Ivan Beschastnikh

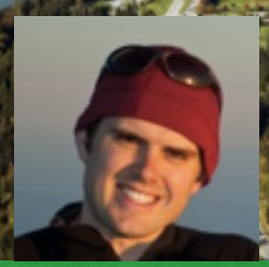
Computer Science
University of British Columbia
Vancouver, Canada

Software Practices

Gail Murphy

Gregor Kiczales

Ron Garcia



Reid Holmes

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

Ivan Beschastnikh

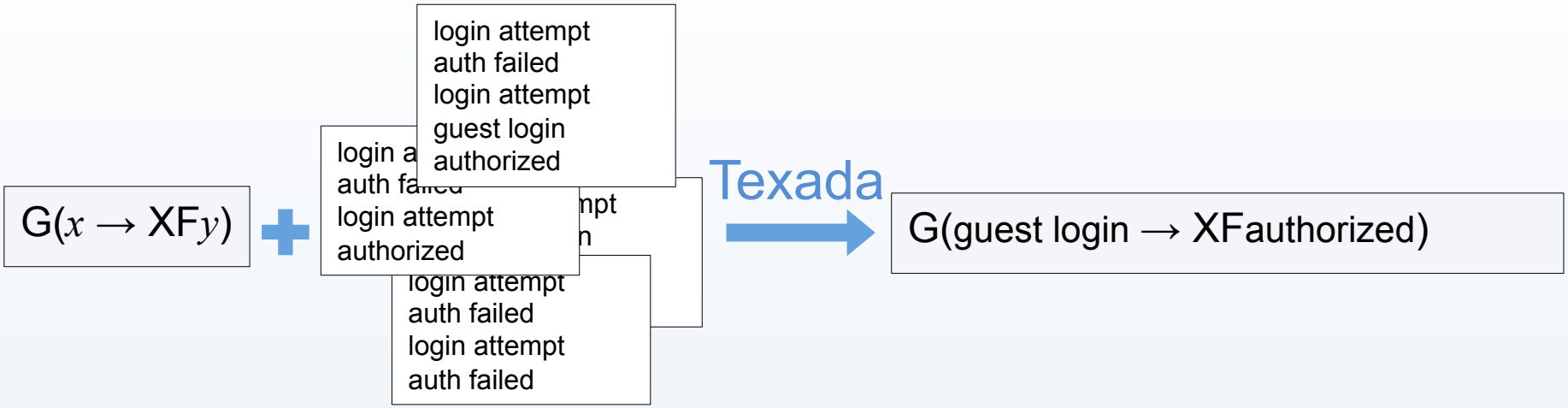
Computer Science
University of British Columbia
Vancouver, Canada

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada



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  - May fall out of date like documentation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

