Artificial Intelligence 289 (2020) 103369

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

SAT-based explicit LTL_f satisfiability checking

Jianwen Li [a,*,1], Geguang Pu [a,*], Yueling Zhang [a], Moshe Y. Vardi [b], Kristin Y. Rozier [c,*]

[a] East China Normal University, China
[b] Rice University, United States of America
[c] Iowa State University, United States of America

ARTICLE INFO

Article history:
Received 29 April 2019
Received in revised form 12 June 2020
Accepted 15 August 2020
Available online 18 August 2020

Keywords:
LTL over finite traces
Satisfiability checking
SAT-based satisfiability checking
Conflict-driven satisfiability checking

ABSTRACT

Linear Temporal Logic over finite traces (LTL_f) was proposed in 2013 and has attracted increasing interest around the AI community. Though the theoretic basis for LTL_f has been thoroughly explored since that time, there are still few algorithmic tools that are able to provide an efficient reasoning strategy for LTL_f. In this paper, we present a SAT-based framework for LTL_f satisfiability checking, which is the foundation of LTL_f reasoning. We use propositional SAT-solving techniques to construct a transition system, which is an automata-style structure, for an input LTL_f formula; satisfiability checking is then reduced to a path-search problem over this transition system. Based on this framework, we further present CDLSC (Conflict-Driven LTL_f Satisfiability Checking), a novel algorithm (heuristic) that leverages information produced by propositional SAT solvers, utilizing both satisfiability and unsatisfiability results. More specifically, the satisfiable results of the SAT solver are used to create new states of the transition system and the unsatisfiable results to accelerate the path search over the system. We evaluate all 5 off-the-shelf LTL_f satisfiability algorithms against our new approach CDLSC. Based on a comprehensive evaluation over 4 different LTL_f benchmark suits with a total amount of 9317 formulas, our time-cost analysis shows that 1) CDLSC performs best on checking unsatisfiable formulas by achieving approximately a 4X time speedup, compared to the second-best solution (K-LIVE [1]); 2) Although no approaches dominate checking satisfiable formulas, CDLSC performs best on 2 of the total 4 tested satisfiable benchmark suits; and 3) CDLSC gains the best overall performance when considering both satisfiable and unsatisfiable instances.

1. Introduction

Linear Temporal Logic over Finite Traces, or LTL_f, is a specification language gaining popularity in the AI community for formalizing and validating system behaviors. While standard Linear Temporal Logic (LTL) is interpreted on infinite traces, LTL_f is interpreted over finite traces [2]. While LTL is typically used in formal-verification settings, where we are interested in nonterminating computations, cf. [3,4],[2] LTL_f is more attractive in AI scenarios focusing on finite behaviors, such as planning [7–11], plan constraints [12,13], and user preferences [14–16]. Due to the wide spectrum of applications of LTL_f

* Corresponding authors.
E-mail addresses: lijwen2748@gmail.com (J. Li), ggpu@sei.ecnu.edu.cn (G. Pu), kyrozier@iastate.edu (K.Y. Rozier).
[1] Part of this work is finished at Iowa State University.
[2] Several works on verification also studied LTL over finite traces [5,6].

https://doi.org/10.1016/j.artint.2020.103369
0004-3702/

in the AI community [17,29], it is worthwhile to study and develop an efficient framework for solving LTL_f-reasoning problems. Just as propositional satisfiability checking is one of the most fundamental propositional reasoning tasks, LTL_f satisfiability checking is a fundamental task for LTL_f reasoning.

Given an LTL_f formula, the satisfiability problem asks whether there is a finite trace that satisfies the formula. A "classical" solution to this problem is to reduce it to the LTL satisfiability problem [2]. The advantage of this approach is that the LTL satisfiability problem has been studied for decades, and many mature tools are available, cf. [18–20]. Thus, LTL_f satisfiability checking can benefit from progress in LTL satisfiability checking. There is, however, an inherent drawback that an extra cost has to be paid when checking LTL formulas, as the tool searches for a "lasso" (a lasso consists of a finite path plus a cycle, representing an infinite trace), whereas models of LTL_f formulas are just finite traces. Based on this motivation, [21] presented a tableau-style algorithm for LTL_f satisfiability checking. They showed that the dedicated tool, *Aalta-finite*, which conducts an explicit-state search for a satisfying trace, outperforms extant tools for LTL_f satisfiability checking.

The conclusion of a dedicated solver being superior to LTL_f satisfiability checking from [21], seems to be out of date by now because of the recent dramatic improvement in propositional SAT solving, cf. [22]. On one hand, SAT-based techniques have led to a significant improvement on LTL satisfiability checking, outperforming the tableau-based techniques of *Aalta-finite* [21]. On the other hand, SAT-based techniques are now dominant in symbolic model checking [23,24]. Our preliminary evaluation indicates that LTL_f satisfiability checking via SAT-based model checking [25,26] or via SAT-based LTL satisfiability checking [27] both outperform the tableau-based tool *Aalta-finite*. Also, a recent LTL_f satisfiability checker *LTL2SAT* implements a similar approach to Bounded Model Checking [28], checking the satisfiability of the LTL_f formulas iteratively over the length of the possible models until the theoretic upper threshold is reached. Such approach also shows the advantage when compared to *Aalta-finite* on particular benchmark suits [29]. Thus, the question raised initially in [18] needs to be re-opened with respect to LTL_f satisfiability checking: is it best to reduce to SAT-based model checking or develop a dedicated SAT-based tool?

Inspired by [27], we present an explicit-state SAT-based framework for LTL_f satisfiability. We construct the LTL_f *transition system* by utilizing SAT solvers to compute the states explicitly. For details, we propose the *neXt Normal Form* (XNF) for an LTL formula, which can be treated as a propositional formula by taking each temporal operator as a new atomic proposition, like the idea presented in [30]. The SAT solver is then utilized to compute states of the transition system w.r.t. the input LTL formula. Furthermore, by making use of both satisfiability and unsatisfiability information from SAT solvers, we propose a *conflict-driven* algorithm, CDLSC, for efficient LTL_f satisfiability checking. We show that by specializing the transition-system approach of [27] to LTL_f and its finite-trace semantics, we get a framework that is significantly simpler and yields a much more efficient algorithm CDLSC than the one in [27].

We conduct a comprehensive comparison among 5 different LTL_f-solving approaches off the shelf. Based on a comprehensive evaluation over 4 different LTL_f benchmark suits with a total amount of 9317 formulas, our time-cost analysis shows that 1) CDLSC performs best on checking unsatisfiable formulas by achieving approximately a 4X time speedup, compared to the second-best solution (K-LIVE [1]); 2) Although no approaches dominate checking satisfiable formulas, CDLSC performs best on 2 of the total 4 tested satisfiable benchmark suits; and 3) CDLSC gains the best overall performance when considering both satisfiable and unsatisfiable instances.

Compared to the previous conference version [31], this paper extends the contribution in the following way:

- Provide the full proofs for all the lemmas and theorems;
- Introduce more examples to help understand our proposed methodology;
- Re-construct the experiments by 1) allowing a timeout of 1 hour rather than 1 minute and 2) applying more benchmark suits from real life, to better support the efficiency of CDLSC.

The rest of this paper is organized as follows. Section *LTL over Finite Traces* introduces definitions for LTL_f and its satisfiability problem; Section *Approach Overview* provides an overview of our new approach in a high level; Section *SAT-based Explicit-State Checking* presents the SAT-based algorithm for LTL_f satisfiability checking; Section *Conflict-Driven* LTL_f *Satisfiability Checking* presents CDLSC, the main contribution of this paper; Section *Experimental Evaluation* demonstrates the experimental results; and finally, Section *Discussion and Concluding Remarks* concludes the paper.

2. LTL over finite traces

Given a set $\mathcal{P}$ of atoms, an LTL_f formula ϕ has the form:

$$\phi ::= \mathtt{tt} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{X}\phi \mid \phi \mathcal{U} \phi;$$

where $\mathtt{tt}$ is true, $\neg$ is the negation operator, $\wedge$ is the and operator, $\mathcal{X}$ is the strong Next operator and $\mathcal{U}$ is the Until operator. We also have the duals $\mathtt{ff}$ (false) for $\mathtt{tt}$, $\vee$ for $\wedge$, $\mathcal{N}$ (weak Next) for $\mathcal{X}$ and $\mathcal{R}$ for $\mathcal{U}$. A *literal* is an atom $p \in \mathcal{P}$ or its negation ($\neg p$). Moreover, we use the notation $\mathcal{G}\phi$ (Globally) and $\mathcal{F}\phi$ (Eventually) to represent $\mathtt{ff}\mathcal{R}\phi$ and $\mathtt{tt}\mathcal{U}\phi$. Notably, $\mathcal{X}$ is the standard *next* operator, while $\mathcal{N}$ is *weak next*; $\mathcal{X}$ requires the existence of a successor state, while $\mathcal{N}$ does not. Thus $\mathcal{N}\phi$ is always true in the last state of a finite trace, since no successor exists there. This distinction is specific to LTL_f.

LTL_f formulas are interpreted over finite traces [2]. Given an atom set $\mathcal{P}$, we define $\Sigma = 2^{\mathcal{P}}$ be the family of sets of atoms. Let $\xi \in \Sigma^+$ be a finite nonempty trace, with $\xi = \sigma_0\sigma_1 \ldots \sigma_n$. We use $|\xi| = n+1$ to denote the length of ξ. Moreover,

for $0 \leq i \leq n$, we denote $\xi[i]$ as the i-th position of ξ, and ξ_i to represent $\sigma_i\sigma_{i+1}\ldots\sigma_n$, which is the suffix of ξ from position i. We define the satisfaction relation $\xi \models \phi$ as follows:

- $\xi \models \mathtt{tt}$; and $\xi \models p$, if $p \in \mathcal{P}$ and $p \in \xi[0]$;
- $\xi \models \neg\phi$, if $\xi \not\models \phi$;
- $\xi \models \phi_1 \wedge \phi_2$, if $\xi \models \phi_1$ and $\xi \models \phi_2$;
- $\xi \models \mathcal{X}\phi$ if $|\xi| > 1$ and $\xi_1 \models \phi$;
- $\xi \models (\phi_1 \mathcal{U} \phi_2)$, if there exists $0 \leq i < |\xi|$ such that $\xi_i \models \phi_2$ and for every $0 \leq j < i$ it holds that $\xi_j \models \phi_1$.

Definition 1 *(LTL_f satisfiability problem).* Given an LTL_f formula ϕ over the alphabet Σ, we say ϕ is satisfiable iff there is a finite nonempty trace $\xi \in \Sigma^+$ such that $\xi \models \phi$.

Notations. We use $cl(\phi)$ to denote the set of subformulas of ϕ. Let A be a set of LTL_f formulas, we denote $\bigwedge A$ to be the formula $\bigwedge_{\psi \in A} \psi$. The two LTL_f formulas ϕ_1, ϕ_2 are semantically equivalent, denoted as $\phi_1 \equiv \phi_2$, iff for every finite trace ξ, $\xi \models \phi_1$ iff $\xi \models \phi_2$. Obviously, we have $(\phi_1 \vee \phi_2) \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\mathcal{N}\psi \equiv \neg\mathcal{X}\neg\psi$ and $(\phi_1 \mathcal{R} \phi_2) \equiv \neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$.

We say an LTL_f formula ϕ is in *Negated Normal Form* (NNF) if all negations are in front of only atoms. For example, the formula $X\neg a$ is in NNF while $\neg Xa$ is not. It is trivial to know that every LTL_f formula has an equivalent NNF. Moreover, we also define the *Tail Normal Form* (TNF) of an LTL_f formula ϕ as below.

Definition 2. Assume ϕ is in NNF, and its TNF $\mathsf{tnf}(\phi)$ is defined as $t(\phi) \wedge \mathcal{F}Tail$, where $Tail$ is a new atom to identify the last state of satisfying traces (Motivated from [2]), and $t(\phi)$ is an LTL_f formula defined recursively as follows:

1. $t(\phi) = \phi$ if ϕ is $\mathtt{tt}, \mathtt{ff}$ or a literal;
2. $t(\mathcal{X}\psi) = \neg Tail \wedge \mathcal{X}(t(\psi))$;
3. $t(\mathcal{N}\psi) = Tail \vee \mathcal{X}(t(\psi))$;
4. $t(\phi_1 \wedge \phi_2) = t(\phi_1) \wedge t(\phi_2)$;
5. $t(\phi_1 \vee \phi_2) = t(\phi_1) \vee t(\phi_2)$;
6. $t(\phi_1 \mathcal{U} \phi_2) = (\neg Tail \wedge t(\phi_1)) \mathcal{U} t(\phi_2)$;
7. $t(\phi_1 \mathcal{R} \phi_2) = (Tail \vee t(\phi_1)) \mathcal{R} t(\phi_2)$.

Theorem 1. *ϕ is satisfiable iff $tnf(\phi)$ is satisfiable.*

Proof. We provide a sketch of the proof here and the full proof can be found in Appendix. Let ξ, ξ' be two non-empty finite traces satisfying $|\xi| = |\xi'|$ and $\xi'[i] = \xi[i]$ for $0 \leq i < |\xi| - 1$ as well as $\xi'[|\xi| - 1] = \xi[|\xi| - 1] \cup \{Tail\}$. We can prove by induction over the type of ϕ that $\xi \models \phi$ iff $\xi' \models \mathsf{tnf}(\phi)$. For the ($\Leftarrow$) direction, we complement the proof that $\mathsf{tnf}(\phi)$ is satisfiable implies there is such a ξ' that $\xi' \models \mathsf{tnf}(\phi)$. □

In the rest of the paper, unless clearly specified, the input LTL_f formula is in TNF. Also, since every TNF has the common part $\mathcal{F}Tail$, we omit it for simplicity in the following. For example, the formula $(\neg Tail \wedge a)\mathcal{U}b$ actually represents the TNF $(\neg Tail \wedge a)\mathcal{U}b \wedge \mathcal{F}Tail$.

3. Approach overview

There is a Non-deterministic Finite Automaton (NFA) $\mathcal{A}_\phi$ that accepts exactly the same language as an LTL_f formula ϕ [2]. Instead of constructing the NFA for ϕ, we generate the corresponding *transition system* (Definition 6), by leveraging SAT solvers. The transition system represents an intermediate structure of the NFA, in which every state consists of a set of subformulas of ϕ. Notably in most cases, we construct the transition system partially to check the satisfiability of LTL_f formulas on the fly.

