Synthesizing controllers: On the Correspondence Between LTL Synthesis and Non-Deterministic Planning

Alberto Camacho[1], Jorge A. Baier[2], Christian Muise[3], and Sheila A. McIlraith[1]

[1] Department of Computer Science. University of Toronto, Canada.
[2] Chilean Center for Semantic Web Research. Pontificia Universidad Católica de Chile, Chile.
[3] IBM Research. Cambridge Research Center. USA.

Abstract. Linear Temporal Logic (LTL) synthesis can be understood as the problem of building a controller that defines a winning strategy, for a two-player game against the environment, where the objective is to satisfy a given LTL formula. It is an important problem with applications in software synthesis, including controller synthesis. In this paper we establish the correspondence between LTL synthesis and fully observable non-deterministic (FOND) planning. We study LTL interpreted over both finite and infinite traces. We also provide the first explicit compilation that translates an LTL synthesis problem to a FOND problem. Experiments with state-of-the-art LTL FOND and synthesis solvers show automated planning to be a viable and effective tool for highly structured LTL synthesis problems.

Keywords: automated planning, controller synthesis, LTL, non-deterministic planning

1 Introduction

The problem of synthesizing software, including controllers, from logical specification is a fundamental problem in AI and computer science more generally. Church's synthesis problem was first posed by Church in 1957 in the context of synthesizing digital circuits from a logical specification [1] and is considered one of the most challenging problems in reactive systems [2]. Two common approaches to solving the problem have emerged: reducing the problem to the emptiness problem of tree automata, and characterizing the problem as a two-player game.

In 1989, Pnueli and Rosner examined the problem of reactive synthesis using Linear Temporal Logic (LTL) [3] as the specification language (henceforth "LTL synthesis") viewing the problem as a two-player game, and showing that this problem was 2EXPTIME-complete [4]. This discouraging result has been mitigated by the identification of several restricted classes of LTL for which synthesis is less complex. For example, if the LTL specification is restricted to the class of so-called *Generalized Reactivity(1)* (GR1) formulae, an N^3-time algorithm exists [2]. Today, a number of synthesis tools exist with varying effectiveness (e.g., Acacia+ [5], Lily [6]).

Recent work has explored various connections between automated planning and synthesis (e.g., [7–12]) but has not provided a full mapping between the two problems, nor have the practical implications of such a mapping been explored from an

automated planning perspective. In this paper we investigate the relationship between (LTL) synthesis and automated planning, and in particular (LTL) Fully Observable Non-Deterministic (FOND) planning. We do so by leveraging a correspondence between FOND and 2-player games. This work is inspired by significant recent advances in the computational efficiency of FOND planning that have produced FOND planners that scale well in many domains (e.g., myND [13] and PRP [14]). Our insights are that just as SAT can be (and has been) used as a black-box solver for a myriad of problems that can be reduced to SAT, so too can FOND be used as a black-box solver for suitable problems. Establishing the connection between FOND and 2-player games not only provides a connection to LTL synthesis – the primary subject of this exploration – it also provides the key to leveraging FOND for other problems.

In Section 3 we establish the correspondence between LTL synthesis and strong solutions to FOND planning. In Section 4 we provide the first automatic translation of a realizability problem into a planning problem, described in the Planning Domain Definition Language (PDDL), the de facto standard input language for automated planners. Experiments with state-of-the-art LTL synthesis and FOND solvers illustrate that the choice of formalism and solver can have a dramatic impact. Indeed, planning-based approaches excel if the problem is highly structured and the uncertainty largely restricted, as is the case for synthesis problems associated with engineered physical devices.

2 Preliminaries

FOND: A FOND planning problem is a tuple $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, where $\mathcal{F}$ is a set of *fluents*; $\mathcal{I} \subseteq \mathcal{F}$ characterizes what holds initially; $\mathcal{G} \subseteq \mathcal{F}$ characterizes the goal condition; and $\mathcal{A}$ is the set of actions. The set of literals of $\mathcal{F}$ is $Lits(\mathcal{F}) = \mathcal{F} \cup \{\neg f \mid f \in \mathcal{F}\}$. Each action $a \in \mathcal{A}$ is associated with $\langle Pre_a, \mathit{Eff}_a \rangle$, where $Pre_a \subseteq Lits(\mathcal{F})$ is the precondition and Eff_a is a set of outcomes of a. We sometimes write $oneof(\mathit{Eff}_a)$ to emphasize that Eff_a is non-deterministic. Each outcome $e \in \mathit{Eff}_a$ is a set of conditional effects of the form $(C \rightarrow \ell)$, where $C \subseteq Lits(\mathcal{F})$ and $\ell \in Lits(\mathcal{F})$. Given a planning state $s \subseteq \mathcal{F}$ and a fluent $f \in \mathcal{F}$, we say that s satisfies f, denoted $s \models f$, iff $f \in s$. In addition $s \models \neg f$ if $f \notin s$, and $s \models L$ for a set of literals L, if $s \models \ell$ for every $\ell \in L$. Action a is *applicable* in state s if $s \models Pre_a$. We say s' *is a result of applying* a *in* s iff, for one outcome e in Eff_a, s' is equal to $s \setminus \{f \mid (C \rightarrow \neg f) \in e, s \models C\} \cup \{f \mid (C \rightarrow f) \in e, s \models C\}$. A *policy* p, is a partial function from states to actions such that if $p(s) = a$, then a is applicable in s. An *execution* π of a policy p in state s is a sequence $s_0, a_0, s_1, a_1, \ldots$ (finite or infinite), where $s_0 = s$, and such that every state-action-state substring s, a, s' are such that $p(s) = a$ and s' is a result of applying a in s. Finite executions ending in a state s are such that $p(s)$ is undefined.