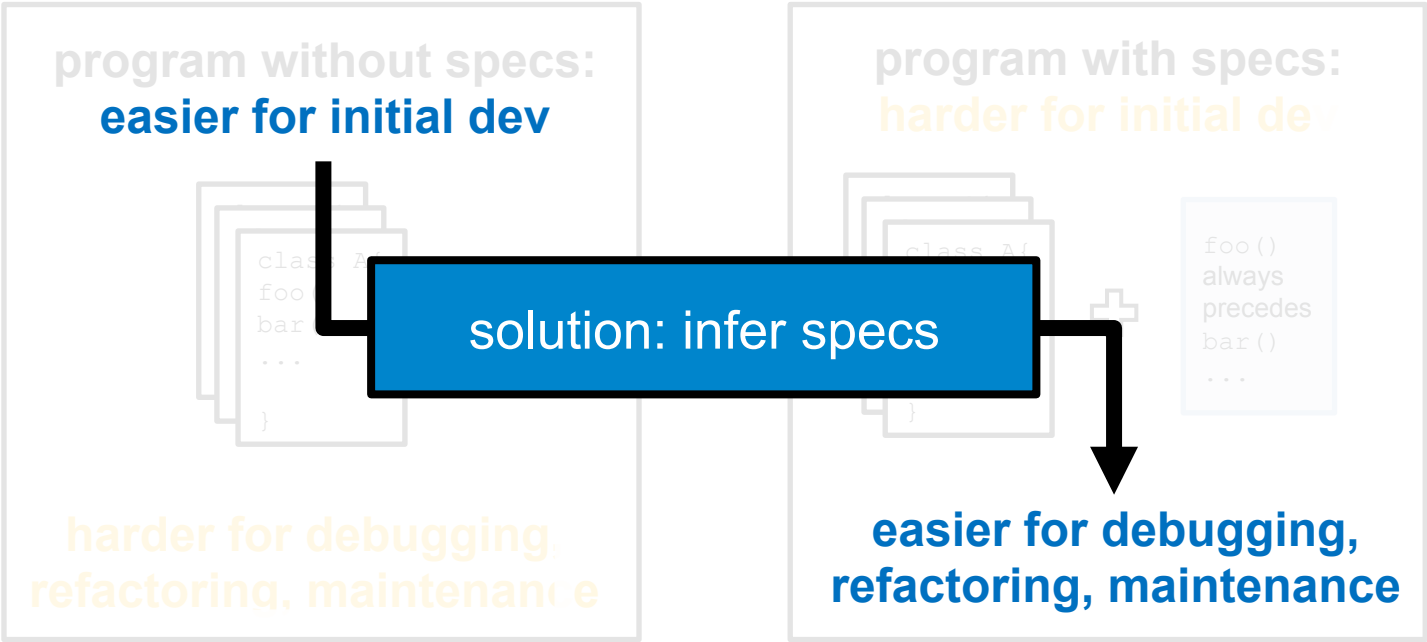
harder for debugging, refactoring, maintenance

program with specs:
harder for initial dev

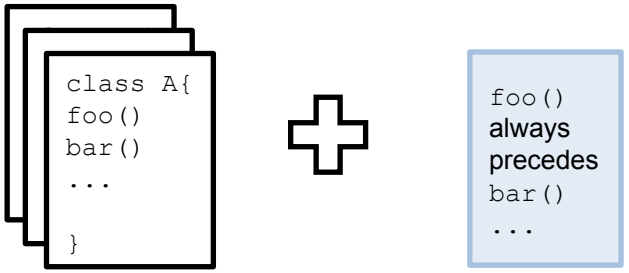
easier for debugging, refactoring, maintenance

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  - May fall out of date like documentation



Uses of Inferred Specs in Familiar Systems



familiar system + **inferred specs**

unfamiliar system + **inferred specs**

- program maintenance[5]
- confirm expected behavior[2,3]
- bug detection[2]
- test generation[4]

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

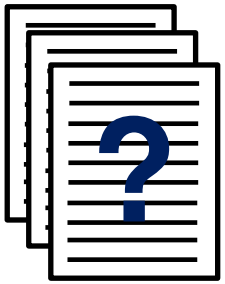
[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. TSE 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE 2001.
[3] G. Ammons, R. Bodik, J. R. Larus. Mining Specifications. POPL 2002
[4] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA 2010.
[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

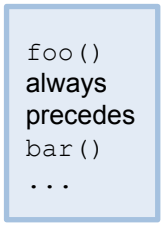
familiar system

inferred specs

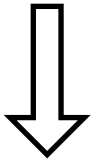
- program maintenance[1]
- confirm expected behavior[2]
- bug detection[2]
- test generation[3]



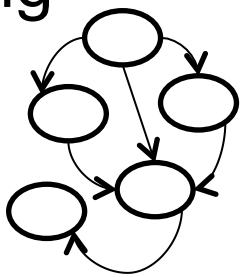
unfamiliar system



inferred specs



- system comprehension[4]
- system modeling[4]
- reverse engineering[5]



[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. TSE 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE 2001.
[3] G. Ammons, R. Bodik, J. R. Larus. Mining Specifications. POPL 2002
[4] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA 2010.
[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

web log

```
sales_page
search
sales_anncs
search
sales_anncs
search
search
sales_anncs
sales_anncs
..
homepage
search
homepage
search
sales_anncs
sales_anncs
homepage
search
```

dining phil.

