Planning with Propositional Temporally Extended Goals Using Heuristic Search

Technical Report CSRG-537, Department of Computer Science, University of Toronto

Jorge A. Baier and **Sheila McIlraith**
Deptartment of Computer Science,
University of Toronto,
Toronto, ON, M5S 3H5,
Canada

Abstract

Temporally extended goals (TEGs) refer to properties that must hold over intermediate and/or final states of a plan. The main strategy for planning with TEGs is to prune the search space during planning via goal progression. Currently, planners for TEGs prune the search space using heuristic search. In this paper we propose a method for planning with propositional TEGs using heuristic search. To this end, we translate an instance of a planning problem with TEGs into an equivalent classical planning problem. With this translation in hand, we exploit heuristic search to determine a plan. Our translation is based on the construction of a nondeterministic finite automaton for the TEG. We propose two alternative translations: the first uses only ADL operators, and the second (more efficient) uses derived predicates. We prove the correctness of our algorithm and analyze the complexity of the resulting representation. The translator is fully implemented and available. Our approach consistently outperforms existing approaches to planning with TEGs, often by orders of magnitude.

1 Introduction

In this paper we address the problem of generating finite plans for temporally extended goals (TEGs) using heuristic search. TEGs refer to properties that must hold over intermediate and/or final states of a plan. From a practical perspective, TEGs are compelling because they encode many realistic but complex goals that involve properties other than those concerning the final state. Examples include achieving several goals in sequence (e.g., book flight after confirming hotel availability),

safety goals such as maintenance of a property, (e.g., $\Box open(door)$, i.e. the door must always be open), and achieving a goal within some number of steps (e.g., at most 3 states after lifting a heavy object, the robot must recharge its batteries).

Planning with TEGs is fundamentally different from using temporally extended domain control knowledge to guide search (e.g., TLPLAN (Bacchus & Kabanza, 1998), and TALPLAN (Kvarnström & Doherty, 2000)). TEGs express properties of the plan we want to generate, whereas domain control knowledge expresses general properties of the search for a class of plans (Kabanza & Thiébaux, 2005). As a consequence, domain control knowledge is generally associated with an additional final-state goal.

A strategy for planning with TEGs, as exemplified by TLPLAN, is to use some sort of blind search on a search space that is constantly pruned by the progression of the temporal goal formula. This works well for safety-oriented goals (e.g., $\Box open(door)$) because it prunes those actions that falisfy the goal. Nevertheless, it is less effective with respect to liveness properties such as $\Diamond at(Robot, Home)$.

Our objective is to exploit heuristic search to efficiently generate plans with TEGs. We envisioned two ways to achieve this. The first was to convert a TEG planning problem to a classical planning problem where the goal was expressed in terms of the final state, and to use existing heuristic search techniques. The second possible approach was to develop our own heuristic for TEG planning. We pursued the former approach, though achieving the latter requires much of the machinery created for the former approach.

In contrast to previous approaches, we propose to represent TEGs in f-LTL, a version of propositional linear temporal logic (LTL) (Pnueli, 1977) which can only be interpreted by finite computations, and is more natural for expressing properties of finite plans. To convert a TEG to a classical planning problem we provide a translation of f-LTL formulae to nondeterministic finite automata (NFA). Once the TEG is represented as an NFA, we provide two alternative constructions of the classical planning problem, one that uses ADL operators and a second construction that uses defined predicates. We prove the correctness of our algorithm, analyze the space complexity of our translations and suggest techniques to reduce space.

Our translators are fully implemented and will be made available on the Web. They output PDDL problem descriptions, which makes our approach amenable to use with a variety of classical planners. We have experimented with two heuristic search planners, FF (Hoffmann & Nebel, 2001) and FF's extension for derived predicates, FF$_{\mathcal{X}}$ (Thiébaux, Hoffmann, & Nebel, 2005). Our experimental results illustrate the significant power heuristic search brings to planning with TEGs. In almost all of our experiments, we consistently outperform existing (non-heuristic) techniques for planning with TEGs. We also show that for complex goals, the translation that uses derived predicates is better than the one that uses only ADL

operators.

There are several pieces of related research that are notable. We group this research into two overlapping categories: 1) work that compiles TEGs into classical planning problems such as that of Rintanen (2000), and Cresswell and Coddington (2004), and 2) work that exploits automata representations of TEGs in order to plan with TEGs, such as Kabanza and Thiébaux's (2005) work on TLPLAN and work by Pistore and colleagues (Pistore, Bettin, & Traverso, 2001; dal Lago, Pistore, & Traverso, 2002). We discuss related work in more depth in Section 6.

2 Preliminaries

In this section we introduce f-LTL, a finite variant of LTL which we use to desribe our TEGs. Then we define general concepts and terminology that are necessary to our work.

2.1 f-LTL: LTL over finite computations

We introduce f-LTL logic, a variant of LTL (Pnueli, 1977) which we define over *finite* rather than infinite sequences of states. We use f-LTL formulae to describe TEGs for finite plans. f-LTL formulae augment LTL formulae with the propositional constant final, which is only true in final states of computation.

Definition 1 (f-LTL formula) *An f-LTL formula over a set $\mathcal{P}$ of propositions is one of the following.*

1. final, true *or* false,
2. *p, for any $p \in \mathcal{P}$,*
3. *$\neg\psi$, $\psi \wedge \chi$, $\bigcirc\psi$, or $\psi \mathcal{U} \chi$, if ψ and χ are f-LTL formulae.*

The semantics of an f-LTL formula is defined over finite sequences of states. A (finite) sequence of states $\sigma = s_0 s_1 \cdots s_n$ over a set of propositions $\mathcal{P}$ is such that $s_i \subseteq \mathcal{P}$, for each $i \in \{0, \ldots, n\}$. For notational convenience, we denote the suffix $s_i s_{i+1} \cdots s_n$ of σ by σ_i.

Let φ be an f-LTL formula. We say that $\sigma \models \varphi$ iff $\sigma_0 \models \varphi$. Furthermore,

1. $\sigma_i \models$ final iff $i = n$.
2. $\sigma_i \models$ true and $\sigma_i \not\models$ false.
3. $\sigma_i \models p$ iff $p \in s_i$

4. $\sigma_i \models \neg\varphi$ iff $\sigma_i \not\models \varphi$.

5. $\sigma_i \models \psi \wedge \chi$ iff $\sigma_i \models \psi$ and $\sigma_i \models \chi$.

6. $\sigma_i \models \bigcirc\varphi$ iff $i < n$ and $\sigma_{i+1} \models \varphi$.

7. $\sigma_i \models \psi \mathsf{U} \chi$ iff there exists a $j \in \{i, \ldots, n\}$ such that $\sigma_j \models \chi$ and for every $k \in \{i, \ldots, j-1\}$, $\sigma_k \models \psi$.

Standard temporal operators such as *always* ($\Box$), *eventually* ($\Diamond$), and *release* (R), and additional binary connectives are defined in terms of the basic elements of the language.

$$\psi \vee \chi \stackrel{\text{def}}{=} \neg(\neg\psi \wedge \neg\chi), \qquad \psi \supset \chi \stackrel{\text{def}}{=} \neg\psi \vee \chi,$$
$$\psi \equiv \chi \stackrel{\text{def}}{=} (\psi \supset \chi) \wedge (\chi \supset \psi), \qquad \psi \mathsf{R} \chi \stackrel{\text{def}}{=} \neg(\neg\psi \mathsf{U} \neg\chi),$$
$$\Box\varphi \stackrel{\text{def}}{=} \mathsf{false}\,\mathsf{R}\,\varphi, \qquad \Diamond\varphi \stackrel{\text{def}}{=} \mathsf{true}\,\mathsf{U}\,\varphi.$$

As in LTL, we can rewrite formulae containing U and R in terms of what has to hold true in the "current" state and what has to hold true in the "next" state. This is accomplished by identities 1 and 2 in the following proposition.

Proposition 1 *The following are identities of f-LTL.*

1. $\psi \mathsf{U} \chi \equiv \chi \vee (\psi \wedge \bigcirc(\psi \mathsf{U} \chi))$.

2. $\psi \mathsf{R} \chi \equiv \chi \wedge (\mathsf{final} \vee \psi \vee \bigcirc(\psi \mathsf{R} \chi))$.

3. $\neg\bigcirc\varphi \equiv \mathsf{final} \vee \bigcirc\neg\varphi$.

