Learning Linear Temporal Properties

Daniel Neider
Max Planck Institute for Software Systems
67663 Kaiserslautern, Germany
Email: neider@mpi-sws.org

Ivan Gavran
Max Planck Institute for Software Systems
67663 Kaiserslautern, Germany
Email: gavran@mpi-sws.org

***Abstract*—We present two novel algorithms for learning formulas in Linear Temporal Logic (LTL) from examples. The first learning algorithm reduces the learning task to a series of satisfiability problems in propositional Boolean logic and produces a smallest LTL formula (in terms of the number of subformulas) that is consistent with the given data. Our second learning algorithm, on the other hand, combines the SAT-based learning algorithm with classical algorithms for learning decision trees. The result is a learning algorithm that scales to real-world scenarios with hundreds of examples, but can no longer guarantee to produce minimal consistent LTL formulas. We compare both learning algorithms and demonstrate their performance on a wide range of synthetic benchmarks. Additionally, we illustrate their usefulness on the task of understanding executions of a leader election protocol.**

I. Introduction

Making sense of the observed behavior of complex systems is an important problem in practice. It arises, for instance, in debugging (especially in the context of distributed systems), reverse engineering (e.g., of malware and viruses), specification mining for formal verification, and modernization of legacy systems, to name but a few examples. However, understanding a system based on examples of its execution is clearly a challenging task that can quickly become overwhelming without proper tool support.

In this paper, we address this problem and develop learning-based techniques to help engineers understand the dynamic (i.e., temporal) behavior of complex systems. More precisely, we solve the problem of learning formulas in Linear Temporal Logic (LTL) [1], which are meant to distinguish between desirable and undesirable executions of a system (e.g., to explain the root-cause of a bug). The particular choice of LTL in this work is motivated by two observations: first, logical formulas often provide concise descriptions of the observed behavior and are relatively easy for humans to comprehend; second, LTL—together with Computational Tree Logic (CTL) [2]—is widely considered to be the de facto standard for specifying temporal properties and, hence, many engineers are familiar with its use.

The precise problem we are aiming at is the following: given a sample $\mathcal{S}$ consisting of two finite sets of positive and negative examples, learn an LTL formula φ that is consistent with $\mathcal{S}$ in the sense that all positive examples satisfy φ, whereas all negative examples violate φ.[1] To be as general and succinct as possible, we here consider examples to be infinite, ultimately periodic words (e.g., traces of a non-terminating system) and assume the standard syntax of LTL. However, our techniques can easily be adapted to the case of finite words and extend smoothly to arbitrary future-time temporal operators, such as "release", "weak until", and so on. We fix all necessary definitions and notations in Section II.

The main contribution of this work are **two novel learning algorithms for LTL formulas from data**, one based on SAT solving, the other on learning decision trees.

SAT-based learning algorithm: The idea of our first algorithm, presented in Section III, is to reduce the problem of learning an LTL formula to a series of satisfiability problems in propositional Boolean logic and to use highly-optimized SAT solvers to search for solutions. Inspired by ideas from bounded model checking [10], our learning algorithm produces a series of propositional formulas $\Phi_n^{\mathcal{S}}$ for increasing values of $n \in \mathbb{N} \setminus \{0\}$ that depend on the sample $\mathcal{S}$ and have the following two properties: (1) $\Phi_n^{\mathcal{S}}$ is satisfiable if and only if there exists an LTL formula of size n (i.e., with n subformulas) that classifies the examples correctly, and (2) a model of $\Phi_n^{\mathcal{S}}$ contains sufficient information to construct such an LTL formula. By increasing the value of n until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, we obtain an effective algorithm that learns an LTL formula that is guaranteed to classify the examples correctly (given that the sample is non-contradictory).

By design, our SAT-based learning algorithm has three distinguished features, which we believe are essential in practice. First, our algorithm learns LTL formulas of minimal size (i.e., with the minimal number of subformulas). As we seek to learn formulas to be read by humans, the size of the learned formula is a crucial metric since larger formulas are generally harder to understand than smaller ones. Second, once an LTL formula has been learned, our algorithm can be queried for further, distinct formulas that are consistent with the sample. We believe that this feature is important in practice as it allows generating multiple explanations for the observed data. Third, our algorithm does not rely on an a priori given

[1]Note that, in contrast to classical computational learning theory [3] and modern statistical machine learning [4], [5], we seek to learn a formula that does not make mistakes on the examples. In fact, separation problems of this sort are of great interest in automata and formal language theory. Prominent examples in this area are the minimization of incompletely-specified state machines [6], [7] and Regular Model Checking [8], [9].

set of templates, which is in stark contrast to existing work on learning temporal properties (e.g., Bombara et al. [11]). To the best of our knowledge, our SAT-based algorithm is in fact the first learning algorithm that is not restricted to a fixed class of templates. However, restrictions to the shape of LTL formulas (e.g., to the popular GR(1)-fragment of LTL [12]) can easily be encoded if desired.

Learning algorithm based on decision trees: Our second learning algorithm, which we present in Section IV, trades in the guarantee of finding minimal solutions in order to attain better scalability. The key idea is to perform the learning in two phases. In the first phase, we run the SAT-based learning algorithm described above on various subsets of the examples. This results in a (small) number of LTL formulas, named "LTL primitives", that classify at least these subsets correctly. In the second phase we use a standard learning algorithm for decision trees [13] to learn a Boolean combination of these LTL primitives that classifies the whole set of examples correctly, though it might not be minimal. Note, however, that we need to carefully choose the subsets of examples such that the resulting LTL primitives (a) separate all pairs of positive and negative examples and (b) are general enough to permit "small" decision trees. We have experimented with numerous strategies to select subsets, but in this paper we present only the two that performed best. A well known advantage of decision trees is that they are simple to comprehend due to their rule-based structure.

In Section V, we evaluate the performance of both learning algorithms on a wide range of synthetic benchmarks that reflect typical patterns of LTL formulas used in practice. Additionally, we illustrate their usefulness for understanding causes of inconsistencies in the leader election used by Zookeeper's atomic broadcast protocol [14].

Details and proofs omitted due to space constraints can be found in an extended version of this paper [15].

Related Work

Learning of temporal properties from examples has recently attracted increasing interest, especially in the area of *Signal Temporal Logic (STL)* [16] and *parametric STL* [17]. Examples include the work by Asarin et al. [17], Kong et al. [18], [19], Vaidyanathan et al. [20], and Bartocci, Bortolussi, and Sanguinetti [21]. In contrast to our SAT-based learning algorithm, however, all of these techniques either rely on user-given templates or can only learn formulas from very restricted syntactic fragments. Various techniques for mining LTL specifications [22], [23] and CTL specifications [24] exist as well, but these also rely on templates or restrict the class of formulas severely. To the best of our knowledge, our SAT-based algorithm is in fact the first that is capable of learning unrestricted LTL formulas without relying on user-given templates. Nonetheless, expert knowledge in form of constraints on the syntax can easily be encoded if desired.

