to specify and plan in the bounded blocks world using TLPLAN. However, none of the other standard domain-independent planners can deal effectively with resource constraints, even simple ones like this. [45]

In this domain, it is easier to use a single operator that simply moves blocks from one location to another (thus avoiding the intermediate "holding" a block state present in the blocks world specification used in Section 3). Table 12 gives the domain's operator. In this case the operator's precondition is quite simple, we must move the object `?x` to a new location, we cannot move the table, and if we move `?x` to the table there must be space on the table. For this domain, `table-space` is a 0-ary described function that must be specified in the initial state and must be properly updated by the operator. The term `(table-space)` evaluates to the quantity of space on the table in the current world (table-space $= n$ means that there is space for $n$ more blocks on the table), and the precondition simply tests to ensure that there is space on the table if that is where we intend on moving `?x`.

This gives an example of TLPLAN ability to handle functions. In particular, by adding the equality predicate `(add (= (table-space) (- (table-space) 1)))` we are specifying that the function `(table-space)` is to have the new value given by its current value minus one. All terms inside of the add and deletes are evaluated in the current world prior to being committed. The evaluator computes the value of the term `(- (table-space) 1)` by looking up the current value of `(table-space)` and subtracting 1 from it using the standard computed function '$-$'.

[44] GRAPHPLAN graphs only handle a limited kind of action concurrency, so shortest GRAPHPLAN parallel length need not be shortest parallel length.

[45] HTN-style planners often have some facilities for dealing with resource constraints, e.g., [62].

Table 12
The bounded blocks world operators

```
(def-adl-operator (puton ?x ?y)
  (pre
    (?x) (clear ?x)
    (?y) (clear ?y)
    (?z) (on ?x ?z)
    (and
      (not (= ?z ?y))            ;Don't put it back where it came from
      (not (= ?x ?y))            ;Can't put a block on itself
      (not (= ?x table))         ;Can't move the table
      (implies (= ?y table)          ;can move to table only if
               (> (table-space) 0)))) ;table has space.

  (add (on ?x ?y))
  (del (on ?x ?z))
  (implies (= ?y table)
    (add (= (table-space) (- (table-space) 1))))
  (implies (= ?z table)
    (add (= (table-space) (+ (table-space) 1))))
  (implies (not (= ?y table))
    (del (clear ?y)))
  (implies (not (= ?z table))
    (add (clear ?z))))
```

The conditional updates are specified using "implies". In particular, since the evaluator short-circuits the evaluation of formulas, the consequent of the implication (in this case the adds and deletes) will not be executed if the antecedent evaluates to FALSE.

Fig. 12 illustrates the performance of TLPLAN in this domain. Each data point represents the average time taken to solve 10 randomly generated bounded blocks problems, were we have only 3 spaces on the table. The data shows that this domain, like the standard blocks world, is very hard without domain-specific search control. There are two different control strategies that can be easily specified for this domain. First, the meta-level notion of a *goodtower* continues to be useful in this modified version of the blocks world. It has, however, a slightly different realization. In particular, we may now need to dismantle a tower of blocks to free some space on the table. We can define an appropriately modified version of *goodtower* as follows:

```
(def-defined-predicate (goodtower ?x)
  ;;Note this goodtower takes into account table space.
  ;;In particular, goodtowers must not occupy needed
  ;;tablespace.
  (and
   (clear ?x)
   (if-then-else
    (= ?x table)
    ;;then
    (> (table-space) 0)
    ;;table is a goodtower if it has space.
```

```
    ;;else
    (goodtowerbelow ?x))))

(def-defined-predicate (goodtowerbelow ?x)
  (or
   (and (on ?x table) (goal (on ?x table)))
   (and (on ?x table)
        (not (exists (?y) (goal (on ?x ?y))))
        (forall (?z) (goal (on ?z table)) (on ?z table)))
   (exists (?y) (on ?x ?y)
     (and
       (not (goal (on ?x table)))
       (not (goal (clear ?y)))
       (forall (?z) (goal (on ?x ?z)) (= ?z ?y))
       (forall (?z) (goal (on ?z ?y)) (= ?z ?x))
       (goodtowerbelow ?y)))))
```

In this version, we classify the table as being a *goodtower* if it can be stacked on (i.e., if there is space). The main difference lies in *goodtowerbelow*, where a block on the table is a *goodtower* if it needs to be on the table, or there is nowhere else it need be and all other blocks that must be on the table are already there. In both of these cases the goal can be achieved without moving that block from the table. The recursive case is just as in the standard *goodtowerbelow* given in Section 3.