A finite execution π *achieves* a set of literals L if its ending state s is such that $s \models L$. An infinite execution π *achieves* a set of literals L if there exists a state s that appears infinitely often in π and that is such that $s \models L$. An infinite execution σ is *fair* iff whenever s, a occurs infinitely often within σ, then so does s, a, s', for every s' that is a result of applying a in s [15]. Note this implies that finite executions are fair. A policy p is a *strong plan* (resp. *strong-cyclic plan*) for a FOND problem $P = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, iff every execution (resp. fair execution) of p over $\mathcal{I}$ satisfies the goal $\mathcal{G}$.

Linear Temporal Logic: Linear Temporal Logic (LTL) is a propositional logic extended with temporal modal operators *next* ($\bigcirc$) and *until* (U). The set of LTL formulae over a set of propositions $\mathcal{P}$ is defined inductively as follows. p is a formula if $p \in \mathcal{P}$ or the constant $\top$. If φ_1 and φ_2 are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\bigcirc\varphi_1$ and $\varphi_1 \,\mathsf{U}\, \varphi_2$. Let $\sigma = s_0, s_1, \ldots$ be an infinite sequence of subsets of $\mathcal{P}$, and φ be an LTL formula. Then σ *satisfies* φ, denoted as $\sigma \models \varphi$ iff $\sigma, 0 \models \varphi$, where:

$\sigma, i \models p$, for each $p \in \mathcal{P} \cup \{\top\}$ iff $s_i \models p$ $\quad$ $\sigma, i \models \neg\varphi$ iff $\sigma, i \models \varphi$ does not hold

$\sigma, i \models \varphi_1 \wedge \varphi_2$ iff $\sigma, i \models \varphi_1$ and $\sigma, i \models \varphi_2$ $\quad$ $\sigma, i \models \bigcirc\varphi$ iff $\sigma, (i+1) \models \varphi$

$\sigma, i \models \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there exists a $j \geq i$ such that
$\sigma, j \models \varphi_2$, and $\sigma, k \models \varphi_1$, for each $k \in \{i, i+1, \ldots, j-1\}$

Intuitively, the *next* operator tells what needs to hold in the next time step, and the *until* operator tells what needs to hold until something else holds. The modal operators *eventually* ($\Diamond$) and *always* ($\Box$) are defined by $\Diamond\varphi \equiv \top \,\mathsf{U}\, \varphi$, $\Box\varphi \equiv \neg\Diamond\neg\varphi$. Additional constants and operators are defined by following conventional rules as follows $\bot \equiv \neg\top$, $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

LTL over finite traces: $\mathsf{LTL_f}$ is a variant of LTL interpreted over finite traces [16]. Such finite variants have been applied to problems in verification, model checking and planning. $\mathsf{LTL_f}$ and LTL share the same syntax, but their interpretations differ. For example, the $\mathsf{LTL_f}$ formula $\Diamond\neg\bigcirc\top$ is true in a finite trace, whereas the same formula in LTL evaluates false on infinite traces. Similarly, *weak next* must often replace *next* to avoid unintended interpretations of LTL over finite traces. (See [16] for details.)

Automata: There is a well-established correspondence between LTL and automata. A *Non-deterministic Büchi Automaton* (NBA) is a tuple $M = (Q, \Sigma, \delta, q_0, Q_{Fin})$, where Q is the set of automaton states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 is the initial state, and $Q_{Fin} \subseteq Q$ is the set of accepting states. The automaton is *deterministic* (DBA) when for each $q \in Q$, and $s \in \Sigma$, there exists a unique $q' \in Q$ such that $(q, s, q') \in \delta$. A *run* of M on an infinite word $\sigma = s_0, s_1, \ldots$ of elements in Σ is a sequence of automaton states, starting in q_0, such that $(q_i, s_i, q_{i+1}) \in \delta$ for all $i \geq 0$. A run is accepting if it visits an infinite number of accepting states. Finally, we say that M accepts σ if there is an accepting run of M on σ. *Non-deterministic Finite-state Automata* (NFAs) differ from NBAs in that the acceptance condition is defined on *finite* words: a word $\sigma = s_0, s_1, \ldots, s_m$ is accepting if $q_{m+1} \in Q_{Fin}$. Finally, *Deterministic Finite-state Automata* are NFAs where the transition relation is deterministic.

Given an LTL formula φ, it is possible to construct an NBA A_φ that accepts σ iff $\sigma \models \varphi$. The construction is worst-case exponential in the size of φ [17]. It is not always possible to construct a DBA, and the construction is double exponential. Similar results hold for $\mathsf{LTL_f}$: it is always possible to construct an NFA (resp. DFA) that accepts σ iff $\sigma \models \varphi$, and the construction is worst-case exponential (resp. double exponential) [18].

LTL FOND: Recently [11] extended FOND with LTL goals. An LTL FOND problem is a tuple $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, where $\mathcal{G}$ is an LTL or $\mathsf{LTL_f}$ formula over $\mathcal{F}$, and $\mathcal{F}, \mathcal{I}, \mathcal{A}$ are defined as in FOND planning. In short, LTL FOND executions are defined just like in FOND, and a policy is a strong-cyclic (resp. strong) plan for problem P if each fair (resp. unrestricted) execution π results in a sequence of states σ such that $\sigma \models \mathcal{G}$.