The classic approach to generate the NFA from an LTL_f formula, i.e., Tableau Construction [32], creates the set of all one-transition next states of the current state. Since the number of these states can be extremely large, we leverage SAT solvers to compute the next states of the current state iteratively. Although both approaches share the same worst case (computing all states in the state space), our new approach is better for on-the-fly checking, as it computes new states only if the satisfiability of the formula cannot be determined based on existing states.

We show the SAT-based approach via an example. Consider the formula $\phi = (\neg Tail \wedge a)\mathcal{U}b$, whose corresponding transition system is shown in Fig. 1. The initial state s_0 of the transition system is $\{\phi\}$. To compute the next states of s_0, we translate ϕ to its equivalent *neXt Normal Form* (XNF), e.g., $\mathsf{xnf}(\phi) = (b \vee ((\neg Tail \wedge a) \wedge \mathcal{X}\phi))$, see Definition 5. If we replace $\mathcal{X}\phi$ in $\mathsf{xnf}(\phi)$ with new propositions p_1, the new formula, denoted $\mathsf{xnf}(\phi)^p$, is a pure Boolean formula. As a result, a SAT solver can compute an assignment for the formula $\mathsf{xnf}(\phi)^p$. Assume the assignment is $\{a, \neg b, \neg Tail, p_1\}$, then we can induce that $(a \wedge \neg b \wedge \neg Tail \wedge \mathcal{X}\phi) \Rightarrow \phi$ is true, which indicates $\{\phi\} = s_0$ is a one-transition next state of s_0, i.e., s_0 has a self-loop

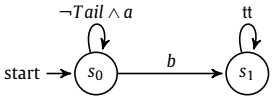

Fig. 1. The corresponding transition system of $\phi = (\neg Tail \wedge a)\mathcal{U}b$, where $s_0 = \{\phi\}$ and $s_1 = \{\mathsf{tt}\}$.

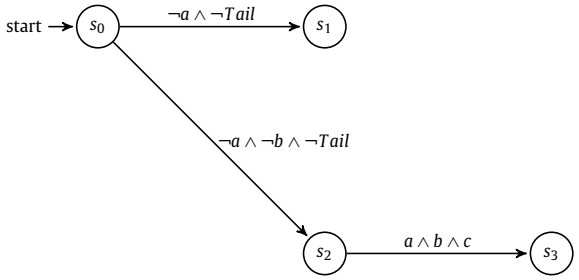

Fig. 2. An illustration to check on-the-fly on a part of the transition system for $\phi = (\neg Tail)\mathcal{U}a \wedge (\neg Tail)\mathcal{U}(\neg a) \wedge (\neg Tail)\mathcal{U}b \wedge (\neg Tail)\mathcal{U}(\neg b) \wedge (\neg Tail)\mathcal{U}c$. In the figure, $s_0 = \{\phi\}$, $s_1 = \{(\neg Tail)\mathcal{U}a \wedge (\neg Tail)\mathcal{U}b \wedge (\neg Tail)\mathcal{U}(\neg b) \wedge (\neg Tail)\mathcal{U}c\}$, $s_2 = \{(\neg Tail)\mathcal{U}a \wedge (\neg Tail)\mathcal{U}b \wedge (\neg Tail)\mathcal{U}c\}$ and $s_3 = \{\mathsf{tt}\}$.

with the label $\{a, \neg b, \neg Tail\}$. To compute another next state of s_0, we add the constraint $\neg p_1$ to the input of the SAT solver. Repeat the above process and we can construct all states in the transition system.

Checking the satisfiability of ϕ is then reduced to finding a *final state* (Definition 7) in the corresponding transition system. Since ϕ is in TNF, a final state s meets the constraint that $Tail \wedge \mathsf{xnf}(\bigwedge s)^p$ (recall s is a set of subformulas of ϕ) is satisfiable. For the above example, the initial state s_0 is actually a final state, as $Tail \wedge \mathsf{xnf}(\phi)^p$ is satisfiable. Because all states computed by the SAT solver in the transition system are reachable from the initial state, we can prove that ϕ is satisfiable iff there is a final state in the system (Theorem 4). Notably, a final state of the transition system defined in Definition 7 is not a traditional final state in an NFA; it is a state that has a transition to the traditional final state.

We present a conflict-driven algorithm, i.e., CDLSC, to accelerate the satisfiability checking. CDLSC maintains a *conflict sequence* $\mathcal{C}$, in which each element, denoted as $\mathcal{C}[i]$ $(0 \leq i < |\mathcal{C}|)$, is a set of states in the transition system that cannot reach a final state in i steps. Starting from the initial state, CDLSC iteratively checks whether a final state can be reached, and makes use of the conflict sequence to accelerate the search. Consider the formula $\phi = (\neg Tail)\mathcal{U}a \wedge (\neg Tail)\mathcal{U}(\neg a) \wedge (\neg Tail)\mathcal{U}b \wedge (\neg Tail)\mathcal{U}(\neg b) \wedge (\neg Tail)\mathcal{U}c$. In the first iteration, CDLSC checks whether the initial state $s_0 = \{\phi\}$ is a final state, i.e., whether $Tail \wedge \mathsf{xnf}(\phi)^p$ is satisfiable. The answer is negative, so s_0 cannot reach a final state in 0 steps and can be added into $\mathcal{C}[0]$. However, we can do better by leveraging the Unsatisfiable Core (UC) returned from the SAT solver. Assume that we get the UC $u_1 = \{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}(\neg a)\}$. That indicates every state s containing u, i.e., $s \supseteq u$, is not a final state. As a result, we can add u instead of s_0 into $\mathcal{C}[0]$, making the algorithm much more efficient.

Now in the second iteration, CDLSC first tries to compute a one-transition next state of s_0 that is not included in $\mathcal{C}[0]$. (Otherwise the new state cannot reach a final state in 0 step.) This can be encoded as a Boolean formula $\mathsf{xnf}(\phi)^p \wedge \neg(p_1 \wedge p_2)$ where p_1, p_2 represent $\mathcal{X}((\neg Tail)\mathcal{U}a)$ and $\mathcal{X}((\neg Tail)\mathcal{U}(\neg a))$ respectively. Assume the new state $s_1 = \{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}(\neg b), (\neg Tail)\mathcal{U}c\}$ is generated from the assignment of the SAT solver. Then CDLSC checks whether s_1 can reach a final state in 0 steps, i.e., $\mathsf{xnf}(\bigwedge s_1)^p \wedge Tail$ is satisfiable. The answer is negative and we can add the UC $u_2 = \{(\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}(\neg b)\}$ to $\mathcal{C}[0]$ as well. Now to compute a next state of s_0 that is not included in $\mathcal{C}[0]$, the encoded Boolean formula becomes $\mathsf{xnf}(\phi)^p \wedge \neg(p_1 \wedge p_2) \wedge \neg(p_3 \wedge p_4)$ where p_3, p_4 represent $\mathcal{X}((\neg Tail)\mathcal{U}b)$ and $\mathcal{X}((\neg Tail)\mathcal{U}(\neg b))$ respectively. Assume the new state $s_2 = \{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}c\}$ is generated from the assignment of the SAT solver. Since $\mathsf{xnf}(\bigwedge s_2)^p \wedge Tail$ is satisfiable, s_2 is a final state and we conclude that the formula ϕ is satisfiable. An illustration to the states computation above is shown in Fig. 2. In principle, there are a total of $2^5 = 32$ states in the transition system of ϕ, but CDLSC succeeds to find the answer by computing only 3 of them (including the initial state).

CDLSC also leverages the conflict sequence to accelerate checking unsatisfiable formulas. Like Bounded Model Checking (BMC) [28], CDLSC searches the model iteratively, but BMC invokes only one SAT call for each iteration, while CDLSC invokes multiple SAT calls. CDLSC is more like an IC3-style algorithm, but achieves a much simpler implementation by using UC instead of the *Minimal Inductive Core* (MIC) like IC3 [25].

4. SAT-based explicit-state checking

Given an LTL_f formula ϕ, we construct the LTL_f *transition system* [21,27] leveraging SAT solvers and then check the satisfiability of the formula over its corresponding transition system.

4.1. LTL_f transition system

First, we show how one can consider LTL_f formulas as propositional ones. This requires considering temporal subformulas as *propositional atoms*.

Definition 3 *(Propositional atoms).* For an LTL_f formula ϕ, we define the set of *propositional atoms* of ϕ, i.e., $\mathsf{PA}(\phi)$, as follows: (1) $\mathsf{PA}(\phi) = \{\phi\}$ if ϕ is an atom, Next, Until or Release formula; (2) $\mathsf{PA}(\phi) = \mathsf{PA}(\psi)$ if $\phi = (\neg\psi)$; (3) $\mathsf{PA}(\phi) = \mathsf{PA}(\phi_1) \cup \mathsf{PA}(\phi_2)$ if $\phi = (\phi_1 \wedge \phi_2)$ or $(\phi_1 \vee \phi_2)$.

Consider $\phi = (a \wedge ((\neg Tail \wedge a)\mathcal{U}b) \wedge \neg(\neg Tail \wedge \mathcal{X}(a \vee b)))$. We have $\mathsf{PA}(\phi) = \{a, Tail, ((\neg Tail \wedge a)\mathcal{U}b), (\mathcal{X}(a \vee b))\}$. Intuitively, the propositional atoms are obtained by treating all temporal subformulas of ϕ as atoms. Thus, an LTL_f formula ϕ can be viewed as a propositional formula over $\mathsf{PA}(\phi)$.

Definition 4. For an LTL_f formula ϕ, let ϕ^p be ϕ considered as a propositional formula over $\mathsf{PA}(\phi)$. A *propositional assignment* A of ϕ^p, is in $2^{\mathsf{PA}(\phi)}$ and satisfies $A \models \phi^p$.

Consider the formula $\phi = (a \vee (\neg Tail \wedge a)\mathcal{U}b) \wedge (b \vee (Tail \vee c)\mathcal{R}d)$. From Definition 4, ϕ^p is $(a \vee p_1) \wedge (b \vee p_2)$ where p_1, p_2 are two Boolean variables representing the truth values of $(\neg Tail \wedge a)\mathcal{U}b$ and $(Tail \vee c)\mathcal{R}d$. Moreover, the set $\{p_1, p_2\}$ is a propositional assignment of ϕ^p. **In the rest of the paper, we do not introduce the intermediate variables and directly say $\{(\neg Tail \wedge a)\mathcal{U}b, (Tail \vee c)\mathcal{R}d\}$ is a propositional assignment of ϕ^p.** The following theorem shows the relationship between the propositional assignment of ϕ^p and the satisfaction of ϕ.

Theorem 2. *For an LTL_f formula ϕ and a finite trace ξ, $\xi \models \phi$ implies there exists a propositional assignment A of ϕ^p such that $\xi \models \bigwedge A$.*

Proof. ($\Rightarrow$) Base case: when ϕ is a literal, Next, Until or Release formula, it is true since there is only one propositional assignment of ϕ^p, i.e. $A = \{\phi\}$. Inductive step: if $\phi = \phi_1 \wedge \phi_2$, $\xi \models \phi$ implies $\xi \models \phi_1$ and $\xi \models \phi_2$. By assumption hypothesis, there is A_i of ϕ_i^p $(i = 1, 2)$ such that $\xi \models \bigwedge A_i$. Let $A = A_1 \cup A_2$, and a consistent A, in which either ψ or $\neg\psi$ cannot be, 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 ξ cannot model $\bigwedge A_1$ and $\bigwedge A_2$ at the same time, which is a contradiction. So A is a propositional assignment of ϕ^p and $\xi \models \bigwedge A$. The proof for $\phi = \phi_1 \vee \phi_2$ is similar. □

We now introduce the *neXt Normal Form* (XNF) of LTL_f formulas, which is useful for the construction of the transition system.

Definition 5 *(neXt normal form).* An LTL_f formula ϕ is in *neXt Normal Form* (XNF) if there are no Until or Release subformulas of ϕ in $\mathsf{PA}(\phi)$.

For example, $\phi = ((\neg Tail \wedge a)\mathcal{U}b)$ is not in XNF, while $(b \vee (\neg Tail \wedge a \wedge (\mathcal{X}((\neg Tail \wedge a)\mathcal{U}b))))$ is. Every LTL_f formula ϕ has a linear-time conversion to an equivalent formula in XNF, which we denoted as $\mathsf{xnf}(\phi)$.

Theorem 3. *For an LTL_f formula ϕ, there is a corresponding LTL_f formula $\mathsf{xnf}(\phi)$ in XNF such that $\phi \equiv \mathsf{xnf}(\phi)$. Furthermore, the cost of the conversion is linear.*

Proof. First, $\mathsf{xnf}(\phi)$ can be constructed recursively as follows: (1) $\mathsf{xnf}(\phi) = \phi$, when ϕ is tt, ff, a literal or $\mathcal{X}\psi$ (note ϕ is $\mathcal{N}$-free); (2) $\mathsf{xnf}(\phi_1\ o\ \phi_2) = \mathsf{xnf}(\phi_1)\ o\ \mathsf{xnf}(\phi_2)$, where o is $\wedge$ or $\vee$; (3) $\mathsf{xnf}(\phi_1\mathcal{U}\phi_2) = \mathsf{xnf}(\phi_2) \vee (\mathsf{xnf}(\phi_1) \wedge \mathcal{X}(\phi_1\mathcal{U}\phi_2))$; and (4) $\mathsf{xnf}(\phi_1\mathcal{R}\phi_2) = \mathsf{xnf}(\phi_2) \wedge (\mathsf{xnf}(\phi_1) \vee \mathcal{X}(\phi_1\mathcal{R}\phi_2))$; Since the construction is built on two expansion rules of Until and Release, and the expansion stops once the Until and Release are in the scope of Next, it preserves the equivalence $\phi \equiv \mathsf{xnf}(\phi)$. Since no expansion is applied to the $\mathcal{X}$ operator and the Next formulas are considered as atomic ones in XNF, the conversion cost is at most linear. □

Observe that when ϕ is in XNF, there can be only Next (no Until or Release) temporal formulas in the propositional assignment of ϕ^p. For $\phi = b \vee (a \wedge \neg Tail \wedge \mathcal{X}(a\mathcal{U}b))$, the set $A = \{a, \neg b, \neg Tail, \mathcal{X}(a\mathcal{U}b)\}$ is a propositional assignment of ϕ^p. Based on LTL_f semantics, we can induce from A that if a finite trace ξ satisfying $\xi[0] \supseteq \{a, \neg b, \neg Tail\}$ and $\xi_1 \models a\mathcal{U}b$, $\xi \models \phi$ is true. This motivates us to construct the transition system for ϕ, in which $\{a\mathcal{U}b\}$ is a next state of $\{\phi\}$ and $\{a, \neg b, \neg Tail\}$ is the transition label between these two states.