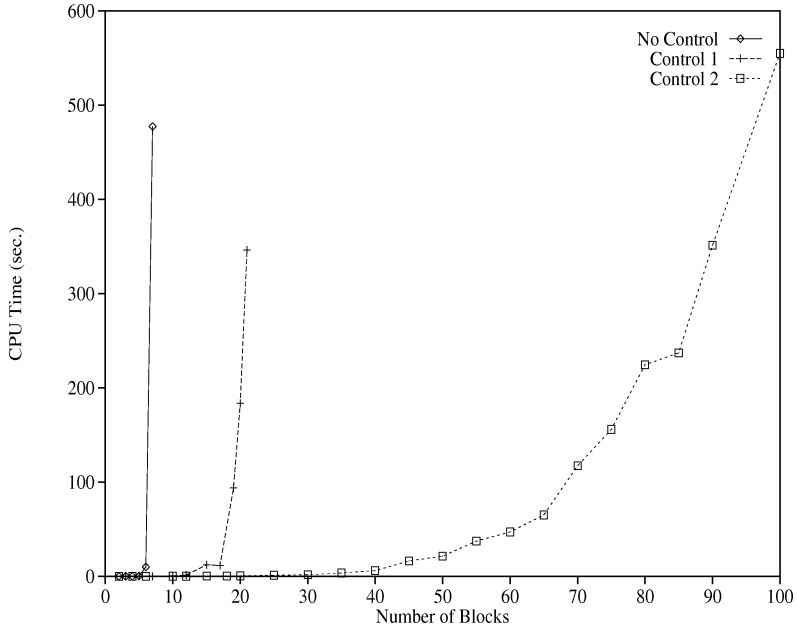


Fig. 12. Search control in the bounded blocks world.

Now we can define the following predicate:

```
(def-defined-predicate (can-move-to-final ?x)
  ;;we can move ?x to its final location when it has
  ;;a final location location and that final location is
  ;;a goodtower.
  (and
   (clear ?x)
   (exists (?y) (goal (on ?x ?y))
     (and
       (not (on ?x ?y))
       (goodtower ?y)))))
```

This predicate is true of a block when it can be moved to its final location. With `can-move-to-final` we can define the following very simple control strategy:

```
(define bbw-control1
  ;;simple trigger control.
  (always
   (and
    ;;never destroy goodtowers.
    (forall (?x) (clear ?x)
      (implies (and (not (= ?x table)) (goodtower ?x))
               (next (goodtowerbelow ?x))))
    ;;if a block exists that can be moved immediately,
    ;;move it.
    (implies
     (exists (?x) (clear ?x)
       (can-move-to-final ?x))
     (exists (?x) (clear ?x)
       (and (can-move-to-final ?x)
            (next (goodtower ?x)))))
     )))
```

This control strategy is a simple "trigger" control. If we are at a state where a block can be moved to its final location do so. Note that if there are multiple blocks that can be moved to their final location the trigger (the existential condition) will remain active until all have been moved. The choice of which block to move first is "non-deterministic".[46] The strategy also involves the obvious of not dismantling towers that don't need to be dismantled.

Fig. 12 shows that the trigger control is quite effective for small problems, and serves to illustrate the fact that considerable gain can often be achieved with minor effort. Nevertheless, although the trigger control knows what to do if it finds certain fortuitous situations, it has no idea of how to achieve those fortuitous situations. Hence, as we increase the size of the problems it becomes less and less useful.

A more complete strategy can also be written. This strategy is more complex, but it is able to solve problems quite effectively without requiring any backtracking. Again the idea

[46] Of course, the implementation picks the blocks in a particular order.

is quite simple. The strategy has four components. The first two are taken from the previous strategy.

(1) Never dismantle goodtowers.
(2) If there are blocks that can be moved to their final positions we pick one such block and move it to its final position.
(3) If locations exists that can be stacked on (i.e., they are goodtowers that are waiting for their next block) then we pick one such location and clear the block that is intended to go there while keeping the location clear. Once the next block is clear we are back to a situation where the previous rule applies: there is a block that can be moved into its final location.
(4) If there are no clear locations that can be stacked on, we pick one such location and clear it. Once we have achieved this we are in a situation where rule (3) applies.

