Formal Methods in System Design (2019) 54:164–190
https://doi.org/10.1007/s10703-018-00326-5

SAT-based explicit LTL reasoning and its application to satisfiability checking

Jianwen Li[1] · Shufang Zhu[2] · Geguang Pu[2] · Lijun Zhang[3] · Moshe Y. Vardi[1]

Published online: 2 January 2019

Abstract
We present here a new explicit reasoning framework for linear temporal logic (LTL), which is built on top of propositional satisfiability (SAT) solving. The crux of our approach is a construction of temporal transition system that is based on SAT-solving rather than tableau to construct states and transitions. As a proof-of-concept of this framework, we describe a new LTL satisfiability algorithm. We tested the effectiveness of this approach by demonstrating that it significantly outperforms all existing LTL-satisfiability-checking algorithms.

Keywords Satisfiability checking · Linear temporal logic · SAT-based LTL checking

1 Introduction

Linear Temporal Logic (LTL) was introduced into program verification in [30]. Since then it has been widely accepted as a language for the specification of reactive systems [25] and it is a key component in the verification of reactive systems [6,17]. Explicit temporal reasoning, which involves an explicit construction of temporal transition systems, is one of the key algorithmic components in this context. For example, explicitly translating LTL formulas to Büchi automata is a key step both in explicit-state model checking [14] and in runtime verification [38]. LTL satisfiability checking, a step that should take place *before* verification,

✉ Geguang Pu
ggpu@sei.ecnu.edu.cn

Jianwen Li
lijwen2748@gmail.com

Shufang Zhu
saffiechu@gmail.com

Lijun Zhang
zhanglj@ios.ac.cn

Moshe Y. Vardi
vardi@cs.rice.edu

1 Rice University, Houston, USA

2 East China Normal University, Shanghai, China

3 University of Chinese Academy of Sciences, Beijing, China

to assure consistency of temporal requirements, can also use explicit reasoning [31]. These tasks are known to be quite demanding computationally for complex temporal properties [14,31,38]. A way to get around this difficulty is to replace explicit reasoning by symbolic reasoning, e.g., as in BDD-based or SAT-based model checking [28,29], but there are cases that the symbolic approach is inefficient [31] or inapplicable [38]. Thus, explicit temporal reasoning remains an indispensable algorithmic tool.

The main approach to explicit temporal reasoning is based on the *tableau* technique, in which a recursive *syntactic* decomposition of temporal formulas drives the construction of temporal transition systems. This approach is based on the technique of *propositional tableau*, whose essence is search via *syntactic splitting* [8]. This is in contrast to modern propositional satisfiability (SAT) solvers, whose essence is search via *semantic splitting* [24]. The tableau approach to temporal reasoning underlies both the efficient LTL-to-automata translator [10] and the LTL-satisfiability checker [22]. Thus, we have a situation where in the symbolic setting much progress is being attained both by the impressive improvement in the capabilities of modern SAT solvers [24] as well as new SAT-based model-checking algorithms [2,5], while progress in explicit temporal reasoning is slower and does not fully leverage modern SAT solving. It should be noted that several LTL satisfiability solvers, including Aalta_v1.2 [21], and ls4 [37] do employ SAT solvers. However, Aalta_v1.2 utilizes the SAT solving as an *aid* to the main reasoning engine, rather than serve as the *main* reasoning engine. While ls4 has a similar SAT-based framework, our new SAT-based approach relies on different data structures and proposes distinguished heuristics to accelerate the satisfiability checking. Detailed comparisons between these two SAT-based methodologies are shown in Sect. 7.

Our main aim in this paper is to study how SAT solving can be *fully leveraged* in explicit temporal reasoning. The key intuition is that explicit temporal reasoning consists of construction of states and transitions, subject to temporal constraints. Such temporal constraints can be reduced to a sequence of Boolean constraints, which enables the application of SAT solving. This idea underlies the complexity-theoretic analysis in [40], and has been explored in the context of modal logic [15,19], but not yet in the context of explicit temporal reasoning. Our belief is that SAT solving would prove to be superior to tableau in that context.

We describe in this paper a general framework for SAT-based explicit temporal reasoning. The crux of our approach is a construction of temporal transition system that is based on SAT-solving rather than tableau to construct states and transitions. The obtained transition system can be used for LTL-satisfiability solving, LTL-to-automata translation, and runtime-monitor construction.

As proof of concept for the new framework, we use it to develop a SAT-based algorithm for LTL-satisfiability checking. We also propose several heuristics to speed up the checking by leveraging SAT solvers. We implemented the algorithm and heuristics in an LTL-satisfiability solver Aalta_v2.0. To evaluate its performance, we compared it against Aalta_v1.2, the existing best-of-breed LTL-satisfiability solver [21,22], which is tableau-based. We also compare it against NuXmv, a symbolic LTL-satisfiability solver that is based on cutting-edge SAT-based model-checking algorithms [2,5], which outperforms Aalta_v1.2. We show that our explicit SAT-based LTL-satisfiability solver outperforms both against the benchmarks under test. In summary, the contributions in this paper are as follows:

- We propose a SAT-based explicit LTL-reasoning framework.
- We show a successful application of the framework to LTL-satisfiability checking, by designing a novel algorithm and efficient heuristics.
- We evaluate the effectiveness of this framework by developing a best-of-breed LTL-satisfiability solver.

– We compare our new framework for LTL-satisfiability checking with existing approaches. The experimental results demonstrate that our tool significantly outperforms other existing LTL satisfiability solvers.

In addition to the missing proofs, this article extends [23] in the following ways:

– Present formal representations for LTL-satisfiability algorithms and heuristics, which is not only for readers to understand but also provides the theoretic support and are easier to follow and re-implement;
– Test more benchmarks in the experimental part to show the scalability of our approach.

The paper is organized as follows. Section 2 provides technical background. Section 3 introduces the new SAT-based explicit-reasoning framework. Section 4 describes an overview the application to LTL-satisfiability checking, and Sect. 5 shows the algorithm details. Section 6 shows the experimental results for LTL-satisfiability checking. Finally Sect. 7 concludes the paper.

2 Preliminaries

Linear Temporal Logic (LTL) is an extension of propositional logic, in which temporal connectives X (Next) and U (Until) are introduced. Let AP be a set of atomic properties. The syntax of LTL formulas is defined by:

$$\phi ::= tt \mid ff \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi\ U\ \phi \mid X\phi$$

where $a \in AP$, tt, ff represent the constants *true* and *false*. We introduce the R (Release) connectives as the dual of U, which means $\phi R \psi$ is semantically equivalent to $\neg(\neg\phi\ U\ \neg\psi)$. We also use the usual abbreviations: $Fa = tt\ U\ a$, and $Ga = ff\, R a$.

We say that a is a *literal* if it is an atomic proposition or its negation. Throughout the paper, we fix L to denote the set of literals, lower case letters a, b, c, l to denote literals, α to denote propositional formulas, and ϕ, ψ for LTL formulas. We consider LTL formulas in Negation Normal Form (NNF), which can be achieved by pushing all negations in front of only atoms. Since we consider LTL in NNF, formulas are interpreted here on infinite literal sequences, whose alphabet is $\Sigma := 2^{AP}$.

A *trace* $\xi = \omega_0\omega_1\omega_2 \ldots$ is an infinite sequence in Σ^ω. For ξ and $k \geq 0$ we use $\xi^k = \omega_0\omega_1 \ldots \omega_{k-1}$ to denote a prefix of ξ, and $\xi_k = \omega_k\omega_{k+1} \ldots$ to denote a suffix of ξ. Thus, $\xi = \xi^k\xi_k$. The semantics of LTL with respect to an infinite trace ξ is given by:

– $\xi \models tt$ and $\xi \not\models ff$;
– $\xi \models a$ iff $a \in \xi^1$, where a is an atomic proposition and $\xi^1 = \omega_0 \in \Sigma$;
– $\xi \models \neg\psi$ iff $\xi \not\models \psi$;
– $\xi \models \phi_1 \wedge \phi_2$ iff $\xi \models \phi_1$ and $\xi \models \phi_2$;
– $\xi \models \phi_1 \vee \phi_2$ iff $\xi \models \phi_1$ or $\xi \models \phi_2$;
– $\xi \models X\ \phi$ iff $\xi_1 \models \phi$;
– $\xi \models \phi_1\ U\ \phi_2$ iff there exists $i \geq 0$ such that $\xi_i \models \phi_2$ and for all $0 \leq j < i$, $\xi_j \models \phi_1$;
– $\xi \models \phi_1\ R\ \phi_2$ iff for all $i \geq 0$, it holds $\xi_i \models \phi_2$ or there exists $0 \leq j \leq i$ such that $\xi_j \models \phi_1$.

The *closure* of an LTL formula ϕ, denoted as $cl(\phi)$, is a formula set such that:

1. ϕ is in $cl(\phi)$;
2. ψ is in $cl(\phi)$ if $X\psi \in cl(\phi)$ or $\neg\psi \in cl(\phi)$;

3. ϕ_1, ϕ_2 are in $cl(\phi)$ if $\phi_1\ op\ \phi_2 \in cl(\phi)$, where op can be $\wedge$, $\vee$, U and R;
4. $(X\psi) \in cl(\phi)$ if $\psi \in cl(\phi)$ and ψ is an Until or Release formula.

We say each ψ in $cl(\phi)$, which is added via rules 1–3, is a *subformula* of ϕ. Note that the standard definition of LTL closure consists only of rules 1–3. Rule 4 is added in this paper due to its usage in later sections. Note that the size of $cl(\phi)$ is linear in the length of ϕ, even with the addition of rule 4.

3 Explicit LTL reasoning

In this section we introduce the framework of explicit LTL reasoning. To demonstrate clearly both the similarity and difference between our approach and previous ones, we organize this section as follows. We first provide a general definition of temporal transition systems, which underlies both our new approach and previous approach. We then discuss how traditional methods and our new one relate to this framework.

3.1 Temporal transition system

As argued in [15,39], the key to efficient *modal* reasoning is to reason about states and transitions *propositionally*. We show here how the same approach can be applied to LTL. Unlike *modal logic*, where there is a clear separation between formulas that talk about the current state and formulas that talk about successor states (the latter are formulas in the scope of G or F in LTL), LTL formulas do not allow for such a clean separation. Achieving such a separation requires some additional work.

We first define propositional satisfiability of LTL formulas.

Definition 1 *(Propositional satisfiability)* For an LTL formula ϕ, a *propositional assignment for* ϕ is a set $A \subseteq cl(\phi)$ such that

- every literal $\ell \in L$ is either in A or its negation is, but not both.
- $(\theta_1 \wedge \theta_2) \in A$ implies $\theta_1 \in A$ and $\theta_2 \in A$,
- $(\theta_1 \vee \theta_2) \in A$ implies $\theta_1 \in A$ or $\theta_2 \in A$,
- $(\theta_1\ U\ \theta_2) \in A$ implies $\theta_2 \in A$ or both $\theta_1 \in A$ and $(X(\theta_1\ U\ \theta_2)) \in A$. In the former case, that is, $\theta_2 \in A$, we say that A satisfies $(\theta_1\ U\ \theta_2)$ *immediately*. In the latter case, we say that A *postpones* $(\theta_1\ U\ \theta_2)$.
- $(\theta_1 R \theta_2) \in A$ implies $\theta_2 \in A$ and either $\theta_1 \in A$ or $(X(\theta_1 R \theta_2)) \in A$. In the former case, that is, $\theta_1 \in A$, we say that A satisfies $(\theta_1 R \theta_2)$ *immediately*. In the latter case, we say that A *postpones* $(\theta_1 R \theta_2)$.

We say that a propositional assignment A *propositional satisfies* ϕ, denoted as $A \models_p \phi$, if $\phi \in A$. We say an LTL formula ϕ is *propositionally satisfiable* if there is a propositional assignment A for ϕ such that $A \models_p \phi$.

