Using Temporal Knowledge in a Constraint-Based Planner

Aaron R. Bradley
arbrad@Stanford.edu

1 Introduction

Incorporating domain-specific temporal knowledge into a constraint-based planner can significantly decrease the computation time. This paper describes a method for compiling and embedding user-specified temporal information into the constraint problem translation of a planning problem. We provide empirical results for our framework implemented within the BLACKBOX [9] planning system; the data indicate that using temporal domain information accelerates planning.

A classical planning problem is usually specified as sets of *objects*, *actions*, propositional *predicates* to describe the relationships of the objects in the world, a description of the initial state of the world, and a description of the desired final state. The actions and predicates make up the planning *domain*, while a specific set of objects, an initial state, and a final state constitute a particular *planning problem*. Actions act on objects; when taken, an action may change what is true about the world through its *effects*. Further, an action may have a constraint, called its *precondition*, specifying what needs to be true about the world for it to be taken; traditionally, the scope of the *precondition* is a subset of the current state, rather than the history of world states. If an action A with precondition P and effect E is taken at time t, then we may describe this event formally by the formula

$$A_t \rightarrow P_t \wedge E_{t+1}, \tag{1}$$

which states that taking action A at time t requires that its precondition hold in the world at time t and its effect hold in the world at time $t+1$. A *plan* consists of a sequence of sets of parallel actions such that (1) holds for each action at each level, for all possible orderings within the sets of parallel actions. In a *serial* plan, the sets of parallel actions are singletons. Planning algorithms are written with the primary goal of finding a plan when one exists; other subgoals may include minimizing the number of actions, minimizing the number of time-steps, or minimizing the time to compute the plan.

The use of *domain-specific information* may be incorporated into a planning algorithm to increase speed, decrease the length or number of actions in a plan, or meet some other goal. Generally, an action's precondition specifies the minimum necessary facts that must hold for the action to be taken; however, in many cases, one may conclude that taking the action except when a stronger condition holds leads to undesirable results such as a longer plan or no plan at all (and, hence, backtracking and trying another action). This extra information may be incorporated as a strengthened precondition, G, which we call an *action guard*. Then taking a guarded action requires

$$A_t \rightarrow P_t \wedge G_t \wedge E_{t+1} \tag{2}$$

to hold. Other information may be incorporated in formulas over world facts as *global constraints*, which restrict the actions that are taken by constraining the world-states. In addition to speeding planning or decreasing plan lengths, global constraints may be actual goals of the plan as, for example, temporally-extended goals [2], [5], [6].

Whereas action guards, like action preconditions, are usually specified over the state at which the action is taken, our work follows that of Bacchus and Kabanza [1] in allowing guards to be specified

over the history of world-states leading up to the action. Further, whereas global constraints often take a restricted form such as a propositional assertion that is invariant throughout the plan, we present a framework for handling arbitrary temporally specified constraints—*i.e.* constraints that may relate facts in world-states arbitrarily far away in time-steps. Reformulating (2) with a temporal action guard, we have

$$A_t \rightarrow P_t \wedge G_{[1,t]} \wedge E_{t+1}. \tag{3}$$

for $G_{[1,t]}$ a formula over the sequence of world-states up to time t.

Unlike Bacchus and Kabanza, who implemented their temporal framework within a forward-chaining planner (see §2), we present a method for using temporally-specified knowledge within a constraint-based planner, which generates a constraint problem from a planning problem (see §2). While we provide further details later, a forward-chaining approach is the natural choice for incorporating temporal knowledge, whereas a constraint-based approach, in requiring all knowledge to be compiled into a constraint problem, offers several challenges: first, preserving temporal semantics and, second, maximizing the ratio between the increase in speed from adding extra constraints and the decrease in speed from increasing the size of the constraint problem. Like TLPLAN, our framework, `tk2sat`, accepts action guards and global constraints using a temporal language based on linear temporal logic. However, our approach diverges there: `tk2sat` generates a propositionally-labeled automaton, modeled after *Büchi automata* but over finite sequences, for each formula in the specification. These automata are in turn instantiated with specific actions and compiled into the constraint problem.

This paper presents a framework for handling temporally-specified action guards and global constraints. We show with examples that nontrivial temporal assertions increase the speed of planning even in a relatively simple domain; moreover, the speed-up is especially apparent in large problems, making previously hard problems tractable. While we focus on augmenting the planner with domain-specific information to accelerate planning, our compilation technique may be applied elsewhere, such as to constraint-based planners that plan over more complicated domains with complex temporal properties. Further, our intermediate representation may even provide a more efficient means of applying constraints in a forward- or backward-chaining planner than is offered by the TLPLAN approach.

§2 introduces the classical planning domain, reviews previous work relevant to our research, and provides an overview of linear temporal logic (LTL), Büchi automata, and the particle-tableau method of translating a LTL assertion to its equivalent automaton. §3 presents our technique for embedding temporal knowledge within a SAT or CSP encoding of a planning problem; we provide a proof of correctness of the encoding, and a bound on its size polynomial in the size of the Büchi automaton representation of the temporal constraint. §4 describes our C++ framework, `tk2sat`, for handling temporal knowledge in the planning system BLACKBOX; additionally, we present some empirical results and discussion. We make some final remarks in §5 and suggest avenues for further research.

2 Background

This section reviews relevant previous work in the classical planning domain. It then provides an introduction to linear temporal logic, metric interval temporal logic, Büchi automata, and the particle-tableau method for converting a linear temporal logic assertion to a Büchi automaton.

2.1 Previous Work in Classical Planning

Classical planning algorithms handle domains described with propositional languages, hence domains with (usually) discrete time, propositionally-specified initial and goal states, and discrete and finite objects. Planning in this context may be described as a problem of *search* and *optimization* over a large but finite search space of either world-states or plans. We outline several algorithms for classical planning; however, David Smith *et al.* provide a superb overview of the classical planning domain, so we focus on algorithms relevant to the rest of this paper and refer the interested reader to [15]. Additionally, Daniel Weld describes several planning algorithms in great depth [16].

The STRIPS language for classical planning specifies a domain with world-state knowledge as literals; a discrete and finite set of domain objects; and actions, parameterized by domain objects, as a set of preconditions and effects. For example, a template for a literal in the `logistics` domain is

```
(in-city ?obj ?city)
```

which, when instantiated by two objects from the problem domain, is true if and only if `?obj` is in `?city`. Objects for a particular problem might include `truck1`, `san-diego`, or `san-diego-post-office`. Finally, the action

```
(:action LOAD-TRUCK
 :parameters (?obj ?truck ?loc)
 :precondition (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
                    (at ?truck ?loc) (at ?obj ?loc))
 :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))
```

describes a template for a specific action that loads `?obj` into `?truck`. It may only be taken in a state where the literals in the precondition hold; when taken, the literals in the effect hold in the subsequent state. Many extensions to basic STRIPS are possible, including conditional effects, disjunctions of preconditions, quantification, and resource constraints. The series of versions of the PDDL [13] specification language provides a good indication of the historical increase in expressibility.

With the notation fixed for specifying a domain and planning problem, we face the problem of actually finding a plan. The most intuitive planning structure is based on forward-chaining search: the planner begins with a planning-node corresponding to the world-state (or world-states, if the initial state is not completely specified) and searches the space of world-states by generating next-states with actions in the domain. The search is complete when it reaches a world-state node that satisfies the goal condition. While forward-chaining planners are generally slower and produce inefficient plans compared to other architectures, at least one successful recent planner is based on forward-chaining: TLPLAN [1], the motivation for the work presented here, allows users to augment domains with temporally-specified constraints, resulting in significantly faster running times and shorter plans. Bacchus and Kabanza use the forward-chaining approach to take advantage of the semantics of linear temporal logic, which specify next-state formulas based on the formula describing the current state. TLPLAN is thus able to generate and evaluate a compact representation of the temporal constraints as the planner moves forward in the search space. We postpone further discussion of TLPLAN until after we formally introduce LTL.

Backward-chaining planners such as HSPr [4] use a similar search space, but find the plan by starting at the goal and working backwards to the initial state. The motivation for these planners

usually is the ability to incorporate powerful heuristics that a forward-chaining architecture either cannot use or uses less efficiently.

GRAPHPLAN [3] and its descendants rely on *mutex constraints* to build a *planning graph* that represents all feasible plans with respect to the action preconditions and effects and the mutex constraints. The mutex constraints are as follows:

- Two actions are mutually exclusive (*mutex*) if and only if
  - one's effects include p, while the other's include $\neg p$;
  - one's effects include p, while the other's preconditions include $\neg p$;
  - one's preconditions include p, the other's include q, and p and q are mutex propositions.
- Two propositions are mutex if and only if
  - one is p, while the other is $\neg p$;
  - the *support* of p is pairwise mutex with the support of q.

The support of a proposition p is the set of actions that include p in their effects. In the first phase of the algorithm, GRAPHPLAN constructs the planning graph level-by-level, stopping at the first level at which all propositions specified by the goal are present and no two are mutex. Then, it searches backwards through the planning graph until it finds a sequence of (possibly parallel) actions such that the state at the first level satisfies the initial condition; alternately, it may fail to find such a sequence. In the latter case, it adds a level to the planning graph and tries again. If searches are allowed to complete, the resulting plan is optimal in length.

Finally, constraint-based planners map a planning problem onto a constraint satisfaction problem (CSP). One specialized CSP structure is SAT, which is the set of constraint problems where each variable has domain $\{\mathbf{true}, \mathbf{false}\}$ and constraints describe the conjunctive normal form (CNF) of a propositional-logic formula. Here, we use a set to represent the disjunction of its members, and a list of such sets to represent the conjunction of clauses. SATPLAN [10] encodes the planning domain and particular problem as a SAT instance, which is solved by a generic SAT-solver.

BLACKBOX [9] encodes the planning graph of GRAPHPLAN into a SAT instance, hands the instance to a SAT solver, and extracts the plan from the resulting proposition-assignment if a solution is returned. It allows users to define a problem-solving strategy consisting partly of a list of the solvers that should attack the SAT instances, and the length they should run before returning without a solution. The compilation introduces a variable at each level for each action and proposition appearing in the planning graph; a **true** assignment to an action variable indicates that the action is taken at that level, while a **true** assignment to a proposition variable indicates that the proposition holds in that world-state. An action A and its preconditions P and effects E are encoded as an implication, $A \rightarrow P \wedge E$, while a mutex between p and q results in $(\neg p \vee \neg q)$. Kambhampati and Do propose encoding the planning graph as a general CSP instance [8]; here, a CSP variable with a domain containing its support and the constant $\perp$ corresponds to each proposition at each level of the planning graph. A mutex between actions A_1 and A_2 with effects p_1 and p_2, respectively, results in the implication $p_1 = A_1 \rightarrow p_2 \neq A_2$. Action A with preconditions p_i and effect q is encoded as $p = A \rightarrow \bigwedge_i p_i \neq \perp$.

Constraint-based planners have several unique advantages over other planning architectures. First, they exploit advances in SAT and CSP solvers by mapping planning onto these domains. For example, with the release of the SAT-solver CHAFF [14], BLACKBOX's speed increased dramatically

with little more than providing an interface to CHAFF. Second, they act on the observation that the ordering implicit in forward or backward search is not necessarily the best way to attack the search problem: SAT and CSP solvers employ heuristics to attack problems in an order that depends on the problem itself, rather than the overall search architecture.

We propose to combine the expressiveness of TLPLAN with the advantages of a constraint-based planner. Whereas TLPLAN exploits the forward-chaining semantics of its specification language, we present a method of efficiently compiling the temporal specification into the SAT or CSP representation of the planning problem. Further, we propose that the intermediate representation of the temporal specification, a propositionally-labeled automaton, could also be used efficiently in forward- and backward-chaining planners as an alternate strategy to generating formulas throughout the search. While we focus on two particular applications, action guards and global constraints, the technique could be applied elsewhere, for example, to temporally-extended goals [2], [5].

Huang *et al.* [7] describes a control framework within BLACKBOX to handle non-temporal action guards and global constraints. A user-specified action guard in their framework essentially strengthens the current-state dependent precondition of the action, although with a language that allows *bounded quantification* and arbitrary conjunction, disjunction, and negation. Bounded quantification is a modification to first-order quantification that limits the scope to a finite domain computable at SAT compile-time. Global constraints take the form of a scoped precondition/effect pair such that if the precondition holds at some level, then the effect must hold at the subsequent level. Both the precondition and effect are formulas over the world-state literals in the planning domain and are specified with bounded quantifiers and arbitrary conjunction, disjunction, and negation. The user specifies the scope of global constraints with bounded universal quantifiers. This control framework is essentially non-temporal: although global constraints consider two states at a time (the precondition and the effect), constraints involving an arbitrary number of world-states in a sequence are impossible. Further, their action guards are limited to a single world-state. Finally, the overall form of the constraints is that of an *invariant*, an assertion that *always* holds; however, in a true temporal framework, one can imagine specifying that something may *eventually* hold, or even more complex temporal relationships.

Nevertheless, Huang *et al.* successfully translates most of the constraints presented in TLPLAN's example control files, resulting in faster planning on nontrivial problems. However, they succeed because most of the temporal constraints presented with TLPLAN are of the form described above: they are invariants, and the local behavior of what is invariant relies on at most two states. In contrast, our work introduces a full temporal language into BLACKBOX for specifying guards and global constraints, while our example controls in §4.2 express complex temporal relationships that require knowledge of and constrain arbitrarily long sequences of world-states. We show in §3 and §4 that the language of the BLACKBOX control specification language is a proper subset of ours: useful constraints can be expressed with our framework that cannot be said non-temporally. One key difference in the two implementations is that Huang's can handle some constraints through pruning the plan graph, while we rely on constraining the SAT instance with extra clauses. Huang shows that for his control framework, neither approach is superior. The latter approach, however, is necessary in our framework to compile the specifications possible with our expressive temporal language.

2.2 Temporal Logic

2.2.1 Notation and Semantics

Linear temporal logic (LTL) [11] describes infinite sequences of propositional worlds. All worlds are described by a set of propositions *Prop*; each world in the sequence is defined by the propositions that hold in that world. Formally, LTL is interpreted over sequences, $\pi : N \to 2^{Prop}$, mapping the discrete (usually natural numbers) time-steps to the set of propositions holding at that time.

