Model-Guided Synthesis for LTL over Finite Traces

Shengping Xiao[1], Yongkang Li[1], Xinyue Huang[1], Yicong Xu[1], Jianwen Li[1⋆], Geguang Pu[1,2], Ofer Strichman[3], and Moshe Y. Vardi[4]

[1] East China Normal University, Shanghai, China
{spxiao,51265902012,52265902016,51215902150}@stu.ecnu.edu.cn,
{jwli,ggpu}@sei.ecnu.edu.cn
[2] Shanghai Trusted Industrial Control Platform Co., Ltd, Shanghai, China
[3] Technion, Haifa, Israel
ofers@technion.ac.il
[4] Rice University, Houston, USA
vardi@cs.rice.edu

Abstract. Satisfiability and synthesis are two fundamental problems for Linear Temporal Logic, both of which can be solved on the automaton constructed from the input formula. In general, satisfiability is easier than synthesis in both theory and practice, as satisfiability needs only to find a satisfying trace, while synthesis has to find a winning strategy. This paper presents a novel technique called MoGuS, which improves the performance of synthesis for LTL_f, a variant of LTL interpreted over finite traces, by repeatedly invoking an LTL_f satisfiability checker to guide its search for a winning strategy. Satiisfiabiity checkers have not been used before in the context of LTL_f synthesis. MoGuS computes a satisfying trace of the input formula, and then uses the formula-progression technique to compute the states on the fly in the automaton run. It then checks whether there exists a winning strategy from each of the states. If not, the current state is marked as a 'failure' state (as it can never produce a winning strategy), the checking rolls back to its predecessor state, and the process repeats. MoGuS returns 'Realizable' if the initial state turns out to be winning, and 'Unrealizable' otherwise. We conducted an extensive experimental evaluation of MoGuS by comparing it to different state-of-the-art LTL_f synthesis algorithms on a large set of benchmarks. The results show that MoGuS has the most stable and the best overall performance on the tested benchmarks.

1 Introduction

Temporal *synthesis* is the automated construction of a reactive system from a given temporal logic formula (specification), e.g., LTL [38], such that the interactive behaviors between the system and the external environment are guaranteed to satisfy the specification [18,39]. The problem of determining whether such a

⋆ Jianwen Li is the corresponding author.

system exists is called *realizability*. LTL realizability and synthesis are major research topics in formal methods, and fruitful works have been accomplished on both the theoretical and practical aspects, e.g., [8,45,36], to name a few. In recent years, an annual synthesis competition [1] has played an important role in motivating tool development in this area. Nevertheless, LTL synthesis is still considered a very challenging problem, as generating deterministic automata from LTL specifications, which is a critical part of the algorithmic solution, involves a doubly-exponential blow-up [2].

An LTL formula is interpreted over infinite traces, so the constructed automaton from the formula has to accept infinite traces as well. Such automata with infinite accepting conditions, e.g., Büchi [11], are notorious for their challenging determinization, e.g., Safra Construction [41], which is a barrier to effective LTL realizability/synthesis. A recent argument, however, has been made that synthesis of the system with finite behaviors is sufficient in practice [20,28].

LTL_f, which is defined over finite traces, has emerged as a popular logic in AI-related domains since its invention [20]. Given an LTL_f formula φ, there exists a non-deterministic finite automaton (NFA) that represents φ's language. Determinization of an NFA can be performed via the classical subset construction [28]. Although the worst-case complexity remains the same (2EXPTIME), this leads to a much simpler synthesis procedure [45], as we will discuss below. Indeed, LTL_f has emerged as a popular temporal description language in AI-related domains, especially for specifying *motion planning* problems [16,14,22,3,4,15,46,27]. LTL_f synthesis is then used to build a model that satisfies these specifications.

Recently, several works have been conducted to study the theory and practice of fundamental problems related to LTL_f, e.g., translation to automata [42,21], satisfiability checking [34,33,35], and synthesis [28,45,43]. The asymptotic complexity of both LTL_f satisfiability and LTL_f synthesis is the same as in LTL, namely PSPACE-complete and 2EXPTIME-complete, respectively [20,28]. The focus of this paper is on using LTL_f satisfiability checking to speed up LTL_f synthesis and realizability.

There are so far two kinds of approaches to solving LTL_f synthesis. The first one is *bottom-up* [45,7], which first constructs the whole (minimal) DFA for the input formula, using the efficient DFA-construction tool MONA [30], and then computes all winning states in the DFA back from the accepting states. Representative LTL_f synthesizers based on this approach include Lisa [7] and Lydia [21]. The second one is *top-down* [43,29], which takes the input formula as the initial state, and then computes the remaining states of the DFA *on-the-fly*, while rolling back once a winning/failure state is identified. Representative LTL_f synthesizers based on this approach include OLFS [43] and Cynthia [29]. Both solutions return 'Realizable' iff the initial state is winning. According to previous studies [43,29], the top-down solution performs better in some particular benchmarks than the bottom-up one, while in general the latter approach gains a better overall performance in the selected benchmarks where synthesis from the formulas is not challenging enough.

The on-the-fly top-down solution has been conducted by using either SAT [24] or Sentential Decision Diagram (SDD) [19] techniques. The former utilizes SAT solvers to compute exactly one (deterministic) state at a time, making the whole framework very flexible. Using SAT solvers for state enumeration is, however, not quite efficient, as the SAT solver cannot distinguish between the system and environment variables. As a result, the satisfying assignment that it finds is rather arbitrary from the perspective of the synthesis process. Meanwhile, the latter solution leverages SDD to encode the variables' information in order, such that the enumerated DFA transitions (with states) can be more compact. As shown in [29], the SDD-based approach is able to outperform the SAT-based one on a considerable number of test instances. Nevertheless, the drawback of this approach is that one SDD computation has to generate all of the one-step successors, which is a much more computationally expensive operation than a single SAT call. So the question is whether there is a way to compute states in a light way, and enumerate the transitions and states in a more 'targeted' way, which will lead to faster convergence.

This paper tries to address this question and proposes to conduct LTL_f synthesis via an algorithm that is based on multiple LTL_f satisfiability checks, hence leveraging the relative efficiency of those tools. We call this approach MoGuS (**M**odel-**G**u**i**ded **S**ynthesis). Instead of computing only one state (and transition), MoGuS utilizes an LTL_f satisfiability solver to generate one satisfying trace at a time, which corresponds to a sequence of states and transitions. The insight is that a satisfying trace is more likely to be compatible with a winning strategy, which can potentially make the state search more precise. MoGuS then progresses backward on the satisfying state sequence, trying each time to prove that the state is winning. If it is not, it asks the LTL_f satisfiability solver for a new assignment that does not go through that state (which we call a 'failure' state), by blocking it. This process is repeated until the initial state becomes winning, or the formula becomes unsatisfiable. In the former case, the formula is declared to be 'Realizable', and in the latter, 'Unrealizable'. The correctness of our procedure relies on the fact that if a satisfying trace runs across a failure state, it cannot be produced by a winning strategy and hence can be blocked.

Generally speaking, MoGuS is a top-down solution for LTL_f synthesis. Compared to the SAT-based approach, it can enumerate states and transitions more precisely, as the search inside is guided by a satisfying trace that is more likely to target a winning state. Compared to the SDD-based approach, MoGuS is more flexible in computing states and transitions, as the state-of-the-art satisfiability solvers, e.g., aaltaf [33], provide a way to compute one state (and transition) at a time.

We implemented MoGuS inside the tool MoGuSer and evaluated its performance by comparing it to the state-of-the-art LTL_f synthesis solvers Cynthia, Lisa, and Lydia on the collected benchmarks from [29] (1454 in total), as well as the *Ascending* benchmarks (1800 in total) generated by Spot [23] for the purpose

of scalability testing[5]. For the collected benchmarks, MoGuSer solves a total of 1287 (out of 1454) instances, which is twice that solved by Cynthia (605) but slightly less than that solved by Lisa (1316) and Lydia (1339). For the *Ascending* benchmarks, MoGuSer solves a total of 1559 (out of 1800) cases, which is better than that solved by all other solvers, i.e., Cynthia (1430), Lisa (1101), Lydia (916). Our tool MoGuSer has the most stable and the best overall performance in these two evaluations.

