LTL_f Synthesis as AND-OR Graph Search: Knowledge Compilation at Work

Giuseppe De Giacomo[1] , **Marco Favorito**[1,2*] , **Jianwen Li**[3] , **Moshe Y. Vardi**[4] , **Shengping Xiao**[3] and **Shufang Zhu**[1†]

[1]Sapienza University of Rome, [2]Banca d'Italia, [3]East China Normal University, [4]Rice University

degiacomo@diag.uniroma1.it, favorito@diag.uniroma1.it, lijwen2748@gmail.com, vardi@cs.rice.edu, 10175101164@stu.ecnu.edu.cn, zhu@diag.uniroma1.it

Abstract

Synthesis techniques for temporal logic specifications are typically based on exploiting symbolic techniques, as done in model checking. These symbolic techniques typically use backward fixpoint computation. Planning, which can be seen as a specific form of synthesis, is a witness of the success of forward search approaches. In this paper, we develop a forward-search approach to full-fledged Linear Temporal Logic on finite traces (LTL_f) synthesis. We show how to compute the Deterministic Finite Automaton (DFA) of an LTL_f formula on-the-fly, while performing an adversarial forward search towards the final states, by considering the DFA as a sort of AND-OR graph. Our approach is characterized by branching on suitable propositional formulas, instead of individual evaluations, hence radically reducing the branching factor of the search space. Specifically, we take advantage of techniques developed for knowledge compilation, such as Sentential Decision Diagrams (SDDs), to implement the approach efficiently.

1 Introduction

Program synthesis aims at automatically generating a program from declarative specifications expressed in temporal logic [Pnueli and Rosner, 1989; Ehlers *et al.*, 2017]. A commonly used logic for program synthesis is Linear Temporal Logic (LTL), typically used also in model checking [Baier and Katoen, 2008]. Recently, synthesis has been investigated for specifications expressed in LTL_f, a finite-trace variant of LTL [De Giacomo and Vardi, 2013]. Roughly speaking, we consider an alphabet of propositions partitioned into those controlled by the agent (one may think of these as a binary encoding of agent actions) and those controlled by the environment (one may think of these as fluents), and then we use LTL_f to specify which finite traces are desirable. The outcome of the synthesis procedure is a program (a finite-state controller) that at every time step, given the values of the environment propositions in the history so far, sets the next value of the agent propositions so that the traces generated satisfy the LTL_f specification [De Giacomo and Vardi, 2015].

LTL_f synthesis has been proven to be one of the most successful synthesis settings so far. Several tools have been developed recently, among which Lisa [Bansal *et al.*, 2020] and Lydia [De Giacomo and Favorito, 2021] are possibly the best performing ones to date. Both these tools are based on first constructing a DFA corresponding to the LTL_f specification, and then considering it as a game arena where the agent tries to get to an accepting state in spite that the environment tries to avoid it. A winning strategy, which is a finite controller returned by the synthesis procedure, can be obtained through a backward fixpoint computation for *adversarial reachability* of the DFA accepting states [De Giacomo and Vardi, 2015]. The main difficulty of this approach is that it requires computing the entire DFA of the LTL_f specification, which can be, in the worst case, doubly exponential in the size of the specification [De Giacomo and Vardi, 2015]. Hence, even though the backward fixpoint computation can be performed symbolically, enabling scalable performance [Zhu *et al.*, 2017], the DFA construction step can become a significant bottleneck [Zhu *et al.*, 2019].

An alternative approach is to expand the arena while searching for the accepting states via forward search [Xiao *et al.*, 2021], which is analogous to the approach taken by most work in adversarial Planning with fully observable nondeterministic domains (FOND), where the agent controls the actions and the environment controls the fluents [Ghallab *et al.*, 2004; Geffner and Bonet, 2013]. The agent has to reach the goal, despite that the environment may choose adversarially the effects of the agent actions (*strong plans* in FOND) [Cimatti *et al.*, 1998; Cimatti *et al.*, 2003; Geffner and Bonet, 2013]. The typical way to deal with this kind of planning is through forward search on an AND-OR graph [Nilsson, 1971], where the OR-nodes correspond to the choices (quantified existentially) of the agent and the AND-nodes correspond to the choices (quantified universally) of the environment [Mattmüller *et al.*, 2010; Mattmüller, 2013; Geffner and Bonet, 2013]. Note that the search space generated for FOND planning with a compactly represented domain, say, in PDDL [Haslum *et al.*, 2019], is at most single-exponential [Rintanen, 2004].

*The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of Banca d'Italia.

†Corresponding Author

Instead, to handle LTL_f synthesis, we need to deal with a state space that can be of double-exponential size. Searching over a double-exponential state space has been studied in Planning in partially observable nondeterministic domains (POND), aka *contingent planning*, where the search procedure must be performed over the *belief-states* [Reif, 1984; Goldman and Boddy, 1996; Bertoli *et al.*, 2006; Geffner and Bonet, 2013]. However, belief-states have a specific structure [Bertoli *et al.*, 2006; Thanh To *et al.*, 2009], the techniques utilized in contingent planning cannot be directly applied to LTL_f synthesis.

In this work, we investigate LTL_f forward synthesis adopting an AND-OR graph search as in FOND Planning [Mattmüller *et al.*, 2010; Mattmüller, 2013], but over a doubly exponential search space, as for contingent planning [Bertoli *et al.*, 2006]. We do not rely on an encoding into PDDL, as [Camacho *et al.*, 2018; Camacho and A. McIlraith, 2019], which may result into a PDDL specification with exponential size. Instead, we develop specific techniques to create the search space on-the-fly while exploring it, such that we can possibly decide realizability/unrealizability before reaching the worst-case double-exponential blowup.

In details, we propose a technique to create on-the-fly the DFA corresponding to the LTL_f specification. This technique avoids a detour to automata theory and instead builds directly deterministic transitions from a current state. In particular, this technique exploits LTL formula progression [Emerson, 1990; Bacchus and Kabanza, 1998] to separate what happens *now* (label) and what should happen *next* accordingly (successor state). Crucially, we exploit the structure that formula progression provides to branch on propositional formulas (representing several evaluations), instead of individual evaluations. This drastically reduces the branching factor of the AND-OR graph to be searched (recall that in LTL_f synthesis, both the agent choices and the environment choices can be exponentially many). More specifically, we label transitions/edges with propositional formulas on propositions controlled by the agent (for OR-nodes) and by the environment (for AND-nodes). Every such propositional formula captures a set of evaluations leading to the same successor node. We leverage Knowledge Compilation (KC) techniques, and in particular Sentential Decision Diagrams (SDDs) [Darwiche, 2011], to effectively generate such propositional formulas for OR-nodes and AND-nodes, and thus reduce the branching factor of the search space. We implemented our approach in a tool called Cynthia and conducted comprehensive experiments by comparing to existing LTL_f synthesis tools, including Lisa, Lydia and Ltlfsyn from [Xiao *et al.*, 2021] and demonstrate the merits of our approach.