In addition to the set of propositions, LTL uses the Boolean operators $\wedge$ (conjunction) and $\neg$ (negation); the modal operators $\bigcirc$ (next) and $\mathcal{U}$ (until); and the constant **true**. Additionally, the Boolean operator $\vee$ (disjunction); the modal operators $\mathcal{W}$ (weak until), $\mathbf{G}$ (henceforth), and $\mathbf{F}$ (eventually); and the constant **false** are useful to define as combinations of the basic operators. The assertion $\bigcirc\varphi$ holds at a state in a sequence if φ holds in the next state. $\varphi_1 \mathcal{U} \varphi_2$ holds at a state if φ_1 holds from that state until φ_2 holds sometime in the future. $\varphi_1 \mathcal{W} \varphi_2$ is similar, except that it can hold even if φ_2 never holds at a future state; then φ_1 holds forever. $\mathbf{G}\varphi$ holds at a state if φ holds at the state and all future states; $\mathbf{F}\varphi$ holds if either the current state or some future state satisfies φ.

LTL has the following formal semantics, given in first-order logic, where we say for $\pi_i \models \varphi$ that the sequence π at time i models the LTL formula φ:

$$
\begin{array}{rll}
\pi_i \models p & \text{iff} & p \in \pi_i, \ \textit{i.e.}, \ p \text{ holds in the world } \pi_i \\
\pi_i \models \neg\varphi & \text{iff} & \pi_i \not\models \varphi \\
\pi_i \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi_i \models \varphi_1 \text{ and } \pi_i \models \varphi_2 \\
\pi_i \models \varphi_1 \vee \varphi_2 & \text{iff} & \pi_i \models \varphi_1 \text{ or } \pi_i \models \varphi_2 \\
\pi_i \models \bigcirc\varphi & \text{iff} & \pi_{i+1} \models \varphi \\
\pi_i \models \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \exists k \geq i. \, (\forall j \in [i,k). \pi_j \models \varphi_1 \ \wedge \ \pi_k \models \varphi_2) \\
\pi_i \models \mathbf{G}\varphi & \text{iff} & \forall j \geq i. \pi_j \models \varphi \\
\pi_i \models \mathbf{F}\varphi & \text{iff} & \exists j \geq i. \pi_j \models \varphi \\
\pi_i \models \varphi_1 \mathcal{W} \varphi_2 & \text{iff} & \exists k \geq i. \, (\forall j \in [i,k). \pi_j \models \varphi_1 \ \wedge \ \pi_k \models \varphi_2) \ \vee \ \forall j \geq i. \pi_j \models \varphi_1
\end{array}
$$

The sequence π *satisfies* φ if φ holds in the first state of π, *i.e.* $\pi_1 \models \varphi$; we may write this $\pi \models \varphi$. The derived modal operators are related to the basic operators as follows:

$$
\begin{array}{rcl}
\mathbf{F}\varphi & \equiv & \mathbf{true} \mathcal{U} \varphi \\
\mathbf{G}\varphi & \equiv & \neg (\mathbf{true} \mathcal{U} \neg\varphi) \\
\varphi_1 \mathcal{W} \varphi_2 & \equiv & \varphi_1 \mathcal{U} \varphi_2 \ \vee \ \mathbf{G}\varphi_1
\end{array}
$$

2.2.2 MITL

In planning, we are concerned with finite sequences of worlds; moreover, we may wish to discuss concrete intervals of time. Hence, we use metric interval temporal logic (MITL) [1] to specify assertions in our framework. The modal operators $\mathcal{U}$, $\mathcal{W}$, $\mathbf{G}$, and $\mathbf{F}$ may be parameterized by an interval $[a, b]$ for natural numbers a, b:

$$
\begin{array}{rll}
\pi_i \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 & \text{iff} & \exists k \in [a,b]. \, (\forall j \in [i,k). \pi_j \models \varphi_1 \ \wedge \ \pi_k \models \varphi_2) \\
\pi_i \models \mathbf{G}_{[a,b]} \varphi & \text{iff} & \forall j \in [a,b]. \pi_j \models \varphi
\end{array}
$$

$$\pi_i \models \mathbf{F}_{[a,b]}\varphi \quad \text{iff} \quad \exists j \in [a,b].\pi_j \models \varphi$$
$$\pi_i \models \varphi_1 \mathcal{W}_{[a,b]} \varphi_2 \quad \text{iff} \quad \exists k \in [a,b].\, (\forall j \in [i,k).\pi_j \models \varphi_1 \ \wedge \ \pi_k \models \varphi_2) \ \vee \ \forall j \in [a,b].\pi_j \models \varphi_1$$

If an interval is not specified, it is assumed to be $[1,k]$, where k is the length of the sequence π—which, as we will see, is also the length of the proposed plan.

2.2.3 From LTL to Büchi Automaton

A useful way of checking if a model π satisfies an assertion φ is to translate φ to an equivalent ω-automaton, a finite-state, propositionally-labeled automaton over infinite sequences (called ω-*words*) of worlds. The language of LTL corresponds to Büchi automata (technically, Büchi automata are more expressive than LTL) [11], which have five parts:

- N, the set of nodes, n_i;
- N_0, the set of initial nodes;
- E, the set of edges, $\langle n_1, n_2 \rangle \in N \times N$;
- μ, the node labeling;
- $\mathcal{F}$, the acceptance condition.

μ, the node labeling, maps nodes to propositional assertions—*i.e.* assertions without temporal operators. If a sequence of world-states $s_1, s_2, s_3, \ldots$ models the LTL assertion φ from which $\mathcal{A}$ is constructed, it induces a path $n_1, n_2, n_3, \ldots$ in $\mathcal{A}$ such that $\forall i.\mu(n_i) \models s_i$ and $n_1 \in N_0$; that is, the state s_i satisfies the node labeling of n_i.

The acceptance condition may be defined in several ways; the most straightforward is Muller's formulation, where $\mathcal{F}$ is a set of sets of nodes. Büchi automata are nondeterministic in general, so a sequence is accepted by an automaton if there is some induced path such that the set of nodes occurring infinitely often in the path is a set in $\mathcal{F}$.

For our domain knowledge framework, we will need to construct a variation on Büchi automata that recognizes languages of finite sequences; hence, we present the particle tableau algorithm [12] for constructing automata from LTL assertions, modifying the relevant sections later to work within our framework. Before presenting the main algorithm, we define several sub-procedures. Given a LTL assertion φ, the first step consists of pushing negatives down to literals; the following rewrite rules (congruences) make the process clearer:

$$\begin{aligned}
\neg\mathbf{F}\varphi &\equiv \mathbf{G}\neg\varphi \\
\neg\mathbf{G}\varphi &\equiv \mathbf{F}\neg\varphi \\
\neg \bigcirc \varphi &\equiv \bigcirc\neg\varphi \\
\neg(\varphi_1 \mathcal{U} \varphi_2) &\equiv (\neg\varphi_2)\mathcal{W}(\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \mathcal{W} \varphi_2) &\equiv (\neg\varphi_2)\mathcal{U}(\neg\varphi_1 \wedge \neg\varphi_2).
\end{aligned}$$

The last two congruences deserve some intuition. Essentially, a state does not satisfy $\varphi_1 \mathcal{U} \varphi_2$ if either a $\neg\varphi_1$-state occurs before φ_2 holds for the first time, or if φ_2 never occurs. The latter case explains why the congruent formula uses $\mathcal{W}$ instead of $\mathcal{U}$. A state does not satisfy $\varphi_1 \mathcal{W} \varphi_2$ if a

$\neg\varphi_1$-state occurs before φ_2 holds for the first time. Since a $(\neg\varphi_1 \wedge \neg\varphi_2)$-state must occur sometime in the future, the congruent formula uses $\mathcal{U}$.

Next, we define the closure $\tilde{\Phi}_\varphi = \mathit{closure}(\varphi)$ of a LTL assertion. The closure is calculated as a fix-point: the closure contains φ; for each member of the closure, the closure contains the member's sub-formulas, where a formula's sub-formulas are the children of its root operator, viewing the formula as a tree; finally, for each member ψ of the set whose root operator is a temporal operator other than $\bigcirc$, the closure contains $\bigcirc\psi$. Formally, the closure is calculated as follows:

SET *closure*(LTL φ)
  SET $\tilde{\Phi}_\varphi := \{ \varphi \}$
  for $\psi \in \tilde{\Phi}_\varphi$
    $\tilde{\Phi}_\varphi := \tilde{\Phi}_\varphi \cup \mathit{subformulas}(\psi)$
  for $\psi \in \tilde{\Phi}_\varphi$ of the form $\mathbf{F}\psi_1, \mathbf{G}\psi_1, \psi_1\mathcal{U}\psi_2$, or $\psi_1\mathcal{W}\psi_2$
    $\tilde{\Phi}_\varphi := \tilde{\Phi}_\varphi \cup \{ \bigcirc\psi \}$
  return $\tilde{\Phi}_\varphi$

For the next procedure, we need to divide modal operators into two expansion tables expressing the operators' *next-state semantics*. They indicate what must hold in the next state given a LTL assertion holding in this state. The first table lists the α-operators, those operators that are conjunctive in nature. If the assertion on the left holds, then the assertions in the set on the right also hold:

α	$\kappa(\alpha)$
$\varphi_1 \wedge \varphi_2$	$\{ \varphi_1, \varphi_2 \}$
$\mathbf{G}\varphi$	$\{ \varphi, \bigcirc\mathbf{G}\varphi \}$

β-operators are disjunctive in behavior: if the assertion on the left holds, then either the first assertion on the right holds, or all of the assertions in the second set hold.

β	$\kappa_1(\beta)$	$\kappa_2(\beta)$
$\varphi_1 \vee \varphi_2$	$\{ \varphi_1 \}$	$\{ \varphi_2 \}$
$\mathbf{F}\varphi$	$\{ \varphi \}$	$\{ \bigcirc\mathbf{F}\varphi \}$
$\varphi_1\mathcal{U}\varphi_2$	$\{ \varphi_2 \}$	$\{ \varphi_1, \bigcirc(\varphi_1\mathcal{U}\varphi_2) \}$
$\varphi_1\mathcal{W}\varphi_2$	$\{ \varphi_2 \}$	$\{ \varphi_1, \bigcirc(\varphi_1\mathcal{W}\varphi_2) \}$

Although $\mathcal{U}$ and $\mathcal{W}$ are treated similarly, recall that $\varphi_1\mathcal{U}\varphi_2$ guarantees that φ_2 eventually holds, while $\varphi_1\mathcal{W}\varphi_2$ does not. This difference becomes apparent when setting the acceptance condition.

Given the closure and the α- and β-formula tables, we may now define the *cover* of a set of LTL assertions. The cover of an assertion φ is a set of subsets of $\tilde{\Phi}_\varphi$ called *particles*, where each particle P is consistent (if $\psi \in P$ then $\neg\psi \notin P$). Intuitively, each subset in the cover of φ describes one possible way of satisfying φ (see Fig. 1 for an example). The cover of a subset of $\tilde{\Phi}_\varphi$ also contains subsets of $\tilde{\Phi}_\varphi$; each of these subsets describes one possible way of satisfying each formula in the original subset. The algorithm proceeds by expanding formulas using the α- and β-tables, both as left-to-right (forward) and right-to-left (inverse) expansions. It finishes when each particle in the set is closed under expansions and inverse expansions.

SET *cover*(SET Φ)

if $\exists\varphi.\varphi \in \Phi \wedge \neg\varphi \in \Phi$
  return $\{\}$
% α-expansion
if $\exists\psi \in \tilde{\Phi}_\varphi.\alpha\text{-}formula(\psi) \wedge \psi \in \Phi \wedge \kappa(\psi) \not\subseteq \Phi$
  return $cover(\Phi \cup \kappa(\psi))$
% α^{-1}-expansion
if $\exists\psi \in \tilde{\Phi}_\varphi.\alpha\text{-}formula(\psi) \wedge \kappa(\psi) \subseteq \Phi \wedge \psi \notin \Phi$
  return $cover(\Phi \cup \{\psi\})$
% β-expansion
if $\exists\psi \in \tilde{\Phi}_\varphi.\beta\text{-}formula(\psi) \wedge \psi \in \Phi \wedge \kappa_1(\psi) \notin \Phi \wedge \kappa_2(\psi) \not\subseteq \Phi$
  return $cover(\Phi \cup \{\kappa_1(\psi)\}) \cup cover(\Phi \cup \kappa_2(\psi))$
% β^{-1}-expansion
if $\exists\psi \in \tilde{\Phi}_\varphi.\beta\text{-}formula(\psi) \wedge \psi \notin \Phi \wedge (\kappa_1(\psi) \in \Phi \vee \kappa_2(\psi) \subseteq \Phi)$
  return $cover(\Phi \cup \{\psi\})$
return $\{\Phi\}$

Note that $\emptyset$ may be a particle; we represent it by $P_\emptyset$ in our diagrams.

Several operations on particles may now be defined. First, the *state* of a particle is the subset of propositional assertions; *i.e.*, the assertions not containing temporal operators:

$$state(P) \equiv \{\ \psi \in P \mid \psi \text{ is not temporal }\}.$$

This set describes what is true about the current state. The successors, $succ(P)$, is the set of particles produced by calling *cover* on the set of assertions implied to hold in the next state:

$$succ(P) = cover(\{\ \psi \mid \bigcirc\psi \in P\ \}).$$

Initial particles are those that contain φ since they are the only ones satisfying φ in the first state. All particles in $cover(\varphi)$ are necessarily initial; however, other particles may be initial because of inverse expansions.

Choosing *final* particles is more complex: it relies on several more properties of particles. As we will use these concepts later to redefine the goal conditions, we present them now in the context of infinite sequences. A *promising* formula has the form $\mathbf{F}\varphi$ or $\varphi_1\mathcal{U}\varphi_2$; the first promises φ will eventually hold, while the second promises φ_2. A *fulfilling* particle is one that either promises nothing or fulfills all promises—*e.g.*, if fulfilling particle P contains $\mathbf{F}\varphi$, then it also contains φ. A *set* of particles is fulfilling if they form a connected component of the particle tableau and together they promise nothing or fulfill all promises; here, one node may fulfill the promise of another. Using the Muller acceptance condition defined above, all fulfilling sets of particles are final. The acceptance condition distinguishes automata over infinite sequences from those over finite sequences: while the latter may have goal nodes, the former must define acceptance in terms of nodes that are visited infinitely often.

