Proceedings of the 2nd Workshop on
Model Checking and
Automated Planning
(MOCHAP-15)

Edited By:
Sergiy Bogomolov, Daniele Magazzeni, Martin Wehrle

Jerusalem, Israel 7/6/2015
Organizing Committee

Sergiy Bogomolov  
IST Austria, Austria

Daniele Magazzeni  
King’s College London, UK

Martin Wehrle  
University of Basel, Switzerland

Program Committee

Sergiy Bogomolov, IST Austria, Austria
Dragan Bosnacki, Eindhoven University of Technology, The Netherlands
Giuseppe Della Penna, University of L’Aquila, Italy
Stefan Edelkamp, University of Bremen, Germany
Georgios Fainekos, Arizona State University, USA
Goran Frehse, Verimag, France
Enrico Giunchiglia, University of Genova, Italy
Klaus Havelund, Jet Propulsion Laboratory, USA
Alan J. Hu, University of British Columbia, Canada
Alberto Lluch Lafuente, Technical University of Denmark, Denmark
Daniele Magazzeni, King’s College London, UK
Fabio Mercorio, University of Milano Bicocca, Italy
Andrea Orlandini, CNR-ISTC, Italy
Doron Peled, Bar Ilan University, Israel
Erion Plaku, Catholic University of America, USA
Andreas Podelski, University of Freiburg, Germany
Sylvie Thiébaux, Australian National University, Australia
Enrico Tronci, University of Rome “La Sapienza”, Italy
Martin Wehrle, University of Basel, Switzerland
Foreword

There has been a lot of work on the exchanges between the two research areas of model checking and automated planning. From a high level perspective, model checking and planning problems are related in the sense that plans (found by a planning system) correspond to error traces (found by a model checker), and vice versa. The two paradigms of “planning via model checking” and “directed model checking” are now widely used in different planning and verification domains.

The purpose of the workshop on Model Checking and Automated Planning (MOCHAP) is to continue to promote a cross-fertilisation between research on planning and verification, incrementing the synergy between the two areas.

After the successful first edition of the workshop, held at ICAPS 2014, this year MOCHAP-15 featured again a very rich program. Topics include planning in nondeterministic domains, planning with LTL, symmetry and partial order reduction, diagnosis and guided search on hybrid systems. Furthermore, two notable researchers have accepted our invitation to complete the program: Paolo Traverso, with a talk on "20 Years of Planning via Model Checking: From Theory to Practice", and Doron Peled, with a talk on "Commutativity based search".

We thank the members of the Program Committee for their dedicated effort in ensuring the quality of the papers presented at MOCHAP-15. We thank the invited speakers and all the authors for presenting their work and for contributing to a successful event.

Sergiy Bogomolov, Daniele Magazzeni, Martin Wehrle
MOCHAP-15 Chairs
# Table of Contents

20 Years of Planning via Model Checking: From Theory to Practice (Invited)
*Paolo Traverso*  5

Stubborn Sets for Fully Observable Nondeterministic Planning
*Dominik Winterer, Robert Mattmüller, Martin Wehrle*  6

Counterexample-Guided Abstraction Refinement for POND Planning
*Jonas Thiem, Robert Mattmüller, Manuela Ortlieb*  13

Compiling Away LTL Planning Goals in Polynomial Time
*Jorge Torres, Jorge Baier*  15

A Unifying Framework for Planning with LTL and Regular Expressions
*Eleni Triantafillou, Jorge Baier, Sheila McIlraith*  23

Commutativity Based Search (Invited)
*Doron Peled*  32

On Combining Symmetry with Partial Order Reduction
*Dragan Bosnacki*  33

UPMurphi Released: PDDL+ Planning for Hybrid Systems
*Giuseppe Della Penna, Benedetto Intrigila, Daniele Magazzeni, Fabio Mercorio*  35

Hybrid Systems: Guided Search, Abstractions, and Beyond
*Sergiy Bogomolov*  40


20 Years of Planning via Model Checking: From Theory to Practice

Paolo Traverso
FBK-ICTirst, Centre for Information Technology
I-38123 Trento - Italy

Abstract
Planning via Model Checking is nowadays a well-known technique. Techniques based on model checking have been successfully applied for dealing with different kinds of planning problems. In classical planning domains, LTL based model checking has been used to guide the search towards the goal. Classical planners based on BDDs have participated to the international planning competition since its early editions. Techniques based on both explicit state and symbolic model checking have been used to address the problem of planning under uncertainty, including planning with full observability in non-deterministic domains (FOND), planning with partial observability (POND), conformant planning, and planning in non-deterministic domains with temporally extended LTL and CTL goals. Techniques for interleaving planning via model checking and partial plan execution have been explored. Model checking techniques have also been used for planning with preferences, and planning in asynchronous domains. Recently, planning via model checking has been successfully applied to hybrid systems.

In my talk, I will review some of the different approaches in planning via model checking. I will then discuss some applications in different domains, e.g., applications for safety critical systems, web services and business processes, the dynamic management of harbor facilities, and planning for services in the field of smart cities and communities. I will discuss some lessons learned and some future challenges for the practical application of planning via model checking.
Stubborn Sets for Fully Observable Nondeterministic Planning

Dominik Winterer and Robert Mattmüller
University of Freiburg, Germany
{wintered, matts}@informatik.uni-freiburg.de

Martin Wehrle
University of Basel, Switzerland
martin.wehrle@unibas.ch

Abstract
The stubborn set method is a state-space reduction technique, originally introduced in model checking and then transferred to classical planning. It was shown that stubborn sets significantly improve the performance of optimal deterministic planners by considering only a subset of applicable operators in a state. Fully observable nondeterministic planning (FOND) extends the formalism of classical planning by nondeterministic operators. We show that stubborn sets are also beneficial for FOND problems. We introduce nondeterministic stubborn sets, stubborn sets which preserve strong cyclic plans. We follow two approaches: Fast Incremental Planning with stubborn sets from classical planning and LAO* search with nondeterministic stubborn sets. Experiments show that both approaches increase coverage and decrease node generations when compared to their respective baselines.

Introduction
Classical planning is the problem of finding a sequence of actions leading from a specified initial state to some goal state. Whereas in classical planning outcomes of actions are uniquely determined, fully observable nondeterministic planning (FOND) permits actions whose outcomes are uncertain. Such nondeterministic actions can be used to model, e.g., the failure of an agent’s action. While this is often addressed by re-planning, strong cyclic plans—trial-and-error strategies—empower the agent to solve failure situations without re-planning.

Recently, research in classical planning has shifted towards techniques orthogonal to heuristics such as partial order reduction which has been transferred from computer aided verification (Valmari 1989; Godefroid 1995) to optimal deterministic planning (Alkhazraji et al. 2012). Further research aimed at improving the efficiency of stubborn set computation and determining a generalized definition of stubborn sets (Wehrle and Helmert 2014). We address the stubborn set method combined with two algorithms, Fast Incremental Planning (FIP) and LAO*.

Fast Incremental Planning is an algorithm for strong cyclic planning which solves FOND problems within multiple runs of an underlying classical planner (Kuter et al. 2008). Planner for Relevant Policies (PRP) combines this idea with a regression search to generalize the policy and substantially outperforms FIP (Muise, McIlraith, and Beck 2012). Our first step towards estimating the potential of stubborn sets for FOND planning is to use FIP with an underlying classical planner in combination with stubborn sets from classical planning. However, the main drawback of such determination approaches is that they may find poor solutions, e.g., strong cyclic plans with high expected costs.

LAO* (Hansen and Zilberstein 2001), originally proposed to solve MDPs, is an algorithm for strong cyclic planning, which finds strong cyclic solutions in the nondeterministic state space. Using an admissible heuristic estimator, it finds strong cyclic plans of minimal expected costs. It has been shown that combining LAO* with pattern database heuristics (Mattmüller et al. 2010) is a successful approach to solving FOND problems. Our contribution is a stubborn set formalism for nondeterministic state spaces, that preserves strong cyclic plans. We evaluated both approaches, FIP with stubborn sets from classical planning and LAO* with our new formalism. Our results show that both approaches increase coverage and reduce node generations when compared to their respective baselines without stubborn sets.

Preliminaries
We use an SAS+ based notation (Bäckström and Nebel 1993) to model fully observable nondeterministic planning problems. States of the world are described by a finite set of state variables \( V \). Every variable \( v \in V \) has an associated finite domain \( D_v \) and an extended domain \( D_v^+ = D_v \cup \{\bot\} \) where \( \bot \) defines the undefined value. A partial state is a function \( s \) with \( s(v) \in D_v^+ \) for all \( v \in V \). We write \( \text{vars}(s) \) for the set of all \( v \) with \( s(v) \neq \bot \). A partial state is a state if \( \text{vars}(s) = V \).

**Definition 1** (nondeterministic planning task). A nondeterministic planning task is a 4-tuple \( \Pi = (V, O, s_0, s_\ast) \), where \( V \) is a finite set of finite-domain variables, \( O \) is a finite set of nondeterministic operators, \( s_0 \) is a state called the initial state and \( s_\ast \) is a partial state called the goal. Each nondeterministic operator \( o = (\text{Pre} \mid \text{Eff}) \) has a partial state \( \text{Pre} \) called precondition, a finite set of partial states \( \text{Eff} \) and an associated non-negative number \( \text{cost}(o) \) called its cost.

An operator \( o \) is applicable in a state \( s \) if \( \text{Pre} \) is sat-
s. The application of a single effect \( \text{eff} \in \text{Eff} \) in \( s \) yields the state \( \text{app}(\text{eff}, s) \) that results from updating the values of \( s \) with the ones of \( \text{eff} \). The application of \( o \) to a state \( s \) yields the set of states \( o(s) := \{ \text{app}(\text{eff}, s) | \text{eff} \in \text{Eff} \} \). The set of applicable operators in a state \( s \) is denoted by \( \text{app}(s) \). Sometimes we want to refer to a particular outcome of an operator \( o = \langle \text{Pre} | \{ \text{eff}_1, \cdots, \text{eff}_k \} \rangle \). The determination of non-deterministic operator \( o \) is \( o^{1}, \cdots, o^{k} \) with every outcome \( o^{i} = \langle \text{Pre} | \{ \text{eff}^{i}_1 \} \rangle \). The all-outcomes determination of planning task \( \Pi = \langle V, O, s_0, s_∗ \rangle \) is \( \text{Det} = \langle V, O_{\text{det}}, s_0, s_∗ \rangle \) where \( O_{\text{det}} \) is the set of all operator outcomes of \( O \).

An operator is deterministic if \( |\text{Eff}| = 1 \). It is non-deterministic if \( |\text{Eff}| \geq 2 \). We say a planning task \( \Pi = \langle V, O, s_0, s_∗ \rangle \) is deterministic if all of its operators are deterministic. We refer to all variables in the preconditions of an operator \( o \) as \( \text{vars}(o) \) and to all variables in its effects as \( \text{effvars} = \bigcup_o \text{vars}(\text{eff}_1) \).

A solution to a FOND planning task \( \Pi \) with set of states \( S \) is a policy \( \pi : S \rightarrow O \cup \{ \bot \} \), which maps a state to an appropriate action or is undefined, e.g. \( \pi(s) = \bot \). Policy \( \pi \) is weak if it defines at least one path from the initial state to a goal state following it. It is closed if following it either leads to a goal state or to a state where the policy is defined. Policy \( \pi \) is proper if from every state visited following it there exists a path to a goal state following it. Policy \( \pi \) is acyclic if it does not revisits already visited states.

**Definition 2** (weak plan, strong cyclic plan, strong plan). Let \( \Pi = \langle V, O, s_0, s_∗ \rangle \) be a planning task.

- A policy \( \pi \) is called a weak plan for \( \Pi \) if it is weak.
- A policy \( \pi \) is called a strong cyclic plan for \( \Pi \) if it is closed and proper.
- A policy \( \pi \) is called a strong plan for \( \Pi \) if it is closed proper and acyclic.

A weak plan is a sequence of actions which leads to the goal if all nondeterministic operator outcomes were deterministic. It corresponds to a plan in classical planning. A strong plan guarantees that after a maximum number of steps a goal state is reached. Strong cyclic planning relaxes that property requiring that the goal is reached within a finite sequence of actions. We want to emphasize that the nondeterminism in FOND planning is not necessarily the same as in model checking with nondeterministic models since unlike strong cyclic plans, counterexamples in model checking are linear sequences.

**Deterministic Stubborn Sets**

The first step towards stubborn sets is the definition of operator interference. We follow the definition of Wehrle and Helmert (2014).

**Definition 3** (interference of deterministic operators). Let \( o_1 \) and \( o_2 \) be operators of a deterministic planning task \( \Pi \) and let \( s \) be a state of \( \Pi \). Operators \( o_1 \) and \( o_2 \) interfere in \( s \) if they are both applicable in \( s \), and

- \( o_1 \) disables \( o_2 \) in \( s \), i.e., \( o_2 \notin \text{app}(o_1(s)) \), or
- \( o_2 \) disables \( o_1 \) in \( s \), i.e., \( o_1 \notin \text{app}(o_2(s)) \), or

![Figure 1: Solid: expensive strong cyclic solution. Dotted: cheap strong cyclic solution. Determinization-based algorithms might not find the cheap solution.

- \( o_1 \) and \( o_2 \) conflict in \( s \), i.e., \( s_{12} = o_1(o_2(s)) \) and \( s_{21} = o_2(o_1(s)) \) are both defined and differ: \( s_{12} \neq s_{21} \).

We approximate deterministic operator interference, by considering it globally for any state \( s \). According to this syntactic notion of interference, two deterministic operators \( o_1 \) and \( o_2 \) interfere if the effect of \( o_1 \) violates the precondition of \( o_2 \) (or vice versa) or if \( o_1 \) and \( o_2 \) have a common variable in their effects which they set to different values. Furthermore, we consider that operators which are never jointly applicable cannot interfere. This is done by checking whether the preconditions of two operators \( o_1 \) and \( o_2 \) are mutually exclusive (Wehrle and Helmert 2014).

For stubborn sets we need two more definitions. A disjunctive action landmark (DAL) in state \( s \) is a set of operators such that all operator sequences leading from \( s \) to a goal state contain some operator in the set. A necessary enabling set (NES) for operator \( o \) in state \( s \) is a set of operators such that all operator sequences that lead from \( s \) to some goal state and include \( o \) contain some operator in the NES before the first occurrence of \( o \). Both sets can be computed by selecting a variable \( v \) whose value differs from either the goal or the precondition of the operator to enable. Then, we add each operator which achieves the desired value of \( v \). As both sets are not uniquely determined, the pruning power and size of stubborn sets depend on their choices (Wehrle and Helmert 2014).

**Definition 4** (deterministic strong stubborn set). Let \( \Pi = \langle V, O, s_0, s_∗ \rangle \) be a deterministic planning task and \( s \) a state. A set \( S \subseteq O \) is a deterministic strong stubborn set (DSSS) in \( s \) if the following conditions hold:

1. \( S \) contains a disjunctive action landmark in \( s \).
2. For all operators \( o \in S \) there is no \( o \notin \text{app}(s) \), \( S \) contains a necessary enabling set for \( o \) in \( s \).
3. For all operators \( o \in S \) there is no \( o \in \text{app}(s) \), \( S \) contains all operators that interfere with \( o \) in \( s \).

We use FIP combined with an underlying classical planner using deterministic stubborn sets. Solving FOND problems with classical planners can lead to costly strong cyclic plans. Although optimality does not play the major role in FOND planning, the possibility of finding arbitrarily bad solutions is undesirable. We show that exactly this might happen.

**Example 1.** Consider a nondeterministic planning task \( \Pi = \langle V, O, s_0, s_∗ \rangle \) with variables \( V = \{ v_1, v_2 \} \) and the following
operators:
• \( o_1 = \langle v_1 = 0 \mid \{ v_1 := 1 \}, \{ v_2 := 2 \} \rangle \)
• \( o_2 = \langle v_1 = 1, v_2 = 0 \mid \{ T \}, \{ v_1 := 0, v_2 := 2 \} \rangle \)
• \( o_3 = \langle v_1 = 0 \mid \{ v_2 := 2 \}, \{ v_2 := 1 \} \rangle \)
• \( o_4 = \langle v_1 = 0, v_2 = 1 \mid \{ T \}, \{ v_2 := 2 \} \rangle \)

As cost function we have \( \text{cost} : \{ o_1 \mapsto 1, o_2 \mapsto 1000, o_3 \mapsto 2, o_4 \mapsto 1 \} \), the initial state is \( s_0 = \{ v_1 \mapsto 0, v_2 \mapsto 0 \} \) and the goal is \( s_\ast = \{ v_2 \mapsto 2 \} \). Assume we perform a run of the FIP algorithm and its first weak plan would be \( o_1\) inducing the fail-state \( o_1^{[1]}(s_0) = 10 \). In a subsequent weak plan search, the algorithm considers both outcomes of \( o_2 \) and adds them to the policy. This yields a clearly non-optimal strong cyclic plan, whereas the optimal solution consists of \( o_3 \) and \( o_4 \) (Figure 1). Applying PRP to this example gives the same solution, since regressing \( o_1^{[2]} \) is ineffective.

**Non-deterministic Stubborn Sets**

Reducing FOND problems to multiple classical planning problems sometimes leads to poor strong cyclic solutions since the individual runs of classical planners only guarantee good weak plans which are not always part of a good strong cyclic plans. To overcome this, it can be beneficial to plan in the nondeterministic state space e.g., with LAO* search (Hansen and Zilberstein 2001) which finds strong cyclic plans with minimum expected costs under certain assumptions. Planning in the nondeterministic state space needs new definitions of stubborn sets and operator interference since the former do not consider nondeterministic operators.

For a given nondeterministic planning problem \( \Pi \), a straightforward approach would be to directly apply the original definition of strong stubborn sets on the all-outcome-determination of \( \Pi \), and additionally, to add for every outcome \( o[i] \) of a nondeterministic operator \( o \) every other outcome of \( o \) in order to respect \( o \)'s nondeterministic nature. However, as the following example shows, such an approach is incomplete.

