

Forward LTL_f Synthesis: DPLL At Work

Marco Favorito

Banca d'Italia, Italy

Abstract

This paper proposes a new AND-OR graph search approach for synthesis of formulas in Linear Temporal Logic on finite traces (LTL_f) that overcomes some limitations of previous works. We devise a procedure inspired by the Davis-Putnam-Logemann-Loveland (DPLL) algorithm to generate the next available agent-environment moves in a truly depth-first fashion, possibly avoiding exhaustive enumeration or costly compilations. We also propose the use of an equivalence check for search nodes based on the syntactic equivalence of state formulas. Since the resulting procedure is not guaranteed to terminate, we identify a stopping condition to abort execution and restart the search with state-equivalence checking based on Binary Decision Diagrams (BDD), which we show to be correct. The experimental results show that in many cases the proposed technique outperforms other state-of-the-art approaches. Our implementation Nike competed in the LTL_f Realizability Track in the 2023 edition of SYNTCOMP, and won the competition.

Keywords

Linear temporal logic on finite traces, LTL_f Synthesis, AND-OR Graph Search

1. Introduction

Program synthesis is the task of finding a program that provably satisfies a given high-level formal specification [2]. A commonly used logic for program synthesis is Linear Temporal Logic (LTL) [3, 4], typically used also in model checking [5]. LTL on *finite traces* (LTL_f) [6], a variant of LTL to specify *finite*-horizon temporal properties, has been recently proposed as specification language for temporal synthesis [7]. The LTL_f synthesis setting considers a set of variables controllable by the agent, a (disjoint) set of variables controlled by the environment, and a LTL_f specification that specifies which finite traces over such variables are desirable. The problem of LTL_f synthesis consists in finding a finite-state controller that, at every time step, given the values of the environment variables in the history so far, sets the next values for each agent proposition such that the generated traces comply with the LTL_f specification.

The basic technique for solving LTL_f synthesis amounts to constructing a deterministic finite automaton (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 regardless of the environment's moves.

IPS-RCRA-SPIRIT 2023: Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy. November 7-9th, 2023, Rome, Italy [1]


✉ marco.favorito@bancaditalia.it (M. Favorito)

🌐 <https://marcofavorito.me/> (M. Favorito)

🆔 0000-0002-0877-7063 (M. Favorito)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

A *winning strategy*, i.e. a finite controller returned by the procedure, can be obtained through a backward fixpoint computation for *adversarial reachability* of the DFA accepting state.

Related works. State-of-the-art tools such as Lydia [8] and Lisa [9] are based on the classical approach. The main drawback of this technique is that it requires to compute the entire DFA of the LTL_f specification, which in the worst case can be doubly exponential in the size of the formula. Therefore, the DFA construction step becomes the main bottleneck.

A natural idea is to consider a forward search approach that expands the arena on-the-fly while searching for a solution, possibly avoiding the construction of the entire arena. Forward-based approaches are at the core of the best solution methods designed for other AI problems: Planning with fully observable non-deterministic domains (FOND) [10, 11, 12, 13], where the agent has to reach the goal, despite that the environment may choose adversarially the effects of the agent actions, and Planning in partially observable nondeterministic domains (POND), also known as *contingent planning*, where the search procedure must be performed over the *belief-states* [14, 15, 16]. However, techniques developed for such problems cannot be applied to ours: in a FOND planning problem, represented with PDDL [17], the search space is at most single-exponential [18], whereas for LTL_f synthesis the state space can be of double-exponential size wrt the size of the formula; hence, we do not rely on an encoding into PDDL, as [19, 20], which may result in a PDDL specification with exponential size. In POND planning, despite the double-exponential size of the state space, belief-states have a specific structure [16, 21], so their solution techniques cannot be directly applied to LTL_f synthesis.

For these reasons, researchers have been looking into forward search techniques specifically conceived for solving LTL_f synthesis. Two notable attempts in this direction have been presented in [22] and [23]. The former work presents an on-the-fly synthesis approach via conducting a so-called Transition-based Deterministic Finite Automata (TDFA) game, where the acceptance condition is defined on transitions, instead of states. The main issue of that approach is the full enumeration of agent-environment moves, which are exponentially many in the number of variables. Moreover, due to the fact that the acceptance condition is defined on transitions, every generated transition has to be checked for acceptance. The latter work instead proposes a search framework for LTL_f synthesis, where the DFA arena is seen as an AND-OR graph. The available moves are found according to the formula associated with the current search node by means of a Knowledge Compilation technique: Sentential Decision Diagrams (SDD) [24]. Notably, they are able to branch on propositional formulas, representing several evaluations, instead of individual ones. This can drastically reduce the branching factor. Nevertheless, for certain types of problem instances, the approach can get stuck with demanding compilations of the state formulas, needed *both* for state equivalence checking and for search node expansion. Moreover, the requirement of having an irreducible representation of players' moves can be of little usefulness if the branching factor of the search problem is already high, hence resulting in an even more significant compilation overhead.

We think there is a need for a search approach that scales well with the increase of computational power and that uses such power for actually exploring the search space rather than wasting time either slavishly enumerating the exponentially many variable assignments, or by finding the minimal representation of the available search moves.

Contributions. First, we focus our attention on two primitive operations for forward LTL_f synthesis: *state-equivalence checking* and *search node expansion*, and explain at high-level how

these are combined in our approach, highlighting limitations of previous related works. Then, we formalize and discuss two well-known instances of equivalence checks; one based on knowledge compilation, and the other on a computationally-cheap syntactical equivalence between state formulas. Furthermore, we propose a novel search graph expansion technique, based on a procedure inspired by the famous Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Finally, we describe the implementation of a new tool, Nike, that integrates the proposed modules in the search procedure, and compare its performance on known benchmarks with other state-of-the-art tools, showing its surprising effectiveness. Nike won the LTL_f Realizability Track in the 2023 edition of SYNTCOMP ¹. The complete technical report, including detailed proofs and comprehensive experimental results, is available on arXiv [25].

2. Preliminaries

LTL_f Basics. Linear Temporal Logic over finite traces, called LTL_f [6] is a variant of Linear Temporal Logic (LTL) [5] 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 that LTL_f formulas are in Negation Normal Form (NNF), i.e., negations only occur in front of atomic propositions):

$$\varphi ::= tt \mid ff \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \text{O}\varphi \mid \bullet\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \mid \varphi_1 \mathcal{R} \varphi_2$$

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 Boolean connectives; and O (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 $\square\varphi \equiv false \mathcal{R} \varphi$. Also for convenience we consider traces $\rho \in (2^{\mathcal{P}})^*$, i.e., we consider also empty traces ϵ as in [26]. More specifically, a trace $\rho = \rho[0], \rho[1], \dots \in (2^{\mathcal{P}})^*$ 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 $\epsilon \models \square false$. $\epsilon \not\models \varphi$ if φ is ff , a literal, \mathcal{U} -formula or O -formula, hence $\epsilon \not\models \diamond true$.

