SAT-Based Explicit LTL Reasoning

Jianwen Li†, Shufang Zhu‡, Geguang Pu†, Moshe Vardi∗
∗Computer Science Department, Rice University, TX, USA
†Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

Abstract—We present here a new explicit reasoning framework for linear temporal logic (LTL), which is built on top of propositional satisfiability (SAT) solving. As a proof-of-concept of this framework, we describe a new LTL satisfiability tool, Aalta_v2.0, which is built on top of the MiniSAT SAT solver. We test the effectiveness of this approach by demonstrating that Aalta_v2.0 significantly outperforms all existing LTL satisfiability solvers. Furthermore, we show that the framework can be extended from propositional LTL to assertional LTL (where we allow theory atoms), by replacing MiniSAT with the Z3 SMT solver, and demonstrating that this can yield an exponential improvement in performance.

I. INTRODUCTION

Linear Temporal Logic (LTL) was introduced into program verification in [25]. Since then it has been widely accepted as a language for the specification of ongoing computations [20] and it is a key component in the verification of reactive systems [4], [14]. Explicit temporal reasoning, which involves an explicit construction of temporal transition systems, is a key algorithmic component in this context. For example, explicitly translating LTL formulas to Büchi automata is a key step both in explicit-state model checking [11] and in runtime verification [31]. Also, LTL satisfiability checking, a step that should take place before verification, to assure consistency of temporal requirements, also uses explicit reasoning [26]. These tasks are known to be quite demanding computationally for complex temporal properties [11], [26], [31]. A way to get around this difficulty is to replace explicit reasoning by symbolic reasoning, e.g., as in BDD-based or SAT-based model checking [23], [22], but in many cases the symbolic approach is inefficient [26] or inapplicable [31]. Thus, explicit temporal reasoning remains an indispensable algorithmic tool.

The dominant approach to explicit temporal reasoning is based on the tableau technique, in which a recursive syntactic decomposition of temporal formulas drives the construction of temporal transition systems. This approach is based on the technique of propositional tableau, whose essence is search via syntactic splitting [6]. This is in contrast to modern propositional satisfiability (SAT) solvers, whose essence is search via semantic splitting [19]. The tableau approach to temporal reasoning underlies both the best LTL-to-automata translator [10] and the best LTL-satisfiability checker [18]. Thus, we have a situation where in the symbolic setting much progress is being attained both by the impressive improvement in the capabilities of modern SAT solvers [19] as well as new SAT-based model-checking algorithms [11], [9]. While progress in explicit temporal reasoning is slower and does not fully leverage modern SAT solving. (It should be noted that several LTL satisfiability solvers, including Aalta [17], TRP++ [15], and ls4 [30] do employ SAT solvers, but they do so as an aid to the main reasoning engine, rather than serve as the main reasoning engine.)

Our main aim in this paper is to study how SAT solving can be fully leveraged in explicit temporal reasoning. The key intuition is that explicit temporal reasoning consists of construction of states and transitions, subject to temporal constraints. Such temporal constraints can be reduced to a sequence of Boolean constraints, which enables the application of SAT solving. This idea underlies the complexity-theoretic analysis in [33], and has been explored in the context of modal logic [12], but not yet in the context of explicit temporal reasoning. Our belief is that SAT solving would prove to be superior to tableau in that context.

Additional motivation to base explicit temporal reasoning on SAT solving is the need to handle LTL formulas with assertional atoms, that is, atoms that are non-boolean state assertions, e.g., assertions about program variables, such as \( k \leq 10 \). Existing explicit temporal-reasoning techniques abstract such assertions as propositional atoms. Consider, for example, the LTL formula \( \phi = \bigwedge_{1 \leq i \leq n} F(k = i) \), which asserts that \( k \) should assume all values between 1 and \( n \). By abstracting \( k = i \) as \( p_i \), we get the formula \( \phi' = \bigwedge_{1 \leq i \leq n} F_{p_i} \), but the transition system for the abstract formula has \( 2^n \) states, while the transition system for the original formula has only \( n \) states. This problem was noted, but not solved in [31], but it is obvious that reasoning about non-boolean assertions requires reasoning at the assertion level. Basing explicit temporal reasoning on SAT solving, would enable us to lift it to the assertion level by using Satisfiability Modulo Theories (SMT) solving. SMT solving is a decision problem for logical formulas in combinations of background theories expressed in classical first-order logic. Examples of theories typically used are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors, and others. SMT solvers have shown dramatic progress over the past couple of decades and are now routinely used in industrial software development [24].

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

As proof of concept for the new framework, we develop a SAT-based LTL-satisfiability solver. To check its performance, we compare it against Aalta, the previous best-of-breed LTL-satisfiability solver [18], [17], which is tableau-based. We also compare it against NuXmv, a symbolic LTL-satisfiability solver that is based on cutting-edge SAT-based model-checking...
algorithms [1], [3], which outperforms Aalta. We show that our explicit SAT-based LTL-satisfiability solver outperforms both. We also demonstrate the extensibility of our framework by demonstrating an exponential improvement for satisfiability checking of LTL formulas with assertional atoms.

In summary, the contributions in this paper are as follows:

- We propose a SAT-based explicit LTL-reasoning framework.
- We evaluate the effectiveness of this framework by developing a best-of-breed LTL-satisfiability solver.
- We demonstrate the extensibility of the framework by showing its effectiveness for LTL formulas with assertional atoms.

This paper is organized as follows. Section II provides technical background. Section III introduces the new SAT-based explicit-reasoning framework. Section IV describes in detail the application to LTL-satisfiability checking. Section V shows the experimental results for LTL-satisfiability checking. Section VI describes the extension to LTL formulas with assertional atoms. Finally Section VII provides concluding remarks. Missing proofs are in the Appendix.

II. Preliminaries

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

\[ \phi ::= tt \mid ff \mid a \mid \neg \phi \mid \phi \land \psi \mid \phi \lor \psi \mid \phi U \psi \mid X \phi \]

where \( a \in AP \), tt is true and ff is false. \( \phi \) is an LTL formula. We can also introduce the R (release) operator as the dual of U, which means \( \phi R \psi \equiv \neg (\neg \phi U \neg \psi) \). Specially, we use the usual abbreviations: \( Fa = tt U a \), and \( Ga = ff Ra \).

