Specification Sketching for Linear Temporal Logic

Simon Lutz[1,2(✉)], Daniel Neider[1,2], and Rajarshi Roy[3]

[1] TU Dortmund University, Dortmund, Germany
simon.lutz@tu-dortmund.de
[2] Center for Trustworthy Data Science and Security, UA Ruhr, Dortmund, Germany
[3] Max Planck Institute for Software Systems, Kaiserslautern, Germany

Abstract. Virtually all verification and synthesis techniques assume that formal specifications are readily available, functionally correct, and fully match the engineer's understanding of the given system. However, this assumption is often unrealistic in practice: formalizing system requirements is notoriously difficult, error-prone, and requires substantial training. To alleviate this hurdle, we propose a novel approach of assisting engineers in writing specifications based on their high-level understanding of the system. We formalize the high-level understanding as an LTL *sketch* that is a partial LTL formula, where parts that are hard to formalize can be left out. Given an LTL sketch and a set of examples of system behavior that the specification should or should not allow, the task of a so-called *sketching algorithm* is then to complete the sketch such that the resulting LTL formula is consistent with the examples. We show that deciding whether a sketch can be completed falls into the complexity class NP and present two SAT-based sketching algorithms. Finally, we implement a prototype with our algorithms and compare it against two prominent LTL miners to demonstrate the benefits of using LTL sketches.

1 Introduction

Due to its unique ability to prove the absence of errors mathematically, formal verification is a time-tested method of ensuring the safe and reliable operation of safety-critical systems. Success stories of formal methods include application domains such as communication system [21,34], railway transportation [3,4], aerospace [16,24], and operating systems [30,50] to name but a few.

However, there is an essential and often overlooked catch with formal verification: virtually all techniques assume that the specification required for the design or verification of a system is available in a suitable format, is functionally correct, and expresses precisely the properties the engineers had in mind. These assumptions are often unrealistic in practice. Formalizing system requirements is notoriously difficult and error-prone [9,38,44,45]. Even worse, the training effort required to reach proficiency with specification languages can be disproportionate to the expected benefits [17], and the use of formalisms such as temporal logics require a level of sophistication that many users might never develop [25,28].

É. André and J. Sun (Eds.): ATVA 2023, LNCS 14216, pp. 26–48, 2023.
https://doi.org/10.1007/978-3-031-45332-8_2

To aid the process of formalizing specifications, we introduce a fundamentally novel approach to writing formal specifications, named *specification sketching*. Inspired by recent advances in automated program synthesis [47,48], our new paradigm allows engineers to express their high-level insights about a system in terms of a partial specification, named *specification sketch*, where parts that are difficult or error-prone to formalize can be left out. To single out their desired specification, our paradigm additionally allows the engineers to provide positive (i.e., desirable) and negative (i.e., undesirable) examples of system execution. Based on this additional data, a so-called *sketching algorithm* fills in the missing low-level details to obtain a complete specification.

To demonstrate how our paradigm works, let us consider a simple scenario. Imagine that an engineer wishes to formalize the following request-response property P: every request p has to be answered eventually by a response q. This property can be expressed in Linear Temporal Logic (LTL)—a popular specification language in software verification—as $\mathsf{G}(p \rightarrow \mathsf{X}\,\mathsf{F}\,q)$ using standard temporal modalities F ("Finally"), G ("Globally"), and X ("neXt"). However, for the sake of this example, assume that the engineer is unsure of how exactly to formalize P. In such a situation, our sketching paradigm allows them to express their high-level insights in the form of a sketch, say $\mathsf{G}(p \rightarrow ?)$, where the question mark indicates the missing part of the specification. Additionally, they can provide example executions. Assume that they provide the following infinite executions of the system: (i) a positive execution $\{p\}\{q\}\{p\}\{q\}\{p\}\{q\}\cdots$, in which every request is answered by a response in the next time point, and (ii) a negative execution $\{p\}\{q\}\{p\}\{p\}\{p\}\cdots$, in which there are infinitely many requests that are not answered by a response. Our sketching algorithm then computes a substitution for the question mark such that the completed LTL formula is consistent with the examples (e.g., $? \coloneqq \mathsf{X}\,\mathsf{F}\,q$). In this example, the engineer left out an entire temporal formula in the sketch. However, our paradigm also allows one to leave out Boolean and temporal operators. For instance, one could also provide $?(p \rightarrow \mathsf{X}\,\mathsf{F}\,p)$ as a sketch, where the question mark now indicates a missing unary operator (G in our example).

While the concept of specification sketching can be conceived for a wide range of specification languages, in this work, we focus on Linear Temporal Logic (LTL) [39]. Our rationale behind choosing LTL is threefold. First, LTL is popular in academia and widely used in industry [23,24,27,49], making it the de facto standard for expressing (temporal) properties in verification and synthesis. Second, LTL is well-understood and enjoys good algorithmic properties [15,39]. Third, its intuitive and variable-free syntax have recently prompted several efforts to adopt LTL (over finite words) also in artificial intelligence (e.g., as explainable models [13,43], as reward functions in reinforcement learning [12], etc.). We introduce LTL and other necessary definitions in Sect. 2.

In Sect. 3, we then formally state the problem of specification sketching for LTL (or LTL sketching for short). It turns out that the LTL sketching problem might not always have a solution: there are sketches for which no substitutions exist that makes them consistent with the given examples (see the example at the end of Sect. 3). However, we show in Sect. 4 that the problem of deciding

whether such a substitution exists is in the complexity class NP. Moreover, we develop an effective decision procedure that reduces the original question to a satisfiability problem in propositional logic. This reduction permits us to apply highly-optimized, off-the-shelf SAT solvers to check whether a consistent substitution exists.

In Sect. 5, we develop two sketching algorithms for LTL. Following Occam's razor principle, both algorithms are biased towards finding "small" (concise) substitutions for the question marks in a sketch. The rationale behind this choice is that small formulas are arguably easier for engineers to understand and, thus, can be safely deployed in practice.

By exploiting the decision procedure of Sect. 4 as a sub-routine, our first algorithm transforms the sketching problem into several "classical" LTL learning tasks (i.e., learning of LTL formulas from positive and negative data). This transformation allows us to apply a diverse array of LTL learning algorithms, which have been proposed during the last five years [13,37,41]. In addition, our algorithm immediately benefits from any advances in this field of research.

While the first algorithm builds on top of existing work and, hence, is easy to use, we observed that it tends to produce non-optimal substitutions for the unspecified parts of a sketch. Our second algorithm tackles this by searching for substitutions of increasing size using a SAT-based approach that is inspired by Neider and Gavran [37]. We formally prove that this algorithm can, in fact, produce small substitutions (if they exist).

In Sect. 6, we present an experimental evaluation of our algorithms using a prototype implementation `LTL-Sketcher`. We demonstrate that our algorithms are effective in completing sketches with different types of missing information. Further, we compare `LTL-Sketcher` against two state-of-the-art specification mining tools for LTL. From the comparison, we demonstrate that `LTL-Sketcher`'s ability to complete missing temporal formulas and temporal operators enables it to complete more specifications. Moreover, we observe that providing high-level insights as a sketch reduces the number of examples required to derive the correct specification. Finally, we conclude in Sect. 7 with a discussion on future work. All the proofs and additional experimental results can be found in the extended version of this paper [35].

Related Work. Specification sketching can be seen as a form of specification mining [1]. In this area, the general idea of allowing partial specifications is not entirely new, but it has not yet been investigated as generally as in this work. For instance, a closely related setting is the one in which so-called templates are used to mine temporal specifications from system executions. In this context, a template is a partial formula similar to a sketch. Unlike a sketch, however, a template is typically completed with a single atomic proposition or a simple, usually Boolean formula (e.g., a restricted Boolean combination of atomic propositions). A prime example of this approach is Texada [31,32], a specification miner for LTL_f formulas (i.e., LTL over a finite horizon). Texada takes a template (property type in their terminology) and a set of system executions

as input and completes the template with atomic propositions such that the resulting LTL formula satisfies all system executions. In contrast to Texada, our paradigm assists engineers in completing more complex temporal formulas in their specifications, thus alleviating an even larger burden off an engineer. Another example in this setting is the concept of temporal logic queries, introduced by Chan [14] for CTL, and later developed by Bruns and Godefroid [10] for a wide range of temporal logics. However, unlike our paradigm, temporal logic queries allow only a single placeholder in their template that can be filled with only atomic propositions.

Various other techniques operate in settings where the templates are even more restricted. For example, Li et al. [33] mine LTL specification based on templates from the GR(1)-fragment of LTL (e.g., $\mathsf{G}\,\mathsf{F}?$, $\mathsf{G}(?_1 \rightarrow \mathsf{X}?_2)$, etc.), while Shah et al. [46] mine LTL formulas that are conjunctions of the set of common temporal properties identified by Dwyer et al. [19]. In addition, Kim et al. [29] consider a set of interpretable LTL templates, widely used in the development of software systems, to obtain LTL formulas robust to noise in the input data. In the context of CTL, on the other hand, Wasylkowski and Zeller [51] mine specifications using templates of the form $\mathsf{A}\,\mathsf{F}?$, $\mathsf{A}\,\mathsf{G}(?_1 \rightarrow \mathsf{F}?_2)$, etc. However, all of the approaches above complete the templates only with atomic propositions (and their negations in some cases).