We consider the semantics of LTL_f as presented in [26], that also works for empty traces. Given a finite trace ρ and an LTL_f formula φ , we inductively define when φ is *true* for ρ at point i ($0 \leq i < |\rho|$), written $\rho, i \models \varphi$, as follows:

- $\rho, i \models tt$ and $\rho, i \not\models ff$;
- $\rho, i \models p$ iff $p \in \rho[i]$, and $\rho, i \models \neg p$ iff $p \notin \rho[i]$;
- $\rho, i \models \varphi_1 \wedge \varphi_2$ iff $\rho, i \models \varphi_1$ and $\rho, i \models \varphi_2$;
- $\rho, i \models \varphi_1 \vee \varphi_2$ iff $\rho, i \models \varphi_1$ or $\rho, i \models \varphi_2$;
- $\rho, i \models \text{O}\varphi$ iff $0 \leq i < |\rho| - 1$ and $\rho, i + 1 \models \varphi$;
- $\rho, i \models \bullet\varphi$ iff $0 \leq i < |\rho|$ implies $\rho, i + 1 \models \varphi$;
- $\rho, i \models \varphi_1 \mathcal{U} \varphi_2$ iff there exists j with $i \leq j < |\rho|$ such that $\rho, j \models \varphi_2$, and $\forall k. i \leq k < j$ we have $\rho, k \models \varphi_1$;

¹<http://www.syntcomp.org/syntcomp-2023-results/>

- $\rho, i \models \varphi_1 \mathcal{R} \varphi_2$ iff for all j with $i \leq j < |\rho|$ either we have $\rho, j \models \varphi_2$, or $\exists k. i \leq k < j$ we have $\rho, k \models \varphi_1$.

An LTL_f formula φ is *true* for ρ , denoted by $\rho \models \varphi$, if and only if $\rho, 0 \models \varphi$. In particular, $\epsilon \models \Box \text{false}$ and $\epsilon \not\models \Diamond \text{true}$.

We denote by $\text{cl}(\varphi)$ the set of subformulas of φ , including *tt* and *ff*. We denote by $\text{pa}(\varphi) \subseteq \text{cl}(\varphi)$ the set of literals and temporal subformulas of φ whose primary connective is temporal [27]. Formally, for an LTL_f formula φ in NNF , we have $\text{pa}(\varphi) = \{\varphi\}$ if φ is a literal or temporal formula; and $\text{pa}(\varphi) = \text{pa}(\varphi_1) \cup \text{pa}(\varphi_2)$ if $\varphi = (\varphi_1 \wedge \varphi_2)$ or $\varphi = (\varphi_1 \vee \varphi_2)$. Having LTL_f formula φ , replacing every temporal formula $\psi \in \text{pa}(\varphi)$ with a propositional variable a_ψ gives us a propositional formula φ^p ; we call this operation *propositionalization* of φ . Note that $\varphi^p \in \mathcal{B}^+(\text{cl}(\varphi))$, i.e. φ^p is a positive Boolean formula over variables $\text{cl}(\varphi)$. Let $\phi = \varphi^p$, we denote with $\phi^{\text{tf}} = \varphi$ the inverse operation of \cdot^p . 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 $C \in 2^{\text{pa}(\varphi_1) \cup \text{pa}(\varphi_2)}$.

An LTL_f formula φ is in *neXt Normal Form (XNF)* if $\text{pa}(\varphi)$ only includes literals, \bigcirc - and \bullet -formulas. For an LTL_f formula φ in NNF , we can obtain its XNF by transformation function $\text{xfn}(\varphi)$, defined as follows:

- $\text{xfn}(\varphi) = \varphi$ if φ is a literal, $\Box \text{false}$, $\Diamond \text{true}$, \bigcirc -, \bullet -formula;
- $\text{xfn}(\varphi_1 \wedge \varphi_2) = \text{xfn}(\varphi_1) \wedge \text{xfn}(\varphi_2)$;
- $\text{xfn}(\varphi_1 \vee \varphi_2) = \text{xfn}(\varphi_1) \vee \text{xfn}(\varphi_2)$;
- $\text{xfn}(\varphi_1 \mathcal{U} \varphi_2) = (\text{xfn}(\varphi_2) \wedge \Diamond \text{true}) \vee (\text{xfn}(\varphi_1) \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2))$;
- $\text{xfn}(\varphi_1 \mathcal{R} \varphi_2) = (\text{xfn}(\varphi_2) \vee \Box \text{false}) \wedge (\text{xfn}(\varphi_1) \vee \bullet(\varphi_1 \mathcal{R} \varphi_2))$.

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

Theorem 1 ([27]). *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 $\text{xfn}(\varphi)$.*

LTL_f Formula Progression [23]. Consider an LTL_f formula φ over \mathcal{P} and a finite trace $\rho = \rho[0], \rho[1], \dots \in (2^{\mathcal{P}})^*$, 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 split an 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 in the yet unavailable suffix. That is to say, formula progression looks at $\rho[i]$ and φ , and progresses a new formula $\text{fp}(\varphi, \rho[i])$ such that $\rho, i \models \varphi$ iff $\rho, i+1 \models \text{fp}(\varphi, \rho[i])$. 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. [28, 29]. Here we use the formalization provided in [23].

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

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

- $\text{fp}(tt, \sigma) = tt$ and $\text{fp}(ff, \sigma) = ff$;
- $\text{fp}(p, \sigma) = tt$ if $p \in \sigma$, otherwise ff ;

- $\text{fp}(\diamond(\text{true}), \sigma) = tt$ and $\text{fp}(\square(\text{false}), \sigma) = ff$;
- $\text{fp}(\neg p, \sigma) = tt$ if $p \notin \sigma$, otherwise ff ;
- $\text{fp}(\varphi_1 \wedge \varphi_2, \sigma) = \text{fp}(\varphi_1, \sigma) \wedge \text{fp}(\varphi_2, \sigma)$;
- $\text{fp}(\varphi_1 \vee \varphi_2, \sigma) = \text{fp}(\varphi_1, \sigma) \vee \text{fp}(\varphi_2, \sigma)$;
- $\text{fp}(\bigcirc\varphi, \sigma) = \varphi \wedge \diamond \text{true}$, and $\text{fp}(\bullet\varphi, \sigma) = \varphi \vee \square \text{false}$;
- $\text{fp}(\varphi_1 \mathcal{U} \varphi_2, \sigma) = \text{fp}(\varphi_2, \sigma) \vee (\text{fp}(\varphi_1, \sigma) \wedge \text{fp}(\bigcirc(\varphi_1 \mathcal{U} \varphi_2), \sigma))$;
- $\text{fp}(\varphi_1 \mathcal{R} \varphi_2, \sigma) = \text{fp}(\varphi_2, \sigma) \wedge (\text{fp}(\varphi_1, \sigma) \vee \text{fp}(\bullet(\varphi_1 \mathcal{R} \varphi_2), \sigma))$.