To facilitate the implementation of this strategy we make the following definitions:

```
(def-defined-predicate (can-stack-on ?x) ()
  ;;this block is ready to be stacked on.
  (and
   (goodtower ?x)
   (exists (?y) (goal (on ?y ?x))
     (not (on ?y ?x)))))

(def-defined-function (depth ?x) ()
 ;;return the depth of location ?x
  (if-then-else
   (clear ?x)
   ;;then
   (:= depth 0)
   ;;else
   (exists (?y) (on ?y ?x)
     (:= depth (+ 1 (depth ?y))))))

;;;A function to find a location that would become
;;;a can-stack-on location if it was clear.
;;;
(def-defined-function (find-can-stack-on-if-clear) ()
  ;;pick the table if that is possible
  (or
   (if-then-else
    (and (= (table-space) 0)
         (exists (?x) (goal (on ?x table))
                      (not (on ?x table))))
    ;;then return the table
    (:= find-can-stack-on-if-clear table)
    ;;else return the top of a goodtower prefix
    (exists (?x) (on ?x table)
      (and (goodtowerbelow ?x)
        (exists (?y) (= ?y (top-of-goodblocks ?x))
          (and
```

```
            ;;such that the tower is incomplete.
            (exists (?z) (goal (on ?z ?y))
              (not (on ?z ?y)))
            (:= find-can-stack-on-if-clear ?y))))))

      ;;first clause fails, so no stackable location exists.
      (:= find-can-stack-on-if-clear *NOSUCHLOCATION*)))

(def-defined-function (top-of-goodblocks ?x) ()
 ;;If we pass this function a block (which should be a block
 ;;that has a goodtower below it) it will examine the tower
 ;;above the block looking for the top of the longest good
 ;;tower above.

  (if-then-else
   (clear ?x)
   ;;if ?x is clear then it is the top of the goodtower
   ;;prefix.
   (:= top-of-goodblocks ?x)
   ;;else there is a block on ?x
   (exists (?y) (on ?y ?x)
     (if-then-else
       (and
         (not (goal (on ?y table)))
         (not (goal (clear ?x)))
         (forall (?z) (goal (on ?y ?z)) (= ?z ?x))
         (forall (?z) (goal (on ?z ?x)) (= ?z ?y)))
       ;;if the block on top does not violate any
       ;;goal on-relations, then recurse upwards.
       (:= top-of-goodblocks (top-of-goodblocks ?y))
       ;;else stop at ?x.
       (:= top-of-goodblocks ?x)))))
```

The predicate `can-stack-on` is true of a location `?x` if that location is ready to be stacked on; it is used to implement rule (3) of the strategy. The function `depth` has already been explained. The function `find-can-stack-on-if-clear` is a function that returns a location that once cleared can be stacked on; it is used to implement rule (4) of the strategy. The function checks to see if the table is such a location (e.g., when we have a tower of blocks that we have not yet started to build) and returns that if possible. Otherwise it employs the recursive function `top-of-goodblocks` to find the top block of a partly completed goodtower. If we can clear that top block we will once again have a location that can be stacked on. One thing to notice in the function `find-can-stack-on-if-clear` is the use of a functional binding of the existential variable `?y` in the line:

```
(exists (?y) (= ?y (top-of-goodblocks ?x))
```

This line specifies that the variable `?y` is to range over the set of objects that are equal to

```
(top-of-goodblocks ?x).
```

There is of course only one such object, and it is computed by evaluating the function (?x has already been bound at this point).

With these definitions the second strategy can be specified as follows:

```
(define bbw-control2
  ;;more complex control
 (always
  (and
   ;;1. never destroy goodtowers.
   (forall (?x) (clear ?x)
     (implies (and (not (= ?x table)) (goodtower ?x))
              (exists (?y) (on ?x ?y)
                (next (on ?x ?y)))))

   ;;2. Immediate moves
   (implies
    ;;if We can make an immediate move.
    (exists (?x) (clear ?x)
      (can-move-to-final ?x))
    ;;pick one and do it.
    (exists (?x) (clear ?x)
      (and (can-move-to-final ?x)
           (next (goodtower ?x)))))

   ;;3. Clear a next block.
   (if-then-else
    ;;if there is a stackable-location (including the table)
    (exists (?x) (clear ?x)
      (can-stack-on ?x))
    ;; then make progress towards uncovering the next block of
    ;; at least one such location. We do this by asserting
    ;; that there is one such block, and an unachieved
    ;; (on ?y ?x)
    ;; relation such that until we achieve it we decrease the
    ;; depth of ?y (i.e., we uncover ?y) while keeping ?x
    ;; clear.
    (exists (?x) (clear ?x)
     (and
      (can-stack-on ?x)
      ;;Need also to pick the next block to clear as if ?x is
      ;;the table there could be more than one ''next block''
      (exists (?y) (goal (on ?y ?x))
       (and (not (on ?y ?x))
            (until
             (and
              (can-stack-on ?x)              ;;Keep ?x clear
              (exists (?d) (= ?d (depth ?y))  ;;and decrease
               (next (or (on ?y ?x) (< (depth ?y) ?d)))))
             (on ?y ?x))))))                 ;;?y's depth the
```

```
                                                    ;;constraint
                                                    ;;is active
                                                    ;;until we
                                                    ;;achieve
                                                    ;;(on ?y ?x)

        ;;4. else we are either completed or we should pick
        ;;a location that once clear will become a can-stack-on
        ;;location and clear it. (This might include the table).

        (exists (?loc) (= ?loc (find-can-stack-on-if-clear))
            (or
             (= ?loc *NOSUCHLOCATION*)
             (exists (?x) (on ?x ?loc)
              (and
               (implies (= ?loc table) (not (goodtowerbelow ?x)))
               (until
                 (exists (?d) (= ?d (depth ?x))
                   (next (or (can-stack-on ?loc)
                             (< (depth ?x) ?d))))
                 (can-stack-on ?loc))))))
        ))))
```

