Mining temporal and data-temporal specifications

Ivan Beschastnikh
Caroline Lemieux
Dennis Park

Software Practices Lab
Networks Systems Security Lab
Computer Science
University of British Columbia
Mining temporal and data-temporal specifications

Ivan Beschastnikh
Caroline Lemieux
Dennis Park

Software Practices Lab
Networks Systems Security Lab
Computer Science
University of British Columbia
Program specifications

• Formally describe program behavior: what should happen
  • Data: \( x \leq y \)
  • Temporal: eventually `socket.close` is invoked
  • Interface contracts: preconditions, postconditions, invariants

• Helpful for numerous SE tasks:
  • Bug detection (e.g., model checking, test case generation)
  • Manageability (capture what’s important)
  • Documentation and communication (more concise than code)
Program specifications

• Formally describe program behavior: what should happen
  • Data: $x \leq y$
  • Temporal: eventually `socket.close` is invoked
  • Interface contracts: preconditions, postconditions, invariants

• Helpful for numerous SE tasks:
  • Bug detection (e.g., model checking, test case generation)
  • Manageability (capture what’s important)
  • Documentation and communication (more concise than code)
Challenge with program specifications

• Formally describe program behavior: what should happen

  • Data: \( x \leq y \)

  • Temporal: eventually `socket.close` is invoked

In practice, developers rarely write formal specifications

• Bug detection (e.g., model checking, test case generation)

• Manageability (capture what’s important)

• Documentation and communication (more concise than code)
Absence of program specifications

• Specification inference/mining
  • Program implements some hidden specification
  • Infer this specification using program analyses
Uses of Inferred Specs in Familiar Systems

- program maintenance\[1\]
- confirm expected behavior\[2\]
- bug detection\[2\]
- test generation\[3\]

- system comprehension\[4\]
- system modeling\[4\]
- reverse engineering\[1\]

Inferred Specs in Unfamiliar Systems

- program maintenance\[^{[1]}\]
- confirm expected behavior\[^{[2]}\]
- bug detection\[^{[2]}\]
- test generation\[^{[3]}\]

- system comprehension\[^{[4]}\]
- system modeling\[^{[4]}\]
- reverse engineering\[^{[1]}\]

Absence of program specifications

• Specification inference/mining
  • Program implements some hidden specification
  • Infer this specification using program analyses

• Sources of information
  • Source code
  • Code comments
  • Documentation
  • Test oracles (asserts)
  • Exceptional control flow
  • Dynamic behavior
Absence of program specifications

• Specification inference/mining
  • Program implements some hidden specification
  • Infer this specification using program analyses

• Sources of information
  • Source code
  • Code comments
  • Documentation
  • Test oracles (asserts)
  • Exceptional control flow
  • Dynamic behavior
Inference using dynamic behavior

- **Advantages**
  - Precise
  - Independent of programming language (mostly)
  - Quality depends on data, can always generate more data

- **Disadvantages**
  - Semantic gap: what to capture in a trace?
  - Gap between inferred spec and program code
  - Neither sound nor complete (false positives/negatives possible)
In this talk

• Overview linear temporal logic (LTL)

• **Texada**: a tool to mine general LTL properties

For more details see ASE 2015 paper: “General LTL Specification Mining”, by Lemieux et al.

• Overview Daikon: a data property miner

• **Quarry**: a tool that combines Daikon and Texada to mine data-temporal properties

• Work in progress
Linear temporal logic (LTL)

- LTL formulas assert a condition over time
- Extends propositional logic with temporal operators
  - U: until
  - X: next
  - F: eventually
  - G: always
  - W: weak until
  - R: release
  - M: strong release
Linear temporal logic (LTL)

• LTL formulas assert a condition over time

• Extends propositional logic with temporal operators

Base operators
• U: until
• X: next

Derived operators
• F: eventually
• G: always
• W: weak until
• R: release
• M: strong release
Linear temporal logic (LTL)

- LTL formulas assert a condition over time
- Extends propositional logic with temporal operators

- $U$: until
- $X$: next
- $F$: eventually
- $G$: always
- $W$: weak until
- $R$: release
- $M$: strong release

Used in the talk
Linear temporal logic (LTL)

• LTL formulas assert a condition over time
• Extends propositional logic with temporal operators
  • U: until
    • $\psi = p \ U q$: exists an event where $q$ is true and $p$ is true on all events before first $q$ event
  • X: next
  • F: eventually
  • G: always

✓ trace satisfying $\psi$: $p \ p \ p \ p \ q \ r \ r \ q \ p \ r$

✗ trace violating $\psi$: $p \ p \ p \ r \ q \ r \ r \ q \ p \ r$
Linear temporal logic (LTL)

Two key differences from classic LTL

- Atomic propositions are event strings
- Finite trace semantics

\[ \psi = p \mathcal{U} q : \text{exists an event where } q \text{ is true and } p \text{ is true on all events before first } q \text{ event} \]

