

Quantitative Learning of LTL from Finite Traces

Mohammad Afzal^{1,2}, Sankalp Gambhir¹, Ashutosh Gupta¹, and
Shankaranarayanan Krishna¹

¹ Indian Institute of Technology, Bombay, India

² TCS Research, Pune, India

Abstract In this paper, we present a novel method for learning LTL properties from a set of traces. The main novelty of our method, as compared to many existing ones, is that we learn formulae in a “quantitative” sense : given a sample $\mathcal{S} = (P, N)$ consisting of positive traces P and negative traces N , we find the formula φ which “best” describes the sample such that all positive traces satisfy φ and all negative traces do not satisfy φ . To decide how good a formula is with respect to the sample, we have developed a scheme of assigning a value for a formula for a given trace under various schemes. We use the schemes to encode the optimal property synthesis problem, namely, finding the best property/LTL formula into an optimization problem. Then we use an SMT solver to find the best fit formula for a given set of traces. Finally, we present a hybrid approach combining classical LTL satisfaction and the ranking scheme that works on a fragment of LTL and greatly improves performance while maintaining reasonable expressiveness. We have developed a tool QUANTLEARN based on our method and applied on some benchmarks. We also compared different valuation schemes. Our experiments suggest that QUANTLEARN is successful in mining formulae which are reasonably good representations of the sample with high resilience to noise in the data.

1 Introduction

Learning the properties of a system by observing its traces is one of the fundamental aspects of science. There has been a whole range of techniques developed [1] to learn the properties of several natural systems. Learning properties that capture the partial order of events is an active area of research [2]. A natural extension of this approach is to learn properties which capture orderings between events that repeat or are periodic. Such properties are succinctly representable using linear temporal logic (LTL). We focus on mining *linear temporal* specifications, i.e., specifications given by formulae in LTL.

There have been many works that learn LTL formulae from traces. One such is the paper [3], which requires a template LTL formula which defines the structure of the formula keeping events undefined, along with a log of traces, and outputs a set of LTL formulae, which are instantiations of the template. [3] implements this approach in the tool Texada. In another direction, [4] works with traces that are in the form of lassos, where the lassos are interpreted as

rational infinite words. The problem of matching formulae with traces is encoded as a constraint system and a satisfying assignment gives us a learned property. However, requiring the inputs to be lassos significantly restricts the method’s application to real scenarios, as it may be hard to obtain infinite behaviour of the system in practice. We seek to combine some of these ideas and try to eliminate their restrictions with a method that may be used on finite traces obtained from real systems, and output results relevant to the user.

An LTL property learning method has two key aspects. The first is to specify the search space of the formulae. The second aspect is to match the formulae with traces and identify the one which specifies not only a correct property but also is most *relevant* to the user. We need the relevancy criterion because we typically find that traces satisfy many formulae in our search space. Hence, we need to have some ways to *rank* the formulae. In this paper, we develop a novel and flexible scheme of ranking the formulae to match the traces by inferring which infinite strings they are likely to be prefixes of. We use the scheme to develop property learning algorithms that uses state-of-the-art solver technology to find the best-ranked formula in a given search space of the formulas.

Contributions. Our main contribution in this paper are learning algorithms, which take as input a sample $\mathcal{S} = (P, N)$ consisting of positive traces P and negative traces N (both are sets of finite words), and produce the best LTL formula consistent with the sample. Our learning algorithms focus on the GF [5,6,7] fragment of LTL. This fragment expresses most interesting steady state properties of systems, and avoids the undesirable ‘stuttering’ patterns that the next operator (X) produces, as well as the exponential complexity increase from until (U) due to recombination of formulae.

We propose 4 variants of the learning algorithm : (i) *Constraint system optimization* which takes as input a sample and a depth d (of some unknown formula) and produces the best LTL GF formula of depth d consistent with the sample, this is in the spirit of [4], but works over finite words, and uses an optimizer to rank the formulae and obtain the one with the highest score, with the real-valued scoring function made parametric to the method, (ii) *Optimized pattern matching*, which takes as input, a sample and a template formula with unknown variables, produces the best LTL GF formula consistent with the sample by instantiating the variables in the template, is similar to ideas presented in [3], combined with the quantitative ranking, (iii) *Hybrid pattern matching*, which takes as input a sample and a “hybrid” formula specifying templates for some subformulae leaving other subformulae free to be fitted, and synthesizes the best LTL GF formula consistent with the sample. These three learning algorithms use a constraint solving optimization approach to obtain a ranking scheme resulting in the “best” formula - the notion of best comes by ranking formulae which are consistent with the sample, associating numerical values with them. The higher the positive score of a formula, the better it is, in describing the sample. The score captures intuition that all part of the formulas must contribute for the satisfaction by the sample. These use a real-valued valuation function that attaches to all formulae (if we know the depth is d , the search space has all GF formulae

of depth $\leq d$) a score, and then the optimizer prunes away some formulae depending on the scores. However, this search space explodes in size as we increase search depth.

To quickly prune this search space, we propose another learning algorithm, (iv) *Compositional Ranking* that takes as input only a search depth along with a sample, and greedily prunes the search space by removing those formulae f for which $\text{F}f$ is not consistent with the sample. This is done iteratively, starting with formulae of depth 0; the consistent ones are then Boolean composed and the process is repeated until we reach the chosen depth. This greatly improves performance, but effectively restricts the possible outputs to ‘steady state’ properties.

To allow the algorithms to quantitatively distinguish formulae, we supplement them with the idea of a ranking scheme as a parameter to the methods. Our ranking scheme quantitatively scores each formula against a finite word. It expands on intuitive ideas used to formulate distances in regular language spaces [8,9]. The suggested scheme assigns a formula a high score if it expresses most features of the word. A formula can score well on a word if it provides longer evidence of validity with respect to larger parts of the word. For example, Ga will score more on the word $aaaa$ than on the word aa , since it contains more evidence for the word to have been a prefix of $a^\omega \in L(Ga)$. Furthermore, our suggested scoring scheme encourages simpler formulae over complex ones in equivalence classes; that is, from an equivalence class of LTL formulae, the ones with smaller parse trees will be preferred. For example, the same set of words satisfy Ga and GGa . However, our intuition suggests that we must prefer Ga . Therefore, we penalize if a formula is unnecessarily complex. We need to ensure Boolean operations are meaningfully used. For example, $G(\phi \vee \psi)$ should rank well on a sample only if both ϕ and ψ are evenly true along the sample. We translate the above intuitions into a formal mathematical scoring scheme. Our experiments suggest that our choice of the scheme matches the intuition.