We say \( \phi \) is a literal if it is an atomic proposition or its negation. Throughout the paper, we use \( L \) to denote the set of literals, lower case letters \( a, b, c, \ldots \) to denote literals, \( \alpha \) to denote propositional formulas, and \( \phi, \psi \) for LTL formulas. We also consider LTL formulas in negation normal form (NNF), which can be achieved by pushing all negations in front of only atoms. LTL formulas are often interpreted over \( (2^{AP})^\omega \). Since we consider LTL in NNF, formulas are interpreted on infinite literal sequences \( \Sigma := (2^k)^\omega \).

A trace \( \xi = \omega_1 \omega_2 \ldots \) is an infinite sequence over \( \Sigma^\omega \). For \( \xi \) and \( k \geq 1 \) we use \( \xi^k = \omega_1 \omega_2 \ldots \omega_{k-1} \) to denote a prefix of \( \xi \), and \( \xi_k = \omega_k \omega_{k+1} \ldots \) to denote a suffix of \( \xi \). Thus, \( \xi = \xi^k \xi_k \).

The semantics of temporal operators with respect to an infinite trace \( \xi \) is given by:

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

We say that a propositional assignment \( A \) propositional satisfies \( \phi \), denoted as \( A \models P \phi \), if \( \phi \in A \). We say an LTL formula \( \phi \) is propositional satisfiable if there is a propositional assignment \( A \) for \( \phi \) such that \( A \models P \phi \).

For example, consider the formula \( \phi = (aUb) \land (b) \). The set \( A_1 = \{ a, (aUb), (b), (X(aUb)) \} \subseteq cl(\phi) \) is a propositional assignment that propositional satisfies \( \phi \). In contrast, the set \( A_2 = \{ (aUb), (b) \} \subseteq cl(\phi) \) is not a propositional assignment.

The following theorem shows the relationship between LTL formula \( \phi \) and its propositional assignment.

**Theorem 1:** For an LTL formula \( \phi \) and an infinite trace \( \xi \in L^\omega \), we have that \( \xi \models \phi \) if and only if there exists an propositional
assignment $A \subseteq \text{cl}(\phi)$ such that $A$ propositionally satisfies $\phi$ and $\xi = \bigwedge A$.

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

Definition 2: Given an LTL formula $\phi$, the transition system $T_\phi$ is a tuple $(S, S_0, T)$ where

- $S$ is the set of states $s \subseteq \text{cl}(\phi)$ that are propositional assignments for $\phi$. The trace of a state $s$ is $s \cap L$, that is, the set of literals in $s$.
- $S_0 \subseteq S$ is a set of initial states, where $\phi \in S_0$ for all $s_0 \in S_0$.
- $T : S \times S$ is the transition relation, where $T(s_1, s_2)$ holds if $(X\theta) \in s_1$ implies $\theta \in s_2$, for all $X\theta \in \text{cl}(\phi)$.

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

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

Theorem 2: Let $r$ be a run of $T_\phi$. If no Until subformula is stuck at $r$, then $\text{trace}(r) \models \phi$. Also, $\phi$ is satisfiable if there is a run of $T_\phi$ so that no Until subformula is stuck at $r$.

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

Remark 1: The standard approach in LTL model checking relies on the translation of LTL formulas to Büchi automata. (Since we do not pursue this line of inquiry further in this paper, we assume familiarity with this approach; cf. [11].) The transition system $T_\phi$ can also be used to construct generalized Büchi automata.

Given an LTL formula $\phi$ and its transition system $T_\phi = (S, S_0, T)$. The generalized Büchi automaton $A(\phi)$ is a tuple $(2^L, S, S_0, \delta, (F_1, \ldots, F_k))$, where

- $s' \in \delta(s, a)$ if $a = \text{trace}(s)$ and $T(s, s')$ holds.
- Let the Until subformulas of $\phi$ be $\psi_1, \ldots, \psi_k$. Then the generalized acceptance condition is $(F_1, \ldots, F_k)$, where $F_i = \{s \in S : \text{ if } \psi_i \in s \text{, then } s \text{ satisfies } \psi_i \text{ immediately}\}$.

Several papers focus on efficient generation of such automata, cf. [11], [7], [29].

B. System Construction

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

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

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

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

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

We now introduce the neXt Normal Form (XNF) of LTL formulas, which separates the “current” and “next-state” parts of the formula, but involves only a linear blow-up.

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

For example, $\phi = (aUb)$ is not in XNF, while $(b \lor (a \land (X(aUb))))$ is in XNF. Every LTL formula $\phi$ can be converted, with linear blow-up, to an equivalent formula in XNF.

Theorem 3: For an LTL formula $\phi$, there is an equivalent formula $\text{xnf}(\phi)$ that is in XNF. Furthermore, the blow-up of the conversion is linear.

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

1) $\text{xnf}(\phi) = \phi$ if $\phi$ is $tt$, $ff$, a literal $l$ or a Next formula $X\psi$;
2) $\text{xnf}(\phi) = \text{xnf}(\phi_1) \lor \text{xnf}(\phi_2)$ if $\phi = (\phi_1 \lor \phi_2)$;
3) $\text{xnf}(\phi) = \text{xnf}(\phi_1) \land \text{xnf}(\phi_2)$ if $\phi = (\phi_1 \land \phi_2)$;
4) $\text{xnf}(\phi) = (\text{xnf}(\phi_2) \lor (\text{xnf}(\phi_1) \land X\phi))$ if $\phi = (\phi_1 U \phi_2)$;
5) $\text{xnf}(\phi) = (\text{xnf}(\phi_1) \lor \text{xnf}(\phi_1) \land X\phi)$ if $\phi = (\phi_1 R \phi_2)$.

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

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

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

Conversely, if there is a propositional assignment $A'$ of $\phi$ that
satisfies \( \phi \), then there is a satisfying assignment \( A \) of \( \phi^p \) such that \( A' \cap PA(\phi) \subseteq A \).

Proof: (\( \Rightarrow \)) Let \( A \) be a satisfying assignment of \( \phi^p \). Then let \( A' \) be the set of all formulas \( \psi \in cl(\phi) \) such that \( A \) satisfies \( (\text{xnf}(\psi))^p \). We clearly have that \( A' \cap PA(\phi) \subseteq A \). According to Definition [1] and because \( \phi \) is in XNF, we have that \( A' \) is a propositional assignment of \( \phi \) that satisfies \( \phi \).

(\( \Leftarrow \)) Let \( A' \) be a propositional assignment of \( \phi \) that satisfies \( \phi \). Then let \( A \) be the assignment that assigns true to \( \psi \in cl(\phi) \) precisely when \( \psi \in A' \). Again, we clearly have that, \( A' \cap PA(\phi) \subseteq A \). According to Definition [1] and because \( \phi \) is in XNF, we have that \( A \) is a satisfying assignment of \( \phi^p \).