Note that $\text{fp}(\varphi, \sigma)$ is a positive Boolean formula on $\text{cl}(\varphi)$, i.e., $\text{fp}(\varphi, \sigma) \in \mathcal{B}^+(\text{cl}(\varphi))$. The following two propositions show that $\text{fp}(\varphi, \sigma)$ strictly follows LTL_f semantics and retains the propositional behavior of LTL_f formulas.

Proposition 1 ([23]). *Let φ be an LTL_f formula over \mathcal{P} in NNF, ρ be a finite nonempty trace, $\text{fp}(\varphi, \sigma)$ be as above. Then $\rho, i \models \varphi$ iff $\rho, i + 1 \models \text{fp}(\varphi, \rho[i])$.*

Proposition 2 ([23]). *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 $\text{fp}(\varphi, \sigma) \sim_p \text{fp}(\psi, \sigma)$ holds.*

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

Proposition 3 ([23]). *Let φ be an LTL_f formula over \mathcal{P} in NNF, ρ be a finite trace. We have that $\rho \models \varphi$ iff $\epsilon \models \text{fp}(\varphi, \rho)$.*

We take the definition of the *remove-next* function RMNEXT from [23], defined over propositionalized LTL_f formulas in XNF , φ^p :

- $\text{RMNEXT}(\diamond \text{true}) = tt$, $\text{RMNEXT}(\square \text{false}) = ff$
- $\text{RMNEXT}(\varphi_1 \wedge \varphi_2) = \text{RMNEXT}(\varphi_1) \wedge \text{RMNEXT}(\varphi_2)$
- $\text{RMNEXT}(\varphi_1 \vee \varphi_2) = \text{RMNEXT}(\varphi_1) \vee \text{RMNEXT}(\varphi_2)$
- $\text{RMNEXT}(\bigcirc\varphi) = \varphi \wedge \diamond \text{true}$, $\text{RMNEXT}(\bullet\varphi) = \varphi \vee \square \text{false}$.

Note that RMNEXT applies to neither \mathcal{U} -, \mathcal{R} - formulas, since they do not appear in XNF , nor literals (p , $\neg p$), as the input of the function is a propositionalized LTL_f formula in XNF form. We have the following proposition:

Proposition 4 ([23]). *Given an LTL_f formula φ in NNF, $\forall \sigma \in 2^{\mathcal{P}}$, $\text{fp}(\varphi, \sigma) \equiv \text{RMNEXT}(\text{XNF}(\varphi)^p|_{\sigma})$, where $\text{XNF}(\varphi)^p|_{\sigma}$ stands for substituting in $\text{XNF}(\varphi)^p$ the variable p with \top if $p \in \sigma$ and \perp if $p \in \mathcal{P} \setminus \sigma$.*

LTL_f Synthesis. Let φ be an LTL_f formula over $\mathcal{P} = \mathcal{X} \cup \mathcal{Y}$, and \mathcal{X}, \mathcal{Y} are two disjoint sets of propositional variables controlled by the *environment* and the *agent*, respectively. The *synthesis* problem $(\varphi, \mathcal{X}, \mathcal{Y})$ consists in computing a strategy $g : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$, such that for an arbitrary infinite sequence $\lambda = X_0, X_1, \dots \in (2^{\mathcal{X}})^{\omega}$, we can find $k \geq 0$ such that $\rho^k \models \varphi$, where $\rho^k = (X_0 \cup g(\epsilon), (X_1 \cup g(X_0)), \dots, (X_k \cup g(X_0, X_1, \dots, X_{k-1})))$. 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) [7]. 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.

Forward LTL_f Synthesis. Two recent papers [22, 23] proposed a *forward* approach to solve the problem of LTL_f synthesis. Their implementations are called, respectively, LtlfSyn and Cynthia. The idea is to build the DFA on-the-fly, while performing an adversarial forward search towards the final states, by considering the DFA as a sort of AND-OR graph [30]. Therefore, a winning strategy might be found before constructing the whole DFA. The state-of-the-art forward technique [23], implemented in the tool Cynthia, is described by the pseudocode in Algorithm 1. The algorithm is basically a top-down, depth-first traversal of the AND-OR graph induced by the on-the-fly DFA construction, proceeding forward from the initial state and excluding strategies that lead to loops. The forward-based generation of the AND-OR graph is based on formula progression and on the abstract functions `GETORARCS` and `GETANDARCS` functions (Line 20 and Line 21, respectively) that, taken in input a search node n , it produces the next available actions and successor states. The presence of loops must be carefully handled; when a loop is detected at node n , the procedure returns false, temporarily considering n as a failure node. Note that node n is not tagged as failure, since it is unknown whether all the or-arcs of n are explored. If later during the search n is discovered as a success node, such information must be propagated from n backwards to the ancestor nodes of n . 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, implemented in `BACKPROP` (Line 28), as illustrated in [31]. The main novelty of Cynthia over LtlfSyn is that the `GETORARCS` and `GETANDARCS` are based on a compilation of the current state formula in a Sentential Decision Diagram [24], which was used both for fast state-equivalence checking and for possibly avoiding the exponential enumeration of players' moves. Indeed, the experimental evaluation of their technique is rather impressive, as its implementation Cynthia, outperformed other state-of-the-art tools on challenging benchmarks, e.g. on the Nim benchmark [32]. For more details on the search algorithm, please refer to the original paper [23].

Algorithm 1 Forward LTL_f Synthesis [23]

```

1: function SYNTHESIS( $\varphi$ ) return strategy
2:   if ISACCEPTING( $\varphi$ ) then
3:     ADDTOSTRATEGY( $\varphi$ , true)
4:     return GETSTRATEGY()
5:   INITIALGRAPH( $\varphi$ )
6:    $n :=$  GETGRAPHROOT()
7:   found := SEARCH( $n$ ,  $\emptyset$ )
8:   if found then return GETSTRATEGY()
9:   return EMPTYSTRATEGY()  $\triangleright \varphi$  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  $\triangleright$  We found a loop
14:     TAGLOOP( $n$ ) return False
15:    $\psi :=$  FORMULAOFNODE( $n$ )
16:   if ISACCEPTING( $\psi$ ) then
17:     TAGSUCCESSNODE( $n$ )
18:     ADDTOSTRATEGY( $\psi$ , true)
19:     return True
20:   for ( $act$ ,  $AndNd$ )  $\in$  GETORARCS( $n$ ) do
21:     for ( $resp$ ,  $succ$ )  $\in$  GETANDARCS( $AndNd$ ) do
22:       found := SEARCH( $succ$ , [path| $n$ ])
23:       if  $\neg$ found then Break
24:       if found then
25:         TAGSUCCESSNODE( $n$ )
26:         ADDTOSTRATEGY( $\psi$ ,  $act$ )
27:         if ISTAGLOOP( $n$ ) then
28:           BACKPROP( $n$ )
29:       return True
30:   TAGFAILURENODE( $n$ )
31:   return False