Our property learning algorithm defines a tree as a template for LTL formulae. We build quantifier-free constraints along with a symbolic expression that encodes the value of the unknown LTL formula with respect to the traces. We use an SMT solving optimizer to find the formula that scores best among the traces. We consider several natural ways to assign a value to a (word,formula) pair. For instance, we consider modeling conjunctions by the minimum function or by multiplication. Such valuation schemes are parametric to our method. We also consider several variations of the problem. In one such, we let the user fix some parts of the formula templates in our search. This allows us to prioritize a class of formulae without expensive optimizations. We also allow users to prioritize the events that are considered to be more important than others.

We have implemented the algorithm in a tool QUANTLEARN. We demonstrate the effectiveness of our method on a set of synthetic traces with random noise generated from a set of common LTL formulae, and by verifying system guarantees for a Dining Philosophers simulation. We consider all synthetic traces from [4] and also add some of our own, and take the traces for dining philosoph-

ers from Texada [3,10]. We show that QUANTLEARN is more expressive and effective in learning LTL properties, especially when working with limited, noisy data.

Related Work. The papers closest to our work are [3] and [4]. The focus of [4] is to produce the minimal formula which is consistent with a rational sample irrespective of the expressiveness, while [3] requires a user defined input template of the LTL formula which they would like to satisfy. Both work with infinite traces : [4] takes rational traces in the form uv^ω where u, v are typically words of length ~ 10 , while [3] mines specifications from finite traces, and appends them with an infinite sequence of terminal events. [11] considers finite traces, and develops a LTL checker which takes an event log and a LTL property and verifies if the observed behaviour matches some bad behaviour. The papers [12,13] look at the application of monitoring the execution of Java programs, and check LTL formulae on finite traces of these programs. In [14], the authors focus on mining quantified temporal rules which help in establishing data flow analysis between variables in a program, while in [15] software bugs are exposed using a mining algorithm, especially for control flow paths. [16] looks at process mining in the context of workflow management. The tool PISA [17] is developed to extract the performance matrix from workflow logs, while a declarative language is developed to formulate workflow-log properties in [18]. Tool Synoptic [19] on the other hand, follows a different approach and takes event logs and regular expressions as input and produces a model that satisfies a temporal invariant which has been mined from the trace.

2 A Motivating Example

In Figure 1(a), we display a simplified trace of events from an FTP server. Our method ranks LTL formulae over the given trace and finds the formula that is ranked best. For example, we have listed formulae in the table in Figure 1(b). We have also listed the score of each of the formulae in the second column.

Let us consider the formula $\text{GF}\text{disconnected}$ to understand the value assignment. We give value to each subformula at each position of the trace. In Figure 1(c), we illustrate the computation of the values. In the first column, we show the position on the trace, and in the second column, we show the propositional variables at the position. Formula `disconnected` has a score of 1 wherever the variable `disconnected` occurs, and 0 otherwise. `Fdisconnected` gets the value at a position depending on the distance from the position we find `disconnected`. If the distance is zero, we assign 0.90, a seemingly arbitrary choice which we will discuss shortly. If `disconnected` is further away from the position, we assign `Fdisconnected` higher value. However, the increase in value keeps decreasing exponentially as the distance increases. Once we have values for `Fdisconnected` at each position, we can compute the value of $\text{GF}\text{disconnected}$ at the first position. We expect `Fdisconnected` to be non-zero at each position. Otherwise, the value of $\text{GF}\text{disconnected}$ will be zero. The value of $\text{GF}\text{disconnected}$ at position 1 is the decaying sum of the values of `Fdisconnected` at each position.

In the above, the rate of decay used is e^{-s} at the s^{th} position. The choice of the initial values and decays are learned from experiments. We tested various

connected

connected,password

connected,login_incorrect,auth_fail

disconnected

connected

connected,password

connected,login_incorrect,auth_fail

disconnected

connected

...

connected,login_incorrect,auth_fail

disconnected

connected,password

connected,logged_in

connected,logged_in

disconnected

(a)

	Formula	Value
	G disconnected	0
	G auth_fail \Leftrightarrow (connected \wedge login_incorrect)	0.146
	F disconnected	0.044
	G F disconnected	0.173

(b)

Position		disconnected	F disconnected	G F disconnected
1	connected	0	0.044	0.173
2	connected,password	0	0.12	-
3	connected,login_incorrect,auth_fail	0	0.33	-
4	disconnected	1	0.90	-
5	connected	0	0.044	-
6	connected,password	0	0.12	-
7	connected,login_incorrect,auth_fail	0	0.33	-
8	disconnected	1	0.90	-
	-

(c)

Figure 1: (a) A simplified trace of a FTP server. (b) Some candidate LTL formulae and their values over the trace. (c) An illustration in computing values for $G F \text{disconnected}$. - in a cell represents value is not relevant for our computation.

choices and found the values that seem to work in our experiments. We also punish formulas to be unnecessarily complex by using a punishing factor. The value of the punishing factor (δ) for this example is 0.9.

We want to search for the formula that has the best value over the trace. We pass a template formula to our method. For example, consider $G \varphi(1)$, where $\varphi(1)$ is an unknown formula with depth 1. We encode all possibilities of internal nodes and leaves of φ as a constraint system. We use an SMT solver to optimize the value for $G \varphi(1)$. We obtain the formula $G F \text{disconnected}$. The technique developed in [3] needs templates that can only have formulae φ with depth 0. We can work with several template choices and obtain a range of properties over traces. Let us consider a more specific template, $G \text{auth_failed} \Leftrightarrow \varphi(1)$, with an unknown φ . Our search obtains $G \text{auth_failed} \Leftrightarrow (\text{connected} \wedge \text{login_incorrect})$, stating that every failed authorization is a result of an invalid login and that every invalid login does indeed result in authorization failure. We learn specification of arbitrary complexity, *choosing* the best fit with little to no input patterns.

3 Preliminaries

Propositional Logic. Let Var be a set of *propositional* variables, which take values from $\mathbb{B} = \{0, 1\}$ (0 interpreted as *false* and 1 as *true*). The set of formulae

\mathcal{W} in propositional logic — with formulae denoted herein as capital Greek letters — is defined inductively as follows:

- (1) $\forall p \in \text{Var}, p \in \mathcal{W}$; (2) $\forall \Phi \in \mathcal{W}, \neg\Phi \in \mathcal{W}$; (3) $\forall \Phi, \Psi \in \mathcal{W}, \Phi \wedge \Psi \in \mathcal{W}$.

We use the usual syntactic sugar $\Phi \vee \Psi$, $\Phi \Rightarrow \Psi$, and $\Phi \Leftrightarrow \Psi$. A *propositional valuation* is defined as a mapping $v : \text{Var} \rightarrow \mathbb{B}$, which maps propositional variables to boolean values. The semantics of this logic, given by the satisfaction relation \models , are defined inductively as: (1) $v \models x$ iff $v(x) = 1$; (2) $v \models \neg\Phi$ iff $v \not\models \Phi$; (3) $v \models \Phi \wedge \Psi$ iff $v \models \Phi$ and $v \models \Psi$. If $v \models \Phi$ we say v models Φ . A formula is said to be *satisfiable* if there exists a model for it.