The specification is a fairly straightforward translation of the four components mentioned above. There are two similar clauses in the strategy, the second one of which is

```
(until
  (exists (?d) (= ?d (depth ?x))
    (next (or (can-stack-on ?loc) (< (depth ?x) ?d))))
  (can-stack-on ?loc))
```

where `?x` is on `?loc`.

This clause asserts a condition that must be true of every state until we reach a state where `(can-stack-on ?loc)`. Its intent is to force the planner to uncover `?x` so that we reach a state where we can clear it off `?loc` in one move. The formula is made a bit cumbersome by the fact that `?loc` can be the table, thus we cannot simply force a decrease in the depth of `?loc`—depth does not apply to the table.

The uncovering of `?x` is accomplished by asserting that every state, prior to the state where `(can-stack-on ?loc)`, the depth of `?x` decreases. One thing to be careful about, however, is that once we reach a state where `?x` is clear, its depth will not decrease in the next state. Instead in the next state we remove `?x` from `?loc`. Hence, we have the disjunction as the `next` condition.

This example shows that our approach can represent a wide range of control strategies. In the previous examples the control strategies expressed obvious “myopic” information about what was bad to do in various situations. The control strategy above is migrating towards a domain-specific program, specifying (in a loose manner) an entire sequence of activities. There are couple of points to be made about such complex strategies. First, our data shows that simple strategies like the trigger strategy can offer a tremendous improvement. So it could be that simple strategies are sufficient to solve the size of problems we are

faced with. Second, from a pragmatic point of view there is no reason why a planner should not be able to take advantage of a detailed domain-specific strategy if one is available.

## 8. Writing the control knowledge

The key issue raised in our approach is that of obtaining appropriate control knowledge. The examples given in the previous section demonstrate that with the appropriate knowledge we can obtain tremendous performance gains. However, these example domains sometimes require quite lengthy control formulas. So the question arises as to just how easy and practical it is to write the required control knowledge. [47]

We cannot provide any definitive answers to this question, at least not until a wider base of users and domains have been examined. Nevertheless, we have a number of reasons for believing that our approach is practical.

The most compelling evidence is anecdotal evidence from student projects. At the University of Waterloo we have used the TLPLAN system in an undergraduate AI course for a number of years. This course is taught to 4th year undergraduates, and is generally their first course in AI. Part of the course evaluation involves a project in which the students implement a planning system for some domain using the TLPLAN system. A quite impressive array of different domains have been implemented, and the students have been very successful at writing effective search control knowledge in the formalism presented here.

For example, in one implementation a car pool planner was developed [56]. This domain allows one to specify a number of locations (providing their $(x, y)$ coordinates), people, cars, and car capacities. There are operators for driving the car, for loading and unloading passengers, deadlines for people's arrival times, and the possibility of dropping passengers at near by locations from which they can walk. The control formulas written by the student included failure detection rules that terminate a plan prefix if a deadline is already missed; rules to stop the car from being driven to useless locations; and always performing a pickup or a dropoff at any location driven to. With this control knowledge the planner was able to generate plans involving 50 steps in about 60 seconds. Most importantly, the control knowledge was effective in pruning away over 80% of the worlds generated during search.

The second piece of evidence comes from the fact that we have found, both in the domains we have implemented and also reflected in the student projects, that there is considerable "reuse" of control knowledge. For example, in almost every transportation-style domain (the car pooling domain is another example of a transportation-style domain) control ideas that were developed for the logistics domain, like not moving vehicles to irrelevant location and doing all of the necessary actions at a location prior to moving, can be reused. Similarly, the idea of preserving a condition that it would be wasteful to destroy is quite common in many domains: in the blocks world good towers are to be preserved and in the tire domain we want to preserve having various tools until they are no longer

[47] Ultimately we will of course like to develop mechanisms for automatically generating the appropriate control knowledge. Research on various learning mechanisms and techniques for static domain analysis is ongoing. However, since this research is currently preliminary, the system still requires the user to write good control knowledge and hence the question of how easy it is to do this remains.

required. In other words various widely applicable meta-control principles seem to exist. Categorizing and formalizing such principles is an interesting topic for future research.