We now present the algorithm for constructing a Büchi automaton $\mathcal{A}$ from a LTL assertion φ. First, a *particle tableau* is constructed by initially adding the particles in $cover(\varphi)$, then constructing and adding the successor particles until no new successors can be added. Directed edges are added from particles to their successors; then the initial and final particles are selected as defined above. The automaton $\mathcal{A}$ is constructed by reproducing the structure of the tableau and labeling each node with the state of the corresponding particle. Formally, we define the algorithm as follows:

```
AUTOMATON automaton(LTL φ)
  TABLEAU T := cover({φ})
  for SET P ∈ T
    for SET Q ∈ succ(P)
      if Q ∉ T
        T := T ∪ {Q}
      T.edges := T.edges ∪ {⟨P, Q⟩}
  for SET P ∈ T
    if φ ∈ P
      T.initial := T.initial ∪ {P}
  Set T's goal condition.
  AUTOMATON A
  for SET P ∈ T
    NODE n
    μ(n) := state(P)
    A.nodes := A.nodes ∪ {n}
  A.edges := T.edges
  A.initial := T.initial
  A.final := T.final
  return A
```

Fig. 1 outlines the execution of $automaton(p\mathcal{U}q\mathcal{U}r)$. The execution of $cover(p\mathcal{U}q\mathcal{U}r)$ is represented by a tree; each node represents a recursive call to *cover* (except the root node). The resulting three particles contain all formulas on the path from the corresponding leaf to the root.

2.3 Temporal Logic and Planning

TLPLAN, as we previously mentioned, allows users to specify temporal constraints in the form of action guards and global constraints. They use the MITL language, augmented with bounded quantifiers, to express the constraints; further, they use the logic's next-state semantics, which are similar to those of LTL shown in the α and β tables, to generate formulas throughout the search. Specifically, for some world-state node in the forward-chaining search and its associated temporal formula, the next-state nodes are generated through applications of the planning actions, while the next-state temporal formula is generated via the next-state semantics. Then, the algorithm prunes next-state nodes that do not satisfy the state constraint of the new formula, associates the new formula with the remaining nodes, and continues the search. The state constraint of a temporal formula is the constraint that the formula places on the current state.

Our approach differs from TLPLAN in three key ways. First, their algorithm is forward-chaining, while we use BLACKBOX's constraint-based approach. Consequently—and second—whereas they may generate the constraining temporal formulas in parallel with planning, we must compile all temporal knowledge into the SAT representation of the planning graph. If the resulting formulas are too large, dealing with their size may negate the benefits of the extra constraints. Third, they use MITL for expressing constraints *and* in the actual planning algorithm, whereas we use temporal logic only for expressing temporal constraints. Indeed, while we use automata as an intermediate representation, the final form is necessarily propositional and, moreover, must be expressed in

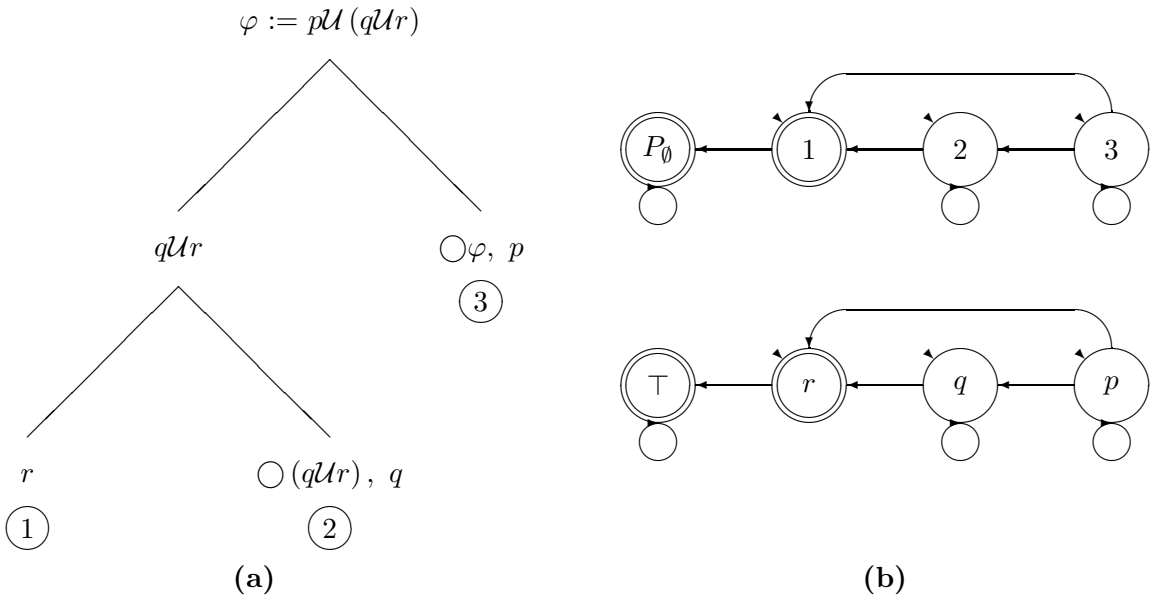


Figure 1: **(a)** A tree representation of the execution of $cover(\varphi)$; both splits are β-expansions. Applying *succ* to the resulting particles results in the special particle $P_\emptyset$ for particle 1, particles 1 and 2 for particle 2 ($cover(q\mathcal{U}r)$ does a β^{-1}-expansion back to φ, making the resulting two particles copies of 1 and 2); and all three particles for 3. **(b)** The resulting tableau showing successor relationships, and the final automaton $\mathcal{A}$.

CNF. Therefore, in building a constraint-based temporal framework, we must develop a temporal knowledge compilation approach that *preserves the temporal semantics* and produces CNF formulas that are *as small as possible*.

3 Compiling MITL to a SAT or CSP Problem

We are now ready to present a method for embedding temporal knowledge in a SAT- or CSP-based planner. *Action guards* and *global constraints* will serve as concrete applications of this technique. In a temporal context, an action guard takes the form

$$A_\ell \rightarrow \varphi_{[1,\ell]};$$

that is, taking the action at time-step ℓ implies a constraint, $\varphi_{[1,\ell]}$, potentially requiring knowledge from any world-state preceding and including the one at ℓ. If φ is specified using MITL, then $\varphi_{[1,\ell]}$ only holds if the sequence of world-states $s_1, \ldots, s_\ell$ models φ. Consequently, any promise made within $[1, \ell]$ must be fulfilled; for example, if $\varphi := \mathbf{F}p$, then at least one of $s_1, \ldots, s_\ell$ must be a p-state for the action A to be taken at time-step ℓ. Like the action's precondition and effect, the scope of φ is the set of objects that are specified as parameters of A, although the guard can also refer to the objects at any point in the preceding sequence. Further, bounded quantifiers allow action guards to refer to objects outside this scope.

Global constraints, the second application, constrain the sequence of world-states to obey a specified behavior. They do not require the plan to achieve any particular final-state goal, although a simple modification allows final-state goals to be incorporated for other applications of global constraints. A global constraint's scope is specified using a scoping bounded quantifier that specifies a subset of objects to which to apply the constraint.

The language we choose to express temporal constraints is MITL augmented with bounded quantification. To make operations on the formulas more tractable, temporal operators may not appear within the scope of a quantifier, except for the case of scoping quantifiers in global constraints. Because of this restriction, existential and universal quantifiers reduce to disjunction and conjunction over finite sets of propositional assertions; therefore, we ignore quantifiers in this section, leaving it as an implementation detail to discuss in §4. Intervals are restricted to discrete sets, here assumed to be sets of natural numbers corresponding to time-steps in the plan. Intervals are always interpreted with respect to the global time.

This section first presents a straightforward method of compiling MITL to SAT, but shows that the resulting expression is too large to be practical. Then, it presents a technique that first converts the MITL specification to an automaton and then compiles the automaton to a SAT or CSP instance; the analysis reveals a significantly smaller SAT expression.

3.1 A Naive Approach

As motivation for our compilation method, we present a simple encoding of temporal knowledge into SAT based on the first-order semantics presented in §2.2.1.

3.1.1 An Example

Consider the temporal formula

$$\varphi := p\mathcal{U}_{[a,b]}q\mathcal{U}_{[c,d]}r,$$

constraining a sequence to satisfy p until sometime between a and b, it satisfies q continuously until sometime between c and d, it satisfies r. More specific behavior results if $[a,b] \cap [c,d] \neq \emptyset$ or $a > d$. Translating into first-order logic, we have

$$\exists t_1 \in [a,b], t_2 \in [c,d]. \, (\forall t < t_1 . p_t \ \wedge \ \forall t \in [t_1, t_2 - 1]. q_t \ \wedge \ r_{t_2}) \, ;$$

since we are describing finite sequences, we can write this formula in the form

$$\bigvee_{t_1 \in [a,b], t_2 \in [c,d]} (p_1 \wedge \cdots \wedge p_{t_1 - 1} \wedge q_{t_1} \wedge \cdots \wedge \ q_{t_2 - 1} \wedge r_{t_2}) \, , \tag{4}$$

where t_1 and t_2 are constant integers in $[a,b]$ and $[c,d]$, respectively. Hence, there are $(b-a+1) \times (d-c+1)$ disjuncts, each explicitly describing one possible sequence satisfying φ. Converting (4) to CNF in the standard way (pushing negations and disjunctions in, moving conjunctions out) introduces an exponential increase in formula size; instead, we introduce one variable, ℓ_{t_1,t_2} (indexed by t_1 and t_2 for clarity) per disjunct to form

$$\begin{array}{l} \cdots \\ \wedge \ (\ell_{t_1,t_2} \ \leftrightarrow \ (p_1 \wedge \cdots \wedge p_{t_1-1} \wedge q_{t_1} \wedge \cdots \wedge \ q_{t_2-1} \wedge r_{t_2})) \\ \cdots \\ \wedge \ \bigvee_{t_1 \in [a,b], t_2 \in [c,d]} \ell_{t_1,t_2}. \end{array} \tag{5}$$

In CNF notation, we write

$$\begin{array}{l} \cdots \\ \{\ell_{t_1,t_2}, \neg p_1, \ldots, \neg p_{t_1-1}, \neg q_{t_1}, \ldots \neg q_{t_2-1}, \neg r_{t_2}\} \\ \{\neg \ell_{t_1,t_2}, p_1\} \\ \cdots \\ \{\neg \ell_{t_1,t_2}, p_{t_1-1}\} \\ \{\neg \ell_{t_1,t_2}, q_{t_1}\} \\ \cdots \\ \{\neg \ell_{t_1,t_2}, q_{t_2-1}\} \\ \{\neg \ell_{t_1,t_2}, r_{t_2}\} \\ \cdots \\ \{\ell_{a,c}, \ldots, \ell_{t_1,t_2}, \ldots, \ell_{b,d}\}. \end{array} \tag{6}$$

This is the final SAT encoding.

3.1.2 Analysis

As mentioned above, (4) contains $(b-a+1) \times (d-c+1)$ disjuncts, suggesting a size dependent on the number of temporal operators. Indeed, for each $\mathcal{U}_{[a,b]}$, $\mathcal{W}_{[a,b]}$, or $\mathbf{F}_{[a,b]}$ appearing in a formula in which all negations have been pushed down to the literals, there are $(b-a+1)$ (or, if $[a,b] = [1,k]$, then the number of time-steps, k) instantiations of the associated constant $t_i \in [a,b]$. The $\mathbf{G}$ operator is conjunctive in nature, so it does not suffer from the same disjunctive choice. Consequently, the size of the formula (before conversion to CNF) is exponential in the number of temporal operators other than $\bigcirc$ and $\mathbf{G}$.

Examining (6), we find for each (t_1, t_2) pair, t_2 clauses of the form

$$\{\neg \ell_{t_1,t_2}, p_i\},$$

and one clause of the form

$$\{\ell_{t_1,t_2}, \neg p_1, \ldots, \neg r_{t_2}\};$$

additionally, there is one global clause of the form

$$\{\ell_{a,c}, \ldots, \ell_{b,d}\}$$

resulting in at least $(c+1) \times (b-a+1) \times (d-c+1)+1$ clauses, and at least $(b-a+1) \times (d-c+1)$ new variables. Generalizing, and recalling our remark above on the $(b-a+1)$ ways of instantiating t_i for naturally disjunctive temporal operators, we find that the resulting CNF encoding of φ has $O(k^{|\mathit{disj\_temporal\_ops}(\varphi)|+1})$ clauses and $O(k^{|\mathit{disj\_temporal\_ops}(\varphi)|})$ new variables, where $\mathit{disj\_temporal\_ops}(\varphi)$ is the number of $\mathcal{U}$, $\mathcal{W}$, and $\mathbf{F}$ operators in φ.

As a concrete example, if $p\mathcal{U}q\mathcal{U}r$ constrains a potential plan with 10 levels, the resulting SAT encoding has 1001 clauses and 100 new variables. We propose a less expensive method.

3.2 A Better Approach

We now describe a method of converting domain knowledge expressed in MITL to SAT that results in significantly less clauses than the straightforward approach presented in §3.1. Given constraint φ, the algorithm generates automaton $\mathcal{A}$ using a modified form of the particle-tableau algorithm introduced in §2.2.3. Finally, it converts the automaton to a SAT or CSP encoding. If φ is guarding an action A, the final stage adds clauses indicating that either the sequence leading up to the action satisfies φ, or A is not taken.

3.2.1 Generating the Automaton

The particle-tableau algorithm discussed in §2.2.3 produces a Büchi automaton from a LTL formula expressing a constraint over an infinite sequence; however, we wish to convert MITL expressions constraining finite sequences of known length to finite propositionally-labeled automata. Therefore, we generalize the α- and β-tables to a single table of interval-augmented operators and redefine the final particles (and, hence, the acceptance condition of the resulting automaton). We end this section with an example.

