Investigating Program Behavior
Using the Texada LTL Specifications Miner

Caroline Lemieux, Ivan Beschastnikh
Computer Science
University of British Columbia
Vancouver, BC, Canada

Abstract—Temporal specifications, relating program events
together through time, are useful for tasks ranging from bug detection
to program comprehension. Unfortunately, such specifications are
often lacking from system descriptions, leading researchers
to investigate methods for inferring these specifications from
code, execution traces, code comments, and other artifacts. This
paper describes Texada, a tool to dynamically mine temporal
specifications in LTL from traces of program activity. We review
Texada’s key features and demonstrate how it can be used to
investigate program behavior through two scenarios: validating
an implementation that solves the dining philosophers problem
and supporting comprehension of a stack implementation. We
also detail Texada’s other, more advanced, usage options. Texada
is an open source tool: https://bitbucket.org/bestchai/texada

I. INTRODUCTION

Program specifications, or formal descriptions of expected
program behavior, are helpful in various software engineering
tasks. Specifications can help in bug detection [19], model
creation [2] and test case generation [3]; they can help with
manageability by capturing what is important [9]; and, being
more concise than code, they can help to document intended
program behavior [8]. However, specifying behavior can be
tedious or difficult, and the resulting specifications can fall out
of date as the system changes. Developers, therefore, rarely
write down specifications of their programs.

One way to overcome the lack of specifications is to
infer, or mine, specifications from existing program artifacts.
For example, a constraint like “a lock must be claimed before
entering function ata_dev_select()”, can be extracted stati-
cally from code or comments [16]. An alternative source of
information, and the focus of this paper, is dynamic program
behavior, which reflects actual program execution.

This paper overviews Texada, a tool for mining linear
temporal logic [7] (LTL) property instances (matching a user-
defined temporal property type) from traces of program be-
behavior, or logs. Figure 1 illustrates the inputs to Texada and
the corresponding outputs through an example. At the top of
the Figure is a log, consisting of four traces of methods
manipulating a Queue instance. The second input below is the
property type “x always precedes y”, which we express in
LTL as ¬y W x. This property type is included in the Texada
distribution so that a user can input this pre-defined property
template without needing to compose the LTL expression
directly. The output for this log has four property instances
(bottom of Figure 1). The first three instances indicate that
the newQueue() method invocation precedes all other method
calls (i.e., it is the first event in every trace). The last property
instance, ¬dequeue() W enqueue(), implies that the dequeue() method is never invoked before enqueue(). So, in these traces
the dequeue instance is used without violating its API contract.

Texada uses LTL to relate dynamic program behaviors, or
events, that appear in a textual log file. The events are user-
defined with regular expressions (not shown in Figure 1) and
may represent a variety of program features. For example, Tex-
ada can be used to analyze console logs generated with printf
statements (events are logged messages), stack traces produced
by stack-tracing tools [1] (events are function names), syscalls
traces produced by tools like strace (events are system calls),
and output from other tools that track program information at
time and emit a text file.

A key feature of Texada is its use of an LTL property type
e.g., ¬y W x in Figure 1). This property type has no structural
constraints: the user can specify any LTL property type without
writing new code. This feature distinguishes Texada from prior
tools that mine a specific set of pre-defined LTL types [9]-
[11], [18], [19]. The Texada repository contains 67 pre-defined
property types based on prior work [2], [6], [19], including all
of the ones mentioned in this paper. These types are usable
without needing to compose any LTL and can be run in
aggregate against an input log.

The Texada algorithms are described in another publica-

\[ \neg y W x \equiv F \neg y \land (\neg y U x) \equiv G(\neg y) \lor (\neg y U x) \]
This paper illustrates how Texada can be used to investigate complex program behavior. We (1) use Texada to validate a solution to the dining philosophers problem, and (2) show how Texada can help understand the usage of the stack data structure.