Finally, the last piece of evidence we can supply has to do with the fact that our approach supports a powerful incremental style of development. In particular, it is very easy to modify the control formula and run the planner to determine the difference in performance. Because the control formula has a compositional semantics, the changes are quite modular. For example, if we add a new conjunct that new conjunct will not alter the pruning achieved by the previous components of the formula. In this manner one can examine the state sequences searched by the planner, often determine if these sequences are doing something that can be avoided, and then modify the control formula to eliminate that behavior. This incremental improvement can be stopped at any time if the planner works well enough for the problem sizes being contemplated. As shown in the previous section, often simple control formulas can yield dramatic improvements.

## 9. Other approaches to utilizing domain information

Our work is by no means the first to suggest the use of domain-specific information in planning. One of the longest traditions in AI planning has been work on HTN planning [48,55,62], and more recently work has been done on formalizing the ideas on which HTN planning is based [21]. HTN planning requires specifying much more information about the planning domain than does classical planning. In particular, in addition to the primitive operators the planner must be given a collection of tasks and task decompositions. These tasks identify common sub-plans in the domain and their decompositions describe the various ways that these sub-plans can be solved. By working down from the top level task to the primitive actions, HTN planners can avoid large parts of the search space. In particular, they will only explore the primitive action sequences that arise from some sequence of task decompositions. Such a hierarchical arrangement can yield an exponential speedup in search time.

The specified task decompositions provide the planner with search control knowledge. In particular, the decompositions eliminate a large number of physically feasible primitive action sequences, much like the search control formulas used in this work. This view of HTN planners was made very clear by Barrett and Weld [8] who showed how the specified task decompositions could be used to prune partially ordered plans composed of primitive actions. The pruning was accomplished with a parsing algorithm.

The language and representation used by HTN planners for their control knowledge is quite distinct from that suggested here, but both seem to be useful. Some pieces of control knowledge seem to be most naturally represented as a hierarchical decomposition of tasks, while other pieces of knowledge seem to be most naturally expressed as information about "bad state sequences" using our formalism. It would seem that there is scope for both types of information, and an interesting topic for future research would be to examine mechanisms for combining both types of information.

Another early planning system to take the issue of control information seriously was the PRODIGY system [16]. PRODIGY employs search control rules, which act like an expert system for guiding search. The PRODIGY approach to specifying control information has

two main disadvantages. First, the approach is very hard to use. In particular, their control rules required one to understand the planning algorithm, as many of the rules had to do with algorithmic choices made by the algorithm. That is, unlike the approach presented here, simple knowledge of the domain is not sufficient to write these control rules. And second, although the control rules give some speedups, these speedups were not that great: even with search control PRODIGY remains a relatively slow planner.

The blocks world illustrates these difficulties well. PRODIGY employed 11 rules for the blocks world. For example, one of the rules is

```
(CONTROL-RULE SELECT-BINDINGS-UNSTACK-CLEAR
  (if (and (current-goal (clear <y>))
           (current-ops (UNSTACK))
           (true-in-state (on <x> <y>))))
  (then select bindings ((<ob> . <x>)
                         (<underob> . <y>))))
```

This rule says that if the planner is currently working on a goal of clearing a block $y$, $x$ is on $y$ in the current state, and it is currently considering regressing the goal through an unstack operator, then it should select a specific binding for the unstack operator. Such "binding" rules require the user to understand how the planner utilizes bindings during planning search. The notion of a binding has nothing to do with the domain, rather it has to do with the planning algorithm.

Another example is the rule

```
(CONTROL-RULE ARM-EMPTY-FIRST
  (if (and (candidate-goal (arm-empty))
           (true-in-state (holding <x>))))
  (then select goal (arm-empty)))
```

This rule says that if the planner is considering the goal of having the robot have its hand empty, and it is true in the current state that it is holding a block $x$, then it should commit to working on the goal hand empty. Again this rule requires that the user know about the difference between a candidate goal and the current goal, and how this difference can affect the planner's operation.

Even with these 11 rules, PRODIGY was unable to solve any of our random blocks world problems that involved more than 9 blocks (and it failed to solve 6 out of the 10 problems involving 9 blocks).

On the other hand, PRODIGY's rules were designed to be learned automatically, so perhaps transparency is not such a critical issue. Nevertheless, current learning algorithms have not yet reached the stage where they can generate truly effective control rules. This often leaves the user of the system with no choice but to attempt to construct some control rules by hand, and as indicated above this can be a very difficult task. A lot of innovative work on learning and reasoning with planning domains has come out of the PRODIGY project, but performance of the scale demonstrated by our approach has not been achieved.

