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

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

Software Practices

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

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: $x \le y$
  - 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

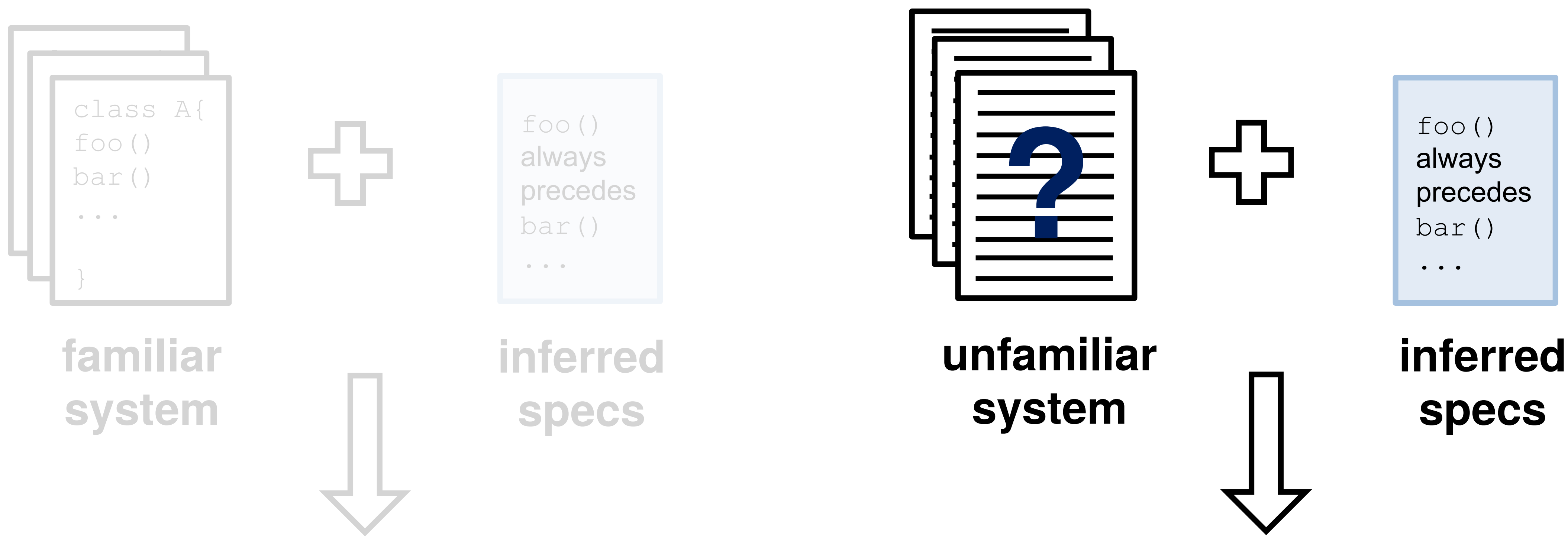
Uses of Inferred Specs in Familiar Systems

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

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

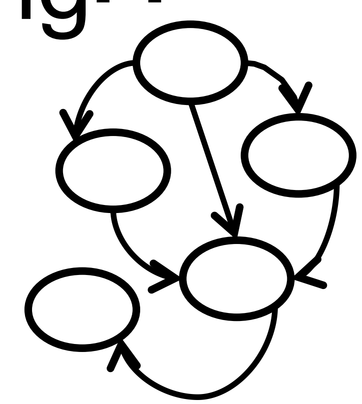
[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. TSE, 613-637, 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE, 27(2):99–123, 2001.
[3] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA, 85-96, 2010.
[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[1]
- confirm expected behavior[2]
- bug detection[2]
- test generation[3]

- 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, 613-637, 2013.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support program evolution. TSE, 27(2):99–123, 2001.
[3] V Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. Generating Test Cases for Specification Mining. ISSTA, 85-96, 2010.
[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.

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
  - U: until
  - X: next

Base operators

  - F: eventually
  - G: always
  - W: weak until
  - R: release
  - M: strong release

Derived operators

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

Used in the talk

  - 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
    - ψ = 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 ψ : p p p p q r r q p r →

✗ trace violating ψ: 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

- ψ = 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 ψ : p p p p q r r q p r

✗ trace violating ψ: 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
    - ψ = X p : the next event is p
  - F: eventually
  - G: always

✓ trace satisfying ψ : p q r r q p r →

✗ trace violating ψ: 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
    - ψ = F p : eventually there is a p event
  - G: always

✓ trace satisfying ψ : q r r q p r p p

✗ trace violating ψ: 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
  - X: next
  - F: eventually
  - G: always
    - ψ = G p : all events are p

✓ trace satisfying ψ : p p p p p p p

✗ trace violating ψ: 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
  - X: next
  - F: eventually
  - G: always

ψ = G(p $\rightarrow$ X F q) : p 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

ψ = G(p $\rightarrow$ X F q) : p 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
  - F: eventually
  - X: next
  - G: always

ψ = G(p $\rightarrow$ X F q) : p is always followed by q

✓ trace satisfying ψ : r s r r p r s q r q

✗ trace violating ψ: 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
- Gabel et al. ICSE 2008: alternating + resource allocation using BDDs
- 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 Yang et al. ICSE 2006

Pattern	Reg. Ex.	LTL
Response	y*(xx*yy*)*	$G(x \rightarrow XF\,y)$
Alternating	(xy)*	$(\neg y\ W\ x) \wedge G((x \rightarrow X(\neg x\ U\ y)) \wedge (y \rightarrow X(\neg y\ W\ x)))$
MultiEffect	(xyy*)*	$(\neg y\ W\ x) \wedge G(x \rightarrow X(\neg x\ U\ y))$
MultiCause	(xx*y)*	$(\neg y\ W\ x) \wedge G(y \rightarrow X(\neg y\ W\ x))$
EffectFirst	y*(xy)*	$G((x \rightarrow X(\neg x\ U\ y)) \wedge (y \rightarrow X(\neg y\ W\ x)))$
CauseFirst	(xx*yy*)	$(\neg y\ W\ x) \wedge G(x \rightarrow XF\,y)$
OneCause	y*(xyy*)*	$G(x \rightarrow X(\neg x\ U\ y))$
OneEffect	y*(xx*y)*	$G(y \rightarrow X(\neg y\ W\ x))$

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

- 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

Patterns:

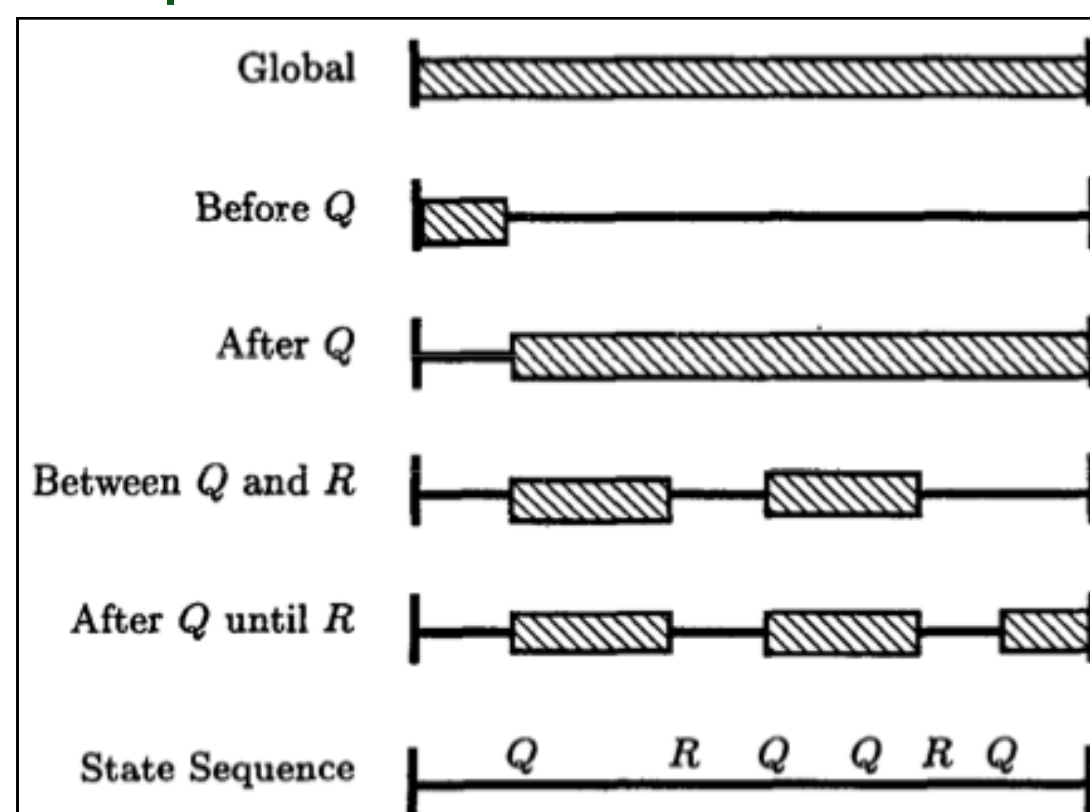
Universality A given state/event occurs throughout a scope.

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.

X

Scopes:



Specification patterns taxonomy

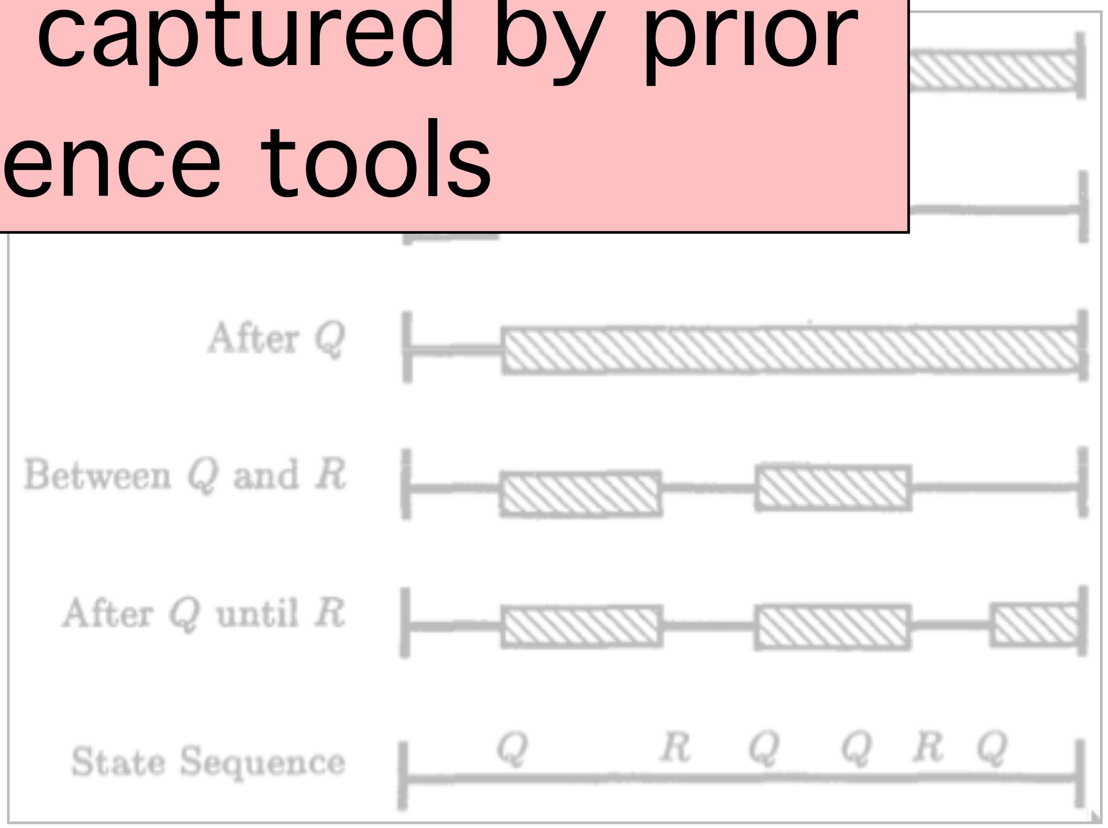
- Dwyer et al. ICSE 1999 formulate "specification patterns" by manually reading many example system specifications
  - Pattern: relation between propositions/events
  - Scope: where the pattern must be true

This taxonomy cannot be captured by prior specification inference tools

Univers... scope.

Precede... 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.



