Computing Infinite Plans for LTL Goals Using a Classical Planner

Fabio Patrizi
Imperial College London
London, UK
fpatrizi@imperial.ac.uk

Nir Lipoveztky
Univ. Pompeu Fabra
Barcelona, Spain
first.last@upf.edu

Giuseppe De Giacomo
Sapienza Univ. di Roma
Rome, Italy
degiacomo@dis.uniroma1.it

Hector Geffner
ICREA & UPF
Barcelona, Spain
first.last@upf.edu

Abstract

Classical planning has been notably successful in synthesizing finite plans to achieve states where propositional goals hold. In the last few years, classical planning has also been extended to incorporate temporally extended goals, expressed in temporal logics such as LTL, to impose restrictions on the state sequences generated by finite plans. In this work, we take the next step and consider the computation of *infinite plans* for achieving *arbitrary* LTL *goals*. We show that infinite plans can also be obtained efficiently by calling a classical planner once over a classical planning encoding that represents and extends the composition of the planning domain and the Büchi automaton representing the goal. This compilation scheme has been implemented and a number of experiments are reported.

1 Motivation

Classical planning has been concerned with the synthesis of finite plans to achieve final states where given propositional goals hold. These are usually called "reachability" problems. In the last few years temporally extended goals, expressed in temporal logics such as LTL, have been increasingly used to capture a richer class of finite plans, where restrictions over the whole sequence of states must be satisfied as well [Gerevini and Long, 2005]. A (temporally) extended goal may state, for example, that any borrowed tool should be kept clean until returning it; a constraint that does not apply to states but, rather, to state sequences. Yet almost all work in planning for LTL goals has been focused on finite plans [Bacchus and Kabanza, 1998; Cresswell and Coddington, 2004; Edelkamp, 2006; Baier and McIlraith, 2006; Baier *et al.*, 2009], while general LTL goals may require infinite plans (see [Bauer and Haslum, 2010]). For instance, in order to monitor a set of rooms, an extended LTL goal may require the agent to always return to each of the rooms, a goal that cannot be achieved by a finite plan.

In this work, we take the next step in the integration of LTL goals in planning and consider the computation of *infinite plans* for achieving *arbitrary* LTL *goals*. It is well known that such infinite plans can be finitely characterized as "lassos": sequences of actions π_1, mapping the initial state of a composite system into some state s, followed by a second action sequence π_2 that maps s into itself, and that is repeated infinitely often [Vardi, 1996]. The composite system is the product of the planning domain and the Büchi automaton representing the goal [De Giacomo and Vardi, 1999]. In this paper we show that such infinite plans can efficiently be constructed by calling a *classical planner* once over a *classical planning problem* P_φ, which is obtained from the PDDL description P of the planning domain, and the Büchi automaton A_φ representing the goal φ.

The crux of our technique is a quite natural observation: since we are looking for lasso sequences, when we reach an accepting state of the Büchi automaton, we can nondeterministically elect the current configuration formed by the state of the automaton and the state of the domain as a "start looping" configuration, and then try to reach the exact same configuration a second time. If we do, we have found an accepting automaton state that repeats infinitely often, satisfying the Büchi condition, i.e., we have found the lasso. In this way we reduce *fair reachability* (the lassos sequences) to plain *reachability* (finite sequences). Such an observation has been made already in the model-checking literature. In particular [Schuppan and Biere, 2004] use this observation to reduce checking of liveness properties ("something good eventually happens"), and, more generally, arbitrary LTL formulas via Büchi automata nonemptiness, to checking of safety properties ("something bad never happens").

Planning technologies have been used before for tackling LTL goals, starting with the pioneer work by Edelkamp [2003]. Also, an earlier computational model for planning with arbitrary LTL goals was developed in [Kabanza and Thiébaux, 2005], where no direct translation into classical planning was present, but a classical planner was invoked to solve a series of subproblems, inside a backtracking search. Strictly related to our approach is the work reported in Albarghouthi, Baier, and McIlraith [2009], where the authors map the model-checking problem over deterministic and nondeterministic transition systems into classical planning problems. They directly exploit the reduction schema devised in [Schuppan and Biere, 2004] to handle the Büchi acceptance condition with the generality required by arbitrary LTL formulas, while adopting specific techniques for safety and liveness properties, demonstrated by promising experiments over the Philosophers domain.

Here we propose instead a direct translation of LTL goals (or better arbitrary Büchi automata goals) into classical planning specifically well cut to exploit state-of-the art planners capabilities, and test it over a variety of domains and goals.