Let ϕ be an LTL_f formula and A be a propositional assignment of ϕ^p, we denote $L(A) = \{l | l \in A \text{ is a literal}\}$ and $X(A) = \{\theta | \mathcal{X}\theta \in A\}$. Now we define the *transition system* for an LTL_f formula.

Definition 6. Given an LTL_f formula ϕ and its literal set $\mathcal{L}$, let $\Sigma = 2^{\mathcal{L}}$. We define the *transition system* $T_\phi = (S, s_0, T)$ for ϕ, where

(1) $S \subseteq 2^{cl(\phi)}$ is the set of states in which every state s except the initial one is obtained as described in (3);
(2) $s_0 = \{\phi\}$ is the *initial state*;
(3) $T : S \times \Sigma \to 2^S$ is the transition relation, where $s_2 \in T(s_1, \sigma)$ $(\sigma \in \Sigma)$ holds iff there is a propositional assignment A of $\mathsf{xnf}(\bigwedge s_1)^p$ such that $\sigma \supseteq L(A)$ and $s_2 = X(A)$.

A *run* of T_ϕ on a finite trace $\xi(|\xi| = n > 0)$ is a finite sequence $s_0, s_1, \ldots, s_n$ such that s_0 is the initial state and $s_{i+1} \in T(s_i, \xi[i])$ holds for all $0 \leq i < n$.

We define the notation $|r|$ for a run r, to represent the length of r, i.e., number of states in r. We say state s_2 is reachable from state s_1 in $i(i \geq 0)$ steps (resp. in up to i steps), if there is a run r on some finite trace ξ leading from s_1 to s_2 and $|r| = i$ (resp. $|r| \leq i$). In particular, we say s_2 is a *one-transition next state* of s_1 if s_2 is reachable from s_1 in 1 steps. Since a state s is a subset of $cl(\phi)$, which essentially is a formula with the form of $\bigwedge_{\psi \in s} \psi$, we mix the usage of the state and formula in the rest of the paper. That is, a state can be a formula of $\bigwedge_{\psi \in S} \psi$, and a formula ϕ can be a set of states, i.e., $s \in \phi$ iff $s \Rightarrow \phi$.

Lemma 1. *Let $T_\phi = (S, s_0, T)$ be the transition system of ϕ. Every state $s \in S$ is reachable from the initial state s_0.*

Proof. Basically, for $s \in T(s_0, \sigma)$ $(\sigma \in \Sigma)$, since there is a propositional assignment A of $\mathsf{xnf}(\bigwedge s_0)^p$ such that $\sigma \supseteq L(A)$ and $s = X(A)$, s is reachable from s_0 in one step. Inductively, assume s is reachable from s_0 in k $(k \geq 1)$ steps. For $s' \in T(s, \sigma)$ $(\sigma \in \Sigma)$, similarly we have s' is reachable from s in one step. As a result, s' is reachable from s_0 in $k+1$ steps. □

Definition 7 *(Final state).* Let s be a state of a transition system T_ϕ. Then s is a *final state* of T_ϕ iff the Boolean formula $Tail \wedge (\mathsf{xnf}(s))^p$ is satisfiable.

By introducing the concept of *final state*, we are able to check the satisfiability of the LTL_f formula ϕ over T_ϕ.

Lemma 2. *s is a final state of T_ϕ, iff there is a finite trace ξ with $|\xi| = 1$ such that $\xi \models s$.*

Proof. From Definition 7, s is a final state iff there is a propositional assignment A of the Boolean formula $Tail \wedge (\mathsf{xnf}(s))^p$ and $Tail \in A$. Recall that every Next subformula in s is associated with $\neg Tail$, so $Tail \in A$ holds iff no Next subformula is in A, and thus iff $L(A) \models \mathsf{xnf}(s)^p$ holds. Let $\xi = \sigma$ $(\sigma \in \Sigma)$ such that $\sigma \supseteq L(A)$, and obviously $\xi \models s$. □

Theorem 4. *Let ϕ be an LTL_f formula. Then ϕ is satisfiable iff there is a final state in T_ϕ.*

Proof. ($\Rightarrow$) Since ϕ is satisfiable, there is a finite trace $\xi \models \phi$. Assume $|\xi| = n(n > 0)$. Based on Theorem 2, there is a propositional assignment A_0 of $\mathsf{xnf}(\phi)^p$ such that $\xi \models \bigwedge A_0$. And according to Definition 6, there is a transition $s_1 \in T(s_0, \sigma_0)$ in T_ϕ where $s_0 = \{\phi\}$, $\sigma_0 \supseteq L(A_0)$ and $s_1 = X(A_0)$. Moreover, we have that $\xi_1 \models s_1$. Recursively, we can prove that for $n > i \geq 0$, there is a transition $s_{i+1} \in T(s_i, \sigma_i)$ in T_ϕ such that $\sigma_i \supseteq L(A_i)$, $s_{i+1} = X(A_i)$ for some propositional assignment A_i of $\mathsf{xnf}(s_i)^p$, and $\xi_{i+1} \models s_{i+1}$ holds. For $i = n-1$, since $|\xi_i| = 1$ and $\xi_i \models s_i$, s_i is a final state according to Lemma 2, and it is reachable from s_0 based on Lemma 1.

($\Leftarrow$) Let s be a final state in T_ϕ, and it is reachable from the initial state s_0 from Lemma 1. Assume a run $r = s_0, \ldots, s_{n-1}, s(n >= 0)$ (when $n = 0$, $s = s_0$ is the initial state) of T_ϕ on $\xi' = \sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ leads from ϕ to s. Moreover according to Lemma 2, there is a finite trace ξ'' with $|\xi''| = 1$ such that $\xi'' \models s$. Let $\xi = \xi' \cdot \xi'' = \sigma_0\sigma_1, \ldots \sigma_n (n \geq 0)$ where $\xi'' = \sigma_n$, and now we prove that $\xi \models \phi$. The proof can be achieved by induction from n to 0. Basically, $(\xi_n = \sigma_n) \models s$ is obviously true. Inductively assume $\xi_i \models s_i$ for $n \geq i \geq 1$, so $\xi_{i-1} = \xi[i-1] \cdot \xi_i$ satisfies $\xi[i-1] \supseteq L$ and $\xi_i \models s_i$ for some $s_i \in T(s_{i-1}, L)$ from the definition of T_ϕ, which means $\xi_{i-1} \models s_{i-1}$. When $i = 0$, we prove that $(\xi = \xi_0) \models (s_0 = \phi)$. □

An intuitive solution from Theorem 4 to check the satisfiability of ϕ is to construct states of T_ϕ until (1) either a final state is found by Definition 7, meaning ϕ is satisfiable; or (2) all states in T_ϕ are generated but no final state can be found, meaning ϕ is unsatisfiable. This approach is simple and easy to implement, however, it does not perform well according to our preliminary experiments.

5. Conflict-driven LTL_f satisfiability checking

In this section, we present a conflict-driven algorithm for LTL_f satisfiability checking. The new algorithm is inspired by [27], where information of both satisfiability and unsatisfiability results of SAT solvers are used. The motivation is as follows:

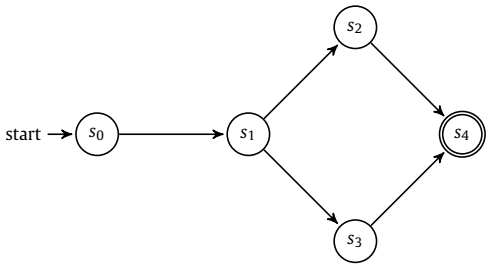

Fig. 3. An example transition system for the conflict sequence.

In Definition 7, if the Boolean formula $Tail \wedge \mathsf{xnf}(s)^p$ is unsatisfiable, the SAT solver is able to provide a UC (Unsatisfiable Core) c such that $c \subseteq s$ and $Tail \wedge \mathsf{xnf}(c)^p$ is still unsatisfiable. It means that c represents a set of states that are not final states. By adding a new constraint $\neg(\bigwedge_{\psi \in c} \mathcal{X}\psi)$, the SAT solver can provide a model (if exists) that avoids re-generation of those states in c, which accelerates the search of final states. More generally, we define the *conflict sequence*, which is used to maintain all information of UCs acquired during the checking process.

Definition 8 *(Conflict sequence).* Given an LTL_f formula ϕ, a conflict sequence $\mathcal{C}$ for the transition system T_ϕ is a finite sequence of sets of states such that:

1. The initial state $s_0 = \{\phi\}$ is in $\mathcal{C}[i]$ for $0 \leq i < |\mathcal{C}|$;
2. Every state in $\mathcal{C}[0]$ is not a final state;
3. For every state $s \in \mathcal{C}[i+1]$ $(0 \leq i < |\mathcal{C}| - 1)$, all the one-transition next states of s are included in $\mathcal{C}[i]$.

We call each $\mathcal{C}[i]$ a *frame*, and i is the *frame level*.

In the definition, $|\mathcal{C}|$ represents the length of $\mathcal{C}$ and $\mathcal{C}[i]$ denotes the i-th element of $\mathcal{C}$. Consider the transition system shown in Fig. 3, in which s_0 is the initial state and s_4 is the final state. Based on Definition 8, the sequence $\mathcal{C} = \{s_0, s_1, s_2, s_3\}, \{s_0, s_1\}, \{s_0\}$ is a conflict sequence. Notably, the conflict sequence for a transition system may not be unique. For the above example, the sequence $\{s_0, s_1\}, \{s_0\}$ is also a conflict sequence for the system. This suggests that the construction of a conflict sequence is algorithm-specific. Moreover, it is not hard to induce that every non-empty prefix of a conflict sequence is also a conflict sequence. For example, a prefix of $\mathcal{C}$ above, i.e., $\{s_0, s_1, s_2, s_3\}, \{s_0, s_1\}$, is a conflict sequence. As a result, a conflict sequence can be constructed iteratively, i.e., the elements can be generated (and updated) in order. Our new algorithm is motivated by these two observations.

An inherent property of conflict sequences is described in the following lemma.

Lemma 3. *Let ϕ be an LTL_f formula with a conflict sequence $\mathcal{C}$ for the transition system T_ϕ, then $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] (0 \leq i < |\mathcal{C}|)$ represents a set of states that cannot reach a final state in up to i steps.*

Proof. We first prove $\mathcal{C}[i] (i \geq 0)$ is a set of states that cannot reach a final state in i steps. Basically from Definition 8, $\mathcal{C}[0]$ is a set of states that are not final states. Inductively, assume $\mathcal{C}[i] (i \geq 0)$ is a set of states that cannot reach a final state in i steps. From Item 3 of Definition 8, every state $s \in \mathcal{C}[i+1]$ satisfies all its one-transition next states are in $\mathcal{C}[i]$, thus every state $s \in \mathcal{C}[i+1]$ cannot reach a final state in $i+1$ steps. Now since $\mathcal{C}[i] (i \geq 0)$ is a set of states that cannot reach a final state in i steps, $\bigcap_{0 \leq j \leq i} \mathcal{C}[j]$ is a set of states that cannot reach a final state in up to i steps. □

We are able to utilize the conflict sequence to accelerate the satisfiability checking of LTL_f formulas, using the theoretical foundations provided by Theorem 5 and 6 below.

Theorem 5. *The LTL_f formula ϕ is satisfiable iff there is a run $r = s_0, s_1, \ldots, s_n (n \geq 0)$ of T_ϕ such that (1) s_n is a final state; and (2) s_i $(0 \leq i \leq n)$ is not in $\mathcal{C}[n-i]$ for every conflict sequence $\mathcal{C}$ of T_ϕ with $|\mathcal{C}| > n - i$.*

Proof. $(\Leftarrow)$ Since s_n is a final state, ϕ is satisfiable according to Theorem 4. $(\Rightarrow)$ Since ϕ is satisfiable, there is a finite trace ξ such that the corresponding run r of T_ϕ on ξ ends with a final state (according to Theorem 4). Let r be $s_0 \rightarrow s_1 \rightarrow \ldots s_n$ where s_n is the final state. It holds that s_i $(0 \leq i \leq n)$ is a state that can reach a final state in $n - i$ steps. Moreover for every $\mathcal{C}$ of T_ϕ with $|\mathcal{C}| > n - i$, $\mathcal{C}[n-i]$ ($\mathcal{C}[n-i]$ is meaningless when $|\mathcal{C}| \leq n - i$) represents a set of states that cannot reach a final state in $n - i$ steps (from Lemma 3). As a result, it is true that s_i is not in $\mathcal{C}[n-i]$ if $|\mathcal{C}| > n - i$. □

Theorem 5 suggests that to check whether a state s can reach a final state in i steps $(i \geq 1)$, finding a one-transition next state s' of s that is not in $\mathcal{C}[i-1]$ is necessary; as $s' \in \mathcal{C}[i-1]$ implies s' cannot reach a final state in $i - 1$ steps (from the proof of Lemma 3). If all one-transition next states of s are in $\mathcal{C}[i-1]$, s cannot reach a final state in i steps.

Algorithm 1 Implementation of CDLSC.