Limiting f-LTL to finite computations results in several obvious discrepancies in the interpretation of LTL and f-LTL formulae. In particular, discrepancies can arise with LTL formulae that force their models to be infinite. For example, in f-LTL the formula $\phi \stackrel{\text{def}}{=} \Box(p \supset \bigcirc q) \wedge \Box(q \supset \bigcirc p)$ is equivalent to $\Box\neg(p \vee q)$. This is because if $\sigma \models \phi$ then "p or q" cannot be true of any state of σ, since otherwise σ could not be a finite computation. A second example is the LTL formula $\Box p$ which in f-LTL is *not* equivalent to $p \wedge \bigcirc\Box p$. If it were, $\Box p$ could never be true in computations with a single state. The interpretation of the $\bigcirc$ operator, represented by identity 3 of Proposition 1, is also a source of discrepancies. The reader familiar with LTL, will note that identity 3 replaces LTL's equivalence $\neg\bigcirc\varphi \equiv \bigcirc\neg\varphi$. This formula does not hold in f-LTL because in f-LTL, $\bigcirc\varphi$ is true in a state iff there exists a next state that satisfies φ. Since our logic is finite, the last state of each model has no successor, and therefore in such states $\neg\bigcirc\varphi$ holds for every φ.

Although there are differences between LTL and f-LTL, their expressive power is similar when it comes to describing temporally extended goals for finite planning. Indeed, f-LTL has the advantage that it is tailored to refer to finite plans. As a consequence, we can express goals that cannot be expressed with LTL. Some examples follow.

Example 1 The following are temporal f-LTL goals together with their intuitive meaning.

- $\Box(\mathsf{final} \supset at(Robot, R1))$: In the final state, $at(Robot, R1)$ must hold. This is one way of encoding final-state goals in f-LTL.
- $\Box((closed(D_1) \wedge \bigcirc \neg closed(D_1)) \supset \bigcirc\bigcirc\bigcirc closed(D_1))$: If D_1 was closed at time step i, and then becomes opened at time step $i+1$, then it must be closed by time step $i+3$, for every i.
- $\neg delivered(R1) \mathsf{U} delivered(R2) \wedge \Diamond delivered(R1)$: Both $R1$ and $R2$ must be delivered, but $R2$ must be delivered before $R1$.
- $\Diamond(p \wedge \bigcirc\bigcirc\mathsf{final})$: p must hold true two states before the plan ends. This is an example of a goal that cannot be expressed in LTL, since it does not have the final constant.

When writing f-LTL goals, one has to be careful not to use formulae that require infinite plans, since they may be reduced to a contradictory formula. Indeed, the algorithm we present in the next section will automatically generate a non-accepting automaton for these types of formulae.

2.2 Planning Problems

A planning problem is a tuple $\langle \mathcal{I}, \mathcal{D}, \mathcal{G}, \mathcal{T} \rangle$, where $\mathcal{I}$ is the *initial state*, represented as a set of first-order (ground) positive facts; $\mathcal{D}$ is the *domain description*; $\mathcal{G}$ is a temporal formula describing the *goal*, and $\mathcal{T}$ is a (possibly empty) set of *derived predicate* definitions.

A domain description is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R} \rangle$, where $\mathcal{C}$ is a set of *causal rules*[1], and $\mathcal{R}$ a set of *action precondition rules*. Causal rules correspond to positive and negative *effect axioms* in the situation calculus (Pednault, 1989; McCarthy & Hayes, 1969). Intuitively, a positive (resp. negative) causal rule defines when a fluent becomes true (resp. false) after performing an action. We represent positive and

[1] Done for notational convenience and clarity. Note that ADL operators can be constructed from causal rules and vice versa (Pednault, 1989).

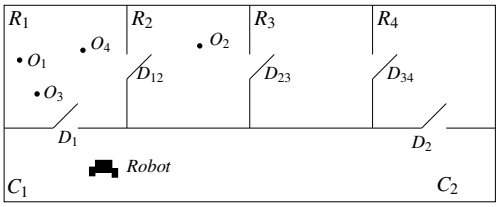

Figure 1: The robot domain.

negative causal rules by the triple $\langle a(\vec{x}), c(\vec{x}), f(\vec{x})\rangle$ and $\langle a(\vec{x}), c(\vec{x}), \neg f(\vec{x})\rangle$ respectively, where $a(\vec{x})$ is an *action term*, $f(\vec{x})$ is a *fluent term*, and $c(\vec{x})$ is a first-order formula, each of them with free variables among those in $\vec{x}$. $\langle a(\vec{x}), \Phi^{+}_{a,f}(\vec{x}), f(\vec{x})\rangle \in \mathcal{C}$ (resp. $\langle a(\vec{x}), \Phi^{-}_{a,f}(\vec{x}), \neg f(\vec{x})\rangle \in \mathcal{C}$) expresses that fluent $f(\vec{x})$ becomes true (resp. false) after performing action $a(\vec{x})$ in the current state if condition $\Phi^{+}_{a,f}(\vec{x})$ (resp. $\Phi^{-}_{a,f}(\vec{x})$) holds. As with ADL operators, the condition $c(\vec{x})$, can contain quantified first-order subformulae. Finally, we assume that for each action term and fluent term, there exists at most one positive and one negative causal rule in $\mathcal{C}$.

The set $\mathcal{R}$ of action precondition rules consists of tuples $\langle a(\vec{x}), \pi(\vec{x})\rangle$, such that $a(\vec{x})$ is an action term, and $\pi(\vec{x})$ is a first-order condition. Intuitively $\langle a, \pi\rangle \in \mathcal{R}$ means that it is possible to execute a in a state that satisfies condition π.

All free variables in rules of $\mathcal{C}$ or $\mathcal{R}$ are regarded as universally quantified.

Example 2 Consider the robot domain defined by Bacchus and Kabanza (1998). In this domain, depicted in Figure 1, there is a robot, some objects and six locations. Four of the locations correspond to rooms ($R_1, \ldots, R_4$), and two of them represent the corridor (C_1 and C_2). Rooms are connected by doors, which can be opened or closed. The robot can move between connected rooms, close or open doors, and grasp or drop objects. It can hold one object at a time. The causal rules for this domain are the following.

Positive	**Negative**
$\langle open(d), \mathsf{true}, opened(d)\rangle$	$\langle close(d), \mathsf{true}, \neg opened(d)\rangle$
$\langle grasp(o), \mathsf{true}, holding(o)\rangle$	$\langle grasp(o), \mathsf{true}, \neg handempty\rangle$
$\langle release(o), \mathsf{true}, handempty\rangle$	$\langle release(o), \mathsf{true}, \neg holding(o)\rangle$
$\langle move(x,y), \begin{array}{l} o = robot \vee \\ holding(o) \end{array}, at(o,y)\rangle$	$\langle move(x,y), \begin{array}{l} o = robot \vee \\ holding(o) \end{array}, \neg at(o,x)\rangle$

2.3 Regression

The causal rules of a domain describe the dynamics of individual fluents. However, to model an NFA in a planning domain, we must also know the dynamics of arbitrary complex formulae, such as for example, the causal rule for $at(o,R_1) \wedge holding(o)$. This is normally accomplished by goal regression (Waldinger, 1977; Pednault, 1989; Reiter, 2001).

To characterize these additional causal rules without articulating them explicitly, we introduce the relation `causes` that holds over the set of valid causal rules. As such, $\langle a, c, (\neg) f \rangle \in$ `causes` if $\langle a, c, (\neg) f \rangle \in \mathcal{C}$. Furthermore, for arbitrary fluent F and action A, if there is no c such that $\langle A, c, (\neg) F \rangle \in \mathcal{C}$, then $(A, \mathsf{false}, (\neg) F) \in$ `causes`.

To define causal laws for boolean formulae of fluents it suffices to define negation and conjunction since disjunction follows from these. The following definitions were obtained using Reiter's regression operator (Reiter, 2001, pg. 64). Here, we assume that $\alpha(\vec{x})$ is a boolean formula of fluents with free variables among the vector of variables $\vec{x}$. Furthermore, $\vec{t}$ is a vector of variables or constants.

Instantiation & Negation: If $(a(\vec{x}), \Phi(\vec{x}), \alpha(\vec{x})) \in$ `causes`, then,

1. $\langle a(\vec{x}), \vec{x} = \vec{t} \wedge \Phi(\vec{x}), \alpha(\vec{t}) \rangle \in$ `causes`, and

2. $\langle a(\vec{x}), \vec{x} = \vec{t} \wedge \Phi(\vec{x}), \neg\neg\alpha(\vec{t}) \rangle \in$ `causes`.