II. TExADA USAGE SCENARIOS

Texada can be used as a command line tool or from a web interface2. Below, as we proceed through the two scenarios, we explain each command line option to Texada and also give the Texada running time3. The traces and arguments used in each example are available in the Texada repository, so that the given results can be replicated.

Validating a solution to the dining philosophers problem.

If a developer knows the specification of a system, Texada can be used to verify whether the specification holds on a set of observed runtime traces from the system. To illustrate this, we use an implementation of the dining philosophers problem (Figure 2). In this classic concurrency problem, the goal is to feed a set of philosophers who are sitting at a table (representing threads of execution) and have to share chopsticks (representing shared resources) to access food on their plates. Between any two adjacent philosophers is one chopstick, which can be held by only one philosopher at a time (exclusive access). A philosopher can be in one of three states: thinking (not accessing chopsticks), hungry (intending to access chopsticks), or eating (actively accessing chopsticks).

We use a multi-threaded Java implementation of a solution to the dining philosophers problem created by Stephen J. Hartley. We modified the logging statements in this implementation to output the state of each of the five philosophers whenever one of them changes state. To study the resulting log we use Texada’s multi-propositional trace parsing feature, which allows us to reason about traces where more than one atomic proposition occurs at a time point. In this scenario, we consider each philosopher’s state to be an atomic proposition.

We used a log consisting of 5 one minute traces of the implementation and found that the implementation (in those minutes) satisfied several correctness properties of the dining philosophers problem.

1. Any two adjacent philosophers, $i$ and $j$, should never eat at the same time. In LTL: $G(\neg i \text{ eating} \rightarrow \neg j \text{ eating})$. We use Texada to derive this set of properties with: ./texada -m -f 'G(x->!y)' --parse-mult-prop log.txt4. The output involving philosopher $i = 2$ included:

\[
G(\neg 2 \text{ eating} \rightarrow \neg 1 \text{ eating}) \\
G(\neg 2 \text{ eating} \rightarrow \neg 3 \text{ eating}) \\
G(\neg 2 \text{ eating} \rightarrow \neg 1 \text{ hungry}) \\
G(\neg 2 \text{ eating} \rightarrow \neg 2 \text{ thinking})
\]

Texada generated similar output for the other four philosophers. The first two properties in this output confirm our expected invariant: philosopher 2 cannot eat at the same time as philosopher 1 or philosopher 3. The bottom two properties confirm the more obvious invariant that philosopher 2 can only be in one state at a time.

2. If we let the solution run long enough, we may be able to detect another desirable property: eventually, non-adjacent philosophers get to eat at the same time. That is, for non-adjacent philosophers $i$ and $k$, we would like to make sure that eventually $i$ is eating and $k$ is eating co-occur. Any interesting solution to the problem should satisfy this property.

This information is in fact implicit in the Texada output from the previous query, with property type $G(x \rightarrow \neg y))$. Since for the binding of $2$ is eating to $x$, the only bindings to $y$ were “1 is eating,” “3 is eating,” “2 is hungry” and “2 is thinking”, clearly at some point all other events co-occurred with “2 is eating” (and similarly for philosophers 0, 1, 3, and 4). This equivalence comes from the fact that $\neg G(x \rightarrow \neg y)) \equiv \neg G(\neg x \vee \neg y) \equiv F(x \wedge y)$. However, the absence of the instantiations of $G(x \rightarrow \neg y))$ with $x$ and $y$ bound to non-adjacent philosophers eating in Texada’s output simply indicates these instantiations were invalidated on at least one trace, not necessarily on all traces. If we explicitly check for $F(x \wedge y)$, we will observe, amongst other instantiations, the following output:

\[
F(\neg 0 \text{ eating} \& \neg 2 \text{ eating}) \\
F(\neg 0 \text{ eating} \& \neg 3 \text{ eating}) \\
F(\neg 1 \text{ eating} \& \neg 3 \text{ eating}) \\
F(\neg 1 \text{ eating} \& \neg 4 \text{ eating}) \\
F(\neg 2 \text{ eating} \& \neg 4 \text{ eating})
\]

This confirms that pairs of non-adjacent philosophers eventually get to eat at the same time in all traces.

