Learning Properties in LTL ∩ ACTL from Positive Examples Only

Rüdiger Ehlers*, Ivan Gavran†, and Daniel Neider†
*Clausthal University of Technology, Clausthal-Zellerfeld, Germany
Email: ruediger.ehlers@tu-clausthal.de
†Max Planck Institute for Software Systems, Kaiserslautern, Germany
Email: {gavran, neider}@mpi-sws.org

***Abstract*—Inferring correct and meaningful specifications of complex (black-box) systems is an important problem in practice, which arises naturally in debugging, reverse engineering, formal verification, and explainable AI, to name just a few examples. Usually, one here assumes that both positive and negative examples of system traces are given—an assumption that is often unrealistic in practice because negative examples (i.e., examples that the system cannot exhibit) are typically hard to obtain.**

To overcome this serious practical limitation, we develop a novel technique that is able to infer specifications in the form of universal very-weak automata from positive examples only. This type of automata captures exactly the class of properties in the intersection of Linear Temporal Logic (LTL) and the universal fragment of Computation Tree Logic (ACTL), and features an easy-to-interpret graphical representation. Our proposed algorithm reduces the problem of learning a universal very-weak automaton to the enumeration of elements in the Pareto front of a specifically-designed monotonous function and uses classical automaton minimization to obtain a concise, finite-state representation of the learned property. In a case study with specifications from the Advanced Microcontroller Bus Architecture, we demonstrate that our approach is able to infer meaningful, concise, and easy-to-interpret specifications from positive examples only.

I. Introduction

The engineering process of reactive systems requires a good understanding of the *specification* that the system should fulfill. For instance, while model checking can prove a system design to be correct with respect to a specification, the resulting proof is only meaningful if the specification captures the requirements of the application. Similarly, while the process of *synthesizing* reactive systems from their specifications is well-researched, both the system specification and the specification of the environment in which the system needs to operate must be correct in order for synthesis to be useful.

Writing correct and complete specifications is hard. Crucial properties of an application are easy to miss, and formalizing the specification as automata or in a logic such as linear temporal logic (LTL) is difficult. The problem can be addressed in multiple ways. Easier to use specification formalisms can support the writing process of specifications. Alternatively, approaches for *inferring* specifications from existing system implementations or examples can avoid the burden of manually writing the specifications. Such *specification learning* techniques are especially useful when a design is already available, so that its implicit specification can be documented by examining the set of its traces. Furthermore, a set of human-given examples may be available from which the wanted requirements should be distilled. Classical specification mining requires both examples that violate the (implicit) specification and examples that satisfy it, as with only one of these classes, either *false* or *true* can be valid specifications, making the problem ill-defined.

Unfortunately, both negative *and* positive examples are not always available. For instance, when inferring the specification of a system that is too big to be fully analyzed, but whose implementation is given, we can extract input/output traces that represent possible executions of the system. Proving that a certain input/output trace is not a possible execution of the system is however a model checking problem, which can be infeasible to solve for complex designs. Similarly, we may want to deduce an environment specification to be used in synthesis from observing the environment. For a black-box environment, we can never know that some behavior observation sequence cannot occur. These observations give rise to the question if there is some way to learn from positive (or negative) example traces only.

To make this problem well-posed, we need to introduce some kind of measure of how *tight* the specification should be that we want to obtain. For the case of positive examples, there is a spectrum of possible specification solutions ranging from *true* all the way to the specification that only allows exactly the set of traces in the example set. Both extremes make little sense, and a learning procedure to solve the problem should be parameterized by a *tightness value* n. At the same time, the intuitive idea of tightness with respect to some parameter n must be concretized in a way such that an efficient learning procedure to learn n-tight specifications can be given *and* we can observe that in practice, the learned specifications capture some relevant specification parts of systems while being easy enough to understand by an engineer.

In this paper, we give such a learning procedure for specifications from positive examples only. We identified *universal very-weak word automata* (UVWs) over infinite words as a specification representation that has a natural definition of tightness, lends itself to an efficient learning procedure, and leads to easily readable learned specifications. This automaton class has been identified as characterizing the class of properties representable both in linear temporal logic (LTL) and in

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_17

This article is licensed under a Creative Commons Attribution 4.0 International License

the universal fragment of computation tree logic (ACTL) [1]. While this implies that there are some ω-regular properties that cannot be learned by our framework, the intersection of LTL and ACTL includes the vast majority of specifications found in case studies on specification shapes [2]. By trading away the full ω-regular expressivity, we get multiple advantages that make learning from only positive examples feasible: UVWs can be decomposed into *simple chains* [3] that each represent a scenario and how the system satisfying the specification is required to react. Thus, they are easy to examine by a specification engineer. We will demonstrate that the maximum length of such a chain is also a natural notion of the complexity of a specification part, making it a good candidate for the concretization of the concept of tightness of a learned specification. Most importantly, simple chains have a natural approximation of language inclusion that enables us to efficiently learn a specification by enumerating all strictest chains that are not in contradiction to any example trace.

The algorithm for learning tight UVWs in this paper starts from a representation of the set of positive traces as *ultimately periodic words*, i.e., words of the form uv^ω for some finite words u and v. It is well-known that ω-regular specifications (or automata) are precisely characterized by the set of ultimately periodic words that satisfy the specification (or are included in the language of the automaton). Since ultimately periodic words can be encoded in a finite format, they are a natural choice of representation for the positive examples that are input to our algorithm.

We evaluate our approach on benchmarks from a case study on the AMBA AHB protocol [4]. Starting from LTL formulas describing the allowed behavior of the AMBA bus clients, we randomly generate sets of positive examples. We run our algorithm on the generated sets of different sizes and note how big the learned UVW is and how long it takes to compute it with our prototype implementation. Our experiments show that if the set of positive examples to learn from is big enough, the algorithm computes a UVW representation of the right LTL formula. The experiments also show that if too few positive examples are available, the UVWs grow quite large to capture the automaton language with the desired tightness value n.

Related Work

The problem of automata learning from data traditionally comes in two different settings: *active* [5]–[7] and *passive* [8]–[10]. In an active setting, the learning algorithm interacts with a *teacher*. The teacher answers two kinds of queries: membership queries (whether a proposed word is in the language of the automaton) and equivalence queries (whether a proposed automaton is the correct one). Learning stops once the teacher answers an equivalence query positively. Having a teacher that is able to answer equivalence queries is a strong assumption. Our work focuses on the passive setting, where the learning algorithm only has access to data, a set of classified examples.

The standard problem formulation of passive learning is that a sample consisting of positive and negative examples is given. For such a setup, several methods have been proposed for learning not only automata [9], [10], but also LTL formulas [11]–[13], or STL formulas [14], [15]. None of these methods provides good results when they are presented with only one class of examples—they return a trivial solution, one that accepts (or rejects) all possible examples.