2 Preliminaries

LTL_f Basics. Linear Temporal Logic over finite traces, called LTL_f [De Giacomo and Vardi, 2013] is a variant of Linear Temporal Logic (LTL) [Pnueli, 1977] that is interpreted over finite traces rather than infinite traces (as in LTL). Given a set of propositions $\mathcal{P}$, the syntax of LTL_f is identical to LTL, and defined as (wlog, we require LTL_f formulas are in Negation Normal Form (NNF), i.e., negations only occur in front of atomic propositions): $\varphi ::= tt \mid ff \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \bullet\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{R}\, \varphi_2$. tt is always true, ff is always false; $p \in \mathcal{P}$ is an *atom*, and $\neg p$ is a *negated atom* (a literal l is an atom or the negation of an atom); $\wedge$ (And) and $\vee$ (Or) are the Booelean connectives; and $\bigcirc$ (Next), $\bullet$ (Weak Next), $\mathcal{U}$ (Until) and $\mathcal{R}$ (Release) are temporal connectives. We use the usual abbreviations $true \equiv p \vee \neg p$, $false \equiv p \wedge \neg p$, $\Diamond\varphi \equiv true\,\mathcal{U}\,\varphi$ and $\Box\varphi \equiv false\,\mathcal{R}\,\varphi$. Also for convenience we consider traces $\rho \in (2^{\mathcal{P}})^*$, i.e., we consider also empty traces ϵ as in [Brafman *et al.*, 2018]. More specifically, a trace $\rho = \rho[0], \rho[1], \ldots \in (2^{\mathcal{P}})^*$ is a finite sequence, where $\rho[i]$ $(0 \leq i < |\rho|)$ denotes the i-th interpretation of ρ, which can be considered as the set of propositions that are $true$ at instant i, and $|\rho|$ represents the length of ρ. We have that $\epsilon \models \varphi$ if φ is tt, an $\mathcal{R}$-formula or $\bullet$-formula, hence $\epsilon \models \Box false$. $\epsilon \not\models \varphi$ if φ is ff, a literal, $\mathcal{U}$-formula or $\bigcirc$-formula, hence $\epsilon \not\models \Diamond true$. Detailed semantics of LTL_f can be found in [De Giacomo and Vardi, 2013; Brafman *et al.*, 2018].

We denote by $\mathsf{cl}(\varphi)$ the set of subformulas of φ, including tt and ff. We denote by $\mathsf{pa}(\varphi) \subseteq \mathsf{cl}(\varphi)$ the set of literals and temporal subformulas of φ whose primary connective is temporal [Li *et al.*, 2019]. Formally, for an LTL_f formula φ in NNF, we have $\mathsf{pa}(\varphi) = \{\varphi\}$ if φ is a literal or temporal formula; and $\mathsf{pa}(\varphi) = \mathsf{pa}(\varphi_1) \cup \mathsf{pa}(\varphi_2)$ if $\varphi = (\varphi_1 \wedge \varphi_2)$ or $\varphi = (\varphi_1 \vee \varphi_2)$.

Having LTL_f formula φ, replacing every temporal formula $\psi \in \mathsf{pa}(\varphi)$ with a propositional variable a_ψ gives us a propositional formula φ^p. As a consequence, two formulas φ_1 and φ_2 are propositionally equivalent, denoted by $\varphi_1 \sim_p \varphi_2$, if, $C \models \varphi_1^p \leftrightarrow C \models \varphi_2^p$ holds for every propositional assignment $C \in 2^{\mathsf{pa}(\varphi_1) \cup \mathsf{pa}(\varphi_2)}$. The equivalence class of a formula $\psi \in \mathsf{cl}(\varphi)$ is denoted by $[\psi]_{\sim_p}$ and defined as $[\psi]_{\sim_p} = \{y \in \mathsf{cl}(\varphi) \mid \psi \sim_p y\}$. The quotient set of a subset $C \subseteq \mathsf{cl}(\varphi)$ is denoted by $C/_{\sim_p}$ and defined as $C/_{\sim_p} = \{[\psi]_{\sim_p} \mid \psi \in C\}$.

Definition 1. *An LTL_f formula φ is in neXt Normal Form (XNF) if $\mathsf{pa}(\varphi)$ only includes literals, $\bigcirc$- and $\bullet$-formulas.*

For an LTL_f formula φ in NNF, we can obtain its XNF by transformation function $\mathsf{xnf}(\varphi)$, defined as follows:
- $\mathsf{xnf}(\varphi) = \varphi$ if φ is a literal, $\Box false$, $\Diamond true$, $\bigcirc$-, $\bullet$-formula;
- $\mathsf{xnf}(\varphi_1 \wedge \varphi_2) = \mathsf{xnf}(\varphi_1) \wedge \mathsf{xnf}(\varphi_2)$;
- $\mathsf{xnf}(\varphi_1 \vee \varphi_2) = \mathsf{xnf}(\varphi_1) \vee \mathsf{xnf}(\varphi_2)$;
- $\mathsf{xnf}(\varphi_1 \mathcal{U} \varphi_2) = (\mathsf{xnf}(\varphi_2) \wedge \Diamond true) \vee (\mathsf{xnf}(\varphi_1) \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2))$;
- $\mathsf{xnf}(\varphi_1 \mathcal{R} \varphi_2) = (\mathsf{xnf}(\varphi_2) \vee \Box false) \wedge (\mathsf{xnf}(\varphi_1) \vee \bullet(\varphi_1 \mathcal{R} \varphi_2))$.

Note that $\Diamond true$ (resp. $\Box false$) guarantees that empty trace is not (resp. is) accepted by $\mathcal{U}$-formulas (resp. $\mathcal{R}$-formulas).

Theorem 1 ([Li *et al.*, 2019]). *Every LTL_f formula φ in NNF can be converted, with linear time in the formula size, to an equivalent formula in XNF, denoted by $\mathsf{xnf}(\varphi)$.*

LTL_f Synthesis. The problem of LTL_f synthesis is described as a tuple $(\varphi, \mathcal{X}, \mathcal{Y})$, where φ is an LTL_f formula over $\mathcal{X} \cup \mathcal{Y}$, and $\mathcal{X}, \mathcal{Y}$ are two disjoint sets of variables controlled by the *environment* and the *agent*, respectively. Sometimes, for simplicity, we do not mention $\mathcal{X}$ and $\mathcal{Y}$ explicitly, if they are clear from the context.

