Towards ILP-based LTLf passive learning

ANTONIO IELO, *University of Calabria, Department of Mathematics and Computer Science, Rende, 87036, Italy.*

MARK LAW, *ILASP Limited, Grantham, NG31 7EU, United Kingdom. Imperial College London, Department of Computing, London, SW7 2AZ, United Kingdom.*

VALERIA FIONDA, *University of Calabria, Department of Mathematics and Computer Science, Rende, 87036, Italy.*

FRANCESCO RICCA, *University of Calabria, Department of Mathematics and Computer Science, Rende, 87036, Italy.*

GIUSEPPE DE GIACOMO, *University of Oxford, Department of Computer Science, Oxford, OX1 3QD, United Kingdom.*

ALESSANDRA RUSSO, *Imperial College London, Department of Computing, London, SW7 2AZ, United Kingdom.*

Abstract

Inferring linear temporal logic over finite traces (LTL_f) formulas from a set of example traces, known as *passive learning*, presents significant challenges due to its combinatorial nature. In this paper, we introduce a novel approach to LTL_f passive learning based on inductive logic programming (ILP), leveraging the inductive learning of answer set programs framework. Our ILP-based method effectively exploits the set of example traces to guide the learning process, and experimental results demonstrate that it o ffers a more efficient solution compared to traditional techniques based on propositional satisfiability.

Keywords: Answer set programming, linear temporal logic over finite traces, learning from answer sets

1 Introduction

Linear temporal logic (LTL) [34] is a well-established, expressive and human-interpretable formalism for specifying and reasoning about the temporal behavior of systems. Over time, LTL has become a cornerstone in the fields of formal verification, model checking and system monitoring to ensure the correctness of both software and hardware systems. However, LTL semantics is defined over infinite sequences, which limits its applicability in scenarios where finite sequences are more relevant, such as when dealing with business processes, where the sequence of events typically terminates [1]. To address this limitation, linear temporal logic over finite traces (LTL_f) [16, 22] was introduced, focusing on properties evaluated over finite sequences.

Despite LTL_f's suitability for many real-world applications, manually developing specifications is becoming increasingly complex due to the dynamic and evolving nature of modern systems.

Vol. 36, No. 2,
https://doi.org/10.1093/logcom/exaf069

Consequently, automated techniques for inferring LTL_f specifications from execution traces have become critical. In particular, *passive learning* of LTL_f formulae [33] refers to the task of inferring an LTL_f formula from a set of traces (some possibly labeled as negative examples). This problem has received significant interest due to the need for human-interpretable models that can explain the behavior of complex systems [3, 21, 33, 35].

EXAMPLE 1.
Consider a system that monitors a manufacturing process where a conveyor belt moves products through different stages. The system generates finite traces representing sequences of sensor readings and actions. Each trace reflects a complete cycle of the process. Normal operation traces are:

$$P = \{\pi_1 = \langle\{\text{belt\_start}\}, \{\text{sens\_active}\}, \{\text{prod\_det}\}, \{\text{sens\_inactive}\}, \{\text{belt\_stop}\}\rangle,$$
$$\pi_2 = \langle\{\text{belt\_start}\}, \{\text{sens\_active}\}, \{\text{prod\_det}\}, \{\text{belt\_stop}\}\rangle\}$$

Erroneous or anomalous system behavior traces are:

$$N = \{\pi_3 = \langle\{\text{belt\_start}\}, \{\text{sens\_active}\}, \{\text{belt\_stop}\}\rangle,$$
$$\pi_4 = \langle\{\text{belt\_start}\}, \{\text{belt\_stop}\}, \{\text{sens\_active}\}\rangle\}$$

If we interpret elements of P as *positive examples*, and elements of N as *negative examples*, the *passive learning problem* consists in computing an LTL_f formula φ such that no $\pi \in N$ is a model of φ, and that all $\pi \in P$ are models of φ, i.e. the formula distinguishes between normal and anomalous traces. Given our examples, a possible solutions is $\varphi = \text{belt\_start} \rightarrow (\text{sens\_active } \mathsf{U} \text{ prod\_det}) \wedge \mathsf{F}(\text{belt\_stop})$, meaning that when the belt starts the sensor remains active until a product is detected and eventually the belt must stop.

However, current model-based techniques for passive learning face substantial challenges and state-of-the-art methods [21] rely on exhaustive search, evaluating one candidate formula at a time. In this paper, we propose the first inductive logic programming (ILP)-based approach for learning LTL_f formulae from positive and negative execution traces. In particular, we leverage the inductive learning of answer set programs (ILASP) system [28], a state-of-the-art learning from answer sets framework that has been shown to handle a wide variety of ILP tasks [27], including those that involve solving complex combinatorial optimization problems. ILP techniques have been prior applyed to learn process models in declarative languages such as DECLARE [1], where patterns are defined using LTL_f formulae [10]. However, unlike previous work, our method does not impose any syntactic restrictions to LTL_f formulae and does not limit the search to predefined patterns. We provide a novel encoding of the passive learning problem in answer set programming (ASP) [7], showing how it can be formulated as a learning task within the ILP framework. We compare our ILP-based approach to satisfiability (SAT) and satisfiability modulo theories (SMT) techniques, as implemented by the SySLite [3] system, and show that it can outperform them on benchmark datasets from cellular network attacks [2, 3].

Outline. The remainder of the paper is organized as follows. Section 2 introduces the background concepts related to LTL_f, ASP and learning from answer sets (LAS). In Section 3, we present an encoding for LTL_f formulas evaluation in a normal logic program. Section 4 formally defines the problem of LTL_f passive learning, including the necessary technical preliminaries. Section 5 discusses the encoding of the LTL_f passive learning in the CLINGO system. Section 6 presents our ILP-based approach using the ILASP framework, detailing the novel encoding we developed for the passive learning task. Section 7 reports the evaluation of our approach, comparing it with existing

SAT- and SMT-based methods on several datasets related to cellular network attacks. Section 8 discusses related work. Finally, Section 9 concludes the paper.

2 Background

In this section, we introduce LTL$_f$ and present the basic notions and terminologies of ASP and LAS used throughout the paper.

2.1 Linear temporal logic over finite traces

LTL [34] 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$_f$ [22] 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:

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

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*)$\mathsf{F}\varphi \equiv \mathit{true}\ \mathsf{U}\ \varphi$; (*Release*)$\varphi_1 \mathsf{R} \varphi_2 \equiv \neg(\neg\varphi_1\ \mathsf{U}\ \neg\varphi_2)$; and (*Always*) $\mathsf{G}\varphi \equiv \mathit{false}\ \mathsf{R}\ \varphi$. 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 $\varphi \in \mathcal{P} \cup \{\mathit{true}, \mathit{false}\}$; $|\circ\,\varphi| = 1 + |\varphi|$ 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:

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

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 LTL$_f$ formula is said to be in *neXt Normal Form* (`xnf`) [13, 29] when all its occurrences of U temporal operators are nested into some X operator. Note that, every LTL$_f$ formula can be transformed into an equivalent formula in `xnf` form in linear time by recursively applying the following transformations [29]:

- $\texttt{xnf}(\varphi) = \varphi$ for $\varphi \in \mathcal{P} \cup \{\mathit{true}, \mathit{false}\}$;
- $\texttt{xnf}(\neg\varphi) = \neg\texttt{xnf}(\varphi)$;

- $\texttt{xnf}(\varphi_1 \circ \varphi_2) = \texttt{xnf}(\varphi_1) \circ \texttt{xnf}(\varphi_2)$ for $\circ \in \{\wedge, \vee\}$;
- $\texttt{xnf}(\mathsf{X}\,\varphi) = \mathsf{X}\,\texttt{xnf}(\varphi)$;
- $\texttt{xnf}(\varphi_1 \,\mathsf{U}\, \varphi_2) = \texttt{xnf}(\varphi_2) \vee (\texttt{xnf}(\varphi_1) \wedge \mathsf{X}(\varphi_1 \,\mathsf{U}\, \varphi_2))$

2.2 Passive learning of LTL$_f$ formulae

The problem of passive learning of LTL$_\text{f}$ formulae, introduced in [33], refers to the challenge of automatically inferring LTL$_\text{f}$ 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 $\text{PL}_{LTL_f} = (\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$ 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$_\text{f}$ formula φ, also referred to as 'target formula', written in $\mathcal{P}$ such that (i) $\pi \models \varphi$, for every $\pi \in \mathcal{E}^+$; and (ii) $\pi \not\models \varphi$, for every $\pi \in \mathcal{E}^-$.