Our problem—learning a specification from system traces—fits into the process mining framework (see Aalst et al. [16] for an overview): given an event log from a process, find a process model that satisfies certain properties. The properties are *fitness* (the model should be consistent with the examples from the log), *precision* (the model should not be overly general, e.g., modeling arbitrary examples), *generalization* (the model should not be overly tight, e.g., consistent only with the examples from the log), and *simplicity* (the model should be simple). Different operationalizations of the four properties give rise to different problem formulations and solutions. By choosing UVWs as our model, we get (structural) simplicity and connect it to the generalization property by the tightness value n, for which we require the tightest possible UVW consistent with the data.

Closely related to our approach is an algorithm by Avellaneda and Petrenko [17] for inferring deterministic automata over finite words (DFAs) from positive finite-word examples alone. Their algorithm searches for an automaton $\mathcal{A}$ with a given number of states n that is consistent with the given positive examples and for which no n-state DFA $\mathcal{A}'$ exists such that the language of $\mathcal{A}'$ is a strict subset of the language of $\mathcal{A}$. Both their approach and ours identify the language to be learned in the limit and use a single additional parameter for choosing the complexity of the language to be learned. Unlike in our approach, the resulting language in their algorithm is not unique for a given value of n. Furthermore, while our approach is relatively simple to adapt to the finite-word setting, their approach is difficult to adapt to the infinite-word setting, which we support in our work. This observation is rooted in the fact that their approach employs a SAT solver to search for candidate solutions, where clauses for the positive examples not found in previous solutions are added step-by-step. For the case of automata over infinite words, this requires the encoding of product runs between the deterministic automaton and the words to be accepted in SAT clauses (as described in [18], [19]). Every positive example requires additional clauses and variables, and for large numbers of positive examples, this easily leads to prohibitive sizes of the SAT instances.

Another direction of previous work is the identification of Live Sequence Charts (LSCs) [20], [21] from system runs. Live Sequence Charts [22] are a specification formalism that is popular for its compliance to the UML standard and the corresponding tools (e.g., IBM RSA). The set of properties representable as Live Sequence Charts, when not using free variables, was shown to be contained in the intersection of LTL and ACTL [23], which is characterized by UVWs (the version with free variables is characterized as a subset of first-order CTL* [24]). The existing work on mining LSCs [20], [21] borrows the concepts of *support* and *consistency* from

data mining [25]. With user-defined thresholds for support and consistency, charts are enumerated until one exceeding that threshold is found. Rather than giving more credibility to patterns occurring most often in the example traces (as it is the case when using the notion of support), our method prefers semantically stronger UVWs, controlled by their size. This lets our approach converge to the same property regardless of the distribution of the traces, as long as all traces (in the form of ultimately periodic words) have a non-zero probability of occurring.

A problem related to ours by the fact that the learning happens over (positive) demonstrations only, is inverse reinforcement learning [26]. There, however, it is the reward function that is being learned. Obtaining only the reward function does not provide a human-understandable task specification. Inspired by inverse reinforcement learning, Vazquez-Chanlatte et al. [27] learn LTL-like temporal specifications from demonstrations. In order to do so, they have to precompute the implication lattice between the possible specifications, which limits the applicability of their approach. This is not necessary in our work, as we take advantage of the syntactic approximation of language inclusion between simple chains of UVWs. On the other hand, they successfully handle noise in the sample.

II. Preliminaries

a) Basics: Given an alphabet Σ, the expression Σ^* represents the set of finite words with characters in Σ, and Σ^ω represents the set of words of infinite length in which each element is in Σ.

Let $\mathbb{B} = \{1, 0\}$ denote the set of Boolean values, with 1 representing **true** and 0 representing **false**. Moreover, let $S_1, \ldots, S_m$ be sets and $\sqsubseteq_i$ for $i \in \{1, \ldots, m\}$ be a partial order over the set S_i. Then, we call a function $f \colon S_1 \times \cdots \times S_m \to \mathbb{B}$ monotone if $s_i \sqsubseteq_i s'_i$ for each $i \in \{1, \ldots, m\}$ implies $f(s_1, \ldots, s_m) \leq f(s'_1, \ldots, s'_m)$. Adopting terminology from multicriterial optimization, we say that some tuple $(s_1, \ldots, s_m)$ is a *Pareto optimum* for f if $f(s_1, \ldots, s_m) = 1$ and for no $(s'_1, \ldots, s'_m) \neq (s_1, \ldots, s_m)$ with componentwise inequality $(s'_1, \ldots, s'_m) \leq (s_1, \ldots, s_m)$, we have $f(s'_1, \ldots, s'_m) = 1$. The set of Pareto optima is called the *Pareto front*. Likewise, we say that some tuple $(s_1, \ldots, s_m)$ is an element of the *co-Pareto front* if $f(s_1, \ldots, s_m) = 0$ and for all $(s'_1, \ldots, s'_m) \neq (s_1, \ldots, s_m)$ with $(s'_1, \ldots, s'_m) \geq (s_1, \ldots, s_m)$, we have $f(s'_1, \ldots, s'_m) = 1$.

b) Automata over infinite words: Given an alphabet Σ, an automaton over infinite words is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$, where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $Q_I \subseteq Q$ is a set of initial states, and F is a set of *final states*.

Given an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$, we say that $\mathcal{A}$ induces a *run* $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ if $\pi_0 \in Q_I$ and for every $i \in \mathbb{N}$, we have that $(\pi_i, a_i, \pi_{i+1}) \in \delta$. An automaton defines a *language* $\mathcal{L}(\mathcal{A})$, i.e., a subset of Σ^ω that it *accepts*. *Universal co-Büchi automata* accept all words w for which all (infinite) runs $\pi = \pi_0 \pi_1 \ldots$ induced by the word w visit states from F only finitely often, i.e., there exists an $i \in \mathbb{N}$ such that for every $j \in \mathbb{N}$ with $i \leq j$ we have $\pi_j \notin F$. The final states are also called *rejecting* states in this case.

Another type of automaton over infinite words are *non-deterministic Büchi automata*, which accept all words that have runs that visit F infinitely often. Such an automaton is furthermore called *deterministic* if for each $(q, a) \in Q \times \Sigma$, we have at most one $q' \in Q$ with $(q, a, q') \in \delta$.