**Example 2.** Consider the following all-outcomes determination \( \Pi_{\text{det}} = \langle \mathcal{V}, \mathcal{O}_{\text{det}}, s_0, s_\ast \rangle \) of nondeterministic planning task \( \Pi \) with variables \( \mathcal{V} = \{ v_1, v_2 \} \) and the following operators:

- \( o_1^{[1]} = \langle v_1 = 0 \mid \{ v_1 := 1 \}, \{ v_2 := 2 \} \rangle \)
- \( o_2^{[1]} = \langle v_2 = 0 \mid \{ v_2 := 1 \}, \{ v_2 := 2 \} \rangle \)
- \( o_3^{[2]} = \langle v_2 = 0 \mid \{ v_2 := 3 \}, \{ v_2 := 4 \} \rangle \)
- \( o_{11} = \langle v_1 = 1, v_2 = 1 \mid \{ v_2 := 5 \} \rangle \)
- \( o_{12} = \langle v_2 = 1, v_2 = 2 \mid \{ v_2 := 5 \} \rangle \)
- \( o_{23} = \langle v_1 = 2, v_2 = 3 \mid \{ v_2 := 5 \} \rangle \)
- \( o_{24} = \langle v_1 = 2, v_2 = 4 \mid \{ v_2 := 5 \} \rangle \)

The initial state is \( s_0 = \{ v_1 \mapsto 0, v_2 \mapsto 0 \} \), and the goal is \( s_\ast = \{ v_2 \mapsto 5 \} \). The set \( \{ o_{11}, o_{12}, o_{23}, o_{24} \} \) is a disjunctive action landmark in \( s_0 \) which we add to the candidate set \( T_{s_0} \). As all operators in this set are inapplicable in \( s_0 \), we have to add a necessary enabling set for all of them. A valid choice for these necessary enabling sets is based on selecting the unsatisfied conditions \( v_2 = 1, v_2 = 2, v_2 = 3 \) and \( v_2 = 4 \) in the preconditions of \( o_{11}, o_{12}, o_{23}, o_{24} \), respectively, and to add the determined operators that set these conditions to true. These achieving operators correspond to all outcomes of \( o_2 \) and \( o_3 \), which are applicable in \( s_0 \) but non-interfering with any operator not in \( T_{s_0} \). Hence, we finally get \( T_{s_0} = \{ o_{11}, o_{12}, o_{23}, o_{24}, o_1^{[1]}, o_2^{[2]}, o_3^{[3]}, o_3^{[4]} \} \).

However, \( T_{s_0} \) is insufficient for our purpose because every strong plan from \( s_0 \) has to start with \( o_1 \); Depending on the nondeterministic outcome of \( o_1 (v_1 = 1 \text{ or } v_2 = 2) \), \( o_2 \) or \( o_3 \) can be applied to satisfy the precondition of an operator to reach the goal. In contrast, starting with \( o_2 \) and applying \( o_1 \) afterwards might lead to outcomes where no goal is reachable anymore (e.g., \( v_1 = 2 \) and \( v_2 = 2 \)). The analogous situation occurs when starting with \( o_3 \) and applying \( o_1 \) afterwards (Figure 2).

The core problem of our straightforward instantiation is that deterministic operator interference is an insufficient criterion for nondeterministic operators. Because changing the order of two non-interfering nondeterministic operators \( o \) and \( o' \) in a strong cyclic plan results in, e.g., outcomes of \( o' \) getting prefixes of weak plans started by \( o \). While this is not an issue for all weak plans which contain outcomes of both operators, it is problematic to those weak plans which start with an outcome of \( o \) but do not contain an outcome of \( o' \). A solution to this is to demand that such prefixes preserve the original weak plan which we address with the following property.

**Definition 5** (prefix-compatibility). Let \( \Pi \) be a planning task and \( \Pi_{\text{det}} = \langle \mathcal{V}, \mathcal{O}_{\text{det}}, s_0, s_\ast \rangle \) its all-outcomes determination. Two operators \( o_1, o_2 \in \mathcal{O}_{\text{det}} \) are prefix compatible if for all operator sequences \( \pi_1 \) and \( \pi_2 \):

- \( o_1 \pi_1 \) is a weak plan implies \( o_2 \pi_1 \pi_2 \) is also a weak plan and
- \( o_2 \pi_2 \) is a weak plan implies \( o_1 \pi_2 \pi_3 \) is also a weak plan

Intuitively, two operators \( o_1 \) and \( o_2 \) are prefix compatible if every weak plan starting with \( o_1 \) is preserved if we put \( o_2 \) to its front and vice versa. Equipped with prefix compatibility, we can formulate the definition of a stubborn set for the nondeterministic state space which has two additional rules compared to the DSS definition.

**Definition 6** (non-deterministic strong stubborn set). Let \( \Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\ast \rangle \) be a nondeterministic planning task,
$$\Pi_{det} = \langle V, O_{det}, s_0, s_\ast \rangle$$ its all-outcomes determination and s a state. A set $$T_s \subseteq O_{det}$$ is a nondeterministic strong stubborn set (NSSS) in s if the following conditions hold:

1. $$T_s$$ contains a disjunctive action landmark in s for $$\Pi_{det}$$.
2. For all operators $$o \in T_s$$ with $$o \notin app(s)$$, $$T_s$$ contains a necessary enabling set for o in s for $$\Pi_{det}$$.
3. For all operators $$o \in T_s$$ with $$o \in app(s)$$, $$T_s$$ contains all operators that interfere with o in s for $$\Pi_{det}$$.
4. For every outcome $$o[i] \in T_s$$ of nondeterministic operator o, $$T_s$$ contains all other outcomes of o.
5. For every outcome $$o[i] \in T_s$$ of nondeterministic operator o, $$T_s$$ contains all nondeterministic operators that are not prefix compatible with o.

**Proposition 1.** Non-deterministic strong stubborn sets preserve completeness for strong cyclic planning.

**Proof.** At first we show completeness for strong planning then we show it for strong cyclic planning. Let $$\pi$$ be a strong plan from state s that induces weak plans $$\pi_i$$ and $$\pi_j$$, such that there is state $$s$$ with $$\pi_i(s) = o[i]$$ and $$\pi_j(s) = o[j]$$. Weak plans $$\pi_i$$ and $$\pi_j$$ have the following structure: $$\pi_i = \alpha o[i] \beta_i$$ and $$\pi_j = \alpha o[j] \beta_j$$ where $$\alpha$$ is a common operator sequence without outcomes of nondeterministic operators, and $$\beta_i, \beta_j$$ contain outcomes of nondeterministic operators respectively. State $$s$$ is the branching point of $$\pi_i$$ and $$\pi_j$$.

Let $$k_i$$ be the smallest index such that operator $$o_{k_i} \in T_s$$ is contained in the nondeterministic stubborn set $$T_s$$, similarly for $$k_j$$ and $$\pi_j$$. We distinguish the following three cases.

1. $$o_{k_i} \in \alpha = o_1 \cdots o_n$$. Clearly $$o_{k_i} = o_{k_j}$$, $$o_{k_j}$$ is applicable since otherwise a necessary enabling set has to be contained in $$T_s$$ and at least one operator has to be applied before $$o_{k_i}$$, contradicting the choice of $$k_i$$. Since $$k_i$$ is the smallest index such that $$o_{k_i} \in T_s$$, $$o_{k_i}$$ does not interfere with any operator of smaller index because otherwise an operator applied before $$o_{k_i}$$ must be contained in $$T_s$$. Also, this contradicts the choice of $$k_i$$.

2. $$o_{k_i} = o[i]$$, $$o_{k_j} \notin \alpha$$ since otherwise $$o_{k_i} \in \alpha$$. Also, $$o_{k_j}$$ cannot be in $$\beta_j$$ because by the definition of the NSSS $$o[j] \in T_s$$.

3. $$o_{k_i} \in \beta_i = o_{n+2} \cdots o_{n+m}$$, $$o_{k_j}$$ cannot be in $$\alpha$$ since otherwise $$o_{k_j} \in \alpha$$. Also $$o_{k_j} \notin o[j]$$ since otherwise by definition of the NSSS, $$o[i]$$ would be included in $$T_s$$, contradicting the choice of $$k_i$$. Therefore $$o_{k_j} \in \beta_j$$.

Since a strong plan is a strong cyclic plan without cycles in the given design choices, we have that the non-interference of $$o_k$$ with operators of smaller index we get $$o_k(o_n \cdots (o_1(s_0))) = o_n(o_{n-1} \cdots (o_1(o_k(s_0))))$$. Hence we can move $$o_k$$ to the front of $$\pi_i$$ and $$\pi_j$$. If $$o_k$$ is an outcome of a nondeterministic operator o' then it has a sibling $$o'$$ which is the smallest index of weak plan $$\pi_k$$. This case is covered by case (2) with $$\pi_i$$ and $$\pi_j$$.

Since a strong plan is a strong cyclic plan without cycles in the given design choices, we have that the non-interference of $$o_k$$ with operators of smaller index we get $$o_k(o_n \cdots (o_1(s_0))) = o_n(o_{n-1} \cdots (o_1(o_k(s_0))))$$. Hence we can move $$o_k$$ to the front of $$\pi_i$$ and $$\pi_j$$. If $$o_k$$ is an outcome of a nondeterministic operator o' then it has a sibling $$o'$$ which is the smallest index of weak plan $$\pi_k$$. This case is covered by case (2) with $$\pi_i$$ and $$\pi_j$$.

Nondeterministic stubborn sets are in general not optimality preserving for strong cyclic planning since prefix compatibility leads to operators being added in front of other ones which can lead to solutions with higher expected costs.

**Approximating Prefix Compatibility**

The exact notion of prefix compatibility is intractable to compute because we have to consider all weak plans. Therefore we outline how to find a sufficient criterion for prefix-compatibility. We define $$Dis(o)$$ as the set of operator-variable pairs $$(o', v) \in O_{det} \times V$$ such that o disables o' on variable v in any state. Further we define $$Neg(o)$$ as the set of goal variables with which o conflicts, i.e., $$eff(o)[v] \neq s_i[v]$$ for goal-related variables v on which o has an effect. If $$Dis(o_1) = Dis(o_2)$$ and $$Neg(o_1) = Neg(o_2)$$ then o_1 and o_2 are prefix compatible. The idea behind this is: if two operators o_1 and o_2 disable the same set of operators on the same variables, then every deterministic operator sequence starting with o_1 remains applicable if we append o_2 to its front. Also weak plans are preserved since o_1 and o_2 do not violate different goal variables.

In some cases, we can weaken this syntactic notion of prefix compatibility. Consider two non-interfering operators $$o_1 = \{Pre | \{eff_1\}\}$$ and $$o_2 = \{Pre | \{eff_1, \ldots, eff_n\}\}$$.

If $$\sigma = \{s \mapsto o_1, o_1(s) \mapsto o_2\}$$ is a subsequence of a strong cyclic plan from state s then exchanging the order of o_1 and o_2 gives an equivalent subsequence since they induce the same set of states, i.e., $$o_1(o_2(s)) = o_2(o_1(s))$$ for all $$i \leq n$$. Therefore if two such operators do not interfere, it suffices to check $$Dis(o_1) \subseteq Dis(o_2)$$ and $$Neg(o_1) \subseteq Neg(o_2)$$.

Sometimes nondeterministic operators contain only one nontrivial effect, i.e., an operator o = $$(Pre | \{eff_i\}, \{T\})$$. For every weak plan o[i] from state s, it exists a finite sequence $$\sigma = o[2] \cdots o[2]$$, repeated applications of o’s trivial effect, such that $$\sigma o[1]$$ is a weak plan from s. Thus, every operator being a prefix of o[1] preserving the weak plan, does also preserve $$\sigma o[1]$$, i.e., $$\sigma o[1]$$.

**Efficient Computation**

As nondeterministic stubborn sets leave open how the disjunctive action landmark and the necessary enabling sets were chosen, the pruning power of stubborn sets depends highly on these design choices. We outlined how prefix compatibility can be syntactically addressed. However, for
applicable nondeterministic operators with more than one nontrivial effect, in the stubborn set we have to add both the interfering and the non-prefix compatible operators. This leads to many operators being added to the stubborn set. It is therefore reasonable to avoid applicable nondeterministic operators with more than one nontrivial effect from being added to the stubborn set. Let nontrivial be the set of operators with more than one nontrivial effect. Our intention is to exclude applicable operators of nontrivial from being added to the stubborn set. We address this by computing a weight whenever we have to add a DAL or NES to the stubborn set. We calculate a weight for each DAL or NES and chose the DAL or NES with lowest weight according to:

\[
\text{weight}(o, s, T_s) = \begin{cases} 
\infty, & \text{if } o \in \text{app}(s) \land o \in \text{nontrivial} \\
K, & \text{if } o \in \text{app}(s) \land o \notin \text{nontrivial} \\
1, & \text{otherwise}
\end{cases}
\]

where \( o \) is an operator not in stubborn set \( T_s \) and \( K \) a nonzero natural number. Our exclude strategy is an extension of a strategy presented by Laarman et al. (2013) which penalizes applicable operators not in the current candidate stubborn set. A coarser strategy towards prefix compatibility for nontrivial operators is to simply assume that an applicable nontrivial operator is not prefix compatible to all other operators. On par with the exclude strategy this is feasible since it avoids the costly computation of the disabling relation.

A Tighter Envelope

Active operators (Chen and Yao 2009; Wehrle et al. 2013) approximate the set of operators which can be part of any weak plan from some state using domain transitions graphs (DTGs). From a more general point of view, Wehrle et al. (2013) denote subsets which preserve at least one weak plan from some state as an envelope. Combining a tight envelope with stubborn sets may not only exclude operators which are not in envelope \( E \) from the stubborn set but also prevent cascades from being added to the stubborn set. Of course, the active operators can also be used for strong cyclic planning since strong cyclic plans consist of multiple weak plans. We additionally exploit the structure of strong cyclic plans and obtain a tighter envelope.

A part-of-a-plan operator \( o \in \mathcal{O} \) in \( s \) is a deterministic operator that is contained in some weak plan starting from \( s \). This notion is intractable to compute so we have to find a sufficient criterion for it.

Definition 7 (active operator). Let \( \Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\ast \rangle \) be a deterministic planning task. An active operator \( o \in \mathcal{O} \) in a state \( s \) is an operator that satisfies the following conditions:

1. For every variable \( v \in \text{prevvars}(o) \), there is a path in \( \text{DTG}(v) \) from \( s[v] \) to \( \text{pre}(o)[v] \), and also from \( \text{pre}(o)[v] \) to the goal value \( s[v] \) if \( v \) is goal-related.
2. For all \( v \in \text{effvars}(o) \cap \text{vars}(s_\ast) \) there is a path in \( \text{DTG}(v) \) from \( \text{eff}(o)[v] \) to \( s_\ast[v] \).

Intuitively, the definition states that an operator is active if it is part of some weak plan from the corresponding abstracted state in every singleton abstraction of \( \Pi \). Thus, a part-of-a-plan operator is always active but not vice versa.

A nondeterministic part-of-a-plan operator \( o \in \mathcal{O} \) in \( s \) is an operator that is contained in some strong plan \( \pi \) starting from \( s \). Like for the part-of-a-plan operators this is intractable to compute.

Proposition 2. Let \( o \) be an applicable nondeterministic operator and \( o^{[i]} \) one of its outcomes. If \( o^{[i]} \) is inactive in state \( s \) then \( o \) cannot be part of any strong cyclic plan from \( s \).

Proof. We show this by contradiction. If there were a strong cyclic plan \( \pi \) from \( s \), such that \( \pi(s) = o \) for some state \( s \) but \( o^{[i]} \) is no part-of-a-plan operator in \( s \). Let further \( \sigma = o_1 \cdots o_n \) be a deterministic operator sequence applicable in \( s \) that leads to \( s \). Since \( o^{[i]} \) is no part-of-a-plan operator in \( s \), no weak plan from \( s \) does contain \( o^{[i]} \). Therefore \( \sigma o^{[i]} \pi' \) for an arbitrary operator sequence \( \pi' \) cannot be a weak plan from \( s \). This implies that \( \pi' \) is no weak plan from \( o^{[i]}(\tilde{s}) \). Thus \( \pi \) is not proper since \( o^{[i]}(\tilde{s}) \) is reachable following \( \pi \) but there is no goal state reachable from \( o^{[i]}(\tilde{s}) \). This contradicts \( \pi \) being a strong plan. It follows that if any outcome of a nondeterministic operator \( o \) is not active in \( s \), then \( o \) cannot be part of a strong cyclic plan from \( s \).

We denote our new envelope by nondeterministic active envelope. It can be used for both the DSSS and NSSS.

Experimental Evaluation

We focused our experimental evaluation on the following two configurations:

1. FIP combined with DSSSs
2. LAO* combined with NSSSs

We further investigated the impact of different envelopes: full, active, nondeterministic active (Table 1 and Table 2). Also, we varied the approximation of prefix compatibility for the NSSS approach (Table 3). We differentiate between the approach which assumes that every nondeterministic operator is not prefix compatible with all other operators (no prefix) and the approach where prefix compatibility is syntactically approximated (syntactic). For the DSSS, disjunctive action landmarks and necessary enabling sets were computed using the laarman strategy. For the NSSS we used the exclude strategy. The interference relation for both the DSSSs and NSSSs is entirely precomputed which is also true for the achievers, the NSSSs need the additional precomputation of the disabling relation. For the underlying classical planner of FIP, we used greedy best first search. As heuristic estimator, we chose the FF heuristic (Hoffmann and Nebel 2001) for all approaches.