Conjunction Suppose the following tuples are in `causes`.

$$\langle a(\vec{x}), \Phi^{+}_{a,\alpha}(\vec{x}), \alpha(\vec{t_1}) \rangle, \qquad \langle a(\vec{x}), \Phi^{-}_{a,\alpha}(\vec{x}), \neg\alpha(\vec{t_1}) \rangle,$$
$$\langle a(\vec{x}), \Phi^{+}_{a,\beta}(\vec{x}), \beta(\vec{t_2}) \rangle, \qquad \langle a(\vec{x}), \Phi^{-}_{a,\beta}(\vec{x}), \neg\beta(\vec{t_2}) \rangle$$

Then

1. $\langle a(\vec{x}), \Phi^{+}_{a,\alpha\wedge\beta}, \alpha(\vec{t_1}) \wedge \beta(\vec{t_2}) \rangle \in$ `causes`, where:

$$\Phi^{+}_{a,\alpha\wedge\beta} = [\Phi^{+}_{a,\alpha}(\vec{x}) \wedge \Phi^{+}_{a,\beta}(\vec{x})] \vee [\alpha(\vec{x}) \wedge \neg\Phi^{-}_{a,\alpha}(\vec{x}) \wedge \Phi^{+}_{a,\beta}(\vec{x})] \vee [\beta(\vec{x}) \wedge \neg\Phi^{-}_{a,\beta}(\vec{x}) \wedge \Phi^{+}_{a,\alpha}(\vec{x})]$$

2. $\langle a(\vec{x}), \Phi^{-}_{a,\alpha\wedge\beta}, \neg(\alpha(\vec{t_1}) \wedge \beta(\vec{t_2})) \rangle \in$ `causes`, where

$$\Phi^{-}_{a,\alpha\wedge\beta} = \Phi^{-}_{a,\alpha}(\vec{x}) \vee \Phi^{-}_{a,\beta}(\vec{x}).$$

The downside of regression is its space complexity; the size of the conditions in the causal laws can grow exponentially. This is justified by the following proposition.

Proposition 2 *Let $\varphi = f_0 \wedge f_1 \wedge \ldots \wedge f_n$. Then, assuming no simplifications are made, $|\Phi^+_{a,\varphi}| = \Omega(3^n(m^+ + m^-))$, where $m^+ = \min_i |\Phi^+_{a,f_i}|$, i.e. m^+ is the size of the smallest condition on a positive effect for action a among all fluents f_i's. Similarly, $m^- = \min_i |\Phi^-_{a,f_i}|$.*

To reduce the size of the resulting formulae, one could try to do boolean simplifications. Unfortunately, simplifying a boolean formula is also exponential in the size of the formula. Despite this bad news, it's important to observe that alternative techniques for planning with TEGs commonly use progression and that *progressed* formulae also grow exponentially. Systems such as TLPLAN commonly use boolean simplification with great success to reduce the size of progressed formulae.

3 Translating f-LTL to NFA

In this section we present an algorithm to translate f-LTL formulae into state-labeled non-deterministic finite automata and then show how to simplifiy them into non-deterministic finite automata. We prove the correctness of our algorithm. Once we have this translation defined, we can use it to compile our planning problems with TEGS into classical planning problems.

For every LTL formula φ over a set of variables $\mathcal{P}$, there exists a Büchi automaton $A_\varphi = (Q, \Sigma, \delta, Q_0, F)$, where $\Sigma = 2^{\mathcal{P}}$ such that given an infinite state sequence σ, A_φ accepts σ if and only if $\sigma \models \varphi$ (Vardi & Wolper, 1994; Vardi, 1995). Although the construction in by Vardi and Wolper (1994) is very impractical, algorithms have been proposed for the generation of pragmatic automata (e.g. Gerth, Peled, Vardi, & Wolper, 1995).

To our knowledge there exists no pragmatic algorithm for translating a finite version of LTL such as the one we use here. In this section, we propose such an algorithm and prove its correctness. Our algorithm is a modification of the one proposed by Gerth et al. (1995), and in the first stage generates a *state-labeled NFA*.

Definition 2 (State-labeled NFAs (SLNFA))
A state-labeled NFA (SLNFA) is a tuple $A = \langle Q, \Sigma, \delta, L, Q_0, F \rangle$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, Σ is the alphabet, $\delta \subseteq Q \times Q$ is

a transition relation, $F \subseteq Q$ is the set of final states, and $L : Q \to 2^{\Sigma}$ is a state-labeling function.

A run is a finite sequence $\rho = q_0 q_1 \dots q_n$, such that $q_0 \in Q_0$ and $(q_i, q_{i+1}) \in \delta$, for all $i \in \{0, \dots, i-1\}$. ρ is an accepting run iff $q_n \in F$. We say that an SLNFA accepts a word $w = x_0 x_1 \cdots x_n$ in Σ^* iff there exists an accepting execution $\rho = q_0 q_1 \dots q_n$ such that for each $i \in \{0, \dots, n\}$, $x_i \in L(q_i)$.

Example Consider the SLNFA $A = \langle \{q_0, q_1\}, \Sigma, \delta, L, \{q_0\}, \{q_1\} \rangle$, where $\Sigma = 2^{\{p,q,r\}}$, $\delta = \{(q_0, q_1), (q_1, q_1)\}$, and $L(q_0) = \Sigma$ and $L(q_1) = \{s \subseteq \Sigma \,|\, q \in s\}$. Observe that this automaton accepts sequences of states over the set of propositions $\{p, q, r\}$; in fact, it accepts every σ such that $\sigma \models \bigcirc\Box q$.

3.1 The algorithm

The translation algorithm (Figure 2) is a modification of the one proposed by Gerth et al. (1995). Our algorithm incorporates the new propositional constant final and modifies the acceptance condition of the generated automaton.

To represent a node of the automaton, the algorithm uses Gerth et al.'s data structure *Node*, which is a tuple $\langle Name, Incoming, New, Old, Next, Father \rangle$. The field *Name* : contains the name of the node; *Incoming* is the list of node names with an outgoing edge leading to *Node*; *New* contains formulas that must hold at the current state but that have not been processed by the algorithm; *Old* contains the formulas that must hold in the node that have been processed by the algorithm; *Next* contains temporal formulae that have to be true in the immediate successors of *Node*. During the construction of the graph, a node can be *split* into two new nodes, these two new nodes contain the name of their father in the field *Father*.

The algorithm takes as input an f-LTL formula in negated normal form, i.e., a formula where all $\neg$'s have been pushed in such that they only occur in front of propositions or constants. Furthermore, the formula can only contain the binary operators $\wedge$ and $\vee$. It is easy to see that using the equivalences in Section 2.1 it is always possible to translate any f-LTL formula to this form.

The procedure `gen_graph`(φ) constructs the graph of A_φ. Initial states are those that contain *Init* in its field *Incoming*.

During the construction of the graph, all the nodes are kept in the set *NodeSet*. Given a formula φ, the algorithm starts processing one node that contains formula φ in its *New* field. Furthermore, set *NodeSet* is empty (line 50).

When processing node *N*, the algorithm checks whether there are any pending formulae in the *New* field. If there are not, then the node is ready to be added to the *NodeSet*. Two cases can hold:

1. If there is already a node in *NodeSet* with the same fields *Old* and *Next*, then its *Incoming* list is updated by adding those nodes in N's incoming list (line 6).

2. If there is no such a node, then N is added to *NodeSet*. Then, a new node is created for processing if $\mathsf{final} \notin \mathit{Old}$. This node contains N in its incoming list, and the field *New* set to N's *Next* field. The fields *New* and *Old* of the new node are empty (lines 8–18).

Otherwise, if *New* is not empty, formula η is removed from *New* and added to *Old*. Depending on the syntax of η, one of the actions below is taken and then the node continues to be processed.

1. In case η is a literal, then if $\neg\eta$ is in *Old*, the node is discarded (a contradiction has occurred). Otherwise, η is added to *Old* and the node continues to be processed.

2. Otherwise, depending on the syntactic form of η, the following actions are taken.

   (a) If $\eta = \varphi \wedge \psi$, both φ, and ψ are added to *New*.

   (b) If $\eta = \bigcirc\psi$, then ψ is added to *Next*.