For example, consider the formula $\phi = (a\ U\ b) \wedge (\neg b)$. The set $A_1 = \{a, (a\ U\ b), (\neg b), (X(a\ U\ b))\} \subseteq cl(\phi)$ is a propositional assignment that propositionally satisfies ϕ. In contrast, the set $A_2 = \{(a\ U\ b), \neg b\} \subseteq cl(\phi)$ is not a propositional assignment.

The following theorem shows the relationship between LTL formula ϕ and its propositional assignment.

Theorem 1 *For an LTL formula ϕ and an infinite trace $\xi \in \Sigma^\omega$, we have that $\xi \models \phi$ iff there exists a propositional assignment $A \subseteq cl(\phi)$ such that A propositionally satisfies ϕ and $\xi \models \bigwedge A$.*

Proof If A propositionally satisfies ϕ and $\xi \models \bigwedge A$, then $\xi \models \phi$, as $\phi \in A$.

For the other direction, assume that $\xi \models \phi$. Let $A = \{\theta \in cl(\phi) : \xi \models \theta\}$. Clearly, $\phi \in A$ is true. It remains to prove that A is a propositional assignment, which we show by structural induction.

- For $\ell \in L$ either $\xi \models \ell$ or $\xi \not\models \ell$, so either $\ell \in A$ or $(\neg\ell) \in A$.
- If $\xi \models (\theta_1 \wedge \theta_2)$, then $\xi \models \theta_1$ and $\xi \models \theta_2$, so both $\theta_1 \in A$ and $\theta_2 \in A$.
- If $\xi \models (\theta_1 \vee \theta_2)$, then $\xi \models \theta_1$ or $\xi \models \theta_2$, so either $\theta_1 \in A$ or $\theta_2 \in A$.
- If $\xi \models (\theta_1\ U\ \theta_2)$, then either $\xi \models \theta_2$, in which case, $\theta_2 \in A$, or $\xi \models \theta_1$ and $\xi \models (X(\theta_1\ U\ \theta_2)$, in which case $\theta_1 \in A$ and $(X(\theta_1\ U\ \theta_2)) \in A$.
- If $\xi \models (\theta_1 R \theta_2)$, then $\xi \models \theta_2$, in which case $\theta_2 \in A$, and either $\xi \models \theta_1$ and $\xi \models (X(\theta_1 R \theta_2))$, in which case $\theta_1 \in A$ and $(X(\theta_1 R \theta_2)) \in A$.

Based on the construction above and Definition 1, we have that A is a propositional assignment of ϕ and $\xi \models \bigwedge A$. □

Since a propositional assignment of LTL formula ϕ contains the information for both current and next states, we are ready to define the *transition systems* of LTL formula.

Definition 2 Given an LTL formula ϕ, the *transition system* T_ϕ is a tuple (S, S_0, T) where

- S is the set of states $s \subseteq cl(\phi)$ that are propositional assignments for ϕ. The *trace* of a state s is $s \cap L$, i.e. $trace(s)$, is the set of literals in s.
- $S_0 \subseteq S$ is a set of *initial states*, where for every state $s \in S$, $\phi \in s$ iff $s \in S_0$.
- $T : S \times S$ is the transition relation, where $T(s_1, s_2)$ holds if $(X\theta) \in s_1$ implies $\theta \in s_2$, for all $X\theta \in cl(\phi)$.

A *run* of T_ϕ is an infinite sequence $s_0, s_1, \ldots$ such that $s_0 \in S_0$ and $T(s_i, s_{i+1})$ holds for all $i \geq 0$.

Every run $r = s_0, s_1, \ldots$ of T_ϕ induces a trace $trace(r) = trace(s_0), trace(s_1), \ldots$ in Σ^ω. In general, it needs not hold that $trace(r) \models \phi$. This requires an additional condition. Consider an Until formula $(\theta_1\ U\ \theta_2) \in s_i$. Since s_i is a propositional assignment for ϕ we either have that s_i satisfies $(\theta_1\ U\ \theta_2)$ immediately or that it postpones it, and then $(\theta_1\ U\ \theta_2) \in s_{i+1}$. If s_j postpones $(\theta_1 U \theta_2)$ for all $j \geq i$, then we say that $(\theta_1\ U\ \theta_2)$ is *stuck* in r.

Theorem 2 *Let r be a run of T_ϕ. If* no *Until subformula is stuck at r, then $trace(r) \models \phi$. Also, ϕ is satisfiable if there is a run r of T_ϕ so that no Until subformula is stuck at r.*

Proof For the first claim, let r be $s_0, s_1, \ldots$ and $r_i = s_i, s_{i+1}, \ldots (i \geq 0)$. Assume that no Until subformua is stuck at r. We prove by induction that $trace(r_i) \models \psi$ for $\psi \in s_i$. It follows that $trace(r) \models \phi$.

- Trivially, for a literal $\ell \in s_i$ we have that $trace(r_i) \models \ell$.
- If $(\theta_1 \wedge \theta_2) \in s_i$, then $\theta_1 \in s_i$ and $\theta_2 \in s_i$. By induction, $trace(r_i) \models \theta_1$ and $trace(r_i) \models \theta_2$, so $trace(r_i) \models (\theta_1 \wedge \theta_2)$. The argument for $(\theta_1 \vee \theta_2) \in s_i$ is analogous.
- If $(X\theta) \in s_i$, then $\theta \in s_{i+1}$. By induction, $trace(r_{i+1}) \models \theta$, so $trace(r_i) \models (X\theta)$
- If $(\theta_1\ U\ \theta_2) \in s_i$, then $\theta_2 \in s_i$ or both $\theta_1 \in s_i$ and $(X(\theta_1\ U\ \theta_2)) \in s_i$, which implies that $(\theta_1\ U\ \theta_2) \in s_{i+1}$. Since $(\theta_1\ U\ \theta_2)$ is not stuck at r, there is some $k \geq i$ such that $\theta_2 \in s_k$, and $\theta_1 \in s_j$ for $i \leq j < k$. Using the induction hypothesis and the semantics of Until, it follows that $trace(r_i) \models (\theta_1\ U\ \theta_2)$.
- If $(\theta_1 R \theta_2) \in s_i$, then $\theta_2 \in s_i$ and either $\theta_1 \in s_i$ or $(X(\theta_1 R \theta_2)) \in s_i$, which implies $(\theta_1 R \theta_2) \in s_{i+1}$. It is possible here for $(\theta_1 R \theta_2)$ to be postponed forever. So for all $k \geq i$, we have that either $\theta_2 \in s_j$ or there exists $i \leq j \leq k$ such that $\theta_1 \in s_j$. Using the induction hypothesis and the semantics of Release, it follows that $trace(r_i) \models (\theta_1 R \theta_2)$.

It follows that if there is a run r of T_ϕ such that no Until subformul is stuck at r then ϕ is satisfiable.

In the other direction, assume that ϕ is satisfiable and there is an infinite trace $\xi \in L^\omega$ such that $\xi \models \phi$. Let $\xi = P_0, P_1, \ldots$, and let $\xi_i = P_i, P_{i+1}, \ldots$ As in the proof of Theorem 1, define $A_i = \{\theta \in cl(\phi) : \xi_i \models \theta\}$. As in the proof of Theorem 1, each A_i is a propositional assignment for ϕ, and, consequently a state of T_ϕ. Furthermore, the semantics of Next implies that we have $T(A_i, A_{i+1})$ for $i \geq 0$, and the semantics of Until ensures that no Until is stuck in the run $A_0, A_1, \ldots$ □

We have now shown that the temporal transition system T_ϕ is intimately related to the satisfiability of ϕ. The definition of T_ϕ is, however, rather nonconstructive. In the next subsection we discuss how to construct T_ϕ.

3.2 System construction

First, we show how one can consider LTL formulas as propositional ones. This requires considering temporal subformulas as *propositional atoms*. We now define the *propositional atoms* of LTL formulas.

Definition 3 *(Propositional atoms)* For an LTL formula ϕ, we define the set of *propositional atoms* of ϕ, i.e. $PA(\phi)$, as follows:

1. $PA(\phi) = \{\phi\}$ if ϕ is an atom, Next, Until or Release formula;
2. $PA(\phi) = PA(\psi)$ if $\phi = (\neg\psi)$;
3. $PA(\phi) = PA(\phi_1) \cup PA(\phi_2)$ if $\phi = (\phi_1 \wedge \phi_2)$ or $\phi = (\phi_1 \vee \phi_2)$.

Consider, for example, the formula $\phi = (a \wedge (a\ U\ b) \wedge \neg(X(a \vee b)))$. Here we have $PA(\phi)$ is $\{a, (a\ U\ b), (X(a \vee b))\}$. Intuitively, the propositional atoms are obtained by treating all temporal subformulas of ϕ as atomic propositions. Thus, an LTL formula ϕ can be viewed as a propositional formula over $PA(\phi)$.

Definition 4 For an LTL formula ϕ, let ϕ^p be ϕ considered as a propositional formula over $PA(\phi)$.

We now introduce the *neXt Normal Form* (XNF) of LTL formulas, which separates the "current" and "next-state" parts of the formula, but costs only linear in the original formula size.

Definition 5 *(neXt Normal form)* An LTL formula ϕ is in *neXt Normal Form* (XNF) if there are no Unitl or Release subformulas of ϕ in $PA(\phi)$.

For example, $\phi = (a\ U\ b)$ is not in XNF, while $(b \vee (a \wedge (X(a\ U\ b))))$ is in XNF. Every LTL formula ϕ can be converted, with linear in the formula size, to an equivalent formula in XNF.

Theorem 3 *For an LTL formula ϕ, there is an equivalent formula $xnf(\phi)$ that is in XNF. Furthermore, the cost of the conversion is linear.*

Proof To construct $xnf(\phi)$, We can apply the expansion rules $(\phi_1 U \phi_2) \equiv (\phi_2 \vee (\phi_1 \wedge X(\phi_1 U \phi_2)))$ and $(\phi_1 R \phi_2) \equiv (\phi_2 \wedge (\phi_1 \vee X(\phi_1 R \phi_2)))$. In detail, we can construct $xnf(\phi)$ inductively:

1. $xnf(\phi) = \phi$ if ϕ is tt, ff, a literal l or a Next formula $X\psi$;
2. $xnf(\phi) = xnf(\phi_1) \wedge xnf(\phi_2)$ if $\phi = (\phi_1 \wedge \phi_2)$;
3. $xnf(\phi) = xnf(\phi_1) \vee xnf(\phi_2)$ if $\phi = (\phi_1 \vee \phi_2)$;
4. $xnf(\phi) = (xnf(\phi_2)) \vee (xnf(\phi_1) \wedge X\phi)$ if $\phi = (\phi_1\ U\ \phi_2)$;
5. $xnf(\phi) = xnf(\phi_2) \wedge (xnf(\phi_1) \vee X\phi)$ if $\phi = (\phi_1 R \phi_2)$.

Since the construction is built on the two expansion rules that preserve the equivalence of formulas, it follows that ϕ is logically equivalent to $xnf(\phi)$. Note that the conversion to $xnf(\phi)$ doubles the size of the converted formula ϕ, but since the conversion puts Until and Release subformulas in the scope of Next, and the conversion stops when it comes to Next subformulas, the cost is at most linear. □

We can now state propositional satisfiability of LTL formulas in terms of satisfiability of propositional formulas. That is, by restricting LTL formulas to XNF, a satisfying assignment of ϕ^p, which can be obtained by using a SAT solver, corresponds precisely to a propositional assignment of formula ϕ.

Theorem 4 *For an LTL formula ϕ in XNF, if there is a satisfying assignment A of ϕ^p, then there is a propositional assignment A' of ϕ that satisfies ϕ such that $A' \cap PA(\phi) \subseteq A$. Conversely, if there is a propositional assignment A' of ϕ that satisfies ϕ, then there is a satisfying assignment A of ϕ^p such that $A' \cap PA(\phi) \subseteq A$.*

Proof ($\Rightarrow$) Let A be a satisfying assignment of ϕ^p. Then let $A' = \{\psi | \psi \in cl(\phi) \wedge A \models \psi^p\}$. It is clear that $A' \cap PA(\phi) \subseteq A$, and now we prove A' is a propositional assignment of ϕ by induction over ϕ. Basically, if ϕ is a literal or Next formula, $\phi \in A$ is true because of $A \in 2^{PA(\phi)}$ and $A \models \phi^p$. Thus, $\phi \in A'$ is true due to the construction of A'. From Definition 1 we know that A' is a propositional assignment of ϕ.

