Towards ILP-based LTL_f passive learning *

Antonio Ielo[1 [0009−0006−9644−7975]], Mark Law[2,4 [0000−0002−8934−9756]], Valeria Fionda[1 [0000−0002−7018−1324]], Francesco Ricca[1 [0000−0001−8218−3178]], Giuseppe De Giacomo[3 [0000−0001−9680−7658]], and Alessandra Russo[4 [000−0002−3318−8711]]

[1] University of Calabria, Rende, Italy, antonio.ielo@unical.it
[2] ILASP Limited, Grantham, United Kingdom
[3] University of Oxford, Oxford, United Kingdom
[4] Imperial College, London, United Kingdom

Abstract. Inferring a LTL_f formula from a set of example traces, also known as passive learning, is a challenging task for model-based techniques. Despite the combinatorial nature of the problem, current state-of-the-art solutions are based on exhaustive search. They use an example at the time to discard a single candidate formula at the time, instead of exploiting the full set of examples to prune the search space. This hinders their applicability when examples involve many atomic propositions or when the target formula is not small. This short paper proposes the first ILP-based approach for learning LTL_f formula from a set of example traces, using a learning from answer sets system called ILASP. It compares it to both pure SAT-based techniques and the exhaustive search method. Preliminary experimental results show that our approach improves on previous SAT-based techniques and that has the potential to overcome the limitation of an exhaustive search by optimizing over the full set of examples. Further research directions for the ILP-based LTL_f passive learning problem are also discussed.

Keywords: Answer Set Programming · Linear temporal logic over finite traces · Learning from answer sets.

1 Introduction

Linear Temporal Logic (LTL) [29] provides a concise, expressive, and human-interpretable language to specify and reason about the temporal behavior of systems. Over the years, it has been widely used in formal verification, model checking, and monitoring to ensure the correctness of software and hardware systems. Unlike LTL, whose specifications are interpreted over infinite sequences, LTL_f [17, 11] deals with properties that are evaluated on finite sequences or

* This work was partially supported by MUR under PRIN project PINPOINT Prot. 2020FNEB27, CUP H23C22000280006; PRIN project HypeKG Prot. 2022Y34XNM, CUP H53D23003710006; PNRR MUR project PE0000013-FAIR, Spoke 9 – WP9.1 and WP9.2; Spoke 5 - WP5.1 and PNRR project Tech4You, CUP H23C22000370006, ERC Advanced Grant WhiteMech (No. 834228), and EU ICT-48 2020 project TAILOR (No. 952215).

traces. LTL_f was developed to deal with scenarios in which the standard semantics over infinite traces were not appropriate, such as when reasoning over business processes, which are typically finite [1]. However, developing manual specifications of system behaviors is becoming difficult due to the complexity, dynamic changes, and evolution of current systems. Automated approaches for inferring LTL_f specifications from data are becoming increasingly important. *Passive learning* of LTL_f formulae [28] refers to the problem of inferring an LTL_f formula from a set of system execution traces (some of which might be labeled as negative examples). The problem has been extensively studied in the literature [2, 16, 28, 30] motivated by the need of learning human-interpretable models that allow explaining the observed behavior of complex systems. Surprisingly, the problem has been proven to be challenging for model-based techniques. In fact, despite its combinatorial nature, the current state-of-the-art [16] is exhaustive-search-based systems that do not apply any pruning to reduce the search space, beyond efficiently evaluating formulae over traces. Although these approaches may be acceptable in certain domains, their reliance on exhaustive search poses limitations in domains with numerous atomic propositions or lengthy target formulae. Specifically, the search space of LTL_f formulae grows exponentially both in the size of the formula being searched and in the number of atomic propositions, thereby preventing their applicability. Thus, investigating the use of techniques capable of exploiting multiple examples to prune the search space, such as ILP, is an important step towards solutions to the passive learning problem that is applicable to real-world problems.

In this paper we propose an approach, based on inductive logic programming (ILP), for learning LTL_f formulae from positive and negative execution traces. Due to the combinatorial complexity of the problem, we make use of ILASP [24], a learning from answer sets system shown to generalize many of the existing ILP methods [23], and to be able to learn specifications expressed in answer set programming [24], a computational environment tailored to solve combinatorial optimization problems. ILP techniques have been applied to the task of learning of DECLARE [1] models [6], a declarative process modeling language whose patterns are defined by LTL_f formulae. However, to the best of our knowledge, this is the first ILP-based method for learning LTL_f formulae from finite execution traces without imposing syntactic restrictions or limiting the search to a set of patterns. Other ILP-based works tackled how to include LTL_f specifications among features used in learning algorithms [32], as well as applications to learning *temporal logic programs* [20]. Specifically, we propose a novel representation of the problem of passive learning of LTL_f formulae in answer set programming [3]. We show how this representation can be adapted to express this problem as a learning from answer set task and as an SAT-based inference task. We then conduct an experimental evaluation of our ILP-based approach over a set of event logs on cellular networks' attacks, (already used in the literature to benchmark systems) and compare its performance with respect to the existing SAT-based methods. Our evaluation shows that our approach improves upon previous SAT- and SMT-based techniques, as implemented by

the SySLite [2] system. The contributions presented in this paper are the first stepping stone towards the development of ILP-based systems for LTL_f passive learning that can be used in real-world settings. To this end, our future work will address the scalability of our proposed method to demonstrate its advantage versus exhaustive search methods, and to solve passive learning problems where execution traces refer also to structured properties of the data.

2 Background

In this section, we introduce linear temporal logic over finite traces and present the basic notions and terminologies of Answer Set Programming (ASP) and Learning from Answer Sets (LAS) used throughout the paper.

2.1 Linear temporal logic over finite traces

Linear temporal logic (LTL) [29] is an extension of propositional logic which allows reasoning about time-related properties in sequences of events by means of *temporal operators*. Classically, LTL formulae are interpreted over infinite traces. When considering LTL on finite traces, the formalism LTL_f [17] keeps the same syntax of standard LTL, but shifts its focus from infinite sequences of events to finite traces. We define now the syntax and semantics of LTL_f.

