https://doi.org/10.1007/s00165-017-0442-2
BCS © 2017
Formal Aspects of Computing (2018) 30: 193–217

Formal Aspects of Computing

An explicit transition system construction approach to LTL satisfiability checking

Jianwen Li[1], Lijun Zhang[2], Shufang Zhu[1], Geguang Pu[1], Moshe Y. Vardi[3] and Jifeng He[1]

[1] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] Computer Science, Rice University, Houston, TX, USA

Abstract. We propose a novel algorithm for the satisfiability problem for linear temporal logic (LTL). Existing automata-based approaches first transform the LTL formula into a Büchi automaton and then perform an emptiness checking of the resulting automaton. Instead, our approach works on-the-fly by inspecting the formula directly, thus enabling to find a satisfying model quickly without constructing the full automaton. This makes our algorithm particularly fast for satisfiable formulas. We construct experiments on different pattern formulas, the experimental results show that our approach is superior to other solvers under automata-based framework.

Keywords: LTL satisfiability checking, Obligation set, LTL transition system

1. Introduction

Model-checking is an autonomous technique checking whether systems have desired properties [CGP99]. When the model does not satisfy a given property, model-checking tools also return a counterexample, which explains the inconsistency between the system model and desired behaviors. Model checking has led to the emergence of assertion-based design, in which the designers first formalize their intent by means of temporal assertions [FKL04]. At early stage of this process, models do not—or only partially—exist, thus model checking cannot be employed yet. On the other side, it is quite likely that such assertions contain errors [PSC+06]. A basic check of these assertions is that of *satisfiability* [SC85]: checking that each temporal assertion can be satisfied by some model and the full set of assertions be satisfied together.

An in-depth empirical study of LTL satisfiability was undertaken by Rozier and Vardi [RV07, RV10]. A basic observation underlying their work is that LTL satisfiability checking can be reduced to model checking. Consider an LTL formula φ over a set *Prop* of atomic propositions. If a model M is *universal*, that is, it contains all possible traces over *Prop*, then φ is satisfiable precisely when the model M does *not* satisfy $\neg\varphi$. Thus, it is easy to add a satisfiability-checking feature to LTL model-checking tools.

Correspondence and offprint requests to: S. Zhu and G. Pu, E-mails: sei_zsf2010@126.com; ggpu@sei.ecnu.edu.cn

LTL model checkers can be classified as *explicit* or *symbolic*. Explicit model checkers, such as SPIN [Hol97] or SPOT [DLP04], construct the state-space of the model explicitly and search for a trace falsifying the assertion [CVWY92]. In contrast, symbolic model checkers, such as CadenceSMV [McM99] or NuSMV [CCGR00], represent the model and analyze it symbolically using binary decision diagrams (BDDs) [BCM+92]. Generally, LTL model checkers follow the automata-theoretic approach [VW86], in which the complemented LTL assertion is explicitly or symbolically translated to a Büchi automaton, which is then composed with the model under verification; see also [Var07]. The model checker checks for *nonemptiness*, by searching for a trace of the model that is accepted by the automaton.

Rozier and Vardi [RV07, RV10] carried out an extensive experimental investigation of LTL satisfiability checking via a reduction to model checking. By using large LTL formulas, they offered challenging model-checking benchmarks to both explicit and symbolic model checkers. For symbolic model checking, they used CadenceSMV and NuSMV. For explicit model checking, they used SPIN as the search engine, and tested essentially all publicly available LTL translation tools. They used a wide variety of benchmark formulas, either generated randomly, as in [DGV99], or using scalable patterns.

Rozier and Vardi reached two major conclusions. First, most LTL translation tools are research prototypes and cannot be considered industrial quality tools. Among all the tools tested, only SPOT can be considered an industrial quality tool. Second, when it comes to LTL satisfiability checking, the symbolic approach is clearly superior to the explicit approach. Even SPOT, the best LTL translator in our experiments, was rarely able to compete effectively against the symbolic tools.

The evidence marshaled by Rozier and Vardi for the conclusion in favor of the symbolic approach is quite compelling, but a close examination shows that it applies only to satisfiability checking via model checking. That is, if one chooses to perform satisfiability checking via a reduction to model checking, then the symbolic approach offers superior performance. It is conceivable, however, that a direct explicit approach to satisfiability checking would outperform the symbolic approach. In explicit model checking, it is possible to perform the nonemptiness test *on the fly*, that is, by letting the search algorithm drive the construction of the automaton [CVWY92]. (In fact, the on-the-fly approach was proposed also for model checking, but was not adopted by SPIN due to its software architecture [Hol97]).

In this paper we revisit the LTL satisfiability problem to examine the advantage of the on-the-fly approach. The driving intuition is that the on-the-fly approach may be quite advantageous in satisfiability checking, since it enables finding a model quickly without constructing the full automaton. Furthermore, the sole focus on satisfiability checking may be amenable to various heuristics that are not applicable in the context of model checking. We report here on a novel LTL satisfiability checking tool, *Aalta*, and demonstrate that it outperforms both SPOT and CadenceSMV.

For a sketch of our approach, we first propose the concept of *obligation set* for every LTL formula, which is a set of *obligations*. An *obligation* is a set of literals that should be fulfilled before the formula can be satisfied. The whole *obligation set* thus collects all possibilities that can make the formula satisfiable. By observing the *obligations*, there is an interesting and useful property that a *consistent obligation* directly implies the satisfiability of the formula. For example, take the formula with the pattern of $\varphi U a$ where φ means arbitrary formula. This until formula has the obligation {a}, (obligation set {{a}}) based on our definition later. Meanwhile, {a} is a consistent obligation due to the sole boolean formula a, which makes the formula $\varphi U a$ be satisfiable. The heuristics can highly speed up the checking process for satisfiable cases. In the general case, if no *consistent obligation* can be found in the formula, we expand the formula with the on-the-fly manner to search an *accepting SCC* (Strongly Connected Component)in the expanded system (which we call the *transition system*). During the process the algorithm returns SAT if either a *consistent obligation* or an *accepting SCC* is found. Otherwise in the worst case, the algorithm returns UNSAT after the whole *transition system* has been explored.

To substantiate our approach we first revisit the experimental methodology of Rozier and Vardi [RV11]. Their focus has been on testing satisfiability of large LTL formulas, either scalable patterns or random ones. But typical temporal assertions are rather small [DAC98]. What makes the LTL satisfiability problem hard is the fact that we need to check *large conjunctions of small temporal formulas*. We describe here a new class of challenging benchmarks, which are random conjunctions of specification patterns from [DAC98]. Our conclusions on the superiority of *Aalta* are based both on the benchmarks of Rozier and Vardi and the newly introduced benchmarks.

An early version [LZP+13] of this approach has been published. This paper is an extension of the previous version and contains the following additional work. (1) The missing proofs are included. We provide a more detailed and precise proof of the central theorem. We introduce the notion of looping formula and discover more properties of it which is presented in Sect. 3.5. Then we prove the soundness and completeness of the central theorem, which gives a boost to our algorithm. (2) In the experimental part, we present the architecture of our tool *Aalta* and introduce the implementation of it. (3) We also did more experiments on missing pattern formulas and the experimental results in 4.2.3 can show the effectiveness of the accelerating technique.

The organization of the paper is as follows. We provide preliminary materials in Sect. 2. In Sect. 3, we describe the novel algorithm underlying *Aalta*. In Sect. 4, we detail our experimental methodology and experimental results. Section 5 discusses related work and Sect. 6 concludes the paper.

2. Preliminaries

2.1. Linear temporal logic

Let AP be a set of atomic properties. The syntax of LTL formulas is defined by:

$$\varphi ::= \mathtt{tt} \mid \mathtt{ff} \mid a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi U \varphi \mid \varphi R \varphi \mid X\varphi$$

where $a \in AP$ is called an *atom*, φ is an LTL formula. We use the usual abbreviations: $Fa = \mathtt{tt}\, Ua$, and $Ga = \mathtt{ff}\, Ra$.

We say φ is a propositional formula if it does not contain temporal operators. We say φ is a *literal* if φ is an atom or its negation. We use L to denote the set of literals, lower case letters a, b, c for atoms, $l_i (i \geq 0)$ for literals, α, β, γ for propositional formulas, and λ, φ, ψ for LTL formulas. Note that, w.l.o.g., we are considering LTL formulas in negation normal form (NNF)—all negations are pushed in front of only atoms. LTL formulas are often interpreted over $(2^{AP})^{\omega}$. Since we consider LTL in NNF forms, literals are the unit and thus formulas are considered to be interpreted on infinite sequences over the alphabet $\Sigma := 2^L$, i.e., infinite sequences of sets of literals.

A *trace* $\xi = \omega_0\omega_1\omega_2 \ldots$ is an infinite sequence over Σ^{ω}. For ξ and $k \geq 1$ we use $\xi^k = \omega_0\omega_1 \ldots \omega_{k-1}$ to denote the prefix of ξ up to its k-th element, and $\xi_k = \omega_k\omega_{k+1} \ldots$ to denote the suffix of ξ from its $(k+1)$-th element. Thus, $\xi = \xi^k\xi_k$. We use $\eta, \eta_0 \ldots$ to denote finite sequences in Σ^*. Let A be a literal set and denote that $\bigwedge A = a_1 \wedge a_2 \wedge \ldots \wedge a_n$ when $\mid A \mid = n$ and $a_i \in A (1 \leq i \leq n)$. Then we define the notion of *consistent traces* first:

Definition 2.1 (*Consistent trace*). We say a literal set A is *consistent* iff A does not contain an atom and its negation at the same time. A trace $\xi = \omega_0\omega_1 \ldots$ is consistent iff ω_i is consistent for all i.

For example, the literal set $\{a, b\}$ is consistent, while $\{a, \neg a\}$ is not according to the definition.

Let $\omega \in \Sigma$ be a consistent set of literals, and α a propositional formula. We define $\omega \models \alpha$ in the standard way: if α is a literal then $\omega \models \alpha$ iff $\alpha \in \omega$, $\omega \models \alpha_1 \wedge \alpha_2$ iff $\omega \models \alpha_1$ and $\omega \models \alpha_2$, and $\omega \models \alpha_1 \vee \alpha_2$ iff $\omega \models \alpha_1$ or $\omega \models \alpha_2$. Moreover, $\omega \models \mathtt{tt}$ and $\omega \not\models \mathtt{ff}$.

The semantics of temporal operators with respect to a consistent trace ξ is given by: $\xi \models \alpha$ iff $\xi^1 \models \alpha$; $\xi \models X\,\varphi$ iff $\xi_1 \models \varphi$; and

1. $\xi \models \varphi_1\ U\ \varphi_2$ iff there exists $i \geqslant 0$ such that $\xi_i \models \varphi_2$ and for all $0 \leqslant j < i$, $\xi_j \models \varphi_1$;
2. $\xi \models \varphi_1\ R\ \varphi_2$ iff either $\xi_i \models \varphi_2$ for all $i \geq 0$, or there exists $i \geq 0$ with $\xi_i \models \varphi_1 \wedge \varphi_2$ and $\xi_j \models \varphi_2$ for all $0 \leq j < i$;

According to the semantics, it holds that $\varphi R \psi = \neg(\neg\varphi U \neg\psi)$. For two LTL formulas φ_1 and φ_2, we denote $\varphi_1 \equiv \varphi_2$ iff φ_1 and φ_2 are semantically equivalent, i.e. for any trace ξ, $\xi \models \varphi_1$ iff $\xi \models \varphi_2$.

Definition 2.2 (*Satisfiability*). We say φ is satisfiable, denoted by $SAT(\varphi)$, if there exists a consistent trace ξ such that $\xi \models \varphi$.

In the remainder of this paper, if not stated explicitly, all traces considered are assumed to be consistent.
Notation. We define some notation that we use throughout this paper.

- For a formula φ, we use $cl(\varphi)$ to denote the set of subformulas of φ. We denote by AP_φ the set of atoms appearing in φ, by L_φ the set of literals over AP_φ, and by Σ_φ the set of consistent literal sets over AP_φ.
- Let $\varphi = \bigwedge_{i\in I}\varphi_i$ such that the root operator of φ_i is not a conjunctive. We define the set of conjuncts of φ as $CF(\varphi) := \{\varphi_i \mid i \in I\}$. When φ does not include a conjunctive as a root operator, $CF(\varphi)$ includes only φ itself. The set of disjuncts $DF(\varphi)$ is defined in an analogous way.
- For a formula φ of the form $\varphi_1 U \varphi_2$ or $\varphi_1 R \varphi_2$, let *left*(φ) (*right*(φ)) be left (right) subformulas of φ.
- For each propositional formula α appearing in the paper, we always check first whether α is satisfiable. If not, we shall replace α by ff.