We say that a automaton is *one-weak* or *very weak* if there exists a ranking function $r \colon Q \to \mathbb{N}$ such that for every $(q, a, q') \in \delta$, we have that either $r(q') < r(q)$ or q and q' are identical. More intuitively, this means that all loops in $\mathcal{A}$ are self-loops.

c) Linear Temporal Logic: The logic LTL (Linear Temporal Logic) [28] extends propositional Boolean logic with temporal modalities, which allow reasoning about sequences of events. Formulas of LTL are inductively defined as follows:

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

As syntactic sugar we add to the set of formulas $true$, $false$, $\psi \wedge \varphi$ and $\psi \rightarrow \varphi$, which are defined as usual for propositional logic. Moreover, we add the derived temporal operators $\mathsf{F}\,\psi :=$ $true \,\mathsf{U}\, \psi$ ("finally") and $\mathsf{G}\,\psi := \neg\,\mathsf{F}\,\neg\psi$ ("globally").

The semantics of propositional operators is defined as usual and here we describe the semantics of temporal modalities.

An LTL formula over some set of atomic propositions AP is evaluated on a word $w = a_0 a_1 \ldots \in (2^{\mathsf{AP}})^\omega$ and a time point $i \in \mathbb{N}$ of the sequence.

- $w, i \models p$ for $p \in \mathsf{AP}$ if $p \in a_i$
- $w, i \models \mathsf{X}\,\varphi$ if $w, i+1 \models \varphi$
- $w, i \models \varphi \,\mathsf{U}\, \psi$ if $\exists j \geq i.\ w, j \models \psi$ and $\forall k. i \leq k < j \Rightarrow w, k \models \varphi$

d) Universal very weak automata: Universal very-weak automata (UVW) are universal co-Büchi automata that are also very-weak. While universal co-Büchi automata are as expressive as Linear Temporal Logic (LTL) [29], universal very-weak automata are less expressive and only capture the properties whose satisfaction by a reactive system can be expressed both in computational tree logic with only universal path quantifiers (ACTL) and linear temporal logic [1], [30].

The language represented by a finite ω-*automaton* (such as a UVW) is uniquely determined by the set of *ultimately periodic words* uv^ω with $u, v \in \Sigma^*$ in the language of the automaton.

A universal very-weak automaton can be decomposed into *simple chains* [3], i.e., such that no state is directly reachable from more than one other state (apart from possibly itself). More formally, a simple chain is a sequence of different states $q_1, \ldots, q_n$ such that for all $i \in \{1, \ldots, n-1\}$, there exists some $a \in \Sigma$ with $(q_i, a, q_{i+1}) \in \delta$.

A simple chain is called *longest* (or *maximal*) in an automaton if it cannot be extended by an additional state at the beginning or the end of the sequence without losing the property that it is contained in the automaton. We say that a UVW is in decomposed form if there are no transitions

between the maximal simple chains of the UVW and for every such simple chain $q_1, \ldots, q_n$, there are no "jumping transitions", i.e., for no $i, j \in \mathbb{N}$ and $a \in \Sigma$, we have $(q_i, a, q_j) \in \delta$ with $j > i + 1$. Without loss of generality, we can assume that in a decomposed UVW, every chain has an initial state and the last state is rejecting, as otherwise the whole chain or the last state, respectively, can be removed.

III. Learning Universal Very-Weak Automata

In this section, we describe our approach to learn universal very weak automata (UVW) from positive examples alone. We first define the notion of n-tightness of a UVW, which specifies what languages we want to learn from positive examples alone. We prove that the languages of n-tight automata are unique, which ensures that the learning problem is well-posed.

We then establish in Section III-B how the simple chains of n-tight automata can be learned. As per the acceptance definition of UVW, the chains describe the words to be rejected. Hence, learning n-tight automata amounts to enumerating all simple chains of length up to n that do not reject any of the positive examples. We show that enumerating them all can be posed as the problem of enumerating the *co-Pareto front elements* of a monotone function.

In Section III-C, we then show how this insight leads to an efficient learning process: we show that the monotone function can be evaluated by solving a relatively simple model checking problem, and for enumerating all chains, we can use a Pareto optima enumeration algorithm from existing work, which outputs the co-Pareto front as a byproduct. To obtain reasonably-sized UVW, the last step is then to run the usual simulation-based automaton minimization steps.

A. Defining tight universal very-weak automata

Given a set of *positive* examples $P \subset \Sigma^\omega$, we want to compute (learn) an automaton $\mathcal{A}$ such that $P \subseteq \mathcal{L}(\mathcal{A})$, where we assume that for each $p \in P$, we have that $p = u_p(v_p)^\omega$ for some finite words $u_p, v_p \in \Sigma^*$ with $|v_p| \geq 1$.

Since there are infinitely many automata $\mathcal{A}$ satisfying this condition, we need an optimization criterion for finding the automaton $\mathcal{A}$. Minimizing the number of states of the solution is not a meaningful optimization criterion in this context, as the smallest automaton is always the one with 0 states – such an automaton does not visit final states, and by the acceptance definition of UVW, this means that all words are accepted.

To permit learning from positive examples only, we hence define an alternative learning criterion: we learn the strictest automaton (i.e., with the smallest language) that satisfies some syntactic cut-off criterion. For UVWs, there is a natural such criterion: the size of the co-domain of the ranking function, or equivalently, the length of the longest chain in a UVW.

Definition 1: Let $P \subset \Sigma^\omega$ be a set of positive examples and $\mathcal{A}$ be a UVW with $\mathcal{L}(\mathcal{A}) \supseteq P$. We say that $\mathcal{A}$ is n-tight for some $n \in \mathbb{N}$ if the following conditions hold:

1) There does not exist a simple chain of states longer than n in $\mathcal{A}$ (or equivalently, there exists a ranking function proving the very-weakness with co-domain $\{1, \ldots, n\}$),
2) For no other UVW $\mathcal{A}'$ with $P \subseteq \mathcal{L}(\mathcal{A}') \subset \mathcal{L}(\mathcal{A})$, we have that all simple chains of states in $\mathcal{A}'$ are of length at most n.

Lemma 1: Given a set of positive examples P and some value $n \in \mathbb{N}$, there exists an n-tight UVW $\mathcal{A}$ with $P \subseteq \mathcal{L}(\mathcal{A})$. All other n-tight UVWs have the same language.

Proof: We construct a universal very weak automaton in its decomposed form, i.e., where the UVW consists of a finite set of simple chains without transitions between them. Let C be a set of all possible simple chains of length up to n. We ignore the state identities/names, so that a chain of length n is characterized completely by transitions between the states, of which there are fewer than $2^{|\Sigma| \cdot (2n-1)}$ many different ones (as there can be at most $2^{|\Sigma|}$ different self-loops on n states and fewer than $2^{|\Sigma| \cdot (n-1)}$ many transitions between different states). This makes the set C finite. We choose an automaton $\mathcal{A}$ to consist of the set of all chains $c \in C$ such that $P \subseteq \mathcal{L}(c)$. We claim that this is an n-tight UVW accepting all words from P and that its language is the language of all n-tight automata.