Another setting is where general (and complex) temporal specifications are learned from system executions without any information about the structure of the specification. The most notable work in this setting is by Neider and Gavran [37], who learn LTL formulas from system executions using a SAT solver. Similar to their work is the work by Camacho et al. [13], which proposes a SAT-based learning algorithm for LTL_f formulas via Alternating Finite Automata as an intermediate representation. Raha et al. [40] present a scalable approach for learning formulas in a fragment of LTL_f without the U-operator, while Roy, Fisman, and Neider [42] consider the Property Specification Language (PSL). However, all of these works are "unguided" in that none of them exploit insights about the structure of the specification to aid the learning/mining process.

Finally, it is worth mentioning that LTL sketching can also be seen as a particular case of syntax-guided synthesis (SyGuS), where syntactic constraints on the resulting formulas are expressed in terms of a context-free grammar. An example of a syntax-guided approach is SySLite [2], a CVC4-based tool for learning Past-time LTL over finite executions. However, to the best of our knowledge, we are unaware of any SyGuS engine that can infer specifications in LTL over infinite (i.e., ultimately-periodic) system executions.

2 Preliminaries

We first set up the notation and definitions that are used throughout the paper.

To model trajectories of a system, we exploit the notion of *words* defined over an alphabet consisting of relevant system events. Formally, an *alphabet* Σ is a nonempty, finite set whose elements are called *symbols*. A *finite word* over an

alphabet Σ is a sequence $u = a_0 \dots a_n$ of symbols $a_i \in \Sigma$ for $i \in \{0, \dots, n\}$. The empty sequence, referred to as *empty word*, is denoted by ε. The length of a finite word u is denoted by $|u|$, where $|\varepsilon| = 0$. Moreover, Σ^* denotes the set of all finite words over Σ, while $\Sigma^+ = \Sigma^* \setminus \varepsilon$ denotes the set of all non-empty words.

An *infinite word* over Σ is an infinite sequence $\alpha = a_0 a_1 \dots$ of symbols $a_i \in \Sigma$ for $i \in \mathbb{N}$. We denote the i-th symbol of an infinite word α by $\alpha[i]$ and the finite infix of α from position i up to (and excluding) position j with $\alpha[i, j) = a_i a_{i+1} \cdots a_{j-1}$. We use the convention that $\alpha[i, j) = \varepsilon$ for any $i \geq j$. Further, we denote the infinite suffix starting at position $j \in \mathbb{N}$ by $\alpha[j, \infty) = a_j a_{j+1} \cdots$. Given $u \in \Sigma^+$, the infinite repetition of u is the infinite word $u^\omega = uu \cdots \in \Sigma^\omega$. An infinite word α is called *ultimately periodic* if it is of the form $\alpha = uv^\omega$ for a $u \in \Sigma^*$ and $v \in \Sigma^+$. Finally, Σ^ω denotes the set of all infinite words over Σ.

Since our algorithms rely on the Satisfiability (SAT) problem, as a prerequisite, we introduce *Propositional Logic*. Let *Var* be a set of propositional variables, which take Boolean values from $\mathbb{B} = \{0, 1\}$ (0 representing *false* and 1 representing *true*). Formulas in propositional (Boolean) logic, denoted by capital Greek letters, are inductively constructed as follows:

- each $x \in \mathit{Var}$ is a propositional formula; and
- if Ψ and Φ are propositional formulas, so are $\neg\Psi$ and $\Psi \vee \Phi$.

Moreover, as syntactic sugar, we allow the formulas *true*, *false*, $\Psi \wedge \Phi$,$\Psi \Rightarrow \Phi$, and $\Psi \Leftrightarrow \Phi$, which are defined as usual. A propositional valuation is a mapping $v : \mathit{Var} \to \mathbb{B}$ that assigns Boolean values to propositional variables. The semantics of propositional logic is given by a satisfaction relation $\models$ that is inductively defined as follows: $v \models x$ if and only if $v(x) = 1, v \models \neg\Phi$ if and only if $v \not\models \Phi$, and $v \models \Psi \vee \Phi$ if and only if $v \models \Psi$ or $v \models \Phi$. In the case that $v \models \Phi$, we say that v satisfies Φ and call it a *model* of Φ. A propositional formula Φ is *satisfiable* if there exists a model v of Φ. The *size* of a formula is the number of its subformulas (defined in the usual way). The satisfiability (SAT) problem is the well-known NP-complete problem of deciding whether a given propositional formula is satisfiable. In the recent past, numerous optimized decision procedures have been designed to handle the SAT problem effectively [6].

Linear Temporal Logic is a logic to reason about sequences of relevant statements about a system by using temporal modalities. Formally, given a set $\mathcal{P}$ of propositions that represent relevant statements about a system, an LTL formula, denoted by small Greek letters, is defined inductively as follows:

- each proposition $p \in \mathcal{P}$ is an LTL formula; and
- 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 allow standard Boolean formulas such as *true*, *false*, $\psi \wedge \varphi$, and $\psi \rightarrow \varphi$ and temporal formulas such as $\mathsf{F}\,\psi := \mathit{true}\,\mathsf{U}\,\psi$ ("Finally") and $\mathsf{G}\,\psi := \neg\,\mathsf{F}\,\neg\psi$ ("Globally"). While we restrict to these formulas, our paradigm extends naturally to all temporal operators (e.g., "Release", "Weak until", etc.).

LTL formulas are interpreted over infinite words $\alpha \in (2^\mathcal{P})^\omega$. To define how an LTL formula is interpreted on a word, we use a valuation function

V. This function maps an LTL formula and a word to a Boolean value and is defined inductively as: $V(p, \alpha) = 1$ if and only if $p \in \alpha[0]$, $V(\neg\varphi, \alpha) = 1 - V(\varphi, \alpha)$, $V(\varphi \vee \psi, \alpha) = \max\{V(\varphi, \alpha), V(\psi, \alpha)\}$, $V(\mathsf{X}\,\varphi, \alpha) = V(\varphi, \alpha[1, \infty))$, and $V(\varphi\,\mathsf{U}\,\psi, \alpha) = \max_{i \geq 0}\{\min\{V(\psi, \alpha[i, \infty)), \min_{0 \leq j < i}\{V(\varphi, \alpha[j, \infty))\}\}\}$. We call $V(\varphi, \alpha)$ the *valuation of* φ *on* α and say that α *satisfies* φ if $V(\varphi, \alpha) = 1$.

For a graphical representation of LTL formulas, we rely on *syntax DAGs*. A syntax DAG is a directed acyclic graph (DAG) obtained from the syntax tree of a formula by merging the common subformulas, resulting in a canonical representation. Figure 1 illustrates the syntax DAG of the formula $(p\,\mathsf{U}\,\mathsf{G}\,q) \vee \mathsf{F}\,\mathsf{G}\,q$.

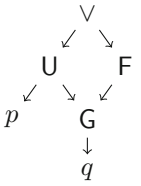


Fig. 1. Syntax DAG of $(p\,\mathsf{U}\,\mathsf{G}\,q) \vee (\mathsf{F}(\mathsf{G}\,q))$

The size of an LTL formula $|\varphi|$ is defined as the number of unique subformulas, which also corresponds to the number of nodes in the syntax DAG of φ. For instance, the size of the formula in Fig. 1 is six.

We denote the set of all LTL operators as $\Lambda = \mathcal{P} \cup \Lambda_U \cup \Lambda_B$. Here, the propositions are the nullary operators, $\Lambda_U = \{\neg, \mathsf{X}, \mathsf{F}, \mathsf{G}\}$ are the unary operators and $\Lambda_B = \{\vee, \wedge, \mathsf{U}\}$ are the binary operators of LTL. Further, let $\mathcal{F}_{\text{LTL}}$ denote the set of all LTL formulas.

3 Problem Formulation

Since the problem of *LTL sketching* relies heavily on LTL sketches, we begin with formalizing them first.

LTL Sketch. An *LTL sketch* is an incomplete LTL formula in which parts that are difficult to formalize can be left out. The left-out parts are represented using placeholders, denoted by ?'s. An example of an LTL sketch can be seen in Fig. 2. We comment on the superscripts on the placeholders in the figure shortly.

Formally, an LTL sketch $\varphi^?$ is simply an LTL formula whose syntax is augmented with placeholders. The placeholders we allow can be of three types: placeholders of arity zero referred to as Type-0 placeholders, that replace missing LTL formulas; placeholders of arity one referred to as Type-1 placeholders, that replace missing unary operators; and placeholders of arity two referred to as Type-2 placeholders, that replace missing binary operators. In Fig. 2 (and throughout the paper), Type-i placeholders are represented using $?^i$.