Finite Words. An *alphabet* Σ is a non-empty, finite set of *symbols*. A *finite word* w over Σ is a finite sequence $a_1 a_2 \dots a_n$ of symbols from Σ . The empty sequence is called the empty word, denoted ϵ . The domain of w , denoted $\text{dom}(w)$ is the set of positions in w . Thus, $\text{dom}(a_1 \dots a_n) = \{1, 2, \dots, n\}$ and $\text{dom}(\epsilon) = \emptyset$. The length of a finite word w is denoted $|w|$, with $|\epsilon| = 0$. The set of all finite words over Σ is denoted Σ^* . Given a finite word α , we refer to the symbol at position i of α by $\alpha(i)$. The subsequence $a_i \dots a_j$ of α is denoted as $\alpha[i:j]$, while $\alpha[i:]$ denotes the suffix $a_i \dots$ of α .

Linear Temporal Logic. Let \mathcal{P} be a set of propositional variables. LTL [20] is an extension of propositional logic with temporal modalities, which allows the expression of temporal properties. Formulae in LTL are defined inductively starting from propositional variables. An LTL formula φ over \mathcal{P} is defined according to the following grammar.

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid X \varphi \mid \varphi U \varphi \quad \text{where } a \in \mathcal{P}$$

Using the above, we have the derived formulae $F\varphi = \text{true} U \varphi$ and $G\varphi = \neg F \neg \varphi$. The size of an LTL formula φ denoted by $|\varphi|$ is the number of subformulae in it. For example, if $\varphi = p U \psi$, then $|\varphi| = |\psi| + 2$. We define the satisfaction relation of an LTL formula as follows.

- $\alpha \models a$ iff $\alpha[0:0] = a$
- $\alpha \models \neg\Phi$ iff $\alpha \not\models \Phi$
- $\alpha \models \Phi \wedge \Psi$ iff $\alpha \models \Phi$ and $\alpha \models \Psi$
- $\alpha \models X\Phi$ iff $\alpha[1:] \models \Phi$
- $\alpha \models \Phi U \Psi$ iff $\exists i$ s.t. $\alpha[i:] \models \Psi$, and for each $j < i$, $\alpha[j:] \models \Phi$

The language $L(\varphi)$ of an LTL formula φ is defined as $\{\alpha \in \Sigma^* | \alpha \models \varphi\}$. Two LTL formulae having the same language are called *equivalent*. In this paper, we learn formulae that are in the GF-fragment of LTL, where only the G and F modalities are allowed apart from Boolean connectives.

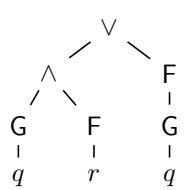


Figure 2: Syntax tree of $[Gq \wedge Fr] \vee (FGq)$.

Since the LTL formulae can be converted into negation normal form (NNF), we learn formulae only in NNF. Given an LTL formula φ , the syntax tree of φ is a tree labeled with variables, Boolean connectives and temporal modalities. The variables always appear at the leaf nodes, while temporal modalities and Boolean connectives are internal nodes. For example, we present the syntax tree of the formula $\varphi = [Gq \wedge Fr] \vee (FGq)$ in Figure 2.

Samples. A *sample* is a pair $\mathcal{S} = (P, N)$ of two finite, disjoint sets $P, N \subseteq (2^{\mathcal{P}})^*$. The words in P are *positive traces* while words in N are *negative traces*. We learn an LTL formula from a given sample.

SMT Solvers. The satisfiability (SAT) problem for propositional logic is to decide whether a given formula is satisfiable. The satisfiability modulo theories (SMT) problem generalizes the SAT problem to include several useful first-order theories. An SMT solver is a tool for deciding satisfiability of given formulae in the theories. Similar to modern SAT solvers, the SMT solvers implement several optimization techniques to solve problems with millions of variables, whilst also incorporating features that vastly improve their expressivity, such as model checking over infinite domains [21].

4 Valuation Functions

In this section we discuss valuation functions that assign a value to a pair (φ, w) consisting of an LTL formula φ and a finite word w . The function indicates “how well” w is represented by φ . The intuition is that a pair scores high if all subformulae of φ participate in accepting w in $L(\varphi)$. For example, $G(p \vee q)$ should score well along with word $\{p\}\{q\}\{p\}\{q\}\{p\}\{q\}$ but should not do well with word $\{p\}\{p\}\{p\}\{p\}\{p\}\{p\}$, since subformula q did not appear in the word.

4.1 A valuation function

Let us present a valuation function first. Let \mathcal{F} represent the set of NNF GF-fragment formulae over \mathcal{P} . We interpret LTL formulae over finite words and define the quantitative semantics in terms of a *valuation mapping* $V : \mathcal{F} \times \Sigma^* \rightarrow \mathbb{R}^+ \cup \{0\}$, where $\Sigma = 2^\mathcal{P}$. The valuation mapping is defined over a word $w \in \Sigma^*$ inductively as follows:

$$\begin{aligned} - V(p, w) &= \begin{cases} 1 & \text{if } p \in w(1) \\ 0 & \text{otherwise} \end{cases} & - V(\varphi \wedge \psi, w) &= \delta \cdot V(\varphi, w) \cdot V(\psi, w) \\ - V(\neg p, w) &= \begin{cases} 1 & \text{if } p \notin w(1) \\ 0 & \text{otherwise} \end{cases} & - V(\varphi \vee \psi, w) &= \delta \cdot \frac{V(\varphi, w) + V(\psi, w)}{2} \\ - V(G\varphi, w) &= \begin{cases} \delta \cdot \sum_{i=0}^{|w|} r^i \cdot V(\varphi, w[i:]) & \text{iff } \exists t, V(\neg\varphi, w[t:]) > 0 \\ 0 & \text{otherwise} \end{cases} & & \\ - V(F\varphi, w) &= \begin{cases} \delta \cdot r^t \cdot V(\varphi, w[t:]) & t = \min\{j \mid V(\varphi, w[j:]) > 0\} \\ 0 & \text{if } \nexists t, V(\varphi, w[t:]) > 0 \end{cases} & & \end{aligned}$$

If $w \models \varphi$ then $V(\varphi, w)$ is non-zero. The valuation scheme is parameterized by two discount factors r and δ . For the literals, we assign valuation zero or one if the word satisfies the literals or not. We interpret conjunction as multiplication, which implies we need both subformulae to do well on the word. We interpret disjunction as an addition, which implies we give a high score to the formula if any of the two subformulae does well on the word. Our interpretation of $G\varphi$ computes the discounted sum of the value of φ at each position of the word. After each letter, we apply a discount of r , where $0 < r < 1$. Our interpretation of $F\varphi$ computes the discounted score of φ at the earliest position where φ has a non-zero score. We also apply a discount factor δ each time we build a more complex formula, where $0 < \delta < 1$. For example, consider the formula $\varphi = Fq$ for $p, q \in \mathcal{P}$ and the word $w = \{p\}\{p, r\}\{p\}\{p, s\}\{p\}\{p, q\}(\{r\}\{q\})^*$. Then q

holds for the first time at the position $t = 7$, and for all $t' < t$, q is not present in w . Thus, $V(q, w[7 :]) = 1$, making $V(\varphi, w) = \delta \cdot r^7 \cdot 1$. In other words, we assign non-zero scores only for satisfiable formulas.

