# General LTL Specification Mining

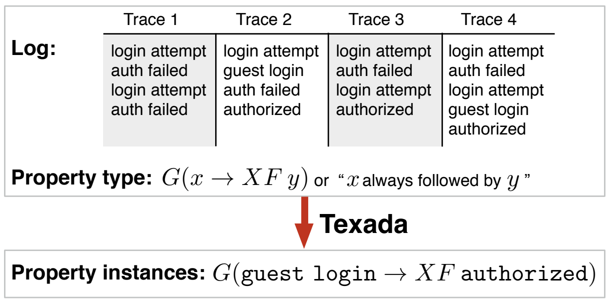
Caroline Lemieux  Dennis Park  Ivan Beschastnikh
Department of Computer Science
University of British Columbia
caro.lemieux@gmail.com, emaildpark@gmail.com, bestchai@cs.ubc.ca

***Abstract*—Temporal properties are useful for describing and reasoning about software behavior, but developers rarely write down temporal specifications of their systems. Prior work on inferring specifications developed tools to extract likely program specifications that fit particular kinds of tool-specific templates. This paper introduces Texada, a new temporal specification mining tool for extracting specifications in linear temporal logic (LTL) of arbitrary length and complexity. Texada takes a user-defined LTL property type template and a log of traces as input and outputs a set of instantiations of the property type (i.e., LTL formulas) that are true on the traces in the log. Texada also supports mining of almost invariants: properties with imperfect confidence. We formally describe Texada's algorithms and evaluate the tool's performance and utility.**

## I. Introduction

Specifying and reasoning about the temporal behavior of programs has been extensively studied [15], [16], [26]. One relatively recent idea is to mine a set of traces generated by a program to derive likely temporal specifications of the program, an API, or some other aspect of the software [1], [11], [19], [38], [46], [47]. For example, consider Figure 1, which lists a set of traces modeled after the /var/log/secure.log file in OSX. These traces satisfy the temporal property "guest login is always followed by authorized", which indicates that either the guest account has no password, or guest logins always succeed. Mined specifications cannot replace manually written specifications created by an expert since a property mined from program traces could be a false positive[1]. Nevertheless, as many programs lack formal specifications, mined specifications are valuable and have been shown to be useful for a wide assortment of tasks, such as testing [10], malware detection [9], data structure repair [12], supporting program evolution [11], [17], and debugging [21], [36], [37], [46].

We present Texada, a tool that implements algorithms to mine linear temporal logic (LTL) [16] properties of arbitrary length and complexity. Note the property "`guestlogin` is always followed by `authorized`" mined from the log in Figure 1 can be expressed in LTL as G(`guest login` → XF `authorized`). Unlike prior specification mining tools that extract a specific set of pre-defined templates [19]–[21], [46], [47], Texada (1) mines specifications described by an arbitrary, user-defined, LTL template, and (2) can mine specifications that satisfy user-defined support and confidence thresholds. While it is unclear how to adapt prior miners to extract new patterns, such as the "1 cause-2 effect precedence chain between two events" ($p$ precedes $s$,$t$ between $q$ and $r$) [15], Texada handles this pattern and all other LTL-based patterns by design. We show that Texada's property types are sufficient to


Log:
Property type: $G(x \rightarrow XF\, y)$ or "$x$ always followed by $y$"
Texada
Property instances: $G($guest login $\rightarrow XF$ authorized$)$

| Trace 1 | Trace 2 | Trace 3 | Trace 4 |
|---|---|---|---|
| login attempt | login attempt | login attempt | login attempt |
| auth failed | guest login | auth failed | auth failed |
| login attempt | auth failed | login attempt | login attempt |
| auth failed | authorized | authorized | guest login |
| | | | authorized |

Fig. 1: (Top) Example inputs to Texada, including an authentication log and a property type. (Bottom) Texada's mined property instances output: a set of LTL formulas based on the property type input template that evaluates to true on each trace in the input log.

capture both existing temporal specification taxonomies, like in Perracotta [47] and Dwyer et al. [15], as well as specification templates used in custom-built miners like Synoptic [7].

Texada can also mine *almost invariants*, or temporal properties that are falsified at some, but not all, locations in the input traces. Texada provides two controls for this: the *confidence* threshold allows a user to control the degree to which a mined property can be violated and the *support* threshold allows a user to control the minimum number of times that a property was validated. For example, if "$a$ always followed by $b$" is mined from a log, the user may interpret this property differently depending on its support (i.e. the number of $a$ events in the log): the more $a$ events there are, the less likely that "$a$ always followed by $b$" is true by chance. More concretely, the property G(`guest login` → XF `authorized`) in Figure 1 has confidence of 1 on each trace in the figure, but a support of 0 on traces 1 and 3, and a support of 1 on traces 2 and 4. As another example, the property F `authorized` (eventually `authorize` appears in the trace), has confidence of 0.75 over the entire log. Texada allows users to specify the support and confidence thresholds that all mined properties must satisfy. Mining of almost invariants is supported by other specification miners [11], [18], [31], [47]. Our contribution is the generalization of the calculation of these statistics to arbitrary LTL properties.

Texada takes two inputs (Figure 1): (1) *log*: a text file containing multiple traces, where each trace is a totally ordered sequence of string events, and (2) *property type*: an LTL formula whose atomic propositions are variables. Texada's output is a set of LTL formulae, or *property instances*, that are instantiations of the input property type. In a property instance, the variable atomic propositions in the property type are replaced with events from the log. Texada guarantees that for the given support and confidence thresholds (1) every returned property instance is valid on each of the input traces in the input log and (2) it returns all valid property instances of the input property type.

[1]The property "guest login is always followed by authorized" would be a false positive if a guest login in which the guest mistyped her password (and failed to authenticate) is missing from the log.

Our work makes the following contributions:

- ⋆ We present two algorithms, a linear miner and a map miner, for mining arbitrary LTL specifications from logs of program behavior. We also present an algorithm to compute support, support potential, and confidence statistics of arbitrary LTL properties. These algorithms are implemented in the Texada tool, which is released as open source [40] and includes ready-to-use property types from [7], [15], [47].
- ⋆ We evaluate Texada's performance and show that it outperforms the specialized temporal property miner in Synoptic [7]. Texada runs in seconds on logs with hundreds of execution traces with each trace containing thousands of events.
- ⋆ We evaluate Texada's utility by using it to (1) mine patterns from a log of website activity studied by two previous tools [23], [35]. We find that Texada-generated property instances are concise and help to focus attention on the relevant properties with particular structure. (2) We use Texada to confirm expected properties of a solution to the Sleeping Barber problem [13]. And, (3) we combine Texada with Daikon [17] to infer likely *data-temporal* properties.

## II. Formal background

We begin by formally describing what we mean by a log, a trace, and an event.

**Definition 1** (Log, trace, and event)**.** The alphabet of *events* is a finite alphabet of strings. An ordered sequence of events is a *trace*, and a set of traces is a *log*.

We use the term *event variable* to denote a place holder for an event. In this paper we use the beginning of the alphabet to denote events (e.g., $a, b, c$) and the end of the alphabet to denote event variables (e.g., $x, y, z$).

We use Linear Temporal Logic [16] to compactly specify specification templates, which we call *property types*. LTL statements assert certain conditions over time and use the operators *until* (U), *next* (X), *eventually* (F), *always* (G), *weak until* (W), *release* (R), and *strong release* (M). In this paper LTL will refer to a restricted version of Propositional Temporal Logic (PLTL), in which the non-temporal parts of formulae are built from atomic propositions and the usual logic operators $\wedge$, $\vee$, and $\neg$. We define LTL formulae in a manner similar to Emerson [16].

**Definition 2** (LTL formula)**.** An atomic proposition, $p$, is an LTL formula. If $p$ and $q$ are LTL formulae, so are $p \wedge q$, $\neg p$, $p \text{ U } q$, and $\text{X}\, p$.

