Learning Interpretable Models in the Property Specification Language

Rajarshi Roy[1] , **Dana Fisman**[2*] and **Daniel Neider**[1*]
[1]Max Planck Institute for Software Systems, Kaiserslautern, Germany
[2]Ben-Gurion University, Be'er Sheva, Israel
rajarshi@mpi-sws.org, dana@cs.bgu.ac.il, neider@mpi-sws.org

Abstract

We address the problem of learning human-interpretable descriptions of a complex system from a finite set of positive and negative examples of its behavior. In contrast to most of the recent work in this area, which focuses on descriptions expressed in Linear Temporal Logic (LTL), we develop a learning algorithm for formulas in the IEEE standard temporal logic: Property Specification Language (PSL). Our work is motivated by the fact that many natural properties, such as an event happening at every n-th point in time, cannot be expressed in LTL, whereas it is easy to express such properties in PSL. Moreover, formulas in PSL can be more succinct and easier to interpret (due to the use of regular expressions in PSL formulas) than formulas in LTL. The learning algorithm we designed, builds on a framework recently proposed for learning LTL formulas. Roughly speaking, our algorithm reduces the learning task to a constraint satisfaction problem in propositional logic and uses a SAT solver to search for a solution. We have implemented our algorithm and performed a comparative study between the proposed method and the existing LTL learning algorithm. Our results illustrate the effectiveness of the proposed approach to provide succinct human-interpretable descriptions from examples.

1 Introduction

Inferring an understandable and meaningful model of a complex system is an important problem in practice. It arises naturally in various areas, including debugging, reverse engineering (e.g., of malware and viruses), specification mining for formal verification, and the modernization of legacy software. Also, this topic clearly falls under explainable AI, as the challenge is to obtain an *explainable* model of the studied phenomena rather than a black box function implementing it.

In recent years, inferring models in Linear Temporal Logic (LTL) has crystallized as one of the most promising approaches to help humans understand the (temporal) behavior of complex systems (see the related work for a detailed discussion). Originally developed by [Pnueli, 1977] in the context of reactive systems, LTL possesses not only a host of desirable theoretical properties (e.g., the ability to effectively translate formulas into finite automata) but also features a compact, variable-free syntax and an intuitive semantics. Specifically, these latter properties make it interesting as an interpretable description language with many applications in the area of artificial intelligence, including plan intent recognition, knowledge extraction, and reward function learning (see [Camacho and McIlraith, 2019] for details).

However, one of the major downsides of LTL is its limited expressive power as compared to other temporal logics. As a consequence, many properties that arise naturally (e.g., an event happening at every n-th point in time) cannot be expressed in LTL. In fact, the class of properties that can be expressed in LTL corresponds exactly to that of star-free ω-languages [Wolper, 1981], which excludes—among others—all properties involving modulo counting.

To overcome this serious limitation, the Property Specification Language (PSL) has been proposed, which has since been adopted by IEEE as an industrial standard for expressing temporal properties [IEEE Standards Association, 2010]. Although PSL is an extension of LTL and, hence, shares many of its beneficial properties, it differs from LTL in three aspects:

1. The expressive power of PSL exceeds that of LTL (it is as expressive as the full class of regular ω-languages [Armoni *et al.*, 2002]). In particular, properties involving modulo counting—as mentioned above—can easily be expressed in PSL.
2. PSL integrates easy-to-understand regular expressions in its syntax.
3. When learning from the observed behavior of a system, models expressed in PSL can be arbitrarily more succinct than those expressed in LTL (see Proposition 1).

We believe that these three properties make PSL particularly well-suited as an interpertable description language.

The main contribution of this paper is an algorithm for learning models (i.e., formulas) in PSL. Following earlier work on learning formulas expressed in LTL [Neider and Gavran, 2018; Camacho and McIlraith, 2019], the precise learning problem our algorithm solves, is as follows: given a sample $\mathcal{S}$ consisting of two finite sets of positive and negative

*The last two authors are ordered alphabetically.

examples, learn a PSL formula φ that is consistent with $\mathcal{S}$ in the sense that all positive examples satisfy φ, whereas all negative examples violate φ. Although we cannot expect algorithms that learn consistent formulas to scale as well as statistical methods that allow for misclassifications (e.g., [Kim *et al.*, 2019]), being able to learn an exact model describing the given data is essential in a multitude of applications, including few-shot learning, debugging of software systems, and many situations in which the observed data is without noise. We refer the reader to [Neider and Gavran, 2018; Camacho and McIlraith, 2019] for more examples where learning consistent formulas is important.

To be as general and succinct as possible, we here assume examples to be infinite, ultimately periodic words (i.e., words of the form uv^ω, where u, v are a finite words and v^ω is the infinite repetition of v) and focus on the core fragment of PSL. However, our algorithm can easily be adapted to learn from finite words and extends smoothly to other future-time temporal operators of PSL.

Our learning algorithm builds on top of the framework proposed by [Neider and Gavran, 2018] for learning formulas in LTL. Its key idea is to reduce the learning task to a series of constraint satisfaction problems in propositional logic and use a highly-optimized SAT solver to search for a solution. By design, our algorithm infers a minimal PSL formula that is consistent with the examples, which is a particularly valuable property in our setting: we seek to learn human-interpretable formulas and the size of the learned formula is a crucial metric for their interpretability (since larger formulas are generally harder to understand than smaller ones). As a result from the fact that PSL makes heavy use of regular expressions, **we also obtain a learning algorithm for minimal regular expressions over finite words as a byproduct of our approach**. Such a learning algorithm has many potential applications, for instance, in the field of natural language processing (e.g., see [Bartoli *et al.*, 2014]).