We evaluated both stubborn set approaches on all FOND domains of the IPC-2008 and variations of these. Furthermore we added two domains from probabilistic planning to our benchmark set.1 All experiments were conducted on a server equipped with AMD Opteron 2.3 GHz CPUs. We set

---

1First-Responders-new consists of larger instances of the First-Responders domain. Forest-new is taken by Muise, McIlraith, and Beck (2012). Tidyup is the Mobile Manipulation domain of Hertle et al. (2014) adapted for FOND planning. Earth-Observation was introduced by Aldinger and Löh (2013).
Table 1: Comparison of plain FIP with FIP using DSSS and different envelopes: full, active (DACT), nondeterministic active (NACT). Nodes of DACT, NACT in % of plain FIP.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Coverage</th>
<th>Node Generations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FIP</td>
<td>DSSS</td>
</tr>
<tr>
<td>FR (75)</td>
<td>94</td>
<td>74</td>
</tr>
<tr>
<td>FR-New (91)</td>
<td>75</td>
<td>-1</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>16</td>
<td>-e4</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>13</td>
<td>-e4</td>
</tr>
<tr>
<td>Earth (40)</td>
<td>35</td>
<td>-2</td>
</tr>
<tr>
<td>Tidyup (10)</td>
<td>5</td>
<td>±0</td>
</tr>
<tr>
<td>Tireworld (40)</td>
<td>3</td>
<td>±0</td>
</tr>
<tr>
<td>BW (30)</td>
<td>25</td>
<td>-1</td>
</tr>
<tr>
<td>Faults (55)</td>
<td>55</td>
<td>±0</td>
</tr>
<tr>
<td>Overall</td>
<td>301</td>
<td>±3</td>
</tr>
</tbody>
</table>

Table 2: Comparison of plain LAO* with LAO* using NSSS and different envelopes: full, active (DACT), nondeterministic active (NACT). Nodes of DACT, NACT in % of plain LAO*.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Coverage</th>
<th>Node Generations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAO*</td>
<td>NSSS</td>
</tr>
<tr>
<td>FR (75)</td>
<td>57</td>
<td>+1</td>
</tr>
<tr>
<td>FR-New (91)</td>
<td>19</td>
<td>-2</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>3</td>
<td>±3</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>6</td>
<td>±1</td>
</tr>
<tr>
<td>Earth (40)</td>
<td>30</td>
<td>±0</td>
</tr>
<tr>
<td>Tidyup (10)</td>
<td>9</td>
<td>±0</td>
</tr>
<tr>
<td>Tireworld (40)</td>
<td>6</td>
<td>±0</td>
</tr>
<tr>
<td>FW (30)</td>
<td>21</td>
<td>±0</td>
</tr>
<tr>
<td>Faults (55)</td>
<td>54</td>
<td>±0</td>
</tr>
<tr>
<td>Overall</td>
<td>208</td>
<td>±3</td>
</tr>
</tbody>
</table>

Table 3: Comparison of no prefix compatibility for nontrivial operators with syntactic approximation. We grouped the domains where coverage and node generations are equal. Nodes of syntactic in % of no prefix approach.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Coverage</th>
<th>Node Generations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no prefix</td>
<td>syntactic</td>
</tr>
<tr>
<td>FR (75)</td>
<td>59</td>
<td>±2</td>
</tr>
<tr>
<td>FR-New (91)</td>
<td>17</td>
<td>+1</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>6</td>
<td>+1</td>
</tr>
<tr>
<td>Forest-New (90)</td>
<td>7</td>
<td>-1</td>
</tr>
</tbody>
</table>
References


Counterexample-Guided Abstraction Refinement for POND Planning

Jonas Thiem and Robert Mattmüller and Manuela Ortlieb
University of Freiburg, Germany
{thiemj,mattmuel,ortlieb}@informatik.uni-freiburg.de

Abstract

Counterexample-guided abstraction refinement (CEGAR) allows to gradually refine a problem until the required detail for a solution is reached. We propose the use of CEGAR to demonstrate unsolvability of partially observable nondeterministic planning tasks while avoiding search through the entire state space.

Partially observable tasks are ubiquitous in planning and robotics (Oliehoek 2010, p. 3). Sometimes, it is important to show unsolvability of such a task fast. Examples include algorithms minimizing necessary sensors, where unsolvability proofs are needed to show that a certain sensor cannot be left out (Mattmüller, Ortlieb, and Wacker 2014), and algorithms for strong and strong cyclic planning (Cimatti et al. 2003) that first try to find a strong plan, and if strong plan non-existence has been established, resort to finding a strong cyclic plan instead. The Counterexample-Guided Abstraction Refinement (CEGAR) technique originating from model checking (Clarke et al. 2003) can be used to speed up unsolvability proofs and has recently been used for classical planning (Seipp 2012), and, in a setting closely related to ours, in the context of games with incomplete information (Dimitrova and Finkbeiner 2008). CEGAR works as follows: It starts with a small initial abstraction of the planning task and searches for an abstract plan. If no such plan exists, it makes use of the fact that abstractions induce over-approximations of reachability and concludes that no concrete plan can exist, either. Otherwise, CEGAR tries to concretize the abstract plan found. Either, the solution is concretizable. Then CEGAR terminates. Otherwise, the solution is spurious and the abstraction must be refined. In our setting, instead of requiring abstractions where every single transition is preserved, preserving goal reachability is sufficient. A central question is how to define abstractions (guaranteeing over-approximations). A straightforward way for POND planning is to define an abstract belief state \( B \) as a set of concrete belief states \( B \), where each such \( B \) consists of the set of world states \( s \) considered possible in \( B \). Then the abstract initial state, goal states, and transitions can be defined easily. E.g., an action precondition \( \varphi \) is satisfied in \( B \) iff it is satisfied in some concrete belief state \( B \) represented by \( B \) (to ensure over-approximation), and \( \varphi \) is satisfied in \( B \) iff it is satisfied in all states \( s \in B \) (to account for the uncertainty of the belief \( B \)). Unfortunately, representing such a set of states \( B \) compactly is hard. On the other hand, representing a set \( B \) of states \( s \) compactly is, although exponential in the worst case, often feasible using binary decision diagrams (BDDs) (Bryant 1986). Therefore, in this work we approximate abstract belief states \( B \) by BDD-encoded sets of world states \( B \). Furthermore, as abstractions we use simple projections to patterns \( P \), i.e., sets of variables (Culberston and Schaeffer 1996). This raises several questions: (a) How to define and compute an (approximate) abstraction to a pattern \( P \) efficiently, (b) how to ensure that goal reachability is preserved, and (c) how to refine an abstraction if necessary. For (a), we use a simple syntactic projection to \( P \) similar to the one used for PDB heuristics in classical planning. However, when we use sets \( B \) of world states as abstract states and thus let the layers “belief” and “abstraction” collapse into one, we introduce an error that violates the over-approximation. We amend this as follows: We only allow variables in \( P \) that can always and unconditionally be observed, or that are known initially and can never become unknown. In addition, we forbid observations of variables outside of the pattern. This guarantees that we only ever produce singleton abstract belief states. Since we forbid some observations, we have no longer an over-approximation, but it can be proven that the goal reachability, possibly along longer paths, is preserved with the chosen restrictions. Regarding (c), we refine an abstraction by collecting all actions in the abstract policy whose precondition is violated in the concrete task and add the violated precondition variables to the refined pattern. If this violates the restriction of patterns, we immediately move to a pattern consisting of all variables in the planning task, i.e., to the identity abstraction. We implemented this variant of the CEGAR algorithm on top of the MYND planner (Mattmüller et al. 2010). For the benchmarked unsolvable problems, CEGAR leads to an increase of 20 to 50 percent in successfully handled unsolvable problems. As a downside, solvable problems suffer a slowdown which gradually widened in our benchmarks. For future work, we plan to investigate the performance of CEGAR as part of the two motivating scenarios, sensor minimization and strong/strong-cyclic planning. We also plan to investigate alternative abstractions such as the doubly exponential one mentioned above.
Acknowledgments
This work was partly supported by the DFG as part of the SFB/TR 14 AVACS.

References
Compiling Away LTL Planning Goals in Polynomial Time

Jorge Torres
Department of Computer Science
Pontificia Universidad Católica de Chile
Santiago, Chile

Abstract
Linear temporal logic (LTL) is an expressive language that allows specifying temporally extended goals and preferences. A general approach to dealing with general LTL properties in planning is by “compiling them away”; i.e., in a pre-processing phase, all LTL formulas are converted into simple, non-temporal formulas that can be evaluated in a planning state. This is accomplished by first generating a finite-state automaton for the formula, and then by introducing new fluents that are used to capture all possible runs of the automaton. Unfortunately, current translation approaches are worst-case exponential on the size of the LTL formula. In this paper, we present a polynomial approach to compiling away LTL goals. Our method relies on the exploitation of alternating automata. Since alternating automata are different from non-deterministic automata, our translation technique does not capture all possible runs in a planning state and thus is very different from previous approaches. We prove that our translation is sound and complete, and evaluate it empirically showing that it has strengths and weaknesses. Specifically, we find classes of formulas in which it seems to outperform significantly the current state of the art.

Introduction
Linear Temporal Logic (LTL) (Pnueli 1977) is a compelling language for the specification of goals in AI planning, because it allows defining constraints on state trajectories which are more expressive than simple final-state goals, such as “deliver priority packages before non-priority ones”, or “while moving from the office to the kitchen, make sure door D becomes closed some time after it is opened”. It was first proposed as the goal specification language of TLPlan system (Bacchus and Kabanza 1998). Currently, a limited but compelling subset of LTL has been incorporated into PDDL3 (Gerevini et al. 2009) for specifying hard and soft goals.

While there are some systems that natively support the PDDL3 subset of LTL [e.g., Coles and Coles, 2011], when planning for general LTL goals, there are two salient approaches: goal progression (Bacchus and Kabanza 1998) and compilation approaches (Rintanen 2000; Cresswell and Coddington 2004; Edelkamp, Jabbar, and Naizih 2006; Baier and McIlraith 2006). Goal progression has been shown to be extremely effective when the goal formula encodes some domain-specific control knowledge that prunes large portions of the search space (Bacchus and Kabanza 2000). In the absence of such expert knowledge, however, compilation approaches are more effective at planning for LTL goals since they produce an equivalent classical planning problem, which can then be fed into optimized off-the-shelf planners.

State-of-the-art compilation approaches to planning for LTL goals exploit the relationship between LTL and finite-state automata (FSA) (Edelkamp 2006; Baier and McIlraith 2006). As a result, the size of the output is worst-case exponential in the size of the LTL goal. Since deciding plan existence for both LTL and classical goals is PSPACE-complete (Bylander 1994; De Giacomo and Vardi 1999), none of these approaches is optimal with respect to computational complexity, since they rely on a potentially exponential compilation. From a practical perspective, this worst case is also problematic since the size of a planning instance has a direct influence on planning runtime.

In this paper, we present a novel approach to compile away general LTL goals into classical goals that runs in polynomial time on the size of the input that is thus optimal with respect to computational complexity. Like existing FSA approaches, our compilation exploits a relation between LTL and automata, but instead of FSA, we exploit alternating automata (AA), a generalization of FSA that does not seem to be efficiently compilable with techniques used in previous approaches. Specifically, our compilation handles each non-deterministic choice of the AA with a specific action, hence leaving non-deterministic choices to be decided at planning time. This differs substantially from both Edelkamp’s and Baier and McIlraith’s approaches, which represent all runs of the automaton simultaneously in a single planning state.

We propose variants of our method that lead to performance improvements of planning systems utilizing relaxed-plan heuristics. Finally, we evaluate our compilation empirically, comparing it against Baier and McIlraith’s—who below we refer to as B&M. We conclude that our translation has strengths and weaknesses: it outperforms B&M’s for classes of formulas that require very large FSA, while B&M’s seems stronger for shallower, simpler formulas.
In the rest of the paper, we outline the required background, we describe our AA construction for finite LTL logic, and then show the details of our compilation approach. We continue describing the details of our empirical evaluation. We finish with conclusions. Refer to Torres and Baier (2015) for a slightly extended version of this paper.

Preliminaries

The following sections describe the background necessary for the rest of the paper.

Propositional Logic Preliminaries

Given a set of propositions $F$, the set of literals of $F$, $\text{Lit}(F)$, is defined as $\text{Lit}(F) = F \cup \{-p \mid p \in F\}$. The complement of a literal $\ell$ is denoted by $\bar{\ell}$, and is defined as $\neg \ell$ if $\ell = p$ and as $\ell$ if $\ell = \neg p$, for some $p \in F$. $L$ denotes $\{\ell \mid \ell \in L\}$.

Given a Boolean value function $\pi : P \rightarrow \{\text{false}, \text{true}\}$, and a Boolean formula $\varphi$ over $P$, $\pi \models \varphi$ denotes that $\pi$ satisfies $\varphi$, and we assume it defined in the standard way. To simplify notation, we use $s \models \varphi$, for a set $s$ of propositions, to abbreviate $\pi_x \models \varphi$, where $\pi_x = \{p \rightarrow \text{true} \mid p \in s\} \cup \{p \rightarrow \text{false} \mid p \in F \setminus s\}$. In addition, we say that $s \models R$, when $R$ is a set of Boolean formulas, iff $s \models r$, for every $r \in R$.

Deterministic Classical Planning

Deterministic classical planning attempts to model decision making of an agent in a deterministic world. We use a standard planning language that allows so-called negative preconditions and conditional effects. A planning problem is a tuple $\langle F, O, I, G \rangle$, where $F$ is a set of propositions, $O$ is a set of action operators, $I \subseteq F$ defines an initial state, and $G \subseteq \text{Lit}(F)$ defines a goal condition.

Each action operator $a$ is associated with the pair $(\text{prec}(a), \text{eff}(a))$, where $\text{prec}(a) \subseteq \text{Lit}(F)$ is the precondition of $a$ and $\text{eff}(a)$ is a set of conditional effects, each of the form $C \rightarrow \ell$, where $C \subseteq \text{Lit}(F)$ is a condition and literal $\ell$ is the effect. Sometimes we write $\ell$ as a shorthand for the unconditional effect $\{\ell\} \rightarrow \ell$.

We say that an action $a$ is applicable in a planning state $s$ iff $s \models \text{prec}(a)$. We denote by $\rho(s, a)$ the state that results from applying $a$ in $s$. Formally,

$$\rho(s, a) = (s \setminus \{p \mid C \rightarrow \neg p \in \text{eff}(a), s \models C\}) \cup \{p \mid C \rightarrow p \in \text{eff}(a), s \models C\}$$

if $s \models F$ and $a$ is applicable in $s$; otherwise, $\delta(a, s)$ is undefined. If $\alpha$ is a sequence of actions and $a$ is an action, we define $\rho(s, \alpha a)$ as $\rho(\delta(s, \alpha), a)$ if $\rho(s, \alpha)$ is defined. Furthermore, if $\alpha$ is the empty sequence, then $\rho(s, \alpha) = s$. An action sequence $\alpha$ is applicable in a state $s$ iff $\rho(s, \alpha)$ is defined. If an action sequence $\alpha = a_1a_2 \ldots a_n$ is applicable in $s$, it induces an execution trace $\sigma = s_1 \ldots s_{n+1}$ in $s$, where $s_i = \rho(I, a_1 \ldots a_{i-1})$, for every $i \in \{1, \ldots, n + 1\}$.

An action sequence is a plan for problem $\langle F, O, I, G \rangle$ if $\alpha$ is applicable in $I$ and $\rho(I, \alpha) \models G$.

Alternating Automata

Alternating automata (AA) are a natural generalization of non-deterministic finite-state automata (NFA). At a definitional level, the difference between an NFA and an AA is the transition function. For example, if $A$ is an NFA with transition function $\delta$, and we have that $\delta(q, a) = \{p, r\}$, then this intuitively means that $A$ may end up in state $p$ or in state $r$ as a result of reading symbol $a$ when $A$ was previously in state $q$. With an AA, transitions are defined as formulas. For example, if $\delta'$ is the transition function for an AA $A'$, then $\delta'(q, a) = p \lor r$ means, as before, that $A'$ ends up in $p$ or $r$ after reading an $a$ in state $q$. Nevertheless, formulas provide more expressive power. For example $\delta'((q, b) = (s \lor t) \lor r$ can be intuitively understood as $A'$ will end up in both $s$ and $t$ or (only) in $r$ after reading a $b$ in state $q$. In this model, only positive Boolean formulas are allowed for defining $\delta$.

Definition 1 (Positive Boolean Formula) The set of positive formulas over a set of propositions $P$—denoted by $B^+(P)$—is the set of all Boolean formulas over $P$ and constants $\bot$ and $\top$ that do not use the connective “$\lnot$”.

The formal definition for AA that we use henceforth follows.

Definition 2 (Alternating Automata) An alternating automata (AA) over words is a tuple $A = (Q, \Sigma, \delta, I, F)$, where $Q$ is a finite set of states, $\Sigma$, the alphabet, is a finite set of symbols, $\delta : Q \times \Sigma \rightarrow B^+(Q)$ is the transition function, $I \subseteq Q$ are the initial states, and $F \subseteq Q$ is a set of final states.

As suggested above, any NFA is also an AA. Indeed, given an NFA with transition function $\delta$, we can generate an equivalent AA with transition function $\delta'$ by simply defining $\delta'(q, a) = \bigvee_{p \in P} p$, when $\delta(q, a) = P$. We observe that this means $\delta'(q, a) = \bot$ when $P$ is empty.

As with NFAs, an AA accepts a word $w$ whenever there exists a run of the AA over $w$ that satisfies a certain property. Here is the most important (computational) difference between AAs and NFAs: a run of an AA is a sequence of sets of states rather than a sequence of states. Before defining runs formally, for notational convenience, we extend $\delta$ for any subset $T$ of $Q$ as $\delta(T, a) = \bigwedge_{q \in T} \delta(q, a)$ if $T \neq \emptyset$ and $\delta(T, a) = \top$ if $T = \emptyset$.

Definition 3 (Run of an AA over a Finite String) A run of an AA $A = (Q, \Sigma, \delta, I, F)$ over word $x_1x_2\ldots x_n$ is a sequence $Q_0Q_1\ldots Q_n$ of subsets of $Q$, where $Q_0 = I$, and $Q_i = \delta(Q_{i-1}, x_i)$, for every $i \in \{1, \ldots, n\}$.

Definition 4 A word $w$ is accepted by an AA $A$ if there is a run $Q_0Q_1\ldots Q_n$ of $A$ over $w$ such that $Q_n \subseteq F$.

For example, if the definition of an AA $A$ is such that $\delta'(q, b) = (s \lor t) \lor r$, and $I = \{q\}$, then both $\{q\}\{s, t\}$ and $\{q\}\{r\}$ are runs of $A$ over word $b$.

Finite LTL

The focus of this paper is planning with LTL interpreted over finite state sequences (Baier and McIraith 2006; De Giacomo and Vardi 2013). At the syntax level, the finite LTL we use in this paper is almost identical to regular LTL, except for the addition of a “weak next” modality (\(\diamond\)). The definition follows.
Figure 1: An NFA for formula $\square(p \rightarrow \Diamond \lozenge q)$ that expresses the fact that every time $p$ becomes true in a state, then $q$ has to be true in the state after or in the future. The input to the automaton is a (finite) sequence $s_0 \ldots s_n$ of planning states.

**Definition 5 (Finite LTL formulas)** The set of finite LTL formulas over a set of propositions $P$, $fLTL(P)$, is inductively defined as follows:

- $p$ is in $fLTL(P)$, for every $p \in P$.
- If $\phi$ and $\psi$ are in $fLTL(P)$ then so are $\neg \phi$, $(\phi \land \psi)$, $(\phi \lor \psi)$, $\Diamond \phi$, $\Box \phi$, and $(\phi R \psi)$.

The truth value of a finite LTL formula is evaluated over a finite sequence of states. Below we assume that those states are actually planning states.