   (c) If η is one of $\varphi \wedge \psi$, $\varphi \,\mathsf{U}\, \psi$, or $\varphi \,\mathsf{R}\, \psi$, then N is split into two nodes. The set $\mathbf{New1}(\eta)$ and $\mathbf{New2}(\eta)$ are added, respectively, to the *New* field of the first and second nodes. These functions are defined as follows:

η	$\mathbf{New1}(\eta)$	$\mathbf{New2}(\eta)$
$\varphi \vee \psi$	$\{\varphi\}$	$\{\psi\}$
$\varphi \,\mathsf{U}\, \psi$	$\{\varphi, \bigcirc(\varphi \,\mathsf{U}\, \psi)\}$	$\{\psi\}$
$\varphi \,\mathsf{R}\, \psi$	$\{\psi, \mathsf{final} \vee \bigcirc(\varphi \,\mathsf{R}\, \psi)\}$	$\{\varphi, \psi\}$

   The intuition of the split lies in standard f-LTL equivalences. For example, $\varphi \,\mathsf{U}\, \psi$ is equivalent to $\psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$, thus one node verifies the condition ψ, whereas the other verifies $\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi)$.

The following definition is required in the proof of the correctness of the algorithm and to define final states of the automaton.

Definition 3 ($\Delta^{-}(q)$) *Let $\Delta(q)$ be the value of the Old field for node q, when node q has been processed. We define $\Delta^{-}(q)$ as the set containing all the literals in $\Delta(q)$.*

```
1  function expand(Node,NodeSet)
2  begin
3      if New(Node) = ∅ then
4          if ∃N ∈ NodeSet and Old(N) = Old(Node) and Next(N) = Next(Node)
           then
5              Incoming(ND) ←
6                  Incoming(ND) ∪ Incoming(Node)
7              return NodeSet
8          else
9              if final ∉ Old(Node) then
10                 return expand([Name ← Father ← new_name(),
11                                Incoming ← Name(Node),
12                                New ← Next(Node), Old ← ∅
13                                Next ← ∅], {Node} ∪ NodeSet)
14             else
15                 if Next(Node) = ∅ then
16                     return {Node} ∪ NodeSet
17                 else
18                     return NodeSet  /* discarded                    */
19     else
20         choose η ∈ New(Node)
21         New(Node) ← New(Node) \ {η}
22         if η ≠ True and η ≠ False then Old(Node) ← Old(Node) ∪ {η}
23         switch η do
24             case η is a literal, True or False
25                 if η = False or η̄ ∈ Old(Node) then
26                     return NodeSet  /* discarded                    */
27                 else
28                     return expand(Node,NodeSet)
29             case η = ○φ
30                 Next(Node) ← Next(Node) ∪ {φ}
31                 return expand(Node,NodeSet)
32             case η = φ ∧ ψ
33                 New(Node) ← New(Node) ∪ ({φ, ψ} \ Old(Node))
34                 return expand(Node,NodeSet)
35             case η = φ ∨ ψ or φ R ψ or φ U ψ
36                 Node1 ← [Name ← new_name(),
37                          Father ← Name(Node),
38                          Incoming ← Incoming(Node),
39                          New ← New(Node) ∪ New1(η),
40                          Old ← Old(Node),
41                          Next ← Next(Node)]
42                 Node2 ← [Name ← new_name(),
43                          Father ← Name(Node),
44                          Incoming ← Incoming(Node),
45                          New ← New(Node) ∪ New2(η),
46                          Old ← Old(Node),
47                          Next ← Next(Node)]
48                 return expand(Node2,expand(Node1,NodeSet))
49 end
50 function gen_graph(φ) begin
```

The set of final states, F is as follows:

$$F = \{q \in Q \,|\, Next(q) = \emptyset \text{ and } \neg\mathsf{final} \notin \Delta^{-}(q)\}.$$

I.e. final states are those states that do not have to satisfy any obligations in the future and that are not forced to be non-final by the actual requirement of the state. The labeling function is the following: $L(q) = \{s \subseteq 2^{\mathcal{P}} \,|\, s \models \Delta^{-}(q) \setminus \{\mathsf{final}, \neg\mathsf{final}\}\}$.

The following theorem states the correctness of the algorithm.

Theorem 1 *Let A_φ be the automaton constructed by the algorithm from formula φ over the set $\mathcal{P}$ of propositions. Then A_φ accepts exactly the set of computations in $(2^{\mathcal{P}})^*$ that satisfy φ.*

Proof: See the appendix.

3.2 Simplifying SLNFAs into NFAs

The algorithm presented above often produces automata that are much bigger than the optimal. To simplify the automata, we have used a modification of the algorithm by Etessami and Holzmann (2000). This algorithm uses a simulation technique to simplify the automaton. In experiments Fritz (2003), it was shown to be slightly better than LTL2AUT (Daniele, Giunchiglia, & Vardi, 1999) at simplifying Büchi automata.

To apply the algorithm we generate a nondeterministic finite automaton that accepts finite computations (strings over $(2^{\mathcal{P}})^*$) equivalent to the SLNFA. A nondeterministic finite state automaton is a tuple $A = \langle Q, \Sigma, \delta, Q_0, F\rangle$, where Q is a finite set of states, $\delta \subseteq Q \times \Sigma \times Q$, and $\Sigma = 2^{\mathcal{P}}$. A run of A over string $w = x_0 x_1 \cdots x_n \in \Sigma^*$ is a sequence of states $\rho = q_0 q_1 \cdots q_n$, where $q_0 \in Q_0$, and $(q_i, a, q_{i+1}) \in \delta$ for some $a \subseteq x_i$, for all $i \in \{0, \ldots, n-1\}$. Run ρ is *accepting* if $q_n \in F$.

It is straightforward to convert a SLNFA to an equivalent NFA by adding one initial state and copying labels of states to any incoming transition. Figure 3 shows examples of NFAs generated by our implementation for some f-LTL formulae. Henceforth, unless stated otherwise, we assume A_φ is the NFA that results from simplifying the SLNFA generated by our algorithm.

3.3 Size complexity of the NFA

Although simplifications normally reduce the number of states of the NFA significantly, the resulting automaton can be exponential in the size of the formula in the worst case.

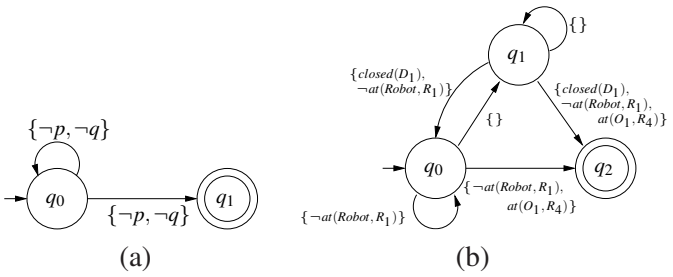

Figure 3: Simplified NFA for (a) $\Box(p \supset \bigcirc q) \wedge \Box(q \supset \bigcirc p)$, and (b) $\Box(at(Robot, R_1) \supset \bigcirc\Diamond closed(D_1)) \wedge \Diamond\Box at(O_1, R_4)$.

Proposition 3 *Let φ be in negated normal form, then the number of states of A_φ is $2^{O(|\varphi|)}$.*

There are simple cases where the translation blows up; e.g., for the formula $\Diamond p_1 \wedge \Diamond p_2 \wedge \ldots \wedge \Diamond p_n$, the resulting NFA has 2^n states. Intuitively each state keeps track of a particular combination of propositions that has been true in the past.

4 Compiling NFAs into a Planning Domain

Now that we are able to represent TEGs as NFAs, we show how the NFA can be encoded into a planning problem. We present two translations, one that represents the domain as a set of causal rules, and the other that represents the domain using both causal rules and derived predicates.

In the planning domain, each state of the automaton is represented by a fluent. More formally, for each state q of the automaton we add to the domain a new fluent E_q. The translation is such that if a sequence of actions $a_1 a_2 \cdots a_n$ is performed in state s_0, generating the succession of states $\sigma = s_0 s_1 \ldots s_n$, then fluent E_q is true in s_n if and only if there is a run of A_φ on σ that ends in state q.

Once the NFA is modeled inside the domain, the temporal goal in the newly generated domain in reduced to a property of the final state alone. Intuitively, this property corresponds to the accepting condition of the automaton.

To represent the dynamics of the states of the automaton, there are two alternatives. The first is to modify the domain's *causal rules* to give an account of their change. The second, is to define them as *derived predicates* or *axioms*.

In the rest of the section, we present both translations, we analyze related complexity issues, and show how our approach can subsume state space pruning by progression. Henceforth we assume the following:

- We start with a planning problem $\mathcal{L} = \langle \mathcal{I}, \mathcal{D}, \mathcal{G}, \mathcal{T} \rangle$, where $\mathcal{G}$ is a temporal formula in f-LTL.
- Temporal goal $\mathcal{G}$ is translated to the NFA $A_{\mathcal{G}} = (Q, \Sigma, \delta, Q_0, F)$.
- To simplify notation, we denote by $Pred(q)$ the set of predecessors of q. E.g., in Fig. 3(b), $Pred(q_0) = \{q_0, q_1\}$.
- Finally, we denote by $\lambda_{p,q}$ the formula $\bigvee_{(q,L,p)\in\delta} \bigwedge L$. E.g., in Fig. 3(b), $\lambda_{q_1,q_0} = closed(D_1) \wedge \neg at(Robot, R_1)$.

