SAT-Based Automata Construction for LTL over Finite Traces

Yingying Shi, Shengping Xiao, Jianwen Li, Jian Guo, Geguang Pu
Software Engineering Institute
East China Normal University
Shanghai, China
{51184501042,10175101164}@stu.ecnu.edu.cn; {jwli,jguo,ggpu}@sei.ecnu.edu.cn

2020 27th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-7281-9553-7/20/$31.00 ©2020 IEEE | DOI: 10.1109/APSEC51365.2020.00008

***Abstract*—In this paper, we consider the automata construction problem for Linear Temporal Logic over finite traces, i.e., LTL_f. We propose a SAT-based approach to translate an LTL_f formula to both of its equivalent Nondeterministic and Deterministic Finite Automata (NFA and DFA). Notably, the generated automata are transition-based instead of state-based, which may potentially be a better fit for the applications that can be achieved on the fly, e.g. LTL_f satisfiability checking and synthesis. Unlike extant approaches to translate LTL_f formulas to the equivalent finite automata, which are indirect and have to introduce intermediate procedures, our methodology enables the direct construction from LTL_f formulas to the finite automata. We evaluated our NFA construction together with other two LTL_f-to-automata approaches implemented in the MONA and SPOT tools, which shows that the performance of our construction is comparable to the other two. We leave the comparison on the DFA construction in the future work.**

***Index Terms*—LTL, LTL_f, LTL_f-to-NFA, LTL_f-to-DFA, SAT, On-the-fly construction**

I. Introduction

Linear Temporal Logic (LTL) was first introduced into Computer Science in 1977 [29], and since then it has been widely used in formal verification like model checking [12], runtime verification [4] and property synthesis [7], [24]. Beyond that, LTL also receives a lot of interests from other research communities such as software engineering [3] and artificial intelligence [1]. The standard LTL formulas are interpreted over infinite (linear) traces, which are suitable for describing the infinite behaviors of the nonterminating systems. Meanwhile, AI applications like motion planning [1], [8], [10], [15], [28], plan constraints [2], [19] and user preferences [5], [6], [31], make more concern on the finite behaviors of the systems.

Recently, LTL_f, a variant of LTL whose semantics are interpreted over finite traces, has raised great interests from the AI community [13], [14], [21]. Researches on LTL_f satisfiability checking [18], [25], [27] and LTL_f synthesis [9], [11], [21], [32], [36] have been extensively investigated. To solve these problems, the crux is how to translate the LTL_f formula to its equivalent finite automaton. Constructing the NFA is sufficient for the satisfiability checking, while constructing the DFA can be used for LTL_f synthesis. This paper focuses on the NFA and DFA generation from a given LTL_f formula.

It is well-known that every LTL formula, which is interpreted over infinite semantics, has an equivalent Büchi automaton such that they accept the same languages [20]. Analogously, for each LTL_f formula interpreted over finite semantics, there is an equivalent NFA (or DFA) such that they accept the same languages [21]. Currently, such finite automata are mainly obtained from SPOT [16] or MONA [17], [23], in an indirect way. SPOT is the state-of-the-art LTL-to-Büchi translator, so to construct the finite NFA of the input LTL_f formula one has to first convert the LTL_f formula to its equi-satisfiable LTL formula, generate the Büchi automaton and then convert the Büchi automaton to the NFA[1]. MONA is a tool that can translate monadic first-order logic [33] to the equivalent minimal DFA. So one can also first convert the LTL_f formula to its equivalent monadic First-Order Logic (FOL) and then utilize MONA to obtain the minimal DFA [36]. Both SPOT and MONA utilize BDD techniques to keep the generated automata small.

However, constructing the NFA/DFA via SPOT or MONA for LTL_f formulas is not only inconvenient but also inefficient. For LTL_f satisfiability checking, it may be too late to do the checking after the whole NFA is generated, since in theory the complexity of LTL_f-to-NFA translation is exponential. It is even worse for LTL_f synthesis, whose solutions rely on the generated DFA that involves a double-exponential complexity. In fact, both the LTL_f satisfiability checking and LTL_f synthesis should benefit from the on-the-fly LTL_f-to-NFA/DFA translation, which only construct the necessary part of automata for the purpose [20].

Inspired from the above motivation, we present in this paper the first on-the-fly construction from a given LTL_f formula to the equivalent transition-based NFA (TNFA) and transition-based DFA (TDFA). TNFA (resp. TDFA) is a variant of NFA (resp. DFA) whose accepting conditions are defined over transitions instead of states. The key point why our algorithm is able to conduct an on-the-fly construction is that it is based on the modern SAT techniques for the state and transition computation. Furthermore, our constructions are direct from LTL_f to the resulting automata without any intermediate steps.

Jianwen Li is the corresponding author.

[1]Details see https://spot.lrde.epita.fr/tut12.html.

2640-0715/20/$31.00 ©2020 IEEE
DOI 10.1109/APSEC51365.2020.00008

To convert a TNFA to the equivalent NFA, we show that there is a simple way which requires to add only one single state. In terms of converting a TDFA to the equivalent DFA, we propose to first obtain the equivalent NFA by applying the TNFA-to-NFA approach and then use the standard Subset Construction [30] to generate the corresponding DFA.

This paper is considered as the pre-processing work for LTL_f on-the-fly synthesis in our future work, and thus focuses on the task to construct the automata on the fly. Despite the inconvenience of converting the TDFA to the corresponding DFA, we consider TDFA as a better data structure to achieve LTL_f synthesis rather than the state-based DFA. On one hand, both the LTL model checking and LTL satisfiability checking results affirm the superiority of the transition-based (Büchi) automata over the state-based ones [22]. This advantage may also be applicable to the transition-based finite automata. On the other hand, TDFA is easily constructed by an on-the-fly algorithm, as we will introduce in the following, and may be potentially better for the synthesis on the fly.

To evaluate the performance of the LTL_f-to-Automata construction, we compared our approach with two extant ones that are implemented in SPOT and MONA tools respectively. Our experiments show that our new approach is comparable to these two ones on the NFA construction. Notably, we leave the DFA construction in the future work. In summary, the contributions of this paper are as follows.

- We present the first direct construction from LTL_f formulas to TNFA and TDFA without any intermediate steps;
- The proposed construction leverages the modern SAT techniques such that the automata can be generated on the fly;
- Although there is already transition-based Büchi automata, this paper is the first to present the concept of transition-based finite automata, to the best of our knowledge.
- We evaluated our construction together with other two LTL_f-to-automata approaches implemented in the MONA and SPOT tools, which shows that the performance of our construction is comparable to the other two.

The rest of the paper are organized as follows. Section II introduces the preliminaries, and Section III introduces the overview of our approach in a high-level way. Then Section IV presents our formal construction from LTL_f to the corresponding finite automata. Section V presents the implementation details of our algorithms. Section VI shows our experimental results. Finally, Section VII discusses and concludes the paper.

II. Preliminaries

This section introduces the basic concepts w.r.t the linear temporal logic over finite traces (LTL_f) and finite automata.