- \( X \): next
- \( F \): eventually
- \( G \): always

\[ \checkmark \text{ trace satisfying } \psi : p \ p \ p \ p \ q \ r \ r \ q \ p \ r \]

\[ \times \text{ trace violating } \psi : p \ p \ p \ r \ q \ r \ r \ q \ p \ r \]
Linear temporal logic (LTL)

• LTL formulas assert a condition over time
• Extends propositional logic with temporal operators
  • U: until
  • X: next
    • $\psi = Xp$: the next event is $p$
  • F: eventually
  • G: always

✓ trace satisfying $\psi : p \ q \ r \ r \ q \ p \ r$

✗ trace violating $\psi : r \ q \ r \ r \ q \ p \ r$
Linear temporal logic (LTL)

• LTL formulas assert a condition over time

• Extends propositional logic with temporal operators
  • U: until
  • X: next
  • F: eventually
    • $\psi = F p$ : eventually there is a $p$ event
  • G: always

✓ trace satisfying $\psi : q \ r \ r \ q \ p \ r \ p \ p$

✗ trace violating $\psi : r \ q \ r \ r \ q \ r \ r$
Linear temporal logic (LTL)

• LTL formulas assert a condition over time

• Extends propositional logic with temporal operators
  • U: until
    ☑ trace satisfying $\psi : p \ p \ p \ p \ p \ p \ p \ p$
  • X: next
    ☠ trace violating $\psi : r \ q \ r \ r \ q \ r \ r$
  • F: eventually
  • G: always
    • $\psi = G \ p :$ all events are $p$
Linear temporal logic (LTL)

- LTL formulas assert a condition over time
- Extends propositional logic with temporal operators
  - $U$: until
  - $X$: next
  - $F$: eventually
  - $G$: always

\[ \psi = G(p \rightarrow X F q) : p \text{ is always followed by } q \]
Linear temporal logic (LTL)

- LTL formulas assert a condition over time
- Extends propositional logic with temporal operators
  - $U$: until
  - $X$: next
  - $F$: eventually
  - $G$: always

$$\psi = G(p \rightarrow X F q) : p \text{ is always followed by } q$$

- Eventually you should see a $q$
- Whenever you see a $p$
- Must be valid on entire trace
Linear temporal logic (LTL)

- LTL formulas assert a condition over time
- Extends propositional logic with temporal operators
  - $U$: until
  - $X$: next
  - $F$: eventually
  - $G$: always

$$
\psi = G(p \rightarrow X F q) : p \text{ is always followed by } q
$$

- ✔ trace satisfying $\psi : r s r r p r s q r q$
- ❌ trace violating $\psi : r q r r r p s$
Mining temporal specifications

- Linear LTL checker; finite traces (process mining)  
  van der Aalst et al. LNCS 2005
- Perracotta: 8 templates + chaining  
  Yang et al. ICSE 2006
- Javert: alternating + resource ownership  
  Gabel et al. FSE 2008
- : alternating + resource allocation using BDDs  
  Gabel et al. ICSE 2008
- Response pattern with support/confidence thresholds  
  Lo et al. JSME 2008
- OCD: anomaly detection, Perracotta types  
  Gabel et al. ICSE 2010

Many of these use REs; can be expressed with LTL
Mining temporal specifications

- Perracotta: 8 templates + chaining

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Reg. Ex.</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>y*(xx<em>yy</em>)*</td>
<td>G(x → XF y)</td>
</tr>
<tr>
<td>Alternating</td>
<td>(xy)*</td>
<td>(¬y W x) ∧ G((x → X(¬x U y)) ∧ (y → X(¬y W x)))</td>
</tr>
<tr>
<td>MultiEffect</td>
<td>(xyy*)*</td>
<td>(¬y W x) ∧ G(x → X(¬x U y))</td>
</tr>
<tr>
<td>MultiCause</td>
<td>(xx<em>y)</em></td>
<td>(¬y W x) ∧ G(y → X(¬y W x))</td>
</tr>
<tr>
<td>EffectFirst</td>
<td>y*(xy)*</td>
<td>G((x → X(¬x U y)) ∧ (y → X(¬y W x)))</td>
</tr>
<tr>
<td>CauseFirst</td>
<td>(xx<em>yy</em>)</td>
<td>(¬y W x) ∧ G(x → XF y)</td>
</tr>
<tr>
<td>OneCause</td>
<td>y*(xyy*)*</td>
<td>G(x → X(¬x U y))</td>
</tr>
<tr>
<td>OneEffect</td>
<td>y*(xx<em>y)</em></td>
<td>G(y → X(¬y W x))</td>
</tr>
</tbody>
</table>
Specification patterns taxonomy

- Dwyer et al. ICSE 1999
  formulate “specification patterns” by manually reading many example system specifications

  - Pattern: relation between propositions/events
  - Scope: where the pattern must be true
Specification patterns taxonomy

- Formulate “specification patterns” by manually reading many example system specifications
  - Pattern: relation between propositions/events
  - Scope: where the pattern must be true

Patterns:

- Universality: A given state/event occurs throughout a scope.
- Precedence: A state/event $P$ must always be preceded by a state/event $Q$ within a scope. Figure 1 gives the key elements of the pattern.
- Response: A state/event $P$ must always be followed by a state/event $Q$ within a scope.