```

3. DPLL-based Forward LTL_f Synthesis

Our aim is to propose a new approach that tries to overcome the above limitations that we consider crucial for a scalable approach. In the first place, we observe that Algorithm 1 can be seen as an abstract specification that depends on two crucial subprocedures: (i) **state-equivalence checking**, denoted with `EQUIVALENCECHECK(n_1, n_2)`, that checks whether the

search nodes n_1 and n_2 can be considered equivalent w.r.t. the current AND-OR search graph; and (ii) **search node expansion**, denoted with `GETANDARCS(n)` (resp. `GETORARCS(n)`), that returns an *iterator* of AND-arcs (resp. OR-arcs) of the AND-node (resp. OR-node) starting from node n . The former is implicitly used, e.g. in the `INPATH` function, while the latter functions are employed at Line 20 and Line 21. Note that this separation was not clearly stated in [23] and [22]; in particular, De Giacomo et al. referred to an `EXPAND(n)` function that *both* computes successors node of n and finds a representation for such successors to be used for state equivalence checking. The high-level search algorithm being used is not different from the De Giacomo et al.’s one. However, this modular separation allows us to focus on each core component separately, possibly giving a computational advantage in the computation of the solution; later we will show that this is indeed the case. While in this paper we consider the same search algorithm of theirs (i.e. a standard depth-first AND-OR search), these arguments apply also to other AND-OR search algorithms, e.g. `AO*` [30]. Sometimes we will only refer to `GETARCS` if the procedure is similar both for `GETORARCS` and `GETANDARCS` (although, in general, they might differ).

The crucial observation is that `GETARCS(n)` does not require that the arcs of search node n have already been computed or, in other words, that the node n has been fully expanded (as done by `EXPAND` function). As per specification, `GETARCS(n)` is an iterator over the available moves from n . The concept of iterator is well-known in the computer science community as a way to decouple algorithms from containers [33]. More interestingly, a special case of iterators, *generators* [34], would allow to compute the next players’ moves iteratively “on-demand”, therefore allowing a depth-first search algorithm to visit the next arc returned by the generator even if all arcs have not been computed yet. We will use a generator-based implementation of `DPLLGETARCS` in Algorithm 2.

In fact, De Giacomo et al.’s approach can be seen as a special case of the proposed framework, in which both `EQUIVALENCECHECK` and `GETARCS` are implemented using SDDs: two search nodes are equivalent if they point to the same SDD node, and `GETARCS` is an iterator that simply scans the children of the root SDD node of n . However, this framework can easily overcome the limiting factors mentioned earlier, namely: (i) computed moves do not have to be disjoint and covering (i.e. different moves that lead to the same successor are allowed, although preferably avoided); (ii) if `GETARCS` is implemented using a generator-like approach, the visit of a child node can happen far before the computation of all the available moves; and (iii) the two main search subtasks, state-equivalence checking and a search node expansion, are implemented by two potentially decoupled functions (`EQUIVALENCECHECK` and `GETARCS`, respectively).

We exploit the above observations to design our new `LTLf` synthesis algorithm. At the core of it, there is a novel search node expansion procedure, `DPLLGETARCS` (Algorithm 2), inspired by the `DPLL` algorithm [35, 36]. Such a procedure is somehow in the middle between full enumeration (as `Ltlfsyn`) and optimal compilation (as `Cynthia`) of players’ moves. Unlike `Cynthia`, it selects an assignment of the relevant variables for the current state formula, hence returning a successor node without requiring the full SDD compilation, which in some cases might be prohibitive. However, unlike `Ltlfsyn`, the recursive nature of the procedure gives room to propositional formula simplification, which might drastically reduce the size of the action space (i.e. a smaller number of branching variables to be considered). As state equivalence checking procedures, we consider (i) `BDDCHECK`, which is based on Binary Decision Diagrams (BDD) [37], and

(ii) `SYNTACTICEQCHECK`, based on syntactic equivalence. Intuitively, the `BDDEQCHECK` compiles the state formula as Cynthia did using SDDs, but without requiring the variable decomposition for finding the next moves. The `SYNTACTICEQCHECK` just compares the formulas structurally, which can be implemented very efficiently. A complete formalization of these components, as well as proof of correctness of the constructions, and how these are combined in a unified procedure, will be extensively discussed in the following sections.

4. BDD-based and Syntactic Equivalence Checks

In this section, we describe two equivalence checks that can be used for forward LTL_f synthesis. The first one is a knowledge-compilation-based equivalence check, that uses BDDs to compile the state formula and achieve constant (propositional) equivalence checking. The second one is a simple and lightweight equivalence check that has never been used in this context and, as we shall see, turns out to be very useful in the experimental evaluation. However, we discuss implications regarding the completeness of the resulting synthesis procedure.

BDD-based EQUIVALENCECHECK. The BDD-based equivalence check is similar to the SDD-based equivalence check performed by Cynthia. That is, for a search node n , we take its associated LTL_f formula ψ with `FORMULAOFNODE` (remember that search node is associated with an LTL_f formula). Then, we compute $\text{xf}(\psi)$, which is propositionally equivalent to ψ . $\text{xf}(\psi)$, by construction, is defined over the set of variables $\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}$, where $\mathcal{Z} = \bigcup_{\theta \in \text{cl}(\varphi)} \{z_\alpha \mid \alpha \in \text{pa}(\text{xf}(\theta)), \alpha \text{ not literal}\}$. Finally, we get its BDD representation, i.e. $B_\psi := \text{BDDREPRESENTATION}(\text{xf}(\psi)^p)$. We do these operations both for search nodes n_1 and n_2 , whose state formulas are ψ_1 and ψ_2 respectively, yielding $B_{\text{xf}(\psi_1)}$ and $B_{\text{xf}(\psi_2)}$. The equivalence check whether the two BDDs point to the same BDD node ($B_{\text{xf}(\psi_1)} = B_{\text{xf}(\psi_2)}$). If that is the case then it means, thanks to the canonicity property of BDDs, that the associated (propositionalized) formulas are propositionally equivalent.

Lemma 1. *Let $(\varphi, \mathcal{X}, \mathcal{Y})$ be a LTL_f synthesis problem instance. The `BDDEQCHECK` procedure for such instance induces a search space for Algorithm 1 with no more than $2^{2^{|\mathcal{O}(\text{cl}(\varphi))|}}$ search nodes.*

Proof. Any LTL_f formula ψ associated to some search node n of Algorithm 1 is such that $\text{xf}(\psi)^p \in \mathcal{B}^+(\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z})$. Since there are at most $2^{|\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}|}$ models, thanks to the canonicity property of BDDs, there can be at most $2^{2^{|\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z}|}}$ propositionally equivalent formulas. Since $\mathcal{Y} \cup \mathcal{X} \cup \mathcal{Z} = \mathcal{O}(\text{cl}(\varphi))$, we get the claim. \square

We preferred the use of BDDs instead of SDDs since we do not need the decomposing feature of SDDs, and also because robust and optimized implementations for BDDs already exists, e.g. CUDD [38], with useful features such as dynamic variable reordering.