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

data struct.

```
StackAr(int)
isFull()
isEmpty()
top()
isEmpty()
topAndPop()
isEmpty()
isFull()
isEmpty()
top()
isEmpty()
push(java.lang.Object)
isFull()
isFull()
isEmpty()
top()
isEmpty()
push(java.lang.Object)
```

data inv. log

```
this.currentSize == this.front
this.currentSize == this.back
this.theArray[] elements == null
this.theArray[].getClass() elements == null
this.currentSize == 0
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.back <= size(this.theArray[])-1
..
this.theArray[] elements == null
this.theArray[].getClass() elements == null
this.currentSize == 0
this.front one of { 0, 6 }
```


[1] R. Alur, P. Cerny, P. Madhusudan, W. Nam. Synthesis of Interface Specifications for Java Classes. In Proceedings of POPL'05.
[2] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*Icomment: Bugs or BadComments?*/. In Proceedings of SOSP'07.
[3] V. B. Livshits and T. Zimmermann. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. In Proceedings of ESEC/FSE'05.

Types of Program Specs

Some tools infer likely 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
```

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

These invariants describe data at specific program points.

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

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

⇨ **`enqueue()` is always followed by `dequeue()`**

These invariants relate events through execution time.

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]

[1] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE'06.
[2] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. FSE'08.
[3] D. Lo, S-C. Khoo, and C. Liu. Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 20 (4), 2008.
[4] G. Reger, H. Barringer, and D. Rydeheard. A Pattern-Based Approach to Parametric Specification Mining. In Proceedings of ASE'13.
[5] D. Fahland, D. Lo, and S. Maoz. Mining Branching-Time Scenarios. In Proceedings of ASE'13.

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

lots of small

Which temporal spec mining tool should I use?

variations of response pattern [1]

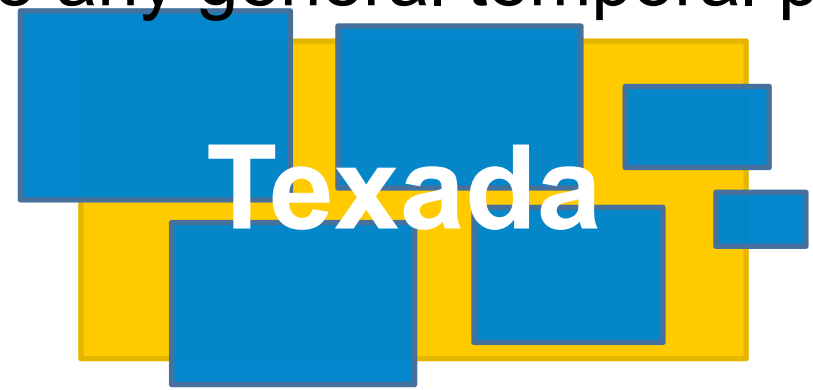
response patterns of arbitrary length [3]