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:

Log:

Trace 1	Trace 2	Trace 3	Trace 4
login attempt auth failed login attempt auth failed	login attempt guest login auth failed authorized	login attempt auth failed login attempt authorized	login attempt auth failed login attempt guest login authorized

Property type: $G(x \to XF\,y)$ or "x always followed by y"

Texada

Output: **Property instances:** $G(\texttt{guest login} \to XF\,\texttt{authorized})$

Texada in one slide

Input:

Log:

Trace 1	Trace 2	Trace 3	Trace 4
login attempt auth failed login attempt auth failed	login attempt guest login auth failed authorized	login attempt auth failed login attempt authorized	login attempt auth failed login attempt guest login authorized

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

Texada

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

"guest login" is always followed by "authorized"

Texada in one slide

Input:

Log:

Trace 1	Trace 2	Trace 3	Trace 4
login attempt auth failed login attempt auth failed	login attempt guest login auth failed authorized	login attempt auth failed login attempt authorized	login attempt auth failed login attempt guest login authorized

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

Texada

Output:

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

"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
....
```

Log

+

Parsing
regular expressions

$$G(x \rightarrow XF\ y)$$

Property type

Texada overview: parsing the log

$$G(x \rightarrow XF\ y)$$

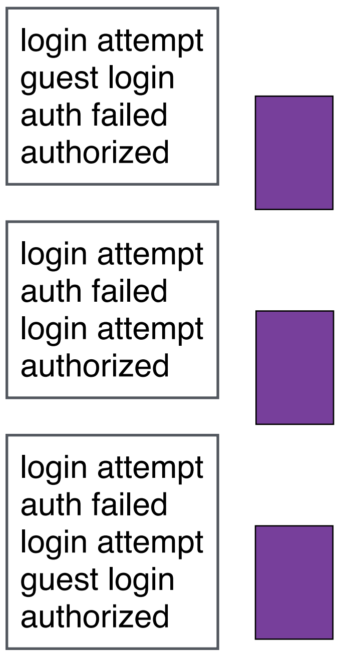
Property type

Log

+

Parsing
regular expressions

Texada overview: parsing the log

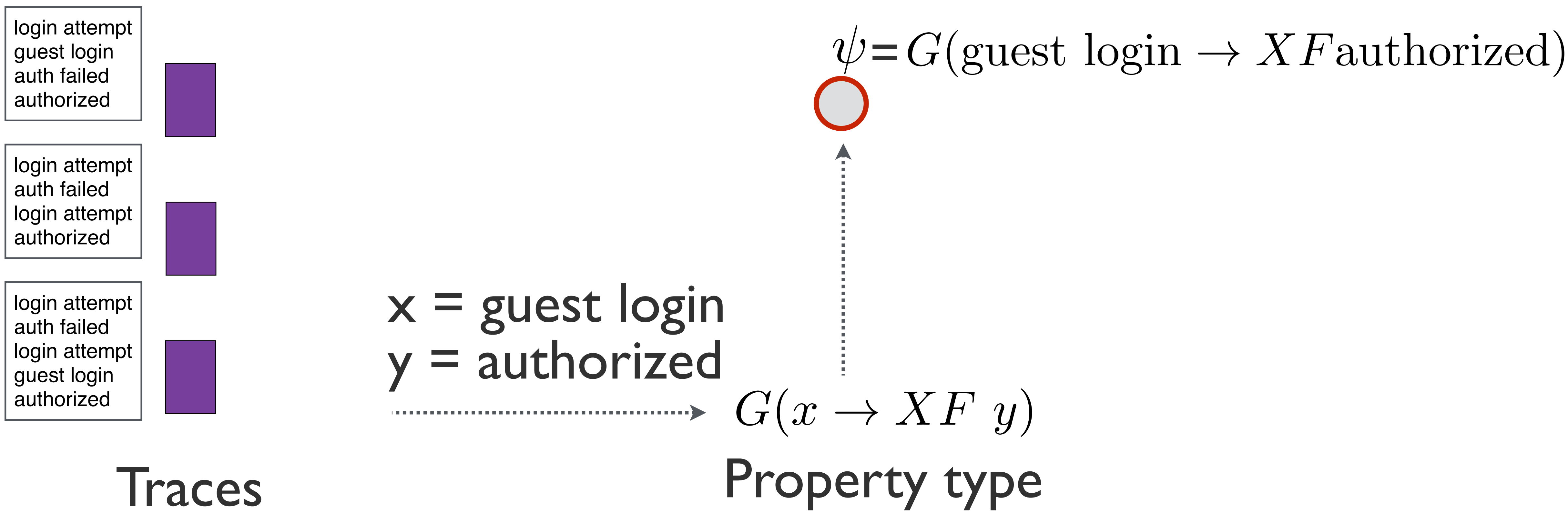


Traces

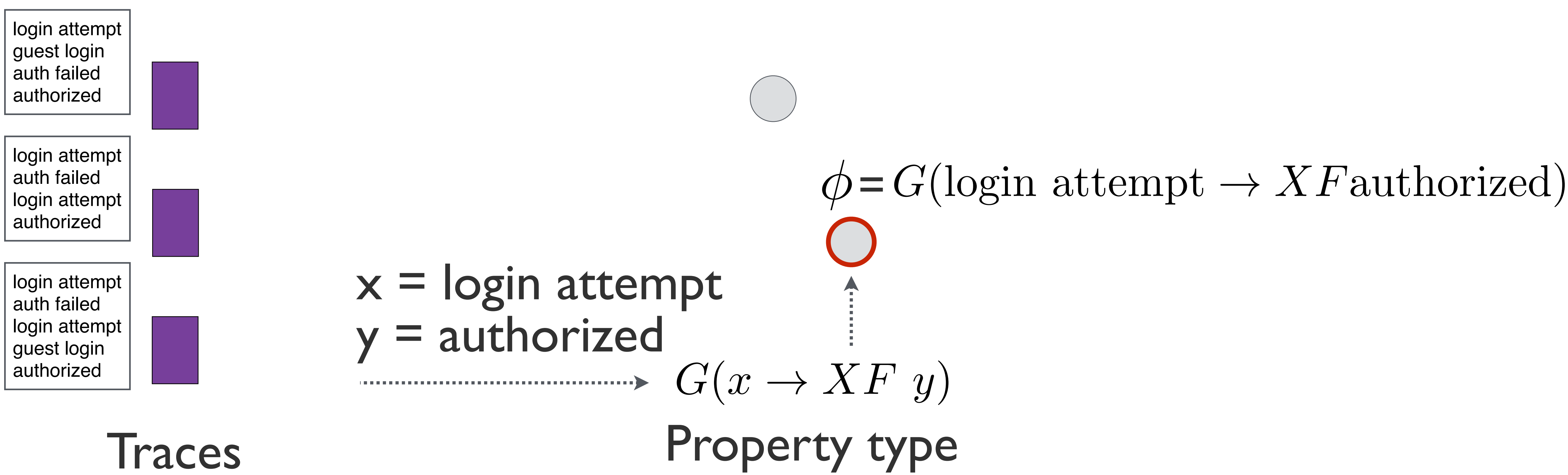
$G(x \rightarrow XF\ y)$

Property type

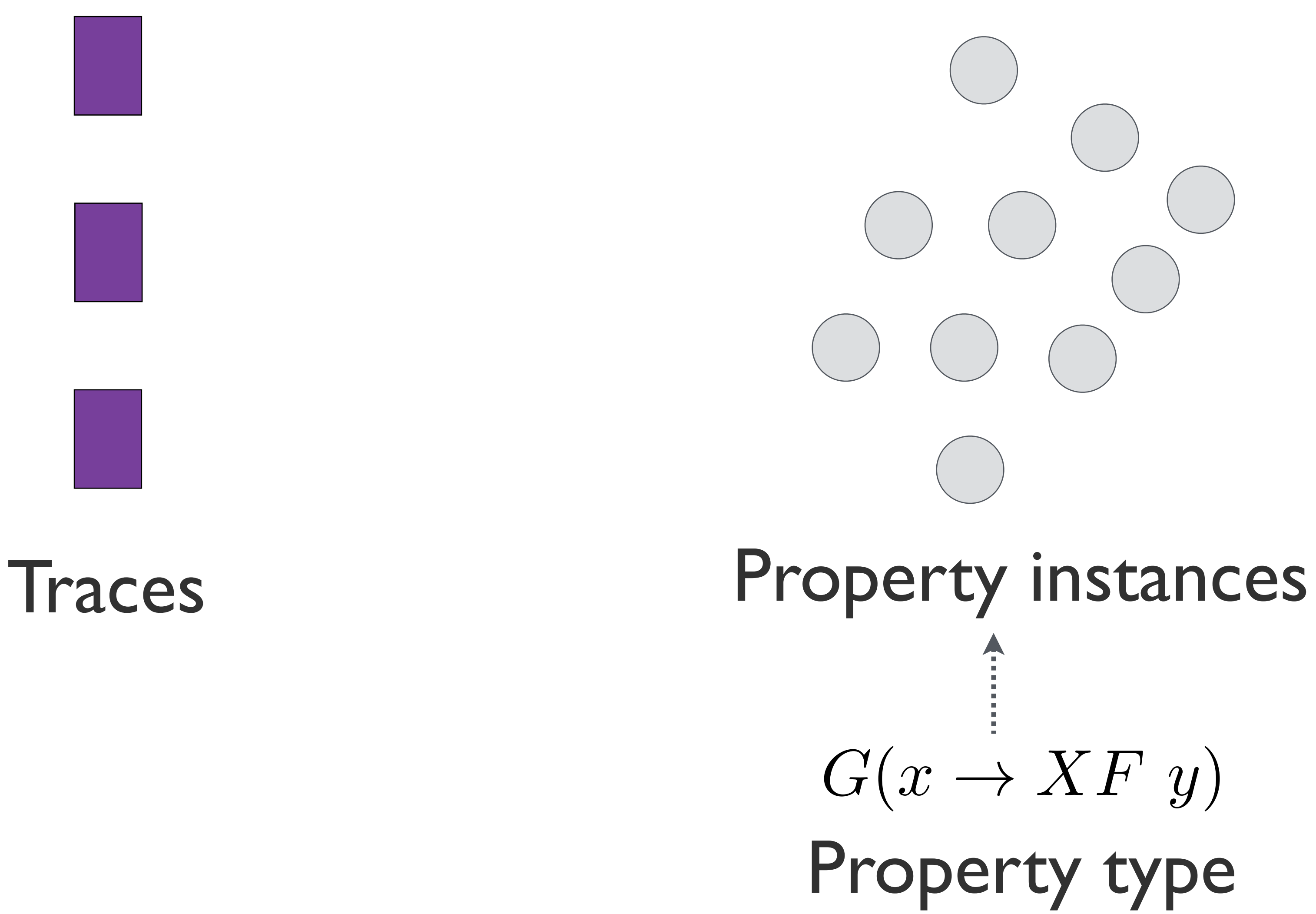
Texada overview: type instantiation



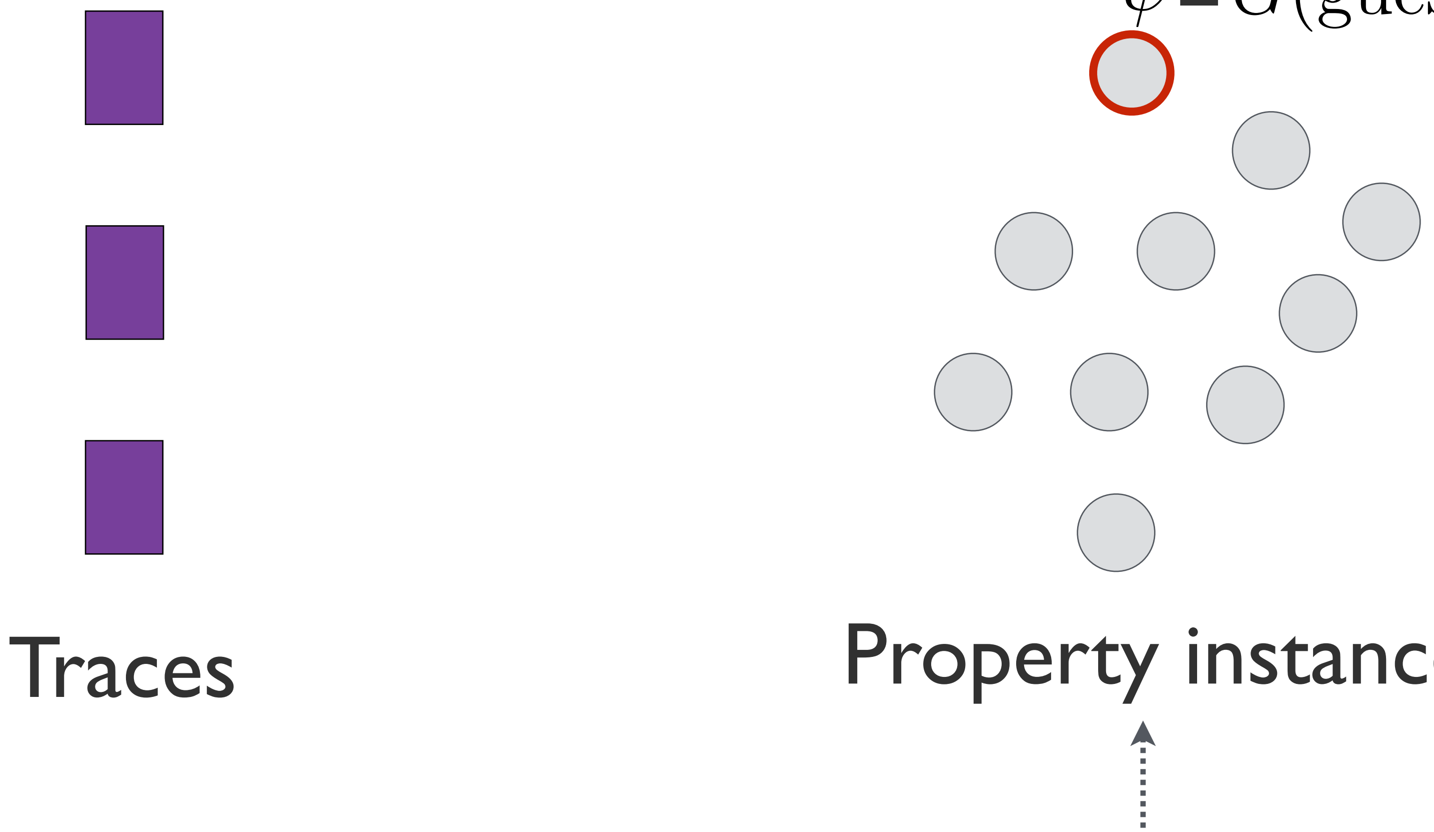