We empirically evaluate a prototype of our algorithm on benchmarks that reflect typical patterns of both LTL and PSL formulas used in practice. This evaluation shows that our algorithm can infer informative PSL formulas and that these formulas are often more succinct than pure LTL formulas learned from the same examples. Moreover, the runtime of our prototype is comparable to the state-of-the-art tool for learning LTL formulas by [Neider and Gavran, 2018].

Related Work. Learning of temporal properties has recently attracted increasing attention. The literature in this area can be broadly structured along three dimensions.

The first dimension is the type of logic used to express models. Examples include learning of models expressed in Signal Temporal Logic [Kong *et al.*, 2017], in Linear Temporal Logic [Neider and Gavran, 2018; Camacho and McIlraith, 2019; Riener, 2019], Computational Temporal Logic [Wasylkowski and Zeller, 2009] and several other temporal logics [Lamma *et al.*, 2007; Chesani *et al.*, 2009]. To the best of our knowledge, learning of models in PSL or an equally expressive logic has not yet been considered.

The second dimension is whether the learning algorithm requires the user to provide templates. Examples of algorithms that require templates are the works of [Li *et al.*, 2011] and [Lemieux *et al.*, 2015], whereas the algorithms for LTL mentioned above do not require templates. Note, however, that providing templates is often a challenging task as it requires the user to have a good understanding of the data. By contrast, our algorithm can learn arbitrary formulas without any assistance from the user.

The third dimension distinguishes between algorithms that learn an exact model and those that learn an approximate one. Like the majority of algorithms mentioned so far, the learning algorithm we devise in this paper is exact (i.e., it learns models that describe the data perfectly; due to our minimality constraint, however, these models generalize the data rather than overfit it). On the other hand, there also exists work that uses statistical methods to derive approximate formulas from noisy data [Kim *et al.*, 2019].

This work is built upon the SAT-based learning algorithm by [Neider and Gavran, 2018]. In fact, constraint solving is often used in learning problems. The perhaps most prominent examples are passive automata learning [Heule and Verwer, 2010; Neider, 2012] and counterexample-guided inductive synthesis [Alur *et al.*, 2018; Alur *et al.*, 2015].

2 Preliminaries

We now introduce the concepts used throughout this paper.

Alphabets and Words. An *alphabet* is a finite, nonempty set Σ, whose elements are called *symbols*.

A *finite word* over Σ is a finite sequence $u = a_0 \dots a_n$ with $a_i \in \Sigma$ for $i \in \{0, \dots, n\}$. The *empty word*, denoted by ε, is the empty sequence, and the length $|u|$ of a finite word u is the number of its symbols (note that $|\varepsilon| = 0$). Moreover, we denote the set of all words by Σ^* and define $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

An *infinite word* over Σ is an infinite sequence $\alpha = a_0 a_1 \dots$ with $a_i \in \Sigma$ for $i \in \mathbb{N}$, and Σ^ω denotes the set of all infinite words over Σ. Given $u \in \Sigma^+$, the infinite word $u^\omega = uuu\dots$ is called the *infinite repetition* of u. An infinite word α is said to be *ultimately periodic* if it is of the form $\alpha = uv^\omega$ for finite words $u \in \Sigma^*$ and $v \in \Sigma^+$.

Given an infinite word $\alpha = a_0 a_1 \dots \in \Sigma^\omega$ and $i, j \in \mathbb{N}$ with $i \leq j$, let $\alpha[i, j) = a_i \dots a_{j-1}$ be the finite *infix* of α from position i up to (and excluding) position j (note that $\alpha[i, i) = \varepsilon$). Moreover, let $\alpha[i] = a_i$ be the symbol at position i and $\alpha[i, \infty) = a_i a_{i+1} \dots$ the infinite *suffix* of α starting at position i. We define $u[i, j)$ and $u[i]$ analogously for finite words $u \in \Sigma^*$ and appropriate indices i, j.

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

$$\Phi ::= x \in \mathit{Var} \mid \neg\Phi \mid \Phi \lor \Phi$$

Additionally, we add syntactic sugar and allow the formulas tt (true), ff (false), $\Phi_1 \land \Phi_2$, $\Phi_1 \rightarrow \Phi_2$, and $\Phi_1 \leftrightarrow \Phi_2$, which are defined as usual.

An *interpretation* is a function $v \colon \mathit{Var} \rightarrow \mathbb{B}$, which assigns a Boolean value to each variable. The *semantics* of propositional logic is given in terms of a satisfaction relation $\models$ that is inductively defined as follows: $v \models x$ with $x \in \mathit{Var}$ if

and only if $v(x) = 1$; $v \models \neg\Phi$ if and only if $v \not\models \Phi$; and $v \models \Phi_1 \lor \Phi_2$ if and only if $v \models \Phi_1$ or $v \models \Phi_2$. If $v \models \Phi$, we say that v satisfies Φ and call it a *model of* Φ. Moreover, a formula Φ is *satisfiable* if there exists a model v of Φ.

The problem of deciding whether a propositional formula is satisfiable is the prototypical NP-complete problem [Cook, 1971]. Despite this fact, modern SAT solvers implement highly-optimized decision procedures that can check the satisfiability of formulas with millions of variables [Balyo *et al.*, 2017]. Moreover, virtually all SAT solvers return a model if the input-formula is satisfiable.