branching live-sequence charts [5]

[1] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE'06.
[2] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. FSE'08.
[3] D. Lo, S-C. Khoo, and C. Liu. Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 20 (4), 2008.
[4] G. Reger, H. Barringer, and D. Rydeheard. A Pattern-Based Approach to Parametric Specification Mining. In Proceedings of ASE'13.
[5] D. Fahland, D. Lo, and S. Maoz. Mining Branching-Time Scenarios. In Proceedings of ASE'13.

“Ultimate” Temporal Spec Inference

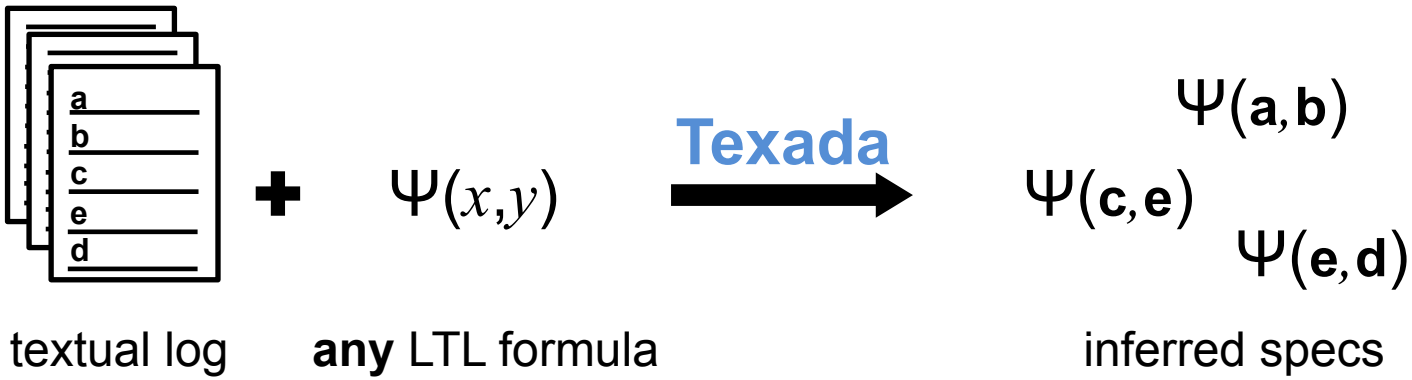
mine any general temporal pattern



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

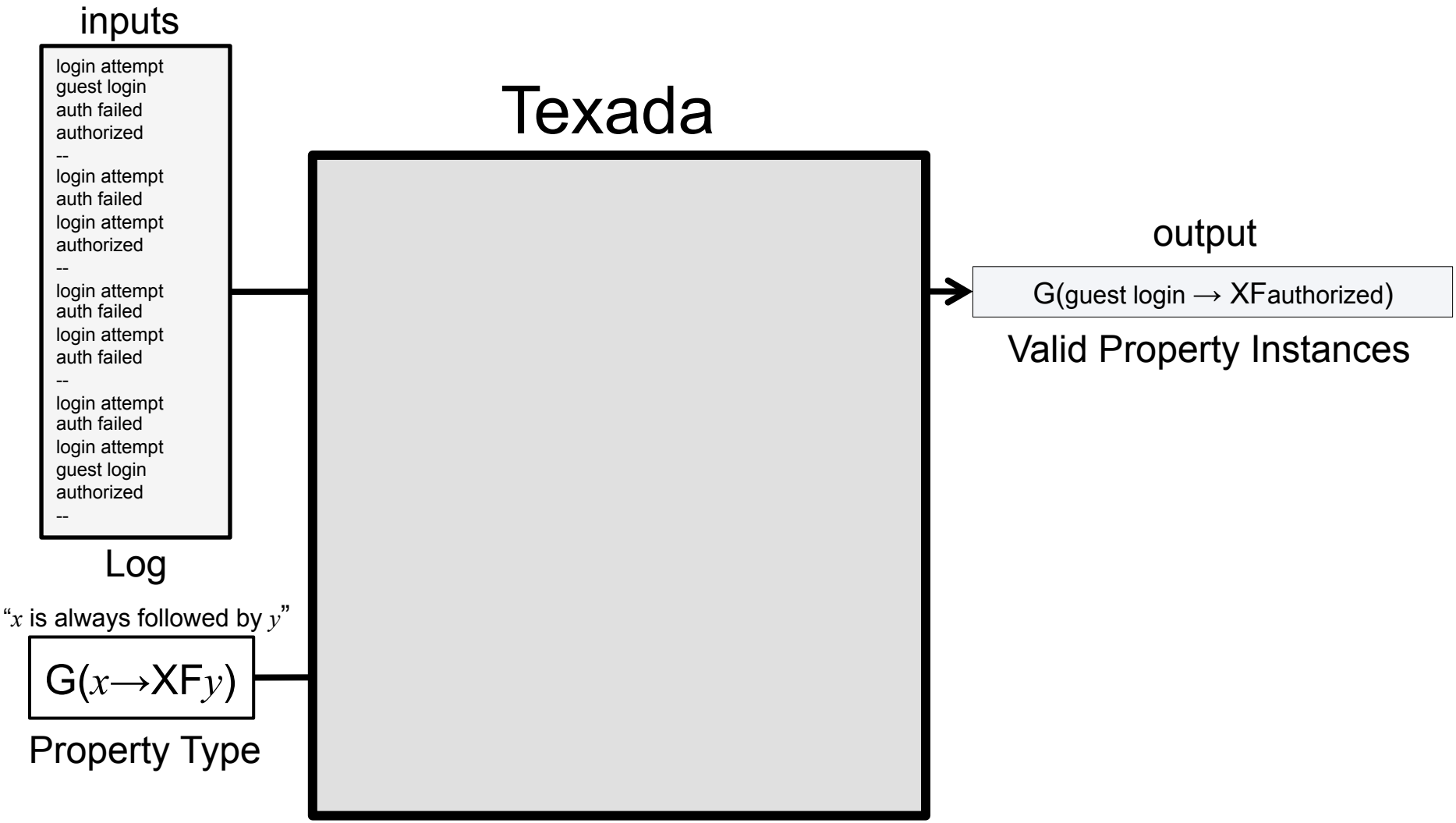
Contributions

- **Texada**: general LTL specification miner



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

Texada Outline



[1] A. Duret-Lutz and D. Poitrenaud. Spot: an Extensible Model Checking Library using Transition-Based Generalized Buchi automata. In Proceedings of MASCOTS'04.

High-level process

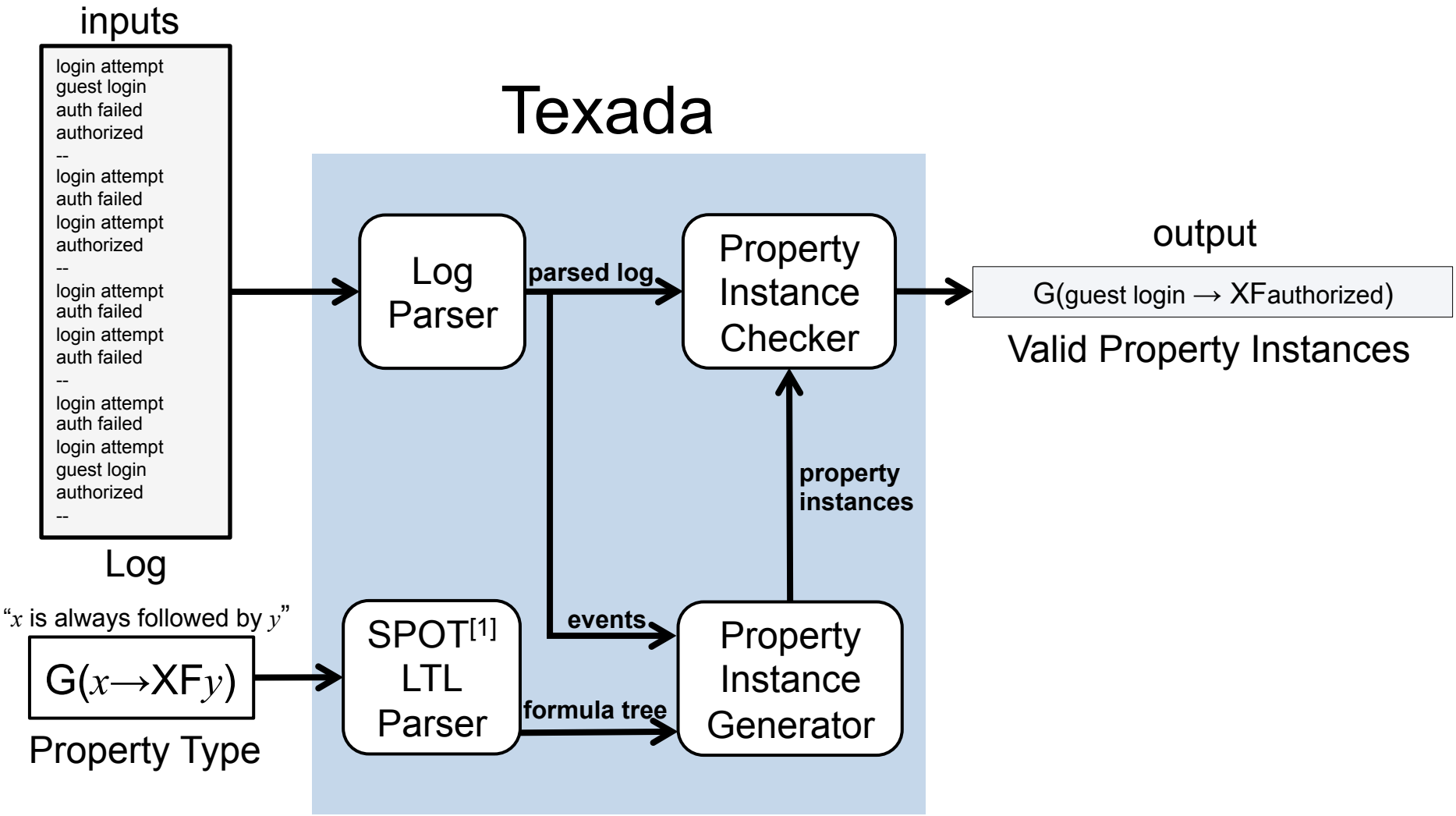
- Parse each property type into interpretable format (tree)