Theorem 2 shows that by requiring the formula \( \phi \) to be in XNF, we can construct the states of the transition system \( T \) via computing satisfying assignments of \( \phi^p \) over \( PA(\phi) \). Let \( t \) be a satisfying assignment of \( \phi^p \) and \( A_t \) be the related propositional assignment of \( \phi \) generated from \( t \) by Theorem 4 the construction is operated as follows:

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

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

Our goal here is to show that we can construct the transition system \( T_0 \) by means of SAT solving. This requires us to refine Theorem 2. A key issue in how a propositional assignment handles an Until formula is whether it satisfies it immediately or postpones it. We introduce new propositions that indicate whether there is an infinite trace \( |s| \) such that \( \phi \) is satisfied.

Theorem 5: For an LTL formula \( \phi, \phi \) is satisfiable iff there is a finite run \( r = s_0, s_1, \ldots, s_n \) in \( T_0 \) such that

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

The significance of Theorem 5 is that it reduces LTL satisfiability checking to searching for a “lasso” in \( T_0 \). Item 1) says that we need to search for a prefix followed by a cycle, while Item 2) provides a way to test that no Until subformula gets stuck in the infinite run in which the cycle \( s_{m+1}, \ldots, s_n \) is repeated infinitely often.

C. Related Work

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

Earlier approach to transition-system construction for LTL formulas, based on tableau [11] and normal form [18], generating the system states explicitly or implicitly via a translation to disjunctive normal form (DNF). In [18], the conversion to DNF is explicit (though various heuristics are used to temper the exponential blow-up) and the states generated corresponds to the disjuncts. In tableau-based tools, cf., [11], the construction is based on iterative syntactic splitting in which a state of the form \( \phi \cup \{ \theta_1 \} \cup \{ \theta_2 \} \) is split to states: \( A \cup \{ \theta_1 \} \) and \( A \cup \{ \theta_2 \} \).

The approach proposed here is based on SAT solving, where the states correspond to satisfying assignments. Satisfying assignments are generated via a search process that is guided by semantic splitting. The advantage of using SAT solving rather than syntactic approaches is the impressive progress in the development of heuristics that have evolved to yield highly efficient SAT solving: unit propagation, two-literal watching, back jumping, clause learning, and more, see [19]. Furthermore, SAT solving continues to evolve in an impressive pace, driven by an annual competition [17]. It should be remarked that an analogous debate, between syntactic and semantic approaches, took place in the context of automated test-pattern generation for circuit designs, where, ultimately, the semantic approach has been shown to be superior [16].

Furthermore, relying on SAT solving as the underlying reasoning technology enables us to decouple temporal reasoning from propositional reasoning. Temporal reasoning is accomplished via a search in the transition system, while the construction of the transition system, which requires propositional reasoning using SAT solving. Using SAT solving also allows us to lift the reasoning from the propositional level to the assertional level, as in discussed below.

IV. LTL Satisfiability Checking

Given an LTL formula \( \phi \), the satisfiability problem is to ask whether there is an infinite trace \( \xi \) such that \( \xi \models \phi \). In the previous section we introduced a SAT-based LTL-reasoning framework and showed how it can be applied to solve LTL satisfiability problems. In this section we use this framework to develop an efficient SAT-based algorithm for LTL satisfiability checking. We design a depth-first-search (DFS) algorithm that constructs the temporal transition system on the fly and searches for a trace per Theorem 5. Furthermore, we propose several heuristics to reduce the search space. Due to the limited space, we offer here a high-level description of the algorithms. Further details can be found in a technical report.

A. The Main Algorithm

The main algorithm, LTL-CHECK, creates the temporal transition system of the input formula on-the-fly, and searches for a lasso in a DFS manner. Several prior works describe algorithms for DFS lasso search, cf., [5], [18], [23]. Here we focus on the steps that are specialized to our algorithm.

The key idea of LTL-CHECK is to create states and their successors using SAT techniques rather than traditional tableau or expansion techniques. Given the current formula \( \phi \), we first compute its XNF version \( \text{xnf}(\phi) \), and then use a SAT solver to compute the satisfying assignments of \( \text{xnf}(\phi)^p \). Let \( P \) be a

1. See http://www.satcompetition.org/
2. http://www.lab205.org/home/pages/hijianwen/data/ltlsatreasoning.pdf
satisfying assignment for \((xnf(\phi))^P\); from the previous section we know that \(X(P) = \{\theta | X\theta \in P\}\) yields a successor state in \(T_\theta\). We implement this approach in the \textit{getState} function, which we improve later by introducing some heuristics. By enumerating all assignments of \((xnf(\phi))^P\) we can obtain all successor states of \(P\). Note, however that LTL-CHECK runs in the DFS manner, under which only a single state is needed at a time, so additional effort must be taken to maintain history information of the next-state generation for each state \(P\).

As soon as LTL-CHECK detects a lasso, it checks whether the lasso is accepting. Previous lasso-search algorithms operate on the Büchi automaton generated from the input formula. In contrast, here we focus directly on the satisfaction of Until subformulas per Theorem 5. We use the example below to show the general idea.

Consider the formula \(\phi = G((Fb) \land (Fc))\). By Theorem 3, \(xnf(\phi) = xnf(Fb) \land xnf(Fc) \land X\phi\), where \(xnf(Fb) = ((b \land v(Fb)) \lor (\neg v(Fb) \land X(Fb)))\) and \(xnf(Fc) = ((c \land v(Fc)) \lor (\neg v(Fc) \land X(Fc)))\). Suppose we get from the SAT solver an assignment of \((xnf(\phi))^P\) \(P = \{v(Fb), \neg v(Fc), b, c, \neg X(Fb), X(Fc), X \phi\}\). By Theorem 4, we create a satisfying assignment \(A\) that includes all formulas in \(cl(\phi)\) that are satisfied by \(P\), and we get the state \(s_0 = P \cup \{\phi, Fb, Fc, (Fb) \land (Fc)\}\). To obtain the next state, we start with \(X(s_0) = \{Fb, \phi\}\), compute \(xnf(Fb) \land \phi\) and repeat the process. After several steps LTL-CHECK may find a path \(s_0 \rightarrow s_1 \rightarrow s_0\), where \(s_1 = \{\phi, Fb, Fc, (Fb) \land (Fc), \neg v(Fb), v(Fc), b, c, X(Fb), \neg X(Fc), X \phi\}\). Now \(s_0\) and \(s_1\) form a lasso. Let \(Q = s_0 \cup s_1\). Both \(Fb\) and \(Fc\) are in \(Q\), but also \(v(Fb)\) and \(v(Fc)\) are in \(Q\). By Theorem 5, \(\phi\) is satisfiable.