A. LTL over Finite Traces (LTL_f)

Classical LTL formulas are interpreted over infinite traces, which are widely used as the formal property languages to describe system behaviors. Meanwhile, the formulas of LTL_f, which is a variant of LTL, are interpreted over finite traces. LTL_f are so far widely used in the AI community to describe finite system bahaviors. Given a set P of atomic propositions, the syntax of an LTL_f formula ϕ is shown as below.

$$\phi := \mathsf{tt} \mid \mathsf{ff} \mid p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathcal{X}\phi \mid \mathcal{N}\phi \mid \phi\mathcal{U}\phi \mid \phi\mathcal{R}\phi$$

In the above, $p \in P$ is an *atom*, while $\mathcal{X}$ (strong Next), $\mathcal{N}$ (weak Next), $\mathcal{U}$ (Until), and $\mathcal{R}$ (Release) are temporal operators. A *literal* is an *atom* $p \in P$ or its negation $\neg p$. $\mathcal{X}$ and $\mathcal{N}$ (resp. $\mathcal{U}$ and $\mathcal{R}$) are dual operators, i.e. $\mathcal{X}\phi \equiv \neg(\mathcal{N}\neg\phi)$ (resp. $\phi_1\mathcal{U}\phi_2 \equiv \neg(\neg\phi_1\mathcal{R}\neg\phi_2)$) is semantically true. Boolean operators such as $\rightarrow$ and $\leftrightarrow$ can be represented by the combination $(\neg, \vee)$ or $(\neg, \wedge)$. Furthermore, we denote the constant true as tt and false as ff. We use the particular notation $\Diamond$ (Future) (resp. $\Box$ (Global)) for $\mathcal{U}$ (resp. $\mathcal{R}$) such that $\Diamond\phi \equiv \mathsf{tt}\mathcal{U}\phi$ (resp. $\Box\phi \equiv \mathsf{ff}\mathcal{R}\phi$) is semantically true. Let $\Sigma = 2^P$ be the set of alphabet and $\eta \in \Sigma^+$ is a finite nonempty trace defined over Σ, with $\eta = \sigma_0\sigma_1 \ldots \sigma_n$. For $0 \leq i \leq n$, we use $\eta[i]$ to denote the i-th element of η (i.e., σ_i), use η_i to denote $\sigma_i \ldots \sigma_n$ which is the suffix of η starting from position i, and use $|\eta| = n + 1$ to denote the length of η. Then the semantics of LTL_f formulas are interpreted as follows.

- $\eta \models \mathsf{tt}$ and $\eta \not\models \mathsf{ff}$;
- $\eta \models p$ iff $p \in \eta[0]$, where $p \in P$;
- $\eta \models \neg\phi$ iff $\eta \not\models \phi$;
- $\eta \models \phi_1 \wedge \phi_2$ iff $\eta \models \phi_1$ and $\eta \models \phi_2$;
- $\eta \models \phi_1 \vee \phi_2$ iff $\eta \models \phi_1$ or $\eta \models \phi_2$;
- $\eta \models \mathcal{X}\phi$ iff $|\eta| > 1$ and $\eta_1 \models \phi$;
- $\eta \models \mathcal{N}\phi$ iff either $|\eta| = 1$, or $\eta_1 \models \phi$;
- $\eta \models \phi_1\mathcal{U}\phi_2$ iff there exists $0 \leq i < |\eta|$ such that $\eta_i \models \phi_2$, and for every $0 \leq j < i$ it holds that $\eta_j \models \phi_1$;
- $\eta \models \phi_1\mathcal{R}\phi_2$ iff either for every $0 \leq i < |\eta|$ it holds $\eta_i \models \phi_2$, or there exists $0 \leq i < |\eta|$ such that $\eta_i \models \phi_1$ and for all $0 \leq j \leq i$ it holds $\eta_j \models \phi_2$.

We use $cl(\phi)$ to denote the set of subformulas of ϕ [34], whose formal definition are as follows: (1) $\phi \in cl(\phi)$; (2) $\psi \in cl(\phi)$ if $\phi = op(\psi)$, where op can be $\neg, \mathcal{X}, \mathcal{N}, \Diamond$ or $\Box$; (3) $\psi_1 \in cl(\phi)$ and $\psi_2 \in cl(\phi)$, if ψ_1 op $\psi_2 \in cl(\phi)$, where op can be $\wedge, \vee, \mathcal{U}$ or $\mathcal{R}$. In the rest of the paper, all LTL_f formulas are considered in the *Negated Normal Form* (NNF), i.e., every negation appears only in front of an atom. It is obvious that every LTL_f formula can be converted to the equivalent NNF, and the conversion cost is linear to the size of the formula.

B. Finite Automata

A finite automaton $\mathcal{A}$ is a tuple $(\Sigma, S, \rho, s_0, \Omega)$ where

- Σ is the set of alphabet;
- S is the set of states;
- $\rho : S \times \Sigma \rightarrow 2^S$ is the transition function. We say (s_1, ω, s_2) is a *transition* iff $s_2 \in \rho(s_1, \omega)$;
- $s_0 \in S$ is the initial state;
- Ω is the set of accepting conditions. We call $\mathcal{A}$ is a state-based (resp. transition-based) automaton if $\Omega \subseteq S$ (resp. $\Omega \subseteq \rho$) is the set of accepting states (resp. accepting transitions).

Let $\eta = \omega_0\omega_1 \ldots \in \Sigma^+$ be a finite trace over Σ. A run r of $\mathcal{A}$ on η is a finite state sequence $s_0s_1 \ldots$ such that s_0 is the

initial state and $s_{i+1} \in \rho(s_i, \omega_i)$ for $0 \leq i < |\eta|$. We say η is accepted by $\mathcal{A}$ iff there is a run r of $\mathcal{A}$ on η such that r ends with some accepting conditions in Ω. $\mathcal{A}$ is called deterministic iff $|\rho(s, \omega)| \leq 1$ for every $s \in S$ and $\omega \in \Sigma$; otherwise, $\mathcal{A}$ is nondeterministic.

We consider two different finite automata, i.e., the state-based and transition-based, which vary according to the accepting conditions. In the rest of our paper, a finite automaton is state-based if it is not stated explicitly.

Theorem 1 ([14]). *Given an LTL$_f$ formula ϕ, there is a nondeterministic finite automaton $\mathcal{A}$ such that $\eta \models \phi$ iff η is accepted by $\mathcal{A}$, for a finite trace $\eta \in \Sigma^+$.*

III. Approach Overview

Our approach first constructs the equivalent transition-based NFA (TNFA) for the input formula and then converts the TNFA to its equivalent NFA. A TNFA is a special NFA whose accepting conditions are defined over transitions instead of states. To convert a TNFA to its equivalent NFA, one can create a new state f and add the transition $s \xrightarrow{\omega} f$ into the resulting NFA for every accepting transition $s \xrightarrow{\omega} s'$ in the TNFA. The new state f is the only accepting state in the resulting NFA. Figure 1 illustrates an example of the whole translation process.

Given an LTL$_f$ formula ϕ, the high-level descriptions of how to generate the equivalent TNFA are as follows. We first convert ϕ to its neXt Normal Form (XNF), which can be considered as a propositional formula over the alphabet composed of only Boolean atoms and neXt formulas. For example, the XNF of the formula $\phi = a\mathcal{R}b$, which is denoted as $xnf(\phi)$, is $b \wedge (a \vee \mathcal{X}(a\mathcal{R}b))$. Notably, the conversion cost from an LTL$_f$ formula to its XNF is linear to the size of the formula. Then we utilize $xnf(\phi)^p$ to denote the propositional formula over the alphabet set $\{a, b, \mathcal{X}(a\mathcal{R}b)\}$.

Taking $xnf(\phi)^p$ as the input formula, an SAT solver is able to compute a satisfying assignment A, e.g., $A = \{\neg a, b, \mathcal{X}(a\mathcal{R}b)\}$. The information inside A indicates that there is a transition from ϕ to itself with the label $\neg a \wedge b$. Analogously, the assignment $A = \{a, b, \neg\mathcal{X}(a\mathcal{R}b)\}$ tells that there is a transition from ϕ to $\mathtt{tt}$ (because there is no $\mathcal{X}$ element in A) with the label $a \wedge b$. Now $\mathtt{tt}$ is a new state and we continue the above process to create new transitions until no new state is generated. Finally we identify the accepting transitions, the details of which are shown below.