Scopes:

- Global
- Before $Q$
- After $Q$
- Between $Q$ and $R$
- After $Q$ until $R$
- State Sequence: $Q$ $R$ $Q$ $Q$ $R$ $Q$
Specification patterns taxonomy

- Formulate “specification patterns” by manually reading many example system specifications.
  - Pattern: relation between propositions/events.
  - Scope: where the pattern must be true.

This taxonomy cannot be captured by prior specification inference tools.
Contribution: Texada

Texada: LTL property miner. Mines LTL properties from a log using an LTL template (a parameterized LTL formula) of arbitrary length and complexity

Texada includes 67 LTL templates

- Specification patterns, Perracotta, etc
- No need to write LTL formulas of your own
- Supersedes prior temporal inference work

- Approximate confidence/support measures for LTL
- Concurrent system analysis (multi-propositional use)
Texada in one slide

**Input:**

<table>
<thead>
<tr>
<th>Log:</th>
<th>Trace 1</th>
<th>Trace 2</th>
<th>Trace 3</th>
<th>Trace 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>login attempt</td>
<td>login attempt</td>
<td>login attempt</td>
<td>login attempt</td>
</tr>
<tr>
<td></td>
<td>auth failed</td>
<td>guest login</td>
<td>auth failed</td>
<td>auth failed</td>
</tr>
<tr>
<td></td>
<td>login attempt</td>
<td>auth failed</td>
<td>login attempt</td>
<td>login attempt</td>
</tr>
<tr>
<td></td>
<td>auth failed</td>
<td>authorized</td>
<td>authorized</td>
<td>guest login</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>authorized</td>
</tr>
</tbody>
</table>

**Property type:** $G(x \rightarrow XF y)$ or “$x$ always followed by $y$”

**Output:**

**Property instances:** $G(\text{guest login} \rightarrow XF \text{ authorized})$
**Texada in one slide**

**Input:**

<table>
<thead>
<tr>
<th>Log:</th>
<th>Trace 1</th>
<th>Trace 2</th>
<th>Trace 3</th>
<th>Trace 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>login attempt auth failed</td>
<td>login attempt guest login auth failed</td>
<td>login attempt auth failed</td>
<td>login attempt auth failed</td>
</tr>
<tr>
<td></td>
<td>login attempt auth failed</td>
<td>guest login authorized</td>
<td>login attempt authorized</td>
<td>login attempt guest login authorized</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Property type:** $G(x \rightarrow XF y)$ or “$x$ always followed by $y$”

**Output:**

**Property instances:** $G(\text{guest login} \rightarrow XF \text{ authorized})$

“guest login” is always followed by “authorized”
# Texada in one slide

<table>
<thead>
<tr>
<th>Input:</th>
<th>Log:</th>
<th>Property type: $G(x \rightarrow XF y)$ or “$x$ always followed by $y$”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trace 1</td>
<td>login attempt auth failed login attempt auth failed</td>
</tr>
<tr>
<td></td>
<td>Trace 2</td>
<td>login attempt guest login auth failed</td>
</tr>
<tr>
<td></td>
<td>Trace 3</td>
<td>login attempt auth failed login attempt authorized</td>
</tr>
<tr>
<td></td>
<td>Trace 4</td>
<td>login attempt auth failed login attempt guest login authorized</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
<th>Property instances: $G(\text{guest login} \rightarrow XF \text{ authorized})$</th>
</tr>
</thead>
</table>

“guest login” is always followed by “authorized”
Texada overview

May 20 16:15:27 my-mac SecurityAgent[130]: Showing Login Window
May 20 16:29:19 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:19 my-mac authorizationhost[129]: Failed to authenticate user <jenny> (tDirStatus: -14090).
May 20 16:29:22 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:22 my-mac SecurityAgent[130]: Login Window Showing Progress

\[ G(x \rightarrow XF y) \]

Property type

Log

Parsing

regular expressions
Texada overview: parsing the log

Log

+ Parsing regular expressions

\[ G(x \rightarrow X F y) \]

Property type
Texada overview: parsing the log

Traces

\[ G(x \rightarrow XFy) \]

Property type
Texada overview: type instantiation

\[ \psi = G(\text{guest login} \rightarrow XF \text{authorized}) \]

\[ x = \text{guest login} \]
\[ y = \text{authorized} \]

Property type
Texada overview: type instantiation

Traces

\[ \phi = G(\text{login attempt} \rightarrow XF \text{authorized}) \]

\[ x = \text{login attempt} \]
\[ y = \text{authorized} \]

Property type
Texada overview: type instantiation

\[ G(x \rightarrow XF y) \]

Property instances

Traces

Property type
Texada overview

\[ \psi = G(\text{guest login} \rightarrow XF \text{authorized}) \]

Property instances

\[ G(x \rightarrow XF y) \]

Property type

Traces
Texada overview: check instances

\[ \psi = G(\text{guest login} \rightarrow XF \text{authorized}) \]

Traces

Satisfies?