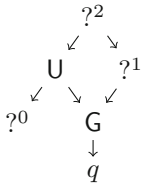


Fig. 2. An LTL sketch

Given (possibly empty) sets Π^0, Π^1 and Π^2 consisting of Type-0, Type-1 and Type-2 placeholders, respectively, we define LTL sketches inductively as follows:

- each element of $\mathcal{P} \cup \Pi^0$ is an LTL sketch; and
- if $\varphi_1^?$ and $\varphi_2^?$ are LTL sketches, $\circ\,\varphi_1^?$ is an LTL sketch for $\circ \in \Lambda_U \cup \Pi^1$ and so is $\varphi_1^? \circ \varphi_2^?$ for $\circ \in \Lambda_B \cup \Pi^2$.

Note that an LTL sketch in which $\Pi^0 = \Pi^1 = \Pi^2 = \emptyset$ is simply an LTL formula. Further, let $\Pi_{\varphi^?} = \Pi^0 \cup \Pi^1 \cup \Pi^2$ denote the set of all placeholders in an sketch $\varphi^?$. For the sketch in Fig. 2, $\Pi_{\varphi^?} = \{?^0, ?^1, ?^2\}$. For brevity, in the rest of the paper, we refer to an LTL sketch as a sketch.

The placeholders are abstract symbols that apriori do not have meaning. To assign meaning to a sketch, we need to substitute all Type-0 placeholders with LTL formulas, all Type-1 placeholders with unary operators, and all Type-2 placeholders with binary operators. We do this using a so-called substitution function (or substitution for short).

Formally, a *substitution* function s maps placeholders and operators present in a sketch to LTL operators and LTL formulas in such a way that: $s(?) \in \mathcal{F}_{\text{LTL}}$ if $? \in \Pi^0$; $s(?) \in \Lambda_U$ if $? \in \Pi^1$; $s(?) \in \Lambda_B$ if $? \in \Pi^2$; and $s(\lambda) = \lambda$ for any LTL operator $\lambda \in \Lambda$. Moreover, a substitution s is said to be *complete* for a sketch $\varphi^?$ if s is defined for every element in $\Lambda \cup \Pi_{\varphi^?}$ in $\varphi^?$. For example, a possible complete substitution s for the sketch $\varphi^?$ in Fig. 2 can be $s(?^0) = p$, $s(?^1) = \mathsf{F}$, $s(?^2) = \vee$, and $s(\lambda) = \lambda$ for $\lambda \in \Lambda$.

A complete substitution s can be applied to a sketch $\varphi^?$ to obtain an LTL formula. To make this precise, we define a function f_s, which is defined recursively on the structure of $\varphi^?$ as: $f_s(\varphi_1^? \; ?^2 \; \varphi_2^?) = f_s(\varphi_1^?) \circ f_s(\varphi_2^?)$, where $\circ = s(?^2)$; $f_s(?^1\varphi^?)) = \circ f_s(\varphi^?)$, where $\circ = s(?^1)$; $f_s(?^0) = s(?^0)$; and $f_s(\varphi^?) = \varphi^?$ if $\Pi_{\varphi^?} = \emptyset$. For the complete substitution s for $\varphi^?$ defined in the last paragraph we get $f_s(\varphi^?) = (p \,\mathsf{U}\, \mathsf{G}\, q) \vee (\mathsf{F}(\mathsf{G}\, q))$.

Input Sample. While there can be many ways to complete a sketch, we direct our search based on two finite, disjoint sets: a set P of positive executions and a set N of negative executions. We consider the executions to be ultimately periodic words, i.e., words of the form uv^ω, where $u \in (2^{\mathcal{P}})^*$ and $v \in (2^{\mathcal{P}})^+$, since they are sufficient to uniquely characterize ω-regular languages [11] (and thus, LTL formulas). We accumulate all the executions in what we call a *sample* $\mathcal{S} = (P, N)$ where $P \cap N = \emptyset$. We define its size to be $|\mathcal{S}| = \sum_{uv^\omega \in P \cup N} |uv|$.

We say that an LTL formula φ is *consistent* with a sample $\mathcal{S} = (P, N)$ if $V(\varphi, uv^\omega) = 1$ for each $uv^\omega \in P$ (i.e., all positive words satisfy φ) and $V(\varphi, uv^\omega) = 0$ for each $uv^\omega \in N$ (i.e., all negative words do not satisfy φ).

The LTL Sketching Problem. We now state the central problem of the paper.

Problem 1 (LTL sketching). Given an LTL sketch $\varphi^?$ and a sample $\mathcal{S} = (P, N)$, find a complete substitution s for $\varphi^?$ such that $f_s(\varphi^?)$ is consistent with $\mathcal{S}$.

Unlike the classical *LTL learning* problem [37], a solution to the *LTL sketching* problem does not always exist. This can be illustrated using the following simple example. Consider the sketch $\mathsf{G}(?^0)$ and a sample consisting of a single positive word $\alpha = \{p\}\{q\}^\omega$ and a single negative word $\beta = \{q\}^\omega$. For this sketch and sample, there does not exist any substitution that leads to an LTL formula consistent with the sample. Towards contradiction, let us assume that there exists an LTL formula $\mathsf{G}(\varphi)$ such that $V(\mathsf{G}(\varphi), \alpha) = 1$ and $V(\mathsf{G}(\varphi), \beta) = 0$.

Based on the semantics of the G-operator, $V(\mathsf{G}(\varphi), \alpha) = V(\mathsf{G}(\varphi), \alpha[1, \infty)) = 1$. On the other hand, $V(\mathsf{G}(\varphi), \beta) = V(\mathsf{G}(\varphi), \alpha[1, \infty)) = 0$ since $\beta = \alpha[1, \infty)$.

Since, for a given LTL sketch and a sample, there might not exist any complete substitution, a naive enumeration-like algorithm to search over all substitutions may not terminate. To show that one can indeed design a terminating sketching algorithm, in the next section, we prove the decidability of *LTL sketching*.

4 Existence of a Complete Sketch

To devise a terminating algorithm for the *LTL sketching* problem, we first introduce the related decision problem, which is the following:

Problem 2 (LTL sketch existence). Given an LTL sketch $\varphi^?$ and a sample $\mathcal{S} = (P, N)$, does there exist a complete substitution s for $\varphi^?$ such that $f_s(\varphi^?)$ is consistent with $\mathcal{S}$.

In what follows, we prove that this problem is indeed decidable and belongs to the complexity class NP. Thereafter, we devise a decision procedure for the problem by exploiting the satisfiability (SAT) problem.

4.1 The Decidability Result

For the decidability result, we begin by introducing some concepts as a preparation. Let us first observe the following key property of ultimately periodic words.

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

This observation indicates that, for a word uv^ω, there exists only a finite number of distinct suffixes of uv^ω, all of which originate in the initial uv portion of uv^ω. Let us then define $suf(uv^\omega) = \{uv^\omega[i, \infty) \mid 0 \leq i < |uv|\}$ as the set of all (possibly) distinct suffixes of a word uv^ω. Moreover, let $suf(\mathcal{S}) = \bigcup_{uv^\omega \in (P \cup N)} suf(uv^\omega)$ to be the set of suffixes of all words in $\mathcal{S}$. Now, Observation 1 also indicates that, to determine the evaluation of an LTL formula φ on an ultimately periodic word uv^ω, it is sufficient to determine its evaluation on the initial $|uv|$ suffixes of uv^ω.

Thus, for a compact representation of the evaluation of φ on uv^ω, we introduce a table notation $T^{\varphi}_{uv^\omega}$. Mathematically speaking, a table $T^{\varphi}_{uv^\omega}$ is a $|\varphi| \times |uv|$ matrix that consists of the satisfaction of all the subformulas φ' of φ on the suffixes of uv^ω. We define the entries of this matrix as: $T^{\varphi}_{uv^\omega}[\varphi', t] = V(\varphi', uv^\omega[t, \infty))$ for all subformulas φ' of φ and $0 \leq t < |uv|$.

Based on the above definition of the table $T^{\varphi}_{uv^\omega}$, we identify three properties of these tables, which form the main building blocks of the decidability proof (i.e., proof of Theorem 1), as we see later.

The first property, or as we call it, the *Semantic* property, is that various rows of the table are related to each other in a way that reflects the semantics of LTL. To explain this further, we use $T^{\varphi}_{uv^\omega}[\varphi', \cdot]$ to represent the row of $T^{\varphi}_{uv^\omega}$ corresponding to the subformula φ'.

We first demonstrate the Semantic property on an example. Consider the formula $\psi = p \vee \mathsf{X}\, q$ and the word $\alpha = \{p, q\}\{p\}\{q\}^\omega$. The table T^ψ_α is illustrated in Fig. 3. From the figure, one can see that the row $T^\psi_\alpha[p \vee \mathsf{X}\, q, \cdot]$ corresponds to the bitwise-OR of the rows $T^\psi_\alpha[p, \cdot]$ and $T^\psi_\alpha[\mathsf{X}\, q, \cdot]$, reflecting the semantics of the $\vee$-operator that combines formulas p and $\mathsf{X}\, q$.