Inductively, since we consider the formula in XNF, we only need to consider the combination of $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$. If $\phi = \phi_1 \wedge \phi_2$ and A is a satisfying assignment of ϕ^p, first we know there exists A_1 and A_2 such that they are the satisfying assignments of ${\phi_1}^p$ and ${\phi_2}^p$ respectively. Moreover, we have that $A = A_1 \cup A_2$, thus causing $A' = {A_1}' \cup {A_2}'$ from the construction of A'. By assumption hypothesis we have ${A_i}'(i = 1, 2)$ is a propositional assignment of ϕ_i, and according to Definition 1 and $A' = {A_1}' \cup {A_2}'$ we know that A' is a propositional assignment of ϕ; If $\phi = \phi_1 \vee \phi_2$ and assume $A_1 \models \phi^p$, we can set $A = A_1$ and thus we know that $A' \supseteq {A_1}'$ from the construction of A'. By assumption hypothesis we have ${A_1}'$ is a propositional assignment of ϕ_1, and according to Definition 1 and $A' \supseteq {A_1}'$ we know that A' is a propositional assignment of ϕ.

($\Leftarrow$) Let A' be a propositional assignment of ϕ. Then let $A = \{\psi \in A' | \psi \text{ is a literal or Next formula}\} \cup \{\psi \in PA(\phi) | \psi \notin A' \wedge (\neg\psi) \notin A'\}$. Clearly we have $A' \cap PA(\phi) \subseteq A$, and now we prove A is a satisfying assignment of ϕ^p by induction over ϕ. Basically, if ϕ is a literal or Next formula, $\phi \in A'$ is true from Definition 1. Thus, $\phi \in A$ and $\neg\phi \notin A$ are true due to the construction of A. So A is a satisfying assignment of ϕ^p.

Inductively, since we consider the formula in XNF, we only need to consider the combination of $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$. If $\phi = \phi_1 \wedge \phi_2$ and A' is a propositional assignment of ϕ, first we know there exists ${A_1}'$ and ${A_2}'$ such that they are the propositional assignments of ϕ_1 and ϕ_2 respectively. Moreover, we have that $A' = {A_1}' \cup {A_2}'$, thus causing $A_1 \cup A_2 \subseteq A$ and A is consistent from the construction of A. By assumption hypothesis we have $A_i (i = 1, 2)$ is a satisfying assignment of ϕ_i, so $A \supseteq A_1 \cup A_2$ is a satisfying assignment of ϕ; If $\phi = \phi_1 \vee \phi_2$ and assume ${A_1}' \models_p \phi$, we have $A' \supseteq {A_1}'$ and thus $A \supseteq A_1$ from the construction of A. By

assumption hypothesis we have A_1 is a satisfying assignment of ϕ_1, and so A is a satisfying assignment of ϕ. □

Theorem 4 shows that by requiring the formula ϕ to be in XNF, we can construct the states of the transition system T_ϕ via computing satisfying assignments of ϕ^p over $PA(\phi)$. Let t be a satisfying assignment of ϕ^p and A_t be the related propositional assignment of ϕ generated from t by Theorem 4, the construction is operated as follows:

1. Let $S_0 = \{A_t \mid t \models \phi^p\}$; and let $S := S_0$,
2. Compute $S_i = \{A_t \mid t \models (xnf(\bigwedge X(s_i)))^p\}$ for each $s_i \in S$, where $X(s_i) = \{\theta \mid (X\theta) \in s_i\}$; and update $S := S \cup S_i$;
3. Stop if S does not change; else go back to step 2.

The construction first generates initial states (step 1), and then all reachable states from initial ones (step 2); it terminates once no new reachable state can be generated (step 3). Theorem 5 shows S is the set of system states and its size is bounded by $2^{|cl(\phi)|}$.

Theorem 5 *In the construction, S is the set of all reachable states from ϕ. Moreover, the size of S is bounded by $2^{|cl(\phi)|}$.*

Proof We first prove that each $S_i (i \geq 0)$ is the set of reachable states from ϕ. The proof is achieved by induction over i. Basically, every $s \in S_0$ is a propositional assignment of ϕ, so $\phi \in s$ and s is an initial state of T_ϕ. Inductively, assume $S_i (i \geq 0)$ is a set of reachable states from ϕ, now we consider the states in S_{i+1}. From the construction, each $s' \in S_{i+1}$ is a propositional assignment of $\bigwedge X(s)$ for some $s \in S_i$. From Definition 2 we know $T(s, s')$ is true, so s' is a reachable state from s. Because $S = \bigcup_{i \geq 0} S_i$, so S is the set of reachable states from ϕ. Moreover, S is updated until no new states can be generated: this means S contains all reachable states. Since every state $s \in 2^{cl(\phi)}$, so the size of S is bounded by $2^{|cl(\phi)|}$. □

Our goal here is to show that we can construct the transition system T_ϕ by means of SAT solving. This requires us to refine Theorem 2. A key issue in how a propositional assignment handles an Until formula is whether it satisfies it immediately or postpones it. We introduce new propositions that indicate which is the case, and we refine the implementation of $xnf()$. Given $\psi = (\psi_1 \, U \, \psi_2)$, we introduce a new proposition $v(\psi)$, and use the following conversion rule: $xnf(\psi) \equiv (v(\psi) \wedge \psi_2) \vee ((\neg v(\psi)) \wedge \psi_1 \wedge (X(\psi)))$. Thus, $v(\psi)$ is required to be true when the Until is satisfied immediately, and false when the Until is postponed. Now we can state the refinement of Theorem 2.

Theorem 6 *For an LTL formula ϕ, ϕ is satisfiable iff there is a* finite *run $r = s_0, s_1, \ldots, s_n$ in T_ϕ such that*

1. *There are $0 \leq m \leq n$ such that $s_m = s_n$;*
2. *Let $Q = \bigcup_{i=m}^{n} s_i$. If $\psi = (\psi_1 U \psi_2) \in Q$, then $v(\psi) \in Q$.*

Proof Suppose first that items 1 and 2 hold. Then the infinite sequence $r' = s_0, \ldots, s_m, (s_{m+1}, \ldots, s_n)^\omega$ is an infinite run of T_ϕ. It follows from Item 2 that no Until subformula is stuck at r'. By Theorem 2, we have that $r' \models \phi$.

Suppose now that ϕ is satisfiable. By Theorem 2, there is an infinite run r' of T_ϕ in which no Until subformula is stuck. Let $r' = s_0, s_1, \ldots$ be such a run. Each $s_i (i \geq 0)$ is a state of T_ϕ, and the number of states is bounded by $2^{|cl(\phi)|}$. Thus, there must be $0 \leq m < n$ such that $s_m = s_n$ and $r' = s_0, \ldots, s_m, (s_{m+1}, \ldots, s_n)^\omega$. Let $Q = \bigcup_{i=m}^{n} s_i$. Since no Until subformula can be stuck at r, if $\psi = \psi_1 U \psi_2 \in Q$, then it is must be that $v(\psi) \in Q$. □

The significance of Theorem 6 is that it reduces LTL satisfiability checking to searching for a "lasso" in T_ϕ [7]. Item 1 says that we need to search for a prefix followed by a cycle, while Item 2 provides a way to test that no Until subformla gets stuck in the infinite run in which the cycle $s_{m+1}, \ldots, s_n$ is repeated infinitely often.

3.3 SAT-based explicit reasoning versus traditional tableau-based reasoning

We introduced our SAT-based reasoning approach above, and in this section we discuss the difference between our SAT-based approach and earlier works.

Earlier approach to transition-system construction for LTL formulas, based on tableau [14] and normal form [22], generates the system states explicitly or implicitly via a translation to *disjunctive normal form* (DNF). In [22], the conversion to DNF is explicit (though various heuristics are used to temper the exponential blow-up) and the states generated correspond to the disjuncts. In tableau-based tools, cf., [9,14], the construction is based on iterative *syntactic splitting* in which a state of the form $A \cup \{\theta_1 \vee \theta_2\}$ is split to states: $A \cup \{\theta_1\}$ and $A \cup \{\theta_2\}$.

The approach proposed here is based on SAT solving, where the states correspond to satisfying assignments. Satisfying assignments are generated via a search process that is guided by *semantic splitting*. The advantage of using SAT solving rather than syntactic approaches is the impressive progress in the development of heuristics that have evolved to yield highly efficient SAT solving: unit propagation, two-literal watching, back jumping, clause learning, and more, see [24]. Furthermore, SAT solving continues to evolve in an impressive pace, driven by an annual competition.[1] It should be remarked that an analogous debate, between syntactic and semantic approaches, took place in the context of automated test-pattern generation for circuit designs, where, ultimately, the semantic approach has been shown to be superior [20].

Furthermore, relying on SAT solving as the underlying reasoning technology enables us to decouple temporal reasoning from propositional reasoning. Temporal reasoning is accomplished via a search in the transition system, which requires proposition reasoning using SAT solving.

4 LTL-satisfiability-checking algorithm overview

Given an LTL formula ϕ, the satisfiability problem is to ask whether there is an infinite trace ξ such that $\xi \models \phi$. In the previous section, we introduced a SAT-based LTL-reasoning framework and showed how it can be applied to solve LTL reasoning problems. We now use this framework to develop an efficient SAT-based algorithm for LTL satisfiability checking. In this section, we show an overview of designing a depth-first-search (DFS) algorithm, that constructs the temporal transition system on-the-fly and searches for a trace per Theorem 6. Furthermore, we propose several heuristics to reduce the search space. The details are introduced explicitly in next section.

4.1 The main algorithm

The main algorithm, LTL-CHECK, creates the temporal transition system of the input formula on-the-fly, and searches for a lasso in a DFS manner. Several prior works describe algorithms

[1] See http://www.satcompetition.org/.

for DFS lasso search, cf. [7,22,34]. Here we focus on the steps that are specialized to our algorithm.

The key idea of LTL-CHECK is to create states and their successors using SAT techniques rather than traditional tableau or expansion techniques. Given the current formula ϕ, we first compute its XNF version $xnf(\phi)$, and then use a SAT solver to compute the satisfying assignments of $(xnf(\phi))^p$. Let P be a satisfying assignment for $(xnf(\phi))^p$; from the previous section we know that $X(P) = \{\theta \mid X\theta \in P\}$ yields a successor state in T_ϕ. We implement this approach in the $getState$ function, which we improve later by introducing some heuristics. By enumerating all assignments of $(xnf(\phi))^p$ we can obtain all successor states of P. Note, however that LTL-CHECK runs in the DFS manner, under which only a single state is needed at a time, so additional effort must be taken to maintain history information of the next-state generation for each state P.

As soon as LTL-CHECK detects a lasso, it checks whether the lasso is accepting. Previous lasso-search algorithms operate on the Büchi automaton generated from the input formula. In contrast, here we focus directly on the satisfaction of Until subformulas per Theorem 6. We use the example below to show the general idea.

Consider the formula $\phi = G((Fb) \wedge (Fc))$. By Theorem 3, $xnf(\phi) = xnf(Fb) \wedge xnf(Fc) \wedge X\phi$, where $xnf(Fb) = ((b \wedge v(Fb)) \vee (\neg v(Fb) \wedge X(Fb)))$ and $xnf(Fc) = ((c \wedge v(Fc)) \vee (\neg v(Fc) \wedge X(Fc)))$. Suppose we get from the SAT solver an assignment of $(xnf(\phi))^p$ $P = \{v(Fb), \neg v(Fc), b, \neg c, \neg X(Fb), X(Fc), X\phi\}$. By Theorem 4, we create a satisfying assignment A' that includes all formulas in $cl(\phi)$ that are satisfied by P, and we get the state $s_0 = P \cup \{\phi, Fb, Fc, (Fb) \wedge (Fc)\}$. To obtain the next state, we start with $X(s_0) = \{Fc, \phi\}$, compute $xnf(Fc \wedge \phi)$ and repeat the process. After several steps LTL-CHECK may find a path $s_0 \rightarrow s_1 \rightarrow s_0$, where $s_1 = \{\phi, Fb, Fc, (Fb) \wedge (Fc), \neg v(Fb), v(Fc), \neg b, c, X(Fb), \neg X(Fc), X\phi\}$. Now s_0 and s_1 form a lasso. Let $Q = s_0 \cup s_1$. Both Fb and Fc are in Q, and also $v(Fb)$ and $v(Fc)$ are in Q. By Theorem 6, ϕ is satisfiable.