### B. Heuristics for State Elimination

While LTL-CHECK uses an efficient SAT solver to compute states of the system in the \textit{getState} function, this approach is effective in creating states and their successors, but cannot be used to guide the overall search. To find a satisfying lasso faster, we add heuristics that drives the search towards satisfaction. The key to these heuristics is smartly choosing the next state given by SAT solvers. This can be achieved by adding more constraints to the SAT solver. Our experiments show that these heuristics are critical to the performance of our LTL-satisfiability tool.

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

The approach we now describe is to satisfy obligations as early as possible during the search, so that a satisfying lasso is also obtained earlier. We now refine the \textit{getState} function, and introduce three heuristics via examples.

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

The \textit{getState} function keeps a global obligation set, collecting all obligations so far not satisfied in the search. The obligation set is initialized with the obligations of initial formulas \(\phi_0\). When an obligation \(o\) is satisfied (i.e., when \(v(\phi)\) is true), \(o\) is removed from the obligation set. Once the obligation set becomes empty in the search, it is reset to contain obligations of current representative formula \(\phi_1\). In Fig. 1, we denote the obligation set by \(O\). \(O\) is initialized to \(\emptyset\), as there is no obligation in \(\phi_0\). \(O\) is then reset in the states \(s_1\) and \(s_3\), when it becomes empty.

The \textit{getState} function runs in the \textbf{ELIMINATION} mode by default, in which it obtains the next state guided by the obligations of current state. For satisfiable formulas, this leads to faster lasso detection. Consider formula \(\phi = G((Fa) \land (F \neg a))\). Parts of the temporal transition system \(T_\phi\) are shown in Fig. 1. In the figure, \(O\) is reset to \(\{(Fa), (F \neg a)\}\) in state \(s_1\), as these are the obligations of \(\phi_1\). To drive the search towards early satisfaction of obligations, we obtain a successor of \(s_1\), by applying the SAT solver to the formula \((xnf(\phi_1) \land v(Fa))\lor v(F \neg a))^P\), to check whether \(Fa\) or \(F \neg a\) can be satisfied immediately. If the returned assignment satisfies \(v(Fa)\), then we get the successor state \(s_2\) with the representative formulas \(\phi_2\), and \((Fa)\) is removed from \(O\). Then the next state is \(s_3\) with the representative formula \(\phi_3\), which removes the obligation \((F \neg a)\). Since \(O\) becomes empty, it is reset to the obligations \(\{(Fa)\}\) of \(\phi_3\). Note that in Fig. 1, there should be transitions from \(s_2\) to \(s_1\) and from \(s_3\) to \(s_2\), but they are never traversed under the \textbf{ELIMINATION} mode.

The \textit{getState} function runs in the \textbf{SAT PURSUING} mode when the obligation set becomes empty. In this mode, we want to check whether the next state can be a state that have been visited before and after that visit the obligation set has become empty. In this case, the generated lasso is accepting, by Theorem 5. In Fig. 1, the obligation set \(O\) becomes empty in state \(s_3\). Previously, it has become empty in \(s_1\). Normally, we find a successor state for \(s_3\) by applying the SAT solver to \((xnf(\phi_3))^P\). To find out if either \(s_0\) or \(s_1\) can be a successor of \(s_3\), we apply the SAT solver to the formula \((xnf(\phi_3) \land (X(\phi_0))\lor X(\phi_1))^P\). Since this formula is satisfiable and indicates a transition from \(s_3\) to \(s_1\) \((X\phi_1\) can be assigned true in the assignment), we
have found that $\text{trace}(s_0), (\text{trace}(s_1), \text{trace}(s_2), \text{trace}(s_3))$\textsuperscript{ω} satisfies $\phi$. In the figure, the transitions labeled $x$ represent failed attempts to generate the lasso when $O$ becomes empty. Although failed attempts have a computational cost, trying to close cycles aggressively does pay off.

The get\textsuperscript{State} function runs in the CONFLICT\_ANALYZE mode if all formulas in the obligation set are postponed in the ELIMINATION mode. The goal of this mode is to eliminate “conflicts” that block immediate satisfaction of obligations. To achieve this, we use a conflict-guided strategy. Consider, for example, the formula $\phi_0 = a \land (Xb) \land F(\lnot a) \land (\lnot b)$. Here the formula $\psi = F(\lnot a) \land (\lnot b)$ is an obligation. We check whether $\psi$ can be satisfied immediately, but it fails. The reason for this failure is the conjunct $a$ in $\phi$, which conflicts with the obligation $\psi$. We identify this conflicting using a minimal unsat core algorithm\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}. To eliminate this conflict, we add the conjunct $\lnot Xa$ to $\phi$, hoping to be able to satisfy the obligation immediately in the next state. When we apply the SAT solver to $(xnf(\phi) \land (\lnot Xa))^p$, we obtain a successor state with the representative formula $\phi_1 = (b \land \psi)$, again with $\psi$ as an obligation. When we try to satisfy $\psi$ immediately, we fail again, since $\psi$ conflicts with $b$. To block both conflicts, we add $\lnot Xb$ as an additional constraint, and apply the SAT solver to $(xnf(\phi) \land (\lnot Xa) \land (\lnot Xb))^p$. This yields a successor state with the representative formula $\phi_2 = \psi$. Now we are able to satisfy $\psi$ immediately, and we are able to satisfy $\psi$ with the finite path $\phi \rightarrow \phi_1 \rightarrow \phi_2$.

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

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

V. EXPERIMENTS ON LTL SATISFIABILITY CHECKING

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

A. Experimental Methodologies

The platform used in the experiments is the DA VinCI\footnote{http://cgi.csc.liv.ac.uk/~konev/software/trp++/} system at Rice University. The system is an IBM iDataPlex consisting of 2304 processor cores in 192 Westmere nodes (12 processor cores per node) at 2.83 GHz with 48 GB of RAM per node (4 GB per core), running the 64-bit Redhat 7 operating system. In our experiments, each tool runs on a single core in a single node. We use the Linux command “time” to evaluate the time cost (in seconds) of each experiment. Timeout was set to 60 seconds, and the out-of-time cases are set to cost 60s.