Require: An LTL_f formula ϕ.
Ensure: SAT or UNSAT.
1: **if** $Tail \wedge \mathsf{xnf}(\phi)^p$ is satisfiable **then**
2: **return** SAT;
3: **end if**
4: Set $\mathcal{C}[0] := get\_uc()$;
5: Set $frame\_level := 0$;
6: **while** true **do**
7: Set $\mathcal{C}[frame\_level + 1] := \emptyset$;
8: **if** $try\_satisfy(\phi, frame\_level)$ returns true **then**
9: **return** SAT;
10: **end if**
11: **if** $inv\_found(frame\_level)$ returns true **then**
12: **return** UNSAT;
13: **end if**
14: $frame\_level := frame\_level + 1$;
15: **end while**

Theorem 6. *The LTL_f formula ϕ is unsatisfiable iff there is a conflict sequence $\mathcal{C}$ and $i \geq 0$ such that $\bigcap_{0\leq j\leq i} \mathcal{C}[j] \subseteq \mathcal{C}[i+1]$.*

Proof. ($\Leftarrow$) $\bigcap_{0\leq j\leq i} \mathcal{C}[j] \subseteq \mathcal{C}[i+1]$ is true implies that $\bigcap_{0\leq j\leq i} \mathcal{C}[j] = \bigcap_{0\leq j\leq i+1} \mathcal{C}[j]$ is true. Also from Lemma 3 we know $\bigcap_{0\leq j\leq i} \mathcal{C}[j]$ is a set of states that cannot reach a final state in up to i steps. Since $\phi \in \mathcal{C}[i]$ is true for each $i \geq 0$, ϕ is in $\bigcap_{0\leq j\leq i} \mathcal{C}[j]$. Moreover, $\bigcap_{0\leq j\leq i} \mathcal{C}[j] = \bigcap_{0\leq j\leq i+1} \mathcal{C}[j]$ is true implies all reachable states from ϕ are included in $\bigcap_{0\leq j\leq i} \mathcal{C}[j]$; otherwise there is a reachable state t from the initial state such that $t \notin \bigcap_{0\leq j\leq i} \mathcal{C}[j]$ but $t \in \bigcap_{0\leq j\leq i+1} \mathcal{C}[j]$, which becomes a contradiction. We have known all states in $\bigcap_{0\leq j\leq i} \mathcal{C}[j]$ are not final states, so ϕ is unsatisfiable.

($\Rightarrow$) If ϕ is unsatisfiable, every state in T_ϕ is not a final state. Let S be a set of all states of T_ϕ. We now define a sequence of sets of states $\mathcal{C}$ as follows: $\mathcal{C}[0] = \mathcal{C}[1] = S$ and $|\mathcal{C}| = 2$. According to Definition 8, it is trivial to check that $\mathcal{C}$ is a conflict sequence. Moreover, $\bigcap_{0\leq j\leq i} \mathcal{C}[j] \subseteq \mathcal{C}[i+1]$ is obviously true for $i = 0$. The proof is done. □

Algorithm design. The algorithm, named CDLSC (Conflict-Driven LTL_f Satisfiability Checking), constructs the transition system on-the-fly. The initial state s_0 is fixed to be $\{\phi\}$ where ϕ is the input formula. From Definition 7, whether a state s is final is reducible to the satisfiability checking of the Boolean formula $Tail \wedge \mathsf{xnf}(s)^p$. If s_0 is a final state, there is no need to maintain the conflict sequence in CDLSC, and the algorithm can return SAT immediately; Otherwise, the conflict sequence is maintained as follows.

- In CDLSC, every element of $\mathcal{C}$ is a set of sets of subformulas of the input formula ϕ. Formally, each $\mathcal{C}[i]$ ($i \geq 0$) can be represented by the LTL_f formula $\bigvee_{c\in\mathcal{C}[i]} \bigwedge_{\psi\in c} \psi$ where c is a set of subformulas of ϕ. We mix-use the notation $\mathcal{C}[i]$ for the corresponding LTL_f formula as well. Every state s satisfying $s \Rightarrow \mathcal{C}[i]$ is included in $\mathcal{C}[i]$.
- $\mathcal{C}$ is created iteratively. In each iteration $i \geq 0$, $\mathcal{C}[i]$ is initialized as the empty set.
- To compute elements in $\mathcal{C}[0]$, we consider an existing state s (e.g., s_0). If the Boolean formula $Tail \wedge \mathsf{xnf}(s)^p$ is unsatisfiable, s is not a final state and can be added into $\mathcal{C}[0]$ from Item 2 of Definition 8. Moreover, CDLSC leverages the Unsatisfiable Core (UC) technique from the SAT community to add a set of states, all of which are not final and include s, to $\mathcal{C}[0]$. This set of states, denoted as c, is also represented by a set of LTL_f formulas and satisfies $c \subseteq s$.
- To compute elements in $\mathcal{C}[i+1]$ ($i \geq 0$), we consider the Boolean formula $(\mathsf{xnf}(s) \wedge \neg\mathcal{X}(\mathcal{C}[i]))^p$, where $\mathcal{X}(\mathcal{C}[i])$ represents the LTL_f formula $\bigvee_{c\in\mathcal{C}[i]} \bigwedge_{\psi\in c} \mathcal{X}(\psi)$. The above Boolean formula is used to check whether there is a one-transition next state of s that is not in $\mathcal{C}[i]$. If the formula is unsatisfiable, all the one-transition next states of s are in $\mathcal{C}[i]$, thus s can be added into $\mathcal{C}[i+1]$ according to Item 3 of Definition 8. Similarly, we also utilize the UC technique to obtain a subset c of s, such that c represents a set of states that can be added into $\mathcal{C}[i+1]$.

As shown above, every Boolean formula sent to a SAT solver has the form of $(\mathsf{xnf}(s) \wedge \theta)^p$ where s is a state and θ is either $Tail$ or $\neg\mathcal{X}(\mathcal{C}[i])$. Since every state s consists of a set of LTL_f formulas, the Boolean formula can be rewritten as $\alpha_1 = (\bigwedge_{\psi\in s} \mathsf{xnf}(\psi) \wedge \theta)^p$. Moreover, we introduce a new Boolean variable p_ψ for each $\psi \in s$, and re-encode the formula to be $\alpha_2 = \bigwedge_{\psi\in s} p_\psi \wedge (\bigwedge_{\psi\in s}(\mathsf{xnf}(\psi) \vee \neg p_\psi) \wedge \theta)^p$. α_2 is satisfiable iff α_1 is satisfiable, and A is an assignment of α_2 iff $A\backslash\{p_\psi | \psi \in s\}$ is an assignment of α_1. Sending α_2 instead of α_1 to the SAT solver that supports assumptions (e.g., Minisat [33]) enables the SAT solver to return the UC, which is a set of s, when α_2 is unsatisfiable. For example, assume $s = \{\psi_1, \psi_2, \psi_3\}$ and α_2 is sent to the SAT solver with $\{p_{\psi_i} | i \in \{1, 2, 3\}\}$ being the assumptions. If the SAT solver returns unsatisfiable and the UC $\{p_{\psi_1}\}$, the set $c = \{\psi_1\}$, which represents every state including ψ_1, is the one to be added into the corresponding $\mathcal{C}[i]$. We use the notation $get\_uc()$ for the above procedure.

The pseudo-code of CDLSC is shown in Algorithm 1. Lines 1-3 consider the case when the input formula ϕ is a final state itself. Otherwise, the first frame $\mathcal{C}[0]$ is initialized to $get_uc()$ (Line 4), and the current frame level is set to 0 (Line 5). After that, the loop body (Line 6-15) keeps updating the elements of $\mathcal{C}$ iteratively, until either the procedure $try\_satisfy$ returns true, which means it found a model of ϕ, or the procedure inv_found returns true, which is the implementation

Algorithm 2 Implementation of $try\_satisfy$.

Require: ϕ: The formula is working on;
$frame\_level$: The frame level is working on.
Ensure: true or false.
1: Let $\psi := \neg\mathcal{X}(\mathcal{C}[frame\_level])$;
2: **while** $(\psi \wedge \mathsf{xnf}(\phi))^p$ is satisfiable **do**
3: Let A be the model of $(\psi \wedge \mathsf{xnf}(\phi))^p$;
4: Let $\phi' := X(A)$, i.e., be the next state of ϕ extracted from A;
5: **if** $frame\_level == 0$ **then**
6: **if** $Tail \wedge \mathsf{xnf}(\phi')^p$ is satisfiable **then**
7: **return** true;
8: **else**
9: Let $c := get\_uc()$;
10: Add c into $\mathcal{C}[0]$;
11: **end if**
12: **else**
13: **if** $try\_satisfy(\phi', frame\_level - 1)$ is true **then**
14: **return** true;
15: **end if**
16: **end if**
17: Let $\psi := \neg\mathcal{X}(\mathcal{C}[frame\_level])$;
18: **end while**
19: Let $c := get\_uc()$;
20: Add c into $\mathcal{C}[frame\_level + 1]$;
21: **return** false;

of Theorem 6. The loop continues to create a new frame in $\mathcal{C}$ if neither of the procedures succeeds to return true. We call each run of the while loop body in Algorithm 1 an *iteration*.

The procedure $try\_satisfy$ updates $\mathcal{C}$. Taking a formula ϕ and the current frame level, $try\_satisfy$ returns true iff a model of ϕ can be found, with the length of $frame\_level + 1$. As shown in Algorithm 2, $try\_satisfy$ is implemented recursively. Each time it checks whether a next state of the input ϕ, which belongs to a lower level (than the input $frame\_level$) frame can be found (Line 2). If such a new state ϕ' is constructed, $try\_satisfy$ first checks whether ϕ' is a final state when $frame\_level$ is 0 and returns true if so. If ϕ' is not a final state, a UC is extracted from the SAT solver and added to $\mathcal{C}[0]$ (Line 5-11). If $frame\_level$ is not 0, $try\_satisfy$ recursively checks whether a model of ϕ' can be found with the length of $frame\_level$ (Line 13-15). The while loop (Line 2-18) will terminate since ψ is continuously updated at Line 17. Finally, if the result is negative and such a state cannot be constructed, a UC is extracted from the SAT solver and added into $\mathcal{C}[frame\_level + 1]$ (Line 19-20). Notably, even though the input of the SAT solver is $(\psi \wedge \mathsf{xnf}(\phi))^p$ at Line 2, get_uc only returns elements from ϕ.

Notably, Item 1 of Definition 8, i.e., $\{\phi\} \in \mathcal{C}[i]$, is guaranteed for each $i \geq 0$, as the original input formula of $try\_satisfy$ is always ϕ (Line 8 in Algorithm 1) and there is some c (Line 20 in Algorithm 2) including $\{\phi\}$ that is added into $\mathcal{C}[i]$, if no model can be found in the current iteration.

The procedure inv_found in Algorithm 1 implements Theorem 6 in a straightforward way: it reduces checking whether $\bigcap_{0\leq j\leq i}\mathcal{C}[j] \subseteq \mathcal{C}[i+1]$ holds on some frame level i, to satisfiability checking of the Boolean formula $\bigwedge_{0\leq j\leq i}\mathcal{C}[j] \Rightarrow \mathcal{C}[i+1]$. The implementation of inv_found then simply enumerates such checking by varying i from 0 to $|\mathcal{C}| - 1$. Theorem 7 provides the theoretical guarantee that CDLSC always terminates correctly.

Lemma 4. *After each iteration of CDLSC with no model found, the sequence $\mathcal{C}$ is a conflict sequence of T_ϕ for the transition system T_ϕ.*

Proof. First, CDLSC sets $\mathcal{C}[0] = \{\phi\}$ after checking $Tail \wedge \mathsf{xnf}(\phi)^p$ is unsatisfiable at Line 4 of Algorithm 1 and Line 10 of Algorithm 2, which meets Item 2 of Definition 8. Secondly after each iteration $i \geq 0$, $try\_satisfy$ guarantees that $\{\phi\}$ is added into each $\mathcal{C}[i]$ if no model is found, which meets Item 1 of Definition 8. By enumerating Line 10 and 20 in $try\_satisfy$ (Algorithm 2), we have that $\mathsf{xnf}(s) \wedge \neg\mathcal{X}(\mathcal{C}[i])$ is unsatisfiable for $s \in \mathcal{C}[i+1](0 \leq i \leq |\mathcal{C}| - 1)$, which meets Item 3 of Definition 8. So $\mathcal{C}$ is a conflict sequence after each iteration with no model found. □

Theorem 7. *The CDLSC algorithm terminates with a correct result.*

Proof. CDLSC runs iteratively, so CDLSC terminates iff either the procedure $try\_satisfy$ or inv_found returns true for some iteration. From Lemma 4, $\mathcal{C}$ is a conflict sequence after each iteration if no model found. After each iteration, $try\_satisfy$ returns true iff a final state is found (Line 6-7) based on Theorem 5. As $\mathcal{C}$ is a conflict sequence after each iteration, inv_found returns true if there is $0 \leq i < |\mathcal{C}|$ such that the Boolean formula $\bigwedge_{0\leq j\leq i}\mathcal{C}[j] \Rightarrow \mathcal{C}[i+1]$ holds, which equivalently means $\bigcap_{0\leq j\leq i}\mathcal{C}[j] \subseteq \mathcal{C}[i+1]$ holds for some i. According to Theorem 6, ϕ is thus unsatisfiable. On the other hand, ϕ is unsatisfiable can imply there is a conflict sequence $\mathcal{C}$ such that $\bigwedge_{0\leq j\leq i}\mathcal{C}[j] \Rightarrow \mathcal{C}[i+1]$ holds for some $i > 0$, due to the fact

that the number of states in the transition system is finite. Therefore, inv_found can eventually return true. As a result, there is always such an iteration, after which CDLSC can terminate and terminate correctly. □

CDLSC is a heuristic algorithm for LTL_f satisfiability checking so it does not improve the lower or upper bound of the problem in theory. That means, the complexity of LTL_f satisfiability preserves to be PSPACE-complete [2] even the problem is solved by CDLSC. As a result, CDLSC may construct at most 2^n states of the transition system for a given LTL_f formula with size n. Since each state of the transition system and element of the conflict sequence are computed via a SAT call, CDLSC can invoke more than 2^n SAT calls in the worst case to check the satisfiability of a formula with size n. Also the states generated in CDLSC are stored explicitly, the actual space used for the algorithm can be at worst exponential to the size of the input formula size. Compared to extant LTL_f satisfiability algorithms, CDLSC has to invoke a massive number of SAT calls, considering that each SAT call requires an NP-complete complexity. However, those cost can successfully pay back thanks to the power of modern SAT solvers and the efficiency of on-the-fly checking as well as invariant finding, according to our experimental evaluation below.