4.2 Heuristics for state elimination

While LTL-CHECK uses an efficient SAT solver to compute states of the system in the $getState$ function, this approach is effective in creating states and their successors, but cannot be used to guide the overall search. To find a satisfying lasso faster, we add heuristics that drive the search towards satisfaction. The key to these heuristics is smartly choosing the next state given by SAT solvers. This can be achieved by adding more constraints to the SAT solver. In summary, the **ELIMINATION** and **SAT_PURSUING** techniques involves a faster check on satisfiable formulas, while the **CONFLICT_ANALYZE** technique speeds up the checking on unsatisfiability formulas. Experiments show these heuristics are critical to the performance of our LTL-satisfiability tool.

The construction of state in the transition system always starts with formulas. At the beginning, we have the input formula ϕ_0 and we take the following steps: (1) Compute $xnf(\phi_0)$; (2) Call a SAT solver to get an assignment P_0 of $(xnf(\phi_0))^p$; and (3) Derive a state P_0' from P_0. Then, to get a successor state, we start with the formula $\phi_1 = \bigwedge X(P_0')$, and repeat steps (1-3). Thus, every state s is obtained from some formula ϕ_s, which we call the *representative formula*. Note that with the possible exception of ϕ_0, all representative formulas are conjunctions. Let $\phi_s = \bigwedge_{1 \le i \le n} \theta_i$ be the representative formula of a state s; we say that $\theta_i (1 \le i \le n)$ is an *obligation* of ϕ if θ_i is an Until formula. Thus, we associate with the state s a set of obligations, which are the Until conjunctive elements of ϕ_s. (The initial state may have obligations if it is a conjunction.) The approach we now describe is to

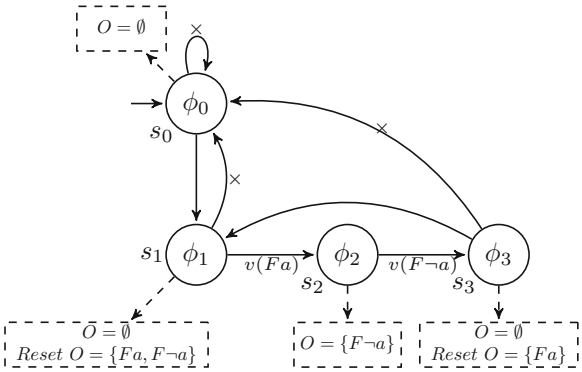

Fig. 1 A satisfiable formula. In the figure $\phi_0 = G((Fa) \wedge (F\neg a))$, $\phi_1 = ((Fa) \wedge (F\neg a) \wedge \phi_0)$, $\phi_2 = ((F\neg a) \wedge \phi_0)$ and $\phi_3 = ((Fa) \wedge \phi_0)$. These representative formulas correspond to states s_0, s_1, s_2, s_3, respectively

satisfy obligations as early as possible during the search, so that a satisfying lasso is obtained earlier. We now refine the *getState* function, and introduce three heuristics via examples.

The *getState* function keeps a global *obligation set*, collecting all obligations so far not satisfied in the search. The obligation set is initialized with the obligations of the initial formula ϕ_0. When an obligation o is satisfied (i.e., when $v(o)$ is true), o is removed from the obligation set. Once the obligation set becomes empty in the search, it is reset to contain obligations of current representative formula ϕ_i. In Fig. 1, we denote the obligation set by O. O is initialized to $\emptyset$, as there is no obligation in ϕ_0. O is then reset in the states s_1 and s_3, when it becomes empty.

The *getState* function runs in the **ELIMINATION** mode by default, in which it obtains the next state guided by the obligations of current state. For satisfiable formulas, this leads to faster lasso detection. Consider formula $\phi = G((Fa) \wedge (F\neg a))$. Parts of the temporal transition system T_ϕ are shown in Fig. 1. In the figure, O is reset to $\{(Fa), (F\neg a)\}$ in state s_1, as these are the obligations of ϕ_1. To drive the search towards early satisfaction of obligations, we obtain a successor of s_1, by applying the SAT solver to the formula $(xnf(\phi_1) \wedge (v(Fa) \vee v(F\neg a)))^p$, to check whether Fa or $F\neg a$ can be satisfied immediately. If the returned assignment satisfies $v(Fa)$, then we get the success state s_2 with the representative formulas ϕ_2, and (Fa) is removed from O. Then the next state is s_3 with the representative formula ϕ_3, which removes the obligation $(F\neg a)$. since O becomes empty, it is reset to the obligations $\{Fa\}$ of ϕ_3. Note that in Fig. 1, there should be transitions from s_2 to s_1 and from s_3 to s_2, but they are never traversed under the ELIMINATION mode.

The *getState* function runs in the **SAT_PURSUING** mode when the obligation set becomes empty. In this mode, we want to check whether the next state can be a state that have been visited before and after that visit the obligation set has become empty. In this case, the generated lasso is accepting, by Theorem 6. In Fig. 1, the obligation set O becomes empty in state s_3. Previously, it has become empty in s_1. Normally, we find a success state for s_3 by applying the SAT solver to $(xnf(\phi_3))^p$. To find out if either s_0 or s_1 can be a successor of s_3, we apply the SAT solver to the formula $(xnf(\phi_3) \wedge (X(\phi_0) \vee X(\phi_1)))^p$. Since this formula is satisfiable and indicates a transition from s_3 to s_1 ($X\phi_1$ can be assigned true in the assignment), we have found that $trace(s_0), (trace(s_1), trace(s_2), trace(s_3))^\omega$ satisfies ϕ. In the figure, the transitions labeled **x** represent failed attempts to generate the lasso when O becomes empty. Although failed attempts have a computational cost, trying to close cycles aggressively does pay off.

The *getState* function runs in the **CONFLICT_ANALYZE** mode if all formulas in the obligation set are postponed in the ELIMINATION mode. The goal of this mode is to eliminate

"conflicts" that block immediate satisfaction of obligations. To achieve this, we use a *conflict-guided* strategy. Consider, for example, the formula $\phi_0 = a \wedge (Xb) \wedge F((\neg a) \wedge (\neg b))$. Here the formula $\psi = F((\neg a) \wedge (\neg b))$ is an obligation. We check whether ψ can be satisfied immediately, but it fails. The reason for this failure is the conjunct a in ϕ, which conflicts with the obligation ψ. We identify this conflict using a *minimal unsat core* algorithm [27]. To eliminate this conflict, we add the conjunct $\neg Xa$ to ϕ, hoping to be able to satisfy the obligation immediately in the next state. When we apply the SAT solver to $(xnf(\phi) \wedge (\neg Xa))^p$, we obtain a successor state with the representative formula $\phi_1 = (b \wedge \psi)$, again with ψ as an obligation. When we try to satisfy ψ immediately, we fail again, since ψ conflicts with b. To block both conflicts, we add $\neg Xb$ as an additional constraint, and apply the SAT solver to $(xnf(\phi) \wedge (\neg Xa) \wedge (\neg Xb))^p$. This yields a successor state with the representative formula $\phi_2 = \psi$. Now we are able to satisfy ψ immediately, and we are able to satisfy ϕ with the finite path $\phi \rightarrow \phi_1 \rightarrow \phi_2$.

As another example, consider the formula $\phi = (G(Fa) \wedge Gb \wedge F(\neg b))$. Since $F(\neg b)$ is an obligation, we try to satisfy it immediately, but fail. The reason for the failure is that immediate satisfaction of $F(\neg b)$ conflicts with the conjunct Gb. In order to try to block this conflict, we add to ϕ the conjunct $\neg XGb$, and apply the SAT solver to $(xnf(\phi) \wedge \neg XGb)^p$. This also fails. Furthermore, by constructing a minimal unsat core, we discover that $(xnf(Gb) \wedge \neg X(Gb))^p$ is unsatisfiable. This indicates that Gb is an "invariant"; that is, if Gb is true in a state then it is also true in its successor. This means that the obligation $F(\neg b)$ can never be satisfied, since the conflict can never be removed. Thus, we can conclude that ϕ is unsatisfiable without constructing more than one state.

In general, identifying conflicts using minimal unsat cores enables to either find satisfying traces faster, or conclude faster that such traces cannot be found.

5 LTL-satisfiability-checking algorithm implementation

In this section, we introduce explicitly the implementation of LTL-satisfiability-checking algorithms under our SAT-based framework. We first talk about the basic (main) checking algorithm, and then propose the heuristics for satisfiable and unsatisfiable instances respectively.

5.1 The main checking algorithm

The main algorithm checks the satisfiability of the input formula on the fly. It implements a *depth-first search* to identify the lasso described in Theorem 6. Algorithm 1 shows the details of the main algorithm, which is named LTL-CHECK.

In Algorithm 1, the function $xnf(\phi)$ (Line 4) is implemented based on Theorem 3. It returns the next normal form of ϕ. The function $getState$ takes an input LTL formula ϕ (initial state) and outputs a system state (assignment of ϕ) P. We have that $T(CF(\phi), \bigwedge X(P))$, where $CF(\phi)$ represents the set of conjuncts of ϕ, and $\bigwedge X(P)$ is one of next states of ϕ. As mentioned previously, P can be obtained from the assignments of $(xnf(\phi))^p$. Another main task of $getState$ is to return a new state never created before in every invocation. More details are shown in Algorithm 2.

LTL-CHECK maintains three global lists: $visited_S$, $visited_P$ and $explored$, which record visited states, visited assignments and explored states respectively. That is, $visited_S[i+1]$ is a next state of $visited_S[i]$ $(i \geq 0)$, and $visited_P[i+1]$ is an assignment of $\bigwedge X(visited_P(i))$.

Algorithm 1 LTL Main Checking Algorithm: LTL-CHECK

Require: An LTL formula ϕ.
Ensure: SAT or UNSAT.
1: **if** $\phi = tt$ (or $\phi = ff$) **then**
2: **return** SAT (or UNSAT);
3: **end if**
4: Let $\phi = xnf(\phi)$: make ϕ ready for SAT solver;
5: CALL $getState(\phi)$: get one system state (assignment of ϕ^p) P from ϕ;
6: **while** $P \neq null$ **do**
7: Let $\psi = \bigwedge X(P)$ be the next state of ϕ;
8: **if** ψ is in $explored$ **then**
9: CALL $getState(\phi)$ again: get another P;
10: Continue;
11: **end if**
12: **if** ψ is in $visited_S$ **then**
13: **if** $model(\psi)$ is true **then**
14: **return** SAT;
15: **end if**
16: **else**
17: Push ψ to $visited_S$, and push P to $visited_P$;
18: **if** LTL-CHECK (ψ) is SAT **then**
19: **return** SAT;
20: **end if**
21: Pop ψ from $visited_S$, and pop P from $visited_P$;
22: **end if**
23: CALL $getState(\phi)$ again: get another P;
24: **end while**
25: Push ϕ to $explored$;
26: **return** UNSAT;

Note that explored states are the states whose all successors are visited but no satisfying model is found. So explored states can be safely blocked in the future state search. The function $model$ (in Line 13) is to check whether the lasso detected (containing ψ) is accepting. The correctness is guaranteed by Theorem 6.