4.1 Translating NFA to Causal Rules

Recall that we have translated our TEG into an NFA and to encode this NFA in the planning domain, we have introduced fluents E_q, one for each state q of the automaton. In this first translation, we encode the dynamics of the fluent E_q as causal rules. For each fluent E_q we generate a new set of causal rules. The resulting new rules are added to the set $\mathcal{C}'$, which is initialized to $\emptyset$.

4.1.1 New causal rules

To understand the intuition behind the translation, consider the NFA shown in Figure 3(b). Suppose E_{q_2} is false in a state s_i. After performing action a_i, fluent E_{q_2} must become true in the resulting state, s_{i+1}, iff either E_{q_0} was true in s_i and $\neg at(Robot, R_1) \wedge at(O_1, R_4)$ is true in s_{i+1} or E_{q_1} was true in s_i and $\neg at(Robot, R_1) \wedge closed(D_1) \wedge at(O_1, R_4)$ is true in s_{i+1}. Notice that $\neg at(Robot, R_1) \wedge closed(D_1) \wedge at(O_1, R_4)$ can be true in s_{i+1} because a made the property true, or because it was true in s_i and a did not make it false.

To write the positive causal rule for E_{q_2} on action a, we must only refer to the state prior to the execution of a. To do so, we appeal once again to regression. For each action a, the positive action rule $\langle a, \Phi^+_{a,E_q}, E_q \rangle$ is added to $\mathcal{C}'$, where:

$$\Phi^+_{a,E_q} = \bigvee_{p \in Pred(q)\setminus\{q\}} E_p \wedge (\Phi^+_{a,\lambda_{p,q}} \vee (\lambda_{p,q} \wedge \neg \Phi^-_{a,\lambda_{p,q}})). \tag{1}$$

Note that $\Phi_{a,\lambda_{p,q}}$ is a condition obtained by regression.

For the negative case, consider state q_0 of the automaton. If E_{q_0} is true in some state s_i, then when getting to state s_{i+1} after performing a, it will become false if $\neg at(Robot, R_1)$ holds in s_{i+1} *and* it does not happen that in E_{q_1} is true in s_i and $\neg at(Robot, R_1) \wedge closed(D_1)$ is true in s_{i+1}.

Again, we need to appeal to regression. For each action a, the positive action rule $\langle a, \Phi^-_{a,E_q}, \neg E_q \rangle$ is added to $\mathcal{C}'$, where:

$$\Phi^{-}_{a,E_q} = \neg\Phi^{+}_{a,E_q} \wedge \neg(\Phi^{+}_{q,\lambda_{q,q}} \vee \lambda_{q,q} \wedge \neg\Phi^{-}_{a,\lambda_{q,q}}). \quad (2)$$

Note that $\lambda_{q,q}$ is false if there is no self transition in q.

Example 2 (cont.) In the robots domain, for the NFA of Figure 3(b) we would add the following positive causal rule for fluent E_{q_2} and action $close(x)$.

$$\begin{aligned}\langle close(x), E_{q_1} \wedge [&at(O_1,R_4) \wedge \neg at(Robot,R_1) \wedge closed(D_1) \vee \\ &at(O_1,R_4) \wedge \neg at(Robot,R_1) \wedge x = D_1)] \vee \\ &E_{q_0} \wedge at(O_1,R_4) \wedge \neg at(Robot,R_1), E_{q_2}\rangle\end{aligned}$$

4.1.2 New initial state

The original initial state must also be modified, since it now must include which fluents E_q are initially true. The new set of facts $\mathcal{I}'$ is the following:

$$\mathcal{I}' = \{E_q \,|\, (p,L,q) \in \delta, p \in Q_0, L \subseteq \mathcal{I}\}.$$

I.e., are the facts E_q such that q is reachable for some initial state p through a transition whose label is a fact that is true in $\mathcal{I}$.

4.1.3 New goal & planning problem

Intuitively, the automaton $A_{\mathcal{G}}$ accepts iff the temporally extended goal $\mathcal{G}$ is satisfied. Therefore, the new goal, $\mathcal{G}' = \bigvee_{p \in F} E_p$, is defined according to the acceptance condition of the NFA, i.e. the goal is achieved if $A_{\mathcal{G}}$ is in some final state. Note that $\mathcal{G}'$ is a non-temporal goal.

The final planning problem L' is $\langle \mathcal{I} \cup \mathcal{I}', \mathcal{C} \cup \mathcal{C}', \mathcal{R}, \mathcal{G}', \mathcal{T}\rangle$.

4.2 Translation by derived predicates

We now examine our second (more efficient) translation that exploits derived predictes. In this translation, as before, we use a fluent E_q to represent each state q of the automaton. However, this time we do not write a causal rule for E_q; rather, we define it as a derived predicate. Nevertheless, we will still need to add a few causal rules. The new causal rules, and defined predicates will be stored respectively in sets $\mathcal{C}'$ and $\mathcal{T}'$, which are initially set to $\emptyset$.

To achieve the translation, we require that the truth value of E_q in a state s_{i+1} be defined in terms of properties that hold true in s_{i+1}. However, as we saw previously,

the truth value of E_q in s_{i+1} depends on whether some fluents E_p hold true in the previous state, where p is a state of the automaton. Therefore, we need a way to represent in state s_{i+1} what fluents E_p were true in the previous state.

Thus, for each state q of the automaton we use an auxiliary fluent $Prev_q$ which is true in a state s iff E_q was true in the previous state. The dynamics of fluent $Prev_q$ is described by the following causal rules, which are added to $\mathcal{C}'$:

$$\langle a, E_q, Prev_q \rangle, \qquad \langle a, \neg E_q, \neg Prev_q \rangle, \tag{3}$$

for each a. The following definitions are added to $\mathcal{T}'$:

$$E_q \stackrel{\text{def}}{=} \bigvee_{p \in Pred(q)} Prev_p \wedge \lambda_{p,q},$$

4.2.1 New initial state

The new initial state must specify which fluents of the form $Prev_q$ are true. These are precisely those facts that correspond to the initial state of the automaton.

$$\mathcal{I}' = \{Prev_q \,|\, q \in Q_0\}.$$

4.2.2 New goal & planning problem

As before, the new goal is defined by $\mathcal{G}' = \bigvee_{p \in F} E_p$, and the new planning problem is $L' = \langle \mathcal{I} \cup \mathcal{I}', \mathcal{C} \cup \mathcal{C}', \mathcal{R}, \mathcal{G}', \mathcal{T} \cup \mathcal{T}' \rangle$.

4.3 Search Space Pruning by Progression

As previously noted, planners for TEGs such as TLPLAN are able to prune the search space by progressing temporal formulae representing the goal. It follows that, a state s is pruned by progression, if the progressed temporal goal in s is equivalent to false. Intuitively, this means that there is no possible sequence of actions that when executed in s would lead to the satisfaction of the goal.

In this section we discuss how to achieve search space pruning, analogous to that of TLPLAN, within our NFA approach. Since our NFAs have no non-final states that do not lead to a final state, if at some state during the plan all fluents E_q are false for every $q \in Q$, then this means that the goal will never be satisfied. In the planning domain this can be achieved in two ways. One way is to add $\bigvee_{q \in Q} E_q$ as a state constraint (or *safety constraint*). The other way is to automatically generate preconditions for actions thereby achieving automated precondition control (Bacchus & Ady, 1999; Rintanen, 2000). The first alternative is the simplest one, but the heuristic planners we know of cannot currently handle it.

To do precondition control the precondition $\langle a, c\rangle \in \mathcal{R}$ can be replaced by $\langle a, c \wedge \neg\pi_a\rangle$, where π_a is such that $(a, \pi_a, \neg\bigvee_{q\in Q} E_q) \in \texttt{causes}$. The size of π_a is unfortunately exponential in the number of states since we need to appeal to regression. The following subsection analyzes other complexity issues.

This technique has the potential to be very useful, but we didn't use it in practice for two reasons: 1) our heuristic search planners didn't support such state constraints, and 2) it would not add to the performance of our system.

4.4 Size Complexity

Planning with the new translated theory is theoretically as hard as planning with the original theory. However, it is important to analyze the size of the translated planning problem. Since there are more fluents in the resulting domain, the planner will take time updating these fluents. This time will be, in general, linear in the size of the conditions that have to be evaluated when performing each action.