The traces in this scenario illustrate how a developer can use Texada to study a non-trivial concurrent program. In this case we used Texada to validate that the runtime traces generated by a solution to the dining philosophers problem satisfy basic correctness and concurrency properties. The commands took 0.035 to 0.044s to run on the log, which consisted of 5 traces, 1088 time points (5440 total events), and 15 unique event types.

Comprehension of stack usage. Texada’s flexibility in temporal property inference makes it well-suited for supporting program comprehension tasks. In the second scenario, we analyze a log with method call traces for a stack data structure. Our goal is to understand how this stack is exercised. In this case, the stack implementation is a Java program provided with the Daikon tool [8] that implements a stack using an array. Note that in this implementation, pop is called topAndPop().

First, we want to know if all elements in the stack are always removed. Here, we use Texada’s support and support potential statistics (detailed further in Section IV). For the $G(x)$ property, the support is the number of $x$ occurrences
and the support potential is the total number of events. Setting --conf-threshold to 0 outputs all instantiations; use-global-thresholds sets this threshold over the entire log and print-stats prints out global statistics for each instantiation. Running ./texada -l -f ‘G(x)’ --conf-threshold 0 --use-global-thresholds --print-stats log.txt, we see the following output:

```
G(push(java.lang.Object))
  support: 654
  support potential: 8212
  confidence: 0.0796
G(topAndPop())
  support: 482
  support potential: 8212
  confidence: 0.0490
```

There are clearly more objects pushed onto the stack than popped off (i.e., 654 > 402). Notwithstanding possible exceptional situations, it is likely that several objects are never removed from the stack in this scenario. To learn more about the order in which push and pop are called, we mine the “x is always followed by y” rule. We summarize Texada’s results by grouping all outputted y instantiations in ( ) brackets:

```
G(isFull()) -> X(F(isEmpty()), "top()")
G(push(java.lang.Object)) -> X(F(isEmpty()), "isFull()", "top()", "topAndPop()")
G(topAndPop()) -> X(F(isEmpty()), "isFull()", "top()")
G("top()" -> X(F(isEmpty()))
```

Above we see that despite the fact that not all of the objects are popped off the stack, at least some object must be, since G("push(java.lang.Object)" -> XF"topAndPop()") holds. We also see that pop is always followed by top, which gives us some information about the workflow.

Finally, we might want to check whether pop is ever called before push: this is a pattern we do not necessarily expect to witness when a human exercises a queue and which could be unsafe. For this, we try to mine “x always precedes y” (~y → W x) since all we are interested in is what needs to happen before pop, we use Texada’s constant event option (-e) to bind y to topAndPop(). The full command is ./texada -m -f '!topAndPop()' W x' -e 'topAndPop()' log.txt and the output is:

```
!topAndPop() W ("isFull()", "isFull()", "StackAr(int)", "top()")
```

Interestingly, we see that push does not appear bound to x, meaning that push does not always precede pop. So in some runs, pop is called before push. As our focus has been on the relationship between push and pop, it might be informative to check the method calls that always precede push; binding y to push(java.lang.Object) returns:

```
!push(java.lang.Object) W ("isFull()", "isFull()", "StackAr(int)", "top()", "topAndPop()")
```

The above tells us that in all executions, any call to push was preceded by a call to pop. This is unusual; we might want to check if this stack has a guard mechanism to protect against calls to pop on an empty stack. For this we mine instances of “x is always immediately followed by y”, or G(x → X(y)). Texada outputs the following instances:

```
G("topAndPop()" -> X("isFull()")
G("top()" -> X("isEmpty()")
G("push(java.lang.Object)" -> X("isFull()")
G("makeEmpty()" -> X("isFull()")
```

The output supports the idea that the system has guards in place to prevent over and under flow; isEmpty() is always called immediately after the call to pop, suggesting that it is called within the method. Similarly, isFull() appears to be called within push.