LTL Synthesis: Intuitively, the LTL synthesis problem [4] describes a two-player game between a controller and the environment. The game consists of an infinite se-

quence of turns. In each turn the environment chooses an action, and the controller then chooses another. Each action corresponds to setting the values of some variables. The controller has a winning strategy if, no matter how the environment plays, the sequences of states generated satisfy a given LTL formula φ. Formally, a synthesis problem is a tuple $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$, the environment variables, and $\mathcal{Y} = \{y_1, \ldots, y_m\}$, the controller variables, are disjoint sets. An LTL formula over $\mathcal{X} \cup \mathcal{Y}$ is *realizable* if there exists a function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ such that for every infinite sequence of subsets of $\mathcal{X}$, $X_1 X_2 \ldots$, it holds that $\pi = (X_1 \cup f(X_1)), (X_2 \cup f(X_1 X_2)) \ldots$ is such that $\pi \models \varphi$. Intuitively, no matter what the choice of the environment is, which is given by the sequence $X_1 X_2 \ldots$, the controller has a strategy, given by f, that ensures formula φ is satisfied in the resulting game. The *synthesis* problem corresponds to computing function f. Synthesis has also been studied over *finite* sequences of turns using $\mathsf{LTL_f}$ specifications (e.g. [10]). In the rest of the paper, we write LTL synthesis to also refer to $\mathsf{LTL_f}$ synthesis, and make the distinction explicit only when necessary.

3 Relationship Between FOND and Synthesis

Both LTL synthesis and FOND are related to two-player games: in both problems an agent (or controller) seeks a solution that achieves a condition regardless of the choices made by the environment. There are, however, two important differences. First, in LTL synthesis the controller reacts to the environment; in other words, the environment "plays first", while the controller "plays second".[4] In FOND, the play sequence is inverted since the environment decides the outcome of an action, which is in turn defined by the agent (controller). Second, state-of-the-art FOND solvers find strong-cyclic solutions predicated on an assumption of fairness in the environment, which is not an assumption inherent to LTL synthesis. Thus a correct mapping between FOND and Synthesis must handle fairness correctly.

Previous work has explored the relation between FOND and synthesis. [9] show how to translate FOND as a reactive synthesis problem by expressing fairness constraints as temporal logic formulae. [16] sketches a mapping from FOND to LTL synthesis, in which the effects of actions are specified using LTL. This approach, however, does not dive into the details of the inverted turns. Neither do the works by [7, 19], which show a correspondence between two-player game structures and FOND planning. In the rest of the section we provide an explicit mapping between LTL FOND and LTL synthesis. Efficiency is the focus of the next section.

To establish a correspondence between LTL synthesis and LTL FOND, we address the inverted turns by considering the negation of realizability. Observe that an instance $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is not realizable iff there exists a sequence $X_1 X_2 X_3 \ldots$ of subsets of $\mathcal{X}$ such that for every function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$:

$$X_1 \cup f(X_1), X_2 \cup f(X_1 X_2), X_3 \cup f(X_1 X_2 X_3) \ldots \models \neg\varphi$$

Note that what comes after the "iff" maps directly into an instance of LTL FOND: we define the problem $P_\varphi = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ such that fluents are the union of all variables (i.e., $\mathcal{F} = \mathcal{X} \cup \mathcal{Y}$), and the set of actions is the set of subsets of $\mathcal{X}$ (i.e., $\mathcal{A} = \{a_x \mid$

[4] The problem with inverted turns, where the agent "plays first", has also been studied (e.g. [5]).

$x \subseteq \mathcal{X}\}$). Intuitively action a_x is always executable (has empty preconditions) and deterministically sets to true the variables in x and to false the variables in $\mathcal{X} \setminus x$. In addition, it non-deterministically sets the values of variables in $\mathcal{Y}$ to every possible combination. Formally, $\mathit{Eff}_{a_x} = \{e_{x,y} \mid y \subseteq \mathcal{Y}\}$, where each $e_{x,y} = \{f \mid f \in x \cup y\} \cup \{\neg f \mid f \in (\mathcal{X} \cup \mathcal{Y}) \setminus (x \cup y)\}$. Finally, we set $\mathcal{I} = \{\}$ and $\mathcal{G} = \bigcirc\neg\varphi$.

A more involved argument follows for $\mathsf{LTL_f}$ synthesis. In this case, an instance $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ is not realizable iff for every finite m there exists a sequence $X_1 X_2 \dots X_m$ of subsets of $\mathcal{X}$ such that, for every function $f : (2^{\mathcal{X}})^* \to 2^{\mathcal{Y}}$:

$$X_1 \cup f(X_1), \dots, (X_1 \dots X_m) \cup f(X_1 \dots X_m) \models \neg\varphi$$

What follows after the "iff" cannot be directly mapped into an instance of LTL FOND, because the formula above has to hold for all m. We can mitigate for this by adding a new variable to $\mathcal{P}_\varphi$, y_{ok}, that acts like any other variable in $\mathcal{Y}$. The goal of $\mathcal{P}_\varphi$ is the $\mathsf{LTL_f}$ formula $\mathcal{G} = \bigcirc(\neg\varphi \wedge \Diamond(y_{ok} \wedge \neg\bigcirc\top)) \vee \Diamond(\neg y_{ok} \wedge \neg\bigcirc\top)$.