Constructing the equivalent DFA from an LTL$_f$ formula is similar to constructing NFA. We can also utilize the SAT techniques to obtain a transition-based DFA (TDFA) directly from the input formula. The idea to generate deterministic transitions are as follows. For the given state q, which essentially is a set of subformulas of the input formula and can be considered as a formula as well, we use the SAT solver to get one of the satisfying assignment A of $xnf(q)^p$ at first. From A we can fix the label, denoted as $L(A)$, on the transition and then enumerate all satisfying assignments of q that include $L(A)$. After that, we create a transition starting from q with the label $L(A)$. Recursively applying the above procedure, all the states of the $TDFA$ can be constructed. Finally, we present a methodology to identify the accepting transitions on the generated automaton, whose details are as below.

From the TDFA we can easily construct the equivalent NFA, from which one can use the Subset Construction to create the corresponding DFA.

IV. SAT-Based Automata Construction

In this section, we present the construction from an LTL$_f$ formula to the equivalent (T)NFA and (T)DFA respectively.

A. LTL$_f$-to-(T)NFA Construction

To leverage SAT solvers for the automata construction, we first need to consider LTL$_f$ formulas as propositional formulas. The motivation comes from treating all temporal subformulas of ϕ as atomic propositions.

Definition 1 (Propositional Atoms). *For an LTL$_f$ formula ϕ, we define the set of propositional atoms, $PA(\phi)$, recursively as follows.*

- *If ϕ is an atom, Next, weak Next, Until or Release formula, $PA(\phi) = \{\phi\}$;*
- *If $\phi = (\neg\psi)$, $PA(\phi) = PA(\psi)$;*
- *If $\phi = (\phi_1 \wedge \phi_2)$ or $(\phi_1 \vee \phi_2)$, $PA(\phi) = PA(\phi_1) \cup PA(\phi_2)$.*

Consider the formula $\phi = (a \wedge ((\neg c \wedge a)\mathcal{U}b) \wedge (c \wedge \mathcal{X}(a \vee b)))$ as an example. According to Definition 1, we have $PA(\phi) = \{a, c, (\neg c \wedge a)\mathcal{U}b), (\mathcal{X}(a \vee b))\}$. Now we define the propositional formulas over propositional atoms w.r.t. an LTL$_f$ formula.

Definition 2 (Propositional Formulas). *Given an LTL$_f$ formula ϕ, let ϕ^p be a propositional formula over $PA(\phi)$.*

Consider $\phi = (a \wedge ((\neg c \wedge a)\mathcal{U}b) \wedge (c \wedge \mathcal{X}(a \vee b)))$. From Definition 2, the corresponding propositional formula $\phi^p = (a \wedge p_1) \wedge (c \wedge p_2)$, in which p_1, p_2 are two Boolean variables representing the truth values of $((\neg c \wedge a)\mathcal{U}b)$ and $(c \wedge \mathcal{X}(a \vee b))$. We define the *next normal form* for an LTL$_f$ formula, whose corresponding propositional formula is the actual input to SAT solvers.

Definition 3 (neXt Normal Form). *For an LTL$_f$ formula ϕ, we define its Next Normal Form, i.e., $xnf(\phi)$, recursively as follows.*

- *$xnf(\mathtt{tt}) = \mathtt{tt}$ and $xnf(\mathtt{ff}) = \mathtt{ff}$;*
- *If ϕ is a literal, $xnf(\phi) = \phi \wedge \mathcal{X}(\mathtt{tt})$;*
- *If $\phi = \mathcal{X}(\psi)$ or $\phi = \mathcal{N}(\psi)$, $xnf(\phi) = \mathcal{X}(\psi)$;*
- *If $\phi = (\phi_1 \wedge \phi_2)$, $xnf(\phi) = xnf(\phi_1) \wedge xnf(\phi_2)$;*
- *If $\phi = (\phi_1 \vee \phi_2)$, $xnf(\phi) = xnf(\phi_1) \vee xnf(\phi_2)$;*
- *If $\phi = (\phi_1 \mathcal{U} \phi_2)$, $xnf(\phi) = xnf(\phi_2) \vee (xnf(\phi_1) \wedge \mathcal{X}(\phi))$;*
- *If $\phi = (\phi_1 \mathcal{R} \phi_2)$, $xnf(\phi) = xnf(\phi_2) \wedge (xnf(\phi_1) \vee \mathcal{X}(\phi))$.*

By restricting the LTL$_f$ formula ϕ to the XNF, we can obtain a satisfying assignment of ϕ^p with the help of the SAT solver. For a satisfying assignment A, we define $X(A) = \{\theta \mid \mathcal{X}\theta \in A\}$ and $L(A) = \{l \mid l \in A \text{ is a literal}\}$. Also for

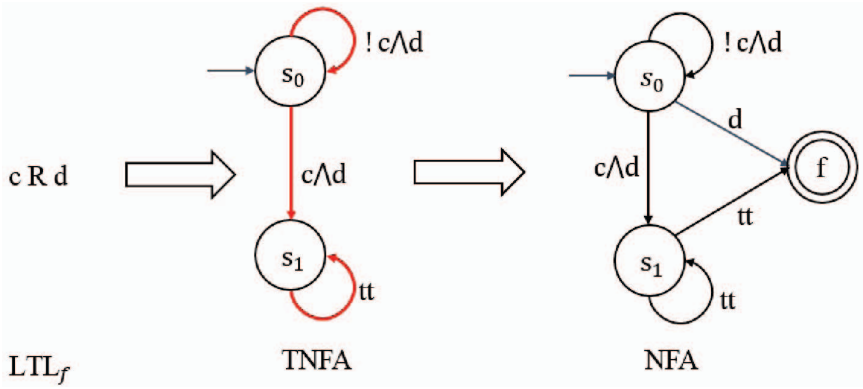

Fig. 1. An example of our approach to construct the NFA from an LTL$_f$ formula. From left to right, the figure displays the input formula ($c\mathcal{R}d$), the equivalent transition-based NFA (TNFA) and the equivalent NFA respectively. The red transitions in the TNFA are accepting transitions.

the sake of simplicity, **we mix the usage of $X(A)$ to denote the conjunction of all elements in the set, i.e., $\bigwedge_{\psi_i \in X(A)} \psi_i$. The same applies to $L(A)$.** The following lemma describes the relationship between an LTL$_f$ formula and the satisfying assignment of its corresponding propositional formula.

Lemma 1. *Given an LTL$_f$ formula ϕ in the XNF and a finite trace $\eta \in \Sigma^+$ with $|\eta| > 1$, $\eta \models \phi$ holds iff there is a satisfying assignment A of ϕ^p such that $\eta[0] \models L(A)$ and $\eta_1 \models X(A)$.*

Proof: ($\Rightarrow$) The proof can be achieved by induction over the types of ϕ. Since ϕ is in XNF, ϕ cannot be an Until or Release formula.

- if ϕ is a literal, we could have a satisfying assignment $A = \{\phi\}$ for ϕ^p. In this situation, $L(A) = \{\phi\}$ and $X(A) = \{\mathtt{tt}\}$. Therefore, it is true that $\eta[0] \models L(A)$ and $\eta_1 \models X(A)$;
- if $\phi = \mathcal{X}\psi$, we could have a satisfying assignment $A = \{\mathcal{X}\psi\}$ for ϕ^p. In this situation, $L(A) = \{\mathtt{tt}\}$ and $X(A) = \{\psi\}$. Considering we have $\eta \models \phi$ i.e., $\eta \models \mathcal{X}\psi$, it is true that $\eta[0] \models L(A)$ and $\eta_1 \models X(A)$;
- if $\phi = \phi_1 \wedge \phi_2$, $\eta \models \phi$ implies $\eta \models \phi_1$ and $\eta \models \phi_2$. By assumption hypothesis, there is A_i of ϕ_i^p $(i = 1, 2)$ such that $\eta[0] \models L(A_i)$ and $\eta_1 \models X(A_i)$. Let $A = A_1 \cup A_2$, and a consistent A, in which either ψ or $\neg\psi$ cannot be together, must exists (A may not be unique because A_1 and A_2 may not be unique). Otherwise, there is $\psi \in A_1$ and $\neg\psi \in A_2$ such that η cannot model $\bigwedge A_1$ and $\bigwedge A_2$ at the same time, which is a contradiction. So A is a satisfying assignment of ϕ^p, which leads to $\eta[0] \models L(A)$ and $\eta_1 \models X(A)$.
- The proof for $\phi = \phi_1 \vee \phi_2$ is analogous to the proof for $\phi = \phi_1 \wedge \phi_2$.