The paper is organized as follows. First, we review the background material: planning domains, LTL, and Büchi automata (Section 2), and the definition of the problem of achieving arbitrary LTL goals φ over planning domains P (Section 3). We then map this problem into the classical planning problem P_φ (Section 4) and test the compilation over various domains and goals (Section 5).

2 Preliminaries

We review the models associated with classical planning, LTL, and Büchi automata.

2.1 Planning Domains

A (classical) *planning domain* is a tuple $\mathcal{D} = (Act, Prop, S, s_0, f)$ where: *(i)* Act is the finite set of domain actions; *(ii)* $Prop$ is the set of domain propositions; *(iii)* $S \subseteq 2^{Prop}$ is the set of domain states; *(iv)* $s_0 \in S$ is the initial state of the domain; and *(v)* $f : A \times S \to S$ is a (partial) state transition function.

Planning languages such as STRIPS or ADL, all accommodated in the PDDL standard, are commonly used to specify the states and transitions in compact form.

A *trace* on a planning domain is a possibly infinite sequence of states $s_0, s_1, s_2, \ldots$ where $s_{i+1} = f(s_i, a)$ for some $a \in Act$ s.t. $f(s_i, a) \neq \bot$. A *goal* is a specification of the desired traces on $\mathcal{D}$. In particular, classical reachability goals, which require reaching a state s where a certain propositional formula φ over $Prop$ holds, are expressed as selecting all those *finite* traces $t = s_0 s_1 \cdots s_n$, such that $s_n \models \varphi$. Using *infinite traces* allows us to consider a richer set of goals, suitably expressed through arbitrary LTL formulas.

2.2 Linear Temporal Logic (LTL)

LTL was originally proposed as a specification language for concurrent programs [Pnueli, 1977]. *Formulas* of LTL are built from a set $Prop$ of propositional symbols and are closed under the boolean operators, the unary temporal operators $\bigcirc$, $\Diamond$, and $\Box$, and the binary temporal operator $\mathcal{U}$.[1] Intuitively, $\bigcirc\varphi$ says that φ holds at the *next* instant, $\Diamond\varphi$ says that φ will *eventually* hold at some future instant, $\Box\varphi$ says that from the current instant on φ will *always* hold, and $\varphi\,\mathcal{U}\,\psi$ says that at some future instant ψ will hold and *until* that point φ holds. We also use the standard boolean connectives $\vee$, $\wedge$, and $\rightarrow$.

The semantics of LTL is given in terms of interpretations over a *linear structure*. For simplicity, we use $\mathbb{N}$ as the linear structure: for an instant $i \in \mathbb{N}$, the successive instant is $i+1$. An *interpretation* is a function $\pi : \mathbb{N} \to 2^{Prop}$ assigning to each element of $Prop$ a truth value at each instant $i \in \mathbb{N}$. For an interpretation π, we inductively define when an LTL formula φ *is true* at an instant $i \in \mathbb{N}$ (written $\pi, i \models \varphi$):

- $\pi, i \models p$, for $p \in Prop$ iff $p \in \pi(i)$.
- $\pi, i \models \neg\varphi$ iff not $\pi, i \models \varphi$.
- $\pi, i \models \varphi \wedge \varphi'$ iff $\pi, i \models \varphi$ and $\pi, i \models \varphi'$.
- $\pi, i \models \bigcirc\varphi$ iff $\pi, i{+}1 \models \varphi$.
- $\pi, i \models \varphi\,\mathcal{U}\,\varphi'$ iff for some $j \geq i$, we have that $\pi, j \models \varphi'$ and for all k, $i \leq k < j$, we have that $\pi, k \models \varphi$.

A formula φ *is true* in π (written $\pi \models \varphi$) if $\pi, 0 \models \varphi$. Given a planning domain (or more generally a transition system), its traces $s_0, s_1, s_2, \ldots$ can be seen as LTL interpretations π such that $\pi, i \models p$ iff $s_i \models p$.