3.2.2 A Modified Particle-Tableau Algorithm

Given MITL assertion $\bar{\varphi} := \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$, we convert it to $\varphi := \varphi_1 \mathcal{U}(t_{[a,b]} \wedge \varphi_2)$, where $t_{[a,b]}$ is a *proposition* parameterized by the time-step; it evaluates to **true** in the final conversion to SAT at time-step ℓ if and only if $\ell \in [a,b]$. $\mathbf{G}_{[a,b]}\varphi$ then becomes $\mathbf{G}(t_{[a,b]} \rightarrow \varphi)$ since

$$\begin{aligned}
\mathbf{G}_{[a,b]}\varphi &\equiv \neg\left(\mathbf{true}\, \mathcal{U}_{[a,b]} \neg\varphi\right) \\
&\equiv \neg\left(\mathbf{true}\, \mathcal{U}(t_{[a,b]} \wedge \neg\varphi)\right) \\
&\equiv \neg\left(\mathbf{true}\, \mathcal{U}\neg(\neg t_{[a,b]} \vee \varphi)\right) \\
&\equiv \mathbf{G}(t_{[a,b]} \rightarrow \varphi),
\end{aligned}$$

while $\mathbf{F}_{[a,b]}\varphi$ easily converts to $\mathbf{F}(t_{[a,b]} \wedge \varphi)$. Evaluating the new next-state semantics of $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ reveals

$$\varphi_1 \mathcal{U}_{[a,b]} \varphi_2 \equiv \varphi_1 \mathcal{U}(t_{[a,b]} \wedge \varphi_2)$$

$$\equiv \quad (t_{[a,b]} \wedge \varphi_2) \ \vee \ \Big(\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2)\Big);$$

consequently, $\mathbf{F}_{[a,b]}\varphi$ has next-state semantics

$$\mathbf{F}_{[a,b]}\varphi \quad \equiv \quad (t_{[a,b]} \wedge \varphi) \ \vee \ \bigcirc(\mathbf{F}_{[a,b]}\varphi)$$

and $\mathbf{G}_{[a,b]}\varphi$ has next-state semantics

$$\begin{aligned}
\mathbf{G}_{[a,b]}\varphi &\equiv \neg\Big((t_{[a,b]} \wedge \neg\varphi) \ \vee \ \bigcirc(\mathbf{true}\ \mathcal{U}_{[a,b]} \neg\varphi)\Big) \\
&\equiv (\neg t_{[a,b]} \vee \varphi) \ \wedge \ \bigcirc\neg(\mathbf{true}\ \mathcal{U}_{[a,b]} \neg\varphi) \\
&\equiv \Big(\neg t_{[a,b]} \wedge \bigcirc\neg(\mathbf{true}\ \mathcal{U}_{[a,b]} \neg\varphi)\Big) \ \vee \Big(\varphi \wedge \bigcirc\neg(\mathbf{true}\ \mathcal{U}_{[a,b]} \neg\varphi)\Big) \\
&\equiv \Big(\neg t_{[a,b]} \wedge \bigcirc\mathbf{G}_{[a,b]}\varphi\Big) \ \vee \ \Big(\varphi \wedge \bigcirc\mathbf{G}_{[a,b]}\varphi\Big)
\end{aligned}$$

Using these new next-state semantics, we reformulate the α and β expansions of LTL as the following β-formula table:

β	$\kappa_1(\beta)$	$\kappa_2(\beta)$
$\varphi_1 \wedge \varphi_2$	$\{\ \varphi_1,\ \varphi_2\ \}$	$\{\ \mathbf{false}\ \}$
$\varphi_1 \vee \varphi_2$	$\{\ \varphi_1\ \}$	$\{\ \varphi_2\ \}$
$\mathbf{F}_{[a,b]}\varphi$	$\{\ \varphi,\ t_{[a,b]}\ \}$	$\{\ \bigcirc\mathbf{F}_{[a,b]}\varphi\ \}$
$\mathbf{G}_{[a,b]}\varphi$	$\{\ \varphi,\ \bigcirc\mathbf{G}_{[a,b]}\varphi\ \}$	$\{\ \neg t_{[a,b]},\ \bigcirc\mathbf{G}_{[a,b]}\varphi\ \}$
$\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$	$\{\ \varphi_2,\ t_{[a,b]}\ \}$	$\{\ \varphi_1,\ \bigcirc(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2)\ \}$
$\varphi_1 \mathcal{W}_{[a,b]} \varphi_2$	$\{\ \varphi_2,\ t_{[a,b]}\ \}$	$\{\ \varphi_1,\ \bigcirc(\varphi_1 \mathcal{W}_{[a,b]} \varphi_2)\ \}$

κ_1 and κ_2 correspond to the disjuncts in the next-state semantics derived above. Since the next-state semantics of $\mathbf{G}_{[a,b]}\varphi$ is disjunctive, unlike $\mathbf{G}\varphi$, we have combined the α- and β-formula tables of LTL into one table. $p \wedge q$ is included by making its κ_2 set contain only **false**. Of course, this table is only a programming convenience—although I think that this unification of the tables is also more elegant. Using the original formulation and simply treating $p\mathcal{U}_{[a,b]}q$ as notation for $p\mathcal{U}(t_{[a,b]} \wedge q)$ produces identical automata. Using this table, *cover* becomes:

```
SET cover(SET Φ)
  if (∃φ.φ ∈ Φ ∧ ¬φ ∈ Φ) ∨ false ∈ Φ
    return {}
  % β-expansion
  if ∃ψ ∈ Φ̃_φ.β-formula(ψ) ∧ ψ ∈ Φ ∧ κ1(ψ) ∉ Φ ∧ κ2(ψ) ⊈ Φ
    return cover(Φ ∪ {κ1(ψ)}) ∪ cover(Φ ∪ κ2(ψ))
  % β^-1-expansion
  if ∃ψ ∈ Φ̃_φ.β-formula(ψ) ∧ ψ ∉ Φ ∧ (κ1(ψ) ∈ Φ ∨ κ2(ψ) ⊆ Φ)
    return cover(Φ ∪ {ψ})
  return {Φ}
```

The second change concerns the definition of a final node. Whereas a Büchi automaton accepts an infinite sequence if it induces a path visiting specified nodes infinitely often, our propositionally-labeled finite automaton should accept a sequence if the induced path ends in a goal node. Recalling that final automaton nodes correspond to final particles, we redefine the criteria for a final particle: a particle is final if and only if

- it promises nothing: it does not contain a formula of the form $\varphi_1 \mathcal{U} \varphi_2$ (promises φ_2), $\mathbf{F}\varphi$ (promises φ), or $\bigcirc\varphi$ (promises φ);
- or it fulfills all promises: it contains φ_2, φ, or φ, meeting the above promises, respectively.

Recall that the $succ(P)$ set used to define successors and directed edges guarantees that if $\bigcirc\varphi$ holds in the current state, then φ holds in the next state. Consequently, in the context of infinite sequences, $\bigcirc\varphi$ is not included as a promising formula because it is always fulfilled; however, for finite sequences, we include $\bigcirc\varphi$ as promising because a sequence ending in a $\bigcirc\varphi$-state should not be accepted.

One possibly useful constraint that is impossible to capture with these semantics is that the interesting behavior of the automaton occurs at the end of the sequence. For example, for guard $p\mathcal{U}q\mathcal{U}r$, a sequence may pass through p, q, and r states early, leaving the remainder of the sequence unconstrained. If this is the behavior we wish to enforce—*i.e.*, we care only that *sometime* before taking action A, such a subsequence occurs—then this guard is useful; however, if instead we want the subsequence to occur and *end* just when action A is taken, then we are out of luck, at least with this pure formulation of MITL. Hence, we want to add the ability to specify that the interesting behavior should occur at the end of the sequence. This specification actually has an easy solution: make non-goals the trivial $\top$-labeled nodes (which do not constrain the state) that would otherwise be goal nodes. Then the end of a sequence cannot be unconstrained, since all remaining goal nodes have some nontrivial label. See below for an example. §4.1.1 shows how the user may specify this constraint.

3.2.3 An Example

Recall the Büchi automaton in the example of §2.2.3 for the LTL assertion $p\mathcal{U}q\mathcal{U}r$. We construct here a finite propositionally-labeled automaton $\mathcal{A}$ for the MITL assertion

$$\varphi := p\mathcal{U}_{[a,b]}q\mathcal{U}_{[c,d]}r.$$

Fig. 2**(a)** shows the tree for $cover(\varphi)$ and the resulting particles. Recalling from §2.2.3 that

$$succ(P) \;=\; cover(\{\ \psi \mid \bigcirc\psi \in P\ \}),$$

finding the successors of the resulting particles leads to two new particles, shown in Fig. 2**(b)**. The final particle tableau is shown in Fig. 3; initial particles are those containing φ as in §2.2.3, while final particles are found as defined above. The resulting automaton is shown in Fig. 4. If the user specifies that the interesting behavior should occur at the end of the sequence, then the goal-node labeled $\top$ is made a non-goal (or simply removed). The resulting automaton will only accept sequences that end in r.

3.2.4 A Final Remark on Automata

The resulting automaton can be useful in some planning frameworks. For example, in a forward-chaining planner, the algorithm may track the possible paths in a constraining automaton induced by a sequence, allowing (for an action guard) an action only if at least one path is currently in a final state, or constraining the choice of successor path nodes for both action guards and global constraints. Tracking is accomplished by keeping with each planning node a bit vector with an

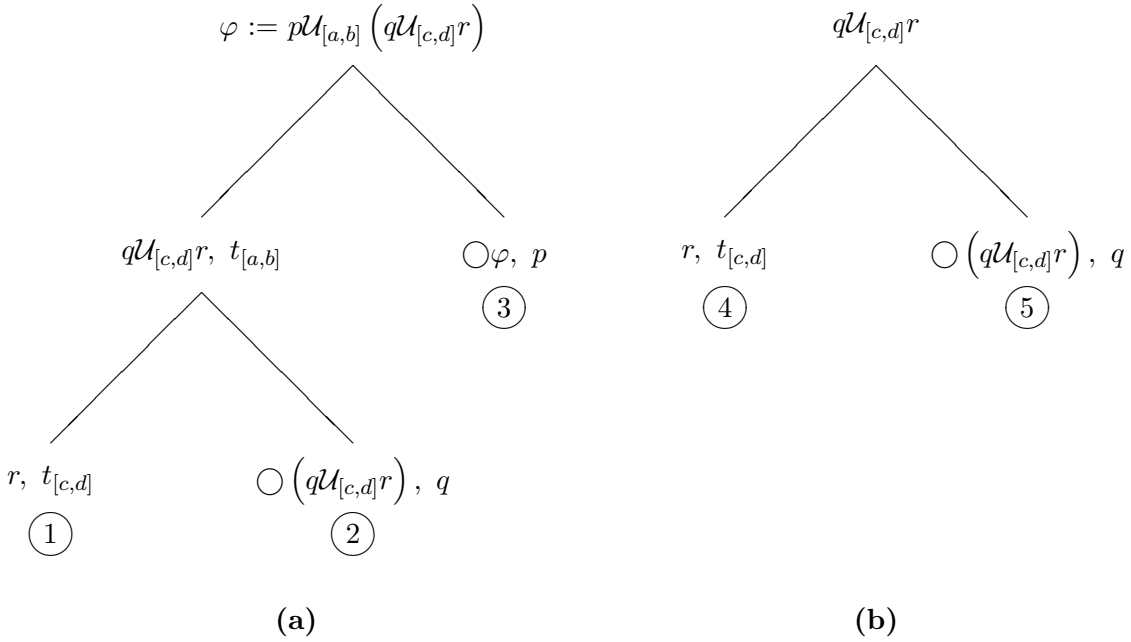


Figure 2: Tree representations of the executions of $cover(\varphi)$ and $cover(q\mathcal{U}_{[c,d]}r)$. The second *cover* is required when finding the successors to particle 2. All particle successors are found within these two *cover*s, so the particle tableau algorithm assigns initial and final particles and terminates.

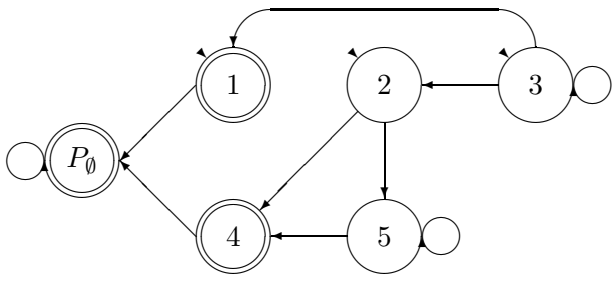


Figure 3: The resulting particle tableau. $P_\emptyset$ is final since it promises nothing; particles 1 and 4 are final because they promise and fulfill r. Particles 1, 2, and 3 are initial because they contain φ.

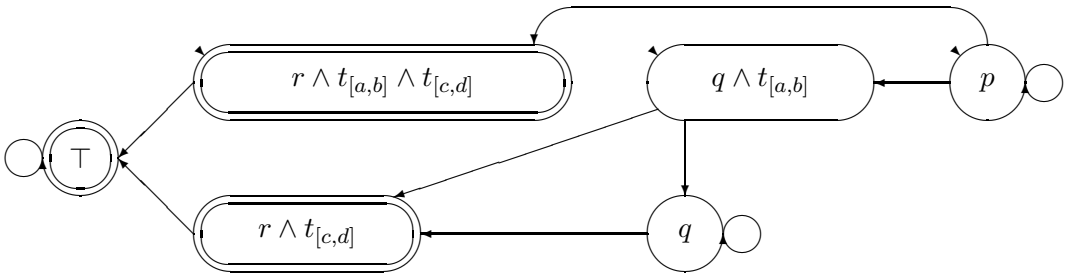


Figure 4: The resulting automaton $\mathcal{A}$. Each node is labeled with the state (non-temporal) formulas from its corresponding particle. Initial and goal nodes are also assigned according to the properties of the corresponding particles.

entry for each automaton node. Initially, only the entries corresponding to initial nodes modeled by the initial conditions of a planning problem are marked with a 1. At each transition from a planning node to the next-state planning node, the vector entries corresponding to successors in the automaton that are modeled by the next-state planning node are marked with a 1; all other entries are set to 0. If no entry is marked with a 1, the new planning node is pruned. Further, for action guards, next-state planning nodes are only generated using actions that are either not guarded or whose vector indicates that the automaton is in an accepting state.

