Forward LTL$_f$ Synthesis: DPLL At Work

Marco Favorito[1]

[1]Banca d'Italia, Italy

Abstract

We present a forward-search-based approach for specifications expressed in Linear Temporal Logic on finite traces (LTL$_f$). We exploit the observation that the DFA game arena coming from the LTL$_f$ can be seen as an AND-OR graph. The idea is that for many problem instances the solution can be found without computing the whole game arena, as done by the classical backward LTL$_f$ synthesis approach. The procedure, implemented in the tool Nike, is a depth-first AND-OR graph search based on two primitives: *state-equivalence checking* and *search node expansion*. State-equivalence checking is based on syntactic equivalence and knowledge compilation techniques, whereas search node expansion is based on a procedure inspired by the famous Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Nike won the LTL$_f$ Realizability Track in the 2023 edition of SYNTCOMP.

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 [1]. A commonly used logic for program synthesis is Linear Temporal Logic (LTL) [2, 3], typically used also in model checking [4]. LTL *on finite traces* (LTL$_f$) [5], a variant of LTL to specify *finite*-horizon temporal properties, has been recently proposed as specification language for temporal synthesis [6]. 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. 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. State-of-the-art tools such as Lydia [7] and Lisa [8] 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,

OVERLAY 2023: 5th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, November 7, 2023, Rome, Italy

marco.favorito@bancaditalia.it (M. Favorito)
https://marcofavorito.me (M. Favorito)
0000-0001-9566-3576 (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)

CEUR-WS.org/Vol-3629/paper11.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

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) [9, 10, 11, 12], 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* [13, 14, 15]. However, techniques developed for such problems cannot be applied to ours directly, which may result in a PDDL specification with exponential size e.g. see [16, 17].

For these reasons, researchers have been looking into forward search techniques specifically conceived for solving LTL$_f$ synthesis, considering the DFA game as an AND-OR graph search. Two notable attempts in this direction have been presented in [18], proposing the tool Ltlfsyn, and [19] with the tool Cynthia. Our work builds on top of them, by improving certain design and implementation details, especially regarding the state-equivalence checking and the search node expansion components, together with a Binary Decision Diagram (BDD)-based check to achieve completeness when certain conditions are met. Nike uses a computationally-cheap syntactical equivalence between state formulas, not used in previous works. Furthermore, a novel search graph expansion technique is proposed, based on a procedure inspired by the famous Davis-Putnam-Logemann-Loveland (DPLL) algorithm. As in [18, 19], the problem is then reduced to an AND-OR graph search, where the OR nodes represent the agent's choices, and the AND nodes represent the environment's choices. The search algorithm used is a classical depth-first AND-OR graph search algorithm. The LTL$_f$ synthesis problem is realizable iff there is a winning strategy for the corresponding AND-OR graph. More details can befound in [20].

2. DPLL-based Forward LTL$_f$ Synthesis

The Search Algorithm. Algorithm 1 describes the AND-OR search procedure that is at the core of Nike. 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 an abstract GetArcs function that, taken in input a search node n, it produces the next available player moves 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 [21]. For more details on the search algorithm, please refer to the original paper [19]. Overall, Algorithm 1 is very similar to the one used by [19], except that instead of relying on the abstract Expand function (see Line 20 of Algorithm 1 of their paper), our approach relies on two primitive operations: *state-equivalence checking* and *search node*

expansion. The state-equivalence check is used to check state equivalence, and it is implicitly used in functions like InPath to detect loops. The search-node expansion is represented by the generating function GetArcs and does not have to precompute all successors (Cynthia), or slavishly enumerating all possible agent's and env's variables assignments (Ltlfsyn).