There has also been some more recent work on utilizing domain-dependent control knowledge by Srivastava and Kambhampati [52] and by Kautz and Selman [34]. Srivastava and Kambhampati present a scheme for taking domain-specific information similar to that

used by TLPLAN and using that information as input to a complex program synthesis system. The end result is an semi-automatically constructed planning system that is customized for that domain. For example, in the logistics domain some of the domain-specific information they utilize includes:

(1) Planes should not make consecutive flights without loading or unloading a package.
(2) Once a package reaches its goal location it should not be moved.

The reader will recognize these rules as part of the domain information we encoded in TLPLAN. In fact, TLPLAN's representation is more general than that allowed by Srivastava and Kambhampati, and all of the domain-specific information mentioned in their paper can easily be encoded in the logic TLPLAN utilizes. Unlike TLPLAN however, their approach requires a complex program synthesis step to make use of this information (a customized planner must first be synthesized). In TLPLAN the control information is simply part of the planner's input. Furthermore, the empirical results presented in [52] show performance that is orders of magnitude inferior to TLPLAN. For example, their customized planners took about one minute each to solve the standard tire "fixit" problem, a 12 package logistics problem, and a 14 block problem. TLPLAN takes 0.06 seconds to solve the tire fixit problem, about 3 seconds on average to solve 12 package logistics problems, and about 0.24 seconds on average to solve 14 block problems. Nevertheless, the methods they developed for synthesizing customized planners demonstrate an interesting alternative approach to utilizing domain-specific information.

Finally, Kautz and Selman [34] have recently investigated the use of domain-specific information in their SATPLAN paradigm. Like us they have adopted an approach in which the domain information is logically represented and is independent of the planner's operation. Specifically, they represent extra domain knowledge as additional propositional clauses, and like us they have noticed that a state-based representation seems to be the most promising for exploiting such knowledge. Their results are still preliminary, but show some promise. In particular, they also show that speedups are possible, but do not attain a speedup that is competitive with TLPLAN's performance. The major hurdle that their approach faces, if it is to be scaled up to the size of problems TLPLAN can handle, is the size of the propositional theories it generates. More effective ways need to be found for dealing with theories of this size or for incrementally simplifying these theories so that smaller theories can be generated. For example, in our experiments we found that logistics problems with 16 packages generated theories containing more than $10^6$ clauses and $10^5$ variables. With theories of this size even polynomial time processing takes a considerable amount of time.

## 10. Conclusions and future work

In this paper we have presented a rich representation for domain-specific control knowledge and we have shown how such knowledge can be utilized by an AI planning system to make planning more efficient. Our empirical evidence indicates that

(1) such information is available in many, if not most, domains and that
(2) with such information we can reach a new level of planning performance.

We believe that the size of problems TLPLAN can solve has never been approached before.

Given the success of this approach the natural and most pressing question becomes: where does the control information come from? In this paper we have taken a pragmatic

approach, and have assumed that it will come from the user just like the other forms of knowledge the user needs to specify when developing a planning domain. Our empirical studies show that this is not an unreasonable approach, and that some form of control knowledge is usually available. Nevertheless, it is equally clear that much of this knowledge has a more abstract form—many of the domains have similar meta-level strategies. Furthermore, it is also clear that some of this knowledge could be automatically derived from the operator descriptions (in conjunction, perhaps, with the initial state). So an important area for future research will be to employ learning and reasoning techniques to automatically generate this domain-specific knowledge. There is a considerable body of work that can be built on in this area, e.g., [22,36,42,45]. The work by McDermott [40] and Bonet et al. [14] can also be viewed in this light. In these works search heuristics are computed dynamically during search. These heuristics try to estimate whether or not the search is making progress towards the goal. Potentially, similar ideas could be used for the off-line construction of search control formulas that provide the same effect as we obtain with our use of the GOAL modality.

Another area in which work could be done is to develop ways in which the temporal logic developed here can be utilized to control other kinds of planning algorithms. It should be relatively easy convert the temporal logic expressions into propositional logic (once we have a fixed initial and goal state), and thus find ways to use our representation in SATPLAN-based approaches.

Finally, we are actively working on methods for extending our approach beyond classical planning. The basic system already handles resources, but we still have empirical work to do to test how effective it can be in domains that make heavy use of resource reasoning. We have extended our approach to generate plans that satisfy temporally extended goals [4]. Such goals generalize the safety and maintenance goals mentioned in [60]. And most recently we have developed a STRIPS database approach to planning and sensing under incomplete knowledge [5]. In future, work we plan to combine this with search control to construct a planner capable of planning and sensing under incomplete knowledge.

## 11. On-line material

The TLPLAN planning system, all of the test suites, and the raw data collected in our experiments is available via the web site http://www.cs.toronto.edu/~fbacchus.