In a backward-chaining planner, swapping the initial and final automaton goals and reversing the directed edges produces an automaton accepting the opposite state sequences of the original automaton. Then applying a similar technique as described for forward-chaining planners allows temporal knowledge to be specified in this context, as well. For action guards, however, tracking begins with the taking of an action; then the rest of the sequence, which precedes the taking of the action in planning time, must induce an accepting path in the associated automaton.

We argue that this approach yields a more efficient planner because all operations involving temporal logic are completed in the preprocessing conversion of formulas to automata. Checking if a planning state models a node label simply involves checking if certain literals are true. Moreover, using automata easily extends temporal logic to a backward-chaining planner. The rest of this section details how MITL may be used in a constraint-based planner.

3.3 Encoding the Automaton

3.3.1 SAT

Recall that the goal of our compilation method is to generate SAT or CSP encodings of temporal constraints that are as small as possible. Intuitively, our automata not only succinctly capture the behavior of finite sequences of world-states, but also provide more accessible structure than the original MITL formulas. Then the algorithm for compiling an automaton into a constraint problem should use this structure to produce a better encoding than the one presented in §3.1.

For each node of $\mathcal{A}$ and time-step $t \in [1, k]$, the algorithm introduces variable $\ell_{n,t}$, which indicates whether an induced path can be at node n at time t. We break the encoding into three macros parameterized by time-step t, where each macro is divided into initial and main cases. Instantiating these macros at some time-step instantiates the time-parameterized propositions of the form $t_{[a,b]}$ to **true** or **false**. We use the sets *init*, *has_pred*, *goal*, and *pred*(n), which contain

the initial nodes, the nodes containing predecessors, the goal nodes, and the predecessors of n in $\mathcal{A}$, respectively.

First, macro $\mathbf{L}_t$ encodes when $\ell_{n,t}$ is true:

$$\mathbf{L}_t \begin{cases} \left(\bigwedge_{n \in init} \ell_{n,1} \;\leftrightarrow\; \mu(n)\right) \;\wedge\; \left(\bigwedge_{n \notin init} \neg \ell_{n,1}\right) & t = 1 \\ \left(\bigwedge_{n \in has\_pred} \ell_{n,t} \;\leftrightarrow\; \mu(n) \wedge \left(\bigvee_{m \in pred(n)} \ell_{m,t-1}\right)\right) \;\wedge\; \left(\bigwedge_{n \notin has\_pred} \neg \ell_{n,t}\right) & t > 1 \end{cases} \tag{7}$$

$\mathbf{L}_t$ defines the new variables at level t. At the initial level, a new variable holds if and only if its associated node's labeling is satisfied and the node is initial; at all other levels, a variable holds if and only if the labeling holds, the node has predecessors, and a variable associated with one of its predecessors held at the previous step. This definition *semantically* captures the sequential nature of the encoding: a plan is in a node at time t if and only if it satisfies the label and was in one of its predecessors previously; but to have been in a predecessor node, the same relationship had to hold then. Carrying this logic back to the initial state, we see that satisfying $\ell_{n,t}$ means that a sequence through $\mathcal{A}$ is satisfied. Yet, importantly, this behavior is captured *syntactically* in a macro referring only to two levels of the plan—which results in a polynomially-sized (in the number of automaton nodes) encoding (see §3.4).

Next, we define a macro $\mathbf{A}_t$ that says that some node label must be satisfied at a given level. This macro is useful for global constraints, where requiring that $\mathbf{A}_t$ hold in the final world-state and using $\mathbf{L}_t$ at every time-step ensures that a plan induces a path through the original automaton. The macro is defined as follows:

$$\mathbf{A}_t \begin{cases} \bigvee_{n \in init} \ell_{n,1} & t = 1 \\ \bigvee_{n \in has\_pred} \ell_{n,t} & t > 1 \end{cases} \tag{8}$$

In our nondeterministic automata, where a node may not have a successor for every possible next-state, requiring that a sequence of states induce *some* path through the automaton is nontrivial. Hence, this macro makes a nontrivial demand on a plan.

Finally, $\mathbf{G}_t$ is a variation on $\mathbf{A}_t$ that is useful for action guards. It says that either some induced path ends in an accepting node, or the guarded action is not taken:

$$\mathbf{G}_t \begin{cases} \left(\bigvee_{n \in init \cap goal} \ell_{n,1}\right) \vee \neg A_1 & t = 1 \\ \left(\bigvee_{n \in has\_pred \cap goal} \ell_{n,t}\right) \vee \neg A_t & t > 1 \end{cases} \tag{9}$$

Using these macros to form the final propositional-logic encoding of a temporally-guarded action, we have

$$\bigwedge_{t \in [1,k]} \left(\mathbf{L}[\wedge \mathbf{G}]\right)_t , \tag{10}$$

where the bracketed expression is included at level t if and only if A_t appears in the planning graph at level t. $\mathbf{A}_t$ is not needed here since $\mathbf{G}_t$ with A_t holding is only satisfied if the world-state sequence induces a path through the automaton (because of $\mathbf{L}_t$) ending in a goal node. Moreover, if the behavior of the plan exits the automaton at some time ℓ, $\mathbf{G}_t$ with A_t holding will never be satisfied, so A cannot be taken for $t \geq \ell$. This is an intuitive argument; we present a formal proof of correctness below.

Global constraints do not have goal states; however, they constrain the behavior of the plan to follow a path through the automaton but not necessarily reach any particular state before the

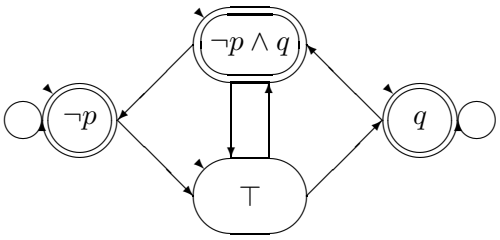


Figure 5: Automaton corresponding to global temporal constraint $\mathbf{G}(p \rightarrow \bigcirc q)$. Huang's control framework handles this type of constraint with a syntax for specifying a precondition and control effect; `tk2sat` handles this constraint as one particular form of global constraint within a general temporal framework.

plan terminates. For example, Huang's BLACKBOX control framework handles global constraints of the form $\mathbf{G}(p \rightarrow \bigcirc q)$ for precondition p and effect q. The corresponding automaton is shown in Fig. 5. One can see that the automaton enforces the constraint that q ought to hold in a state if p held in the preceding time-step; yet, it does not require that the final state of the plan satisfy any particular propositional constraint. Temporally-extended goals [1], which one can view as global constraints, would be implemented with goal conditions; however, in the usual case, the encoding is

$$\bigwedge_{t \in [1,k]} (\mathbf{L}[\wedge \mathbf{A}])_t \tag{11}$$

where the bracketed expression is only included at level k, the final level of the plan. This use of $\mathbf{A}_k$ constrains the behavior of the plan to stay within the automaton, since $\mathbf{A}_k$ holds if and only if the final state satisfies a node labeling in $\mathcal{A}$ and there exists a path through the automaton leading to that node.

The macros convert to CNF with little increase in size:

- $\mathbf{L}_t$ becomes

$$\begin{array}{l} \{\neg \ell_{n,t}, \mu(n)\} \\ \{\neg \ell_{n,t}, \ell_{m_1,t-1}, \ldots, \ell_{m_j,t-1}\} \\ \{\ell_{n,t}, \neg \mu(n), \neg \ell_{m_1,t-1}\} \\ \ldots \\ \{\ell_{n,t}, \neg \mu(n), \neg \ell_{m_j,t-1}\} \end{array} \tag{12}$$

  for each node n with predecessors $m_1, \ldots, m_j$, and

$$\{\neg \ell_{n,t}\} \tag{13}$$

  for each node n without predecessors. For $t = 1$, we have

$$\begin{array}{l} \{\neg \ell_{n,t}, \mu(n)\} \\ \{\ell_{n,t}, \neg \mu(n)\} \end{array} \tag{14}$$

  for each initial node n, and

$$\{\neg \ell_{n,t}\} \tag{15}$$

  for each non-initial node n.

- $\mathbf{A}_t$ becomes
$$\{\ell_{n_1,t}, \ldots, \ell_{n_j,t}\} \tag{16}$$
for nodes $n_i \in \mathit{has\_pred}$ (or init if $t = 1$).

- $\mathbf{G}_t$ becomes
$$\{\ell_{n_1,t}, \ldots, \ell_{n_j,t}, \neg A_t\} \tag{17}$$
for nodes $n_i \in \mathit{has\_pred} \cap \mathit{goal}$ (or $\mathit{init} \cap \mathit{goal}$ if $t = 1$).

We now prove a simple theorem stating the correctness of our SAT encoding. Given the equivalence of the automaton and its source MITL assertion, correctness implies that the SAT encoding captures the constraints specified by the user.

Theorem 3.1 *The assertion*
$$\bigwedge_{t\in[1,\ell]} (\boldsymbol{L}[\wedge \boldsymbol{G}])_t\,, \tag{18}$$
for $\ell \leq k$ and action A taken at time-step ℓ, is satisfied by the sequence of world-states $s_1, s_2, \ldots, s_\ell$ if and only if the corresponding automaton $\mathcal{A}$ accepts the sequence, i.e.*, the sequence induces a path*
$$n_1, n_2, \ldots, n_\ell \text{ such that } \forall i \in [1,\ell].\mu(n_i) \models s_i, \tag{19}$$
for initial node n_1 and final node n_ℓ.

Proof We prove the "if" direction first. Assume the sequence is accepted by $\mathcal{A}$ as described; we prove that (18) holds. Since the sequence induces the path in (19), the $\mu(n_i)$ clause in
$$\ell_{n_i,i} \;\leftrightarrow\; \mu(n_i) \wedge \left(\bigvee_{m\in \mathit{pred}(n_i)} \ell_{m,t-1} \right)$$
from $\mathbf{L}_i$ is satisfied for each i; and since n_{i-1} is a predecessor of n_i for each $i \in [2,\ell]$, the second conjunct on the right side is also satisfied. For $i = 1$, the second conjunct does not exist. Further, n_1 is initial and n_i for $i > 1$ have predecessors, so the members of the second set of conjuncts in $\mathbf{L}_t$ are satisfied. Hence, $\ell_{n_i,i}$ holds for each i. n_ℓ is final, so $\mathbf{G}_\ell$ holds whether or not A_ℓ is taken. Hence, (18) follows.

For the other direction, assume (18) is satisfied; we show that there is an induced path as described in (19) through $\mathcal{A}$. By $\mathbf{G}_\ell$, at least one $\ell_{n_\ell,\ell}$ holds, so by $\mathbf{L}_\ell$, $\mu(n_\ell)$ and $\bigvee_{m\in \mathit{pred}(n_\ell,\ell)} \ell_{m,\ell-1}$ also hold. Tracing backwards and using the second conjunct at each level, we choose the positive $\ell_{n_i,i}$ such that the n_i form a path through $\mathcal{A}$. By the first clause, $\mu(n_i) \models s_i$ at each i. $\mathbf{G}_\ell$ guarantees that since A_ℓ, then n_ℓ is a goal node, while the second set of conjuncts in $\mathbf{L}_i$ guarantees that n_1 is initial and n_i for $i > 1$ have predecessors (so $\ell_{n_i,i}$ for initial node n_i without predecessors cannot be satisfied trivially, which would allow a late entry into the automaton). Thus, the constructed path satisfies (19), completing the proof. ∎

It is easy to see that the global constraint encoding (11) matches the specified temporal semantics since $\mathbf{A}_t$ is simply a different "goal" that must be satisfied without reference to an action. Acceptance is now just the requirement that $\mu(n_i) \models s_i$ for all time-steps i.

3.3.2 CSP

Using Kambhampati's [8] CSP encoding of the planning graph, the encoding discussed above is almost identical in the CSP setting. The changes are the following:

- variables $\ell_{n,t}$ are introduced as usual, but they have domains $\{\top, \bot\}$;
- literals p and $\neg p$ are represented as $p \neq \bot$ and $p = \bot$, respectively;
- $\neg A_t$ is encoded as

$$\bigwedge_{p \in effect(A_t)} p \neq A_t.$$

3.4 Analysis

Analyzing the CNF encoding of (10) reveals a number of clauses and new variables polynomial in the size of the automaton:

Theorem 3.2 *The numbers of clauses and new variables in the* SAT *formulation (10) are polynomial in the number of nodes in the source automaton and the length of the plan.*

Proof We analyze the contribution of each CNF expression in (12) through (17), skipping the special case $t = 1$ since the CNF encoding is smaller. One variable $\ell_{n,t}$ is introduced for each node at each level, resulting in

$$|nodes| \times k \text{ new variables.} \tag{20}$$

Counting clauses, we have

- for each node with predecessors at each level t, $|conjuncts(\mu(n))|$ (the number of conjuncts in $\mu(n)$—all disjuncts were removed when $\mathcal{A}$ was generated) clauses of the type on the first line of (12); one clause of the type on the second line of (12); and $|pred(n)|$ clauses of the form of the remaining clauses in (12);
- for each node without predecessors at each level t, one clause of the type in (13);
- one clause from (16);
- one clause from (17).

Assuming all nodes have predecessors, and recalling that the formulations above use either $\mathbf{A}_t$ or $\mathbf{G}_t$ but not both, the final result is

$$\left[\left(\sum_{n \in nodes} (|conjuncts(\mu(n))| + |pred(n)| + 1)\right) + 1\right] \times k \text{ clauses,} \tag{21}$$

for *nodes* in $\mathcal{A}$ and a k-level plan. Since each node has all nodes as predecessors in the worst case, the method introduces $O(|nodes|k)$ new variables and $O(|nodes|^2 k)$ clauses for each guarded action, a result polynomial in the size of the automaton, as desired. Alternately, we may express this result as

$$\left[\left(\sum_{n \in nodes} |conjuncts(\mu(n))|\right) + |edges| + n + 1\right] \times k \text{ clauses,} \tag{22}$$

or $O((|nodes| + |edges|)k)$ clauses. ∎

While the size of the automaton is theoretically exponentially upper-bounded in the number of temporal operators, in practice, the particle tableau algorithm produces small automata; hence, we may expect relatively small encodings. The encoding for global constraints is smaller since $\mathbf{A}_t$ is instantiated only at the last level, however the quadratic number of clauses and linear number of new variables does not change. Considering our example from §3.1.2, this encoding produces 160 clauses and introduces 40 new variables (compare to 1001 and 100, respectively). In both cases, restricting $[a, b]$ or $[c, d]$ reduces the number of clauses, but the analysis is unenlightening.