For a sample \mathcal{S} , the valuation of a formula φ is taken as the sum of valuations over all positive traces in the sample, that is, $V(\varphi, \mathcal{S} = (P, N)) = \sum_{w \in P} V(\varphi, w)$.

The scheme attempts to match intuition about the operators. Our experiments illustrate that our choice is promising. In the next section, we will consider the other possible natural choices for the valuation function.

4.2 Variations in valuation functions

There are multiple (subjective) choices for the valuation function, none of which have obvious theoretical advantages. However, they behave very differently in practice. We consider various choices for the valuation function. Here, we present examples that differentiate the capabilities and uses of each valuation function.

- We may assign $V(\varphi \wedge \psi, w)$ as the minimum of $V(\varphi, w)$ and $V(\psi, w)$ [22]. This valuation ensures that both φ and ψ must score high for $\varphi \wedge \psi$ to score high. However, the function is not sensitive to the formula that has a higher value. Therefore, the learning algorithm becomes unguided for one part of the formula. This suggests a modification to the valuation function that takes both the subformulae into account symmetrically, without flattening one of the subformulae. Our valuation function for the conjunction of two formulae, defined as their product $V(\varphi, w) \times V(\psi, w)$ is based on this idea.
- Symmetrically, we may assign $V(\varphi \vee \psi, w)$ as the maximum of $V(\varphi, w)$ and $V(\psi, w)$. Just as in the previous case, this unnecessarily discards potential information carried by the valuation of a smaller subformula. In our earlier evaluation, we used the arithmetic mean of the two subformulae, which helps model disjunction such that if either of the parts evaluates high, then the score goes high and both parts have an incentive to score well. For example, consider $G(p \vee q)$. If no q occurs in w then $V(p \vee q, w[i :])$ will be scored lower than if both p and q occur in w .
- In our valuation of $V(G\psi, w)$, we compute the discounted sum of $V(\psi, w[i :])$. We may choose a variety of discounted sums that have a different decay rate than the exponential decay, which allows us to give varied weight to longer words.
- $V(F\psi, w)$ needs to represent the strength of evidence of ψ being true at $w[i :]$ for some i . We can give some discounted weights to the true occurrences and compute the average. In addition to being computationally expensive, this scheme may have an adverse effect that, in case of ψ being true at each $w[i :]$, then $V(F\psi, w)$ may rank higher than $V(G\psi, w)$. Therefore, in our earlier scheme, we gave weight only to the earliest occurrence.

5 Learning Algorithms

As our main contribution, we propose learning algorithms to solve the following problem. *Given a sample $\mathcal{S} = (P, N)$ over finite words, compute an LTL formula φ in the GF-fragment that best describes \mathcal{S} and is consistent with \mathcal{S} . That is, φ*

Algorithm 1 Computing the optimal formula given a sample

```
1: procedure CONSTRAINTOPT( $\mathcal{S} = (P, N), d$ )            $\triangleright$  Returns optimal formula
2:   construct  $\Phi_d^{\mathcal{S}}$                                  $\triangleright$  Constraints in eq(1)
3:   maximize  $\min(\{y_{1,0}^{\tau} \mid \tau \in P\})$  with  $\Phi_d^{\mathcal{S}}$  as constraint
4:   if optimization succeeds with model m then           $\triangleright$  SAT
5:     construct formula tree from m
6:     return optimized formula tree
7:   else
8:     return UNSAT
```

has the highest score, based on the valuation described above, among all formulae such that for all $w \in P$, $w \models \varphi$ and for all $w' \in N$, $w' \not\models \varphi$. Given a sample, there may be several LTL formulae of a given size which are consistent with it. We develop a method to rank all such formulae and compute the one that best represents the temporal patterns displayed by the words in the sample. In other words, we work to produce an LTL formula which quantitatively describes the sample, as opposed to existing qualitative approaches. In order to propose a ranking scheme for LTL formulae, given a word $w \in \Sigma^*$, we associate to each formula φ a score, based on how well the word w satisfies φ . This notion leads to a quantitative version of satisfiability, and the higher the score a formula has, with respect to a word, the higher the satisfiability.

To achieve the goal of ranking formulae, we propose the techniques of *Constraint System Optimization* (Section 5.1) *Optimized Pattern Matching* (Section 5.2) and *Hybrid Pattern Matching* (Section 5.3). In the first one, we get a sample and a depth n as input. We encode the syntax tree of this unknown formula of depth n along with constraints to compute the score of each node in the tree. The second one makes use of a formula template pattern provided by the user, but has unknown propositional variables. We encode constraints which allow mapping these variables to unique variables occurring in the sample. The third one is a “hybrid” approach, a middle ground incorporating both of the above techniques. In the above methods, we use an optimizing SMT solver to solve the above constraints to find the *best* formula for the sample that has highest score according to the valuation function described above.

In the above approaches, we consider the entire search space of formulae (up to given depth, or following a template), and score all of them using the valuation function; the optimizer then prunes away those which do not have the highest score. To get around situations where the optimization problem is computationally intensive and also to reduce search space, we consider an alternative approach. In this approach (Section 5.4), we reduce the search space of all possible formulae (up to the given depth) that must be enumerated and whose scores must be computed. Since we focus on the GF-fragment of LTL, we prune the search space of formulae by eliminating those formulae f such that $\mathbf{F}f$ is not consistent with the sample. We also consider the Boolean combinations of the rest till we reach the given depth. This aggressive pruning is not complete and may fail to compute the optimal formulae.

5.1 Constraint System Optimization

In Algorithm 1, we present the method to compute the optimal formula with the highest score for a sample $\mathcal{S} = (P, N)$ along with the desired depth d of the formula. This is obtained by computing and optimizing the scores of a class of formulae of depth d , constrained to be well formed and to be satisfied by \mathcal{S} . We reduce the construction of an LTL formula for a sample \mathcal{S} to a constraint system $\Phi_d^{\mathcal{S}}$, which is constructed in three parts as:

$$\Phi_d^{\mathcal{S}} = \varphi_d^{\text{ST}} \wedge \bigwedge_{\tau \in P} \varphi_d^{\tau} \wedge \bigwedge_{\tau' \in N} \neg \varphi_d^{\tau'} \quad (1)$$

The first part encodes the structure of a *syntax tree* (ST) representing an unknown formula, while the second and third encode the functional constraints on the *score* of each node, which is enforced by the operators that make up the formula.