Definition 2. *The* synthesis *problem* $(\varphi, \mathcal{X}, \mathcal{Y})$ *aims to computing a strategy* $g : (2^{\mathcal{X}})^* \to 2^{\mathcal{Y}}$*, such that for an arbitrary infinite sequence* $\lambda = X_0, X_1, \ldots \in (2^{\mathcal{X}})^\omega$*, we can find* $k \geq 0$ *such that* $\rho^k \models \varphi$*, where* $\rho^k = (X_0 \cup g(\epsilon)), (X_1 \cup g(X_0)), \ldots, (X_k \cup g(X_0, X_1, \ldots, X_{k-1}))$*. If such a strategy does not exist, then* φ *is unrealizable.*

LTL_f synthesis can be solved by reducing to an adversarial reachability game on the corresponding Deterministic Finite Automaton (DFA) [De Giacomo and Vardi, 2015]. Hence, a strategy can also be represented as a finite-state controller $g : \mathcal{S} \mapsto 2^{\mathcal{Y}}$, where $\mathcal{S}$ denotes the state space of the DFA.

Sentential Decision Diagrams (SDDs). SDDs [Darwiche, 2011] is a Knowledge Compilation (KC) technique designed for an efficient representation and manipulation of Boolean functions. In order to represent a Boolean function, the classical method is applying Shannon decomposition, as done in Ordered Binary Decision Diagrams (BDDs) [Bryant, 1992]. Intuitively, BDD decomposes Boolean functions with one variable at a time. Therefore, the canonicity of BDD is determined wrt a specific ordering of variables. SDD, instead, utilizes a more general decomposition technique that decomposes Boolean functions with a set of variables at each round. Let $f(\mathcal{Y} \cup \mathcal{X})$ be a Boolean function over variables $\mathcal{Y} \cup \mathcal{X}$, where $\mathcal{Y}, \mathcal{X}$ are disjoint. Given an $(\mathcal{Y}, \mathcal{X})$-partition, where $\mathcal{Y}$ variables are considered *primary* and $\mathcal{X}$ variables are considered *subsequent*, the SDD of f, with respect to the $(\mathcal{Y}, \mathcal{X})$-partition, can be written as $\bigvee_{i=1}^{n}[\mathsf{prime}_i(\mathcal{Y}) \wedge \mathsf{sub}_i(\mathcal{X})]$. Intuitively, SDD decomposes f into n children, each of which consists of Boolean functions $\mathsf{prime}_i(\mathcal{Y})$ (what are satisfied *in primary*) and $\mathsf{sub}_i(\mathcal{X})$ (what should be satisfied *in subsequent*, according to $\mathsf{prime}_i(\mathcal{Y})$). In particular, besides that all the primes are disjoint and covering, i.e., $\mathsf{prime}_i \wedge \mathsf{prime}_j = \mathit{false}$ for $i \neq j$, and $\bigvee_{i=1}^{n} \mathsf{prime}_i = \mathit{true}$, SDD also guarantees that all the subs are compressed, i.e., $\mathsf{sub}_i(\mathcal{X}) \neq \mathsf{sub}_j(\mathcal{X})$ for $i \neq j$. Hence, the canonicity of SDDs is determined wrt a specific partition of variables.

3 DFA Construction from LTL_f

The classical approach to LTL_f synthesis first constructs the complete DFA, and then solves an adversarial reachability game through a backward fixpoint computation on this DFA [De Giacomo and Vardi, 2015]. An alternative approach presented in [Xiao *et al.*, 2021] is an on-the-fly synthesis technique that is able to construct the automaton while solving the game in a forward way. Yet, the game arena generated there is explicit, s.t. during search, there can be an exponential number of options to explore at every state, leading to a major drawback for scalability. We now present a new DFA construction based on an incremental technique called *formula progression* that is suitable for exploiting SDDs.

LTL_f Formula Progression. Consider an LTL_f formula φ over $\mathcal{P}$ and a finite trace $\rho = \rho[0], \rho[1], \ldots \in (2^{\mathcal{P}})^*$, in order to have $\rho \models \varphi$, we can start from φ, progress or push φ through ρ. The idea behind *formula progression* is to consider LTL_f formula φ into a requirement about *now* $\rho[i]$, which can be checked straightaway, and a requirement about the future that has to hold on the yet unavailable suffix. That is to say, formula progression looks at $\rho[i]$ and φ, and progresses a new formula $\mathsf{fp}(\varphi, \rho[i])$ such that $\rho, i \models \varphi$ iff $\rho, i+1 \models \mathsf{fp}(\varphi, \rho[i])$. This procedure is analogous to DFA reading trace ρ, where reaching accepting states is essentially achieved by taking one transition after another. Formula progression has been studied in prior work, cf. [Emerson, 1990; Bacchus and Kabanza, 1998]. Here we use it for constructing DFA from LTL_f formulas.

Note that, since ρ is a finite trace, it is necessary to clarify when the trace ends. To do so, we introduce two new formulas $\Box \mathit{false}$ and $\Diamond \mathit{true}$, which, intuitively, refer to *finite trace ends* and *finite trace not ends*, respectively. For simplicity, we enrich $\mathsf{cl}(\varphi)$, the set of proper subformulas of φ, to include them such that $\mathsf{cl}(\varphi)$ is reloaded as $\mathsf{cl}(\varphi) \cup \mathsf{cl}(\Diamond \mathit{true}) \cup \mathsf{cl}(\Box \mathit{false})$.

Definition 3 (LTL_f Formula Progression)**.** *For an* LTL_f *formula* φ *in* NNF*, the progression function* $\mathsf{fp}(\varphi, \sigma)$*, where* $\sigma \in 2^{\mathcal{P}}$*, is defined as follows:*

- $\mathsf{fp}(\mathit{tt}, \sigma) = \mathit{tt}$ *and* $\mathsf{fp}(\mathit{ff}, \sigma) = \mathit{ff}$*;*
- $\mathsf{fp}(p, \sigma) = \mathit{tt}$ *if* $p \in \sigma$*, otherwise* ff*;*
- $\mathsf{fp}(\neg p, \sigma) = \mathit{tt}$ *if* $p \notin \sigma$*, otherwise* ff*;*
- $\mathsf{fp}(\varphi_1 \wedge \varphi_2, \sigma) = \mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\varphi_2, \sigma)$*;*
- $\mathsf{fp}(\varphi_1 \vee \varphi_2, \sigma) = \mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\varphi_2, \sigma)$*;*
- $\mathsf{fp}(\bigcirc \varphi, \sigma) = \varphi \wedge \Diamond \mathit{true}$*;*
- $\mathsf{fp}(\bullet \varphi, \sigma) = \varphi \vee \Box \mathit{false}$*;*
- $\mathsf{fp}(\varphi_1 \mathcal{U} \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \vee (\mathsf{fp}(\varphi_1, \sigma) \wedge \mathsf{fp}(\bigcirc(\varphi_1 \mathcal{U} \varphi_2), \sigma))$*;*
- $\mathsf{fp}(\varphi_1 \mathcal{R} \varphi_2, \sigma) = \mathsf{fp}(\varphi_2, \sigma) \wedge (\mathsf{fp}(\varphi_1, \sigma) \vee \mathsf{fp}(\bullet(\varphi_1 \mathcal{R} \varphi_2), \sigma))$*.*

Note that $\mathsf{fp}(\varphi, \sigma)$ is a positive Boolean formula on $\mathsf{cl}(\varphi)$, i.e., $\mathsf{fp}(\varphi, \sigma) \in \mathcal{B}^+(\mathsf{cl}(\varphi))$.