Our SAT-based learning algorithm is inspired by bounded model checking [10] and earlier work of the first author on learning (minimal) automata over finite words [7], [9]. However, since regular languages are strictly more expressive than LTL (the former being equivalent to monadic second-order logic [25], while the latter being equivalent to fist-order logic [26]), automata learning techniques—including active learning algorithms [27], [28] that operate in Angluin's active learning framework [29]—are not immediately applicable. However, lifting the methods developed in this work to an active learning setup, without a detour via automata, is part of our plans for future work.

Using decision trees to learn Signal Temporal Logic (STL) formulas has been explored by Bombara et al. [11], whose main contribution is an adaptation of the classical impurity measure to account for STL formulas. However, this work still requires user-defined STL primitives to be provided, which serve as the features for the decision tree learning algorithm. By contrast, our technique uses the SAT-based learning algorithm to infer LTL primitives fully automatically.

Learning of logical formulas has also been studied in the context of *probably approximately correct learning (PAC)* [3]. Grohe and Ritzert [30], for instance, considered learning of first-order definable concepts over structures of small degree. Subsequently, Grohe, Löding, and Ritzert [31] studied the learning of hypotheses definable using monadic second order logic on strings. Due to the fundamental differences between PAC learning and the learning model considered here (one being approximate and the other being exact), their techniques cannot easily be applied.

II. Preliminaries

In this section, we set up definitions and notations used throughout the paper.

Finite and Infinite Words: An *alphabet* Σ is a nonempty, finite set. The elements of this set are called *symbols*.

A *finite word* over an alphabet Σ is a sequence $u = a_0 \dots a_n$ of symbols $a_i \in \Sigma$, $i \in \{0, \dots, n\}$. The empty sequence is called *empty word* and written as ε. The length of a finite word u is denoted by $|u|$, where $|\varepsilon| = 0$. Moreover, Σ^* denotes the set of all finite words over the alphabet Σ, while $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of all non-empty words.

An *infinite word* over Σ is an infinite sequence $\alpha = a_0 a_1 \dots$ of symbols $a_i \in \Sigma$, $i \in \mathbb{N}$. We denote the i-th symbol of an infinite word α by $\alpha(i)$ and the infinite suffix starting at position j by $\alpha[j, \infty)$. Given $u \in \Sigma^+$, the infinite word $u^\omega = uu \ldots \in \Sigma^\omega$ is the infinite repetition of u. An infinite word α is called *ultimately periodic* if it is of the form $\alpha = uv^\omega$ for a $u \in \Sigma^*$ and $v \in \Sigma^+$. Finally, Σ^ω denotes the set of all infinite words over the alphabet Σ.

Propositional Boolean Logic: Let *Var* be a set of propositional variables, which take Boolean values from $\mathbb{B} = \{0, 1\}$ (0 representing *false* and 1 representing *true*). Formulas in *propositional (Boolean) logic*—which we denote by capital Greek letters—are inductively constructed as follows:

- each $x \in \mathit{Var}$ is a propositional formula; and
- if Ψ and Φ are propositional formulas, so are $\neg\Psi$ and $\Psi \vee \Phi$.

Moreover, we add syntactic sugar and allow the formulas $true$, $false$, $\Psi \wedge \Phi$, $\Psi \Rightarrow \Phi$, and $\Psi \Leftrightarrow \Phi$, which are defined as usual.

A *propositional valuation* is a mapping $v\colon Var \to \mathbb{B}$, which maps propositional variables to Boolean values. The semantics of propositional logic is given by a satisfaction relation $\models$ that is inductively defined as follows: $v \models x$ if and only if $v(x) = 1$, $v \models \neg\Psi$ if and only if $v \not\models \Psi$, and $v \models \Psi \vee \Phi$ if and only if $v \models \Psi$ or $v \models \Phi$. In the case that $v \models \Phi$, we say that v *satisfies* Φ and call it a *model* of Φ. A propositional formula Φ is *satisfiable* if there exists a model v of Φ. The *size* of a formula is the number of its subformulas (as defined in the usual way).

The satisfiability problem of propositional logic is the problem to decide whether a given formula is satisfiable. Although this problem is well-known to be NP-complete [32], modern SAT solvers implement optimized decision procedures that can check satisfiability of formulas with millions of variables [33]. Moreover, SAT solvers also return a model if the input-formula is satisfiable.

Linear Temporal Logic: Linear Temporal Logic (LTL) [1] is an extension of propositional Boolean logic with modalities that allow expressing temporal properties. Starting with a finite, nonempty set $\mathcal{P}$ of *atomic propositions*, formulas in LTL—which we denote by small Greek letters—are inductively defined as follows:

- each atomic proposition $p \in \mathcal{P}$ is an LTL formula;
- if ψ and φ are LTL formulas, so are $\neg\psi$, $\psi \vee \varphi$, $\mathsf{X}\,\psi$ ("next"), and $\psi\,\mathsf{U}\,\varphi$ ("until").

Again, we add syntactic sugar and allow the formulas $true \coloneqq p \vee \neg p$ for some $p \in \mathcal{P}$, $false \coloneqq \neg true$, as well as $\psi \wedge \varphi$ and $\psi \rightarrow \varphi$, which are defined as usual. Moreover, we allow the additional temporal formulas $\mathsf{F}\,\psi \coloneqq true\,\mathsf{U}\,\psi$ ("finally") and $\mathsf{G}\,\psi \coloneqq \neg\,\mathsf{F}\,\neg\psi$ ("globally"). The *size* of an LTL formula φ, which we denote by $|\varphi|$, is the number of its subformulas. Finally, let $\mathcal{C} = \{\wedge, \vee, \neg, \rightarrow, \mathsf{F}, \mathsf{G}, \mathsf{U}, \mathsf{X}\}$ be the set of LTL operators.

LTL formulas are interpreted over infinite words $\alpha \in (2^{\mathcal{P}})^\omega$, though there exist various semantics for LTL over finite words and our techniques smoothly extend to these situations. For the sake of a simpler presentation, we define the semantics of LTL in a slightly non-standard way by means of a *valuation function* V. This functions maps pairs of LTL formulas and infinite words to Boolean values and is inductively defined as follows: $V(p, \alpha) = 1$ if and only if $p \in \alpha(0)$, $V(\neg\varphi, \alpha) = 1 - V(\varphi, \alpha)$, $V(\varphi \vee \psi, \alpha) = \max\{V(\varphi, \alpha), V(\psi, \alpha)\}$, $V(\mathsf{X}\,\varphi, \alpha) = V(\varphi, \alpha[1, \infty))$, and $V(\varphi\,\mathsf{U}\,\psi, \alpha) = \max_{i \geq 0}\{\min\{V(\psi, \alpha[i, \infty)), \min_{0 \leq j < i}\{V(\varphi, \alpha[j, \infty))\}\}\}$. We call $V(\varphi, \alpha)$ the *valuation of* φ *on* α and say that α *satisfies* φ if $V(\varphi, \alpha) = 1$.

Our SAT-Based learning algorithm relies on a canonical syntactic representation of LTL formulas, which we call *syntax DAGs*. A syntax DAG is essentially a syntax tree (i.e., the unique tree labeled with atomic propositions as well as Boolean and temporal operators that is derived from the inductive definition of an LTL formula) in which common subformulas are shared. This sharing turns the syntax tree into a directed, acyclic graph (DAG), whose number of nodes coincides with the number of subformulas of the represented LTL formula. As an example, Figure 1b (on Page 4) depicts the (unique) syntax DAG of the formula $(p\,\mathsf{U}\,\mathsf{G}\,q) \vee (\mathsf{F}\,\mathsf{G}\,q)$, in which the subformula $\mathsf{G}\,q$ is shared; the corresponding syntax tree is depicted in Figure 1a. Note that syntactically distinct formulas have different (i.e., non-isomorphic) syntax DAGs.