With these few calls to Texada, we have learned that the system calls more pushes than pops, likely contains guards to prevent over and under flow, and has no strict rules on push and pop method call order; in fact, pop appears to always be called before push. The runtimes of these commands ranged between 0.034s and 0.065s. The log consisted of 87 traces, totaling 8211 events and 7 unique event types.

In both scenarios Texada was useful because we had a precise idea of which “correct” invariants we were looking for. In general, it may be difficult to tell the difference between correct and incorrect invariants [15]. Invariant filtering mechanisms, either built into or on top of Texada, may be necessary to make Texada accessible to more users.

### III. TEXADA DESIGN

This section overviews Texada’s design. A more detailed treatment of Texada’s algorithms and design appears in our previous paper [13]. Figure 3 illustrates Texada’s high level operation. Texada works by taking an input log and a property type, and then steps through the process illustrated in Figure 3 to output the instantiations of that property type which are valid over the log. We describe the steps below.

**Parsing the log.** The input execution traces are parsed into an interpretable format (step 1 in Figure 3) using user-supplied regular expressions (see Section IV). The parsed unique event types are passed to the property instance generator.

Texada supports two trace representations: linear and map. The linear representation is the natural one, consisting of an ordered sequence of trace events which must be traversed sequentially. The map representation is a set of maps, each of which represents a trace: the keys are trace events and the values are sets of positions at which the event occurs.

**Parsing property types.** We use the SPOT [5] LTL parser to parse the input LTL property type into a tree structure (step 2 in Figure 3). An example of such an LTL formula tree is shown in Figure 3. Our checking of property instances is based on traversal of the property tree. We use this tree representation to eliminate redundant computation by performing checking state memoization, since two instantiations of the same property type may have nearly identical tree representations.

Our version of LTL is an extension of propositional logic with the following temporal operators6:

- **X(p):** p occurs at the next time point
- **p U q:** p holds up to the first occurrence of q (which must exist)

---

6We also support the W, R, and M operators, which are variations of U.
execution traces
property type
Repeat the checking computation. Memoizing of checking state
variables in the property template, Texada will needlessly
these instantiations may share the assignment of events to
property instance when an invalidating trace is found). Because
property instances on each trace (it stops checking a prop-
now make some notes regarding its performance.

Generally the option to choose if efficiency is desired.

Map optimizations preclude the careful checking necessary to
'skip' over large sections of the trace, for many property
result of checking at the nodes). Because this algorithm can
node in the formula tree holds (instead of just getting the
find-last-occurrence
and
find-first-occurrence

Absolute positions. This algorithm also traverses the LTL for-
matic semantics to evaluate property instances on finite traces.

Creating the space of property instances. The event
types taken from the parsed input traces, along with the LTL
formula tree, are passed into a property instance generator,
which creates the space of property instances (step 3 in Figure
3). The space of property instances is by default generated
on-the-fly, cycling through all possible bindings of events to
property type variables. Pre-generating the instantiations is also
an option. By default, bindings where two unique variables are
bound to the same event are eliminated.

Checking property instances over traces. Finally, prop-
erty types are checked against the input execution traces (step
4 in Figure 3). Checking property instances over the linear
and map representations of a log requires different algorithms.
The checking procedure over the linear trace benefits from the
natural linear definition of LTL operators, making it both
extensible and reliable. The algorithm proceeds by traversing the
LTL formula tree (like the one in Figure 3); at each node,
it applies the formal definition of the nodes to the trace, often
traversing the trace multiple times, which limits its scalability.

In contrast, the map trace algorithm uses the fact that
most LTL operators, apart from the next operator, rely on the
relative positions of events in the trace instead of their
absolute positions. This algorithm also traverses the LTL for-
mula tree, but employs the subroutines find-first-occurrence
and find-last-occurrence to determine whether a high-level
node in the formula tree holds (instead of just getting the
result of checking at the nodes). Because this algorithm can
'skip' over large sections of the trace, for many property
types it is more efficient than the linear checker. However,
the map optimizations preclude the careful checking necessary to
calculate support, support potential, and confidence statistics.
The map checker also involves some memoization and is
generally the option to choose if efficiency is desired.