($\Leftarrow$) Let A be a satisfying assignment of ϕ^p, so $A \models \phi^p$ implies $(\bigwedge A) \Rightarrow \phi$. Also, $\eta[0] \models L(A)$ and $\eta_1 \models X(A)$ implies $\eta \models \bigwedge A$. Therefore, it is true that $\eta \models \phi$. □

Lemma 1 does not cover the case when $|\eta| = 1$, which yields the following corollary.

Corollary 1. *Given an LTL$_f$ formula ϕ in the XNF and a finite trace $\eta \in \Sigma^+$ with $|\eta| = 1$, $\eta \models \phi$ holds implies there is a satisfying assignment A of ϕ^p such that $\eta[0] \models L(A)$.*

Proof: The proof is similar to that of Lemma 1 for the ($\Rightarrow$) direction, the details of which are omitted here. □

Informally speaking, if ϕ is a state in the NFA, $X(xnf(\phi)^p)$ represents one of its successors and $L(xnf(\phi)^p)$ represents the label on the transition from ϕ to $X(xnf(\phi)^p)$. Now we present the construction from an LTL$_f$ formula to its equivalent transition-based NFA.

Definition 4 (Transition-based NFA (TNFA))**.** *Given an LTL$_f$ formula ϕ, the corresponding transition-based NFA ${\mathcal{A}_\phi}^{tn}$ is defined as a tuple $(\Sigma, S, \rho, s_0, T)$ such that*

- *$\Sigma = 2^L$ is the set of alphabet, where L is the literal set of ϕ;*
- *$S \subseteq 2^{cl(\phi)}$ is the set of states;*
- *$\rho : S \times \Sigma \to 2^S$ is the transition function, where $s_2 \in \rho(s_1, \omega)(\omega \in \Sigma)$ holds iff there is a satisfying assignment A of $xnf(s_1)^p$ such that $s_2 = X(A)$ and $\omega \models L(A)$;*
- *$s_0 = \{\phi\}$ is the initial state;*
- *$T \subseteq \rho$ is the set of accepting transitions. A transition $s_1 \xrightarrow{\omega} s_2$ is in T iff $\omega \models s_1$ holds.*

As Definition 4 shows, a state in the NFA is a set of subformulas of the input ϕ. For a simple description, **we mix the usage of a state s such that it can also represent a Boolean formula $\bigwedge_{\psi_i \in s} \psi_i$, i.e., the conjunction of all subformulas in s.**

A run r of the transition-based NFA ${\mathcal{A}_\phi}^{tn}$ on η is a finite state sequence $s_0 s_1 \ldots$ such that s_0 is the initial state and $s_{i+1} \in \rho(s_i, \omega_i)$ for $0 \leq i < |\eta|$. η is accepted by ${\mathcal{A}_\phi}^{tn}$ iff there exists a run r of ${\mathcal{A}_\phi}^{tn}$ on η which ends with an accepting transition in T.

Since in Definition 4, the satisfaction of a finite trace with length one of an LTL$_f$ formula is used to determine the accepting conditions, we introduce the following theorem to show that this process can be achieved in a much easier way than the regular satisfaction check in which the finite trace can

have an arbitrary length.

Theorem 2. *Given a finite trace $\eta \in \Sigma^+$ with $|\eta| = 1$, and an LTL$_f$ formula ϕ, it is true that $\eta \models \phi$ holds iff*

- *ϕ is* tt*; or*
- *$\phi \in \eta$ when ϕ is a literal; or*
- *$\phi = \mathcal{N}\psi$ is a weak Next formula; or*
- *$\eta \models \phi_2$ holds when $\phi = \phi_1 \mathcal{U} \phi_2$ is an Until formula; or*
- *$\eta \models \phi_2$ holds when $\phi = \phi_1 \mathcal{R} \phi_2$ is a Release formula; or*
- *$\eta \models \phi_1$ and $\eta \models \phi_2$ hold when $\phi = \phi_1 \wedge \phi_2$; or*
- *$\eta \models \phi_1$ or $\eta \models \phi_2$ holds when $\phi = \phi_1 \vee \phi_2$.*

Proof: The theorem is self-explained which can be derived from the semantics of LTL$_f$. □

We present the following theorem to guarantee the correctness of our construction.

Theorem 3. *Given an LTL$_f$ formula ϕ and the TNFA ${\mathcal{A}_\phi}^{tn}$ constructed by Definition 4, a finite trace $\eta \models \phi$ holds iff η is accepted by ${\mathcal{A}_\phi}^{tn}$.*

Proof: Assume $\eta = \omega_0\omega_1 \dots \omega_n$ $(n \geq 0)$.

($\Leftarrow$) We prove by induction over the values of n.

- According to Definition 4, η is accepted by ${\mathcal{A}_\phi}^{tn}$ implies that there is a run $r = s_0 s_1 \dots s_n s_{n+1}$ of ${\mathcal{A}_\phi}^{tn}$ on η such that $\omega_n \models s_n$. So basically when $k = n$, it holds that $\eta_k \models s_k$;
- Inductively, assume $\eta_k \models s_k$ holds for $0 < k \leq n$. Because $s_{k-1} \xrightarrow{\omega_{k-1}} s_k$ is a transition in the TNFA and $\eta_k \models s_k$ holds, the set $A_{k-1} = \omega_{k-1} \cup \{\mathcal{X}\theta | \theta \in s_k\}$ is a satisfying assignment of $xnf(s_{k-1})^p$, from Definition 4. Then based on Lemma 1 and since $\eta_{k-1} = \omega_{k-1} \cdot \eta_k$ is true, we have that $\eta_{k-1} \models s_{k-1}$.

When $k = 0$, we prove that $\eta(= \eta_0) \models \phi(= s_0)$ is true.

($\Rightarrow$) We first prove that $\eta \models \phi$ implies there is a run r of ${\mathcal{A}_\phi}^{tn}$ on η. If $n = 0$, Corollary 1 shows that there is a satisfying assignment A of $xnf(\phi)^p$ such that $\eta[0] \models L(A)$. From Definition 4, $\phi \xrightarrow{\eta[0]} X(A)$ is a transition of the TNFA. As a result, there is a run $r = s_0(= \phi)s_1(= X(A))$ of ${\mathcal{A}_\phi}^{tn}$ on η.

If $n > 0$, according to Lemma 1, $\eta \models \phi$ implies there is a satisfying assignment A of $xnf(\phi)^p$ such that $\omega_0 \models L(A)$ and $\eta_1 \models X(A)$. From Definition 4, $\phi(= s_0) \xrightarrow{\omega_0} X(A)(= s_1)$ is a transition in the TNFA. Recursively applying the above process to $\eta_i \models s_i (i \geq 1)$, one can prove there is a transition $s_i \xrightarrow{\omega_i} s_{i+1}$ in the TNFA. As a result, we have $\eta \models \phi$ implies there is a run r of ${\mathcal{A}_\phi}^{tn}$ on η.

