General LTL Specification Mining

Caroline Lemieux, Dennis Park and Ivan Beschastnikh

University of British Columbia
Department of Computer Science

source: https://bitbucket.org/bestchai/texada
Program Specifications

• Formal expectation of how a program should work
• Specs are useful, but **rarely specified by developers**
  – May be difficult to write out
  – May fall out of date like documentation

```
program without specs:
easier for initial dev

```class A{
  foo()
  bar()
  ...  
}
```

```
program with specs:
harder for initial dev

```class A{
  foo()
  bar()
  ...  
}
```
```
foo() always precedes bar() ...
```
```
easier for debugging, refactoring, maintenance
```
```
harder for debugging, refactoring, maintenance
```
Program Specifications

• Formal expectation of how a program should work
• Specs are useful, but **rarely specified by developers**
  – May be difficult to write out
  – May fall out of date like documentation

```
program without specs: easier for initial dev

class A{
  foo()
  bar()
  ...
}

harder for debugging, refactoring, maintenance

program with specs: harder for initial dev

class A{
  foo()
  bar()
  ...
}

foo() always precedes bar()
...

easier for debugging, refactoring, maintenance
```
Program Specifications

- Formal expectation of how a program should work
- Specs are useful, but rarely specified by developers
  - May be difficult to write out
  - May fall out of date like documentation

program without specs:
  easier for initial dev
  harder for debugging, refactoring, maintenance

program with specs:
  harder for initial dev
  easier for debugging, refactoring, maintenance
Program Specifications

- Formal expectation of how a program should work
- Specs are useful, but *rarely specified by developers*
  - May be difficult to write out
  - May fall out of date like documentation

program without specs:
- easier for initial dev
- harder for debugging, refactoring, maintenance

program with specs:
- harder for initial dev
- easier for debugging, refactoring, maintenance

solution: infer specs
Uses of Inferred Specs in Familiar Systems

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

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

\textsuperscript{4} I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE, 267–277, 2011.
Inferred Specs in Unfamiliar Systems

- program maintenance\textsuperscript{[1]}
- confirm expected behavior\textsuperscript{[2]}
- bug detection\textsuperscript{[2]}
- test generation\textsuperscript{[3]}
- system comprehension\textsuperscript{[4]}
- system modeling\textsuperscript{[4]}
- reverse engineering\textsuperscript{[1]}

\textsuperscript{[4]} I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE, 267–277, 2011.
Spec Mining Sources

- Specs can be mined from various program artifacts.
  - Source code [1]
  - Documentation [2]
  - Revision histories [3]

- Focus of talk: textual logs (e.g., execution traces)
  - Easy to instrument, extensible

---

In this talk, focus on mining temporal specs
  - `open()` is always followed by `close()` (response pattern)

Many temporal properties could be mined:

- variations of response pattern [1]
- strict response pattern + resource allocation [2]
- lots of small patterns to combine into big ones [4]
- response patterns of arbitrary length [3]
- branching live-sequence charts [5]
Spec Patterns to Mine

• In this talk, focus on mining temporal specs
  – open() is always followed by close() (response pattern)
• Many temporal properties could be mined:

Which temporal spec mining tool should I use?

- variations of response pattern [1]
- lots of small patterns to combine into big ones [2]
- response patterns of arbitrary length [3]
- branching live-sequence charts [5]

mine any general temporal pattern

- **pattern-based**: can output a set of simple patterns, or more general patterns
- **patterns specified in LTL**, includes 67 pre-defined templates
Contributions

- **Texada**: general LTL specification miner

  textual log + \( \Psi(x,y) \) \rightarrow \text{Texada} \rightarrow \Psi(a,b), \Psi(c,e), \Psi(e,d)

- Approximate confidence/support measures for LTL
- Concurrent system analysis
  - Dining Philosophers
  - Sleeping Barber
Texada Outline

**G(x → XFy)**

```
login attempt
guest login
auth failed
authorized
--
login attempt
auth failed
login attempt
authorized
--
login attempt
auth failed
login attempt
auth failed
--
login attempt
auth failed
login attempt
guest login
authorized
--
```

**Outputs:**

```
G(guest login → XFauthorized)
```

Valid Property Instances

---

**Inputs:**

```
“x is always followed by y”
```

Property Type Mining

- Parse each property type into interpretable format (tree)
- For each property type, dynamically generate and check property instances on log:

\[ G(x \rightarrow XFy) \]

“\( x \) is always followed by \( y \)”