2.3 LTL and Büchi Automata

There is a tight relation between LTL and Büchi automata on infinite words, see e.g., [Vardi, 1996]. A *Büchi automaton (on infinite words)* [Thomas, 1990] is a tuple $A = (\Sigma, Q, Q_0, \rho, F)$ where: *(i)* Σ is the input alphabet of the automaton; *(ii)* Q is the finite set of automaton states; *(iii)* $Q_0 \subseteq Q$ is the set of initial states of the automaton; *(iv)* $\rho : Q \times \Sigma \to 2^Q$ is the automaton transition function (the automaton does not need to be deterministic); and *(v)* $F \subseteq Q$ is the set of accepting states. The *input words* of A are infinite words $\sigma_0\sigma_1 \cdots \in \Sigma^\omega$. A *run* of A on an infinite word $\sigma_0\sigma_1 \cdots$ is an infinite sequence of states $q_0 q_1 \cdots \in Q^\omega$ s.t. $q_0 \in Q_0$ and $q_{i+1} \in \rho(q_i, \sigma_i)$. A run r is *accepting* iff $lim(r) \cap F \neq \emptyset$, where $lim(r)$ is the set of states that occur in r infinitely often. In other words, a run is accepting if it gets into F infinitely many times, which means, being F finite, that there is at least one state $q_f \in F$ visited infinitely often. The *language* accepted by A, denoted by $L(A)$, is the set of (infinite) words for which there is an accepting run.

The *nonemptiness* problem for an automaton A is to decide whether $L(A) \neq \emptyset$, i.e., whether the automaton accepts at least one word. The problem is NLOGSPACE-complete [Vardi and Wolper, 1994], and the nonemptiness algorithm in [Vardi and Wolper, 1994] actually returns *a witness for nonemptiness, which is a finite prefix followed by a cycle.*

The relevance of the nonemptiness problem for LTL follows from the correspondence obtained by setting the automaton alphabet to the propositional interpretations, i.e., $\Sigma = 2^{Prop}$. Then, an infinite word over the alphabet 2^{Prop} represents an interpretation of an LTL formula over $Prop$.

Theorem 1 [Vardi and Wolper, 1994] *For every* LTL *formula φ one can effectively construct a Büchi automaton A_φ whose number of states is at most exponential in the length of φ and such that $L(A_\varphi)$ is the set of models of φ.*

Typically, formulas are used to compactly represent subsets of $\Sigma = 2^{Prop}$. We extend the transition function of a Büchi automaton to propositional formulas over $Prop$ as: $\rho(q, W) \doteq \{q' \mid \exists s \; s.t. \; s \models W \wedge q' \in \rho(q, s)\}$.

3 The Problem

A *plan* π over a planning domain $\mathcal{D} = (Act, Prop, S, s_0, f)$ is an infinite sequence of actions $a_0, a_1, a_2, \ldots \in Act^\omega$. The *trace* of π (starting from the initial state s_0) is the infinite sequence of states $tr(\pi, s_0) = s_0, s_1, \ldots \in S^\omega$ s.t. $s_{i+1} = f(s_i, a_i)$ (and hence $f(s_i, a) \neq \bot$). A plan π *achieves* an LTL formula φ iff $tr(\pi, s_0) \in L(A_\varphi)$, where $A_\varphi = (2^{Prop}, Q, Q_0, \rho, F)$ is the automaton that accepts exactly the interpretations that satisfy φ.

[1] In fact, all operators can be defined in terms of $\bigcirc$ and $\mathcal{U}$.

How can we synthesize such a plan? We can check for nonemptiness the Büchi automaton $A_{\mathcal{D},\varphi} = (\Sigma_{\mathcal{D}}, Q_{\mathcal{D}}, Q_{\mathcal{D}0}, \rho_{\mathcal{D}}, F_{\mathcal{D}})$ that represents the product between the domain $\mathcal{D}$ and the automaton A_φ, where: *(i)* $\Sigma_{\mathcal{D}} = Act$; *(ii)* $Q_{\mathcal{D}} = Q \times S$; *(iii)* $Q_{\mathcal{D}0} = Q_0 \times \{s_0\}$; *(iv)* $(q_j, s_j) \in \rho_{\mathcal{D}}((q_i, s_i), a)$ iff $s_j = f(s_i, a)$ and $q_j \in \rho(q_i, W)$, with $s_i \models W$; and *(v)* $F_{\mathcal{D}} = F \times S$. It can be shown that the above construction is sound and complete:

Theorem 2 [De Giacomo and Vardi, 1999] *A plan π for the planning domain $\mathcal{D}$ achieves the LTL goal φ iff $\pi \in L(A_{\mathcal{D},\varphi})$ for the automaton $A_{\mathcal{D},\varphi}$.*

It is also easy to see that if a plan π is accepted by the Büchi automaton $A_{\mathcal{D},\varphi}$, and hence π achieves the LTL goal φ over $\mathcal{D}$, then π can be seen as forming a lasso, namely: an action sequence π_1 followed by a loop involving an action sequence π_2. This is because π must generate a run over the automaton $A_{\mathcal{D},\varphi}$ that includes some accepting state (q_i, s_i) an infinite number of times. It follows from this that:

Theorem 3 *The goal φ is achievable in a planning domain $\mathcal{D}$ iff there is a plan π made up of an action sequence π_1 followed by the action sequence π_2 repeated an infinite number of times, such that π achieves φ in $\mathcal{D}$.*

4 Compilation Into Classical Planning

Theorem 3 says that the plans to achieve an arbitrary LTL goal have all the same form: a sequence π_1 mapping the initial state of the product automaton $A_{\mathcal{D},\varphi}$ into an accepting state, followed by another sequence π_2 that maps this state into itself, that is repeated for ever. This observation is a direct consequence of well known results. What we want to do now is to take advantage of the factored representation P of the planning domain $\mathcal{D}$ afforded by standard *planning languages*, for transforming the problem of finding the sequences π_1 and π_2 for an arbitrary LTL goal φ, into the problem of finding a standard *finite plan* for a *classical problem* P_φ, where P_φ is obtained from P and the automaton A_φ (that accepts the interpretations that satisfy φ). Such classical plans, that can be obtained using an off-the-shelf classical planner, will all have the form $\pi'_1, loop(q), \pi'_2$, where π'_1 and π'_2 are the action sequences π_1 and π_2 extended with auxiliary actions, and $loop(q)$ is an auxiliary action to be executed exactly once in any plan for P_φ, with q representing an accepting state of A_φ. The $loop(q)$ action marks the current state over the problem P_φ, as the first state of the lasso. This is accomplished by making the $loop(q)$ action dynamically set the goal of the problem P_φ to the pair (q, s) (extended with a suitable boolean flag) if s represents the state of the literals over $Prop$ when the $loop(q)$ was done. That is, the action sequence π'_2 that follows the $loop(q)$ action, starts with the fluents encoding the state (q, s) true, and ends when these fluents have been true once again, thus capturing the loop.

The basis of the classical planning problem P_φ is the intermediate description P', an encoding that captures simple *reachability* in the product automaton $A_{\mathcal{D}}$. If $P = \langle Prop, s_0, Act\rangle$ is the PDDL description of the planning domain, and $A_\varphi = \langle 2^{Prop}, Q, Q_0, \rho, F\rangle$ is the Büchi automaton accepting the interpretations that satisfy φ, then P' is the tuple $\langle Prop', s'_0, Act'\rangle$ where:

- $Prop' = Prop \cup \{p_q, n_q \mid q \in Q\} \cup \{f_0, f_1, f_2\}$,
- $s'_0 = s_0 \cup \{p_q \mid q \in Q_0\} \cup \{f_1\}$,
- $Act' = Act \cup \{mv_1, mv_2\}$,

where the actions in Act' that come from P, i.e. those in Act, have the literal f_0 as an extra precondition, and the literals $\neg f_0$ and f_1 as extra effects. The booleans f_i are flags that force infinite plans $a_0, a_1, a_2, \ldots$ in P' to be s.t. a_0 is an action from P, and if a_i is an action from P, $a_{i+1} = mv_1$, $a_{i+2} = mv_2$, and a_{i+3} is an action from P again. That is, plans for P' are made of sequences of three actions, the first from P, followed by mv_1 and mv_2. For this, mv_1 has precondition f_1 and effects f_2 and $\neg f_1$, and mv_2 has precondition f_2 and effects f_0 and $\neg f_2$.

The actions mv_1 and mv_2 keep track of the fluents p_q that encode the states q of the automaton A_φ. Basically, if state q' may follow q upon input formula W in A_φ, then action mv_1 will have the conditional effects

$$W \wedge p_q \;\rightarrow\; n_{q'} \wedge \neg p_q$$

and mv_2 will have the conditional effects

$$n_q \;\rightarrow\; p_q \wedge \neg n_q$$

for all the states q in A_φ. So that if p_q and W are true right before mv_1, then $p_{q'}$ will be true after the sequence mv_1, mv_2 iff $q' \in \rho(q, W)$ for the transition function ρ of A_φ. It can be shown then that:

Theorem 4 *Let $P = \langle Prop, s_0, Act\rangle$ be the PDDL description of the planning domain $\mathcal{D}$, and $A_\varphi = \langle 2^{Prop}, Q, Q_0, \rho, F\rangle$ be the Büchi automaton accepting the interpretations that satisfy φ. The sequence $\pi = a_0, a_1, a_2, \ldots, a_{i*3+2}$ non-deterministically leads the product automaton $A_{\mathcal{D},\varphi}$ to the state (q, s) iff in the planning domain description P', π achieves the literal p_q and the literals L over $Prop$ iff L is true in s.*