The following two lemmas show that $\mathsf{fp}(\varphi, \sigma)$ strictly follows LTL_f semantics and retains the propositional behavior of LTL_f formulas.

Lemma 1. *Let* φ *be an* LTL_f *formula over* $\mathcal{P}$ *in* NNF*,* ρ *be a finite nonempty trace,* $\mathsf{fp}(\varphi, \sigma)$ *be as above. We have that* $\rho, i \models \varphi$ *iff* $\rho, i+1 \models \mathsf{fp}(\varphi, \rho[i])$*.*

Lemma 2. *Let* φ *and* ψ *be two* LTL_f *formulas over* $\mathcal{P}$ *in* NNF *s.t.* $\varphi \sim_p \psi$*, and* $\sigma \in 2^{\mathcal{P}}$*. Then* $\mathsf{fp}(\varphi, \sigma) \sim_p \mathsf{fp}(\psi, \sigma)$ *holds.*

We generalize LTL_f formula progression from single instants to finite traces by defining $\mathsf{fp}(\varphi, \epsilon) = \varphi$, and $\mathsf{fp}(\varphi, \sigma u) = \mathsf{fp}(\mathsf{fp}(\varphi, \sigma), u)$, where $\sigma \in 2^{\mathcal{P}}$ and $u \in (2^{\mathcal{P}})^*$.

Lemma 3. *Let* φ *be an* LTL_f *formula over* $\mathcal{P}$ *in* NNF*,* ρ *be a finite trace. We have that* $\rho \models \varphi$ *iff* $\epsilon \models \mathsf{fp}(\varphi, \rho)$*.*

Given an LTL_f formula φ, we can consider it as the initial state, and recursively apply formula progression to obtain all reachable states (through deterministic transitions), denoted by $\text{Reach}(\varphi) = \{\mathsf{fp}(\varphi, \rho) \mid \rho \in (2^{\mathcal{P}})^*\}$. Note that once applying propositional equivalence, there can only be $2^{2^{|\mathsf{cl}(\varphi)|}}$ elements in $\text{Reach}(\varphi)/_{\sim_p}$. Lemma 3 shows that a state $\psi \in \text{Reach}(\varphi)/_{\sim_p}$ can be recognized as accepting iff $\epsilon \models \psi$, indicating that there exists a trace ρ such that $\psi = \mathsf{fp}(\varphi, \rho)$ and ρ is completely "consumed" by formula progression, returning a formula, corresponding to an accepting state, that holds on the empty trace ϵ. In particular, given that every state ψ is actually a positive Boolean formula on $\mathsf{cl}(\varphi)$, checking $\epsilon \models \psi$ only requires dealing with Boolean operators of disjunction and conjunction, which can be done in linear time. The correctness and complexity of our DFA construction are stated below.

Theorem 2. *Given* LTL_f *formula* φ*, the following DFA recognizes* $\mathcal{L}(\varphi)$*:* $\mathcal{A} = (2^{\mathcal{P}}, S, s_0, \delta, Acc)$*, where the states* $S = \text{Reach}(\varphi)/_{\sim_p}$*, the initial state* $s_0 = \varphi_{\sim_p}$*, the transition function* $\delta([\psi]_{\sim_p}, \sigma) = \mathsf{fp}([\psi]_{\sim_p}, \sigma), \forall \sigma \in 2^{\mathcal{P}}$ *and the accepting states* $Acc = \{\psi \mid \epsilon \models \psi\}$*.*

Theorem 3. *Let* φ *be an* LTL_f *formula, the constructed DFA* $\mathcal{A}$ *can have, in the worst case,* $2^{2^{|\text{cl}(\varphi)|}}$ *states.*

4 LTL_f Synthesis as AND-OR Graph Search

Recall that LTL_f synthesis can be viewed as an *adversarial reachability* game on the DFA of the given formula. Interestingly, this game can actually be considered as an AND-OR graph, where the OR-nodes indicate the agent actions (quantified existentially), and the AND-nodes indicate the environment responses (quantified universally). In this case, the DFA construction approach described in the previous section allows us to solve LTL_f synthesis via on-the-fly AND-OR graph search. Now, we present our approach of solving LTL_f synthesis via on-the-fly AND-OR graph search, and explain how to leverage KC techniques, Sentential Decision Diagrams (SDDs) [Darwiche, 2011] to significantly reduce the branching factor of the constructed graph.

4.1 Synthesis Algorithm

Given problem $(\varphi, \mathcal{X}, \mathcal{Y})$, our synthesis algorithm searches for a strategy by exploring the constructed AND-OR graph on the fly. This algorithm is basically a top-down traversal of the search space, proceeding forward from the initial, and excluding strategies that lead to loops. Since we apply the crucial step of propositional equivalence check whenever computing a new state, for simplicity, we omit the propositional equivalence symbol $\sim_p$ and denote every newly constructed DFA state by ψ, instead of $[\psi]_{\sim_p}$, e.g., the initial state is denoted by φ, instead of $[\varphi]_{\sim_p}$. Every DFA state is stored as an OR-node, each outgoing transition (or-arc) leads to an AND-node. Every or-arc is stored as an action-AndNode pair $(act, \textit{AndNd})$. Every outgoing transition (and-arc) of an AND-node is stored as a response-OrNode pair $(resp, n)$. A strategy is stored as a set of state-action pairs. If φ is unrealizable, we obtain strategy as an empty set. In order to avoid exploring the same state over and over, we assign a tag to its associated OR-node n after exploring it. More specifically, n is tagged as *success* if the corresponding DFA state ψ is accepting or there exists an act such that, regardless of what the environment $resp$ is, all corresponding followup OR-nodes are already tagged as *success*. In this case, we also add state-action pair (ψ, act) to strategy. If such act does not exist, n is tagged as *failure*. Moreover, we also put a *loop* tag on an OR-node n if a loop is detected on n, which is considered as temporary failure.