	0	1	2
p	1	1	0
q	1	0	1
$\mathsf{X}\, q$	0	1	1
$p \vee \mathsf{X}\, q$	1	1	1

Fig. 3. Table T^ψ_α for $\psi = p \vee \mathsf{X}\, q$ and $\alpha = \{p, q\}\{p\}\{q\}^\omega$

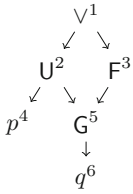


Fig. 4. Syntax DAG of $(p\,\mathsf{U}\,\mathsf{G}\, q) \vee \mathsf{F}(\mathsf{G}\, q)$ with identifiers (in superscripts)

To define these semantic relations between the rows, we must uniquely identify the subformula that corresponds to each row. As a result, we assign unique identifiers $i \in \{1, \ldots, n\}$ to each node of the syntax DAG of φ enabling us to denote the subformula rooted at Node i using $\varphi[i]$. For assigning identifiers, we follow the strategy that: (i) we assign the root node with 1; and (ii) we assign each node with an identifier smaller than its children (i.e., if it has any). Note that one can analogously assign identifiers to syntax DAGs of sketches. Figure 4 demonstrates identifiers for the formula $(p\,\mathsf{U}\,\mathsf{G}\, q) \vee \mathsf{F}(\mathsf{G}\, q)$. We further define a function $\ell\colon \{1, \ldots, n\} \mapsto \Lambda$ that maps the identifiers to the corresponding operators in the syntax DAG.

We now describe the set of equations that formalize the relation between the rows. How a row $T^\varphi_{uv^\omega}[\varphi[i], \cdot]$ relates to the others depends on the operator $\ell(i)$ in the root node of $\varphi[i]$. For instance, if $\ell(i) = p$ for some proposition p, then we have the following relation:

$$T^\varphi_{uv^\omega}[\varphi[i], t] = \begin{cases} 1 \text{ if } p \in uv^\omega[t] \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

If, on the other hand, $\ell(i)$ is a X-operator and Node j is the left child of Node i, we have the following relation:

$$T^\varphi_{uv^\omega}[\varphi[i], t] = \begin{cases} T^\varphi_{uv^\omega}[\varphi[j], t+1] & \text{for } 0 \le t < |uv| - 1 \\ T^\varphi_{uv^\omega}[\varphi[j], |u|] & \text{for } t = |uv| - 1 \end{cases} \tag{2}$$

The above equation exploits the semantics of the X-operator. Further, it exploits Observation 1 and determines the entry $T^\varphi_{uv^\omega}[\varphi[i], |uv| - 1]$ using the evaluation of $\varphi[j]$ at $uv^\omega[|u|, \infty)$, i.e., the start of the periodic part.

If $\ell(i)$ is a $\vee$-operator, and Node j and Node j' are the left and right children of Node i, respectively, then we have the following relation:

$$T^{\varphi}_{uv^\omega}[\varphi[i],t] = T^{\varphi}_{uv^\omega}[\varphi[j],t] \vee T^{\varphi}_{uv^\omega}[\varphi[j'],t] \text{ for } 0 \leq t < |uv| \tag{3}$$

Again, one can see that the above equation follows from the semantics of the $\vee$-operator. For other LTL operators, the relation between rows follows the semantics of the corresponding LTL operator in a similar fashion (see extended version [35] for details). Whenever necessary, we use Observation 1 to determine the semantics of the operator by "looping" around in the period part of uv^ω.

Next, we describe the second property, the *Consistency* property. This property ensures that $T^{\varphi}_{uv^\omega}[\varphi,0] = 1$ if and only if uv^ω satisfies φ. Thus, for an LTL formula φ consistent with $\mathcal{S}$, we have the following relation:

$$T^{\varphi}_{uv^\omega}[\varphi,0] = 1 \text{ for all } uv^\omega \in P, \text{ and } T^{\varphi}_{uv^\omega}[\varphi,0] = 0 \text{ for all } uv^\omega \in N \tag{4}$$

The final property we observe is called the *Suffix* property. This property originates from the fact that LTL, being a future-time logic, has the same evaluation on equal suffixes, i.e., $V(\varphi, u_1v_1^\omega[t,\infty)) = V(\varphi, u_2v_2^\omega[t',\infty))$ for $u_1v_1^\omega[t,\infty) = u_2v_2^\omega[t',\infty)$. Formally, we state the property as follows:

$$T^{\varphi}_{u_1v_1^\omega}[\varphi,t] = T^{\varphi}_{u_2v_2^\omega}[\varphi,t'] \text{ for all } u_1v_1^\omega[t,\infty) = u_2v_2^\omega[t',\infty) \tag{5}$$

This property becomes significant later, especially for constructing LTL formulas to substitute Type-0 placeholders.

With the prerequisites set up, we now proceed to describe an NP algorithm for deciding the *LTL sketch existence* problem. For an easy presentation of the algorithm, we consider the simple (but crucial) case where the only missing information in $\varphi^?$ is a single Type-0 placeholder. While one might assume that non-deterministically guessing a substitution for the placeholder should suffice; it does not. This is because, apriori, the size of the LTL formula required to substitute the Type-0 placeholder is not known.

Thus, in our NP algorithm, instead of guessing substitutions, we guess the entries of the table $T^{\varphi^?}_{uv^\omega}$ for each uv^ω in $\mathcal{S}$. Note that the tables have a finite dimension, precisely $|\varphi^?| \times |uv|$, for each word uv^ω. Thus, the overall process of simply guessing the table entries can be done in time $\mathcal{O}(\mathsf{poly}(|\varphi^?|, |\mathcal{S}|))$.

After guessing the table entries, we must verify that the guessed tables satisfy the three properties, Semantic, Consistency, and Suffix, discussed earlier in this section. It is easy to verify that checking the first two properties for the tables requires time $\mathcal{O}(\mathsf{poly}(|\varphi^?|, |uv|))$ (i.e., polynomial in $|\varphi^?|$ and $|uv|$) for each uv^ω in $\mathcal{S}$. For checking the Suffix property, one must identify the equal suffixes in $suf(\mathcal{S})$. This can be also done in time $\mathcal{O}(\mathsf{poly}(|\mathcal{S}|))$, simply by unrolling the periodic part of the suffixes to a fixed length (see extended version [35] for the details).

This algorithm naturally also extends to multiple Type-0 placeholder. The following lemma now asserts that if the guessed tables satisfy the three properties, then one can find a suitable complete LTL formula. We present a proof sketch of the lemma here (for the full proof, see the extended version [35]).

Lemma 1. *Let $\mathcal{S} = (P, N)$ be a sample and $\varphi^?$ be a sketch with only Type-0 placeholders. Then, the following holds: there exists tables $T^{\varphi^?}_{uv^\omega}$ (i.e., $|\varphi^?| \times |uv|$*

matrices with $\{0,1\}$ *entries) for each* $uv^\omega \in P \cup N$ *that satisfy the Semantic, Consistency, and Suffix properties if and only if there exists a substitution* s *such that LTL formula* $f_s(\varphi^?)$ *is consistent with* $\mathcal{S}$.

Proof sketch: For simplicity, we consider $\varphi^?$ to consist of only one Type-0 placeholder $?^0$. For the forward direction, we explicitly construct an LTL formula for $?^0$, based on tables $T^{\varphi^?}_{uv^\omega}$. Towards this, we first construct a sample $\mathcal{S}' = (P', N')$ as: $P' = \{uv^\omega[t,\infty) \in suf(\mathcal{S}) \mid T^{\varphi^?}_{uv^\omega}[?^0, t] = 1\}$, $N' = \{uv^\omega[t,\infty) \in suf(\mathcal{S}) \mid T^{\varphi^?}_{uv^\omega}[?^0, t] = 0\}$. Since the tables satisfy the Suffix property, $P' \cap N' = \emptyset$. We can now construct a generic LTL formula ψ consistent with $\mathcal{S}'$ using *LTL learning* [37]. For the other direction, we construct $T^{\varphi^?}_{uv^\omega}$ based on $T^{f_s(\varphi^?)}_{uv^\omega}$ as: $T^{\varphi^?}_{uv^\omega}[\varphi^?[i], \cdot] = T^{f_s(\varphi^?)}_{uv^\omega}[f_s(\varphi^?)[i], \cdot]$ for each $uv^\omega \in P \cup N$ and $0 \leq i < |\varphi^?|$. □

With this, we conclude the NP algorithm for the case where $\varphi^?$ only has Type-0 placeholders. We can easily extend the algorithm to the case where $\varphi^?$ consists of Type-1 and Type-2 placeholders. In particular, we first guess the operators to be substituted for the Type-1 and Type-2 placeholders and substitute them. We then obtain a sketch consisting of only Type-0 placeholders. We now apply our algorithm that relies on guessing tables, as described above.

Theorem 1. *The LTL sketch existence problem is in* NP.

We conjecture that the complexity lower-bound of *LTL sketch existence* is NP-hard based on the NP-hardness of *LTL learning* for certain fragments of LTL [22]. However, we have to leave the exact lower-bound of the problem for future work.

4.2 The Decision Procedure

Based on the NP algorithm described above, we now devise a decision procedure to decide the *LTL sketch existence* problem. The decision procedure relies upon reducing the existence of tables $T^{\varphi}_{uv^\omega}$ satisfying the three properties discussed in Sect. 4.1 to a satisfiability (SAT) problem.