Algorithm 2 Implementation of $getState$

Require: An LTL formula ϕ;
Ensure: A new system state of ϕ (assignment of ϕ^p);
1: Let $history$ be a state list;
2: Let α be
$\phi^p \wedge (\bigwedge_{\psi \in explored} \neg(X\psi)^p)^1 \wedge (\bigwedge_{\psi \in history} \neg\psi^p)^2$;
3: **if** α is satisfiable **then**
4: Let P be an assignment of α;
5: $history = history \cup \{\bigwedge X(P)\}$;
6: **return** P;
7: **else**
8: **return** $null$
9: **end if**

At the very beginning, LTL-CHECK checks whether the formula is tt or ff (Line 1–3), in which cases the satisfiability is determined immediately. Then the next normal form of input formula ϕ is computed (Line 4), obtaining a state P from $(xnf(\phi))^p$ (Line 5). If P does not exist (Line 6), i.e. ϕ is checked unsatisfiable after exploring all its next states, then it is pushed to $explored$ (Line 25) and LTL-CHECK returns UNSAT (Line 26). Otherwise,

LTL-CHECK makes sure that the chosen new next state ψ of ϕ is not explored (Line 8–11). Later it checks whether ψ has been visited before (Line 12). If so a lasso has been detected and the *model* function is invoked to check whether a satisfying model is found as well (Line 13–15). If the checking fails, another P is required for further process (Line 23). If ψ is not visited yet, it is pushed into $visited_S$ and LTL-CHECK is invoked recursively by taking ψ as the new input (Line 17,18). If ψ is checked to be satisfiable, so does ϕ; else ψ is popped from $visited_S$ (Line 21) and LTL-CHECK selects another P to continue checking (Line 23). The algorithm terminates as soon as all states from $(xnf(\phi))^p$ are constructed.

Algorithm 3 Implementation of *model*

Require: An LTL formula ϕ;
Ensure: *tt* or *ff*;
1: Let *pos* be the position of ϕ in $visited$;
2: Let $U = \{\psi \text{ is an Until formula} \mid \psi \in CF(visited[pos])\}$;
3: Let $P = \bigcup_{pos \le i \le visited.size-1} visited[i]$;
4: Let $V(U) = \{v(\psi) | \psi \in U\}$, i.e. the set of boolean variables representing for the Until formulas;
5: **if** $V(U) \subseteq P$ **then**
6: **return** *tt*;
7: **else**
8: **return** *ff*;
9: **end if**

The *model* function (Algorithm 3) checks whether all Until formulas in $CF(\phi)$ are satisfied in the cycle by seeking their corresponding boolean variables. Notably, it is sufficient to check the satisfaction of Until formulas in $CF(\phi)$ rather than $cl(\phi)$. For others not appeared in $CF(\phi)$, one can see easily they are satisfied in the cycle found.

The task of $getState$ is to compute a system state, which was never created before, from the input formula. To achieve this, we introduce another data structure $history$ to store all states that are already created so far for each current formula ϕ. Then, the assignment of α in Algorithm 2, is able to be distinguished with those created before. Note in Line 2, the expression labeled 1 erases those states already explored, which are shown can be blocked. And the expression labeled 2 guarantees those assignments that already appeared before cannot be chosen again. By adding these two constraints, SAT solvers can avoid generating duplicated assignments. The notation *null* in Line 7 indicates the desired state does not exist.

However, simply avoiding the generation of duplicated states is not efficient enough to check on a system, whose size of state space is exponential larger than the one of the input formula. In the following section, we present some heuristics to guide SAT solvers to compute the specialized assignment for faster checking.

5.2 Guided state generation

Recall our basic reasoning theorem (Theorem 6), the principle to check whether a lasso is accepting relies on the fulfillment checking of Until formulas in $CF(\psi)$, where ψ is a state in the lasso. So an intuitive idea to speed up the checking process is to locate such a satisfying lasso as soon as possible. Because the satisfying lasso keeps fulfilling the Until formulas, we follow this way and require SAT solvers to always compute assignments that can fulfill some Until formulas in $CF(\psi)$, which denote the set of conjuncts of ψ and ψ is the current state.

We now re-design the $getState$ function in three modes, which focus on different tasks. The ELIMINATION mode tries to fulfill the Until formulas in a global set, the

Algorithm 4 Implementation of the ELIMINATION mode

Require: An LTL formula ϕ and a global set U;
Ensure: A new system state of ϕ (assignment of ϕ^p);
1: **if** U is $\emptyset$ **then**
2: Turn into the SAT_PURSUING mode.
3: Reset U to be $U(\phi)$, where ϕ is the current state and $U(\phi) \subseteq CF(\phi)$ is the set of Until formulas.
4: **end if**
5: Let P be an assignment of
$(xnf(\phi))^p \wedge \bigvee V(U) \wedge \bigwedge_{\psi \in explored} \neg(X(\psi))^p$, where $V(U) = \{v(u) \mid u \in U\}$;
6: **if** P is empty **then**
7: Turn into the CONFLICT_ANALYZE mode.
8: **else**
9: Update $U = U \backslash S$, where $S = \{u \mid v(u) \in P \text{ and } u \in U\}$;
10: **return** P;
11: **end if**

SAT_PURSUING mode is to pursue a satisfying lasso, and the CONFLICT_ANALYZE mode is to pursue an unsatisfiable core if all Until subformulas remained in the global set are postponed. The $getState$ function runs in the ELIMINATION mode by default. The implementation of the ELIMINATION mode is shown in Algorithm 4.

In the ELIMINATION mode, a global get U is used to keep the Until formulas postponed so far. It is initialized as $U(\phi)$, which is the set of Until formulas in $CF(\phi)$ (ϕ is the input formula). The task of the ELIMINATION mode is to check whether some elements in U can be fulfilled (Line 5). If U becomes empty, the SAT_PURSUING mode is invoked to seek a satisfiable lasso (Line 1,2). If the SAT_PURSUING mode does not succeed, the set U is reset to $U(\psi)$, where ψ is the current state. Then the effort to satisfy the elements in U is tried again (Line 5). Now if the try is successful then U is updated and the computed assignment is returned to LTL-CHECK and $getState$ terminates (Line 9,10). Otherwise, it turns into the CONFLICT_ANALYZE mode. Note the two constraints added in Line 5 enables SAT solvers to prune those states which postpone all elements in U and whose next states are already explored.

Algorithm 5 Implementation of the SAT_PURSUING mode

Require: An LTL formula ϕ and a global set U;
Ensure: Checking termination, or turning back to the ELIMINATION mode;
1: Let

$$\psi = \begin{cases} visited_S[0](\phi) & \text{if U = } \emptyset \text{ for the first time;} \\ & \text{otherwise,} \\ \bigvee_{i \leq pos} visited_S[i] & \text{pos is the previous position} \\ & \text{when U becomes } \emptyset. \end{cases}$$

2: Let P be an assignment of
$(xnf(\phi) \wedge X(\psi))^p \wedge \bigwedge_{\lambda \in explored} \neg(X(\lambda))^p$;
3: **if** P is not empty **then**
4: Main algorithm CHECK can terminate with satisfiable (A satisfying loop has been found).
5: **else**
6: Turn back into the ELIMINATION mode.
7: **end if**

To find a satisfying lasso, the SAT_PURSUING mode tries to check whether there is a visited state whose position in $visited_S$ is less than or equal to the position where U becomes empty in previous time. Specially if U becomes empty for the first time then only the initial

state can be considered. Line 1 of Algorithm 5 assigns the disjunction of these states to be a constraint ψ. Then Line 2 shows the inquiry to SAT solvers to compute an assignment whose next state appears before the position where U becomes empty in previous time. If the inquiry succeeds, the SAT_PURSUING mode returns the assignment to LTL-CHECK, and $getState$ terminates (Line 4). Otherwise the SAT_PURSUING mode turns back to the ELIMINATION mode for further process (Line 6).

Consider the visited state ψ whose position in $visited_S$ is before the one where U becomes empty in previous time (If U becomes empty for the first time, ψ must be the input formula ϕ). So the lasso formed by the SAT_PURSUING mode succeeds to satisfy all Until formulas in $CF(\psi)$—as U is reset to be $U(\psi)$ after ψ and all elements in U are satisfied when U becomes empty again. According to Theorem 6 a satisfying model is found.

5.3 Unsat core extraction

It may happen that all elements in U are postponed in the ELIMINATION mode. Then the function turns into the CONFLICT_ANALYZE mode, trying to figure out whether

- The Until subformula ψ is finally satisfied; or
- ψ is postponed forever.

The trivial way to confirm that all reachable states of ϕ postpone ψ is proven not efficient due to its large cost, and we here introduce a more efficient methodology. Let's first consider the reason why ψ is postponed in ϕ. Formally speaking, ψ is postponed in ϕ iff the formula $xnf(\phi) \wedge v(\psi)$ is unsatisfiable. So there must be a minimal unsat core $S_1 \subseteq CF(\phi)$ such that

- $xnf(\bigwedge S_1) \wedge v(\psi)$ is unsatisfiable; and
- for each $S_1' \subset S_1$, $xnf(\bigwedge S_1') \wedge v(\psi)$ is satisfiable.

Note there are already some works on computing such minimal unsat core (see [27]), and we directly apply the techniques here.

Now the task changes to check whether there exists a next state ϕ_1 of ϕ that can avoid the appearance of S_1, i.e. $S_1 \nsubseteq CF(\phi_1)$. We can achieve this by feeding SAT solvers the formula $xnf(\phi) \wedge \neg X(\bigwedge S_1)$. If the formula is satisfiable, the modeling assignment induces the next state that can avoid S_1; Otherwise, there must be a minimal unsat core $S_2 \subseteq CF(\phi)$ to $\bigwedge S_1$, making $xnf(\bigwedge S_2) \wedge \neg X(\bigwedge S_1)$ is unsatisfiable—as S_1 to ψ. Then the task changes to check whether the avoidance of S_2 can be achieved in the next state of ϕ.

Apparently, this is a recursive process and one can see we may maintain a sequence of minimal unsat cores $\rho = S_1, S_2, \ldots$ during computing the next state of ϕ. Then the question raises up that how the recursive computation terminates.

Let $\theta_i = \psi \wedge \bigwedge_{1 \le j \le i} S_j$, i.e. the formula which conjuncts ψ and minimal unsat cores from S_1 to S_i. Obviously, it holds $\theta_{i+1} \Rightarrow \theta_i (i \ge 1)$ and what we expect to find a next state is $\theta_i \not\Rightarrow \theta_{i+1}$. Once $\theta_i \Rightarrow \theta_{i+1}$ holds as well, it indicates θ_i is the unsat core, implying that the Until formula ψ is postponed forever from ϕ. The reason is, $xnf(\theta_i) \wedge \neg X(\theta_i)$ is unsatisfiable under this case, which means θ_i can be never avoided and cause ψ being postponed forever.

Assume a next state ϕ_1 of ϕ is found according to the above strategy and a sequence $\rho = S_1, S_2, \ldots, S_k (k \ge 1)$ is maintained. Now ϕ_1 tries to avoid S_{k-1} in its next state—note S_k is not in $CF(\phi_1)$ but S_{k-1} is. (If $k = 1$ then ϕ_1 tires to avoid ψ in its next state). The corresponding formula is $xnf(\phi_1) \wedge \neg X(S_{k-1})$. This attempt may not succeed, and there may be another minimal unsat core S_k' (not S_k) to S_{k-1}. We must also maintain this information in the sequence. So it turns out the sequence we have to maintain is the sequence of set of minimal unsat cores, i.e. $\rho = Q_1, Q_2, \ldots, Q_k$ where each Q_i is a set of minimal unsat cores.

For example, consider $\phi = (a\ U\ \neg b) \wedge b \wedge Xb \wedge XXb$, we can see easily $\psi = a\ U\ \neg b$ is postponed currently and $Q_1 = \{\{b\}\}$. Moreover $Q_2 = \{\{Xb\}\}$ and $Q_3 = \{\{XXb\}\}$. According to our strategy above, we first try to avoid elements in Q_3, that is to check $xnf(\phi) \wedge \neg X(\bigvee_{S \in Q_3} \bigwedge S)$. Then we get the next state $\phi_1 = (a\ U\ \neg b) \wedge b \wedge Xb$. Similarly we get $\phi_2 = (a\ U\ \neg b) \wedge b$ and $\phi_3 = a\ U\ \neg b$. By then we know $\psi = a\ U\ \neg b$ is not postponed.