Syntax. Let $\mathcal{P}$ be a finite, non-empty, set of propositional symbols. An LTL_f formula is inductively defined according to the following grammar:

$$\varphi ::= true \mid false \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi \ \mathsf{U} \ \varphi$$

where $p \in \mathcal{P}$ and φ is an LTL_f formula. The set $\{\wedge, \vee, \neg\}$ includes the standard conjunction, disjunction and negation operators of classical logic, while X and U denote respectively the *next* and *until* temporal operators. We assume the standard propositional logic equivalence rewriting that defines logical implication $\varphi \rightarrow \varrho$ as $\neg\varphi \vee \varrho$. Furthermore, we define the following derived temporal operators: (*Weak Next*) $\mathsf{X_w}\varphi \equiv \neg\mathsf{X}\neg\varphi$; (*Eventually*) $\mathsf{F}\varphi \equiv true \ \mathsf{U} \ \varphi$; (*Release*) $\varphi_1 \mathsf{R} \varphi_2 \equiv \neg(\neg\varphi_1 \ \mathsf{U} \ \neg\varphi_2)$; and (*Always*) $\mathsf{G}\varphi \equiv false \ \mathsf{R} \ \varphi$. The size of a formula φ, denoted by $|\varphi|$, is the total number of symbols (temporal operators, boolean connectives, and propositional symbols) included in φ. That is, $|\varphi| = 1$ if $\varphi \in \mathcal{P} \cup \{true, false\}$; $|\circ \varphi| = 1 + |\varphi|$ if $\circ \in \{\neg, \mathsf{X}, \mathsf{X_w}, \mathsf{F}, \mathsf{G}\}$; and, $|\varphi_1 \circ \varphi_2| = 1 + |\varphi_1| + |\varphi_2|$ if $\circ \in \{\wedge, \vee, \rightarrow, \mathsf{U}, \mathsf{R}\}$.

Semantics. A finite *trace* over propositional symbols in $\mathcal{P}$ is a sequence $\pi = \pi_0 \cdots \pi_{n-1}$ of *states*, where each state $\pi_i \subseteq \mathcal{P}$ is a set of propositional symbols that hold at time instant i. The *length* of a trace is the number of states over which it is defined, and it is indicated as $|\pi|$.

Given a formula φ and a trace π, we define that π *satisfies* φ at time instant i, denoted $\pi, i \models \varphi$, inductively as follows:

$\pi, i \models p$ iff $p \in \pi_i$;
$\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$
$\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$
$\pi, i \models \varphi_1 \vee \varphi_2$ iff $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$
$\pi, i \models \mathsf{X}\varphi_1$ iff $i < |\pi| - 1$ and $\pi, i+1 \models \varphi_1$;
$\pi, i \models \varphi_1 \mathsf{U} \varphi_2$ iff $\exists j$ with $i \leq j \leq |\pi|$ s.t. $\pi, j \models \varphi_2$ and $\forall k$ with $i \leq k < j$ $\pi, k \models \varphi_1$

Given a trace π and a formula φ, we say that π is a *model* of φ if $\pi, 0 \models \varphi$, denoted in brief as $\pi \models \varphi$. An $\mathrm{LTL_f}$ formula is said to be in *neXt Normal Form* (`xnf`) [8, 25] when all its occurrences of U temporal operators are nested into some X operator. Note that, every $\mathrm{LTL_f}$ formula can be transformed into an equivalent formula in `xnf` form in linear time by recursively applying the following transformations:

- $\mathtt{xnf}(\varphi) = \varphi$ for $\varphi \in \mathcal{P} \cup \{true, false\}$;
- $\mathtt{xnf}(\neg\varphi) = \neg\mathtt{xnf}(\varphi)$;
- $\mathtt{xnf}(\varphi_1 \circ \varphi_2) = \mathtt{xnf}(\varphi_1) \circ \mathtt{xnf}(\varphi_2)$ for $\circ \in \{\wedge, \vee\}$;
- $\mathtt{xnf}(\mathsf{X}\ \varphi) = \mathsf{X}\ \mathtt{xnf}(\varphi)$;
- $\mathtt{xnf}(\varphi_1\ \mathsf{U}\ \varphi_2) = \mathtt{xnf}(\varphi_2) \vee (\mathtt{xnf}(\varphi_1) \wedge \mathsf{X}(\varphi_1\ \mathsf{U}\ \varphi_2))$

2.2 Passive Learning of $\mathrm{LTL_f}$ formulae

The problem of passive learning of $\mathrm{LTL_f}$ formulae, introduced in [28], refers to the challenge of automatically inferring $\mathrm{LTL_f}$ formulae from observed traces of system behavior, usually partitioned into sets of positive and negative examples, such that the positive traces are models of the formula and the negative traces are not models of the formula. This can be formally defined as follows:

Definition 1 (PL_{LTL_f} Passive Learning Task). *Let $\mathcal{P}$ be a set of propositional symbols. A* PL_{LTL_f} passive learning task *is a tuple* $\mathrm{PL}_{LTL_f} = (\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$ *where $\mathcal{E}^+$ is a set of traces over $\mathcal{P}$ called* positive traces*, and $\mathcal{E}^-$ is a set of traces over $\mathcal{P}$ called* negative traces *such that $\mathcal{E}^+ \cap \mathcal{E}^- = \emptyset$. A* solution *of a* PL_{LTL_f} *task is an LTL_f formula φ, written in $\mathcal{P}$ such that (i) $\pi \models \varphi$, for every $\pi \in \mathcal{E}^+$; and (ii) $\pi \not\models \varphi$, for every $\pi \in \mathcal{E}^-$.*

Note that, a PL_{LTLf} passive learning task always accepts a *trivial* solution given by the formula $\phi = \bigvee_{\pi \in \mathcal{E}^+} \bigwedge_{i \in [0,\ldots,|\pi|-1]} \mathsf{X}^i(\bigwedge_{p \in \pi_i} p \wedge \bigwedge_{p \notin \pi_i} \wedge \neg p) \wedge \neg\mathsf{X}^{|\pi|} true$, where X^i denotes the nested application of the X operator i times. We therefore focus on *optimal solutions* of a PL_{LTLf} passive learning task, as defined below.