As shown in Algorithm 1, the SYNTHESIS procedure takes a given LTL_f formula φ as input ($\mathcal{X}, \mathcal{Y}$ are omitted for simplicity), and first checks whether the initial state φ is accepting. If this is the case, state-action pair (φ, true) is added to strategy and returned (Line 4). This is because the agent can do whatever it wants (i.e., assign any value to its variables $\mathcal{Y}$) after reaching an accepting state. Otherwise, we initialize the graph by creating an OR-node n out of φ, and start

Algorithm 1 SDD-based Forward Synthesis

```
1:  function SYNTHESIS(φ) return strategy
2:      if ISACCEPTING(φ) then
3:          ADDTOSTRATEGY(φ, true)
4:          return GETSTRATEGY()
5:      INITIALGRAPH(φ)
6:      n := GETGRAPHROOT()
7:      found := SEARCH(n, ∅)
8:      if found then return GETSTRATEGY()
9:      return EMPTYSTRATEGY() ▷ φ is unrealizable

10: function SEARCH(n, path) return True/False
11:     if ISSUCCESSNODE(n) then return True
12:     if ISFAILURENODE(n) then return False
13:     if INPATH(n, path) then ▷ We found a loop
14:         TAGLOOP(n) return False
15:     ψ :=FORMULAOFNODE(n)
16:     if ISACCEPTING(ψ) then
17:         TAGSUCCESSNODE(n)
18:         ADDTOSTRATEGY(ψ, true)
19:         return True
20:     EXPAND (n) ▷ Uses SDD to partition ψ wrt Y and X
21:     for (act, AndNd) ∈GETORARCS(n) do
22:         for (resp, succ) ∈GETANDARCS(AndNd) do
23:             found :=SEARCH(succ, [path|n])
24:             if ¬found then Break
25:         if found then
26:             TAGSUCCESSNODE(n)
27:             ADDTOSTRATEGY(ψ, act)
28:             if ISTAGLOOP(n) then
29:                 BACKPROP(n)
30:             return True
31:     TAGFAILURENODE(n)
32:     return False
```

the main procedure SEARCH. The SEARCH procedure is a recursive routine, taking an OR-node n and the path leading to n as inputs, returning True (resp. False) indicating that a strategy is (resp. isn't) found by the current recursion. Hence, if the outmost SEARCH returns True, a strategy consisting of all state-action pairs added until then is returned (Line 8). Otherwise, an empty strategy is returned.

SEARCH processes an OR-node n by first checking whether n is tagged already, if so, it returns True for *success* tag, and False for *failure* tag. Then, if n exists on path thus leading to a loop, we put a *loop* tag on node n, and return False. Intuitively, when a loop is detected at node n, the procedure returns False, temporally considering n as a failure node. Note that we do not tag n as failure, since it is unknown here whether all the or-arcs of n are explored. Indeed, the returned False will be taken into account when tagging the ancestor nodes of n. Therefore, when later n is tagged as *success*, this information needs to be propagated back to the ancestor nodes of n.

Later on, the procedure continues by checking whether the associated DFA state ψ of n is accepting, if so, n is tagged as

Algorithm 2 Propagate Success Backwards

```
1: function BACKPROP(n)
2:     N := ENQUEUE(EPQUEUE,
3:     while !ISEMPTY(N) do
4:         n_p := DEQUEUE(N)
5:         for (act, AndNd) ∈GETORARCS(n_p) do
6:             if ALLCHILDRENSUCCESS(AndNd) then
7:                 TAGSUCCESSNODE(n_p)
8:                 ψ :=FORMULAOFNODE(n_p)
9:                 ADDTOSTRATEGY(ψ, act)
10:                Ns :=ENQUEUE (N, FAILUREPNS(n_p))
11:                Break
```

Algorithm 3 SDD-based ExpandGraph from An OrNode

```
1: function EXPAND(n)
2:     ψ :=FORMULAOFNODE(n)
3:     T := SDDREPRESENTATION(xnf(ψ))
4:     for child ∈ GETSDDCHILDREN(T) do
5:         act := GETSDDPRIME(child)
6:         AndNd := GETSDDSUB(child)
7:         ADDORARCS(n, act, AndNd)
8:         for child ∈ GETSDDCHILDREN(AndNd) do
9:             resp := GETSDDPRIME(child)
10:            sub := GETSDDSUB(child)
11:            succ := RMNEXT (sub)
12:            ADDANDARCS(AndNd, resp, succ)
```

success, and (ψ, true) is added to the strategy. If none of these checks succeeds, n is expanded by EXPAND, which constructs all its or-arcs $(act, \textit{AndNd})$, and and-arcs $(resp, succ)$ of every *AndNd*. The crucial constraint is that all the agent actions acts of OR-node n should be disjoint and covering, the same with environment responses $resp$ of every *AndNd*. Indeed, EXPAND is based on SDDs, see Section 4.2. As a side-effect, the EXPAND function stores the newly constructed nodes and arcs from n. We explore OR-node n, by iteratively processing the list of AND-nodes *AndNd* connected to n, until a strategy is found (Lines 21-32). In Line 23, we recursively call SEARCH with the by n extended path. For every *AndNd*, once False is detected for searching some $succ$, we give up on the current *AndNd* and proceed with the next one (Line 24). If searching every $succ$ of *AndNd* returns True (Line 25), n is tagged as *success*, and the corresponding state-action pair (ψ, act) is stored. Moreover, if n carries a *loop* tag, it is easy to see that n has been temporally considered as a *failure* node, and this information has been taken into account when tagging the ancestor nodes of n. Therefore, it is necessary to propagate this success information from n backwards to the ancestor nodes of n (Lines 28-29). If no strategy is found after exploring n, we tag it as *failure*, and SEARCH returns False. It should be noted that, in a forward search on an AND-OR graph, it is critical to handle loops with the assistance of this backward propagation, by BACKPROP, as illustrated in [Scutellà, 1990].

As shown in Algorithm 2, BACKPROP is basically a bottom-up traversal of the subgraph rooted at n, that starts from the leaves, and propagates success backwards. In particular, only the nodes that are tagged as *failure* must be considered. This is because once a node n is tagged as *success*, it indicates that n is not affected by any temporary failure of its children. We start from the direct parents of n, and put them in a queue N. For every direct failure parent node n_p of n, n_p can be tagged as *success* only if there exists an or-arc $(act, \textit{AndNd})$, such that all the followup OR-nodes are already tagged as *success*. In this case, the corresponding state-action pair (ψ, act) is stored. Moreover, the success information of n_p should also be propagated, since n_p was tagged as *failure*, which could have affected the tag information of the direct parent nodes of n_p. Therefore, we add the *failure* nodes of them to N. The propagation continues until N gets empty. It is easy to see that the backward propagation does not change the forward nature of the SEARCH procedure, since the backward propagation has to be considered only as an instrument to correctly propagate the *success* whenever needed, i.e., in the presence of loops.