We implemented the satisfiability-checking algorithms introduced in this paper, and named the tool Aalta\_v2.0. We compare here Aalta\_v2.0 with its previous version, Aalta\_v1.2, which is mainly based on a normal-form-expansion technique (though it does use some SAT solving for acceleration)\footnote{It can be downloaded at http://www.lab205.org/aalta}. The SAT engine used in both Aalta\_v1.2 and Aalta\_v2.0 is MiniSAT\footnote{http://cgi.csc.liv.ac.uk/~konev/software/trp++/}. Two resolution-based LTL satisfiability solvers, TRP++ \footnote{http://www.rcsg.rice.edu/sharecore/davinci/} and ls4 \footnote{http://cgi.csc.liv.ac.uk/~konev/software/trp++/}, also utilize SAT solving, and we include them in our comparison.

As shown in\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, LTL satisfiability checking can be reduced to model checking. While BDD-based model checker were shown to be competitive for LTL satisfiability solving in\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, they were shown later not to be competitive with specialized tools\footnote{http://cgi.csc.liv.ac.uk/~konev/software/trp++/}. We do, however, include in our comparison the model checker NuXmv\footnote{http://cgi.csc.liv.ac.uk/~konev/software/trp++/}, which integrates the latest SAT-based model checking techniques. It uses MiniSAT as the SAT engine as well. Although standard bounded model checking (BMC) is not complete for the LTL satisfiability checking, there are techniques to make it complete, for example, incremental bounded model checking (BMC-INC)\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, which is implemented in NuXmv. In addition, NuXmv implements also new SAT-based techniques, IC3\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, which can handle liveness properties with the K-liveness technique\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}. We included IC3 with K-liveness in our comparison.

To compare with the K-liveness checking algorithm, we ran NuXmv using the command “check\_ltlspec\_klive -d". For the BMC-INC comparison, we run NuXmv with the command “check\_ltlspec\_sbmc\_inc -c". Aalta\_v2.0 and Aalta\_v1.2 tools were used running their default parameters. For the other tools, ls4 runs with “\_r2l" and TRP++ runs with “\_sBFS -FSR”. Since the input of TRP++ and ls4 must be in SNF (Separated Normal Form\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}), an SNF generator is required for running these tools. A generator translate is available from the TRP++ website\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}. The parameters of translate is “\_s -r".

In the experiments we consider the benchmark suite from\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, referred to as schuppan-collected. This suite collects formulas from several prior works, including\footnote{http://www.rcsg.rice.edu/sharecore/davinci/}, and has a total of 7446 formulas (3723 representative formulas and their negations). (Testing also the negation of each formula is in essence a check for validity.) In our experiments, we did not find any inconsistency among the solvers that did not time out.

B. Results

The experimental results are shown in Table 1. In the table, the first column lists the different benchmarks in the suite, and the second to seventh columns display the results from different solvers. Each result in a cell of the table is a tuple $(t, n)$, where $t$ is the total checking time for the corresponding benchmark, and $n$ is the number of unsolved formulas due to timeout in the benchmark. Finally, the last row of the table lists the total checking time and number of unsolved formulas for each solver.
The results show that while the tableau-based tool Aalta_v1.2, outperforms ls4 and TRP++, it is outperformed by NuXmv-BMCINC and NuXmv-IC3-Klive, both of which are outperformed by Aalta_v2.0, which is faster by about 6,000 seconds and solves 47 more instances than NuXmv-IC3-Klive.

Aalta_v2.0 outperforms its predecessor Aalta_v1.2 dramatically, faster by 20,000 seconds and solving 271 more instances. Indeed, when Aalta_v1.2 fails it is often due to timeout during the heavy-duty normal-form generation, which Aalta_v2.0 simply avoids (generating XNF is rather lightweight).

The results also demonstrate the effectiveness of the heuristics presented in the paper. For example, the “/trp/N12” and “/robots” benchmarks are mostly unsatisfiable formulas, which Aalta_v1.2 does not handle well. Yet the unsat-core extraction heuristic, which is described in the CONFLICT_ANALYZE mode of getState function, enables Aalta_v2.0 to solve all these formulas. For satisfiable formulas, the results from “/anzu/ambta” and “/anzu/genbuf” formulas, which are satisfiable, show the efficiency of the ELIMINATION and SAT_PUSHER heuristics in the getState function, which are necessary to solve the formulas.

Note that NuXmv-IC3-Klive is able to solve more cases than Aalta_v2.0 in some benchmarks, such as “/lift” and “/schuppan/philt” in which unsatisfiable formulas are not handled well enough by Aalta_v2.0. Currently, Aalta_v2.0 requires large number of SAT calls to identify an unsatisfiable core. We plan to use a specialized MUS (minimal unsatisfiable core) solver in future work to address this challenge.

VI. SMT-BASED TEMPORAL REASONING

So far, we described how to use SAT solving for checking satisfiability of propositional LTL formulas. In many applications, however, we need to handle LTL formulas with assertion atoms, that is, atoms that are non-boolean state assertions, e.g., assertions about program variables. For example, Spin model checker uses temporal properties expressed in LTL using assertions about Promela state variables [13]. Existing explicit temporal-reasoning tools, e.g., SPOT [8], abstract such assertions as propositional atoms.

Recall that we utilize SAT solvers in our approach to compute assignments of formulas \( \phi^p \) (with \( \phi \) is in XNF). The states of transition system are then obtained from these assignments. When \( \phi \) is an assertional LTL formula, the formula \( \phi^p \) is not a propositional formula, but a Boolean combination of theory atoms, for an appropriate theory. Thus, our approach is still applicable, except that we need to replace the underlying SAT solver by an SMT solver.

Consider, for example the formula \( \phi = (F(k = 1) \land F(k = 2)) \). The XNF of \( \phi \), i.e. \( \text{xnf}(\phi) \), is \( \{v(F(k = 1)) \land (k = 1) \lor \neg v(F(k = 1)) \land XF(k = 1)\} \lor \{v(F(k = 2)) \land (k = 2) \lor \neg v(F(k = 2)) \land XF(k = 2)\} \). If we use a SAT solver, we can obtain an assignment such as \( A = \{(k = 1), v(F(k = 1)), \neg XF(k = 1), (k = 2), v(F(k = 2)), \neg XF(k = 2)\} \), which is consistent propositionally, but inconsistent theoretically. This can be avoided by using an SMT solver. Generally, for a formula \( \phi_n = \bigwedge_{1 \leq i \leq n} F(k = i) \), there are \( O(2^n) \) states generated in the transition system by the SAT-based approach, but only \( n \) states need to be generated. This can be achieved by replacing the SAT solver in our approach by an SMT solver. The performance gap between the SAT-based approach and the SMT-based approach would be exponential. Indeed, SPOT performance on the formulas \( \phi_n \) is exponential in \( n \).