Syntactic EQUIVALENCECHECK. We now consider an equivalence check procedure that is based on *structural equivalence*: two search nodes n_1 and n_2 are considered equivalent if their formulas ψ_1 and ψ_2 have the same syntax tree, i.e.: `SYNTACTICEQCHECK`(n_1, n_2) := `FORMULAOFNODE`(n_1) = `FORMULAOFNODE`(n_2). To make the comparison fast, we can use *hash consing* [39] which is a technique used to share values that are structurally equal. Using hash consing, two formulas can be stated as structurally equivalent if they point to the same

memory address, achieving constant time equality check. Unfortunately, naïvely applying formula progression might give false negatives, as shown by the following result:

Theorem 2. *Algorithm 1 with SYNTACTICEQCHECK for EQUIVALENCECHECK is sound but not complete for LTL_f synthesis.*

Proof. Soundness follows from the soundness of hash-consing based equivalence check. To disprove completeness, consider the synthesis problem with $\varphi = \Box a \mathcal{U} \Diamond b$, $\mathcal{Y} = \{a\}$ and $\mathcal{X} = \{b\}$. Let $\sigma = \{a\}$, $\varphi_0 = \varphi$ and $\varphi_n = \text{fp}(\varphi_{n-1}, \sigma)$. It can be shown by induction on n the following statement: for all $n \geq 1$, we have $\text{xfn}(\varphi_n) = (((b \wedge \Diamond \text{true}) \vee \bigcirc \Diamond b) \wedge \Diamond \text{true}) \vee (\text{xfn}(\varphi_{n-1}) \wedge (((a \vee \Box \text{false}) \wedge \bullet \Box a) \vee \Box \text{false}) \wedge \Diamond \text{true})$. By correctness of fp , the set of formulas $\varphi_0, \varphi_1, \dots$ are semantically equivalent but structurally different, ending up in a infinite loop, which is undetected by the SYNTACTICEQCHECK. \square

At the core of the issue is that, by how the formula progression works, there are some cases in which a new structurally different formula can be always produced by some particular sequence of applications of formula progression rules, although propositionally equivalent formulas have been already produced earlier during the search. Nevertheless, such equivalence check is very computationally cheap and, as we shall see in the experimental section, often it performs better than the BDD-based equivalence check.

To guarantee the termination of this version of the search algorithm, we propose the following procedure: given a synthesis problem, first execute Algorithm 1 with SYNTACTICEQCHECK as equivalence check and some search node expansion procedure. As soon as, during the execution, the size of the formula of any generated search node becomes greater than a given threshold t , then abort the execution and resort to the search algorithm by using BDDCHECK as equivalence check. In other words, if the problem does not present the pathological corner case shown in the proof of Theorem 2, then try to solve it, without getting stuck with onerous BDD-based compilations.

Lemma 2. *Algorithm 1 with SYNTACTICEQCHECK for EQUIVALENCECHECK with size formula threshold t always terminates and is correct.*

Proof. Correctness follows from [23, Theorem 5], whereas termination follows from considering that (i) the number of distinct state formulas of size at most t is finite, and (ii) in case the threshold is hit, by the correctness of the BDD-based equivalence check (Lemma 1). \square

Lemma 2 says that the threshold guarantees that only a finite number of structurally equivalent formulas can be computed. Empirically, we found that a good threshold that suitably postpones the detection of pathological instances is three times the size of the initial formula: $t = 3 \cdot |\varphi|$.

5. DPLL-based Search Node Expansion

In this section, we describe our main novel approach for search node expansion, that we argue is the key ingredient in achieving state-of-the-art performances for forward LTL_f synthesis, that allows to overcome some limitations of previous works discussed in previous sections. In particular, GETARCS is implemented using a DPLL-based procedure (Algorithm 2). We claim

the DPLL-based GETARCS to be effective for solving LTL_f synthesis in a forward fashion, as also shown by the experimental evidence.

Algorithm 2 DPLL-based GETARCS

```

1: function DPLLGETARCS( $n$ ) return  $Gen[move, node]$ 
2:    $\psi \leftarrow \text{XNF}(\text{FORMULAOFNODE}(n))$ 
3:    $ass \leftarrow \{\}$   $\triangleright$  propositional assignment
4:   if ISORNODE( $n$ ) then
5:     yield from DPLLGETORARCS( $\psi^p, ass$ )
6:   else
7:     yield from DPLLGETANDARCS( $\psi^p, ass$ )
8:   function DPLLGETORARCS( $\phi, ass$ )
9:      $\mathcal{Y}' \leftarrow \text{GETAGENTVARS}(\phi)$ 
10:    if  $\mathcal{Y}' \neq \emptyset$  then
11:       $\ell \leftarrow \text{GETBRANCHINGLITERAL}(\phi)$ 
12:       $\phi_\ell \leftarrow \text{REPLACE}(\phi, \ell)$ 
13:      yield from DPLLGETORARCS( $\phi_\ell, ass \cup \{\ell\}$ )
14:       $\phi_{\neg\ell} \leftarrow \text{REPLACE}(\phi, \neg\ell)$ 
15:      yield from DPLLGETORARCS( $\phi_{\neg\ell}, ass \cup \{\neg\ell\}$ )
16:    else  $\triangleright$  No branching on agent variables available
17:      yield ( $ass, \phi^{\text{tf}}$ )  $\triangleright \phi^{\text{tf}}$  is the next AND node
18:   function DPLLGETANDARCS( $\Psi, ass$ )
19:      $\mathcal{X}' \leftarrow \text{GETENVVARS}(\Psi)$ 
20:     if  $\mathcal{X}' \neq \emptyset$  then
21:        $\ell \leftarrow \text{GETBRANCHINGLITERAL}(\Psi)$ 
22:        $\Psi_\ell \leftarrow \text{REPLACE}(\Psi, \ell)$ 
23:       yield from DPLLGETANDARCS( $\Psi_\ell, ass \cup \ell$ )
24:        $\Psi_{\neg\ell} \leftarrow \text{REPLACE}(\Psi, \neg\ell)$ 
25:       yield from DPLLGETANDARCS( $\Psi_{\neg\ell}, ass \cup \neg\ell$ )
26:     else  $\triangleright$  No branching on environment variables available
27:        $\psi' \leftarrow \text{RMNEXT}(\Psi)$ 
28:       yield ( $ass, \psi'$ )  $\triangleright \psi'$  is the next OR node

```

variables to branch on. Both the computed set of assignments resulting from the sequence of recursive calls, ass (initialized at Line 3), and what remains of the formula $\phi = \text{XNF}(\text{FORMULAOFNODE}(n))^p$ after the chosen literals have been replaced with their assigned truth value, are *yielded* such that they can be consumed by the caller function (see Line 17 and 28; the instruction **yield** allows a generator to provide a value to the caller).