A major challenge arises, however, when looking into the branching factor of this AND-OR graph. Note that, in EXPAND, if we simply use $Y \in 2^{\mathcal{Y}}$ as act for every OR-node n, and $X \in 2^{\mathcal{X}}$ as $resp$ for every *AndNd* connected to n, there can be far too many directions to explore, which leads to crucial performance limitation. Another challenge comes from the propositional equivalence check, which needs to be performed whenever computing a new state. We now explain how to use Sentential Decision Diagrams (SDDs) [Darwiche, 2011] to tackle both these challenges.

4.2 SDD-based EXPAND

The crucial reason of adopting SDDs in the implementation of EXPAND (Algorithm 3), rather than other KC techniques, e.g., BDDs, is that, while maintaining canonicity to check propositional equivalence in constant time, SDDs can provide a disjoint, covering and compressed partition of a Boolean function, wrt a hierarchy of $(\mathcal{Y}, \mathcal{X})$-partition. This allows us to easily partition the transition labels into disjoint agent moves and disjoint environment moves, compressed as much as possible, and so labeling transitions *symbolically* by propositional formulas. Let ψ be the associated DFA state of n, the input of EXPAND(n). The algorithm starts from computing $\mathsf{xnf}(\psi)$, which is equivalent to ψ, and intuitively, encodes all the possibilities of what happens *now*, expressed by $\mathcal{Y} \cup \mathcal{X}$ variables, and what happens *next* accordingly, expressed by variables $\mathcal{Z} = \bigcup_{\theta \in \mathsf{cl}(\varphi)} \{z_\alpha | \alpha \in \mathsf{pa}(\mathsf{xnf}(\theta)), \alpha \text{ not literal}\}$. Note that it is crucial to consider the closure of the original LTL$_f$ formula φ, instead of the current state ψ, as the propositional equivalence check between two states requires their SDDs to be defined over the same set of variables.

In Line 3, we represent $\mathsf{xnf}(\psi)$, considering it as a propositional formula over $\mathsf{pa}(\mathsf{xnf}(\psi))$, into an SDD $T := \bigvee_{i=1}^{n} [\mathsf{prime}_i(\mathcal{Y}) \wedge \mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})]$ such that all Ys leading to the same set of possible successors $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$ (X is not decided yet) are clustered into a propositional formula $\mathsf{prime}_i(\mathcal{Y})$. $\mathsf{prime}_i(\mathcal{Y})$ and $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z})$ are extracted as act and corresponding *AndNd*, respectively (Lines 5&6). Moreover, $\mathsf{sub}_i(\mathcal{X} \cup \mathcal{Z}) = \bigvee_{j=1}^{m} [\mathsf{prime}_{i,j}(\mathcal{X}) \wedge \mathsf{sub}_{i,j}(\mathcal{Z})]$ is

such that all Xs leading to the same successor are clustered into formula $\mathsf{prime}_j(\mathcal{X})$, and $\mathsf{sub}_{i,j}(\mathcal{Z})$ refers to the successor state of agent-env choices $(\mathsf{prime}_i(\mathcal{Y}), \mathsf{prime}_{i,j}(\mathcal{X}))$. They are extracted as $resp$ and corresponding $succ$, respectively (Lines 9-11). Note that SDDs guarantee that all disjuncts generated are disjoint, covering and compressed, hence we can use SDDs to reduce the branching factor as much as possible. In particular, every successor state $succ$ is obtained by stripping $\circ$ and $\bullet$, introduced by XNF, through the *remove-next* function RMNEXT, defined below:

- $\text{RMNEXT}(\Diamond true) = tt$, $\text{RMNEXT}(\Box false) = ff$
- $\text{RMNEXT}(\varphi_1 \wedge \varphi_2) = \text{RMNEXT}(\varphi_1) \wedge \text{RMNEXT}(\varphi_2)$
- $\text{RMNEXT}(\varphi_1 \vee \varphi_2) = \text{RMNEXT}(\varphi_1) \vee \text{RMNEXT}(\varphi_2)$
- $\text{RMNEXT}(\circ\varphi) = \varphi \wedge \Diamond true$, $\text{RMNEXT}(\bullet\varphi) = \varphi \vee \Box false$

Note that RMNEXT applies to neither $\mathcal{U}$-,$\mathcal{R}$- formulas, since they do not appear in XNF, nor literals (p, $\neg p$), since its input is a propositional formula over variables $\mathcal{Z}$ that does not contain literals.

Lemma 4. *Algorithm 3 is correct, i.e., given an OR node n,* EXPAND *correctly expands the search graph.*

Proof. (Sketch) We prove the lemma by first showing that the XNF of the LTL_f formula ψ, associated to n, essentially captures all the transitions that can be obtained by applying formula progression on ψ. The SDD representation of $\mathsf{xnf}(\psi)$ correctly provides a disjoint and covering partitions of all the transitions. Hence, the SDD-based EXPAND is correct. □

Theorem 4. *Algorithm 1 terminates in at most double-exponential time, in the size of φ of problem $(\varphi, \mathcal{X}, \mathcal{Y})$.*

Proof. (Sketch) This is guaranteed by the fact that the number of recursive calls in SEARCH is bounded by the worst-case doubly-exponential number of states in the constructed DFA via the SDD-based technique. Note that every recursive call first checks for *success*, *failure*, and loop (Lines 11-19). Then if the recursive call gets to Line 20, it will eventually tag the current node as *success* or *failure*. Therefore, every node is explored only once in a forward manner. Note that if a *success* node n was also tagged *loop*, BACKPROP is called before completing the current recursive call. BACKPROP is essentially a Breadth First Search (BFS) on the subgraph rooted at n. Since there can be at most linear number of $(n_1, (act, resp), n_2)$ edges in this subgraph, and every $(n_1, (act, resp), n_2)$ edge is visited only once during the BFS, BACKPROP terminates in linear time in the size of the subgraph rooted at n. Hence, we conclude that Algorithm 1 terminates in at most double-exponential time, in the size of φ of problem $(\varphi, \mathcal{X}, \mathcal{Y})$. □

Theorem 5. *Algorithm 1 is correct, i.e., it returns a non-empty strategy iff the given synthesis problem is realizable.*

Proof. (Sketch) We prove by showing the main recursive procedure SEARCH is correct. If SEARCH does not detect any loops on the graph, we can see that once an OR-node n is tagged, the tag is stored until the algorithm terminates. n is tagged as *success* if either n is detected as accepting, or there exists an agent act, following which, all the successors are also tagged as *success*, regardless of what the environment $resp$ is, and the current recursion returns True. Note that, only in this case, the corresponding state-action pair is added to strategy. If neither condition happens, n is tagged as *failure*, and the current recursion returns False.