**Definition 6** Given a sequence of states $\sigma = s_0 \ldots s_n$ and a formula $\phi \in fLTL(P)$, we say that $\sigma$ satisfies $\phi$, denoted as $\sigma \models \phi$, iff it holds that $\sigma, 0 \models \phi$, where, for every $i \in \{0,\ldots,n\}$:

1. $\sigma, i \models p$ iff $s_i \models p$, when $p \in P$.
2. $\sigma, i \models \neg \phi$ iff $\sigma, i \not\models \phi$.
3. $\sigma, i \models \phi \land \chi$ iff $\sigma, i \models \phi$ and $\sigma, i \models \chi$.
4. $\sigma, i \models \phi \lor \chi$ iff $\sigma, i \models \phi$ or $\sigma, i \models \chi$.
5. $\sigma, i \models \phi U i$ iff there exists $\phi \in \{i, \ldots, n\}$ such that $\sigma, k \models \chi$ and for each $j \in \{i, \ldots, k-1\}$, it holds that $\sigma, j \models \psi$.
6. $\sigma, i \models \Box \psi$ iff $\sigma, i \models \psi$.  
7. $\sigma, i \models \psi$ iff for each $k \in \{i, \ldots, n\}$ it holds that $\sigma, k \models \chi$ or there exists a $j \in \{i, \ldots, k-1\}$ such that $\sigma, j \models \psi$.

We sometimes use the macros $\text{true} \equiv p \lor \neg \phi$, false $\equiv \neg \text{true}$, and $\phi \Rightarrow \psi$ as $\neg \phi \lor \psi$. Additionally, $\Diamond \phi$, pronounced as "eventually $\phi$" is defined as true $U \phi$, and $\Box \phi$, pronounced as "always $\phi$" is defined as $\neg \Diamond \neg \phi$.

**Deterministic Planning with LTL goals**

A planning problem with a finite LTL goal is a tuple $P = (F, O, I, G)$, where $F$, $O$, and $I$ are defined as in classical planning problems, but where $G$ is a formula in $fLTL(F)$. An action sequence $\alpha$ is a plan for $P$ if $\alpha$ is applicable in $I$, and the execution trace $\sigma$ induced by the execution of $\alpha$ in $I$ is such that $\sigma \models G$.

There are two approaches to compiling away LTL via non-deterministic finite-state automata (Edelkamp, Jabbar, and Naizh 2006; Baier and McIlraith 2006). B&M’s approach compiles away LTL formulas exploiting the fact that for every finite LTL formula $\phi$ it is possible to build an NFA that accepts the finite models of $\phi$. To illustrate this, Figure 1 shows an NFA for $\Box(p \rightarrow \Diamond \lozenge q)$. B&M represent the NFA within the planning domain using one fluent per automaton state. In the example of Figure 1, this means that the new planning problem contains fluents $E_{q_0}$ and $E_{q_2}$. The translation is such that if $\alpha$ is a sequence of actions that induces the execution trace $\sigma = s_1 \ldots s_n$, then $E_{q_0}$ is true in $s_n$, iff there is some run of the automaton over $\sigma$ that ends in state $q$. B&M’s translation has the following property.

**Theorem 1 (Follows from (Baier 2010))** Let $P$ be a classical planning problem, $\phi$ be a finite LTL formula, and $P'$ be the instance that results from applying the B&M translation to $P$. Moreover, let $\alpha$ be a sequence of actions applicable in the initial state of $P$, and let $\sigma$ and $\sigma'$ be, respectively, the sequences (of planning) states induced by the execution of $\alpha$ in $P$ and $P'$. Finally, let $A_{\phi}$ be the NFA for $\phi$. Then the following are equivalent statements.

1. There exists a run $\rho$ of $A_{\phi}$ ending in $q$.
2. $E_{q}$ is true in the last state of $\sigma'$.

As a corollary of the previous theorem, one obtains that satisfaction of finite LTL formulas can be determined by checking whether or not the disjunction $\bigvee_{j \in F} E_j$ holds, where $F$ denotes the set of final states of $A_{\phi}$.

Unfortunately, B&M’s translation is worst-case exponential (Baier 2010); for example, an NFA for $\bigwedge_{i=1}^n \Diamond p_i$ has $2^n$ states. Baier (2010) proposes a formula-partitioning technique that allows the method to generate more compact translations for certain formulas. The method, however, is not applicable to any formula.

Edelkamp’s approach is similar to B&M’s: it builds a Büchi automaton (BA), whose states are represented via fluents, compactly representing all runs of the automaton in a single planning state. The main difference is that the state of the automaton is updated via specific actions—a process that they call synchronized update. We modify this idea in the compilation we give below; however, our compilation is significantly different since it does not represent all runs of the automaton in the same planning state. It is important to remark that the use of BA interpreted as NFA does not yield a correct translation for general LTL goals, although it is correct for the PDDL3 subset of LTL (De Giacomo, Masellis, and Montali 2014).

**Alternating Automata and Finite LTL**

A central part of our approach is the generation of an AA from an LTL formula. To do this we modify Muller, Saoudi, and Schupp’s AA (1988) for infinite LTL formulas. Our AA is equivalent to a recent proposal by De Giacomo, Masellis, and Montali (2014). The main difference between our construction and De Giacomo, Masellis, and Montali’s is that we do not assume a distinguished proposition becomes true only in the final state. On the other hand, we require a special state ($q_F$) that indicates the sequence should finish. The use of such a state is the main difference between our AA for finite LTL and Muller, Saoudi, and Schupp’s AA for infinite LTL.

We require the LTL input formula to be written in negation normal form (NNF); i.e., a form in which negations can be applied only to atomic formula. This transformation can be done in linear time (Gerth et al. 1995).
Let $\varphi$ be in $\mathcal{LTL}(F)$ and $\text{sub}(\varphi)$ be the set of the subformulas of $\varphi$, including $\varphi$. We define $A_{\varphi} = (Q, 2^F, \delta, q_0, \{q_F\})$, where $Q = \{q_n \mid \alpha \in \text{sub}(\varphi)\} \cup \{q_F\}$ and:

$$\delta(q_n, s) = \begin{cases} \top, & \text{if } \ell \in \text{Lit}(F) \text{ and } s \models \ell \\ \bot, & \text{if } \ell \in \text{Lit}(F) \text{ and } s \not\models \ell \\ \end{cases}$$

$$\delta(q_F, s) = \bot$$

$$\delta(q_{0 \land \beta}, s) = \delta(q_0, s) \land \delta(\beta, s)$$

$$\delta(q_{0 \lor \beta}, s) = \delta(q_0, s) \lor \delta(\beta, s)$$

$$\delta(q_0, s) = q_0$$

$$\delta(q_{\bullet}, s) = q_F \lor q_0$$

$$\delta(q_{0 \land \beta}, s) = \delta(q_0, s) \land (\delta(q_\beta, s) \lor q_0 \land \beta)$$

$$\delta(q_{0 \lor \beta}, s) = \delta(q_\beta, s) \lor (\delta(q_0, s) \lor q_0 \land \beta)$$

Theorem 2 Given an LTL formula $\varphi$ and a finite sequence of states $\sigma$, $A_{\varphi}$ accepts $\sigma$ iff $\sigma \models \varphi$.

Proof sketch: Suppose that $\sigma = x_1 x_2 \ldots x_n \in \sigma^*$. Then it can be proven by induction on the construction of $\varphi$ the following lemma: $\sigma, i \models \varphi$ if and only if exists a sequence $r = Q_1 \ldots Q_n$, such that: $Q_1 = \{\varphi\}$, $Q_\alpha \subseteq \{q_\beta\}$, for each subset $Q_j$ in the sequence $r$ it holds that $Q_j \subseteq \text{sub}(\varphi) \cup \{q_\beta\}$ and for each $j \in \{i, i + 1, \ldots, n\}$ it holds that $Q_j \models \delta(Q_{j-1}, x_j)$.

Compiling Away finite LTL Properties

Now we propose an approach to compiling away finite LTL properties using the AA construction described above.

First, we argue that the idea underlying both Edelkamp’s and B&M’s translations would not yield an efficient translation if applied to AA. Recall in both approaches if $E_{q_1}, \ldots, E_{q_n}$ are true in a planning state $s$, then there are $n$ runs of the automaton, each of which ends in $q_1, \ldots, q_n$ (Theorem 1). In other words, the planning state keeps track of all of the runs of the automaton. To apply the same principle to AA, we would need to introduce one fluent for each subset of states of the AA, therefore generating a number of fluents exponential on the size of the original formula. This is because runs of AA are sequences of sets of states, so we would require states of the form $E_R$, where $R$ is a set of states.

To produce an efficient translation, we renounce the idea of representing all runs of the automaton in a single planning state. Our translation will then only keep track of a single run.

Translating LTL via LTL Synchronization

Our compilation approach takes as input an LTL planning problem $P$ and produces a new planning problem $P'$, which is is like $P$ but contains additional fluents and actions. Like previous compilations, $A_{\varphi}$ is represented in $P'$ with additional fluents, one for each state of the automaton for $G$. Like in Edelkamp’s compilation $P'$ contains specific actions—below referred to as synchronization actions—whose only purpose is to update the truth values of those additional fluents. A plan for $P'$ alternates one action from the original problem $P$ with a number of synchronization actions. Unlike any other previous compilation, $P'$ does not represent all possible runs of the automaton in a single planning state.

Synchronization actions update the state of the automaton following the definition of the $\delta$ function. The most notable characteristic that distinguishes synchronization from the Edelkamp’s translation is that non-determinism inherent to the AA is modeled using alternative actions, each of which represents the different non-deterministic options of the AA. As such if there are $n$ possible non-deterministic choices, via the applications of synchronization actions there will be $n$ reachable planning states, each representing a single run.

Given a planning problem $P = (F, O, I, G)$, our translation generates a problem $P'$ in which there is one (new) fluent $q$ for each state $q$ of the AA $A_G$. The compilation is such that the following property holds: if $\alpha = a_1 a_2 \ldots a_n$ is applicable in the initial state of $P$, then there exists a set $A_\alpha$ of action sequences of the form $q_0 a_1 q_1 a_2 q_2 \ldots a_n q_n$, where each $a_i$ is a sequence of synchronization actions whose sole objective is to update the fluents representing $A_G$’s state.

Our theoretical result below says that our compilation can represent all runs, but only one run at a time. Specifically, each of the sequences of $A_\alpha$ corresponds to some run of $A_G$ over the state sequence induced by $\alpha$ over $P$. Moreover, if $\alpha' \in A_\alpha$, $q_\alpha$ is true in the state resulting from performing sequence $\alpha'$ in $P'$ iff $q$ is contained in the last element of a run that corresponds to $\alpha'$.

We are ready to define $P'$. Assume the AA for $G$ has the form $A_G = (Q, \Sigma, \delta, q_0, \{q_F\})$.

Fluents $P'$ has the same fluents as $P$ plus fluents for the representation of the states of the automaton $(Q)$, flags for controlling the different modes (copy, sync, world), and a special fluent $\text{ok}$, which becomes false if the goal has been falsified. Finally, it includes the set $Q^S = \{q^S \mid q \in Q\}$ which are “copies” of the automata fluents, which we describe in detail below. Formally, $F' = F \cup Q \cup Q^S \cup \{\text{copy, sync, world, ok}\}$.

The set of operators $O'$ is the union of the sets $O_w$ and $O_s$.

World Mode Set $O_w$ contains the same actions in $O$, but preconditions are modified to allow execution only in “world mode”. Effects, on the other hand are modified to allow the execution of the $\text{copy}$ action, which initiates the synchronization phase, and which is described below. Formally, $O_w = \{a' \mid a \in O\}$, and for all $a'$ in $O_w$: $\text{prec}(a') = \text{prec}(a) \cup \{\text{ok, world}\}$, $\text{eff}(a') = \text{eff}(a) \cup \{\text{copy, -world}\}$.

Synchronization Mode The synchronization mode can be divided in three consecutive phases. In the first phase, we execute the $\text{copy}$ action which in the successor states adds a copy $q^S$ for each fluent $q$ that is currently true, deleting $q$. Intuitively, during synchronization, each $q^S$ defines the state of the automaton prior to synchronization. The precondition
<table>
<thead>
<tr>
<th>Sync Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans($q^S_0$)</td>
<td>{sync, ok, $q^S_0$}</td>
<td>{$q^S_0$, ¬ok}</td>
</tr>
<tr>
<td>trans($q^S_{αβ}$)</td>
<td>{sync, ok, $q^S_{αβ}$}</td>
<td>{$q^S_{αβ}$, ¬$q^S_{αβ}$}</td>
</tr>
<tr>
<td>trans($q^F_0$)</td>
<td>{sync, ok, $q^F_0$}</td>
<td>{$q^F_0$, ¬$q^F_0$}</td>
</tr>
<tr>
<td>trans($q^F_{αβ}$)</td>
<td>{sync, ok, $q^F_{αβ}$}</td>
<td>{$q^F_{αβ}$, ¬$q^F_{αβ}$}</td>
</tr>
<tr>
<td>trans($q^R_0$)</td>
<td>{sync, ok, $q^R_0$}</td>
<td>{$q^R_0$, ¬$q^R_0$}</td>
</tr>
<tr>
<td>trans($q^R_{αβ}$)</td>
<td>{sync, ok, $q^R_{αβ}$}</td>
<td>{$q^R_{αβ}$, ¬$q^R_{αβ}$}</td>
</tr>
<tr>
<td>trans($q^R_{αβ}$)</td>
<td>{sync, ok, $q^R_{αβ}$}</td>
<td>{$q^R_{αβ}$, ¬$q^R_{αβ}$}</td>
</tr>
<tr>
<td>trans($q^R_{αβ}$)</td>
<td>{sync, ok, $q^R_{αβ}$}</td>
<td>{$q^R_{αβ}$, ¬$q^R_{αβ}$}</td>
</tr>
<tr>
<td>trans($q^R_{αβ}$)</td>
<td>{sync, ok, $q^R_{αβ}$}</td>
<td>{$q^R_{αβ}$, ¬$q^R_{αβ}$}</td>
</tr>
</tbody>
</table>

Table 1: The synchronization actions for LTL goal $G$ in NNF. Above $ℓ$, $α R β$, $α U β$, and $□α$ are assumed to be in the set of subformulas of $G$. In addition, $ℓ$ is assumed to be a literal.

of $copy$ is simply $\{ copy, ok \}$, while its effect is defined by:

$eff(copy) = \{ q \rightarrow q^S, q \rightarrow \neg q \mid q \in Q \} \cup \{ \text{sync, } \neg \text{copy} \}$

As soon as the sync fluent becomes true, the second phase of synchronization begins. Here the only executable actions are those that update the state of the automaton, which are defined in Table 1. Note that one of the actions deletes the ok fluent. This can happen, for example while synchronizing a formula that actually expresses the fact that the action sequence has to conclude now.

When no more synchronization actions are possible, we enter the third phase of synchronization. Here only action world is executable; its only objective is to reestablish world mode. The precondition of world is $\{ \text{sync, ok} \} \cup Q^S$, and its effect is $\{ \text{world}, \neg \text{sync} \}$.

The set $O_s$ is defined as the one containing actions $copy$, world, and all actions defined in Table 1.

New Initial State The initial state of the original problem $P$ intuitively needs to be “processed” by $A_G$ before starting to plan. Therefore, we define $I'$ as $I \cup \{ q_G, copy, ok \}$.

New Goal Finally, the goal of the problem is to reach a state in which no state fluent in $Q$ is true, except for $q_f$, which may be true. Therefore, we define $G' = \{ \text{world, ok} \} \cup (Q \setminus \{ q_F \})$.

Properties There are two important properties that can be proven about our translation. First, our translation is correct.

**Theorem 3 (Correctness)** Let $P = \langle F, O, I, G \rangle$ be a planning problem with an LTL goal and $P' = \langle F', O', I', G' \rangle$ be the translated instance. Then $P$ has a plan $α_0α_1α_2...α_n$ iff $P'$ has a plan $α_0α_1α_2...α_nα_0$ in which for each $i \in \{1, 2, \ldots, n\}$, $α_i$ is a sequence of actions in $O_s$.

**Proof sketch:** We show each sequence of actions $α_i$ simulates the behavior of the automata, i.e., whenever $I$ is a planning state whose next action must be $copy$ and $q_β \in t$, then $ρ(t, α_i)$ satisfies $δ(q_β, t)$.

For this, let’s define $t^S$ as the subset of all the automata fluents $Q^S$ that are added during the execution of the sequence of actions $α_i$. We will prove the following lemma by induction on the construction of $φ$: If $q^S_φ \in t^S$, then $ρ(t, α_i) \models δ(q_φ, t)$.

Observe that if $q^S_φ \in t^S$, then there must be an action $trans(q^S_φ)$ that was executed in $α_i$. This is because $ρ(t, α_i) \cap Q^S = ∅$ and only $trans(q^S_φ)$ can delete $q^S_φ$ from the current state. The second observation is: If some action $trans$ adds $q^S_φ$, then $q^S_φ \in t^S$. This is by definition of $t^S$. If the action adds $q^S_ψ$, then $q^S_ψ \in ρ(t, α_i)$, because the only action that deletes fluents in $Q$ is $copy$.

- $φ = ℓ$. Assume $ℓ$ is positive literal. Then there is a planning state $s$ in which $trans(q^S_φ)$ was executed. Since the precondition requires $ℓ \in s$ and $ℓ$ can only be added by an action from $O_w$, then $ℓ \in t$. By definition, $δ(q_φ, t) = T$, and it is clear that $ρ(t, α_i) \models δ(q_φ, t)$. The argument is analogous for a negative literal $ℓ$.

We will not consider the case for $q_F$. It is never desirable to synchronize that state, because the special fluent $ok$ is removed, leading to a dead end. Now, assume that $q^S_φ \in t^S$ implies $ρ(t, α_i) \models δ(q_φ, t)$ for every $φ$ with less than $m$ operators. The proof sketch for each case can be verified by the reader as follows:

- For each $φ$, it is clear that a version of $trans(q^S_φ)$ is executed due to the first observation.
- If $q_φ$ is added by $trans$, then $q_ψ \in ρ(t, α_i)$ due to the second observation. This implies that $ρ(t, α_i) \models q_ψ$.
- If $q^S_φ$ is added by $trans$, then $q^S_ψ \in t^S$. By induction hypothesis, $ρ(t, α_i) \models δ(q_ψ, t)$, because $α$ is a strict subformula of $φ$ and has less than $m$ operators.
- Finally, using entailment (for positive boolean formulae) and the definition of the transitions for the alternating automata $A_G$, it can be verified that $ρ(t, α_i) \models δ(q_ψ, t)$.