2.2. Normal form expansion

Our algorithm extends the given formula based on the notion of *normal form* for LTL formulas defined as follows:

Definition 2.3 (*Normal form*). The *normal form* of an LTL formula φ, denoted as $NF(\varphi)$, is a set defined as follows:

1. $NF(\varphi) = \{\varphi \wedge X(\mathsf{tt})\}$ if $\varphi \not\equiv \mathsf{ff}$ is a propositional formula. If $\varphi \equiv \mathsf{ff}$, we define $NF(\mathsf{ff}) = \emptyset$;
2. $NF(X\varphi) = \{\mathsf{tt} \wedge X(\psi) \mid \psi \in DF(\varphi)\}$;
3. $NF(\varphi_1 U \varphi_2) = NF(\varphi_2) \cup NF(\varphi_1 \wedge X(\varphi_1 U \varphi_2))$;
4. $NF(\varphi_1 R \varphi_2) = NF(\varphi_1 \wedge \varphi_2) \cup NF(\varphi_2 \wedge X(\varphi_1 R \varphi_2))$;
5. $NF(F\varphi_1) = NF(\varphi_1) \cup NF(XF\varphi_1)$;
6. $NF(G\varphi_1) = NF(\varphi_1 \wedge XG\varphi_1)$;
7. $NF(\varphi_1 \vee \varphi_2) = NF(\varphi_1) \cup NF(\varphi_2)$;
8. $NF(\varphi_1 \wedge \varphi_2) = \{(\alpha_1 \wedge \alpha_2) \wedge X(\psi_1 \wedge \psi_2) \mid \forall\, i = 1,2.\ \alpha_i \wedge X(\psi_i) \in NF(\varphi_i)\}$;

For each $\alpha \wedge X\varphi_i \in NF(\varphi)$, we call α a *current formula* and φ_i is a *next formula* of φ. From the definition it is obvious that if $\alpha \wedge X(\psi) \in NF(\varphi)$, then α is a conjunctive clause, namely a conjunction of literals.

For a formula φ, our algorithm will detect successor formulas based on the set $NF(\varphi)$. First, we shall show that the formula φ is logically equivalent to $\bigvee NF(\varphi)$, here $\bigvee NF(\varphi)$ represents the formula $\bigvee_{1\le j\le k}(\alpha_j \wedge X\varphi_j)$ with $\alpha_j \wedge X\varphi_j \in NF(\varphi)$ and $k = \mid NF(\varphi) \mid$. We note that the empty disjunction (OR-ing over an empty set of operands) is defined as ff. Then, we establish the equivalence property:

Theorem 2.1 For the formula φ, it holds that $\varphi \equiv \bigvee NF(\varphi)$.

Proof We prove it by structural induction over φ:

- The case that φ is a propositional formula or a next formula is trivial by definition.
- If $\varphi = \varphi_1 \vee \varphi_2$, then applying induction hypothesis we have $\varphi \equiv \varphi_1 \vee \varphi_2 \equiv \bigvee NF(\varphi_1) \vee \bigvee NF(\varphi_2) \equiv \bigvee(NF(\varphi_1) \cup NF(\varphi_2)) \equiv \bigvee NF(\varphi_1 \vee \varphi_2)$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then applying induction hypothesis we have $\varphi \equiv \varphi_1 \wedge \varphi_2 \equiv (\bigvee NF(\varphi_1)) \wedge (\bigvee NF(\varphi_2))$. By inspection, this is equivalent to $\bigvee NF(\varphi_1 \wedge \varphi_2)$.
- If $\varphi = \varphi_1 U \varphi_2$, then by Definition 2.3 we know $NF(\varphi) = NF(\varphi_2) \cup NF(\varphi_1 \wedge X\varphi)$, and that $\varphi \equiv \varphi_2 \vee (\varphi_1 \wedge X\varphi)$. By induction hypothesis we have that $\varphi_2 \equiv \bigvee NF(\varphi_2)$. Moreover, we also proved previously that $\varphi_1 \wedge X\varphi \equiv \bigvee NF(\varphi_1 \wedge X\varphi)$. Thus we can prove that $\varphi \equiv \varphi_2 \vee (\varphi_1 \wedge X\varphi) \equiv \bigvee NF(\varphi_2) \vee \bigvee NF(\varphi_1 \wedge X\varphi) \equiv \bigvee NF(\varphi)$;
- If $\varphi = \varphi_1 R \varphi_2$, then from Definition 2.3 we have $NF(\varphi) = NF(\varphi_1 \wedge \varphi_2) \cup NF(\varphi_2 \wedge X\varphi)$ and $\varphi \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_2 \wedge X\varphi)$. By induction hypothesis we have that $\varphi_1 \wedge \varphi_2 \equiv \bigvee NF(\varphi_1 \wedge \varphi_2)$. Moreover, we also proved previously that $\varphi_2 \wedge X\varphi \equiv \bigvee NF(\varphi_2 \wedge X\varphi)$. Thus we can prove that $\varphi \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_2 \wedge X\varphi) \equiv \bigvee NF(\varphi_1 \wedge \varphi_2) \vee \bigvee NF(\varphi_2 \wedge X\varphi) \equiv \bigvee NF(\varphi)$; □

The theorem below states that along the expansion the set of subformulas is non-increasing(except the constant tt):

Theorem 2.2 If $\alpha \wedge X\psi \in NF(\varphi)$, then $CF(\psi) \subseteq cl(\varphi) \cup \{\mathsf{tt}\}$.

Proof First, let $cl'(\varphi) = cl(\varphi) \cup \{\mathtt{tt}\}$. We prove that $CF(\psi) \subseteq cl'(\varphi)$ by structural induction over φ. The base cases $\varphi = \mathtt{tt}, \mathtt{ff}$ and propositional formulas are trivial. Otherwise:

- If $\varphi = \varphi_1 \vee \varphi_2$. Then $NF(\varphi) = NF(\varphi_1) \cup NF(\varphi_2)$. So $\alpha \wedge X\psi \in NF(\varphi_i)$ with $i = 1$ or $i = 2$. By induction hypothesis we have $CF(\psi) \subseteq cl'(\varphi_i) \subseteq cl'(\varphi)$.
- If $\varphi = X\varphi_1$. In this case we have $NF(\varphi) = \{\mathtt{tt} \wedge X\varphi_1' \mid \varphi_1' \subseteq DF(\varphi_1)\}$. Since $CF(\varphi_1') \subseteq cl'(\varphi_1) \subseteq cl'(\varphi)$, so we have $CF(\psi) \subseteq cl'(\varphi)$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then we know for every $\alpha \wedge X\psi \in NF(\varphi)$ there exists $\alpha_1 \wedge X\psi_1 \in NF(\varphi_1)$ and $\alpha_2 \wedge X\psi_2 \in NF(\varphi_2)$ such that $\alpha = \alpha_1 \wedge \alpha_2$ and $\psi = \psi_1 \wedge \psi_2$. Since by induction hypothesis we know $CF(\psi_1) \subseteq cl'(\varphi_1)$ and $CF(\psi_2) \subseteq cl'(\varphi_2)$, so $CF(\psi) \subseteq cl'(\varphi)$ holds.
- If $\varphi = \varphi_1 U \varphi_2$ then we have two cases: Either we have the right expansion $\alpha \wedge X\psi \in NF(\varphi_2)$, in which case $CF(\psi) \subseteq cl'(\varphi_2) \subseteq cl'(\varphi)$ follows directly by induction hypothesis. For the left expansion case, we have $\alpha \wedge X\psi \in NF(\varphi_1 \wedge X\varphi)$, implying that there exists $\alpha_1 \wedge X\psi_1 \in NF(\varphi_1)$ such that $\psi = \psi_1 \wedge \varphi$. So $CF(\psi) \subseteq cl'(\varphi)$ follows by exploiting the induction hypothesis that $CF(\psi_1) \subseteq cl'(\varphi_1) \subseteq cl'(\varphi)$.
- If $\varphi = \varphi_1 R \varphi_2$ then we also have two cases: Either we have $\alpha \wedge X\psi \in NF(\varphi_1 \wedge \varphi_2)$, in which case $CF(\psi) \subseteq cl'(\varphi_1 \wedge \varphi_2) \subseteq cl'(\varphi) \cup \{\varphi_1 \wedge \varphi_2\}$ follows directly by induction hypothesis. Since $\varphi_1 \wedge \varphi_2$ cannot be in $CF(\psi)$ as from the definition of $CF(\psi)$ a conjunction formula cannot be its element. For the other case, we have $\alpha \wedge X\psi \in NF(\varphi_2 \wedge X\varphi)$, implying that there exists $\alpha_1 \wedge X\psi_1 \in NF(\varphi_2)$ such that $\psi = \psi_1 \wedge \varphi$. So $CF(\psi) \subseteq cl'(\varphi)$ follows by exploiting the induction hypothesis that $CF(\psi_1) \subseteq cl'(\varphi_2) \subseteq cl'(\varphi)$. □

2.3. LTL transition system

Let φ be an LTL formula and $\alpha \wedge X\varphi_1$ be an element of $NF(\varphi)$. Essentially $\alpha \wedge X\varphi_1 \in NF(\varphi)$ means that if the current property α can be satisfied, then φ can go to the next state (formula) φ_1. Recursively the same process can be done on φ_1 and there are also new next formulas generating from φ_1. Following this intuition, we can construct a *transition system* for φ, in which all next formulas (including φ) are states and current formulas are labels on the edges. Formally, for a given formula φ we define as below a labeled transition system T_φ:

Definition 2.4 (*LTL transition system*). Let φ be the input formula. The labeled transition system T_φ is a tuple $\langle Act, S_\varphi, \rightarrow, \varphi\rangle$ where:

1. φ is the initial state;
2. Act is the set of conjunctive formulas over L_φ;
3. the transition relation $\rightarrow \subseteq S_\varphi \times Act \times S_\varphi$ is defined by: $\psi_1 \xrightarrow{\alpha} \psi_2$ iff there exists $\alpha \wedge X(\psi_2) \in NF(\psi_1)$;
4. S_φ is the smallest set of formulas such that $\psi_1 \in S_\varphi$, and $\psi_1 \xrightarrow{\alpha} \psi_2$ implies $\psi_2 \in S_\varphi$.

From the definition above, the set of states is the set of formulas reachable from φ, with φ being the initial state. Also there can be more transitions between two states. There is no outgoing transition in a state φ iff for all $\alpha \wedge X\psi \in NF(\varphi)$ s.t. α is equivalent to ff. In this case φ is not satisfiable. Now we introduce the notion of accepting traces:

Definition 2.5 A *run* of T_φ is an infinite path $r = \varphi \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \ldots$ in T_φ. A trace $\xi = \omega_0\omega_1 \ldots \in \Sigma^\omega$ is accepted by the run r if $\omega_i \models \alpha_i$ for all i.

For $\omega \in \Sigma$, we write $\varphi \xrightarrow{\omega} \psi$ if there exists $\varphi \xrightarrow{\alpha} \psi$ such that $\omega \models \alpha$. For a finite sequence $\eta = \omega_0\omega_1..\omega_k$, we write $\varphi \xrightarrow{\eta} \psi$ iff $\varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_1} \psi_2 \xrightarrow{\omega_2} \ldots \xrightarrow{\omega_k} \psi_{k+1} = \psi$. More specially, we write $\varphi \xrightarrow{\xi} \varphi$ iff ξ can be written as $\xi = \eta_0\eta_1\eta_2 \ldots$ such that η_i is a finite sequence and $\varphi \xrightarrow{\eta_i} \varphi$ for all $i \geq 0$.

Theorem 2.2 implies the following properties of $\mid S_\varphi \mid$:

Corollary 2.1 For any formula φ, it holds:

1. for all $\psi \in S_\varphi$, it holds $CF(\psi) \subseteq cl(\varphi) \cup \{\mathtt{tt}\}$,
2. $\mid S_\varphi \mid \leq 2^n + 1$ where n denotes the number of subformulas of φ.

Proof The first clause follows by a simple induction over the path from $\varphi \rightarrow \psi$. That means:

- Base case, if ψ satisfies $\alpha \wedge X\psi \in NF(\varphi)$ for some α, then from Theorem 2.2 one can directly get that $CF(\psi) \subseteq cl(\varphi) \cup \{\mathtt{tt}\}$;

- Inductive case, if there exists $\psi' \in S_\varphi$ such that $CF(\psi') \subseteq cl(\varphi) \cup \{\mathtt{tt}\}$ holds, then for each ψ satisfying $\alpha \wedge X\psi \in NF(\psi')$ it holds that $CF(\psi) \subseteq cl(\psi') \cup \{\mathtt{tt}\}$. Thus, it is true that $CF(\psi) \subseteq cl(\varphi) \cup \{\mathtt{tt}\}$.

The second clause is obtained directly from the first one. Note the constant 1 is due to the possibility of producing tt along the expansion. □

3. New satisfiability checking algorithm

In this section we introduce our new satisfiability checking algorithm explicitly. We first illustrate the main idea of our methodology by showing some running examples. Note that some concepts may be used before their formal definitions are given when we introduce the sketch of our approach.

3.1. Approach overview

We first illustrate the main idea of our algorithm. The key of our on-the-fly approach is the notion of *obligation set* (Definition 3.2). As we show below, satisfying an obligation set gives a sufficient condition for satisfying a given formula. For a given formula, the obligation set contains several possible obligations, each obligation consists of some literals that characterize a possible way of satisfying the formula. We give the flavor of this notion in terms of a few examples:

- for the formula aUb or aRb the obligation set is $\{\{b\}\}$;
- for the formula $\bigvee_{1\leq i\leq n} a_i U b_i$ the obligation set is $\{\{b_1\}, \{b_2\}, \ldots, \{b_n\}\}$;
- for the formula $\bigwedge_{1\leq i\leq n} a_i U b_i$ the obligation set is $\{\{b_1, b_2, \ldots, b_n\}\}$.

For a formula φ, if one of its obligations $O \subseteq 2^L$ is consistent, i.e. $\bigwedge_{a\in O} a \not\equiv \mathtt{ff}$, the trace $\xi = O^\omega$ is consistent, and moreover, it satisfies the corresponding formula (φ). If there is no consistent obligation, we construct a *Transition System* T_φ for φ with *on-the-fly* manner. As introduced previously, states of T_φ consist of reachable formulas, and transitions are obtained by *unrolling* the state formula according to the normal form expansion.