For the formula $\phi = Fa \wedge G\neg a$, we know $\psi = Fa$ is postponed currently and $Q_1 = \{\{Fa, G\neg a\}\}$. Since we know that $\theta_1 = \psi \wedge \bigvee_{S \in Q_1} \bigwedge S = Fa \wedge G\neg a$ and $\theta_1 \wedge \neg X\theta_1$ is unsatisfiable—which means ψ will be postponed from ϕ forever. So we can terminate by returning the unsatisfiable result.

For a more complicated example, we consider $\phi = F(\neg a \wedge \neg b) \wedge a \wedge G((a \rightarrow Xb) \wedge (b \rightarrow Xa))$. The Until formula $\psi = F(\neg a \wedge \neg b)$ is postponed currently and $Q_1 = \{\{a\}\}$. To avoid the elements in Q_1 in next state, i.e. $xnf(\phi) \wedge \neg X(\bigvee_{S \in Q_1} \bigwedge S)$, we can get the next state $\phi_1 = F(\neg a \wedge \neg b) \wedge b \wedge G((a \rightarrow Xb) \wedge (b \rightarrow Xa))$. Now ψ is also postponed due to $b \in CF(\phi_1)$, and we update $Q_1 = \{\{a\}, \{b\}\}$. Then we collect existed states ϕ, ϕ_1 together into set Sts, and try to avoid elements in Q_1 in the next state of states—the formula is $(\bigvee_{\phi \in Sts} xnf(\phi)) \wedge \neg X(\bigvee_{S \in Q_1} \bigwedge S)$. But this attempt still fails. So we get $Q_2 = \{\{a, G((a \rightarrow Xb) \wedge (b \rightarrow Xa))\}, \{b, G((a \rightarrow Xb) \wedge (b \rightarrow Xa))\}\}$. As we see $\theta_2 = \psi \wedge (\bigvee_{S \in Q_1} \bigwedge S) \wedge (\bigvee_{S \in Q_2} \bigwedge S)$ is an invariant, i.e. $xnf(\theta_2) \wedge X\neg\theta_2$ is unsatisfiable, so we know θ_2 is the unsat core and ϕ is unsatisfiable.

Now we start to define the sequence we maintain in the CONFLICT_ANALYZE mode, which we call *avoidable sequence*.

Definition 6 *(Avoidable sequence)* For an LTL formula ϕ, let ψ is an Until subformula of ϕ. The avoidable sequence of ψ is a sequence $\rho = Q_0, Q_1, \ldots$, where $Q_i \subseteq 2^{cl(\phi)}$ and

- $Q_0 = \{\{\psi\}\}$;
- For $i \geq 0$, let $\theta_i = \bigwedge_{0 \leq j \leq i} (\bigvee_{S \in Q_j} \bigwedge S)$ then $S' \in Q_{i+1}$ $(S' \subseteq cl(\phi))$ if,
  1. $(\theta_i \wedge \neg X(\theta_i))$ is satisfiable;
  2. $(\bigwedge S') \wedge (\theta_i \wedge \neg X(\theta_i))$ is unsatisfiable;
  3. For each $S'' \subset S'$, $(\bigwedge S'') \wedge (\theta_i \wedge \neg X(\theta_i))$ is satisfiable.

Specially, we say ρ is an unavoidable sequence if there is $k > 0$ such that $\theta_k \Rightarrow \theta_{k+1}$.

The avoidable sequence is an abstract way to represent set of states that postpone ψ. If a state ϕ' in the transition system T_ϕ satisfies $\phi' \Rightarrow \theta_i$, then ϕ' is represented by θ_i. Let $S_{\theta_i}(S_{\theta_{i+1}})$ be the set of states represented by $\theta_i(\theta_{i+1})$, it is easy to see $S_{\theta_{i+1}} \subseteq S_{\theta_i}$ due to $\theta_{i+1} \Rightarrow \theta_i$. For example, assume the avoidable sequence $\rho = \{\{a\}\}, \{\{b\}\}$, then we know $\theta_1 = a$ and $\theta_2 = a \wedge b$. Apparently the state $a \wedge \neg b$ can be represented by θ_1, but not by θ_2. Since the set of states in T_ϕ is finite, so the avoidable sequence must also be finite.

Lemma 1 *For an LTL formula ϕ and ψ is an Until subformula of ϕ, the avoidable sequence of ψ is finite.*

Proof It is directly derived from the fact that elements in the sequence are finite. □

Specially when ρ is an unavoidable sequence, i.e. $\theta_{k-1} \Rightarrow \theta_k (k > 0)$, it means essentially $S_{\theta_i} = S_{\theta_{i+1}}$. We can prove that in this case θ_i covers all states that postpone ψ forever. Before that we introduce the following lemma.

Lemma 2 *For an LTL formula ϕ, let ψ be an Until subformula of ϕ. If $\rho = Q_0, Q_1, \ldots, Q_k$ $(k \geq 1)$ is an unavoidable sequence of ψ, then it holds that $xnf(\theta_k) \wedge \neg X(\theta_k)$ is unsatisfiable.*

Proof Since ρ is an unavoidable sequence, so $S_{\theta_{k-1}} = S_{\theta_k}$. Assume that $xnf(\theta_k) \wedge \neg X(\theta_k)$ is satisfiable, so there is an assignment in which we can extract a next state λ of states in S_{θ_k} and $\lambda \notin S_{\theta_k}$. But since $S_{\theta_{k-1}} = S_{\theta_k}$, it indicates λ is also the next state that is not in $S_{\theta_{k-1}}$ but it is in S_{θ_k}. So $S_{\theta_{k-1}} \neq S_{\theta_k}$ must be true, which is a contradiction. □

Algorithm 6 Implementation of the CONFLICT_ANALYZE mode

Require: An LTL formula ϕ postpone all Until formulas in U;
Ensure: A finite path satisfies at least one element of U or an unsat core;
1: Get some reachable states from ϕ and put them into Sts (including ϕ);
2: Let $\rho = \{\{U\}\}$ and $pos = 0$;
3: Let $\theta_{pos} = \bigwedge_{0 \le i \le pos} \bigvee_{S \in \rho[i]} \bigwedge S$;
4: **while** true **do**
5: **while** $(xnf(\bigvee Sts) \wedge \neg X(\theta_{pos}))^p$ is satisfiable **do**
6: Let P be the assignment and add $X(P)$ to Sts;
7: $pos = pos - 1$ and update θ_{pos} (pos is changed);
8: **if** $pos < 0$ **then**
9: **return** the finite path leading from ϕ to $X(P)$;
10: **end if**
11: **end while**
12: Add computed set of minimal unsat cores to $\rho[pos + 1]$ (if $\rho[pos + 1]$ does not exist, then extend it);
13: $pos = pos + 1$ and update θ_{pos};
14: **while** $xnf(\bigvee Sts) \wedge \neg X(\theta_{pos})$ is unsatisfiable **do**
15: Add computed set of minimal unsat cores to $\rho[pos + 1]$ (if $\rho[pos + 1]$ does not exist, then extend it);
16: Set $pos = pos + 1$ and update θ_{pos};
17: **if** $xnf(\theta_{pos}) \wedge \neg X(\theta_{pos})$ is unsatisfiable **then**
18: **return** θ_{pos} as the unsat core;
19: **end if**
20: **end while**
21: Let P be the assignment of $(xnf(\bigvee Sts) \wedge \neg X(\theta_{pos}))^p$ and add $X(P)$ to Sts;
22: $pos = pos - 1$ and update θ_{pos};
23: **end while**

Let S_ρ be the set of states represented by the avoidable sequence ρ, then we have

Theorem 7 *For an LTL formula ϕ and ψ is an Until subformula of ϕ, if ρ is an unavoidable sequence of ψ, then all states represented by S_ρ are explored.*

Proof From Lemma 2 we know all next states of states in S_ρ are also in S_ρ. And since every state represented by S_ρ can postpone ψ, so all states in S_ρ together can postpone ψ forever. According to the meaning of explored states, all states represented by S_ρ are explored ones. □

Now we present the improved algorithm for CONFLICT_ANALYZE mode. The algorithm maintains the information of avoidable sequence in the mode and utilizes it to locate the result. We should claim that, computing elements of avoidable sequence is relatively expensive so far, and especially for extending the length of the sequence. Consider that if finally the Until formula turns out to be satisfiable, then it may waste time maintaining unnecessarily long sequence. To balance these situations, our algorithm starts from a set of states reachable from the initial state rather than only itself, in which case it can increase the possibility to fulfill the postponed Until formula as soon as possible.

Let $\rho = Q_0, Q_1, \ldots, Q_k$ and we use $\rho[i]$ to represent Q_i in the algorithm. The variable *pos* points to the position of ρ in which the elements should be avoided currently. The notation $X(P)$ means the set of Next formulas in P (they form the next state indeed). In Line 1, users can define their own strategies to obtain the set of reachable states.

6 Experiments on LTL satisfiability checking

In this section we discuss the experimental evaluation for LTL satisfiability checking. We first describe the methodology used in experiments and then show the results.

6.1 Experimental methodologies

The platform used in the experiments is an IBM iDataPlex consisting of 2304 processor cores in 192 Westmere nodes (12 processor cores per node) at 2.83 GHz with 48 GB of RAM per node (4 GB per core), running the 64-bit Redhat 7 operating system. In our experiments, each tool runs on a single core in a single node. We use the Linux command "time" to evaluate the time cost (in seconds) of each experiment. Timeout was set to be 60 s, and the out-of-time cases are set to cost 60 s.

We implemented the satisfiability-checking algorithms introduced in this paper, and named the tool Aalta_v2.0.[2] We compare Aalta_v2.0 with Aalta_v1.2, which is the latest explicit LTL-satisfiability solver (though it does use some SAT solving for acceleration) [21]. (The SAT engine used in both Aalta_v1.2 and Aalta_v2.0 is Minisat [11].) In the literature [21], Aalta_v1.2 is shown to outperform the BDD-based model checking (the representative solver is NuSMV [4]) and the tableau-based (e.g. solvers like pltl [33] approaches,[3]) so we omit the comparison with these solvers in this paper. Two resolution-based LTL satisfiability solvers, TRP++ [18] and ls4 [37], which are not compared comprehensively in our previous work [21], are also included in the comparison in this paper. (Note that ls4 also utilize SAT solving.)

As shown in [31], LTL satisfiability checking can be reduced to model checking. While BDD-based model checker were shown to be competitive for LTL satisfiability solving in [31], they were shown later not to be competitive with specialized tools, such as Aalta_v1.2 [22]. We do, however, include in our comparison the model checker NuXmv [3], which integrates the latest SAT-based model checking techniques. It uses Minisat as the SAT engine as well. Although standard bounded model checking (BMC) is not complete for the LTL satisfiability checking, there are techniques to make it complete, for example, *incremental bounded model checking* (BMC-INC) [16], which is implemented in NuXmv. In addition, NuXmv implements also new SAT-based techniques, IC3 [2], which can handle liveness properties with the K-liveness technique [5]. We included IC3 with K-liveness in our comparison.

To compare with the K-liveness checking algorithm, we ran NuXmv using the command "check_ltlspec_klive -d". For the BMC-INC comparison, we run NuXmv with the command "check_ltlspec_sbmc_inc -c". Aalta_v2.0, Aalta_v1.2 and ls4 tools were run using their default parameters. For the other tool, TRP++ runs with "-sBFS -FSR". Since the input of TRP++ and ls4 must be in SNF (Separated Normal Form [12]), an SNF generator is required for running these tools. We use the *TST-translate* tool, which belongs to the ls4 tool suit, to generate the SNF format models from LTL formulas.

[2] It can be downloaded at www.lab301.cn/aalta.

[3] Although a most recent tableau-based solver has been presented in [1], our preliminary results show its performance is at least 10 times worse than Aalta_2.0. As a result, we rule out the comparison with this work.