Now we prove that the run r is an accepting run. If $n = 0$, the run r is $s_0 s_1$ such that $\eta[0] \models s_0(= \phi)$ holds. According to Definition 4, the transition $s_0 \xrightarrow{\eta[0]} s_1$ is an accepting transition. As a result, r is an accepting run and η is accepted by ${\mathcal{A}_\phi}^{tn}$.

If $n > 0$, we can also prove that for the run $r = s_0 s_1 \dots s_n s_{n+1}$, $s_n \xrightarrow{\omega_n} s_{n+1}$ is an accepting transition, due to the fact that $\omega_n \models s_n$. The proof is done. □

We discuss the upper bound of the generated TNFA in the following theorem.

Theorem 4. *For an LTL$_f$ formula ϕ, the size of the corresponding TNFA ${\mathcal{A}_\phi}^{tn}$ generated by Definition 4 is at most $2^{|cl(\phi)|}$, where $cl(\phi)$ is the set of subformulas of ϕ.*

Proof: The proof is straightforward as every state in ${\mathcal{A}_\phi}^{tn}$ is from $2^{cl(\phi)}$, according to Definition 4. □

We have shown the construction from an LTL$_f$ formula to its equivalent TNFA. The following theorem indicates that there is a trivial way to convert a TNFA to its equivalent NFA.

Theorem 5. *For a TNFA $\mathcal{A}^{tn}$, there is an equivalent NFA $\mathcal{A}^n$ such that η is accepted by $\mathcal{A}^{tn}$ iff η is accepted by $\mathcal{A}^n$, for $\eta \in \Sigma^+$.*

Proof: Assume $\mathcal{A}^{tn} = (\Sigma, S, \rho, s_0, T)$ and one can construct the NFA $\mathcal{A}^n = (\Sigma, S', \rho', s_0, F)$ such that

- $S' = S \cup \{f\}$ where f is a new and only accepting state in $\mathcal{A}^n$;
- $\rho' = \rho \cup \{s_1 \times \omega \rightarrow f |\ s_1 \times \omega \rightarrow s_2 \in T\}$;
- $F = \{f\}$.

We prove the equivalence of $\mathcal{A}^{tn}$ and $\mathcal{A}^n$ as follows. On one hand, every accepting run of $\mathcal{A}^{tn}$ which ends with an accepting transition in T can also be an accepting run of $\mathcal{A}^n$ which ends with the accepting state f. On the other hand, every accepting run of $\mathcal{A}^n$ which ends with the accepting state f can also be an accepting run of $\mathcal{A}^{tn}$ which ends with an accepting transition in T. □

B. LTL$_f$-to-(T)DFA Construction

Based on the NFA construction in the previous section, the equivalent DFA w.r.t the LTL$_f$ formula may be obtained by applying the Subset Construction to the generated NFA. However, such DFA construction is not on the fly, as it has to be postponed until the NFA is fully generated. Inspired from Definition 4, which enables to construct the TNFA on the fly, we present here an on-the-fly construction directly from the LTL$_f$ formula to the equivalent transition-based DFA (TDFA).

Let ϕ be a propositional formula and we utilize $AllSat(\phi)$ to represent the set of all satisfying assignments of ϕ. Given a literal set L, we also define the projection of $AllSat(\phi)$ under L, i.e., $AllSat(\phi)_{|L}$, which is a subset of $AllSat(\phi)$ such that $A \supseteq L$ iff $A \in AllSat(\phi)_{|L}$. Informally speaking, $AllSat(\phi)_{|L}$ represents exactly the set of satisfying assignments of ϕ including L. Now we are ready to present the TDFA construction.

Definition 5 (Transition-based DFA)**.** *Given an LTL$_f$ formula ϕ, the corresponding transition-based DFA ${\mathcal{A}_\phi}^{td}$ is defined as a tuple $(\Sigma, Q, \delta, q_0, T^d)$ such that*

- *$\Sigma = 2^L$ is the set of alphabet, where L is the literal set of ϕ;*
- *$Q \subseteq 2^{2^{cl(\phi)}}$ is the set of states;*
- *$\delta : Q \times \Sigma \rightarrow Q$ is the transition function, where $q_2 = \delta(q_1, \omega)(\omega \in \Sigma)$ holds iff $q_2 = \{X(A) | A \in AllSat(xnf(q_1)^p)_{|\omega}\}$;*

- $q_0 = \{\{\phi\}\}$ *is the initial state;*
- $T^d \subseteq \delta$ *is the set of accepting transitions. A transition* $q_1 \xrightarrow{\omega} q_2$ *is in* T^d *iff there is a* $s \in q_1$ *such that* $\omega \models s$ *holds.*

Analogous to the state representation of the TNFA, we mix the usage of a state q in the TDFA such that q also represents the LTL$_f$ formula $\bigvee_{s\in q}\bigwedge_{\psi\in s}\psi$.

A run r of the transition-based DFA ${\mathcal{A}_\phi}^{td}$ on η is a finite state sequence $q_0q_1\ldots$ such that q_0 is the initial state and $q_{i+1} = \delta(q_i, \omega_i)$ for $0 \leq i < |\eta|$. Since ${\mathcal{A}_\phi}^{td}$ is deterministic, the run r on η is unique. η is accepted by ${\mathcal{A}_\phi}^{td}$ iff the only run r of ${\mathcal{A}_\phi}^{tn}$ on η ends with some accepting transitions in T^d. The following theorem guarantees the correctness of the TDFA construction shown in Definition 5.

Theorem 6. *Given an LTL$_f$ formula ϕ and the TDFA ${\mathcal{A}_\phi}^{td}$ constructed by Definition 5, a finite trace $\eta \models \phi$ holds iff η is accepted by ${\mathcal{A}_\phi}^{td}$.*

Proof: Assume ${\mathcal{A}_\phi}^{tn}$ is the TNFA constructed by Definition 4 w.r.t ϕ. Intuitively, ${\mathcal{A}_\phi}^{td}$ from Definition 5 is equivalent to the TDFA constructed by applying the Subset Construction to ${\mathcal{A}_\phi}^{tn}$. However, the Subset Construction is not standardized for transition-based finite automata such that the above intuition is not straightforward. Instead, we follow the proof of Theorem 3 and prove this theorem analogously as follows.

Assume $\eta = \omega_0\omega_1\ldots\omega_n$ $(n \geq 0)$.

($\Leftarrow$) According to Definition 5, η accepted by ${\mathcal{A}_\phi}^{td}$ implies that there is a run $r = q_0q_1\ldots q_nq_{n+1}$ of ${\mathcal{A}_\phi}^{td}$ on η such that $q_n \xrightarrow{\omega_n} q_{n+1} \in \delta$ and $\omega_n \models s$ for some $s \in q_n$. To prove η is accepted by ${\mathcal{A}_\phi}^{td}$ implies $\eta \models \phi$ holds, we only need to prove η is accepted by ${\mathcal{A}_\phi}^{td}$ implies $\eta_{n-i} \models q_{n-i}$ for i=0...n. Then we could have $\eta(=\eta_0) \models \phi(=q_0)$ when $i = n$.

We prove by induction over the values of i.