This paper is organized as follows. The next section introduces preliminaries. Section 3 presents the construction from an LTL_f formula to its corresponding TDFA, by leveraging the formula progression technique. Section 4 describes the details of MoGuS and its correctness guarantee. Section 5 shows the experimental results. And we discuss a brief history of LTL synthesis in Section 6. Finally, Section 7 summarizes the contributions and discusses future work.

2 Preliminaries

2.1 LTL over Finite Traces (LTL_f)

Linear Temporal Logic over finite traces, or LTL_f [20], extends propositional logic with finite-horizon temporal connectives. Generally speaking, LTL_f is a variant of Linear Temporal Logic (LTL) [38] that is interpreted over finite traces. Given a set of atomic propositions $\mathcal{P}$, the syntax of LTL_f is identical to LTL, and defined as:

$$\varphi ::= tt \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where tt represents the *true* formula, $p \in \mathcal{P}$ is an atomic proposition, $\neg$ represents *negation*, $\wedge$ represents *and*, $\bigcirc$ represents the *strong Next* operator and $\mathcal{U}$ represents the *Until* operator. We also have the corresponding dual operators ff (*false*) for tt, $\vee$ (or) for $\wedge$, $\bullet$ (weak Next) for $\bigcirc$ and $\mathcal{R}$ (Release) for $\mathcal{U}$. Moreover, we use the notation $\mathcal{G}\varphi$ (Global) and $\mathcal{F}\varphi$ (Future) to represent $ff\,\mathcal{R}\,\varphi$ and $tt\,\mathcal{U}\,\varphi$, respectively. Notably, $\bigcirc$ is the standard *Next* operator, while $\bullet$ is *weak Next*; $\bigcirc$ requires the existence of a successor instance, while $\bullet$ does not. Thus $\bullet\phi$ is always true in the last instance of a finite trace, since no successor exists there.

A finite *trace* $\rho = \rho[0], \rho[1], \cdots, \rho[n]$ is a sequence of propositional interpretations (sets), in which $\rho[m] \in 2^{\mathcal{P}}$ $(0 \leq m < |\rho|)$ is the m-th interpretation of ρ, and $|\rho| = n + 1$ represents the length of ρ. Intuitively, $\rho[m]$ is interpreted as the set of propositions which are *true* at instance m. We denote ρ^i to represent $\rho[0], \rho[1], \ldots, \rho[i-1]$ $(i \geq 1)$, which is the prefix of ρ to position i (not including i), and ρ_i to represent $\rho[i], \rho[i+1], \cdots, \rho[n]$, which is the suffix of ρ from position i (including i). Two finite traces, ρ_1 and ρ_2, can be concatenated to one trace ρ, denoted by $\rho = \rho_1 \cdot \rho_2$.

[5] From the preliminary evaluations, our previous synthesizer OLFS [43] performs much worse than other tested tools, so it is excluded in the comparison.

LTL_f formulas are interpreted over finite traces. For a finite trace ρ and an LTL_f formula φ, we define the satisfaction relation $\rho \models \varphi$ (i.e., ρ is a model of φ) as follows:

- $\rho \models tt$;
- $\rho \models p$ iff $p \in \rho[0]$, where p is an atomic proposition;
- $\rho \models \neg\varphi$ iff $\rho \not\models \varphi$;
- $\rho \models \varphi_1 \wedge \varphi_2$ iff $\rho \models \varphi_1$ and $\rho \models \varphi_2$;
- $\rho \models \mathsf{O}\varphi$ iff $|\rho| > 1$ and $\rho_1 \models \varphi$;
- $\rho \models \varphi_1 \mathcal{U} \varphi_2$ iff there exists i with $0 \leq i < |\rho|$ such that $\rho_i \models \varphi_2$, and for every j with $0 \leq j < i$ it holds that $\rho_j \models \varphi_1$.

The set of finite traces that satisfy LTL_f formula φ is the language of φ, denoted as $\mathcal{L}(\varphi) = \{\rho \in (2^{\mathcal{P}})^+ \mid \rho \models \varphi\}$. Two LTL_f formulas φ_1 and φ_2 are semantically equivalent, denoted as $\varphi_1 \equiv \varphi_2$, iff for every finite trace ρ, $\rho \models \varphi_1$ iff $\rho \models \varphi_2$. A *literal* is an atom $p \in \mathcal{P}$ or its negation ($\neg p$). We say an LTL_f formula is in *Negation Normal Form* (*NNF*), if the negation operator appears only in front of an atom. Every LTL_f formula can be converted into its *NNF* in linear time. We assume that all LTL_f formulas are in *NNF* in this paper.

2.2 Transition-based DFA

The Transition-based Deterministic Finite Automaton (TDFA) is a variant of the Deterministic Finite Automaton (DFA) [42].

Definition 1 (Transition-based DFA). *A transition-based* DFA *(*TDFA*) is a tuple* $\mathcal{A} = (2^{\mathcal{P}}, S, s_0, \delta, T)$ *where*

- $2^{\mathcal{P}}$ *is the alphabet;*
- S *is the set of states;*
- $s_0 \in S$ *is the initial state;*
- $\delta : S \times 2^{\mathcal{P}} \to S$ *is the transition function;*
- $T \subseteq \delta$ *is the set of accepting transitions.*

For simplicity, we use the notation $s_1 \xrightarrow{\omega} s_2$ to denote $\delta(s_1, \omega) = s_2$. The run r of a TDFA $\mathcal{A}$ on a finite trace $\rho = \rho[0], \rho[1], \cdots, \rho[n] \in (2^{\mathcal{P}})^+$ is a finite state sequence $r = s_0, s_1, \cdots, s_n$ such that s_0 is the initial state, $s_i \xrightarrow{\rho[i]} s_{i+1}$ is true for $0 \leq i < n$. Note that runs of TDFA do not need to include the destination state of the last transition, which is implicitly $s_{n+1} = \delta(s_n, \rho[n])$, since the starting state (s_n) together with the labels of the transition ($\rho[n]$) are sufficient to determine the destination. r is called *acyclic* iff $(s_i = s_j) \Leftrightarrow (i = j)$ for $0 \leq i, j < n$. Also, we say that ρ *runs across* s_i iff s_i is in the corresponding run r. The trace ρ is accepted by $\mathcal{A}$ iff the corresponding run r ends with an accepting transition, i.e., $\delta(s_n, \rho[n]) \in T$. The set of finite traces accepted by a TDFA $\mathcal{A}$ is the language of $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$.

According to [42], TDFA has the same expressiveness as the normal DFA, and for an LTL_f formula φ, there is a TDFA $\mathcal{A}_\varphi$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$. As a result, the LTL_f satisfiability-checking problem can be solved on the corresponding TDFA. That is, an LTL_f formula φ is satisfiable iff there is a finite trace accepted by its corresponding TDFA $\mathcal{A}_\varphi$ [42].

2.3 LTL_f realizability, synthesis, and TDFA games

Definition 2 (LTL_f realizability). *Let φ be an LTL_f formula whose alphabet is $\mathcal{P}$ and $\mathcal{X}, \mathcal{Y}$ be two subsets of $\mathcal{P}$ such that $\mathcal{X} \cap \mathcal{Y} = \emptyset$ and $\mathcal{X} \cup \mathcal{Y} = \mathcal{P}$. $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of output variables controlled by the system. φ is realizable with $\langle \mathcal{X}, \mathcal{Y} \rangle$ if there exists a strategy $g : (2^{\mathcal{X}})^* \to 2^{\mathcal{Y}}$ such that for an arbitrary infinite sequence $\lambda = X_0, X_1, \cdots \in (2^{\mathcal{X}})^\omega$ of propositional interpretations over $\mathcal{X}$, there is $k > 0$ such that $\rho \models \phi$ holds, where $\rho = (X_0 \cup g(\epsilon)), (X_1 \cup g(X_0)), \cdots, (X_k \cup g(X_0, \cdots, X_{k-1}))$. ($\epsilon$ means the empty trace.)*

Less formally, an LTL_f formula φ is realizable if there exists a winning strategy g for the outputs, namely that for every sequence of inputs, its combination with g's outputs up to some k, satisfies φ (it can be a different value of k for different input sequences). Notably, the synthesis defined above is called *system-first* synthesis, which means that at each point the value of Y depends on the value history of X. This paper focuses on the system-first synthesis.