For technical reasons, our construction requires the input formula φ to be *tagged* before the system construction. The goal is to distinguish atoms coming from different Until subformulas of φ. For $a \in AP_\varphi$, there may be several copies of a in φ: taking $\varphi = aU(a \wedge b)$ as an example and there are two copies of a in φ. Tagging φ would distinguish these two as in φ, which results in a new formula $\varphi_t = a_1 U(a_2 \wedge b)$. However it should be mentioned that, tagging does not affect the semantics of the original formula, it only involves in changes on the syntactical level. In other words, although in $\varphi_t = a_1 U(a_2 \wedge b)$ it does not hold $a_1 = a_2$ (syntactically equivalent), it is true that $a_1 \equiv a_2$(semantically equivalent).

On the construction process of T_φ, by taking actually φ_t as the input, our algorithm is seeking such an SCC B of T_φ that literals along the edges of B is the superset of some obligation of ψ where ψ is a state in B. To present briefly we name such SCC is an *accepted SCC*. Finally if no accepted SCC can be found after the whole transition system T_φ is established, which is considered as the worst case, our algorithm returns the result of unsatisfiable for the input formula φ.

we need first to tag the subformulas such that all the literals are identified by their positions in until subformulas. We need the notion of accepting SCCs:

Definition 3.1 Let an SCC (Strongly Connected Component) B of a transition system T be a sub-system of T such that all states in B are connected by others. Moreover, we denote by $L(B)$ the set of all literals appearing in transitions between states in B. We say B is accepting if $L(B)$ is a superset of some obligation $O \in Olg(\psi)$ and $\psi \in B$.

We show that the formula is satisfiable if an SCC is found that contains a consistent obligation.
Summarizing, combining with the trivial on-the-fly checking, our approach works as follows:

1. If a *consistent obligation* is found in the processed states so far, then the formula is satisfiable;
2. If an accepting SCC is found during the generation of the transition system, then the formula is also satisfiable.
3. In the worst case, the formula is unsatisfiable after exploring whole transition system.

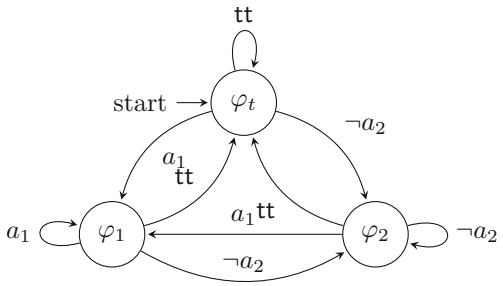

Fig. 1. The transition system of $\varphi = GFa \wedge GF\neg a$. Here $\varphi_t = GFa_1 \wedge GF\neg a_2$ is the tagged formula of φ. Also $\varphi_1 = \varphi_t \wedge F\neg a$ and $\varphi_2 = \varphi_t \wedge Fa$

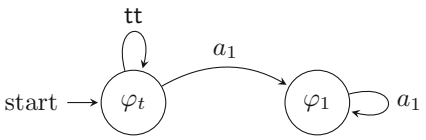

Fig. 2. The transition system of $\varphi = Ga \wedge GF\neg a$. Here $\varphi_t = Ga_1 \wedge GF\neg a_2$ is the tagged formula of φ, and $\varphi_1 = \varphi \wedge F\neg a$

Here we provide some examples to illustrate the process of our approach.

Example 1 (i) Ga is directly satisfiable without unrolling since it contains a consistent obligation $\{a\}$, according to the first process of the algorithm;

(ii) $GFa_1 \wedge GF\neg a_2$ is satisfiable, as an accepting SCC ($\varphi_1 \xrightarrow{\neg a_2} \varphi_2 \xrightarrow{a_1} \varphi_1$) can be found in its corresponding transition system (see Fig. 1).

(iii) $Ga_1 \wedge GF\neg a_2$ is unsatisfiable, as no accepting SCC can be found in its corresponding transition system (see Fig. 2).

Tagging the formula may cause potential larger size of the generated transition system, and we leave the discussion in Sect. 3.3. Now we present our approach explicitly in the following subsections.

3.2. Obligation set

The key of our on-the-fly satisfiability algorithm is the notion of obligation set, which is defined for the input formula φ:

Definition 3.2 (*Obligation set*). For a formula φ, we define its obligation set, denoted by $Olg(\varphi)$, as follows:

1. $Olg(\mathsf{tt}) = \{\emptyset\}$ and $Olg(\mathsf{ff}) = \{\{\mathsf{ff}\}\}$;
2. If φ is a literal, $Olg(\varphi) = \{\{\varphi\}\}$;
3. If $\varphi = X\psi$, $Olg(\varphi) = Olg(\psi)$;
4. If $\varphi = \psi_1 U \psi_2$ or $\psi_1 R \psi_2$, $Olg(\varphi) = Olg(\psi_2)$;
5. If $\varphi = \psi_1 \vee \psi_2$, $Olg(\varphi) = Olg(\psi_1) \cup Olg(\psi_2)$;
6. If $\varphi = \psi_1 \wedge \psi_2$, $Olg(\varphi) = \{O_1 \cup O_2 \mid O_1 \in Olg(\psi_1)$ *and* $O_2 \in Olg(\psi_2)\}$.

For $O \in Olg(\varphi)$, we refer to it as an *obligation* of φ. Moreover, we say O is a consistent obligation iff $\bigwedge a \not\equiv \mathsf{ff}$ holds, where $a \in O$.

The obligation set $Olg(\varphi)$ enumerates all obligations the given formula φ is subject to. Each obligation $O \in Olg(\varphi)$ characterizes a possible way to resolve the obligations proposed by the formula, in the sense that a formula is satisfiable if one of its obligations can be resolved accordingly. It means that the resolved obligation is a consistent obligation. Note that the particular obligation $\{\mathsf{ff}\}$ can never be resolved. The obligation $\{\mathsf{tt}\}$ is consistent.

The power of this characterization is best explained by the following theorem:

Theorem 3.1 Assume $O \in Olg(\varphi)$ is a consistent obligation. Then, $O^\omega \models \varphi$.

Proof We prove it by structural induction over φ. The basic cases when φ is tt, ff and any literal are trivial. For the induction step we consider:

- If $\varphi = X\psi$ then we have $O \in Olg(\varphi) = Olg(\psi)$. By inductive hypothesis we have $O^\omega \models \psi$. Thus, $O^\omega \models \varphi$ as well;
- If $\varphi = \varphi_1 U \varphi_2$ then we have $O \in Olg(\varphi_2)$, according to the definition of obligation set (Definition 3.2). By inductive hypothesis we have $O^\omega \models \varphi_2$. Moreover according to LTL semantics we know $O^\omega \models \varphi$ as well;
- If $\varphi = \varphi_1 R \varphi_2$ then we have $O \in Olg(\varphi_2)$, from Definition 3.2. By inductive hypothesis we have $O^\omega \models \varphi_2$. Moreover according to LTL semantics we know $O^\omega \models \varphi$;
- If $\varphi = \varphi_1 \vee \varphi_2$, we have $O \in Olg(\varphi_1)$ or $O \in Olg(\varphi_2)$. Assume $O \in Olg(\varphi_2)$ without loss of generality. By inductive hypothesis we have $O^\omega \models \varphi_2$, implying $O^\omega \models \varphi$ as well;
- If $\varphi = \varphi_1 \wedge \varphi_2$ then there exist $O_1 \in Olg(\varphi_1)$ and $O_2 \in Olg(\varphi_2)$ such that $O = O_1 \cup O_2$. Since O is consistent so both O_1 and O_2 must be consistent. By inductive hypothesis we have $O_1^\omega \models \varphi_1$ and $O_2^\omega \models \varphi_2$. Again since O is consistent we have $O^\omega \models \varphi_1 \wedge \varphi_2$. □

We illustrate the usefulness of the theorem by the following example:

Example 2 (i) Consider $G(aRb)$. It has only one obligation $\{b\}$ which is consistent. Thus, the trace $\{b\}^\omega$ satisfies $G(aRb)$.

(ii) Consider the formula $\varphi := GF(a \wedge b) \wedge F(\neg a)$: first, the obligation $\{a, b, \neg a\}$ is not consistent. Further, the normal form $NF(\varphi)$ contains $\neg a \wedge X(GF(a \wedge b)))$. Thus we can reach the formula $GF(a \wedge b)$ along $\neg a$. Moreover, $GF(a \wedge b)$ has a consistent obligation set $O = \{a, b\}$. Theorem 3.1 then provides a trace ξ with: $\xi := \{\neg a\} O^\omega \models \varphi$.

(iii) The opposite direction of Theorem 3.1 does not hold. Consider for example the formula $F(a) \wedge G(X \neg a)$. It has a single obligation $\{a, \neg a\}$ which is not consistent. However, $\{a\}\{\neg a\}^\omega$ is a satisfying trace. Consider another formula $F(a) \wedge G(\neg a)$ which has the same obligation. This formula is obviously not satisfiable.

Theorem 3.1 is indeed very useful: it returns an affirmative answer as soon as a consistent obligation is found for the current candidate. In the following sections, we exploit this notion to derive an on-the-fly algorithm for all formulas.

3.3. Tagging input formulas

First, from the discussions and definitions above, we observe that the obligation set ignores the left subformulas of until and release operators. If Theorem 3.1 does not give an affirmative answer, the left subformulas then play a role in our construction. As a preparation for the general case, we first need to *tag* the atoms in the input formula in our approach such that they can be differentiated. The example below illustrates why tagging is useful in our construction:

Example 3 Consider $\varphi := (a \vee b) U (Ga)$, in which the atom a appears twice. Without tagging, we can see there exists a transition $\varphi \xrightarrow{a} \varphi \xrightarrow{b} \varphi$ which forms a SCC B, and $L(B) = \{a, b\}$ is a superset of the obligation $\{a\}$. However, obviously, the infinite path through this SCC can not satisfy φ.

On the other side, our algorithm first tags the formula to $\varphi_t = (a_1 \vee b) U (Ga_2)$. Then the transition system for the tagged formula will be constructed. The *tagged* SCC B has label $L(B) = \{a_1, b\}$ which is not a superset of the obligation $\{a_2\}$. Thus B is not an accepting SCC, and the infinite path through SCC $\varphi \xrightarrow{a_1} \varphi \xrightarrow{b} \varphi$ can not satisfy φ.

We need some notations to formalize the tagging process. Consider a given input formula φ. For each atom a appearing in φ, we enumerate all occurrences of a by $S_a := \{a_1, a_2, \ldots, a_n\}$, provided a appears n times in φ. The easiest tagging function is the identity function, i.e., we consider all a_i *syntactically different*, but *semantically equivalent*. The complexity of our approach will depend on the number of syntactically different atoms. This tagging is inefficient: below we give an improved tagging function.

Given a formula φ we denote $U(\varphi)$ the set of until subformulas of φ and $right(\varphi)$ the set of right subformulas of φ. Then:

Definition 3.3 (*Tagging formula*). Let $a \in AP$ be an atom appearing in φ. Then, the tagging function $F_a : S_a \to 2^{U(\varphi)}$ is defined as: $\psi \in F_a(a_i)$ iff a_i appears in $right(\psi)$.

We define the *tagged formula* φ_t as the formula obtained by replacing a_i by $a_{F_a(a_i)}$ for each $a_i \in S_a$.

Thus, after tagging AP_{φ_t} will contain more atoms. Note that all these new copies are semantically equivalent to a, i.e., $a_{F_a(a_i)} \equiv a$ for all $a_{F_a(a_i)}$. Given a tagging function F_a, two copies a_i, a_j are syntactically equivalent iff $F_a(a_i) = F_a(a_j)$. More explicitly, $a_1 = a_2 \Leftrightarrow F_a(a_1) = F_a(a_2)$.

As an example, consider $\varphi = aU(a \wedge aU\neg a)$. Let $\psi_u = aU\neg a$, and $S_a = \{a_1, a_2, a_3, a_4\}$. From Definition 3.3 we know $F_a(a_1) = \emptyset$, $F_a(a_2) = F_a(a_3) = \{\varphi\}$, and $F_a(a_4) = \{\varphi, \varphi_u\}$. So the tagging function will introduce three syntactically different copies of a, and we denote φ_t by $a_1 U(a_2 \wedge a_2 U\neg a_4)$. Here even a_1, a_2, a_4 are syntactically different, they are semantically equivalent. Thus it holds for example $a_2 \wedge \neg a_4 \equiv \mathsf{ff}$. The size of subformulas may increase after tagging. According to Definition 3.3, the following lemma is obvious:

Lemma 3.1 (Tagging Cost). Let φ be the input formula and φ_t the formula obtained after tagging φ. Then, $| cl(\varphi_t) | \leq 2^m \cdot | cl(\varphi) |$, where $m = | U(\varphi) |$.

Proof It can be directly proven following Definition 3.3. □

We note that for all formula φ, it holds $\varphi \equiv \varphi_t$. This implies $SAT(\varphi)$ iff $SAT(\varphi_t)$. As our approach will work with the tagged formula φ_t, in the remaining of the paper:

- Syntactically: for a given input formula φ, all atoms are ranging over the tagged atoms appearing in φ_t, thus $AP = AP_{\varphi_t}$, $L = L_{\varphi_t}$ and $\Sigma = \Sigma_{\varphi_t} = 2^L$.
- Semantically: tagged atoms are equivalent to the original atom. Thus, the notion of consistent traces and consistent obligations are defined by taking the semantical equivalences of tagged atoms into consideration.