The complexity of passive learning has been analyzed within the context of LTL. It has been proved that, in the general case (where temporal operator can be used without restrictions but with a known upper bound on the size of the target formula) this problem is NP-complete [5]. Furthermore, the problem remains NP-complete in several syntactical fragments of the logic, as well as in some finite trace settings [15, 30][1].

In the setting of this paper, i.e. generic LTL$_\text{f}$ formulae with an upper bound on the size of the corresponding syntax tree, the passive learning problem remains in the NP complexity class. To establish this result, we derive an upper bound on the number of possible LTL$_\text{f}$ formulae of size $k \in \mathbb{N}$ by analyzing the number of binary trees of size k, where each node is labeled either by a propositional symbol, a Boolean connective or a temporal operator, effectively representing the syntax trees of these formulas. Determining whether a trace π satisfies a formula φ (encoded by a given syntax tree) can be done in polynomial time by applying the sematics rules shown in the previous section. This process corresponds to a top-down traversal of the formula's syntax tree, which can be performed in polynomial time with respect to the tree size and must be repreated for each positive and negative example trace. Consequently, non-deterministically guessing a formula and evaluating it across all example traces yields this problem within the NP complexity class. The ASP-based solutions we present in Section 4 mirrors this approach.

Note that, a PL$_{LTLf}$ passive learning task always accepts a *trivial* solution given by the formula $\phi = \bigvee_{\pi \in \mathcal{E}^+} \bigwedge_{i \in [0,\ldots,|\pi|-1]} \mathsf{X}^i(\bigwedge_{p \in \pi_i} p \wedge \bigwedge_{p \notin \pi_i} \wedge \neg p) \wedge \neg \mathsf{X}^{|\pi|} \mathit{true}$, 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 $\text{PL}_{LTL_f} = (\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$ 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 $|\varphi'| < |\varphi|$.

[1] We refer the interested reader to Section 3.1 of [5], which provides a thorough overview and comparison of the complexity results achieved in [30] and [5], as well as a detailed discussion of the differences between the underlying hypotheses and assumptions.

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

ASP [7, 20] is a knowledge representation formalism based on the stable model semantics of logic programs, which 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 [19].

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 `l{`$h_1, \ldots,$ `h`$_k$`}u` is called a *cardinality constraint*. In an aggregate `l` and `u` are integers and `h`$_i$, for `1` $\leq$ `i` $\leq$ `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[2]. 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 [11] 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 LAS paradigm [25, 28]. In ASP, the main 'output" of a program is a set of answer sets. So LAS takes as (positive and negative) examples *(partial) interpretations*, 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}.

[2] We use the simplified definitions of the *reduct* for choice rules presented in [26].

Many ILP systems (e.g. [24]) 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.[3] Given a mode bias $M = \langle M_h, M_b \rangle$, a rule R is 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 LAS* task, denoted as $\text{ILP}^{context}_{LAS}$, 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 $\text{ILP}^{context}_{LAS}$ tasks)
Let T be a $\text{ILP}^{context}_{LAS}$ learning task. A hypothesis H is an *optimal inductive solution* of T if and only 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

[3] 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.

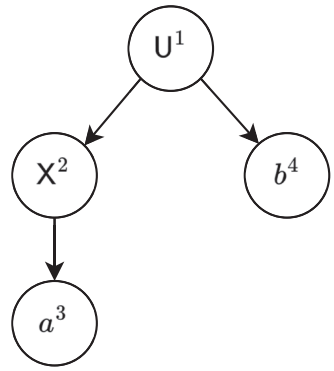

FIGURE 1 Syntax tree of the formula $\varphi = (\mathsf{X}a)\ \mathsf{U}\ b$. Superscripts in node labels represent the assigned node identifier.

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, $\pi^i \in \mathcal{E}^-$, e.g. 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 $\mathcal{E}$, i.e. $P(\mathcal{E}) = \bigcup_{\pi^i \in \mathcal{E}} P(\pi^i)$.

EXAMPLE 2.
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 [33] 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 that 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 3.
Consider the formula $(\mathsf{X}a)\ \mathsf{U}\ b$, depicted in Figure 1 shows its syntax tree. 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(·)` 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 $|\pi^i| = t$, i.e. π^i_t is the last state of π^i. The definition of these rules, which we denote by P_{LTL_f}, follows the `xnf(·)` 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:

LISTING 1.1 The logic program P_{LTL_f}

```
  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,_).
```

For values $t' > t$ of the second term of the atom *holds*/3, it is possible to represent subsequent instants of each `xnf(·)` 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) [9], 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 [12]. In particular, by observing that the rules implement the recursive application of `xnf(·)`, 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 with plain ASP

A first way to model the passive learning problem is to frame it as an abduction problem in ASP [14], 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:

LISTING 1.2 The logic program P_{tree}

```
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,RJ), id(I+1,RI), RI <= RJ.
```

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 breadth first search (BFS) traversal of the tree starting from the root node, which is adapted from [17] to our setting, which does not require labeled edges.

EXAMPLE 4. (Symmetry breaking constraint)
Consider the two complete binary trees on Figure 2. The left labeling conforms to BFS traversal order; the right one does not, cfr. nodes highlighted in red in the trees. By assigning to each node the additional identifier $v^2 + u$ (a *pairing function*, a bijection between $\mathbb{N}$ and $\mathbb{N}^2$, [38]), the symmetry breaking constraint discards the rightmost tree.

Program P_{tree} is then joined with the following program, named P_{label}.