The synthesis problem can be reduced to TDFA *games* [28,43] specified by $\mathcal{A}_\varphi$, with the help of definitions on the *winning/failure* states.

Definition 3 (System Winning/Failure State [43]). *For a* TDFA *game specified by $\mathcal{A} = (2^{\mathcal{Y} \cup \mathcal{X}}, S, s_0, \delta, T)$, $s \in S$ is a* system winning state *iff there is $Y \in 2^{\mathcal{Y}}$ such that for every $X \in 2^{\mathcal{X}}$, either $\delta(s, Y \cup X) = s'$ is an accepting transition or s' is a system winning state. State s is a* system failure state *iff s is not a system winning state.*

Theorem 1. *Given an LTL_f formula φ with $\langle \mathcal{X}, \mathcal{Y} \rangle$, a* TDFA *$\mathcal{A}_\varphi = (2^{\mathcal{Y}} \times 2^{\mathcal{X}}, S, s_0, \delta, T)$ such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, and the* TDFA *game specified by $\mathcal{A}_\varphi$, the following are equivalent:*

- *φ with $\langle \mathcal{X}, \mathcal{Y} \rangle$ is realizable;*
- *the system wins the game specified by $\mathcal{A}_\varphi$;*
- *s_0 is a system-winning state of the game specified by $\mathcal{A}_\varphi$.*

3 LTL_f-to-tdfa via Progression

In our new framework MoGuS, we translate the formula to its TDFA via *formula progression*. The progression technique originates in [5] for goal planning with temporal logics, and a definition of LTL_f progression has been used in [29]. Here we adapt the progression to finite traces instead of single propositions and adjust the translation process for TDFA.

Definition 4 (Formula Progression for LTL_f). *Given an LTL_f formula φ and a non-empty finite trace ρ, the progression formula $\mathsf{fp}(\varphi, \rho)$ is recursively defined as follows:*

- *$\mathsf{fp}(tt, \rho) = tt$ and $\mathsf{fp}(ff, \rho) = ff$;*

- $\mathsf{fp}(p, \rho) = tt$ *if* $p \in \rho[0]$; $\mathsf{fp}(p, \rho) = ff$ *if* $p \notin \rho[0]$;
- $\mathsf{fp}(\neg\varphi, \rho) = \neg\mathsf{fp}(\varphi, \rho)$;
- $\mathsf{fp}(\varphi_1 \wedge \varphi_2, \rho) = \mathsf{fp}(\varphi_1, \rho) \wedge \mathsf{fp}(\varphi_2, \rho)$;
- $\mathsf{fp}(\varphi_1 \vee \varphi_2, \rho) = \mathsf{fp}(\varphi_1, \rho) \vee \mathsf{fp}(\varphi_2, \rho)$;
- $\mathsf{fp}(\bigcirc\varphi, \rho) = \varphi$ *if* $|\rho| = 1$; *Else* $\mathsf{fp}(\bigcirc\varphi, \rho) = \mathsf{fp}(\varphi, \rho_1)$;
- $\mathsf{fp}(\bullet\varphi, \rho) = \varphi$ *if* $|\rho| = 1$; *Else* $\mathsf{fp}(\bullet\varphi, \rho) = \mathsf{fp}(\varphi, \rho_1)$;
- $\mathsf{fp}(\varphi_1 \,\mathcal{U}\, \varphi_2, \rho) = \mathsf{fp}(\varphi_2, \rho) \vee (\mathsf{fp}(\varphi_1, \rho) \wedge \mathsf{fp}(\bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2), \rho))$;
- $\mathsf{fp}(\varphi_1 \,\mathcal{R}\, \varphi_2, \rho) = \mathsf{fp}(\varphi_2, \rho) \wedge (\mathsf{fp}(\varphi_1, \rho) \vee \mathsf{fp}(\bullet(\varphi_1 \,\mathcal{R}\, \varphi_2), \rho))$.

The following lemmas are not hard to obtain based on Definition 4, whose proofs are omitted here.

Lemma 1. *Given an* LTL_f *formula* φ *and two non-empty finite traces* ρ_1 *and* ρ_2, $\rho_2 \models \mathsf{fp}(\varphi, \rho_1)$ *implies* $\rho_1 \cdot \rho_2 \models \varphi$.

Lemma 2. *Given an* LTL_f *formula* φ *and two non-empty finite traces* ρ_1 *and* ρ_2, *it holds that* $\mathsf{fp}(\mathsf{fp}(\varphi, \rho_1), \rho_2) = \mathsf{fp}(\varphi, \rho_1 \cdot \rho_2)$.

Lemma 3. *Given an* LTL_f *formula and a non-empty finite trace* ρ, $\rho \models \varphi$ *implies* $\rho_i \models \mathsf{fp}(\varphi, \rho^i)$ *for every* $0 \leq i < |\rho|$.

Now we re-construct the TDFA for an LTL_f formula.

Definition 5 (LTL_f to TDFA). *Given an* LTL_f *formula* φ, *the* TDFA $\mathcal{A}_\varphi$ *is a tuple* $(2^{\mathcal{P}}, S, \delta, s_0, T)$ *such that*

- $2^{\mathcal{P}}$ *is the alphabet, where* $\mathcal{P}$ *is the set of atoms of* φ;
- $S = \{\varphi\} \cup \{\mathsf{fp}(\varphi, \rho) \mid \forall \rho \in (2^{\mathcal{P}})^+\}$ *is the set of states;*
- $s_0 = \varphi$ *is the initial state;*
- $\delta : S \times 2^{\mathcal{P}} \to S$ *is the transition function such that* $\delta(s, \sigma) = \mathsf{fp}(s, \sigma)$ *for* $s \in S$ *and* $\sigma \in 2^{\mathcal{P}}$ *(Here* σ *is considered a trace with length 1);*
- $T = \{s_1 \xrightarrow{\sigma} s_2 \in \delta \mid \sigma \models s_1\}$ *is the set of accepting transitions.*

Theorem 2. *Given an* LTL_f *formula* φ *and the* TDFA $\mathcal{A}_\varphi$ *constructed by Definition 5, it holds that* $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}_\varphi)$.

Proof. Let $|\rho| = n + 1$ $(n \geq 0)$ and the corresponding run r of $\mathcal{A}_\varphi$ on ρ is $s_0, s_1, \ldots, s_n$, where $s_0 = \varphi$.

($\Leftarrow$) According to Definition 5, ρ is accepted by $\mathcal{A}_\varphi$ implies $(\rho_n = \rho[n]) \models s_n$ and $s_n = \mathsf{fp}(\varphi, \rho^n)$. Then from Lemma 1, we have $(\rho^n \cdot \rho_n = \rho) \models (s_0 = \varphi)$.

($\Rightarrow$) First from Definition 5, every $\mathsf{fp}(\varphi, \rho^i)$ for $0 \leq i \leq n$ is a state of $\mathcal{A}_\varphi$. Secondly, $\delta(\mathsf{fp}(\varphi, \rho^i), \rho[i]) = \mathsf{fp}(\varphi, \rho^{i+1})$ is true for $0 \leq i \leq n$, because $\mathsf{fp}(\varphi, \rho^{i+1}) = \mathsf{fp}(\mathsf{fp}(\varphi, \rho^i), \rho[i])$ is true (Lemma 2). Therefore, let $s_i = \mathsf{fp}(\varphi, \rho^i)$ $(0 \leq i \leq n)$ and the state sequence $r = s_0, s_1, \ldots, s_n$ is a run of $\mathcal{A}_\varphi$ on ρ. Finally, $\rho \models \varphi$ implies that $\rho_n \models (s_n = \mathsf{fp}(\varphi, \rho^n))$ is true because of Lemma 3. So ρ is accepted by $\mathcal{A}_\varphi$. □

Theorem 3 (Complexity). *Given an* LTL_f *formula* φ *and the* TDFA $\mathcal{A}_\varphi$ *constructed by Definition 5, the size of* $\mathcal{A}_\varphi$ *is at most* $2^{2^{|cl(\varphi)|}}$, *where* $cl(\varphi)$ *is the set of all subformulas of* φ.