In the next section we present our core satisfiability checking algorithm.

3.4. On-the-fly satisfiability algorithm

First, we introduce some notations for convenience:

- We fix λ as our input formula in this section. Let T_λ be the transition system for the tagged formula λ_t.
- For all $\varphi \in S_\lambda$, we denote ST_φ the subsystem of T_λ consisting all states reachable from φ.

Now we present our main theorem:

Theorem 3.2 Let $\varphi \in S_\lambda$. Then, $SAT(\varphi)$ iff there exists a SCC B of ST_φ and a state ψ in B such that $L(B)$ is a superset of some obligation $O \in Olg(\psi)$.

The full proof is given in the next section. We sketch the proof idea first, which is best illustrated in Fig. 3. Let $\xi = \omega_0\omega_1 \ldots$ be a (consistent) trace such that $\xi \models \varphi$. Then, there is a run in ST_φ accepting ξ, i.e., we have $\varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_1} \ldots$. After some prefix ξ^n, since there are only finitely many states reachable, we will be able to partition the suffix into $\eta_1\eta_2 \ldots$ where all η_i are finite sequences, and all η_i lead from ψ to ψ itself. Such formula ψ will be referred to a *looping formula*.

Looping formulas arising from U and R operators must be treated differently. For instance $aRb \xrightarrow{b} aRb$ resolves the obligation $\{b\}$, however $aUb \xrightarrow{a} aUb$ does not. To characterize this difference, we shall memorize atoms appearing along the edges and check whether the obligation $\{b\}$ is met. Interestingly, R operators are easy to handle, but things get more involved if the same atom appears on both sides of U operators, such as aUa. Here we make use of the fact that we are working on the tagged formula, and our transition system is labeled with tagged atoms. Thus we can efficiently check whether appearing atoms correspond to those obligations for U formulas. With these notions, the theorem can be proven by the following idea: any edge label is a propositional formula that is not ff, thus any run in the transition system induces a *consistent* trace, which can be proven to satisfy the formula iff the collected atoms along the trace can produce an obligation. Thus, the formula is satisfiable if and only if we can find an SCC B such that $O \subseteq L(B)$.

The above theorem states that the satisfiability of an LTL formula λ can be checked directly on the transition system T_λ. Together with Theorem 3.1, we arrive at the following on-the-fly algorithm, which we refer to as *OFOA*(λ):

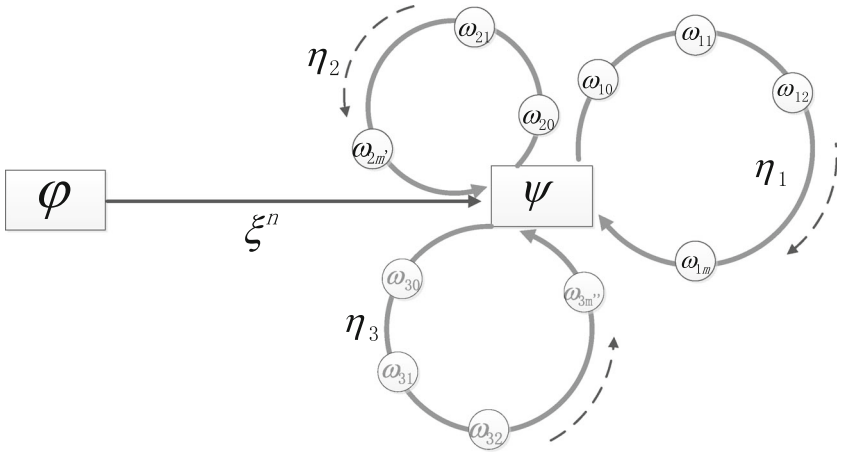

Fig. 3. A snapshot illustrating the relation $\xi \models \varphi$

1. Whenever a formula is found, we compute the obligation set. In case that it contains a consistent obligation set, we return true because of Theorem 3.1 (line 1–2 in Algorithm 1);
2. The algorithm is processed with the depth-first manner (line 3 and 7–9 in Algorithm 1)
3. If a SCC B is reached, $\varphi \in B$, and $L(B)$ is a superset of some obligation set $O \in Olg(\varphi)$, then we return true (line 4–6 in Algorithm 1);
4. If all SCCs are explored, and all do not have the property in step 3, we return false (line 10 in Algorithm 1).

Algorithm 1: The pseudo-code of OFOA algorithm.

```
   Input: The tagged formula λ_t;
   Output: SAT or UNSAT.
1  if λ_t has a consistent obligation then
2      return SAT;
3  for each α ∧ Xφ ∈ NF(λ_t) do
4      if φ is visited then
5          if there is an accepting SCC containing φ then
6              return SAT;
7      else
8          if OFOA(φ) returns SAT then
9              return SAT;
10 return UNSAT;
```

We discuss briefly the complexity of the proposed algorithm. First, we remark that the worst case scenario happens if all extended formulas do not contain any consistent sets, which happens for instance for the formula $GF(a) \wedge GF(\neg a)$. Given the input formula λ, we first construct λ_t. By Lemma 3.1, we have $\mid cl(\lambda_t) \mid \leq 2^m \cdot \mid cl(\lambda) \mid$, where $m = \mid U(\lambda) \mid$ is the number of until subformulas of λ. By Corollary 2.1, the number of states is bounded by $\mid S_\varphi \mid \leq 2^n + 1$, where $n = 2^m \cdot \mid cl(\lambda) \mid$. In addition, for each reachable state φ, we compute the obligation $Olg(\varphi)$, which is exponential in the number of conjunctions in φ, but linear in other operators.

3.5. Proof of theorem 3.2

This section is devoted to the proof of Theorem 3.2. We organize the proof as follows. We first introduce the notion of looping formulas and discuss their properties. We then continue with the soundness and completeness proofs of the theorem.

Assumption. Throughout the section, we have the following assumptions:

- λ denotes the fixed input formula, T_λ is the transition system for λ.
- all traces are over Σ^ω where $\Sigma \subseteq 2^{L_{\lambda_t}}$, i.e., the set of consistent literals.
- all formulas appearing in this section are taken from the set of states S_λ, i.e., $\varphi, \psi ... \in S_\lambda$. Thus, all formulas in this section will be ranging over tagged atoms appearing in λ_t.

3.5.1. Looping formulas and their properties

We start with a simple theorem about the relation between satisfiability and the transitions:

Theorem 3.3 Let $\xi \in \Sigma^\omega$ be a trace and φ be a formula. Then, for all $n \geq 1$, there exists ψ such that $\xi \models \varphi \Leftrightarrow \varphi \xrightarrow{\xi^n} \psi \wedge \xi_n \models \psi$.

Proof Let $\xi = \omega_0\omega_1 \ldots$. We prove the theorem by induction on n.

- For the base case we let $n = 1$. Then: $\xi \models \varphi \Leftrightarrow \xi \models \bigvee NF(\varphi) \Leftrightarrow \exists \alpha \wedge X\psi \in NF(\varphi) \cdot \xi \models (\alpha \wedge X\psi)$ $\Leftrightarrow \exists \alpha \wedge X\psi \in NF(\varphi) \cdot \xi_1 \models \psi \wedge \omega_0 \models \alpha \Leftrightarrow \exists \psi \cdot \varphi \xrightarrow{\xi^1} \psi \wedge \xi_1 \models \psi$.
- For the induction step, we assume the lemma holds for all $n = 1, 2, \ldots, k$ and prove that it holds for $n = k+1$ as well. Applying the induction hypothesis on k, we have: $\xi \models \varphi \Leftrightarrow \varphi \xrightarrow{\xi^k} \psi \wedge \xi_k \models \psi$ holds. Further, for $\xi_k \models \psi$ we apply the induction hypothesis with respect to the base case and obtain $\xi_k \models \psi \Leftrightarrow \psi \xrightarrow{\xi_k^1} \psi' \wedge \xi_{k+1} \models \psi'$, so we can conclude that $\xi \models \varphi \Leftrightarrow \varphi \xrightarrow{\xi^{k+1}} \psi' \wedge \xi_{k+1} \models \psi'$. The proof is done. □

Essentially, $\xi \models \varphi$ is equivalent to the fact that we can reach a formula ψ along the prefix ξ^n such that the suffix ξ_n satisfies ψ. Thus, if $\xi \models \varphi$ holds, then ξ will be accepted by a run in ST_φ.

Now we introduce the notion of looping formulas:

Definition 3.4 (*Looping formula*). We say φ is a *looping formula* iff there exists a trace $\xi \in \Sigma^\omega$ which can be written as an infinite sequence $\xi = \eta_0\eta_1\eta_2 \ldots$ such that η_i is a finite sequence and $\varphi \xrightarrow{\eta_i} \varphi$ for all $i \geq 0$. We write $\varphi \xrightarrow{\xi} \varphi$ in this case, and say φ is a looping formula with respect to ξ.

For convenience in the rest of the paper when we say φ is a looping formula, it really means φ is a looping formula with respect to some infinite trace ξ. The following corollary is a direct consequence of Theorem 3.3 and the fact that we have only finitely many formulas in S_φ:

Corollary 3.1 If $\xi \models \varphi$, then there exists $n \geq 1$ and ψ such that $\varphi \xrightarrow{\xi^n} \psi$ and $\xi_n \models \psi$ and $\psi \xrightarrow{\xi_n} \psi$.

Proof From Theorem 3.3 $\xi \models \varphi$ implies there is an infinite expansion path $\sigma = \varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_1} \psi_2 \xrightarrow{\omega_2} \ldots$ such that $\xi_i \models \psi_i$ for all $i \geq 1$. Since we have only finitely states, there must exist a ψ reachable from φ such that it appears infinitely often along this path. Obviously, this formula ψ is a looping formula as required. □

This corollary gives the hint that after a finite prefix we can focus on the satisfiability of looping formulas. Now we give a lemma stating a nice property for the release operator:

Lemma 3.2 If $\varphi = \varphi_1 R \varphi_2$ and $\varphi \xrightarrow{\xi} \varphi$, then $\xi \models \varphi$.

Proof Since $\varphi \xrightarrow{\xi} \varphi$, so we have $\exists\, n \cdot \varphi \xrightarrow{\xi^n} \varphi \wedge \varphi \xrightarrow{\xi_n} \varphi$. Let $\eta_i = \omega_i\omega_{i+1} \ldots \omega_n (0 \leq i \leq n)$. Here all expansions for the formula φ along the path must be from the right subformula φ_2 of φ, i.e., $\alpha \wedge X\psi \in NF(\varphi_2 \wedge X\varphi)$. Since $\varphi \xrightarrow{\xi^n} \varphi$, we have that $\forall\, 0 \leq i \leq n \cdot \varphi_2 \xrightarrow{\eta_i} \mathrm{tt}$, which implies $\xi_j \models \varphi_2$ for all $0 \leq j \leq n$. Inductively for $\varphi \xrightarrow{\xi_n} \varphi$ we can get the same property. So $\forall\, j \geq 0$ we have $\xi \models \varphi_2$, implying $\xi \models \varphi$. □

Lemma 3.2 indicates that if φ is a Release formula and φ is a looping formula with respect to ξ, then $\xi \models \varphi$. There is another simple but useful property that looping formulas have, which we list as below:

Lemma 3.3 If φ is a looping formula, then there must be an Until or Release formula ψ in $CF(\varphi)$.

Proof This lemma is directly proven according to LTL semantics, as for an arbitrary trace ξ it is impossible that $\varphi \xrightarrow{\xi} \varphi$ holds without an Until or Release formula in $CF(\varphi)$. □

Finally, we shall introduce an order on formulas to identify structural properties of looping formulas. We propose the following order for formulas:

Definition 3.5 (*Poset on formulas*). For formulas φ, ψ, we write $\varphi \preceq \psi$ iff $\varphi \in cl(\psi)$.

The order $\preceq$ is obviously a partial order. For a looping formula φ, the set $CF(\varphi)$ possess at least one minimal element (w.r.t. the order $\preceq$). Below we prove that all minimal elements of the set expand either to $\mathtt{tt}$ or themselves.

Lemma 3.4 If $\varphi \xrightarrow{\eta} \varphi$ then for all minimal elements $\psi \in CF(\varphi)$ we have $\psi \xrightarrow{\eta} \mathtt{tt}$ or $\psi \xrightarrow{\eta} \psi$.

Proof Let ψ be a minimal element in $CF(\varphi)$. Since $\varphi \xrightarrow{\eta} \varphi$ and $\psi \in CF(\varphi)$, there must exist ψ' such that $\psi \xrightarrow{\eta} \psi'$ and $CF(\psi') \subseteq CF(\varphi)$ or $\psi' = \mathtt{tt}$. If $\psi' \neq \mathtt{tt}$, then according to Theorem 2.2 we know $CF(\psi') \subseteq cl(\psi)$. However, $cl(\psi) \cap CF(\varphi) = \{\psi\}$ because of the minimality of ψ. Thus $\psi' = \psi$. □

3.5.2. Soundness proof of theorem 3.2

We first introduce the relation $\models_f$:

Definition 3.6 Let $\eta = \omega_0\omega_1 \ldots \omega_n$ $(n \geq 0)$. Then, we say the finite sequence η satisfies the formula φ, denoted by $\eta \models_f \varphi$, iff $\varphi \xrightarrow{\eta} \varphi'$ for some φ' and there exists $O \in \mathit{Olg}(\varphi)$ such that $O \subseteq \eta$. Here $O \subseteq \eta$ is an abbreviation for $O \subseteq \cup_{i=0}^{n} \omega_i$.