LISTING 1.3 The logic program P_{label}

```
unary_label(neg; next; eventually; always).
binary_label(and; until; or).
```

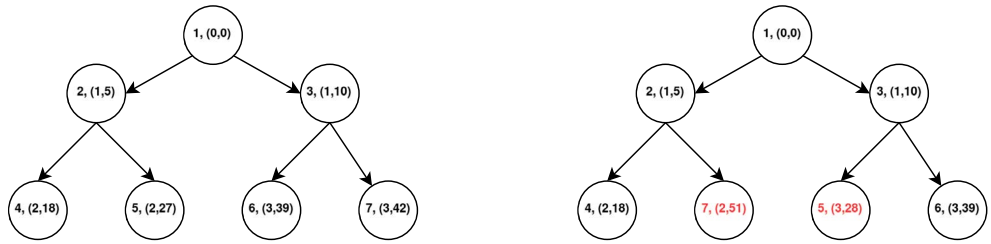

FIGURE 2 Example of BFS-conformant labeling of trees. Given a node labeled $(i,(j,k))$, the integer i is used for tree-generation purposes (referred to as 'node identifier'), while the pair (j,k) is an additional identifier that is used for symmetry breaking purposes, where j is the parent of i in the tree and $k = i^2 + j$.

```
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).
```

Each answer set of the resulting program, 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:

LISTING 1.4 The logic program P_{goal}

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

Thus, every answer set, projected onto the predicates `label/2`, `edge/2`, `node/1` and `order/3` as shown in Example 3, 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, i.e. a solution to the passive learning problem instance $(\mathcal{P}, \mathcal{E}^+, \mathcal{E}^-)$. A slight variation to P_{tree}, enables to find solutions of size *at most* n exploiting weak constraints:

LISTING 1.5 The logic program P_{tree} modified to compute solutions of size at most n

```
#const n.
node(0).
{ node(X): X=1..n }.
:- node(X), X > 0, not node(X-1).
:~node(X). [1@1,X]
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.
```

This version of the encoding searches over formulae of at most n nodes. The constraint enforces 'used node identifiers' to be successive integers, to avoid redundant syntax trees in the search space. The optimal answer sets identify smallest formulae matching the abduction goal.

5 LTL$_f$ passive learning with multi-shot CLINGO APIs

The encoding shown in the previous section requires the target size of the formula, n, to be provided in input to the ASP system. However, such n cannot be known beforehand; thus, solving the passive learning problem optimally requires multiple calls to an ASP system in an *iterative deepening-like* fashion, increasing the value of n at each call. Thus, it is required to ground the logic program P_{tree} multiple times, for increasing values of n. The optimization version of the encoding partially solves this problem, as it does not require knowing the exact size of the target formula but an upper bound. However, if the size of the solution is much smaller than the chosen n, it incurs in a overhead due both to grounding (e.g. reasoning over a large number of syntax trees) and solving (e.g. solving an optimization task rather than a decision task). Due to the recursive nature of trees — trees of size $n + 1$ are obtained by 'appending' a node to a leaf in a tree of size n —, such a problem is a natural candidate for multi-shot solving capabilities of the CLINGO system [23]. This enables for the incremental generation of trees rather than the up-front for a fixed size n. Thus, the first solution to be found will also be a minimal one, and optimal for our purposes.

We assume the reader to be familiar with the notions *incremental composition* of ASP programs as by [23]. Howerver, to help the reader going trough the description of the multi-shot encoding, we first recall some basics. The CLINGO system enables to partition a logic program into distinct subprograms, distinguished by the `#program` directive. Each subprogram is assigned a list of *parameters*, that act as placeholders in the subprogram's rules. The CLINGO APIs enable to instantiate programs by assigning values to parameters, and to separately ground program parts, incrementally adding them to the logic program being solved. One of the assumption underlying multi-shot solving and grounding, which has impact on rendering our ASP programs ameanable to multi-shot execution, is that *distinct subprograms cannot define an atom twice*, e.g. atoms cannot be in the head of rules from distinct subprograms.[4]

The encoding we propose in this section defines two subprograms, namely *search*(t) and *eval*(t). The *search*(t) subprogram encodes trees of increasing size, while *eval*(t) labels and evaluates trees according to the passive learning problem definition. Formulae and traces are encoded using the same as in the previous section.

5.1 Incremental generation of syntax trees

The logic program *search*(t) models the generation of trees of size t, assuming rules to model generation of trees of size $t - 1$ is alredy in the solver. Trees of size t are obtained by *appending*

[4] Theoretical underpinnings of ASP program composition are in [23].

a node to a tree of size $t-1$. Thus, we define *search*(t) as *choosing a parent node* among nodes $\{0,\ldots,t-1\}$ for the newly to-be-added node t:

LISTING 1.6 The program part *search*(t)

```
#program search(t).
node(t).
pair(X,t):- node(X), node(t), X < t.
1 { edge(X,t): pair(X,t) } 1:- node(t), t > 1.
:- node(X), 3 { edge(X,Y) }.
parent(t, (U,t*t+U)):- edge(U,t).
:- parent(I,RI), parent(J,RJ), I < J, RI >= RJ.
```

Similarly, the symmetry breaking constraint is introduced one-node-at-a-time, whenever *search*(t) is grounded with a specific value of t. In this version of the encoding, connectedness of the underlying graph structure is guaranteed since we are 'appending' the node t to a previously grounded node definition.

5.2 Incremental evaluation

In the tree generation phase, the issue of defining atoms twice can be naturally avoided, since node are assigned an unique, increasing identifier. The definition of program part *eval*(n), which assigns labels to the tree, evaluates it on the traces and constraints solutions, requires more care. In fact, evaluating the LTL_f expression requires rules involving *all the nodes in the tree*, and not just the newly added node. Hence, atoms *holds*(tid, t, x) modeling that the subformula rooted in x holds true at time t for the trace π^{tid} will be in the head of rules which span multiple subprograms.

To avoid this issue, we introduce an auxiliary term in the predicates *arity*/2, *order*/3, *holds*/3, *label*/2. The new term is an integer, which denotes the n-th *shot of evaluation.* The change is purely syntactical in nature, and each predicate keeps the same semantics as in the previous encoding. Thus, we obtain the subprogram *eval*(n):

LISTING 1.7 The program part *eval*(n)

```
#program eval(n).
#external shot(n).
:- shot(n), holds(n,TID,0,1), neg(TID).
:- shot(n), not holds(n,TID,0,1), pos(TID).
arity(n,X,C) :- node(X), C = #count{Y: edge(X,Y)}, shot(n).
1 { label(n,X,A): proposition(A) } 1:- shot(n), node(X),
arity(n,X,0).
1 { label(n,X,L): symbol(L,C) } 1:- shot(n), node(X),
arity(n,X,C), C > 0.
order(n,X,LHS,RHS):- shot(n), node(X), edge(X,LHS), edge(X,RHS),
LHS < RHS.
holds(n, TID, T, X):- label(n,X,A), trace(TID, T, A), shot(n).
holds(n, TID, T, X):- label(n,X,next), edge(X,Y),
        holds(n, TID, T+1, Y), not last(TID,T), trace(TID,T),
shot(n).
holds(n, TID, T, X):- label(n,X,until), order(n,X,LHS,RHS),
        holds(n,TID,T,RHS), trace(TID,T), shot(n).
```

```
  holds(n, TID, T, X):- label(n,X,until), order(n,X,LHS,RHS),
          holds(n,TID,T+1,X), holds(n,TID,T,LHS), trace(TID,T),
shot(n).
  holds(n, TID, T, X):- label(n, X, eventually), edge(X,Y),
          holds(n, TID, T, Y), shot(n).
  holds(n, TID, T, X):- label(n, X, eventually), holds(n, TID, T+1,
X),
          shot(n),trace(TID,T).
  holds(n, TID, T, X):- label(n, X, always), edge(X,Y),
last(TID,T),
          holds(n, TID, T, Y), shot(n).
  holds(n, TID, T, X):- label(n, X, always), edge(X,Y),
          holds(n, TID, T+1, X), holds(n, TID, T, Y), shot(n).
  holds(n, TID, T, X):- label(n,X,and), edge(X,Y1), edge(X,Y2),
          Y1 < Y2, holds(n, TID, T, Y1), holds(n, TID, T, Y2),
shot(n).
  holds(n, TID, T, X):- label(n,X,or), edge(X,Y),
          holds(n,TID,T,Y), shot(n).
  holds(n, TID, T, X):- label(n,X,neg), edge(X,Y), not holds(n,TID,
T,Y),
          trace(TID,T), shot(n).
  holds(n, TID, T, X):- label(n,X,implies), order(n,X,LHS,RHS),
          holds(n,TID,T,RHS), shot(n).
  holds(n, TID, T, X):- label(n,X,implies), order(n,X,LHS,RHS),
          not holds(n,TID,T,LHS), trace(TID,T), shot(n).
```

It is also necessary to *disable* constraints grounded (concerning the satisfiability/unsatisfiability over certain examples) in the previous evaluation shot. This is in order to avoid our incremental program becoming *permanently false*. We do so by means of the external atom *shot*(n). The atom *shot*(n) is set to True (from the API) whenever the *current* solving shot is the n-th, and to False otherwise. By setting *shot*(n) to permanently false, we clean up the solver from 'outdated' clauses, which were required to evaluate trees of size less than n.