We can then define all other logic operators, like $\vee$ and $\rightarrow$. Any non-temporal formula is true on a trace if it is true on the first event of the trace. The other temporal operators follow:

- Until: $p \text{ U } q$, holds if there is an event in the trace where $q$ holds and if $p$ is true on all events up to the first event where $q$ holds.
- Next: $\text{X}\, p$, holds if $p$ is true on the event immediately following the trace's first event.
- Finally/Eventually: $\text{F}\, p \equiv \textit{true} \text{ U } p$, holds if there exists a future event in the trace where $p$ is true.
- Globally/Always: $\text{G}\, p \equiv \neg\text{F}(\neg p)$, holds on a trace if $p$ holds on all events in the trace.

The operators W, R, and M are variations of until and are defined as in [2]. Standard LTL semantics assume an infinite sequence of events. Since Texada mines specifications from finite traces, we redefine the LTL semantics to fit our context (see end of this section).

A property type captures the temporal relationships between a set of event variables and (optionally) a set of events.

**Definition 3** (Property type)**.** A *property type* is an LTL formula in which all of the atomic propositions are either event variables or events.

For example, the property type $\text{G}(x \rightarrow \text{XF}\, y)$ represents the "$x$ is always followed by $y$" relationship between $x$ and $y$. Referring to events in a property type allows us to, amongst other things, establish scope. For example, if we are interested in expressing "$x$ is always followed by $y$" between open_file (denoted of) and close_file (denoted cf), then we can use these events in the property type to create a scope (as in [15]): $\text{G}((\texttt{of} \wedge \neg\texttt{cf} \wedge \text{F}\,\texttt{cf}) \rightarrow (x \rightarrow (\neg\texttt{cf} \text{ U } (y \wedge \neg\texttt{cf})))\text{ U } \texttt{cf}$

A property instance corresponds to a property type and has identical LTL structure, but with each event variable in the property type replaced by an event. Following Beschastnikh et al. [5], we call the map which associates each variable with an event a *binding*. For example, a property instance of the "$x$ is always followed by $y$" property type in a program trace may be "open_file is always followed by close_file", i.e., $\text{G}(\texttt{open\_file} \rightarrow \text{XF}\,\texttt{close\_file})$.

**Definition 4** (Property instance)**.** Let $\Pi$ be a property type. Then, $\pi$ is a *property instance* of $\Pi$ (or an *instantiation* of $\Pi$) if $\pi$ is an LTL formula that is identical to $\Pi$ in structure and in which all of the atomic propositions are events.

**Definition 5** (Binding)**.** Let $E$ be an alphabet of events and let $V$ be a finite set of event variables. Then, a *binding*[2] is a function $b\colon V \rightarrow E$.

Applying a binding to the variables in a property type creates a property instance corresponding to that binding: applying $\{x \rightarrow \texttt{open\_file}, y \rightarrow \texttt{close\_file}\}$ to $\text{G}(x \rightarrow \text{XF}\, y)$ creates the property instance $\text{G}(\texttt{open\_file} \rightarrow \text{XF}\,\texttt{close\_file})$. We use "property" in place of "property instance" and "property type" when the context makes it clear which one we mean.

We say that a binding and its corresponding property instance are *valid* on a log if the property instance holds on each trace in the log. Generally, we are interested in mining all of the valid property instances. But, we may also be interested in those instances that are most likely (e.g., because a log is incomplete or contains anomalous or buggy executions). To help with these cases we generalize the notion of validity.

**Definition 6** (Trace support potential)**.** The support potential of a property instance $\pi$ on a trace $t$ is the number of time points of $t$ which *could* falsify $\pi$.

**Definition 7** (Trace support)**.** The support of a property instance $\pi$ on a trace $t$ is the number of time points of $t$ which could falsify $\pi$, but *do not* falsify $\pi$.

To explain these further, consider checking $\text{G}a$ on some trace. One way to do this is to iterate along the trace and check whether the event at each position is $a$. If at any time point $\neg a$ is found to be true, $\text{G}a$ is falsified. Therefore, for $\text{G}a$,

[2]A binding may bind multiple variables in $V$ to the same event. This may produce trivial results: for example, $\text{G}(x \rightarrow \text{F}\, y)$, a common formulation of always followed by, is true any time $x$ and $y$ bind to the same event. This is why in this paper we use $\text{XF}\, y$ instead of $\text{F}\, y$, i.e., $\text{G}(x \rightarrow \text{XF}\, y)$.

```
1  function texada(property type string prop_s, log L, thresholds T)
2  // Returns: valid property instances of prop_s on L.
3  let valid_instances = []
4  prop = parse−property−type(prop_s)
5  traces = parse−log(L)
6  for binding in gen−binding(prop.vars, traces.events)
7    prop_instance = instantiate(prop, binding)
8    valid = check(prop_instance, traces, T)
9    if (valid) valid_instances.add(prop_instance)
10 return valid_instances
```

Fig. 2: A high-level description of Texada. The two versions of the check function (line 8) are listed in Figures 3 (linear miner) and Figure 6 (map miner).

the evaluation of $a$ at each position in the trace is potentially falsifiable. So the support potential for this property instance is the length of the trace, and its support is the number of positions in the trace at which $a$ appears. In general, the number of falsifiable time points depends on both the structure of the property instance and the sequence of events in the trace. We elaborate further on falsifiable time points in Section IV-A.

We use *trace confidence* to denote the ratio of trace support to trace support potential (if both are 0, confidence is 1). A confidence of 1 means that the instance is not falsified at any time point. We say that a property instance is *vacuously true* if its support and support potential are both 0. The *global support* and *global support potential* of a property instance is the sum of its support/support potential over all traces in the log; *global confidence* is the ratio of global support to global support potential. We use the terms support, support potential, and confidence when it is clear from the context if we mean the trace-based or global versions.

We can now formulate Texada's objective. Texada takes as inputs a log, a property type, and support, support potential, and confidence thresholds[3]. Texada outputs a set of property instances such that each instance has support/support potential/-confidence greater than or equal to the corresponding threshold.

**Finite trace semantics.** To check a property on a finite trace we effectively transform the finite trace into an infinite trace by appending an infinite sequence of terminal events $\omega$. This extension defines exactly the infinite trace we are checking on, so more advanced adaptations of LTL to finite traces are not necessary [3]. For many property types, this extension does not affect the validity of the property instances. For example, consider checking "$a$ is immediately followed by $b$," $\mathrm{G}(a \rightarrow \mathrm{X}\, b)$, on a finite trace which ends in $a$. Adding $\omega$s to the end of this trace does not change the invalidity of this property instance on that trace: $\omega$ is $\neg b$ and $\omega$ immediately follows $a$ so the property remains invalid. However, Texada evaluates $\mathrm{G}\, a$ to true on a finite trace consisting solely of $a$ events. This is despite the fact that on the extended trace, i.e., $aa\ldots a\omega\omega\ldots$, $\mathrm{G}\, a$ does not hold. It this case Texada checks $a\ \mathrm{U}\ \omega$ instead of $\mathrm{G}\, a$. Effectively, Texada determines the validity of the property instances *before* $\omega$. Instead of requiring the user to incorporate this additional $\omega$ bound into the property type, Texada implicitly evaluates the property type as if it contained the $\omega$ bound.

[3]When these thresholds are omitted, Texada uses the default confidence threshold of 1, and support and support potential thresholds of 0. The user may specify the trace or global versions of each of these three thresholds.

## III. Texada design overview

Figure 2 lists the pseudocode for Texada's algorithm. We overview the design behind this code and then explain Texada's two property checking algorithms in Sections IV and V.

**(1) Representing a property type.** Texada parses an input property type into a tree structure whose leaves are atomic propositions (line 4 of Figure 2). Figure 4(c) shows an example of such a tree. Texada traverses this tree when checking the validity of a property instance.

This style of checking contrasts with other pattern-mining tools that use automata to represent temporal properties (e.g., [7], [35], [39]). Critically, we use the tree representation to memoize and re-use evaluation results (see Section VI). Since the sub-trees of two different property instances may be identical, the memoization significantly speeds up the checking of many (potentially thousands of) similar property instances.

**(2) Representing a trace.** Texada parses each input trace into one of two representations (line 5 of Figure 2): (1) a linear array form, or (2) a map form in which each event is mapped to a sorted list of trace locations where the event appears. For example, the linear form of the trace $a, a, b, b$ is $[a, a, b, b, \omega]$. Its map form is $\{a \rightarrow [0, 1], b \rightarrow [2, 3], \omega \rightarrow [4]\}$. The map form allows the property checking algorithm to avoid traversing sections of the trace that do not contain relevant events.