The argument is similar for the other versions of $trans$.

To conclude our theorem, note that if $t$ is a planning state, $q_β \in t$ and the next action to execute is $copy$, then $q^S_β \in t^S$. Using the lemma, this implies $ρ(t, α_i) \models δ(q_β, t)$.

Second, the size of the plan for $P'$ is linear on the size of the plan for $P$.

**Theorem 4 (Bounded synchronizations)** If $T$ is a reachable planning state from $I'$ and $T' \cap Q^S = \emptyset$, then there is a sequence of trans actions $α$ such that $δ(T, copy·α) \cap Q^S = \emptyset$ and $|α| \in O(G)$.

**Proof:** Note that $T$ is a state in world mode getting ready to go into synchronization mode after the $copy$ action has been executed. The main idea of the proof is to choose the order of the subformulae to be synchronized, where the first one corresponds to the largest subformula of the current state, the second one corresponds to the second largest subformula and so on. Note that when an action $trans(q^S_β)$ is executed, it always happens that at most two fluents $q^S_β$ and $q^S_γ$ are added, and the formulae $β$ and $γ$ are strict subformulae of $α$. This means that a subformula will never get synchronized twice in a single synchronization phase $σ$. Since the number
of subformulas is linear on $|G|$, this means that the length of $\sigma$ must be $O(|G|)$. ■

Towards More Efficient Translations

The translation we have presented above can be modified slightly for obtaining improved performance. The following are modifications that we have considered.

An Order for Synchronization Actions

Consider the goal formula is $\alpha \land \beta$ and that currently both $q_a$ and $q_\beta$ are true. The planner has two equivalent ways of completing the synchronization: by executing first $\text{trans}(q_a)$ and then $\text{trans}(q_\beta)$, or by inverting this sequence. By enforcing an order between these synchronizations, we can reduce the branching factor at synchronization phase. Such an order is simple to enforce by modifying preconditions and effects of synchronization actions so that states are synchronized following a topological order of the parse tree of $G$.

Positive Goals

The goal condition of the translated instance requires being in and every $q \in Q$ to be false. On the other hand, action copy, which has to be performed after each world action, has precisely the effect of making every $q \in Q$ false. This may significantly hurt performance if search relies on heuristics that relax negative effects of actions, like the FF heuristic (Hoffmann and Nebel 2001), which is key to the performance of state-of-the-art planning systems (Richter and Helmer 2009). To improve heuristic guidance, we define a new fluent $q^D$, for each $q \in Q$, with the intuitive meaning that $q^D$ becomes true when $\text{trans}(q)$ cannot be executed in the future. For every action $\text{trans}(q_a^S)$ that does not add $q_a$, we include the conditional effect $\{q_\beta | \beta \in \text{super}(\alpha)\} \rightarrow q^D_\beta$, where $\text{super}(\alpha)$ is the set of subformulas of $G$ that are proper superformulas of $\alpha$. Using a function $f$ that takes a $\text{LTLL}(F)$ formula and generates a propositional formula, the new goal $f(G)$ can be recursively written as follows:

- If $\varphi = p$ and $p \in \text{Lit}(F)$, then $f(p) = q^D_p$.
- If $\varphi = \alpha \land \beta$, then $f(\varphi) = f(\alpha)^D \land f(\beta)$.
- If $\varphi = \alpha \lor \beta$, then $f(\varphi) = \neg f(\alpha)^D \lor f(\beta)$.
- If $\varphi = \neg \beta$ or $\varphi = \blacksquare \beta$, then $f(\varphi) = q^D_{\neg \beta} \land f(\beta)$.
- If $\varphi = \alpha * \beta$, where $* \in \{U, R\}$, then $f(\varphi) = q^D_{\alpha \ast} \land f(\beta)$.

Empirical Evaluation

The objective of our evaluation was to compare our approach with existing translation approaches, over a range of general LTL goals, to understand when it is convenient to use one or other approach. We chose to compare to B&M’s rather than Edelkamp’s because the former seems to yield better performance (Baier, Bacchus, and McIlraith 2009). We do not compare against other existing systems that handle PDDL3 natively, such as LPRPG-P (Coles and Coles 2011), because efficient translations for the (restricted) subset of LTL of PDDL3 into NFA are known (Gerevini et al. 2009).

We considered both LAMA (Richter, Helmer, and Westphal 2008) and FF$_X$ (Thiébaut, Hoffmann, and Nebel 2005), because both are modern planners supporting derived predicates (required by B&M). We observed that LAMA’s preprocessing time where high, sometimes exceeding planning time by 1 to 2 orders of magnitude, and thus decided to report results we obtained with FF$_X$. We used an 800MHz-CPU machine running Linux. Processes were limited to 1 GB of RAM and 15 min. runtime.

There are no planning benchmarks with general LTL goals, so we chose two of the domains (rovers and openstacks) of the 2006 International Planning Competition, which included LTL preferences (but not goals), and generated our own problems, with some of our goals inspired by the preferences. In addition, we considered the blocksworld.

Our translator was implemented in SWI-Prolog. It takes a domain and a problem in PDDL with an LTL goal as input and generates PDDL domain and problem files. It also receives an additional parameter specifying the translation mode which can be any of the following: simple, OSA, PG, and OSA+PG, where simple is the translation of Section , and OSA, PG are the optimizations described in Section . OSA+PG is the combination of OSA and PG.

Table 2 shows a representative selection of the results we obtained. It shows translation time (TT), plan length (PL), planning time (PT), the number of planning states that were evaluated before the goal was reached (PS). Times are displayed in seconds. For our translators we also include the length of the plan without synchronization actions (WPL). NR means the planner/translator did not return a plan.

We observe mixed results. B&M yields superior results on some problems: for example p01 and p03 of openstacks, which are of the form $\bigwedge_{i=1}^n \lhd p_i$, where $n = 3$ and $n = 5$ for p01 and p03, respectively. The performance gap is probably due to the fact that (1) the B&M problem requires fewer actions in the plan and (2) B&M’s output for these goals is quite compact on the size of the formula. On the other hand, there are other goal formulas in which our approach outperforms B&M. For example, p09 and p11 in openstacks, which are of the form: $\bigvee_{i=1}^n \lhd p_i$, where $n = 5$ and $n = 4$ for p09 and p11, respectively. Even though the outer $\bigvee$ can be removed yielding an equivalent formula, B&M generates an output exponential in $n$, which results in higher translation time and eventually in the planner running out of memory.

By observing the rest of the data, we conclude that B&M returns an output that is significantly larger than our approaches for the following classes of formulas: $\alpha \bigvee_{i=1}^n \lhd p_i$, $\alpha \bigwedge_{i=1}^n \lhd p_i \cup \gamma_i$, $(\bigvee_{i=1}^n \lhd \alpha_i) \cup \beta$, and $(\bigvee_{i=1}^n \lhd \alpha_i \bigvee R \beta_i) \cup \beta$, with $n \geq 4$, yielding finally an “NR”. Being polynomial, our translation handles these formulas reasonably well: low translation times, and a compact output. In many cases, this allows the planner to return a solution.

The use of positive goals has an important influence in performance possibly because the heuristic is more accurate, leading to fewer expansions. OSA, on the other hand, seems to negatively affect planning performance in FF$_X$. The reason is the following: FF$_X$ will frequently choose the wrong synchronization action and therefore its enforced hill climbing algorithm will often fail. This behavior may not be observed in planners that use complete search algorithms.
<table>
<thead>
<tr>
<th>Rovers Domain</th>
<th>Openstacks Domain</th>
<th>Blocksworld Domain</th>
<th>Conclusions</th>
<th>References</th>
</tr>
</thead>
</table>


A Unifying Framework for Planning with LTL and Regular Expressions

Eleni Triantafillou
Department of Computer Science
University of Toronto
Toronto, Canada

Jorge A. Baier
Depto. de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile

Sheila A. McIlraith
Department of Computer Science
University of Toronto
Toronto, Canada

Abstract
Temporally extended goals are critical to the specification of many real-world planning problems. Such goals are typically specified in the subset of Linear Temporal Logic (LTL) found in the Planning Domain Description Language, PDDL3.0. In this paper, we propose LTL-RE, a high-level language that supports the specification of a wide variety of temporal goals, not only using LTL but also using regular expressions. LTL-RE derives its formal foundation from finite Linear Dynamic Logic (LDLf), and its expressive power is no less than that of regular expressions. LTL-RE augments LDLf with planning-friendly syntax including LTL and typical programming language constructs. It is also designed for use with AI automated planning transition systems, supporting both state-, action-, and path-oriented temporal goal specification. Building on recent work focused on LTL, we propose a translation of LTL-RE into Alternating Automata, which are then embedded directly in domain descriptions for use with classical planners. We evaluate the behavior of our translator and the resultant planning problems, with comparison to alternative LTL translators.

1 Introduction
Most real-world planning problems involve complex goals that are temporally extended, necessitate the optimization of preferences or other quality measures, require adherence to safety constraints and directives, and/or may require or benefit from following a prescribed high-level script that specifies how the task is to be realized. By way of illustration, consider a logistics company that transports packages. Package delivery may be governed by the following types of (possibly inconsistent) goals:

- Always ship frozen food in a refrigerated truck.
- Prefer to deliver priority packages before regular packages.
- If the customer is a preferred customer, then always apply a 15% discount to the final bill.
- Prefer to deliver domestic packages within 24 hours of receipt.
- While a truck is at a location that is not full, load all packages bound for a different destination on the truck; drive to the next destination; unload all packages to be delivered to this destination.

While some forms of non-classical goal specification were initially realized via special-purpose planners such as the Hierarchical Task Network (HTN) planner SHOP2 (e.g., (Erol, Hendler, and Nau 1994)) or TLPLAN, the pioneering planning system that accepts Linear Temporal Logic (LTL) pruning rules (Baçküs and Kabanza 1998), more recent efforts have focused on incorporating such non-classical goals, which include both temporally extended goals and preferences, into state-of-the-art domain independent planners (e.g., (Rintanen 2000; Doherty and Kvarnström 2001; Cresswell and Coddington 2004; Edelkamp 2006; Baier and McIlraith 2006; Benton, Kambhampati, and Do 2006; Baier, Fritz, and McIlraith 2007; Coles and Coles 2011; Lago, Pistore, and Traverso 2002)). Such systems have been used in service of a diversity of planning and non-planning applications from genom rearrangement (Uras and Erdem 2010) and program test generation (Razavi, Farzan, and McIlraith 2014) to story generation (Haslum 2012), automated diagnosis (Grastien et al. 2007; Sohrabi, Baier, and McIlraith 2010), and verification (Albarghouthi, Baier, and McIlraith 2009; Patrizi et al. 2011). In recognition of the planning community’s need for non-classical planning objectives, PDDL3.0 (Gerevini et al. 2009) was designed to capture a useful subset of LTL constraints which can either be cast as hard constraints on the plan or aggregated in a weighted sum to construct an objective function of soft constraints for optimization in the context of plan generation.

The provision of planning systems that accept temporally extended goals is at the heart of synergies between the AI Automated Planning community and the Model Checking community, where similar types of constraints are used to specify safety and liveness properties for software and hardware verification, and to specify target behavior for the synthesis of software and controllers. Historically the model checking community has specified such properties in a temporal logic such as LTL or one of its branching time counterparts – CTL or CTL*. Such temporal logics are very good at specifying state-centric properties but they don’t provide a natural vehicle for specifying action-centric properties – procedural properties involving the actions of a domain.

In a series of well-received lectures between 2011 and 2013, Moshe Vardi advocated convincingly for both the benefits of LTL, but also for its limitations in the context of industry-driven verification tasks (e.g., (Vardi 2012)). In response, Vardi advocated for Linear Dynamic Logic (LDL), a temporal logic that combines LTL and Regular Expressions (REs) in a manner that avoids the exponential blowup that typically plagues REs in such a context. Subsequently, De Giacomo and Vardi (2013) proposed LDLf, which defines

---

1A classical planning goal is limited to a conjunction of properties that must hold in the final state.
LDL over finite traces, citing automated planning among the applications for the logic.

While the AI planning community has increasingly studied state-centric path constraints in the form of temporally extended goals (TEGs), there has been far less examination of temporally extended goals that take the form of REs. An exception to this is the work of Baier, Fritz, and McIraith (2007; 2008) which supports planning with action-centric procedural control/goals in a Golog-like language that captures the syntax of REs. A second exception is the work by Shaparau, Pistore, and Traverso (2008) which, building on previous work on the EAGLE goal language (Lago, Pistore, and Traverso 2002), also provides a form of REs for temporally extended goal specification.

In this paper we propose a goal specification language, LTL-RE, that supports both LTL and REs. However, unlike previous work noted above, it has its formal underpinnings in one uniform language – LDLf – capturing the semantics of LDLf while at the same time augmenting LDLf with further syntax that we believe is more compelling to an end user charged with specifying goals or constraints for plan generation. LTL-RE is able to specify goals with respect to planning domain actions as well as state properties in the form of REs, using compelling programming constructs such as “if then else” and “while” loops. We define the syntax and semantics of LTL-RE and examine how to plan for LTL-RE goals using state-of-the-art domain independent planners via a reformulation into finite state automata. Unlike previous reformulation approaches that exploited Non-deterministic Finite State Automata (NFAs) (e.g., (Baier and McIraith 2006)), we exploit an approach based on Alternating Automata following Torres and Baier (2015) that avoids the worst-case exponential blow-up inherent to NFAs. This workshop paper represents a work in progress. We present our algorithm and report on experimental results to date, contrasting the efficacy of our reformulation to LTL-specific LDL reformulations based on NFAs and Alternating Automata.

2 Preliminaries

In this section we recount how transition systems are compactly described using a planning language and review the syntax and semantics of LTL, LDL and their finite trace counterparts, LTLf and LDLf.

2.1 A Planning-Language Transition System

The objective of this paper is to show how to plan for a rich goal language based on LTL and REs uniformly captured in LDLf. We assume that the “world” for which we want to build our plan is described by a deterministic transition system compactly described by an initial state and a set of actions. In AI automated planning such a transition system is typically specified using the Planning Domain Description Language (PDDL) of which there are several variants of differing expressivity (McDermott 1998).

Formally, a transition system is given by a tuple \((P, A, \mathcal{I})\), where \(P\) is a set of propositions, which we use to describe a state, \(A\) is a set of actions, and \(\mathcal{I} \subseteq P\) is the initial state. For every action \(a \in A\), \(prec(a)\) and \(eff(a)\) denote, respectively, the preconditions and effects of \(a\). \(prec(a)\) is a set of fluent literals over \(P\) and \(eff(a)\) is a set of elements of the form \(C \rightarrow L\), where \(C\) is a set of literals over \(P\) and \(L\) is a literal over \(P\). When \(C \rightarrow L\) is an effect of \(a\) and \(a\) is applied on a state \(s\) in which \(C\) holds, then \(L\) must hold in the state that results from applying \(a\) in \(s\).

An action \(a\) is applicable in a state \(s \subseteq 2^P\) if \(s \models prec(a)\).

If \(a\) is applicable in \(s\), then partial function \(\delta : 2^2^P \times A\) is defined such that:

\[
\delta(s, a) = s' \{ p \mid C \rightarrow \neg p \in eff(a) \} \cup \{ p \mid C \rightarrow p \in eff(a) \}
\]

If \(a\) is not applicable in \(s\), then \(\delta(a, s)\) is undefined.

A sequence of actions \(a_0a_1 \ldots a_{n-1}\) is applicable in \(s_0\) if \(\delta(s_i, a_i)\) is defined for each \(i \in \{0, \ldots, n-1\}\). A state trace \(s_0s_1 \ldots s_{n+1}\) is induced by the execution of \(\alpha = a_0a_1 \ldots a_n\) in a state \(s\) iff (1) \(\alpha\) is applicable in \(s\), (2) \(s = s_0\), and (3) \(\delta(s_i, a_i) = s_{i+1}\), for every \(i \in \{0, \ldots, n - 1\}\).

2.2 From Propositional Dynamic Logic to LDL

Propositional Dynamic Logic (PDL) was introduced by Fischer and Ladner (1979) to describe interesting properties of programs, such as correctness and termination. In PDL, terms are actions and propositions, and modal operators are exploited to directly reference regular programs within the language. This allows, for instance, the use of a test operator which results in the blocking of the program if the property that is being tested is false. It also supports nondeterministic choice of actions, sequencing of actions in a program, and the repetition of a program for a nondeterministic number of iterations. Within PDL, it is also possible to define programming constructs such as "if then else" and "while do".

LDL is an extension of PDL, which carries over the rich expressive properties of PDL but interpreted with respect to linear traces, just as LTL is used in planning and model checking to express interesting state-centric properties of linear traces (e.g., TEGs). LDL is a logic that is expressively equivalent to Monadic Second Order Logic (MSO), and is strictly more powerful than first order logic (FOL), or equivalently, LTL. In their 2013 paper, De Giacomo and Vardi argue that LDLf witnesses the marriage of the best properties of REs on finite traces (REf) and LTLf, namely the rich expressivity of RE with the declarative convenience LTLf (De Giacomo and Vardi 2013). Since, however, in our opinion LDLf is still not a very intuitive specification mechanism, we augment it with syntax to allow for a more intuitive expression of temporal and dynamic properties and clarify its use in the context of a transition system expressed via a planning language, such as PDDL.

2.3 LTLf and LDLf

LTLf: LTL is a modal temporal logic, first proposed for verification (Pnueli 1977). It supports the expression of rich path properties using modalities that include always (\(\Box\)), eventually (\(\Diamond\)), until (\(U\)), and next (\(O\)). These temporal modalities can be arbitrarily nested over well-formed formulae defined over standard logical constructs such as \(\neg, \lor, \land\), etc. LTLf is a finite variant of LTL that has been used extensively for
the specification of TEGs in automated planning. Below we review the semantics of LTLf.

Given a finite trace \( \pi \) over an alphabet \( \mathcal{P} \), and an instant, \( i \) of the trace, the LTLf operators are defined below.