Please note that the relation is defined by checking syntactic inclusion. Thus, assuming the input formula is aUa, which is $a_1 U a_2$ after tagging. According to the above definition, $\{a_1\} \not\models_f a_2$. The reader shall bear this in mind in the remaining of this section.

Below we present some simple properties of the relation $\models_f$ which is useful later:

Lemma 3.5 1. Assume $\eta \models_f \varphi$, then $\eta \models_f \varphi \vee \psi$.
2. Assume $\eta \models_f \varphi$ and $\eta \models_f \psi$, then $\eta \models_f \varphi \wedge \psi$.

Proof Let $\eta = \omega_0\omega_1 \ldots \omega_n$. We consider the first case: $\eta \models_f \varphi$ implies that there exists $O \in \mathit{Olg}(\varphi)$ such that $O \subseteq \eta$. Since $O \in \mathit{Olg}(\varphi) \subseteq \mathit{Olg}(\varphi \vee \psi)$, we have $\eta \models_f \varphi \vee \psi$. For the second case: $\eta \models_f \varphi$ implies that there exists $O_1 \in \mathit{Olg}(\varphi)$ such that $O_1 \subseteq \eta$. Similarly, $\eta \models_f \psi$ implies that there exists $O_2 \in \mathit{Olg}(\psi)$ such that $O_2 \subseteq \eta$. Since we have $O_1 \cup O_2 \in \mathit{Olg}(\varphi \wedge \psi)$ and $O_1 \cup O_2 \subseteq \eta$, we have $\eta \models_f \varphi \vee \psi$. □

The following lemma corresponds to Lemma 3.2 for until formulas with respect to the finite satisfaction relation:

Lemma 3.6 Let $\varphi = \varphi_1 U \varphi_2$ and $\varphi \xrightarrow{\eta} \varphi$. Then, $\eta \not\models_f \varphi$.

Proof Let $\eta = \omega_0\omega_1 \ldots \omega_n$ and we rewrite $\varphi \xrightarrow{\eta} \varphi$ as $\varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_2} \ldots \xrightarrow{\omega_n} \varphi$. Note it is apparent that $\varphi \xrightarrow{\omega_0} (\varphi \wedge \varphi_1')$ with $\varphi_1 \xrightarrow{\omega_0} \varphi_1'$. Thus, by induction one can show that along the path $\varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_1} \psi_2 \ldots$ it holds $\varphi \in CF(\psi_i)$ for all i. Since the transition for conjunctive formula ψ_i is obtained by combining transitions for each $\psi' \in CF(\psi_i)$, the transition for the subformula $\varphi \in CF(\psi_i)$ must be from left subformula φ_1, i.e., $\alpha \wedge X\psi \in \mathit{NF}(\varphi_1 \wedge X\varphi)$. As a result, for each i, the label ω_i must be a superset of $CF(\alpha_i)$, and $CF(\alpha_i)$ contains the literals from φ_1. Moreover, the tagging function has been used to classify the atoms. According to Definition 3.3 the atoms in φ_1 and φ_2 are never tagged the same—this is because the atoms in φ_2 will tag φ while those in φ_1 will not. So φ_1 and φ_2 still don't have common atoms, thus $CF(\alpha_i)$ contains no obligation literals from φ_2;

According to Definition 3.2, the obligation literals of φ are all from those of φ_2. Thus, from the definition of $\models_f$ (Definition 3.6), we know $\eta \not\models_f \varphi$. □

The following lemma says that if there exists a partitioning $\xi = \eta_1\eta_2 \ldots$ that makes φ expanding to itself by each η_i and $\eta_i \models_f \varphi$ holds, then $\xi \models \varphi$.

Lemma 3.7 Given a looping formula φ and a trace ξ, let $\xi = \eta_1\eta_2\ldots$. If $\forall\, i \geq 1 \cdot \varphi \xrightarrow{\eta_i} \varphi \wedge \eta_i \models_f \varphi$, then $\xi \models \varphi$.

Proof We enumerate the set $CF(\varphi) = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, and we shall prove $\xi \models \bigwedge CF(\varphi)$. Let S_0 denote the minimal elements of $CF(\varphi)$ with respect to the partial order $\preceq$. Moreover, we define $S_{i+1} = S_i \cup \{\psi' \in CF(\varphi) \mid \exists\, \psi \in S_i.\psi \preceq \psi'\}$. Obviously, there is a finite index k such that $S_k = CF(\varphi)$. We proceed with induction over the index:

1. Basic step: Let $\psi \in S_0$. By Lemma 3.4 if $\varphi \xrightarrow{\eta_i} \varphi$, then $\psi \xrightarrow{\eta_i} \mathtt{tt}$ or $\psi \xrightarrow{\eta_i} \psi$. If $\exists\, \eta_i \cdot \psi \xrightarrow{\eta_i} \mathtt{tt}$ holds, $\xi \models \psi$ follows from Theorem 3.3. Otherwise we have $\psi \xrightarrow{\eta_i} \psi$ for all $i \geq 1$. According to LTL semantics ψ must be either until or release formula. Applying Lemma 3.6 we know that ψ cannot be an Until formula, and thus be a Release formula from Lemma 3.3. Then Lemma 3.2 implies that $\xi \models \psi$, and therefore $\xi \models \bigwedge S_0$.
2. For induction step we assume $\xi \models \bigwedge S_k$. Consider arbitrary η_i: let $\psi \in S_{k+1} \setminus S_k$ and assume $\psi \xrightarrow{\eta_i} \psi'$. Since $\varphi \xrightarrow{\eta_i} \varphi$, we have $CF(\psi') \subseteq CF(\varphi)$. By the construction of the set S_i, $CF(\psi')$ does not contain any other elements in S_{k+1}, thus we have $CF(\psi') \subseteq \{\psi\} \cup S_k$. First we assume $CF(\psi') \subseteq S_k$. Then from the induction hypothesis we know $\eta_{i+1}\eta_{i+2}\ldots \models \psi'$ so $\eta_i\eta_{i+1}\ldots \models \psi$ (From Theorem 3.3), thus $\xi \models \psi$. Now consider the case $\psi \in CF(\psi')$:$\psi \xrightarrow{\eta_i} \psi$ implies that ψ must be a Release formula. (From Lemma 3.3 ψ is either an Until or Release formula, and it cannot be an Until formula due to Lemma 3.6.) Then Lemma 3.2 implies again that $\xi \models \psi$, and therefore $\xi \models \bigwedge S_{k+1}$. □

Now we are ready to prove the soundness part of the theorem:

Lemma 3.8 (Soundness) Let $\varphi \in S_\lambda$. Assume that there exists an SCC B of ST_φ such that $\psi \in B$ and $L(B)$ is a superset of some obligation set $O \in Olg(\psi)$. Then, $SAT(\varphi)$.

Proof As B is a SCC, we have a path $\delta := \psi(\psi_1) \xrightarrow{\omega_1} \psi_2 \xrightarrow{\omega_2} \ldots \xrightarrow{\omega_{k-1}} \psi$ such that η visits all states in B and all transitions between states in B ($\eta = \omega_1\omega_2\ldots\omega_{k-1}$). Since all states in ST_φ are reachable from φ, there must exist a finite sequences η_0 such that $\varphi \xrightarrow{\eta_0} \psi$. We construct $\xi := \eta_0\eta^\omega$. By assumption there exists $O \in Olg(\psi)$ such that $O \subseteq L(B) = \cup_{i=1}^{k-1}\omega_i$. So according to the definition of $\models_f$ we have $\eta \models_f \psi$. Thus from Lemma 3.7 we have $\xi \models \varphi$. So φ is satisfiable. □

3.5.3. Completeness proof of theorem 3.2

Lemma 3.9 $\xi \models \varphi \Rightarrow \exists\, n \cdot \xi^n \models_f \varphi$.

Proof We prove it by structural induction over the formula φ. For the base case assume φ is tt or a literal, $\xi^1 \models_f \varphi$ by definition. Moreover, φ can not be ff. Now we consider the induction step:

- If $\varphi = X\psi$, then $\xi \models \varphi \Rightarrow \xi_1 \models \psi$. By induction hypothesis we know $\exists\, n \cdot {\xi_1}^n \models_f \psi$ holds, so $\xi^{n+1} \models_f \varphi$ holds.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $\xi \models \varphi_1 \wedge \xi \models \varphi_2$. By induction hypothesis we have $\exists\, n_1 \cdot \xi^{n_1} \models_f \varphi_1$ and $\exists\, n_2 \cdot \xi^{n_2} \models_f \varphi_2$ hold. Observe that $\xi^n \models_f \varphi$ implies $\xi^m \models_f \varphi$ for all $m \geq n$. Now from Lemma 3.5 we have that $\xi^n \models_f \varphi$ with $n := \max(n_1, n_2)$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $\xi \models \varphi_1 \vee \xi \models \varphi_2$. By induction hypothesis we have $\exists\, n_1 \cdot \xi^{n_1} \models_f \varphi_1$ or $\exists\, n_2 \cdot \xi^{n_2} \models_f \varphi_2$ holds. Without loss of generality, assume $\exists\, n_1 \cdot \xi^{n_1} \models_f \varphi_1$. Lemma 3.5 implies then $\xi^{n_1} \models_f \varphi_1 \vee \varphi_2$.
- If $\varphi = \varphi_1 U \varphi_2$, $\xi \models \varphi_1 U \varphi_2$ implies that there exists $i \geq 0$ such that $\xi_i \models \varphi_2$. By induction hypothesis we have $\exists\, n \cdot {\xi_i}^n \models_f \varphi_2$ hold, thus there exists an obligation $O \in Olg(\varphi_2)$ such that $O \subseteq {\xi_i}^n$. This implies $O \subseteq \xi^{i+n}$, thus $\xi^{i+n} \models_f \varphi$ holds.
- If $\varphi = \varphi_1 R \varphi_2$, we observe first that $\xi \models \varphi_2$ must hold. By induction hypothesis we know $\exists\, n \cdot \xi^n \models_f \varphi_2$. According to Definition 3.6 we have that $\xi^n \models_f \varphi$ holds as well. □

Lemma 3.10 Let ξ be an infinite trace, φ be a looping formula, and assume $\varphi \xrightarrow{\xi} \varphi$ and $\xi \models \varphi$ hold. Then there exists a partitioning $\xi = \eta_1\eta_2\ldots$ and $\forall\, i \geq 1 \cdot \varphi \xrightarrow{\eta_i} \varphi \wedge \eta_i \models_f \varphi$ holds.

Proof Assume $\varphi \xrightarrow{\xi} \varphi \wedge \xi \models \varphi$. We first prove that there exist n such that $\varphi \xrightarrow{\xi^n} \varphi \wedge \xi^n \models_f \varphi \wedge (\varphi \xrightarrow{\xi_n} \varphi \wedge \xi_n \models \varphi)$. From Lemma 3.9 $\xi \models \varphi$ implies that there exists k such that $\xi^k \models_f \varphi$. Since φ is a looping formula with respect

to ξ, we can find the $n \geq k$ such that $\varphi \xrightarrow{\xi^n} \varphi$ and $\varphi \xrightarrow{\xi_n} \varphi$ hold. For $n \geq k$, $\xi^n \models_f \varphi$ holds as well. Now we apply Theorem 3.3: $\xi \models \varphi$ implies that $\varphi \xrightarrow{\xi^n} \varphi$ and $\xi_n \models \varphi$.

Since $\varphi \xrightarrow{\xi_n} \varphi \wedge \xi_n \models \varphi$, applying the arguments above inductively yields the lemma. □

The above lemma states that if $\varphi \xrightarrow{\xi} \varphi$ as well as $\xi \models \varphi$, we can find a partitioning $\eta_1\eta_2\ldots$ that makes φ expend to itself by each η_i and $\eta_i \models_f \varphi$ holds. Combining Lemma 3.9, Lemma 3.10 and Corollary 3.1, we have the completeness of our central theorem:

Lemma 3.11 (Completeness) Let $\varphi \in S_\lambda$. Then, $SAT(\varphi)$ implies that there exists a SCC B of ST_φ such that $\psi \in B$ and $L(B)$ is a superset of some obligation set $O \in Olg(\psi)$.

Proof Since $SAT(\varphi)$, let ξ be such that $\xi \models \varphi$. Then by Corollary 3.1 there exists $n \geq 0$ and $\psi \in S_\varphi$ such that $\varphi \xrightarrow{\xi^n} \psi$, $\psi \xrightarrow{\xi_n} \psi$ and $\xi_n \models \psi$. Moreover, by Lemma 3.10, there exists a partition $\xi_n = \eta_1\eta_2\ldots$ such that for every finite sequence η_i we have $\psi \xrightarrow{\eta_i} \psi$ as well as $\eta_i \models_f \psi$. As the state ψ is visited infinitely often, there must be a SCC B such that $\psi \in B$. According to the definition of $\models_f$ we know that there exists $O \in Olg(\psi)$ such that $O \subseteq \eta_1$. Obviously, $\psi \xrightarrow{\eta_1} \psi$ is contained in some SCC B, thus $\eta \subseteq L(B)$, implying $O \subseteq L(B)$. □

We have proposed our new satisfiability checking methodology for LTL formulas. We also prove the correctness above for the approach. And in the next section we talk about the evaluations among our techniques as well as other existing solutions.