Indeed, for a tighter UVW $\mathcal{A}'$ (i.e., such that $P \subseteq \mathcal{L}(\mathcal{A}') \subset \mathcal{L}(\mathcal{A})$) with maximal chain length n, there must exist $\alpha \in \Sigma^\omega$ such that $\alpha \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{A}')$. The fact that $\alpha \notin \mathcal{L}(\mathcal{A}')$ means that a run of $\mathcal{A}'$ on α will end up in one of its final (rejecting) states going through a chain of up to n states. But by $P \subseteq \mathcal{L}(\mathcal{A}')$ and by our definition of $\mathcal{A}$, this chain should be a part of $\mathcal{A}$. Therefore, $\alpha \notin \mathcal{L}(\mathcal{A})$, which yields a contradiction.

Let now $\mathcal{A}$ and $\mathcal{A}'$ be two n-tight automata. If they are not equivalent, then there exists a word $\alpha \in \Sigma^\omega \setminus P$ accepted by one of them but not by the other. Wthout loss of generality, let $a \notin \mathcal{L}(\mathcal{A})$. Since all chains in $\mathcal{A}$ and $\mathcal{A}'$ are of length at most n, this means that the word is rejected by one of such chains in $\mathcal{A}$. As the chain can be added to $\mathcal{A}'$ without making it reject a word in P, this proves that $\mathcal{A}'$ is not n-tight, yielding a contradiction. Hence, the assumption that the two automata $\mathcal{A}$ and $\mathcal{A}'$ are not equivalent but both n-tight cannot be fulfilled. □

The lemma shows that for a given set of positive examples P, n-tight automata have a unique language. It also shows how such an automaton can be computed: we first enumerate all simple chains of length n that a decomposed automaton accepting all elements from P could have. Taking these chains together, we obtain an n-tight UVW.

B. Enumerating All Simple Chains of a UVW to be Learned

The n-tightness definition of the previous subsection states what language the automaton that we want to learn from a set of positive examples should have. However, enumerating all simple chains that are consistent with the given positive examples is computationally inefficient as their number grows exponentially with n and the size of the alphabet. We show in this subsection how this problem can be mitigated.

To do so, we represent simple chains syntactically by so-called *chain strings*. Then, we define a partial order over these strings that is consistent with language inclusion between automata consisting only of the represented chains. In order

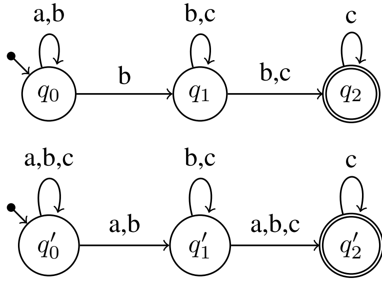

Fig. 1. Two example simple chains, where the lower one is syntactically stronger than the first one.

to obtain n-tight UVWs, we then only need to enumerate all chain strings that are *strongest* according to this partial order.

We visualize this idea in Figure 1 for the case of $n = 3$ and $\Sigma = \{a, b, c\}$. The simple chains given there are represented by the chain strings $(\{a, b\}, \{b\}, \{b, c\}, \{b, c\}, \{c\})$ and $(\{a, b, c\}, \{a, b\}, \{b, c\}, \{a, b, c\}, \{c\})$, which denote the edge labels along the chain, alternating between self-loops and edges between states. Assuming that both chains are compatible with some set of positive examples over the alphabet $\Sigma = \{a, b, c\}$, the lower one is *stronger* than the upper one in the sense that it rejects strictly more words.

This can be seen from the fact that both chains have the same length, and at each self-loop and each edge between the states, the labels for the lower chain are supersets of the respective labels for the upper automaton. On the chain string level, we can easily see that by looking at every pair of elements in the string and comparing the respective sets for set inclusion. Hence, every rejecting run for the upper chain is a rejecting run for the lower one as well. Chain strings induce a natural order by element-wise inclusion, and as already mentioned, the main idea of our approach is to enumerate only the largest chain strings with respect to the partial order that are consistent with the set of positive examples, which decreases the number of chains to be enumerated.

To simplify the presentation henceforth, the formal chain string definition also permits interrupted chains of states, which are not simple chains according to their definition in Section II. Furthermore, we only care about chains in which exactly the first state is initial and exactly the last state is rejecting. The generality is, however, not lost: if a simple chain does not have this form (so it has additional initial or rejecting states), then it contains another shorter simple chain of this form. This shorter simple chain can be extended to a chain of length n by duplicating the last (rejecting) state and rerouting the outgoing transitions of the previously last state to the new last state. This yields another chain of length n that is not missed when enumerating *all* maximal (w.r.t. their partial order) chain strings of length n that are compatible with P according to the definitions to follow. Figure 2 depicts this observation. The leftmost chain is split into a chain for the rejecting state q_2 and a chain for the rejecting state q_3. The now shorter chain is post-processed to a longer chain by duplicating the last state.

Definition 2: Let Σ and n be given. A *chain string* for Σ and n is of the form $s = (l_1, m_1, l_2, m_2, \ldots, l_n) \in (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma$. Such a string s induces a chain-like automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$ with

- $Q = \{q_1, \ldots, q_n\}$;
- $Q_I = \{q_1\}$;
- $\delta = \{(q_i, x, q_i) \mid x \in l_i, i \in \{1, \ldots, n\}\} \cup \{(q_i, x, q_{i+1}) \mid x \in m_i, i \in \{1, \ldots, n-1\}\}$; and
- $F = \{q_n\}$.

Note that the induced automaton $\mathcal{A}$ consists of at most one single simple chain that is reachable from an initial state.

The main idea of the following enumeration procedure is to cast the problem of finding all strongest simple chains as a problem of finding the *co-Pareto front* of a monotonous function f_n over chain strings. This enables the use of a Pareto front enumeration algorithm [31] for monotone functions to enumerate all simple chains that are consistent with the given positive examples.

The said algorithm however finds the Pareto front elements of a rectangular finite subset of $\mathbb{N}^u$ for some $u \in \mathbb{N}$. To make it compatible with the problem of finding simple chains, we have to encode chain strings into $\mathbb{N}^u$. The fact that all chain string elements are powersets enables a relatively simple encoding. We set $u = |\Sigma| \cdot (2n - 1)$ and for every chain string $s = (l_1, m_1, l_2, m_2, \ldots, l_n) \in (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma$, the corresponding encoded string in $\mathbb{N}^u$ is of the form $s' = (l_1^1, \ldots, l_1^{|\Sigma|}, m_1^1, \ldots, m_1^{|\Sigma|}, l_2^1, \ldots, l_2^{|\Sigma|}, \ldots, l_n^1, \ldots, l_n^{|\Sigma|})$, where every every element l_j^i and m_j^i is either 0 or 1, depending on whether the ith element of Σ is part of the encoded l_j. The order of the elements of Σ used in this encoding is arbitrary but fixed.