As proof of concept, we checked satisfiability of the formulas \( \phi_n \), for \( n = 1, \ldots, 100 \), by Aalta_v2.0. We then replaced MiniSAT by Z3, a state-of-the-art SMT solver [24]. The performance results show indeed an exponential gap between the SAT-based approach and the SMT-based approach. (Of course, we also gain in correctness: the formula \( F(k = 1 \land k = 2) \) is satisfiable when considered propositionally, but unsatisfiable when considered assertionally.) Applying SMT-based techniques in other temporal-reasoning tasks, such as translating LTL to Büchi automata [11] or to runtime monitors [31], is a promising research direction.

VII. CONCLUDING REMARKS

We described in this paper a SAT-based framework for explicit LTL reasoning. As proof of concept, we implemented an LTL satisfiability solver, whose performance dominates all

Table I. Experimental results on the Schuppan-collected benchmark. Each cell lists a tuple \((t, n)\) where \(t\) is the total checking time (in seconds), and \(n\) is the total number of unsolved formulas.

| Formula type       | ls4  | TRP++ | NuXmv-BMCINC | Aalta_v1.2 | NuXmv-IC3-Klive | Aalta_v2.0 |
|-------------------|------|-------|--------------|------------|----------------|------------|
| /acacia/example    | 155  | 0     | 192          | 0          | 1              | 0          |
| /acacia/demo-v3    | 2668 | 32    | 2834         | 38         | 0              | 660        |
| /acacia/demo-v22   | 60   | 9     | 63           | 2          | 1              | 2          |
| /alaska/lift       | 13581| 247   | 15602        | 254        | 1919           | 26         |
| /alaska/szymanski  | 27   | 0     | 28           | 4          | 1              | 0          |
| /anzu/ambta        | 6120 | 102   | 6120         | 102        | 536            | 7          |
| /anzu/genbuf       | 7200 | 120   | 7200         | 120        | 782            | 11         |
| /rozier/countier   | 62   | 1     | 64           | 1          | 1855           | 44         |
| /rozier/formula   | 1175 | 29    | 1793         | 29         | 1058           | 15         |
| /rozier/philt      | 144  | 0     | 146          | 0          | 567            | 9          |
| /rozier/counter    | 448  | 0     | 495          | 1          | 2768           | 46         |
| /rozier/robots     | 3634 | 532   | 45739        | 735        | 3570           | 58         |
| /rozier/lift       | 18811| 256   | 19142        | 265        | 4049           | 67         |
| /rozier/countier   | 990  | 0     | 1303         | 0          | 1085           | 18         |
| Total             | 114321| 1675  | 163142       | 2428       | 24769          | 376        |

*Figure is provided in Appendix.*
similar tools. We also demonstrated that our approach can be extended from propositional LTL to assertional LTL, yielding exponential improvement in performance.

Extending the explicit SAT/SMT-based approach to other applications of LTL reasoning, such as translating LTL to automata and monitors is a promising research direction. Current best-of-breed translators, e.g., [8], are tableau-based, and the SAT/SMT approach may yield significant performance improvement.

Of course, the ultimate temporal-reasoning task is model checking. Explicit model checkers such as SPIN [14] start with a translation of LTL to Büchi automata, which are then used by the model-checking algorithm. An alternative approach is to construct the automaton on-the-fly using SAT/SMT techniques, using the framework developed here.

Furthermore, current symbolic model-checking tools, such as NuXmv, do rely heavily on SAT solvers to implement algorithms such as BMC [13] or IC3 [1]. The success of the SAT-based explicit LTL-reasoning approach for LTL satisfiability checking suggests that this approach may also be successful in SAT-based model checking. This remains a highly intriguing research possibility.

REFERENCES

[1] A. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, Verification, Model Checking, and Abstract Interpretation, volume 6538 of Lecture Notes in Computer Science, pages 70–87. Springer Berlin Heidelberg, 2011.
[2] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In CAV, pages 334–342, 2014.
[3] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In Gianpietro Cabodi and Satnam Singh, editors, FMCAD, pages 52–59. IEEE, 2012.
[4] E.M. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.
[5] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. Formal Methods in System Design, 1:275–288, 1992.
[6] M. d’Agostino. Tableau methods for classical propositional logic. In Handbook of tableau methods, pages 45–123. Springer, 1999.
[7] N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In Proc. 11th Int. Conf. on Computer Aided Verification, volume 1633 of Lecture Notes in Computer Science, pages 249–260. Springer, 1999.
[8] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In Proc. 12th Int’l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 76–83. IEEE Computer Society, 2004.
[9] N. Eén and N. Sörensson. An extensible SAT-solver. In SAT, pages 502–518, 2003.
[10] M. Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. Journal of Logic and Computation, 7(4):429–456, 1997.
[11] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembiski and M. Sredniawa, editors, Protocol Specification, Testing, and Verification, pages 3–18. Chapman & Hall, 1995.
[12] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedure - the case study of modal K. In Proc. 13th Int’l Conf. on Automated Deduction, volume 1104 of Lecture Notes in Computer Science, pages 583–597. Springer, 1996.
[13] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. Rajamani., editors, Computer Aided Verification, volume 3576 of Lecture Notes in Computer Science, pages 98–111. Springer Berlin Heidelberg, 2005.
[14] G.J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.
[15] U. Hustadt and B. Konev. Trp++ 2.0: A temporal resolution prover. In In Proc. CADE-19, LNAI, pages 274–278. Springer, 2003.
[16] T. Larabee. Test pattern generation using boolean satisfiability. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 11(1):4–15, 1992.
[17] J. Li, G. Pu, L. Zhang, M. Y. Vardi, and J. He. Fast LTL satisfiability checking by SAT solvers. CoRR, abs/1401.5677, 2014.
[18] J. Li, L. Zhang, G. Pu, M. Vardi, and J. He. LTL satisfiability checking revisited. In The 20th International Symposium on Temporal Representation and Reasoning, pages 91–98, 2013.
[19] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. Commun. ACM, 52(8):76–82, 2009.
[20] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, 1992.
[21] J. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In K. Sakallah and L. Simon, editors, Theory and Applications of Satisfiability Testing - SAT 2011, volume 6695 of Lecture Notes in Computer Science, pages 159–173. Springer Berlin Heidelberg, 2011.
[22] K. McMillan. Interpolation and SAT-based model checking. In J. Hunt, Warren A., and Fabio Somenzi, editors, Computer Aided Verification, volume 2725 of Lecture Notes in Computer Science, pages 1–13. Springer Berlin Heidelberg, 2003.
[23] K.L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.
[24] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
[25] A. Pnueli. The temporal logic of programs. In Proc. 18th IEEE Symp. on Foundations of Computer Science, pages 46–57, 1977.
[26] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. Int’l J. on Software Tools for Technology Transfer, 12(2):123–137, 2010.
[27] V. Schuppan and L. Darmawan. Evaluating LTL satisfiability solvers. In Proceedings of the 9th international conference on Automated technology for verification and analysis, AVTA’11, pages 397–413. Springer-Verlag, 2011.
[28] S. Schwon and J. Esparza. A note on on-the-fly verification algorithms. In Proc. 11th Int’l Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 3440, pages 174–190. Springer, 2005.
[29] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In Proc. 12th Int. Conf. on Computer Aided Verification, volume 1855 of Lecture Notes in Computer Science, pages 248–263. Springer, 2000.
[30] M. Suda and C. Weidenbach. A PLTL-prover based on labelled superposition with partial model guidance. In Automated Reasoning, volume 7364 of Lecture Notes in Computer Science, pages 537–543. Springer Berlin Heidelberg, 2012.
[31] D. Tabukov, K.Y. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. Formal Methods in System Design, 41(3):236–268, 2012.
[32] M. Vardi. On the complexity of epistemic reasoning. In Proceedings of the Fourth Annual Symposium on Logic in Computer Science, pages 243–252, Piscataway, NJ, USA, 1989. IEEE Press.
[33] M.Y. Vardi. United verification theory. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, Temporal Logic in Specification, volume 398, pages 202–212. Springer, 1989.
[34] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Proc. 1st IEEE Symp. on Logic in Computer Science, pages 332–344, 1986.
APPENDIX