Samples and Consistency: Throughout this paper, we assume that the data we learn from is given as two (potentially empty) finite, disjoint sets $P, N \subset (2^{\mathcal{P}})^\omega$ of ultimately periodic words. The words in P are interpreted as *positive examples*, while the words in N are interpreted as *negative examples*. We call the pair $\mathcal{S} = (P, N)$ a *sample*. Since we want to work with the ultimately periodic words in a sample algorithmically, we assume that they are stored as pairs (u, v) of finite words $u \in (2^{\mathcal{P}})^*$ and $v \in (2^{\mathcal{P}})^+$, which can be accessed individually. To measure the complexity of a sample, we define its *size* to be $|\mathcal{S}| = \sum_{uv^\omega \in P \cup N} |u| + |v|$.

Given an LTL formula φ and a sample $\mathcal{S} = (P, N)$, both over a set $\mathcal{P}$ of atomic propositions, we call φ *consistent* with $\mathcal{S}$ if $V(\varphi, uv^\omega) = 1$ for each $uv^\omega \in P$ (i.e., all positive examples satisfy φ) and $V(\varphi, uv^\omega) = 0$ for each $uv^\omega \in N$ (i.e., all negative examples do not satisfy φ); in this case, we also say that φ *separates* P and N. We call φ *minimally consistent with* $\mathcal{S}$ if φ is consistent with $\mathcal{S}$ and no consistent LTL formula of smaller size exists.

III. A SAT-based Learning Algorithm

The fundamental task we solve in this section is:

> *"given a sample $\mathcal{S}$, compute an LTL formula of minimal size that is consistent with $\mathcal{S}$".*

We call this task *passive learning of LTL formulas*—as opposed to active learning [29] where the learning algorithm is permitted to actively query for additional data. Note that this problem can have more than one solution as there can be multiple, non-equivalent LTL formulas that are minimally consistent with a given sample.

Before we explain our learning algorithm in detail, let us briefly comment on the minimality requirement in the definition above. On the one hand, we observe that the problem becomes simple if no restriction on the size is imposed: for $\alpha \in P$ and $\beta \in N$, construct a formula $\varphi_{\alpha,\beta}$ with $V(\varphi_{\alpha,\beta}, \alpha) = 1$ and $V(\varphi_{\alpha,\beta}, \beta) = 0$ that describes the first symbol where α and β differ using a sequence of X-operators and an appropriate propositional formula; then, $\bigvee_{\alpha \in P} \bigwedge_{\beta \in N} \varphi_{\alpha,\beta}$ is consistent with $\mathcal{S}$ since we assume P and N to be disjoint. However, simply characterizing all differences between positive and negative examples is clearly overfitting the sample and, hence, arguably of little help in practice. On the other hand, we believe that small formulas are easier for humans to comprehend than large ones, which justifies spending effort on learning a smallest formula. However, we do not impose any preference amongst minimal consistent formulas (which is an interesting topic for future work).

Let us now turn to describing our learning algorithm. Its underlying idea is to reduce the construction of a minimally

consistent LTL formula to a satisfiability problem in propositional logic and use a highly-optimized SAT solver to search for solutions. More precisely, given a sample $\mathcal{S}$ and a natural number $n \in \mathbb{N} \setminus \{0\}$, we construct a propositional formula $\Phi_n^{\mathcal{S}}$ of size polynomial in n and $|\mathcal{S}|$ that has the following two properties:

1) $\Phi_n^{\mathcal{S}}$ is satisfiable if and only if there exists an LTL formula of size n that is consistent with $\mathcal{S}$; and
2) if v is a model of $\Phi_n^{\mathcal{S}}$, then v contains sufficient information to construct an LTL formula ψ_v of size n that is consistent with $\mathcal{S}$.

By increasing the value of n by one and extracting an LTL formula ψ_v from a model v of $\Phi_n^{\mathcal{S}}$ as soon as it becomes satisfiable (indeed, any model is sufficient), we obtain an effective algorithm that learns an LTL formula of minimal size that is consistent with $\mathcal{S}$. This idea is shown in pseudo code as Algorithm 1. In fact, the existence of a trivial solution for the passive LTL learning task (as sketched at the beginning of this section) shows that Algorithm 1 is guaranteed to terminate, and the size of this solution provides an upper bound on the value of n.

Algorithm 1: SAT-based learning algorithm

Input: a sample $\mathcal{S}$

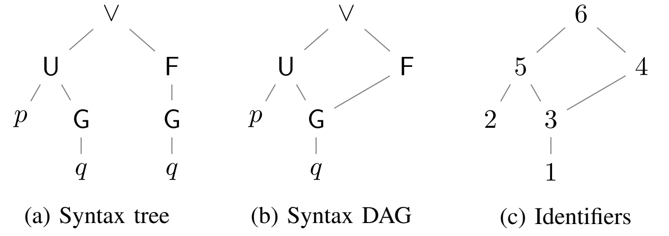
1 $n \leftarrow 0$;
2 **repeat**
3 $\quad n \leftarrow n + 1$;
4 $\quad$ Construct and solve $\Phi_n^{\mathcal{S}}$;
5 **until** $\Phi_n^{\mathcal{S}}$ *is satisfiable, say with model* v;
6 Construct and **return** ψ_v;

The key idea of the formula $\Phi_n^{\mathcal{S}}$ is to encode the syntax DAG of an (unknown) LTL formula $\varphi^\star$ with n subformulas and then constrain the variables of $\Phi_n^{\mathcal{S}}$ such that $\varphi^\star$ is consistent with the sample $\mathcal{S}$. To simplify our encoding, we assign to each node of this syntax DAG a unique *identifier* $i \in \{1, \ldots, n\}$ such that (a) the identifier of the root is n and (b) if the identifier of an inner node is i, then the identifiers of its children are less than i. Note that such a numbering scheme is not unique for a given syntax DAG, but it entails that the root always has identifier n and the node with identifier 1 is always labeled with an atomic proposition. We refer the reader to Figures 1b and 1c for an example.

We encode a syntax DAG using three types of propositional variables:

- $x_{i,\lambda}$ where $i \in \{1, \ldots, n\}$ and $\lambda \in \mathcal{P} \cup \mathcal{C}$;
- $l_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$; and
- $r_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$.

