SAT-Based Explicit LTL_f Satisfiability Checking*

Jianwen Li, Kristin Y. Rozier
Iowa State University
Ames, IA, USA
{jianwen,kyrozier}@iastate.edu

Geguang Pu, Yueling Zhang
East China Normal University
Shanghai, China
{ggpu,ylzhang}@sei.ecnu.edu.cn

Moshe Y. Vardi
Rice University
Houston, TX, USA
vardi@cs.rice.edu

Abstract

We present a SAT-based framework for LTL_f (Linear Temporal Logic on Finite Traces) satisfiability checking. We use propositional SAT-solving techniques to construct a transition system for the input LTL_f formula; satisfiability checking is then reduced to a path-search problem over this transition system. Furthermore, we introduce CDLSC (Conflict-Driven LTL_f Satisfiability Checking), a novel algorithm that leverages information produced by propositional SAT solvers from both satisfiability and unsatisfiability results. Experimental evaluations show that CDLSC outperforms all other existing approaches for LTL_f satisfiability checking, by demonstrating an approximate four-fold speed-up compared to the second-best solver.

Introduction

Linear Temporal Logic over Finite Traces, or LTL_f, is a formal language gaining popularity in the AI community for formalizing and validating system behaviors. While standard Linear Temporal Logic (LTL) is interpreted on infinite traces (Pnueli 1977), LTL_f is interpreted over finite traces (De Giacomo and Vardi 2013). While LTL is typically used in formal-verification settings, where we are interested in nonterminating computations, cf. (Vardi 2007), LTL_f is more attractive in AI scenarios focusing on finite behaviors, such as planning (Bacchus and Kabanza 1998; De Giacomo and Vardi 1999; Calvanese, De Giacomo, and Vardi 2002; Patrizi et al. 2011; Camacho et al. 2017), plan constraints (Bacchus and Kabanza 2000; Gabaldon 2004), and user preferences (Bienvenu, Fritz, and McIlraith 2006; 2011; Sohrabi, Baier, and McIlraith 2011). Due to the wide spectrum of applications of LTL_f in the AI community (De Giacomo, Masellis, and Montali 2014), 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 (De Giacomo and Vardi 2013). The advantage of this approach is that the LTL satisfiability problem has been studied for at least a decade, and many mature tools are available, cf. (Rozier and Vardi 2007; 2010). 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, (Li et al. 2014) 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 (Li et al. 2014), seems to be out of date by now because of the recent dramatic improvement in propositional SAT solving, cf. (Malik and Zhang 2009). On one hand, SAT-based techniques have led to a significant improvement on LTL satisfiability checking, outperforming the tableau-based techniques of *Aalta-finite* (Li et al. 2014). (Also, the SAT-based tool *ltl2sat* for LTL_f satisfiability checking outperforms *Aalta-finite* on particular benchmarks (Fionda and Greco 2016).) On the other hand, SAT-based techniques are now dominant in symbolic model checking (Cavada et al. 2014; Vizel, Weissenbacher, and Malik 2015). Our preliminary evaluation indicates that LTL_f satisfiability checking via SAT-based model checking (Bradley 2011; Een, Mishchenko, and Brayton 2011) or via SAT-based LTL satisfiability checking (Li et al. 2015) both outperform the tableau-based tool *Aalta-finite*. Thus, the question raised initially in (Rozier and Vardi 2007) 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 (Li et al. 2015), 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. 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

*A full report is at http://temporallogic.org/research/AAAI19/. Geguang Pu and Kristin Y. Rozier are corresponding authors.

specializing the transition-system approach of (Li et al. 2015) 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 (Li et al. 2015).

We conduct a comprehensive comparison among different approaches. Our experimental results show that the performance of CDLSC dominates all other existing LTL_f-satisfiability-checking algorithms. On average, CDLSC achieves an approximate four-fold speed-up, compared to the second-best solution (IC3 (Bradley 2011)+K-LIVE (Claessen and Sörensson 2012)) tested in our experiments. Our results re-affirm the conclusion of (Li et al. 2014) that the best approach to LTL_f satisfiability solving is via a dedicated tool, based on explicit-state techniques.