A. Proof of Theorem 2

Proof: If A propositionally satisfies φ and ξ |= \( \bigwedge \ A \), then ξ |= φ, as φ ∈ A.

For the other direction, assume that ξ |= φ. Let A = \{θ ∈ cl(φ) : ξ |= θ\}. Clearly, φ ∈ A. It remains to prove that A is a propositional assignment, which we show by structural induction.

- For \( \ell \in L \) either ξ |= \( \ell \) or ξ \( \not\models \) \( \ell \), so either \( \ell \in A \) or \( \neg \ell \in A \).
- If \( ξ \models (θ_1 \land θ_2) \), then ξ |= θ_1 and ξ |= θ_2, so both θ_1 ∈ A and θ_2 ∈ A.
- If ξ |= (θ_1 ∨ θ_2), then ξ |= θ_1 or ξ |= θ_2, so either θ_1 ∈ A or θ_2 ∈ A.
- If ξ |= (θ_1 U θ_2), then either ξ |= θ_2, in which case, θ_2 ∈ A, or ξ |= θ_1 and ξ |= (X(θ_1 U θ_2)), in which case θ_1 ∈ A and (X(θ_1 U θ_2)) ∈ A.
- If ξ |= (θ_1 R θ_2), then ξ |= θ_2, in which case, θ_2 ∈ A, and either ξ |= θ_1 and ξ |= (X(θ_1 R θ_2)), in which case θ_1 ∈ A and (X(θ_1 R θ_2)) ∈ A.

B. Proof of Theorem 3

Proof: For the first claim, let r be \( s_0, s_1, \ldots \) and \( r_i = s_i, s_{i+1}, \ldots \) (i ≥ 0). Assume that no Until subformula is stuck at r. We prove by induction that trace(r_i) |= ψ for ψ ∈ s_i. It follows that trace(r) |= ψ.

- Trivially, for a literal \( \ell \in s_i \) we have that trace(r_i) |= \( \ell \).
- If (θ_1 ∧ θ_2) ∈ s_i, then θ_1 ∈ s_i and θ_2 ∈ s_i. By induction, trace(r_i) |= θ_1 and trace(r_i) |= θ_2, so trace(r_i) |= (θ_1 ∧ θ_2). The argument for (θ_1 ∨ θ_2) ∈ s_i is analogous.
- If (Xθ) ∈ s_i, then θ ∈ s_{i+1}. By induction, trace(r_{i+1}) |= θ, so trace(r_i) |= (Xθ).
- If (θ_1 U θ_2) ∈ s_i, then θ_2 ∈ s_i or both θ_1 ∈ s_i and (X(θ_1 U θ_2)) ∈ s_i, which implies that (θ_1 U θ_2) ∈ s_{i+1}. Since (θ_1 U θ_2) is not stuck at r, there is some k ≥ i such that \( θ_2 \in s_k \), and \( θ_1 \in s_j \) for \( i \leq j \leq k \). Using the induction hypothesis and the semantics of Until, it follows that trace(r_i) |= (θ_1 U θ_2).
- If (θ_1 R θ_2) ∈ s_i, then θ_2 ∈ s_i and either θ_1 ∈ s_i or (X(θ_1 R θ_2)) ∈ s_i, which implies that (θ_1 R θ_2) ∈ s_{i+1}. It is possible here for (θ_1 R θ_2) to be postponed forever. So for all k ≥ i, we have that either \( θ_2 \in s_j \) or there exists i ≤ j ≤ k such that \( θ_1 \in s_j \). Using the induction hypothesis and the semantics of Release, it follows that trace(r_i) |= (θ_1 R θ_2).

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

In the other direction, assume that ψ is satisfiable and there is an infinite trace ξ ∈ L^∞ such that ξ |= ψ. Let ξ = θ_0, θ_1, ..., and let \( ξ_i = \theta_i, \theta_{i+1}, \ldots \). As in the proof of Theorem 2 define \( A_i = \{ θ ∈ cl(φ) : ξ_i |= θ \} \). As in the proof of Theorem 2 each \( A_i \) is a propositional assignment for \( φ \), and, consequently a state of \( T_φ \). Furthermore, the semantics of Next implies that we have \( T(A_i, A_{i+1}) \) for i ≥ 0. Furthermore, the semantics of Until ensures that no Until is stuck in the run \( A_0, A_1, \ldots \).