Intuitively, the variables $x_{i,\lambda}$ encode a labeling of the syntax DAG in the sense that if a variable $x_{i,\lambda}$ is set to *true*, then node i is labeled with λ (recall that each node is labeled with either an atomic proposition from $\mathcal{P}$ or an operator from $\mathcal{C}$). The variables $l_{i,j}$ and $r_{i,j}$, on the other hand, encode the structure of the syntax DAG (i.e., the left and/or right child of inner nodes): if variable $l_{i,j}$ ($r_{i,j}$) is set to *true*, then j is the identifier of the left (right) child of node i. By convention, we ignore the variables $r_{i,j}$ if node i of the syntax DAG is labeled with an unary operator; similarly, we ignore both $l_{i,j}$ and $r_{i,j}$ if node i is labeled with an atomic proposition. Note that in the case of $l_{i,j}$ and $r_{i,j}$, the identifier i ranges from 2 to n because node 1 is always labeled with an atomic proposition and, hence, cannot have children. Moreover, j ranges from 1 to $i-1$ to reflect the fact that identifier of children have to be smaller than the identifier of the current node.



(a) Syntax tree (b) Syntax DAG (c) Identifiers

Fig. 1: Syntax tree, syntax DAG, and identifiers of the syntax DAG for the LTL formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \lor (\mathsf{F}\, \mathsf{G}\, q)$

TABLE I: Constraints enforcing that the variables $x_{i,\lambda}$ encode a syntax DAG

$$\Big[\bigwedge_{1 \leq i \leq n} \bigvee_{\lambda \in \mathcal{P} \cup \mathcal{C}} x_{i,\lambda}\Big] \land \Big[\bigwedge_{1 \leq i \leq n} \bigwedge_{\lambda \neq \lambda' \in \mathcal{P} \cup \mathcal{C}} \lnot x_{i,\lambda} \lor \lnot x_{i,\lambda'}\Big] \quad (1)$$

$$\Big[\bigwedge_{2 \leq i \leq n} \bigvee_{1 \leq j < i} l_{i,j}\Big] \land \Big[\bigwedge_{2 \leq i \leq n} \bigwedge_{1 \leq j < j' < i} \lnot l_{i,j} \lor \lnot l_{i,j'}\Big] \quad (2)$$

$$\Big[\bigwedge_{2 \leq i \leq n} \bigvee_{1 \leq j < i} r_{i,j}\Big] \land \Big[\bigwedge_{2 \leq i \leq n} \bigwedge_{1 \leq j < j' < i} \lnot r_{i,j} \lor \lnot r_{i,j'}\Big] \quad (3)$$

$$\bigvee_{p \in \mathcal{P}} x_{1,p} \quad (4)$$

To enforce that the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ in fact encode a syntax DAG, we impose the constraints listed in Table I. Formula (1) ensures that each node is labeled with exactly one label. Similarly, Formulas (2) and (3) enforce that each node (except for node 1) has exactly one left and exactly one right child (although we ignore certain children if the node represents an unary operator or an atomic predicate). Finally, Formula (4) makes sure that node 1 is labeled with an atomic proposition.

Let Φ_n^{DAG} now be the conjunction of Formulas (1) to (4). Then, one can construct a syntax DAG from a model v of Φ_n^{DAG} in a straightforward manner: simply label node i with the unique label λ such that $v(x_{i,\lambda}) = 1$, designate node n as the root, and arrange the nodes of the DAG as uniquely described by $v(l_{i,j})$ and $v(r_{i,j})$. Moreover, we can easily derive an LTL formula from this syntax DAG, which we denote by ψ_v. Note, however, that ψ_v is not yet related to the sample $\mathcal{S}$ and, thus, might or might not be consistent with it.

To enforce that ψ_v is indeed consistent with $\mathcal{S}$, we now constrain the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ further. More precisely,

we add for each ultimately periodic word uv^ω in $\mathcal{S}$ a propositional formula $\Phi_n^{u,v}$ that tracks the valuation of the LTL formula encoded by Φ_n^{DAG} (and all its subformulas) on uv^ω. The observation that enables us to do this is the following.

Observation 1: Let $uv^\omega \in (2^{\mathcal{P}})^\omega$, ψ be an LTL formula over $\mathcal{P}$, and $k \in \mathbb{N}$. Then, $uv^\omega[|u|+k,\infty) = uv^\omega[|u|+m,\infty)$ with $m \equiv k \mod |v|$. In addition, $V(\varphi, uv^\omega[|u|+k,\infty)) = V(\varphi, uv^\omega[|u|+m,\infty))$ holds for every LTL formula φ.

Intuitively, Observation 1 states that the suffixes of a word uv^ω eventually repeat periodically. As a consequence, the valuation of an LTL formula on a word uv^ω can be determined based only on the finite prefix uv (recall that the semantics of temporal operators only depend on the suffixes of a word). To illustrate this claim, consider the LTL formula $\mathsf{X}\,\varphi$ and assume that we want to determine the valuation $V(\mathsf{X}\,\varphi, uv^\omega[|uv|-1,\infty))$ (i.e., $\mathsf{X}\,\varphi$ is evaluated at the end of the prefix uv). Then, Observation 1 permits us to compute this valuation based on $V(\varphi, uv^\omega[|u|,\infty))$, as opposed to the original semantics of the X-operator, which recurs to $V(\varphi, uv^\omega[|uv|,\infty))$ (i.e., the valuation at the next position). Note that similar, though more involved ideas can be applied to all other temporal operators.

Each formula $\Phi_n^{u,v}$ is built over an auxiliary set of propositional variables $y_{i,t}^{u,v}$ where $i \in \{1,\dots,n\}$ is a node in the syntax DAG and $t \in \{0,\dots,|uv|-1\}$ is a position in the finite word uv. The meaning of these variables is that the value of $y_{i,t}^{u,v}$ corresponds to the valuation $V(\varphi_i, uv^\omega[t,\infty))$ of the LTL subformula φ_i that is rooted at node i. Note that the set of variables for two distinct words from the sample must be disjoint.

To obtain this desired meaning of the variables $y_{i,t}^{u,v}$, we impose the constraints listed in Table II, which are inspired by bounded model checking [10]. Formula (5) implements the LTL semantics of atomic propositions and ensures that if node i is labeled with $p \in \mathcal{P}$, then $y_{i,t}^{u,v}$ is set to 1 if and only if $p \in uv(t)$. Next, Formulas (6) and (7) implement the semantics of negation and disjunction, respectively: if node i is labeled with $\neg$ and node j is its left child, then $y_{i,t}^{u,v}$ is the negation of $y_{j,t}^{u,v}$; on the other hand, if node i is labeled with $\vee$, node j is its left child, and node j' is its right child, then $y_{i,t}^{u,v}$ is the disjunction of $y_{j,t}^{u,v}$ and $y_{j',t}^{u,v}$. Moreover, Formula (8) implements the semantics of the X-operator, following the idea of "returning to the beginning of the periodic part v" as sketched above. Finally, Formula (9) implements the semantics of the U-operator. More precisely, the first conjunction in the consequent covers the positions $t \in \{0,\dots,|u|-1\}$ in the initial part u, while the second conjunct covers the positions $t \in \{|u|,\dots,|uv|-1\}$ in the periodic part v. Thereby, the second conjunct relies on an auxiliary set $t \looparrowright_{u,v} t'$ defined by

$$t \looparrowright_{u,v} t' := \begin{cases} \{t,\dots,t'-1\} & \text{if } t < t'; \\ \{|u|,\dots,t'-1,t,\dots,|uv|-1\} & \text{if } t \geq t', \end{cases}$$

which contains all positions in v "between t and t'". To avoid cluttering this section too much, we have omitted the description of the missing operators $\wedge$, $\rightarrow$, F, G and the constants *true* and *false*, which are implemented analogously. Moreover, our SAT encoding is extensible, and additional LTL operators such as weak until or weak and strong release can easily be added.