Linear Temporal Logic. The logic *LTL*, short for *Linear Temporal Logic* [Pnueli, 1977], is an extension of propositional logic that enables reasoning about time. The main building blocks of LTL are so-called *atomic propositions*, which, intuitively, correspond to interesting properties about the system in consideration. Given a finite set $\mathcal{P}$ of atomic propositions, formulas in LTL—which we denote by small Greek letters—are inductively constructed as follows:

$$\varphi ::= p \in \mathcal{P} \mid \neg\varphi \mid \varphi \lor \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

In addition to the temporal operators X ("next") and U ("until"), we also allow the derived operators F ("finally"), defined by $\mathsf{F}\,\varphi := tt\,\mathsf{U}\,\varphi$, and "globally", defined by $\mathsf{G}\,\varphi := \neg\,\mathsf{F}\,\neg\varphi$ (note that our technique seamlessly extends to any future-time temporal operator, such as "release", "weak until", and so on). Analogous to propositional logic, we also allow the formulas tt, ff, $\varphi \land \psi$, $\varphi \rightarrow \psi$, and $\varphi \leftrightarrow \psi$.

Formulas in LTL are evaluated over infinite words $\alpha \in \Sigma^\omega$ with $\Sigma = 2^\mathcal{P}$ (i.e., over infinite sequences of sets of atomic propositions, modeling which propositions hold true at which points in time). Similar to propositional logic, the semantics of LTL is defined in terms of a satisfaction relation $\models$, which formalizes when an infinite word $\alpha \in (2^\mathcal{P})^\omega$ *satisfies* an LTL formula: $\alpha \models p$ if and only if $p \in \alpha[0]$; $\alpha \models \neg\varphi$ if and only if $\alpha \not\models \varphi$; $\alpha \models \varphi_1 \lor \varphi_2$ if and only if $\alpha \models \varphi_1$ or $\alpha \models \varphi_2$; $\alpha \models \mathsf{X}\,\varphi$ if and only if $\alpha[1,\infty) \models \varphi$; and $\alpha \models \varphi_1\,\mathsf{U}\,\varphi_2$ if and only if there exists a $j \in \mathbb{N}$ such that $\alpha[j,\infty) \models \varphi_2$ and $\alpha[i,\infty) \models \varphi_1$ for each $i \in \{0,\ldots,j-1\}$. Note that the satisfaction of a formula, due to the temporal operators, depends on the satisfaction of its subformulas on (potentially different) infinite suffixes of α.

It is well-known that LTL cannot express natural properties such as modulo counting. To alleviate this serious restriction, the Property Specification Language (PSL) has been developed (e.g., see [Eisner and Fisman, 2006]), which makes extensive use of regular expressions. The remainder of this section introduces regular expressions and PSL in detail.

Regular Expressions. To simplify the definition of PSL, we define regular expressions in a slightly non-standard way. Firstly, we use propositional formulas rather than symbols of an alphabet as atomic expressions (e.g., for $\mathcal{P} = \{p, q\}$, the formula $p \lor q$ represents the set $\{\{p\},\{q\},\{p,q\}\}$ of symbols from $\Sigma = 2^\mathcal{P}$), whereas $p \land \neg q$ represents the singleton set $\{\{p\}\}$). Secondly, we take an operational view on regular expressions in terms of a matching relation rather than the classical view as generators of regular languages.

Regular expressions are inductively constructed as follows, where the left-hand-side describes the construction of atomic expressions and the right-hand-side describes the construction of general regular expressions:

$$\xi ::= p \in \mathcal{P} \mid \neg\xi \mid \xi \lor \xi \qquad \rho ::= \varepsilon \mid \xi \mid \rho + \rho \mid \rho \circ \rho \mid \rho^*$$

As usual, the regular operator $+$ stands for choice, $\circ$ stands for concatenation, and * for finite repetition (Kleene star). As syntactic sugar, we also allow the Boolean operators $\land$, $\rightarrow$, and $\leftrightarrow$ in atomic expressions.

Let us first give a meaning to atomic expressions. To this end, we assign to each atomic expression ξ a set $[\![\xi]\!] \subseteq 2^\mathcal{P}$ of symbols in the following way: $[\![p]\!] = \{A \in 2^\mathcal{P} \mid p \in A\}$; $[\![\neg\xi]\!] = 2^\mathcal{P} \setminus [\![\xi]\!]$; and $[\![\xi_1 \lor \xi_2]\!] = [\![\xi_1]\!] \cup [\![\xi_2]\!]$.

To define the semantics of regular expressions, we introduce a *matching relation* $\vdash$, which formalizes when an infix $u[i,j)$ of a finite word $u \in (2^\mathcal{P})^*$ *matches* a regular expression. Formally, the matching relation is defined as follows: $u[i,j) \vdash \varepsilon$ if and only if $j = i$; $u[i,j) \vdash \xi$ if and only if $j = i+1$ and $u[i] \in [\![\xi]\!]$; $u[i,j) \vdash \rho_1 + \rho_2$ if and only if $u[i,j) \vdash \rho_1$ or $u[i,j) \vdash \rho_2$; $u[i,j) \vdash \rho_1 \circ \rho_2$ if and only if there exists a $k \in \{i,\ldots,j\}$ such that $u[i,k) \vdash \rho_1$ and $u[k,j) \vdash \rho_2$; and $u[i,j) \vdash \rho^*$ if and only if $j = i$ or there exists a $k \in \{i+1,\ldots,j\}$ such that $u[i,k) \vdash \rho$ and $u[k,j) \vdash \rho^*$. Note that this definition applies to finite infixes $\alpha[i,j)$ of infinite words $\alpha \in (2^\mathcal{P})^\omega$ as well.