If SEARCH detects a loop on n in the graph, the presence of the loop leads to temporary failure. It could happen that a parent node n_p (also further ancestor nodes) of n is tagged as *failure* due to this temporary failure of n. Therefore, once n is tagged as *success*, the *success* tag should be propagated in the loops through BACKPROP. BACKPROP is correct, since the tag of a parent node n_p (also further ancestor nodes) changes from *failure* to *success* iff there exists an agent act, following which, all the successors are also tagged as *success*, regardless of what the environment $resp$ is. Therefore, BACKPROP is able to eliminate the temporary failure caused by loops. Hence, if a *failure* tag stays until the algorithm terminates, this is a confirmed *failure* that is not affected by any temporary failure. As a result, Algorithm 1 terminates with the initial node tagged as *success* and returns a non-empty strategy iff the given synthesis problem is realizable. □

5 Implementation and Empirical Evaluations

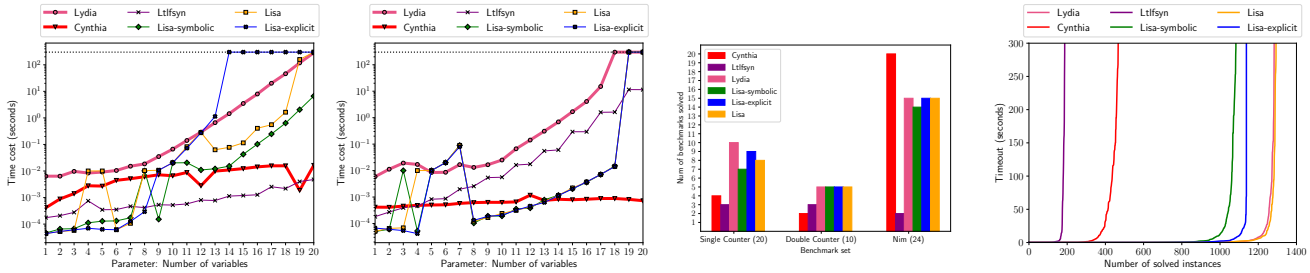
We implemented the forward synthesis problem presented in Section 4 in a tool called Cynthia in C++ [1]. Cynthia is able to take an LTL_f synthesis problem $(\varphi, \mathcal{X}, \mathcal{Y})$ and constructs a strategy that realizes φ if one exists. We make use of library SDD-2.0 (http://reasoning.cs.ucla.edu/sdd) to handle all SDD related operations.

Optimizations. Cynthia applies some optimizations to speed up the synthesis procedure. First, right before EXPAND an OR-node n, we perform the pre-processing techniques described in [Xiao *et al.*, 2021]. More specifically, we check: (*i*) there exists a one-step strategy that reaches accepting states from n, then n is tagged as success; or (*ii*) there does not exist an agent move that can avoid sink state (a non-accepting state only going back to itself) from n, then n is tagged as failure. Moreover, despite being a depth-first search, the SDD-based $\text{EXPAND}(n)$, in fact, constructs all the connected *AndNd* of n, and followup OR-nodes $succ$ at once, which allows us to conduct a "look-ahead" check. More specifically, this "look-ahead" check tries to tag constructed $succ$ by the pre-processing techniques to speed up further search.

Experimental Methodology. We evaluated the efficiency of Cynthia, by comparing against the following tools: Lisa [Bansal *et al.*, 2020] and Lydia [De Giacomo and Favorito, 2021] are state-of-the-art backward LTL_f synthesis approaches. Both tools compute the complete DFA first, and then solve an adversarial reachability game following the symbolic backward computation technique described in [Zhu *et al.*, 2017]. Ltlfsyn [Xiao *et al.*, 2021] implements a SAT-based on-the-fly forward synthesis approach.

Experiment Setup. Experiments were run on a computer cluster, where each instance took exclusive access to a computing node with Intel-Xeon processor running at 2.6 GHz, with 8GB of memory and 300 seconds of time limit. The correctness of Cynthia was empirically verified by comparing

[1] Tool available at https://whitemech.github.io/cynthia.



(a) GF-pattern. (b) U-pattern. (c) *Two-player-Games*. (d) *Random*.

Figure 1: Comparison results on all benchmarks.

the results with those from all baseline tools. No inconsistencies were encountered for all solved instances.

Benchmarks. We collected, in total, 1494 LTL_f synthesis instances from literature, consisting of 3 benchmark families: 40 *Patterns* instances [Xiao *et al.*, 2021]; 54 *Two-player-Games* instances [Tabajara and Vardi, 2019; Bansal *et al.*, 2020]; 1400 *Random* instances [Zhu *et al.*, 2017; De Giacomo and Favorito, 2021].

Results. Figure 1a and Figure 1b show the running time of each tool on every instance of the GF-, and U-pattern, respectively. Across these instances, we observe that Cynthia is able to solve all instances with much less time comparing to backward approaches, represented by Lisa and Lydia. Comparing to Ltlfsyn, Cynthia is able to achieve comparative performance on the GF-pattern instances, with time cost difference of $<$1 second (y-axis is in log scale), see Figure 1a. On the U-pattern instances, Cynthia shows significantly better performance, see Figure 1b. On the *Two-player-Games* benchmarks, see Figure 1c, we observe that Cynthia is able to dominate all other tools on the *Nim* instances. Yet, on both *Counter(s)* instances, backward approaches show better performance over all forward approaches, and Cynthia is almost on par with Ltlfsyn. On the *Random* benchmarks, Cynthia, in general, performs better than Ltlfsyn, by solving more instances with less time, see Figure 1d. Nevertheless, Cynthia cannot beat backward approaches.

Analysis. It is clear from the plots that Cynthia, in general, shows an overall better performance than Ltlfsyn, illustrating the efficiency and better scalability of our approach. In particular, there is a notable outperformance of Cynthia on the U-pattern instances, see Figure 1b. The challenge in the U-pattern instances lies mostly in proving realizable, and can be achieved by just satisfying variables under control. Since every variable appears only once on the right side of the $\mathcal{U}$ operator, our approach is able to compress the branching labels as propositional formulas, such that highly reducing the branching factor and thus speeding up the search procedure.

When comparing Cynthia with backward approaches integrated tools, it should be noted that, in general, forward approaches perform well on the instances where the result can be obtained far before exploring the whole search space. In our benchmarks, this is exactly what happens for *Nim* and *Pattern* instances, where Cynthia shows dominating performance over all tools, which demonstrates the promising efficiency of forward synthesis.