This reduction relies on a symbolic encoding of the entries of the tables. To this end, we introduce propositional variables $y^{u,v}_{i,t}$ for each $i \in \{1, \ldots, n\}$, $t \in \{0, \ldots, |uv| - 1\}$, and $uv^\omega \in P \cup N$. A variable $y^{u,v}_{i,t}$ encodes the entry $T^{\varphi}_{uv^\omega}[\varphi[i], t]$. Further, we encode the operators to be substituted for the Type-1 and Type-2 placeholders in $\varphi^?$ using the following variables: (i) $x_{i,\lambda}$ for each Node i where $\ell(i)$ is a Type-1 placeholder and each $\lambda \in \Lambda_U$; and (ii) $x_{i,\lambda}$ for each Node i where $\ell(i)$ is a Type-2 placeholder and each $\lambda \in \Lambda_B$.

We now impose constraints on the introduced variables to ensure that the prospective tables satisfy the three properties necessary for inferring a consistent LTL formula. We achieve this by constructing a propositional formula $\Phi^{\varphi^?, \mathcal{S}}$. This formula ensures that variables $y^{u,v}_{i,t}$ encode appropriate tables and using Lemma 1, its satisfiability ensures the existence of a suitable substitution for $\varphi^?$.

Internally, $\Phi^{\varphi^?, \mathcal{S}} := \Phi^{1,2}_? \wedge \Phi_{sem} \wedge \Phi_{con} \wedge \Phi_{suf}$ is a conjunction of four formulas. The first conjunct $\Phi^{1,2}_?$ ensures that the Type-1 and Type-2 placeholders are

substituted by appropriate operators. The conjuncts Φ_{sem}, Φ_{con} and Φ_{suf} ensure that the variables $y_{i,t}^{u,v}$ encode entries of tables that satisfy the Semantic property (e.g., Eqs. 1, 2, etc.), the Consistency property (Eq. 4) and the Suffix property (Eq. 5), respectively. In the remainder of the section, we describe the construction of each of the four formulas.

We begin by introducing the constraints required for $\Phi_?^{1,2}$. For each Node i labeled with a Type-1 placeholder (i.e., $\ell(i) \in \Pi^1$), we design the following constraint:

$$\Big[\bigvee_{\lambda \in \Lambda_U} x_{i,\lambda}\Big] \wedge \Big[\bigwedge_{\lambda \neq \lambda' \in \Lambda_U} \neg x_{i,\lambda} \vee \neg x_{i,\lambda'}\Big], \tag{6}$$

which ensures that the Type-1 placeholders are substituted with a unique unary operator. For Type-2 placeholders, we have the exact same constraint except that the operators range from the set of binary operators Λ_B. We now construct $\Phi_?^{1,2}$ simply by taking a conjunction of all such constraints for the nodes labeled with Type-1 and Type-2 placeholders.

Next, we define Φ_{sem} as the conjunction $\bigwedge_{uv^\omega \in P \cup N} \Phi^{u,v}$, where $\Phi^{u,v}$ denotes a formula that ensures that the variables $y_{i,t}^{u,v}$ satisfy the semantic relations for the word uv^ω. In the formula $\Phi^{u,v}$, for each Node i labeled with the X-operator (i.e., $\ell(i) = \mathsf{X}$) and having Node j as its left child, we have the following constraint:

$$\Big[\bigwedge_{0 \leq t < |uv|-1} \Big[y_{i,t}^{u,v} \leftrightarrow y_{j,t+1}^{u,v}\Big]\Big] \wedge \Big[y_{i,|uv|-1}^{u,v} \leftrightarrow y_{j,|u|}^{u,v}\Big] \tag{7}$$

This constraint ensures that the variables $y_{i,t}^{u,v}$ satisfy Eq. 2 for the word uv^ω. For nodes labeled with other operators, we construct similar constraints based on their corresponding semantic relations. If the nodes are labeled with Type-1 or Type-2 placeholders, we additionally rely on variables $x_{i,\lambda}$ to determine the operator λ to be substituted in Node i. Based on the operator label λ, we devise appropriate semantic constraints. Finally, we construct $\Phi^{u,v}$ as the conjunction of all such semantic constraints.

We construct the following constraint to ensure Eq. 4 is satisfied for the prospective tables:

$$\Phi_{con} \coloneqq [\bigwedge_{uv^\omega \in P} y_{1,0}^{u,v}] \wedge [\bigwedge_{uv^\omega \in N} \neg y_{1,0}^{u,v}] \tag{8}$$

Finally, for Φ_{suf}, we have the following constraint for each Node i labeled with a Type-0 placeholder (i.e., $\ell(i) \in \Pi^0$):

$$\bigwedge_{u_1 v_1^\omega[t,\infty) = u_2 v_2^\omega[t',\infty) \in suf(\mathcal{S})} \Big[y_{i,t}^{u_1,v_1} \leftrightarrow y_{i,t'}^{u_2,v_2}\Big], \tag{9}$$

which ensures that Eq. 5 is satisfied for the prospective tables.

Overall, we construct a formula $\Phi^{\varphi^?,\mathcal{S}}$ that ranges over $\mathcal{O}(n + nm)$ variables and is of size $\mathcal{O}(n + nm^3 + m^2)$, where $n = |\varphi^?|$ and $m = |\mathcal{S}|$. We conclude this section by stating the correctness of $\Phi^{\varphi^?,\mathcal{S}}$.

Theorem 2. *Let $\varphi^?$ be a sketch, $\mathcal{S}$ a sample, and $\Phi^{\varphi^?,\mathcal{S}}$ the formula as defined above. Then, $\Phi^{\varphi^?,\mathcal{S}}$ is satisfiable if and only if there exists a complete substitution s such that $f_s(\varphi^?)$ is consistent with $\mathcal{S}$.*

Proof sketch: For the forward direction, based on a model v of $\Phi^{\varphi^?,\mathcal{S}}$, we construct a complete substitution s such that $f_s(\varphi^?)$ is consistent with $\mathcal{S}$. First, due to constraints like Constraint 6, we can substitute any Type-1 or Type-2 placeholder, say at Node i, with the unique operator λ for which $v(x_{i,\lambda}) = 1$. Second, we construct substitutions for Type-0 placeholders by relying on tables $T^{\varphi^?}_{uv^\omega}$ that we construct from v as follows: $T^{\varphi^?}_{uv^\omega}[\varphi^?[i], uv^w[t,\infty)] = v(y^{u,v}_{i,t})$ for each $uv^\omega \in P \cup N$ and $i \in \{1,\ldots,n\}$. Due to Constraints 7, 8, and 9, the constructed tables $T^{\varphi^?}_{uv^\omega}$ satisfy the Semantic, Consistency, and Suffix properties. As a result, one can explicitly construct substitutions for Type-0 placeholders based on tables $T^{\varphi^?}_{uv^\omega}$, exploiting Lemma 1. For the other direction, based on the substitution s, we simply construct a unique assignment v that satisfies $\Phi^{\varphi^?,\mathcal{S}}$. □

5 Algorithms to Complete an LTL Sketch

We now describe two novel algorithms for solving the *LTL sketching* problem, which aim at searching for concise LTL formulas from sketches, as alluded to in the introduction. Thus, our first algorithm relies on existing techniques to learn *minimal* LTL formulas. Our second algorithm, alternatively, searches for formulas of increasing size based on constraint solving.

5.1 Algorithm Based on LTL Learning

This algorithm, which we refer to as `Algo1`, builds upon the decision procedure for checking the existence of a complete substitution presented in Sect. 4.2. In particular, it relies on $\Phi^{\varphi^?,\mathcal{S}}$ from the decision procedure to construct substitutions for placeholders of a sketch. While it is straightforward to substitute Type-1 and Type-2 placeholders, the algorithm relies on the classic LTL learning problem to substitute Type-0 placeholders.

The first step of the algorithm is to construct $\Phi^{\varphi^?,\mathcal{S}}$ from the given sample and sketch, as described in Sect. 4.2. If $\Phi^{\varphi^?,\mathcal{S}}$ is unsatisfiable, the algorithm straightaway returns that no solution exists, as established by Theorem 2. If satisfiable, we use a model, say v, of $\Phi^{\varphi^?,\mathcal{S}}$ (obtained from any off-the-shelf SAT solver) to complete $\varphi^?$, the details of which we describe next.

Given a model v of $\Phi^{\varphi^?,\mathcal{S}}$, one can substitute the Type-1 and Type-2 placeholders in $\varphi^?$ as follows: for each Node i where $\ell(i)$ is a Type-1 and Type-2 placeholders, assign $s(\ell(i)) = \lambda$, where λ is the unique operator for which $v(x_{i,\lambda}) = 1$.