Theorem 1. LTL *synthesis problem* $\langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ *is realizable iff* P_φ *has no strong plan.*

In the other direction, let $P = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ be an LTL FOND problem. We now construct a synthesis problem $\langle \mathcal{X}_P, \mathcal{Y}_P, \varphi_P \rangle$ following well-known encodings of planning as SAT [20]; we use LTL to establish a connection between a state and its successors, instead of different variables, and we consider explicitly that actions have a number of outcomes. The specification φ_P is: $\varphi_P := \varphi_{init} \to (\varphi_{env} \to (\varphi_{agt} \wedge \varphi_g))$. Intuitively, φ_{init} models the initial state $\mathcal{I}$, φ_{env} and φ_{agt} model the dynamics in $\mathcal{P}$, and φ_g is the LTL goal formula $\mathcal{G}$.

For each action in $a \in \mathcal{A}$, we create a variable $a \in \mathcal{X}_P$. Each fluent $f \in \mathcal{F}$ is also a variable in $\mathcal{X}_P$. Variables in $\mathcal{Y}_P$ are used to choose one of the non-deterministic outcomes of each action; this way if the action with the largest number of outcomes has n outcomes, we create $\lceil \log n \rceil$ variables, whose objective is to "choose" the outcome for an action. To model the preconditions of the action, we conjoin in φ_{env}, for each action a the formula $\Box(a \to \bigwedge_{\ell \in Pre_a} \ell)$. We express the fact that only one action can execute at a time by conjoining to φ_{env} the formulae $\Box\bigvee_{a \in \mathcal{A}} a$, and $\Box(a \to \neg a')$, for each $a' \in \mathcal{A}$ different from a. To model the fact that the environment selects the outcome being performed, for each action outcome e we create a variable a_e in $\mathcal{X}_P$. For each action $a \in \mathcal{A}$ and outcome $e \in \mathit{Eff}_a$, φ_{agt} has formulae of the form $\Box(a \wedge \chi_{a,e} \to a_e)$, where $\chi_{a,e}$ is a formula over $\mathcal{Y}_P$, which intuitively "selects" outcome e for action a. For space, we do not go into the details of how to encode $\chi_{a,e}$. However, these formulae have the following property: for any action a, given an assignment for $\mathcal{Y}_P$ variables there is exactly one $e \in \mathit{Eff}_a$ for which $\chi_{a,e}$ becomes true. This models the fact that the $\mathcal{Y}_P$ variables are used to select the outcomes.

Finally, we now conjoin to φ_{env} formulae to express the dynamics of the domain. Specifically we add successor-state-axiom-like expressions [21] of the form:

$$\Box(\bigcirc f \equiv (\phi_f^+ \vee (f \wedge \neg\phi_f^-)), \quad \text{for each } f \in \mathcal{F}$$

where ϕ_f^+ is a formula that encodes the conditions under which f becomes true after an outcome has occurred, and where ϕ_f^- encodes the conditions under which f becomes false in the next state. Both of these formulae can be computed from Eff_a[21], and have fluents a_e for $e \in \mathit{Eff}_a$. Finally, φ_{init} is the conjunction of the fluents in the initial

state $\mathcal{I}$, and φ_g is the goal formula, $\mathcal{G}$. When the goal of $\mathcal{P}$ is an $\mathsf{LTL_f}$ formula, the construction conjoins $\circ\top$ to the successor state axioms in φ_{env}.

Now, it is not hard to see that there exists a strong solution to the LTL problem P iff there exists a (finite for $\mathsf{LTL_f}$ goals, infinite for LTL) sequence of settings of the $\mathcal{X}_P$ variables, such that for every sequence of settings of the $\mathcal{Y}$ variables (i.e., for every function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$), it holds that $(X_1 \cup Y_1), (X_2 \cup Y_2), (X_3 \cup Y_3), \ldots \models \varphi_P$.

Theorem 2. *LTL FOND problem $P = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ has a strong plan if and only if $\langle \mathcal{X}_P, \mathcal{Y}_P, \neg\varphi_P \rangle$ is not realizable.*

4 Approach

In Section 3 we established the correspondence between existence of solutions to LTL synthesis, and existence of strong solutions to LTL FOND planning. In this section we introduce the first translation from LTL synthesis into FOND planning (and by inclusion, into LTL FOND), and a translation for $\mathsf{LTL_f}$ specifications.

Our approach to solve an LTL synthesis problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ as FOND consists of three stages. First, we pre-process $\mathcal{P}$. Second, we compile it into a standard FOND problem $\mathcal{P}'$. Finally, we solve $\mathcal{P}'$ with a strong-cyclic planner. Extracting a strategy for $\mathcal{P}$ from a solution to $\mathcal{P}'$ is straightforward, and we omit the details for lack of space.

Automaton Transformation: In a pre-processing stage, we simplify the specification by removing from $\mathcal{X}$ and $\mathcal{Y}$ those variables that do not appear in φ. Then, we transform φ into an automaton, $A_\varphi = (Q, \Sigma, \delta, q_0, Q_{Fin})$, that can be a DBA when the LTL formula is interpreted over infinite traces, or an NFA (or DFA, by inclusion) when the specification is an $\mathsf{LTL_f}$ formula. In addition to DBAs, our algorithm can seamlessly handle NBAs at the cost of losing its completeness guarantee. NBAs are a good alternative to DBAs as they are usually more compact, and only a subset of LTL formulae can be transformed into DBAs. The transition relation δ in A_φ implicitly defines the conditions under which the automaton in state q is allowed to transition to state q'. These conditions are known as *guards*. Formally, $guard(q, q') = \bigvee_{(q,s,q') \in \delta} s$. In our case, elements of the alphabet Σ are conjunctions of boolean variables, that allow for guard formulae to be described in a compact symbolic form. In what follows, we assume guard formulae $guard(q, q') = \bigvee_m c_m$ are given in DNF, where each clause c_m is a conjunction of boolean state variables. We denote as $\delta^\star$ the set of tuples $T_m = (q, c_m, q')$ for each pair (q, q') with $guard(q, q') \neq \bot$, and for each clause c_m in $guard(q, q')$. For convenience, we write $guard(T_m) = c_m$, and refer to elements of $\delta^\star$ as *transitions*. Wherever convenient, we drop the subindex of transitions and simply write T.