Property Specification Language. In this paper, we consider the core fragment of the *Property Specification Language* [Eisner and Fisman, 2006], which we here abbreviate as *PSL* for the sake of brevity. This fragment extends LTL with a so-called *triggers operator* $\rho \mapsto \varphi$ where ρ is a regular expression and φ is a PSL formula. Intuitively, a word $\alpha \in (2^\mathcal{P})^\omega$ satisfies the PSL formula $\rho \mapsto \varphi$ if φ holds every time the regular expression ρ matches on a finite prefix of α. To define the semantics of the triggers operator formally, we extend the satisfaction relation of LTL by $\alpha \models \rho \mapsto \varphi$ if and only if $\alpha[0,i) \vdash \rho$ implies $\alpha[i-1,\infty) \models \varphi$ for all $i \in \mathbb{N} \setminus \{0\}$. Finally, we define the *size* $|\varphi|$ of a PSL formula φ to be the number of its unique subformulas and subexpressions.

PSL is a popular specification language in industrial applications, having been standardized by IEEE [IEEE Standards Association, 2010]. It is as expressive as ω-regular languages [Armoni *et al.*, 2002] (i.e., languages accepted by nondeterministic Büchi automata) and, hence, exceeds the expressive power of LTL [Wolper, 1981]. A simple property that cannot be expressed in LTL is that a proposition p holds at every second point in time, which can be expressed in PSL as $(tt \circ tt)^* \mapsto p$ [Eisner and Fisman, 2018].

3 The Learning Problem

In this section, we formally define the learning problem studied in this paper. We assume the data to learn from is given as a pair $\mathcal{S} = (P, N)$ consisting of two finite, disjoint sets $P, N \subset \Sigma^\omega$ of ultimately periodic words such that $P \cap N = \emptyset$. We call this pair a *sample*. Moreover, we say that a PSL formula φ is *consistent* with a sample $\mathcal{S} = (P, N)$ if $\alpha \models \varphi$ for each $\alpha \in P$ and $\alpha \not\models \varphi$ for each $\alpha \in N$.

Having defined the setting, we can now state the learning task as *"given a sample $\mathcal{S}$, compute a PSL formula of minimal size that is consistent with $\mathcal{S}$"*. Note that this definition asks to

Algorithm 1: SAT-based learning algorithm for PSL

Input: A sample $\mathcal{S}$
1 $n \leftarrow 0$
2 **repeat**
3 $\quad n \leftarrow n+1$
4 $\quad$ Construct $\Phi_n^{\mathcal{S}}$ and check its satisfiability
5 **until** $\Phi_n^{\mathcal{S}}$ *is satisfiable, say with model* v
6 **return** φ_v

construct a PSL formula that is minimal among all consistent formulas. The motivation for this requirement is threefold. Firstly, we observe that the problem becomes simple without a restriction on the size: for $\alpha \in P$ and $\beta \in N$, one can easily construct a formula $\varphi_{\alpha,\beta}$ with $\alpha \models \varphi_{\alpha,\beta}$ and $\beta \not\models \varphi_{\alpha,\beta}$, which 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 trivially consistent with $\mathcal{S}$. However, simply enumerating all differences of a sample is clearly of little help towards the goal of learning a descriptive model. Secondly, small formulas are easier for humans to interpret, which justifies spending effort on learning a formula that is as small as possible. Thirdly, small formulas tend to provide good generalization of the behavior represented by the sample and avoids the possibility of simply overfitting it.

Before we explain our learning algorithm in detail, let us show that models expressed in PSL can be arbitrarily more succinct than those expressed in LTL, which follows from Theorem 4.1 of [Wolper, 1981].

Proposition 1. *Let $n \in \mathbb{N}$ and $\mathcal{S}_n = (P_n, N_n)$ over $\mathcal{P} = \{p\}$ with $P_n = \{\{p\}^{2n+1}\emptyset\{p\}^\omega\}$ and $N_n = \{\{p\}^{2n}\emptyset\{p\}^\omega\}$. Then $(p \circ p)^* \mapsto \mathsf{X}\, p$ is a PSL formula (of constant size) consistent with $\mathcal{S}_n$, whereas every LTL formula that is consistent with $\mathcal{S}_n$ has size greater than $2n$.*

4 The Learning Algorithm

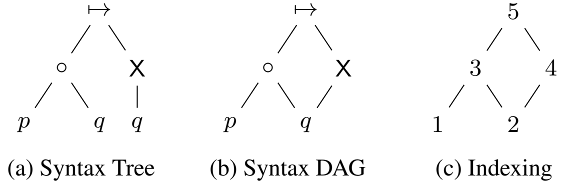
The idea underlying our algorithm is to reduce the construction of a minimally consistent PSL formula to a constraint satisfaction problem in propositional logic and to use a highly-optimized SAT solver to search for a solution. More precisely, given a sample $\mathcal{S}$, we construct a series $\left(\Phi_n^{\mathcal{S}}\right)_{n=1,2,\ldots}$ of propositional formulas that have the following properties:

1. there exists a PSL formula of size $n \in \mathbb{N} \setminus \{0\}$ that is consistent with $\mathcal{S}$ if and only if $\Phi_n^{\mathcal{S}}$ is satisfiable; and
2. given a model v of $\Phi_n^{\mathcal{S}}$, we can extract a PSL formula φ_v of size n that is consistent with $\mathcal{S}$.

By incrementing n (starting from 1) until $\Phi_n^{\mathcal{S}}$ becomes satisfiable, we obtain an effective learning algorithm for models expressed in PSL, as shown in Algorithm 1. Note that termination of this algorithm follows from the existence of a trivial solution (see Section 3). Moreover, its correctness follows from Properties 1 and 2 of $\Phi_n^{\mathcal{S}}$.