We work through CDLSC by reusing the satisfiable instance $\phi = (\neg Tail)\mathcal{U}a \wedge (\neg Tail)\mathcal{U}(\neg a) \wedge (\neg Tail)\mathcal{U}b \wedge (\neg Tail)\mathcal{U}(\neg b) \wedge (\neg Tail)\mathcal{U}c$. First, CDLSC checks whether $Tail \wedge \mathsf{xnf}(\phi)^p$ is satisfiable (Line 1 of Algorithm 1). The answer is negative and $\{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}(\neg a)\}$ is the returned UC, which is added into $\mathcal{C}[0]$ (Line 4 of Algorithm 1). Then, CDLSC tries to check whether $\mathsf{xnf}(\phi)^p \wedge \neg(p_1 \wedge p_2)$ is satisfiable (Line 2 of Algorithm 2), where p_1, p_2 represent $\mathcal{X}((\neg Tail)\mathcal{U}a)$ and $\mathcal{X}((\neg Tail)\mathcal{U}(\neg a))$ respectively. Assume the new state $\phi' = \{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}(\neg b), (\neg Tail)\mathcal{U}c\}$ is generated from the assignment of the SAT solver. Then CDLSC checks whether ϕ' can reach a final state in 0 steps, i.e., $\mathsf{xnf}(\phi')^p \wedge Tail$ is satisfiable (Line 6 of Algorithm 2). The answer is negative and we can add the UC $\{(\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}(\neg b)\}$ to $\mathcal{C}[0]$ as well (Line 10 of Algorithm 2). Now to compute a next state of ϕ that is not included in $\mathcal{C}[0]$, the encoded Boolean formula becomes $\mathsf{xnf}(\phi)^p \wedge \neg(p_1 \wedge p_2) \wedge \neg(p_3 \wedge p_4)$ where p_3, p_4 represent $\mathcal{X}((\neg Tail)\mathcal{U}b)$ and $\mathcal{X}((\neg Tail)\mathcal{U}(\neg b))$ respectively (Line 6 of Algorithm 2). Assume the new state $\phi'' = \{(\neg Tail)\mathcal{U}a, (\neg Tail)\mathcal{U}b, (\neg Tail)\mathcal{U}c\}$ is generated from the assignment of the SAT solver. Since $\mathsf{xnf}(\phi'')^p \wedge Tail$ is satisfiable, ϕ'' is a final state and we conclude that the formula ϕ is satisfiable (Line 7 of Algorithm 2). As discussed before, there are a total of $2^5 = 32$ states in the transition system of ϕ, but CDLSC succeeds to find the answer by computing only 3 of them (including the initial state).

In the previous section, it has been shown how CDLSC accelerates the checking of satisfiable formulas in the previous section. For unsatisfiable instances, consider $\phi = (\neg Tail)\mathcal{U}a \wedge (Tail)\mathcal{R}\neg a \wedge (\neg Tail)\mathcal{U}b$. CDLSC first checks that $Tail \wedge \mathsf{xnf}(\phi)^p$ is unsatisfiable at Line 1 of Algorithm 1, where the SAT solver returns $c = \{(\neg Tail)\mathcal{U}a, Tail\mathcal{R}\neg a\}$ as the UC. So c is added into $\mathcal{C}[0]$ (Line 4 of Algorithm 1). Then CDLSC checks that $(\mathsf{xnf}(\phi) \wedge \neg\mathcal{X}(\mathcal{C}[0]))^p$ is still unsatisfiable (Line 2 of Algorithm 2), in which $c = \{(\neg Tail)\mathcal{U}a, Tail\mathcal{R}\neg a\}$ is still the UC. So c is added into $\mathcal{C}[1]$ as well (Line 20 of Algorithm 2). Since $\mathcal{C}[0] \subseteq \mathcal{C}[1]$ and according to Theorem 6, CDLSC terminates with the unsatisfiable result (Line 11 to Line 13 of Algorithm 1). In this case, CDLSC only visits one state for the whole checking process. For a more general instance like $\phi \wedge \psi$, where ψ is a large LTL_f formula, checking by CDLSC enables to achieve a significantly improvement compared to the checking by traditional tableau approach.

Summarily, CDLSC is a conflict-driven on-the-fly satisfiability checking algorithm, which successfully leads to either an earlier finding of a satisfying model, or the faster termination with the unsatisfiable result.

6. Experimental evaluation

In this section, we introduce a comprehensive evaluation among different LTL_f satisfiability checkers with a large amount of benchmark suits that are available so far, to the best of our knowledge.

6.1. Benchmark suits

Our extensive experimental evaluation, checking 9317 formulas, uses four classes of benchmark suits: 7442 LTL-as-LTL_f (since LTL formulas share the same syntax as LTL_f) that are originally collected in [34], 1700 LTL_f-Specific benchmark suits, which are common LTL_f patterns that are all satisfiable by finite traces (but not necessarily by infinite traces), 63 widely used LTL_f patterns from NASA-Boeing [35] and 112 DECLARE benchmark suits that are from the business process management dataset and used in a recent work [36]. We check both execution time and correctness; checking also correctness, as in [18], ensures we are comparing performance of tools finding the *same* results.

The LTL-as-LTL_f benchmark suits consists of 7 different formula classes from different scenarios, i.e., the classes of *acacia*, *alaska*, *anzu*, *forobots*, *rozier*, *schuppan* and *trp* formulas. As an example, Table 1 shows the different patterns from the *rozier* benchmark. Readers are referred to [34] for details of other benchmarks.

- **Random Formulas** generated as in [19], vary the number of variables {1, 2, 3}, formula length {5, …, 100}, and probability of choosing a temporal operator {0.3, 0.5, 0.7, 0.95} from the operator set $\{\neg, \vee, \wedge, \mathcal{X}, \mathcal{U}, \mathcal{R}, \mathcal{G}, \mathcal{F}, \mathcal{GF}\}$.
- **Counter Formulas** scale four, temporally complex patterns that describe large state spaces: n-bit binary counters for $1 \leq n \leq 20$ [18]. The four templates differ in variables and nesting of $\mathcal{X}$'s.
- **Pattern Formulas** encode eight scalable patterns (from [37], and are generated by code from [18]) scaling to $n = 100$.

Table 1
LTL-as-LTL$_f$ benchmark suits: the de facto standard benchmark suite for LTL satisfiability checking can also be used for LTL$_f$ satisfiability.

Name	LTL$_f$ Formalization	Answer
Random	500 formulas per set: $\{Vars \in \{1 \dots 3\}, length \in \{5 \dots 100\}\}$	varies
Counter	n bit counter, 2 vars, no carry	unsat
CounterCarry	n bit counter, 3 vars, with carry	unsat
CounterLinear	formulates linearly instead of quadratically	unsat
CounterCarryLinear	n bit counter, 3 vars, with carry and linear subformulas	unsat
$S(n)$	$\bigwedge_{i=1}^{n} \Box p_i$	sat
$E(n)$	$\bigwedge_{i=1}^{n} \Diamond p_i$	sat
$Q(n)$	$\bigwedge(\Diamond p_i \vee \Box p_{i+1})$	sat
$U(n)$	$(\dots(p_1 \,\mathcal{U}\, p_2) \,\mathcal{U}\, \dots) \,\mathcal{U}\, p_n$	sat
$U_2(n)$	$p_1 \,\mathcal{U}\, (p_2 \,\mathcal{U}\, (\dots p_{n-1} \,\mathcal{U}\, p_n)\dots)$	sat
$C_1(n)$	$\bigvee_{i=1}^{n} \Box\Diamond p_i$	sat
$C_2(n)$	$\bigwedge_{i=1}^{n} \Box\Diamond p_i$	sat
$R(n)$	$\bigwedge_{i=1}^{n}(\Box\Diamond p_i \vee \Diamond\Box p_{i+1})$	sat

The LTL$_f$-Specific benchmark suits, whose description are shown in Table 2, consist of the following patterns.

- **Random Conjunction formulas** combine 19 common LTL$_f$ formulas from [17,38] (see the first 19 patterns in Table 2) as random conjunctions in the style of [39] in two sets of 500 formulas:
  - 20 variables, varying the number of conjuncts in {10, 30, 50, 70, 100}. For each number of conjuncts, we generate 100 formulas;
  - 50 conjuncts, varying the number of variables in {10, 30, 50, 70, 100}. For each number of variables, we generate 100 formulas.
- **Pattern Formulas** 7 scalable patterns (see the last 7 patterns in Table 2) inspired by [40] up to length 100. The total number of generated formulas is 700.

In addition to those described above, we introduce two more kinds of benchmark suits from the industry, whose instance lengths are normally larger than those from LTL-as-LTL$_f$ and LTL$_f$-specific benchmarks.

- The NASA-Boeing benchmark suits consist of 63 in total real-world LTL$_f$ specifications in which
  - 49 LTL$_f$ specifications used for designing Boeing AIR 6110 wheelbraking system [42]; and
  - 14 LTL$_f$ specifications used for designing NASA NextGen air traffic control (ATC) system [43].
- The DECLARE benchmark suits consists of a total of 112 LTL$_f$ patterns that are widely used in the business process management, as shown in [36].

Based on our checking results from all testing solvers, the NASA-Boeing and DECLARE benchmark suits contain only satisfiable formulas.

6.2. Experimental setup

We implement CDLSC in the tool *aaltaf*[3] and use Minisat 2.2.0 [33] as the SAT engine. According to Algorithm 1 and 2, our tool *aaltaf* enables to use the SAT solver in an incremental way. We compare it with two extant LTL$_f$ satisfiability solvers: Aalta-finite [21] and LTL2SAT [29]. Notably, LTL2SAT utilizes Aalta-finite as the heuristic engine dedicated for LTL$_f$ formulas. As a result, we consider both the LTL2SAT performances with and without Aalta-finite enabled. We also compared with the state-of-art LTL solver Aalta-infinite [27], using the LTL$_f$-to-LTL satisfiability-preserving reduction described in [2]. As LTL satisfiability checking is reducible to model checking, as described in [18], we also compared with this reduction, using nuXmv 1.1.1 with the K-LIVE back-end [23], as an LTL$_f$ satisfiability checker. Aalta-finite, Aalta-infinite and aaltaf are ran with their default parameters. LTL2SAT is written in Java and the command to run the tool is "java -jar LTL2SAT.jar -t time formula", where "time" is the running timeout and "formula" is the LTL$_f$ formula to be checked. In particular, LTL2SAT also provides the "-u" option to disable the invoke of Aalta-finite. To run K-LIVE in nuXmv, we utilize the following (nuXmv) commands:

```
read_model
flatten_hierarchy
encode_variables
build_boolean_model
check_ltlspec_klive -d
quit
```

[3] https://github.com/lijwen2748/aaltaf.

Table 2
LTL_f**-Specific Benchmark suits**: formulas specifically designed for LTL_f from previous works, adapted to be benchmark suits for our experiments. To create benchmark suits from Declare Templates, we substituted variables for branches, then created formula-generating scripts. We choose these as benchmark suits because a) they are common patterns; b) they are all satisfiable in LTL_f (but not necessarily LTL), allowing us to check correctness of sat/unsat results.

Name	LTL_f Formalization	Description	Answer
Declare Patterns		From [17,38]	sat*
Existence	$\Diamond a$	a must be executed at least once	sat*
Absence 2	$\neg\Diamond(a \wedge \Diamond a)$	a can be executed at most once	sat*
Choice	$\Diamond a \vee \Diamond b$	a or b must be executed	sat*
Exclusive Choice	$(\Diamond a \vee \Diamond b) \wedge \neg(\Diamond a \wedge \Diamond b)$	Either a or b must be executed, but not both	sat*
Resp. existence	$\Diamond a \rightarrow \Diamond b$	If a is executed, then b must be executed as well	sat*
Coexistence	$(\Diamond a \rightarrow \Diamond b) \wedge (\Diamond b \rightarrow \Diamond a)$	Either a and b are both executed, or none of them is executed	sat*
Response	$\Box(a \rightarrow \Diamond b)$	Every time a is executed, b must be executed afterwards	sat*
Precedence	$\neg b \mathcal{W} a$	b can be executed only if a has been executed before	sat*
Succession	$\Box(a \rightarrow \Diamond b) \wedge (\neg b \mathcal{W} a)$	b must be executed after a, and a must precede b	sat*
Alt. Response	$\Box(a \rightarrow \mathcal{X}(\neg a U b))$	Every a must be followed by b, without any other b in between	sat*
Alt. Precedence	$(\neg b \mathcal{W} a) \wedge \Box(b \rightarrow \mathcal{X}(\neg b \mathcal{W} a))$	Every b must be preceded by a, without any other b in between	sat*
Alt. Succession	$\Box(a \rightarrow \mathcal{X}(\neg a U b)) \wedge (\neg b \mathcal{W} a) \wedge \Box(b \rightarrow \mathcal{X}(\neg \mathcal{W} a))$	Combination of alternate response and alternate precedence	sat*
Chain Response	$\Box(a \rightarrow \mathcal{X} b)$	If a is executed then b must be executed next	sat*
Chain Precedence	$\Box(\mathcal{X} b \rightarrow a)$	Task b can be executed only immediately after a	sat*
Chain Succession	$\Box(a \leftrightarrow \mathcal{X} b)$	Tasks a and b must be executed next to each other	sat*
Not Coexistence	$\neg(\Diamond a \wedge \Diamond b)$	Only one among tasks a and b can be executed, but not both	sat*
Neg. Succession	$\Box(a \rightarrow \neg\Diamond b)$	Task a cannot be followed by b, and b cannot be preceded by a	sat*
Neg. Chain Succession	$\Box(a \leftrightarrow \mathcal{X}\neg b)$	Tasks a and b cannot be executed next to each other	sat*
End	$\Diamond(a \wedge \neg\mathcal{X}(a \vee \neg a))$	a occurs *last*; translated to LTL_f from [41]	sat*
Declare Templates		formula-generating code inspired by constraints from [40]	sat*
RespondedExistence (n)	$\Diamond x \rightarrow \Diamond\left(\bigvee_{i=1}^{n} y_i\right)$		sat*
Response (n)	$\Box\left(x \rightarrow \Diamond\left(\bigvee_{i=1}^{n} y_i\right)\right)$		sat*
AlternateResponse(n)	$\Box\left(x \rightarrow \mathcal{X}\left(\neg x \mathcal{U} \bigvee_{i=1}^{n} y_i\right)\right)$		sat*
ChainResponse(n)	$\Box\left(x \rightarrow \mathcal{X}\left(\bigvee_{i=1}^{n} y_i\right)\right)$		sat*
Precedence(n)	$(\neg x)\mathcal{W}\left(\bigvee_{i=1}^{n} y_i\right)$		sat*
AlternatePrecedence(n)	$Precedence(n) \wedge \Box(x \rightarrow \mathcal{X} Precedence(n))$		sat*
ChainPrecedence(n)	$\Box\left((\mathcal{X} x) \rightarrow \left(\bigvee_{i=1}^{n} y_i\right)\right)$		sat*

We ran the experiments on a RedHat 6.0 cluster with 2304 processor cores in 192 nodes (12 processor cores per node), running at 2.83 GHz with 48 GB of RAM per node. Each tool executed on a dedicated node with a 8 GB memory and timeout of 1 hour, measuring execution time with Unix `time`. Those timeout, out of memory or running-error test cases will be assigned a penalty of 1 hour. We check the correctness by comparing the results from different solvers and it turns out the results from all solvers are consistent excluding those timeouts.