**(3) Representing the space of property instances.** In line 6 of Figure 2 Texada iterates over the set of property instances (represented by bindings of variables $V$ to events $E$). Texada considers all possible bindings[4], or the space of all $|V|$-element permutations of $E$. Texada generates instantiations dynamically, which bounds space usage.

**(4) Checking property instances over traces.** Finally, Texada checks the validity of each property instance on all traces in the log (line 8, Figure 2) before moving on to the next instance. The check function on line 8 has two variants, one for each of the two trace representations (linear/map).

## IV. Linear miner

A natural way to evaluate a property instance on a linear trace is to iterate over the trace and recursively evaluate each operator according to its semantics (Section II). The core section of this linear checking algorithm is listed in Figure 3.

We describe the algorithm by checking a property instance on trace 2 in Figure 1. The linear trace representation is listed in Figure 4(a) and the property instance we will check is:

$$(\neg\texttt{authorized}\ \mathrm{W}\ \texttt{guest login}) \land \\ \mathrm{G}(\texttt{guest login} \rightarrow \mathrm{XF}\, \texttt{authorized})$$

This is the "`guest login` always followed by `authorized`" property with the added restriction that `authorized` cannot occur before `guest login` (the CauseFirst pattern in Table I). Internally, Texada represents this property as a tree (Figure 4(c)) and checks this property instance by traversing this tree. The root of the tree is $\land$; to check it, we check its children on the first event of the trace (see Figure 3, lines 7–8).

**Left sub-tree of $\land$:** To check $\neg$`authorized` W `guest login`, we follow the checking code for U in Figure 3, lines 25–34 (but as W is a weak until, we will return true on line 29). We begin

[4]Texada explores the space of permutations generated without replacement by default, and has a flag to consider permutations with replacement.

```
1   function check−linear(property instance $prop$, trace $t$)
2   // Returns: true if $prop$ holds on $t$, false otherwise.
3
4     // Operations that do not require trace traversal:
5     if ($prop$ is event)
6       return ($t$.first˙event() == $prop$)
7     else if ($prop$ == $p \wedge q$)
8       return check−linear($p$, $t$) && check−linear($q$, $t$)
9     else if ($prop$ == $p \vee q$)
10      return check−linear($p$, $t$) || check−linear($q$, $t$)
11    else if ($prop$ == $\neg p$)
12      return !check−linear($p$, $t$)
13
14    // Operations that require trace traversal:
15    else if ($prop$ == X $p$)
16      if ($t$.first˙event().is˙terminal())
17        // this creates the infinite sequence of terminal events.
18        return check−linear($p$, $t$)
19      return check−linear($p$, $t$.next˙event())
20
21    else if ($prop$ == G $p$)
22      if ($t$.first˙event().is˙terminal()) return true
23      return check−linear($p$, $t$) && check−linear($prop$, $t$
              .next˙event())
24
25    else if ($prop$ == $p$ U $q$)
26      if ($t$.first˙event().is˙terminal())
27        // If we get here, we have not seen q:
28        // need q to occur for until to hold.
29        return false
30      else if ($q$ holds on $t$) return true
31      else if ($\neg p$ holds on $t$) return false
32      // Here, p and !q hold, so we need to look
33      // for a !p or q further down the trace.
34      return check−linear($prop$, $t$.next˙event())
```

Fig. 3: A section of the linear checking algorithm.

traversing the linear representation of the trace in Figure 4(a) at the first event, which is `login attempt`. Since this does not evaluate to true on `guest login` nor on $\neg$(¬`authorized`), we check (¬`authorized` W `guest login`) on the next event. Here we return true because `guest login` has been reached without violating ¬`authorized`.

**Right sub-tree of $\wedge$:** In general, to check G $p$ we must check $p$ on each event in the trace (Figure 3, lines 21–23), recursing until $\neg p$ is found or the trace ends. The implication operator $p \rightarrow q$ works like $\neg p \vee q$, short-circuiting if $\neg p$ is found. So, to check G(`guest login` $\rightarrow$ XF `authorized`)), we recurse down the trace. When we reach `guest login`, we need to check the implication, XF `authorized`. At this point the X operator takes a step along the trace. Then, checking F `authorized` involves stepping down the trace once more until we find `authorized`. We find it at position 4 and return that XF `authorized` holds. Continuing, we find the G sub-property is not violated, and the full property holds on this trace (it does not hold on the log in Figure 1 as it is violated in trace 3).

Boolean operators omitted from Figure 3, such as the $\rightarrow$, $\leftrightarrow$, and $\oplus$ are implemented by combining the results of checking $p$ and $q$ with the corresponding operator, as with the other boolean operators. Evaluation can short-circuit on $\wedge$, $\vee$, and $\rightarrow$ operators. The W operator evaluation is identical to U except that it evaluates to true in the base case.

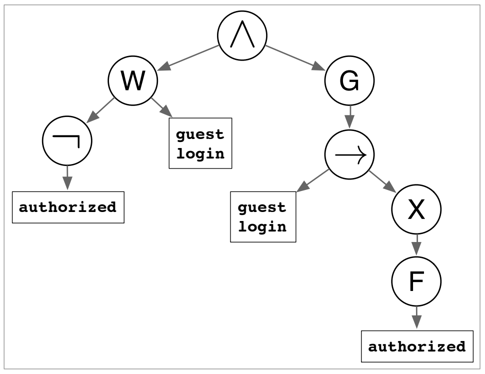
In both the linear miner and the map miner (Section V), we take special care to check property types at the end of the trace to satisfy the finite trace semantics (see Section II). Most operators have a simple boolean base case for the terminal event $\omega$. The next operator, X $p$, requires a more extensive base

| | |
|---|---|
| 1 | `login attempt` |
| 2 | `guest login` |
| 3 | `auth failed` |
| 4 | `authorized` |
| 5 | $\omega$ |

**(a)**

`login attempt` $\rightarrow$[0],
`guest login` $\rightarrow$[1],
`auth failed` $\rightarrow$[2],
`authorized` $\rightarrow$[3]

**(b)**



**(c)**

Fig. 4: **(a)** Linear and **(b)** map representations of trace 2 from Figure 1. **(c)** Syntactic tree for property: (¬`authorized` W `guest login`) $\wedge$ G(`guest login` $\rightarrow$ XF `authorized`).

case, since evaluating X $p$ on $\omega$ requires evaluating $p$ on $\omega$ (creating the appearance of an infinite trace). This evaluation terminates because the formula tree for any $p$ is finite, thus Texada will eventually reach an event at a leaf node, whose checking does not recurse.

## A. *Computing support and support potential*

The linear miner returns property instances that are never falsified. We now describe an algorithm to calculate the trace support for a property instance. Interestingness measures of temporal properties have been considered previously [11], [27]. However, these measures were restricted to property types of the form "$a$ is always followed by $b$" or "$a$ always precedes $b$", or nested variations of these. Texada computes an approximation of support and support potential for arbitrary LTL formulae, whose definitions are not captured in this previous work.

Figure 5 lists the pseudocode for the support algorithm[5]. It has the same recursive structure as the linear mining algorithm in Figure 3, but instead of combining boolean values, the calculation in the recursive call combines counts. The support *potential* algorithm (not shown for space reasons) deviates from the support algorithm in Figure 5 at the atomic propositions: support returns 1 at *satisfied* falsifiable time points while support potential returns 1 at *all* time points.

**Key challenge: equivalence of $p \rightarrow q$ and $\neg p \vee q$.** Note that the support calculation for $\vee$ in Figure 5 differs significantly from Figure 3. This is because the equivalence of $p \rightarrow q$ and $\neg p \vee q$ is central to our notion of falsifiability. For example, we would like the support of "$a$ is always followed by $b$", G($a \rightarrow$ XF$b$), to be the number of time points at which $a$ occurs and is eventually followed by $b$, and the support potential to be the total number of time points at which $a$ occurs. For this formula, the absence of $\neg a$ designates a falsifiable time point. However, since a formula like G($a \vee b$) can be re-written as G($\neg a \rightarrow b$) or G($\neg b \rightarrow a$), it is unclear which side of the implication should determine falsifiability.