Algorithm 1 Forward LTL_f 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:     for (act, AndNd) ∈GetArcs(n) do
21:         for (resp, succ) ∈GetArcs(AndNd) do
22:             found :=Search(succ, [path|n])
23:             if ¬found then Break
24:         if found then
25:             TagSuccessNode(n)
26:             AddToStrategy(ψ, act)
27:             if IsTagLoop(n) then
28:                 BackProp(n)
29:             return True
30:     TagFailureNode(n)
31:     return False
```

State-Equivalence Checks. We now describe two state-equivalence checking approaches: *BDD-based* and *hash-consing based.* The first one (BddBasedEqCheck) is that, 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 $\mathsf{xnf}(\psi)$, which is propositionally equivalent to ψ. Finally, we get its BDD representation, i.e. B_ψ:=BddRepresentation$(\mathsf{xnf}(\psi)^p)$. We do these operations both for n_1 and n_2, yielding $B_{\mathsf{xnf}(\psi_1)}$ and $B_{\mathsf{xnf}(\psi_2)}$. The equivalence check whether the two BDDs point to the same BDD node ($B_{\mathsf{xnf}(\psi_1)}= B_{\mathsf{xnf}(\psi_2)}$). If true, then it means, by the canonicity property of BDDs, that the associated (propositionalized) formulas are propositionally equivalent.

The second check (HashConsingEqCheck) 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.To make the comparison fast, we can use *hash consing* [22] 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. Since this equivalence check is sound but not complete, 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 HashConsingEqCheck as equivalence check and the search node expansion procedure (DpllGetArcs, see below). 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 Algorithm 1 based on BddBasedEqCheck and DpllGetArcs. Currently, we use a threshold $t = 3 \cdot |\varphi|$.

DPLL-based Search Node Expansion. Given a search node n, our expansion node function DpllGetArcs returns a generator over pairs $(\mathsf{move}, \mathsf{node})$, where move is a mapping from variables to truth values (the absence of a variable is considered a *don't care*), and node is a LTL_f formula that, as required by ours and [19] search framework, represents a search node (either AND or OR). Informally, an agent/environment move is found by picking a variable, assigning a boolean value, and replacing the value to the state formula. The resulting formula is processed again until there are no agent/environment variables. The overall assignment is considered as the next move to explore, and the successor state is computed by applying formula

progression rules (See the function RmNext and Proposition 4 of [20]). Note that such kind of procedure is 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 on the number of variables (possibly better thanks to simplifications). Note that DpllGetArcs abstract specification that can be customized by the way variables are chosen and by which value is assigned to them first. In our tool, we consider them in alphabetical order (as future work we aim to provide less naive and more meaningful orderings), and the assignments strategies are three: *True-First* (i.e. the first assigment considered is always **true**), *False-First* (i.e. the first assigment considered is always **false**), and *random* (i.e. the assignment considered is random). Due to lack of space, we underspecified some details in the description of the theory behind the tool, which can be found in [20].

3. Implementation and Evaluation

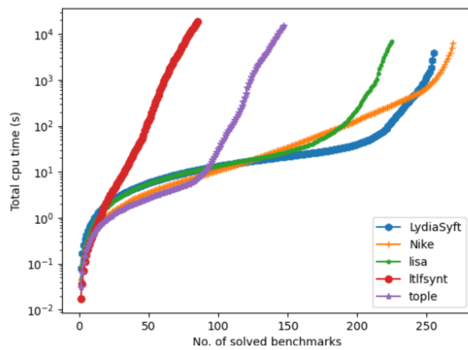

Figure 1: SYNTCOMP23 results for the LTL_f track.

Our prototype implementation, Nike, is an open-source tool implemented in C++11 (github.com/marcofavorito/nike). More specifically, Nike uses Syfco to parse the synthesis problems described in TLSF format [23] to obtain the LTL_f specification and the partition of agent/environment propositions. Nike integrates the preprocessing techniques presented in [18] to perform one-step realizability/unrealizability checks, which is implemented using CUDD (see below), at the beginning of the synthesis procedure. If neither one-step check succeeds, the AND-OR search begins. The search algorithm used by Nike is a recursive depth-first search algorithm, which is detailed below. 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 BDD library CUDD-3.0.0 [24] instead is used for the BDD-based state-equivalence checking. Figure 1 shows the experimental evaluation over benchmarks from the literature [25]. As can be seen, Nike achieved the best performances among other competitive tools for LTL_f synthesis.

4. Conclusion

In this paper we presented the tool Nike, 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. We think this work sets the foundations for a new family of forward LTL_f synthesis algorithms, and opens several research avenues for investigating effective branching heuristics for the DPLL-based search graph expansion (e.g. non-chronological backtracking), or better termination strategies for searching with hash-consing-based 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] A. Church, Application of recursive arithmetic to the problem of circuit synthesis, Journal of Symbolic Logic 28 (1963).
[2] A. Pnueli, The temporal logic of programs, in: FOCS, 1977.
[3] A. Pnueli, R. Rosner, On the Synthesis of a Reactive Module, in: POPL, 1989.
[4] C. Baier, J. Katoen, Principles of model checking, 2008.
[5] G. De Giacomo, M. Y. Vardi, Linear Temporal Logic and Linear Dynamic Logic on Finite Traces, in: IJCAI, 2013.
[6] G. De Giacomo, M. Y. Vardi, Synthesis for LTL and LDL on Finite Traces, in: IJCAI, 2015.
[7] G. De Giacomo, M. Favorito, Compositional approach to translate $\text{LTL}_f/\text{LDL}_f$ into deterministic finite automata, in: ICAPS, 2021.
[8] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications, in: AAAI, 2020.
[9] M. Ghallab, D. S. Nau, P. Traverso, Automated planning - theory and practice, 2004.
[10] H. Geffner, B. Bonet, A Concise Introduction to Models and Methods for Automated Planning, 2013.
[11] A. Cimatti, M. Roveri, P. Traverso, Strong planning in non-deterministic domains via model checking, in: AIPS, 1998.
[12] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking. 1–2 (2003).
[13] J. H. Reif, The complexity of two-player games of incomplete information, JCSS 29 (1984).
[14] R. P. Goldman, M. S. Boddy, Expressive planning and explicit knowledge, in: AIPS, 1996.
[15] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Strong planning under partial observability, Artif. Intell. 170 (2006).
[16] A. Camacho, J. A. Baier, C. J. Muise, S. A. McIlraith, Finite LTL Synthesis as Planning, in: ICAPS, 2018.
[17] A. Camacho, S. A. McIlraith, Strong fully observable non-deterministic planning with LTL and LTL_f goals, in: IJCAI, 2019.
[18] 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.
[19] 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.
[20] M. Favorito, Forward ltlf synthesis: Dpll at work, arXiv preprint arXiv:2302.13825 (2023).
[21] M. G. Scutellà, A note on dowling and gallier's top-down algorithm for propositional horn satisfiability, J. Log. Program. 8 (1990) 265–273.
[22] L. P. Deutsch, An interactive program verifier (1973).

[23] S. Jacobs, G. A. Perez, P. Schlehuber-Caissier, The temporal logic synthesis format tlsf v1.2, 2023. arXiv:2303.03839.
[24] F. Somenzi, CUDD: CU Decision Diagram Package. Univ. of Colorado at Boulder (2016).
[25] G. Perez, SYNTCOMP 2023 Results | The Reactive Synthesis Competition, 2023. URL: http://www.syntcomp.org/syntcomp-2023-results/.