6 LTL$_f$ passive learning with 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. In order to do so, we have to provide a suitable mode bias, background knowledge and example encodings. Figure 3 provides an (informal) graphical representation of such mapping.

Intuitively, the role of P_{label}, P_{tree} and P_{goal} — i.e. generating a labeled tree encoding a LTL$_f$ formula — is replaced by the mode bias of the learning task, as well as a special positive example (which we will refer to as e^*) that constraints the learned hypothesis. The rules to evaluate LTL$_f$ formulae are provided in the background knowledge. Finally, labeled traces are encoded into context-dependant examples.

In the following, we describe each section of the learning from answer set task.

Mode bias. We use the following mode bias, which we report in the input language of ILASP [25]. This specifies available operators (via the type `op`), the maximum size of the formula (via

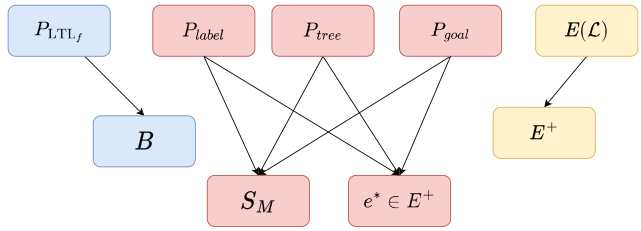

FIGURE 3 Mapping the plain ASP abductive approach into a learning from answer sets task.

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 3, which can be interpreted as the encoding of a LTL_f formula.

LISTING 1.8 Mode bias for the learning task.

```
#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 $a \in \mathcal{P}$. Since the evaluation of the inductive solution over examples is automatically handled by the ILASP system, it is not required anymore to provide different symbols to evaluate the formula over multiple traces in the encoding. Thus, with respect to the encoding described in Section 5, we drop the first term of the predicates `holds/3`, `last/2`, `sat/1`, `unsat/1`. Since this is only a syntactical transformation, we omit the full listing, which is available in Appendix A.2.

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}.

LISTING 1.9 Example e^* to constraint the learned hypothesis.

```
% 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).
```

Furthermore, by exploiting positive examples such as e^* it is possible to provide syntactical constraints on the formula being searched. This can be exploited to provide a 'syntactical bias' on the searched formula, e.g. searching for a solution matching a particular pattern.

EXAMPLE 5.
Suppose we are interested in learning formulae of the form $\mathsf{G}\,\varphi$. This amounts to adding to e^* facts that fix the formula structure $\mathsf{G}(\cdot)$, while leaving the system free to explore for φ. To do so, it is sufficient to add the fact $\mathit{label}(1, \mathit{always})$ to the background knowledge. If, on the other hand, we were interested in learning target formulae in *negation normal form*, i.e. where negation appears only above propositional terms, we could add the constraint:

```
:- label(X, neg), edge(X,Y), label(Y,Q), not proposition(Q).
```

that discards syntax trees where negation appears on top of which is not labeled with a propositional symbol.

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.

EXAMPLE 6.
A positive example consisting of the trace $\pi = \{a\} \cdot \{b\} \cdot \{c\}$ is encoded as follows:

```
#pos(0, {sat}, {},{
  trace(0).
  trace(0,a).
  trace(1).
  trace(1,b).
  trace(2).
  trace(2,c).
}).
```

The natural way to model negative traces is by using ILASP's `neg` qualifier. However, implementation-wise, our early experiments have shown that using ILASP's `#neg` qualifier caused severe performance issues.

Thus, we encode negative traces by means of positive examples on *complementary* inclusion sets, swapping out `sat` with `unsat` (which is defined as `not sat` in the background knowledge).

7 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, e.g. *time-invariants* of the system. A comprehensive description of each log is available in [3] and its technical report. We compare our ILASP-based solution with our plain ASP solution and multi-shot 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 [39]. We report execution time in seconds, with a timeout of 3600 s execution time on each event log. Currently, the most appropriate ILASP version for the task, due to incremental example processing, is ILASP 2i, which is the one we use to run the experiments.

Data. The dataset is composed by 18 logs: AKA Bypass (AKA), Authentication 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

TABLE 1 Summary statistics of the event logs used in the experiments. The columns represent the logs alphabet size $|\mathcal{A}|$ (i.e. number of distinct propositional symbols), total number of positive examples $|\mathcal{E}^+|$ and negative examples $|\mathcal{E}^-|$, total number of events $|\mathcal{L}|$ (i.e. the sum of the lenghts of the traces in the log) and the target formula size $|\varphi|$.

Event log	$\|\mathcal{A}\|$	$\|\mathcal{E}^+\|$	$\|\mathcal{E}^-\|$	$\|\mathcal{L}\|$	$\|\varphi\|$	**Event log**	$\|\mathcal{A}\|$	$\|\mathcal{E}^+\|$	$\|\mathcal{E}^-\|$	$\|\log \mathcal{L}\|$	$\|\varphi\|$
AKA	22	200	200	13014	7	AF	18	500	500	24024	3
BT	2	400	400	7947	5	CWP	2	500	500	7480	7
DSP	3	500	500	14700	10	EMM	19	500	500	36602	3
FI	2	500	500	9764	8	GLBA	3	500	500	15024	7
HIPPA-1	3	500	500	14945	7	HIPPA-2	3	500	500	9788	5
IM	21	500	500	28796	3	IMSI-1	20	500	500	27933	3
IMSI-2	26	500	500	39377	3	MR	27	500	500	47334	7
NE	21	500	500	26058	3	NA	20	500	500	15836	5
IMSI-3	27	500	500	38560	5	RLF	22	500	500	48301	7

(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) and RLF Report (RLF). Table 1 summarizes descriptive statistics about the event logs. 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 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.

Systems. In particular, we indicate by SySLite^L, $L \in$ {sygus, sat, 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 `MS Abduction` column refers to our (incremental) plain ASP encoding that uses abduction, while `SS Abduction` refers to the basic ASP encoding, exploiting minimization of *node*/1 rather than multi-shot evaluation. 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 (e.g., formula-size minimization, we assume the maximum size of the target formula to be known before-hand. This upper bound on solution size is also used in the iterative-deepening multi-shot approach, as well as in the plain abductive approach, and is set to 10, which is the least n required to find a solution for all event logs (with the exception of DSP). All systems support the same set of temporal logic operators $\{\mathsf{X}, \mathsf{U}, \mathsf{F}, \mathsf{G}, \wedge, \vee, \neg, \rightarrow\}$. We run the different systems on each event log, with an one hour timeout.

Results. The solution based on ILASP compares favorably with the algorithms implemented in SySLite, and outperforms it on the majority of event logs, as shown in Table 2. The scatter plot in Figure 4 reports a pair-wise comparison of the execution times of the different systems wrt ILASP's performance. As the majority of points lie above the bisector, we can see that overall the ILASP-based solution is the most effective on in this setting. This is confirmed by the cactus plot shown

TABLE 2 Execution time in seconds for compared methods over the event logs. Best execution time is in bold. T.O denotes timeout (execution time >3600 s).

Event log	SySLite^{sygus}	ILASP 2i	MS abduction	SySLite^{sat}	SySLite^{guided}_sat	SS abduction
AKA	144.2	**62.542**	612.31	T.O.	T.O.	1958
AF	**1.98**	3.32	4.705	360.7	T.O.	109
BT	2.17	**1.437**	13.239	659.23	133.19	41
CWP	22.01	**7.739**	148.891	T.O.	T.O.	471
DSP	T.O.	T.O	T.O	T.O.	T.O.	T.O.
EMM	**4.64**	5.78	7.207	1155.18	T.O.	2468
FI	51.59	**19.865**	510.189	T.O.	T.O.	793
GLBA	26.86	**13.606**	183.542	T.O.	T.O.	782
HIPPA-1	**25.96**	31.994	194.694	T.O.	T.O.	631
HIPPA-2	2.41	**1.693**	15.962	707.43	153.41	246
IM	4.04	4.483	5.837	977.77	T.O.	136
IMSI-1	3.72	**3.584**	4.877	774.34	T.O.	856
IMSI-2	6.89	**4.782**	8.092	1144.58	T.O.	2764
MR	995.85	**55.622**	T.O	T.O.	T.O.	T.O.
NE	**2.43**	4.301	4.706	480.18	T.O.	121
NA	2.54	**2.545**	2.8	1877.79	T.O.	731
IMSI-3	16.55	**11.652**	98.643	T.O.	T.O.	T.O.
RLF	560.44	**23.266**	T.O	T.O.	T.O.	T.O.

in Figure 5, that provides an aggregate picture of the overall performance across all the event logs. Overall, the multi-shot ASP solution is noticeably slower than the `sygus` SMT-based algorithm and ILASP alike. However, the multi-shot version improves on the 'single-shot' abduction with ASP.

Figure 6 shows a (linear) regression analysis that relates the systems' runtimes to the size of target formulae in each event log. All systems are affected by target formula size; an increase in target formula size causes an exponential effect on runtimes—which is expected, due to the nature of the problem and the solvers being based on SAT, SMT and ASP solvers. Appendix E reports regression analysis for other attributes reported in the summary statistics in Table 1, which do not affect runtimes as critically as the target formula size.

These results suggests that ILP techniques, in our case based on the ILASP system, might be a viable approach to scale beyond current model-based techniques without over-relying on pure enumerative approaches.

8 Related work

The seminal work [33] 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 [35], in order to improve

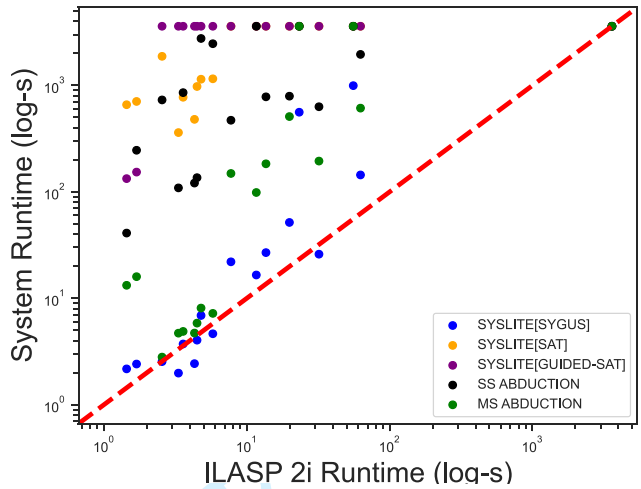

FIGURE 4 Pair-wise comparison of system's execution time, with respect to ILASP. A point (x, y) in the scatter plot means that a given passive learning task was solved in x seconds by ILASP and y seconds by the other system.

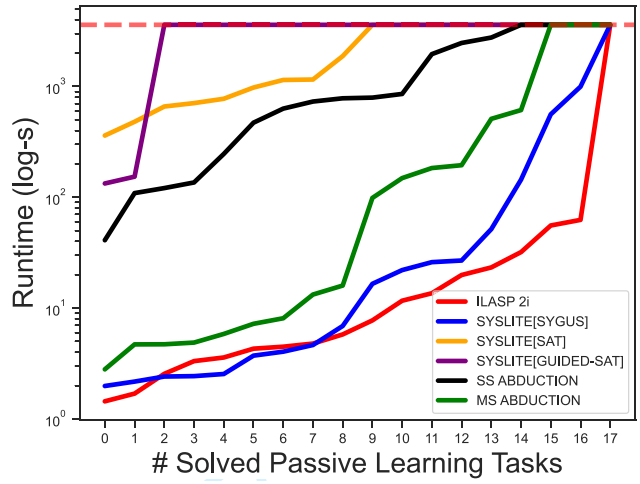

FIGURE 5 Comparison of system's execution time across all passive learning instances. A point (x, y) in the cactus plot means that a given system solves its x-th fastest instance in y seconds.

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* [8]), which can then be translated into LTL$_f$ formulae. The SySLite [3] 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 [36] as well as SMT-based *syntax-guided synthesis* [36] enumeration which exploits bit-vector theories for fast evaluation. Recently, an approach based solely on a highly optimized exhaustive search has been proposed [21] that

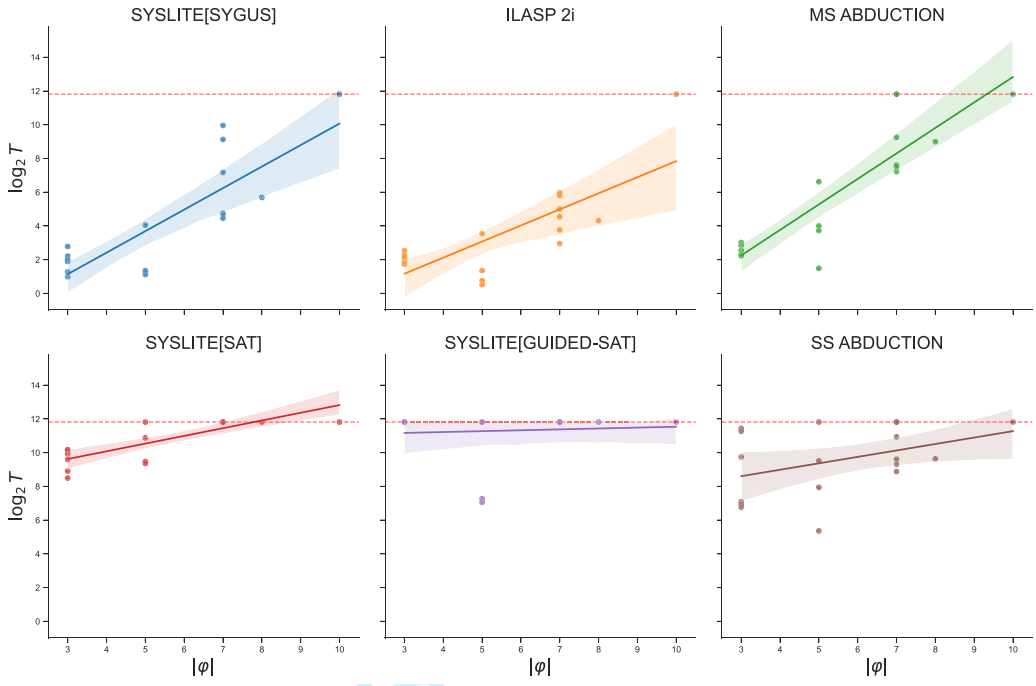

FIGURE 6 Regression analysis: Effect of target formula size on system (log-)runtimes.

enumerates formulae of increasing size and performs pruning on syntactic and semantic criteria on a single trace at a time. A direct comparison with [21] 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 [21]. Similarly, other works [40] focus on exploiting GPUs to improve scalability. Indeed, the passive learning problem has been explored also in similar formalisms, such as computational tree logic [6] and signal temporal logic [4]. Finally, learning specifications from positive examples only [37] can be considered a special case of passive learning (with no negative examples). From the theoretical standpoint, computational complexity-wise, the authors of [15] 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 [8] passive learning of arbitrary LTL_f formulae can be done with an exponential number of examples under some conditions.

9 Conclusion

In this paper, we presented an ILP approach based on the ILASP system for the passive learning of LTL_f 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 aim to improve the scalability of our proposed method, and possibly extend it to take into account data attached to the events that occur during the system's execution. A comparison with the approach of [21] is also in our plans to possibly demonstrate that there is an advantage versus exhaustive search methods. Another interesting extension we we will consider, which would extend the applicability of passive learning in real-world settings, is to apply our techniques to *noisy domains* [18, 32] (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 [31] could enhance the performance of the abductive approach, as we suspect that the number of symbols generated by evaluating candidate solutions over the event log is one of the causes of performance degradation.