The Type-0 placeholders, however, are more challenging to substitute. This is because they represent entire LTL formulas. Towards substituting Type-0 placeholders, for every Node i for which $\ell(i)$ is a Type-0 placeholder (i.e., $\ell(i) \in \Pi^0$), we first construct a sample $\mathcal{S}_i = (P_i, N_i)$ as $P_i = \{uv^\omega[t,\infty) \in \mathit{suf}(\mathcal{S}) \mid v(y^{u,v}_{i,t}) = 1\}$, and $N_i = \{uv^\omega[t,\infty) \in \mathit{suf}(\mathcal{S}) \mid v(y^{u,v}_{i,t}) = 0\}$. We now learn a minimal

LTL formula φ_i consistent with the sample $\mathcal{S}_i$ (using some *LTL learning* algorithm [37,40,41]) for substituting $\ell(i)$. Intuitively, such formulas φ_i ensure that the tables $T^{\varphi}_{uv^\omega}$ of φ obtained by completing $\varphi^?$ satisfy the Semantic, Consistency and Suffix properties described in Sect. 4.1.

We now establish the correctness of the algorithm using the following theorem:

Theorem 3. *Given sketch $\varphi^?$ and sample $\mathcal{S}$, `Algo1` completes $\varphi^?$ to output an LTL formula that is consistent with $\mathcal{S}$ if such a formula exists, otherwise returns that no such formula exists.*

Observe that this algorithm constructs new samples for each Type-0 placeholder, each of which have size $\mathcal{O}(|suf(\mathcal{S})|) = \mathcal{O}(|\mathcal{S}|^2)$. This poses a challenge to the scalability of this algorithm. Furthermore, the new samples are not optimized to produce the minimal possible substitutions. Our next algorithm improves both the runtime and the size of the inferred specification.

5.2 Algorithm Based on Incremental SAT Solving

We now describe an algorithm, abbreviated as `Algo2`, that reduces *LTL sketching* to a series of SAT solving problems, inspired by the SAT-based algorithm of Neider and Gavran [37]. Given a sample $\mathcal{S}$ and a number $n \in \mathbb{N}\backslash\{0\}$, we construct a propositional formula $\Psi_n^{\varphi^?,\mathcal{S}}$, of size $\mathsf{poly}(|\varphi^?|, |\mathcal{S}|)$, that has the properties that: (i) $\Psi_n^{\varphi^?,\mathcal{S}}$ is satisfiable if and only if one can complete $\varphi^?$ to obtain an LTL formula of size at most n that is consistent with $\mathcal{S}$; and (ii) using a model v of $\Psi_n^{\varphi^?,\mathcal{S}}$, one can complete $\varphi^?$ to construct a consistent LTL formula of size at most n.

However, in contrast to the algorithms by Neider and Gavran, we first solve $\Phi^{\varphi^?,\mathcal{S}}$ (discussed in Sect. 4.2) to determine the existence of a complete substitution. If and only if $\Phi^{\varphi^?,\mathcal{S}}$ is satisfiable, our algorithm checks the satisfiability of $\Psi_n^{\varphi^?,\mathcal{S}}$ for increasing values of n (starting from $|\varphi^?| - 1$) to search for an LTL formula of size at most n that has the same syntactic structure as $\varphi^?$. We construct the resulting LTL formula by substituting the placeholders in $\varphi^?$ based on a model v of the formula $\Psi_n^{\varphi^?,\mathcal{S}}$, similar to what we do in `Algo1`. The termination of this algorithm is guaranteed by the decision procedure encoded by $\Phi^{\varphi^?,\mathcal{S}}$. The procedure ensures that we search for a solution only if there exists a complete and consistent LTL formula, to begin with. Moreover, the properties of $\Psi_n^{\varphi^?,S}$ ensure that we find the suitable LTL formula if there exists one.

On a technical level, the formula $\Psi_n^{\varphi^?,\mathcal{S}}$ is obtained by modifying certain parts of the formula $\Phi^{\varphi^?,\mathcal{S}}$. Precisely, $\Psi_n^{\varphi^?,\mathcal{S}} := \Phi_?^{1,2} \wedge \Phi'_{sem} \wedge \Phi_{con} \wedge \Phi^0_{?,n}$ and it introduces two modifications in $\Phi^{\varphi^?,\mathcal{S}}$: a new formula $\Phi^0_{?,n}$ replaces Φ_{suf}; and Φ'_{sem} adds more constraints to Φ_{sem}. The formula $\Phi^0_{?,n}$ encodes the structure of LTL formulas that substitute the Type-0 placeholders. Φ'_{sem}, again as in Φ_{sem}, ensures that the variables $y^{u,v}_{i,t}$ encode table entries $T^{\varphi}_{uv^\omega}[\varphi[i], t]$ that satisfy equations (e.g., Eqs. 1, 2, etc.) describing the Semantic property. We now briefly describe the constraints for the newly introduced formulas.

The formula $\Phi^0_{?,n}$ relies on an additional set of variables: (i) $x_{i,\lambda}$ for each Node i where $\ell(i)$ is a Type-0 placeholder or $i \in \{|\varphi^?|+1,\ldots,n\}$, and each $\lambda \in \Lambda$; and (ii) $l_{i,j}$ and $r_{i,j}$ for each Node i where $\ell(i)$ is a Type-0 placeholder or $i \in \{|\varphi^?|+1,\ldots,n\}$, and each $j \in \{\max(i,|\varphi^?|+1),\ldots,n\}$. The variable $x_{i,\lambda}$, again, encodes that Node i is labeled with λ. The variables $l_{i,j}$ (and $r_{i,j}$) encode that the left child (and the right child) of Node i is Node j. Together the new variables encode the structure of the prospective LTL formulas for Type-0 placeholders.

We now impose constraints, similar to Constraint 6, on the variables $x_{i,\lambda}$ to ensure each node is labeled by a unique LTL operator from Λ. Further, we impose constraints to ensure that each Node i has a unique left and right child. Finally, we construct $\Phi^0_{?,n}$ as the conjunction of all such structural constraints.

The formula Φ'_{sem} also relies on new variables $y^{u,v}_{i,t}$ for each Node i labeled with a Type-0 variables or $i \in \{|\varphi^?|+1,\ldots,n\}$, each $t \in \{0,\ldots,|uv|-1\}$ and each uv^ω in $\mathcal{S}$. Now, we construct semantic constraints such as:

$$\left[x_{i,\mathsf{X}} \wedge l_{i,j}\right] \rightarrow \bigwedge_{0 \le t < |uv|-1} \left[y^{u,v}_{i,t} \leftrightarrow y^{u,v}_{j,t+1}\right] \wedge \left[y^{u,v}_{i,|uv|-1} \leftrightarrow y^{u,v}_{j,|u|}\right], \tag{10}$$

that ensures that the $y^{u,v}_{i,t}$ variables encode entries of table that satisfy Eq. 2. We construct Φ'_{sem} as the conjunction of Φ_{sem} and the new semantic constraints.

We establish the correctness guarantees using the following theorem:

Theorem 4. *Given sketch $\varphi^?$ and sample $\mathcal{S}$, `Algo2` completes $\varphi^?$ to output an LTL formula that is consistent with $\mathcal{S}$ if such a formula exists, otherwise returns no such formula exists.*

`Algo2` searches for substitutions of Type-0 placeholders of increasing size and, thus, is able to find small substitutions for the sketch. However, it may not always find a minimal consistent LTL formula because a minimal formula may require the parts of the substitution to share subformulas from the existing sketch.

To demonstrate this, consider the sketch $\mathsf{F}(?_0) \vee \mathsf{F}\,\mathsf{G}\,p$ and the sample consisting of one positive word $\{\}\{p\}^\omega$ and one negative word $\{\}^\omega$. For this input, a possible output by `Algo2` is the formula $\mathsf{F}\,p \vee \mathsf{F}\,\mathsf{G}\,p$, which is of size 5. However, the minimal consistent formula $\mathsf{F}\,\mathsf{G}\,p \vee \mathsf{F}\,\mathsf{G}\,p$ is of size 4. In this example, substituting $?_0$ with $\mathsf{G}\,p$ produces a smaller formula than substituting it with p, since $\mathsf{G}\,p$ allows more sharing of subformulas.

While `Algo2` may not always return a minimal formula, we can provide an upper bound on its size, thus ensuring its conciseness. To compute this bound, we define the syntax size $|\varphi|_s$ of a formula φ to be the number of operators and propositions appearing in φ. Typically, the syntax size $|\varphi|_s$ is larger than the (DAG) size $|\varphi|$, since it counts all the operators and propositions, including the repeating ones. For instance, for $\varphi = \mathsf{F}\,\mathsf{G}\,q \vee \mathsf{F}\,\mathsf{G}\,q$, $|\varphi|_s = 7$, while $|\varphi| = 4$.

We now state the guarantee on the size of the formula returned by `Algo2` in the following theorem. Intuitively, the theorem states the size $|\varphi|$ of the formula

that `Algo2` returns is bounded by the syntax size $|\varphi^*|_s$ of the minimal (DAG size) solution φ^*.