- For each property type, dynamically generate and check property instances on log:

"x is always followed by y"

$$G(x \rightarrow XFy)$$

G(authorized → XFguest login)	✗
G(authorized → XFlogin attempt)	✗
G(authorized → XFauth failed)	✗
G(guest login → XFauthorized)	✓
G(auth failed→ XFauthorized)	✗
G(auth failed→ XFguest login)	✗
G(auth failed → XFauthorized)	✗
G(guest login→ XFlogin attempt)	✗
G(guest login→ XFauth failed)	✗
G(login attempt → XFauthorized)	✗
G(login attempt → XFguest login)	✗
G(login attempt → XFauth failed)	✗

Texada Outline



[1] A. Duret-Lutz and D. Poitrenaud. Spot: an Extensible Model Checking Library using Transition-Based Generalized Buchi automata. In Proceedings of MASCOTS'04.

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

[1] C,Lemieux, D. Park and I. Beschastnikh. General LTL Specification Mining. ASE 2015

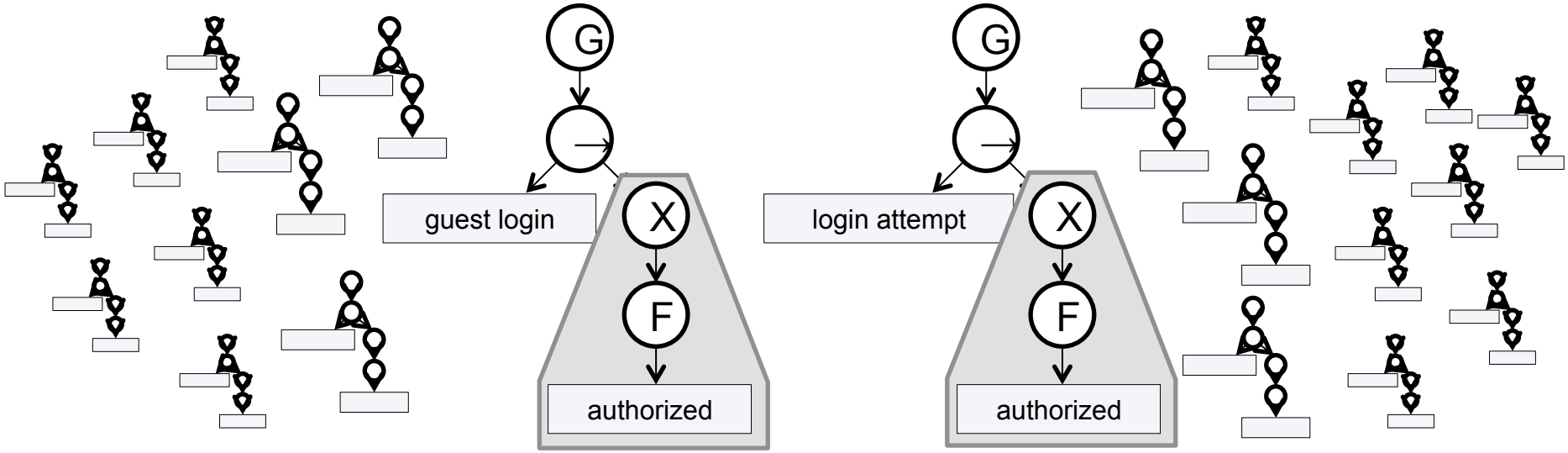
The interesting technical bits

- 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

[1] C,Lemieux, D. Park and I. Beschastnikh. General LTL Specification Mining. ASE 2015