To encode the formula structure, we use a syntax tree with identifiers $\mathcal{N} = \{1, 2, \dots, n\}$, for the set of nodes, where $n = |\mathcal{N}|$. We assume herein that the root node is identified as 1. We have the child relations L and R , such that $(i, j) \in L$ (or R) iff the node j is the left (right) child of node i . The only child of unary operators is considered as a left child by assumption. For each node $i \in \mathcal{N}$ and possible label $\lambda \in \mathcal{O} \cup \mathcal{P}$, we introduce a Boolean variable $x_{i,\lambda}$ indicating whether the node is labelled with an operator (\mathcal{O}) or variable (\mathcal{P}). φ_n^{ST} is constructed as the conjunction of formulae 2 through 10.

$$\left[\bigwedge_{1 \leq i \leq n} \bigvee_{\lambda \in \mathcal{P} \cup \mathcal{O}} x_{i,\lambda} \right] \wedge \left[\bigwedge_{1 \leq i \leq n} \bigwedge_{\lambda \neq \lambda' \in \mathcal{P} \cup \mathcal{O}} (\neg x_{i,\lambda} \vee \neg x_{i,\lambda'}) \right] \quad (2)$$

$$\bigwedge_{\exists j(i,j) \in L} \bigvee_{p \in \mathcal{P}} x_{i,p} \quad (3)$$

The constraint given by expression (2) ensures the two properties that each node must accept atleast one label, and that it must accept atmost one label, while expression (3) ensures that the leaf nodes are labelled with propositional variables, and not operators (since they do not have children).

Next, we encode the functional constraints imposed by the operators. For this, to each node, we attach a set of variables over \mathbb{R} , representing its *score* at each point in a trace $\tau \in \mathcal{S} : Y_i^{\tau} = \{y_{i,t}^{\tau} \mid 0 \leq t \leq |\tau|, i \in \mathcal{N}\}$, where $y_{i,t}^{\tau}$ is defined below. For a given trace τ we construct φ_d^{τ} as the conjunction of:

$$y_{1,0}^{\tau} > 0 \quad (4)$$

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{p \in \mathcal{P}} x_{i,p} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^{\tau} = \begin{cases} 1 & \text{if } p \in \tau(t) \\ 0 & \text{if } p \notin \tau(t) \end{cases} \right] \quad (5)$$

$$\bigwedge_{1 \leq i \leq n, (i,j) \in L} x_{i,\neg} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^{\tau} = \delta \cdot \max(0, 1 - y_{j,t}^{\tau}) \right] \quad (6)$$

$$\bigwedge_{1 \leq i \leq n, (i,j) \in L, (i,j') \in R} x_{i,\wedge} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^{\tau} = \delta \cdot y_{j,t}^{\tau} \cdot y_{j',t}^{\tau} \right] \quad (7)$$

$$\bigwedge_{1 \leq i \leq n, (i,j) \in L, (i,j') \in R} x_{i,\vee} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^{\tau} = \delta \cdot \frac{y_{j,t}^{\tau} + y_{j',t}^{\tau}}{2} \right] \quad (8)$$

$$\bigwedge_{\substack{1 \leq i \leq n \\ (i,j) \in L}} x_{i,G} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \delta \cdot \sum_{t \leq t' < |\tau|} r^{t'-t} \cdot y_{j,t'}^\tau \right] \wedge \left[\bigwedge_{t \leq t' < |\tau|} (y_{j,t'}^\tau > 0) \right] \quad (9)$$

$$\bigwedge_{1 \leq i \leq n, (i,j) \in L} x_{i,F} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} \exists t'. \left[y_{i,t}^\tau = \delta \cdot r^{t'-t} \cdot y_{j,t'}^\tau \right] \wedge (t \leq t') \wedge \left[\bigwedge_{t < t'' < |\tau|} \neg(y_{j,t''}^\tau > 0) \right] \wedge (y_{j,t'}^\tau > 0) \right] \quad (10)$$

Corresponding to boolean variables and to each operator, the constraints encode the calculation of the valuation of a given node as a function of the valuation of its children, as defined in section 4. The score for a node labelled G at a position t in the trace is described by the constraint (9). The first conjunct encodes the actual score as a function of its child, adding the child's score over all positions in the input word, scaled by an exponential. The second conjunct simply ensures that the G-property holds in a classical sense, i.e., that its child has positive valuation at all positions. Similarly, F is encoded in constraint (10), where we look for the first position t' where its child has positive valuation. The score of its child is exponentially scaled so as to diminish the contribution from an occurrence far away from the start.

Finally, we optimize the score of the entire syntax tree using the score of the root node $y_{1,0}^\tau$ w.r.t. the constraint system Φ_d^S (see Algorithm 1). Then we use the resulting model to label the tree, obtaining the optimal formula of chosen depth. By iterating over d , we may obtain the minimal such formula.

5.2 Optimized Pattern Matching

We now present a variation of Algorithm 1 where the input consists of a sample along with a user provided formula pattern, where the propositional variables are unknown. This approach is in the spirit of [3] where we take a formula template instead of just a depth as in Algorithm 1. We generate a static syntax tree by parsing the given pattern, with placeholder propositional variables becoming the leaves. In addition to the constraints discussed in Section 1, constraint (11) ensures the mapping of the placeholder variables (also called pattern variables) to exactly one variable in the given sample, where Var represents the set of variables in the given pattern and $m_{x,p}$ stands for the mapping of pattern variable x to the sample variable p .

$$\varphi_P^{Var} = \left[\bigwedge_{x \in Var} \bigvee_{p \in P} m_{x,p} \right] \wedge \left[\bigwedge_{x \in Var} \bigwedge_{p \neq p' \in P} \neg m_{x,p} \vee \neg m_{x,p'} \right] \quad (11)$$

The constraint (11) specifies that each pattern variable x is mapped to a sample variable, and no pattern variable is mapped to two sample variables.

As before, we optimize the score of the entire tree over the sample w.r.t. these constraints to find the appropriate formula satisfying the pattern. This technique is much faster than a constraint system for the entire tree, and is very useful when the type of property to be mined for is known beforehand, such as safety or liveness properties. An `unsat` result indicates the pattern cannot hold. Such a result can be used to verify the said safety properties.