Definition 2 (Optimal solution of a PL_{LTL_f} Passive Learning Task). *Let $\mathrm{PL}_{LTL_f} = (\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$ be a PL_{LTL_f} passive learning task. An LTL_f formula φ, written in $\mathcal{P}$, is an* optimal solution *of PL_{LTL_f} if and only if φ is a solution of PL_{LTL_f} and there is no LTL_f formula φ' written in $\mathcal{P}$ that is a solution of PL_{LTL_f} and $|\varphi'| < |\varphi|$.*

Solving a PL_{LTL_f} passive learning task means searching for an optimal (i.e. minimal-size) solution with respect to a fixed set of propositional symbols $\mathcal{P}$.

2.3 Answer Set Programming

Answer Set Programming (ASP) [3, 15] is a knowledge representation formalism based on the stable model semantics of logic programs, that allows modeling in a declarative way problems up to the second level of the polynomial hierarchy. We recall in the following some basic notions of ASP and assume the reader is familiar with the input language of Clingo [14].

Typically an ASP program includes four types of rules: normal rules, choice rules, hard and soft constraints. In this paper, we consider ASP programs composed of only normal rules, choice rules, and hard constraints. Given atoms h, $h_1,\ldots,\ h_k,\ b_1,\ldots,\ b_n,\ c_1,\ldots,\ c_m$, a *normal rule* is of the form h :- $b_1,\ldots,b_n$, `not` $c_1,\ldots,$ `not` c_m, with h as the *head* and $b_1,\ldots,b_n$, `not` $c_1,\ldots,$ `not` c_m (collectively) as the *body* ("`not`" represents negation as failure); a *constraint* is a rule of the form :- $b_1,\ldots,b_n$, `not` $c_1,\ldots,$ `not` c_m; and a *choice rule* is a rule of the form $l\{h_1,\ldots,h_k\}u$:- $b_1,\ldots,b_n$, `not` $c_1,\ldots,$ `not` c_m where $\texttt{l}\{\texttt{h}_1,\ldots,\texttt{h}_\texttt{k}\}\texttt{u}$ is called an *aggregate*. In an aggregate `l` and `u` are integers and $\texttt{h}_\texttt{i}$, for $\texttt{1} \leq \texttt{i} \leq \texttt{k}$, are atoms. A choice rule specifies that when the *body* is satisfied at least `l`, and no more than `u`, atoms from the *head* must evaluate true.

Given an ASP program P, the Herbrand Base of P, denoted as HB_P, is the set of ground (variable free) atoms that can be formed from the predicates and constants that appear in P. Subsets of HB_P are (Herbrand) interpretations of P. Informally, a model of an ASP program P, called *Answer Set* of P, is defined in terms of the notion of *reduct* of P, which is constructed by applying the following transformation steps to the grounding of P[5]. Given a program P and an Herbrand interpretation $I \subseteq HB_P$, the reduct P^I is constructed from the grounding of P by first removing rules whose bodies contain the negation of an atom in I; secondly, we remove all negative literals from the remaining rules; thirdly, we set $\bot$ (note $\bot \notin HB_P$) to be the head to every constraint, and in every choice rule whose head is not satisfied by I we replace the head with $\bot$; and finally, we replace any remaining choice rule $l\{h_1,\ldots,h_m\}u$:- $b_1,\ldots,b_n$ with the set of rules $\{h_i$:- $b_1,\ldots,b_n \mid h_i \in I \cap \{h_1,\ldots,h_m\}\}$. Any $I \subseteq HB_P$ is an *answer set* of P if it is the minimal model of the *reduct* P^I. We denote an answer set of a program P with A and the set of answer sets of P with $AS(P)$. A program P is said to be satisfiable (resp. unsatisfiable) if $AS(P)$ is non-empty (resp. empty).

2.4 Learning from Answer Sets

Many ILP systems learn from (positive and negative) examples of atoms which should be true or false, as many ILP systems are targeted at learning Prolog programs, where the main "output" of a program is a query of a single atom. In this paper, we make use of the Learning from Answer Sets (LAS) paradigm. In ASP, the main "output" of a program is a set of answer sets. So learning from Answer Sets takes as (positive and negative) examples *(partial) interpretations*,

[5] We use the simplified definitions of the *reduct* for choice rules presented in [22]

which should or should not (respectively) be answer sets of the learned ASP program. A *partial interpretation* e is a pair of sets of atoms $\langle e^{inc}, e^{exc} \rangle$, referred to as the *inclusions* and *exclusions* respectively. An interpretation I is said to *extend* e if and only if $e^{inc} \subseteq I$ and $e^{exc} \cap I = \emptyset$. A *context-dependent partial interpretation* (CDPI) is a tuple $e = \langle e_{pi}, e_{ctx} \rangle$, where e_{pi} is a partial interpretation and e_{ctx} is an ASP program called a *context*. A CDPI e is *accepted* by a program P if and only if there is an answer set of $P \cup e_{ctx}$ that extends e_{pi}.