Given a search node n , DPLLGETARCS returns a generator over pairs (move, node), where move is a mapping from variables to truth values (the absence of a variable is considered as *don't care*), and node is a LTL_f formula that, as required by ours and De Giacomo et al.'s search framework, represents a search node (either AND or OR). Depending on whether n is an OR-node or an AND-node, the DPLLGETORARCS function (Line 5) or the DPLLGETANDARCS function (Line 7) is called, respectively. The DPLLGETORARCS function takes in input a propositionalization of ψ , i.e. $\phi := \psi^p$, and the current variables' assignment ass . If there is still some agent variable in \mathcal{Y} to assign (Line 9), then we decide the next branching literal ℓ (by calling the function GETBRANCHINGLITERAL, Line 11), we substitute its truth value to the formula ϕ , and simplify it by calling the function REPLACE (Line 12), obtaining ϕ_ℓ . Then, we do the recursive call to DPLLGETORARCS with the new propositionalized formula ϕ_ℓ and updated assignment $ass \cup \{\ell\}$, and start generating the next moves with a fixed value for literal ℓ . Intuitively, this step represents a

The DPLL algorithm is a very famous algorithm for deciding the satisfiability of proposition logic formulas in conjunctive normal form (CNF). Many variants of it have been proposed that work for general non-clausal formulas [40, 41], motivated by the fact that, quite often, conversion of a boolean formula to CNF is both unnecessary and undesirable, e.g. because of loss of structural information and due to the worst-case exponential blow-up of the size of the formula. We agree with this view, and in the following we assume to deal with propositionalized LTL_f formulas in non-clausal form.

We are interested in designing a DPLL-like procedure to identify the next moves and successor nodes from a search node n . Our proposed procedure (Algorithm 2), like any DPLL procedure, runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively applying the same procedure to the simplified formula, until there are no agent or environment

transition to another node of the search tree of a DPLL algorithm. The instruction **yield from** allows a generator to forward the generation of results to another generating function. When the generation terminates, the negated literal $\neg\ell$ is replaced to the original formula ϕ , yielding another propositionalized LTL_f formula $\phi_{\neg\ell}$, and the available moves starting from this branch are generated. Intuitively, the last step represents the exploration of the opposite branch of the current node of the DPLL search tree, with the branching literal ℓ set at the opposite truth value $\neg\ell$. Note that in the base case, we return the pair (ass, ϕ^{tf}) , where ass contains all the chosen literals in the current final assignment, and ϕ^{tf} is the LTL_f formula that represents the next AND node. The `DPLLGETANDARCS` is analogous to `DPLLGETORARCS` but for AND nodes; therefore, it aims at finding an assignment of environment variables \mathcal{X} rather than of agent variables \mathcal{Y} . Another difference with `DPLLGETANDARCS` is that in the base case, we use the propositional formula Ψ (the result of the substitutions of chosen literals and the subsequent simplifications) to compute the next search node formula ψ' , using the function `RmNEXT`, at Line 27. Note that, at this stage, Ψ is a propositional formula over \mathcal{Z} state variables only. By Proposition 4, since $\Psi = \text{xf}(\psi)^p|_{\sigma}^{\mathcal{P}}$, we have that $\psi' = \text{RmNEXT}(\Psi) = \text{fp}(\psi, \sigma)$, i.e. the correct next state.

According to the needs of the search algorithm, the procedure can be run exhaustively, i.e. until all available moves from node n have been produced. Still, the simplification step can possibly avoid a large part of the naive search space over \mathcal{Y} and \mathcal{X} ; this is an improvement wrt the `Ltlfsyn` approach, which blindly enumerates all possible assignments. The simplification step recursively applies the usual propositional simplification rules, e.g. considering the absorbing or neutral boolean values of binary operators. We suggest to simplify the propositional formula to a great extent, but without resorting to any compilations. Instead, we leave the formula in non-clausal form, aiming at eliminating branching variables from the resulting formula. Such variables will be considered as *don't care* in the current assignment.

We argue that such kind of procedures, like the one described in Algorithm 2, are suitable for our use-case because of their depth-first nature, which implies a low-space requirement, and because of their “responsive” nature: a candidate move is proposed in linear time in the number of variables (possibly better thanks to simplifications). It is interesting to observe that the full trace of a DPLL execution can be seen as a compilation of the propositional theory [42]. Note that Algorithm 2 is an abstract specification that can be customized by different realizations of `GETBRANCHINGLITERAL` and `REPLACE`.

Lemma 3. *Let $(\varphi, \mathcal{X}, \mathcal{Y})$ be a LTL_f synthesis problem instance. `DPLLGETARCS` correctly expands the search graph.*

Proof sketch. By construction, both `DPLLGETORARCS` and `DPLLGETANDARCS` are complete because they consider all possible agent’s and environment’s variable assignments (possibly avoiding exhaustive enumeration thanks to `REPLACE`’s simplifications). Moreover, when `DPLLGETORARCS`(ψ) reaches its base case, by definition of AND-OR graph of φ , (ass, ϕ^{tf}) is a valid transition from ψ to AND node ϕ^{tf} , while `DPLLGETORARCS`(ϕ^{tf}) returns (ass, ψ') , where $\psi' = \text{RmNEXT}(\text{xf}(\psi)^p|_{ass}^{\mathcal{P}})$ is the correct successor node by Proposition 4. \square

Theorem 3. *Algorithm 1 with `BDDCHECK` for state-equivalence checking and Algorithm 2 for search node expansion is correct and always terminates.*

Proof. Termination follows from Lemma 1 and Thm. 4 of [23]. Correctness follows from Lemma 1, 3, and Thm. 5 of [23]. \square

Theorem 4. *Algorithm 1 with SYNTACTICEQCHECK and formula size threshold t for state-equivalence checking and Algorithm 2 for search node expansion is correct and always terminates.*

Proof. Termination follows from Lemma 2 and Thm. 4 of [23]. Correctness follows from Lemma 2, 3, and Thm. 5 of [23]. \square

6. Implementation and Experiments

We implemented the presented synthesis methods in a tool called Nike, which resulted the winner in the LTL_f Realizability Track of SYNTCOMP 2023. Nike is an open-source tool implemented in C++11. It uses Syfco to parse the synthesis problems described in TLSF format [43] to obtain the LTL_f specification and the partition of agent/environment propositions. Nike integrates the preprocessing techniques presented in [22] to perform one-step realizability/unrealizability checks, which is implemented using the BDD library CUDD [38], at the beginning of the synthesis procedure. If neither one-step check succeeds, the AND-OR search begins. Since the procedure is correct and terminates, either the search procedure does not find a winning strategy, in which case the answer to the LTL_f synthesis problem is “unrealizable”, or a winning strategy is found, and therefore the outcome is “realizable”. We use n-ary trees with hash-consing for representing the LTL_f formulas and performing the hash-based state-equivalence checking. The CUDD library is used for the BDD-based state-equivalence checking. Nike, as Cynthia and Ltfsyn, applies some optimizations to speed up the synthesis procedure. First, when visiting an OR-node n for the first time, we perform the pre-processing techniques described in [22]. 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. Nike can run in two modes: using BDD-based state-equivalence checking (BDD), and hash-consing-based state-equivalence checking (Hash). In the DPLL-based search node expansion, we considered variables in alphabetical order, and we combined them with three simple branching strategies: *True-first* (TF) that first sets variables to true, *False-first* (FF) that first sets variables to false; and *Random* (Rand) that sets variables at random. This yields six combinations of Nike that we included in these experiments. We also include a parallel version of Nike, Nike-P, that runs in Hash mode all the three branching strategies in parallel.