The algorithm uses a helper function called interval (not listed), which takes a proposition and returns the interval on which this proposition must be checked. This function returns the interval containing only $t$.first_event() on atomic propositions, the interval from $t$.first_event() to the terminal

[5]In our implementation, the algorithm for linear property instance checking is bundled with support and support potential calculation. We explain them separately to simplify our presentation.

```
1  function support(instance prop, trace t)
2  // Returns: the number of atomic proposition evaluations on t which
3  // could have falsified prop on which prop was satisfied
4
5    if (prop is event)
6      if (t.first˙event() == prop) return 1
7      return 0
8
9    else if (prop == $p \vee q$)
10      // if a child with an earlier interval is satisfied,
11      // then the property instance is vacuously true
12      if (interval(p, t) > interval(q, t))
13         if (q is satisfied by t) return 0
14         return support(p, t)
15      else if (interval(p, t) < interval(q, t))
16         if (p is satisfied by t) return 0
17         return support(q, t)
18      else
19         if (both p and q are satisfied by t) return 0
20         else if (support(p,t) >= support(q,t) return support(p,t)
21         return support(q,t)
22
23    else if (prop == $\neg p$)
24      if (t.first˙event() != p) return 1
25      return 0
26
27    else if (prop == Gq)
28      if (t.first˙event().is˙terminal()) return 0
29      return support(q, t) + support(prop, t.next˙event())
30
31    else if (prop == Fq)
32      if (t.first˙event().is˙terminal()) return 0
33      else if (q holds on t) return support(q, t)
34      return support(prop, t.next˙event())
35
36   else if (prop == Xq)
37      if (t.first˙event().is˙terminal()) return 0
38      return support(q, t.next˙event())
```

Fig. 5: A section of the support calculation algorithm.

event on G, F, U, the interval of X's child node (in the formula tree) pushed by one on X, and the union of the interval of the children of $\wedge$ and $\vee$ on these operators. An interval $i$ is smaller than another interval $j$ if $i$ and $j$ are disjoint and $i$ occurs first.

Getting back to the always followed by example, $G(\neg a \vee XFb)$: since the interval for $\neg a$ is strictly before the interval for X F$b$, we execute lines 15-17 in Figure 5 and return the support of XF$b$. Here, the algorithm approximates which side's falsification indicates a falsifying time point by comparing the temporal "order" of each side, given by the interval calculation.

The other operators are simpler. The support of G$p$ is the sum of support of $p$ at each event. Property instances are in negative normal form[6], so the $\neg p$ evaluation treats $p$ as an atomic proposition. The operators not listed for space reasons correspond in structure to the linear checking algorithm.

Overall, the linear miner and support/support potential algorithms have a simple recursive structure. Next, we describe the map miner, which was designed to reduce linear scanning over traces and reduce redundant computation.

## V. Map Miner

The map miner uses the map trace representation (e.g. Figure 4(b)). The map-based property instance checker leverages a key property of LTL: an LTL formula describes the sequence of events *relative* to each other. The map checker, therefore,

```
1  function check−map(property instance prop, index i)
2  // Returns: true if prop holds starting at i, false otherwise.
3  // Omitted: boolean connectives, and operators F, R.
4
5    if (prop == X p)
6      // Special case for the end of the trace.
7      if (i == omega_pos)
8        return check−map(p, i)
9      // Else, move forward and check p
10     return check−map(p, $i+1$)
11
12   if (prop == G p)
13     first_not_p = first−occurrence($\neg p$, [i, omega_pos − 1])
14     // If !p never occurs, G(p) holds.
15     if (first_not_p == false) return true
16     return false
17
18   if (prop == p U q)
19     // Find q.
20     first_q = first−occurrence(q, [i, omega_pos − 1]);
21     // If q never occurs, until does not hold.
22     if (first_q == false) return false
23     first_not_p = first−occurrence($\neg p$, [i, omega_pos − 1])
24     // If !p never occurs, until holds.
25     if (first_not_p == false) return true
26     // Make sure !p did not occur before q.
27     if (first_not_p < first_q) return false
28     return true
```

Fig. 6: A section of the map miner checking algorithm.

can skip over the trace between the relevant events (using the map trace representation). As we will show in Section VII-B, this makes the map checker much more efficient than the linear checker, especially on long traces.

The miner is structured as three functions: check-map, first-occurrence, and last-occurrence. Each one takes a node in the property instance tree (e.g., Figure 4(c)) and an interval in the trace, and then traverses the sub-tree rooted at the node. The check-map function (Figure 6) traverses the tree to implement operator semantics on a node given the validity of the node's sub-trees in the trace at certain positions. This function returns a boolean and evaluates non-temporal operators identically to check-linear (Figure 3, lines 7–12).

Temporal operators are implemented by using the first and last occurrence information provided by the first-occurrence and last-occurrence functions. first-occurrence (Figure 7) returns the first position in some interval where a sub-formula rooted at a node in the property instance tree evaluates to true, or returns false if no such position exists. For example, in lines 13–16 of Figure 6, the check-map function evaluates G $p$ at index $i$ in the trace by using the result of first-occurrence $(\neg p, [i, omega\_pos - 1])$. If first-occurrence returns false, then $\neg p$ never occurs after $i$, which means that G $p$ is true on this interval; otherwise G $p$ is false on this interval.

When first-occurrence is called with an event node (i.e., a leaf in the sub-tree) it runs a binary search on the sorted positions list associated with the event (stored in the map trace representation) to find the first occurrence of the event in a given interval (line 5 in Figure 7). However, first-occurrence must also traverse the sub-tree when it is called with a non-leaf node. Lines 8–20 in Figure 7 detail the case of first-occurrence $(p \text{ U } q, intvl)$: first-occurrence makes a recursive call first-occurrence $(q, intvl)$, and uses the returned position in a call to last-occurrence $(\neg p, [intvl.\text{start}, j])$. Then the first position where $p$ U $q$ holds in this interval is the point at which there are no longer any $\neg p$ events until $q$ or where $q$ occurs (if $q$

[6]Prior to calculating support, Texada converts property instances into negative normal form, pushing the logical not operators to the leaves.

```
function first−occurrence(instance $prop$, interval $intvl$)
// Returns: first index in $intvl$ where $prop$ is true, false otherwise.

  if ($prop$ is event)
    return binary−search($prop$, $intvl$)

  else if ($prop$ == $p$ U $q$)
    // Find first position where $q$ is true.
    $first\_q$ = first−occurrence($q$, $[intvl.\text{start}, omega\_pos - 1]$)
    // Until needs $q$ to occur.
    if ($first\_q$ == false) return false
    // Find the last $!p$ that occurs before the first $q$
    $last\_not\_p$ = last−occurrence($\neg p$, $[intvl.\text{start}, first\_q - 1]$)
    // If $!p$ does not occur before the first $q$, $p$ U $q$ holds
    // on the first element of the interval.
    if ($last\_not\_p$ == false) return $intvl$.start
    // If the last $!p$ before $q$ is after the end of our original
    // interval, $p$ U $q$ holds nowhere on that interval.
    if ($last\_not\_p \geq intvl$.end) return false
    return $last\_not\_p + 1$
```

Fig. 7: Pseudocode for a section of the first-occurrence function that facilitates the check-map function in Figure 6.

is the start of the trace, $p$ U $q$ holds trivially since there is nowhere for $\neg p$ to occur before $q$). We omit last-occurrence for space reasons, but it looks similar to first-occurrence.

Returning to the example used in the linear miner description, consider checking the same CauseFirst property instance in Figure 4(c). We check the same trace, this time using the map representation listed in Figure 4(b). check-map starts at the root node, $\wedge$, and recursively checks its two children.