- By Definition 5, we have that $\omega_n \models s$ for some $s \in q_n$. Since $q_n = \bigvee_{s\in q_n}\bigwedge_{\psi\in s}\psi$, it is true that $s \implies q_n$ for every $s \in q_n$. So $\omega_n \models s$ implies $\omega_n \models q_n$, which means basically when $i = 0$, it holds that $\eta_{n-0} \models q_{n-0}$;
- Inductively, assume $\eta_{n-i} \models q_{n-i}$ holds by satisfying $\eta_{n-i} \models s$ for some $s \in q_{n-i}$ $(0 < i \leq n)$. Because $q_{n-i-1} \xrightarrow{\omega_{n-i-1}} q_{n-i}$ is a transition in the TDFA and $\eta_{n-i} \models q_{n-i}$ holds, the set $A_{n-i-1} = \omega_{n-i-1} \cup \{\mathcal{X}\theta | \theta \in s\}$ is a satisfying assignment of $xnf(q_{n-i-1})^p$ in $AllSat(xnf(q_{n-i-1})^p)_{|\omega_{n-i-1}}$, from Definition 5. Then based on Lemma 1 and since $\eta_{n-i-1} = \omega_{n-i-1} \cdot \eta_{n-i}$ is true, we have that $\eta_{n-i-1} \models q_{n-i-1}$.

($\Rightarrow$) We first prove that $\eta \models \phi$ implies that there is a run r of ${\mathcal{A}_\phi}^{td}$ on η. If $n = 0$, Corollary 1 shows that there is a satisfying assignment A of $xnf(\phi)^p$ such that $\eta[0] \models L(A)$. From Definition 5, $\phi \xrightarrow{\eta[0]} q$, where $q = \{X(A') | A' \in AllSat(xnf(\phi)^p)_{|L(A)}\}$, is a transition of the TDFA. As a result, there is a run $r = q_0(=\phi)q$ of ${\mathcal{A}_\phi}^{td}$ on η.

If $n > 0$, according to Lemma 1, $\eta \models \phi$ implies there is a satisfying assignment A of $xnf(\phi)^p$ such that $\omega_0 \models L(A)$ and $\eta_1 \models X(A)$. From Definition 5, $\phi(=q_0) \xrightarrow{\omega_0} q_1$, where $q_1 = \{X(A') | A' \in AllSat(xnf(\phi)^p)_{|\omega_0}\}$, is a transition in the TDFA. Recursively applying the above process to $\eta_i \models q_i (i \geq 1)$, one can prove there is a transition $q_i \xrightarrow{\omega_i} q_{i+1}$ in the TDFA. As a result, we have $\eta \models \phi$, which implies there is a run r of ${\mathcal{A}_\phi}^{td}$ on η.

Now we have that $\eta \models \phi$ implies there is a run $r = q_0...q_nq_{n+1}$ of ${\mathcal{A}_\phi}^{td}$ on η and $\eta_i \models q_i$ for $i = 0...n$. When $i = n$, $\eta_n \models q_n$ i.e., $\omega_n \models q_n$ holds. According to Definition 5, the transition $q_n \xrightarrow{\omega_n} q_{n+1}$ is an accepting transition. As a result, r is an accepting run and η is accepted by ${\mathcal{A}_\phi}^{td}$. □

We now discuss the upper bound of the generated TDFA in the following theorem.

Theorem 7. *For an LTL$_f$ formula ϕ, the size of the corresponding $TDFA$ ${\mathcal{A}_\phi}^{td}$ generated by Definition 5 is at most* $2^{2^{|cl(\phi)|}}$.

Proof: The proof is straightforward as every state in ${\mathcal{A}_\phi}^{td}$ is from $2^{2^{cl(\phi)}}$, according to Definition 5. □

It should be noted that, the conversion from a TNFA to its equivalent NFA shown in Theorem 5 is not applicable to that from a TDFA to its equivalent DFA. In fact, the resulting automaton can only be an NFA. One still has to apply the Subset Construction to the NFA for the equivalent DFA. However, we argue that for real application purposes, such as LTL$_f$ synthesis that requires DFA, working on TDFA has the same or even better performance. Currently, this is out of this paper's scope.

Algorithm 1: Construction of the TNFA

Input: LTL$_f$ formula ϕ
Output: TNFA ${\mathcal{A}_\phi}^{tn} = (\Sigma, S, \rho, s_0, T)$

$\Sigma := 2^L$;
$s_0 := \{\phi\}$;
$S := \{s_0\}$;
$\rho := \emptyset$;
$T := \emptyset$;
$to\_process := \{s_0\}$;
while $to\_process \neq \emptyset$ **do**
  get a state s from $to\_process$;
  $\psi := xnf(s)^p$;
  while ψ *is satisfiable* **do**
    get a model A of ψ;
    $label := L(A)$;
    $next := X(A)$;
    if $next \notin S$ **then**
      add $next$ into S;
      add $next$ into $to\_process$;
    add $s \times label \rightarrow next$ into ρ;
    if $label \models s$ **then**
      add $s \times label \rightarrow next$ into T;
    $\psi := \psi \wedge (\neg(\bigwedge A))$;
  remove s from $to\_process$;
return $(\Sigma, S, \rho, s_0, T)$;

V. Implementation

For each LTL$_f$ formula, there are (T)NFA and DFA that accept the same language as the formula. In this section, we first give the algorithms for constructing TNFA from the LTL$_f$ formula and converting TNFA to NFA. Then we introduce the algorithm for constructing TDFA directly from a given LTL$_f$ formula. All algorithms are presented in pseudo-code.

A. LTL$_f$-to-(T)NFA

We define the construction of the TNFA from a given LTL$_f$ formula as shown in Algorithm 1.

To get the automaton, we need to solve all the successor states and related transitions of each state. In the algorithm, $to\_process$ is the set of states whose successor states have not yet been calculated. Each time the first layer of the while loop (Line 7) is executed, the successor states and transitions of a state will be calculated. After a state is processed, it will be removed from $to\_process$. State s corresponds to the formula $xnf(s)^p$, and one model of $xnf(s)^p$ corresponds to one transition and one successor. When we have a model A of $xnf(s)^p$ by the SAT solver, we can get $label \coloneqq L(A)$ as the label of the transition and $next \coloneqq X(A)$ as the successor. If the state corresponding to $next$ appears for the first time, it will be added into S and $to\_process$. The transition $s \times label \to next$ should be added into ρ. Then we check whether the transition is an accepting transition. If $label \models s$, we add it into T. We update ψ to be $\psi \wedge (\neg(\bigwedge A))$ in order to avoid duplicate model of $xnf(s)^p$. The algorithm terminates when $to\_process$ is empty, which means that all states and their transitions and successors are solved.

After constructing the TNFA, we introduce the conversion from TNFA to NFA in Algorithm 2.

Algorithm 2: Conversion from TNFA to NFA

Input: TNFA ${\mathcal{A}_\phi}^{tn} = (\Sigma, S, \rho, s_0, T)$
Output: NFA ${\mathcal{A}_\phi}^{n} = (\Sigma, S', \rho', s_0, F)$

1 $\Sigma \coloneqq 2^L$;
2 $s_0 \coloneqq \{\phi\}$;
3 $S' \coloneqq S \cup \{f\}$;
4 $\rho' \coloneqq \rho$;
5 $F \coloneqq \{f\}$;
6 **for** $s_1 \times \omega \to s_2 \in T$ **do**
7 **add** $s_1 \times \omega \to f$ into ρ';
8 **return** $(\Sigma, S', \rho', s_0, F)$;

The algorithm converts the TNFA to NFA by a few modifications to the original automaton. Compared with the TNFA, we add an accepting state f and a new transition $s_1 \times \omega \to f$ for every accepting transition $s_1 \times \omega \to s_2 \in T$.

B. LTL$_f$-to-TDFA

As shown in Algorithm 3, we can construct a TDFA from an LTL$_f$ formula ϕ on the fly.