All artifacts for enabling reproducibility, including benchmark formulas and their generators, are available from the paper website at https://drive.google.com/open?id=1eOYGvm3C8sQ-9iyfZ8qx42K54hgrFNTC.

6.3. Results

Fig. 4 shows the results for LTL_f satisfiability checking on LTL-as-LTL_f benchmark suits. In summary, CDLSC outperforms all other approaches. CDLSC performs best with a total time cost of 931 minutes, while K-LIVE in nuXmv performs the second-best with the total time cost of 1158 minutes. CDLSC checks the LTL_f formula directly, while K-LIVE must take the input of the LTL formula translated from the LTL_f formula. As a result, KLIVE may take extra cost, e.g., finding a satisfying lasso for the model, to the satisfiability checking. Meanwhile, CDLSC can benefit from the heuristics dedicated for LTL_f that are proposed in [21]. Compared to the results shown in [31], nuXmv is able to decrease the performance gap between CDLSC and K-LIVE. That is because we extend the running timeout from 1 minute to 1 hour, which makes nuXmv solves more instances but no change for CDLSC. This phenomenon indicates that nuXmv has a better scalable performance than our implementation *aaltaf*. However, it should not be surprising due to the fact that the nuXmv is a mature software which has been developing for decades. Also, Aalta-infinite for LTL_f does not perform better than model checking, using K-LIVE for, in contrast to the results for LTL in [27]. We conjecture that, since Aalta-infinite is a dedicated solver for LTL

Table 3
A summary of the experimental results on different classes in LTL-as-LTL$_f$ benchmark suits. Each cell of the table has the format of $\langle n/t \rangle$, where n represents the number of unsolved instances within 1 hour and t means the corresponding time cost (seconds).

	Number	Aalta-infinite	Aalta-finite	LTL2SAT	nuXmv	CDLSC	CDLSC without heuristics
acacia	140	1/3665	0/1	62/225857	0/35	0/1	0/433
alaska	280	2/7573	0/2	86/315354	0/306	3/10806	7/32606
anzu	222	0/48	106/389365	88/322816	0/890	4/14495	4/20435
rozier	4640	74/271749	33/120345	220/6027477	8/30347	0/85	10/36795
schuppan	144	37/135303	2/7371	0/1311	10/43904	8/29453	20/72632
trp	1940	2/8441	113/412790	0/2324	0/1181	0/64	33/133527
forobots	76	0/2	38/139080	0/592	0/17	0/1	0/213
Total	7442	116/426781	292/1068954	456/6895731	18/69480	15/55905	74/296641

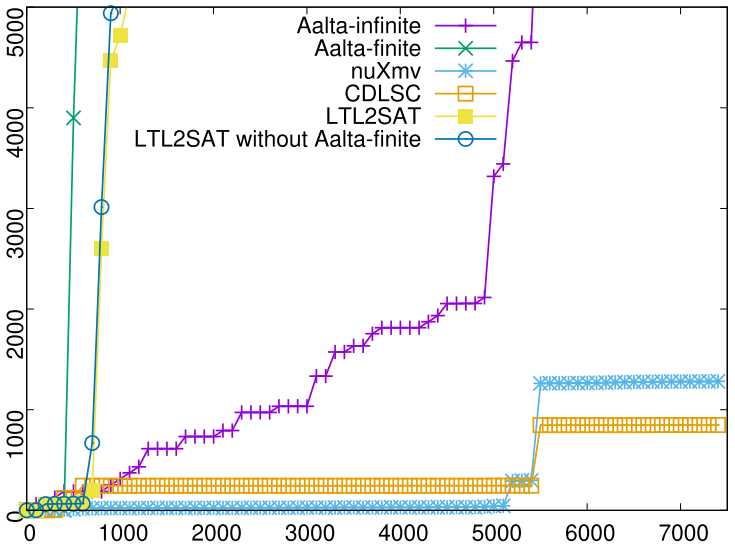

Fig. 4. Result for LTL$_f$ Satisfiability Checking on LTL-as-LTL$_f$ benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the accumulated checking time (minute).

formulas, its LTL-specific heuristics do not apply well to LTL$_f$ formulas. Finally, the performance of LTL2SAT (with and without Aalta-finite) is highly tied to its performance for unsatisfiability checking as most of the timeout cases for LTL2SAT are unsatisfiable. For Aalta-finite, its performance is restricted by the heavy cost of the Tableau Construction.

Since the LTL-as-LTL$_f$ benchmark suits consist of a diverse of different classes, we summarize the corresponding statistics for each class in Table 3. Although CDLSC has the best overall performance, different solvers can perform best on certain classes. For example, Aalta-infinite performs best on the "anzu" class and Aalta-finite performs best on the "acacia" and "alaska" classes. LTL2SAT (with Aalta-finite) does not perform best on any class listed in the table, and LTL2SAT without Aalta-finite performs even worse than LTL2SAT (with Aalta-finite) on the LTL-as-LTL$_f$ benchmarks, whose results are omitted in Table 3. The question on how to conduct a portfolio solver that can perform best on all kinds of classes thus raises up, which we leave for future work. Also, CDLSC integrates the heuristics presented in Aalta-finite, and Table 3 shows the performance on CDLSC with and without heuristics. As shown in the table, the heuristics in our previous work plays an important role on the satisfiability checking: CDLSC with heuristics solves 53 more instances than CDLSC without heuristics. However with no doubt, the new SAT-based approach in CDLSC accelerates the satisfiability checking significantly as well, from the results comparison between CDLSC and Aalta-finite.

Fig. 5 and Fig. 6 show the comparing results on satisfiable and unsatisfiable LTL-as-LTL$_f$ benchmark suits respectively. LTL2SAT with and without Aalta-finite produce the best performance on the satisfiable formulas, though CDLSC and nuXmv perform similarly. Recall that LTL2SAT implements a Bounded-Model-Checking (BMC) [28] style for checking; it can perform best on the satisfiable benchmark suits since BMC is shown superior to finding bugs (checking satisfiability) [44]. The conjecture that LTL2SAT integrates Aalta-finite is to help to solve unsatisfiable formulas, which is confirmed from Fig. 6 that the performance of LTL2SAT is better than LTL2SAT without Aalta. We show more evidences below. Obviously, CDLSC performs best on checking unsatisfiable formulas, followed by nuXmv and Aalta-infinite. As a result, CDLSC performs best on unsatisfiable and almost best on satisfiable formulas, which make it the best approach on the overall performance.

A case-to-case performance comparison between LTL2SAT with and without Aalta on satisfiable and unsatisfiable formulas is shown in Fig. 7 and Fig. 8 respectively. These two figures show the clearer evidences that while LTL2SAT without Aalta-finite performs better than LTL2SAT (with Aalta-finite) on checking satisfiable formulas, LTL2SAT (with Aalta-finite) does have a better performance on checking unsatisfiable formulas than LTL2SAT without Aalta-finite. The results in these

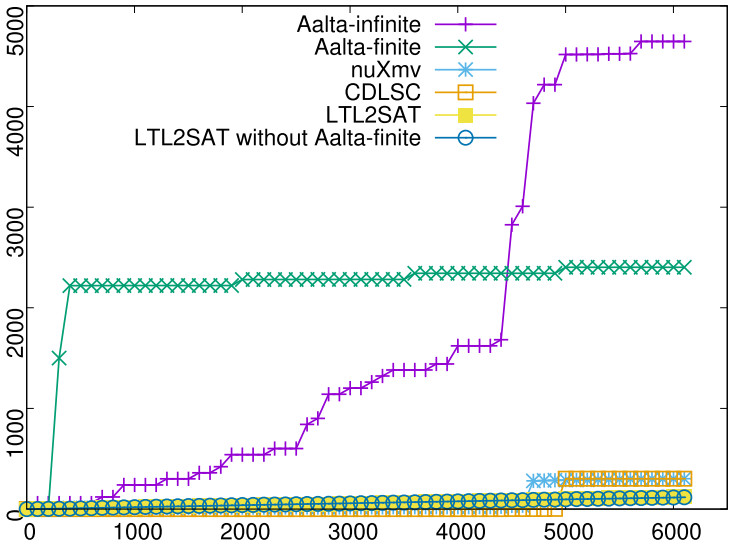

Fig. 5. Result for LTL_f Satisfiability Checking on Satisfiable LTL-as-LTL_f benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the accumulated checking time (minute).

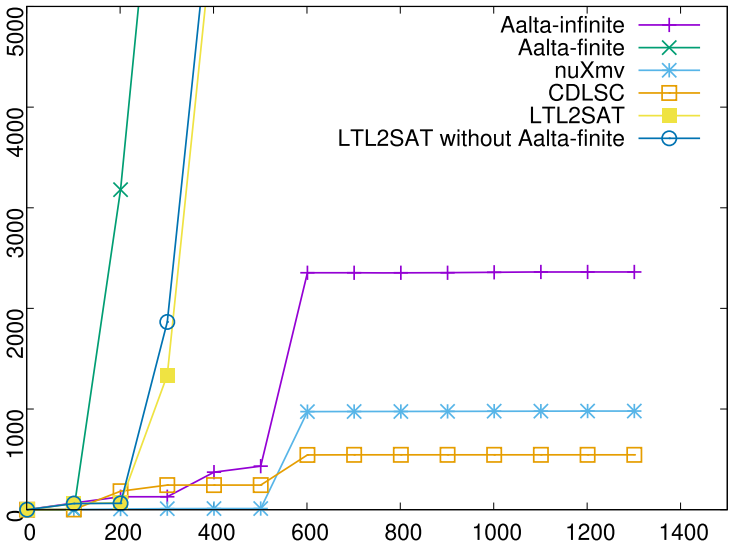

Fig. 6. Result for LTL_f Satisfiability Checking on Unsatisfiable LTL-as-LTL_f benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the accumulated checking time (minute).

two figures explain the consistency between the performance of LTL2SAT and the theoretic foundation behind the tool: the inherent BMC-style checking strategy enables the tool to perform well on satisfiable formulas, but meanwhile, additional heuristics (e.g. from Aalta-finite) have to be imported to accelerate the tool's checking on unsatisfiable formulas.[4]

Table 4 shows the results for LTL_f-specific experiments. Column 1 shows the types of LTL_f formulas under test and Columns 2-7 show the checking times by formula types in seconds. The dedicated LTL_f solvers perform extremely fast on the seven scalable pattern formulas (Column 3 and 7), because their heuristics work well on these patterns. For the difficult random conjunction benchmark suits, which mainly consists of unsatisfiable formulas, CDLSC still outperforms all other solvers. Notably, CDLSC solves all instances in the LTL_f-specific benchmarks.

Finally, we test more satisfiable benchmark suits to evaluate the performance among different solvers. Fig. 9 shows the results on checking the 63 satisfiable NASA-Boeing benchmark suits. CDLSC performs best while LTL2SAT without Aalta and Aalta-infinite perform slightly less than CDLSC on checking these challenging benchmark suits. These three approaches outperform all others. Fig. 10 shows the results on checking the 112 satisfiable DECLARE benchmark suits. Surprisingly,

[4] It is well known that BMC is good at checking satisfiability but not at checking unsatisfiability.

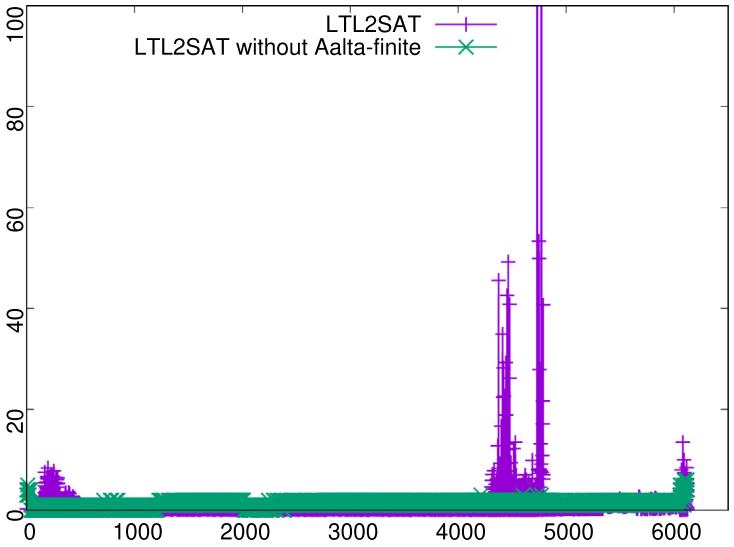

Fig. 7. Comparison between LTL2SAT with and without Aalta-finite on Satisfiable LTL-as-LTL$_f$ benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the checking time (second).

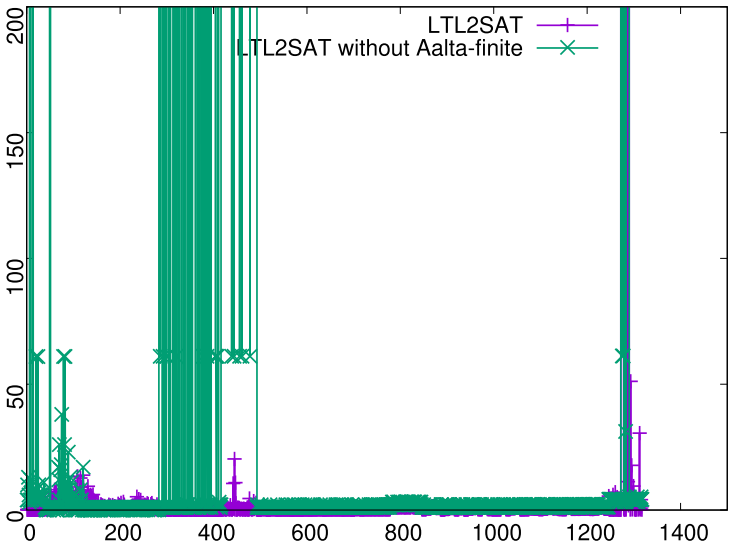

Fig. 8. Comparison between LTL2SAT with and without Aalta-finite on Unsatisfiable LTL-as-LTL$_f$ benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the checking time (second).

Table 4
Results for LTL$_f$ Satisfiability Checking on LTL_f-specific benchmark suits. The time unit for the cost of each pattern is second.