In the second stage, we compile $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ with automaton A_φ into a parameterized FOND problem $\mathcal{P}'(\mathcal{X}, \mathcal{Y}, A_\varphi, H) = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ that integrates the dynamics of A_φ with a two-player game between the environment and the agent. Before introducing the technical details of the compilation, we give an overview. The compilation simulates automaton states by means of fluents q, one for each automaton state with the same name. Planning states s have the following property: *an automaton fluent q is true in s iff for some σ, a run σ of A_φ finishes in q*. Notably, the input word σ can be obtained directly from the state-action plan that leads the initial state to s in the search

tree. When the input to the algorithm is a non-deterministic automaton (NBA or NFA), planning states can simultaneously capture multiple runs of the automaton in parallel by simultaneously having multiple q fluents set to true.

The acceptance condition behaves differently for Büchi and non-Büchi automata, and this is reflected in our compilation. For Büchi automata, the planning process expands a graph that simulates the environment and agent moves, and the search for solutions becomes a search for strong cyclic components that hit an accepting state infinitely often. The latter property is checked by means of *tokenized* fluents q^t, one for each q. Intuitively, the truth of $q \wedge q^t$ in state s indicates a commitment to progress runs finishing in q into runs that reach an accepting state. Conversely, $s \models q \wedge \neg q^t$ represents that such a commitment has been accomplished. The role of the parameter H is twofold: it bounds the horizon in the search for cycles, and it allows the use of strong-cyclic solvers to find solutions to a problem whose non-determinism has unrestricted fairness.

The compilation runs in two sequentially alternating modes simulating each two-player turn. The *environment mode* simulates the environment moves, which are non-deterministic and uncontrollable to the agent. In *automaton mode*, the agent moves are simulated and the automaton state fluents are synchronized according to valid transitions in $\delta^\star$. Auxiliary fluents q^s and $q^{s,t}$ are used to store the value of automaton state fluents q and q^t prior to synchronization, so that more than one transition can be simulated in the case of non-deterministic automata compilations. When an accepting state q is recognized, the agent can set the token fluents q^t to commit to progress the runs that finish in q into a run that hits another accepting state.

The dynamics of the compilation are similar for non-Büchi automata. The exception is that accepting runs are recognized whenever an accepting automaton state fluent is reached, and there is no need to commit to reaching another accepting state. Consequently, tokenized fluents q^t and $q^{s,t}$ are not needed but have been kept, for generality, in the algorithm below.

The sets of fluents ($\mathcal{F}$) and actions ($\mathcal{A}$) of the problem are listed below. In what follows, we describe the technical details of the compilation.

$$\begin{aligned}
\mathcal{F} &= \left\{q, q^s, q^t, q^{s,t} \mid q \in Q\right\} \cup \{goal\} \cup \{next(h{+}1, h)\}_{0 \le h < H} \cup \{turn_k\}_{1 \le x \le |\mathcal{X}|} \\
&\quad \cup \{at_horizon(h)\}_{0 \le h \le H} \cup \{env_mode, aut_mode, can_switch, can_accept\} \\
&\quad \cup \{v_x, v_{\neg x}\}_{x \in \mathcal{X}} \cup \{v_y, v_{\neg y}\}_{y \in \mathcal{Y}} \\
\mathcal{A} &= \{move_k\}_{1 \le k \le |\mathcal{X}|} \cup \{trans_T\}_{T \in \delta^\star} \cup \{switch2aut, switch2env, accept\}
\end{aligned}$$

Environment Mode: In the environment mode, the dynamics of the problem simulates the move of the environment. As this move is not controllable by the agent, it can be simulated with a non-deterministic action that has $2^{|\mathcal{X}|}$ effects, each simulating an assignment to variables in $\mathcal{X}$. Fluents v_l simulate the truth value of variables in $\mathcal{X} \cup \mathcal{Y}$. More precisely, v_x (resp. $v_{\neg x}$) indicates that $x \in \mathcal{X}$ is made true (resp. false), and similarly for $y \in \mathcal{Y}$. In order to reduce the explosion in non-deterministic action effects, we simulate the environment's move with a cascade of non-deterministic actions $move_k$, each one setting (v_{xk}) or unsetting ($v_{\neg xk}$) the value of a variable x_k in $\mathcal{X}$,

$$\begin{aligned}
Pre_{move_k} &= \{env_mode, turn_k\} \\
Eff_{move_k} &= oneof(\{v_{xk}, \neg v_{\neg xk}\}, \{\neg v_{xk}, v_{\neg xk}\}) \cup \Psi_k
\end{aligned}$$

where $\Psi_k = \{turn_{k+1}, \neg turn_k\}$ if $k < |\mathcal{X}|$, and $\Psi_k = \{can\_switch, \neg turn_k\}$ if $k = |\mathcal{X}|$. After the environment's move has been simulated, the *switch2aut* action switches the dynamics to automaton mode, and the automaton configuration (represented by fluents of the form q and q^t) is *frozen* into copies q^s and $q^{s,t}$. Special predicates $at\_horizon(h)$ capture the number of turns from the last recognized accepting state in the plan. If $h < H$, the horizon value is incremented by one.