Memoization (reuse of computation)

- To check property type, check each instance on log
  - for N unique events, M variables, ~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

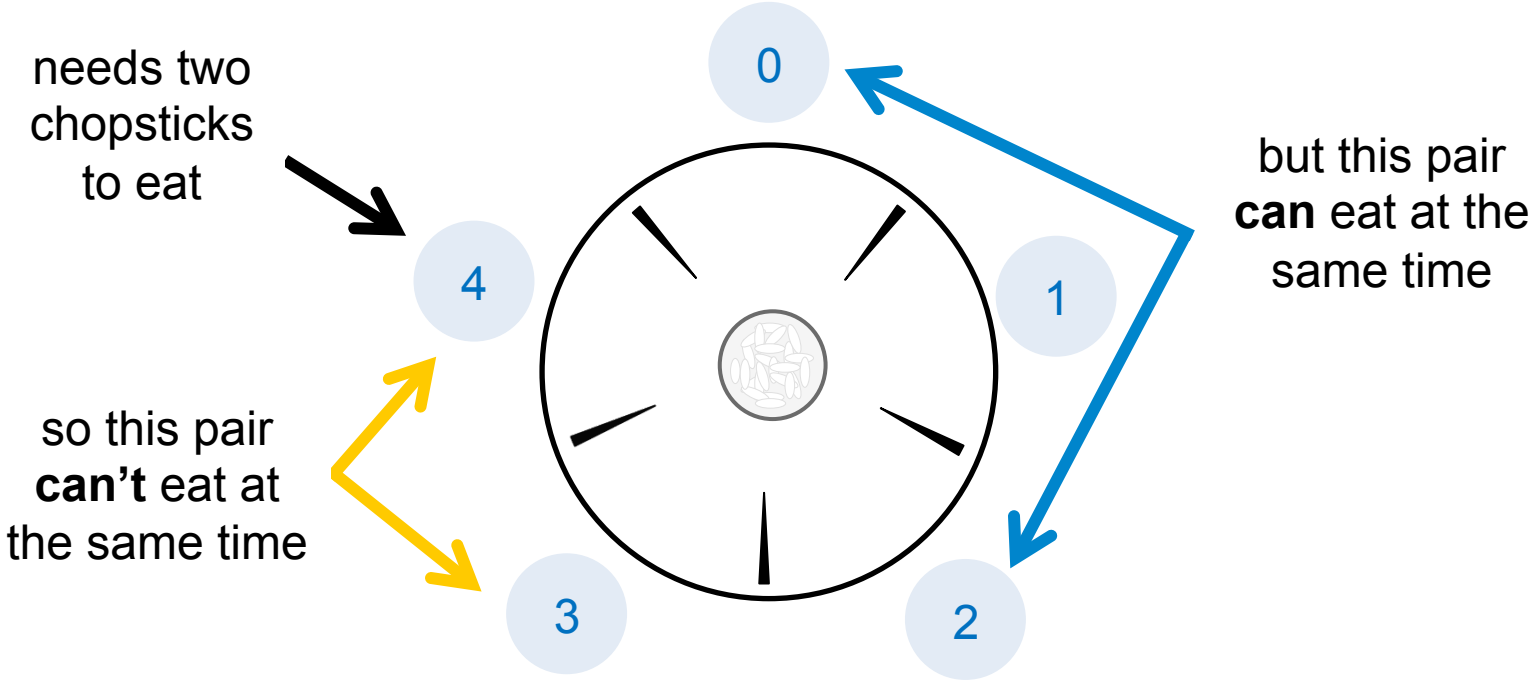
	Name	Regex	LTL
– Synoptic[1]	Always Followed by		G(x→XFy)
	Never Followed by		G(x→XG!y)
	Always Precedes		(!y W x)
– Perracotta[2]	Alternating	(xy)*	(!y W x) & G((x→X(!x U y)) & (y→ X(!y W x)))
	MultiEffect	(xyy*)*	(!y W x) & G(x→X(!x U y))
	MultiCause	(xx*y)*	(!y W x) & G((x→XFy) & (y→X(!y W x)))
	EffectFirst	y*(xy)*	G((x→X(!x U y)) & (y→ X(!y W x)))
	OneCause	y*(xyy*)*	G(x→X(!x U y))
	CauseFirst	(xx*yy*)*	(!y W x) & G(x→XFy)
	OneEffect	y*(xx*y)*	G((x→XFy) & (y→X(!y W x)))

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

[1] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. FSE11.
[2] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. ICSE06.

Dining Philosophers

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



- 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

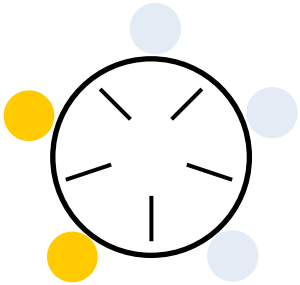
multiple events at single time point

time point separator