4. Experiments

In this section we detail our experimental methodology and results. To test the efficiency of the proposed algorithm, we have implemented it in a tool called *Aalta* [1]. We call Theorem 3.1 as the *obligation acceleration* technique (OA, for short). Similarly, we refer to the technique that underlies Theorem 3.2 as the *on-the-fly* technique (OF, for short). In the tool we have the following two configurations: (i) OF: On-the-fly checking without OA, (ii) OFOA: On-the-fly checking with obligation acceleration. By default *Aalta* is implemented with the OFOA configuration.

We run all our experiments on the SUG@R cluster in Rice University[2]. SUG@R is an Intel Xeon compute cluster. It contains 134 SunFire x4150 nodes from Sun Microsystems. Each node has two quad-core Intel Xeon processors running at 2.83GHz, yielding a system-wide total of 1064 processor cores. The OS is Red Hat Enterprise 5 Linux, 2.6.18 kernel. The time cost is measured using the Unix time command. All time measurements are "end-to-end": That means we measure the time starting from formula input to the satisfiability-checking result (SAT or UNSAT). If timeout occurs before the result is given, then we count the timeout as the checking time in the current test.

We set up our experiment in this paper by following related previous work. In [LZP+13] we follow the strategies in [RV11], which aims to focus the comparison among automata-based LTL satisfiability solvers. Some details are missing in [LZP+13] and we will discuss them explicitly in this paper. We introduce them in the following.

4.1. Tool implementation

The architecture of tool *Aalta* is represented in Fig. 4, *Aalta* provides two interfaces, *input* and *output*. And the core modules consist of *Parser*, *NNF Converter* and *LTL Checker*.

Input: The *input* of *Aalta* is a string. Also *Aalta* recognizes propositional operators such as $\wedge$ and temporal operators, for instance, X, R. Special operators like F and G can be recognized by *Aalta*, too. $G\varphi = ffR\varphi$, $F\varphi = ttU\varphi$. Table 1 shows the mapping relationship between formula operators and representations in *Aalta*.

Output: *Aalta* shows the satisfiability checking result of input formula. *sat* means the input formula is satisfiable, *unsat* is not. Users can choose configuration parameters such as "-" to get an trace satisfying the input formula which is *sat*. To present infinite traces, "(s)" is used by *Aalta* to denote the infinite occurrence of string s. For instance, an output like "a(xy)" is an infinite trace "$a(xy)^\omega$".

[1] www.lab205.org/aalta
[2] http://www.rcsg.rice.edu/sugar/

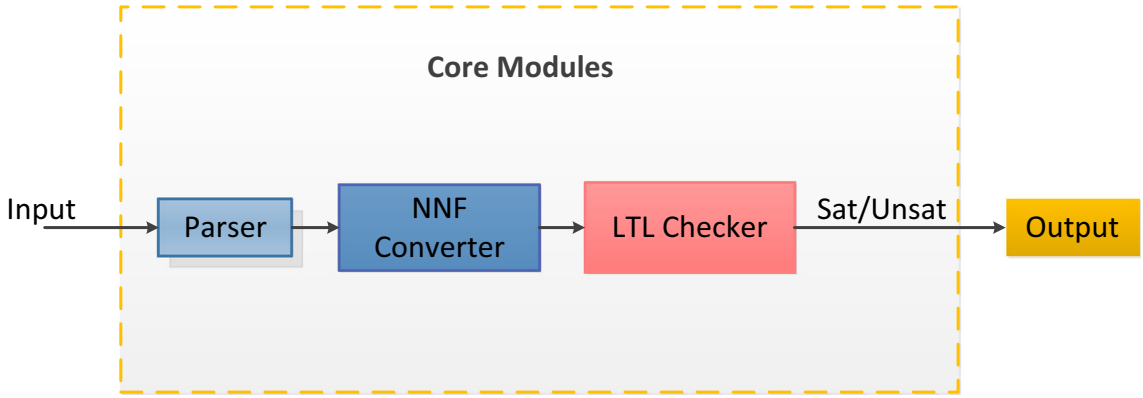

Fig. 4. The architecture of *Aalta*

Table 1. The operators representations in *Aalta*

operators	symbols
$\neg$	!,~
$\wedge$	&, &&
$\vee$	|, ||
X	X
U	U
R	R,V
G	G,[]
F	F,⟨⟩
$\rightarrow$	$\rightarrow$
$\leftrightarrow$	$\leftrightarrow$
tt	$true$, $TRUE$
ff	$false$, $FALSE$

Parser: This part is to recognize the input string of *Aalta*, mainly cope with operators shown in Table 1 to normalize the input formula. Formula simplification and Tagging are done in this process after which we can get an abstract semantic tree.

NNF Converter: Our algorithm requires the formula under check to be in NNF, so we implemented the *NNF Converter* to automatically generate NNF form of each input formula according to Definition 2.3. For each NNF formula, we will compute the obligation set.

LTL Checker: In this part, we need to check the consistency of obligation set in every new generated state. If a *consistent obligation* is found, then return *sat* according to Algorithm1, Line 1–2. If there are no *consistent obligation* and the state has been visited before, we will try to find whether an accepting SCC containing φ has occurred. Return *unsat* if there are no accepting SCC, else return *sat*. For each state generated by *NNF Converter*, *LTL Checker* will be invoked which means our checking process will run *on-the-fly*.

4.2. Comparing with automata-based solvers

In this section we follow the strategy introduced in [RV11].

4.2.1. Testing tools

We compare the performance of *Aalta* with two other LTL satisfiability solvers: PANDA+CadenceSMV [RV11] and SPOT [DLP04]. SPOT is considered as the best explicit LTL-to-Büchi translator [RV07, RV10]. Its most recent version (1.0.2 at the time of writing the present paper) has an integrated emptiness checking implementation (with "-e" flag) and it is considerably improved since the benchmarking in [RV07, RV10]. That benchmarking showed the superiority of CadenceSMV for LTL satisfiability checking, and this has been further improved in PANDA+CadenceSMV [RV11]. Thus, we benchmarked all three tools. Note that PANDA consists of 30 different symbolic encodings, we run all these encodings in parallel and choose the best result among them.

4.2.2. Benchmarks

We use here the benchmarks from [RV07, RV10, RV11]. These include random, pattern and counter formulas. We tested over 60,000 random formulas and all eight kinds of pattern (lengths varying from 1 to 1000) and four kinds of counter formulas (lengths varying from 1 to 20). These benchmarks are suitable for testing satisfiability of *large* formulas. As described in [RV07, RV10], random formulas are created by randomly choosing N, L and P, where N represents the number of variables in the formula, L is the length of the generated formula and P is the probability of occurrence of the Until operator. The pattern formulas are those with special format and may be used quite often in practice. The counter formulas are a kind of patterns whose satisfiability checking requires generation of the whole systems (automata). For more details on these formulas we refer to [RV07, RV10]. There are already off-the-shelf perl scripts that can generate these formulas[3]. Our benchmarks are obtained directly from those executable scripts.

Typical temporal assertions are, however, quite small in practice [DAC98]. What makes the LTL satisfiability problem hard is the fact that we need to check *large conjunctions of small temporal formulas*, as we need to check that the conjunction of all input assertions is also satisfiable. We introduce here a novel class of challenging LTL benchmarks, which are *random conjunctions* of specification patterns from [DAC98]. Formally, a random conjunction formula $RC(n)$ has the form: $RC(n) = \bigwedge_{1 \leq i \leq n} P_i(v_1, v_2, \ldots, v_k)$, where n is the number of conjuncts elements and $P_i (1 \leq i \leq m)$ is a randomly chosen property pattern formula used frequently in practice [DAC98]. The propositions $\{v_1, v_2, \ldots, v_k\}$ used in these formulas are also chosen randomly. More precisely, we generate the class of random conjunction formulas in the following way:

1. We extract all pattern formulas[4].
2. For a formula in $RC(n)$, we conjoin n pattern formulas selected randomly. In each pattern formula, we instantiate the variables as random literals (positive or negative) over a set of six atomic propositions.
3. In our experiments we generated 500 random formulas for each n.

In this part of the experiment, each test is run on a single core of SUG@AR with a timeout of 10 minutes for each formula. When the timeout occurs in a test, we count it the maximum cost of 600 seconds (10 minutes). To test *Aalta*'s correctness, we assume that the results from PANDA+CadenceSMV and SPOT are correct and we compare the results with *Aalta*'s. *Aalta* successfully passes all the tests.

4.2.3. Experimental results

The experimental results are reported in this section. Generally speaking, our results demonstrate that *Aalta* outperforms both SPOT and PANDA+CadenceSMV. The experimental results show that, *Aalta* performs best for random and random conjunction formulas, and also better for 6 out of 8 pattern formulas. *Note that when we say Aalta performs best in this section, we actually means Aalta performs best among the three testing tools.* However, the performance of *Aalta* on counter formulas is slightly worse than the other two solvers. In the following we explicit these aspects respectively.

Aalta performs best for random formulas. We first compare the three tools on random benchmarks. We use here three atomic propositions and formula length of up to 200. In total, we tested 20,000 random formulas. Figure 5 shows performance results for the three tools, where for each length we report average running time on 500 formulas.

[3] http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/benchmarking_scripts.html
[4] http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml

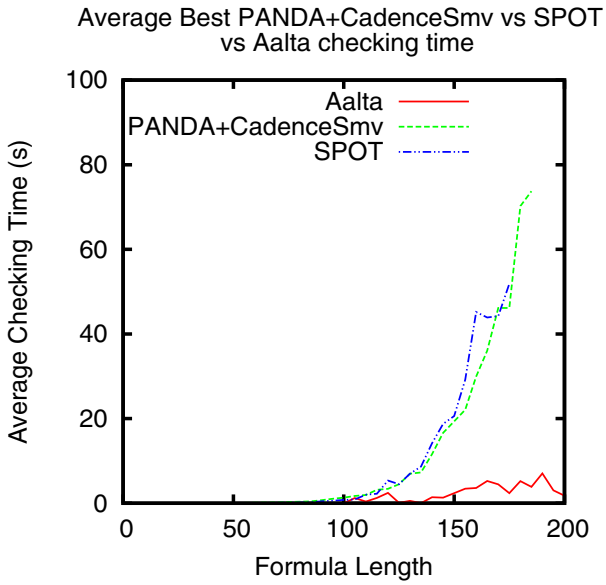

Fig. 5. Experimental results for random formulas with 3 variables

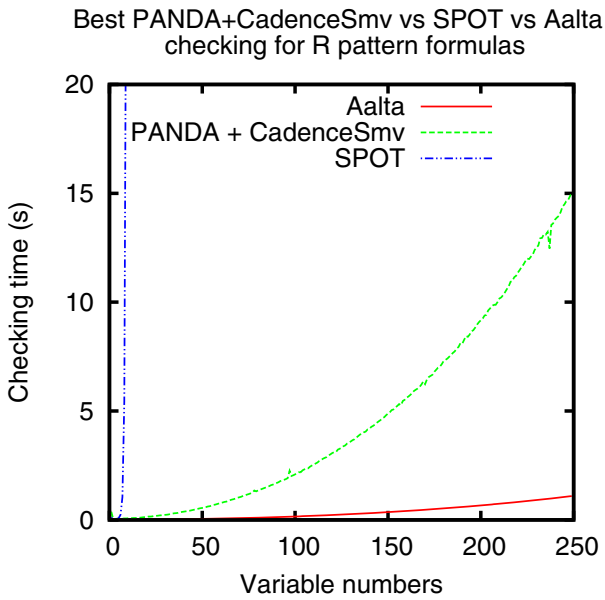

Fig. 6. Experimental results for pattern $R(n) = \bigwedge_{i=1}^{n}(GFp_i \vee FGp_{i+1})$

We can see that *Aalta* outperforms the other tools on random formulas. In fact, *Aalta* significantly outperforms the other tools; for 60% of the formulas, *Aalta* returns in a few milliseconds, while SPOT and PANDA+CadenceSMV takes tens of seconds. In total, *Aalta* completes checking all 20,000 formulas in one hour, while neither SPOT nor PANDA+CadenceSMV are able to complete in 24 hours. The superiority of *Aalta* stems from the fact that 95% of the test formulas turn out to be satisfiable; furthermore, 80% of them are checked by using the *obligation acceleration* technique. This indicates the power of *obligation acceleration* on checking satisfiable formulas. Indeed, on unsatisfiable formulas PANDA+CadenceSMV is faster than *Aalta*, performing nearly twice as faster than *Aalta*. Overall, however, *Aalta*'s heuristics for quick satisfiability testing do pay off.