Type	NuXmv	Aalta-finite	Aalta-infinite	LTL2SAT	LTL2SAT without Aalta-finite	CDLSC
Alternate Response	134	1	48	123	76	3
Alternate Precedence	154	3	70	380	54	4
Chain Precedence	127	2	45	83	43	2
Chain Response	79	1	41	49	54	2
Precedence	132	2	14	124	67	1
Responded Existence	130	1	14	327	111	1
Response	155	1	41	53	54	2
Random Conjunction	1669	19564	4443	20477	3897	115

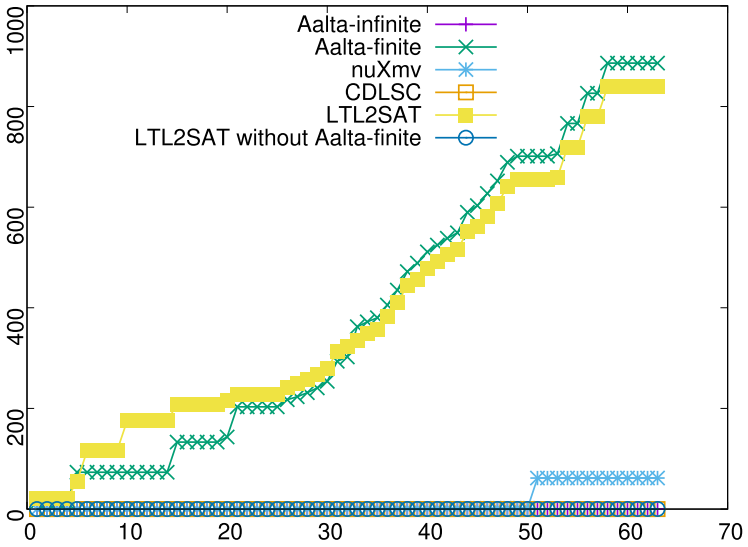

Fig. 9. Result for LTL_f Satisfiability Checking on the NASA-Boeing benchmark suits. The X axis represents the number of benchmark suits, and the Y axis is the accumulated checking time (minute).

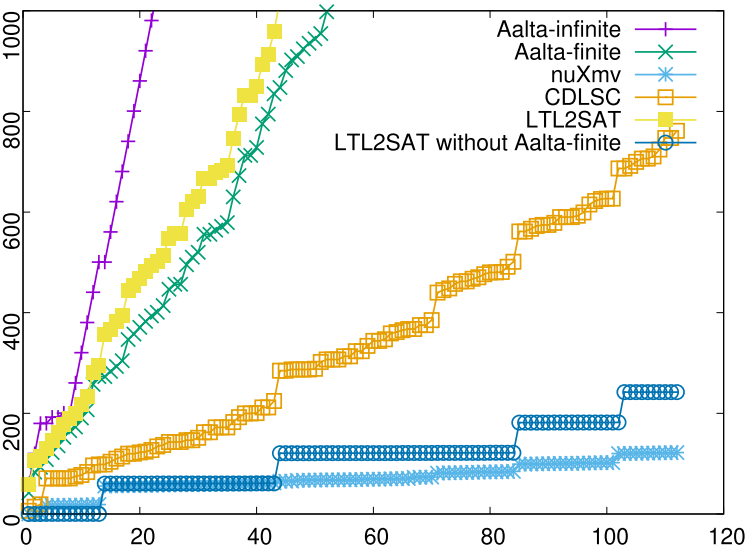

Fig. 10. Result for LTL_f Satisfiability Checking on the DECLARE benchmark suits [36]. The X axis represents the number of benchmark suits, and the Y axis is the accumulated checking time (minute).

nuXmv performs best on checking these satisfiable benchmark suits, followed by LTL2SAT without Aalta and CDLSC. It indicates that there is not such an approach that can dominate all satisfiable benchmark suits. So far, we can find only one unsatisfiable benchmark (from LTL-as-LTL_f) and CDLSC performs best on these unsatisfiable formulas.

7. Discussion and concluding remarks

There are two ways to apply Bounded Model Checking (BMC) to LTL_f satisfiability checking. The first one is to check the satisfiability of the LTL formula from the input LTL_f formula. Since [27] showed this approach performs worse than K-LIVE, CDLSC outperforming K-LIVE implies that CDLSC also outperforms BMC. The second approach is to check the satisfiability of the LTL_f formula ϕ directly, by unrolling ϕ iteratively. In the worst case, BMC can terminate (with UNSAT) once the iteration reaches the upper bound. This is exactly what is implemented in LTL2SAT [29].

Our experiments demonstrate that CDLSC outperforms Aalta-infinite and K-LIVE, which are designed for LTL satisfiability checking, showing the advantage of a dedicated algorithm for LTL_f. Notably, CDLSC maintains a conflict sequence, which is similar to the state-of-art model checking technique IC3 [25]. CDLSC does not require the conflict sequence to be monotone, and simply use the UC from SAT solvers to update the sequence. Meanwhile, IC3 requires the sequence to be strictly

monotone, and has to compute its dedicated MIC (Minimal Inductive Core) to update the sequence. In fact, CDLSC is motivated from a more recent model checking algorithm CAR (Complementary Approximate Reachability) [45]. We conclude that CDLSC outperforms other existing approaches for LTL_f satisfiability checking.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported by NASA ECF NNX16AR57G, NSF CAREER Awards CNS-1552934, CNS-1664356, NSF grants CCF-1319459, CCF-1704883 and IIS-1527668, IIS-1830549, NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering," NSFC projects No. 6157297, 61632005, 61532019, and China HGJ project No. 2017ZX01038102-002.

Appendix A

A.1. Proof of Theorem 1

We first introduce the following lemmas that are useful for the proof.

Lemma 5. *If $tnf(\phi)$ is satisfiable, there is a non-empty finite trace ξ such that $\neg Tail \in \xi[i]$ for $0 \leq i < |\xi| - 1$, $Tail \in \xi[|\xi| - 1]$ and $\xi \models tnf(\phi)$.*

Proof. Since $\mathsf{tnf}(\phi)$ is satisfiable, there is a non-empty finite trace ξ' such that $\xi' \models \mathsf{tnf}(\phi)$. Recall that $\mathsf{tnf}(\phi)$ has the form of $t(\phi) \wedge \mathcal{F}Tail$, so $\xi' \models \mathsf{tnf}(\phi)$ implies $\xi' \models t(\phi)$ and there is $0 \leq k < |\xi'|$ such that $Tail \in \xi'[k]$ and $Tail \notin \xi'[j]$ for every $j < k$. Actually, we may set k to be the position where $Tail$ first appears in ξ'. We define $tp(\xi') = \xi'[0]\xi'[1]\ldots\xi'[k]$, and first prove that $\xi' \models t(\phi)$ implies $tp(\xi') \models t(\phi)$. Let $\xi = tp(\xi')$, and we prove by induction over the type of ϕ that $\xi \models t(\phi)$.

1. If $\phi = \mathsf{tt}$, then $t(\phi) = \mathsf{tt}$ and of course $\xi \models t(\phi)$;
2. If $\phi = l$ is a literal, then $t(\phi) = l$ and $\xi' \models t(\phi)$ implies $l \in \xi'[0] = \xi[0]$. Therefore, $\xi \models t(\phi)$;
3. If $\phi = \phi_1 \wedge \phi_2$, then $t(\phi) = t(\phi_1) \wedge t(\phi_2)$, and $\xi' \models t(\phi)$ implies $\xi' \models t(\phi_1)$ and $\xi' \models t(\phi_2)$. By hypothesis assumption, $\xi' \models t(\phi_1)$ implies $\xi \models t(\phi_1)$ and $\xi' \models t(\phi_2)$ implies $\xi \models t(\phi_2)$. So $\xi \models t(\phi)$ is true. If $\phi = \phi_1 \vee \phi_2$, the proof is similar;
4. If $\phi = \mathcal{X}\psi$, then $t(\phi) = \neg Tail \wedge \mathcal{X}(t(\psi))$, and $\xi' \models t(\phi)$ implies that $Tail \notin \xi'[0]$ and $\xi'_1 \models t(\psi)$. Let $\xi_1 = tp(\xi'_1)$. By hypothesis assumption, $\xi'_1 \models t(\psi)$ implies $\xi_1 \models t(\psi)$ is true. Moreover, because $Tail \notin \xi'[0]$, $\xi = tp(\xi') = \xi'[0] \cdot tp(\xi'_1) = \xi'[0] \cdot \xi_1$ from its definition. As a result, $\xi \models t(\phi)$ is true;
5. If $\phi = \mathcal{N}\psi$, then $t(\phi) = Tail \vee \mathcal{X}(t(\psi)) = Tail \vee (\neg Tail \wedge \mathcal{X}(t(\psi)))$, and $\xi' \models t(\phi)$ implies that $Tail \in \xi'[0]$ or $\xi' \models \neg Tail \wedge \mathcal{X}(t(\psi))$. In the first case, $\xi = \xi'[0]$ and obviously $\xi \models t(\phi)$. For the second case, the proof is the same as that if $\phi = \mathcal{X}\psi$;
6. If $\phi = \phi_1 \mathcal{U} \phi_2$, then $t(\phi) = (\neg Tail \wedge t(\phi_1)) \mathcal{U} t(\phi_2)$, and $\xi' \models t(\phi)$ implies that there is $0 \leq i < |\xi'|$ such that $\xi'_i \models t(\phi_2)$ and for every $0 \leq j < i$ it holds $\xi'_j \models \neg Tail \wedge t(\phi_1)$. As a result, we have that $\xi = tp(\xi') = \xi'[0]\ldots\xi'[i-1] \cdot tp(\xi'_i)$, and thus $\xi_i = tp(\xi'_i)$ and $\xi_j = \xi'[j]\ldots\xi'[i-1] \cdot tp(\xi'_i) = tp(\xi'_j)$. By hypothesis assumption, $\xi'_i \models t(\phi_2)$ implies $\xi_i \models t(\phi_2)$ and $\xi'_j \models \neg Tail \wedge t(\phi_1)$ implies $\xi_j \models \neg Tail \wedge t(\phi_1)$. As a result, $\xi \models t(\phi)$ is true;
7. If $\phi = \phi_1 \mathcal{R} \phi_2$, then $t(\phi) = (Tail \vee t(\phi_1)) \mathcal{R} t(\phi_2)$, and $\xi' \models t(\phi)$ implies that for all $0 \leq i < |\xi'|$ it holds that, $\xi'_i \models t(\phi_2)$ or there is $0 \leq j < i$ such that $\xi'_j \models Tail \vee t(\phi_1)$. Since $\xi = tp(\xi')$, so $\xi_i = tp(\xi'_i)$ for $0 \leq i < |\xi|$. By hypothesis assumption, $\xi'_i \models t(\phi_2)$ implies $\xi_i \models t(\phi_2)$ for every $0 \leq i < |\xi| - 1$. Moreover, it is true that $Tail \in \xi[|\xi| - 1]$, which implies $\xi[|\xi| - 1] \models Tail \vee t(\phi_1)$. Therefore, we have that $\xi \models t(\phi)$.

Because $tp(\xi') \models t(\phi)$ is true, and $tp(\xi') \models \mathcal{F}Tail$ is obviously true, we prove finally that $\xi = tp(\xi') \models \mathsf{tnf}(\phi)$. □

Lemma 6. *Let ξ, ξ' be two non-empty finite traces satisfying $|\xi| = |\xi'|$ and $\xi'[i] = \xi[i]$ for $0 \leq i < |\xi| - 1$ as well as $\xi'[|\xi| - 1] = \xi[|\xi| - 1] \cup \{Tail\}$. Then $\xi \models \phi$ iff $\xi' \models tnf(\phi)$.*

Proof. We prove by induction over the type of ϕ.

1. If ϕ is tt, ff or a literal l, obviously $\xi \models \phi$ holds iff $\xi' \models \mathsf{tnf}(\phi)$ holds;
2. If $\phi = \mathcal{X}\psi$, then $\xi \models \phi$ holds iff $|\xi| > 1$ and $\xi_1 \models \psi$ holds. By hypothesis assumption, $\xi_1 \models \psi$ holds iff $\xi'_1 \models \mathsf{tnf}(\psi)$ holds, and $\xi'_1 \models \mathsf{tnf}(\psi)$ holds iff $\xi' \models \neg Tail \wedge \mathcal{X}(\mathsf{tnf}(\psi))$ holds (because $\neg Tail \in \xi'[0]$). As a result, we have the following equations:

$$\begin{aligned}
&\xi \models \mathcal{X}\psi \\
&\Leftrightarrow \quad \xi' \models \neg Tail \wedge \mathcal{X}(\mathsf{tnf}(\psi)) \\
&\Leftrightarrow \quad \xi' \models \neg Tail \wedge \mathcal{X}(t(\psi) \wedge \mathcal{F}Tail) \\
&\Leftrightarrow \quad \xi' \models \neg Tail \wedge \mathcal{X}(t(\psi)) \wedge \mathcal{F}Tail
\end{aligned}$$

Since $\mathsf{tnf}(\phi) = \neg Tail \wedge \mathcal{X}(t(\psi)) \wedge \mathcal{F}Tail$, so $\xi \models \phi$ iff $\xi' \models \mathsf{tnf}(\phi)$ is true;