Acknowledgements

This work was partially supported by MUR under PRIN project PINPOINT Prot. 2020FNEB27, CUP H23C22000280006; PRIN project HypeKG – CUP H53D23003710006; PRIN 2022 PNRR project DISTORT funded by European Union Next Generation EU Mission 4 Component 1 – CUP: H53D23008170001; 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).

References

[1] W. M. P. van der Aalst, M. Pesic and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, **23**, 99–113, 2009.

[2] M. F. Arif, D. Larraz, M. Echeverria, A. Reynolds, O. Chowdhury and C. Tinelli. Syslite github repository traces. https://github.com/CLC-UIowa/SySLite.

[3] M. F. Arif, D. Larraz, M. Echeverria, A. Reynolds, O. Chowdhury and C. Tinelli. *SYSLITE: Syntax-Guided Synthesis of PLTL Formulas from Finite Traces*, pp. 93–103. FMCAD, 2020.

[4] E. Bartocci, C. Mateis, E. Nesterini and D. Nickovic. Survey on mining signal temporal logic specifications. *Information and Computation*, **289**, 104957, 2022. https://doi.org/10.1016/j.ic.2022.104957. https://www.sciencedirect.com/science/article/pii/S0890540122001122.

[5] B. Bordais, D. Neider and R. Roy. The complexity of learning LTL, CTL and ATL formulas. In *42nd International Symposium on Theoretical Aspects of Computer Science (STACS 2025). Leibniz International Proceedings in Informatics (LIPIcs)*, Volume **327**, pp. 19:1–19:20, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. https://doi.org/10.4230/LIPIcs.STACS.2025.19.

[6] B. Bordais, D. Neider and R. Roy. Learning branching-time properties in CTL and ATL via constraint solving. In *Formal Methods. FM 2024*, A. Platzer, K. Y. Rozier, M. Pradella and M. Rossi, eds, Lecture notes in computer science, vol. **14933**. Springer, Cham, 2025. https://doi.org/10.1007/978-3-031-71162-6_16.

[7] G. Brewka, T. Eiter and M. Truszczynski. Answer set programming at a glance. *Communications of the ACM*, **54**, 92–103, 2011.

[8] A. Camacho and S. A. McIlraith. Learning interpretable models expressedin linear temporal logic. *Proceedings of the Twenty-Ninth International Conference on Automated Planning and*

Scheduling, ICAPS 2019,Berkeley, CA, USA, July 11-15, 2019, J. Benton, N. Lipovetzky, E. Onaindia, D.E. Smith and S. Srivastava, eds, pp. 621–630. AAAI Press, 2019. https://ojs.aaai.org/index.php/ICAPS/article/view/3529.

[9] S. Ceri, G. Gottlob and L. Tanca. *Logic Programming and Databases. Surveys in Computer Science*. Springer, Berlin, Heidelberg, 1990.

[10] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi and S. Storari. Exploiting inductive logic programming techniques for declarative process mining. *Transactions on Petri Nets and Other Models of Concurrency*, **2**, 278–295, 2009.

[11] A. Cropper and S. Dumancic. Inductive logic programming at 30: A new introduction. *Journal of Artificial Intelligence Research*, **74**, 765–850, 2022.

[12] E. Dantsin, T. Eiter, G. Gottlob and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, **33**, 374–425, 2001.

[13] C. Dodaro, V. Fionda and G. Greco. LTL on weighted finite traces: Formal foundations and algorithms. *Pro-ceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, L.D. Raedt, ed., pp. 2606–2612. ijcai.org, 2022. https://doi.org/10.24963/IJCAI.2022/361,10.24963/ijcai.2022/361.

[14] T. Eiter, G. Gottlob and N. Leone. Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science*, **189**, 129–177, 1997.

[15] N. Fijalkow and G. Lagarde. *The Complexity of Learning Linear Temporal Formulas from Examples*, pp. 237–250. ICGI, 2021.

[16] V. Fionda and G. Greco. LTL on finite and process traces: Complexity results and a practical reasoner. *Journal of Artificial Intelligence Research*, **63**, 557–623, 2018.

[17] D. Furelos-Blanco, M. Law, A. Jonsson, K. Broda and A. Russo. Induction and exploitation of subgoal automata for reinforcement learning. *Journal of Artificial Intelligence Research*, **70**, 1031–1116, 2021.

[18] J. Gaglione, D. Neider, R. Roy, U. Topcu and Z. Xu. Maxsat-based temporal logic inference from noisy data. *Innovations in Systems and Software Engineering*, **18**, 427–442, 2022.

[19] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. *Answer Set Solving in Practice*. Springer International Publishing, Cham, 2012.

[20] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, **9**, 365–385, 1991.

[21] E. Ghiorzi, M. Colledanchise, G. Piquet, S. Bernagozzi, A. Tacchella and L. Natale. Learning linear temporal properties for autonomous robotic systems. *IEEE Robotics and Automation Letters*, **8**, 2930–2937, 2023.

[22] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dy-namic logic on finite traces. In *IJCAI 2013, Proceedingsof the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, F. Rossi, ed., pp. 854–860. IJCAI/AAAI, 2013. http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

[23] R. Kaminski, J. Romero, T. Schaub and P. Wanko. How to build your own asp-based system?! *Theory and Practice of Logic Programming*, **23**, 299–361, 2023.

[24] M. Kazmi, P. Schüller and Y. Saygin. Improving scalability of inductive logic programming via pruning and best-effort optimisation. *Expert Systems with Applications*, **87**, 291–303, 2017.

[25] M. Law, A. Russo and K. Broda. The ILASP system for learning answer set programs . www.ilasp.com, 2015.

[26] M. Law, A. Russo and K. Broda. *Simplified reduct for choice rules in ASP*. In *Tech. Rep., Department of Computing (DTR2015–2)*, Imperial College London, 2015. https://www.doc.ic.ac.uk/research/technicalreports/2015/DTR15-2.pdf.

[27] M. Law, A. Russo and K. Broda. The complexity and generality of learning answer set programs. *Artificial Intelligence*, **259**, 110–146, 2018.

[28] M. Law, A. Russo and K. Broda. Logic-based learning of answer set programs. In *Reasoning Web. Explainable Artificial Intelligence*, M. Krötzsch and D. Stepanova, eds, Lecture notes in computer science, vol. **11810**. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-31423-1_6.

[29] J. Li, G. Pu, Y. Zhang, M. Y. Vardi and K. Y. Rozier. Sat-based explicit ltlf satisfiability checking. *Artificial Intelligence*, **289**, 103369, 2020.

[30] C. Mascle, N. Fijalkow and G. Lagarde. *Learning Temporal Formulas from Examples Is Hard*, 2023. https://arxiv.org/abs/2312.16336.

[31] G. Mazzotta, F. Ricca and C. Dodaro. *Compilation of Aggregates in ASP Systems*, *AAAI*, pp. 5834–5841. AAAI Press, 2022.

[32] A. Mrowca, M. Nocker, S. Steinhorst and S. Gunnemann. Learning temporal specifications from imperfect traces using bayesian inference. In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. p. 96. ACM, 2019. https://doi.org/10.1145/3316781.3317847.

[33] D. Neider and I. Gavran. Learning linear temporal properties. In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30- November 2, 2018*, N.S. Bjørner and A. Gurfinkel, eds, pp. 1–10. IEEE, 2018. https://doi.org/10.23919/FMCAD.2018.8603016.

[34] A. Pnueli. The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October -1 November 1977*. pp. 46–57. IEEE Computer Society, 1977. https://doi.org/10.1109/SFCS.1977.32.

[35] R. Raha, R. Roy, N. Fijalkow and D. Neider. Scalable anytime algorithms for learning fragments of linear temporal logic. *In: TACAS. Lecture Notes in Computer Science*, **13243**, 263–280, 2022. Cham, Springer International Publishing.

[36] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett and C. Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification. CAV 2019*, I. Dillig and S. Tasiran, eds, Lecture notes in computer science(), vol. **11562**. Springer, Cham, 2019. https://doi.org/10.1007/978-3-030-25543-5_5.

[37] R. Roy, J.-R. Gaglione, N. Baharisangari, D. Neider, Z. Xu and U. Topcu. Learning interpretable temporal properties from positive examples only. *Proceedings of the AAAI Conference on Artificial Intelligence*, **37**, 6507–6515, 2023. https://doi.org/10.1609/aaai.v37i5.25800.

[38] M. Szudzik. An elegant pairing function. In *Special NKS 2006 Wolfram Science Conference (pp. 1-12)*, Wolfram Research, ed, 2006. http://szudzik.com/ElegantPairing.pdf.

[39] O. Tange. *GNU Parallel 2018, March 2018*, 2018. https://doi.org/10.5281/zenodo.1146014.

[40] M. Valizadeh, N. Fijalkow and M. Berger. LTL learning on GPUs. In *Computer Aided Verification. CAV 2024*, A. Gurfinkel and V. Ganesh, eds, Lecture notes in computer science, vol. **14683**. Springer, Cham, 2024. https://doi.org/10.1007/978-3-031-65633-0_10.

Appendix A. ILASP encoding

This section provides full listings for the ILASP mode bias and background knowledge of the passive learning task.

A.1 Mode bias