C. Proof of Theorem 5

Proof: Suppose first that items 1) and 2) hold. Then the infinite sequence \( r' = s_0, s_1, \ldots, \) is an infinite run of \( T_φ \). It follows from Item 2) that no Until subformula is stuck at \( r' \). By Theorem 2 we have that \( r' \models φ \).

Suppose now that φ is satisfiable. By Theorem 2 there is an infinite run \( r' \) of \( T_φ \) in which no Until subformula is stuck. Let \( r' = s_0, s_1, \ldots \) be such a run. Each \( s_i (i \geq 0) \) is a state of \( T_φ \), and the number of states is bounded by \( 2^{111} \). Thus, there must be \( 0 \leq m < n \) such that \( s_m = s_n \). Let \( Q = \bigcup_{i=m}^n s_i \). Since no Until subformula can be stuck at r, if \( ψ = ψ_1 U ψ_2 \in Q \), then it is must be that \( v(ψ) \in Q \).

D. More experiments on LTL-Satisfiability Checking

This section shows more experimental results on LTL-satisfiability checking. Table 11 shows a complete version of Table 10 in which we explore the different performance of Aalta_v2.0 with/without heuristics. From the table we can learn that the heuristics described in this paper play a key role on the SAT-based LTL-satisfiability checking under our framework. Moreover we find that the performance of Aalta_v2.0 without heuristics can be even worse than its previous version Aalta_v1.2, which is a tableau-based solver. We can explain it via an example. Actually, if the formula is \( φ_1 \lor φ_2 \), the traditional tableau method splits the formula and at most creates two nodes. However under our pure SAT reasoning framework, it may create three nodes which contain only \( φ_1 \) or \( φ_2 \), or both of them. This shows that the space generated by SAT solvers may be in general larger than that generated by tableau expansion, and it affirms as well the importance of heuristics proposed in this paper.

In additional to the schuppan-collected benchmarks, we also run the random conjunction formulas, which is proposed in [19], by all testing solvers. A random conjunction formula \( RC(n) \) has the form of \( \bigwedge_{1 \leq i \leq n} P_i(v_1, v_2, \ldots, v_k) \), where \( n \) is the number of conjunctive elements and \( P_i(1 \leq i \leq n) \) is a randomly chosen pattern formula used frequently in practice. The motivation is that typical temporal assertions may be quite small in practice. And what makes the LTL satisfiability problem hard is that we need to check large conjunctions of small temporal formulas, so that we need to check that the conjunction of all input assertions is also satisfiable. In our experiment, the number of \( n \) varies from 1 to 30, and for each \( n \) a set of 100 formulas are randomly chosen. The experimental results are shown in Fig. 2. It shows that Aalta_v2.0 still performs best among tested solvers, and comparing to the second best solver NuXmv it achieves approximately the 30% speed-up.

E. Demonstration Results on SMT Reasoning

Fig. 3 shows the evaluation results for LTL satisfiability checking by leveraging SMT solver Z3. Obviously the figure shows a clear performance gap between the SAT-based and SMT-based approach on checking LTL formulas with assertional atoms.

\[^1\]http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml
TABLE II. EXPERIMENTAL RESULTS ON THE SCHUPPAN-COLLECTED BENCHMARK. EACH CELL LISTS A TUPLE \((t, n)\) WHERE \(t\) IS THE TOTAL CHECKING TIME (IN SECONDS), AND \(n\) IS THE TOTAL NUMBER OF UNSOLVED FORMULAS.

| Formula type            | ls4   | TRP++ | NuXmv-BMCINC | Aalta_v1.2 | NuXmv-IC3-Klive | Aalta_v2.0 without heuristics | Aalta_v2.0 with heuristics |
|-------------------------|-------|-------|--------------|------------|----------------|-----------------------------|--------------------------|
| /acacia/example         | 155   | 0     | 192          | 0          | 1              | 0                          | 1                        |
| /acacia/demo-v3         | 2668  | 32    | 2834         | 38         | 3              | 0                          | 0                        |
| /acacia/demo-v22        | 60    | 0     | 67           | 0          | 1              | 0                          | 0                        |
| /alaska/ls4             | 13381 | 227   | 15602        | 254        | 1919           | 26084                      | 63 567                   |
| /alaska/szymanski       | 27    | 0     | 283          | 4          | 1              | 0                          | 2                        |
| /alantur/ambtb          | 6120  | 102   | 6120         | 102        | 536            | 7                           | 2086 9062 8 4876 60 928 |
| /antr/ambient           | 7200  | 120   | 7200         | 120        | 782            | 11                          | 3343 1350 13 5243 94 827 |
| /router/counter         | 4934  | 62    | 4491         | 44         | 3865           | 64                          | 9238 60 3988 65 3328 35 2649 |
| /router/formulas        | 167   | 0     | 37533        | 523        | 1258           | 19                          | 1372 20 664 0 1672 25 363 |
| /router/pattern         | 14616 | 228   | 15450        | 237        | 1505           | 8                           | 8 3252 13 8 0 9 0          |
| /schuppan/01formula     | 2193  | 34    | 2778         | 35         | 14             | 0                           | 2 55 0 2 0 2 0            |
| /schuppan/02formula     | 2284  | 35    | 2566         | 41         | 1781           | 26                          | 2 742 7 2 0 2 0            |
| /schuppan/phltl         | 1771  | 27    | 1793         | 29         | 1538           | 15                          | 1253 247 153 1333 247 133 |
| /rip/N5x                | 144   | 0     | 146          | 0          | 567            | 9                           | 300 0 187 0 219 0 15 0    |
| /rip/N5y                | 448   | 0     | 495          | 1          | 2268           | 46                          | 116 0 102 0 316 0 16 0    |
| /rip/N12x               | 36345 | 532   | 45739        | 735        | 370            | 58                          | 768 0 705 0 768 0 175 0   |
| /rip/N12y               | 18811 | 256   | 19142        | 265        | 4049           | 67                          | 7413 60 979 0 7413 100 154 |
| /forobots               | 990   | 0     | 1303         | 0          | 1085           | 18                          | 2250 32 37 0 2130 40 524 |
| Total                   | 114321| 1675  | 163142       | 2428       | 24769          | 376                         | 28208 350 14261 120 31554 435 7668 |

Fig. 2. Results for LTL-satisfiability checking on Random Conjunction Formulas.
Fig. 3. Results for LTL-satisfiability checking on $\bigwedge_{1 \leq i \leq n} F(k = i)$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.9\textwidth]{figure3.png}
\caption{Results for LTL-satisfiability checking on $\bigwedge_{1 \leq i \leq n} F(k = i)$.}
\end{figure}