P' thus captures simple reachability in the automaton $A_{\mathcal{D}}$ that is the product of the planning domain described by P and the automaton A_φ representing the goal φ. The *classical planning problem* P_φ that captures the plans for φ over P is defined as an extension of P'. The extension enforces a correspondence between the 'loopy' plans π for φ over P of the form 'π_1 followed by loop π_2', and the finite plans for the classical problem P_φ of the form '$\pi'_1, loop(q), \pi'_2$', where π_1 and π_2 are the action sequences before and after the $loop(q)$ action with the auxiliary actions removed. The encoding P_φ achieves this correspondence by including in the goal the literal p_q encoding the state q of the A_φ as well as all the literals L over $Prop$ that were true when the action $loop(q)$ was done. This is accomplished by making a copy of the latter literals in the atoms $req(L)$. More precisely, if $P = \langle Prop, s_0, Act\rangle$ and $P' = \langle Prop', s'_0, Act'\rangle$, P_φ is the tuple $P'' = \langle Prop'', s''_0, Act'', Goal''\rangle$ where:

- $Prop'' = Prop' \cup \{req(L) \mid L \in Prop\} \cup \{Ls, Lf\}$
- $s''_0 = s'_0$
- $Act'' = Act' \cup \{loop(q) \mid q \in F\}$
- $G'' = \{Lf\} \cup \{L \equiv req(L) \mid L \in Prop\}$.

Here $L \in Prop$ refers to the literals defined over the $Prop$ variables, and the new fluents $req(L)$, Ls, and Lf stand for 'L required to be true at the end of the loop', 'loop started', and 'loop possibly finished' respectively. In addition, the new $loop(q)$ actions have preconditions p_q, f_0, $\neg Ls$, and effects Ls and

$$L \rightarrow req(L)$$

for all literals L over $Prop$, along with the effects $p_q \rightarrow \neg p_{q'}$ for all the automaton states q' different than q. The effects $L \rightarrow req(L)$ 'copy' the literals L that are true when the action $loop(q)$ was done, into the atoms $req(L)$ that cannot be changed again. As a result, the goals $L \equiv req(L)$ in G'' capture the equivalence between the truth value of L when the $loop(q)$ action was done, and when the goal state of P_φ is achieved.

The effects $p_q \rightarrow \neg p_{q'}$, on the other hand, express a commitment to the automaton state q associated with the $loop(q)$ action, setting the fluents representing all other states q' to false. In addition, all the non-auxiliary actions in Act'', namely those from P, are extended with the effect $Ls \rightarrow Lf$ that along with the goal Lf ensures that some action from P must be done as part of the loop. Without the Lf fluent ('loop possibly finished') in the goal and these conditional effects, the plans for P_φ would finish right after the $loop(q)$ action without capturing a true loop.

From the goal G'' above that includes both Lf and

$$L \equiv req(L)$$

for all literals L over $Prop$, this all means that a $loop(q)$ action must be done in any plan for P_φ, after an initial action sequence π'_1, and before a second action sequence π'_2 containing an action from Act. The sequence π'_2 closes the 'lasso'; namely, it reproduces the state of the product automaton where the action $loop(q)$ was done.[2]

Theorem 5 (Main) *π is a plan for the* LTL *goal φ over the planning domain described by P iff π is of the form 'π_1 followed by the loop π_2', where π_1 and π_2 are the action sequences from P, before and after the $loop(q)$ action in any* classical plan *for P_φ.*

5 Use of the Classical Planner

Theorem 5 states that the plans for an arbitrary LTL goal φ over a domain description P can be obtained from the plans for the *classical planning* problem P_φ. The goal of P_φ is a classical goal that includes the literal Lf and the equivalences $L \equiv req(L)$ for $L \in Prop$. Classical planners usually deal with precondition, conditions, and goals that are conjunctions of literals, eliminating other formulas. For this, they apply standard transformations as a preprocessing step [Gazen and Knoblock, 1997]. In our use of planners, we have found useful to compile the equivalences $L \equiv req(L)$ away from the goal by including extra actions and fluents. In particular, a new action $End?$ is introduced that can be applied at most once as the last action of a plan (this is managed by an extra boolean flag). The precondition of $End?$ is Lf and its effects are

$$L, req(L) \rightarrow end(L)$$

over all L over $Prop$, where $end(L)$ are new atoms. It is easy to see that π is a classical plan for the original encoding P_φ iff π followed by the $End?$ action is a classical plan in the revised encoding where the equivalences $L \equiv req(L)$ in the goal have been replaced by the atoms $end(L)$. This transformation is general and planner independent.