LTL over Finite Traces

Given a set $\mathcal{P}$ of atomic propositions, 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 (De Giacomo and Vardi 2013). 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 $\xi_1 \models \psi$;
- $\xi \models (\phi_1\mathcal{U}\phi_2)$, 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 *Tail Normal Form* (TNF) if ϕ is in *Negated Normal Form* (NNF) and $\mathcal{N}$-free. It is trivial to know that every LTL_f formula has an equivalent NNF. Assume ϕ is in NNF, $\mathsf{tnf}(\phi)$ 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 (De Giacomo and Vardi 2013)), 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 $\mathsf{tnf}(\phi)$ is satisfiable.*

In the rest of the paper, unless clearly specified, the input LTL_f formula is in TNF.

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 (De Giacomo and Vardi 2013). Instead of constructing the NFA for ϕ, we generate the corresponding *transition system* (Definition 5), 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.

The classic approach to generate the NFA from an LTL_f formula, i.e., Tableau Construction (Gerth et al. 1995), 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 $\phi = (\neg Tail \wedge a)\mathcal{U}b$. 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 4. If we replace $\mathcal{X}\phi$ in $\mathsf{xnf}(\phi)$ with a 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 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 6) 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).

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 step, 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. 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) (Biere et al. 1999), 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 (Bradley 2011).

SAT-based Explicit-State Checking

Given an LTL_f formula ϕ, we construct the LTL_f *transition system* (Li et al. 2014; 2015) leveraging SAT solvers and then check the satisfiability of the formula over its corresponding transition system.

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 2 (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 atomic propositions. Thus, an LTL_f formula ϕ can be viewed as a propositional formula over $\mathsf{PA}(\phi)$.

Definition 3. *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 3, 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 $\{\neg a, p_1((\neg Tail \wedge a)\mathcal{U}b), \neg b, p_2((Tail \vee c)\mathcal{R}d)\}$ is a propositional assignment of ϕ^p. In the rest of the paper, we do not introduce the intermediate variables and directly say $\{\neg a, (\neg Tail \wedge a)\mathcal{U}b, \neg b, (Tail \vee c)\mathcal{R}d\}$ 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 $\xi \models \bigwedge A$; On the other hand, $\xi \models \bigwedge A$ where A is a propositional assignment of ϕ^p, also implies $\xi \models \phi$.*

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

Definition 4 (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.*

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 $L(A) = \{l | l \in A \textit{ is a literal}\}$ and $X(A) = \{\theta | \mathcal{X}\theta \in A\}$. Now we define the *transition system* for an LTL_f formula.

Definition 5. *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* $S \subseteq 2^{cl(\phi)}$ *is the set of states,* $s_0 = \{\phi\} \in S$ *is the* initial state*, and*

- $T : S \times \Sigma \to 2^S$ *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*.*

Definition 6 (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.

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

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 6, 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.

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 (Li et al. 2015), where information of both satisfiability and unsatisfiability results of SAT solvers are used. The motivation is as follows: In Definition 6, 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 7 (Conflict Sequence)**.** *Given an* LTL_f *formula* ϕ*, a conflict sequence* $\mathcal{C}$ *for the transition system* T_ϕ *is a finite sequence of set 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 $\mathcal{C}[i]$ *is 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 Figure 1, in which s_0 is the initial state and s_4 is the final state. Based on Definition 7, 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.

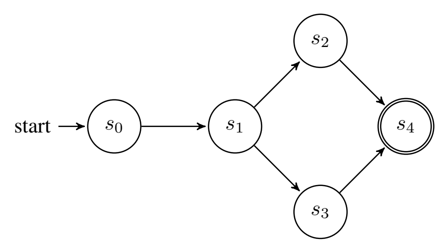

Figure 1: An example transition system for the conflict sequence.

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

Lemma 2. *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 step. Basically from Definition 7, $\mathcal{C}[0]$ 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 7, 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. $\square$

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 the proof of Lemma 2). As a result, it is true that s_i is not in $\mathcal{C}[n-i]$ if $|\mathcal{C}| > n-i$. $\square$

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 2). If all one-transition next states of s are in $\mathcal{C}[i-1]$, s cannot reach a final state in i steps.