This concludes the outline of Texada's overall design. We
now make some notes regarding its performance.

Memoizing checking state. Texada checks nearly all
property instances on each trace (it stops checking a property
instance when an invalidating trace is found). Because
these instantiations may share the assignment of events to
variables in the property template, Texada will needlessly
repeat the checking computation. Memoizing of checking state
avoids this by storing the checking results from prior runs
(for a specific trace position and a formula tree node). The
memoization is currently implemented in the map checker’s
find-first-occurrence and find-last-occurrence subroutines.

Runtime performance. Our extensive evaluation [13]
demonstrates that Texada performs favorably against a spe-
cialized property type miner, Synoptic [2]. Texada’s map miner
mines the Synoptic property types faster than Synoptic over all
parameters (increasing trace length, trace number, and number of
unique events). For example, mining the Synoptic property
types on a log of twenty 10,000-event long traces with 980
unique events, Texada’s map miner took 59s compared to
Synoptic’s 69s. We believe that Texada is sufficiently fast to
accommodate a variety of log input sizes and use-cases.

IV. OTHER USAGE OPTIONS

Besides the basic options reviewed in Section II, Texada
includes a number of advanced options that we explain here.

Parsing the input log. In the scenarios we elided the
log format, assuming that the logs are in a format that is
compatible with Texada. However, Texada supports custom
regular expressions (regexes) to extract relevant events from
log lines. Using the -r option, a user can input a list of regular
expressions, each specifying the structure of a matching log
line. The regex arguments require a capturing group with the
name <ETYPE>. The Texada parser (step 1 in Figure 3) attempts
to match each log line with one of these regular expressions,
in order. If a match is found, the string captured by the named
group <ETYPE> becomes the event type of the line7.

Mining temporal properties from a single execution is
brittle, so Texada works best when there are multiple traces
in the log file. The user can specify a custom trace separator
regular expression, using --trace-separator, which partitions
the sequence of lines in a log into traces8.

If the -i or --ignore-mlines option is not specified,
texada will stop and show an error when a line fails to match
the provided regular expressions. With the -i option Texada
will instead ignore non-matching lines.

Texada includes several distinct algorithms – each of which
can mine LTL property instances – which have different
trade-offs. The user must specify either the linear (-l or
-llinear-trace) or the map (-m or --map-trace) checker algo-

7The default regex is (?<ETYPE>.*).
8The default trace separator is --
**Property instance generation.** As discussed in Section III, Texada is configured to generate property instances on the fly. It can be configured to pre-generate property instances with the \texttt{--pregen-instants} option.

By default, Texada will not allow distinct variables in the input formula to be bound to the same event. For example, if the input formula is \( G(x \rightarrow XFy) \), Texada will not check if \( G(a \rightarrow XFb) \) holds on a log. This double-binding can be enabled with the \texttt{--allow-same-bindings} flag.

**Property support and confidence thresholds.** While we used a confidence threshold in the comprehension of stack usage scenario, the topic of support, support potential, and confidence warrants some further detail. Briefly, support potential is the number of time points where the property type could be falsified; support is the number of such time points where the property type is not falsified, and confidence is the ratio of support to support potential.

For example, we expect the support potential of “\( a \) is always followed by \( b' \)” to be the number of \( a \) events, and its support to be the number of \( a \) events which are eventually followed by a \( b \) event. We expect both support and confidence to rise if more \( a \) events which are followed by \( b \) are added to a trace; if \( a \) events not followed by \( b \) are added to a trace, we would like support potential to increase while support stays constant, lowering confidence. Note that in this formulation the addition of \( b \) events or unrelated events has no effect on the confidence of this property instance. The “\( a \) is always followed by \( b' \)” case is the ideal one for support and support potential, and these statistics are currently approximated for other property types, but may not quite reflect our intuition about what they should be. See our previous publication [13] for more details.