## References

[1] AIPS98, Artificial Intelligence Planning Systems 1998 planning competition. http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html, 1998.

[2] F. Bacchus, F. Kabanza, Planning for temporally extended goals, in: Proc. AAAI-96, Portland, OR, 1996, pp. 1215–1222.

[3] F. Bacchus, F. Kabanza, Using temporal logic to control search in a forward chaining planner, in: M. Ghallab, A. Milani (Eds.), New Directions in AI Planning, IOS Press, Amsterdam, 1996, pp. 141–153.

[4] F. Bacchus, F. Kabanza, Planning for temporally extended goals, Ann. Math. Artificial Intelligence 22 (1998) 5–27.

[5] F. Bacchus, R. Petrick, Modeling and agent's incomplete knowledge during planning and execution, in: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-98), Trento, Italy, 1998, pp. 432–443.

[6] M. Barbeau, F. Kabanza, R. St-Denis, Synthesizing plant controllers using real-time goals, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 791–798.
[7] A. Barrett, K. Golden, J.S. Penberthy, D. Weld, UCPOP user's manual (version 2.0), Technical Report TR-93-09-06, University of Washington, Department of Computer Science and Engineering, 1993. (ftp://cs.washington.edu/pub/ai/).
[8] A. Barrett, D. Weld, Task-decomposition via plan parsing, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1117–1122.
[9] A. Barrett, D.S. Weld, Partial-order planning: Evaluating possible efficiency gains, Artificial Intelligence 67 (1) (1994) 71–112.
[10] M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger, Integrated plan generation and recognition—A logic-based approach, Technical Report RR-91-26, DFKI, 1991.
[11] S. Biundo, W. Stephan, Modeling planning domains systematically, in: Proc. European Conference on Artificial Intelligence (ECAI-96), Wiley, New York, 1996, pp. 599–603.
[12] S. Biundo, W. Stephan, System assistance in structured domain model development, in: Proc. IJCAI-97, Nagoya, Japan, Morgan Kaufmann, San Mateo, CA, 1997, pp. 1240–1245.
[13] A. Blum, M. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1997) 281–300.
[14] B. Bonet, G. Loerincs, H. Geffner, A robust and fast action selection mechanism for planning, in: Proc. AAAI-97, Providence, RI, 1997, pp. 714–719.
[15] C. Boutilier, R. Dearden, Using abstractions for decision-theoretic planning with time constraints, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1016–1022.
[16] J.G. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Khan, C. Knoblock, S. Minton, A. Pérez, S. Reilly, M. Veloso, X. Wang, Prodigy 4.0: The manual and turorial, Technical Report CMU–CS–92–150, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.
[17] K. Currie, A. Tate, O-plan: The open planning architecture, Artificial Intelligence 52 (1991) 49–86.
[18] T. Dean, L.P. Kaelbling, J. Kerman, A. Nicholson, Planning with deadlines in stochastic domains, in: Proc. AAAI-93, Washington, DC, 1993, pp. 574–579.
[19] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Elsevier Science, Amsterdam/MIT Press, Cambridge, MA, 1990, Chapter 16, pp. 997–1072.
[20] K. Erol, D.S. Nau, V.S. Subrahmanian, On the complexity of domain-independent planning, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 381–386.
[21] K. Erol, Hierarchical task network planning: Formalization, analysis, and implementation, Ph.D. Thesis, University of Maryland, College Park, MD, 1995.
[22] O. Etzioni, Acquiring search-control knowledge via static analysis, Artificial Intelligence 62 (2) (1993) 255–302.
[23] J.W. Garson, Quantification in modal logic, in: D. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, Vol. II, Reidel, Dordrecht, 1977, pp. 249–307.
[24] A. Gerevini, L. Schubert, Accelerating partial-order planners: Some techniques for effective search control and pruning, J. Artificial Intelligence Res. 5 (1996) 95–137.
[25] C. Green, Application of theorem proving to problem solving, in: Proc. IJCAI-69, Washington, DC, 1969, pp. 219–239.
[26] N. Gupta, D.S. Nau, On the complexity of blocks-world planning, Artificial Intelligence 56 (1992) 223–254.
[27] J.Y. Halpern, M.Y. Vardi, Model checking vs. theorem proving: A manifesto, in: J.A. Allen, R. Fikes, E. Sandewall (Eds.), Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-91), Cambridge, MA, Morgan Kaufmann, San Mateo, CA, 1991, pp. 325–334.
[28] M. Hennessy, The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics, Wiley, New York, 1990.
[29] D.S. Johnson, A catalog of complexity classes, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A, Elsevier Science, Amsterdam/MIT Press, Cambridge, MA, 1990, Chapter 2, pp. 69–161.
[30] D. Joslin, M. Pollack, Least-cost flaw repair: A plan refinement strategy for partial-order planning, in: Proc. AAAI-94, Seattle, WA, Morgan Kaufmann, San Mateo, CA, 1994, pp. 1004–1009.
[31] P.C. Kanellakis, Elements of relational database theory, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Elsevier Science, Amsterdam, 1990, pp. 1071–1156.