Texada overview: type instantiation



Texada overview: type instantiation



Texada overview



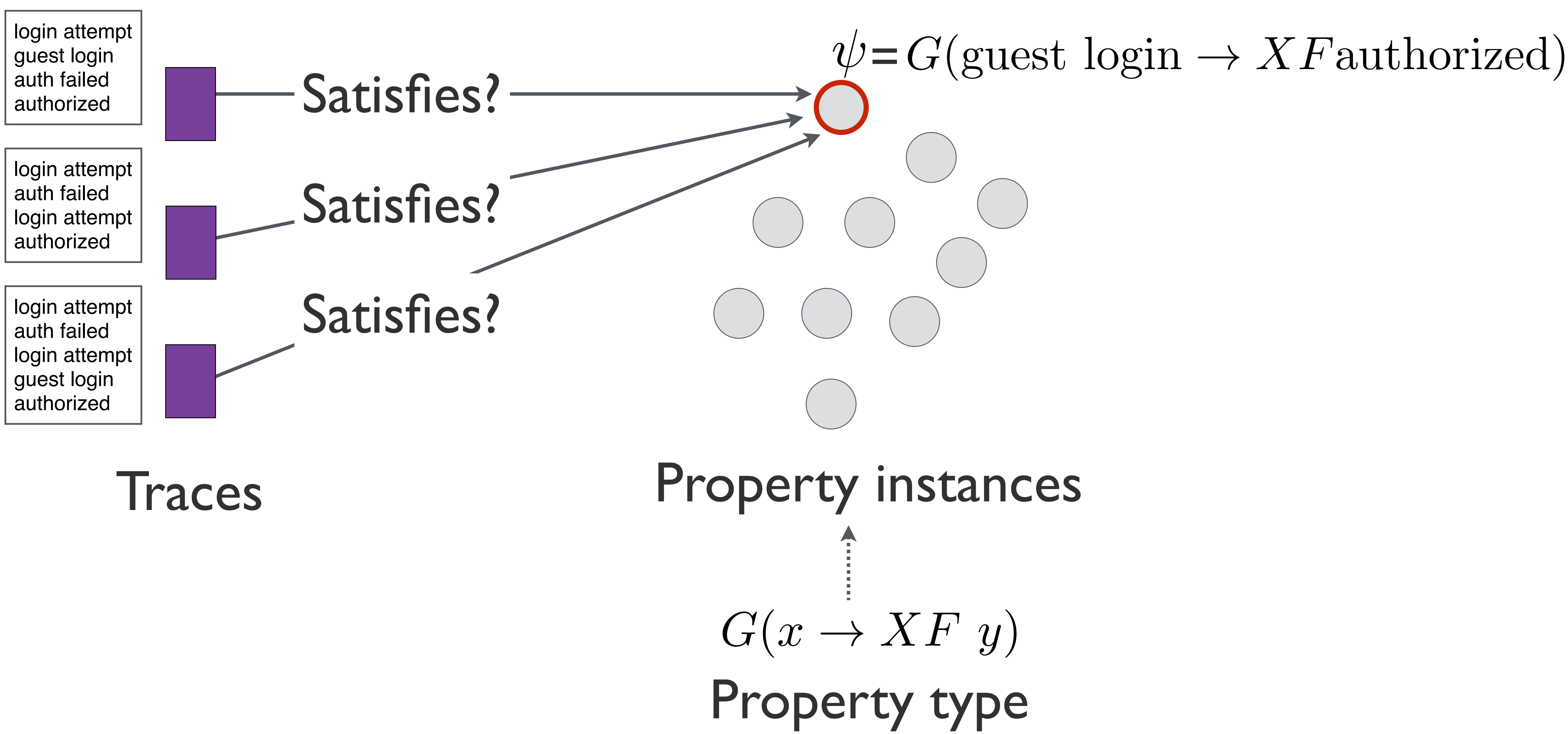
$\psi = G(\text{guest login} \rightarrow XF\text{authorized})$

Property instances

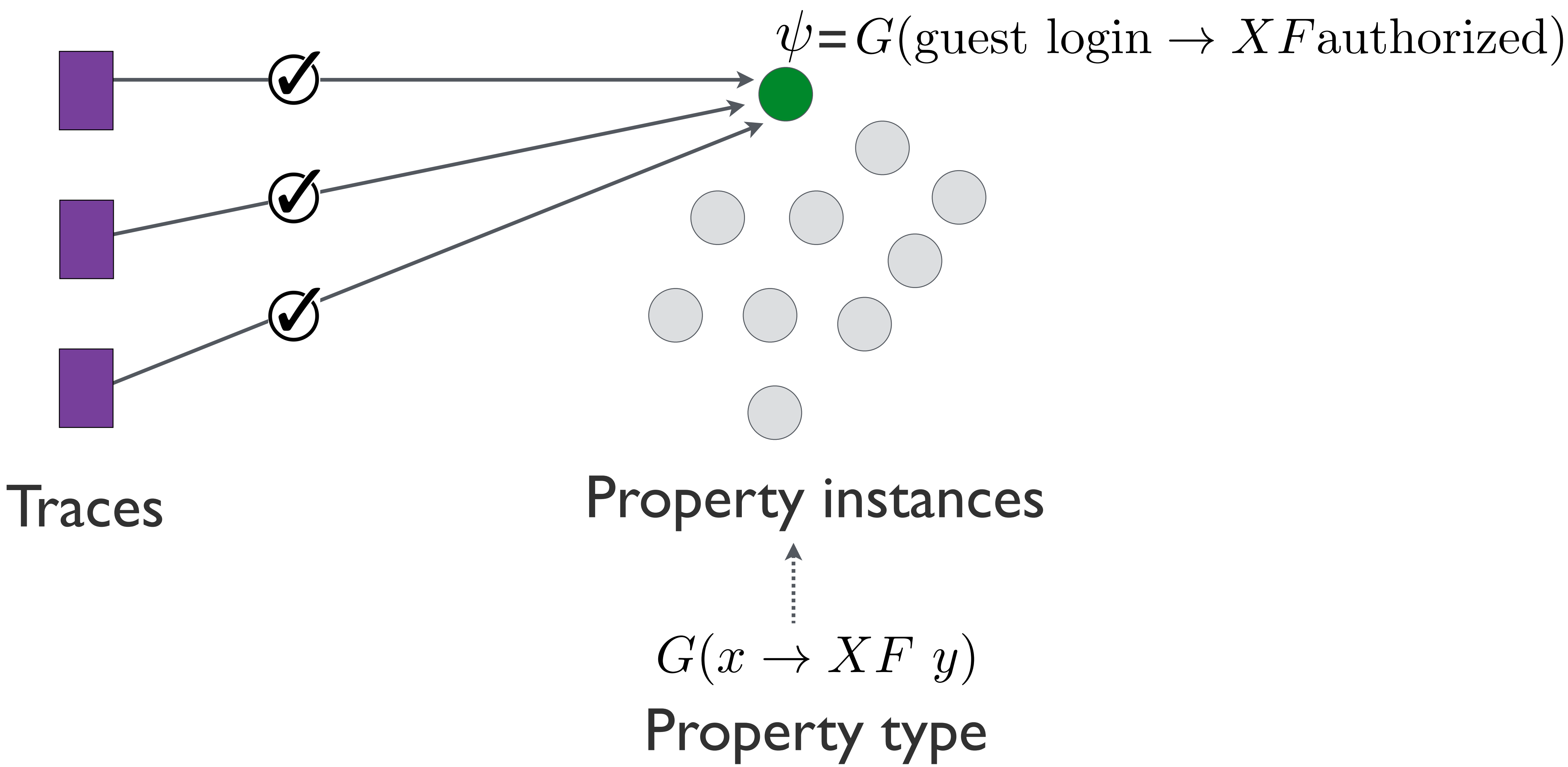
$G(x \rightarrow XF\ y)$

Property type

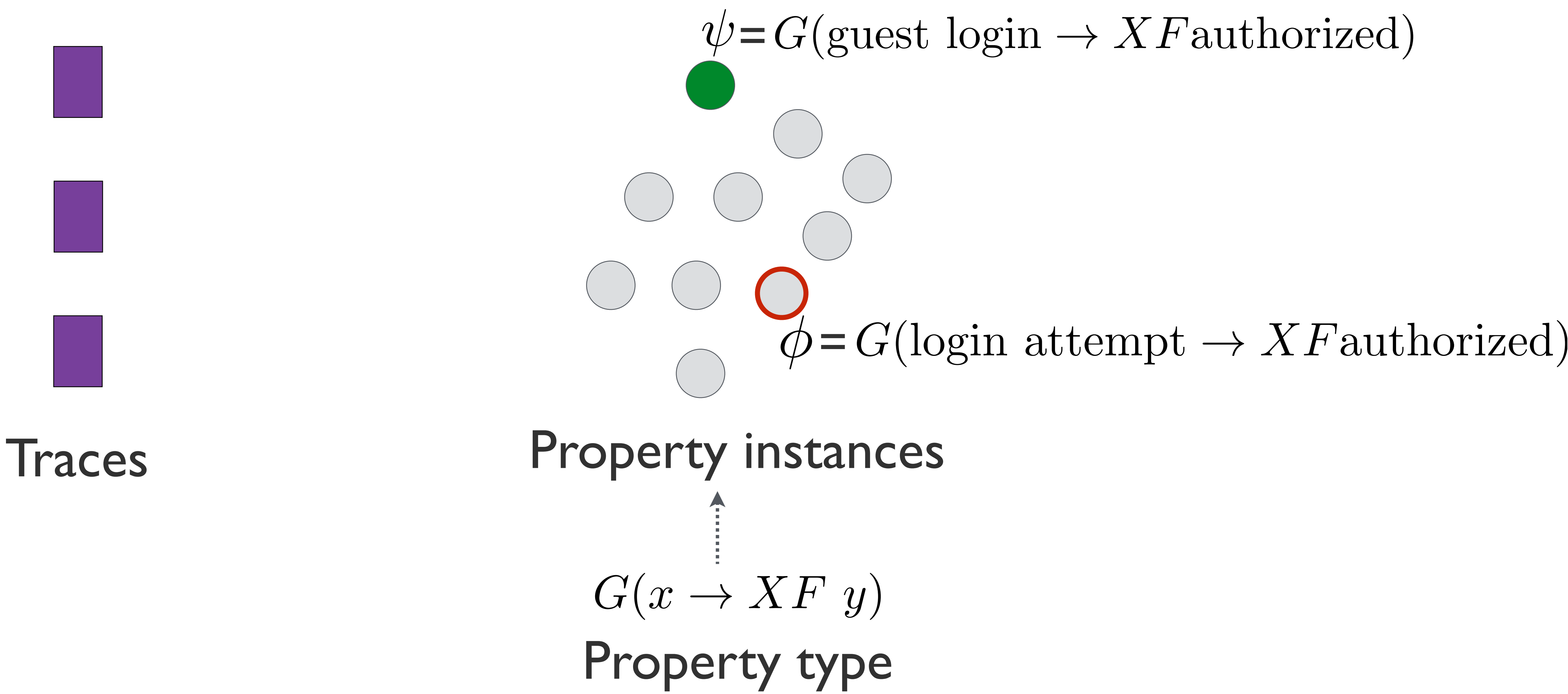
Texada overview: check instances



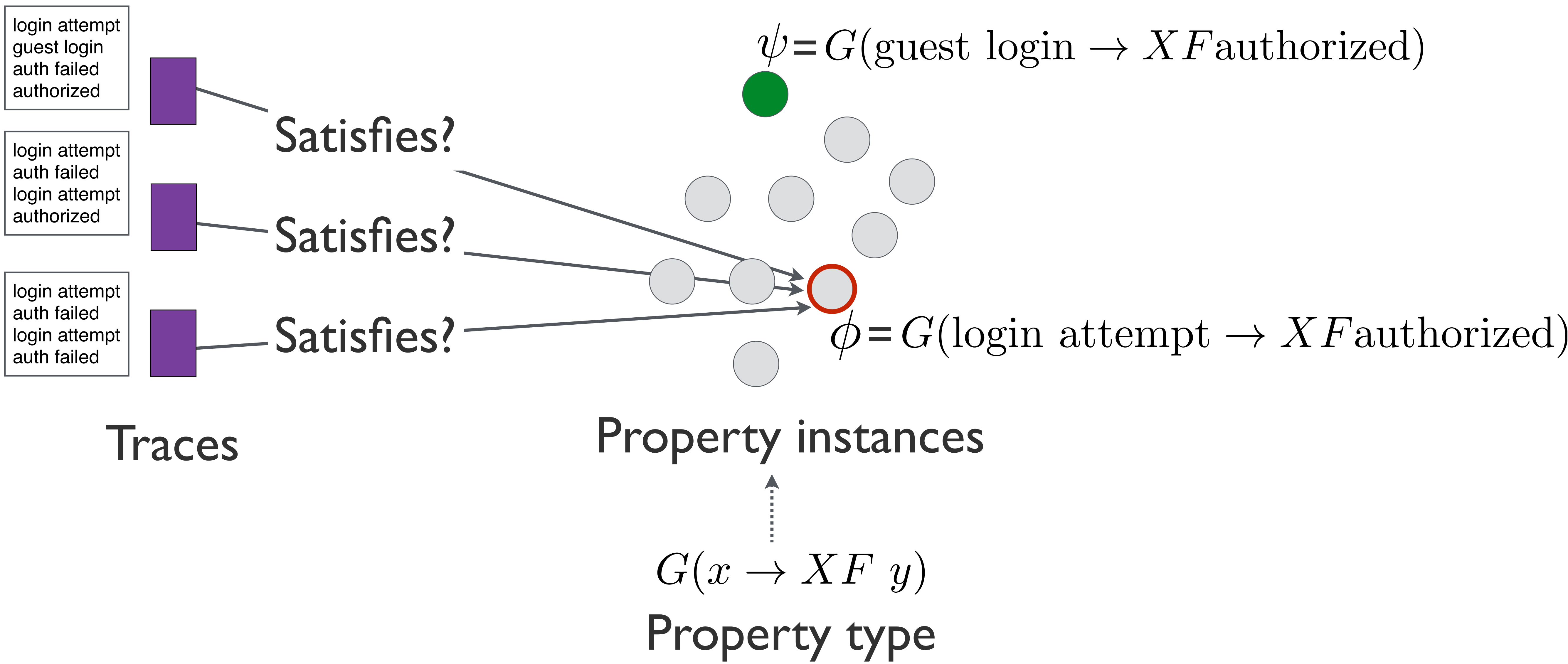
Texada overview: check instances