These statistics can help a user to reason about property instances that hold over a fraction of the traces, but perhaps not all traces (or not completely on all traces). While these thresholds may help the user analyze imperfect traces, Texada has no mechanism reason about traces or logs beyond determining the satisfiability of property instances. A log that requires data cleaning should be pre-processed prior to being used as input to Texada.

A series of options allow the user to specify thresholds for these statistics for output property instantiations with the linear checker. These thresholds can be set with \texttt{--sup-threshold}, \texttt{--sup-pot-threshold} and \texttt{--conf-threshold}; the default values for these thresholds are 0, 0, and 1 respectively. The \texttt{--no-vacuous-findings} option stops Texada from outputting vacuously true (invalidated but not concretely supported) expressions by setting the support threshold to 1. Note that enabling these options will cause a slowdown in the linear checker. The option \texttt{--use-global-thresholds} sets all the input thresholds as global (over the entire log instead of each individual trace). This option causes even more slowdown since some inter-trace optimizations can no longer be used. Full statistics for each outputted instantiation can be printed with the \texttt{--print-stats} option (though this will disable most optimizations).

**V. RELATED WORK**

Texada is not the only temporal specification mining tool, but is to our knowledge the only one supporting fully general user-specified templates. Van der Aalst et al. developed an almost-fully general LTL checker tool [17], whose logic resembles the Texada linear algorithm’s logic, but can only check one property instance, as opposed to discovering valid property instances from a specified pattern.

Unlike prior work [12], [14], [19], which focuses on mining a few specific temporal patterns, Texada allows users to infer properties matching any pattern expressible in LTL. Since LTL may not be widely known, the Texada tool provides LTL version of the patterns in Dwyer et al.’s work on temporal specifications [6], the Perracotta patterns [19], and the property types used by the Synoptic miner [2], giving a total of 67 included property types.

Several specification miners are based around the response pattern, “\( y \) responds to \( x' \)” or “\( x \) is always followed by \( y' \)” as we have called it previously. Perracotta mines 8 variations of the response patterns out of execution traces, forming larger patterns by chaining the strictest response pattern (i.e., \((xy)^*\)), which the authors call “alternating” [19]. Javert mines both the alternating and resource allocation (i.e., \((xy^+z)^*\)) patterns from dynamic traces, and combines these into even more complex patterns [9]; the authors have also developed an algorithm to mine these based on binary decision diagrams [10].

Researchers have also developed methods to handle imperfect traces by computing, as just one example, property instance interestings scores. The Perracotta tool has thresholds for property interestings; Gabel et al. develop thresholds as part of their BDD-based inference tool [11]; and Lo et al. allow users to specify support and confidence thresholds to determine which mined properties are statistically significant [4].

Texada can mine all the properties these tools mine, and most of their templates are distributed with Texada. Another contrasting feature is that Texada includes algorithms to compute support and confidence measures, which can be used on imperfect traces or to allow the user to input thresholds for statistical significance. In addition, we have found that Texada’s performance compares favorably to the performance of a specialized invariant miner found in Synoptic [2].

**VI. CONCLUSION**

This paper overviewed the Texada tool, which mines arbitrary LTL properties over textual logs regardless of the properties’ form. We presented two usage scenarios to demonstrate how Texada can be used to (1) validate key properties of a concurrent program, and (2) support comprehension of program behavior. We believe that Texada is generally applicable and is especially useful for constructing more advanced analyses tools that require LTL specification mining. For example, we have used Texada to mine temporal properties between data invariants, an unanticipated use-case of the tool. Texada is distributed with 67 pre-defined property types from prior work [2], [6], [19]. Texada is an open source tool and is available at: https://bitbucket.org/bestchai/txeda

**ACKNOWLEDGMENTS**

This project has been funded by an NSERC discovery award, an NSERC USRA award, and the Office of the Privacy Commissioner of Canada (OPC); the views expressed herein are those of the author(s) and do not necessarily reflect those of the OPC.
REFERENCES