Theorem 1. *Given a sample $\mathcal{S}$, Algorithm 1 terminates and outputs a minimal PSL formula that is consistent with $\mathcal{S}$.*

Corollary 1. *A simple modification of Algorithm 1 learns a minimal regular expression from (finite) sample of finite words.*



(a) Syntax Tree (b) Syntax DAG (c) Indexing

Figure 1: Different representations of the PSL formula $(p \circ q) \mapsto \mathsf{X}\, q$

Roughly speaking, the formula $\Phi_n^{\mathcal{S}}$ is the conjunction $\Phi_n^{\mathcal{S}} := \Phi_n^{\text{str}} \wedge \Phi_n^{\text{cst}}$, where Φ_n^{str} encodes the structure of the prospective PSL formula and Φ_n^{cst} enforces that the prospective PSL formula is consistent with the sample. In the remainder of this section, we describe both Φ_n^{str} and Φ_n^{cst} in detail.

Structural Constraints. The formula Φ_n^{str} relies on a canonical syntactic representation of PSL formulas, which we call *syntax DAGs*. A syntax DAG is essentially a syntax tree (i.e., the unique tree that is derived from the inductive definition of a PSL formula) in which common subformulas are merged. This merging results into a directed, acyclic graph (DAG), whose number of nodes coincides with the number of subformulas of the prospective PSL formula. Figures 1a and 1b illustrate syntax trees and syntax DAGs, respectively.

To simplify our encoding, we assign a unique identifier $k \in \{1, \ldots, n\}$ to each node of a syntax DAG such that (a) the identifier of the root is n and (b) the identifier of an inner node is larger than the identifiers of its children (see Figure 1c). Note that this encoding entails that Node 1 is always a leaf, which is necessarily labeled with an atomic proposition.

Let now $\Lambda_R = \{\neg, \vee, +, \circ, {}^*\} \cup \mathcal{P}$ be the set of operators and atomic propositions that can appear in regular expressions and $\Lambda_P = \Lambda_R \cup \{\mathsf{X}, \mathsf{U}, \mapsto\}$ be the set of all PSL operators and atomic propositions. Then, we can encode a syntax DAG using the following propositional variables:

- $x_{k,\lambda}$ where $k \in \{1, \ldots, n\}$ and $\lambda \in \Lambda_P$
- $l_{k,\ell}$ where $k \in \{2, \ldots, n\}$ and $\ell \in \{1, \ldots, k-1\}$
- $r_{k,\ell}$ where $k \in \{2, \ldots, n\}$ and $\ell \in \{1, \ldots, k-1\}$

Intuitively, the variables $x_{k,\lambda}$ encode the labeling of a syntax DAG in the sense that if $x_{k,\lambda}$ is set to true, then node k is labeled by λ. Similarly, the variables $l_{k,\ell}$ and $r_{k,\ell}$ encode the left and right child of node k, respectively. By convention, we ignore the variables $r_{k,\ell}$ (resp. $r_{k,\ell}$ and $l_{k,\ell}$) if node k is labeled with an unary operator (resp. an atomic proposition).

To enforce that these variables in fact encode a syntax DAG, we first need to make sure that for each $k \in \{1, \ldots, n\}$ there exists precisely one $\lambda \in \Lambda_P$ such that $x_{k,\lambda}$ is set to true. This can be done with the following constraint:

$$\Big[\bigwedge_{1 \leq k \leq n} \bigvee_{\lambda \in \Lambda_P} x_{k,\lambda} \Big] \wedge \Big[\bigwedge_{1 \leq k \leq n} \bigwedge_{\lambda \neq \lambda' \in \Lambda_P} \neg x_{k,\lambda} \vee \neg x_{k,\lambda'} \Big]$$

Similarly, we assert that for each $k \in \{1, \ldots, n\}$ there exists precisely one $\ell \in \{1, \ldots, k-1\}$ and one $\ell' \in \{1, \ldots, k-1\}$ such that $l_{k,\ell}$ and $r_{k,\ell'}$ is set to true, respectively.

Next, we have to ensure that the labeling of the syntax DAG respects the type of the operators (e.g., children of a regular

expression are also regular expressions). The constraint below exemplifies this for the concatenation operator $\circ$:

$$\bigwedge_{\substack{1\leq k\leq n\\ 1\leq \ell,\ell'<k}} [x_{k,\circ} \wedge l_{k,\ell} \wedge r_{k,\ell'}] \rightarrow \Big[\bigvee_{\lambda\in\Lambda_R} x_{\ell,\lambda} \wedge \bigvee_{\lambda\in\Lambda_R} x_{\ell',\lambda} \Big]$$

We add analogous constraints for all other operators. Note that the constraint for the triggers operator is slightly different as it combines a regular expression and a PSL formula.

It is left to enforce that Node 1 is always labeled with an atomic proposition. We do so using the constraint $\bigvee_{p\in\mathcal{P}} x_{1,p}$.

Finally, let Φ_n^{str} be the conjunction of all constraints discussed above. Then, one can construct a syntax DAG from a model v of Φ_n^{str} in a straightforward manner: label Node k with the unique $\lambda \in \Lambda_P$ such that $v(x_{k,\lambda}) = 1$, designate Node n as the root, and arrange the nodes as described uniquely by $v(l_{k,\ell})$ and $v(r_{k,\ell})$. Subsequently, we can derive a PSL formula from this syntax DAG, which we denote by φ_v. To ensure that φ_v is consistent with $\mathcal{S}$, we add further constraints (i.e., a formula Φ_n^{cst}), which we describe next.

Constraints for Consistency. To construct the propositional formula Φ_n^{cst}, we exploit a simple observation about PSL.