3.5 Working Within the Planning Framework

The previous sections presented and analyzed a framework for compiling MITL guards into a SAT expression. Clearly, this is the main work in adapting temporal domain knowledge for a constraint-based planner; however, a few more points are worth noting before moving onto implementation details.

First, what are literals in the planning domain? So far, we have used generic literals like p, q, or r, but clearly this is not interesting for planning. For a given planning problem, the literals are the instantiated domain predicates. For the `logistics` domain [9] with predicates `(OBJ ?obj)` and `(in-city ?obj ?city)`—the first providing type information, the second providing more complex information about the changing world—possible instantiations include `(OBJ package1)` and `(in-city airplane1 bos)`, respectively, resulting in literals `OBJ_package1` and `in-city_airplane1_bos`. These literals, however, are too specific and thus not useful to the user; hence, the user specifies the guards as *templates*, with variables of the same form as the predicates. For example, for the action

```
(:action LOAD-TRUCK
 :parameters (?obj ?truck ?loc)
 :precondition (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
                    (at ?truck ?loc) (at ?obj ?loc))
 :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))
```

a corresponding (vacuous) user-provided guard template might be

```
(:action LOAD-TRUCK
 :guard (until (not (at ?truck ?loc)) (at ?truck ?loc)))
```

In the notation used so far, the guard is

$$\neg\texttt{(at ?truck ?loc)}\mathcal{U}\texttt{(at ?truck ?loc)};$$

when instantiated for a particular action, say `LOAD-TRUCK` with parameters `(package1 truck1 loc1)`, the one literal is `at_truck1_loc1`. In BLACKBOX, this literal is still not completely instantiated, as it is parameterized by the time-step. As suggested earlier with the notation $\mathbf{L}_t$, the literal will be fully instantiated in the SAT expression.

Next, in addition to specifying temporal knowledge, the user may wish to use quantifiers in expressing domain knowledge. In a finite domain, universal quantification corresponds to conjunction, while existential quantification corresponds to disjunction. However, we may make these operators

more than just a notational convenience by adding an extra predicate that can be evaluated at SAT compile-time, resulting in *bounded quantifiers* [1] that prune the feasible domain of objects down to those satisfying the predicate. Notationally, we say

$$\exists x : \alpha(x).\beta(x)$$

for propositional formula α and temporal assertion β. β may contain further quantifiers, while both α and β may refer to other variables assigned a value by an outer scope. $\alpha(x)$ must be evaluatable at compile-time—*i.e.* after the problem has been read, but before solving the problem. Usually, α provides typing information, but it may in general refer to static world-facts. The user may also specify

$$\forall x : \alpha(x).\beta(x),$$

although this can easily be implemented as

$$\neg\exists x : \alpha(x).\neg\beta(x).$$

In addition to these logical quantifiers, the user must also provide a scope for global constraints. While semantically conjunctive—the scope says that *for all* parameter assignments satisfying a scoping constraint, the plan should meet the associated instantiated global temporal constraint—the division between the scoping constraint and the temporal constraint necessitates a third quantifier,

$$\texttt{scope}\ x : \alpha(x).\beta(x),$$

that returns a set of parameters satisfying α and β, where both are now propositional formulas. For example, in the global constraint (adapted from [7])

```
(:scope (scope (?trk) (truck ?trk)
          (scope (?obj) (obj ?obj)
            (scope (?loc) (location ?loc)
              (and (in_wrong_city ?obj ?loc) (not (AIRPORT ?loc))))))
 :constraint (henceforth (or (not (and (at ?trk ?loc) (at ?obj ?loc)))
               (next (at ?trk ?loc)))))
```

the `scope` clause returns a list of parameter instantiations to use for the temporal constraint. In this sense, `scope` is a quantifier.

Finally, the user may wish to take advantage of the other set of information available at compile-time: goals [1]. To allow the user to incorporate this information in constraints, we introduce the predefined predicate `(goal` φ`)` for propositional formula φ, which evaluates to **true** if and only if φ is a goal of the problem. In the `logistics` domain, for example, the assertion `(goal (at ?obj ?loc))` evaluates to **true** when instantiated if the instantiated predicate is a goal. Paralleling the `goal` case, the user may also specify that a predicate is a static world-fact with the predefined predicate `(init` φ`)` for propositional formula φ. This clause indicates that the value of φ may be evaluated from only its initial value since it necessarily remains the same throughout the plan.

4 Implementation and Empirical Results

4.1 Implementation

We implemented our constraint-based framework, `tk2sat`, for temporally-specified domain knowledge in the BLACKBOX [9] planning system. BLACKBOX uses GRAPHPLAN [3] to build the planning

graph for a specified problem, then compiles it into a SAT instance that is given to one of several SAT solvers. With the additional knowledge framework, the user specifies an additional input file, the `tk2sat` control file, which includes user-defined predicates, action guards, and global constraints; then, the system generates additional SAT clauses guarding actions appearing within the generated planning graph and constraining the behavior of objects in the scopes of global constraints. §4.1.1 discusses the control file; §4.1.2 outlines the implementation.

4.1.1 The `tk2sat` Control File

Users specify domain-specific information with the control file. The syntax is as follows:

- **Header:** specifies the name of the module and the domain:

```
(define (tkcontrol <module>)
        (:domain <domain>)
 <defpredicates, guards, and global constraints>
)
```

- **User-defined predicates:** allows users to build predicates using existing predicates:

```
(:defpredicate <predicate>
 :parameters (<var1> <var2> ...)
 :body (<\mitl\ formula>))
```

- **Guards:** specifies an action and its associate guard; the guard may refer only to the parameters of the action:

```
(:action <action>
 :guard (<\mitl\ formula>))
```

- **Global constraints:** specifies a temporal constraint and a scope to which to apply the constraint:

```
(:scope (<scope clause>)
 :constraint (<\mitl\ formula>))
```

MITL formulas are specified using the following operators:

- $\varphi_1 \wedge \varphi_2$: `(and` φ_1 φ_2`)`.
- $\varphi_1 \vee \varphi_2$: `(or` φ_1 φ_2`)`.
- $\neg\varphi$: `(not` φ`)`.
- $\varphi_1 \mathcal{U} \varphi_2$: `(until` φ_1 φ_2`)`.
- $\varphi_1 \mathcal{W} \varphi_2$: `(waitfor` φ_1 φ_2`)`.

- **F**φ: `(eventually` φ`)`.
- **G**φ: `(henceforth` φ`)`.
- $\bigcirc\varphi$: `(next` φ`)`.
- **goals**: `(goal p)`, which evaluates to **true** if and only if the propositional formula `p` is a goal of the problem.
- **static facts**: `(init p)`, which evaluates to **true** if and only if the propositional formula `p` is true in the initial state of the problem—the system does not verify that its value does not change.
- $\exists p : \alpha(p).\beta(p)$: `(exists (p) (`$\alpha$`(p)) (`$\beta$`(p)))`, where `p` is an object in the problem and α and β are propositional and MITL assertions possibly parameterized by p, respectively. α bounds the domain for the existential operator; it must be evaluatable at SAT compile-time, so it may only use propositions true of the initial state (*e.g.*, types, static world-facts, and `goal` statements). If the outer operator of the α clause is not `goal`, it is implicitly assumed to be an `init`. Evaluates to a disjunction of instantiated β`(p)`.
- $\forall p : \alpha(p).\beta(p)$: `(forall (p) (`$\alpha$`(p)) (`$\beta$`(p)))`, where the parameter constraints are the same as for `exists`. Evaluates to a conjunction of instantiated β`(p)`s.
- **end-constraint**: `(end` φ`)`, for MITL assertion φ; it is only useful in the context of a global constraint. Makes **true**-labeled goal nodes non-goals, forcing the interesting behavior of the automaton to constrain the part of the sequence ending at the action decision.
- `scope` $p : \alpha(p).\beta(p)$: `(scope (p) (`α`(p)) (`β`(p)))`, where α and β are both propositional and MITL assertions possibly parameterized by p. Both must also be evaluatable at SAT compile-time. This operator cannot be used in an arbitrary formula; it may only appear within the `scope` environment. Evaluates to a list of mappings from the `scope` parameter to a plan object; multiple embedded `scope`s evaluate to a list of mappings from the `scope` parameters to plan objects.

Every operator may take MITL sub-formulas as arguments, except as specified. Propositional formulas may contain the operators `and`, `or`, `not`, `goal`, `init`, `exists`, and `forall`. The current syntax does not support interval specifications, as these were not found useful within the BLACKBOX environment, although the core temporal framework does support these specifications.

An Example. The following control file specifies two action guards and a global constraint for the `logistics` domain. The `UNLOAD-TRUCK` action guard, part of the `LOAD-TRUCK` action guard, and the user-defined predicate `in_wrong_city` are adapted from [7]. We show empirical results for variations on this control file in §4; this particular control file is only meant to give an example of how to use the `tk2sat` interface.

```
(define (tkcontrol logcontrol)
        (:domain logistics)

; True iff ?obj is in the wrong city when it is at ?loc.  The init
```

```
; operator is used as an optimization since a location cannot be moved
; from a city; hence, the predicate in-city... is evaluated at
; compile-time.
(:defpredicate in_wrong_city
 :parameters (?obj ?loc)
 :body (exists (?goal_loc) (goal (at ?obj ?goal_loc))
         (exists (?city) (in-city ?loc ?city)
           (not (init (in-city ?goal_loc ?city))))))

; Specifies that the truck should not be loaded if (1) ?obj should be
; at ?loc to satisfy a goal, (2) ?loc is an AIRPORT and ?obj is at
; ?loc and in the wrong city, or (3) ?obj has been in ?truck before.
(:action LOAD-TRUCK
 :guard (and
          (eventually (henceforth (not
            (or (goal (at ?obj ?loc))
                (and (in_wrong_city ?obj ?loc) (AIRPORT ?loc))))))
          (henceforth (not (in ?obj ?truck)))))

; Specifies that the truck should not be unloaded if (1) the object is in
; the wrong city and not at an airport, or (2) in the right city, but not
; at the right location.
(:action UNLOAD-TRUCK
 :guard (eventually (henceforth (not
          (or (and (in_wrong_city ?obj ?loc) (not (AIRPORT ?loc)))
              (and (not (in_wrong_city ?obj ?loc))
                   (not (goal (at ?obj ?loc)))))))))

; True iff ?obj is in ?city.
(:defpredicate obj-in-city
 :parameters (?obj ?city)
 :body (exists (?loc) (LOCATION ?loc)
         (and (at ?obj ?loc) (init (in-city ?loc ?city)))))

; Restricts global behavior so that an object's path in the plan follows:
;  1. in a location in a city
;  2. in a truck
;  3. in another location in the same city
;  4. in a plane
;  5. in a location in a new city
;  6. in a truck
;  7. in another location in the same city
; where any step may be skipped.
(:scope (scope (?obj) (OBJ ?obj)
          (scope (?city-1) (CITY ?city-1)
            (scope (?city-2) (CITY ?city-2)
              (and (init (obj-in-city ?obj ?city-1))
                   (goal (obj-in-city ?obj ?city-2))))))
 :constraint (until (obj-in-city ?obj ?city-1)
              (until (exists (?truck-1) (TRUCK ?truck-1) (in ?obj ?truck-1))
              (until (obj-in-city ?obj ?city-1)
```

```
            (until (exists (?plane) (AIRPLANE ?plane) (in ?obj ?plane))
            (until (obj-in-city ?obj ?city-2)
            (until (exists (?truck-2) (TRUCK ?truck-2) (in ?obj ?truck-2))
                   (henceforth (obj-in-city ?obj ?city-2)))))))))
)
```

Because the formula $\mathbf{FG}\varphi$ for propositional formula φ only requires φ to hold at the end of the finite sequence, the formula is semantically non-temporal. Hence, the action guard for `UNLOAD-TRUCK` is non-temporal; see §4.1.2 for how `tk2sat` optimizes the compiled SAT representation in such a case. However, the `(henceforth (not (in ?obj ?truck)))` clause of `LOAD-TRUCK`'s action guard is temporal because it refers to the entire history before the action is taken. The global constraint is also temporal. Note the use of `init` and `goal` to restrict the domains of `?obj`, `?city-1`, and `?city-2`: `?obj` should initially be in `?city-1`, and it should finally be in `?city-2`. In the `exists` clauses, the predicate `(TRUCK ?truck-1)` can be evaluated at compile-time, resulting in a smaller disjunction of instantiations of `(in ?obj ?truck-1)`.

4.1.2 `tk2sat`: Under the Hood

`tk2sat` is implemented in C++. The logic framework defines classes for a recursively structured representation of MITL. The entire class hierarchy is as follows:

`functor` Defines basic interface and behavior of MITL classes, including setting and returning arguments, equality checks, syntactic simplification, conversion to CNF, and pushing in of negations. Syntactic simplification consists of handling cases in which an argument is a `t` or `f`.

- `oftr (operator)` Base class of MITL operators.
  - `bftr (Boolean operator)` Base class of Boolean operators. Implements the state property. Each subclass provides a specialized equality check, simplification, pushing in of negations, conversion to CNF, and copying.
    - `and`
    - `or`
    - `not`
    - `t` (**true**)
    - `f` (**false**)
  - `tftr (temporal operator)` Base class of temporal operators. Defines intervals and functionality for finding the closure of an MITL assertion. Each subclass provides a specialized equality check, simplification, pushing in of negations, conversion to CNF, copying, and returning of κ sets, as defined in §2.2.3.
    - `until`
    - `waitfor`
    - `eventually`
    - `henceforth`
    - `next`

`fset (functor set)` Defines specialized set functions for working with MITL assertions. `fset` itself is a `functor`, so it can hold instances of `fset`s as elements.

`quantifier` Base class of quantifiers.

`exists` Bounded existential operator; takes three arguments: the planning domain variable, a domain-restriction propositional clause, and a body MITL formula. The domain-restriction clause is evaluated at SAT compile-time to determine the objects in the planning domain to which to apply the body. The result of evaluation is a disjunction of the body assertion instantiated with objects satisfying the domain-restriction clause. Tightly coupled with the BLACKBOX data structures.