<table>
<thead>
<tr>
<th>Property Instance</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G(\text{authorized} \rightarrow XF\text{guest login}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{authorized} \rightarrow XF\text{login attempt}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{authorized} \rightarrow XF\text{auth failed}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{guest login} \rightarrow XF\text{authorized}) )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>( G(\text{auth failed} \rightarrow XF\text{authorized}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{auth failed} \rightarrow XF\text{guest login}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{auth failed} \rightarrow XF\text{auth failed}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{login attempt} \rightarrow XF\text{authorized}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{login attempt} \rightarrow XF\text{guest login}) )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( G(\text{login attempt} \rightarrow XF\text{auth failed}) )</td>
<td>( \times )</td>
</tr>
</tbody>
</table>
## Linear Log Parsing

Texada parses logs by regexes (specify event line format, trace separator)

<table>
<thead>
<tr>
<th>login attempt</th>
<th>guest login</th>
<th>auth failed</th>
<th>authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>guest login</td>
<td>auth failed</td>
<td>authorized</td>
</tr>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>login attempt</td>
<td>auth failed</td>
<td>login attempt</td>
<td>authorized</td>
</tr>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>login attempt</td>
<td>auth failed</td>
<td>login attempt</td>
<td>auth failed</td>
</tr>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>login attempt</td>
<td>auth failed</td>
<td>login attempt</td>
<td>guest login</td>
</tr>
</tbody>
</table>

set of traces in linear format
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

<table>
<thead>
<tr>
<th></th>
<th>guest login</th>
<th>login attempt</th>
<th>auth failed</th>
<th>authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>guest login</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- G
- X
- F

Nodes:
- G: guest login
- X: guest login
- F: authorized
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace $\iff$ holds on first event of trace
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace \(\iff\) holds on first event of trace

\[G(p):\text{ check if } p \text{ holds at every time point}\]
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- Holds on trace $\iff$ holds on first event of trace

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest login</td>
<td>login attempt</td>
<td>auth failed</td>
<td>authorized</td>
</tr>
</tbody>
</table>

$G(p)$: check if $p$ holds at every time point

$q \rightarrow r$: check if $q \rightarrow r$
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest login</td>
<td>login attempt</td>
<td>auth failed</td>
<td>authorized</td>
</tr>
</tbody>
</table>

G(\(p\)): check if \(p\) holds at every time point

\(q \rightarrow r\): check if \(q \rightarrow r\)

guest login

r
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace $\iff$ holds on first event of trace

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest login</td>
<td>login attempt</td>
<td>auth failed</td>
<td>authorized</td>
</tr>
</tbody>
</table>

$G(p)$: check if $p$ holds at every time point

$q \rightarrow r$: check if $q \rightarrow r$

$X(s)$: check if $s$ holds at next time point
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

\begin{tabular}{c|c|c|c}
 guest login & login attempt & auth failed & authorized \\
\end{tabular}

\begin{align*}
\mathcal{G}(p) & : \text{check if } p \text{ holds at every time point} \\
\mathcal{X}(s) & : \text{check if } s \text{ holds at next time point} \\
\mathcal{F}(a) & : \text{check if } a \text{ holds at some time point}
\end{align*}
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest login</td>
<td>login attempt</td>
<td>auth failed</td>
<td>authorized</td>
</tr>
</tbody>
</table>

$G(p)$: check if $p$ holds at every time point

$q \rightarrow r$: check if $q \rightarrow r$

$X(s)$: check if $s$ holds at next time point

$F(a)$: check if $a$ holds at some time point
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0 1 2 3

| guest login | login attempt | auth failed | authorized |

G(p): check if p holds at every time point
q→r: check if q→r
X(s): check if s holds at next time point
F(a): check if a holds at some time point
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace ⇔ holds on first event of trace

0 1 2 3
| guest login | login attempt | auth failed | authorized |

G(p): check if p holds at every time point

q→r : check if q→r

X(s): check if s holds at next time point
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace $\iff$ holds on first event of trace

\[ G(p): \text{check if } p \text{ holds at every time point} \]

\[ q \rightarrow r : \text{check if } q \rightarrow r \]
Property Instance Checking (Linear Alg)

- Check each instance on each trace in log
- holds on trace $\iff$ holds on first event of trace

$G(p)$: check if $p$ holds at every time point
Linear Algorithm Observations

- Linear checker works but … is slow.
- Notice: most temporal operators rely on relative positions
- Optimization: use **map format**

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0,2]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[1,3]</td>
</tr>
</tbody>
</table>
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

![Diagram]

G → X → F

guest login → F → authorized
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

\[ G(p) \text{ holds at 0 if } !p \text{ never occurs} \]

find first occurrence of \( !p \)
Checking on Map Traces

• Check on trace in map form also tree-based
  – but also uses the negation of nodes
• Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

\[ G(p) \text{ holds at 0 if } \neg p \text{ never occurs} \]