Many ILP systems (e.g. [19]) use mode declarations as a form of language bias to specify hypothesis spaces. We adopt a similar notion of language bias. A *mode bias* is defined as a pair of sets of mode declarations $\langle M_h, M_b \rangle$, where M_h (resp. M_b) are called the *head* (resp. *body*) *mode declarations*. Each mode declaration is a literal whose abstracted arguments are either $var(t)$ or $const(t)$, for some constant t (called a *type*). Informally, a literal is *compatible* with a mode declaration m if it can be constructed by replacing every instance of $var(t)$ in m with a variable of type t, and every $const(t)$ with a constant of type t.[6] Given a mode bias $M = \langle M_h, M_b \rangle$, a rule R if compatible with M if (i) the head of R is compatible with a mode declaration in M_h; (ii) each body literal of R is compatible with a mode declaration in M_b; and (iii) no variable occurs with two different types. We indicate with S_M the set of rules compatible with a given language bias $M = \langle M_h, M_b \rangle$, and we refer to it as the *hypothesis space* S_M.

We can now define the notion of context-dependent Learning from Answer Sets. This consists of an ASP background knowledge B, a hypothesis space, and sets of context-dependent positive and negative partial interpretation examples. The goal is to find a hypothesis H that has at least one answer set (when combined with B) that extends each positive example, and no answer set that extends any negative examples. Note that each positive example could be extended by a different answer set of the learned program. This can be formally defined as follows.

Definition 3 (Context-dependent learning from answer sets). *A* Context-dependent Learning from Answer Sets *task, denoted as* $ILP^{context}_{LAS}$*, is a tuple* $T = \langle B, S_M, E^+, E^- \rangle$ *where* B *is an ASP program,* S_M *is a set of ASP rules, and* E^+ *and* E^- *are finite sets of CDPIs. A hypothesis* $H \subseteq S_M$ *is an inductive solution of* T *if and only if (i)* $\forall e \in E^+$, $B \cup H$ *accepts* e*; and (ii)* $\forall e \in E^-$, $B \cup H$ *does not accept* e*.*

It is common practice in ILP to search for "optimal" hypotheses. This is usually defined in terms of the number of literals in the hypothesis. Given a hypothesis H, the length of the hypothesis, $|H|$, is the number of literals that appear in H.

Definition 4 (Optimal solution of $ILP^{context}_{LAS}$ tasks). *Let* T *be a* $ILP^{context}_{LAS}$ *learning task. A hypothesis* H *is an* optimal inductive solution *of* T *if and only*

[6] The set of constants of each type is assumed to be given with a task, together with the maximum number of variables in a rule, giving a set of variables $V_1, \ldots, V_{max}$ that can occur in a hypothesis. Whenever a variable V of type t occurs in a rule, the atom $t(V)$ is added to the body of the rule to enforce the type.

if H is an inductive solution of T, and there is no inductive solution H' of T such that $|H'| < |H|$.

3 Formalizing LTL_f semantics in ASP

In this section, we present an encoding for evaluating LTL_f formulae over traces, by embedding LTL_f semantics into a normal logic program. We present our encoding into different subsections, addressing how to represent traces, formulae, and temporal logic operators' evaluation rules in logic programs.

Encoding traces. We assume traces to be uniquely indexed by integers, and in particular we will assume that a trace π^i is referred to by the integer i. We encode a trace π^i over $\mathcal{P}$ as a set of facts matching the predicates `trace/3` and `trace/2`. The atom $trace(i, t, a)$ models that $a \in \pi^i_t$. In order to be able to model empty states, we introduce the atoms $trace(i, t)$ for $0 \leq t < |\pi^i|$. Further information about the trace can be encoded by auxiliary predicates which refer to the trace identifier in the first term. In this paper, the only additional information to encode is whether each trace is a positive, $\pi^i \in \mathcal{E}^+$, or negative, $\pi^i \in \mathcal{E}^-$, example, and this is done through the atoms $pos(i), neg(i)$ respectively. We denote by $P(\pi)$ the set of facts that encode the trace π. With a slight abuse of notation, we will also denote by $P(\mathcal{E})$ the set of facts that encode the set of traces $\mathcal{E}$, that is $P(\mathcal{E}) = \bigcup_{\pi^i \in \mathcal{E}} P(\pi^i)$.

Example 1. Consider the trace $\pi^0 = \{a\} \cdot \{a, b\} \cdot \{\}$, and assume $\pi^0 \in \mathcal{E}^+$. This is encoded by the following facts:

```
trace(0,0,a). trace(0,1,a). trace(0,1,b).
trace(0,0). trace(0,1). trace(0,2). pos(0).
```

Encoding formulae. We encode a formula by reifying its syntax tree, in a similar way as authors of [28] do in SAT, by means of the predicates `edge/2`, `order/3`, `label/2` and `node/1`. The predicates `node/1`, `edge/2` model trees in a natural way, where we use natural numbers to identify nodes. The atoms $node(x), edge(y, x)$ model that x is a node of the tree, and that y is its parent. The predicate `label/2` models logic operators (or propositions) associated with each node in the tree. An atom $label(x, j)$ encodes that the node x is labeled with $j \in \mathcal{O} \cup \mathcal{P}$, where $\mathcal{O}$ is the set of available temporal and propositional logic operators. In this paper, we assume $\mathcal{O} = \{\neg, \vee, \wedge, \mathsf{X}, \mathsf{U}, \rightarrow, \mathsf{F}, \mathsf{G}\}$. The atom $order(i, lhs, rhs)$ distinguishes between left and right of node i, which is needed for the evaluation of the non-commutative operators $\{\mathsf{U}, \rightarrow\}$. We denote by $P(\varphi)$ the set of facts which encode a formula φ. Without loss of generality, we will assume that the node identified by 1 is the root of a formula's tree.