`forall` Bounded universal operator; results in conjunction. Tightly coupled with the BLACKBOX data structures.

`scope` Bounded scoping operator; takes the same arguments as `exists` and `forall`, but evaluates to a list of mappings from its parameter to a plan object. Multiple embedded `scope`s return a list of mappings from the combined parameters to plan objects. Tightly coupled with the BLACKBOX data structures.

`tcons` (time-step constraint) Time-step constraint; evaluates to **true** or **false** at SAT compile-time when its parent operator is instantiated with a time. These constraints are used when a temporal MITL operator is given a non-$[1, k]$ interval, for the length of the plan k; however, within the BLACKBOX framework, this constraint is not used.

`var` (variable) Defines a MITL literal. For example, if the domain defines the predicate `(at ?obj ?loc)`, the resulting variable has the name `at_?obj_?loc`, as in the BLACKBOX framework.

`goal` Special operator. Evaluates to **true** or **false** at SAT compile-time according to whether its argument is a goal in the particular planning problem. Also implements `init` behavior.

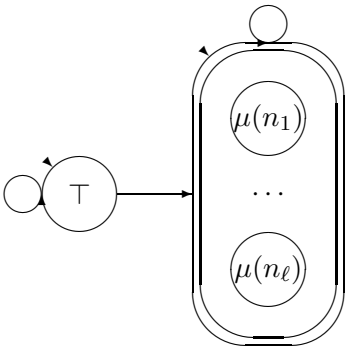
`paut` (propositional automaton) Defines a propositionally-labeled automaton; constructed by `tk2sat`. Returns MITL representation of itself as instantiations of the macros defined in §3.3.1, although the time-step is left uninstantiated.

`tk2sat` Wrapper class for particle-tableau algorithm. Given an MITL assertion, returns the automaton representation as a `paut`.

`tk2wff` Interface between `tk2sat` and BLACKBOX. Defines the one global function called by BLACKBOX. Searches the planning graph for instantiations of guarded actions, and adds the guard clauses. Tightly coupled with the BLACKBOX data structures.

Except for the quantifiers and `tk2wff`, which must have complete knowledge of the problem and domain, the framework is generally independent of the BLACKBOX implementation. Hence, moving most of the code to a different application is feasible.

Optimizing for Non-Temporal Guards. Automata of the form



are semantically non-temporal. For clarity, all nodes are initial, and nodes n_i are fully connected and final. The assertion

$$\bigvee_{i \in [1,\ell]} \mu(n_i)$$

must only hold at the time-step of the action being guarded; hence, this case matches the control framework of [7]. This automaton can be described by (and generated by) the MITL assertion

$$\mathbf{FG} \bigvee_{i \in [1,\ell]} \mu(n_i);$$

see §4.1.1 for an example. To avoid the SAT clause overhead of the temporal framework, this case is recognized and handled separately. Specifically, if action A appears at time-step t, the clause

$$\{\mu(n_1), \ldots, \mu(n_\ell), \neg A_t\}$$

is appended; otherwise, `tk2sat` does not generate any new clauses.

4.1.3 Bringing it All Together

Now that we have described the control file and the overall BLACKBOX temporal knowledge framework, we may present the high-level control flow. First, `tk2sat` reads the user-specified control file, building a template guard consisting of parameters and a template automaton for each user-specified guard, and a scope expression and a template automaton for each global constraint. After BLACKBOX encodes the planning graph as SAT, `tk2sat` searches the graph for instantiated actions. For each instantiated action, the corresponding guard template is recalled and instantiated in the conversion from automaton to MITL; however, at this stage, the MITL assertions (corresponding to $\mathbf{L}_t$ and $\mathbf{G}_t$) are still parameterized by time-step. With the parameters instantiated, bounded quantifiers are evaluated and simplified to disjunctions or conjunctions.

Next, `tk2sat` queries the guard as to whether the guard is semantically temporal. If not, the optimized encoding discussed in §4.1.2 is used. Otherwise, $\mathbf{L}_t$ is instantiated at each level in the planning graph (here, finally, converted to the GRAPHPLAN-chosen variable ID), while $\mathbf{G}_t$ is instantiated at levels at which the action appears. During this time-step instantiation, formulas are syntactically simplified in case expressions become trivially **true** or **false**. For example, $p \vee \mathbf{true}$ becomes **true**, while $p \wedge \mathbf{true}$ becomes p. If a literal p does not appear in a level (by the closed-world assumption, this implies that the literal does not hold), then p simplifies to **false** and $\neg p$ to **true**. CNF clauses containing **true** are not added to the SAT encoding; **false** members of clauses are removed. This completes the SAT encoding of the guard.

Finally, `tk2sat` compiles all global constraints. For a given constraint, `tk2sat` first evaluates the scope clause, which returns a list of tuples of parameter instantiations. For each tuple, it instantiates and compiles the associated automaton as in the case of guards, except that it leaves out the goal clause, using $\mathbf{A}_k$ at the final level k instead. This completes the SAT encoding of the global constraint.

4.2 Results

We ran BLACKBOX with the `tk2sat` framework on a Linux-based 1GHz dual-processor Pentium 4 with 2GB RAM. For testing, we chose to use only the CHAFF SAT-solver [14] because of its superior performance compared to BLACKBOX's other solvers.

4.2.1 Domain-Specific Information for `Logistics`

The `logistics` domain [9] models a world of locations in cities, objects, and trucks and airplanes that move objects between locations. It is formally defined as follows:

```
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
               (AIRPLANE ?airplane) (CITY ?city) (AIRPORT ?airport)
               (at ?obj ?loc) (in ?obj ?obj) (in-city ?obj ?city))

(:action LOAD-TRUCK
  :parameters (?obj ?truck ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
                     (at ?truck ?loc) (at ?obj ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action LOAD-AIRPLANE
  :parameters (?obj ?airplane ?loc)
  :precondition (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
                     (at ?obj ?loc) (at ?airplane ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?airplane)))

(:action UNLOAD-TRUCK
  :parameters (?obj ?truck ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
                     (at ?truck ?loc) (in ?obj ?truck))
  :effect (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action UNLOAD-AIRPLANE
  :parameters (?obj ?airplane ?loc)
  :precondition (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
                     (in ?obj ?airplane) (at ?airplane ?loc))
  :effect (and (not (in ?obj ?airplane)) (at ?obj ?loc)))

(:action DRIVE-TRUCK
  :parameters (?truck ?loc-from ?loc-to ?city)
  :precondition (and (TRUCK ?truck) (LOCATION ?loc-from)
```

```
                    (LOCATION ?loc-to) (CITY ?city)
                    (at ?truck ?loc-from) (in-city ?loc-from ?city)
                    (in-city ?loc-to ?city))
  :effect (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action FLY-AIRPLANE
  :parameters (?airplane ?loc-from ?loc-to)
  :precondition (and (AIRPLANE ?airplane) (AIRPORT ?loc-from)
                    (AIRPORT ?loc-to) (at ?airplane ?loc-from))
  :effect (and (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to)))
)
```

Note that action preconditions specify the minimum requirements for the action to be taken; the reader may see immediately cases in which the action can be taken, but should not. We define several action guards and global constraints below to direct the planner away from bad choices.

Huang wrote a control file for their control framework based on Bacchus and Kabanza's example for TLPlan [1]. His controls include both action guards and global constraints. Our main adaptation uses the action guards, which, because of the optimization discussed above, works well in the `tk2sat` framework; however, because we did not optimize for the special form of global constraint $\mathbf{G}(p \rightarrow \bigcirc q)$, their global constraints invariably . Most likely, the benefit of the extra constraints did not overcome the negative consequence of the larger constraint problem. The resulting *non-temporal* control file with only non-temporal action guards follows:

```
(define (tkcontrol logcontrol)
        (:domain logistics)

(:defpredicate in_wrong_city
 :parameters (?obj ?loc)
 :body (exists (?goal_loc) (goal (at ?obj ?goal_loc))
         (exists (?city) (in-city ?loc ?city)
           (not (in-city ?goal_loc ?city)))))

(:action LOAD-TRUCK
 :guard (eventually (henceforth (not
          (or (goal (at ?obj ?loc))
              (and (in_wrong_city ?obj ?loc) (AIRPORT ?loc)))))))

(:action UNLOAD-TRUCK
 :guard (eventually (henceforth (not
          (or (and (in_wrong_city ?obj ?loc) (not (AIRPORT ?loc)))
              (and (not (in_wrong_city ?obj ?loc))
                   (not (goal (at ?obj ?loc)))))))))

(:action LOAD-AIRPLANE
 :guard (eventually (henceforth (in_wrong_city ?obj ?loc))))

(:action UNLOAD-AIRPLANE
 :guard (eventually (henceforth (not (in_wrong_city ?obj ?loc)))))
)
```

`LOAD-TRUCK` and `UNLOAD-TRUCK` are essentially the same as the example guards in §4.1.1. Loading

an airplane only makes sense if the object is not in the right city; unloading it makes sense only when the object is in the right city. The non-temporal global constraints, which we tested for completeness, follow:

```
; don't move a truck if there is an object that needs to be moved
(:scope (scope (?trk) (truck ?trk)
          (scope (?obj) (obj ?obj)
            (scope (?loc) (location ?loc)
              (and (in_wrong_city ?obj ?loc) (not (AIRPORT ?loc))))))
 :constraint (henceforth (or (not (and (at ?trk ?loc) (at ?obj ?loc)))
               (next (at ?trk ?loc)))))

; don't move a truck if there is an object that needs to be unloaded
(:scope (scope (?trk) (truck ?trk)
          (scope (?obj) (obj ?obj)
            (scope (?loc) (AIRPORT ?loc)
              (in_wrong_city ?obj ?loc))))
 :constraint (henceforth (or (not (and (at ?trk ?loc) (in ?obj ?trk)))
               (next (at ?trk ?loc)))))

; don't move a truck if there is an object in the truck and at the goal
(:scope (scope (?trk) (truck ?trk)
          (scope (?obj) (obj ?obj)
            (scope (?loc) (location ?loc)
              (goal (at ?obj ?loc)))))
 :constraint (henceforth (or (not (and (at ?trk ?loc) (in ?obj ?trk)))
               (next (at ?trk ?loc)))))

; don't move an airplane if there is an object that needs to be moved
(:scope (scope (?pln) (AIRPLANE ?pln)
           (scope (?obj) (obj ?obj)
            (scope (?loc) (AIRPORT ?loc)
              (in_wrong_city ?obj ?loc))))
 :constraint (henceforth (or (not (and (at ?obj ?loc) (at ?pln ?loc)))
               (next (at ?pln ?loc)))))

; don't move an airplane if there is an object that needs to be unloaded
(:scope (scope (?pln) (AIRPLANE ?pln)
          (scope (?obj) (obj ?obj)
            (scope (?loc) (AIRPORT ?loc)
              (not (in_wrong_city ?obj ?loc)))))
 :constraint (henceforth (or (not (and (at ?pln ?loc) (in ?obj ?pln)))
               (next (at ?pln ?loc)))))
```

We defined temporal action guards for `LOAD-TRUCK` and `UNLOAD-TRUCK`; both indicate that an object may be loaded only if it was not previously in the truck or airplane. By referring to the entire history before an action, the guards are temporal. The guards may be integrated with other guards, including Huang's.

```
(:action LOAD-TRUCK
 :guard (henceforth (not (in ?obj ?truck))))
```

```
(:action LOAD-AIRPLANE
 :guard (henceforth (not (in ?obj ?airplane))))
```

Finally, the following temporal global constraint guides the overall plan flow. We noted in plans from several sample problems that in all cases, objects are driven to an airport, flown to another city, and driven to a final location, where any of these steps may be skipped. Hence, we encoded that behavior as follows:

```
(define (tkcontrol logcontrol)
        (:domain logistics)

(:defpredicate obj-in-city
 :parameters (?obj ?city)
 :body (exists (?loc) (LOCATION ?loc)
         (and (at ?obj ?loc) (init (in-city ?loc ?city)))))

(:scope (scope (?obj) (OBJ ?obj)
          (scope (?city-1) (CITY ?city-1)
            (scope (?city-2) (CITY ?city-2)
              (and (init (obj-in-city ?obj ?city-1))
                   (goal (obj-in-city ?obj ?city-2))))))
 :constraint (until (obj-in-city ?obj ?city-1)
             (until (exists (?truck-1) (TRUCK ?truck-1) (in ?obj ?truck-1))
             (until (obj-in-city ?obj ?city-1)
             (until (exists (?plane) (AIRPLANE ?plane) (in ?obj ?plane))
             (until (obj-in-city ?obj ?city-2)
             (until (exists (?truck-2) (TRUCK ?truck-2) (in ?obj ?truck-2))
                    (henceforth (obj-in-city ?obj ?city-2)))))))))
)
```

One could specify exactly in which truck `?obj` is by replacing

```
             (exists (?truck-1) (TRUCK ?truck-1) (in ?obj ?truck-1))
```

with

```
             (exists (?truck-1)
                     (and (TRUCK ?truck-1) (obj-in-city ?truck-1 ?city-1))
                     (in ?obj ?truck-1))
```

This change, however, seems to have little effect on the results, perhaps because the clauses are easy to satisfy as originally stated.

4.2.2 Empirical Results

We ran two sets of tests. In the first set, we used combinations of the constraints introduced above to create control files for just `tk2sat`, allowing analysis of the relative value of non-temporal and temporal constraints when compiled into the SAT instance. The data show that temporal information provides valuable constraints. The second set examines performance: we divided the non-temporal

and temporal constraints into separate files, using Huang's control framework for the non-temporal constraints, and `tk2sat` for the temporal ones. Hence, the overall system—BLACKBOX augmented with Huang's control framework and our `tk2sat`—gains a performance boost, relative to BLACKBOX with `tk2sat` alone, of pruning the planning graph using the non-temporal constraints before compiling to SAT. Further, the data show that this combination produces a superior planner to just BLACKBOX with Huang's control system.