G(p) holds at 0 if \( \neg p \) never occurs
find first occurrence of \( \neg p \)
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

\[ G(p) \text{ holds at } 0 \text{ if } !p \text{ never occurs} \]

- Find first occurrence of \( !p \)
  - Search for first occurrence of guest login (1)
- First occurs at last occurrence of authorized (3)
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

$G(p)$ holds at 0 if $!p$ never occurs
find first occurrence of $!p$

first occurs at last occurrence of authorized (3)

search for first occurrence of guest login (1)

first occ $\geq 3$
Checking on Map Traces

- Check on trace in map form also tree-based
  - but also uses the negation of nodes
- Map form allows algorithm to skip over trace

<table>
<thead>
<tr>
<th>event</th>
<th>posns</th>
</tr>
</thead>
<tbody>
<tr>
<td>login attempt</td>
<td>[0]</td>
</tr>
<tr>
<td>guest login</td>
<td>[1]</td>
</tr>
<tr>
<td>auth failed</td>
<td>[2]</td>
</tr>
<tr>
<td>authorized</td>
<td>[3]</td>
</tr>
</tbody>
</table>

G(p) holds at 0 if $\neg p$ never occurs
find first occurrence of $\neg p$

$\neg p$ never occurs in trace, $G(p)$ holds.
Memoization (reuse of computation)

- To check property type, check each instance on log
  - for $N$ unique events, $M$ variables, $\sim N^M$ instances
  - tree form allows for specialized memoization

- Preliminary memo over 3 instantiations: 7% speedup
Memoization (reuse of computation)

- To check property type, check each instance on log
  - for $N$ unique events, $M$ variables, $\sim N^M$ instances
  - tree form allows for specialized memoization

- Preliminary memo over 3 instantiations: 7% speedup
Support, Confidence for LTL

• Want to know which instances “almost never” violated
• check guest login is always followed by authorized:

login attempt
guest login
auth failed
authorized
guest login
authorized
guest login

only one guest login not followed by authorized – guest login is almost always followed by authorized

• Can we formalize this?
Initial Support, Confidence Concept

- Proposal: support for \( G(p) = \# \) number of time points where \( p \) holds

- But: support for \( G(p \rightarrow XFq) \)

\[
\begin{align*}
\text{qqqq} & \quad \text{qpqq} & \quad \text{pppp} \\
\text{sup } G(p) &= 0 & \text{sup } G(p) &= 1 & \text{sup } G(p) &= 4 \\
\text{pppq} & \quad \text{pqpp} & \quad \text{rrrr} \\
\text{sup } G(p \rightarrow XFq) &= 4 & \text{sup } G(p \rightarrow XFq) &= 2 & \text{sup } G(p \rightarrow XFq) &= 4
\end{align*}
\]
Initial Support, Confidence Concept

- Proposal: support for $G(p) = \# \text{ number of time points where } p \text{ holds}$
  
  \[
  \begin{array}{ccc}
  qqqq & qpqq & pppp \\
  \text{sup } G(p) = 0 & \text{sup } G(p) = 1 & \text{sup } G(p) = 4
  \end{array}
  \]

- But: support for $G(p \rightarrow XFq)$
  
  \[
  \begin{array}{ccc}
  pppq & pqpp & rrrr \\
  \text{sup } G(p \rightarrow XFq) = 4 & \text{sup } G(p \rightarrow XFq) = 2 & \text{sup } G(p \rightarrow XFq) = 4
  \end{array}
  \]
## Initial Support, Confidence Concept

- **Proposal:** support for $G(p) = \# \text{ number of time points where } p \text{ holds}

<table>
<thead>
<tr>
<th>qqqq</th>
<th>qpqq</th>
<th>pPPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sup \ G(p) = 0$</td>
<td>$sup \ G(p) = 1$</td>
<td>$sup \ G(p) = 4$</td>
</tr>
</tbody>
</table>

- **But:** support for $G(p \rightarrow XFq)$

<table>
<thead>
<tr>
<th>pPPP</th>
<th>pqPP</th>
<th>rrrr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sup \ G(p \rightarrow XFq) = 4$</td>
<td>$sup \ G(p \rightarrow XFq) = 2$</td>
<td>$sup \ G(p \rightarrow XFq) = 4$</td>
</tr>
</tbody>
</table>
Initial Support, Confidence Concept

- Proposal: support for $G(p) = \#$ number of time points where $p$ holds

\[
\begin{align*}
\text{qqqq} & \quad \text{qpqq} & \quad \text{pppp} \\
\sup G(p) = 0 & \quad \sup G(p) = 1 & \quad \sup G(p) = 4
\end{align*}
\]