- \( \pi, i \models \bigcirc \varphi \iff i < \text{last} \land \pi, i + 1 \models \varphi \)
- \( \pi, i \models \varphi_1 \cup \varphi_2 \iff \) for some \( j \) such that \( i \leq j \leq \text{last} \), we have that \( \pi, j \models \varphi_2 \) and for all \( k, i \leq k < j \), we have that \( \pi, k \models \varphi_2 \)

The operators \( \square \) and \( \diamond \) can be defined in terms of the above modal operators. Intuitively, \( \square \phi \) denotes that formula \( \phi \) holds in every state of the trace from the current instant forward, while \( \diamond \phi \) denotes that \( \phi \) will hold at some instant in the subtrace from the current instant forward. More formally,

- \( \pi, i \models \diamond \varphi \iff \pi, j \models \varphi \land i < j \leq \text{last} \)
- \( \pi, i \models \square \varphi \iff \pi, j \models \varphi \land i < j < \text{last} \)

LTLf - LDLf (De Giacomo and Vardi 2013) features the properties of LDL, but formulae are evaluated over finite linear traces. The syntax of LDLf is defined as follows:

\[
\begin{align*}
\varphi & ::= A \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \langle \rho \rangle \varphi \\
\rho & ::= \phi \mid \varphi_1^? \mid \rho_1 \mid \rho_2 \mid \rho^* \\
\end{align*}
\]

where \( A \) denotes atomic propositions, \( \phi \) denotes a propositional formula over atomic propositions, \( \rho \) denotes path expressions, which are regular expressions over propositional formulas \( \phi \), together with the test construct \( ? \). Finally, \( \varphi \) denotes LDLf formulas formed by applying Boolean connectives, combined with the modal operator \( \langle \rho \rangle \).

The modal operator \( \langle \rho \rangle \varphi \) evaluates to \( \mathit{true} \) in a state if there exists a trace, starting from the current state, which satisfies \( \rho \) and ends in a state which satisfies \( \varphi \). Its dual operator, \( \rho \varphi \), which can be defined as \( \neg \langle \rho \rangle \neg \varphi \), is \( \mathit{true} \) in a state if all traces starting from that state which satisfy \( \rho \) end in a state that satisfies \( \varphi \).

For a given finite trace \( \pi \) over an alphabet \( \mathcal{P} \), and an instant, \( i \), of the trace, \( i \in \{0, \ldots, \text{last}\} \), we inductively define what it means for an LDLf formula \( \varphi \) to be \( \mathit{true} \), i.e.

- \( \pi, i \models A \iff A \in \pi(i) \)
- \( \pi, i \models \neg \varphi \iff \pi, i \not\models \varphi \)
- \( \pi, i \models \varphi \land \varphi' \iff \pi, i \models \varphi \) and \( \pi, i \models \varphi' \)
- \( \pi, i \models \langle \rho \rangle \varphi \iff \) for some \( j \) such that \( i \leq j \leq \text{last} \), we have that \( (i, j) \in R(\rho, \pi) \) and \( j \models \varphi \)

where the relation \( R(\rho, \pi) \) is defined inductively as follows:

- \( R(\phi, s) = \{(i, i + 1) \mid \pi(i) \models \phi\} \) (\( \phi \) propositional)
- \( R(\varphi?, s) = \{(i, i) \mid \pi, i \models \phi\} \)
- \( R(\rho_1 \cup \rho_2, s) = R(\rho_1, s) \cup R(\rho_2, s) \)
- \( R(\rho_1 \rho_2, s) = \{(i, j) \mid \exists k \text{ such that } (i, k) \in R(\rho_1, s) \text{ and } (k, j) \in R(\rho_2, s)\} \)
- \( R(\rho^*, s) = \{(i, i) \cup (i, j) \mid \exists k \text{ such that } (i, k) \in R(\rho, s) \text{ and } (k, j) \in R(\rho^*, s)\} \)

## 3 A Goal Language over LTL & REs

### 3.1 Overview

In this paper we propose Linear Temporal Logic with Regular Expressions for finite traces (LTL-RE)\(^2\), a high-level language for the specification of temporally extended goals that are evaluated over finite traces. This language functions as a unifying framework, by supporting syntax from LTL and LDL, programming constructs “if-then-else” and “while”, and a modality “final”, for expressing properties that must hold in the last state of a finite trace. It also supports direct reference to planning language actions from within LTL-RE through the use of a special predicate, “occ” which ranges over the ground actions in a planning problem description, \( \mathcal{A} \).

LTL-RE is as expressive as LDLf, which is equivalent to Monadic Second Order Logic. This is strictly more expressive than LTLf. This fact allows the definition of LTL operators, and the constructs “if then else”, “while” and “final” in terms of the syntax of LDLf, as we demonstrate in a following section.

### 3.2 The Syntax of LTL-RE

Given a transition system \((P, A, T)\), as defined in Section 2.1, the syntax of LTL-RE is given by the following grammar:

\[
\begin{align*}
\phi & ::= p : p \in P \mid \text{occ}(a) : a \in A \mid \neg \phi \mid \varphi_1 \land \varphi_2 \\
\varphi & ::= \phi \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \mathit{final} \phi \mid \langle \rho \rangle \varphi \mid \varphi_1 \cup \varphi_2 \mid \\
& \quad \square \varphi \mid \diamond \varphi \mid \langle \rho \rangle \varphi \mid \langle \rho \rangle \varphi_1 \land \varphi_2 \mid \\
& \quad \langle \rho \rangle \varphi_1 \cup \varphi_2 \mid \langle \rho \rangle \varphi_1 \land \varphi_2 \mid \langle \rho \rangle \varphi_1 \cup \varphi_2 \mid \\
& \quad \mathit{if-then-else}(\varphi, \rho_1, \rho_2) \mid \mathit{while}(\varphi, \rho) \\
\end{align*}
\]

With this syntax, we can express typical LTL goals such as “Always have your phone and eventually be at home.”

\( \square \mathit{have}(\text{Phone}) \land \diamond \mathit{at}(\text{Home}) \)

but also TEGs that take the form of regular expressions, such as “If it’s night time then take a taxi home, else take the subway.”

\( \mathit{if-then-else}((\mathit{night}, \text{occ}(\text{taxiHome})), \mathit{occ}(\text{subwayHome})) \)

### 3.3 Semantics

LTL-RE is interpreted over a pair \( \pi = (\sigma, \alpha) \), where \( \alpha \) is a sequence of actions, and \( \sigma \) is the sequence of states induced by the execution of \( \alpha \) in a certain state.

We say that \( \pi \models \varphi \), where \( \varphi \) is an LTL-RE formula, \( \pi = (\sigma, \alpha) \), \( \sigma = s_0 \ldots s_n \), and \( \alpha = a_0 \ldots a_{n-1} \) iff \( \pi, 0 \models \varphi \).

Now we assume we include the same definitions in LDLf’s semantics that were listed in the previous section, taking into account that \( \pi(i) \) now refers the the \( i \)-th state, i.e., \( s_i \). In addition we add the following rule for the occ operator:

- \( \pi, i \models \text{occ}(a) \) iff \( i < n \) and \( a_i = a \)

\(^2\)not to be confused with RELTL (e.g., (Eisner and Fisman 2007)).
This operator is what makes it possible to directly refer to actions within the language, and not merely through their effects on the state properties. Specifically, \( \text{occ}(a) \) is true in the current state, if \( a \) is the next planning action to be executed.

Now we define the semantics for the “if-then-else” and “while” programming constructs. Following (Fischer and Ladner 1979), we can express these constructs in terms of standard LDL operators.

- \( \pi, i \models \text{if-then-else}(\psi, \varphi_1, \varphi_2) \) if \( \pi, i \models \psi \land \varphi_1 \lor \neg \psi \land \varphi_2 \)
- \( \pi, i \models \text{while}(\psi, \varphi) \) if \( \pi, i \models (\psi \land \langle \psi; \varphi \rangle) \).

We also define the semantics for the LTL operators \( \bigcirc, \bigcirc_X, \bigcirc_U \) and \( U \) can be rewritten using the syntax of LDLf while preserving their semantics as defined in the previous section. This is shown in (De Giacomo et al. 2014).

- \( \pi, i \models \bigcirc \psi \iff \pi, i \models (\text{true}) \land \pi, i \models \psi \land \pi, i \models \text{i < last} \)
- \( \pi, i \models \bigcirc X \psi \iff \pi, i \models (\text{true}) \land \pi, i \models X \psi \land \pi, i \models \langle \psi; \text{true} \rangle \psi \)
- \( \pi, i \models \psi U \varphi \iff \pi, i \models (\langle \psi; \text{true} \rangle \varphi) \)

Finally, we define the semantics for the modality final:

- \( \pi, i \models \text{final} \varphi \iff \pi, i \models \varphi \land i = \text{last} \)

### 3.4 Planning for an LTL-RE Goal

We end this section by defining what it means to plan for an LTL-RE goal.

**Definition 3.1.** Let \( R = (P, A, I) \) be a transition system and \( \varphi \) be an LTL-RE formula. Then the sequence of action \( \alpha \) is a plan for \( \varphi \) over \( R \) if \( \alpha \) is applicable in \( I \) and generates a state trace \( \sigma \) over \( I \) such that \( (\sigma, \alpha) \models \varphi \).

### 4 Planning for LTL-RE Goals with Standard Planners

In this section we show how we can plan for LTL-RE goals using state-of-the-art planners. To this end, we use a two-step approach that follows (Torres and Baier 2015). In the first stage, we build an alternating automaton for the LTL-RE formula. Then, we show how this automaton can be used to compile the temporal logic into a non-temporal (final-state) goal. We do this by exploiting the fact that the dynamics of an alternating automaton can be encoded efficiently in a new transition system that is built from the original planning domain transition system.

#### 4.1 Alternating Automata on Words

An Alternating Automaton on Words (AA) on the alphabet \( 2^P \) is a tuple \( A = (2^P, Q, q_0, \delta, F) \), where \( Q \) is a finite nonempty set of states, \( q_0 \) is the initial state, \( F \) is a set of accepting states, and \( \delta \) is a transition function \( \delta : Q \times 2^P \to B^+(Q) \), where \( B^+(Q) \) is a set of positive Boolean formulas whose atoms are states of \( Q \).

A run of an AA \( A = (2^P, Q, q_0, \delta, F) \) over word \( w = b_1 \ldots b_n \) is a sequence of subsets of \( Q \), \( Q_0, Q_1, \ldots, Q_n \), such that \( Q_0 = \{q_0\} \), and \( Q_{i+1} = \delta(q, b_i) \), for every \( q \in Q_i \), and every \( i \in \{0, \ldots, n-1\} \). An AA accepts a word \( w \) if it has a run ending in a subset of \( F \).

### 4.2 From LDLf to AA

Following (Fischer and Ladner 1979; De Giacomo and Vardi 2013), the Fisher-Ladner Closure of a LDLf formula \( \varphi \) is a set \( CL_\varphi \) of LDLf formulas, recursively defined as follows:

- if-then-else
- \( \neg \psi \) if \( \psi \in CL_\varphi \) and \( \psi \) not of the form \( \neg \psi' \)
- \( \varphi_1 \land \varphi_2 \in CL_\varphi \) implies \( \varphi_1, \varphi_2 \in CL_\varphi \)
- \( \langle \rangle \psi \in CL_\varphi \) implies \( \psi \in CL_\varphi \)
- \( \neg \psi \in CL_\varphi \) implies \( \psi \in CL_\varphi \)
- \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \) implies \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \)
- \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \) implies \( \langle \varphi_1 \varphi_2 \rangle \in CL_\varphi \)
- \( \langle \varphi_1 \varphi_2 \rangle \in CL_\varphi \) implies \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \)
- \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \) implies \( \langle \varphi_1 \varphi_2 \rangle \in CL_\varphi \)
- \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \) implies \( \langle \varphi_1 \varphi_2 \rangle \in CL_\varphi \)
- \( \langle \varphi_1 \rangle \varphi_2 \in CL_\varphi \) implies \( \langle \varphi_1 \varphi_2 \rangle \in CL_\varphi \)

De Giacomo and Vardi define an AA which accepts all and only the traces that satisfy a given LDLf formula. Specifically, given a set of propositions \( P \), and a LDLf formula \( \varphi \) which is in Negation Normal Form (NNF), the automaton for \( \varphi \) that is defined in (De Giacomo and Vardi 2013) is given by: \( A_\varphi = (2^P, CL_\varphi, \varphi, [\{\}], \langle \rangle \) \), where \( 2^P \) is the alphabet, \( CL_\varphi \), denoting the Fisher-Ladner Closure of the formula \( \varphi \) is the state set, and \( \langle \rangle \) is their transition function.

We define an automaton \( A_\varphi \) for a set of propositions \( P \) and a LDLf formula in NNF \( \varphi \) as follows: \( A_\varphi = (2^P, CL_\varphi \cup \{q_f\}, \varphi, \delta, \{q_f\}) \), where \( q_f \) is a special automaton state in which the AA transitions when the trace has ended. Also, \( \delta \) is the transition function, which differs from the aforementioned \( \delta' \) in the transition for \( \langle \varphi \rangle \), as we elaborate on later. For an interpretation \( I \), and assuming that \( A \) stands for a propositional formula, we define \( \delta \) below.

\[
\begin{align*}
\delta(A, \Pi) &= \text{true} \text{ if } A \in \Pi \\
\delta(A, \Pi) &= \text{false} \text{ if } A \notin \Pi \\
\delta(q_f, \Pi) &= \text{false} \\
\delta(\varphi_1 \land \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \lor \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \lor \delta(\varphi_2, \Pi) \\
\delta(\langle \varphi \rangle, \Pi) &= \{\varphi \text{ if } \Pi \models \varphi \} \\
\delta(\langle \neg \varphi \rangle, \Pi) &= \{\neg \varphi \text{ if } \Pi \models \varphi \} \\
\delta(\langle \varphi_1 \varphi_2 \rangle, \Pi) &= \delta(\langle \varphi_1 \rangle, \Pi) \lor \delta(\langle \varphi_2 \rangle, \Pi) \\
\delta(\langle \varphi_1 \varphi_2 \rangle, \Pi) &= \delta(\langle \varphi_1 \rangle, \Pi) \land \delta(\langle \varphi_2 \rangle, \Pi) \\
\delta(\langle \varphi_1 \rangle \varphi_2, \Pi) &= \{\varphi_2 \text{ if } \rho \text{ is test-only} \} \\
\delta(\langle \neg \varphi \rangle \varphi_2, \Pi) &= \{\neg \varphi_2 \text{ if } \rho \text{ is test-only} \} \\
\delta(\langle \varphi \rangle, \Pi) &= \{\varphi \text{ if } \Pi \models \varphi \} \\
\delta(\langle \neg \varphi \rangle, \Pi) &= \{\neg \varphi \text{ if } \Pi \models \varphi \} \\
\delta(\langle \varphi_1 \varphi_2 \rangle, \Pi) &= \{\varphi_1 \text{ if } \rho \text{ is test-only} \} \\
\delta(\langle \neg \varphi \rangle \varphi_2, \Pi) &= \{\neg \varphi_2 \text{ if } \rho \text{ is test-only} \} \\
\end{align*}
\]

It is important to note that a LDLf formula can be rewritten to an equivalent LDLf formula which is in NNF in linear time. Further, that the state set of the AA for a LDLf formula \( \varphi \) is linear in the size of \( \varphi \).

**Theorem 4.1.** Let \( \varphi \) be an LDLf formula and \( A_\varphi \) the AA defined above. Then, for any interpretation \( \pi \), \( \pi \models \varphi \) iff \( A_\varphi \) accepts \( \pi \).
Proof. The correctness of the theorem stems from Theorem 17 in (De Giacomo and Vardi 2013). Our only modification to the AA presented in that paper is in the transition \( \delta([\phi]_\varphi, I) \): In the first of the two cases for this transition (i.e., when \( I \models \phi \)), we add the disjunction with \( q_f \). To see why this is necessary, consider the case where \( I \) contains a single state, and in that state \( \phi \) is true and \( \varphi \) is false. Then, since \( I \models \phi \), we are in the first of the two cases of this transition, so we transition to a state in which \( \varphi \lor q_f \), i.e., false \( \lor q_f \), is true. Had we not included \( q_f \) in this disjunction, the automaton would not accept this trace, which is an incorrect behavior. By allowing \( q_f \) as an option for this transition, we provide the automaton with the choice to end the trace and accept it, as it should. \(\Box\)

4.3 Building an AA for LTL-RE

LTL-RE augments LDL\(_f\) with LTL programming constructs, and a final modality, in order to make goal specification easier. These constructs can all be defined in terms of native LDL\(_f\). The most significant extension of LTL-RE over LDL\(_f\), from the perspective of translation, is the addition of the occ predicate that enables a goal to reference the occurrence of a particular ground action.

Let \((P, A, I)\) be a transition system, \(\alpha\) be a sequence of action applicable in \(I\), and \(\sigma\) be the state trace that is induced by the execution of \(\alpha\) in \(I\). Then we define a word \(\beta = b_0 b_1 \ldots b_n\) in which \(b_i = s_i \cup \{a_i\}\), for \(i \in \{0, \ldots, n - 1\}\), and \(b_n = s_n\).

Given an LTL-RE formula \(\varphi\), we first replace all program constructs (“if-then-else” and “while”), as well as any LTL constructs by the corresponding LDL\(_f\) equivalent given by the semantics defined in the previous section (i.e., we replace any occurrence of \(\Box \varphi\) by \(\text{true}^*\varphi\), and so forth). Then, as with LDL\(_f\) we put the resulting formula in negation normal form. Let \(\varphi'\) be the resulting formula. The AA for the resulting formula is like \(A_{\varphi'}\) described above for LDL\(_f\), but includes the following additional definition for \(\delta\):

\[
\delta(\text{occ}(a), I) = \text{occ}(a)
\]

where \(\text{occ}(a)\) is the special fluent from the set \(\text{Occ}\), which is described in a following paragraph. By making \(\text{occ}(a)\) true, we ensure that the ground action \(a\) must be the next action of the plan in order for this trace to be accepting. The automaton for \(\varphi\) that results from applying these steps is denoted as \(A_{\varphi}\).

Theorem 4.2. Let \(A_{\varphi}, \sigma\) be a state trace induced by the execution of action sequence \(\alpha\), and \(\beta\) be defined as above. Then \(\langle\sigma, \alpha\rangle \models \varphi\) iff \(A_{\varphi}\) accepts \(\beta\).

4.4 Compiling away LTL-RE goals

Now we describe a method to compile away LTL-RE goals by representing the AA within a new (output) transition system, constructed from the original planning transition system. Our method is based on the one proposed in (Torres and Baier 2015), which in their case compiled away LTL planning expressed as an AA. Although Torres and Baier only deal with LTL rather than LDL, the main technical difference between their method and ours is the treatment of action-centric constraints and in particular the special handling of the occ predicate in order to support REs over actions.