Algorithm 2 Computing the optimal formula given a partial pattern

```

1: procedure HYBRIDPATTERN( $\mathcal{S} = (P, N)$ , pattern)       $\triangleright$  Returns optimal formula
   fitting a pattern
2:   parse pattern
3:   construct  $\varphi_P^{Var}$  for propositional patterns in tree
4:   construct  $\varphi_n^{ST}$  for every subformula pattern  $\varphi(n)$  in the tree
5:   constraint  $\Phi \leftarrow \varphi_P^{Var} \wedge \bigwedge \varphi_n^{ST}$ 
6:   maximize  $\min(\{y_{1,0}^{\tau} \mid \tau \in P\})$  with  $\Phi$  as constraint
7:   if optimization succeeds with model m then                       $\triangleright$  SAT
8:     construct formula tree from m
9:     return optimized formula tree
10:  else
11:    return UNSAT

```

5.3 Hybrid Pattern Matching

The algorithms defined in sections 5.1 and 5.2 suffers from a common drawback, though at different ends of the spectrum. In the first, we work with increasing depth to find the optimal formula, constraint sizes may grow quickly with increase in the depth. In the second, we start with a formula template and many formulae are not considered since we are guided by the template pattern. This makes this approach *insufficiently* expressive in comparison with constrained system optimization.

To remedy this, we introduce a middle ground, where, instead of attempting to learn formulae from scratch or from explicit patterns, we learn *subformulae* within some pattern. A subformulae argument $\varphi(d)$ with d being a prescribed maximum depth for the subtree is provided as part of the pattern, parsed into the tree as an abstracted empty formula with constraints constructed for the specified nodes explicitly, and for the subformulae recursively in the manner as described for section 5.1 (Algorithm 2). In Figure 3, we show an example of the hybrid pattern $G(\varphi(2) \vee x)$, where $\varphi(2)$ is an unknown formula of depth ≤ 2 and x is an unknown proposition.

5.4 Composition Ranking

We describe here an alternative greedy search for optimal formula, which bypasses constraint solving and optimizations, by pruning the search space of formulae. We begin by enumerating all formulae of depth zero, i.e., all literals in our system as obtained after parsing input traces, we consider all compositions of these with the operators present. After enumerating the literals, we then perform an “F-check”. For any formula φ , the F-check tests whether in any of the input samples, $F\varphi$ holds. If a formula passes a F-check, it is kept in the set to produce formulae of higher depth, and is removed otherwise (see Algorithm 3). Due to this greedy pruning of the search space, however, this algorithm restricts itself to recognizing only certain kinds of properties. As an example, properties that partition the traces like $Gp \vee Gq$ get pruned away by the F-check as their subformulae may not be consistent on all the traces. Formulae of the form $G(p \rightarrow Fs)$ have

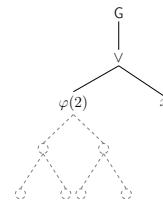


Figure 3: Subtree for hybrid pattern matching

Algorithm 3 Compositional Ranking

```
1: procedure COMPRANK( $\mathcal{S} = (P, N)$ ,  $depth$ )  $\triangleright$  Returns list of satisfying formulae  
sorted by score  
2:   curr_depth  $\leftarrow 0$ , formulae  $\leftarrow \{\text{literals in } \mathcal{S}\}$   
3:   used  $\leftarrow \{\}$ , unary  $\leftarrow \{G, F\}$ , binary  $\leftarrow \{\wedge, \vee\}$   
4:   while  $curr\_depth \neq depth$  do  $\triangleright$  Search till user required depth  
5:     for all  $f \in \text{formulae}$  do  
6:       if  $F(f)$  does not hold on any trace then  $\triangleright$  F-Check  
7:         formulae  $\leftarrow \text{formulae} \setminus \{f\}$   
8:       used  $\leftarrow \text{used} \cup \text{formulae}$   
9:       if  $curr\_depth \neq depth - 1$  then  
10:        formulae  $\leftarrow \bigcup_{T \in \text{unary}} T(\text{used}) \cup \bigcup_{T \in \text{binary}} T(\text{used}, \text{used})$   
11:        curr_depth  $\leftarrow curr\_depth + 1$   
12:    scores  $\leftarrow \{(f, V(f, P, N)) | f \in \text{formulae}\}$   
13: return sort(scores)  $\triangleright$  sort list w.r.t. scores
```

reduced recognition at higher depths (<50% accuracy at depth 4 and above). This is due to the algorithm tending to latch onto “steady” properties (of the form $G \dots$) instead of “impulsive” properties (of the form $p \rightarrow \dots$). For example, on a sample having traces generated using the formula $G(q \rightarrow (G(p \rightarrow Fs)))$, the algorithm returns $G(G(q \wedge p) \rightarrow Fs)$ (see Figure 5)(b). While this formula gives us useful information about the underlying property, it disregards the impulsive nature of the response generated by the implication, i.e., s should occur *after* q and p have appeared. This does however present an opportunity to run the tool on subsets of traces with some human guidance. In essence, this scoring scheme can be leveraged to recognize steady-state temporal properties with high performance at the cost of losing completeness. In our implementation, we have more custom options for users to prioritise certain parts of search space (see appendix A).

6 Experiments

We have implemented the above algorithm in a tool called QUANTLEARN. In this section, we present the results of QUANTLEARN on a set of synthetic traces and also analyze the traces generated by the dining philosopher problem. QUANTLEARN is implemented in C++. QUANTLEARN is publicly available at <https://github.com/sankalpgambhir/quantl>

For the optimization, it takes a set of positive traces, a (possibly empty) set of negative traces, and a formula template (which can simply be $\varphi(n)$, a search depth of n with no specification) or a combination, while the compositional ranking, takes as input, the traces along with a search depth. Our implementation uses SMT solver Z3 [21] for the optimizations. We performed tests with TensorFlow as well, but in preliminary testing, we saw unreliable convergence and results, possibly attributable to the highly non-linear binary constraint system. The performance could be improved by using more advanced numerical optimization techniques at the cost of losing determinism. Z3 in particular was chosen due to its ability to handle high levels of non-linearity and large disjunctive constraints. All our queries to Z3 are quantifier-free. For optimization, QUANTLEARN returns

a formula with maximal score per our scheme, while for compositional ranking, it returns a list of all formulae found, sorted by score. In our experiments, we used the retardation factor r as e^{-1} and also used 0.8 to decay each time we build deeper formula in order to bias the ranking towards simpler formulae. We evaluated the performance of QUANTLEARN on a 64-bit Linux system with an AMD Renoir Ryzen 5 (4500U) laptop CPU. We used 1000 seconds timeout in our experiments. Now we present our the experimental results.