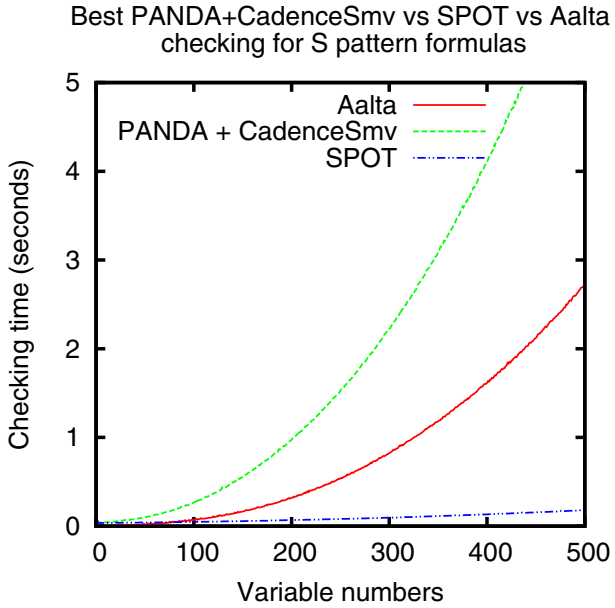

Fig. 7. Experimental results for pattern $S(n) = \bigvee_{1 \leq i \leq n} Gp_i$

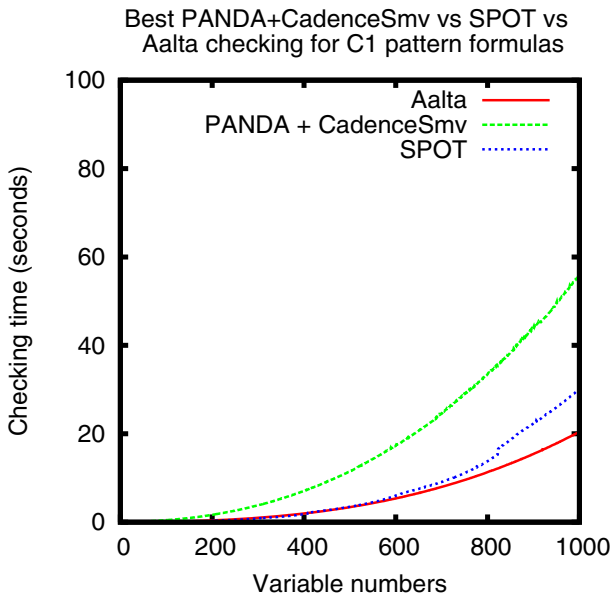

Fig. 8. Experimental results for pattern $C1(n) = \bigvee_{1 \leq i \leq n} GFp_i$

Aalta performs best for most of the pattern formulas. Our experiments show that *Aalta* performs best for all pattern formulas except the S-pattern formula, where SPOT performs best among three tools. For example, Fig. 6 displays the comparing results for the R-pattern formulas, where SPOT scales exponentially with formula length, while PANDA+CadenceSMV is quicker, and *Aalta* performs the best.

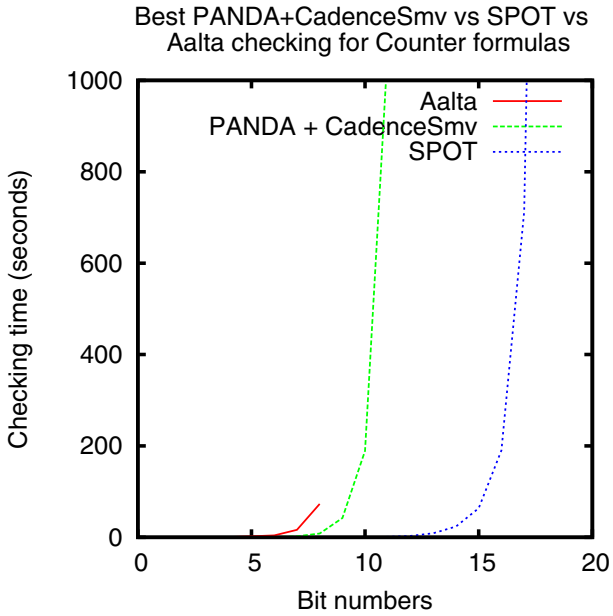

Fig. 9. Experimental results for counter formulas

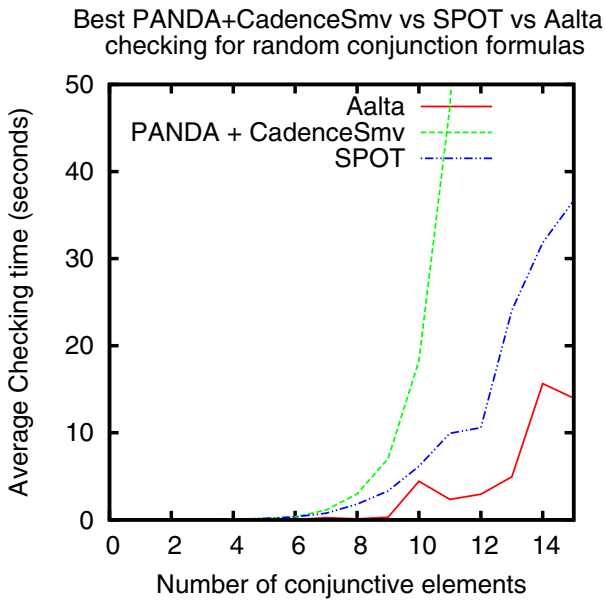

Fig. 10. Experimental results for random conjunctive formulas

Here it is clear that SPOT pays the price for not performing the automaton non-emptiness test on-the-fly, as the automata scale exponentially. In fact, even *Aalta* scales exponentially for R-pattern formulas without the *obligation acceleration* technique. There are other patterns such as C2 ($C2(n) = \bigwedge_{1\leq i\leq n} GFp_i$), E ($E(n) = \bigwedge_{1\leq i\leq n} Fp_i$), Q ($Q(n) = \bigwedge_{1\leq i\leq n}(Fp_i \vee Gp_{i+1})$), U ($U(n) = (\ldots(p_1 Up_2)\ldots)Up_n$), and U2 ($U2(n) = p_1 U(p_2 U(\ldots(p_{n-1} Up_n)))$), for which the results from *Aalta* perform best among tested tools, and especially exponentially faster than those from SPOT.

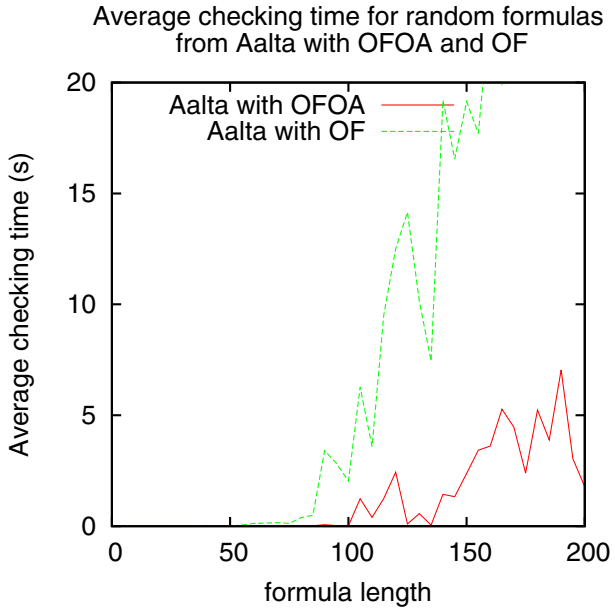

Fig. 11. Experimental results for 3-variable random formulas from *Aalta* with OFOA and OF

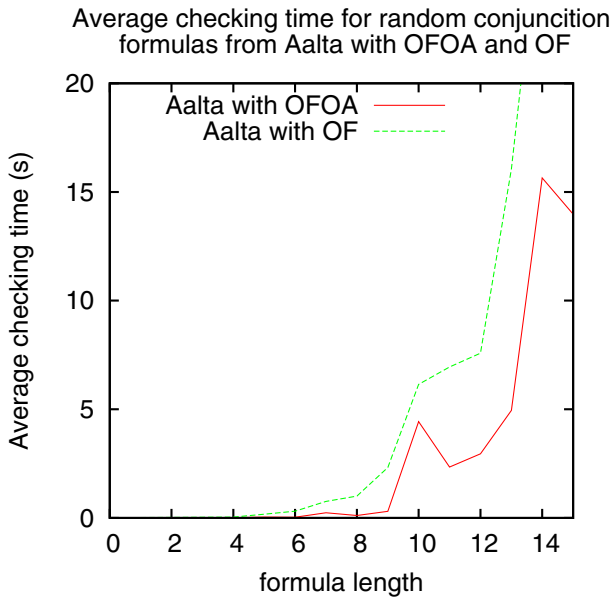

Fig. 12. Experimental results for random conjunction formulas from *Aalta* with OFOA and OF

There are two kinds of pattern formulas for which SPOT is able to check in linear time with the size. That are, S and C1 patterns (shown in Figs. 7, 8). For S-pattern formulas, all three tools scale polynomially, since automata size scales linearly. Note here, SPOT performs better than *Aalta* on S pattern, which is because that automaton construction in SPOT is faster due to its highly optimized development during the years. The same case occurs when considering the counter formulas, in which SPOT performs best, and follows PANDA+CadenceSMV and then our tool *Aalta*. The results are shown in Fig. 9. Actually, a counter formula is a special kind of pattern which constructs the Büchi automaton with the exponential size to the bit number given in the formula. This automaton has only one accepting condition, which has to be explored until the whole automaton is generated.

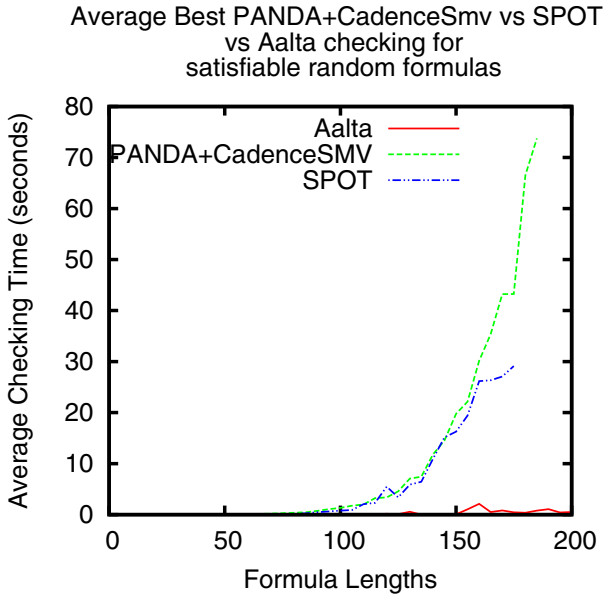

Fig. 13. Experimental results for satisfiable random formulas with 3 variables

As a result, for all three tools, no optimizations can be used to check it satisfiable instead of spending the worst time to generate the automaton. It is not surprising that SPOT performs best on this kind of formulas, since it is already equipped with best automata construction. But as mentioned together, even SPOT must spend exponential cost to check the results.

Aalta performs best for random conjunction formulas. Checking satisfiability of random conjunction formulas is quite challenging, but *Aalta* still performs best. The results are shown in Fig. 10. The number of conjuncts extends only to 15 (with average formula length of 100) and all tools time out for larger formulas. The advantage of *Aalta* here is less marked; it performs about twice as fast as SPOT and PANDA+CadenceSMV. Checking satisfiability for random-conjunction formulas emerges as a challenging problem, requiring further research. It would be interesting to combine *Aalta* with the abstraction technique of [CRST07].

The obligation acceleration enhances on-the-fly checking. One of the effective heuristics of *Aalta* is the OA technique: a consistent obligation implies satisfiability directly. Now we compare here the results from *Aalta* implemented with the OFOA and pure OF strategies in checking random and random conjunction formulas. Figures 11 and 12 indicate that OA indeed plays a key role. For random formulas the OFOA strategy performs much faster than the pure OF strategy. Moreover, the OFOA strategy can be even exponentially better for special cases, such as the R pattern formulas mentioned above. Although the advantage declines for random conjunction formulas, the OFOA strategy is still twice as fast as the OF strategy.

Aalta contributes mainly on satisfiable formulas. Since *Aalta* follows the on-the-fly checking framework, it should contribute more apparently for satisfiable rather than unsatisfiable formulas. The results from Figs. 13 and 14 confirm the intuition. Figure 13 shows the comparing results for satisfiable random formulas, while Fig. 14 shows that for unsatisfiable ones. It takes *Aalta* almost less than 1 second in average to check the satisfiable formulas, which can be seen from Fig. 13. For unsatisfiable formulas, Fig. 14 shows the performances between three tools are not obvious: PANDA+CadenceSMV performs better than *Aalta*. Comparing to the results in Fig. 5, one can conclude easily that *Aalta* contributes mainly for the satisfiable formulas.

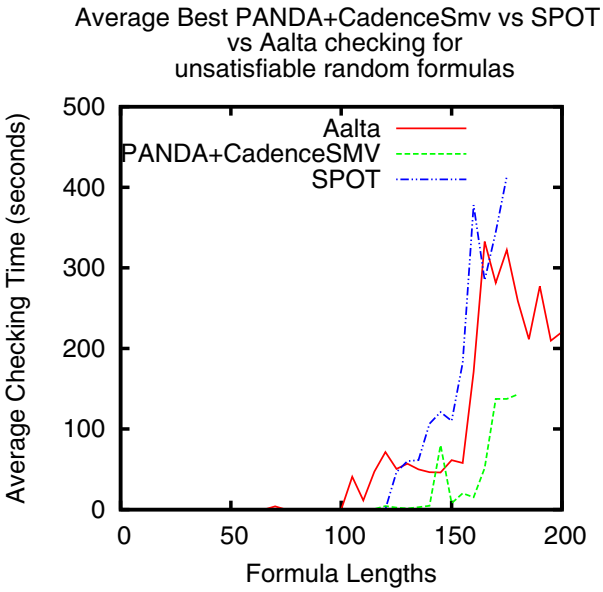

Fig. 14. Experimental results for unsatisfiable random formulas with 3 variables

4.3. Discussion

In this section we have set up the experiment which follows the strategies in [RV11] and focus on merely the comparison among automata-based LTL satisfiability solvers. From this we find our approach has a fairly big advantage.