On the other hand, backward approaches perform better when it is necessary to explore the entire search space. In the case of the *Counter(s)* instances, due to their specific structure, in order to obtain a strategy, the searching space to explore grows exponentially fast. In particular, the branching factor of AND-nodes, even after clustering, can remain exponential in the number of environment variables, and so leaving no space to further reduce. Nevertheless, backward approaches can leverage powerful minimization to highly reduce the searching space such that achieving better performance, as also observed in [Tabajara and Vardi, 2019]. For the *Random* instances, which are randomly conjuncted LTL_f formulas, the advantage of possibly being lucky and finding a solution quickly without exploring the entire search space is overwhelmed by the fact that backward approaches integrated with composition techniques [Bansal *et al.*, 2020; De Giacomo and Favorito, 2021] are able to first decompose the conjuncted formula into smaller pieces, obtain the minimized DFA of each conjunct and then compose them for final game solving. It might be possible that similar composition ideas could be leveraged to forward synthesis approaches as well, although further research is necessary in this direction.

6 Conclusions

We investigated the effectiveness of forward search in LTL_f synthesis. We observed that even an uninformed search is able to drastically improve the synthesis capability in several cases (as in the *Nim* benchmarks above). This shows that our approach is quite promising, especially considering that we could move from the uninformed search presented here to informed search exploiting heuristics [Mattmüller, 2013; Jiménez and Torras, 2000].

Acknowledgments

This work is partially supported by the ERC Advanced Grant WhiteMech (No. 834228), the EU ICT-48 2020 project TAILOR (No. 952215), the PRIN project RIPER (No. 20203FFYLK), the JPMorgan AI Faculty Research Award "Resilience-based Generalized Planning and Strategic Reasoning", NSFC No. 62002118, No. U21B2015 and Shanghai Pujiang Talent Plan No. 20PJ1403500, NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD

MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

References

[Bacchus and Kabanza, 1998] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. Math. Artif. Intell.*, 22(1-2), 1998.

[Baier and Katoen, 2008] C. Baier and JP. Katoen. *Principles of model checking*. 2008.

[Bansal *et al.*, 2020] S. Bansal, Y. Li, L. M. Tabajara, and M. Y. Vardi. Hybrid Compositional Reasoning for Reactive Synthesis from Finite-Horizon Specifications. In *AAAI*, 2020.

[Bertoli *et al.*, 2006] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Strong planning under partial observability. *Artif. Intell.*, 170(4-5), 2006.

[Brafman *et al.*, 2018] R. I. Brafman, G. De Giacomo, and F. Patrizi. LTL_f/LDL_f non-markovian rewards. In *AAAI*, 2018.

[Bryant, 1992] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3), 1992.

[Camacho and A. McIlraith, 2019] A. Camacho and S. A. McIlraith. Strong fully observable non-deterministic planning with LTL and LTL_f goals. In *IJCAI*, 2019.

[Camacho *et al.*, 2018] A. Camacho, J. A. Baier, C. J. Muise, and S. A. McIlraith. Finite LTL Synthesis as Planning. In *ICAPS*, 2018.

[Cimatti *et al.*, 1998] A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *AIPS*, 1998.

[Cimatti *et al.*, 2003] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. 1–2(147), 2003.

[Darwiche, 2011] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 2011.

[De Giacomo and Favorito, 2021] G. De Giacomo and M. Favorito. Compositional approach to translate LTL_f/LDL_f into deterministic finite automata. In *ICAPS*, 2021.

[De Giacomo and Vardi, 2013] G. De Giacomo and M. Y. Vardi. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*, 2013.

[De Giacomo and Vardi, 2015] G. De Giacomo and M. Y. Vardi. Synthesis for LTL and LDL on Finite Traces. In *IJCAI*, 2015.

[Ehlers *et al.*, 2017] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi. Supervisory control and reactive synthesis: a comparative introduction. *Discret. Event Dyn. Syst.*, 27(2), 2017.

[Emerson, 1990] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, 1990.

[Geffner and Bonet, 2013] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. 2013.

[Ghallab *et al.*, 2004] M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning - theory and practice*. 2004.

[Goldman and Boddy, 1996] R. P. Goldman and M. S. Boddy. Expressive planning and explicit knowledge. In *AIPS*, 1996.

[Haslum *et al.*, 2019] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. *An Introduction to the Planning Domain Definition Language*. 2019.

[Jiménez and Torras, 2000] P. Jiménez and C. Torras. An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artif. Intell.*, 124(1), 2000.

[Li *et al.*, 2019] J. Li, K. Y. Rozier, G. Pu, Y. Zhang, and M. Y. Vardi. Sat-based explicit LTL_f satisfiability checking. In *AAAI*, 2019.

[Mattmüller *et al.*, 2010] R. Mattmüller, M. Ortlieb, M. Helmert, and P. Bercher. Pattern database heuristics for fully observable nondeterministic planning. In *ICAPS*, 2010.

[Mattmüller, 2013] R. Mattmüller. *Informed progression search for fully observable nondeterministic planning*. PhD thesis, 2013.

[Nilsson, 1971] N. J. Nilsson. *Problem-solving methods in artificial intelligence*. 1971.

[Pnueli and Rosner, 1989] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *POPL*, 1989.

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

[Reif, 1984] J. H. Reif. The complexity of two-player games of incomplete information. *JCSS*, 29(2), 1984.

[Rintanen, 2004] J. Rintanen. Complexity of planning with partial observability. In *ICAPS*, 2004.

[Scutellà, 1990] M. G. Scutellà. A note on dowling and gallier's top-down algorithm for propositional horn satisfiability. *J. Log. Program.*, 8(3):265–273, 1990.

[Tabajara and Vardi, 2019] L. M. Tabajara and M. Y. Vardi. Partitioning Techniques in LTL_f Synthesis. In *IJCAI*, 2019.

[Thanh To *et al.*, 2009] S. Thanh To, E. Pontelli, and T. Cao Son. A conformant planner with explicit disjunctive representation of belief states. In *ICAPS*, 2009.

[Xiao *et al.*, 2021] S. Xiao, J. Li, S. Zhu, Y. Shi, G. Pu, and M. Y. Vardi. On-the-fly synthesis for LTL over finite traces. In *AAAI*, 2021.

[Zhu *et al.*, 2017] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi. Symbolic LTL_f Synthesis. In *IJCAI*, 2017.

[Zhu *et al.*, 2019] S. Zhu, G. Pu, and M. Y. Vardi. First-Order vs. Second-Order Encodings for LTL_f-to-Automata Translation. In *TAMC*, 2019.