Observation 1. *Let $uv^\omega \in (2^{\mathcal{P}})^\omega$ and φ be a PSL formula. Then, $uv^\omega[|u|+i,\infty) = uv^\omega[|u|+j,\infty)$ for $j \equiv i \mod |v|$. Thus, $uv^\omega[|u|+i,\infty) \models \varphi$ if and only if $uv^\omega[|u|+j,\infty) \models \varphi$.*

Intuitively, Observation 1 states that there exists only a finite number of distinct infinite suffixes of a word uv^ω, which eventually repeat periodically. Since the semantics of PSL is defined in terms of the suffixes of a word, we can determine whether a word uv^ω satisfies a PSL formula based on its finite prefix uv alone. To illustrate this claim, consider the formula $\mathsf{X}\,\varphi$ and suppose that we want to determine whether $uv^\omega[|uv|-1,\infty) \models \mathsf{X}\,\varphi$ holds (i.e., the satisfaction of $\mathsf{X}\,\varphi$ at the end of the prefix uv). Then, Observation 1 allows us to reduce this question to checking whether $uv^\omega[|u|,\infty) \models \varphi$ holds, instead of the original semantics of the X-operator, which depends on whether $uv^\omega[|uv|,\infty) \models \varphi$ is satisfied.

For reasoning about matchings of regular expressions, however, it is not enough to just consider the prefix uv. For instance, consider the ultimately periodic word $uv^\omega = \emptyset\{p\}(\emptyset)^\omega$ and the PSL formula $\varphi := (tt \circ tt)^* \mapsto p$ (stating that p is true at every second position). By just considering the prefix $uv = \emptyset\{p\}\emptyset$, it seems that $uv^\omega \models \varphi$. However, *unrolling* the repeating part $v = \emptyset$ once more, resulting in the prefix $uvv = \emptyset\{p\}\emptyset\emptyset$, immediately shows that $uv^\omega \not\models \varphi$.

Similar to Observation 1, the next lemma provides a bound $b \in \mathbb{N}$ on the number of unrollings required to gather enough information to determine the satisfaction of a triggers operator. This bound depends on the number n of nodes of the syntax DAG and the function $M_{u,v} \colon \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$M_{u,v}(j) = \begin{cases} j & \text{if } j < |uv|\text{; and} \\ |u| + ((j-|u|)\,\%\,|v|) & \text{if } j \geq |uv|, \end{cases}$$

where $a\,\%\,b$ is the remainder of the division a/b. Intuitively, $M_{u,v}$ maps a position j in the word uv^ω to an appropriate position within the prefix uv. The lemma uses finite automata as representations of regular expressions to derive the bound.

Lemma 1. *Let $uv^\omega \in (2^{\mathcal{P}})^\omega$, $\psi = \rho \mapsto \varphi$ with $|\psi| = n$, and $b = 2^{n+1}$. Then, $uv^\omega[i,\infty) \models \psi$ if and only if $uv^\omega[i,j) \vdash \rho$ implies $uv^\omega[M_{u,v}(j-1),\infty) \models \varphi$ for all $j \leq |u| + b|v|$.*

Note an important property of Lemma 1: reasoning about regular expressions and the triggers operator $\mapsto$ requires us to consider the prefix uv^b, while the prefix uv is sufficient for reasoning about the remaining PSL operators.

Towards the definition of the formula Φ_n^{cst}, we construct for each ultimately periodic word uv^ω in $\mathcal{S}$ a propositional formula $\Phi_n^{u,v}$ that tracks the satisfaction of the PSL formula encoded by Φ_n^{cst} (and all its subformulas/subexpressions) on uv^ω. Each of these formulas is built over auxiliary variables:

- $y_{i,k}^{u,v}$ with $0 \leq i < |uv|$ and $k \in \{1,\dots,n\}$
- $z_{i,j,k}^{u,v}$ with $0 \leq i \leq j \leq |uv^b|$, $b = 2^{n+1}$ as in Lemma 1, and $k \in \{1,\dots,n\}$

The meaning of these variables is that $y_{i,k}^{u,v}$ is set to true if and only if $uv^\omega[i,\infty)$ satisfies the PSL formula rooted at Node k (if that node is labeled with a PSL operator); similarly, $z_{i,j,k}^{u,v}$ is set to true if and only if $uv^\omega[i,j)$ matches the regular expression rooted at Node k (if that node is labeled with a regular expression operator). Note that we have to create both the variables $y_{i,k}^{u,v}$ and $z_{i,j,k}^{u,v}$ for each node since the "type" of a node is determined dynamically during SAT solving.

It is left to enforce that the variables $y_{i,k}^{u,v}$ and $z_{i,j,k}^{u,v}$ have the desired meaning. For the Boolean and temporal operators (except the triggers operator), we reuse the constraints proposed by [Neider and Gavran, 2018]. For instance, the constraint for the atomic propositions is

$$\bigwedge_{1\leq k\leq n} \bigwedge_{p\in\mathcal{P}} x_{k,p} \rightarrow \bigwedge_{0\leq i<|uv|} \begin{cases} y_{i,k}^{u,v} & \text{if } p \in uv[i]\text{; and} \\ \neg y_{i,k}^{u,v} & \text{if } p \notin uv[i]. \end{cases}$$

Intuitively, this constraint states that if Node k is labeled with the atomic proposition $p \in \mathcal{P}$, then the variables $y_{i,k}^{u,v}$ capture precisely the presence or absence of p in the k-th position of the prefix uv. Similarly, the constraint for the X-operator is

$$\bigwedge_{1\leq k\leq n,\ 1\leq \ell<k} [x_{k,\mathsf{X}} \wedge l_{k,\ell}] \rightarrow \Big[\bigwedge_{0\leq i<|uv|-1} \big[y_{i,k}^{u,v} \leftrightarrow y_{i+1,\ell}^{u,v} \big] \Big] \wedge \big[y_{|uv|-1,k}^{u,v} \leftrightarrow y_{|u|,\ell}^{u,v} \big],$$

which states that if Node k is labeled with X and its left child is Node ℓ, then the satisfaction of the formula rooted at Node k at time i (i.e., $y_{i,k}^{u,v}$) equals the satisfaction of the subformula rooted at Node ℓ at time $i+1$ (i.e., $y_{i+1,\ell}^{u,v}$), except at time $|uv|-1$, where it "wraps around" to time $|u|$ (see Observation 1).