We can obtain quite different results depending on the type of translation being used. For the case of the translation into causal rules, the size complexity is linear in the number of states and transitions, but is exponential in the size of the labels of the automaton. Indeed, observe that expressions (1) and (2) are linear in the number of preceding states of q. However, the size of $\Phi^{+}_{a,\lambda_{p,q}}$, $\Phi^{-}_{a,\lambda_{p,q}}$, $\Phi^{+}_{a,\lambda_{p,p}}$, and $\Phi^{-}_{a,\lambda_{p,p}}$ is exponential in the number of binary connectives in each of these formulae (this is a consequence to Proposition 2). Moreover, since the number of causal rules in $\mathcal{C}'$ is proportional to the number of states and the number of actions, the total size as a function of the number of transitions is therefore, $O(n|Q|2^{\ell})$, where ℓ is the maximum size of a transition in $A_{\mathcal{G}}$, and n is the number of action terms in the domain.

For the case of the translation into derived predicates, we obtain a much better result. Observe that each of the definitions of (3) is linear in the size of $\lambda_{p,q}$ and the number of preceding states. Therefore, we obtain that the size of $\mathcal{T}'$ is $O(n|Q|\ell)$. Furthermore, the size of $\mathcal{C}'$ is only $O(n|Q|)$.

4.5 Reducing $|Q|$ and Handling Quantification

In the previous section we saw that the size of the resulting translation depends on the number of states of the automaton, $|Q|$. Previously, we also saw that $|Q|$ is generally exponential in the size of the temporal formula. This means that we could be generating quite big translations even if we choose to use derived predicates.

Fortunately, there is a way to reduce the size complexity by splitting a formula into different goals. Consider for example the formula $\varphi = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n$, which we know has an exponential NFA. We know that φ will be satisfied if each of the

conjuncts $\Diamond p_i$ is satisfied. If instead of generating a unique NFA for φ we generated a *different* NFA *for each* $\Diamond p_i$, then we could just plan for a goal equivalent to the conjunction of the acceptance conditions of each of those automata. For this particular φ this means that the number of states in the new planning problem is linear in n instead of exponential.

The same idea can be generalized to any combination of boolean formulae. In our implementation, we benefit from this property by exploting it even more. We preprocess the TEG formula using the follwing f-LTL equivalences:

$$(\phi \wedge \psi) \mathsf{U} \chi \equiv (\phi \mathsf{U} \chi) \wedge (\psi \mathsf{U} \chi),$$
$$\phi \mathsf{U} (\psi \vee \chi) \equiv (\phi \mathsf{U} \psi) \vee (\psi \mathsf{U} \chi),$$

and other similar equivalences that hold for the temporal connectives R and $\bigcirc$, effectively "pulling" binary connectives up in the formulae. With this technique, we generate more automata but avoid the risk of exponential explosion. This technique is also useful when handling f-LTL formulae that come from quantified formula. We elaborate upon this point in future work.

5 Implementation and Experiments

We implemented a compiler for the two proposed translations. The compiler takes a domain, described by causal rules, and a temporal goal, described in f-LTL, and generates a new planning problem as described in Section 4. Furthermore, a module of the program can convert the new problem into a PDDL domain/problem (McDermott, 1998), thereby enabling its use with a wide variety of available planners.

We conducted several experiments in the robots domain to test the effectiveness of our approach. In each experiment, we compiled the planning problem to PDDL. To evaluate the translation to causal rules (CR), we used FF as our heuristic planning engine (CR+FF). For the translation to derived predicates (DP), we used $\text{FF}_{\mathcal{X}}$ (DP+$\text{FF}_{\mathcal{X}}$), an extension of FF proposed by Thiébaux et al. (2005) that supports derived predicates.

Table 1 presents results obtained for various temporal goals by our translation and the C version of TLPLAN (TLPLAN-C). The second and third columns show statistics about the translation. The second column shows the time taken in each translation, and the third shows the number of states of the automata representing the goal. The rest of the columns show the time (t) and length (ℓ) of the plans for each approach. In the case of the TLPLAN-C, two times are presented. In the first (t) no additional search control was added to the planner, i.e. the planner was using only the goal to prune the search space. In the second (t-ctrl) we added (by hand)

Prb.	Comp.	No.	CR+FF		DP+FF$_{\mathcal{X}}$		TLPLAN-C		
	CR/DP	Sts.	t	ℓ	t	ℓ	t	$t_{\text{-ctrl}}$	ℓ
1	.02/.02	2	.02	6	.02	6	.07	.01	6
2	.02/.01	2	.02	8	.01	8	.04	.03	8
3	.09/.06	15	.04	10	.04	10	.20	.02	10
4	.06/.07	5	.03	6	.02	6	.38	.10	6
5	.07/.03	6	.04	15	.03	15	.5	.19	13
6	.49/.39	37	.19	16	.16	16	.51	.17	18
7	.05/.03	6	.05	9	.11	10	.96	.31	10
8	.07/.06	15	.05	10	.04	12	1.40	.04	10
9	.01/.02	4	.03	18	.03	18	13.90	.15	14
10	.04/.05	6	.07	32	.05	15	17.52	.40	14
11	.08/.04	5	.06	22	.03	20	m	m	–
12	.09/.02	5	.50	25	.03	24	m	m	–
13	.09/.05	6	m	–	.04	28	m	m	–
14	.32/.05	5	m	–	.10	33	m	m	–
15	.07/.03	5	.11	31	.09	34	m	m	–
16	.09/.04	10	m	–	.07	46	m	m	–

Table 1: Our approach compared to TLPLAN-C

additional control information to "help" TLPLAN do a better search. The character 'm' stands for *ran out of memory*.

TLPLAN-C is significantly outperformed by our approach, unless the goals are very simple (e.g. problem 1 is $\Diamond\Box(at(Robot,C_1) \wedge at(O_1,R_2))$, starting in the initial state depicted in of Figure 1). Only TLPLAN-C with extra control information can be somewhat competitive with our approach for more complex goals. This is the case for goal 3, which is defined as $\bigcirc^5 at(O_1,R_4) \vee \bigcirc^6 at(O_1,R_4) \wedge \Diamond\Box(at(Robot,C_1) \wedge at(O_1,R_2))$[2]. In this case, TLPLAN-C is able to perform better because its control information (added by hand) advises it not to grab any object different from O_1. Such information was not added to our compiler.

Another interesting case is the simplest problem for which TLPLAN-C runs out of memory. This is goal 11, which corresponds to $\Diamond[(at'(R_4) \vee at'(R_3)) \wedge \bigcirc AllClosed)] \wedge \Diamond\Box at(O_1,C_1)$, where *AllClosed* stands for a formula where all doors are closed, and $at'(r)$ stands for "both O_3 and O_4 are in r." On the other hand, goal 14 is like goal 11, but when at' stands "all objects are in r." In this case, FF runs out of memory in the preprocessing phase. This is due to the huge conditions that are part of the conditional effects of some actions, which in turn is due to the underlying regression used by the translation.

[2] $\bigcirc^5 p$ abbreviates $\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc p$.

Table 2 compares our approach's performance to that of the planner presented in (Kabanza & Thiébaux, 2005) (henceforth, TPBA), which uses Büchi automata to control search. Again, our planner clearly outperforms TPBA. This planner offers four templates to write automata. We have used one of them, which is of the form $\Diamond(p_1 \wedge \bigcirc(\Diamond p_2 \wedge \ldots \wedge \bigcirc\Diamond p_n)\ldots)$ [3].

Again, TPBA is significantly outperformed by our approach, even in the presence of extra control information added by hand (this is indicated by the '+c' in the table). In dfs mode, TPBA is able to solve every problem but more slowly and with less quality. In the bfs mode with no control information, TPBA fails for goal 4, which is "O_1 must eventually be at R_2, then at R_4, then at C_1, then at R_3, and finally at C_2". On the other hand, TPBA fails in bfs mode with control information for goal 10, which is defined as "eventually O_1 at R_2, then eventually all objects in R_4, and finally all objects in C_1." The control information added by hand in this case is "do not close any doors."

The results presented above, though very good, were expected. None of the planners we have compared to uses heuristic search, which means they may not have enough information to determine which action to choose during search. The TLPLAN family of planners is particularly efficient when control information is added to the planner. Usually this information is added by an expert in the planning domain. However, control information, while natural for classical goals, might be harder to write for temporally extended goals. The advantage of our approach is that we do not need to write this information and still be efficient. Moreover, control information can be added in the context of our approach by integrating it into the goal formula.