Example 2. Consider the formula $(\mathsf{X}a)\ \mathsf{U}\ b$. This is encoded by the following facts:

```
node(1..4). label(1, until). label(2, next). label(3, a).
label(4,b). edge(1,2). edge(1,4). edge(2,3). order(1,2,4).
```

Encoding semantics. We encode the semantics of each supported operator by simulating the recursive application of the `xnf`$(\cdot)$ transformation by means of normal recursive rules. In particular, each subformula is identified by the node identifier of its root in the syntax tree. The atom $holds(i, t, x)$ models that the $\pi^i, t \models \varphi_x$ where φ_x is the subformula of φ rooted in the node identified by integer x. The atom $last(i, t)$ models that $|\pi^i| = t$, that is π^i_t is the last state of π^i. The definition of these rules, which we denote by $P_{\mathrm{LTL_f}}$, follows the `xnf`$(\cdot)$ definitions in Section 2.1. The encoding for operators in $\{\wedge, \vee, \neg, \mathsf{U}, \mathsf{X}, \rightarrow\}$, denoted by the constants `and`, `or`, `neg`, `until`, `next` and `implies` respectively, is as follows:

```
holds(TID, T, X)
  :- label(X, A), trace(TID, T, A).
holds(TID, T, X)
  :- label(X, next), edge(X, Y), holds(TID, T+1, X), not last(TID, T).
holds(TID, T, X)
  :- label(X, until), order(X,LHS,RHS), holds(TID, T, RHS).
holds(TID, T, X)
  :- label(X, until), order(X,LHS,RHS), holds(TID, T, LHS), holds(TID, T+1, X).
holds(TID, T, X)
  :- label(X, and), order(X,A,B), holds(TID, T, A), holds(TID, T, B).
holds(TID, T, X)
  :- label(X, or), edge(X, A), holds(TID, T, A).
holds(TID, T, X)
  :- label(X, neg), edge(X, Y), not holds(TID, T, Y), trace(TID, T).
holds(TID, T,X)
  :- label(X,implies), order(X,LHS,RHS), holds(TID, T,RHS), holds(TID, T,LHS).
holds(TID, T,X)
  :- label(X,implies), order(X,LHS,RHS), not holds(TID, T,LHS), trace(TID, T).
holds(TID, T, X)
  :- label(X, eventually), edge(X,Y), holds(TID, T,Y).
holds(TID, T, X)
  :- label(X, eventually), holds(TID, T+1, X), trace(TID, T).
holds(TID, T, X)
  :- label(X, always), edge(X, Y), holds(TID, T, Y), last(TID, T).
holds(TID, T, X)
  :- label(X, always), edge(X, Y), holds(TID, T, Y), holds(TID, T+1, X).
last(TID, T) :- trace(TID, T), not trace(TID, T+1).
sat(TID) :- holds(TID, 0,1).
unsat(TID) :- not sat(TID), trace(TID,_).
```

Listing 1.1. The logic program $P_{\mathrm{LTL_f}}$

For values $t' > t$ of the second term of the atom $holds/3$ it is possible to represent subsequent instants of each `xnf`$(\cdot)$ formula. For the evaluation of `xnf` formulae, it is sufficient to evaluate the current state and next state of the trace. Evaluation of this kind of rules produces a locally-stratified program (i.e., the resulting ground instantiation is stratified) [5], since whenever $holds(i, t, x)$ is in the head of a rule the body of the rule can contain only atoms $holds(i, t, \_)$ or $holds(i, t + 1, \_)$. Thus, when solved with the other subprograms encoding traces and formulae (that are only facts) has a unique answer set [7]. In particular, by observing that the rules implement the recursive application of `xnf`$(\cdot)$ which yields an equivalent formula to φ, it can be proved that,

$\pi^i \models \varphi$ (resp. $\pi^i \not\models \varphi$) if $holds(i, |\pi^i| - 1, 1)$ is (is not) in the unique answer set of $P(\pi^i) \cup P(\varphi) \cup P_{LTLf}$.

4 LTL$_f$ passive learning in plain ASP

A first way to model the passive learning problem is to frame it as an abduction problem in ASP [9], where the set of abducibles corresponds to facts matching the predicates `node/1`, `edge/2`, `label/2`, which reify into facts the syntax tree of a LTL$_f$ formula. The goal of the abduction task is to find an LTL$_f$ formula φ for which all $e \in \mathcal{E}^+$ we have that $e \models \varphi$ and for all $e \in \mathcal{E}^-$ we have that $e \not\models \varphi$. The following rules encode, denoted P_{tree} and P_{label} respectively, the abduction of an LTL$_f$ formula of size n:

```
node(1..n).
pair(X,Y) :- node(X), node(Y), X < Y.
1 { edge(Y,X): pair(Y,X) } 1 :- node(X), X > 1.
reach(1). reach(X) :- edge(Y,X), reach(Y).
:- node(X), not reach(X).
id(1,(0,0)).
id(V,(U,V*V+U)) :- edge(U,V).
:- id(I,RI), id(I+1,RJ), RI >= RJ.
:- id(I,RJ), id(I+1,RI), RI <= RJ.
```

Listing 1.2. The logic program P_{tree}

Each answer set of P_{tree} encodes a tree of size n. Due to the combinatorial nature of the problem, we introduce basic symmetry-breaking constraints in order to avoid generating isomorphic trees. In particular, we assume that each node has an identifier that is greater than its parent's identifier. Furthermore, the last constraints force the node identifiers to respect the order of a BFS traversal of the tree starting from the root node, which is adapted from [12] to our setting which does not require labeled edges.

```
unary_label(neg; next; eventually; always).
binary_label(and; until; or).
leaf(X)   :- node(X), not edge(X,_).
unary(X)  :- node(X), not leaf(X), not binary(X).
binary(X) :- edge(X,Y), edge(X,Y'), Y < Y'.
1 {label(X,L): unary_label(L)  }  1 :- unary(X).
1 {label(X,L): binary_label(L) } 1 :- binary(X).
proposition(A) :- trace(_,_,A).
1 {label(X,L): proposition(L)  }  1 :- leaf(X).
```