TABLE II: Constraints enforcing that the variables $y_{i,t}^{u,v}$ track the valuation of the prospective LTL formula on ultimately periodic words

$$\bigwedge_{1\leq i\leq n}\bigwedge_{p\in\mathcal{P}} x_{i,p} \rightarrow \Big[\bigwedge_{0\leq t<|uv|} \begin{cases} y_{i,t}^{u,v} & \text{if } p \in uv(t) \\ \neg y_{i,t}^{u,v} & \text{if } p \notin uv(t)\end{cases}\Big] \quad (5)$$

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j<i}} (x_{i,\neg}\wedge l_{i,j}) \rightarrow \bigwedge_{0\leq t<|uv|}\Big[y_{i,t}^{u,v} \leftrightarrow \neg y_{j,t}^{u,v}\Big] \quad (6)$$

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j,j'<i}} (x_{i,\vee}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow \bigwedge_{0\leq t<|uv|}\Big[y_{i,t}^{u,v} \leftrightarrow (y_{j,t}^{u,v}\vee y_{j',t}^{u,v})\Big] \quad (7)$$

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j<i}} (x_{i,\mathsf{X}}\wedge l_{i,j}) \rightarrow \Big[\Big[\bigwedge_{0\leq t<|uv|-1} y_{i,t}^{u,v}\leftrightarrow y_{j,t+1}^{u,v}\Big]\wedge\Big[y_{i,|uv|-1}^{u,v}\leftrightarrow y_{j,|u|}^{u,v}\Big]\Big] \quad (8)$$

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j,j'<i}} (x_{i,\mathsf{U}}\wedge l_{i,j}\wedge r_{i,j'}) \rightarrow \Big[\Big[\bigwedge_{0\leq t<|u|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{t\leq t'<|uv|}\Big[y_{j',t'}^{u,v}\wedge\bigwedge_{t\leq t''<t'} y_{j,t''}^{u,v}\Big]\Big]\wedge \Big[\bigwedge_{|u|\leq t<|uv|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{|u|\leq t'<|uv|}\Big[y_{j',t'}^{u,v}\wedge\bigwedge_{t''\in t\looparrowright_{u,v}t'} y_{j,t''}^{u,v}\Big]\Big]\Big] \quad (9)$$

For each $uv^\omega \in P\cup N$, let $\Phi_n^{u,v}$ now be the conjunction of Formulas (5) to (9). Then, we define

$$\Phi_n^{\mathcal{S}} := \Phi_n^{DAG}\wedge\Big[\bigwedge_{uv^\omega\in P}\Phi_n^{u,v}\wedge y_{n,0}^{u,v}\Big]\wedge\Big[\bigwedge_{uv^\omega\in N}\Phi_n^{u,v}\wedge\neg y_{n,0}^{u,v}\Big].$$

Note that the subformula $\Phi_n^{u,v}\wedge y_{n,0}^{u,v}$ makes sure that $uv^\omega\in P$ satisfies the prospective LTL formula (more concretely, uv^ω starting from position 0 satisfies the LTL formula at the root of the syntax DAG), while $\Phi_n^{u,v}\wedge\neg y_{n,0}^{u,v}$ ensures that $uv^\omega\in N$ does not satisfy it.

To prove the correctness of our learning algorithm, we first establish that the formula $\Phi_n^{\mathcal{S}}$ has in fact the desired properties.

Lemma 1: Let $\mathcal{S} = (P,N)$ be a sample, $n\in\mathbb{N}\setminus\{0\}$, and $\Phi_n^{\mathcal{S}}$ the propositional formula defined above. Then, the following holds:

1) If an LTL formula of size n that is consistent with $\mathcal{S}$ exists, then the propositional formula $\Phi_n^{\mathcal{S}}$ is satisfiable.
2) If $v \models \Phi_n^{\mathcal{S}}$, then ψ_v is an LTL formula of size n that is consistent with $\mathcal{S}$.

Termination and correctness of Algorithm 1 then follow from Lemma 1.

Theorem 1: Given a sample $\mathcal{S}$, Algorithm 1 terminates eventually and outputs an LTL formula of minimal size that is consistent with $\mathcal{S}$.

Proof: Since there exists a consistent LTL formula for every non-contradictory sample, Part 1 of Lemma 1 guarantees

that Algorithm 1 terminates. Moreover, Part 2 ensures that the output is indeed an LTL formula that is consistent with $\mathcal{S}$. Since n is increased by one in every iteration of the loop until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, the output of Algorithm 1 is a consistent LTL formula of minimal size. □

It is important to emphasize that the size of $\Phi_n^{\mathcal{S}}$ and, hence, the performance of Algorithm 1 depends on the size of a sample $\mathcal{S} = (P, N)$, as summarized next.

Remark 1: The formula $\Phi_n^{\mathcal{S}}$ ranges over $\mathcal{O}(n^2 + n|\mathcal{S}|)$ variables and is of size $\mathcal{O}(n^2 + n^3 \sum_{uv^\omega \in P \cup N} |uv|^3)$.

Finally. we conclude this section with a remark on incorporating expert knowledge into the learning process.

Remark 2: By adding constraints to the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$, one can easily incorporate expert knowledge (e.g., syntactic templates) into the learning process.

IV. A Decision Tree Based Learning Algorithm

The SAT-based algorithm described in Section III is an elegant, out-of-the-box way to discover minimal LTL formulas describing a sample. Even though it scales well beyond toy examples, its running time seems too prohibitive for real-world examples (as discussed in Section V). That is why we now present a learning algorithm based on a combination of SAT solving and decision tree learning.

Our second algorithm proceeds in two phases, outlined in Algorithm 2. In the first phase, we run Algorithm 1 on small subsets of P and N. This is repeated until we obtain a set Π of LTL formulas (we call them *LTL primitives*) that separate all pairs of words from P and N. In the second phase, formulas from Π are used as features for a standard decision tree learning algorithm [13]. The resulting decision tree is a Boolean combination of LTL formulas $\varphi_i \in \Pi$ that is consistent with the sample.

Algorithm 2: Learning algorithm based on decision trees

Input: a sample $\mathcal{S}$

1. Run Algorithm 1 on small subsets of P and N to construct a set $\Pi = \{\varphi_1, \dots, \varphi_n\}$ of LTL formulas such that for each pair $u_1 v_1^\omega \in P$ and $u_2 v_2^\omega \in N$ there exists a $\varphi_i \in \Pi$ with $V(\varphi_i, u_1 v_1^\omega) = 1$ and $V(\varphi_i, u_2 v_2^\omega) = 0$;
2. Learn a decision tree t with LTL primitives from Π as features and **return** the resulting Boolean combination ψ_t of LTL primitives (which is consistent with $\mathcal{S}$);

Note that this relaxes the problem addressed in Section III: we can no longer guarantee finding a formula of minimal size. However, decision trees are among the structures that are the easiest to interpret by end-users. That makes them suitable for our use-case, and the minimality of formulas is replaced by structural simplicity of decision trees.