Given a transition system \(T = (P, A, I)\), and an LTL-RE goal \(\varphi\), our method generates a new transition system \(T' = (P', A', I')\). A plan for \(T\) and goal \(\varphi\) can be obtained by finding a sequence of action that reaches a final state where a distinguished property in \(t'\) holds. Satisfaction of this property corresponds to successful transitioning through the automaton \(A_{\varphi}\). Thus, the problem of planning for a temporal goal is reduced to the problem of finding a classical plan for which this distinguished property holds. Such a plan can be obtained using optimized off-the-shelf classical planners.

Following (Torres and Baier 2015), in \(T'\) there is one (new) fluent \(q\) for each state \(q\) of \(A_{\varphi}\). If \(\alpha = a_1 a_2 \ldots a_n\) is applicable in the initial state of \(T\), then there will exist a corresponding set of action sequences (denoted \(A_{\alpha}\)) of the form \(a_0 a_1 a_2 a_3 \ldots a_n\alpha\), where each \(\alpha_i\) is a sequence of so-called “synchronization actions” which did not appear in \(P\) and whose objective is to update the state of \(A_{\varphi}\).

Also in keeping with Torres and Baier, all actions in \(T\) also appear in \(T'\). Execution in \(T'\) can be understood as having two “modes”. In the so-called world mode, actions from the original transition system \(T\) can be executed. In the so-called synchronization mode actions that update the state of the automaton can be executed. The set of propositions \(P'\) contains additional propositions representing the state of the AA for \(\varphi\) plus additional flags that are used to switch appropriately between modes. Synchronization actions update the state of the automaton following the definition of the \(\delta\) function.

Fluents \(P'\) has the same fluents as \(P\) plus fluents that represent the states of the automaton \((Q)\), flags for controlling the different modes (copy, sync, world), and a special fluent ok, which becomes false if the goal has been falsified. Finally, it includes the set \(Q^S = \{q^S \mid q \in Q\}\) which are “copies” of the automata fluents (described in detail below), and \(\text{Occ}\) which contains a fluent \(\text{occ}(a')\) for each action such that \(\text{occ}(a)\) is a subformula of the original goal formula \(\varphi\). Formally, the set of fluents \(P' = F \cup Q \cup Q^S \cup \{\text{copy}, \text{sync}, \text{world}, \text{ok}\} \cup \text{Occ}\).

The set of planning operators \(O'\) is the union of the sets \(O_w\) and \(O_s\) for the world-mode and synchronization-mode actions, as follows.

World Mode Operators

The set \(O_w\) contains the same actions in \(A\), but preconditions are modified to allow execution only in “world mode”. Effects, on the other hand, are modified to allow the execution of the copy action, which initiates the synchronization phase, and which is described below. Formally, \(O_w = \{a' \mid a \in A\}\), and for all \(a'\) in \(O_w\):

\[
\text{prec}(a') = \text{prec}(a) \cup \{\text{ok, world}\} \cup \text{notOcc}(a'),
\]

\[
\text{eff}(a') = \text{eff}(a) \cup \{\text{copy, −world}\},
\]

where \(\text{notOcc}(a') = \{\neg \text{occ}(a) \mid a \neq a'\} \text{ and } \text{occ}(a) \in \text{Occ}\). I.e., the preconditions for executing \(a'\) are all the original preconditions for action \(a\), that the flags indicate the planner is in world mode and the goal has not been falsified,
Table 1: The synchronization actions generated for the translation of an LTL-RE goal \( \varphi \) in NNF. \( \ell \) is assumed to be a literal, and a (used in the last line) is assumed to be a (ground) world action. The transition \( tr(q^S_{α}) \) applies in the case where \( α \) is propositional (otherwise one of the earlier rules would be used). Similarly for \( tr(q^S_{(α ∧ β)}) \). We say that \( α \) is test-only if it is a finite regular expression whose atoms are only tests \( \psi \).

<table>
<thead>
<tr>
<th>Sync Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α}, \ell )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{β}) )</td>
<td>{sync, ok, ( q^S_{β} )}</td>
<td>{( q^S_{β}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α, β}) )</td>
<td>{sync, ok, ( q^S_{α, β} )}</td>
<td>{( q^S_{α, β}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α ∨ β}) )</td>
<td>{sync, ok, ( q^S_{α ∨ β} )}</td>
<td>{( q^S_{α ∨ β}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
<tr>
<td>( tr(q^S_{α}) )</td>
<td>{sync, ok, ( q^S_{α} )}</td>
<td>{( q^S_{α}, \neg \text{ok} )}</td>
</tr>
</tbody>
</table>

and it’s not the case that any of the other actions mentioned in the temporal goal \( \varphi \) (the set \( \text{Occ} \)) are occurring now.

Synchronization Mode Operators The set of synchronizing mode operators, \( O_s \), contains the actions \( \text{copy}, \text{world}, \) and all actions defined in Table 1. Collectively these actions realize the bookkeeping associated with the transition of the AA \( A_ϕ \) as a result of the actions executed in so-called world mode.

Synchronization mode is divided into three consecutive parts. In the first part, we execute the \( \text{copy} \) action which in the successor states adds a copy \( q^S \) for each fluent \( q \) that is currently true, deleting \( q \). Intuitively, during synchronization, each \( q^S \) defines the state of the automaton prior to synchronization. In addition, \( \text{copy} \) removes any propositions of the form \( \text{occ}(a) \). The precondition of \( \text{copy} \) is \{\{\text{copy, ok}\}, \text{while its effect is defined by}: 
\[
\text{eff}(\text{copy}) = \{q \rightarrow q^S, q \rightarrow \neg q \mid q \in Q\} \cup \{\text{sync}, \neg \text{copy}\} \cup \text{Occ}
\]

As soon as the \( \text{sync} \) fluent becomes true, the second phase of synchronization begins. Here the only executable actions are those that update the state of the automaton, which are defined in Table 1. Note that one of the actions deletes the \( \text{ok} \) fluent. This can happen, for example, while synchronizing a formula that actually expresses the fact that the action sequence has to conclude now.

When no more synchronization actions are possible—i.e., when there are no fluents of the form \( q^S \), we enter the third phase of synchronization. Here only action \( \text{world} \) is executable; its only objective is to reestablish world mode.

The precondition of \( \text{world} \) is \{\{\text{sync, ok}\}\} \cup Q^S, and its effect is \{\text{world, \neg sync}\}.

New Initial State The initial state of the original problem \( P \) intuitively needs to be “processed” by \( A_ϕ \) before starting to plan. Therefore, we define \( I' \) as \( I \cup \{q_f, \text{copy, ok}\} \).

New Goal Finally, the goal of the problem is to reach a state in which no state fluent in \( Q \) is true, except for \( q_f \), which may be true. Therefore we define \( G' = \{\text{world, ok}\} \cup \text{Occ} \).

5 Experimental Results
We have implemented the translator for LTL-RE. Three important questions to assess in an experimental evaluation are: 1) how large are the automata resulting from translation of the LTL-RE formulae, 2) how fast and space efficient is the translation, and 3) how effectively do the automata help guide search for a satisfying plan. Some of these are best evaluated on realistic benchmarks but such benchmarks exist only in limited ways and only for the LTL fragment of LTL-RE. As such, the comparative experimental analysis reported here is solely for the LTL fragment of LTL-RE.

For the purpose of experimentally evaluating our approach, we compare the performance of our LTL-RE translator both to an NFA-based LTL translator initially introduced in (Baier and McIlraith 2006), and to an AA-based LTL translator (Torres and Baier 2015), similar to ours. The NFA-based LTL translator is highly optimized to avoid, in most cases, the exponential blow up in the size of the automata characteristic of NFA-based representations of LTL. The AA translator exists in several versions including a naive version without engineering optimizations, and an optimized version. In order to fairly compare against our LTL-RE translator which currently lacks the implementation of analogous optimizations, we compared against the similarly unoptimized AA-based LTL translator. (The work reported here remains in progress, and optimization of our translator, analogous to those used in the NFA and AA-based LTL translators, constitutes ongoing work.) Even this comparison is not entirely appropriate. The LTL-RE translator is designed to translate all of LDLi including regular expressions, LTL, and additional programming language constructors. One might expect that a special-purpose translator that
Table 2: Results for domain *Blocksworld*, depicting translation time (TT), plan length (PL), total planning time (PT), number of planning states that were evaluated before the goal was reached (PS), and world plan length (WPL), which denotes the number of *planning* actions in PL; not including the actions that are responsible for the automaton synchronization.

<table>
<thead>
<tr>
<th>NFA translator</th>
<th>AA-LTL</th>
<th>LTL-RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT PL PT PS TT PL WPL PT PS TT PL WPL PT PS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p01 0.051 2 0.00 3 0.0108 15 2 0.00 73 0.112 15 2 0.00 48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p02 0.044 3 0.00 4 0.093 22 3 0.00 139 0.110 22 3 0.00 96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p03 0.051 7 0.00 16 0.113 50 7 0.00 719 0.113 53 7 0.00 547</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p04 0.058 10 0.00 27 0.112 75 10 0.01 3351 0.115 83 10 0.01 2959</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p05 0.049 14 0.00 43 0.115 104 13 0.03 15575 0.139 121 13 0.04 16672</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p06 0.303 14 0.00 43 0.117 99 13 0.04 16213 0.133 110 13 0.04 17153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p07 0.077 4 0.00 6 0.095 32 4 0.00 1555 0.115 32 4 0.00 1454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p08 0.586 7 0.00 11 0.116 55 6 0.00 23920 0.125 52 6 0.09 33252</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p09 72.556 9 0.02 20 0.113 67 7 0.117 74360 0.133 78 7 0.57 136892</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p10 72.614 9 0.02 20 0.119 68 7 0.22 89464 0.144 79 7 1.01 227686</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Results for domain *Logistics*, depicting translation time (TT), plan length (PL), total planning time (PT), number of planning states that were evaluated before the goal was reached (PS), and world plan length (WPL), which denotes the number of *planning* actions in PL; not including the actions that are responsible for the automaton synchronization.

<table>
<thead>
<tr>
<th>NFA translator</th>
<th>AA-LTL</th>
<th>LTL-RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT PL PT PS TT PL WPL PT PS TT PL WPL PT PS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p01 0.057 7 0.00 10 0.109 41 7 0.04 14217 0.107 59 7 0.56 126511</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p02 0.055 10 0.00 17 0.112 71 10 0.39 147017 0.115 81 10 18.35 1460496</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p03 0.061 21 0.00 64 0.114 127 21 8.82 1010542 0.107 0 0 13 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p04 0.060 27 0.02 121 0.112 0 0 NR NR 0.125 0 0 NR NR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p05 0.056 0 NR NR 0.116 65 10 0.14 60253 0.124 71 10 3.69 374535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p06 0.062 14 0.00 55 0.114 78 13 0.52 148994 0.136 78 13 35.10 1642692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p07 0.058 21 0.00 61 0.115 113 21 0.83 221874 0.118 42 0 23.31 1006596</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p08 0.045 20 0.00 70 0.085 111 20 0.98 226412 0.092 40 7 23.36 1006596</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p09 0.058 10 0.00 14 0.118 73 10 10.92 1097329 0.089 106 10 11.57 1045964</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Results for domain *ZenoTravel*, depicting translation time (TT), plan length (PL), total planning time (PT), number of planning states that were evaluated before the goal was reached (PS), and world plan length (WPL), which denotes the number of *planning* actions in PL; not including the actions that are responsible for the automaton synchronization.

<table>
<thead>
<tr>
<th>NFA translator</th>
<th>AA-LTL</th>
<th>LTL-RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT PL PT PS TT PL WPL PT PS TT PL WPL PT PS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p01 0.057 5 0.00 7 0.107 31 5 0.01 5237 0.104 29 5 0.02 4455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p02 0.051 11 0.00 16 0.106 53 9 0.16 57597 0.109 52 9 0.21 61045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p03 0.050 15 0.00 28 0.110 74 13 2.65 436249 0.110 74 13 2.65 436249</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p04 0.059 19 0.00 46 0.111 99 17 6.24 770784 0.128 96 17 20.71 1429295</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p05 0.059 7 0.00 9 0.116 52 7 0.22 76082 0.118 53 7 0.22 54321</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p06 0.051 12 0.00 20 0.123 76 11 2.89 485525 0.126 81 11 2.56 435910</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p07 0.056 0 NR NR 0.117 0 0 NR NR 0.127 0 0 NR NR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p08 0.053 5 0.00 9 0.097 39 5 0.01 4015 0.112 39 5 0.01 4702</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p09 0.058 8 0.00 14 0.114 68 8 0.05 20804 0.139 120 8 0.21 45778</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p10 0.065 0 NR NR 0.120 0 0 NR NR 0.138 0 0 NR NR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p11 0.058 9 0.00 19 0.119 62 9 0.16 52978 0.125 75 9 0.12 72760</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p12 0.060 11 0.00 22 0.116 89 11 5.84 747680 0.123 104 11 1.84 263967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p13 0.063 13 0.00 42 0.111 101 13 0.93 224707 0.132 149 13 2.77 334909</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p14 0.063 15 0.00 46 0.121 0 0 NR NR 0.139 0 0 NR NR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
is tuned only to LTL would be more efficient than the LTL portion of a more general LTL-RE translator – at least before implementation of optimizations.

The experiments were performed on three domains from the International Planning Competition (IPC): Blocksworld, Logistics and ZenoTravel. Goals arose from those introduced in IPC 2002. All experiments were run on a 64-bit machine, with a CPU of 1600 MHz. Each experiment was limited to 15 minutes runtime and 1GB of memory. The results are illustrated in Tables 2, 3, and 4.

Tables 2, 3, and 4 illustrate the performance of three reformulations: the NFA-based one in the first column, the AA-based LTL reformulation in the second and our AA-based LTL-RE reformulation in the third. The headers of these tables include translation time (TT), plan length (PL), total planning time (PT), number of planning states that were evaluated before the goal was reached (PS), and finally, world plan length (WPL), which denotes the number of planning actions in PL: not including the actions that are responsible for the automaton synchronization.

A merit of an AA-based translation approach relative to an NFA-based approach is the avoidance of the exponential blowup in size that theoretically exists with the latter. Indeed, this was an original impetus for selection of an AA-based approach. Nevertheless, the NFA-based translator is so optimized that it did not exhibit this theoretical blowup when originally developed and analyzed experimentally (Baier and McIlraith 2006). Consistent with this, we observe that the optimized NFA-based translator outperforms the two AA-based translators in many cases. However, there are problem instances that the NFA translator does less well on, marked by the drastic increase in total NFA translation time seen for example in p07 - p10 of Blocksworld. These instances correspond to goal formulas of the form \( \Diamond (\Diamond p_1 \land \Diamond p_2 \land \ldots \land \Diamond p_n) \), where \( n = 3, 5, 6, 7 \) for the instances p07 - p10 respectively.

The comparison of the two AA-based translators is particularly interesting. The two methods generate plans of the same world plan length for all instances. The computational cost that is associated with the use of the more general framework of LTL-RE, however, is reflected in the number of states expanded before the goal is reached, as well as in the total planning time, and in the total plan length (which includes both “world” actions, and actions to synchronize the automaton).

The difference in the total length of the translations is evident in every domain, reflecting that the syntactic rewriting of LTL formulas using LDL syntax is not always the most compact way of representing them. The difference in the number of expanded states and the planning time, on the other hand, is exhibited most dramatically in the Logistics domain, shown in Table 3. Instance p06, in particular, exemplifies the gap in the planning times of the two last translators, while almost all instances of this domain showcase the difference in the number of states that these two translators generate, in favor of the AA-based LTL translator.

There are, however, instances in which our method outperforms the AA LTL translator. Some examples of this can be found in the ZenoTravel domain, in Table 4. In the case of p12 for example, our translator takes significantly less time to generate the plan, and expands significantly less states while doing so.

We plan to further investigate the relationship between these two reformulations, and run more experiments to shed light on which formulas are better suited for each one. We also plan to implement an optimized version of the translator, in order to be able to fairly compare with more efficient reformulations of the AA-based LTL translation.

Our experiments did not examine the effectiveness of the translation of regular expressions and programming constructs. We note that Baier, Fritz, and McIlraith (2007; 2008) developed an automata-based translator for a Golog-like language that included regular expressions. Comparison with this translator would be interesting if suitable benchmarks could be found or constructed.

6 Discussion and Concluding Remarks

In this paper, we introduce LTL-RE: a high-level language for goal specification, which is rich enough to capture linear temporal formulas, as well as regular expressions. LTL-RE offers a convenient set of syntactic constructors, thus serving as a compelling vehicle for goal specification. A further contribution of our work is the implementation of a translation of LTL-RE goals into classical planning domains, making it feasible to plan for LTL-RE goals using state-of-the-art classical planners. We experimentally evaluate our approach by comparing the performance of this translator with an NFA-based translator and another AA-based translator, both specific to LTL formulas.

We are currently still running experiments, aiming to enhance our understanding of the differences in these reformulations. We also plan to experiment with Golog domains, to examine our translator’s performance on goals which are equivalent to regular expressions. Finally, we are interested in equipping our implementation with optimizations similar to those in (Torres and Baier 2015).

Acknowledgements

We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References


Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art
planners. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 26–33.


Commutativity Based Search

Doron A. Peled
Department of Computer Science
Bar Ilan University

Abstract
The problem of state space search is fundamental to both concurrent system verification and planning. Often, the state space is huge, so optimizing the search may be crucial. We consider the problem of visiting all states in a graph where edges are generated by actions and the (reachable) states are not known in advance. Some of the actions may commute, i.e., they result in the same state for every order in which they are taken. We show different methods in which we use commutativity to achieve full coverage of the states while traversing considerably fewer edges.
On Combining Symmetry with Partial Order Reduction

Dragan Bošnacki
Eindhoven University of Technology

We present some recent results (Bosnacki and Scheffers 2015) that combine two of the most successful state space reduction techniques in explicite state model checking: symmetry reduction and partial order reduction.

Partial order reduction (Godefroid 1996; Valmari 1996; Peled 1994) exploits the independence of the checked property from the execution order of the system actions. More specifically, two actions \( a, b \) are allowed to be permuted precisely when, if for all sequences \( v, w \) of actions: if \( vabw \) (where juxtaposition denotes concatenation) is an accepted behavior, then \( vbw \) is an accepted behavior as well. In a sense, instead of checking all the execution sequences, the desired property is checked only on representative sequences, which results in significant savings in space and time. Thus, the corner stone of the independence relation is the confluence condition as given in Fig. 1a. The confluence requires that from each state \( s \) of the state space the permutations of two independent actions \( a \) and \( b \) will lead to the same state \( s' \). The actual reduction of the state space is realized during the state space exploration by limiting the search from a given state \( s \) to only a subset of the actions that are executable in \( s \).