Listing 1.3. The logic program P_{label}

Each answer set, projected on the predicates `node/1`, `edge/2`, `label/2`, `order/3`, corresponds to the encoding of an LTL$_f$ formula. In order to encode the goals of the abduction, we have to constrain the generated formulae to accept positive examples and reject negative examples. The following rules, P_{goal}, can encode the goal of the abduction:

```
sat(TID) :- holds(TID,1,1).
unsat(TID) :- trace(TID,_), not sat(TID).
```

```
:- sat(TID), neg(TID).
:- unsat(TID), pos(TID).
```

Listing 1.4. The logic program P_{goal}

Thus, every answer set, projected onto the predicates `label/2`, `edge/2`, `node/1` and `order/3` as shown in Example 2, of the logic program $P_{tree} \cup P_{label} \cup P_{goal} \cup P(\mathcal{E}^+) \cup P_{\text{LTL}_\text{f}} \cup P(\mathcal{E}^-)$, can be mapped to an LTL_f formula of size n accepting all positive examples and rejecting all negative examples, that is a solution to the passive learning problem instance $(\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$. In order to find an optimal, size-minimal, solution the above approach can be easily implemented in an incremental fashion using Clingo incremental solving APIs, in order to search for solutions of increasing formula length n. This allows generating trees incrementally, rather than up-front for a fixed size n. Thus, the first solution to be found will be a minimal one. We omit this for brevity since it is very similar to well-known examples in literature [18].

5 LTL_f passive learning using ILASP

In this section, we frame an instance $(\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$ of the passive learning problem as a context-dependent learning from answer sets task, by providing suitable mode biases, background knowledge and encoding of the examples $\mathcal{E}^+ \cup \mathcal{E}^-$.

Mode bias. We use the following mode bias, which we report in the input language of ILASP [21]. This specifies available operators (via the type `op`), the maximum size of the formula (via the type `node_id`) to search for as well as the available atomic propositions (via the type `atom`) (here omitted, since it depends on the traces in $\mathcal{E}^+ \cup \mathcal{E}^-$). We assume a constant declaration of type `atom` for each $p \in \mathcal{P}$. Analogously to the plain ASP encoding, the predicates `edge/2`, `label/2` encode the labeled syntax tree of a LTL_f formula. The constant `n` below refers to the maximum size of the target formula to be inferred. Notice the mode bias consists only of `#modeh` directives: thus the inductive solution will be a set of facts matching the predicates in Example 2, which can be interpreted as the encoding of a LTL_f formula.

```
#constant(node_id, 1..n).
#constant(op, next).
#constant(op, until).
#constant(op, eventually).
#constant(op, always).
#constant(op, and).
#constant(op, neg).
#constant(op, or).
#constant(op, implies).
#modeh( edge(const(node_id), const(node_id)) ).
#modeh( label(const(node_id), const(op)) ).
#modeh( label(const(node_id), const(atom)) ).
```

Background knowledge. We assume the logic program P_{LTL_f} to be the background knowledge of our LAS task. We assume an atom $\mathit{proposition}(a)$ for each $a \in \mathcal{P}$. Basically, since the evaluation of the inductive solution over examples is automatically handled by the ILASP system, it is not needed to handle the evaluation of a formula

over multiple traces in the encoding. Thus, we drop the first term of the predicates `holds/3`, `last/2`, `sat/1`, `unsat/1`.

Encoding constraints over search space. A candidate inductive solution $H \subseteq S_M$ models a syntax tree encoding a LTL_f formula according to the previously defined fact schema, using predicates `node/1`, `edge/2`, `label/2`. The following ASP program represents the context of a special positive example e^*, which imposes syntactical constraints over the syntax tree of the candidate inductive solution. This addresses the same constraints as the ones in P_{tree} and P_{label}.

```
% Nodes are terms that appear in edge/2 or first term of label/2
node(X) :- label(X,_).
node(X) :- edge(_,X).
node(X) :- edge(X,_).
% Don't skip nodes
node(X) :- node(X+1), X >= 1.
% The syntax tree of the formula must be connected
reach(1). reach(T) :- edge(R,T), reach(R).
:- node(X), not reach(X).
:- node(X), not edge(_,X), X > 1.
% Bounded fan-out for logic operators
:- node(X), 3 #count { Z: edge(X,Z) }.
% Exactly one label per node
:- node(X), not label(X,_).
:- label(X,A), label(X,B), A < B.
% Syntax tree admits a BFS-indexing
id(1,(0,0)).
id(V,(U,V*V+U)) :- edge(U,V).
:- id(I,RI), id(I+1, RJ), RI >= RJ.
:- id(I+1,RI), id(I,RJ), RI <= RJ.
% Labels must match node's arity
arity(X,0) :- node(X), not edge(X,_).
arity(X,2) :- node(X), edge(X,Y), edge(X,Y1), Y < Y1.
arity(X,1) :- node(X), not arity(X,0), not arity(X,2).
:- arity(X,N), label(X,Y), not symbol(Y,N).
symbol(A,0) :- proposition(A).
symbol(next,1).  symbol(until,2).  symbol(eventually,1).  symbol(always,1).
symbol(neg,1).  symbol(and,2).  symbol(or,2).  symbol(implies,2).
```

Encoding examples. We encode $\pi \in \mathcal{E}^+$ as a positive example which includes the constant `sat` in the inclusion set and an empty exclusion set, while $\pi \in \mathcal{E}^-$ includes the constant `unsat` in the inclusion set and an empty exclusion set. In both cases, the context of the example consists of $P(\pi)$ for $\pi \in \mathcal{E}^+ \cup \mathcal{E}^-$. Similarly, we drop the term `TID` from the predicates `trace/3`, `trace/2` as for the background knowledge program.

6 Evaluation

This section reports an experimental evaluation that aims at assessing both ASP-based approaches presented in the previous section, and to compare existing solutions based

on SAT and SMT. In the experimental evaluation we used the event logs pertaining to the passive learning of *attack signatures* on cellular networks, and partitioned each event log into positive and negative traces. *Signatures* are formulae of kind $\mathsf{G}\varphi$, which characterize the positive traces of each log on each time instant. A comprehensive description of each log is available in [2] and its technical report. We compare our ILASP-based solution with our plain ASP solution in order to assess whether ILP can help in this setting, as well as other SAT-based approaches previously implemented in literature, referring to their implementation in the SySLite system. Experimental data and full encodings are available on GitHub (`https://github.com/ilp2023-27/data`).