Theorem 5. *Given sketch $\varphi^?$ and sample $\mathcal{S}$, let φ^* be a minimal formula that is consistent with $\mathcal{S}$ and can be obtained by completing $\varphi^?$. Then, `Algo2` returns a formula φ that is consistent with $\mathcal{S}$, can be obtained by completing $\varphi^?$ and has size $|\varphi| \leq |\varphi^*|_s$.*

Proof sketch: Towards contradiction, we assume that the LTL formula φ returned by `Algo2` has size $|\varphi| > |\varphi^*|_s$. Now, based on the property of $\Psi_n^{\varphi^?,\mathcal{S}}$ (see first paragraph of Sect. 5.2), $\Psi_n^{\varphi^?,\mathcal{S}}$ is satisfiable for $n = |\varphi^*|_s$. This is because there exists a consistent LTL formula of size at most $n = |\varphi^*|_s$, φ^* itself. Thus, `Algo2`, due to its incremental search, should have returned φ^*, contradicting our assumption. □

6 Experimental Evaluation

In this section, we design experiments to answer the following research questions:

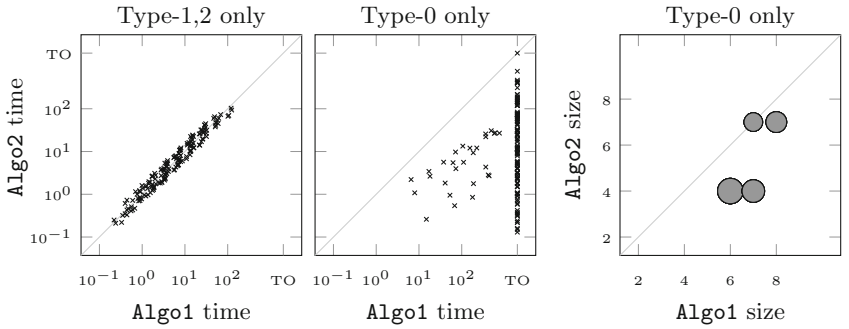
RQ1: Which of the two presented sketching algorithms is more effective?
RQ2: How do our algorithms compare against other specification mining tools for LTL?

To answer these questions, we have implemented a prototype of our algorithms in `Python3`, named `LTL-Sketcher`[1]. In `LTL-Sketcher`, we additionally implement two heuristics to improve the runtime of our algorithms, both of which are directed toward optimizing the SAT encoding used in the algorithms. We briefly mention the idea behind the heuristics.

The first heuristic is inspired by the SAT encoding used in Bounded Model Checking [8]. The encoding exploits a succinct description of the semantics of LTL using expansion laws [5]. Exemplarily, the expansion law for the U-operator is $\varphi \,\mathsf{U}\, \psi = \psi \vee (\varphi \wedge \mathsf{X}(\varphi \,\mathsf{U}\, \psi))$, which relies on checking satisfaction in the next position using X-operator. Using the LTL expansion laws reduces the number of variables required in Φ_{sem}. In the second heuristic, we create variables $y_{i,t}^{u,v}$ only for the distinct suffixes uv^ω in $\mathcal{S}$. This is sufficient because LTL formulas have the same evaluation on equal suffixes (which is also the basis for Eq. 5). Hence, if two words share a suffix, we can create the variables encoding the semantics only once, reducing the total number of variables. Also, in this heuristic, the constraint Φ_{suf} imposing the Suffix property becomes unnecessary. We refer interested readers to the extended version [35] for the details of the heuristics.

Benchmarks. For evaluating our algorithms, following the literature in *LTL learning*, we rely on benchmarks generated synthetically using common LTL formulas used in practice [19]. We choose the same nine formulas also chosen by

[1] The code can be found in https://github.com/rajarshi008/LTLSketcher.



(a) Runtime comparison (b) Size comparison

Fig. 5. Comparison of `Algo1` and `Algo2` with respect to runtime (in seconds) and the size of inferred formulas. The points below the diagonal are where `Algo2` performs better. In Fig. 5a, "TO" denotes timeouts. In Fig. 5b, the size of a bubble is proportional to the number of cases.

Neider and Gavran [37] for generating their benchmarks. We, however, deviate from their method of generating benchmarks. This is because, as observed by Raha et al. [40], their method, being fairly naive, consumes more time and often does not generate adequately different trajectories from a chosen LTL formula. We, in contrast, design a novel method of generating samples based on random sampling of words from Büchi automata [7] constructed from the LTL formulas (using `Spot` [18]). Overall, we generate 18 samples for each of the nine formulas (i.e., 162 samples in total), with the number of examples varying from 20 to 800 and the length of words varying from 4 to 16. We conduct all the experiments on a single core of an Intel Xeon E7-8857 CPU (at 3 GHz) using upto 6 GB of RAM.

RQ1: Comparison of Sketching Algorithms. To answer RQ1, we compare `Algo1` (from Sect. 5.1) and `Algo2` (from Sect. 5.2) based on their running times and the size of formula inferred. For this comparison, as sketches, we remove parts (upto 50% in size) of each formula to construct two kinds of sketches: one with only Type-1 or Type-2 placeholders and one with only Type-0 placeholders (see extended version [35] for the entire list). Exemplarily, for $\mathsf{G}(q \rightarrow \mathsf{G}(\neg p))$, we construct two sketches: $?^1_1(q \rightarrow ?^1_2(\neg p))$ and $\mathsf{G}(q \rightarrow ?^0)$; for $\mathsf{F}(q) \rightarrow (p \,\mathsf{U}\, q)$, we construct two sketches $?^0 \rightarrow (p \,\mathsf{U}\, q)$ and $?^1(q) \rightarrow (p ?^2 q)$, etc. We now run the algorithms on the 18 samples and two sketches generated from each of the nine formulas with a timeout of 900 secs.

We depict the runtime comparisons in Fig. 5a. We observe that while both the algorithms have comparable runtime on sketches with only Type-1 or Type-2 placeholders, `Algo1` performs significantly worse on sketches with only Type-0 placeholders with 134 timeouts. We also depict the comparison of formula size in Fig. 5b. We notice that `Algo2` returns smaller formulas than `Algo1` in many

cases. The reason `Algo1` performs slow and returns large formulas is that it solves *LTL learning* on potentially large intermediate samples for sketches with Type-0 placeholders. Thus, we answer RQ1 in favor of `Algo2`.

RQ2: Comparison Against LTL Mining Tools. To address RQ2, we compared `LTL-Sketcher` against two prominent approaches for mining specifications in LTL. The first approach completes user-defined templates with (Boolean combinations of) atomic propositions. For this approach, we select the popular LTL miner `Texada` [31]. The second approach learns LTL formulas of minimal size without syntactic constraints. For this approach, we choose `Flie` [37] as a prototypical example of this class of algorithms.

The setting of `Texada` differs from ours in that it permits positive examples only, and these examples have to be finite words. Thus, in order to have a fair comparison, we make minor modifications to our SAT encoding (specifically to the X-operator) to handle finite words. Furthermore, our tool does not require one to provide negative examples and, hence, can immediately be applied.

Table 1. Number of successes for completing sketches

Sketch	Tool	$\mathsf{F}(q) \rightarrow (\neg p \,\mathsf{U}\, q)$	$\mathsf{F}(q) \rightarrow (p \,\mathsf{U}\, q)$	$\mathsf{G}(q \rightarrow \mathsf{G}(\neg p))$
full	`LTL-Sketcher`	10	10	10
	`Texada`	6	9	10
medium	`LTL-Sketcher`	10	10	10
	`Texada`	0	1	10
small	`LTL-Sketcher`	10	10	10
	`Texada`	0	1	0

To compare `Texada` and `LTL-Sketcher`, we considered six of the nine formulas used in RQ1, dropping the smallest three. For each formula, we created ten samples with only positive, finite words by truncating ultimately periodic and ensuring consistency with the formula. Also, we created three sketches for each formula, retaining different amounts of information: in a *full sketch*, we only replaced each atomic proposition with a different Type-0 placeholder; in a *medium sketch*, we replaced a larger subformula containing at least one temporal operator; and in a *small sketch*, we replaced the formula with a single Type-0 placeholder. As an example, from formula $\mathsf{F}(q) \rightarrow (\neg p \,\mathsf{U}\, q)$, we constructed the full sketch $\mathsf{F}(?^0_1) \rightarrow (\neg ?^0_2 \,\mathsf{U}\, ?^0_3)$, the medium sketch $\mathsf{F}(?^0_1) \rightarrow ?^0_2$ and the small sketch $?^0_1$.

We ran `Texada` and `LTL-Sketcher` on each of these sketches and all corresponding samples and counted the cases in which the tools could provide a substitution. A selection of the results on three prototypical formulas is shown in Table 1. The remaining results follow the same trend and can be found in the extended version [35]. We notice that `Texada` found substitutions for the

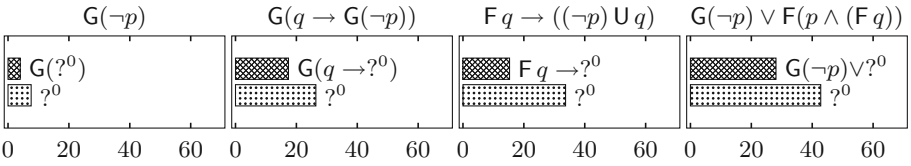


Fig. 6. The average sample sizes required to recover the original formula (mentioned in the chart titles). In each chart, the trivial sketch $?^0$ indicates the run for `Flie`, while the other one indicates the run for `LTL-Sketcher`.

full sketches in most cases. However, when we removed more structural information from the specifications (i.e., medium and small sketches), `Texada` was rarely able to complete a sketch. By contrast, `LTL-Sketcher` provided a substitution in every benchmark. The reason is that `Texada`'s strategy of exclusively searching for atomic propositions is only feasible if the user can provide a detailed template where all temporal operators are specified. Our tool, in contrast, alleviates the burden of writing complex temporal operators and, thus, is more flexible.