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 2 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 $\bigcap_{0 \leq j \leq i} \mathcal{C}[j]$. 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 the set of states of T_ϕ. According to Lemma 2, $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] (i \geq 0)$ contains the set of states that are not final in up to i steps. Now we let $\mathcal{C}$ satisfy that $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] (i \geq 0)$ contains all states that are not final in up to i steps, so $\bigcap_{0 \leq j \leq i} \mathcal{C}[j]$ includes all reachable states from ϕ, as ϕ is unsatisfiable. However, because $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] \supseteq \bigcap_{0 \leq j \leq i+1} \mathcal{C}[j] \supseteq S (i \geq 0)$, there must be an $i \geq 0$ such that $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] = \bigcap_{0 \leq j \leq i+1} \mathcal{C}[j]$, which indicates that $\bigcap_{0 \leq j \leq i} \mathcal{C}[j] \subseteq \mathcal{C}[i+1]$ is true. $\square$

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 6, 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 set 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 7. 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 7. 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 (Eén and Sörensson 2003)) 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 1-2 consider the case when the input formula ϕ is a final state itself. Otherwise, the first frame $\mathcal{C}[0]$ is initialized to $\{\phi\}$ (Line 3), and the current frame level is set to 0 (Line 4). After that, the loop body (Line 5-11) 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 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, $frame\_level$, $try\_satisfy$ 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

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: Set $\mathcal{C}[0] := \{\phi\}$;
4: Set $frame\_level := 0$;
5: **while** true **do**
6:     **if** $try\_satisfy(\phi, frame\_level)$ returns true **then**
7:         **return** SAT;
8:     **if** $inv\_found(frame\_level)$ returns true **then**
9:         **return** UNSAT;
10:     $frame\_level := frame\_level + 1$;
11:     Set $\mathcal{C}[frame\_level] = \emptyset$;

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 12-13). 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 14-15).

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 $\mathcal{C}[frame\_level]$;
11:             Continue;
12:     **if** $try\_satisfy(\phi', frame\_level - 1)$ is true **then**
13:         **return** true;
14: Let $c = get\_uc()$;
15: Add c into $\mathcal{C}[frame\_level + 1]$;
16: **return** false;

Notably, Item 1 of Definition 7, 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 6 in Algorithm 1) and there is some c (Line 15 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

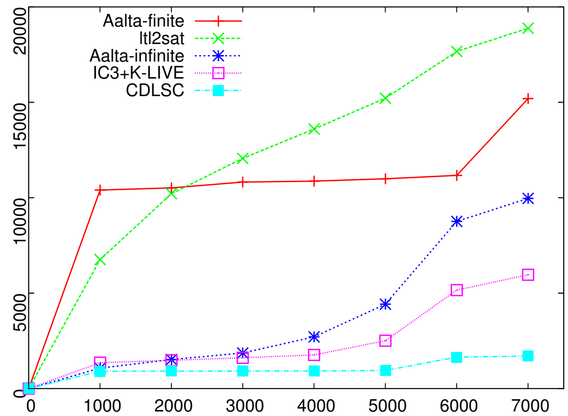

Figure 2: Result for LTL_f Satisfiability Checking on LTL-as-LTL_f Benchmarks. The X axis represents the number of benchmarks, and the Y axis is the accumulated checking time (s).

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 $\bigwedge_{1 \leq j \leq i} \mathcal{C}[j] \Rightarrow \mathcal{C}[i+1]$. Theorem 7 provides the theoretical guarantee that CDLSC always terminates correctly.

Lemma 3. *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.*

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

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.

Experimental Evaluation

Benchmarks[1] Our extensive experimental evaluation, checking 9142 formulas, uses two classes of benchmarks: 7442 LTL-as-LTL_f (since LTL formulas share the same syntax as LTL_f) and 1700 LTL_f-Specific benchmarks, which are common LTL_f patterns that are all satisfiable by finite traces (but not necessarily by infinite traces). We check both execution time and correctness; checking also correctness, as in (Rozier and Vardi 2007), ensures we are comparing performance of tools finding the *same* results.

LTL-as-LTL_f benchmarks consist of the following. **Random Formulas** generated as in (Rozier and Vardi 2011), vary the number of variables $\{1, 2, 3\}$, formula length $\{5, \ldots, 100\}$, and probability of choosing a temporal operator $\{0.3, 0.5, 0.7, 0.95\}$ from the operator set $\{\neg, \vee, \wedge, \mathcal{X}, \mathcal{U}, \mathcal{R}, \mathcal{G}, \mathcal{F}, \mathcal{GF}\}$. We generate all formulas prior to testing for repeatability. **Counter Formulas** scale four, temporally complex patterns that describe large state

[1] All artifacts for enabling reproducibility, including benchmark formulas and their generators, are available from the paper website at http://temporallogic.org/research/AAAI19.