Algorithm 3: Construction of the TDFA

Input: LTL$_f$ formula ϕ
Output: TDFA ${\mathcal{A}_\phi}^{td} = (\Sigma, Q, \delta, q_0, T^d)$

1 $\Sigma \coloneqq 2^L$;
2 $Q \coloneqq \{q_0\}$;
3 $\delta \coloneqq \emptyset$;
4 $q_0 \coloneqq \{\{\phi\}\}$;
5 $T^d \coloneqq \emptyset$;
6 $to\_process \coloneqq \{q_0\}$;
7 **while** $to\_process \neq \emptyset$ **do**
8 **get** a state q from $to\_process$;
9 $\psi \coloneqq xnf(q)^p$;
10 **let** $label\_next$ be a **map**;
11 **while** ψ *is satisfiable* **do**
12 **get** a model A of ψ;
13 $label \coloneqq L(A)$;
14 $next \coloneqq X(A)$;
15 **if** $label\_next[label] = NULL$ **then**
16 $label\_next[label] \coloneqq \{next\}$;
17 **else**
18 $label\_next[label] \coloneqq label\_next[label] \cup \{next\}$;
19 $\psi \coloneqq \psi \wedge (\neg(\bigwedge A))$;
20 **for** $(Label, Next)$ *in* $label\_next$ **do**
21 **if** $Next \notin Q$ **then**
22 **add** $Next$ into Q;
23 **add** $Next$ into $to\_process$;
24 **add** $q \times Label \to Next$ into δ;
25 **if** $Label \models q$ **then**
26 **add** $q \times Label \to Next$ into T^d;
27 **remove** q from $to\_process$;
28 **return** $(\Sigma, Q, \delta, q_0, T^d)$;

Obviously, Algorithm 3 is developed from Definition 5. In the algorithm, the function of $to\_process$ is the same as in Algorithm 1. Each time the while loop in the first level (Line 7) is executed, a state is taken from $to\_process$ to solve all its successor states, and new states that have not appeared are added to Q and $to\_process$. After solving the successor of a state, it is removed from $to\_process$. When we are to get the successor of a state q, we process all models of the $xnf(q)^p$. The map $label\_next$ is used to integrate the $X(A)$ part of the models with the same $X(A)$ (Line 16-18). Then the for loop (Line 20) adds new states into Q and $to\_process$, and adds the transition in $label\_next$ into δ and T^d (if accepting condition holds).

VI. Experiments

In this section, we present the results of our experimental evaluation.

TABLE I
COMPARISON ON THE NUMBER OF STATES FOR GENERATED AUTOMATA.

NUMBER of FORMULAS[a]	MONA	SPOT	AALTAF
36	0	4	2
77	0	4	4
18	0	5	2
18	0	5	4
82	4	4	2
82	5	4	2
123	5	4	4
82	5	5	2
82	5	5	4
600[b]	451	600	600

[a]The first column categorizes our benchmarks according to the number of states.
[b]The last row summarizes the number of formulas for which each tool generates automata successfully.

A. Experimental Setup

Tools We implemented the on-the-fly construction from LTL_f-to-TNFA using aaltaf [26]. Then we compared the performance on the LTL_f-to-Automata construction among three tools, i.e., MONA, SPOT and aaltaf. Each tool is able to generate an equivalent automaton corresponding to the input LTL_f formula.

Benchmarks We conducted the comparison of MONA, SPOT and aaltaf in the context of LTL_f-to-Automata construction, therefore only satisfiable formulas are the candidate benchmarks. For that purpose, we first checked the satisfiability of all the LTL_f formulas and ruled out those unsatisfiable ones. In summary, we collected a sum of 600 LTL-as-LTL_f formulas from [35] as our benchmarks, taking into the fact that LTL_f and LTL have the same syntax [29].

Platform To explore the difference between MONA, SPOT and aaltaf on the LTL_f-to-Automata construction, we ran each formula on Ubuntu 20.04 with 4 processor cores, 3.2 GHz and 8GB of RAM. The timeout was set to be 120 seconds for each tested formula. Instances that cannot generate the corresponding automata within timeout will receive a penalty of 120 seconds.

Comparison Criteria As introduced before, aaltaf, MONA and SPOT generate different kinds of automata, i.e., aaltaf generates the TNFA, while MONA and SPOT generate the NFA. As a result, we evaluate the performance in terms of the number of states, transitions and the time consuming, which are the three general criteria for automata construction.

B. Experimental Results

We first evaluate each tool in terms of the number of generated states. Table I shows the results for the LTL_f-to-Automata translation on the LTL-as-LTL_f benchmarks. We select the data of Row 2 for explanation and others are analogous. There are 36 instances on which MONA failed to complete the automata construction. Meanwhile, SPOT successfully generated the corresponding automata with each having 4 states for the whole 36 cases. At the same time,

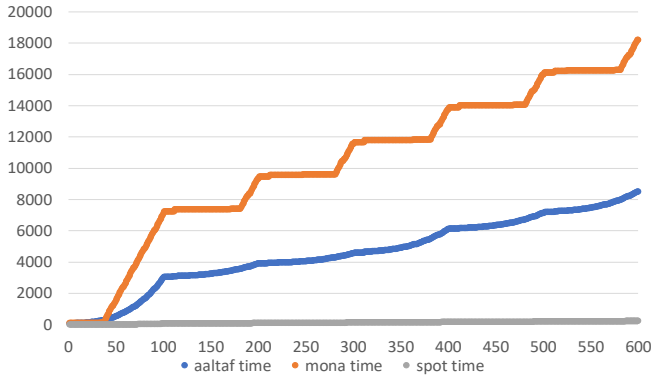

Fig. 2. Time Comparison among different tools.

our tool aaltaf successfully finished the automata construction with 2 states for each automaton, which is the least number among all three tools. The last row summarizes the number of each tool-generated automata. Our benchmarks have a total amount of 600 formulas. MONA succeeded in transforming 451 formulas. SPOT successfully transformed 600, which is the same with aaltaf.

As shown in Table I, column 2 denotes the number of states of automata generated by MONA. There are 149 instances that MONA failed to output corresponding automata. For these cases, MONA responds with "token too large, exceed YYLMAX", which means that the scale of BDD is too large to handle for MONA. The reason is, MONA utilizes BDD packages to compose the structure of DFA. Unlike MONA, SPOT and aaltaf successfully completed all 600 instances for the LTL_f-to-Automata construction. One can see that aaltaf produces fewer or as many states as the other two tools for the whole 600 benchmarks.

In addition to the comparison of states, we conducted the comparison of transitions among those tool-generated automata. Table II shows the experimental results for our comparison on the number of transitions among three tools.

TABLE II
COMPARISON ON THE NUMBER OF TRANSITIONS FOR THE GENERATED AUTOMATA.

NUMBER of FORMULAS[d]	MONA	SPOT	AALTAF
73	A[a]	A	A
4	A	A	B[b]
12	A	B	A
7	B	A	A
12	B	A	B
8	B	B	A
186	B	B	B
8	B	B	C[c]
33	B	C	B
27	C	B	B
33	C	B	C
16	C	C	B
32	C	C	C
90	0	C	C
25	0	B	C
34	0	C	B
600	451	600	600

[a]A represents the range [1,500), which means the number is greater than or equal to 1, and less than 500.
[b]B represents the range [500, 2000), which means the number is greater than or equal to 500, and less than 2000.
[c]C represents the range [2000, 7000), which means the number is greater than or equal to 2000, and less than 7000.
[d]The first column categorizes our benchmarks according to the number of transitions for the generated automata.

Here we select the data of Row 2 for explanation and others are analogous. There are 73 instances for which each tool successfully generated the corresponding automata. To be more precise, the number of transitions of each generated automaton is in the range of 1 to 500. In total, aaltaf has a number of 518 instances that produce fewer or as many transitions as the other two tools. For SPOT and MONA, the numbers are 497 and 348 respectively. The results show that our tool aaltaf is competitive to SPOT and MONA on the NFA construction.