Learning Decision Trees: We assume familiarity with decision tree learning and refer the reader to a standard textbook for further details [5]. As illustrated in Figure 3, the decision trees we seek to learn are tree-shaped structures whose inner nodes are labeled with LTL formulas from Π and whose leaves are labeled with either *true* or *false*. The LTL formula represented by such a tree t is given by $\psi_t \coloneqq \bigvee_{\rho \in \mathfrak{P}} \bigwedge_{\varphi \in \rho} \varphi$ where $\mathfrak{P}$ is the set of all paths from the root to a leaf labeled with *true* and $\varphi \in \rho$ denotes that φ occurs on ρ (negated if the path follows a dashed edge).

To learn a decision tree over LTL primitives, we perform a preprocessing step and modify the sample as follows. For each word $uv^\omega \in P \cup N$, we use the LTL primitives as features and create a Boolean vector of size $|\Pi|$ with the i-th entry set to $V(\varphi_i, uv^\omega)$; this vector is then labeled with *true* if $uv^\omega \in P$ or with *false* if $uv^\omega \in N$. In the second step, we apply a standard learning algorithm for decision trees to this modified sample (we used Gini impurity [34] as split heuristic in our experiments). Since we are interested in a tree that classifies our sample correctly, we disable heuristics such as pruning.

Obtaining LTL Primitives: Meaningful features are essential for a successful classification using decision trees. In our algorithm, features are generated from the set of LTL primitives Π. We used two different strategies, called Strategy α and Strategy β, for obtaining Π.

Strategy α iteratively chooses subsets $P' \subset P$ and $N' \subset N$ of size k according to probability distributions prob_P and prob_N on P and N, respectively. After a formula φ separating P' and N' is found using Algorithm 1 and added to Π, prob_P and prob_N are updated to increase the likelihood of any word that is not yet classified correctly by any of the $\varphi \in \Pi$ to be selected. This process is repeated until all pairs of positive and negative examples are separated by some LTL primitive or restarted after a user-given number of iterations. Although this strategy is, in general, not guaranteed to terminate due to its probabilistic nature, it always did in our experiments.

Strategy β computes LTL primitives in a more aggressive way. Starting with the set $S = P \times N$, it uniformly at random selects k pairs from S and uses Algorithm 1 to compute an LTL primitive φ that separates those pairs. Then, it removes all pairs separated by φ from S and repeats the process until S becomes empty (i.e., all pairs of examples are separated).

We refer to the extended version of this paper [35] for a detailed explanation of both strategies.

Correctness: The correctness of Algorithm 2 is formalized below.

Theorem 2: Given a sample $\mathcal{S}$, Algorithm 2 learns a (not necessarily minimal) formula ψ_t that is consistent with $\mathcal{S}$.

Theorem 2 follows from the fact that Step 1 of Algorithm 2 constructs a set of LTL primitives that allows separating any pair of positive and negative examples. Once such a set is constructed, any decision tree learner produces a decision tree t that is guaranteed to classify the examples correctly. The resulting LTL formula ψ_t, hence, is consistent with $\mathcal{S}$.

V. Evaluation

In this section, we answer questions that arise naturally: how performant is Algorithm 1 and what is the performance gain of Algorithm 2. Furthermore, what is the complexity of

TABLE III: Common LTL patterns used in practice [37]

Absence	Existence	Universality
$\mathsf{G}(\neg p_0)$	$\mathsf{F}(p_0)$	$\mathsf{G}(p_0)$
$\mathsf{F}(p_1) \rightarrow (\neg p_0 \,\mathsf{U}\, p_1)$	$\mathsf{G}(\neg p_0 \vee \mathsf{F}(p_0 \wedge \mathsf{F}(p_1)))$	$\mathsf{F}(p_1) \rightarrow (p_0 \,\mathsf{U}\, p_1)$
$\mathsf{G}(p_1 \rightarrow \mathsf{G}(\neg p_0))$	$\mathsf{G}(p_0 \wedge (\neg p_1 \rightarrow (\neg p_1 \,\mathsf{U}(p_2 \wedge \neg p_1))))$	$\mathsf{G}(p_1 \rightarrow \mathsf{G}(p_0))$

the learned decision trees in terms of the number of decision nodes, and, finally, how do different parameters influence the performance of Algorithm 2. After answering these questions with experiments performed on synthetic data, we demonstrate the usefulness of our algorithms for understanding executions of a leader-election algorithm.

We implemented both learning algorithms in a Python tool[2] using Microsoft Z3 [36]. All experiments were conducted on Debian machines with Intel Xeon E7-8857 CPUs at 3 GHz, using up to 5 GB of RAM.

Performance on Synthetic Data: To simulate real-world use-cases, we generated samples based on common LTL patterns [37], which are shown in Table III. Starting from a pattern formula ψ, we generated sets of random words and separated them into P and N depending on whether they are a model of ψ or not. Thereby, we fixed $|u| + |v| = 10$ for all words in the sample and added noise in form of one additional atomic proposition that is not constrained by the pattern formula. The size of the generated samples ranges between 50 and 5000. In total, we generated 192 samples.

Figure 2 compares the running times of Algorithm 1 and Algorithm 2 (using Strategy α and $k = 3$) on samples of varying sizes. (So as not to clutter the presentation too much, we selected four LTL patterns that showed a typical behavior of our learning algorithms. The complete results are available in the technical report [35].) Overall, Algorithm 1 produces minimal formulas consistent with a sample. It does so even for samples of considerable size, but if the sample size grows beyond 2000 (varies over samples), the SAT-based learner (Algorithm 1) frequently times out. When Algorithm 2 (using decision tree learning) is applied to these samples—as shown on the right-hand-side of Figure 2—none of the computations timed out and the running times significantly improved.

What kind of trees does Algorithm 2 produce? An example output of the algorithm is shown in Figure 3. Moreover, as Table IV illustrates, Algorithm 2 learns small trees, often with less than five inner nodes. Upon closer inspection, we noticed that it often happens that one of the LTL primitives was the specified formula itself. This suggests that small subsets already characterize our samples completely.

To be able to compare decision trees to the formulas learned by Algorithm 2, we measure the *size* of a tree t in terms of the size of the formula ψ_t this tree encodes. In our experiments, the formulas learned by Algorithm 2 were on average 1.41 times larger than those learned by Algorithm 1. However, there are outlier trees that are four times bigger than the one learned by Algorithm 1. Nonetheless, about 70 % are of the same size. Even for the outliers, as emphasized previously, the readability

[2]Our tool is publicly available at https://github.com/gergia/samples2LTL.