Experimental Methodology. We evaluated the efficiency of all variants of Nike, by comparing against the following tools: Lisa [9] and Lydia [8] 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 [44]. We excluded Ltfsyn from the comparison since it was already superseded by Cynthia.

Experiment Setup. Experiments were run on a VM instance on Google Cloud, type c2-standard-4, endowed with Intel(R) Xeon(R) CPU 3.10GHz, 4 logical CPU threads, 16 GB of memory and 300 seconds of time limit. The correctness of Nike was empirically verified by comparing the results with those from all baseline tools. No inconsistency was found.

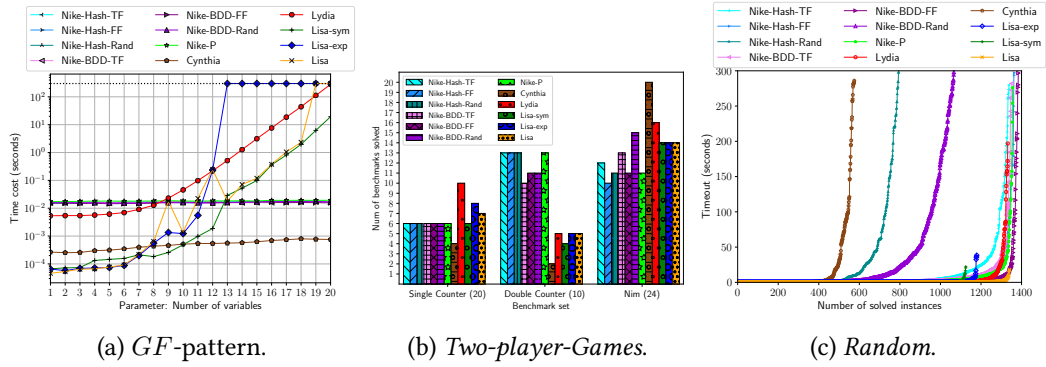


Figure 1: Comparison results on all benchmarks.

Benchmarks. We collected 1494 LTL_f synthesis instances from literature: 20 unrealizable GF -*pattern* and 20 realizable U -*pattern* instances, of the form $GF(n) = \Box(p_1) \wedge \Diamond(q_2) \wedge \dots \wedge \Diamond q_n$ and $U(n) = p_1 \mathcal{U}(p_2 \mathcal{U}(\dots p_{n-1} \mathcal{U} p_n))$, respectively [45, 46]; 54 *Two-player-Games* instances [47, 9]: *Single-Counter*, where the agent stores an n -bit counter (where n is the scaling parameter) which it must increment upon a signal by the environment. The agent wins if the counter eventually overflows to 0; *Double-Counter* is similar to the *Single-Counter* one, except that in this case there are two n -bit counters, one incremented by the environment and another by the agent. The goal of the agent is for its counter to eventually catch up with the environment’s counter; and *Nim* is a generalized version of the game of Nim with n heaps of m tokens each [32]. Finally, we considered 1400 *Random* instances, of which 400 are from [44] and 1000 from [8], constructed from LTL synthesis datasets *Lily* and *Load Balancer* [48].

Analysis. Figure 1a shows the running time of each tool on every instance of the GF -*pattern* dataset. Across these instances, we can observe that all variants of Nike solve instances very quickly, thanks to the pre-processing techniques. This is done with much less time comparing to backward approaches, represented by Lisa and Lydia, simply because these tools do not have such optimizations. Cynthia solved it in less time, but we attribute this to the set up time of the CUDD BDD manager that worsens the performances. Nevertheless, this amounts to a negligible time cost difference of $\ll 1$ second. Results are similar for the U -*pattern* dataset, shown in the supplementary material. On the *Two-player-Games* benchmarks, see Figure 1b, we observe that Nike variants dominate all other tools on the *Double-Counter* instances, while competing with backward approaches on the other instances. On *Nim*, Cynthia is the best performing tool, but on the other benchmarks Nike shows to be better. The Nike-BDD combinations performs slightly worse on *Double Counter* than the Nike-Hash combinations. On the *Random* benchmarks, all variants of Nike, except the ones using Rand, are competitive with state-of-the-art backward approaches, and far better than Cynthia.

It is clear from the plots that Nike, in general, shows an overall better performance than Cynthia, illustrating the efficiency and better scalability of our approach. In particular, there is a notable outperformance of Cynthia on the *Double-Counter* and in the *Random* instances. We attribute this to the ability of Nike to not be stuck with compilation processes that can easily become intractable, both on hand-designed datasets like *Double-Counter*, and in randomly

generated intractable cases. On the *Nim* benchmark, our tool does not perform as good as the others, but its performance are still competitive, especially in the variant Nike-BDD-FF. This is because the *Nim* formulas were manageable enough for SDD compilation (Cynthia) and for DFA construction (Lydia/Lisa), whereas the blind branching strategies of Nike were not effective in this case, as most of the time is spent on generating successors that have been already visited. The worse performance of the Rand branching strategy on the *Random* benchmark can be explained by the fact that the TF and the FF strategies might exploit a particular problem structure of these instances, that allow to easily arrive at success nodes or failure nodes, and frees the algorithm to explore more moves thanks to the short-circuit evaluation of the search outcome (see Lines 23 and 24 in Algorithm 1). The best configuration is Nike-BDD-FF, which suggests that for this benchmark the state compilation is not too hard and the canonicity of the representation helps to prevent the revisit of propositionally-equivalent states.

Overall, despite the simplicity of the DPLL-based expansion, performances are very surprising with respect to backward approaches; this suggests that our approach is very promising and worth of future research.

7. Conclusions

We proposed the best forward search LTL_f synthesis approach so far, and the first that is truly competitive with the considered state-of-the-art tools based on backward computation (as in the *Random* benchmark). Our implementation ranked first in the LTL_f Realizability Track of the 2023 edition of SYNTCOMP. We think this work sets the foundations for a new family of LTL_f synthesis algorithms, and opens several research avenues for investigating effective branching heuristics [49] for the DPLL-based search graph expansion, e.g. non-chronological backtracking, better order in which branching variables are chosen [50], or better termination strategies for searching with syntactic state-equivalence checking.