Type	Number	Result	IC3+K-LIVE	Aalta-finite	Aalta-infinite	ltl2sat	CDLSC
Alternate Response	100	sat	134	1	48	123	3
Alternate Precedence	100	sat	154	3	70	380	4
Chain Precedence	100	sat	127	2	45	83	2
Chain Response	100	sat	79	1	41	49	2
Precedence	100	sat	132	2	14	124	1
Responded Existence	100	sat	130	1	14	327	1
Response	100	sat	155	1	41	53	2
Practical Conjunction	1000	varies	1669	19564	4443	20477	115

Table 1: Results for LTL_f Satisfiability Checking on LTL_f-specific Benchmarks.

spaces: n-bit binary counters for $1 \leq n \leq 20$ (Rozier and Vardi 2007). The four templates differ in variables and nesting of $\mathcal{X}$'s. **Pattern Formulas** encode eight scalable patterns (from (Geldenhuys and Hansen 2006), and are generated by code from (Rozier and Vardi 2007)) scaling to $n = 100$. **Other LTL formulas** that were used as specifications in realistic case studies: (Bloem et al. 2007; De Wulf et al. 2008; Filiot, Jin, and Raskin 2009).

LTL_f-Specific benchmarks consist of the following. **Conjunctive Formulas** combine common LTL_f formulas from (De Giacomo, Masellis, and Montali 2014; Ciccio and Mecella 2015; Prescher, Di Ciccio, and Mendling 2014) as random conjunctions in the style of (Li et al. 2013) in two sets of 500 formulas: (1) 20 variables, varying the number of conjuncts in $\{10, 30, 50, 70, 100\}$; and (2) 50 conjuncts, varying the number of variables in $\{10, 30, 50, 70, 100\}$. **Pattern Formulas** scalable patterns inspired by (Di Ciccio, Maggi, and Mendling 2016) up to length 100; see Table 1.

Experimental Setup We implement CDLSC in the tool *aaltaf*[2] and use Minisat 2.2.0 (Eén and Sörensson 2003) as the SAT engine. We compare it with two extant LTL_f satisfiability solvers: Aalta-finite (Li et al. 2014) and ltl2sat (Fionda and Greco 2016). We also compared with the state-of-art LTL solver Aalta-infinite (Li et al. 2015), using the LTL_f-to-LTL satisfiability-preserving reduction described in (De Giacomo and Vardi 2013). As LTL satisfiability checking is reducible to model checking, as described in (Rozier and Vardi 2007), we also compared with this reduction, using nuXmv with the IC3+K-LIVE back-end (Cavada et al. 2014), as an LTL_f satisfiability checker.

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 48GB of RAM per node. Each tool executed on a dedicated node with a timeout of 60 seconds, measuring execution time with Unix `time`. Excluding timeouts, all solvers found correct verdicts for all formulas. All artifacts are available in the supplemental material.

Results Figure 2 shows the results for LTL_f satisfiability checking on LTL-as-LTL_f benchmarks. CDLSC outperforms all other approaches. On average, CDLSC performs about 4 times faster than the second-best approach IC3+K-LIVE (1705 seconds vs. 6075 seconds). CDLSC checks the LTL_f formula directly, while IC3+K-LIVE must take the input of the LTL formula translated from the LTL_f formula. As a result, IC3-KLIVE may take extra cost, e.g., finding a satisfying lasso for the model, to the satisfiability checking.

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

Meanwhile, CDLSC can benefit from the heuristics dedicated for LTL_f that are proposed in (Li et al. 2014). Finally, the performance of ltl2sat 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.

Table 1 shows the results for LTL_f-specific experiments. Columns 1-3 show the types of LTL_f formulas under test, the number of test instances for each formula type, and the results by formula type. Columns 4-8 show the checking times by formula types in seconds. The dedicated LTL_f solvers perform extremely fast on the seven scalable pattern formulas (Column 5 and 8), because their heuristics work well on these patterns. For the difficult conjunctive benchmarks, CDLSC still outperforms all other solvers.