Finally, we present the time cost for each tool on each instance, which is shown in Fig. 2. SPOT spent the least to compete the benchmarks, followed by aaltaf and MONA in order. It is not surprised to see MONA cost the most, as its main advantage is to generate DFA rather than NFA. Also, SPOT outperforms aaltaf on the time cost is mainly because SPOT is highly optimized after decades' development. Compared to that, the automata construction inside aaltaf is just a prototype so far and many heuristics can be applied to improve the performance. We leave it as the future work.

VII. DISCUSSION AND CONCLUSION

As far as we know, there are two approaches to convert an LTL_f formula to its equivalent finite automaton. The first one is to generate the Büchi automaton using SPOT and then convert it to the corresponding NFA. Since SPOT only supports the translation of automata from LTL formulas that are over infinite traces, one should first rewrite the input LTL_f formula to its equi-satisfiable LTL formula, in which a new proposition *alive* should be introduced to identify the LTL_f-semantic satisfaction under the LTL semantics. After that, the equivalent Büchi automaton w.r.t. the LTL formula can be obtained via SPOT, whose transitions are labeled with transitions containing either *alive* or $\neg$*alive*. Finally, by deleting all transitions with $\neg$*alive* labeled, removing the *alive* proposition on the other transitions and simplifying the left parts, one is able to obtain the finite NFA. SPOT utilizes the BDD techniques to achieve the automata translation. An illustrative example is shown at the website of SPOT[2].

The second approach converts the input LTL_f formula to the monadic First-Order Logic (FOL) and then leverages the MONA tool [17] to construct the automata [36]. The correctness guarantee underlies this approach replies on the fact that there is an equivalent monadic FOL formula for an arbitrary LTL_f formula. The generated automaton is represented by the shared, multi-terminal BDD.

Our approach introduced in this paper differs from the above two methods as follows. (1) The translations from an LTL_f formula to its equivalent TNFA and TDFA are straightforward without involving any intermediate processes. (2) The resulting automata of our approach are transition-based, which has been shown more efficient than the state-based ones from previous works [22]. (3) Most importantly, our approach leverages the SAT instead of the BDD techniques for automata construction, which makes it potentially more useful for on-the-fly applications such as planning (via synthesis). Actually, this

[2]https://spot.lrde.epita.fr/tut12.html

paper is considered as the pre-processing work for LTL_f on-the-fly synthesis in our future work, which focuses on the task of establishing the fundamental part to construct the automata on the fly. The preliminary results provided in the previous section shows that in general the cost of TNFA construction should not be heavier than the other two automata-constructing approaches.

To conclude, we present the direct translations from an LTL_f formula to its equivalent TNFA and TDFA respectively. The translations leverage the SAT techniques such that the automata can be constructed on the fly. As far as we know, this is the first work that proposes to generate the transition-based finite automata for the LTL_f formulas, and we will evaluate the corresponding advantages in the real applications such as planning (via synthesis) in our future work.

ACKNOWLEDGMENT

We thank anonymous reviewers for their useful comments. This work was supported in part by the Project of Science and Technology Commitment of Shanghai (Grant #19511103602) and National Natural Science Foundation of China (Grant #62002118).

REFERENCES

[1] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[2] F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[3] Luciano Baresi, Pourhashem Kallehbasti, Mohammad Mehdi, and Matteo Rossi. Efficient scalable verification of LTL specifications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE 15, pages 711–721. IEEE Press, 2015.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4), September 2011.

[5] M. Bienvenu, C. Fritz, and S. McIlraith. Planning with qualitative temporal preferences. In *KR*, pages 134–144, Lake District, UK, June 2006.

[6] M. Bienvenu, C. Fritz, and S. A. McIlraith. Specifying and computing preferred plans. *Artificial Intelligence*, 175:1308 – 1345, 2011.

[7] Aaron Bohy, Emmanuel Filiot, and Naiyong Jin. Acacia+, a tool for ltl synthesis. In *of Lecture Notes in Computer Science*, pages 652–657. Springer-Verlag, 2012.

[8] D. Calvanese, G. De Giacomo, and M.Y. Vardi. Reasoning about actions and planning in LTL action theories. In *Principles of Knowledge Representation and Reasoning*, pages 593–602. Morgan Kaufmann, 2002.

[9] A. Camacho, J. A. Baier, C. Muise, and S. A. McIlraith. Finite ltl synthesis as planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, pages 29–38. ICAPS, 2018.

[10] A. Camacho, J.A. Baier, C. Muise, and A.S. McIlraith. Bridging the gap between LTL synthesis and automated planning. Technical report, U. Toronto, 2017.

[11] A. Camacho, C. Muise, J. Baier, and S. McIlraith. Synkit: Ltl synthesis as a service. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI 18, pages 5817–5819. AAAI Press, 2018.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[13] G. De Giacomo, R. De Masellis, and M. Montali. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *AAAI*, pages 1027–1033, 2014.

[14] G. De Giacomo and M. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 2000–2007. AAAI Press, 2013.

[15] G. De Giacomo and M.Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *Proc. European Conf. on Planning*, Lecture Notes in AI 1809, pages 226–238. Springer, 1999.

[16] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized büchi automata. In *Proc. 12th Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 76–83. IEEE Computer Society, 2004.

[17] J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 516–520. Springer, 1998.

[18] V. Fionda and G. Greco. The complexity of LTL on finite traces: Hard and easy fragments. In *AAAI*, pages 971–977. AAAI Press, 2016.

[19] A. Gabaldon. Precondition control and the progression algorithm. In *KR*, pages 634–643. AAAI Press, 2004.

[20] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[21] G. De Giacomo and M. Y. Vardi. Synthesis for ltl and ldl on finite traces. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI 15, pages 1558–1564. AAAI Press, 2015.

[22] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to büchi automata. In *Proc. 22nd IFIP Int'l Conf. on Formal Techniques for Networked and Distributed Systems*, pages 308–326, 2002.

[23] J.G. Henriksen, J.L. Jensen, M.E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. 1st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[24] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Proc. 19th Int. Conf. on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 258–262, 2007.

[25] J. Li, K. Y. Rozier, G. Pu, Y. Zhang, and M. Y. Vardi. Sat-based explicit ltlf satisfiability checking. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pages 2946–2953. AAAI Press, 2019.

[26] J. Li, Y. Yao, G. Pu, L. Zhang, and J. He. Aalta: an LTL satisfiability checker over infinite/finite traces. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pages 731–734, 2014.

[27] J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL_f satisfiability checking. In *ECAI*, pages 91–98, 2014.

[28] F. Patrizi, N. Lipoveztky, G. De Giacomo, and H. Geffner. Computing infinite plans for LTL goals using a classical planner. In *IJCAI*, pages 2003–2008. AAAI Press, 2011.

[29] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[30] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:115–125, 1959.

[31] S. Sohrabi, J. A. Baier, and S. A. McIlraith. Preferred explanations: Theory and generation via planning. In *AAAI*, pages 261–267, August 2011.

[32] L. Tabajara and M. Vardi. Partitioning techniques in ltlf synthesis. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI 19, pages 5599–5606. AAAI Press, 2019.

[33] B.A. Trakhtenbrot. Finite automata and monadic second order logic. *Siberian Math. J*, 3:101–131, 1962. Russian; English translation in: AMS Transl. 59 (1966), 23-55.

[34] P. Wolper. Temporal logic can be more expressive. In *Proc. 22nd IEEE Symp. on Foundations of Computer Science*, pages 340–348, 1981.

[35] S. Zhu, G. Pu, and M. Vardi. First-order vs. second-order encodings for LTL_f-to-automata translation. In *Theory and Applications of Models of Computation - 15th Annual Conference*, TAMC 2019, pages 684–705. Springer, 2019.

[36] S. Zhu, L. Tabajara, J. Li, G. Pu, and M. Vardi. Symbolic ltlf synthesis. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI 17, pages 1362–1369. AAAI Press, 2017.