Acknowledgements

This line of research has started from earlier research work supported by the ERC-ADG White-Mech (No. 834228).

References

- [1] R. De Benedictis, N. Gatti, M. Maratea, A. Murano, E. Scala, L. Serafini, I. Serina, E. Tosello, A. Umbrico, M. Vallati, Preface to the Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (IPS-RCRA-SPIRIT 2023), in: Proceedings of the Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (IPS-RCRA-SPIRIT 2023) co-located with

22th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2023), 2023.

- [2] A. Church, Application of recursive arithmetic to the problem of circuit synthesis, *Journal of Symbolic Logic* 28 (1963).
- [3] A. Pnueli, The temporal logic of programs, in: *FOCS*, 1977.
- [4] A. Pnueli, R. Rosner, On the Synthesis of a Reactive Module, in: *POPL*, 1989.
- [5] C. Baier, J. Katoen, *Principles of model checking*, 2008.
- [6] G. De Giacomo, M. Y. Vardi, Linear Temporal Logic and Linear Dynamic Logic on Finite Traces, in: *IJCAI*, 2013.
- [7] G. De Giacomo, M. Y. Vardi, Synthesis for LTL and LDL on Finite Traces, in: *IJCAI*, 2015.
- [8] G. De Giacomo, M. Favorito, Compositional approach to translate LTL_f/LDL_f into deterministic finite automata, in: *ICAPS*, 2021.
- [9] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications, in: *AAAI*, 2020.
- [10] M. Ghallab, D. S. Nau, P. Traverso, *Automated planning - theory and practice*, 2004.
- [11] H. Geffner, B. Bonet, *A Concise Introduction to Models and Methods for Automated Planning*, 2013.
- [12] A. Cimatti, M. Roveri, P. Traverso, Strong planning in non-deterministic domains via model checking, in: *AIPS*, 1998.
- [13] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking. 1–2 (2003).
- [14] J. H. Reif, The complexity of two-player games of incomplete information, *JCSS* 29 (1984).
- [15] R. P. Goldman, M. S. Boddy, Expressive planning and explicit knowledge, in: *AIPS*, 1996.
- [16] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Strong planning under partial observability, *Artif. Intell.* 170 (2006).
- [17] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise, *An Introduction to the Planning Domain Definition Language*, 2019.
- [18] J. Rintanen, Complexity of planning with partial observability, in: *ICAPS*, 2004.
- [19] A. Camacho, J. A. Baier, C. J. Muise, S. A. McIlraith, Finite LTL Synthesis as Planning, in: *ICAPS*, 2018.
- [20] A. Camacho, S. A. McIlraith, Strong fully observable non-deterministic planning with LTL and LTL_f goals, in: *IJCAI*, 2019.
- [21] S. Thanh To, E. Pontelli, T. Cao Son, A conformant planner with explicit disjunctive representation of belief states, in: *ICAPS*, 2009.
- [22] S. Xiao, J. Li, S. Zhu, Y. Shi, G. Pu, M. Y. Vardi, On-the-fly synthesis for LTL over finite traces, in: *AAAI*, 2021.
- [23] G. De Giacomo, M. Favorito, J. Li, M. Y. Vardi, S. Xiao, S. Zhu, Ltlf synthesis as AND-OR graph search: Knowledge compilation at work, in: *IJCAI*, 2022.
- [24] A. Darwiche, SDD: A new canonical representation of propositional knowledge bases, in: *IJCAI*, 2011.
- [25] M. Favorito, Forward ltlf synthesis: Dpll at work, 2023. [arXiv:2302.13825](https://arxiv.org/abs/2302.13825).
- [26] R. I. Brafman, G. De Giacomo, F. Patrizi, LTL_f/LDL_f non-markovian rewards, in: *AAAI*, 2018.
- [27] J. Li, K. Y. Rozier, G. Pu, Y. Zhang, M. Y. Vardi, Sat-based explicit LTL_f satisfiability

- checking, in: AAAI, 2019.
- [28] E. A. Emerson, Temporal and modal logic, in: Handbook of Theoretical Computer Science, 1990.
 - [29] F. Bacchus, F. Kabanza, Planning for temporally extended goals, *Ann. Math. Artif. Intell.* 22 (1998).
 - [30] N. J. Nilsson, Problem-solving methods in artificial intelligence, 1971.
 - [31] M. G. Scutellà, A note on dowling and gallier’s top-down algorithm for propositional horn satisfiability, *J. Log. Program.* 8 (1990) 265–273.
 - [32] C. L. Bouton, Nim, a game with a complete mathematical theory, *Annals of Mathematics* 3 (1901).
 - [33] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Deutschland GmbH, 1995.
 - [34] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, Iteration abstraction in sather, *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18 (1996) 1–15.
 - [35] M. Davis, H. Putnam, A computing procedure for quantification theory, *J. ACM* 7 (1960).
 - [36] M. Davis, G. Logemann, D. W. Loveland, A machine program for theorem-proving, *Commun. ACM* 5 (1962).
 - [37] R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, *ACM Comput. Surv.* 24 (1992).
 - [38] F. Somenzi, CUDD: CU Decision Diagram Package. Univ. of Colorado at Boulder (2016).
 - [39] L. P. Deutsch, An interactive program verifier (1973).
 - [40] C. Thiffault, F. Bacchus, T. Walsh, Solving non-clausal formulas with DPLL search, in: SAT, 2004.
 - [41] H. Jain, E. M. Clarke, Efficient SAT solving for non-clausal formulas using dpLL, graphs, and watched cuts, in: DAC, ACM, 2009, pp. 563–568.
 - [42] J. Huang, A. Darwiche, The language of search, *J. Artif. Intell. Res.* 29 (2007) 191–219.
 - [43] S. Jacobs, G. A. Perez, P. Schlehuber-Caissier, The temporal logic synthesis format tLsf v1.2, 2023. [arXiv:2303.03839](https://arxiv.org/abs/2303.03839).
 - [44] S. Zhu, L. M. Tabajara, J. Li, G. Pu, M. Y. Vardi, A Symbolic Approach to Safety LTL Synthesis, in: HVC, 2017.
 - [45] K. Y. Rozier, M. Y. Vardi, LTL satisfiability checking, in: SPIN, volume 4595 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 149–167.
 - [46] J. Geldenhuys, H. Hansen, Larger automata and less work for LTL model checking, in: SPIN, volume 3925 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 53–70.
 - [47] L. M. Tabajara, M. Y. Vardi, Partitioning Techniques in LTL_f Synthesis, in: IJCAI, 2019.
 - [48] R. Ehlers, Unbeast: Symbolic bounded synthesis, in: TACAS, volume 6605 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 272–275.
 - [49] J. P. M. Silva, The impact of branching heuristics in propositional satisfiability algorithms, in: EPIA, volume 1695 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 62–74.
 - [50] P. Liberatore, On the complexity of choosing the branching literal in dpLL, *Artificial intelligence* 116 (2000) 315–326.