6.1 Performance on synthetic data

We considered the samples generated from the languages of commonly used LTL formulae [23,4]. These are the properties checking *Absence*, *Existence* and *Universality*. We also added to this list, *Response*. We present 8 formulae representing these common LTL patterns in Figure 4. We generated 88 samples from these formulae (11 samples per formulae, of varying length) each containing 20 positive traces and 1 negative trace by running their equivalent automaton and resolving non-determinism probabilistically. Increasing the number of negative traces speeds up convergence due to better pruning, so the worst case was chosen, modelling a real-world scenario where obtaining negative or 'faulty' system traces may be impossible. The sample trace lengths range between 10 and 10^4 . Furthermore, we use random unrelated variables in the traces. We evaluated QUANTLEARN on various partial formulae from these as well.

Figure 4: Common LTL formulae [4][23]
Property Formulae

Property	Formulae
Absence	$G\neg p$, $G(q \rightarrow G(\neg p))$
Response	$G(p \rightarrow F_s)$, $G(q \rightarrow G(p \rightarrow F_s))$
Existence	Fp , $G(\neg p \vee F(p \wedge Fq))$
Universality	Gp , $G(p \rightarrow Gq)$

The samples generated using formulae in Figure 4 have been used for the experiments in Figure 5.

Runtimes with optimization for scores.

In Figure 5a, we show the running times with different patterns of user specification over the different lengths of traces for constraint system optimization and optimized pattern matching. We generate the traces by randomly deciding whether a proposition should occur at a point, constraining that it is allowed by the formula itself. For adding noise, a maximum number of 'noisy' propositional variables were generated, which did not occur in the formula, and with a probability p_{noise} each could occur at any position in the trace. In our experiments, QUANTLEARN is able to extract and learn formulae of up to depth 3 reliably over several test runs, with high resilience to noise (upto $p_{noise} = 75\%$ with 3 noisy variables does not affect results for trace sizes over 20), while extracting them often from single traces as small as 10-15 in length. Due to the nature of our algorithm, the runtime increases significantly as the traces size increases. However, we observe that QUANTLEARN can operate in a scenario where the unknown formulae are up to depth 3 and traces are of effective size upto 2000 (here, 20 traces of size 100), to extract the generator or a similar property reliably. Outside of Figure 5, we tested the response property of depth 3, generating traces from $G(p \rightarrow F_s)$, and QUANTLEARN learnt the optimal formula $(G\neg p) \vee G(F_s)$ which is functionally quite similar. This trend however does not continue with increase in trace size due to the excess non-linearity generated by added trace points and the op-

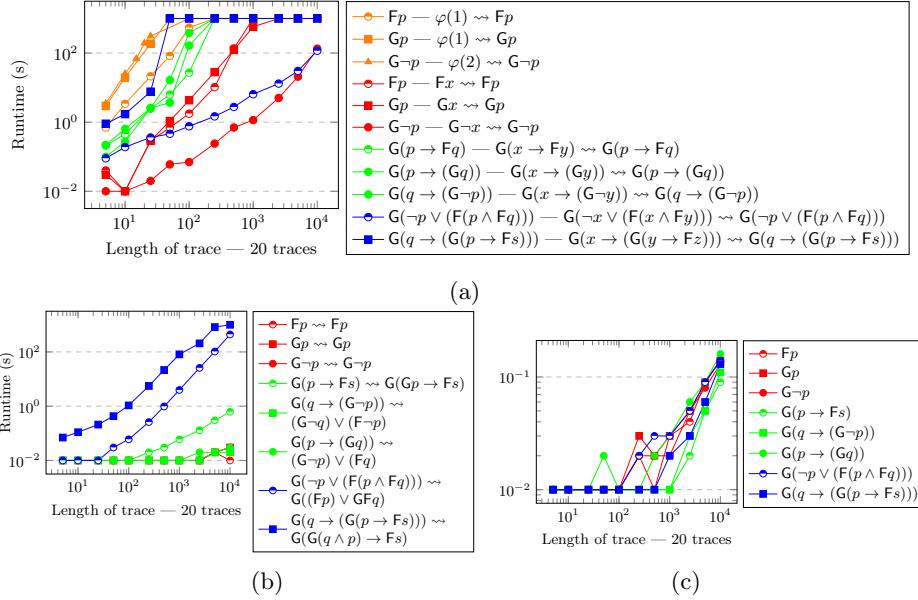


Figure 5: Runtime for traces generated for formulae. (a) full and partial pattern specification. (b) compositional ranking, and only a depth as input. (c) Texada [10], with complete pattern specification. In the legend of (a), $\varphi - pattern \rightsquigarrow \psi$ represents the following : φ is the formula used to generate the traces; $pattern$ is the input to QUANTLEARN and ψ is the learned output formula. Similarly in the legend of (b), $\varphi \rightsquigarrow \psi$ indicates the traces in the sample were generated using φ , and ψ is the learnt formula.

timization timed out for traces with effective size over a few thousands (summed over 20 traces). This can be countered by guiding the tool and providing partial patterns instead. As we increase the level of user specification using patterns of higher depth, we are able to find formulae of up to depth 6, with a correspondingly raised limit of reasonable execution time with increasing trace sizes. For some of the long traces and deep search, QUANTLEARN timed out. However, QUANTLEARN is able to reliably extract properties with no or partial inputs. The state-of-the-art tool Texada does not handle the case.

Runtimes with compositional ranking for scores. In Figure 5b we show runtimes for synthetic traces generated from formulae in Figure 4 with composition ranking. The length of the individual traces is varied from 5 to 10,000. QUANTLEARN scales well for tested sizes (effective size 200,000). As the complexity of formulae increases, it focuses on steady temporal patterns more often than impulse patterns, as discussed in subsection 5.4. The tool returns a list of formulae ranked by score. The top result is shown in the figure for each of the formulae used to generate the traces.

Runtimes of Texada. In Figure 5c we show runtimes for the generated traces with Texada. The comparison with the tool is difficult because it requires complete formula pattern as input while we do not. The tests are performed with

Input	Output	Interpretation
$G(\varphi(1))$	$GFp1 \text{ is thinking}$	Liveness property
$G(x \rightarrow \varphi(1))$	$G((p4 \text{ is eating}) \rightarrow \neg(p3 \text{ is eating}))$	Mutual exclusion
$G(x \rightarrow Fy \wedge Fz)$	$G((p1 \text{ is hungry}) \rightarrow (F((p1 \text{ is eating}) \wedge F(p1 \text{ is thinking}))))$	Deadlock freedom

Figure 6: Results of running QUANTLEARN on Dining Philosophers traces

complete formula pattern as input, with which Texada outputs a list of possible formulae with propositions substituted in. For formula depth less than 4, Texada and compositional ranking perform comparably despite QUANTLEARN requiring no pattern input.