Symmetry reduction (Ip and Dill 1996; Emerson and Sistla 1996; Clarke et al. 1996) is one of the most successful techniques to tackle the state space explosion problem in model checking. The technique exploits the inherent symmetry of the model which is present in many systems, like mutual exclusion algorithms, cache coherence protocols, bus communication protocols, etc. After observing that the symmetry in the description of the model results in a symmetric state space, the key idea is to partition the state space into equivalence classes of (symmetric) states. Then, the state space exploration can be performed in the usually smaller quotient state space that consists only of (representatives of the) equivalence classes.

The problem of finding canonical, i.e., unique, representatives of equivalence classes is also known as the orbit problem. The orbit problem is equivalent to the graph isomorphism problem (Clarke et al. 1996), for which no polynomial algorithm is known. As a result, often with symmetry reduction the verification time can become critical. On the other hand, finding multiple (non-canonical) representatives usually boils down to sorting algorithms (Emerson and Sistla 1996; Bosnacki, Dams, and Holenderski 2000). An obvious drawback of the multiple representatives is that they provide less state space reduction compared to the canonical representatives. However, in practice it often turns out that, with an acceptable increase of the state space, the verification time can be improved significantly by using multiple representatives (Ip and Dill 1996; Bosnacki, Dams, and Holenderski 2000).

It turns out that symmetry and partial order reduction are orthogonal in the sense that they exploit different aspects of the system for reduction of the state space.

A combination of symmetry based on canonical representatives with partial order reduction was presented in (Emerson, Jha, and Peled 1997). This paper can be seen as a follow-up of (Emerson, Jha, and Peled 1997) which brings an extra contribution the symmetry case with multiple representatives. Along the lines of (Emerson, Jha, and Peled 1997) we derive our result in the more general setting of bisimulation preserving reductions. As symmetry reduction is then considered a special case of a bisimulation preserving reduction, all results are valid for symmetry too.

The main contribution of this paper is the use of a new kind of independence, which is weaker than the standard one. As mentioned above, in the usual definition of independence we insist on confluence, i.e., we require that the two paths obtained by permuting the independent actions \( a \) and \( b \) meet in the same state \( s' \). Instead, in the new definition we relax the confluence condition by allowing the permutations to lead to bisimilar states \( s_1' \) and \( s_2' \), as represented in Fig. 1b.

It turns out that almost all property preservation results, like absence of deadlock, safety, and liveness (\( LTL \) and \( CTL^* \) without the next operator), that can be found in the literature can be reused with a straightforward adaptation.

One can combine symmetry and partial order also in the model checking of timed systems which use discrete time (Bosnacki 2002).

References

Figure 1: Confluence of independent actions.


UPMurphi Released: PDDL+ Planning for Hybrid Systems

Giuseppe Della Penna
University of L’Aquila
Italy
giuseppe.dellapenna@univaq.it

Benedetto Intrigila
University of Roma "Tor Vergata"
Italy
intrigil@mat.uniroma2.it

Daniele Magazzeni
King’s College London
United Kingdom
daniele.magazzeni@kcl.ac.uk

Fabio Mercorio
University of Milan-Bicocca
Italy
fabio.mercorio@unimib.it

Abstract
In this tool paper, we present the release of UPMurphi, a universal planner for PDDL+ domains. Planning for hybrid domains has found increasing attention in the planning community, motivated by the need to address more realistic scenarios. While a number of techniques for planning with a subset of PDDL+ domains have been proposed, UPMurphi is able to handle the full range of PDDL+ features, including nonlinear continuous processes, exogenous events, Timed Initial Literals and numeric Timed Initial Fluents.

This paper describes the UPMurphi framework and presents its main features, together with a guide for using the tool, and some examples where UPMurphi has been successfully applied.

1 Introduction
A hybrid system is one in which there are both continuous control parameters and discrete logical modes of operation. It represents a powerful model to describe the dynamic behaviour of modern engineering artefacts. Hybrid systems frequently occur in practice, e.g., in robotics or embedded systems. Dealing with hybrid systems is becoming more and more an important challenge, as many real-world scenarios feature a mixture of discrete and continuous behaviours. Some example applications include coordination of activities of a planetary lander, oil refinery management, autonomous vehicles. Such scenarios motivate the need to reason with mixed discrete-continuous domains.

Planning for hybrid domains has found increasing attention in the planning community, motivated by the need to address more realistic scenarios and to interact with robotics and control frameworks. PDDL+ (Fox and Long 2006) is the extension of PDDL which allows the modelling of hybrid domains through the use of discrete actions, processes (that model continuous change over time), and exogenous events (that model changes that are initiated by the environment).

A number of techniques for PDDL+ planning have been proposed (Penberthy and Weld 1994; McDermott 2003; Li and Williams 2008; Coles et al. 2012; Shin and Davis 2005; Coles and Coles 2014; Molineaux, Klenk, and Aha 2010; Bryce and Gao 2015; Bogomolov et al. 2014; 2015). However, despite the recent efforts in proposing new algorithms and approaches for this domain, UPMurphi is currently the only available tool able to handle the full range of PDDL+ features.

The purpose of this paper is to accompany the release of the UPMurphi tool. To this aim, in the rest of the paper we overview the main features of the planner and we then describe the main techniques used for dealing with hybrid domains. In Section 3 we describe the general framework and provide a guide for using the tool. In Section 4 we survey some applications for which UPMurphi has been successfully used. Section 5 concludes the paper.

What UPMurphi Can Do. UPMurphi is a forward-search planner for PDDL+ domains. It can handle the whole PDDL+ language, including nonlinear continuous processes, exogenous events, Timed Initial Literals and numeric Timed Initial Fluents. It can be used either to find a single plan from the initial state to a goal state or to find a universal plan, i.e., a policy for handling the state space generated from the initial state.

2 How UPMurphi Works
UPMurphi is based on the planning-as-model-checking paradigm (Cimatti et al. 1997), and it is built on top of the CMurphi model checker (Cached Murphi Web Page 2006).

Planning in hybrid domains is challenging because in addition to the discrete state explosion problem, the continuous behaviour causes the reachability problem generally even to be undecidable. UPMurphi handles the hybrid dynamics through discretisation of time and continuous variables and by planning within a finite horizon. In this way, the state space is finite.

UPMurphi implements the Discretise and Validate approach (Della Penna et al. 2009) which is sketched in Figure 1. Here, the continuous dynamics of the system is relaxed into a discretised model, where discrete time steps and corresponding step functions for continuous values are used in place of the original continuous dynamics. Then, UPMurphi performs a forward reachability analysis in the discretised state space, searching for a path from the initial state to a state satisfying the goal condition. The discrete solution is then validated against the continuous model through the plan validator VAL (Howey, Long, and Fox 2004) to check whether the solution is valid or not. If it is invalid, the discretisation is refined and the process iterates. If UPMurphi
fails to find a plan at one discretisation the process can be iterated at a finer grained discretisation. The validation output can guide the user in identifying a suitable finer discretisation.

![Diagram of D&V approach](image)

Figure 1: Graphical representation of the D&V approach

## 3 How to Use UPMurphi

The overall UPMurphi architecture is sketched in Figure 3. UPMurphi can be invoked through the upm command by passing both the PDDL+ domain and problem as arguments. This initiates the following chain of operations.

**PDDL+ translation:** the PDDL-to-UPmurphi compiler, built on top of the VAL PDDL+ parser, takes as input a PDDL+ model and outputs a semantically equivalent UPMurphi model (up to the discretisation of time and continuous variables)\(^1\) `<domain_name>.m`. This, in turn, is compiled into an executable `<domain_name>_planner`.

**PDDL+ Planning:** The executable generated in the previous phase can now be invoked to start the planning/universal planning tasks. Several options can be specified to fine-tune this phase, as we describe below. The UPMurphi engine applies an explicit algorithm for building the system dynamics, and searches it through a forward search.

**Plan output:** Once the final plan(s) have been generated, they are written by default as PDDL+ plans, but the user can choose among a variety of different output formats (i.e., text, binary, and CSV), even in verbose mode.

**Plans Validation:** UPMurphi is designed to automatically interface with the VAL plan validator, so that the generated plans are executed and validated. To enable this functionality, the user must install VAL separately.

This process is completely automatic. When running the planner, the user can specify the discretisation to be used for time and continuous variables, although the default settings can be used.

### 3.1 UPMurphi Features

In this section we give details of the main UPMurphi features.

**Disk-based Search.** Starting with an initial discretisation (e.g., the one provided by default), the Discretise and Validate process should be iterated until a valid discretisation is used. However, the finer the discretisation, the larger the resulting state space, and this may lead to the state explosion phenomenon too early. To mitigate this issue, in this release UPMurphi employs the disk during the forward-search for storing both the state space generated so far and the current solution. Specifically, UPMurphi exploits the disk to store the full state description (i.e., the state values) whereas only the state signature is stored in memory using 40-bits for the encoding. This approach is beneficial for continuous domains as they often present a high number of discretised real values that grow the state size. Furthermore, it also allows trying several discretisation settings without affecting the number of states that can be visited during the search. UPMurphi is able to adapt its algorithm to increase or decrease the disk usage with respect to the user specified options and the size of the system under analysis. Furthermore, to avoid an excessive time overhead, the disk structures have been designed and implemented by taking into account their usage patterns, i.e., how (and how frequently) each structure is accessed during each phase of the planning process. This makes it possible to reduce the number of disk seek-and-read operations, which are the bottleneck of any disk algorithm, as seeks suffer from a latency time that is much higher than the actual read/write time. To give an example, UPMurphi privileges sequential read/writes, at the cost of duplicating some information and/or requiring more disk space, which is not a problem as large disks are nowadays very common.

**Serialisation.** Thanks to the disk-based exploration, UPMurphi can store the system graph, and access to it directly without having to load data into memory. This would allow one to resume the analysis by loading a (previously visited) system graph also on another machines.

**State Compression.** UPMurphi inherits from CMurphi a number of techniques to optimise the state representation (i.e., the bit compression and hash-compaction), and adapts them working even with the disk-based exploration described above.

**Exploration Strategy.** UPMurphi distinguishes between two planning modalities, namely (i) planning in which a feasible plan is generated to reach a goal and (ii) Universal Planning that could be seen as a collection of plans (aka a set of policies) able to bring the system to the goal from any reachable state for which a plan exists in the given setting. Note that both these modalities support the specification of the optimality requirement for minimising the plan makespan\(^2\).

---

\(^1\)The translation process is fully detailed in (Della Penna, Magazzeni, and Mercorio 2012).

\(^2\)Here optimality is dependent on the discretisation and the finite horizon used for planning.
(Some) Exploration Settings. UPMurphi provides some other options that can be set for customising the state space exploration. They include the specification of the (highest) amount of memory to be used by the planning process, the enabling of a deadlock check (here intended as a non-goal state without any action applicable), and the specification of either the maximum number of BFS levels to explore or a maximum plan length.

Stepwise Exploration. UPMurphi allows the use of a step-by-step exploration useful for debugging purposes. At each step the user can specify which action has to be applied (among the ones applicable in the current state). The values of each PDDL+ predicate and fluent are shown to the user and the process iterates.

Discretisation settings. By default, UPMurphi discretises the time to 0.1 units while real scale and real fraction digits are set to 8 and 2 respectively. Although we found these values suitable for a number of planning problems, the user is free to specify different values by passing them as arguments while invoking the UPMurphi planner.

Supporting the PDDL+ semantics. UPMurphi has been designed to support the PDDL+ semantics according to the start-process-stop model introduced by (Fox and Long 2006), that works by transforming a durative-action into a chain of PDDL+ elements, namely: (i) a pair start/end actions that apply the discrete effects at start and at end of the action respectively; (ii) a process that applies the continuous change over the action execution (iii) and an event that checks whether all the overall durative-action conditions are satisfied during its execution. This motivated the need to properly model the processes and events interaction within UPMurphi to fully support the whole PDDL+ semantics. Figure 2 shows an example of how UPMurphi represents and discretised time-line. The time is uniformly discretised in clock ticks (T) and a built-in action time-passing (TP) is responsible for advancing the time accordingly. Then, an action A1 can activate a process P1 that, after three clock ticks, triggers event E3, which in turn activates process P2. UPMurphi is able to handle process/events interleaving as well as Timed Initial Literals and numeric Timed Initial Fluents. Clearly, a fine enough discretisation must be used in order to capture the happenings of TIFs and TILs. On the other hand, the time granularity of TIFs and TILs can be used as a guidance for choosing the initial time discretisation.

Limitations. The main limitation of UPMurphi is that currently there is no heuristic to guide the search, and a blind BFS is performed. UPMurphi also requires the PDDL+ domain to be typed for being processed. Finally, the only metric actually supported by UPMurphi is :minimize total-time.

4 UPMurphi’s Track Record

UPMurphi has been applied to several challenging PDDL+ domains. In the following we present some of them.

The Planetary Lander (Fox and Long 2006). A rover has to perform two observation tasks, that require either to perform the corresponding preparation tasks or to execute a single cumulative preparation task for both observations. The goal is to find a solution minimising the plan makespan. As a challenge, the domain presents a nonlinear system dynamics, deriving from the system equations (e.g., the energy generated by solar panels is influenced by the position of the sun), concurrence between processes (i.e., the rover may generate energy while is charging the battery), and processes/events interactions that may invalidate the plan due to tight resources and time constraints. In Figure 4 we show a log of a UPMurphi execution for the planetary lander. UPMurphi here searches for a feasible plan using up to 1Gb RAM and outputs the resulting plan in PDDL+ format. Finally, the plan is shown and saved into a file as well. More details on the use of UPMurphi in this domain can be found in (Della Penna, Magazzeni, and Mercorio 2012).

The Batch Chemical Plant. A production system is designed to obtain a concentrated saline solution recycling the remaining part for the next cycle. The system has many tightly connected components, regulated by a nonlinear dynamics (due to the equations modelling temperature and concentration variations), unknown action durations and a large set of safety constraints. UPMurphi has been used to synthesise a set of policies for many different initial production configurations. Note that thanks to the efficient use of the disk during the state space exploration, UPMurphi was able to generate up to 7 million plans. More details on the use of UPMurphi for the batch chemical plant can be found in (Della Penna et al. 2010).
Table 1: Some statistics for Planetary Lander and Chemical Plant domains

<table>
<thead>
<tr>
<th></th>
<th>Planetary Lander</th>
<th>Chemical Plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Space Size</td>
<td>$10^{14}$</td>
<td>$10^{14}$</td>
</tr>
<tr>
<td>Reachable States</td>
<td>31,965,220</td>
<td>29,968,861</td>
</tr>
<tr>
<td>Generated Plans</td>
<td>5,901,014</td>
<td>7,184,404</td>
</tr>
</tbody>
</table>

Table 2: Universal Plan statistics for the generator domain with time discretisation from 5.0 down to 1.0 seconds

<table>
<thead>
<tr>
<th>Time discretisation (sec)</th>
<th>5.0</th>
<th>2.5</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>State space size</td>
<td>$10^{14}$</td>
<td>$10^{14}$</td>
<td>$10^{14}$</td>
</tr>
<tr>
<td>Reachable states</td>
<td>26,276</td>
<td>399,189</td>
<td>29,119,047</td>
</tr>
<tr>
<td>Generated plans</td>
<td>0</td>
<td>10,015</td>
<td>126,563</td>
</tr>
<tr>
<td>Total synthesis time</td>
<td>3.7</td>
<td>20.71</td>
<td>1,430.11</td>
</tr>
<tr>
<td>Valid</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

Figure 4: Log of a UPMurphi execution for the Planetary Lander domain

Nonlinear generator. It is the continuous model of the well-known generator domain (Howey and Long 2003). A generator is powered by a fuel tank with a limited capacity of 60 fuel units and consumes one fuel unit per second. During the generator activity (modelled by the consume durative action), two fuel tanks of 25 fuel units each can be used to refuel it (through the refuel durative action). The refuelling process has a variable duration (i.e., its duration must be decided by the planner) and is described by the Torricelli’s law, which makes the system dynamics nonlinear. Moreover, the domain also involves concurrency, since the consume and refuel actions take place continuously and concurrently, and are modelled through continuous processes. The goal is to make the generator run for 100 seconds. Table 2 summarises the results of the universal planning process after three Discretise and Validate iterations. We first considered a time discretisation of 5.0 and 2.5, both resulting in invalid solutions. We then refined the discretisation to 1.0 which proved to be fine enough for obtaining valid plans.

5 Concluding Remarks

In this paper we presented the release of the PDDL+ planner UPMurphi, overviewing its main features that allow it to handle the full range of PDDL+ features, including non-linear continuous processes, exogenous events, Timed Initial Literals and numeric Timed Initial Fluents. On a practical note, UPMurphi has been designed to work natively on Linux distributions (Ubuntu specifically), but it has been extensively tested on Windows with Cygwin environment, too. Finally, a MacOS compilation is also supported. Please refer to the UPMurphi web page (UPMurphi Web Page 2015) for download, installation instructions, more details, and news about UPMurphi development.
References


Hybrid Systems: Guided Search, Abstractions, and Beyond

Sergiy Bogomolov
IST Austria, Austria
sergiy.bogomolov@ist.ac.at

Abstract

Hybrid systems represent an important and powerful formalism for modeling real-world applications such as embedded systems. A verification tool like SpaceEx is based on the exploration of a symbolic search space (the region space). As a verification tool, it is typically optimized towards proving the absence of errors. In some settings, e.g., when the verification tool is employed in a feedback-directed design cycle, one would like to have the option to call a version that is optimized towards finding an error path in the region space. A recent approach in this direction is based on guided search. Guided search relies on a cost function that indicates which states are promising to be explored, and preferably explores more promising states first. In this talk, we present two approaches to define and compute efficient cost functions. We develop our approaches on the top of the symbolic hybrid model checker SpaceEx which uses regions as its basic data structures.

In the first part of the talk, we introduce a box-based distance measure which is based on the distance between regions in the concrete state space. In the second part of the talk, we discuss an abstraction-based cost function based on pattern databases for guiding the reachability analysis. For this purpose, a suitable abstraction technique that exploits the flexible granularity of modern reachability analysis algorithms is introduced. We illustrate the practical potential of our approaches in several case studies.