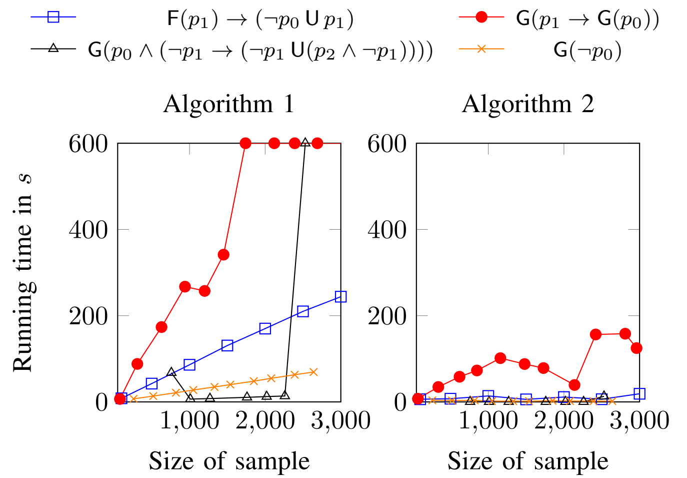
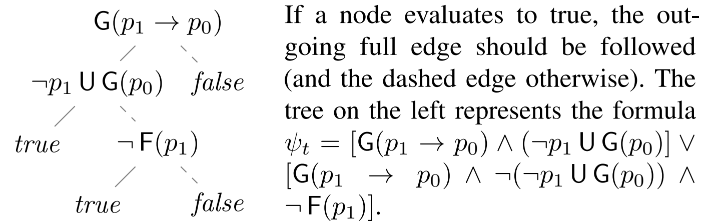


Fig. 2: Comparison of Algorithm 1 and Algorithm 2



If a node evaluates to true, the outgoing full edge should be followed (and the dashed edge otherwise). The tree on the left represents the formula $\psi_t = [\mathsf{G}(p_1 \rightarrow p_0) \wedge (\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)] \vee [\mathsf{G}(p_1 \rightarrow p_0) \wedge \neg(\neg p_1 \,\mathsf{U}\, \mathsf{G}(p_0)) \wedge \neg\, \mathsf{F}(p_1)]$.

Fig. 3: A decision tree obtained from a sample generated from the LTL pattern $\mathsf{G}(p_1 \rightarrow \mathsf{G}(p_0))$

does not degrade completely because the rule-based structure of decision trees is known to be easily understandable by humans. Note that the runtime and size of decision trees depends on the parameters of Algorithm 2, which we discuss next.

Tuning the Decision Tree-Based Algorithm: As described in Section IV, Algorithm 2 can be tuned by various parameters (sampling strategy for obtaining LTL primitives, size of sample subsets, probability increase rate, and number of repetitions inside a single sampling). In this subsection, we explore how those parameters affect the performance of the algorithm.

TABLE IV: Different parameters used for Algorithm 2

Sampling strategy	Subset size k	Number of timeouts	Avg. running time in s	Avg. number of nodes in a tree
α	3	0 / 192	21.00	3.05
α	6	4 / 192	35.28	1.47
α	10	8 / 192	42.72	1.2
β	3	4 / 192	30.92	1.37
β	6	12 / 192	48.46	1.19
β	10	21 / 192	48.11	1.06

Table IV shows the performance of Algorithm 2 for different parameters, averaged over all 192 benchmarks. As the table indicates, the less aggressive method of separating sets, Strategy α, performs better. It seems that if the subset sizes are increased, or Strategy β is used, the sampled subsets already describe the specified formula completely. Finally, we chose Strategy α and $k = 3$ to be our default parameters. Varying the probability decrease rate and the number of repetitions inside

a single sampling did not influence the performance much.

Explaining Executions of a Leader Election Protocol: A number of methods exist for finding errors or reproducing certain behavior in distributed systems through systematic testing [38], [39]. However, finding an execution and a corresponding schedule is only a first step towards understanding an issue. In the following, we demonstrate how to apply our technique in order to obtain a minimal LTL description of a specific inconsistency in a leader election protocol.

The leader election protocol we consider is the *Fast Leader Election* algorithm [14], [40] used by Apache Zookeeper. In this protocol, every node has a unique ID and initially tries to become the leader. To this end, every node sends messages to all other nodes proclaiming its leadership. Upon receiving a message by an aspirant leader with a higher ID, a node gives up its claim and acknowledges its support for the aspirant. If a node learns that an aspirant node has a support of a majority of all nodes, it commits (after waiting for a constant time for new messages) to the aspirant as the leader. Once committed, the node never again changes its decision and informs any other node of its commitment (one example is the message depicted by the dotted arrow in Figure 5). If a node has not committed and learns about another node that has committed, it commits to the same leader.

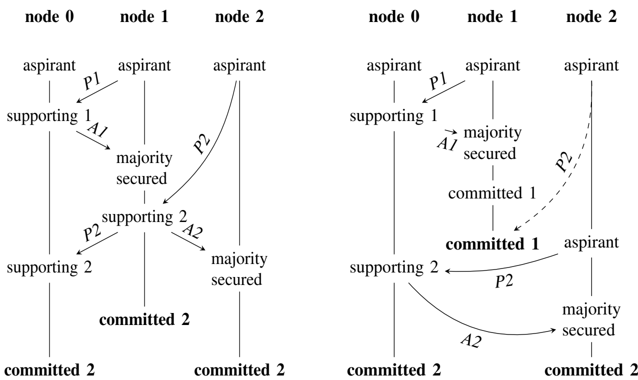


Fig. 4: Consistent schedule for an execution of the leader election protocol

Fig. 5: Inconsistent schedule for an execution of the leader election protocol

Figure 4 shows an example of a successful leader election with three nodes in an UML-style message sequence chart. The messages exchanged between nodes are proposing the leader i (P_i) and node j acknowledging the claim of a leader (A_j). The arrows indicate exchanged messages and imply a precedence of events. Note that not all messages are shown in the figures, but only the ones important for understanding the protocol.

In Figure 4 all the nodes have committed to the same leader. On the other hand, Figure 5 shows a schedule that ends up in an inconsistent state where nodes committed to different leaders. This schedule was discovered by the PCTCP algorithm [41], which systematically explores the space of possible executions of distributed algorithms. The situation in Figure 5 is caused by the asynchronous communication: for performance reasons, nodes commit as quickly as possible and then discard any messages, which otherwise would have changed their commitment (indicated as a dashed line in Figure 5). Note, however, that this is not a bug in Zookeeper's broadcast algorithm, as a leader without a quorum will not be allowed to perform any action in the later phase.

To better understand how this inconsistent state arises, our goal is to generate an LTL formula that describes the difference between the schedules in Figures 4 and 5. To this end, we constructed a sample by generating 20 linearizations of the schedule from Figure 4 and 20 linearizations of the schedule from Figure 5. Since we seek an explanation for the inconsistent behavior, the former (with consistent outcomes) correspond to negative examples (set N), and the latter (with inconsistent outcomes) correspond to positive examples (set P). The set of atomic propositions used to construct the examples contains twelve elements: $\mathit{recv}(i, j)$ for $i, j \in \{1, 2, 3\}$ (meaning that node j received a message from node i) and $\mathit{comm}(i)$ for $i \in \{1, 2, 3\}$ (meaning that node i committed to a leader).[3]

Finally, we ran Algorithm 1 on this sample. The result was the formula $\neg \mathit{recv}(2, 1) \, \mathsf{U} \, \mathit{comm}(1)$. Intuitively, node 1 did not receive a message from node 2 before it committed to a leader. That is exactly the difference between the schedules in Figures 4 and 5. Also, it hints at a specific reason for the inconsistency in Figure 5, thus potentially helping the engineers improve the system. Note, however, that this experiment still required a significant amount of manual effort. In order to apply the technique in practice, more automation is needed.