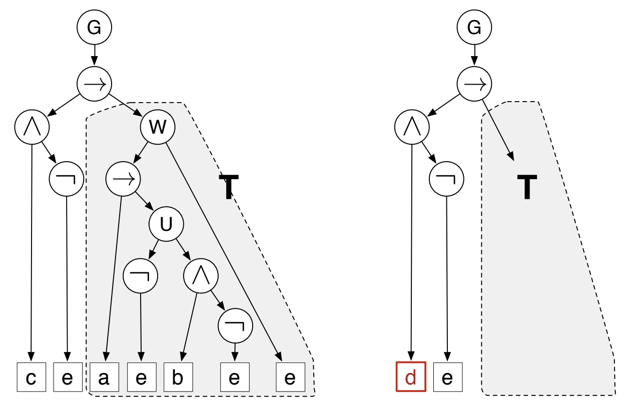
**Left sub-tree of** $\wedge$: To check $\neg$`authorized` W `guest login` we follow Figure 6, lines 19–28 (with line 22 returning true for the W operator). We find the first occurrence of `guest login`: 1. Then, we find the first occurrence of $\neg(\neg$`authorized`$) \equiv$ `authorized`: 3. Since `guest login` occurs before `authorized` $(1 < 3)$, we return true.

**Right sub-tree of** $\wedge$: To check G(`guest login` $\rightarrow$ XF`authorized`)) we find the first occurrence of $\neg$(`guest login` $\rightarrow$ XF`authorized`) $\equiv$ (`guest login` $\wedge$ XG$\neg$`authorized`). We do this by finding the first occurrence of `guest login` and of XG$\neg$`authorized` on our desired interval; if they do not co-occur, we repeat the process on an interval with a later start point. Since both do not occur at one point in the trace, we find no first occurrence of (`guest login`$\wedge$XG$\neg$`authorized`) and find the right sub-tree to be true. Both sub-trees of $\wedge$ are true, so the property holds.

## VI. Checking state memoization

As Texada may check thousands of bindings for a given property type, it will check many LTL formulae with similar syntactic trees. This results in redundant computation. Texada uses memoization to reduce such duplicate computation. A benefit of the tree representation of property types is that we can re-use the result of evaluating one instantiation in evaluating a similar instantiation. This is possible because two instantiations will frequently have identical sub-formulae, or sub-trees. For example, Figure 8 shows two property instances of the property type "$x$ is always followed by $y$ after $w$ until $z$" [15] with nearly identical syntactic trees. The **T** sub-tree in Figure 8 corresponds to the sub-formula $(a \rightarrow (\neg e \text{ U } (b \wedge \neg e)))$ W $e$), which is common to both property instances.

We have implemented a memoization strategy in the map checker to store the result of evaluating `first − occurrence`



**(a)** $G(c \wedge \neg e \rightarrow \boxed{T})$ **(b)** $G({d} \wedge \neg e \rightarrow \boxed{T})$

Fig. 8: **(a)** Syntactic tree for $\text{G}(c \wedge \neg e \rightarrow ((a \rightarrow (\neg e \text{ U } (b \wedge \neg e))) \text{ W } e))$, **(b)** Syntactic tree for $\text{G}(d \wedge \neg e \rightarrow ((a \rightarrow (\neg e \text{ U } (b \wedge \neg e))) \text{ W } e))$. The tree on the right differs in a single node (left-most terminal $d$ node). Therefore, we can determine the truth value of the highlighted sub-tree by using a memoization of the computation of an identical sub-tree that appears on the left.

and `last − occurrence` on a subformula for a given interval. We started with memoizing these functions as they are frequently invoked. We plan to extend memoization to other intermediate checking state in our future work. Our implementation retains memoized state until the mining process completes. Our evaluation in Section VII-B indicates that on short formulas the memoized state is small compared to that of the input log, which must be maintained in memory.

## VII. Evaluation

In this section we show that Texada generalizes over prior work (Section VII-A), that it is fast (Section VII-B), and that it is useful (Section VII-C, VII-D, and VII-E).

### A. *Expressiveness of property types*

Our survey of related work indicates that Texada property types are sufficient to capture both the various temporal specification taxonomies, like in Perracotta [47] and Dwyer et al. [15], as well as specification templates used in the numerous custom-built miners like Synoptic [7].

Perracotta [47] is a tool to mine eight different kinds of temporal properties. These properties cover common program patterns. For example, Perracotta's Response pattern represents a pattern in which an action $x$ is followed by a response $y$. Table I lists the property patterns mined by Perracotta and the corresponding Texada LTL property types.

Dwyer et al. [15] defined a set of LTL specification patterns based on a survey of a variety of specifications. A specification pattern in their formulation consists of a *pattern* and a *scope* over which the pattern must be true. Table II lists two specification patterns as property types. All of the specification patterns trivially translate to Texada property types.

Synoptic [7] is a tool to infer an FSM model from a log of system behavior to support programmer comprehension. Its algorithm relies on the mining of three temporal property types listed in Table III. These properties have also been used to infer more accurate models in InvariMint [5] and CSight [6].

### B. *Performance evaluation*

We evaluated Texada's performance on a machine running 64-bit Ubuntu 14.04 TLS with 8GB RAM and an Intel i5

TABLE I: Property patterns mined by the Perracotta tool [47] and their equivalent property types in Texada.

| **Pattern** | **Reg. Ex.** | **LTL** |
|---|---|---|
| Response | y*(xx*yy*)* | $G(x \rightarrow XF y)$ |
| Alternating | (xy)* | $(\neg y \text{ W } x) \wedge G((x \rightarrow X(\neg x \text{ U } y)) \wedge (y \rightarrow X(\neg y \text{ W } x)))$ |
| MultiEffect | (xyy*)* | $(\neg y \text{ W } x) \wedge G(x \rightarrow X(\neg x \text{ U } y))$ |
| MultiCause | (xx*y)* | $(\neg y \text{ W } x) \wedge G(x \rightarrow XF y) \wedge G(y \rightarrow X(\neg y \text{ W } x))$ |
| EffectFirst | y*(xy)* | $G((x \rightarrow X(\neg x \text{ U } y)) \wedge (y \rightarrow X(\neg y \text{ W } x)))$ |
| CauseFirst | (xx*yy*) | $(\neg y \text{ W } x) \wedge G(x \rightarrow XF y)$ |
| OneCause | y*(xyy*)* | $G(x \rightarrow X(\neg x \text{ U } y))$ |
| OneEffect | y*(xx*y)* | $G(x \rightarrow XF y) \wedge G(y \rightarrow X(\neg y \text{ W } x))$ |

TABLE II: Two example specification patterns from [15], each one demonstrates a different scope.

| **Pattern ; Scope** | **LTL** |
|---|---|
| $p$ responds to $s$, $t$; after $q$ | $G(q \rightarrow G((s \wedge XF t) \rightarrow X(\neg t \text{ U } (t \wedge F p))))$ |
| never $p$; between $q$ and $r$ | $G((q \wedge \neg r \wedge F r) \rightarrow (\neg p \text{ U } r))$ |

TABLE III: Property patterns used in Synoptic [7] and their equivalent LTL property types in Texada.

| **Pattern** | **LTL** |
|---|---|
| Always followed by | $G(x \rightarrow XF y)$ |
| Always precedes | $F y \rightarrow (\neg y \text{ U } x)$ |
| Never followed by | $G(x \rightarrow XG(\neg y))$ |

Haswell quad-core 3.2GHz processor. Texada is implemented in about 7,500 lines of C++, depends on the Boost library[7], and uses the SPOT library [14] to parse and traverse LTL property types. For all performance experiments comparing to Synoptic, we use Texada revision a410 [40].

We first compare Texada's linear and map miners against the temporal miner in Synoptic [7], which mines the three temporal property types in Table III[8]. Synoptic mines instances of these property types from the input log and guarantees that they are satisfied in the inferred FSM model. Synoptic is an interesting point of comparison because although it is implemented in Java, it is highly optimized for mining the three temporal properties. We were interested to see if Texada's general property miners could out-perform Synoptic's specialized miner. For a proper comparison, these experiments used a confidence threshold of 1 and a support threshold of 0.

We generated a set of random synthetic logs, each with a specific number of traces, unique events, and trace length. We ran Texada's map miner, linear miner, and Synoptic's miner (using the --onlyMineInvariants option) 5 times on each log input and report the average runtime. We compared the Texada miners against Synoptic's miner along three dimensions: varying the number of traces (Figure 9), the length of the traces (Figure 10), and the number of unique events (Figure 11). These three dimensions determine both Synoptic's and Texada's performance. The take-away from these experiments is that on the three Synoptic invariant types in Table III the map miner dominates the linear miner and the Synoptic miner.