Execution environment. All experiments were executed on an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 512GB RAM machine, using Clingo version 5.4.0, Python 3.10, ILASP 4.2.0 and the version of SySLite available in the authors' repository. All experiments were run in parallel using GNU Parallel [33]. We report execution time in seconds, with a timeout of 3600 seconds execution time on each event log. Currently, the most appropriate ILASP version for the task is ILASP 2i, which is the one we use to run the experiments.

Data. The dataset is composed by 18 logs: AKA Bypass (AKA), uthentication Failure (AF), Bank Transaction (BT), Chinese Wall Policy (CWP), Dynamic Separation Policy (DSP), EMM Information (EMM), Financial Institute (FI), GLBA, HIPPA 16450A2 (HIPPA-1), HIPPA 16450A3 (HIPPA-2), Identity Malformed (IM), IMSI Catching (IMSI-1), IMSI Cracking (IMSI-2), Measurement Report (MR), NULL Encryption (NE), Numb Attack (NA), Paging with IMSI (IMSI-3), RLF Report (RLF). We considered each log as an instance for the passive learning problem. Since SySLite algorithms target pure-past formulae, we reverse each trace in the log in order to use our encodings

Table 1. Execution time in seconds for compared methods over the event logs. Best execution time is in bold. T.O denotes timeout (execution time >3600s).

Event Log	SySLite$^{\text{sygus}}$	ILASP 2i	Abduction	SySLite$^{\text{sat}}$	SySLite$^{\text{guided_sat}}$
AKA	144.2	**62.542**	612.31	T.O.	T.O.
AF	**1.98**	3.32	4.705	360.7	T.O.
BT	2.17	**1.437**	13.239	659.23	133.19
CWP	22.01	**7.739**	148.891	T.O.	T.O.
DSP	T.O.	T.O	T.O	T.O.	T.O.
EMM	**4.64**	5.78	7.207	1155.18	T.O.
FI	51.59	**19.865**	510.189	T.O.	T.O.
GLBA	26.86	**13.606**	183.542	T.O.	T.O.
HIPPA-1	**25.96**	31.994	194.694	T.O.	T.O.
HIPPA-2	2.41	**1.693**	15.962	707.43	153.41
IM	**4.04**	4.483	5.837	977.77	T.O.
IMSI-1	3.72	**3.584**	4.877	774.34	T.O.
IMSI-2	6.89	**4.782**	8.092	1144.58	T.O.
MR	995.85	**55.622**	T.O	T.O.	T.O.
NE	**2.43**	4.301	4.706	480.18	T.O.
NA	2.54	**2.545**	2.8	1877.79	T.O.
IMSI-3	16.55	**11.652**	98.643	T.O.	T.O.
RLF	560.44	**23.266**	T.O	T.O.	T.O.

and samples2ltl tool. In this way, all approaches are able to learn the same formulae up to dual relabeling of temporal operators involved in the formulae. In particular, we indicate by SySLite^L, $L \in \{\text{sygus}, \text{sat}, \text{sat\_guided}\}$ the different algorithms available in SySLite, implementing SMT- and SAT-based algorithms. In particular, the `sygus` algorithm is SMT-based, exploiting bit-vector theories for efficient computations. `ILASP 2i` column refers to our ILASP encoding, and `Abduction` column refers to our (incremental) plain ASP encoding that uses abduction. Since the SySLite tool targets pure-past formulae rooted on the *historically* operator (the pure-past dual of G), we (i) invert the traces before defining our LAS and ASP encoding; (ii) add to our encoding the constraint that target formulae must be rooted in G. Since ILASP currently does not support incremental solving (wrt the definition of the hypothesis space), but rather solves a complexity-wise harder optimization task, we assume the maximum size of the target formula is known beforehand. All systems support the same set of temporal logic operators $\{\mathsf{X}, \mathsf{U}, \mathsf{F}, \mathsf{G}, \wedge, \vee, \neg\}$. We run the different algorithms available in SySLite with a timeout of one hour, along with ILASP and our abductive encoding, on a suite of event logs. The solution based on ILASP compares favorably, as shown in Table 1, with the algorithms implemented in SySLite, and even outperforms it on some event logs. Our plain ASP solution is noticeably slower than the `sygus` SMT-based algorithm and ILASP alike. This suggests that ILP based on ILASP might be a viable approach to scale beyond current model-based techniques without over-relying on pure enumerative approaches.

7 Related works

The seminal work [28] defines two algorithms for the passive learning problem of LTL_f formulae. One of them introduces SAT solvers as practical tools for LTL_f passive learning, encoding formulae's syntax trees and their evaluation over traces as a satisfiability problem, while the other exploits a decision tree to propositionally combine smaller LTL_f formulae, addressing scalability but dropping the "optimality" of the solution (in terms of formula size). Another approach [30], in order to improve scalability, targets the *directed* fragment of LTL_f, which however is not as expressive as LTL_f as it is unable to express the *until* temporal operator. Other SAT-based works target equivalent formalisms (such as *alternate automata* [4]), that can then be translated into LTL_f formulae. The SySLite [2] system targets pure-past LTL_f formulae of the form $\mathsf{H}\varphi$, where H is the past version of the operator G. It implements different SAT-based algorithms (the ones in [31] as well as SMT-based *syntax-guided synthesis* [31] enumeration which exploits bit-vector theories for fast evaluation. Recently, an approach based solely on a highly optimized exhaustive search has been proposed [16] that enumerates formulae of increasing size and performs pruning on syntactic and semantic criteria on a single trace at a time. A direct comparison with [16] could not be implemented since the tool does not expose an API to force learning of specific formulae, e.g. starting with G. Thus implementing a comprehensive empirical comparison, in the specific case of learning signatures, would require a modification of the tool of [16].From the theoretical standpoint, computational complexity-wise, the authors of [10] identify multiple fragments of LTL for which the passive learning problem is already NP-complete, and sample complexity-wise (e.g., how many examples are required to guarantee a given formula is learned) it is known [4] passive learning of arbitrary LTL_f formulae can be done with an exponential number of examples under some conditions.