Property instances

\[ G(x \rightarrow XF y) \]

Property type
Texada overview: check instances

\[ \psi = G(\text{guest login} \rightarrow XF \text{authorized}) \]

Traces

Property instances

\[ G(x \rightarrow XF y) \]

Property type
Texada overview: check instances

\[ \psi = \mathcal{G}(\text{guest login} \rightarrow X F \text{authorized}) \]

\[ \phi = \mathcal{G}(\text{login attempt} \rightarrow X F \text{authorized}) \]

Traces

Property instances

\[ \mathcal{G}(x \rightarrow X F \ y) \]

Property type
**Texada overview: check instances**

\[ \psi = G(\text{guest login} \rightarrow X F \text{authorized}) \]

\[ \phi = G(\text{login attempt} \rightarrow X F \text{authorized}) \]

Property instances

\[ G(x \rightarrow X F y) \]

Property type

Traces

Satisfies?

Satisfies?

Satisfies?
Texada overview: check instances

\[ \psi = G(\text{guest login} \rightarrow XF\text{authorized}) \]

\[ \phi = G(\text{login attempt} \rightarrow XF\text{authorized}) \]

Traces

Property instances

\[ G(x \rightarrow XF y) \]

Property type
Texada overview

Traces

Property instances

\[ G(x \rightarrow X F y) \]

Property type
Texada overview

Property instances

$G(x \rightarrow XF \ y)$

Property type

Property instances that are true on all input traces

Texada output
Trace representation

- Linear array of events
- Optimized representations
  - Map (event to a list of positions inside a trace)
  - Prefix tree (collapse identical prefixes)
Linear property instance checking

- LTL tree traversal and recursive trace traversal
  \((\neg \text{authorized } U \text{ guest login}) \land G(\text{guest login } \rightarrow XF \text{ authorized})\)
Linear property instance checking

- LTL tree traversal and recursive trace traversal

\[
(\neg \text{authorized} \ U \ \text{guest login}) \land G(\text{guest login} \rightarrow XF \ \text{authorized})
\]
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Evaluate left child, if false stop. Else, return right child result.
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Find first instance of right child, evaluate left child at each position preceding right child.
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Find first instance of right child, evaluate left child at each position preceding right child

If event at current position is “guest login” return true, else false.
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Find first instance of right child, evaluate left child on each event that precedes right child

Return negation of child
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Find first instance of right child, evaluate left child on each event that precedes right child.

If event at current position is "authorized" return true, else false.
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Evaluate left child, if false stop. Else, return right child result
Linear property instance checking

- LTL tree traversal and recursive trace traversal

Traverse all events and check child at each one
Key optimization: checking memoization

- Many property instances have a similar structure

\[ \psi = G(c \land \neg e \rightarrow ((a \rightarrow (\neg e \cup (b \land \neg e)))W e)) \]

\[ \phi = G(d \land \neg e \rightarrow ((a \rightarrow (\neg e \cup (b \land \neg e)))W e)) \]
Key optimization: checking memoization

- Many property instances have a similar structure

\[
\psi = G(c \land \neg e \rightarrow ((a \rightarrow (\neg e \mathcal{U} (b \land \neg e)))) W e))
\]

\[
\phi = G(d \land \neg e \rightarrow ((a \rightarrow (\neg e \mathcal{U} (b \land \neg e)))) W e))
\]
Checking memoization

- Many property instances have a similar structure

\[ \psi = G(c \land \neg e \rightarrow ((a \rightarrow (\neg e \cup (b \land \neg e))) W e)) \]
\[ \phi = G(d \land \neg e \rightarrow ((a \rightarrow (\neg e \cup (b \land \neg e))) W e)) \]
Checking memoization

- Many property instances have a similar structure

\[ \psi = G(c \land \neg e \rightarrow ((a \rightarrow (\neg e U (b \land \neg e))) W e)) \]

\[ \phi = G(d \land \neg e \rightarrow ((a \rightarrow (\neg e U (b \land \neg e))) W e)) \]
Checking memoization

- Many property instances have a similar structure

\[ \psi = G(c \land \neg e \rightarrow T) \]

\[ \phi = G(d \land \neg e \rightarrow T) \]
Checking memoization

• Many property instances have a similar structure

ψ = G(c ∧ ¬e → T)

φ = G(d ∧ ¬e → T)