**Varying the number of traces (Figure 9)**. The trace length was held constant at 10,000 and the number of unique events was held constant at 50. Synoptic threw an OutOfMemoryError at 260 traces. All three miners exhibit a linear slowdown indicating a fixed performance penalty to processing an additional input trace.

**Varying trace length (Figure 10)**. The number of traces per log was held constant at 20, and the number of unique events was held constant at 100. This figure is similar to Figure 9 in that all miners slow down linearly. The linear miner has the worst performance since each additional event in a trace requires further recursion (e.g., for the G operator for "always followed by" property in Table III).

**Varying number of unique events (Figure 11)**. The number of traces per log was held constant at 20 and the length of a trace was held constant at 10,000. The linear miner does especially poorly with an exponential slow down, while the Synoptic and map miners have similarly good performance.

The above results indicate that the map miner generally outperforms the linear miner. This is because the map miner often avoids recursive trace traversal that the linear miner is forced to perform. However, we did find property types on which the map miner performed worse than the linear miner. We are working to further characterize their relative performance.

**Varying support and confidence thresholds.** Texada's code is optimized for the default thresholds of support of 0 and confidence of 1. For example, the linear miner can short-circuit further recursion along the trace for some of the operators[9]. Non-default support and confidence thresholds disable most of these optimizations. This is because to calculate the exact support or support potential of a property instance, Texada must count all possible falsifiable time points. We found that decreasing the trace confidence threshold generally increases runtime. In Figure 12 we see that increasing the support threshold causes a variety of effects while mining the always followed by, always precedes, and never followed by properties[10]. In each case an increase in support threshold causes an initial jump in runtime, followed by a flat section and sometimes a decrease in runtime. Introducing the support threshold forces Texada to stop short-circuiting on the evaluation of several operands; however, once the support threshold is set too high, the property instance will fail the threshold on some trace. At this point Texada will begin to short-circuit evaluation by moving to the next instantiation, instead of checking the current instantiation on the remaining traces in the log. Note that this trace short-circuiting is unsafe if thresholds are specified at the global level, as this forces evaluation over the entire log.

We discuss the advantages of introducing support and confidence thresholds despite the large runtime cost in Section VII-C.

**Benefit of memoization.** To evaluate the effect of memoization, we carried out an experiment with three instantiations of the property "$p$ occurs at most twice between $q$ and $r$ and $s$ occurs at most twice between $q$ and $r$" [15]. We selected this property type because we found that the map checker was slower than usual in evaluating instances of this type. To compare the runtime of the map checker with and without memoization we slowed down the checker by using a log containing a single, especially long, trace that we synthesized. The trace consisted of 36 million events, sampled at random from a set of 10 event types. We measured the runtime in checking 3 instantiations[11]:

[7]http://www.boost.org

[8]Synoptic is one of the few miners that was available to us for evaluation.

[9]Some examples of short-circuiting: if one branch is found to be false in evaluating $\wedge$, if $\neg p$ occurs in evaluating $Gp$, or if a property instance is found to be false on one trace in the log.

[10]These properties were mined on a randomly-generated log over 18 events with 20 event traces, each with 10,000 events.

[11]We used Texada revision df18 [40]

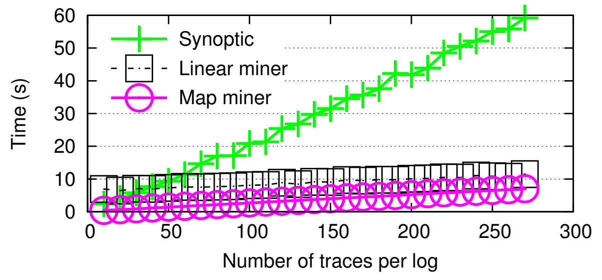


Fig. 9: Time to mine the three Synoptic property types using Synoptic and Texada's linear and map miners for logs with varying numbers traces per log.

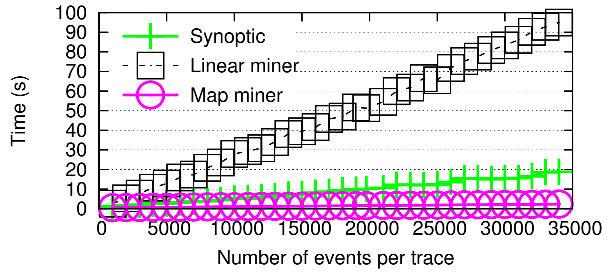


Fig. 10: Time to mine the three Synoptic property types using Synoptic and Texada's linear and map miners for logs with varying number of events per trace.

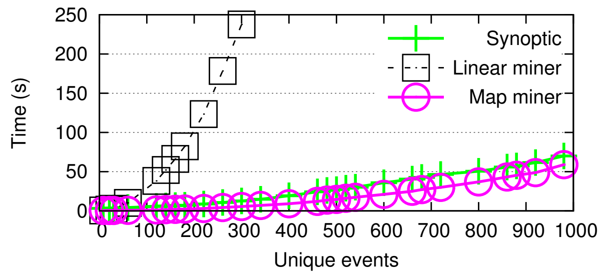


Fig. 11: Time to mine the three Synoptic property types using Synoptic and Texada's linear and map miners for logs with varying number of unique events.

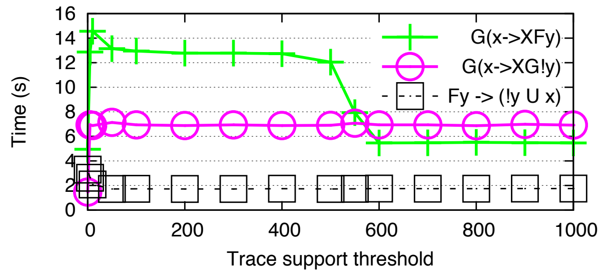


Fig. 12: Time to mine each Synoptic property with the Texada linear miner given different trace support thresholds.

*A*, *B*, and *C*. We designed the bindings for these so that the memoized state of *B* and *C* would help in evaluating *A*, and vice-versa. However, evaluating *B* would not help in evaluating *C*, and vice-versa.

We ran the map checker on *A*, *B*, and *C* in four configurations: (1) $[A,B,C]$ with memoization, (2) $[A,B,C]$, clearing memoized state after checking *A*, (3) $[B,C,A]$ with memoization, (4) $[B,C,A]$, clearing memoized state after checking *C*. We ran each configuration 100 times and report the average runtime.

The two orders that do not clear memoized state had nearly identical runtimes of about 5.7s, indicating that for this example the checking order does not matter. We believe that this will generally hold true since we memoize the evaluation of every sub-formula. The two orders that cleared the memoized state also had similar runtimes of about 6.1s. In this experiment, memoization decreased the total runtime by 7%. In practice, we expect the speed-up to be significantly higher as many more (than just three) instantiations would share the memoized state.

Memory use in these experiments peaked at 342MiB and was primarily used to store the input trace. The experiments generated 14 KiB of memoized state, or 5 KiB per property instance. Mining all possible instances of the same property type (with four variables) over the same log (over 10 event types) requires checking 5,040 property instances[12]. At the rate of 5 KiB per property instance, this process will generate about 25 MiB of memoized state. We aim to improve on the memory use of memoized state by developing expiration policies to regularly delete memoized state that is not going be reused.

### C. Mining patterns of user activity

To evaluate Texada's utility we applied it to a web log used to evaluate the BEAR framework [23] and Perfume [35]. This log records web requests for a real estate website on which users browse or search for houses and apartments to rent or buy. Each request has a timestamp and an anonymized IP address; we use these to interpret the log as separate executions of the web-site, one execution per client who accesses the site, where events are the visited site pages.