To compare `Flie` and `LTL-Sketcher`, we estimated how many examples are required to infer the desired specification. For this experiment, we used the same set of nine LTL formulas and sketches with varying amounts of missing information, some of which can be seen in Fig. 6. To calculate the number of examples required, we designed a counterexample-guided strategy to compute a *minimal* sample required for both tools to obtain the desired formula from a sketch of it. In this strategy, if a tool does not return the desired formula with the current sample, we add one of the shortest counterexamples to the sample that helps eliminate the current solution formula. We continue this process and end up with a minimal sample of both tools for each sketch.

Figure 6 presents the average size of minimal samples (over ten runs) required to recover the desired formulas from their sketches. While we present the result for some formulas here, the remaining results follow the same pattern (see extended version [35]). We observed that `Flie` required more examples than `LTL-Sketcher` to single out the correct specifications in all the cases. This asserted the fact that providing high-level insights as a sketch reduces the number of examples required to derive the desired specification. Thus, to answer RQ2, the ability to handle sketches provides `LTL-Sketcher` an edge over existing LTL mining tools.

7 Conclusion and Future Work

In this work, we introduce LTL sketching—a novel way of writing formal specifications in LTL. The key idea is that a user can write a partial specification, i.e., a sketch, which is then completed based on given examples of desired and undesired system behavior. We have shown that the sketching problem is in NP, presented two SAT-based sketching algorithms and some heuristics to improve their performance. Our experimental evaluation has shown that our algorithms

can effectively complete sketches consisting of different types of missing information. Further, the ability to handle sketches provides our algorithms an edge over existing LTL mining approaches.

A natural direction for future work is to lift the idea of specification sketching to other specification languages, such as Signal Temporal Logic (STL) [36], the Property Specification Language (PSL) [20], or even visual specifications, such as UML (high-level) message sequence charts [26]. Moreover, we intend to extend the notion of sketching beyond the use of examples (e.g., by allowing the engineer to constrain placeholders using simple logical formulas or regular expressions).

Acknowledgements. This work has been financially supported by Deutsche Forschungsgemeinschaft, DFG Project numbers 434592664 and 459419731, and the Research Center Trustworthy Data Science and Security (https://rc-trust.ai), one of the Research Alliance centers within the UA Ruhr (https://uaruhr.de).

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, 16–18 January 2002, pp. 4–16. ACM (2002). https://doi.org/10.1145/503272.503275
2. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: syntax-guided synthesis of PLTL formulas from finite traces. In: FMCAD, pp. 93–103. IEEE (2020)
3. Bacherini, S., Fantechi, A., Tempestini, M., Zingoni, N.: A story about formal methods adoption by a railway signaling manufacturer. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 179–189. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_13
4. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: roissy VAL. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005). https://doi.org/10.1007/11415787_20
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT competition 2016: recent developments. In: 31st AAAI Conference on Artificial Intelligence, AAAI '17, pp. 5061–5063. AAAI Press (2017)
7. Bernardi, O., Giménez, O.: A linear algorithm for the random sampling from regular languages. Algorithmica **62**(1–2), 130–145 (2012)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers, vol. 58, pp. 117–148. Elsevier (2003). https://doi.org/10.1016/S0065-2458(03)58003-2, https://www.sciencedirect.com/science/article/pii/S0065245803580032
9. Bowen, J.P.: Gerard o'regan: concise guide to formal methods: theory, fundamentals and industry applications. Form. Aspects Comput. **32**(1), 147–148 (2020)
10. Bruns, G., Godefroid, P.: Temporal logic query checking. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, 16–19

June 2001, Proceedings, pp. 409–417. IEEE Computer Society (2001). https://doi.org/10.1109/LICS.2001.932516
11. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational ω-languages. In: Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 554–566. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58027-1_27
12. Camacho, A., Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: LTL and beyond: formal languages for reward function specification in reinforcement learning. In: Kraus, S. (ed.) Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, 10–16 August 2019, pp. 6065–6073. ijcai.org (2019). https://doi.org/10.24963/ijcai.2019/840
13. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. In: ICAPS, pp. 621–630. AAAI Press (2019)
14. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_34
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
16. Cofer, D., Miller, S.: DO-333 certification case studies. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 1–15. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_1
17. Courtois, P.J., Seidel, F., Gallardo, F., Bowell, M.: Licensing of safety critical software for nuclear reactors. Common position of international nuclear regulators and authorised technical support organisations, December 2015. https://doi.org/10.13140/RG.2.1.2789.8968
18. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP, pp. 7–15. ACM (1998)
20. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Series on Integrated Circuits and Systems, Springer, New York (2006). https://doi.org/10.1007/978-0-387-36123-9
21. Fecko, M.A., et al.: A success story of formal description techniques: Estelle specification and test generation for MIL-STD 188–220. Comput. Commun. **23**(12), 1196–1213 (2000)
22. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: ICGI. Proceedings of Machine Learning Research, vol. 153, pp. 237–250. PMLR (2021)
23. Fix, L.: Fifteen years of formal property verification in intel. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 139–144. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_8
24. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: automated air traffic control design space exploration. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_1

25. Greenman, B., Saarinen, S., Nelson, T., Krishnamurthi, S.: Little tricky logic: misconceptions in the understanding of LTL. Art Sci. Eng. Program. **7**(2), 7:1–7:37 (2023)
26. Harel, D., Thiagarajan, P.S.: Message sequence charts. In: UML for Real - Design of Embedded Real-Time Systems, pp. 77–105. Kluwer (2003). https://doi.org/10.1007/0-306-48738-1_4
27. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
28. Holzmann, G.J.: The logic of bugs. In: SIGSOFT FSE, pp. 81–87. ACM (2002)
29. Kim, J., Muise, C., Shah, A., Agarwal, S., Shah, J.: Bayesian inference of linear temporal logic specifications for contrastive explanations. In: IJCAI, pp. 5591–5598. ijcai.org (2019)
30. Klein, G., et al.: SeL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010)
31. Lemieux, C., Beschastnikh, I.: Investigating program behavior using the texada LTL specifications miner. In: ASE, pp. 870–875. IEEE Computer Society (2015)
32. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining (T). In: ASE, pp. 81–92. IEEE Computer Society (2015)
33. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: MEMOCODE, pp. 43–50. IEEE (2011)
34. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using FDR. Softw. Concepts Tools **17**(3), 93–102 (1996)
35. Lutz, S., Neider, D., Roy, R.: Specification sketching for linear temporal logic. arXiv preprint arXiv:2206.06722 (2022)
36. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
37. Neider, D., Gavran, I.: Learning linear temporal properties. In: FMCAD, pp. 1–10. IEEE (2018)
38. Pakonen, A., Pang, C., Buzhinsky, I., Vyatkin, V.: User-friendly formal specification languages - conclusions drawn from industrial experience on model checking. In: ETFA, pp. 1–8. IEEE (2016)
39. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE Computer Society (1977)
40. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable anytime algorithms for learning fragments of linear temporal logic. In: TACAS 2022. LNCS, vol. 13243, pp. 263–280. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_14
41. Riener, H.: Exact synthesis of LTL properties from traces. In: FDL, pp. 1–6. IEEE (2019)
42. Roy, R., Fisman, D., Neider, D.: Learning interpretable models in the property specification language. In: IJCAI, pp. 2213–2219. ijcai.org (2020)
43. Roy, R., Gaglione, J.R., Baharisangari, N., Neider, D., Xu, Z., Topcu, U.: Learning interpretable temporal properties from positive examples only. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, no. 5, pp. 6507–6515, June 2023. https://doi.org/10.1609/aaai.v37i5.25800, https://ojs.aaai.org/index.php/AAAI/article/view/25800
44. Rozier, K.Y.: Specification: the biggest bottleneck in formal methods and autonomy. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 8–26. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_2

45. Schlör, R., Josko, B., Werth, D.: Using a visual formalism for design verification in industrial environments. In: Margaria, T., Steffen, B., Rückert, R., Posegga, J. (eds.) Services and Visualization Towards User-Friendly Design. LNCS, vol. 1385, pp. 208–221. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053507
46. Shah, A., Kamath, P., Shah, J.A., Li, S.: Bayesian inference of temporal task specifications from demonstrations. In: NeurIPS, pp. 3808–3817 (2018)
47. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5–6), 475–495 (2013)
48. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294. ACM (2005)
49. Vardi, M.Y.: Branching vs. linear time: final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_1
50. Verhulst, E., de Jong, G.: OpenComRTOS: an ultra-small network centric embedded RTOS designed using formal modeling. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 258–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74984-4_16
51. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. **18**(3–4), 263–292 (2011)