A Pareto-front enumeration algorithm necessarily also enumerates the co-Pareto front to be sure it found all Pareto front points [31], which we exploit to find all strongest chain strings, as these form the co-Pareto front. The monotone function itself implements a *model checking* step of all elements in P against the chain, which due to the lasso-like structure of the examples is relatively easy to solve.

Lemma 2: Let P be a set of positive examples over the alphabet Σ, $n \in \mathbb{N}$, and $f_n \colon (2^\Sigma \times 2^\Sigma)^{n-1} \times 2^\Sigma \to \mathbb{B}$ be a function that maps a chain string over Σ and n to 1 if and only if the automaton induced by the string rejects some element in P. Then, the function f_n is monotone.

Proof: Let $s = (l_1, m_1, l_2, \ldots, m_{n-1}, l_n)$ and $s' = (l'_1, m'_1, l'_2, \ldots, m'_{n-1}, l'_n)$ be two chain strings with $l_i \subseteq l'_i$ for each $i \in \{1, \ldots, n\}$ and $m_i \subseteq m'_i$ for each $i \in \{1, \ldots, n-1\}$. Furthermore, let $\mathcal{A}_s = (Q_s, \Sigma, \delta_s, Q_{I,s}, F_s)$ and $\mathcal{A}_{s'} = (Q_{s'}, \Sigma, \delta_{s'}, Q_{I,s'}, F_{s'})$ be the corresponding UVWs as in Definition 2 with $Q_s = Q'_s = \{q_1, \ldots, q_n\}$.

As $l_i \subseteq l'_i$ for each $i \in \{1, \ldots, n\}$ and $m_i \subseteq m'_i$ for each $i \in \{1, \ldots, n-1\}$, by the fact that by Definition 2, the transition relation of $\mathcal{A}_s$ is monotone in $l_1 \ldots l_n$ and $m_1 \ldots m_n$, we have that $\delta_s \subseteq \delta_{s'}$. Hence, every run π of $\mathcal{A}_s$ for some word $w \in \Sigma^\omega$ is also a run of $\mathcal{A}_{s'}$ for the same word.

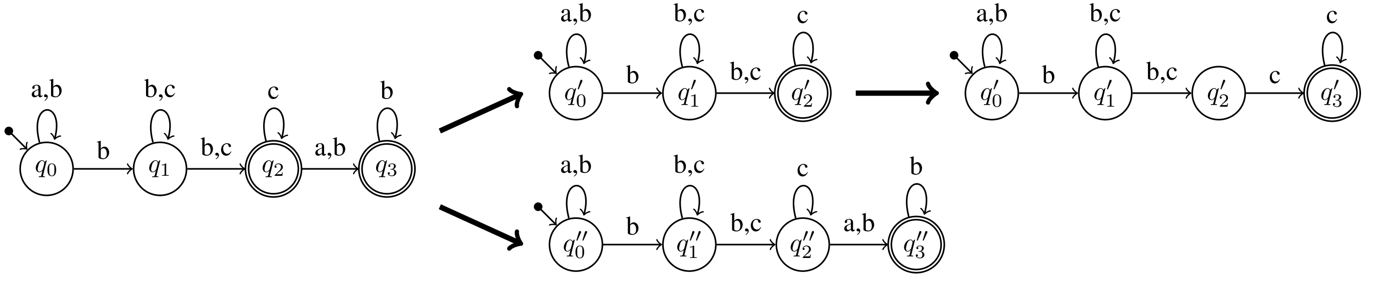

Fig. 2. Splitting a chain with multiple rejecting states

As universal automata accept all words that do not induce any rejecting run, this means that all words rejected by $\mathcal{A}_s$ will also be rejected by $\mathcal{A}_{s'}$, and hence, we have $\mathcal{L}(\mathcal{A}_{s'}) \subseteq \mathcal{L}(\mathcal{A}_s)$.

To show that f_n is monotone, recall that the function f_n maps a chain string t to whether the UVW $\mathcal{A}_t$ rejects a word in P. Towards a contradiction, assume that $f_n(\mathcal{A}_s) = 1$ but $f_n(\mathcal{A}_{s'}) = 0$. This means that there exists a word $w \in P$ such that $w \notin \mathcal{L}(\mathcal{A}_s)$ and $w \in \mathcal{L}(\mathcal{A}_{s'})$. Thus, w witnesses $\mathcal{L}(\mathcal{A}_{s'}) \not\subseteq \mathcal{L}(\mathcal{A}_s)$, which is a contradiction to the previous part of the proof. In conclusion, we obtain that f_n is monotone. □

We obtain the following corollary:

Corollary 1: Let P be a set of positive examples over the alphabet Σ, $n \in \mathbb{N}$, and A be the set of automata induced by the co-Pareto front elements of the function f_n. The automaton for the language $\bigcap_{\mathcal{A} \in A} \mathcal{L}(\mathcal{A})$ is n-tight for P and Σ.

The automaton from this corollary can be built easily, as universal very-weak automata are closed under language intersection by just merging the state sets, transition relations, and initial states [3]. This enables us to simply merge all chains found together into a single UVW.

C. Engineering Considerations of the Learning Algorithm

After the co-Pareto front of strongest chains has been enumerated, the last step in the construction of the UVWs is merging them to a single UVW. We add the chains one-by-one to a solution UVW. After every such step, we use the automaton minimization techniques described in [2] to reduce the size of the automaton. If the process is stopped prematurely, the result is still useful—a UVW that accepts a subset of the language that the final automaton (given sufficient computation resources) would accept. This property makes it possible to use the algorithm in the *anytime* fashion, stopping it when a given resource budget is exceeded.

It remains to be described how f_n can be computed efficiently. We implemented this process as follows: let $P = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ and $\mathcal{A} = (Q, \Sigma, \delta, Q_I, F)$ with $Q = \{q_1, \ldots, q_n\}$ be an automaton induced by a chain string to be checked. For every $j \in \{1, \ldots, m\}$, we translate (u_j, v_j) to a deterministic Büchi automaton $\mathcal{A}'$ accepting exactly $u_j(v_j)^\omega$. Such an automaton has $|u_j| + |v_j| + 1$ states. We then check if $\mathcal{A}'$ admits a word rejected by $\mathcal{A}$, i.e., if $\mathcal{L}(\mathcal{A}') \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$. Since the complement of a universal co-Büchi word automaton can be obtained in the form of a non-deterministic Büchi automaton by just interpreting $\mathcal{A}$ as such, the standard product construction from linear-time model checking can be applied to test if $\mathcal{L}(\mathcal{A}') \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$. The function f_n can then simply iterate over all examples $j \in \{1, \ldots, m\}$ and test if this is the case for any of them. Whenever it finds that $\mathcal{L}(\mathcal{A}') \cap \overline{\mathcal{L}(\mathcal{A})} \neq \emptyset$ for some automaton $\mathcal{A}'$ built from a positive example, the function f_n returns 1. Otherwise, it returns 0 after iterating through all values for $j \in \{1, \ldots, m\}$.