```
#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)) ).
```

Additionally, a special positive example e^* is used to constraint the hypothesis space to well-formed LTL_f formulae:

```
% 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).
```

A.2 Background knowledge

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

Appendix B. ASP plain abduction (exact *n*)

This section provides full listing to model the passive learning task using ASP. In particular, this encoding assumes the target formula size to be known (e.g. its answer sets are LTL$_f$ formulae of size *n* that solve an instance of the passive learning problem).

```
#const 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.
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).
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,1,1).
unsat(TID):- trace(TID, _), not sat(TID).
  :- sat(TID), neg(TID).
  :- unsat(TID), pos(TID).
```

Appendix C. ASP plain abduction (upper bound on *n*)

This section provides full listing to model the passive learning task using ASP. In particular, this encoding assumes an upper bound size on target formula size to be known (e.g. its answer sets are LTL_f formulae of size at most *n* that solve an instance of the passive learning problem).

```
#const n.
size_range(1..n).
{ formula_size(X): size_range(X) } = 1.
node(X):- size_range(X), formula_size(K), X <= K.
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.
:~ node(X). [1@1,X]
...
```

The rest of the encoding (rules to label the tree and evaluate the resulting formula) is the same as in the previous section for the exact n version of the encoding.

Appendix D. ASP multi-shot abduction

This section provides full listing to model the passive learning task using multi-shot ASP. In particular, this encoding does not make assumptions about the target formula size, but explores syntax trees incrementally wrt their size.

```
  #program search(t).
node(t).
pair(X,t):- node(X), node(t), X < t.
  1 { edge(X,t): pair(X,t) } 1:- node(t), t > 1.
  :- node(X), 3 { edge(X,Y) }.
parent(t, (U,t*t+U)):- edge(U,t).
  :- parent(I,RI), parent(J,RJ), I < J, RI >= RJ.
  #program eval(n).
  #external shot(n).
  :- shot(n), holds(n,TID,0,1), neg(TID).
  :- shot(n), not holds(n,TID,0,1), pos(TID).
arity(n,X,C):- node(X), C = #count{Y: edge(X,Y)}, shot(n).
  1 { label(n,X,A): proposition(A) } 1:- shot(n), node(X),
arity(n,X,0).
  1 { label(n,X,L): symbol(L,C) } 1:- shot(n), node(X),
arity(n,X,C), C > 0.
order(n,X,LHS,RHS):- shot(n), node(X), edge(X,LHS), edge(X,RHS),
LHS < RHS.
holds(n, TID, T, X):- label(n,X,A), trace(TID, T, A), shot(n).
holds(n, TID, T, X):- label(n,X,next), edge(X,Y),
  holds(n, TID, T+1, Y), not last(TID,T), trace(TID,T), shot(n).
holds(n, TID, T, X):- label(n,X,until), order(n,X,LHS,RHS),
  holds(n,TID,T,RHS), trace(TID,T), shot(n).
holds(n, TID, T, X):- label(n,X,until), order(n,X,LHS,RHS),
  holds(n,TID,T+1,X),  holds(n,TID,T,LHS),  trace(TID,T), shot(n).
```

```
holds(n, TID, T, X):- label(n, X, eventually), edge(X,Y),
  holds(n, TID, T, Y), shot(n).
holds(n, TID, T, X):- label(n, X, eventually), holds(n, TID, T+1,
X),
     shot(n), trace(TID,T).
holds(n, TID, T, X):- label(n, X, always), edge(X,Y), last(TID,T),
    holds(n, TID, T, Y), shot(n).
holds(n, TID, T, X):- label(n, X, always), edge(X,Y),
  holds(n, TID, T+1, X), holds(n, TID, T, Y), shot(n).
holds(n, TID, T, X):- label(n,X,and), edge(X,Y1), edge(X,Y2),
  Y1 < Y2, holds(n, TID, T, Y1), holds(n, TID, T, Y2), shot(n).
holds(n, TID, T, X):- label(n,X,or), edge(X,Y),
     holds(n,TID,T,Y), shot(n).
holds(n, TID, T, X):- label(n,X,neg), edge(X,Y),
not holds(n,TID,T,Y), trace(TID,T), shot(n).
holds(n, TID, T, X):- label(n,X,implies), order(n,X,LHS,RHS),
  holds(n,TID,T,RHS), shot(n).
holds(n, TID, T, X):- label(n,X,implies), order(n,X,LHS,RHS),
  not holds(n,TID,T,LHS), trace(TID,T), shot(n).
```

D.1 Driver code

Here, we provide the companion Python code, which uses the multi-shot solving CLINGO APIs. Here, `ctl` consists of a `clingo.Control` instance [23]. The constants `SEARCH_LP` and `LOG` refer respectively to a source file containing the logic program in Listing 1.5 and 1.6.