3. If $\phi = \mathcal{N}\psi$, then $\xi \models \phi$ iff $|\xi| = 1$ or $|\xi| > 1$ and $\xi_1 \models \psi$ holds. We only prove the case when $|\xi| = 1$ and the other case has been proved above. Due to the facts that $|\xi| = 1$, the construction of ξ', and $\mathsf{tnf}(\phi) = (Tail \vee \mathsf{tnf}(\mathcal{X}(\psi))) \wedge \mathcal{F}Tail$, it is true that $\xi \models \phi$ holds iff $\xi' \models \mathsf{tnf}(\phi)$ holds;
4. If $\phi = \phi_1 \wedge \phi_2$, then $\xi \models \phi$ holds iff both $\xi \models \phi_1$ and $\xi \models \phi_2$ hold. By hypothesis assumption, we have $\xi \models \phi_1$ holds iff $\xi' \models \mathsf{tnf}(\phi_1)$ holds, and $\xi \models \phi_2$ holds iff $\xi' \models \mathsf{tnf}(\phi_2)$ holds. As a result, $\xi \models \phi$ holds iff $\xi' \models \mathsf{tnf}(\phi_1) \wedge \mathsf{tnf}(\phi_2) = t(\phi_1) \wedge t(\phi_2) \wedge \mathcal{F}Tail = t(\phi_1 \wedge \phi_2) \wedge \mathcal{F}Tail = \mathsf{tnf}(\phi_1 \wedge \phi_2)$ holds;
5. If $\phi = \phi_1 \vee \phi_2$, then $\xi \models \phi$ holds iff $\xi \models \phi_1$ or $\xi \models \phi_2$ holds. By hypothesis assumption, we have $\xi \models \phi_1$ holds iff $\xi' \models \mathsf{tnf}(\phi_1)$ holds, or $\xi \models \phi_2$ holds iff $\xi' \models \mathsf{tnf}(\phi_2)$ holds. As a result, $\xi \models \phi$ holds iff $\xi' \models \mathsf{tnf}(\phi_1) \vee \mathsf{tnf}(\phi_2) = (t(\phi_1) \vee t(\phi_2)) \wedge \mathcal{F}Tail = t(\phi_1 \vee \phi_2) \wedge \mathcal{F}Tail = \mathsf{tnf}(\phi_1 \vee \phi_2)$ holds;
6. If $\phi = \phi_1 \mathcal{U} \phi_2$, then $\xi \models \phi$ holds iff there exists $0 \leq i < |\xi|$ such that $\xi_i \models \phi_2$, and for every $0 \leq j < i$ it holds that $\xi_j \models \phi_1$. By hypothesis assumption, $\xi_i \models \phi_2$ holds iff $\xi'_i \models \mathsf{tnf}(\phi_2)$ holds, and moreover, $\xi_j \models \phi_1$ holds iff $\xi'_j \models \mathsf{tnf}(\phi_1)$ holds. Because of $0 \leq j < i$ and $0 \leq i < |\xi|$, j does not equal to $|\xi| - 1$, which means $\neg Tail \in \xi'[j]$. As a result, $\xi'[j] \models \neg Tail \wedge \mathsf{tnf}(\phi_1)$. Therefore, $\xi'_i \models \phi_2$ holds and for every $0 \leq j < i$, $\xi'_j \models \neg Tail \wedge \mathsf{tnf}(\phi_1)$ is true, which means $\xi' \models (\neg Tail \wedge \mathsf{tnf}(\phi_1)) \mathcal{U} \mathsf{tnf}(\phi_2)$ is true. Finally, we have

$$\begin{aligned}
&\xi \models \phi_1 \mathcal{U} \phi_2 \\
&\Leftrightarrow \xi' \models (\neg Tail \wedge \mathsf{tnf}(\phi_1)) \mathcal{U} \mathsf{tnf}(\phi_2) \\
&\Leftrightarrow \xi' \models (\neg Tail \wedge t(\phi_1) \wedge \mathcal{F}Tail) \mathcal{U} (t(\phi_2) \wedge \mathcal{F}Tail) \\
&\Leftrightarrow \xi' \models (\neg Tail \wedge t(\phi_1)) \mathcal{U} t(\phi_2) \wedge \mathcal{F}Tail \\
&\Leftrightarrow \xi' \models \mathsf{tnf}(\phi)
\end{aligned}$$

7. If $\phi = \phi_1 \mathcal{R} \phi_2$, then $\xi \models \phi$ holds iff for every $0 \leq i < |\xi|$ it holds $\xi_i \models \phi_2$, or there exists $0 \leq j \leq i$ such that $\xi_j \models \phi_1$ holds. By hypothesis assumption, $\xi_i \models \phi_2$ holds iff $\xi'_i \models \mathsf{tnf}(\phi_2)$ holds, and, $\xi_j \models \phi_1$ holds iff $\xi'_j \models \mathsf{tnf}(\phi_1)$ holds. Therefore, it is true that $\xi \models \phi$ holds iff $\xi' \models \mathsf{tnf}(\phi_1) \mathcal{R} \mathsf{tnf}(\phi_2)$ holds. Because of $0 \leq j \leq i$ and $0 \leq i < |\xi|$, so $Tail \in \xi'[j]$ is true when $j = |\xi'| - 1$. As a result, $\xi'_j \models Tail \vee \mathsf{tnf}(\phi_1)$. Therefore, $\xi'_i \models \phi_2$ holds for every $0 \leq i < |\xi'|$, or there exists $0 \leq j \leq i$ such that $\xi'_j \models Tail \vee \mathsf{tnf}(\phi_1)$ is true, which means $\xi' \models (Tail \vee \mathsf{tnf}(\phi_1)) \mathcal{R} \mathsf{tnf}(\phi_2)$ is true. Finally, we have

$$\begin{aligned}
&\xi \models \phi_1 \mathcal{R} \phi_2 \\
&\Leftrightarrow \xi' \models \mathsf{tnf}(\phi_1) \mathcal{R} \mathsf{tnf}(\phi_2) \\
&\Leftrightarrow \xi' \models (t(\phi_1) \wedge \mathcal{F}Tail) \mathcal{R} (t(\phi_2) \wedge \mathcal{F}Tail) \\
&\Leftrightarrow \xi' \models t(\phi_1) \mathcal{R} t(\phi_2) \wedge \mathcal{F}Tail \\
&\Leftrightarrow \xi' \models (Tail \vee t(\phi_1)) \mathcal{R} t(\phi_2) \wedge \mathcal{F}Tail \\
&\Leftrightarrow \xi' \models \mathsf{tnf}(\phi)
\end{aligned}$$

The proof is done. □

We are ready now to prove Theorem 1.

Proof. ($\Rightarrow$) If ϕ is satisfiable, there is a non-empty finite trace ξ such that $\xi \models \phi$. From Lemma 6, we know that there is a corresponding finite trace ξ' satisfying $|\xi| = |\xi'|$ and $\xi'[i] = \xi[i]$ for $0 \leq i < |\xi| - 1$ as well as $\xi'[|\xi| - 1] = \xi[|\xi| - 1] \cup \{Tail\}$ such that $\xi' \models \mathsf{tnf}(\phi)$. So $\mathsf{tnf}(\phi)$ is satisfiable.

($\Leftarrow$) If $\mathsf{tnf}(\phi)$ is satisfiable, there is a finite trace ξ' satisfying $Tail \notin \xi'[i]$ for $0 \leq i < |\xi| - 1$ and $Tail \in \xi'[|\xi| - 1]$ such that $\xi' \models \mathsf{tnf}(\phi)$, from Lemma 5. Moreover, according to Lemma 6, there is a corresponding finite trace satisfying $|\xi| = |\xi'|$ and $\xi[i] = \xi'[i]$ for $0 \leq i < |\xi| - 1$ as well as $Tail \notin \xi[|\xi| - 1]$ such that $\xi \models \phi$. So ϕ is satisfiable. □

References

[1] K. Claessen, N. Sörensson, A liveness checking algorithm that counts, in: FMCAD, IEEE, 2012, pp. 52–59.
[2] G. De Giacomo, M. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: IJCAI, AAAI Press, 2013, pp. 2000–2007.

[3] K. Rozier, Linear temporal logic symbolic model checking, Comput. Sci. Rev. 5 (2) (2011) 163–203, https://doi.org/10.1016/j.cosrev.2010.06.002.
[4] M. Vardi, Automata-theoretic model checking revisited, in: VMCAI, in: LNCS, vol. 4349, Springer, 2007, pp. 137–150.
[5] E. Bartocci, R. Bloem, D. Nickovic, F. Roeck, A counting semantics for monitoring LTL specifications over finite traces, in: H. Chockler, G. Weissenbacher (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2018, pp. 547–564.
[6] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D. Van Campenhout, Reasoning with temporal logic on truncated paths, in: Proc. 15th Int'l Conf. on Computer Aided Verification, in: Lecture Notes in Computer Science, vol. 2725, Springer, 2003, pp. 27–39.
[7] F. Bacchus, F. Kabanza, Planning for temporally extended goals, Ann. Math. Artif. Intell. 22 (1998) 5–27.
[8] G. De Giacomo, M. Vardi, Automata-theoretic approach to planning for temporally extended goals, in: Proc. European Conf. on Planning, in: Lecture Notes in AI, vol. 1809, Springer, 1999, pp. 226–238.
[9] D. Calvanese, G. De Giacomo, M. Vardi, Reasoning about actions and planning in LTL action theories, in: Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 2002, pp. 593–602.
[10] F. Patrizi, N. Lipoveztky, G. De Giacomo, H. Geffner, Computing infinite plans for LTL goals using a classical planner, in: IJCAI, AAAI Press, 2011, pp. 2003–2008.
[11] A. Camacho, J. Baier, C. Muise, A. McIlraith, Bridging the gap between LTL synthesis and automated planning, Tech. rep., U. Toronto, 2017, http://www.cs.toronto.edu/~acamacho/papers/camacho-genplan17.pdf.
[12] F. Bacchus, F. Kabanza, Using temporal logic to express search control knowledge for planning, Artif. Intell. 116 (1–2) (2000) 123–191.
[13] A. Gabaldon, Precondition control and the progression algorithm, in: KR, AAAI Press, 2004, pp. 634–643.
[14] M. Bienvenu, C. Fritz, S. McIlraith, Planning with qualitative temporal preferences, in: KR, Lake District, UK, 2006, pp. 134–144.
[15] M. Bienvenu, C. Fritz, S.A. McIlraith, Specifying and computing preferred plans, Artif. Intell. 175 (7C8) (2011) 1308–1345.
[16] S. Sohrabi, J.A. Baier, S.A. McIlraith, Preferred explanations: theory and generation via planning, in: AAAI, 2011, pp. 261–267.
[17] G. De Giacomo, R.D. Masellis, M. Montali, Reasoning on LTL on finite traces: insensitivity to infiniteness, in: AAAI, 2014, pp. 1027–1033.
[18] K. Rozier, M. Vardi, LTL satisfiability checking, in: SPIN, in: LNCS, vol. 4595, Springer, 2007, pp. 149–167.
[19] K. Rozier, M. Vardi, A multi-encoding approach for LTL symbolic satisfiability checking, in: Int'l Symp. on Formal Methods, in: LNCS, vol. 6664, Springer, 2011, pp. 417–431.
[20] K. Rozier, M. Vardi, LTL satisfiability checking, Int. J. Softw. Tools Technol. Transf. 12 (2) (2010) 123–137.
[21] J. Li, L. Zhang, G. Pu, M.Y. Vardi, J. He, LTL_f satisfiability checking, in: ECAI, 2014, pp. 91–98.
[22] S. Malik, L. Zhang, Boolean satisfiability from theoretical hardness to practical success, Commun. ACM 52 (8) (2009) 76–82.
[23] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The NuXMV symbolic model checker, in: CAV, 2014, pp. 334–342.
[24] Y. Vizel, G. Weissenbacher, S. Malik, Boolean satisfiability solvers and their applications in model checking, Proc. IEEE 103 (11) (2015) 2021–2035.
[25] A. Bradley, SAT-based model checking without unrolling, in: R. Jhala, D. Schmidt (Eds.), Verification, Model Checking, and Abstract Interpretation, in: LNCS, vol. 6538, Springer, 2011, pp. 70–87.
[26] N. Een, A. Mishchenko, R. Brayton, Efficient implementation of property directed reachability, in: FMCAD, 2011, pp. 125–134.
[27] J. Li, S. Zhu, G. Pu, M. Vardi, SAT-based explicit LTL reasoning, in: HVC, Springer, 2015, pp. 209–224.
[28] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 1579, Springer, 1999.
[29] V. Fionda, G. Greco, The complexity of LTL on finite traces: hard and easy fragments, in: AAAI, AAAI Press, 2016, pp. 971–977.
[30] A. Cimatti, M. Roveri, V. Schuppan, S. Tonetta, Boolean abstraction for temporal logic satisfiability, in: Proc. 15th Int'l Conf. on Computer Aided Verification, in: Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 532–546.
[31] J. Li, K.Y. Rozier, G. Pu, Y. Zhang, M.Y. Vardi, SAT-based explicit LTLf satisfiability checking, in: AAAI Conference on Artificial Intelligence, AAAI, 2019.
[32] R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: P. Dembiski, M. Sredniawa (Eds.), Protocol Specification, Testing, and Verification, Chapman & Hall, 1995, pp. 3–18.
[33] N. Eén, N. Sörensson, An extensible SAT-solver, in: SAT, 2003, pp. 502–518.
[34] V. Schuppan, L. Darmawan, Evaluating LTL satisfiability solvers, in: Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis, AVTA'11, Springer-Verlag, 2011, pp. 397–413.
[35] R. Dureja, K.Y. Rozier, More scalable LTL model checking via discovering design-space dependencies (D^3), in: D. Beyer, M. Huisman (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2018, pp. 309–327.
[36] V. Fionda, G. Greco, Ltl on finite and process traces: complexity results and a practical reasoner, J. Artif. Intell. Res. 63 (2018) 557–623.
[37] J. Geldenhuys, H. Hansen, Larger automata and less work for LTL model checking, in: SPIN, in: LNCS, vol. 3925, Springer, 2006, pp. 53–70.
[38] J. Prescher, C. Di Ciccio, J. Mendling, From declarative processes to imperative models, in: SIMPDA, vol. 1293, 2014, pp. 162–173.
[39] J. Li, L. Zhang, G. Pu, M. Vardi, J. He, LTL satisfiability checking revisited, in: TIME, 2013, pp. 91–98.
[40] C. Di Ciccio, F. Maggi, J. Mendling, Efficient discovery of target-branched declare constraints, Inf. Syst. 56 (C) (2016) 258–283, https://doi.org/10.1016/j.is.2015.06.009.
[41] C.D. Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, ACM Trans. Manag. Inf. Syst. 5 (4) (2015) 24:1–24:37, https://doi.org/10.1145/2629447.
[42] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, S. Tonetta, Formal design and safety analysis of AIR6110 wheel brake system, in: D. Kroening, C.S. Păsăreanu (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2015, pp. 518–535.
[43] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, K.Y. Rozier, Model checking at scale: automated air traffic control design space exploration, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2016, pp. 3–22.
[44] J. Li, R. Dureja, G. Pu, K.Y. Rozier, M.Y. Vardi Simplecar, An efficient bug-finding tool based on approximate reachability, in: H. Chockler, G. Weissenbacher (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2018, pp. 37–44.
[45] J. Li, S. Zhu, Y. Zhang, G. Pu, M.Y. Vardi, Safety model checking with complementary approximations, in: ICCAD, 2017.