Automaton Mode: The automaton mode simulates the assignment to variables in $\mathcal{Y}$ and the automaton state transitions. Whereas the update in the automaton configuration is usually understood as a *response* to the observation to variables in $\mathcal{X} \cup \mathcal{Y}$, the dynamics of the encoding take a different perspective: the agent can decide which automaton transitions to perform, and then set the variables in $\mathcal{Y}$ so that the transition guards are satisfied. Such transitions are simulated by means of $trans_T$ actions, one for each $T = (q_i, guard(T), q_j) \in \delta^\star$:

$$Pre_{trans_T} = \{aut\_mode, q_i^s, \neg q_j\} \cup \{\neg v_{\neg l}\}_{l \in guard(T)}$$
$$\mathit{Eff}_{trans_T} = \{q_j\} \cup \{v_l\}_{l \in guard(T)} \cup \Psi_{trans_T}$$

where $\Psi_{trans_T} = \{q_i^{s,t} \rightarrow q_j^t\}$ if $q_j \notin Q_{Fin}$ and $\Psi_{trans_T} = \{can\_accept\}$ otherwise. A transition $T = (q_i, guard(T), q_j)$ can be simulated when there exists a run of the automaton finishing in q_i (as such, q_i had to be *frozen* into q_i^s by means of *switch2aut*). Preconditions include the set $\{\neg v_{\neg l} \mid l \in guard(T)\}$, that checks that the transition guard is not violated by the current assignment to variables. Here, we abuse notation and write $l \in guard(T)$ if the literal l appears in $guard(T)$. As usual, we use the equivalence $\neg(\neg l) = l$. The effects $\{v_l \mid l \in guard(T)\}$ set the variables in $\mathcal{Y}$ so that the guard is satisfied and T can be fired. In parallel, the automaton state fluent q_j is set, as to reflect the transition T. According to the semantics of the tokenized fluents, when $q_i^{s,t}$ holds in the current state the token is progressed into q_j^t to denote a commitment to reach an accepting state. If q_j is indeed an accepting state, then the tokenized fluent is not propagated and instead the fluent can_accept is set. Notably, the conditional effects $q_i^{s,t} \rightarrow q_j^t$ do *not* delete the copies q_i^s and $q_i^{s,t}$. This allows the agent to simulate more than one transition when the automaton is an non-deterministic, thereby capturing multiple runs of the automaton in the planning state (although it is not obliged to simulate all transitions). When the automaton is deterministic, the effects of $trans_T$ allow for at most one transition can be simulated. Finally, the fluent q_j appears negated in the preconditions of $trans_T$ merely for efficiency purposes, as executing $trans_T$ when q_j is true has no value to the plan (and $trans_T$ can be safely pruned).

The agent has two action mechanisms to switch back to environment mode: $accept$ and $switch2env$. At any time during automaton mode, the agent can execute *switch2env* causing all *frozen* copies q^s and $q^{s,t}$ to be deleted. The purpose of *Regularize*, which is optional and defined as $\{\neg v_z, \neg v_{\neg z} \mid z \in \mathcal{X} \cup \mathcal{Y}\}$, is to improve search performance.

$$Pre_{switch2env} = \{aut\_mode\}$$
$$\mathit{Eff}_{switch2env} = \{env\_mode, \neg aut\_mode, turn_1\} \cup \{\neg q^s, \neg q^{s,t} \mid q \in Q\} \cup Regularize$$

The *accept* action is useful to compilations based on Büchi automata, and recognizes runs that have satisfied a commitment to hit an accepting state. At least one of these runs exist if fluent can_accept (which is part of the preconditions) holds true. By executing *accept*, the agent *forgets* those runs that did not satisfy the commitment to hit

an accepting state, and commits to progress the rest of the runs into runs that hit another accepting state. The agent can postpone action *accept* as much as necessary in order to progress all relevant runs into runs that hit an accepting state. Action *accept* has a non-deterministic effect *goal*, introduced artificially as a method to find infinite plans that visit accepting states infinitely often. Full details can be found in [11].

$$
\begin{aligned}
&Pre_{accept}(h) = \{aut\_mode, can\_accept, at\_horizon(h)\}\\
&Eff_{accept}(h) = oneof(\{goal\},\\
&\quad \{turn_1, at\_horizon(0), \neg at\_horizon(h)\} \cup \{env\_mode, \neg aut\_mode, \neg can\_accept\} \cup\\
&\quad \{q^s \rightarrow \neg q^s, q^{s,t} \rightarrow \neg q^{s,t} \mid q \in Q\} \cup \{q \wedge q^t \rightarrow \{\neg q, \neg q^t\} \mid q \in (Q \setminus Q_{Fin})\} \cup\\
&\quad \{q \wedge \neg q^t \rightarrow q^t \mid q \in Q\} \cup Regularize)
\end{aligned}
$$

Initial and Goal States: The initial state of is $\mathcal{I} = \{next(h+1, h) \mid h \in 0 \ldots H\} \cup \{q_0, env\_mode, turn_1, at\_horizon(0)\}$. When the input of the algorithm is a Büchi automaton, the goal is $\mathcal{G} = \{goal\}$. For NFAs and DFAs, the goal is $\mathcal{G} = \{can\_accept\}$.

These steps comprise our compilation of LTL synthesis into FOND, Syn2FOND.