Proof. From Definition 5, every state in $\mathcal{A}_\varphi$ excluding the initial one is computed via progression. It is not hard to prove that every formula from progression can be converted into the form of $\bigvee\bigwedge\psi$ where $\psi \in cl(\varphi)$. Since there are at most $2^{2^{|cl(\varphi)|}}$ formulas with the form $\bigvee\bigwedge\psi$ (including φ), the size of $\mathcal{A}_\varphi$ is also at most $2^{2^{|cl(\varphi)|}}$. □

4 Guided LTL$_f$ Synthesis with Satisfiable Traces

The previous forward synthesis approaches [43,29] require determining whether the state in the corresponding TDFA game is a winning or a failure state. In this process, the edges from the state are explored enumeratively, which is performed randomly without direction. By Definition 3, winning states are recursively defined with its base case falling on the accepting edges of TDFA. Therefore, we can intuitively infer that edges associated with some satisfiable traces are more likely to make the current state determined as winning. Inspired by that, we propose our new synthesis algorithm MoGuS (Model-Guided Synthesis). In the following, we first illustrate our new synthesis approach at a high level with an example (Sec. 4.1), introduce the details of the approach (Sec. 4.2), and then run the algorithmic details using the example again (Sec. 4.3).

4.1 An Example

We will use the formula

$$\varphi = \bigcirc\mathcal{F}(\bigcirc a \wedge \mathcal{G}b) \tag{1}$$

with $\mathcal{X} = \{a\}$ and $\mathcal{Y} = \{b\}$ as a running example. It is clearly unrealizable, because no system can guarantee $\bigcirc a$, since a is an input. In the beginning, we cannot determine whether s_0 is winning or failure and a finite trace ρ that satisfies φ is computed by an LTL$_f$ satisfiability solver (Figure 1 (a)). Suppose that $\rho = a \wedge b, a \wedge b, a \wedge b$. The corresponding TDFA run on ρ is $r = s_0, s_1, s_2$, where

$$s_1 = \mathcal{F}(\bigcirc a \wedge \mathcal{G}b), \tag{2}$$

and

$$s_2 = (a \wedge \mathcal{G}b) \vee (\bigcirc\mathcal{F}(\bigcirc a \wedge \mathcal{G}b)). \tag{3}$$

We now check whether s_2 is a winning state. As illustrated in Figure 1 (b), for every $Y \in 2^{\mathcal{Y}}$ there exists $X \in 2^{\mathcal{X}}$ such that it forms a loop from s_2. So it concludes that s_2 is a failure state and rolls back to s_1. For $Y = b$, it leads to a known failure state s_2, and for $Y = \neg b$, it detects a loop (Figure 1 (c)). This implies that s_1 is a failure state. Similarly, the initial state s_0 is found to be a failure state, and then MoGuS returns 'Unrealizable'.

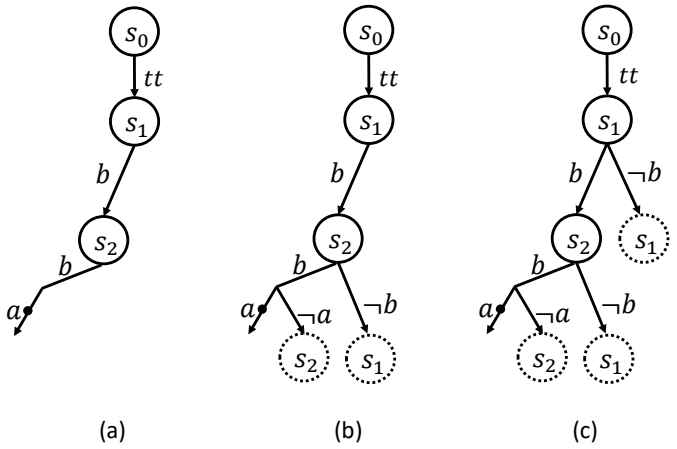


Fig. 1: Given the input formula $\bigcirc\mathcal{F}(\bigcirc a \wedge \mathcal{G}b)$ (recall that $\bigcirc$ denotes *strong Next*) with $\mathcal{X} = \{a\}$ and $\mathcal{Y} = \{b\}$, these diagrams show the progression of $\mathsf{MoGuS}(\varphi, \langle \mathcal{X}, \mathcal{Y} \rangle)$ in its satisfiability-based search for a winning strategy. Dashed circles represent loops, and points on the arrows represent accepting transitions. For simplicity, we merge edges with the same successor, i.e., tt, b, and $\neg b$ represent 4, 2, and 2 edges respectively. And here $s_0 = \bigcirc\mathcal{F}(\bigcirc a \wedge \mathcal{G}b)$, $s_1 = \mathcal{F}(\bigcirc a \wedge \mathcal{G}b)$, and $s_2 = (a \wedge \mathcal{G}b) \vee (\bigcirc\mathcal{F}(\bigcirc a \wedge \mathcal{G}b))$.

4.2 The Synthesis Algorithm MoGuS

Given a formula φ with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, we explore the TDFA by on-the-fly construction and perform a top-down traversal of the search space. Every time a TDFA state is visited, we first try to determine based on known information whether it is winning, failure, or forming a loop. If one of these three cases occurs, we backtrack to the previous state. Otherwise, we invoke the LTL_f satisfiability solver to find a satisfiable trace, subsequently exploring the states in the corresponding TDFA run. This is the primary distinction from the approach in [43]. Our search direction is guided by a satisfiable trace, rather than randomly selected by the (Boolean) SAT solver.

Algorithm 1 shows the implementation of MoGuS. It first declares four global sets: *winning* and *failure* to store the known winning and failure states respectively, *to_win* to collect winning state-edge pairs $\langle s, X \cup Y \rangle$ such that $X \cup Y \models s$ or $\mathsf{fp}(s, X \cup Y)$ is a winning state, and *to_fail* to collect state-edge pairs $\langle s, X \cup Y \rangle$ such that $X \cup Y \not\models s$ and $\mathsf{fp}(s, X \cup Y)$ is a failure state. *to_win* and *to_fail* are maintained to compute the edge constraint (Line 8). At the main entry of the algorithm, the parameter *path* refers to the state sequence that leads from the initial state to the current state ψ.

At Line 5, MoGuS checks whether ψ is winning or failure currently based on the state information collected so far, and Algorithm 2 presents the implementation of `currentWinning`. `currentWinning`(ψ) returns 'Winning' if (1) ψ is already in *winning*, or (2) based on Definition 3, there is $Y \in 2^{\mathcal{Y}}$ such that

$X \cup Y \models \psi$ holds or $\mathsf{fp}(\psi, X \cup Y)$ is in *winning*, for every $X \in 2^{\mathcal{X}}$. During the process, those transitions accepting or leading to a winning state are added into *to_win*. The analogous process is performed to check whether ψ is a failure state currently.

An edge constraint is computed for current ψ at Line 8, which blocks all edges not requiring further exploration. Formally, `edgeConstraint`(ψ) assigns *edge_constraint* as:

$$\bigwedge_{Y \in Y_f} \neg Y \ \wedge \ \bigwedge_{Y \in Y_u} \left(Y \rightarrow \bigwedge_{X \in X_w(Y)} \neg X \right). \tag{4}$$

Y_f (subscripts 'f'/'u' stand for 'failure'/'unknown') denotes a set of values for output variables, with which some assignments for input variables can lead the system to fail in the TDFA game.

$$Y_f = \left\{ Y \in 2^{\mathcal{Y}} \mid \exists X \in 2^{\mathcal{X}}. \langle \psi, X \cup Y \rangle \in to\_fail \right\} \tag{5}$$

For some output values collected in Y_u, the system retains the potential for winning. It no longer needs to explore inputs values that are known to lead the system winning, which are denoted by $X_w(Y)$ (subscript 'w' stands for 'winning'). And the right part of Equation 4 addresses this scenario.

$$Y_u = \left\{ Y \in 2^{\mathcal{Y}} \mid Y \notin Y_f \text{ and } \exists X \in 2^{\mathcal{X}}. \langle \psi, X \cup Y \rangle \notin to\_win \right\} \tag{6}$$

$$X_w(Y) = \left\{ X \in 2^{\mathcal{X}} \mid \langle \psi, X \cup Y \rangle \in to\_win \right\} \tag{7}$$

It checks the satisfiability of the current state ψ under the edge constraint at Line 9. If 'sat' is returned, the solver would compute a satisfiable trace ρ (Line 10). The first *for*-loop at Lines 12-18 generates each state in the run on ρ by formula progression and checks whether the new states are winning, failure, or forming a loop. Then in the second *for*-loop (Lines 19-24), it recursively checks whether each state $r[i]$ is winning or failure and add $r[i]$ to *winning* or *failure* respectively. Notably, the check order has to be reverse, i.e., from $|r|-1$ to 0, as MoGuS performs a Depth-First Search (DFS). At last, it recursively checks the current state ψ (Line 25) and returns the results.