We created a set of test problems, each containing fifteen to sixteen packages, eight cities with one truck and two to three locations each, and two to three airplanes. The high ratio of packages and cities to airplanes makes the problems more difficult since all packages must pass through one of the airplanes at some point in the plan. Indeed, the problems proved to be nontrivial: by itself, BLACKBOX took at least ten minutes on all but one tested problem and did not finish several within the allotted one hour. We only applied BLACKBOX without additional controls to the subset of problems that we used for our first test; given the difficulty of the remaining problems, one can assume that it probably would not have found solutions within one hour. Table 1 describes the relevant features of the problems, omitting the number of cities and number of trucks per city, which are a constant eight and one, respectively. We indicate easy, medium, and hard problems based on the results in Table 5, where these categories describe problems generally solved in under 100 seconds, in several hundred seconds, and in over 1000 seconds, respectively.

Problem	**Pkgs**	**Lctns**	**Plns**	**Problem**	**Pkgs**	**Lctns**	**Plns**
prob131.pddl	15	2	2	**prob133.pddl**	16	2	2
prob134.pddl	15	2	2	*prob136.pddl*	15	2	2
prob137.pddl	15	2	2	*prob138.pddl*	15	2	2
prob139.pddl	15	2	2	**prob140.pddl**	15	2	2
prob141.pddl	15	2	3	prob142.pddl	15	2	3
prob143.pddl	16	2	3	prob144.pddl	16	2	2
prob145.pddl	15	3	2	*prob146.pddl*	15	3	2
prob147.pddl	15	2/3	2	**prob148.pddl**	15	2/3	2
prob149.pddl	16	3	2	prob150.pddl	16	3	2
prob151.pddl	16	3	2	*prob153.pddl*	15	2/3	2
prob154.pddl	16	3	2	**prob155.pddl**	16	3	2
prob156.pddl	16	3	2	*prob157.pddl*	16	2	2

Table 1: Description of problems. The **Pkgs** and **Plns** columns list the number of packages and airplanes, respectively. The **Lctns** column lists the number of locations per city; 2/3 indicates that half of the cities contain two locations, while the other half contain three. Plain text indicates a relatively easy problem, italicized text indicates a problem of medium difficulty, and bold text indicates a hard problem.

For the first set of tests, we constructed the `tk2sat` control files described in Table 2. Table 3 displays the running time in seconds for BLACKBOX with `tk2sat`; dashes indicate that a plan was not found within one hour. The results indicate that for this set of problems, using the non-temporal and temporal action guards and temporal global constraint produced the most consistently fast planner: not only did BLACKBOX with `all` finish all of the tests, but it also finished more quickly on the larger problems. For example, on four of the five problems that required over 1000 seconds

for `huang`, `all` finished in less time. As the problems become harder, the disadvantage of creating and solving larger constraint problems is overcome by the advantages of the extra constraints. Further, comparing the performance of the single temporal global constraint to that of the set of non-temporal global constraints reveals that, for our compilation scheme, inherently temporal information can be stronger than non-temporal information. We show below, however, that using Huang's pruning algorithm allows us to take advantage of both types of information.

Control	Description
`none`	no constraints
`huang`	non-temporal action guards
`huangall`	non-temporal action guards and global constraints
`glbl`	temporal global constraint
`action`	temporal action guards
`huang_glbl`	non-temporal action guards and temporal global constraint
`all`	non-temporal action guards, temporal action guards, and temporal global constraint

Table 2: The control files used for the tests in Table 3.

Timing results for `tk2sat` on domain `logistics`							
Problem	**none**	**huang**	**huangall**	**glbl**	**action**	**huang_glbl**	**all**
prob131.pddl	1004	71	405	289	141	110	129
prob134.pddl	1161	790	1552	—	1090	626	849
prob136.pddl	—	1021	1648	—	664	1657	1041
prob137.pddl	207	104	415	385	110	107	114
prob138.pddl	1320	1305	—	—	1948	1080	1061
prob139.pddl	1091	117	424	374	147	105	136
prob140.pddl	—	—	—	—	—	—	1275
prob141.pddl	2029	74	374	381	112	80	110
prob142.pddl	—	125	412	730	184	182	157
prob143.pddl	1587	204	448	825	132	139	163
prob144.pddl	1904	68	493	765	216	170	181
prob145.pddl	749	84	619	358	197	202	261
prob146.pddl	—	1042	2835	3580	2287	2660	991
prob147.pddl	—	2669	—	—	—	—	2015

Table 3: Timing results for the `logistics` domain, where all constraints are compiled as additional clauses, using the `tk2sat` framework alone. All times are in seconds; a dash indicates that the problem was not solved within one hour.

In the second test set, we focused on measuring the speed and robustness of particular combinations of constraints. Since our goal is to make the fastest planner possible, we combined Huang *et al.*'s work with ours so that non-temporal information is manipulated as efficiently as possible. Table 4 describes the control files used for this set of tests. Each test combines a control file for

Huang's system with a control file for `tk2sat`; we also show results for just Huang's system. Table 5 shows the time in seconds required to solve each problem. We can immediately see a few significant trends: first, the value of adding temporal constraints through `tk2sat` is especially apparent on the difficult problems. Second, the two control file pairs that resulted in solving the most problems contain temporal information: one contains all of the constraints, while the other has the non-temporal action guards and temporal global constraint. Further, all pairs containing at least some temporal information solved more problems than just BLACKBOX with Huang's system alone.

Control	Description
Huang's BLACKBOX control system	
`act`	non-temporal action guards
`all`	non-temporal action guards and global constraints
`tk2sat`	
`none`	no constraints
`act`	temporal action guards
`glbl`	temporal global constraint
`all`	temporal action guards and global constraint

Table 4: Control files used for the tests in Table 5 and Table 6.

In an attempt to understand why certain problems are solved more quickly with certain control files, we examined the number of clauses `tk2sat` adds to each final SAT instance. Table 6 shows for each problem and control file pair the number of clauses in the final SAT instance from which the solution was extracted. In general, the temporal action guards increase the number of clauses in the SAT instance by about 2-4%, while the global constraint adds approximately 8-12% more clauses. Given this uniform increase in size, the data does not indicate a correlation between the relative number of extra clauses in a particular problem and whether those temporal constraints result in solving the problem in the allotted time. Further, while trivially measurable characteristics of the problems themselves such as the number of packages, airplanes, and locations indicate the overall difficulty of the problems, there seems to be no obvious correlation between a particular configuration of parameters and the performance of certain constraints. Most likely, it is the subtle play between problem constraints and domain knowledge constraints, as well as the strategy of the SAT solver, that results in the variation in performance; studying and predicting these characteristics is a nontrivial research endeavor in itself. Fortunately, however, certain control files give more consistent overall performance on problems of the type in our test set, so one could choose a strategy based entirely on this observation. For example, using all the constraints for one hour, followed by just the action constraints for the next, could be one fruitful strategy. Alternately, one might run them in parallel.

5 Conclusion

Given the time-dependent sequential nature of the planning problem, using temporal knowledge is a natural next step in specifying and incorporating domain-specific information. Bacchus and Kabanza showed with TLPLAN that temporally specified constraints can allow even a forward-

Timing results for Huang's system and `tk2sat` on domain `logistics`								
Problem	**none act**	**none all**	**act act**	**act all**	**all act**	**all all**	**glbl act**	**glbl all**
prob131.pddl	38	51	65	31	52	50	73	55
prob133.pddl	—	—	—	—	—	3168	1048	—
prob134.pddl	693	241	664	557	163	370	275	449
prob136.pddl	624	602	302	301	289	289	460	515
prob137.pddl	35	26	28	29	72	55	65	42
prob138.pddl	276	220	791	419	818	525	270	997
prob139.pddl	48	77	55	119	43	161	64	80
prob140.pddl	1366	1115	2965	—	1322	2184	1494	712
prob141.pddl	40	53	41	45	64	62	67	56
prob142.pddl	41	97	51	70	186	65	75	73
prob143.pddl	54	36	56	56	48	73	47	56
prob144.pddl	51	44	106	94	72	63	64	59
prob145.pddl	32	38	66	31	55	55	76	48
prob146.pddl	774	505	531	486	688	189	327	417
prob147.pddl	994	482	2150	891	738	667	503	854
prob148.pddl	—	—	880	601	3314	—	—	2845
prob149.pddl	—	3205	—	1764	2031	2215	3554	3455
prob150.pddl	139	93	118	101	232	154	152	151
prob151.pddl	2408	—	712	2039	1598	951	1032	1481
prob153.pddl	320	250	533	278	449	259	170	264
prob154.pddl	—	—	2705	—	2412	2895	2366	—
prob155.pddl	3222	—	1142	1877	2561	2305	2335	—
prob156.pddl	—	—	3089	3121	—	1965	2593	—
prob157.pddl	241	1352	309	1490	716	388	230	284

Table 5: Timing results for the `logistics` domain, where non-temporal constraints are processed with Huang's system and temporal constraints with `tk2sat`. All times are in seconds; a dash indicates that the problem was not solved within one hour.

chaining algorithm to produce plans quickly. We proved that temporal knowledge can be compiled into a constraint problem representation of a planning problem succinctly enough to significantly decrease planning time on hard problems. Moreover, we showed that even in simple planning domains, one may specify nontrivial and effective temporal constraints that cannot be specified non-temporally.

We presented our method of compiling MITL into a SAT or CSP instance in §3. For a given MITL specification, the method first generates an equivalent propositionally-labeled automaton that accepts a finite sequence of world-states if and only if the sequence satisfies the original specification. We outlined in §3.2.4 how forward- or backward-chaining algorithms could incorporate temporal constraints through this representation more effectively than with the original MITL representation. For a constraint-based planner, however, the automaton is only an intermediate representation. The final stage of our technique compiles the automaton into a set of SAT CNF-clauses or CSP

Number of clauses for Huang's system and `tk2sat` on domain `logistics`						
Problem	**act act**	**act all**	**all act**	**all all**	**glbl act**	**glbl all**
prob131.pddl	124	124	136	136	133	133
prob133.pddl	—	—	—	183	177	—
prob134.pddl	141	141	154	155	151	151
prob136.pddl	133	134	146	147	143	143
prob137.pddl	117	117	128	129	126	126
prob138.pddl	145	146	159	159	155	156
prob139.pddl	134	135	147	147	143	144
prob140.pddl	159	—	174	175	170	171
prob141.pddl	158	158	169	170	165	166
prob142.pddl	146	147	157	158	154	154
prob143.pddl	144	144	155	155	151	152
prob144.pddl	122	122	134	135	131	132
prob145.pddl	134	134	146	146	143	143
prob146.pddl	147	148	160	161	157	158
prob147.pddl	152	153	166	167	162	163
prob148.pddl	159	159	173	—	—	170
prob149.pddl	—	165	180	180	176	177
prob150.pddl	164	165	179	180	175	176
prob151.pddl	178	179	194	195	190	191
prob153.pddl	152	153	166	167	162	163
prob154.pddl	165	—	181	181	177	—
prob155.pddl	176	177	192	193	188	—
prob156.pddl	166	166	—	182	177	—
prob157.pddl	140	141	154	155	151	151

Table 6: Number of clauses in final SAT instances; numbers are in thousands.

constraints that is appended to the rest of the constraint problem. We showed that the size of this set is quadratic in the number of nodes in the automaton and linear in the number of levels in the planning graph. Though theoretically exponentially upper-bounded in size, the particle tableau algorithm produces small automata in practice; hence, our technique produces relatively small SAT and CSP encodings.

Our implementation, the `tk2sat` framework, is embedded within the BLACKBOX constraint-based planning system. Users specify an extra control file containing user-defined predicates, temporal action guards, and temporal global constraints. The system then instantiates action guards and global constraints for actions and planning literals present in the planning graph. In §4.2, we presented empirical results showing, first, that temporally-specified action guards and global constraints decrease the planning times for hard problems, and second, that combining Huang *et al.*'s control framework and `tk2sat` produces a planner that is superior to either system alone. Indeed, our test set contains several problems that were only solved within an hour by using the two systems to process non-temporal and temporal constraints.

There are at least two major opportunities for future research. First, a planning system could *learn* temporal rules using finite-state machine learning algorithms and techniques from inductive logic programming (ILP). Such a system would use ILP techniques to extract a set of compact world-state descriptions common to sequences of world-states in a specific domain. Then, using sequences leading to successful plans, as well as sequences from failed paths, the system would learn a propositionally-labeled automaton representing a global constraint. For action guards, the system would look at sequences ending at a given action, separating instances of effective uses of the action from ineffective uses. An effective use is one that appears in the final plan. Finally, the system would learn an action guard automaton from these sequences.

Second, our automata-based framework for using temporal knowledge in forward-chaining, backward-chaining, and constraint-based planning algorithms could be used for applications other than action guards and global constraints meant to accelerate planning. Indeed, the latest version of PDDL incorporates temporally-extended goals; one can imagine later versions allowing temporally-dependent actions. Further, a reactive planner, one that plans and reacts to a changing environment, could use temporal information to create complex history-dependent actions.

Acknowledgments

I want to thank my thesis advisor, Dr. Pandurang Nayak, for providing insightful comments and suggestions during the research process, as well as reading and suggesting changes to drafts of this thesis. The STeP research group generously provided the computational resources to run the empirical tests. Additionally, I thank the individuals of the STeP group for their friendship and comments on this and other work. Finally, I am deeply grateful for Professor Zohar Manna's mentorship over the past three years.

References

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of Second International Workshop On Temporal Representation and Reasoning (TIME)*, 1995.

[2] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[3] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[4] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[5] Giuseppe De Giacomo and Moshe Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *ECP*, pages 226–238, 1999.

[6] Giuseppe De Giacomo and Moshe Y. Vardi. Reasoning about actions and planning in LTL action theories. In *KR*, 2002.

[7] Yi-Cheng Huang, Bart Selman, and Henry A. Kautz. Control knowledge in planning: Benefits and tradeoffs. In *AAAI/IAAI*, pages 511–517, 1999.

[8] S. Kambhampati. Planning graph as (dynamic) CSP: Exploiting EBL, DDB and other CSP techniques in Graphplan. *IJCAI*, 1999.

[9] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14-16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[10] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359–363, 1992.

[11] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[12] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, New York, 1995.

[13] Drew McDermott. PDDL—the planning domain definition language. AIPS-98 Competition Committee, June 1998.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[15] D. Smith, J. Frank, and A. J'onsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1):61–94, 2000.

[16] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.