We reused the event types from the BEAR study by pre-processing the log to remove irrelevant events, like those generated by web crawlers, and by assigning semantically identical events to the same label. The pre-processed log contained about 12,000 lines, with 13 different events. We used Texada revision e436 [40] to mine the property types in Table II with the linear miner. In the analysis we ignored 4 rarely occurring events to simplify inspection. Due to space constraints we discuss the implications of a select set of mined property instances. The log had no ground truth to compare our results to, but we believe the following results show utility. For each result below we report a runtime that is an average over 5 runs.

| **1.** | F *news_page* $\rightarrow$ (!*news_page* U *news_article*) |
|---|---|

*Visits to news article always precede visits to the news page.* (Texada runtime: 1.6s, instances returned: 15, support threshold: 8,000, confidence threshold: 0.98.) This is an instantiation of "*x* always precedes *y*" in Table III and has 0.99 confidence and 9,605 support. This instantiation suggests that the news articles generate much more initial interest than the news page, and that this page is only accessed by users who have taken the time to access an article and want more content. It may indicate the news page needs to be redesigned for broader appeal.

| **2.** | G(*sales_page* $\rightarrow$ XF($\neg$*sales_anncs*)) |
|---|---|

*After visiting a sales page, the sales announcement pages is always visited.* (Texada runtime: 7.5s, instances returned: 3, confidence threshold: 0.80.) This is an instantiation of "*x* is always followed by *y*" in Table III and has 0.87 confidence and

[12]The default configuration is to generate bindings without replacement: no two variables are bound to the same event.

TABLE IV: Number of instances of $G(x \rightarrow XG(\neg y))$ mined from the BEAR log with the linear miner using varying global support and global confidence thresholds. The cell highlighted in the upper left corresponds to the default Texada thresholds.

| **supp.** \ **conf.** | **1** | **0.95** | **0.9** | **0.85** | **0.8** | **0.7** | **0.5** | **0.3** | **0.1** |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 11 | 120 | 141 | 150 | 165 | 169 | 182 | 182 | 182 |
| **200** | 5 | 105 | 122 | 127 | 142 | 145 | 155 | 155 | 155 |
| **500** | 2 | 96 | 111 | 116 | 130 | 133 | 143 | 143 | 143 |
| **5,000** | 0 | 87 | 100 | 105 | 118 | 121 | 130 | 130 | 130 |
| **15,000** | 0 | 71 | 78 | 81 | 90 | 93 | 99 | 99 | 99 |
| **50,000** | 0 | 47 | 51 | 53 | 59 | 61 | 64 | 64 | 64 |
| **100,000** | 0 | 29 | 32 | 33 | 35 | 37 | 39 | 39 | 39 |
| **200,000** | 0 | 17 | 18 | 19 | 21 | 21 | 21 | 21 | 21 |

2,232 support. We expect users interested in buying or selling a property to navigate from the main sales page to the sales announcements. However, the lower confidence of the property suggests there may be a block to easy navigation between the two. The sales page could be revised to better funnel users towards announcements.

**3.** $G(\mathit{search} \rightarrow G((\mathit{news\_article} \wedge XF\,\mathit{renting\_anncs}) \rightarrow X(\neg \mathit{renting\_anncs}\ U\ (\mathit{renting\_anncs} \wedge F\,\mathit{sales\_anncs}))))$

*Users who visit news article pages eventually visit a sales announcement page.* (Texada runtime: 12.8s, instances returned: 158) We ran this property with confidence threshold 1 and support threshold 0. This is an instantiation of "$p$ responds to $s$, $t$ after $q$" in Table II. This property says that after a search, every time a user accesses a news article and then a renting announcement, the user will then subsequently access a sales announcement (something that does not happen every time users visit a renting announcement after search). This may indicate that news articles impact users' navigation, which can prompt work on news article accessibility.

The support and confidence thresholds allow the user to focus on the most likely instantiations. The filtering effect due to support and confidence is illustrated in Table IV, where increasing global confidence and support thresholds decreases the number of instances of the never followed by property found on the BEAR log. We also see that high support automatically filters out low-confidence properties; the bottom row of the table shows that there were no additional properties with support at least 200K and a confidence value below 0.8. The properties in the bottom row likely include the key patterns in the log.

Two features distinguish the above properties from results derived using other tools on the same log [23], [35]. **(1)** Texada-generated properties are concise and allow a developer to focus on and filter by a set of relevant events without needing to understand other events in the trace. **(2)** The properties have a well-defined LTL structure stipulated by the property type.

### D. Validating expected behaviour

The sleeping barber problem is a classic concurrency problem attributed to Edsger Dijkstra [13]. In the problem a barber alternates between cutting hair, sleeping, and checking for customers in his waiting room. If a customer walks in and finds the barber sleeping, the customer wakes up the barber and gets a haircut. If the customer walks in and the barber is cutting hair, the customer waits in the waiting room if there is space, otherwise the customer leaves. When the barber is done cutting hair, he checks if there are customers in the waiting room. If there are, then he cuts one of the customer's hair; otherwise, he goes to sleep. In our version of the problem, each customer desires a certain number of haircuts.

Our solution is based on a solution by Teemu Kerola and represents the barber and each customer as a thread. The implementation tracks the state of the barber and each customer and prints the state of each thread whenever (1) a customer enters, (2) a customer exits, (3) a customer receives the desired number of haircuts, and (4) the barber retrieves a customer from the waiting room. We treat the barber's and customers' states as events. The resulting trace includes multiple events for each of the above time points. We call this a *multi-propositional* trace: multiple atomic propositions (customer and barber states) are logged at each time point.

We checked a log from five runs of our solution to the sleeping barber problem. The runs had seven customers, each of which desired two haircuts. There were 38 unique events in the 4874 lines long log, with traces having 892–1036 lines (99-115 time points). We used Texada to confirm that desired properties hold over these runs. This analysis is not a proof of correctness, but as we show below, it can reveal bugs.

**No two customers receive haircuts at the same time.** Instantiations of the $G(x \rightarrow \neg y)$ will test this property when $x$ and $y$ are bound to $\text{Customer-}i\text{-GettingHairCut}$ and $\text{Customer-}j\text{-GettingHairCut}$, respectively, with distinct $i$ and $j$. Running this property with the map option took 0.064 seconds and outputted 424 property instances. We filtered these instances to find those of the form $G(\text{Customer-}i\text{-GettingHairCut} \rightarrow \neg\text{Customer-}j\text{-GettingHairCut})$, for all distinct $i, j$ pairs, demonstrating no violations of the mutex in these runs.

**Each customer receives 2 haircuts.** We check this property by confirming that each customer transitions to the GettingHairCut state from another state exactly twice. A property that expresses this is an extension of the Bounded Existence property with global scope [15]: $(\neg x\ W\ (x\ W\ (\neg x\ W\ (x\ W\ G\neg x)))) \wedge F(x \wedge XF(\neg x \wedge XFx))$. The first half of this property assures that all transitions to $x$ occur *at most* twice, and the second half assures that these transitions occur *at least* twice[13].

With the map miner, Texada took 0.031s to check this property, returning 13 property instances. These instances bind $x$ to $\text{Customer-}i\text{-Waiting}$ and $\text{Customer-}i\text{-GettingHairCut}$ for all $i$, except Customer-4. Examining the log, we found that in one trace, Customer-4 transitioned to the GettingHairCut state only once. The second time the customer was observed to enter, to be the only one in the waiting room when the barber checked it, and to exit the barbershop. This turned out to be a bug in our logging code, not the implementation: no time point was logged during the haircut for Customer-4. Texada helped us find this bug.

For this example we have also used Texada to validate the property *Customers are served in the order in which they sit down*. In addition, we used Texada in a similar way to validate an implementation of the dining philosophers problem. We omit these for space reasons.

### E. Mining data-temporal properties

Texada can be used to develop more advanced program analyses. We prototyped a tool that combines Texada with Daikon [17] (a tool to infer likely data invariants from program traces). Our prototype infers likely *data-temporal* properties.

[13]This is an example of an LTL property that is checkable with Texada, but which may be easier to specify as an automaton.

As an example, consider a `Queue` class with fields `size` and `capacity`, which represent the current size and the maximum size of the queue, respectively. For this class Daikon may infer a data invariant like $\texttt{size} \leq \texttt{capacity}$. With Texada, we can infer temporal relations between these data invariants. For example, the `Queue` may also have an `isFull` flag. While Daikon can infer a data invariant like $(\texttt{isFull} == \texttt{true}) \iff (\texttt{size} == \texttt{capacity})$ at some program points, a more powerful property can be formulated temporally: $(\texttt{isFull} == \texttt{false})$ U $(\texttt{size} == \texttt{capacity})$. That is, "`isFull` is `false` until `size` is equal to `capacity`"; it is an instance of "*x* holds until *y* becomes true". This data-temporal property captures an important correctness condition: the queue is not flagged as full until it reaches capacity.