In the experiments, we consider two kinds of benchmarks. One is from [32], which is referred to as *schuppan-collected*. This benchmark collects formulas from several prior works, including [31], and has a total of 7550 formulas. We note that this benchmark consists of 3775 pattern formulas and their negations, which can be easily checked satisfiable by most of tested solvers. As a result, we only take into account the 3775 pattern formulas in our experiments. Another one is called *random conjunction* formulas, which is proposed in [22] to test the scalability of LTL satisfibility solvers.

A random conjunction formula $RC(n)$ has the form of $\bigwedge_{1\leq i\leq n} P_i(v_1, v_2, \ldots, v_k)$, where n is the number of conjunctive elements and $P_i(1 \leq i \leq n)$ is a randomly chosen pattern formula used frequently in practice.[4] The motivation is that typical temporal assertions may be quite small in practice. And what makes the LTL satisfiability problem often hard is that we need to check *large collections of small temporal formulas*, so we need to check that the conjunction of all input assertions is satisfiable. In our experiment, the number of n varies from 1 to 30, and for each n a set of 100 conjunctions formulas are randomly chosen. In our experiments, we did not find any inconsistency among the solvers that did not time out.

6.2 Results

The overall experimental results on the *schuppan-collected* benchmark are shown in Table 1. In the table, the first column lists the different benchmarks in the suite, the second column shows the number of instances in the benchmarks, and the third to ninth columns display the results from different solvers. Each result in a cell of the table is a tuple $\langle t/n\rangle$, where t is the total checking time for the corresponding benchmark, and n is the number of solved formulas. Finally, the last row of the table lists the total checking time and number of solved formulas for each solver.

The results show that while the tableau-based tool Aalta_v1.2, outperforms TRP++, it is outperformed by ls4, NuXmv-BMCINC and NuXmv-IC3-Klive, all of which are outperformed by Aalta_v2.0, which is faster by about 6000 s and solves 47 more instances than the second best solver NuXmv-IC3-Klive.

Our framework is explicit and closest to that is underlaid behind Aalta_v1.2. From the results, Aalta_v2.0 with heuristic outperforms Aalta_v1.2 dramatically, faster by more than 23,000 s and solving 485 more instances. One reason is, when Aalta_v1.2 fails it is often due to timeout during the heavy-duty normal-form generation, which Aalta_v2.0 simply avoids (generating XNF is rather lightweight).

Generating the states in a lightweight way, however, is not efficient enough. By running Aalta_v2.0 without heuristics, it cannot perform better than Aalta_v1.2, see the data in column 6 and 8 of Table 1. It can even be worse in some benchmarks such as "/anzu/amba" and "anzu/genbuf". We can explain the reason via an example. Assume the formula is $\phi_1 \vee \phi_2$, the traditional tableau method splits the formula and creates at most two nodes. Under our pure SAT-reasoning framework, however, it may create three nodes which contain $\phi_1 \wedge \neg\phi_2$ or $\neg\phi_1 \wedge \phi_2$, or $\phi_1 \wedge \phi_2$. This indicates that the state space generated by SAT solvers may in general be larger than that generated by tableau expansion.

To overcome this challenge, we propose some heuristics by adding specific constraints to SAT solvers, which at the mean time succeeds to reduce the searching space of the overall system. The results shown in column 9 of Table 1 demonstrate the effectiveness of heuristics presented in the paper. For example, the "/trp/N12/" and "/forobots/" benchmarks are mostly unsatisfiable formulas, which Aalta_v1.2 and Aalta_v2.0 without heuristic do not handle well.

[4] http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml.

Yet the *unsat-core extraction* heuristic, which is described in the CONFLICT_ANALYZE mode of $getState$ function, enables Aalta_v2.0 with heuristic to solve all these formulas. For satisfiable formulas, the results from "/anzu/amba/" and "/anzu/genbuf" formulas, which are satisfiable, show the efficiency of the ELIMINATION and SAT_PURSUING heuristics in the $getState$ function, which are necessary to solve the formulas.

In summary, Aalta_v2.0 with heuristic performed best on satisfiable formulas, solving 3079 instances, followed in order by NuXmv-BMCIMC (3043), NuXmv-IC3-Klive (3029), ls4 (2999), Aalta_v1.2 (2902) and TRP++ (1852). For unsatisfiable formulas, NuXmv-IC3-Klive performs best, solving 620 instances, followed in order by Aalta_v2.0 with heuristic (617), ls4 (496), NuXmv-BMCINC (356), Aalta_v1.2 (309), and TRP++ (307). Detailed statistics are in Tables 2 and 3, where we omit the results from Aalta_v2.0 without heuristics. (So Aalta_v2.0 in the tables represents exactly Aalta_v2.0 with heuristics.) Notably, the sum of accumulated time in Tables 2 and 3 for each solver may not equal to the corresponding one shown in Table 1, as there are instances that cannot be solved by the solver and the cost to check these instances (60 s for each), does not count in Tables 2 and 3.

NuXmv-IC3-Klive is able to solve more cases than Aalta_v2.0 with heuristic in some benchmarks, such as "/alaska/lift", in which unsatisfiable formulas are not handled well enough by Aalta_v2.0. Currently, Aalta_v2.0 requires large number of SAT calls to identify an unsatisfiable core. In future work we plan to use a specialized MUS (minimal unsatisfable core) solver to address this challenge. Although ls4 and Aalta_v2.0 have similar SAT-based framework to compute states of the automata on-the-fly, they have different performance according to the results shown above. A conjecture to explain why their performance varies is because they use different heuristics to accelerate the satisfiability checking. More discussion is in Sect. 7.

The experimental results on *random conjunction formulas* are shown in Fig. 2. It shows that Aalta_v2.0 (with heuristic) performs best among tested solvers, and comparing to the second best solver (NuXmv), it achieves approximately a 30% speed-up.

7 Concluding remarks

We described in this paper a SAT-based framework for explicit LTL reasoning. We showed one of its applicaitons to LTL-satisfiability checking, by proposing basic algorithms and efficient heuristics. As proof of concept, we implemented an LTL satisfiability solver, whose performance dominates all similar tools against the benchmarks under test.

The tool ls4 [37] is implemented on a SAT-based framework as well. Aalta_v2.0 takes the original LTL formula as the input and internally translates the formula to its equivalent XNF, the Boolean form of which is the input of SAT solvers. Meanwhile, ls4 takes the SNF (Separated Normal Form [13]) of the original formula as the input and internally generates the TST (Temporal Satisfiability Tasks [36,37]) on which SAT solvers are applied. Notably, a SNF of the LTL formula is equi-satisfiable to, while the XNF used in the paper is equivalent to the original formula. For the methodologies, ls4 and Aalta_v2.0 have similar SAT-invoking principles, e.g. they both compute the states of the system (the system is essentially the Büch automaton) explicitly one by one such that the satisfiability checking is performed on the fly, and they both try to find a satisfiable lasso as soon as possible to accelerate checking satisfiable formulas. However, Aalta_v2.0 mainly differs ls4 by proposing the distinguished conflcit-driven heuristics to speed up the unsatisfiability checking. Aalta_v2.0 introduces the *avoidable sequence* to guide the search, while ls4 utilizes the *labeled layer* [36,37] to

Table 1 Experimental results on the Schuppan-collected benchmark

Type	Number	ls4	TRP++	NuXmv- BMCINC	Aalta_v1.2	NuXmv- IC3-Klive	Aalta_v2.0 $^{-}$	Aalta_v2.0
/acacia/example	25	155/25	192/25	1/25	1/25	8/25	1/25	1/25
/acacia/demo-v3	36	68/36	1417/17	3/36	660/36	30/36	630/36	3/36
/acacia/demo-v22	10	60/10	67/10	1/10	2/10	4/10	2/10	1/10
/alaska/lift	136	3190/103	7801/9	1919/110	4084/73	2867/112	4610/76	1431/109
/alaska/szymanski	4	27/4	240/0	1/4	1/4	2/4	1/4	1/4
/anzu/amba	51	1450/29	3060/0	536/44	2686/26	1062/43	1938/21	928/47
/anzu/genbuf	60	1100/45	3600/0	782/49	3343/16	1350/47	2672/13	827/56
/rozier/counter	76	4267/32	4291/32	4465/12	4328/16	4488/11	4328/21	2649/57
/rozier/formulas	2000	167/2000	37,533/1477	1258/1981	1372/1980	664/2000	1672/1975	363/2000
/rozier/pattern	244	2216/206	13,450/7	1505/236	8/244	3252/238	8/224	9/224
/schuppan/O1	28	1097/11	1069/11	14/28	2/28	95/27	2/28	2/28
/schuppan/O2	28	1142/14	1283/8	1581/2	2/28	742/20	2/28	2/28
/schuppan/phltl	18	885/5	896/4	1058/8	833/8	753/12	660/8	767/14
/trp/N5x	240	144/240	46/240	567/231	1309/218	187/240	219/240	15/240
/trp/N5y	140	448/130	95/139	2768/94	1216/104	102/140	316/140	16/140
/trp/N12x	400	1672/374	22,869/33	3570/342	9768/240	705/400	768/400	175/400
/trp/N12y	240	1905/192	8021/108	4049/173	7413/130	979/240	7413/140	154/239
/forobots	39	990/39	1303/39	1085/21	2280/24	37/39	2130/9	524/39
Total	3775	20,983/3495	103,142/2159	24,769/3399	31,208/3211	14,261/3649	30,554/3350	7868/3696

Each cell from Column 3–9 lists a tuple $\langle t/n \rangle$ where t is the total checking time (in seconds), and n is the total number of solved formulas. In the last two columns, Aalta_v2.0 (resp. Aalta_v2.0 $^{-}$) represents the implementation with (resp. without) heuristics

Table 2 Experimental results on the Schuppan-collected benchmark for satisfiable formulas

Type	Number	ls4	TRP++	NuXmv-BMCINC	Aalta_v1.2	NuXmv-IC3-Klive	Aalta_v2.0
/acacia/example	25	76/25	86/25	1/25	1/25	8/25	1/25
/acacia/demo-v3	36	748/36	554/17	3/36	3/36	30/36	3/36
/acacia/demo-v22	10	30/10	37/10	1/10	1/10	4/10	1/10
/alaska/lift	136	487/81	322/7	282/102	2084/63	529/83	367/79
/alaska/szymanski	4	27/4	43/4	1/4	1/4	2/4	1/4
/anzu/amba	51	100/29	0/0	116/44	1186/26	582/43	273/47
/anzu/genbuf	60	225/45	0/0	122/49	843/16	570/47	422/54
/rozier/counter	76	1214/32	1851/32	25/12	328/16	88/11	289/57
/rozier/formulas	2000	163/1890	6087/1370	88/1890	1372/1880	649/1890	28/1890
/rozier/pattern	244	936/206	330/7	1025/236	8 /244	2232/238	9/244
/schuppan/phltl	18	87/5	33/1	135/4	233/4	78/5	1/10
/trp/N5x	240	81/131	83/131	10/131	309/130	145/131	12/131
/trp/N5y	140	334/84	331/94	8/94	16/94	84/94	10/94
/trp/N12x	400	4425/225	1639/33	33/225	768/220	531/220	94/225
/trp/N12y	240	3173/173	3062/104	29/173	8413/120	318/173	12/173
/forobots	39	696/14	914/14	3/14	280/14	27/14	30/14
Total	3775	12,002/2999	16,556/1852	1994/3043	14,451/2902	6189/3029	1554/3079

Each cell lists a tuple $\langle t, n\rangle$ where n is the total number of solved formulas and t is the total checking time for solving these n cases (in seconds). Note that the pattern formulas that are all unsatisfiable are omitted, and the special tuple $\langle 0/0\rangle$ means the solver cannot solve any satisfiable formulas

Table 3 Experimental results on the Schuppan-collected benchmark for unsatisfiable formulas