```
ctl = clingo.Control([])
ctl.load(SEARCH_LP)
ctl.load(LOG)
ctl.ground([("base", [])])
continue_searching = True
max_tree_size = 20
tree_size = 1
while continue_searching and tree_size <= max_tree_size:
     #Ground SEARCH(n) and EVAL(n)
     ctl.ground([("search", [clingo.Number(tree_size)])])
     ctl.ground([("eval", [clingo.Number(tree_size)])])
     #Release SHOT(n-1)
     ctl.release_external(clingo.Function("shot",
[clingo.Number(tree_size-1)]))
     ctl.cleanup()
     #Assign true SHOT(n)
     ctl.assign_external(clingo.Function("shot",
[clingo.Number(tree_size)]), True)
     ans = ctl.solve(on_model=model)
     total_time_solve += solving_time
     if ans.satisfiable:
        return model
```

```
        tree_size+ = 1
if tree_size > max_tree_size:
    return False
```

Appendix E. Regression analysis

For completeness, we report (linear) regression analysis for the other event logs' attributes in Table 1, similarly to Figure 6.

Figures E7 and E8 show how the alphabet size and total number of processed events affect passive learning systems' performance in the experimental section. Overall, both these properties seem to contribute to the exponential growth in all systems' runtimes, but to a less critical extent than the target formula size.

Curiously, and somewhat counterintuitively, higher alphabet sizes seem to make the problem slightly easier in some systems. This depends on *which event logs the system timeouts on*. We provide an example of such behavior, focusing on the ILASP 2i plot. As shown in Table 2, ILASP 2i timeouts only on the DSP event log (with alphabet size 3), while it is able to solve all other tasks; this causes the corresponding regression line in Figure E7 to tilt downwards as the alphabet size increases.

With respect to the total number of events in the logs[5] , we see it indeed correlates with an increase in systems' runtimes. Here, concerning the ASP-based systems and ILASP, we observe that the

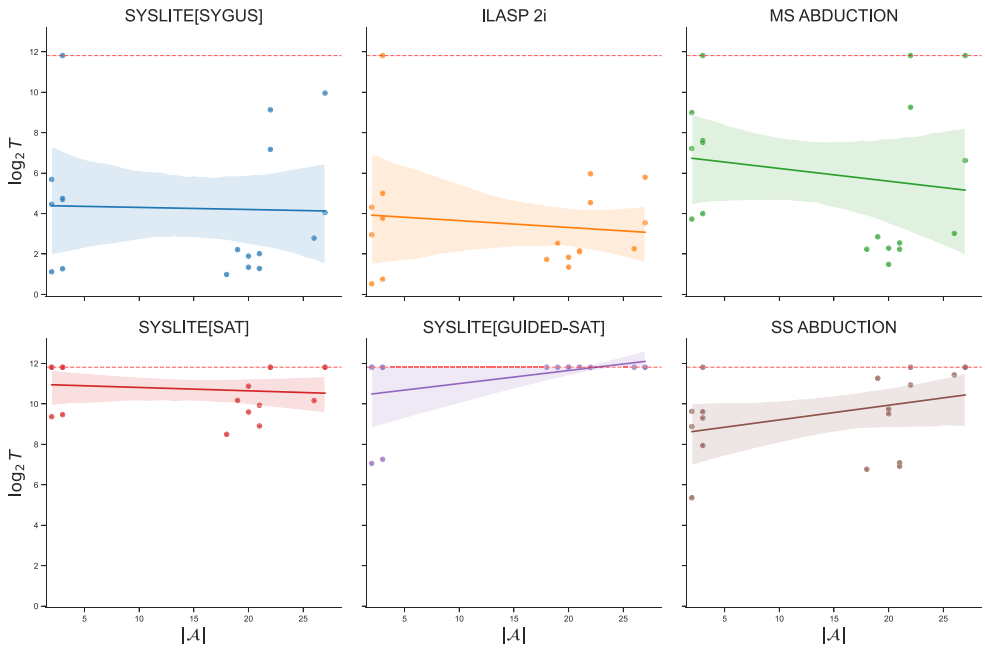

FIGURE E7 Regression analysis: Effect of log alphabet size (i.e. number of atomic events, propositional symbols) on system (log-)runtimes.

[5] All the event logs (except two) consist of 500 positive examples and 500 negative examples; thus, we focus on *total number of events*, the sum of lengths of the traces in the log, rather than number of positive and negative examples.

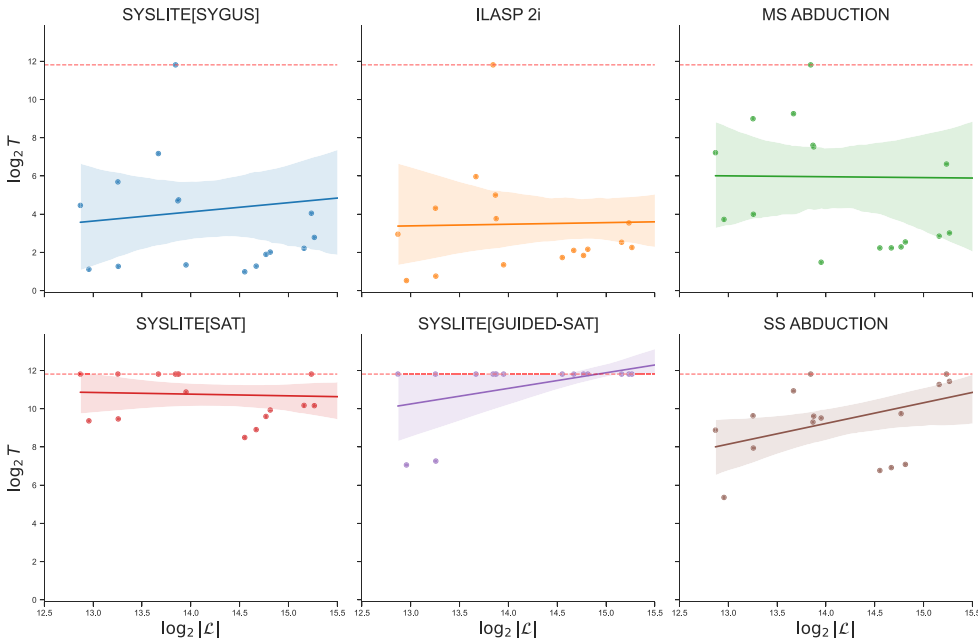

FIGURE E8 Regression analysis: Effect of log size (total number of events) on system (log-)runtimes.

single-shot abductive solution is most severely affected by the increasing log size. This matches our intuition: the single-shot abductive solution requires, as explained in Section C, to 'evaluate all examples in a single solve call'. The multi-shot encoding still requires all examples to be processed together, but the search space is built incrementally, so that if the event log admits a small solution (in terms of target formula size), the effect of the event logs' size is mitigated. Finally, ILASP 2i is the least affected by the increase in log size, since it processes examples one-by-one incrementally.

Received 5 November 2024