Definition 1 (Syn2FOND). *For* LTL *synthesis problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$, (NBA, DBA, NFA, or DFA) automaton A_φ, and parameter H, the* Syn2FOND compilation *constructs the FOND problem $\mathcal{P}'(\mathcal{X}, \mathcal{Y}, A_\varphi, H) = \langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ as described above.*

Solutions to the compiled problem $\mathcal{P}'$ yield solutions to $\mathcal{P}$ (cf. Theorem 3). The iterated search of solutions to Syn2FOND compilations (with $H = 1, 2 \ldots, 2^{|Q|}$) is guaranteed to succeed, if $\mathcal{P}$ is realizable, when the input automaton is a DBA, NFA, or DFA (cf. Theorem 4). This follows, intuitively, from the fact that if a solution exists, then a strong cyclic policy can be unfolded and simulated in a Syn2FOND compilation search graph. If the agent's strategy cannot always guarantee hitting an accepting state within $H \leq 2^{|Q|}$ turns, then the environment can force a non-accepting cycle – i.e., the environment has a winning strategy that prevents the agent from satisfying the specification. With deterministic automata, the bound can be lowered to $H \leq |Q|$. We illustrate below with a counter-example that completeness is not guaranteed for NBAs.

Theorem 3 (soundness). *Strong-cyclic plans to the Syn2FOND compilation $\mathcal{P}'$ correspond to solutions for $\mathcal{P}$.*

Theorem 4 (completeness). *For a* realizable LTL *synthesis problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$, and NFA automaton A_φ, the* Syn2FOND compilation *$\mathcal{P}'$ is solvable for some $H \leq 2^{|Q|}$. When A_φ is a DBA or DFA, the bound can be lowered to $H \leq |Q|$.*

Incompleteness of the NBA-based compilation The NBA-based compilation is not guaranteed to preserve solutions. Let $\varphi = \bigcirc(\Box x \vee \Diamond \neg x)$, and consider the NBA A_φ with states $Q = \{q_0, q_1, q_2, q_3\}$, $\delta^\star = \{(q_0, \top, q_1), (q_0, \top, q_2), (q_1, x, q_1), (q_2, x, q_2), (q_3, \top, q_3)\}$, and $Q_{Fin} = \{q_1, q_3\}$. The environment can consistently play x a finite, but unbounded number of times before playing $\neg x$ – at which point the runs of the automaton that finish in q_2 must not have been forgotten. There is no bounded parameter H that can satisfy such a requirement.

5 Evaluation

Our main objective for the evaluation is to give a sense of when to choose one formalism over another. Although the same problems can be represented as either LTL synthesis or FOND planning, the choice of formalism can have a dramatic impact on the time required to find a solution. We would expect the FOND setting to be better suited for problems with more "structure", and our results serve to illustrate this hypothesis. In our experiments, we used state-of-the-art synthesis and FOND tools Acacia+ [5] and PRP [14]. Our Syn2FOND algorithm was implemented in a tool we named SynKit. SynKit uses Spot [22] to transform LTL formulae into NBA, and PRP as FOND planner.

We considered some representative problems from both synthesis and FOND perspectives, retrieved from the Syntcomp and IPC competitions, respectively. The first group of problems – **lily** and **loadcomp** (*load*, for short) – come from the synthesis community. The second group of problems – **ttw** and **ctw** – come from the FOND benchmark tireworld. We performed experiments with Acacia$^+$ and our synthesis tool Syn2FOND equipped with PRP as strong cyclic planner. Additionally, we tested PRP in some problems directly encoded as FOND. The first thing to note is the drastic performance hit that can occur converting from one formalism to another. Going from LTL synthesis to FOND is workable in some instances, but the opposite direction proved impossible for even the simplest **ttw** problems that can be solved very efficiently in its native FOND formulation. Automata transformations become a bottleneck in the synthesis tools, causing time (TLE, 30 min) and memory (MLE, 512MB) limit exceptions. This is because the specification requires complex constraints to properly maintain the reachable state-space. It is this "structure" that we conjecture the synthesis tools struggle with, and test separately below with two newly introduced domains.

We encoded directly and compactly in both LTL and FOND the newly introduced domain **build**. The **build** domain addresses the problem of building maintenance, and requires the agent to maintain which rooms have their lights on or off depending on the time of day and whether or not people are in the room. The environment controls the transition between day and night, as well as when people enter or leave a room. The agent must control the lights in response. Problems **build-p#** have # rooms, and problems **build-irr-p#-n** introduce n rooms that may non-deterministically be vacuumed at night (controlled by the environment). Note that this is irrelevant to the task of turning lights on or off (the vacuuming can be done in any lighting condition). For the **build-irr** domain, we could see that the synthesis tools scale far worse as the number of rooms increases (both in generating automata and performing the synthesis). Further, as the number of rooms that need to be vacuumed (which is irrelevant to computing a controller), the relevance reasoning present in the FOND planner is able to cope by largely ignoring the irrelevant aspects of the environment. Conversely, the synthesis component of Acacia+ struggles a great deal. This highlights the strength of the FOND tools for leveraging state relevance to solve problems efficiently.