• Can only re-use results if evaluated at same point in the trace

• Memory vs. compute trade-off
Checking memoization

• Many property instances have a similar structure

\[ \psi = G(c \land \neg e \rightarrow \text{T} \) \]
\[ \phi = G(d \land \neg e \rightarrow \text{T} \) \]

Current strategy:

• Memoize eval result at each <tree node, location in the trace>

• Throw away memoized state after checking one trace against all property instances
Support/confidence computation

• Consider checking G(a) on three traces
  • Trace1: aaaaa
  • Trace2: aaaab
  • Trace3: abbbb
Support/confidence computation

- Consider checking $G(a)$ on three traces
  - Trace1: aaaaa ✓
  - Trace2: aaaab ✗
  - Trace3: abbbb ✗

- But, Trace2 and Trace3 are qualitatively different
- Useful to differentiate these, depending on use-case
  - Anomaly detection, bug finding, ...
- Want to get a handle on log incompleteness (finite log!)
Support/confidence computation

- Consider checking \( G(a) \) on three traces
  - Trace1: aaaaa  ✔
  - Trace2: aaaab  ✗
  - Trace3: abbbb  ✗

- Support of \( G(a) \): number of positions in which ‘a’ appears
- Support potential of \( G(a) \): length of the trace
- Confidence = support / support potential
Support/confidence computation

- Consider checking G(a) on three traces
  - Trace1: aaaaa  ✓  sup: 5  conf: 1.0
  - Trace2: aaaaab  ✗  sup: 4  conf: 0.8
  - Trace3: abbbb  ✗  sup: 1  conf: 0.2

- Support of G(a) : number of positions in which ‘a’ appears
- Support potential of G(a) : length of the trace
- Confidence = support / support potential
Support/confidence computation

- Consider checking \( G(a) \) on three traces
  - Trace1: aaaaa \( \checkmark \) sup: 5 conf: 1.0

Generalizing support/confidence for arbitrary property:

- Support: count locations where instance is true
- Support potential: compute whether a “false” evaluation is possible (depending on trace contents)

- Support potential of \( G(a) \): length of the trace
- Confidence = support / support potential
Texada implementation

• Open source project, in C++
• Uses SPOT lib for parsing LTL property templates
• Includes 67 pre-defined templates (no need to write your own templates!)
  • Dwyer et. al’s patterns (55)
  • Perracotta patterns (8)
  • Synoptic patterns (4)
Texada Evaluation

- Can Texada mine a wide enough variety of temporal properties?
- Can Texada help comprehend unknown systems?
  - Real estate web log
  - StackAr
- Can Texada confirm expected behavior of systems?
  - Dining Philosophers
  - Sleeping Barber
- Is Texada fast?
  - Texada vs. Synoptic (Beschastnikh et al., ESEC/FSE 2011)
  - Texada vs. Perracotta (Yang et al., ICSE 2016)
- Can we use Texada’s results to build other tools?
  - Quarry prototype
Texada Evaluation

• Can Texada mine a wide enough variety of temporal properties?

• Can Texada help comprehend unknown systems?
  – Real estate web log
  – StackAr

• Can Texada confirm expected behavior of systems?
  – Dining Philosophers
  – Sleeping Barber

• Is Texada fast?
  – Texada vs. Synoptic
  – Texada vs. Perracotta

• Can we use Texada’s results to build other tools?
  – Quarry prototype

For more details see ASE 2015 paper: “General LTL Specification Mining”, by Lemieux et al.
Expressiveness of Property Types

- Texada can express properties from prior work

- Synoptic\[1\]

  - Always Followed by
    - RegEx: G(x→XFy)
  - Never Followed by
    - RegEx: G(x→XG!y)
  - Always Precedes
    - RegEx: (!y W x)
  - Alternating
    - RegEx: (xy)* \& G((x→X(!x U y)) \& (y→ X(!y W x)))
  - MultiEffect
    - RegEx: (xyy)* \& G((x→X(!x U y)) \& (y→ X(!y W x)))
  - MultiCause
    - RegEx: (xx*y)* \& G((x→XFy) \& (y→X(!y W x)))
  - EffectFirst
    - RegEx: y*(xy)* \& G((x→X(!x U y)) \& (y→ X(!y W x)))
  - OneCause
    - RegEx: y*(xy)* \& G((x→X(!x U y))
  - CauseFirst
    - RegEx: (xx*yy)* \& G((x→XFy)
  - OneEffect
    - RegEx: y*(xx*y)* \& G((x→XFy) \& (y→X(!y W x)))

- Perracotta\[2\]

- Patterns in Property Specifications for Finite-State Verification
  [Dwyer et al. ICSE’99]

---


Expressiveness of Property Types

- Texada can express properties from prior work
  - Synoptic [1]
  - Perracotta [2]