While giving the search direction, the satisfiability solver also helps prevent the exploration of states that do not appear in any accepting run. The following lemma indicates that these states are system failure states.

Lemma 4. *Given a* TDFA *game over $\mathcal{A}$, s is a system failure state if s is not in any accepting run.*

And then, ψ can be determined as a failure state when $\psi \wedge edge\_constraint$ is unsatisfiable (Lines 31-32).

Lemma 5. *When the Algorithm 1 reaches Line 9, state ψ is a failure state if $\psi \wedge edge\_constraint$ is unsatisfiable, where edge_constraint is computed from Equations 4-7.*

Algorithm 1: MoGuS: Model-Guided Synthesis

Input: LTL_f formula φ with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$
Output: Realizable or Unrealizable

```
1  winning, failure, to_win, to_fail := ∅
2  return isWinning(φ, [φ]) ? Realizable : Unrealizable
3
4  Function isWinning(ψ, path)
5      peek := currentWinning(ψ, path)
6      if peek ≠ Unknown then
7          return peek =Winning
8      edge_constraint :=edgeConstraint(ψ)
9      if ltlfSat(ψ ∧ edge_constraint)=sat then
10         ρ := getModel()
11         Initialize r as an empty state sequence
12         for i from 0 to |ρ| − 2 do
13             s := fp(ψ, ρ[0 : i])
14             if s ∈ winning ∪ failure ∪ path then
15                 break
16             else
17                 r.pushBack(s)
18                 path.pushBack(s)
19         for i from |r| − 1 to 0 do
20             if isWinning(r[i], path) then
21                 winning := winning ∪ {r[i]}
22             else
23                 failure := failure ∪ {r[i]}
24             path.popBack()
25         if isWinning(ψ, path) then
26             winning := winning ∪ {r[i]}
27             return true
28         else
29             failure := failure ∪ {r[i]}
30             return false
31     else
32         return false
```

Proof. At Line 9, ψ has not been determined as winning, which implies $Y_f \cup Y_u = 2^{\mathcal{Y}}$. Therefore, there are two cases for $Y \in 2^{\mathcal{Y}}$:

- $Y \in Y_f$. By Equation 5, there exists $X \in 2^{\mathcal{X}}$ such that $\mathsf{fp}(\psi, X \cup Y)$ is a failure state or forming a loop.
- $Y \in Y_u$. Here we consider $X \in 2^{\mathcal{X}}$ such that $X \notin X_w(Y)$. By Equations 4, it holds that $X \cup Y \models \mathit{edge\_constrain}$. Therefore, $\psi \wedge \mathit{edge\_constraint}$ is

Algorithm 2: Implementation of `currentWinning`

Input: A TDFA state ψ and a state sequence *path* storing the states visited from the initial state to ψ.
Output: Winning, Failure, or Unknown

```
1  if isCurrentWinning(ψ) then
2      return Winning
3  if isCurrentFailure(ψ) then
4      return Failure
5  return Unknown
6
7  function isCurrentWinning(ψ)
8      if ψ ∈ winning then
9          return true
10     for each Y ∈ 2^𝒴 do
11         all_win :=true
12         for each X ∈ 2^𝒳 do
13             if X ∪ Y ⊭ ψ and fp(ψ, X ∪ Y) ∉ winning then
14                 all_win := false
15                 break
16             else
17                 to_win := to_win ∪ {⟨ψ, X ∪ Y⟩}
18         if all_win then
19             return true
20     return false
21 function isCurrentFailure(ψ)
22     if ψ ∈ failure then
23         return failure
24     for each Y ∈ 2^𝒴 do
25         Let exist_fail :=false
26         for each X ∈ 2^𝒳 do
27             if X ∪ Y ⊭ ψ and fp(ψ, X ∪ Y) ∈ failure ∪ path then
28                 exist_fail :=true
29                 to_fail := to_fail ∪ {⟨ψ, X ∪ Y⟩}
30                 break
31         if ¬exist_fail then
32             return false
33     return true
```

unsatisfiable implying that $\mathsf{fp}(\psi, X \cup Y)$ cannot appear in any accepting run. By Lemma 4, $\mathsf{fp}(\psi, X \cup Y)$ is a failure state.

Based on the above cases, ψ is a failure state by Definition 3. □

With Lemma 5 and Definition 3, we explicitly state this corollary in the context of Algorithm 1 for better understanding.

Corollary 1. *Given a* TDFA *state* ψ*,*

- ψ *is a winning state, if there exists* $Y \in 2^{\mathcal{Y}}$ *such that for every* $X \in 2^{\mathcal{X}}$*,* $X \cup Y \models \psi$ *or* $\mathsf{fp}(\psi, X \cup Y) \in winning$*;*
- ψ *is a failure state, if either* $\psi \wedge edge\_constrainr$ *is unsatisfiable, or for every* $Y \in 2^{\mathcal{Y}}$ *there exists* $X \in 2^{\mathcal{X}}$*,* $X \cup Y \not\models \psi$ *and* $\mathsf{fp}(\psi, X \cup Y) \in failure \cup path$*.*

Theorem 4. *Algorithm 1 is complete and sound. That is, given an* LTL_f *formula* φ *with inputs* $\mathcal{X}$ *and outputs* $\mathcal{Y}$*,*

- *Algorithm 1 can terminate within time of* $O(2^{|\mathcal{X} \cup \mathcal{Y}|} \cdot 2^{2^{|cl(\varphi)|}})$*;*
- φ *with* $\langle \mathcal{X}, \mathcal{Y} \rangle$ *is realizable iff Algorithm 1 returns* 'Realizable'*.*

Proof. **Completeness.** The number of states related to recursive calls to `isWinning` is bounded by the worst case doubly-exponential number of states in the constructed TDFA (Theorem 3). Before every newly computed state is checked recursively, MoGuS first checks for winning, failure, and loop (Line 14). Then *edge_constraint* helps enumerate the edges and successors of each state, which guarantees that each state is visited at most $2^{|\mathcal{X} \cup \mathcal{Y}|}$ times. Hence, the time complexity of Algorithm 1 is $O(2^{|\mathcal{X} \cup \mathcal{Y}|} \cdot 2^{2^{|cl(\varphi)|}})$.

Soundness. ($\Leftarrow$) In Algorithm 1, there are two cases in which MoGuS returns 'Realizable'. The first one (Line 5) is when `currentWinning`$(\psi, path)$ returns 'Winning', which implements exactly Definition 3. This indicates that φ is already a winning state and according to Theorem 1, φ with $\langle \mathcal{X}, \mathcal{Y} \rangle$ is realizable. The second case is recursively calling `isWinning` (Line 25), which finally falls into the base case, i.e., `currentWinning`$(\psi, path)$ returns 'Winning'. This situation has been discussed before.