The second transformation that we have found useful to improve performance involves *changes in the planner itself*. We made three changes in the state-of-the-art FF planner [Hoffmann and Nebel, 2001] so that the sequences made up of a normal domain action followed by the auxiliary actions mv_1 and mv_2, that are part of all plans for the compiled problems P_φ, are executed as if the 3-action sequence was just one "primitive" action. For this, every time a normal action a is applied in the search, the whole sequence a, mv_1, mv_2 is applied instead. In addition, the two auxiliary actions mv_1 and mv_2 that are used to capture the ramifications of the normal actions over the Büchi automata, are not counted in the evaluation of the heuristic (that counts the number of actions in the relaxed plans), and the precondition flag f_1 of the action mv_1 appearing in the relaxed plans is not taken into account in the identification of the "helpful actions", as all the actions applicable when f_1 is false and f_0 is true, add f_1 Finally, we have found critical to disable the goal agenda mechanism, as the compiled problems contain too many goals: as many as literals. Without these changes FF runs much slower over the compiled problems. In principle, these problems could be avoided with planners able to deal properly with "action macros" or "ramifications", but we have found such planners to be less robust than FF.

6 Experiments

Let us describe through a sample domain what LTL goals can actually capture. In this domain, a robotic 'animat' lives on a $n \times n$ grid, whose cells may host a food station, a drink station, the animat's lair, and the animat's (beloved) partner. In our instances the partner is at the lair. The animat status is described in terms of levels of power (p), hunger (h), and thirst (t). The animat can $move$ one cell up, down, right, and left, can $drink$ (resp. eat), when in a drink (food) station, and can $sleep$, when at the lair. Each action affects (p, h, t), as follows: $move$:$(-1, +1, +1)$, $drink$:$(-1, +1, 0)$, eat:$(-1, 0, +1)$, and $sleep$:$(max, +1, +1)$. The value max is a parameter adjusted depending on the grid size n. Initially, $(p, h, t) = (max, 0, 0)$.

The objective of the animat is not to reach a particular goal as in classical planning but to carry on a happy life. The animat is happy if it is not (too) hungry, thirsty or weak, and, importantly, if it can get back to its lair and see its partner, every now and then, and do something different as well. Its life is happy if this condition is always verified. Formally, animat's happiness is expressed by the fol-

[2] The theorem below doesn't require the presence of a NO-OP action in P, yet many LTL goals require such an action (e.g., the goal 'eventually always hold block A'). Also, the finite plans π that can be used to achieve some type of LTL goals (e.g., 'eventually hold block A'), map then into the infinite plans where π is followed by a NO-OP loop.

Instance	Total time	Plan Length
animat_3x3	30.96	76
animat_4x4	133.87	85
animat_5x5	948.87	115
animat_6x6,7x7,8x8,9x9	> 1079.73	(Out of mem)

Table 1: Results for *animat* domain. Times in seconds. Plan length includes aux. actions (effective length is 1/3 approx.).

lowing LTL formula: $\Box((h \neq max) \wedge (t \neq max) \wedge (p \neq 0)) \wedge \Box\Diamond(with\_partner) \wedge \Box\Diamond(\neg with\_partner)$, which requires an infinite plan such that: *(i)* h, t and p are guaranteed to never reach their max/min values; *(ii)* the animat visits its partner infinitely often; and *(iii)* the animat does something else than visiting its partner infinitely often.

As a first set of experiments[3], we tested the performance of FF (with the modifications previously discussed) in solving *animat* instances. Specifically, we increased the grid size from 3 to 9, and max from 15 (for $n = 3$) to 27 (for $n = 9$), adding 2 units each time n was increased by 1. As for the goal formula, we used exactly the same as seen above, by just setting the value of max depending on n. This problem is challenging for FF because it requires building a non-trivial lasso for which the EHC search fails. In Table 1 we show the results, with times expressed in seconds, and plan lengths including the auxiliary actions (number of domain actions is approx. 1/3). In this domain, the failure of the more focused EHC triggers a greedy best first search that runs out of memory over the largest domains. Still, this search produces non-trivial working 'loopy' plans, including almost 40 actions in the largest instance solved.

We carried out two additional classes of experiments on standard planning domains. In the first class, we test the overhead of the translation for purely classical problems, and hence reachability goals, with the NO-OP action added. For this we compare the performance of FF over the classical planning problems P' with goal G with the performance of FF over the translation P_φ where P is P' but with the goal G removed, and φ is the LTL formula $\Diamond G$. Results are shown in Table 2. As it can be seen from the table, there is a performance penalty that comes in part from the extra number of actions and fluents in the compiled problems (columns OP and FL). Still, the number of nodes expanded in the compiled problems remains close to that of nodes expanded in the original problems, and while times are higher, coverage over the set of instances does not change significantly (columns S). The scalability of FF over classical problems vs. their equivalent compiled LTL problems is shown in Fig. 1 for Gripper, as the number of balls is increased. While the times grow for the latter, the degradation appears to be polynomial as the number of expanded nodes is roughly preserved.