- Texada can mine a wide variety of properties
- Texada can mine concurrent sys. properties
- Texada has reasonable performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Regex</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always Followed by</td>
<td>G(x→XFy)</td>
<td></td>
</tr>
<tr>
<td>Never Followed by</td>
<td>G(x→XG!y)</td>
<td></td>
</tr>
<tr>
<td>Always Precedes</td>
<td>(!y W x)</td>
<td></td>
</tr>
<tr>
<td>Alternating</td>
<td>(!y W x) &amp; G((x→XFy) &amp; (y→X(!y W x)))</td>
<td></td>
</tr>
<tr>
<td>MultiEffect</td>
<td>(!y W x) &amp; G(x→X(!x U y)) &amp; (y→X(!y W x))</td>
<td></td>
</tr>
<tr>
<td>MultiCause</td>
<td>(xx<em>yy)</em></td>
<td>(!y W x) &amp; G(x→XFy)</td>
</tr>
<tr>
<td>EffectFirst</td>
<td>y*(xx<em>y)</em></td>
<td>G((x→XFy) &amp; (y→X(!y W x)))</td>
</tr>
<tr>
<td>OneCause</td>
<td>y((xyy)*</td>
<td>G(x→X(!! y y)))</td>
</tr>
<tr>
<td>CauseFirst</td>
<td>(xx<em>yy)</em></td>
<td>(!y W x) &amp; G(x→XFy)</td>
</tr>
<tr>
<td>OneEffect</td>
<td>y*(xx<em>y)</em></td>
<td>G((x→XFy) &amp; (y→X(!y W x)))</td>
</tr>
</tbody>
</table>

Patterns in Property Specifications for Finite-State Verification

[Dwyer et al. ICSE’99]

Dining Philosophers

• Classic concurrency problem: philosophers sit around a table, thinking, hungry, or eating.

needs two chopsticks to eat

so this pair can’t eat at the same time

but this pair can eat at the same time

• These specs could not be checked with previous temporal spec miners!
Multi-Propositional Traces

- LTL: multiple atomic propositions may hold at a time
- Standard log model: **one event at each time point**
- Texada supports multi-propositional logs: **multiple events can occur at one time point**
- Dining philosophers log: 5 one minute traces, 6.5K lines

```
0 is THINKING
1 is HUNGRY
2 is THINKING
3 is THINKING
4 is THINKING
...
0 is THINKING
1 is EATING
2 is THINKING
3 is THINKING
4 is THINKING
...
```
Dining Phil. Mutex (safety property)

• Two adjacent philosophers never eat at the same time
• Property pattern: $G(x \rightarrow !y)$ “if $x$ occurs, $y$ does not”

Texada output for $G(x \rightarrow !y)$ includes:

- $G(0 \text{ is EATING } \rightarrow {!}4 \text{ is EATING})$
- $G(0 \text{ is EATING } \rightarrow {!}3 \text{ is EATING})$
- $G(1 \text{ is EATING } \rightarrow {!}2 \text{ is EATING})$
- $G(2 \text{ is EATING } \rightarrow {!}3 \text{ is EATING})$
- $G(3 \text{ is EATING } \rightarrow {!}4 \text{ is EATING})$

Together, mean that two adjacent philosophers never eat at the same time.
Dining Phil. Efficiency (liveness property)

- Non-adjacent philosophers eventually eat at the same time
- Property pattern: $F(x \& y)$ “eventually $x$ and $y$ occur together”

Texada output for $F(x \& y)$ includes:

- $F(2\text{ is EATING} \& 4\text{ is EATING})$
- $F(4\text{ is EATING} \& 2\text{ is EATING})$

Together, mean that non-adjacent philosophers eventually eat at the same time
Dining Phil. Efficiency (liveness property)

- Non-adjacent philosophers eventually eat at the same time
- Property pattern: $F(x \land y)$ “eventually $x$ and $y$ occur together”

- Texada can mine a wide variety of properties
- Texada can mine concurrent sys. properties
- Texada has reasonable performance

$F(0 \text{ is EATING } \land 2 \text{ is EATING})$
$F(0 \text{ is EATING } \land 3 \text{ is EATING})$
$F(1 \text{ is EATING } \land 3 \text{ is EATING})$
$F(1 \text{ is EATING } \land 4 \text{ is EATING})$
$F(2 \text{ is EATING } \land 4 \text{ is EATING})$

Together, mean that non-adjacent philosophers eventually eat at the same time
Texada vs. Synoptic

- Texada performs favourably against Synoptic’s miner on three property types it is *specialized* to mine.