8 Conclusion

In this paper, we presented an ILP approach based on the ILASP system for the passive learning of LTLf formulae. Our approach embeds LTL_f semantics into a normal logic program, similar to previous works based on SAT, which is provided as the background knowledge. We outperform SAT-based techniques as implemented in SySLite and compare favorably against its best-performing SMT-based syntax-guided enumeration algorithm. We also implement an abduction-based algorithm based on ASP, proving our performance gains are due to ILASP's inductive loop rather than the use of plain ASP with respect to SAT or SMT encoding. As future work we plan to improve the scalability of our proposed method and extend it to take into account data attached to events that occur during the system's execution. A comparison with the approach of [16] is also in our plans to possibly demonstrate there is an advantage versus exhaustive search methods. Another interesting extension we are interested in, which would extend the applicability of passive learning in real-world settings, is to apply our techniques to *noisy domains* [13, 27] (where traces or their labels might contain errors) by exploiting ILASP's support for *example's penalties* and ASP optimization techniques. It would also be interesting to check whether a compilation-based ASP system [26] can be beneficial to improve the performance of the abductive approach, where we conjecture the number of symbols generated by evaluating candidate solutions over the event log is one of the causes of performance degradation.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Comput. Sci. Res. Dev. **23**(2), 99–113 (2009)
2. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: FM-CAD. pp. 93–103 (2020)
3. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
4. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: ICAPS. pp. 621–630 (2019)
5. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Surveys in computer science, Springer (1990)
6. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. Trans. Petri Nets Other Model. Concurr. **2**, 278–295 (2009)
7. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3), 374–425 (2001)
8. Dodaro, C., Fionda, V., Greco, G.: LTL on weighted finite traces: Formal foundations and algorithms. In: IJCAI. pp. 2606–2612 (2022)
9. Eiter, T., Gottlob, G., Leone, N.: Abduction from logic programs: Semantics and complexity. Theor. Comput. Sci. **189**(1-2), 129–177 (1997)
10. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: ICGI. pp. 237–250 (2021)
11. Fionda, V., Greco, G.: LTL on finite and process traces: Complexity results and a practical reasoner. J. Artif. Intell. Res. **63**, 557–623 (2018)

12. Furelos-Blanco, D., Law, M., Jonsson, A., Broda, K., Russo, A.: Induction and exploitation of subgoal automata for reinforcement learning. J. Artif. Intell. Res. **70**, 1031–1116 (2021)
13. Gaglione, J., Neider, D., Roy, R., Topcu, U., Xu, Z.: Maxsat-based temporal logic inference from noisy data. Innov. Syst. Softw. Eng. **18**(3), 427–442 (2022)
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2012)
15. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
16. Ghiorzi, E., Colledanchise, M., Piquet, G., Bernagozzi, S., Tacchella, A., Natale, L.: Learning linear temporal properties for autonomous robotic systems. IEEE Robotics Autom. Lett. **8**(5), 2930–2937 (2023)
17. Giacomo, G.D., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 854–860. IJCAI/AAAI (2013)
18. Kaminski, R., Romero, J., Schaub, T., Wanko, P.: How to build your own asp-based system?! Theory Pract. Log. Program. **23**(1), 299–361 (2023)
19. Kazmi, M., Schüller, P., Saygın, Y.: Improving scalability of inductive logic programming via pruning and best-effort optimisation. Expert Systems with Applications **87**, 291–303 (2017)
20. Kolter, R.: Inductive temporal logic programming. Ph.D. thesis, University of Kaiserslautern (2009)
21. Law, M., Russo, A., Broda, K.: The ILASP system for learning answer set programs. `www.ilasp.com` (2015)
22. Law, M., Russo, A., Broda, K.: Simplified reduct for choice rules in ASP. Tech. rep., Department of Computing (DTR2015-2), Imperial College London (2015)
23. Law, M., Russo, A., Broda, K.: The complexity and generality of learning answer set programs. Artif. Intell. **259**, 110–146 (2018)
24. Law, M., Russo, A., Broda, K.: Logic-based learning of answer set programs. In: Reasoning Web. pp. 196–231 (2019)
25. Li, J., Pu, G., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: Sat-based explicit ltlf satisfiability checking. Artificial Intelligence **289**, 103369 (2020)
26. Mazzotta, G., Ricca, F., Dodaro, C.: Compilation of aggregates in ASP systems. In: AAAI. pp. 5834–5841. AAAI Press (2022)
27. Mrowca, A., Nocker, M., Steinhorst, S., Günnemann, S.: Learning temporal specifications from imperfect traces using bayesian inference. In: DAC. p. 96. ACM (2019)
28. Neider, D., Gavran, I.: Learning linear temporal properties. In: FMCAD. pp. 1–10 (2018)
29. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57. IEEE Computer Society (1977)
30. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning fragments of linear temporal logic. In: TACAS. Lecture Notes in Computer Science, vol. 13243, pp. 263–280 (2022)
31. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In: CAV. pp. 74–83 (2019)
32. Ribeiro, T., Folschette, M., Magnin, M., Okazaki, K., Kuo-Yen, L., Inoue, K.: Diagnosis of Event Sequences with LFIT. In: The 31st International Conference on Inductive Logic Programming (ILP) (2022)
33. Tange, O.: GNU parallel: The command-line power tool. login Usenix Mag. **36**(1) (2011)