The constraints for regular expressions follow the matching relation and refer to the variables $z_{i,j,k}^{u,v}$ instead of $y_{i,k}^{u,v}$. Exemplarily, we present the constraints for the $\circ$-operator (constraints for the other regular operators are analogous):

$$\bigwedge_{1\leq k\leq n,\ 1\leq \ell,\ell'<k} [x_{k,\circ} \wedge l_{k,\ell} \wedge r_{k,\ell'}] \rightarrow \bigwedge_{0\leq i\leq j\leq |uv^b|} \Big[z_{i,j,k}^{u,v} \leftrightarrow \bigvee_{i\leq t\leq j} z_{i,t,\ell}^{u,v} \wedge z_{t,j,\ell'}^{u,v} \Big]$$

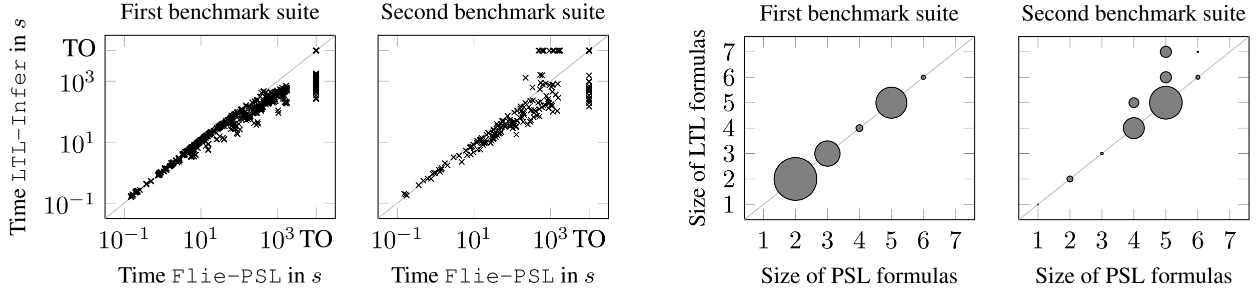


Figure 2: Comparison of Flie-PSL and LTL-Infer. Bubble size is ∝ to no. of benchmarks with the formula size. "TO" indicates timeouts.

Finally, the constraint below captures the semantics of the triggers operator $\mapsto$ by relating the variables $y^{u,v}_{i,k}$ and $z^{u,v}_{i,j,k}$.

$$\bigwedge_{1\leq k\leq n,\ 1\leq \ell,\ell'<k} [x_{k,\mapsto} \land l_{k,\ell} \land r_{k,\ell'}] \rightarrow \bigwedge_{0\leq i<|uv|} \Big[y^{u,v}_{i,k} \leftrightarrow \bigwedge_{i\leq j\leq |uv^b|} \Big[z^{u,v}_{i,j,\ell} \rightarrow y^{u,v}_{M_{u,v}(j-1),\ell'} \Big]\Big]$$

As the final step, we define the formula Φ^{cst}_n by

$$\Phi^{\text{cst}}_n := \Big[\bigwedge_{uv^\omega \in P} \Phi^{u,v}_n \land y^{u,v}_{0,n} \Big] \land \Big[\bigwedge_{uv^\omega \in N} \Phi^{u,v}_n \land \neg y^{u,v}_{0,n} \Big],$$

which enforces that all positive words in $\mathcal{S}$ satisfy the prospective PSL formula ($y^{u,v}_{0,n}$ has to be true), while all negative words violate it ($y^{u,v}_{0,n}$ has to be false).

5 Evaluation

We have implemented a prototype of our learning algorithm, named Flie-PSL[1] (Formal Language Inference Engine for PSL), which is written in Python and uses Z3 [de Moura and Bjørner, 2008] as SAT solver. Deviating slightly from Algorithm 1, we have implemented the following improvement: instead of using $y^{u,v}_{i,k}$ and $z^{u,v}_{i,j,k}$ for each node, we use the latter variables only for $0 \leq m < n$ nodes and the former variables for the remaining $n - m$ nodes (the constraints generated for these variables remain as in Section 4. This modification essentially limits the size of a regular expression in the final PSL formula to m. To obtain a complete algorithm, we iterate over all valid values for m before increasing n.

To assess the performance of our prototype, we have compared it to the LTL learning algorithm by [Neider and Gavran, 2018], which we call LTL-Infer for brevity. To make this comparison as fair as possible, we have used two benchmark suites. The first benchmark suite is taken directly from [Neider and Gavran, 2018] and contains 1217 samples, which were generated from common LTL properties. The second benchmark suite simulates real-world PSL use-cases and contains 390 synthetic samples, which we have generated from PSL formulas that commonly appear in practice (e.g., $(p_1 \circ p_2)^* \mapsto q$; see [Eisner and Fisman, 2006] for more examples). Our procedure to generate these samples is similar to the one by [Neider and Gavran, 2018] and proceeds as follows: firstly, we select a formula φ from our pool of PSL formulas; secondly, we generate samples consisting of a number (ranging from 10 to 500) of ultimately periodic words uv^ω with $|u| + |v| \leq 15$; thirdly, we partition the words in each sample into sets P and N depending on their satisfaction of φ. In total, the median size of the samples in the second benchmark suite is 100 words. All experiments were conducted on a single core of an Intel Xeon E7-8857 V2 CPU (at 3.6 GHz) with a timeout of 1800 s.