Note that in an actual implementation of f_n, there is no need to explicitly build $\mathcal{A}'$ or construct the product Büchi automaton. Rather, the implementation can make use of the fact that only the last state of the simple chain is rejecting. So it can compute the states of the product that are reachable and then check if state q_n in the $\mathcal{A}$ component of the product is reachable while at the same time, all characters in v_j are contained in the self-loop label of state q_n. If and only if that is the case, positive example number j is rejected by $\mathcal{A}$.

IV. Empirical Evaluation

We implemented the approach from this paper in a prototype toolchain [32], which is available on Github. The enumeration of the simple chains is performed by a tool written in `C++`, while the subsequent minimization of the resulting UVWs is implemented in `Python 3`.

In order to assess the performance of our approach on practically relevant properties, we considered the specification of the industrial on-chip bus arbiter of the AMBA AHB bus [33]. Specifically, we considered ten assumptions made for the master of the AHB bus, as described in [4]. For simplicity we abstracted from the concrete variable names and rewrote predicates over categorial values into individual propositions. For instance, the original property A8 from [4] referring to a burst sequence of unspecified length (denoted by the value INCR) is $\mathsf{G}[\text{HLOCK} \wedge (\text{HBURST} = \text{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}(\neg\text{REQ_VLD})]$. It is rewritten into $\mathsf{G}[(a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}(\neg c)]$. All the resulting formulas are shown on the left-hand-side of Table I.

Except for Property 3, all properties can be represented in UVW form by a single simple chain with two states each. For Property 3, we need two chains of length 3. The properties employing *two to four* atomic propositions have been learned over words with characters that encode this number of propositions. Propery 6 has been learned over positive examples in which each character has three proposition values, while for Property 9, we used two propositions. This deviation was necessary to

TABLE I
MEAN COMPUTATION TIMES FOR UVW LEARNING FOR THE TEN LTL PROPERTIES CONSIDERED IN SECTION IV

	Property	Time in s	
		chain len. 2	chain len. 3
1)	$\mathsf{G}[a \to (b \vee c \vee d)]$	0.763	timeout
2)	$\mathsf{G}[a \to (b \vee c)]$	0.517	1.029
3)	$\mathsf{G}[\mathsf{X}\,\neg a \to (\neg b \leftrightarrow \mathsf{X}(\neg b))]$	0.493	1.184
4)	$\mathsf{G}[a \to \neg b]$	0.408	0.713
5)	$\mathsf{G}[a \to (\neg b \wedge \neg c)]$	0.533	1.059
6)	$\mathsf{G}\,a$	0.526	1.057
7)	$\mathsf{G}[a \to \mathsf{F}\,b]$	0.442	0.870
8)	$\mathsf{G}[(a \wedge b) \to \mathsf{X}\,\mathsf{F}(\neg c)]$	0.634	119.123
9)	$\mathsf{G}\,\mathsf{F}\,a$	0.423	0.685
10)	$\mathsf{G}\,\mathsf{F}(\neg a \wedge \neg b)$	0.428	0.702

ensure that there are enough distinct positive examples for these properties.

For each property, we computed 50,000 different ultimately periodic words uv^ω that satisfy the property, where $|u|$ is of length 0, 1, 2, 3, or 4, while $|v|$ is of length 1, 2, 3, or 4. The characters of the words are the subsets of propositions holding, and all word part lengths are equally likely to be chosen. We also use a uniform probability distribution over the characters when computing the positive examples. Whenever a non-positive example for the property is found during the positive example computation, it is discarded and another example word is computed instead. We ran every experiment on 10 different example sets generated in this way and report the mean values obtained in the following.

The experiments were conducted on a computer with four AMD EPYC 7251 processors running at 2,1 GHz and an x64 version of Linux. The available main memory per run of the learner was restricted to 3 GB. We used a computation time limit of 600 s per learning problem.

Table I contains the mean computation times for all properties when using all 50,000 positive examples as input in each case. It can be seen that for most combinations, our approach computes a UVW rather quickly. Only for one property with a higher number of atomic propositions and an unnecessarily long chosen chain length, the toolchain times out.

Figure 3 shows for nine of the ten properties how big the computed UVW are, where sizes for both chain lengths of 2 and 3 are reported. Here, we varied the number of positive examples provided to the learner along the X axis (minimum: 100, in steps of 100). For very low number of examples, our toolchain often times out. This is rooted in the fact that the tightest UVW is often very large when not enough positive examples are available. It can also be observed that for a lower chain length, the computed UVW converges to a small one much earlier.

Figure 4 depicts the relationship between computation time and the sizes of the computed UVWs in more detail, using Property 3, the one that was left out of Figure 3. It can be observed that computation times are very short when enough positive examples are available, and they grow only very mildly with additional positive examples. When, however, not enough examples are available, the approach computes a much larger number of simple chains, which also increases the workload of the UVW minimization heuristic.

Finally, Figure 5 depicts the UVWs learned for Property 3. The property can only be learned correctly with a chain length of 3, and the two paths through the UVW on the right-hand side show the two conjunctive requirements that the LTL property $\mathsf{G}[\mathsf{X}\,\neg a \to (\neg b \leftrightarrow \mathsf{X}(\neg b))]$ imposes, namely that (1) after a character with $b = \mathit{true}$ is seen, b needs to retain a value of true until a gets a false value afterwards and (2) after a character with $b = \mathit{false}$ is seen, b needs to retain a value of false until a gets a false value afterwards. The automaton has a simple structure and is quite easy to read. The computed UVW for a chain length of 2 is, as expected, an overapproximation of the language to be learned. Interestingly, the encoded language is a liveness language, even though the approximated LTL property is not.

For all ten LTL properties considered in our experiments, the learned UVWs for the correct chain lengths represent the correct languages and have a minimal number of states. Moreover, the resulting UVWs are fairly easy to understand (we refer the reader to the extended version of the experiments, available in the code repository [32] for their depiction), which underpins the use of UVWs as an easy-to-understand specification formalism.