```
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"



G(3 is EATING → ! 4 is EATING)

⬍

G(4 is EATING → ! 3 is EATING)

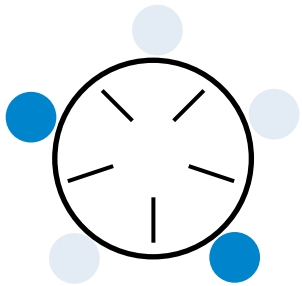
- Texada output for G($x \rightarrow !y$) includes

G(0 is EATING → ! 1 is EATING)
G(0 is EATING → ! 4 is EATING)
G(1 is EATING → ! 2 is EATING)
G(2 is EATING → ! 3 is EATING)
G(3 is EATING → ! 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"



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

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
```

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

Describe data at specific program points

Temporal invariants
(Texada)

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

⇨ **`enqueue()` is always followed by `dequeue()`**

Relate events through 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

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()

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**

```
size >= 3
..
size >= 3
..
size == 4
..
size >= 3
..
```

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

`size >= 3` **and** `size == 4` **are different strings**

```
size >= 3
..
size >= 3
..
size == 4
..
size >= 3
..
```

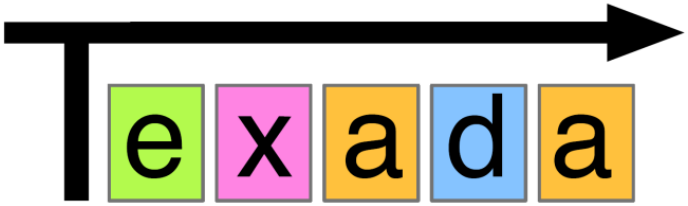
Data invariant semantics: yes

`size == 4` **is stronger than** `size >= 3`

- 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/

[1] C. Lemieux, D. Park and I. Beschastnikh. General LTL Specification Mining. ASE 2015