In the second class, we tested our approach on three classical problems (Blocksworld, Gripper and Logistics) using more complex LTL goals. Such experiments aim at evaluating the effectiveness of our approach wrt the general problem of finding infinite plans that satisfy generic LTL goals. We

[3]Experiments run on a dual-processor Xeon 'Woodcrest', 2.66 GHz CPU, 8 GB of RAM, with a process timeout of 30 minutes and memory limit of 2 GB).

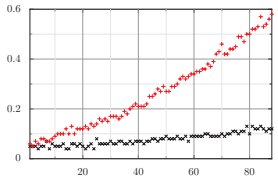

Figure 1: FF scalability over classical vs. LTL Gripper encodings (X-axis: #of balls. Y-axis: times in sec.). While the times grow for the LTL version, the degradation is polynomial.

Domain	I	S	E	AT	OP	FL
Blocks+LTL	50	31	141,573	72.84	4.5	5.4
Blocks	50	34	81,832	5.26		
Logistics+LTL	28	28	97	0.21	4.2	6.0
Logistics	28	28	94	0.07		
Satellite+LTL	20	20	103	0.45	2.3	8.0
Satellite	20	20	95	0.02		
TPP+LTL	30	24	21,513	123.27	2.5	7.1
TPP	30	30	15,694	8.19		
Grid+LTL	5	3	208	2.15	3.0	13.0
Grid	5	5	81	0.03		
Gripper+LTL	50	50	130	0.15	4.1	5.2
Gripper	50	50	102	0.06		

Table 2: Comparison between FF solving classical planning problems and FF solving the same problems stated as LTL reachability. Columns show domain name (+LTL for LTL version), # of instances (I), # of solved instances (S), av. # of expanded nodes (E), av. sol. time in sec (AT), av. factor of operators wrt classical (OPS), av. factor of fluents wrt classical (OPS). Times in seconds.

used five different classes of LTL formulas as goals:

- (Type 1) $\bigwedge_{i=1}^{n} \Diamond p_i$;
- (Type 2) $\Diamond(p_1 \wedge \bigcirc\Diamond(p_2 \wedge \ldots \wedge \bigcirc\Diamond(p_n)\ldots))$;
- (Type 3) $\bigwedge_{i=1}^{n} \Box\Diamond p_i$;
- (Type 4) $(\ldots(p_1 \,\mathcal{U}\, p_2)\,\mathcal{U}\ldots)\,\mathcal{U}\, p_n$;
- (Type 5) $(\Box\Diamond p_1 \rightarrow \Box\Diamond p_2) \wedge \ldots \wedge (\Box\Diamond p_{n-1} \rightarrow \Box\Diamond p_n)$.

Types 1, 3 and 4, appear among those proposed in [Rozier and Vardi, 2010] for model-checkers' performance comparison; type 2 is a type-1 variant, which forces the planner to plan for sequential goals; and type 5 formulas are built from *strong fairness* formulas $\Box\Diamond p \rightarrow \Box\Diamond q$, so as to generate large Büchi automata.

For all domains and classes of formulas above, we generated a set of instances, obtained by increasing several parameters. For Blocksworld, we increased the number of blocks, for Gripper the number of balls, and for Logistics the number of packages, airplanes and locations within each city, fixing the number of cities to 4. In addition to these, for each problem, we increased the LTL formula length, i.e., the number of boolean subformulas occurring in the LTL formula. Then, we compiled such instances into classical planning problems, according to the schema above, and solved them using FF. The results are shown in Table 3.

We tried to solve these collections of instances using the well-known symbolic model checker NuSMV [Cimatti *et al.*, 2002], so as to compare our approach with a state-of-the-art model checker [Rozier and Vardi, 2010]. In order to do so,