($\Rightarrow$) We perform this part of proof by contraposition. If Algorithm 1 returns 'Unrealizable', then φ with $\langle \mathcal{X}, \mathcal{Y} \rangle$ is unrealizable. There are three cases leading MoGuS to return 'Unealizable'. The first and second cases (Lines 5, 25) are analogous to those of ($\Leftarrow$). And the third case at Line 32 exactly corresponds to the conclusion of Lemma 5. □

4.3 Back to Our Example

Figure 2 illustrates how MoGuS works in detail, based on the example described in subsection 4.1. We now describe the main steps of Algorithm 1 when applied to this example. At Line 5 of Algorithm 1, `currentWinning` finds a transition $\langle s_0, a \wedge \neg b \rangle$, adds it to to_fail, and returns 'Unknown'. Then *edge_constraint* is assigned as b at Line 8. And `ltlfSat` computes a model of φ (Line 9), which

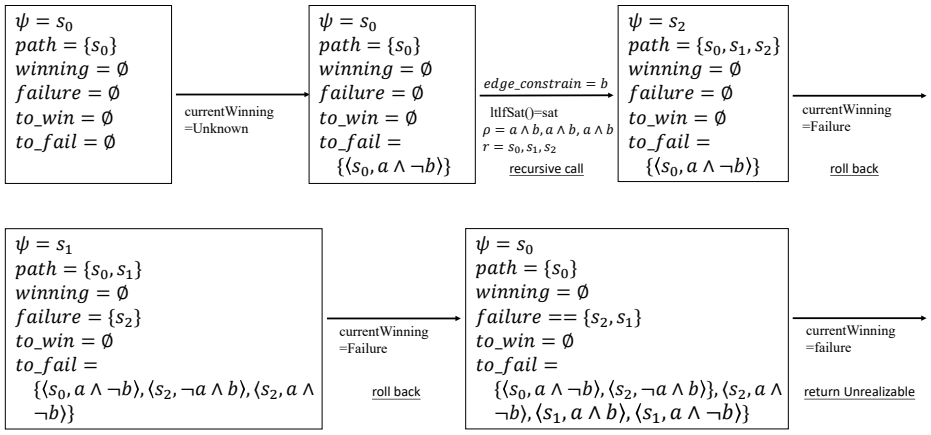


Fig. 2: The main steps of MoGuS$(\varphi, \langle \mathcal{X}, \mathcal{Y} \rangle)$ when checking the realizability of Equation (1) with $\mathcal{X} = \{a\}$ and $\mathcal{Y} = \{b\}$, where we have $\varphi = s_0 = \mathsf{O}\mathcal{F}(\mathsf{O}a \wedge \mathcal{G}b)$, $s_1 = \mathcal{F}(\mathsf{O}a \wedge \mathcal{G}b)$ and $s_2 = (a \wedge \mathcal{G}b) \vee (\mathsf{O}\mathcal{F}(\mathsf{O}a \wedge \mathcal{G}b))$.

in the figure returns $\rho = \{a \wedge b\}, \{a \wedge b\}, \{a \wedge b\}$. The corresponding run of ρ is $r = s_0, s_1, s_2$, of which states are checked whether winning, failure, or loop and added to *path* at Lines 14-18. Next, MoGuS recursively checks whether the new states in r (with a reverse order) can be winning (Line 20). As shown in the figure, s_2 turns out to be a failure state by `currentWinning` and is added to $failure$, since for every $Y \in 2^{\mathcal{Y}}$ there is a chance of forming a loop through s_2 (see Figure 1). Then, the algorithm rolls back state by state and sequentially updates the failure states and transitions into $failure$ and to_fail respectively. The recursive process finally goes back to s_0 and concludes that φ with $\langle \mathcal{X}, \mathcal{Y} \rangle$ is unrealizable.

5 Experimental Evaluation

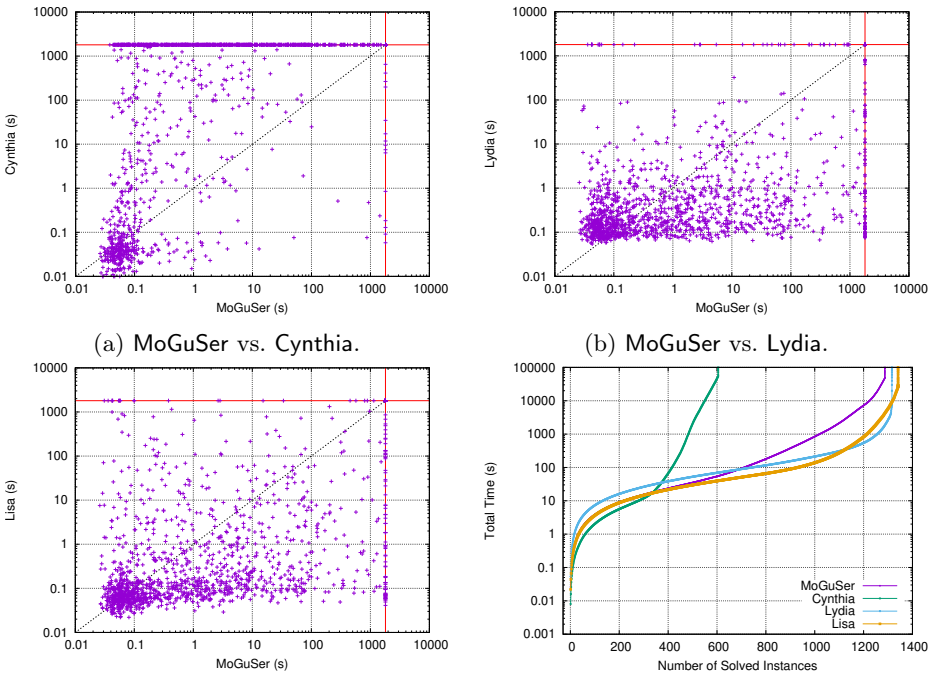
5.1 Experimental Set-up

Tools. We implemented MoGuS in a tool called MoGuSer using C++, and integrated aaltaf [33] as the engine for LTL$_f$ satisfiability checking. We compared the results with three state-of-the-art synthesis tools, Lisa [7], Lydia [21], and Cynthia [29]. The first two tools are based on the bottom-up approach, and Cynthia is SDD-based and performs forward synthesis. All three tools were run with their default parameters.

Benchmarks. We ran the experiment with the collected benchmarks in [29], which are in total 1454 instances, including 1400 *Random* instances, and 54 *Two-player-Games* instances. Based on our preliminary experimental results, the majority of existing cases can be solved by the evaluated tools. To better compare the scalability of different tools, we also created a new set of benchmarks, which we call *Ascending*. We employed the `randltl` command in Spot [23]

to generate a batch of random LTL formulas, which are treated as LTL_f formulas (since LTL_f formulas share the same syntax as LTL), and then randomly divide the atomic variables from these formulas into input and output variables. The `--tree-size`[6] option of `randltl` specifies the tree size of the generated formulas, and we generated 200 test cases for each size ranging from 100 to 900 (1800 in total). Although the *Ascending* benchmark consists of random formulas as well, they have much larger sizes than those in [29] and are more suitable to evaluate the tools' scalability.

Platform. We ran the experiments on a CentOS 7.4 cluster, where each instance had exclusive access to a processor core of the Intel Xeon 6230 CPU running at 2.1 GHz, with 8 GB of memory and a 30-minute time limit. We measured execution time with the Unix command `time`. When collecting the data, we recorded a running time of 1801 seconds for all instances that could not be solved by the tested tool within this time limit. We verified that the results emitted by the four tools are consistent (excluding those that timed out).



(a) MoGuSer vs. Cynthia. (b) MoGuSer vs. Lydia.

(c) MoGuSer vs. Lisa (d) Total number of solved instances.

Fig. 3: Comparison of different solvers' performance on Random and Two-player-Games benchmarks together. Points located on the red line represent instances that fail to be solved within the given time and memory resources.

[6] See https://spot.lre.epita.fr/man/randltl.1.html.

5.2 Results and Analysis I: Random and Two-player-Games Benchmarks

We first evaluated the four tools on benchmarks collected from previous literature, i.e., Random and Two-player-Games benchmarks. The pairwise comparison of MoGuSer against Cynthia, Lydia, and Lisa is shown in Figures 3a-3c, respectively. Figure 3d illustrates the total number of successfully solved cases accumulated over time. Tables 1 and 2 present quantitative summaries of results on these benchmarks.

Comparing MoGuSer with the other top-down tool Cynthia, we can observe that MoGuS has significantly improved the top-down solving capability for LTL_f synthesis problems. MoGuSer could solve many more instances than Cynthia, and when both tools can solve an instance, MoGuSer demonstrates faster performance (Figure 3a). As shown in Table 1, MoGuSer solved 918 unrealizable Random instances, while Cynthia only solved 254. These facts indicate that the targeted search strategy of MoGuS can avoid unnecessary searches (especially in unrealizable cases), resulting in faster convergence. Besides, MoGuSer achieves better results on both *Counter(s)* instances than Cynthia, while Cynthia still keeps a distinct advantage over other tools on the *Nim* test cases.

Table 1: Results on Random instances.

Tools	Realizable		Unrealizable		Total	
	Solved	Uniquely solved	Solved	Uniquely solved	Solved	Uniquely solved
MoGuSer	347	0	918	6	1265	6
Cynthia	324	0	254	0	578	0
Lydia	350	0	932	0	1282	0
Lisa	351	0	960	11	1311	11

Table 2: Results on Two-player-Games instances.