Finally, we created a synthetic domain that lets us tune the level of "structure" in a problem: more structure leads to fewer possibilities for the environment to act without violating a constraint or assumption. In the **switches** domain, a total of n switches, $s_1 \dots s_n$, initially switched on, need to be all switched off eventually. The environment affects the state of the switches non-deterministically. However, the dynamics of the

	Acacia+			Syn2FOND(PRP)				PRP
Problem	Aut	$N \times \lvert Q \rvert$	Syn	H	Aut	$\lvert Q \rvert$	Search	Search
p4-0	212	6×131	0.01	3	0.16	13	3.1	0
p4-1	11.1	6×195	0.03	3	0.17	13	2.56	0.32
p4-2	1.55	6×181	0.02	3	0.16	13	1.26	0.66
p4-3	0.58	6×46	0.02	3	0.1	12	1.04	0.22
p5-0	TLE	—	—	3	1.33	15	1.02	0
p5-1	TLE	—	—	3	1.59	15	0.88	3.3
p5-2	16991	7×548	0.02	3	1.25	15	0.94	2.14
p5-3	533	7×452	0.08	3	1.34	15	15.8	3.82
p5-4	111	7×236	0.06	3	0.59	14	10.22	4.66
p6-0	TLE	—	—	3	9.11	17	1.9	0
p6-1	TLE	—	—	3	11.8	17	1.8	4.9
p6-2	TLE	—	—	3	11.4	17	1.78	5.54
p6-3	TLE	—	—	3	10.7	17	1.52	15.66
p6-4	TLE	—	—	3	10.4	17	3.66	25.44
p6-5	TLE	—	—	3	6.15	16	3.32	26.18

Table 1: Performance of LTL synthesis and FOND planning on the **switches** problems.

environment is such that immediately after the agent switches off s_k, the environmental non-determinism can only affect the state of a certain number of switches $s_{k'}$, with $k' > k$. A trivial strategy for the agent is to switch off s_1 to s_n in that order. We encoded a series of **switches** problems natively as LTL specifications in TLSF format and also as FOND. Table 1 shows how Acacia+, Syn2FOND, and PRP fared with these problems. They are in three distinct sets (each of increasing number of switches), and within each set the problems range from most structured to least (by varying k). The problems in the first two groups are solved quite readily by PRP, and so the trend is less clear, but for the larger problems we find that PRP struggles when there is less structure and the environment can flip many switches without violating an assumption. This trend also manifests in the Syn2FOND compilations. On the other side, we find that the most structured problems are the most difficult for Acacia+, and the compilation into automata becomes the bottleneck again. On the other hand, the synthesis becomes easier when there is less structure (i.e., more switches can be flipped).

The structure we tune in the **switches** domain is one property of a problem that may favour FOND technology over the synthesis tools. Another is the presence of state variables that are irrelevant. Other notions, such as causal structure in the problem, may also play an important role in features that separate the effectiveness of the two formalisms. We plan to investigate these possibilities in future work.

6 Concluding Remarks

LTL synthesis is an important and challenging problem for which broadly effective tools have remained largely elusive. Motivated by recent advances in the efficiency of FOND planning, this work sought to examine the viability of FOND planning as a computational tool for the realization of LTL synthesis. To this end, we established the theoretical correspondence between LTL synthesis and strong solutions to FOND planning. We also provided the first approach to automatically translate a realizability problem, given by a specification in LTL or $\mathsf{LTL_f}$, into a planning problem described in PDDL. Experiments with state-of-the-art LTL synthesis and FOND solvers highlighted properties that

challenged or supported each of the solvers. Our experiments show automated planning to be a viable and effective tool for highly structured LTL synthesis problems.

Acknowledgements: The authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Fondecyt grant numbers 1150328 and 1161526.

References

1. Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic, Cornell University 1957 **1** (1957) 3–50
2. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of Reactive (1) designs. Journal of Computer and System Sciences **78**(3) (2012) 911–938
3. Pnueli, A.: The temporal logic of programs. In: FOCS. (1977) 46–57
4. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. (1989) 179–190
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: CAV. (2012) 652–657
6. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD. (2006) 117–124
7. De Giacomo, G., Felli, P., Patrizi, F., Sardiña, S.: Two-player game structures for generalized planning and agent composition. In: AAAI. (2010)
8. Patrizi, F., Lipovetzky, N., Geffner, H.: Fair LTL synthesis for non-deterministic systems using strong cyclic planners. In: IJCAI. (2013)
9. Sardiña, S., D'Ippolito, N.: Towards fully observable non-deterministic planning as assumption-based automatic synthesis. In: IJCAI. (2015) 3200–3206
10. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: IJCAI. (2015) 1558–1564
11. Camacho, A., Triantafillou, E., Muise, C., Baier, J.A., McIlraith, S.A.: Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In: AAAI. (2017) 3716–3724
12. Camacho, A., Baier, J.A., Muise, C.J., McIlraith, S.A.: Finite LTL synthesis as planning. In: ICAPS. (2018) To appear.
13. Mattmüller, R., Ortlieb, M., Helmert, M., Bercher, P.: Pattern database heuristics for fully observable nondeterministic planning. In: ICAPS. (2010) 105–112
14. Muise, C., McIlraith, S.A., Beck, J.C.: Improved non-deterministic planning by exploiting state relevance. In: ICAPS. (2012) 172–180
15. Geffner, H., Bonet, B.: A concise introduction to models and methods for automated planning. Synthesis Lectures on Artificial Intelligence and Machine Learning **7**(2) (2013) 1–141
16. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. (2013)
17. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation **115**(1) (1994) 1–37
18. Baier, J.A., McIlraith, S.A.: Planning with temporally extended goals using heuristic search. In: ICAPS. (2006) 342–345
19. Kissmann, P., Edelkamp, S.: Solving fully-observable non-deterministic planning problems via translation into a general game. In: KI09. (2009) 1–8
20. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence **170**(12-13) (2006) 1031–1080
21. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge, MA (2001)
22. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: ATVA. (2016) 122–129