		COMPIL.		SOL. (FF)			
Domain	**I**	**C**	**ACT**	**S**	**NS**	**AST**	**TAT**
Blocks+LTL1	100	90	0.33	80	0	9.69	10.01
Blocks+LTL2	100	90	0.00	85	0	12.40	12.40
Blocks+LTL3	100	90	28.15	68	0	16.83	44.97
Blocks+LTL4	100	87	13.45	14	51	0.46	13.91
Blocks+LTL5	100	80	83.22	59	10	0.57	83.79
Logistics+LTL1	243	243	0.02	242	0	0.61	0.63
Logistics+LTL2	243	243	0.04	241	0	18.45	18.49
Logistics+LTL3	243	243	0.23	99	0	119.09	119.32
Logistics+LTL4	243	243	1.85	151	48	29.68	31.53
Logistics+LTL5	243	243	55.83	180	0	123.85	179.68
Gripper+LTL1	100	100	0.45	80	0	0.45	0.91
Gripper+LTL2	100	100	0.00	100	0	0.13	0.13
Gripper+LTL3	100	100	75.61	100	0	0.13	75.73
Gripper+LTL4	100	90	14.93	60	0	1.08	16.01
Gripper+LTL5	100	80	156.72	60	0	0.67	157.39

Table 3: Results for FF over compilations P_φ for different domains P and LTL goals φ. Columns show domain and class of LTL formula, # of instances (I), # of instances compiled successfully (C), avg. compilation time (ACT), # of solved instances (S), # of instances found unsolvable (NS), avg. solution time (AST), and avg. compilation+solution times (TAT).

we translated the LTL goals (before compilation into classical planning) into LTL model-checking ones, using a very natural schema, where ground predicates are mapped into boolean variables, and ground actions act as values for a variable. The model checker, however, runs out of memory on even the simplest instances of Blocksworld and Logistics with classical goals, and on most of the Gripper instances, and had even more problems when non-classical goals were used instead. The sheer *size* of these problems appears thus to pose a much larger challenge to model checkers than to classical planners.

7 Conclusion

We have introduced a general scheme for compiling away arbitrary LTL goals in planning, and have tested it empirically over a number of domains and goals. The transformation allows us to obtain infinite 'loopy' plans for an extended goal φ over a domain description P, from the finite plans that can be obtained with any classical planner from a problem P_φ. The result is relevant to both planning and model-checking: to planning, because it enables classical planners to produce a richer class of plans for a richer class of goals; to model-checking, because it enables the use of classical planning to model-check arbitrary LTL formulas over deterministic and non-deterministic domains. We have experimentally shown indeed that state-of-the-art model-checkers do not appear to scale up remotely as well as state-of-the-art planners that search with automatically derived heuristics and helpful actions. In the future, we want to test the use of the P_φ translation for model-checking rather than planning, and extend these ideas to planning settings where actions have non-deterministic effects, taking advantage of recent translations developed for conformant and contingent problems.

Acknowledgements

This work was partially supported by grants TIN2009-10232, MICINN, Spain, EC-7PM-SpaceBook, and EU Programme FP7/2007-2013, 257593 (ACSI).

References

[Albarghouthi *et al.*, 2009] A. Albarghouthi, J. Baier, and S. McIlraith. On the use of planning technology for verification. In *Proc. ICAPS'09 Workshop VV&PS*, 2009.

[Bacchus and Kabanza, 1998] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. of Math. and AI*, 22:5–27, 1998.

[Baier and McIlraith, 2006] J.A. Baier and S.A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proc. AAAI'06*, 2006.

[Baier *et al.*, 2009] J.A. Baier, F. Bacchus, and S.A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Art. Int.*, 173(5-6), 2009.

[Bauer and Haslum, 2010] A. Bauer and P. Haslum. LTL goal specifications revisited. In *Proc. ECAI'10*, 2010.

[Cimatti *et al.*, 2002] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. CAV'02*, 2002.

[Cresswell and Coddington, 2004] S. Cresswell and A. Coddington. Compilation of LTL goal formulas into PDDL. In *Proc. ECAI'04*, 2004.

[De Giacomo and Vardi, 1999] G. De Giacomo and M. Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *Proc. ECP'99*, 1999.

[Edelkamp, 2003] S. Edelkamp. Promela planning. In *Proc. SPIN'03*, 2003.

[Edelkamp, 2006] S. Edelkamp. On the compilation of plan constraints and preferences. In *ICAPS'06*, 2006.

[Gazen and Knoblock, 1997] B. Gazen and C. Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Proc. ECP'97*, 1997.

[Gerevini and Long, 2005] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, Univ. of Brescia, 2005.

[Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[Kabanza and Thiébaux, 2005] F. Kabanza and S. Thiébaux. Search control in planning for temporally extended goals. In *Proc. ICAPS'05*, 2005.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. In *Proc. FOCS'77*, 1977.

[Rozier and Vardi, 2010] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

[Schuppan and Biere, 2004] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004.

[Thomas, 1990] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

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

[Vardi, 1996] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, volume 1043 of *LNCS*. Springer, 1996.