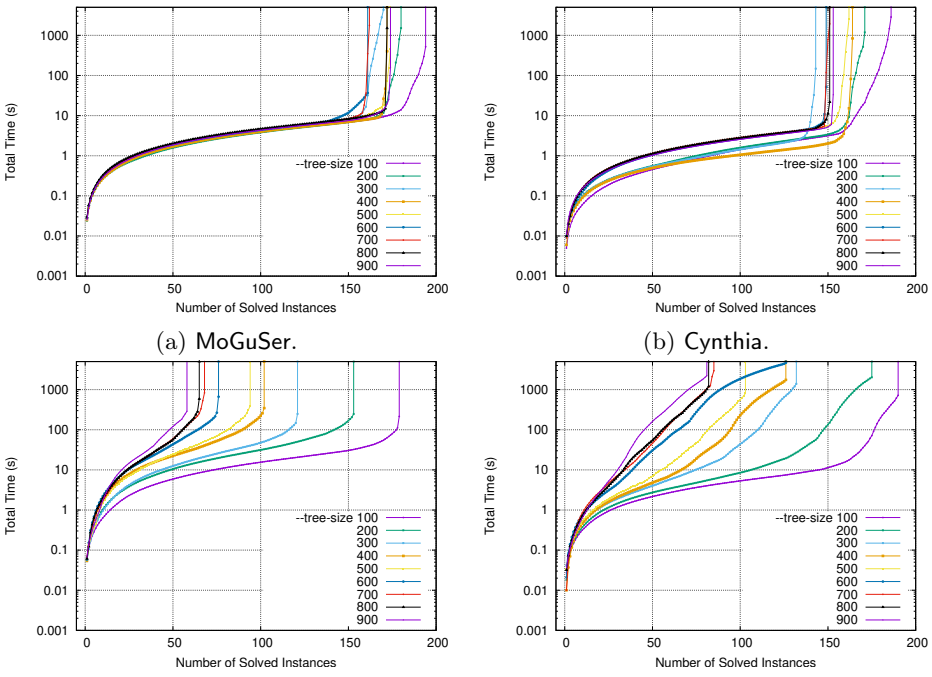
Tools	Single-counter		Double-counters		Nim	
	Solved	Uniquely solved	Solved	Uniquely solved	Solved	Uniquely solved
MoGuSer	8	0	6	1	8	0
Cynthia	4	0	2	0	21	4
Lydia	11	3	6	0	17	0
Lisa	8	0	7	0	13	1

As for the two bottom-up tools, Lydia and Lisa achieve similar performance in most aspects and slightly outperform MoGuSer. The distribution pattern of

points in Figure 3b closely resembles that in Figure 3c. We can observe that each tool has its own merits in solving speed, MoGuSer performs almost the same as Lydia/Lisa among instances that can be solved by both tools. From the endpoints of the curves in Figure 3d, Tables 1 and 2, the number of instances solved by MoGuSer is slightly lower than that of Lydia and Lisa. Thus we conclude that the MoGuS has improved the top-down solving capability for LTL_f synthesis problems to a level nearly equivalent to that of bottom-up tools (Lydia and Lisa).

Taking a comprehensive view of Tables 1 and 2, the bottom-up approach still holds a slight advantage over the top-down approach. However, no LTL_f synthesis tool dominates all other tools, since each of them can uniquely solve some instances. Similarly to hardware model checking [44], this emphasizes the need for maintaining a portfolio consisting of diverse approaches for LTL_f synthesis.

5.3 Results and Analysis II: Ascending Benchmarks



(a) MoGuSer. (b) Cynthia.

(c) Lydia. (d) Lisa.

Fig. 4: The cumulative number of instances solved by each tool over time on the *Ascending* benchmarks.

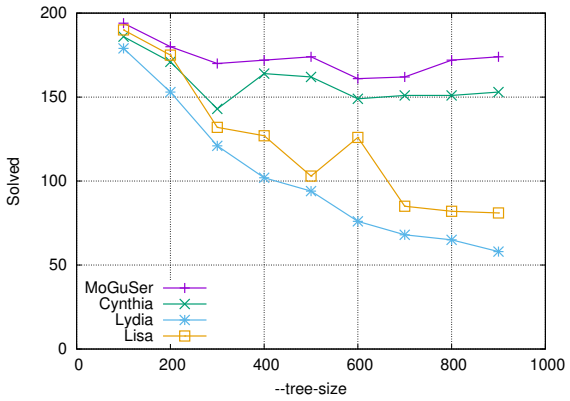


Fig. 5: The number of solved instances as a function of the formula's tree size.

We further evaluated the scalability of the four tools by a collection of test cases sorted in ascending order based on the tree sizes of formulas. Figure 4 depicts the cumulative number of instances solved by each tool over time on benchmarks of varying sizes. Meanwhile, Figure 5 illustrates the number of solved instances as a function of the formula's tree size.

Both MoGuSer and Cynthia demonstrate robust scalability, but MoGuSer performs better than Cynthia. The endpoints of the curves in Figures 4a and 4b are not clearly located by the tree sizes of the formulas, and the curves corresponding to MoGuSer and Cynthia in Figure 5 fluctuate but are generally stable. These suggest that the solving capabilities of MoGuSer and Cynthia are not significantly limited by the problem size. Besides, MoGuSer solved more cases than Cynthia, which reflects the efficiency of the top-down methodology by MoGuSer. Besides, Cynthia achieves relatively satisfactory results on the *Ascending* benchmarks. We infer that this can be attributed to the fact that realizable instances account for a large portion of the *Ascending* benchmarks, since Cynthia's ability to solve realizable instances is comparable to the other three tools (see Table 1). Excluding 101 instances that cannot be solved by any tool, there are 1475 realizable instances among the remaining 1699 instances.

On the other hand, the performance of the two tools (Lisa and Lydia) based on the bottom-up method drops significantly as the LTL_f formula size grows. The curves in Figures 4c and 4d indicate that the solving speed of Lisa and Lydia gradually slows down as the tree sizes of the formulas increase. From the positions of the endpoints of the curves in Figures 4c, 4d and the trend of the lower two curves in Figure 5, the number of instances successfully solved by Lydia and Lisa also change significantly. When `-tree-size` is set to 100, Lydia and Lisa solve 179 and 190 respectively, and when –tree-size increases to 900, only 58 and 81 are solved respectively. We speculate that the reason is that both Lydia and Lisa require the construction of complete DFA before synthesis, which relies heavily on BDDs [10]. BDDs can require exponential space in the number of variables, which limits the capability of synthesis tools.

6 LTL synthesis – Related Work

We have discussed various works related to LTL_f synthesis in the introduction. Let us mention here some comments about related work. Temporal synthesis is a classical problem, first proposed by A. Church in the 1960s [17]. The original logic specification to be synthesized was expressed by an S1S formula, for which the complexity to solve the problem is non-elementary [12,40]. The first work to consider LTL synthesis is [39], which solves the synthesis problem by reducing it to a Rabin game [25]. This approach constructs a non-deterministic Büchi automaton from the input LTL formula, and then determinizes it to its equivalent Rabin automaton, a process which takes worst-case double-exponential time. The complexity of solving a Rabin game is NP-Complete [25]. Nowadays, the standard approach is to reduce LTL synthesis to the parity game [26], because a parity game can be solved in quasi-polynomial time [13], even though the doubly-exponential process to obtain a deterministic parity automaton cannot be avoided. LTL synthesis tools like ltlsynt [37] and Strix [36], are built using the parity-game approach. Because of the challenge to determinize an ω automaton, researchers also consider other possibilities, e.g., by reducing LTL synthesis to the bounded safety game [31]. Acacia+ [9] is a representative tool following the safety-game approach. The annual reactive synthesis competition [1] drives progress in this field, yet the scalability issue is still a major problem.