- More results in paper.
- Texada algs benefit from log-level short-circuiting.
Texada vs. Perracotta

- Perracotta performs favourably against Texada:

<table>
<thead>
<tr>
<th>Unique events (10K events/trace, 20 traces/log)</th>
<th>Perracotta</th>
<th>Texada (map miner)</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>0.85 s</td>
<td>2.42 s</td>
</tr>
<tr>
<td>160</td>
<td>0.97 s</td>
<td>4.07 s</td>
</tr>
<tr>
<td>260</td>
<td>1.42 s</td>
<td>10.21 s</td>
</tr>
</tbody>
</table>

- Perracotta’s algorithm particularly effective at reducing instantiation effect on runtime.
- Further memoization work (along with good expiration policies) might help reduce instantiation effect
Texada vs. Perracotta

- Perracotta performs favourably against Texada:
  - Perracotta’s algorithm particularly effective at reducing instantiation effect on runtime.
  - Further memoization work (along with good expiration policies) might help reduce instantiation effect

- Texada can mine a wide variety of properties
- Texada can mine concurrent sys. properties
- Texada has reasonable performance
Texada demo

Project page:
https://bitbucket.org/bestchai/texada

Online tool:
http://bestchai.bitbucket.org/texada/
In this talk

• Overview linear temporal logic (LTL)
• **Texada**: a tool to mine general LTL properties

• Overview Daikon: a data property miner
• **Quarry**: a tool that combines Daikon and Texada to mine data-temporal properties
In this talk

- Overview linear temporal logic (LTL)
- Texada: a tool to mine general LTL properties

- Overview Daikon: a data property miner
- Quarry: a tool that combines Daikon and Texada to mine data-temporal properties
  - Work in progress
Daikon

Concrete program values + program points (control flow)

Tests

Program Source

Data traces

PP1:
\[ x = y - 1 \]

PP2:
\[ x = 1, y = 1 \]

PP1:
\[ x = 0, y = 0 \]

PP2:
\[ x = 0, y = 1 \]

PP1:
\[ x = 1, y = 2 \]

PP2:
\[ x = 0, y = 0 \]

PP1:
\[ x = 2, y = 3 \]

PP2:
\[ x = 1, y = 1 \]

PP1:
\[ x = 0, y = 0 \]

PP2:
\[ x = 0, y = 1 \]

PP1:
\[ x = y - 1 \]

PP2:
\[ x == y \]

Likely data invariants
Daikon applied to a queue

- Likely invariants
  - \( \text{size} \leq \text{capacity} \)
  - isFull one of \{true, false\}

\[
\text{vars} : \{\text{size, capacity, isFull}\}
\]
Ongoing work: mining data-temporal specs

Data invariants (Daikon)

```plaintext
enqueue():enter
size == 0
enqueue():exit
size == 1
enqueue():enter
size == 1
enqueue():exit
size == 2
dequeue():enter
size == 2
dequeue():exit
size == 4
```

Describe data at specific program points

![Diagram showing invariants at exit of enqueue()]

at exit of enqueue(), size >= 1

Temporal invariants (Texada)

```plaintext
create()
enqueue(5)
enqueue(3)
dequeue()
enqueue(7)
enqueue(2)
enqueue(25)
dequeue()
dequeue()
enqueue(8)
enqueue(16)
dequeue()
```

Relate events through time.

enqueue() is always followed by dequeue()
Ongoing work: mining data-temporal specs

Data invariants
(Daikon)

enqueue()::enter
size == 0
enqueue()::exit
size == 1
enqueue()::enter
size == 1
enqueue()::exit
size == 2
dequeue()::enter
size == 2
dequeue()::exit
size == 4

But: data values may interact through time

Temporal invariants
(Texada)

create()
enqueue(5)
enqueue(3)
enqueue(8)
enqueue(16)
dequeue()
enqueue() is always followed by dequeue()

Describe data at specific program points

Relate events through time.
Daikon applied to a queue

- Likely invariants
  - size $\leq$ capacity
  - isFull one of {true, false}
- True over all time: $G(size \leq capacity)$

What if we consider non-global scope?
Daikon applied to a queue

- Likely invariants
  - size <= capacity
  - isFull one of \{true, false\}
- True over all time: \(G(\text{size} \leq \text{capacity})\)

**What if we consider non-global scope?**

- Example:
  - (isFull == false) \(U\) (size == capacity)
Quarry

Concrete program values + program points (control flow)

Program Source → Tests → Data traces → Daikon

PP1:
- x=2, y=3
- x=1, y=2
- x=1, y=2
- x=0, y=0

PP2:
- x=0, y=0
- x=1, y=1
- x=1, y=1
- x=1, y=1

Likely data invariants

PP1:
- x = y - 1, x <= 2
- x==y, x in {0,1}

PP2:
- x = y - 1, x <= 2
- x==y, x in {0,1}
Quarry

Program Source

Tests

Data traces

PP1:
\( x = 2, y = 3 \)

PP2:
\( x = 0, y = 1 \)

PP1:
\( x = 0, y = 1 \)

PP2:
\( x = 0, y = 0 \)

PP1:
\( x = 1, y = 2 \)

PP2:
\( x = 0, y = 0 \)

PP1:
\( x = y - 1, x \leq 2 \)

PP2:
\( x = y, x \in \{0, 1\} \)