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 (Li et al. 2015) showed this approach performs worse than IC3+K-LIVE, CDLSC outperforming IC3+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 (Fionda and Greco 2016).

Our experiments demonstrate that CDLSC outperforms Aalta-infinite and IC3+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 (Bradley 2011). 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. We conclude that CDLSC outperforms other existing approaches for LTL_f satisfiability checking.

Acknowledgments. We thank anonymous reviewers for helpful comments. This work is supported by NASA ECF NNX16AR57G, NSF CAREER Award CNS-1552934, NSF grants CCF-1319459 and IIS-1527668, 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.

References

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

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

Bienvenu, M.; Fritz, C.; and McIlraith, S. 2006. Planning with qualitative temporal preferences. In *KR*, 134–144.

Bienvenu, M.; Fritz, C.; and McIlraith, S. A. 2011. Specifying and computing preferred plans. *Artificial Intelligence* 175(7C8):1308 – 1345.

Biere, A.; Cimatti, A.; Clarke, E.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer.

Bloem, R.; Galler, S.; Jobstmann, B.; Piterman, N.; Pnueli, A.; and Weiglhofer, M. 2007. Automatic hardware synthesis from specifications: A case study. In *DATE*, 1188–1193.

Bradley, A. 2011. SAT-based model checking without unrolling. In Jhala, R., and Schmidt, D., eds., *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer. 70–87.

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

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

Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S. 2014. The NuXMV symbolic model checker. In *CAV*, 334–342.

Ciccio, C. D., and Mecella, M. 2015. On the discovery of declarative control flows for artful processes. *ACM Trans. Manage. Inf. Syst.* 5(4):24:1–24:37.

Claessen, K., and Sörensson, N. 2012. A liveness checking algorithm that counts. In *FMCAD*, 52–59. IEEE.

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

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

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

De Wulf, M.; Doyen, L.; Maquet, N.; and Raskin, J.-F. 2008. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *TACAS*, volume 4963 of *LNCS*, 63–77. Springer.

Di Ciccio, C.; Maggi, F.; and Mendling, J. 2016. Efficient discovery of target-branched declare constraints. *Inf. Syst.* 56(C):258–283.

Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In *SAT*, 502–518.

Een, N.; Mishchenko, A.; and Brayton, R. 2011. Efficient implementation of property directed reachability. In *FMCAD*, 125–134.

Filiot, E.; Jin, N.; and Raskin, J.-F. 2009. An antichain algorithm for LTL realizability. In *CAV*, 263–277.

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

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

Geldenhuys, J., and Hansen, H. 2006. Larger automata and less work for LTL model checking. In *SPIN*, volume 3925 of *LNCS*, 53–70. Springer.

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

Li, J.; Zhang, L.; Pu, G.; Vardi, M.; and He, J. 2013. LTL satisfibility checking revisited. In *TIME*, 91–98.

Li, J.; Zhang, L.; Pu, G.; Vardi, M. Y.; and He, J. 2014. LTL_f satisfibility checking. In *ECAI*, 91–98.

Li, J.; Zhu, S.; Pu, G.; and Vardi, M. 2015. SAT-based explicit LTL reasoning. In *HVC*, 209–224. Springer.

Malik, S., and Zhang, L. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52(8):76–82.

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

Pnueli, A. 1977. The temporal logic of programs. In *IEEE FOCS*, 46–57.

Prescher, J.; Di Ciccio, C.; and Mendling, J. 2014. From declarative processes to imperative models. *SIMPDA* 1293:162–173.

Rozier, K., and Vardi, M. 2007. LTL satisfiability checking. In *SPIN*, volume 4595 of *LNCS*, 149–167. Springer.

Rozier, K., and Vardi, M. 2010. LTL satisfiability checking. *STTT* 12(2):123–137.

Rozier, K., and Vardi, M. 2011. A multi-encoding approach for LTL symbolic satisfiability checking. In *Int'l Symp. on Formal Methods*, volume 6664 of *LNCS*, 417–431. Springer.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2011. Preferred explanations: Theory and generation via planning. In *AAAI*, 261–267.

Vardi, M. 2007. Automata-theoretic model checking revisited. In *VMCAI*, LNCS 4349, 137–150. Springer.

Vizel, Y.; Weissenbacher, G.; and Malik, S. 2015. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE* 103(11):2021–2035.