7 Concluding Remarks

We have presented a new approach called MoGuS for synthesizing LTL_f formulas. By invoking an LTL_f satisfiability checker, MoGuS performs the search for a winning strategy in a more targeted way compared to previous top-down approaches. An empirical comparison of this method to state-of-the-art LTL_f synthesizers suggests that it can achieve the best overall performance. Several future works are being considered. Firstly, MoGuS relies heavily on an LTL_f satisfiability checker, and hence it can benefit from performance improvements of this stage. We can explore incrementally invoking the LTL_f satisfiability checker. Secondly, both the bottom-up tools Lydia and Lisa integrated with composition techniques [7,21,6], which decomposes the formula on conjunctions, perform synthesis of each conjunct, and then combine the results in order to solve the original problem. Similar composition ideas could also be applied to top-down synthesis approaches. Finally, it is interesting to check whether the optimizations described in this article can accelerate LTL synthesis of safety properties [32,47].

Acknowledgements This work is supported by National Natural Science Foundation of China (Grant #U21B2015 and #62372178), "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, Shanghai Collaborative Innovation Center of Trusted Industry Internet Software, by US NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, and by US DoD MURI grant N00014-20-1-2787.

Data-Availability Statement To support the experimental results, the source code of MoGuSer and benchmarks is available at https://drive.google.com/file/d/1ohOa4Kl4R4br095k-kVJcWV87U5XON5q/view?usp=sharing.

References

1. The reactive synthesis competition. http://www.syntcomp.org/
2. Althoff, C., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. In: Proc. 10th Int. Conf. on the Implementation and Application of Automata. Lecture Notes in Computer Science, vol. 3845, pp. 262–272. Springer (2005)
3. Aminof, B., De Giacomo, G., Murano, A., Rubin, S.: Synthesis under assumptions. In: Sixteenth International Conference on Principles of Knowledge Representation and Reasoning. pp. 615–616. AAAI Press (2018)
4. Aminof, B., De Giacomo, G., Murano, A., Rubin, S.: Planning under LTL environment specifications. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling. pp. 31–39. AAAI Press (2019)
5. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence **22**, 5–27 (1998)
6. Bansal, S., Giacomo, G.D., Stasio, A.D., Li, Y., Vardi, M.Y., Zhu, S.: Compositional safety LTL synthesis. In: Verified Software: Theories, Tools, and Experiments (VSTTE) (2022)
7. Bansal, S., Li, Y., Tabajara, L., Vardi, M.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence. vol. 34, pp. 9766–9774. AAAI Press (2020)
8. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012), in Commemoration of Amir Pnueli
9. Bohy, A., Filiot, E., Jin, N.: Acacia+, a tool for LTL synthesis. In: of Lecture Notes in Computer Science. pp. 652–657. Springer-Verlag (2012)
10. Bryant, R.: Graph-based algorithms for Boolean-function manipulation. IEEE Transactions on Computing **C-35**(8), 677–691 (1986)
11. Büchi, J.: On a decision method in restricted second order arithmetic. In: Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960. pp. 1–12. Stanford University Press (1962)
12. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS **138**, 295–311 (1969)
13. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing. p. 252–263. STOC 2017, Association for Computing Machinery, New York, NY, USA (2017)
14. Camacho, A., Bienvenu, M., McIlraith, S.A.: Finite LTL synthesis with environment assumptions and quality measures. In: Sixteenth International Conference on Principles of Knowledge Representation and Reasoning. pp. 454–463. AAAI Press (2018)
15. Camacho, A., McIlraith, S.A.: Strong fully observable non-deterministic planning with LTL and LTLf goals. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. pp. 5523–5531 (2019)

16. Camacho, A., Triantafillou, E., Muise, C.J., Baier, J.A., McIlraith, S.A.: Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. pp. 3716–3724. AAAI Press (2017)
17. Church, A.: Logic, arithmetics, and automata. In: Proc. Int. Congress of Mathematicians, 1962. pp. 23–35. Institut Mittag-Leffler (1963)
18. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Journal of Symbolic Logic **28**(4), 289–290 (1963)
19. Darwiche, A.: Sdd: A new canonical representation of propositional knowledge bases. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence. p. 819–826. AAAI Press (2011)
20. De Giacomo, G., Vardi, M.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence. pp. 854–860. AAAI Press (2013)
21. De Giacomo, G., Favorito, M.: Compositional approach to translate LTLf/LDLf into deterministic finite automata. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 31, pp. 122–130 (2021)
22. De Giacomo, G., Rubin, S.: Automata-theoretic foundations of fond planning for LTLf and LDLf goals. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence. p. 4729–4735. AAAI Press (2018)
23. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22). Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (Aug 2022). https://doi.org/10.1007/978-3-031-13188-2_9
24. Eén, N., Sörensson, N.: An extensible SAT-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
25. Emerson, E., Jutla, C.: The complexity of tree automata and logics of programs. In: Proc. 29th IEEE Symp. on Foundations of Computer Science. pp. 328–337 (1988)
26. Emerson, E., Jutla, C.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd IEEE Symp. on Foundations of Computer Science. pp. 368–377 (1991)
27. Fuggitti, F.: FOND planning for LTLf and PLTLf goals (2020). https://doi.org/10.48550/ARXIV.2004.07027
28. Giacomo, G.D., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Proceedings of the 24th International Conference on Artificial Intelligence. pp. 1558–1564. AAAI Press (2015)
29. Giacomo, G.D., Favorito, M., Li, J., Vardi, M.Y., Xiao, S., Zhu, S.: LTLf synthesis as and-or graph search: Knowledge compilation at work. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence. pp. 3292–3298. AAAI Press (2022)
30. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Proc. 1st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 1019, pp. 89–110. Springer (1995)
31. Kupferman, O.: Avoiding determinization. In: Proc. 21st IEEE Symp. on Logic in Computer Science. pp. 243–254 (2006)
32. Kupferman, O., Vardi, M.: Model checking of safety properties. In: Proc. 11th Int. Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 1633, pp. 172–183. Springer (1999)

33. Li, J., Rozier, K.Y., Pu, G., Zhang, Y., Vardi, M.Y.: SAT-based explicit LTLf satisfiability checking. In: The Thirty-Third AAAI Conference on Artificial Intelligence. pp. 2946–2953. AAAI Press (2019)
34. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL$_f$ satisfibility checking. In: Proceedings of the Twenty-First European Conference on Artificial Intelligence. p. 513–518. IOS Press (2014)
35. Luo, W., Wan, H., Du, J., Li, X., Fu, Y., Ye, R., Zhang, D.: Teaching LTLf satisfiability checking to neural network. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence. pp. 3292–3298. AAAI Press (2022)
36. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018)
37. Michaud, T., Colange, M.: Reactive synthesis from LTL specification with spot. In: Proceedings Seventh Workshop on Synthesis, SYNT@CAV 2018. Electronic Proceedings in Theoretical Computer Science, vol. xx, p. xx (2018)
38. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
39. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Proceedings of the 16th International Colloquium on Automata, Languages and Programming. p. 652–671. ICALP '89, Springer-Verlag, Berlin, Heidelberg (1989)
40. Rabin, M.: Automata on infinite objects and Church's problem. Amer. Mathematical Society (1972)
41. Safra, S.: On the complexity of ω-automata. In: Proc. 29th IEEE Symp. on Foundations of Computer Science. pp. 319–327 (1988)
42. Shi, Y., Xiao, S., Li, J., Guo, J., Pu, G.: SAT-based automata construction for LTL over finite traces. In: 27th Asia-Pacific Software Engineering Conference (APSEC). pp. 1–10. IEEE (2020). https://doi.org/10.1109/APSEC51365.2020.00008
43. Xiao, S., Li, J., Zhu, S., Shi, Y., Pu, G., Vardi, M.Y.: On the fly synthesis for LTL over finite traces. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence. pp. 6530–6537. AAAI Press (2021)
44. Zhang, X., Xiao, S., Xia, Y., Li, J., Chen, M., Pu, G.: Accelerate safety model checking based on complementary approximate reachability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **42**(9), 3105–3117 (2023). https://doi.org/10.1109/TCAD.2023.3236272
45. Zhu, S., Tabajara, L., Li, J., Pu, G., Vardi, M.: Symbolic LTLf synthesis. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. pp. 1362–1369. AAAI Press (2017)
46. Zhu, S., Giacomo, G.D., Pu, G., Vardi, M.Y.: LTLf synthesis with fairness and stability assumptions. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence. pp. 3088–3095. AAAI Press (2020)
47. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: A symbolic approach to safety LTL synthesis. In: Strichman, O., Tzoref-Brill, R. (eds.) Hardware and Software: Verification and Testing. pp. 147–162. Springer International Publishing, Cham (2017)