6 Related Work

There are several notable pieces of related work. The relationship between automata theory and temporal logic has been known for decades (Vardi, 1995), and it has been applied successfully in the software verification area. In relation to planning, de Giacomo and Vardi (1999) have considered the planning problem with TEGs represented as LTL formulae from a theoretical perspective. They reduce the problem of planning with TEGs to decision problems in automata theory, obtaining several complexity results about the decidability of planning under various settings.

The temporal extension of TLPLAN to search control used in our experiments (and referred to as TPBA) (Kabanza & Thiébaux, 2005), considers a Büchi au-

[3] Three more are available. One is for classical goals, another is for cyclic (infinite) goals, and the third is very similar to the one we are using.

Prb.	DP+FF$_{\mathcal{X}}$		TPBA/$_{\text{dfs+c}}$		TPBA/$_{\text{dfs}}$		TPBA/$_{\text{bfs+c}}$		TPBA/$_{\text{bfs}}$	
	t	ℓ	t	ℓ	t	ℓ	t	ℓ	t	ℓ
1	.00	2	.06	2	0.3	2	0.24	2	0.44	2
2	.01	5	.51	15	30	563	0.96	5	44.42	5
3	.01	6	.58	17	29.56	563	1.3	5	47.91	5
4	.02	7	1.20	25	m	–	3.29	7	m	–
5	.01	13	1.53	34	m	–	11.66	10	m	–
6	.01	16	1.68	38	m	–	28.87	12	m	–
7	.02	17	2.00	45	m	–	82.57	15	m	–
8	.02	17	2.13	49	m	–	35.69	17	m	–
9	.03	21	2.50	52	m	–	13.37	20	m	–
10	.07	41	7.18	91	m	–	126.25	35	m	–
11	.09	46	8.66	101	m	–	m	–	m	–
12	.10	49	10.06	113	m	–	m	–	m	–
13	.28	67	19.89	131	m	–	m	–	m	–
14	2.45	74	28.28	236	m	–	m	–	m	–
15	4.54	115	43.07	300	m	–	m	–	m	–

Table 2: Our approach compared to search control with Büchi automata

tomaton that is constructed for the goal. Then, it uses the automaton to guide the planning by following a path in its graph from an initial state to a final state. If there is more than one path, the search has to choose between one of the paths and possibly backtrack to another path. In combination with this, it uses search control information to make planning more efficient. However, since no heuristics are employed, though it follows a path to accepting states in the automaton, it can easily "get lost" in the absence of good control information. Furthermore, the fact that it needs a single automaton for the TEG makes it more vulnerable to exponential blowup. Finally, in contrast to our approach, TPBA is able to handle infinite plans, including cyclic plans.

Approaches for planning as symbolic model checking have also used automata to encode the goals (e.g Pistore et al., 2001; dal Lago et al., 2002). These approaches use different languages for extended goals, and are not heuristic. Rather, they associate a set of domain states to each of the states of the automaton. They start by associating the whole set of domain states to each state of the automaton and then iteratively refine them until a fixed point is reached. Then they are able to extract the plan from the automaton.

Cresswell and Coddington (2004) propose a translation of LTL formulae into PDDL. They translate LTL formulae to a deterministic finite state machine (FSM), and then they translate the FSM into an ADL-only domain. The FSM is generated by successive applications of the *progress* operator of Bacchus and Kabanza (1998)

to the TEG. Since they use pure LTL, their finite state machine does not have accepting states. The accepting condition, then, must be determined by simulating an infinite repetition of the last state. The use of deterministic automata makes it very prone to exponential blowup with very simple goals; e.g., $\Diamond(p \wedge \bigcirc^n q)$ produces an FSM that is exponential in n. The authors' code was unavailable for comparison with our work. Nevertheless, they report that their technique is no more efficient than TLPLAN.

Finally, Rintanen (2000) proposes a translation of a subset of LTL into a set of ADL operators. Their translation does not use automata. As a consequence, their approach is restricted to TEGs of the form $\kappa \vee p \,\mathsf{U}\, q$, $\kappa \vee p \,\mathsf{R}\, q$, and $\kappa \vee \bigcirc p$, where p and q are propositional literals and κ is a disjunction of literals.

7 Discussion

In this paper we proposed a method to generate plans with TEGs using heuristic search. To this end, we proposed two translation methods that take as input a planning problem with a TEG and produce a classical planning problem. The first translation produces domains described only by ADL operators, the second uses derived predicates. We have implemented both translation methods. Our implementation outputs PDDL problem descriptions, which enables us to use a wide variety of planners and planning systems. Experimental results with the FF and $\text{FF}_{\mathcal{X}}$ planners show excellent performance compared to existing (non-heuristic) planners for TEGs.

To represent TEGs we used f-LTL, a finite version of LTL. This logic is particularly well-suited to represent properties of finite plans. Previous approaches have used LTL, which can only be interpreted for infinite sequences of states. To interpret a formula over a finite plan, these approaches must interpret a sequence of states $s_0 s_1 \cdots s_{n-1} s_n$ as $s_0 s_1 \cdots s_{n-1} s_n s_n \cdots$. However, for certain LTL formulae, this interpretation may produce unnatural plans.

References

Bacchus, F., & Ady, M. (1999). Precondition control. Unpublished manuscript.

Bacchus, F., & Kabanza, F. (1998). Planning for temporally extended goals.. *Annals Mathematics Artificial Intelligence*, *22*(1-2), 5–27.

Cresswell, S., & Coddington, A. M. (2004). Compilation of LTL goal formulas into PDDL. In de Mántaras, R. L., & Saitta, L. (Eds.), *Proceedings of the*

16th European Conference on Artificial Intelligence (ECAI-04), pp. 985–986, Valencia, Spain. IOS Press.

dal Lago, U., Pistore, M., & Traverso, P. (2002). Planning with a language for extended goals. In *Proceedings of AAAI/IAAI*, pp. 447–454, Edmonton, Alberta, Canada.

Daniele, M., Giunchiglia, F., & Vardi, M. Y. (1999). Improved automata generation for linear temporal logic.. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV-99)*, Vol. 1633 of *LNCS*, pp. 249–260, Trento, Italy. Springer.

de Giacomo, G., & Vardi, M. Y. (1999). Automata-theoretic approach to planning for temporally extended goals. In Biundo, S., & Fox, M. (Eds.), *ECP*, Vol. 1809 of *LNCS*, pp. 226–238, Durham, UK. Springer.

Etessami, K., & Holzmann, G. J. (2000). Optimizing Büchi automata.. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR-00)*, Vol. 1877 of *LNCS*, pp. 153–167, University Park, PA. Springer.

Fritz, C. (2003). Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata.. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA-03)*, Vol. 2759 of *LNCS*, pp. 35–48, Santa Barbara, CA. Springer.

Gerth, R., Peled, D., Vardi, M. Y., & Wolper, P. (1995). Simple on-the-fly automatic verification of linear temporal logic.. In *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV-95)*, pp. 3–18, Warsaw, Poland.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, *14*, 253–302.

Kabanza, F., & Thiébaux, S. (2005). Search control in planning for temporally extended goals. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, pp. 130–139.

Kvarnström, J., & Doherty, P. (2000). Talplanner: A temporal logic based forward chaining planner. *Annals of Mathematics Artificial Intelligence*, *30*(1-4), 119–169.

McCarthy, J., & Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press.

McDermott, D. V. (1998). PDDL — The Planning Domain Definition Language. Tech. rep. TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Pednault, E. P. D. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference of Knowledge Representation and Reasoning (KR-89)*, pp. 324–332, Toronto, Canada.

Pistore, M., Bettin, R., & Traverso, P. (2001). Symbolic techniques for planning with extended goals in non-deterministic domains. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, Toledo, Spain.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS-77)*, pp. 46–57.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA.

Rintanen, J. (2000). Incorporation of temporal logic control into plan operators.. In Horn, W. (Ed.), *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00)*, pp. 526–530, Berlin, Germany. IOS Press.

Thiébaux, S., Hoffmann, J., & Nebel, B. (2005). In defense of PDDL axioms. *Artificial Intelligence*, *168*(1-2), 38–69.

Vardi, M. Y. (1995). An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, Vol. 1043 of *LNCS*, pp. 238–266. Springer.

Vardi, M. Y., & Wolper, P. (1994). Reasoning about infinite computations. *Information and Computation*, *115*(1), 1–37.

Waldinger, R. (1977). Achieving several goals simultaneously. In *Machine Intelligence 8*, pp. 94–136. Ellis Horwood, Edinburgh, Scotland.