[32] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proc. AAAI-96, Portland, OR, 1996, pp. 1194–1201.
[33] H. Kautz, B. Selman, Blackbox: A new approach to the application of theorem proving to problem solving, 1998. System available at http://www.research.att.com/\string~kautz.
[34] H. Kautz, B. Selman, The role of domain-specific knowledge in the planning as satisfiability framework, in: Proc. International Conference on Artificial Intelligence Planning, 1998, pp. 181–189.
[35] D. Kibler, P. Morris, Don't be stupid, in: Proc. IJCAI-81, Vancouver, BC, 1981, pp. 345–347.
[36] C. Knoblock, Automatically generating abstractions for planning, Artificial Intelligence 68 (2) (1994) 243–302.
[37] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, in: European Conference on Planning, 1997, pp. 273–285. System available at http://www.informatik.uni-freiburg.de/ koehler/ipp.html.
[38] J. Laird, A. Newell, P. Rosenbloom, SOAR: An architecture for general intelligence, Artificial Intelligence 33 (1) (1987) 1–67.
[39] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specication, Springer, New York, 1992.
[40] D. McDermott, A heuristic estimator for means-end analysis in planning, in: Proc. Third International Conference on AI Planning Systems, 1996.
[41] S. Minton, J. Bresina, M. Drummond, Total-order and partial-order planning: A comparative analysis, J. Artificial Intelligence Res. 2 (1994) 227–262.
[42] S. Minton, Learning Search Control Knowledge, Kluwer Academic, Dordrecht, 1988.
[43] S. Parekh, A study of procedural search control in simon, 1996. http://www.cs.washington.edu/homes/sparekh/quals.ps.
[44] E. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, in: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-89), Toronto, Ont., 1989, pp. 324–332.
[45] M. Poet, D.E. Smith, Threat-removal strategies for partial-order planning, in: Proc. AAAI-93, Washington, DC, 1993, pp. 492–499.
[46] S.J. Rosenschein, Plan synthesis: A logical perspective, in: Proc. IJCAI-81, Vancouver, BC, 1981, pp. 115–119.
[47] S. Russell, P. Norvig, Artificial Intelligence A Modern Approach, Prentice Hall, Englewood Cliffs, NJ, 1995.
[48] E. Sacerdoti, Planning in a hierarchy of abstraction spaces, Artificial Intelligence 5 (1974) 115–135.
[49] B. Selman, Near-optimal plans, tractability and reactivity, in: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-94), Bonn, Germany, 1994, pp. 521–529.
[50] A.P. Sistla, Safety, liveness, and fairness in temporal logic, Formal Aspects of Computing 6 (1994) 495–511.
[51] A.P. Sistla, E.M. Clarke, The complexity of propositional linear temporal logic, J. ACM 32 (1985) 733–749.
[52] B. Srivastava, S. Kambhampati, Synthesizing customized planners from specifications, J. Artificial Intelligence Res. 8 (1998) 93–128.
[53] W. Stephan, S. Biundo, Deduction-based refinement planning, in: Proc. International Conference on Artificial Intelligence Planning, AAAI Press, Menlo Park, CA, 1996, pp. 213–220.
[54] J. Tash, S. Russell, Control strategies for a stochastic planner, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1079–1085.
[55] A. Tate, Generating project networks, in: Proc. IJCAI-77, Cambridge, MA, 1977, pp. 888–893.
[56] C. Thompson, Carpoolworld, Undergraduate project in CS486, University of Waterloo, Ontario, 1998.
[57] M.Y. Vardi, The complexity of relational query langauges, in: Proc. 14th Ann. ACM Symp. on Theory of Computing, 1982, pp. 176–185.
[58] M.Y. Vardi, Computational model theory: An overview, Logic J. IGPL 6 (4) (1998) 601–623.
[59] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: The PRODIGY architecture, J. Experimental and Theoretical Artificial Intelligence 7 (1) (1995).
[60] D. Weld, O. Etzioni, The first law of robotics (a call to arms), in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1042–1047.
[61] D.S. Weld, An introduction to least commitment planning, AI Magazine 15 (4) (1994) 27–61.
[62] D. Wilkins, Practical Planning: Extending the Classical AI Planning Paradigm, Morgan Kaufmann, San Mateo, CA, 1988.