Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada
General LTL Specification Mining
Caroline Lemieux, Dennis Park and Ivan Beschastnikh

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  – May fall out of date like documentation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
- 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  – May fall out of date like documentation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{[5]}
- confirm expected behavior\textsuperscript{[2,3]}
- bug detection\textsuperscript{[2]}
- test generation\textsuperscript{[4]}

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

\[\text{familiar system} \quad + \quad \text{inferred specs} \quad \Downarrow \quad \text{unfamiliar system} \quad + \quad \text{inferred specs}\]

\textsuperscript{[3]} G. Ammons, R. Bodik, J. R. Larus. Mining Specifications. POPL 2002
\textsuperscript{[5]} I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE 2011.
Inferred Specs in Unfamiliar Systems

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

\textbullet\ system comprehension\textsuperscript{[4]}
\textbullet\ system modeling\textsuperscript{[4]}
\textbullet\ reverse engineering\textsuperscript{[5]}

\textsuperscript{[3]} G. Ammons, R. Bodik, J. R. Larus. Mining Specifications. POPL 2002
\textsuperscript{[5]} I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE 2011.
Spec Mining Sources

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

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

---

Types of Program Specs

Some tools infer likely data invariants (Daikon)

<table>
<thead>
<tr>
<th>Method</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enqueue()::enter</code></td>
<td><code>size == 0</code></td>
</tr>
<tr>
<td><code>enqueue()::exit</code></td>
<td><code>size == 1</code></td>
</tr>
<tr>
<td><code>enqueue()::enter</code></td>
<td><code>size == 1</code></td>
</tr>
<tr>
<td><code>enqueue()::exit</code></td>
<td><code>size == 2</code></td>
</tr>
<tr>
<td><code>dequeue()::enter</code></td>
<td><code>size == 2</code></td>
</tr>
<tr>
<td><code>dequeue()::exit</code></td>
<td><code>size == 4</code></td>
</tr>
</tbody>
</table>

These invariants describe data at specific program points.

at exit of `enqueue()`, `size >= 1`

Some tools infer likely temporal invariants (Perracotta, Javert, Texada)

<table>
<thead>
<tr>
<th>Method</th>
<th>Event Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create()</code></td>
<td>enqueue (5) enqueue (3) dequeue () enqueue (7) enqueue (2) enqueue (25) dequeue () dequeue () enqueue (8) enqueue (16) dequeue ()</td>
</tr>
</tbody>
</table>

These invariants relate events through execution time.

enqueue () is always followed by dequeue ()
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:
  - strict response pattern + resource allocation [2]
  - lots of small patterns to combine into big ones [4]
  - variations of response pattern [1]
  - 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]
- strict response pattern
- lots of small patterns
- response patterns of arbitrary length [3]
- branching live-sequence charts [5]

“Ultimate” Temporal Spec Inference

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

**Texada**

mine any general temporal pattern
Contributions

- **Texada**: general LTL specification miner

  ![Textual Log](image)

  **Texada**

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

inputs

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

Log

"x is always followed by y"

G(x→XFy)

Valid Property Instances

G(guest login → XFauthorized)

Property Type

High-level process

- 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 \( y \)”
Texada Outline

inputs

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

Texada

Log Parser

Property Instance Checker

Valid Property Instances

output

parsed log

property instances

Log

G(x→XFy)

“x is always followed by y”

SPOT[1]

LTL Parser

Property Instance Generator

Challenges

- Space of instantiations is huge (efficiency)
- Number of true instantiations huge (comprehension)
The interesting technical bits

- Space of instantiations is huge (efficiency)
  - Checking LTL semantics directly (linear alg. slow)
  - Use a mapping data structure/algorithm
  - Memoize checking result of LTL sub-trees

- Number of true instantiations huge (comprehension)
  - General-purpose support and confidence measures for LTL formulae

The interesting technical bits

• Space of instantiations is huge (efficiency)
  – Checking LTL semantics directly (linear alg. slow)
  – Use a mapping data structure/algorithm
  – Memoize checking result of LTL sub-trees

• Number of true instantiations huge (comprehension)
  – General-purpose support and confidence measures for LTL formulae

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
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 program behavior?
  - 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 is a General LTL miner

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

- Synoptic[1]

- Perracotta[2]

- 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 mined 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
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”

G(3 is EATING $\rightarrow$ ! 4 is EATING)
G(4 is EATING $\rightarrow$ ! 3 is EATING)

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

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

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

Together, mean that non-adjacent philosophers eventually eat at the same time.
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
```

Describe data at specific program points

at exit of `enqueue()`, `size >= 1`

Temporal invariants (Texada)

```
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

Temporal invariants (Texada)
- create()
- enqueue(5)
- enqueue(3)
- enqueue(8)
- enqueue(16)
- dequeue()

But: data values may interact through time

Describe data at specific program points

Relate events through time.
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 and size == 4 are different strings

  Data invariant semantics: yes
  - size == 4 is stronger than size >= 3
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

  \[
  \text{size} \geq 3 \text{ and size} = 4 \text{ are different strings}
  \]

  Data invariant semantics: yes

  \[
  \begin{align*}
  \text{size} & \geq 3 \\
  \vdots & \text{..} \\
  \text{size} & = 4 \\
  \vdots & \text{..} \\
  \text{size} & \geq 3 \\
  \vdots &
  \end{align*}
  \]

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

- Many temporal spec miners, unclear which to use
- Texada: general LTL spec miner
  - confirms expected behavior, discovers unexpected use patterns
  - prototyped confidence measures
  - can examine concurrent system logs

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