Appendix

Definition 4 ($Old^{-}(q)$) *We define $Old^{-}(q)$ as the set containing all the literals in $Old(q)$.*

Proof for Theorem 1 It follows directly from Lemmas 6 and 7 below.

In the following proofs, if X is a set of temporal formulae, we denote by $\bigwedge X$ the conjuction the elements in X.

Lemma 1 (Analogous to Lemma 4.1 in (Gerth et al., 1995)) *If nodes q_1 and q_2 are split from a node q (in lines 35-48), then the following property holds.*

$$\begin{aligned}(\bigwedge Old^{-}(q) \wedge \bigwedge New(q) \wedge \beta(q)) \equiv \\ (\bigwedge Old^{-}(q_1) \wedge \bigwedge New(q_1) \wedge \beta(q_1)) \vee \\ (\bigwedge Old^{-}(q_2) \wedge \bigwedge New(q_2) \wedge \beta(q_2))\end{aligned} \tag{4}$$

where $\beta(q) = \bigcirc \bigwedge Next(q)$ if $Next(q) \neq \emptyset$ and $True$ otherwise. Similarly, when node q is updated to become a new node q' (in lines 23-34), the following holds

$$\begin{aligned}(\bigwedge Old^{-}(q) \wedge \bigwedge New(q) \wedge \beta(q)) \equiv \\ (\bigwedge Old^{-}(q') \wedge \bigwedge New(q') \wedge \beta(q'))\end{aligned} \tag{5}$$

Proof: Straightforward from properties of f-LTL. □

Following the proof by Gerth et al. (1995), we define an ancestor relation between nodes R, such that $(p,q) \in R$ iff $Father(q) = Name(p)$. We deonete by R^* the transitive closure of R. Furthermore, we say a node is *rooted* if it has no ancestors; i.e., $Father(q) = Name(q)$.

Lemma 2 (Analogous to Lemma 4.2 by Gerth et al. (1995)) *Let p be a rooted node, and $q_1, \ldots, q_n$ be such that $(p, q_i) \in R^*$. Then, the following holds:*

$$\bigwedge New(p) \equiv \bigvee_{1 \leq i \leq n} (\bigwedge \Delta^{-}(q_i) \wedge \beta(q_i)).$$

Proof: By induction in the construction using Lemma 1. □

Lemma 3 *Let graph G_φ be generated by the algorithm for formula φ. Let $R = \{r_1, \ldots, r_n\}$ be the successors of node p. Moreover let $\psi = \bigwedge Next(p)$. Then the graph that results by invoking the algorithm with ψ, G_ψ, is isomorphic with the graph that results from removing every state from G_φ that is unreachable any node in R. Furthermore, the isomorphism is such that if it maps q to q', then $\Delta(q) = \Delta(q')$.*

Proof: Since nodes in R are direct successors of p, we know that there exists a common ancestor r' in G of all nodes in R such that, at the beginning of its construction, $New(r')$ is equal to $Next(p)$.

Likewise, when invoking the algorithm with ψ, the starting node, say r'', will contain its New field equal to ψ. Without loss of generality, lets choose an execution of the algorithm in which the conjunction $\psi = \bigwedge Next(p)$ is broken such that eventually $New = Next(p)$.

We start mapping node r' to r''. Each time we split a node into two nodes, we look at G_φ and map the two successors accordingly. We repeat this process recursively.

The resulting mapping is effectively an isomorphism, since graph G_ψ is constructed using the same procedure as the subgraph of G_φ rooted in r'. □

Lemma 4 *If there is an accepting run $\rho = q_0 q_1 \cdots q_n$ for σ in A_φ, and $Next(q_0) = \psi$, then there is an accepting run for σ_1 in A_ψ.*

Proof: Since q_1 is successor of q_0, the proof results directly from Lemma 3. □

Lemma 5 *If $\rho = q_0 q_1 \cdots q_n$ is an accepting run for $\sigma = s_0 s_1 \cdots s_n$, then $\sigma_0 \models \Delta^-(q_0)$.*

Proof: Since ρ is a run, $s_0 \models \bigwedge \Delta^-(q_0) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$. Since s_0 does not contain temporal formulae, $\sigma_0 \models \bigwedge \Delta^-(q_0) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$. Now we have two cases.

- $n = 0$. Since q_0 is final, we know $\neg\mathsf{final} \notin \Delta^-(q_0)$ (by definition of final state). Now, whether or not $\mathsf{final} \in \Delta^-(q_0)$, $\sigma_0 \models \bigwedge \Delta^-(q_0)$.
- $n > 0$. Since q_0 has a successor, then $\mathsf{final} \notin \Delta^-(q_0)$ (see condition in line 9 in the algorithm). Now, whether or not $\neg\mathsf{final} \in \Delta^-(q_0)$, $\sigma_0 \models \bigwedge \Delta^-(q_0)$. □

Lemma 6 *If there is an accepting run for σ in A_φ, then $\sigma \models \varphi$.*

Proof: By induction in the length of σ.

- Base case ($\sigma = s_0$). Then, there is an initial state q in A_φ which is also final. By Lemma 5, $\sigma \models \bigwedge \Delta^-(q)$. Since q is final, we have that $Next(q) = \emptyset$. From Lemma 2, we conclude immediately that $\sigma \models \varphi$.
- Induction. Suppose $\rho = q_0 q_1 \ldots q_k$ is an accepting run for σ in A_φ. Then, by Lemma 5, $\sigma_0 \models \Delta^-(q_0)$. Moreover, from Lemma 4, there is an accepting run for σ_1 in A_ψ, where $\psi = Next(q_0)$.

  By inductive hypothesis, $\sigma_1 \models Next(q_0)$. Therefore, by f-LTL equivalence, we have that $\sigma \models \Delta^-(q_0) \wedge \bigcirc Next(q_0)$. Then, by Lemma 2, we conclude immediately that $\sigma \models \varphi$. □

Lemma 7 *If $\sigma \models \varphi$, then there is an accepting run for σ in A_φ.*

Proof: By induction in the length of σ.

- Base case ($|\sigma| = 1$). By Lemma 2, we have that

$$\sigma \models \bigwedge \Delta^{-}(q) \wedge \beta(q), \tag{6}$$

for some initial state q.

We can conclude the following.

1. $Next(q) = \emptyset$, i.e, $\beta(q) \equiv \mathsf{true}$. Otherwise, we would have that $\sigma \models \bigcirc\xi$, for some ξ, which is not possible if $|\sigma| = 1$.
2. $\sigma \models \bigwedge \Delta^{-}(q)$, which is implied by the condition above and (6).
3. $\neg\mathsf{final} \notin \Delta^{-}(q)$, because $|\sigma| = 1$.

From 1. and 3. we conclude that q is a final state. From 2., we conclude that $s_0 \models \bigwedge \Delta^{-}(q) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$. Hence σ has an accepting run in A_φ.

- Induction. By Lemma 2, we conclude that $\sigma \models \bigwedge \Delta^{-}(q) \wedge \beta(q)$ for some initial state q. We have two cases.

  – $Next(q) = \emptyset$. This implies that $\sigma_0 \models \bigwedge \Delta^{-}(q)$. As before, this means that $s_0 \models \bigwedge \Delta^{-}(q)$, because $\bigwedge \Delta^{-}(q)$ is atemporal. Therefore q can be the initial state of a run.

  Furthermore, since $|\sigma| > 0$, $\sigma \not\models \mathsf{final}$, and therefore $\mathsf{final} \notin \Delta^{-}(q)$. This implies $s_0 \models \Delta^{-}(q) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$.

  Moreover, the algorithm will construct a successor to state q. Let's denote this successor by q'. The following holds true:

  * $\Delta^{-}(q') = \emptyset$, because $Next(q) = \emptyset$.
  * State q' is final (by definition of final state).
  * State q' has a transition to itself. The algorithm constructs this transition in line 6.

  All this means that σ has an accepting run in A_φ, in fact such run is $q(q')^n$.

  – $Next(q) \neq \emptyset$. Again, we have $s_0 \models \Delta^{-}(q) - \{\mathsf{final}, \neg\mathsf{final}\}$. Since, as before, $\sigma \not\models \mathsf{final}$, we have a transition from q to a state, say q'. State q' was initially invoked with $New(q') = Next(q)$, so, by Lemma 3 and the induction hypothesis, we have that any run for σ_1 from q' has an accepting run in A_ψ, with $\psi = \bigwedge Next(q')$. Since any path in A_ψ has an isomorphic path in A_φ, then σ has a run in A_φ. □