V. Discussion of the Proposed Approach

In this section, we discuss potential challenges for the application of our approach as well as strategies to mitigate them.

First, our learning algorithm depends on a well-chosen tightness value n: if the value is too small, then the resulting UVW is too permissive and imprecise; if the value is chosen too large, on the other hand, the computational effort for learning an UVW can become prohibitive. Determining an appropriate value for n remains an open question. A potential strategy to find such a value in practice—apart from relying on domain knowledge—could be to perform a search that starts with a reasonably small tightness value and then increments the value until the resulting UVW does no longer change. As Table I suggest, our learning algorithm is fast enough (less than 1 s given sufficiently many examples) so that such a search is a viable approach. On a more general note, however, we would like to reiterate that without such a parameter, the problem of learning from only positive examples is ill-defined.

Second, as our experimental evaluation has shown, our learning algorithm requires a fair amount of positive examples (several thousands) to perform well. However, compared to most other learning algorithms, which require negative examples, we believe that this is not a major restriction in practice because (a) positive examples are usually much easier to obtain than negative examples and (b) positive examples are often readily available (e.g., from log files) or can be generated automatically (e.g., by means of simulations).

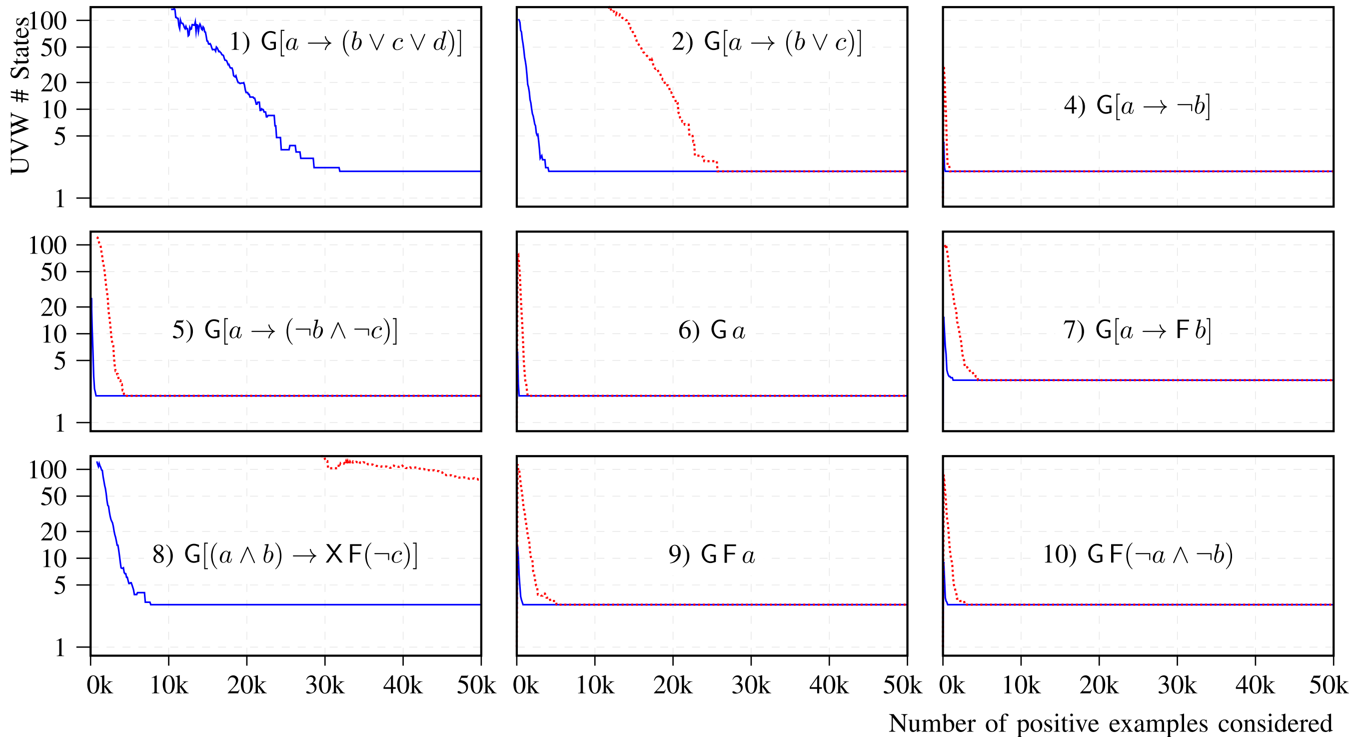

Fig. 3. UVW sizes for nine of the ten examples. The dotted lines are for a UVW chain length of 3, while the solid lines are for a UVW chain length of 2. The number of positive examples given to the learner ranges between 100 and 50,000 and is displayed on the x axis of each chart. Parts in the charts with absent lines represent timeouts, which were common for low numbers of positive examples.

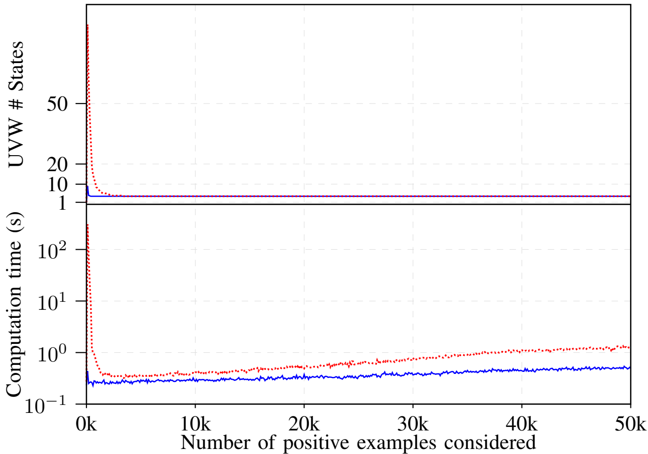

Fig. 4. Joint plot for the computation time and UVW sizes for Property 3).

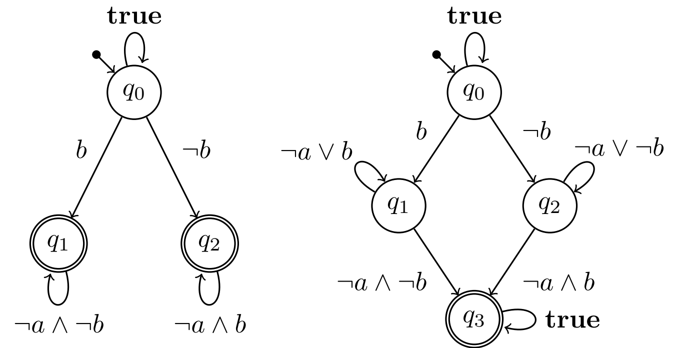

Fig. 5. Learned UVW (for chain lengths of 2 on the left and 3 on the right) from the positive examples for the LTL property $\mathsf{G}[\mathsf{X}\,\neg a \to (\neg b \leftrightarrow \mathsf{X}(\neg b))]$.