Type	Number	ls4	TRP++	NuXmv-BMCINC	Aalta_v1.2	NuXmv-IC3-Klive	Aalta_v2.0
/alaska/lift	136	73/22	3/2	77/8	384/10	38/29	544/30
/rozier/formulas	2000	4/110	66/107	29/91	40/100	15/110	1/110
/schuppan/O1	28	103/10	27/9	7/28	2/28	36/27	2/28
/schuppan/O2	28	106/9	16/3	3/2	2/28	69/20	5/28
/schuppan/phltl	18	0/0	19/3	22/4	89/4	15/7	36/4
/trp/N5x	240	62/109	62/109	17/100	139/88	42/109	8/109
/trp/N5y	140	113/46	104/45	0/0	130/10	18/46	16/46
/trp/N12x	400	621/146	0/0	56/117	456/20	174/175	64/175
/trp/N12y	240	277/19	180/4	0/0	34/13	95/67	238/66
/forobots	39	293/25	388/25	2/7	280/10	10/25	32/25
Total	3775	1723/496	906/307	215/356	1556/309	512/620	946/617

Each cell lists a tuple $\langle t/n \rangle$ where n is the total number of solved formulas and t is the total checking time for solving these n cases (in seconds). Note that the pattern formulas that are all satisfiable are omitted, and the special tuple $\langle 0/0 \rangle$ means the solver cannot solve any unsatisfiable formulas

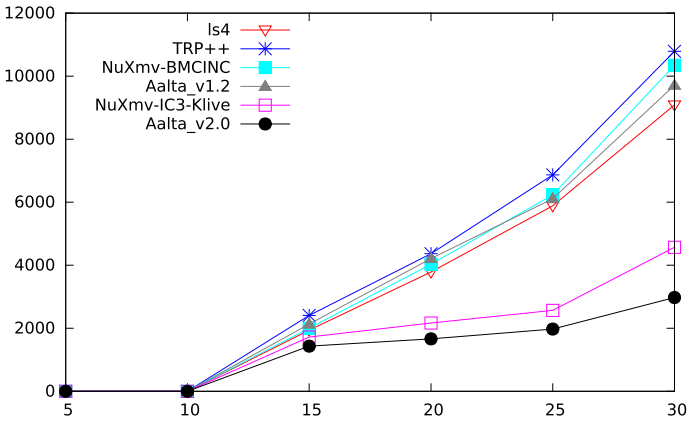

Fig. 2 Results for LTL-satisfiability checking on Random Conjunction Formulas. X-axis represents the number of conjuncts, and Y-axis represents the total checking time (s)

record the proofs and conflicts found during search. During the checking process, there is only one avoidable sequence maintained at one time by Aalta_v2.0, while multiple layers have to be recorded in ls4. The unsatisfiability checking in Aalta_v2.0 is achieved by finding an invariant on the avoidable sequence, which indicates the obligations recorded at the head of the sequence can never be satisfied. ls4 relies on detecting a *layer repetition* [37], which indicates also that the goals in the TST cannot be satisfied. The invariant of the avoidable sequence can be checked semantically by leveraging SAT computations, while the layer repetition is recognized syntactically by detecting exactly the same two layers, as described in [37].

Our approach also utilize the *Minimal Unsat Subset* (MUS) [27] technique as a key heuristics for LTL satisfiability checking. Although MUS is an expensive procedure in theory, our experiments show that it contributes significantly to the performance of our new approach. To temper the cost consumed by MUS, a solution may be to use some lighter techniques like "prime implicant" [26]. We will explore this direction in our future work. A relevant work [42] also utilizes the MUS techniques. However, instead of computing one MUS, [42] requires to compute all the MUS together at one time, which results in a much heavier task. Besides that, the approach in [42] takes the SNF as the input, and therefore is also underlied by the symbolic checking strategy.

Extending the explicit SAT-based approach to other applications of LTL reasoning, is a promising research direction. For example, the standard approach in LTL model checking [41] relies on the translation of LTL formulas to Büchi automata. The transition systems T_ϕ that is used for LTL satisfiability checking can also be used in the translation from LTL to Büchi automata. Current best-of-breed translators, e.g., [9,10,14,35] are tableau-based, and the SAT approach may yield significant performance improvement.

Of course, the ultimate temporal-reasoning task is model checking. Explicit model checkers such as SPIN [17] start with a translation of LTL to Büchi automata, which are then used by the model-checking algorithm. An alternative approach could be to construct the automaton on the fly, using the SAT-based framework developed here. Although on-the-fly automata-based model checking has been proposed for decades [14], there is no such extant model checker that supports this mechanism. One reason may be for traditional tableau construction, computing all next states of the current state at a time is still expensive. Our

SAT-based approach suggests a lighter way to compute one next state of the current state at a time, and thus may be also useful for the explicit model checking area. This remains a highly intriguing research possibility.

Acknowledgements The authors thank anonymous reviewers for useful comments. The work is supported in part by NSF Grants CCF-1319459, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", and by BSF Grant 9800096. Jianwen Li is partially supported by NSFC Projects No. 61572197 and No. 61632005. Geguang Pu is partially supported by MOST NKTSP Project 2015BAG19G02 (Grant No. ZF1213) and STCSM Project No. 16DZ1100600. Lijun Zhang is supported by NSFC Grant No. 61532019.

References

1. Bertello M, Gigante N, Montanari A, Reynolds M (2016) Leviathan: a new LTL satisfiability checking tool based on a one-pass tree-shaped tableau. In: Proceedings of the twenty-fifth international joint conference on artificial intelligence, IJCAI'16, pp 950–956. AAAI Press. http://dl.acm.org/citation.cfm?id=3060621.3060753
2. Bradley A (2011) SAT-based model checking without unrolling. In: Jhala R, Schmidt D (eds) Verification, model checking, and abstract interpretation. Lecture notes in computer science, vol 6538. Springer, Berlin, pp 70–87
3. Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S (2014) The nuXmv symbolic model checker. In: CAV, pp 334–342
4. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: an opensource tool for symbolic model checking. In: Brinksma E, Larsen KG (eds) Computer aided verification. Lecture notes in computer science, vol 2404. Springer, Berlin, pp 359–364
5. Claessen K, Sörensson N (2012) A liveness checking algorithm that counts. In: Cabodi G, Singh S (eds) FMCAD, pp 52–59. IEEE
6. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
7. Courcoubetis C, Vardi M, Wolper P, Yannakakis M (1992) Memory efficient algorithms for the verification of temporal properties. Form Methods Syst Des 1:275–288
8. D'Agostino M (1999) Tableau methods for classical propositional logic. In: D'Agostino M, Gabbay D, Haehnle R, Posegga J (eds) Handbook of tableau methods. Springer, Dordrecht, pp 45–123
9. Daniele N, Guinchiglia F, Vardi M (1999) Improved automata generation for linear temporal logic. In: Proceedings of the 11th international conference on computer aided verification. Lecture notes in computer science, vol 1633. Springer, Berlin, pp 249–260
10. Duret-Lutz A, Poitrenaud D (2004) SPOT: an extensible model checking library using transition-based generalized büchi automata. In: Proceedings of the 12th international workshop on modeling, analysis, and simulation of computer and telecommunication systems. IEEE Computer Society, pp 76–83
11. Eén N, Sörensson N (2003) An extensible SAT-solver. In: SAT, pp 502–518
12. Fisher M (1997) A normal form for temporal logics and its applications in theorem-proving and execution. J Log Comput 7(4):429–456
13. Fisher M, Dixon C, Peim M (2001) Clausal temporal resolution. ACM Trans Comput Log 2(1):12–56
14. Gerth R, Peled D, Vardi M, Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic. In: Dembiski P, Sredniawa M (eds) Protocol specification, testing, and verification. Chapman & Hall, Boca Raton, pp 3–18
15. Giunchiglia F, Sebastiani R (1996) Building decision procedures for modal logics from propositional decision procedure—the case study of modal K. In: Proceedings of 13th international conference on automated deduction. Lecture notes in computer science, vol 1104. Springer, Berlin, pp 583–597
16. Heljanko K, Junttila T, Latvala T (2005) Incremental and complete bounded model checking for full PLTL. In: Etessami K, Rajamani S (eds) Computer aided verification. Lecture notes in computer science, vol 3576. Springer, Berlin, pp 98–111
17. Holzmann G (2003) The SPIN model checker: primer and reference manual. Addison-Wesley, Boston
18. Hustadt U, Konev B (2003) Trp++ 2.0: a temporal resolution prover. In: Proceedings of CADE-19. LNAI. Springer, Berlin, pp 274–278
19. Kaminski M, Tebbi T (2013) Inkresat: modal reasoning via incremental reduction to sat. In: Bonacina MP (ed) International conference on automated deduction—CADE-24. Springer, Berlin, pp 436–442

20. Larrabee T (1992) Test pattern generation using boolean satisfiability. IEEE Trans Comput Aided Des Integr Circuits Syst 11(1):4–15
21. Li J, Pu G, Zhang L, Vardi MY, He J (2014) Fast LTL satisfiability checking by SAT solvers. CoRR arXiv:1401.5677
22. Li J, Zhang L, Pu G, Vardi M, He J (2013) LTL satisfibility checking revisited. In: The 20th international symposium on temporal representation and reasoning, pp 91–98
23. Li J, Zhu S, Pu G, Vardi M (2015) SAT-based explicit LTL reasoning. Springer, Berlin, pp 209–224
24. Malik S, Zhang L (2009) Boolean satisfiability from theoretical hardness to practical success. Commun ACM 52(8):76–82
25. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems: specification. Springer, Berlin
26. Manquinho VM, Flores PF, Silva JPM, Oliveira AL (1997) Prime implicant computation using satisfiability algorithms. In: Proceedings of ninth IEEE international conference on tools with artificial intelligence, pp 232–239
27. Marques-Silva J, Lynce I (2011) On improving MUS extraction algorithms. In: Sakallah K, Simon L (eds) Theory and applications of satisfiability testing—SAT 2011. Lecture notes in computer science, vol 6695. Springer, Berlin, pp 159–173
28. McMillan K (1993) Symbolic model checking. Kluwer Academic Publishers, Dordrecht
29. McMillan K (2003) Interpolation and SAT-based model checking. In: Hunt WA Jr., Somenzi F (eds) International conference on computer aided verification. Lecture notes in computer science, vol 2725. Springer, Berlin, pp 1–13
30. Pnueli A (1977) The temporal logic of programs. In: Proceedings of 18th IEEE symposium on foundations of computer science, pp 46–57
31. Rozier K, Vardi M (2010) LTL satisfiability checking. Int J Softw Tools Technol Transf 12(2):123–137
32. Schuppan V, Darmawan L (2011) Evaluating LTL satisfiability solvers. In: Proceedings of the 9th international conference on automated technology for verification and analysis, ATVA'11. Springer, Berlin, pp 397–413
33. Schwendimann S (1998) A new one-pass tableau calculus for PLTL. In: Proceedings of the international conference on automated reasoning with analytic tableaux and related methods. Springer, Berlin, pp 277–292
34. Schwoon S, Esparza J (2005) A note on on-the-fly verification algorithms. In: Proceedings 11th international conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 3440. Springer, Berlin, pp 174–190
35. Somenzi F, Bloem R (2000) Efficient Büchi automata from LTL formulae. In: Proceedings of 12th international conference on computer aided verification. Lecture notes in computer science, vol 1855. Springer, Berlin, pp 248–263
36. Suda M (2015) Variable and clause elimination for LTL satisfiability checking. Math Comput Sci 9(3):327–344
37. Suda M, Weidenbach C (2012) A PLTL-prover based on labelled superposition with partial model guidance. In: International joint conference on automated reasoning. Lecture notes in computer science, vol 7364. Springer, Berlin, pp 537–543
38. Tabakov D, Rozier K, Vardi MY (2012) Optimized temporal monitors for SystemC. Form Methods Syst Des 41(3):236–268
39. Vardi M (1989) On the complexity of epistemic reasoning. In: Proceedings of the fourth annual symposium on logic in computer science. IEEE Press, Piscataway, pp 243–252
40. Vardi M (1989) Unified verification theory. In: Banieqbal B, Barringer H, Pnueli A (eds) Proceedings of temporal logic in specification, vol 398. Springer, Berlin, pp 202–212
41. Vardi M, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of 1st IEEE symposium on logic in computer science, pp 332–344
42. Williams R, Konev B (2013) Propositional temporal proving with reductions to a sat problem. In: Bonacina MP (ed) International conference on automated deduction—CADE-24. Springer, Berlin, pp 421–435