- But: support for $G(p \rightarrowXF q)$

\[
\begin{align*}
\text{pppq} & \quad \text{pqpp} & \quad \text{rrrr} \\
\sup G(p \rightarrowXF q) = 4 & \quad \sup G(p \rightarrowXF q) = 2 & \quad \sup G(p \rightarrowXF q) = 4
\end{align*}
\]
• Proposal: support for $G(p) = \#\text{ number of time points where } p \text{ holds}$

\[
\begin{align*}
\text{qqqqq} & : \sup G(p) = 0 \\
\text{qpqqq} & : \sup G(p) = 1 \\
\text{ppppp} & : \sup G(p) = 4
\end{align*}
\]

• But: support for $G(p \rightarrow XFq)$

\[
\begin{align*}
\text{pppqq} & : \sup G(p \rightarrow XFq) = 4 \\
\text{pqppp} & : \sup G(p \rightarrow XFq) = 2 \\
\text{rrrrr} & : \sup G(p \rightarrow XFq) = 4
\end{align*}
\]
• Proposal: support for $G(p) = \# \text{ number of time points where } p \text{ holds}$

$\begin{align*}
\text{qqqq} & \quad \text{qpqq} & \quad \text{pppp} \\
\text{sup } G(p) &= 0 & \text{sup } G(p) &= 1 & \text{sup } G(p) &= 4
\end{align*}$

• But: support for $G(p \rightarrow XFq)$

$\begin{align*}
\text{pppq} & \quad \text{pqpp} & \quad \text{rrrr} \\
\text{sup } G(p \rightarrow XFq) &= 4 & \text{sup } G(p \rightarrow XFq) &= 2 & \text{sup } G(p \rightarrow XFq) &= 4
\end{align*}$
Initial Support, Confidence Concept

• Proposal: support for $G(p) = \# \text{ number of time points where } p \text{ holds}$

\[
\begin{align*}
\text{qqqq} & \quad \sup G(p) = 0 \\
\text{qpqq} & \quad \sup G(p) = 1 \\
\text{pppp} & \quad \sup G(p) = 4
\end{align*}
\]

• But: support for $G(p \rightarrow XFq)$

\[
\begin{align*}
\text{pppq} & \quad \sup G(p \rightarrow XFq) = 4 \\
\text{pqpp} & \quad \sup G(p \rightarrow XFq) = 2 \\
\text{rrrr} & \quad \sup G(p \rightarrow XFq) = 4
\end{align*}
\]
Support, Confidence Heuristic

• What we do: focus on falsifiability

  guest login → XF authorized vacuously true on

• Call these vacuously true time points not falsifiable

• Approximate support, support potential for arbitrary LTL
  – Support potential of $\Psi$: number of falsifiable time points
  – Support of $\Psi$: number of falsifiable time points on which $\Psi$ is satisfied
  – Confidence of $\Psi$: support/support potential (or 1 if both are 0)
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
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
Expressiveness of Property Types

- Texada can express properties from prior work

<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>(xy)*</td>
<td>(!y W x) &amp; G((x→X(!x U y)) &amp; (y→ X(!y W x)))</td>
</tr>
<tr>
<td>MultiEffect</td>
<td>(xyy)*</td>
<td>(!y W x) &amp; G(x→X(!x y))</td>
</tr>
<tr>
<td>MultiCause</td>
<td>(xx<em>y)</em></td>
<td>(!y W x) &amp; G((x→XFy) &amp; (y→X(!y W x)))</td>
</tr>
<tr>
<td>EffectFirst</td>
<td>y*(xy)*</td>
<td>G((x→X(!x U y)) &amp; (y→ X(!y W x)))</td>
</tr>
<tr>
<td>OneCause</td>
<td>y*(xyy)*</td>
<td>G(x→X(!x U 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>

- Synoptic[1]

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

- 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>OneCause</td>
<td>y (xyy)*</td>
<td>G((x→X(x→y)) &amp; (y→X(!y W x)))</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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
...
```

Multiple events at single time point

```
0 is THINKING
1 is EATING
2 is THINKING
3 is THINKING
4 is THINKING
...
```

Time point separator
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:

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

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”

```
F(2 is EATING & 4 is EATING)
F(4 is EATING & 2 is EATING)
```

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

```
F(0 is EATING & 2 is EATING)
F(0 is EATING & 3 is EATING)
F(1 is EATING & 3 is EATING)
F(1 is EATING & 4 is EATING)
F(2 is EATING & 4 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: $F(x \& y)$ “eventually $x$ and $y$ occur together”

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

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

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

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 on runtime.

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

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

• Many temporal spec miners, unclear which to use
• Texada: general LTL spec miner
  – confirms expected behavior, discovers unexpected use patterns
  – prototyped confidence measures (future work to improve this)
  – can examine concurrent system logs

• Open source and ready to use:
  https://bitbucket.org/bestchai/texada/