Likely data invariants

Multi-propositional invariant traces
Quarry

Program Source

Tests

Data traces

PP1: \( x = y - 1, x \leq 2 \)
PP2: \( x = 0, y = 0 \)
PP1: \( x = 1, y = 1 \)
PP2: \( x = 0, y = 0 \)

Daikon

Likely data invariants

PP1: \( x = y - 1, x \leq 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

Multi-propositional invariant traces

Texada

Data-temporal properties

PP1: \( x = y - 1, x \leq 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 2, y = 3 \)
PP2: \( x = 1, y = 1 \)

PP1: \( x = 0, y = 0 \)
PP2: \( x = 0, y = 0 \)

PP1: \( x = 1, y = 2 \)
PP2: \( x = 0, y = 0 \)
Quarry applied to a queue

- $G(size \leq capacity)$
- $(isFull == false) \mathbf{U} (size == capacity)$
- $G(this.back \leq size(this.theArray[]) - 1)$
  - True with confidence $< 100\%$
  - Either bug, or initialization behavior

- Ongoing work
  - Data invariant semantics for atomic propositions (instead of string semantics)
Challenges in data-temporal spec mining

• Data invariant semantics for atomic propositions
  – Does “size >= 3” always hold on the following trace?

- Current string semantics: no
  size >= 3
  size >= 3
  size == 4
  size >= 3
  are different strings

- Data invariant semantics: yes
  size >= 3
  size >= 3
  size == 4
  size >= 3

• What does it mean for “size >= 3” to be true at a program point where size is not in scope?
Conclusion

Program specifications: important, but often missing

- **Texada**: a tool to mine LTL properties from traces
  - General-purpose, 67 pre-defined LTL property types
  - Fast: 1 million log lines in 3s
- **Quarry**: a tool that combines Daikon and Texada to mine data-temporal properties
  - Work in progress

Open source and ready for use:
https://bitbucket.org/bestchai/texada
Texada evaluation: performance

• Compared performance of Texada against Synoptic’s miner on three property types
  • x always followed by y : $G(x \rightarrow XF \ y)$
  • x never followed by y : $G(x \rightarrow G(\neg y))$
  • x always precedes y : $F \ y \rightarrow (\neg y \ U \ x)$
  • x immediately followed by y : $G(x \rightarrow Xy)$

• Synthetic logs, uniformly randomly distributed events
• Average tool runtime over 5 executions on log input
Eval: vary number of traces

- 10K events/trace, 50 event types
Eval: vary number of traces

- 10K events/trace, 50 event types

Synoptic miner:
- 1 million log lines: 21s
- 2 million log lines: 42s

Texada map miner:
- 1 million log lines: 3s
- 2 million log lines: 6s
Eval: vary trace length

- 20 traces, 100 event types
Eval: vary event types

- 20 traces, 100 events/trace
Texada evaluation: utility

- Run Texada on an anonymized real estate website HTTP access log
  - 12K events, 13 event types
  - Use a subset of the property types from
  - Texada’s runtime < 1s

- Ghezzi et al. ICSE 2014
- Ohmann et al. ASE 2014
- Dwyer et al. ICSE 1999
Texada evaluation: utility

- HTTP access log for a real estate website

Users who visit news article pages eventually visit a sales announcement page.

Users do not visit the search page as they navigate to the homepage from the contacts and news pages.
Support/confidence in LTL mining

- Number of instances mined for “always followed by” template on the HTTP access log, varying global support/confidence thresholds.

<table>
<thead>
<tr>
<th>supp.</th>
<th>conf.</th>
<th>1</th>
<th>0.95</th>
<th>0.9</th>
<th>0.85</th>
<th>0.8</th>
<th>0.7</th>
<th>0.6</th>
<th>0.5</th>
<th>0.3</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>11</td>
<td>120</td>
<td>141</td>
<td>150</td>
<td>165</td>
<td>169</td>
<td>175</td>
<td>182</td>
<td>182</td>
<td>182</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>5</td>
<td>105</td>
<td>122</td>
<td>127</td>
<td>142</td>
<td>145</td>
<td>150</td>
<td>155</td>
<td>155</td>
<td>155</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>2</td>
<td>96</td>
<td>111</td>
<td>116</td>
<td>130</td>
<td>133</td>
<td>138</td>
<td>143</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>5,000</td>
<td></td>
<td>0</td>
<td>87</td>
<td>100</td>
<td>105</td>
<td>118</td>
<td>121</td>
<td>126</td>
<td>130</td>
<td>130</td>
<td>130</td>
</tr>
<tr>
<td>15,000</td>
<td></td>
<td>0</td>
<td>71</td>
<td>78</td>
<td>81</td>
<td>90</td>
<td>93</td>
<td>97</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>50,000</td>
<td></td>
<td>0</td>
<td>47</td>
<td>51</td>
<td>53</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td>0</td>
<td>29</td>
<td>32</td>
<td>33</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>200,000</td>
<td></td>
<td>0</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>