We prototyped a version of a tool to mine data-temporal properties. It works as follows. First, it uses one of Daikon's frontends to instrument a program to collect data traces from a set of program executions (e.g., the program's test suite). Then, it uses Daikon to infer likely data invariants from these traces. The Daikon invariants are spliced into the control flow of the data traces, matching invariants with their corresponding program points. This generates a set of totally ordered invariant traces (which are multi-propositional, as defined in Section VII-D). Finally, the tool runs Texada on these traces to generate data-temporal properties.

We applied this technique to QueueAr, a program distributed with Daikon that implements the `Queue` class discussed above. We used QueueAr's test suite to generate invariant traces, one per `Queue` instance. Using Texada we mined:
$\neg$"$\texttt{this.currentSize} \geq 1$" W "$\texttt{this.currentSize} == 0$"
This instantiation reflects an important (but expected) property of the queue's test suite: the Queue is always created empty.

We think that data-temporal properties can be useful in test-case generation and for bug detection. Evaluation of data-temporal properties is part of our future work.

## VIII. Related work

The problem of checking an LTL formula on a finite trace has been considered by van der Aalst et al. [42]; their algorithm is similar to Texada's linear checker. There are also efficient techniques for checking an LTL formula on a single trace using Alternating Finite Automata [39]. However, the generated AFA depends on both the property instance and the trace, making the approach not scalable to our context, which requires checking a large set of LTL properties across many traces.

Many specification mining tools have appeared in prior work [38]. Existing tools mine LTL properties, but unlike Texada, these are designed to mine a particular set of property types. The Perracotta tool by Yang et al. [47] mines several two variable response patterns and chains alternating properties together to form multi-variable properties. It also supports approximate inference to filter out uninteresting properties. Lo et al. extend Yang et al.'s work with a miner that produces quantified temporal rules that capture data flow relationships among events [33]. Li et al. [28] extend Perracotta to mine simple LTL patterns from traces and merge these to analyze digital circuits. These same LTL patterns are mined between data invariants in [4], [8]. Although prior work has not explicitly mined temporal relationships *between* data invariants, there has been work on augmenting models, such as FSMs and Live Sequence Diagrams, with data invariants [30], [34], [45].

Weimer et al. [46] mine alternating and response patterns on exceptional control-flow to effectively identify bugs. Javert [19] also mines alternating patterns, along with resource ownership patterns (i.e., (ab*c)*) and composes them into more complex properties. Reger et al. use a similar pattern-composition technique [24] and extend it to accommodate imperfect traces [25]. Gabel et al. [20] improve the efficiency of mining the alternating and resource allocation patterns by using BDDs to represent property types. Like all these miners, Texada is a dynamic analysis technique and may produce false positives. One promising approach to validate mined specifications of programs whose code is available is to use deductive specification inference [22].

A number of tools mine association rules (response patterns between events sets): DynaMine [29] examines revision history of a program and observes program behaviour to filter behaviour and to avoid blow-up of potential patterns. Thummalapenta et al. [41] mine such rules to infer exception-handling rules. Lo et al. [11] develop an algorithm to mine response patterns between sequences of events.

InvariMint [5] is a declarative model inference specification approach that requires an algorithm designer to specify model inference algorithms as parameterized FSM templates with bindings functions that resemble Texada property types. Texada can be used to mine the legal bindings in InvariMint.

Recent work on inferring various types of models from source code, execution traces, and log files, relies on temporal properties [6], [7], [32], [44]. For example in [44] temporal properties are specified manually by a user to guide specification mining, and in [6], [7] a set of properties are mined automatically from the input traces. Texada can improve both types of work — the Texada-mined property instances can be presented to the user for selection, or used en masse.

The recursive-descent-parsing-style checking of LTL in Texada does not work for classic LTL with infinite trace semantics. The standard way to check an LTL formula is to derive a Büchi Automaton [43] and to intersect it with the model. Using existing model-checking tools out-of-the-box (e.g., SPOT [14]) for specification mining is not typically possible because of the mis-matched trace semantics.

## IX. Conclusion

Texada is a general LTL miner that is a swiss army knife of property mining. It replaces a suite of tools that are based on specific templates and supports tasks ranging from log exploration to property validation. We presented two algorithms in Texada that mine arbitrary LTL properties, and an algorithm to compute the support and confidence for an arbitrary LTL property. Our evaluation demonstrates that Texada is fast and can outperform Synoptic, an existing special-purpose miner. We also demonstrated Texada's utility by mining properties from a web log of user activity, by validating expected properties of a solution to the Sleeping Barber problem, and by mining data-temporal properties from traces of Daikon invariants. Property types in Texada encompass those proposed in prior work and can capture a wide variety of temporal patterns. We encourage other researchers to build their software analyses on top of Texada, which is open sourced [40] and includes 67 pre-defined property types from prior work [7], [15], [47].

## Acknowledgments

This research is funded by an NSERC discovery award and an NSERC USRA award.

## References

[1] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 182–195, New York, NY, USA, 2003. ACM.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] A. Bauer, M. Leucker, and C. Schallhart. The Good, the Bad, and the Ugly, But How Ugly Is Ugly? In *Runtime Verification*, pages 126–138. Springer Berlin Heidelberg, 2007.

[4] M. Bertasi, G. Di Guglielmo, and G. Pravadelli. Automatic generation of compact formal properties for effective error detection. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sept 2013.

[5] 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 Transactions on Software Engineering (TSE)*, 41(4):408–428, April 2015.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *SIGSOFT FSE*, pages 267–277, 2011.

[8] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli. Dynamic Property Mining for Embedded Software. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 187–196, New York, NY, USA, 2012. ACM.

[9] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 5–14, New York, NY, USA, 2007. ACM.

[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96, Trento, Italy, 2010.

[11] S.-C. K. David Lo and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[12] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 233–243, Portland, ME, USA, July 2006.

[13] E. W. Dijkstra. Cooperating sequential processes. 1968.

[14] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized büchi automata. In *Proceedings of the 2004 Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 76–83, Oct 2004.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[16] E. A. Emerson. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter Temporal and Modal Logic, pages 995–1072. J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, 1990.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, 2001.

[18] D. Fahland, D. Lo, and S. Maoz. Mining branching-time scenarios. In *Proceedings of the Conference on Automated Software Engineering (ASE)*, pages 443–453, Nov 2013.

[19] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA, 2008.

[20] M. Gabel and Z. Su. Symbolic Mining of Temporal Specifications. In *Proceedings of the 2008 International Conference on Software Engineering*, 2008.

[21] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[22] M. Gabel and Z. Su. Testing mined specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Cary, NC, USA, 2012.

[23] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, Hyderabad, India, 2014.

[24] H. B. Giles Reger and D. Rydeheard. A Pattern-Based Approach to Parametric Specification Mining. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2013, November 11-15 2013.

[25] H. B. Giles Reger and D. Rydeheard. Automata-based Pattern Mining from Imperfect Traces. In *Proceedings of the Second International Workshop on Software Mining*. ASE 2013, November 11-15 2013.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[27] T.-D. B. Le and D. Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 331–340, March 2015.

[28] W. Li, A. Forin, and S. A. Seshia. Scalable Specification Mining for Verification and Diagnosis. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 755–760, New York, NY, USA, 2010. ACM.

[29] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305. ACM, September 2005.

[30] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 387–396, Antwerp, Belgium, 2010.

[31] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, 2007.

[32] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2009.

[33] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. *Sci. Comput. Program.*, 77(6):743–759, 2012.

[34] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[35] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral Resource-Aware Model Inference. In *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, September 2014.

[36] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012.

[37] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012.

[38] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, May 2013.

[39] V. Stolz and E. Bodden. Temporal Assertions Using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, May 2006.

[40] Texada. https://bitbucket.org/bestchai/texada.

[41] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[42] W. van der Aalst, H. de Beer, and B. van Dongen. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer Berlin Heidelberg, 2005.

[43] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin Heidelberg, 1996.

[44] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–257, L'Aquila, Italy, September 2008.

[45] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring Extended Finite State Machine Models from Software Executions. *Empirical Software Engineering*, pages 1–43, 2015.

[46] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[47] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 282–291, Shanghai, China, 2006.