Summary: Algorithm 2 significantly improves upon the performance of Algorithm 1, though with a small increase in the size of the formula. The original motivation of getting readable explanations for the behavior of a system is preserved due to the fact that decision-trees are easy to comprehend. Algorithm 2 works the best using Strategy α and subsets of size $k = 3$. Finally, our techniques are able to give interesting insight into real-world systems.

VI. Conclusion

We have presented two novel algorithms for learning LTL formulas from examples. Our first algorithm is based on SAT solving, while the second algorithm extends the first with techniques for learning decision trees. We have shown that both algorithms are able to learn LTL formulas for a comprehensive set of benchmarks that we have derived from common LTL patterns. Moreover, we have demonstrated how our methods can help understand distributed algorithms.

Interesting directions of future work include the integration of LTL past-time operators, lifting our techniques to an active learning setup [29], as well as the development of similar learning algorithms for CTL. Furthermore, we plan to investigate the use of maximum-margin classifiers, such as support vector machines. To this end, one needs to develop a notion of distance between temporal formulas and words, which is clearly of independent, theoretical interest as well.

[3] While we could have included more information into propositions, we had to obscure some in order to avoid "stating the obvious" of the form "node 1 committed to node 1 as a leader, while node 2 committed to node 2".

REFERENCES

[1] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[2] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs*, ser. Lecture Notes in Computer Science, vol. 131. Springer, 1981, pp. 52–71.

[3] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984. [Online]. Available: http://doi.acm.org/10.1145/1968.1972

[4] A. Blum, J. Hopcroft, and R. Kannan, *Foundations of Data Science*, January 2018. [Online]. Available: https://www.cs.cornell.edu/jeh/book.pdf

[5] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997. [Online]. Available: http://www.worldcat.org/oclc/61321007

[6] C. P. Pfleeger, "State reduction in incompletely specified finite-state machines," *IEEE Trans. Computers*, vol. 22, no. 12, pp. 1099–1102, 1973. [Online]. Available: https://doi.org/10.1109/T-C.1973.223655

[7] D. Neider, "Computing minimal separating dfas and regular invariants using SAT and SMT solvers," in *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7561. Springer, 2012, pp. 354–369.

[8] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, "Regular model checking," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 403–418. [Online]. Available: https://doi.org/10.1007/10722167_31

[9] D. Neider and N. Jansen, "Regular model checking using solver technologies and automata learning," in *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7871. Springer, 2013, pp. 16–31. [Online]. Available: https://doi.org/10.1007/978-3-642-38088-4_2

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003. [Online]. Available: https://doi.org/10.1016/S0065-2458(03)58003-2

[11] G. Bombara, C. I. Vasile, F. Penedo, H. Yasuoka, and C. Belta, "A decision tree approach to data classification using signal temporal logic," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*. ACM, 2016, pp. 1–10.

[12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012. [Online]. Available: https://doi.org/10.1016/j.jcss.2011.08.007

[13] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, 2011, pp. 245–256. [Online]. Available: https://doi.org/10.1109/DSN.2011.5958223

[15] D. Neider and I. Gavran, "Learning linear temporal properties," *CoRR*, vol. abs/1806.03953, 2018. [Online]. Available: http://arxiv.org/abs/1806.03953

[16] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3253. Springer, 2004, pp. 152–166. [Online]. Available: https://doi.org/10.1007/978-3-540-30206-3_12

[17] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric identification of temporal properties," in *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7186. Springer, 2011, pp. 147–160. [Online]. Available: https://doi.org/10.1007/978-3-642-29860-8_12

[18] Z. Kong, A. Jones, A. M. Ayala, E. A. Gol, and C. Belta, "Temporal logic inference for classification and prediction from data," in *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*. ACM, 2014, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2562059.2562146

[19] Z. Kong, A. Jones, and C. Belta, "Temporal logics for learning and detection of anomalous behavior," *IEEE Trans. Automat. Contr.*, vol. 62, no. 3, pp. 1210–1222, 2017. [Online]. Available: https://doi.org/10.1109/TAC.2016.2585083

[20] P. Vaidyanathan, R. Ivison, G. Bombara, N. A. DeLateur, R. Weiss, D. Densmore, and C. Belta, "Grid-based temporal logic inference," in *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*, 2017, pp. 5354–5359. [Online]. Available: https://doi.org/10.1109/CDC.2017.8264452

[21] E. Bartocci, L. Bortolussi, and G. Sanguinetti, "Learning temporal logical properties discriminating ECG models of cardiac arrhytmias," *CoRR*, vol. abs/1312.7523, 2013. [Online]. Available: http://arxiv.org/abs/1312.7523

[22] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. IEEE, 2011, pp. 43–50. [Online]. Available: https://doi.org/10.1109/MEMCOD.2011.5970509

[23] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 81–92. [Online]. Available: https://doi.org/10.1109/ASE.2015.71

[24] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Autom. Softw. Eng.*, vol. 18, no. 3-4, pp. 263–292, 2011. [Online]. Available: https://doi.org/10.1007/s10515-011-0084-1

[25] J. R. Büchi, "On a decision method in restricted second-order arithmetic," in *Int. Congr. for Logic, Methodology and Philosophy of Science*. Stanford Univ. Press, 1962, pp. 1–11.

[26] A. W. Kamp, "Tense logic and the theory of linear order," Ph.D. dissertation, University of California, Los Angeles, 1968.

[27] A. Farzan, Y. Chen, E. M. Clarke, Y. Tsay, and B. Wang, "Extending automated compositional verification to the full class of omega-regular languages," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 2–17.

[28] D. Angluin and D. Fisman, "Learning regular omega languages," *Theor. Comput. Sci.*, vol. 650, pp. 57–72, 2016.

[29] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[30] M. Grohe and M. Ritzert, "Learning first-order definable concepts over structures of small degree," in *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. [Online]. Available: https://doi.org/10.1109/LICS.2017.8005080

[31] M. Grohe, C. Löding, and M. Ritzert, "Learning mso-definable hypotheses on strings," in *International Conference on Algorithmic Learning Theory, ALT 2017, 15-17 October 2017, Kyoto University, Kyoto, Japan*, ser. Proceedings of Machine Learning Research, vol. 76. PMLR, 2017, pp. 434–451. [Online]. Available: http://proceedings.mlr.press/v76/grohe17a.html

[32] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[33] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017, pp. 5061–5063.

[34] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees. Wadsworth, 1984*. Routledge, 1993.

[35] D. Neider and I. Gavran, "Learning linear temporal properties," *CoRR*, vol. abs/1806.03953, 2018. [Online]. Available: http://arxiv.org/abs/1806.03953

[36] 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, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[37] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the Second Workshop on Formal Methods in Software Practice*, ser. FMSP '98. New York, NY, USA: ACM, 1998, pp. 7–15. [Online]. Available: http://doi.acm.org/10.1145/298595.298598

[38] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, 2010, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736040

[39] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?" *PACMPL*, vol. 2, no. POPL, pp. 46:1–46:24, 2018. [Online]. Available: http://doi.acm.org/10.1145/3158134

[40] A. Medeiros, "Zookeeper's atomic broadcast protocol: Theory and practice," 2012.

[41] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Berfrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees," in *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2018, to appear.