Note we measure the results by the time cost of checking in the paper. That means, we count each test the time from the original input to the checking result being obtained. In particular, if timeout occurs then the time for this test is considered to be the timeout (10 minutes). We display the results by taking a group of formulas as a unit, and thus we show the total (or average) time cost of the given group of formulas. In this way every test has the same maximum cost and thus the total cost succeeds to indicate the number of solved cases for a group of formulas. In other words, the less total time cost is paid in our statistics, the more cases the corresponding checker can solve within the same maximum timeout. The selection of timeouts depends on the difficulties of benchmarks: The one used in the experiment is simple for all three testing tools so we can set the timeout to be 10 minutes without increasing our whole experimental time (about 24 hours).

5. Related work

The classical approach to LTL satisfiability checking is by reduction to model checking. This can be implemented using either explicit-state techniques or symbolic techniques. Rozier and Vardi [RV07, RV10] studied this approach and benchmarked several tools. They concluded that the combination of SPIN [Hol97] and SPOT [DLP04] yields the best performance for the explicit-state approach, but symbolic tools such as CadenceSMV [McM99] or NuSMV [CCG+02, CRST07] yields better performance. In follow-up work [RV11], Rozier and Vardi studied several symbolic encodings of automata for LTL formulas and described a tool, PANDA, built on top of CadenceSMV, which implements a portfolio approach, running many symbolic encodings in parallel and selecting the best performing one.

It should be mentioned that, the later version of NuSMV (since 2002) integrates bounded model checking [BCCZ99, CBRZ01, CPRS02] as well. The bounded model checking framework encodes both the system and properties symbolically, and then uses SAT solvers [ES03] to compute the results. Thanks to the rapid development of SAT techniques for years, the performance of bounded model checking becomes very efficient. Although this method is not complete, which means that it cannot check any unsatisfiable formulas, it provides a promising direction to achieve a best solution for LTL satisfiability checking. Several related works such as IMC[McM03]

and IC3 [Bra11] are received well concern recently, and we leave the comparison between our approach and them in the future work.

In earlier works, the decision problem of LTL satisfiability checking is known to be PSPACE-complete [SC82]. One kind of checking algorithm is considered as a 2-phase procedure: First use a tableau procedure to create a graph, and then apply another one to check whether all eventuality (Until) formulas are fulfilled. The second step often leads to an exploration on all the strongly connected components (SCC) [Tar72] of the graph. The representatives on this method include [Wol85, KMMP93]. Theoretically, this kind of solution works in EXPTIME.

However, a subsequent work using the tableau structure succeeds with only 1 phase [Sch98]. That means, instead of a graph construction, the tableau procedure generates a tree structure, and then check whether all eventuality (Until) formulas are fulfilled along each branch of the tree. In that way, only one branch is necessary to keep in memory during checking process, which they claim to be more considerable when executing in parallel. The solver pltl is the representative of this approach. The complexity of this approach is considered to be 2EXPTIME. But dramatically, Goranko, Kyrilov, and Shkatov [VGS10] showed that, in practice, the one-phase procedure is more efficient than the two-phase one. They compared these two methods by the benchmarks from [RV07, RV10], testing all the random, pattern and counter formulas, and found that the one-phase procedure can even perform exponentially better than the two-phase one. The conclusion becomes one evidence that, theoretically worst-case complexity results do not always reflect truly in practice.

The satisfiability checking of LTL formulas is also able to be achieved by temporal resolution [FDP01, Sch10]. The main idea is that, the input formula must first be translated to its Separated Normal Form (SNF) [Fis91], and then the clauses in the SNF can be deductive according to some basic rules (or say, axioms for LTL global formulas as each clause in SNF is a global one). The inductive system can be treated as a direct graph, and each time at least one of the clauses in SNF intends to be erased. If all the clauses are erased, then the algorithm recognizes that the input formula is unsatisfiable, otherwise the formula is satisfiable. One can see that this approach performs better for checking unsatisfiable formulas, while the satisfiable formula must be determined after all the exploration. Another bottleneck of this approach is that, translating the input formula to its SNF may lead to an exponential up cost, which is inefficient for on-the-fly checking. One representative solver of this solution is TRP++.

There is also another LTL satisfiability checking strategy based on antichains [DDMR08]. The work follows the model-checking-based framework, but to avoid the potentially exponential up cost on generating NBW (Non-deterministic Büchi Automata), they construct the ABW (Alternating Büchi Automata) instead. They encode the ABW symbolically, and use the combination of BDDs [Bry86, Bry92] and antichains to present the structure efficiently. Then an emptiness checking is taken on the generated ABW. A representative solver is Alaska. The advantages of this method are: first they use the antichains rather than enumerating the whole alphabet to construct the BDDs; And second the checking procedure is processed on the ABW instead of NBW, which potentially performs exponentially faster than the classical model-checking-based approach.

Schuppan and Darmawan [SD11] performed a comprehensive experimental evaluation of LTL satisfiability solvers. They considered a wide range of solvers implementing three major classes of algorithms, based on model checking, tableau, and temporal resolution. They concluded that no solver dominates or solves all instances, and recommend a portfolio approach, similar to that of [RV11]. Our tool, *Aalta*, is closest in spirit to the model-checking approach, but it combines automaton generation and non-emptiness checking in an on-the-fly approach. Briefly, we first construct a transition system for the input formula based on the observation that every formula has an equivalent normal form [LPZ+13]. Then a central theorem is proposed to guarantee the checking on the transition system with an on-the-fly manner. Beside that, some accelerations are also introduced from the obligation set, which can be naturally extracted from an LTL formula. Note that the concept of normal form is also involved in [DTZ08]. In this paper we not only demonstrate its performance advantage over automata-based tools, but also set up a comprehensive comparison in the style of [SD11].

6. Conclusions

In this paper, we proposed a novel on-the-fly satisfiability checking approach for LTL formulas. Our approach exploits the notion of obligation set, which provides efficient ways for identifying many satisfiable formulas. We

have implemented a tool, *Aalta*, and run experiments using existing and new benchmarks. In most of the cases, *Aalta* significantly outperforms existing automata-based LTL satisfiability solvers.

Acknowledgements

Jianwen Li is partially supported by NSFC Project No. 61572197 and No. 61632005. Geguang Pu is partially supported by MOST NKTSP Project 2015BAG19B02 and STCSM Project No. 16DZ1100600. Lijun Zhang is supported by the National Natural Science Foundation of China (Grants 61532019, 61472473). Jifeng He is partially supported by project Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (ZF1213). Moshe Vardi is supported in part by NSF Grants CCF-1319459 and IIS-1527668, and by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering".
Funding Funding was provided by National Natural Science Foundation of China (Grant Nos. 91118007, 61021004, 61361136002), National Science Foundation (Grant No. CNS 1049862).

References

[BCCZ99] Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of the 5th international conference on tools and algorithms for the construction and analysis of systems, volume 1579 of Lecture notes in computer science. Springer

[BCM+92] Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170

[Bra11] Bradley A (2011) Sat-based model checking without unrolling. In: Jhala R, Schmidt D (eds) Verification, model checking, and abstract interpretation, volume 6538 of Lecture notes in computer science, pp 70–87. Springer, Berlin

[Bry86] Bryant RE (1986) Graph-based algorithms for Boolean-function manipulation. IEEE Trans Comput C-35(8):677–691

[Bry92] Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3):293–318

[CBRZ01] Clarke EM, Bierea A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34

[CCG+02] Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: an opensource tool for symbolic model checking. In: Computer aided verification, Lecture notes in computer science 2404, pp 359–364. Springer

[CCGR00] Cimatti A, Clarke EM, Giunchiglia F, Roveri M (2000) NuSMV: a new symbolic model checker. Int J Softw Tools Technol Transf 2(4):410–425.

[CGP99] Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge

[CPRS02] Cimatti A, Pistore M, Roveri M, Sebastiani R (2002) Improving the encoding of ltl model checking into sat. In: Revised papers from the third international workshop on verification, model checking, and abstract interpretation, VMCAI '02, pp 196–207. Springer, London

[CRST07] Cimatti A, Roveri M, Schuppan V, Tonetta S (2007) Boolean abstraction for temporal logic satisfiability. In: Proceedings of the 15th international conference on computer aided verification, volume 4590 of Lecture notes in computer science, pp 532–546. Springer

[CVWY92] Courcoubetis C, Vardi MY, Wolper P, Yannakakis M (1992) Memory efficient algorithms for the verification of temporal properties. Formal Methods Syst Des 1:275–288

[DAC98] Dwyer MB, Avrunin GS, Corbett JC (1998) Property specification patterns for finite-state verification. In: Proceedings of the 2nd workshop on formal methods in software practice, pp 7–15. ACM

[DDMR08] De Wulf M, Doyen L, Maquet N, Raskin J-F (2008) Antichains: alternative algorithms for ltl satisfiability and model-checking. In: Tools and algorithms for the construction and analysis of systems, volume 4963 of Lecture notes in computer science, pp 63–77. Springer

[DGV99] Daniele N, Guinchiglia F, Vardi MY (1999) Improved automata generation for linear temporal logic. In: Proceedings of the 11th intenational conference on computer aided verification, volume 1633 of Lecture notes in computer science, pp 249–260. Springer

[DLP04] 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, pp 76–83. IEEE Computer Society

[DTZ08] Duan Z, Tian C, Zhang L (2008) A decision procedure for propositional projection temporal logic with infinite models. Acta Inf 45(1):43–78

[ES03] Eén N, Sörensson N (2003) An extensible sat-solver. In: SAT, pp 502–518

[FDP01] Fisher M, Dixon C, Peim M (2001) Clausal temporal resolution. ACM Trans Comput Logic 2(1):12–56

[Fis91] Fisher M (1991) A resolution method for temporal logic. In: In proceedings of the twelfth international joint conference on artificial intelligence, pp 99–104. IJCAI, Morgan Kaufman

[FKL04] Foster HD, Krolnik A, Lacey DJ (2004) Assertion-based design. Springer, Berlin

[Hol97] Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295

[KMMP93] Kesten Y, Manna Z, McGuire H, Pnueli A (1993) A decision algorithm for full propositional temporal logic. In: Courcoubeti C (ed) Proceedings of the 5th conference on computer aided verification, volume 697 of Lecture notes in computer science, pp 97–109. Springer

[LPZ+13] Li J, Pu G, Zhang L, Wang Z, He J, Larsen KG (2013) On the relationship between ltl normal forms and büchi automata. In: Liu Z, Jim W, Zhu H (eds) Theories of programming and formal methods, volume 8051 of Lecture notes in computer science, pp 256–270. Springer

[LZP+13] 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

[McM99] McMillan K (1999) The SMV language. Technical report, Cadence Berkeley Lab

[McM03] McMillan K (2003) Interpolation and sat-based model checking. In: Jr. Hunt, WarrenA, Somenzi Fabio (eds) Computer aided verification, volume 2725 of Lecture notes in computer science, pp 1–13. Springer, Berlin

[PSC+06] Pill I, Semprini S, Cavada R, Roveri M, Bloem R, Cimatti A (2006) Formal analysis of hardware requirements. In: Proceedings of the 43rd design automation conference, pp 821–826. ACM

[RV07] Rozier KY, Vardi MY (2007) LTL satisfiability checking. In: Proceedings of the 14th international SPIN workshop, volume 4595 of Lecture notes in computer science, pp 149–167. Springer

[RV10] Rozier KY, Vardi MY (2010) LTL satisfiability checking. Int J Softw Tools Technol Transf 12(2):1230–137

[RV11] Rozier KY, Vardi MY (2011) A multi-encoding approach for LTL symbolic satisfiability checking. In: Proceedings of the 17th International symposium on formal methods, volume 6664 of Lecture notes in computer science, pp 417–431. Springer

[SC82] Sistla AP, Clarke EM (1982) The complexity of propositional linear temporal logics. In: Proceedings of the 14th annual ACM symposium on theory of computing, pp 159–168

[SC85] Sistla AP, Clarke EM (1985) The complexity of propositional linear temporal logic. J ACM 32:733–749

[Sch98] 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, pp 277–292. Springer

[Sch10] Schuppan V (2010) Towards a notion of unsatisfiable cores for ltl. In: Fundamentals of software engineering, pp 129–145

[SD11] Schuppan V, Darmawan L (2011) Evaluating ltl satisfiability solvers. In: Proceedings of the 9th international conference on Automated technology for verification and analysis, AVTA'11, pp 397–413. Springer

[Tar72] Tarjan RE (1972) Depth first search and linear graph algorithms. SIAM J Comput 1(2):146–160

[Var07] Vardi MY (2007) Automata-theoretic model checking revisited. In: Proceedings of the 8th international conference on verification, model checking, and abstract interpretation, volume 4349 of Lecture notes in computer science, pp 137–150. Springer

[VGS10] Kyrilov A, Goranko V, Shkatov D (2010) Tableau tool for testing satisfiability in ltl: Implementation and experimental analysis. Electr Notes Theory Comput Sci 262:113–125

[VW86] Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st IEEE symposium on logic in computer science, pp 332–344

[Wol85] Wolper P (1985) The tableau method for temporal logic: an overview. Logique Anal 110–111:119–136

Received 14 June 2016

Accepted in revised form 27 September 2017 by Jim Woodcock

Published online 9 November 2017