6.2 Mining Formulae from the traces of Dining Philosophers

The dining philosophers problem [24] is a widely used example of a control problem in distributed systems and has become an important benchmark for testing expressiveness of concurrent languages and resource allocation strategies. We consider the problem with five philosophers p1, p2, p3, p4, p5 sitting at a round table. They are been served food, with a fork placed between each pair. Each philosopher proceeds to think till they are hungry, after which they attempt to pick up the forks on both of their sides, eating till they are full, but only when forks on both sides are available. After they are done eating, they put the forks down back on to either of their sides, and continue thinking. The goal is to establish *lockout-freedom*, i.e., each hungry philosopher is eventually able to eat. We use QUANTLEARN to mine LTL formulae from a trace of size 250 from Texada tests [10]. We searched several mined properties using different templates presented in Figure 6.

- When we gave the pattern $G(\varphi(1))$ (invariant, depth 1) to QUANTLEARN and required it to learn a property, we obtained the property $GFp1 \text{ is thinking}$. The mined property acts as a verification of the liveness of the system. QUANTLEARN took 12 seconds to find the property. While this provides a general fact about the system, building upon this result, we can guide the tool to find more relevant properties for higher depths.
- When we gave the pattern $G(x \rightarrow \varphi(1))$ to QUANTLEARN and required it to learn a property, we obtained the property $G((p4 \text{ is eating}) \rightarrow \neg(p3 \text{ is eating}))$. The mined property illustrates that adjacent philosophers cannot acquire forks at the same time, ensuring that our *lock*, the availability of forks does indeed prevent philosophers from eating. QUANTLEARN took 72 seconds to find the property.
- When we gave the pattern $G(x \rightarrow Fy \wedge Fz)$ to QUANTLEARN and required it to learn a property, we obtained $G((p1 \text{ is hungry}) \rightarrow (F((p1 \text{ is eating}) \wedge F(p1 \text{ is thinking}))))$. This property is a richer demonstration of deadlock freedom for philosopher 1, ensuring that they both enter and exit their *critical section*, i.e., the `eating` state. QUANTLEARN took 165 seconds to find the property.

7 Conclusion and Future work

In this paper, we presented a novel scheme to assign a value to LTL formulae that emphasizes on good representability of a word by the formula more than mere satisfiability by the word. We presented a method that uses the scheme to mine LTL formulae from given traces of reactive systems. We plan to improve our scheme to be more solver friendly. For example, the optimizer currently has trouble with the non-linearity and discreteness of our constraints, and developing a linear, more continuous formalism may improve efficiency.

References

1. Ujjwal M., Sanghamitra B., and Anirban M. *Multiobjective Genetic Algorithms for Clustering - Applications in Data Mining and Bioinformatics*. Springer, 2011.
2. Pallavi M., Rahul G., Aditya K., and Rupak M. Partial order reduction for event-driven multi-threaded programs. In *TACAS 2016*.
3. Caroline L, Dennis P, and Ivan. General ltl specification mining (t). In *ASE 2015*.
4. D. Neider and I. Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.
5. Andreas G, Jan K, and Javier E. Rabinizer: Small deterministic automata for ltl(f, G). In *ATVA 2012*.
6. Jan K and Javier E. Deterministic automata for the (f, g)-fragment of LTL. In *CAV 2012*.
7. Javier E, Jan K, and Salomon S. One theorem to rule them all: A unified translation of LTL into ω -automata. In *LICS 2018*.
8. Sean A Fulop and David Kephart. Topology of language classes. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, pages 26–38, 2015.
9. Austin J Parker, Kelly B Yancey, and Matthew P Yancey. Regular language distance and entropy. *arXiv preprint arXiv:1602.07715*, 2016.
10. Texada. <https://github.com/ModelInference/texada>.
11. Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *ODBASE 2005*.
12. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *ASE 2001*.
13. K Havelund and G Roşu. Synthesizing monitors for safety properties. In *TACAS*, 2002.
14. David Lo, V-Prasad R, G R, and Kapil V. Mining quantified temporal rules: Formalism, algorithms, and evaluation. *Science of Computer Programming*, 2012.
15. Westley W and George C. Mining temporal specifications for error detection. In *TACAS*, 2005.
16. Rakesh A, Dimitrios G, and Frank L. Mining process models from workflow logs. In *EDBT*, 1998.
17. Michael Z M and Michael R. Workflow-based process monitoring and controlling-technical and organizational issues. In *HICSS*, 2000.
18. I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Softw. Eng.*, 2015.
19. Ivan B, Yuriy B, Sigurd S, Michael S, and Michael D E. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE*, 2011.
20. Christel B and Joost-Pieter K. *Principles of model checking*. MIT Press, 2008.

21. Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer Berlin Heidelberg, 2008.
22. Paulo Tabuada and Daniel Neider. Robust linear temporal logic. *arXiv preprint arXiv:1510.08970*, 2015.
23. Matthew B D, George S A, and James C C. Property specification patterns for finite-state verification. In *Second workshop on FMSP*, 1998.
24. Edsger W Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. Springer, 1971.

A Prioritize Variables

Finally, we have added one more heuristic in our implementation. In case there is a large set of events and we want bias the focus of our search towards certain letters in the traces that do not occur very often, we may adjust the value $V(p, w)$ assigned to each propositional variable p . In our default scheme, we assign $V(p, w) = 1$ if $w(1)$ contains p . A user may assign a value greater than 1 to variables p that are desirable and assign less than 1 for the variables p that are not. This allows for mining specifications pertaining to the prioritized variables in cases where several competing well ranked specifications are present. Let π be the map from the propositional variables \mathcal{P} to their priority. We replace equation (5) by the following formula where we return score $\pi(p)$ instead of 1.

$$\bigwedge_{1 \leq i \leq N} \bigwedge_{p \in \mathcal{P}} x_{i,p} \rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \begin{cases} \pi(p) & \text{if } p \in \tau(t) \\ 0 & \text{if } p \notin \tau(t) \end{cases} \right] \quad (12)$$

Recall that we considered the trace in figure 1, where we wanted to find a property $\mathbf{Gauth_failed} \Leftrightarrow \varphi(1)$. We may further abstract the property $\mathbf{G}\varphi(0) \Leftrightarrow \varphi(1)$ by setting the priority map $\pi = \{\mathbf{auth_failed} \mapsto 10, \text{otherwise} \mapsto 1\}$. We obtain the same result from QUANTLEARN.

In the Dining Philosophers problem, we may wish to verify individually whether the properties are being satisfied for a single philosopher (thread). By giving a higher weight to the properties of this philosopher, we can guide the tool to learn the relevant properties and verify them. This can be expanded to studying specific applications or threads in varied noisy data where the target of interest is either known apriori or is inferred from preliminary unguided results.

The suggested variations and their results indicate that our method is viable to be adapted to an application at hand, where we want to bias our ranking to give preference to a desired class of formulae.