The two diagrams on the left of Figure 2 compare the runtime of Flie-PSL and LTL-Infer on the first and second benchmark suite, respectively. In general, Flie-PSL was moderately slower than LTL-Infer and timed out 1.34 times more often (Flie-PSL timed out 38.4% and 56.2% of the times on the first and second benchmark suite, respectively, whereas LTL-Infer timed out 24.8% and 53.6% of the times). This was surprising given that learning PSL formulas is inherently more complex than learning LTL formulas (both are essentially search problems, and the search space for PSL formulas subsumes the one for LTL formulas). Moreover, for PSL formulas that are not LTL formulas, one has to consider longer prefixes of the word uv^ω (see Lemma 1), resulting in a more complex SAT encoding. In fact, there were even 25 benchmarks on which Flie-PSL outperformed LTL-Infer because it was able to learn smaller formulas.

The two diagrams on the right of Figure 2 compare the size of the learned formulas. On the first benchmark suite, we observe that Flie-PSL mainly produced pure LTL formulas of the same size as LTL-Infer (a likely explanation for this is that these benchmarks have explicitly been designed to capture LTL properties). However, on 68 benchmarks of the second suite, Flie-PSL learned PSL formulas containing regular expressions and was able to recover the exact PSL property used to generate the sample in 40 of the benchmarks. Overall, Flie-PSL learned a smaller formula in 52 benchmarks.

6 Conclusion

We have developed an algorithm for learning interpretable models expressed in PSL and have shown empirically that this algorithm infers interesting PSL formulas with only little overhead as compared to learning LTL formulas. For future work, we plan to extend our algorithm to noisy data and, orthogonally, to learn models expressed as ω-regular expressions.

Acknowledgements

This work was partly supported by DFG grant no. 434592664.

[1] The tool is available at https://github.com/ifm-mpi/Flie-PSL.

References

[Alur *et al.*, 2015] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[Alur *et al.*, 2018] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, 2018.

[Armoni *et al.*, 2002] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The forspec temporal logic: A new temporal property-specification language. In *8th International Conference of Tools and Algorithms for the Construction and Analysis of Systems, TACAS '02*, volume 2280 of *LNCS*, pages 296–211. Springer, 2002.

[Balyo *et al.*, 2017] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In *31st AAAI Conference on Artificial Intelligence, AAAI '17*, pages 5061–5063. AAAI Press, 2017.

[Bartoli *et al.*, 2014] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *IEEE Computer*, 47(12):72–80, 2014.

[Camacho and McIlraith, 2019] Alberto Camacho and Sheila A. McIlraith. Learning interpretable models expressed in linear temporal logic. In *29th International Conference on Automated Planning and Scheduling, ICAPS '18*, pages 621–630. AAAI Press, 2019.

[Chesani *et al.*, 2009] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Exploiting inductive logic programming techniques for declarative process mining. *Trans. Petri Nets Other Model. Concurr.*, 2:278–295, 2009.

[Cook, 1971] Stephen A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing, STOC '71*, pages 151–158. ACM, 1971.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *14th International Conference of Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[Eisner and Fisman, 2006] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006.

[Eisner and Fisman, 2018] Cindy Eisner and Dana Fisman. Functional specification of hardware via temporal logic. In *Handbook of Model Checking*, pages 795–829. 2018.

[Heule and Verwer, 2010] Marijn Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *10th International Colloquium of Grammatical Inference: Theoretical Results and Applications, ICGI '10*, volume 6339 of *LNCS*, pages 66–79. Springer, 2010.

[IEEE Standards Association, 2010] IEEE Standards Association. IEEE 1850-2010 – IEEE standard for property specification language (PSL), 2010.

[Kim *et al.*, 2019] Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *28th International Joint Conference on Artificial Intelligence, IJCAI '19*, pages 5591–5598. ijcai.org, 2019.

[Kong *et al.*, 2017] Zhaodan Kong, Austin Jones, and Calin Belta. Temporal logics for learning and detection of anomalous behavior. *IEEE Trans. Automat. Contr.*, 62(3):1210–1222, 2017.

[Lamma *et al.*, 2007] Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Inducing declarative logic-based models from labeled traces. In *BPM*, volume 4714 of *LNCS*, pages 344–359. Springer, 2007.

[Lemieux *et al.*, 2015] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL specification mining (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE '15*, pages 81–92. IEEE Computer Society, 2015.

[Li *et al.*, 2011] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE '11*, pages 43–50. IEEE, 2011.

[Neider and Gavran, 2018] Daniel Neider and Ivan Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design, FMCAD '18*, pages 1–10. IEEE, 2018.

[Neider, 2012] Daniel Neider. Computing minimal separating dfas and regular invariants using SAT and SMT solvers. In *10th International Symposium of Automated Technology for Verification and Analysis, ATVA '12*, volume 7561 of *LNCS*, pages 354–369. Springer, 2012.

[Pnueli, 1977] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium of Foundations of Computer Science, FOCS '77*, pages 46–57. IEEE Computer Society, 1977.

[Riener, 2019] Heinz Riener. Exact synthesis of LTL properties from traces. In *2019 Forum for Specification and Design Languages, FDL '19*, pages 1–6. IEEE, 2019.

[Wasylkowski and Zeller, 2009] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. In *24th IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 295–306. IEEE Computer Society, 2009.

[Wolper, 1981] Pierre Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundations of Computer Science, FOCS '81*, pages 340–348. IEEE Computer Society, 1981.