Finally, our learning algorithm is designed to learn properties of infinite words and, hence, requires infinite words (in the form of ultimately periodic words uv^ω) as input. In practice however, one will only be able to observe or simulate finite executions. To mitigate this challenge, we propose two strategies. If the examples are obtained from simulations, it is fairly easy to detect repetitions of system states that can be used to partition the execution into an initial fragment u and a repeating part v. On the other hand, if the examples are obtained from observing an existing system without access to its internal state, one can use acceleration-like techniques [34] to detect cycles based on repeating patterns in the observations.

VI. Conclusion and Direction for Future Work

We have developed an effective method for learning formal specifications in the form of universal very-weak automata from positive examples only. Our learning algorithm reduces the problem of learning such an automaton to the enumeration of elements in a Pareto front and uses an effective minimization technique to obtain a unique finite-state representation of the learned property. Experiments with properties from

the Advanced Microcontroller Bus Architecture (AMBA) have demonstrated that our approach is able to infer concise and easy-to-interpret specifications from positive examples.

For future work, we plan to adapt our learning algorithm to be able to learn from finite rather than infinite words. A relatively straightforward way to do this would be to restrict the chain enumeration to only consider chains that have $\circledcirc\!\!\circlearrowleft \mathbf{true}$ as final state. The class of languages learnable by this approach would then exactly be the set of languages that can be accepted by so-called universal very-weak finite automata, studied, for instance, by Bojańczyk [30].

Finally, we are interested in determining an appropriate tightness value automatically from the sample, which seems to be a non-trivial problem.

References

[1] M. Maidl, "The common fragment of CTL and LTL," in *FOCS 2000, Proceedings*, 2000, pp. 643–652. [Online]. Available: https://doi.org/10.1109/SFCS.2000.892332

[2] K. Adabala and R. Ehlers, "A fragment of linear temporal logic for universal very weak automata," in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, 2018, pp. 335–351.

[3] R. Ehlers, "ACTL ∩ LTL synthesis," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, pp. 39–54.

[4] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of AMBA AHB from formal specification: a case study," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 585–601, Oct 2013.

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

[6] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[7] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Inf. Comput.*, vol. 103, no. 2, pp. 299–347, 1993.

[8] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," in *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific, 1992, pp. 49–61.

[9] M. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *ICGI*, ser. Lecture Notes in Computer Science, vol. 6339. Springer, 2010, pp. 66–79.

[10] D. Neider and N. Jansen, "Regular model checking using solver technologies and automata learning," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, vol. 7871. Springer, 2013, pp. 16–31.

[11] D. Neider and I. Gavran, "Learning linear temporal properties," in *FMCAD*. IEEE, 2018, pp. 1–10.

[12] A. Camacho and S. A. McIlraith, "Learning interpretable models expressed in linear temporal logic," in *ICAPS*. AAAI Press, 2019, pp. 621–630.

[13] J. Kim, C. Muise, A. Shah, S. Agarwal, and J. Shah, "Bayesian inference of linear temporal logic specifications for contrastive explanations," in *IJCAI*. ijcai.org, 2019, pp. 5591–5598.

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

[15] S. Mohammadinejad, J. V. Deshmukh, A. G. Puranic, M. Vazquez-Chanlatte, and A. Donzé, "Interpretable classification of time-series data using efficient enumerative techniques," in *HSCC*. ACM, 2020, pp. 9:1–9:10.

[16] W. M. P. van der Aalst, J. Carmona, T. Chatain, and B. F. van Dongen, "A tour in process mining: From practice to algorithmic challenges," *Trans. Petri Nets Other Model. Concurr.*, vol. 14, pp. 1–35, 2019.

[17] F. Avellaneda and A. Petrenko, "Inferring DFA without negative examples," in *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, 2018, pp. 17–29. [Online]. Available: http://proceedings.mlr.press/v93/avellaneda19a.html

[18] R. Ehlers, "Minimising deterministic Büchi automata precisely using SAT solving," in *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, ser. Lecture Notes in Computer Science, O. Strichman and S. Szeider, Eds., vol. 6175. Springer, 2010, pp. 326–332.

[19] S. Baarir and A. Duret-Lutz, "SAT-based minimization of deterministic ω-automata," in *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, 2015, pp. 79–87.

[20] D. Lo and S. Maoz, "Specification mining of symbolic scenario-based models," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'08, Atlanta, Georgia, November 9-10, 2008*, 2008, pp. 29–35.

[21] ——, "Mining scenario-based triggers and effects," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, 2008, pp. 109–118.

[22] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 45–80, 2001.

[23] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal logic for scenario-based specifications," in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 445–460. [Online]. Available: https://doi.org/10.1007/b107194

[24] W. Damm, T. Toben, and B. Westphal, "On the expressive power of live sequence charts," in *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, 2006, pp. 225–246.

[25] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011. [Online]. Available: http://hanj.cs.illinois.edu/bk3/

[26] A. Y. Ng and S. J. Russell, "Algorithms for inverse reinforcement learning," in *ICML*. Morgan Kaufmann, 2000, pp. 663–670.

[27] M. Vazquez-Chanlatte, S. Jha, A. Tiwari, M. K. Ho, and S. A. Seshia, "Learning task specifications from demonstrations," in *Advances in Neural Information Processing Systems 31, NeurIPS*, 2018, pp. 5372–5382. [Online]. Available: http://papers.nips.cc/paper/7782-learning-task-specifications-from-demonstrations

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

[29] O. Kupferman and M. Y. Vardi, "Safraless decision procedures," in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*. IEEE Computer Society, 2005, pp. 531–542.

[30] M. Bojańczyk, "The common fragment of ACTL and LTL," in *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS*, ser. Lecture Notes in Computer Science, R. M. Amadio, Ed., vol. 4962. Springer, 2008, pp. 172–185.

[31] R. Ehlers, "Computing the complete pareto front," *CoRR*, vol. abs/1512.05207, 2015. [Online]. Available: http://arxiv.org/abs/1512.05207

[32] I. Gavran, R. Ehlers, and D. Neider, "unite - Uvw learNer from posITive Examples," Aug. 2020. [Online]. Available: https://github.com/TUC-ES/unite

[33] "ARM Ltd. Amba specification (rev. 5)," 2019. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033/index.html

[34] B. Jonsson and M. Saksena, "Systematic acceleration in regular model checking," in *CAV*, ser. Lecture Notes in Computer Science, vol. 4590. Springer, 2007, pp. 131–144.