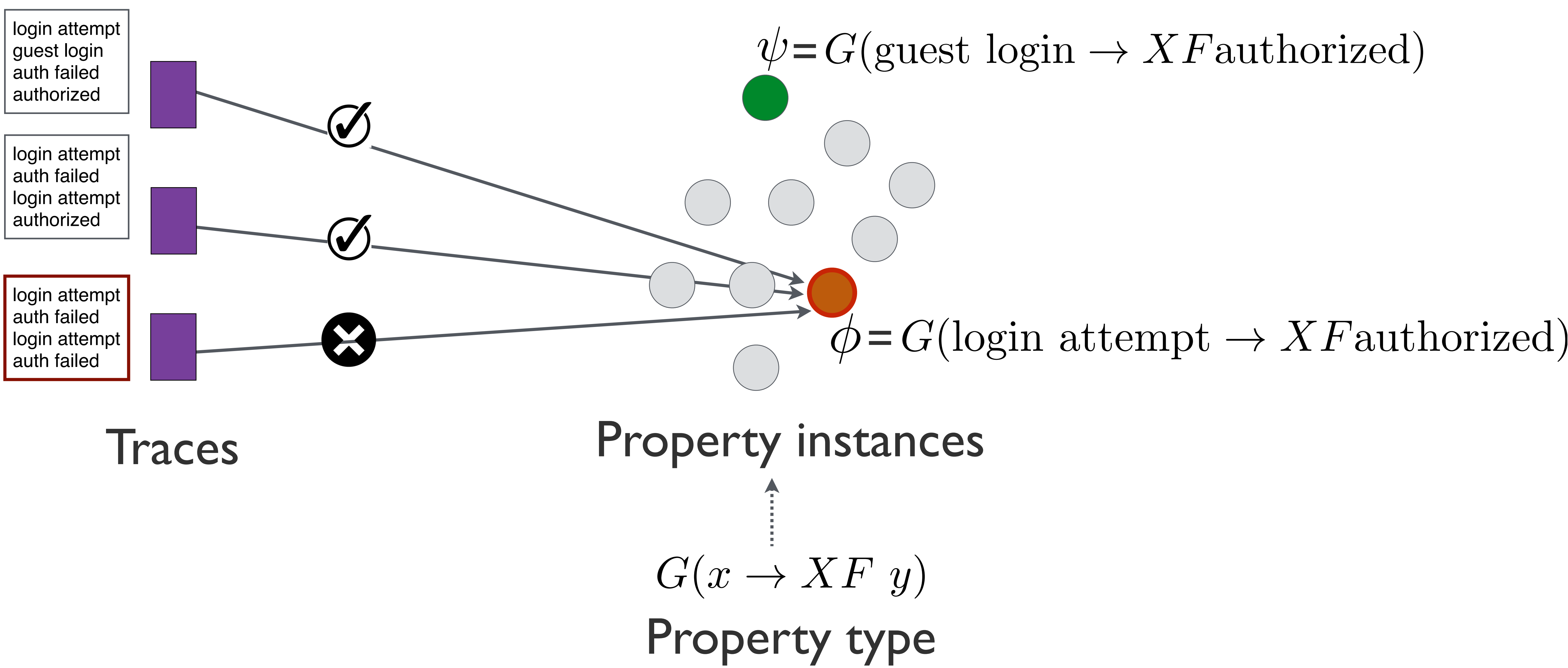
Texada overview: check instances



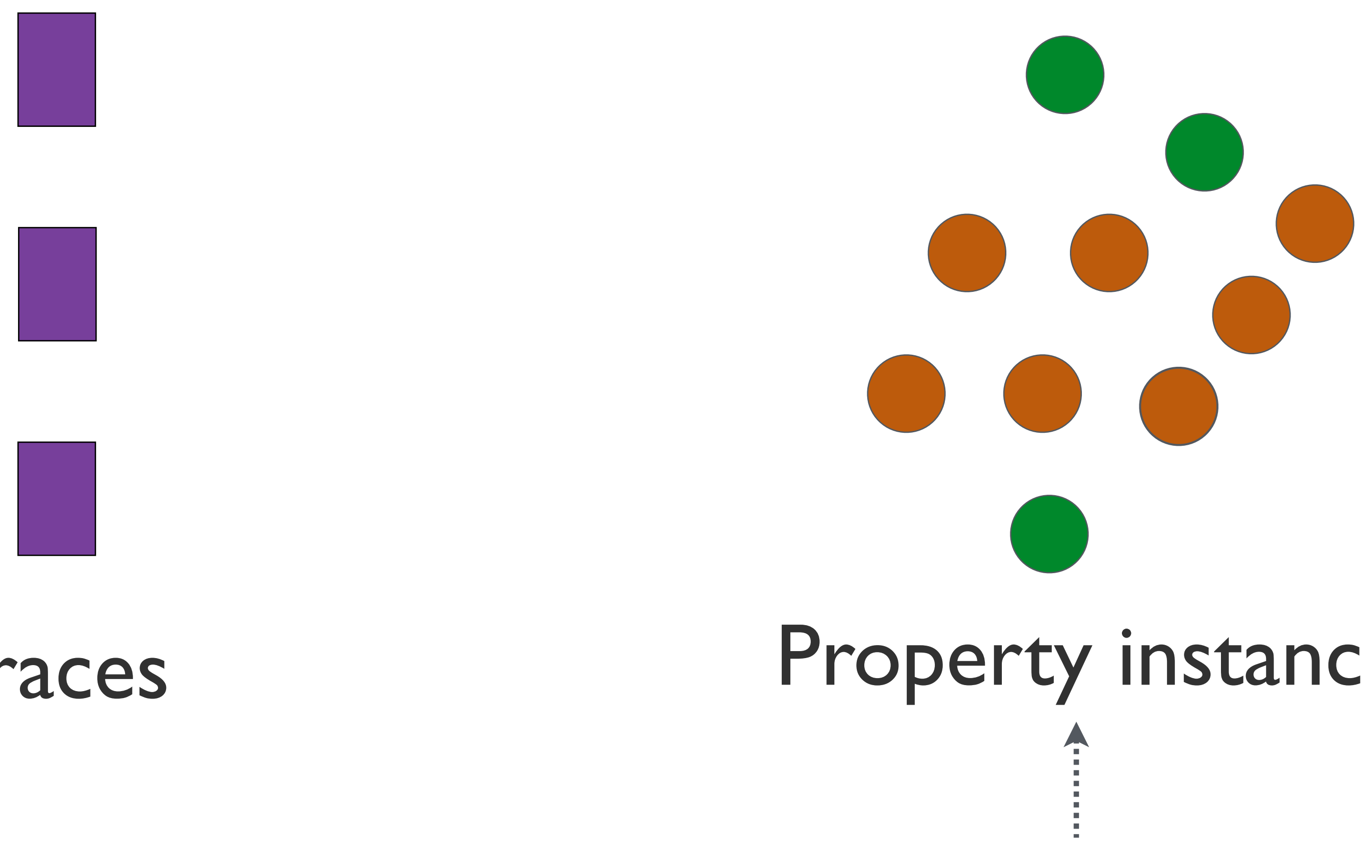
Texada overview: check instances



Texada overview: check instances



Texada overview



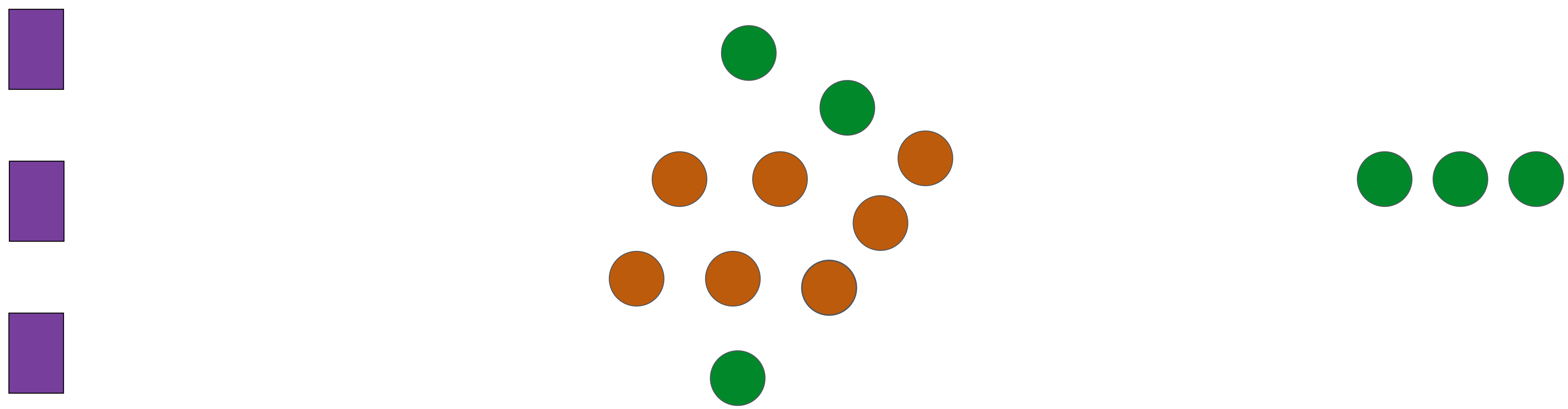
Traces

Property instances

$$G(x \to XF\ y)$$

Property type

Texada overview



Traces

Property instances

$G(x \rightarrow XF\ y)$

Property type

Texada output

Property instances that are true on all input traces

Trace representation

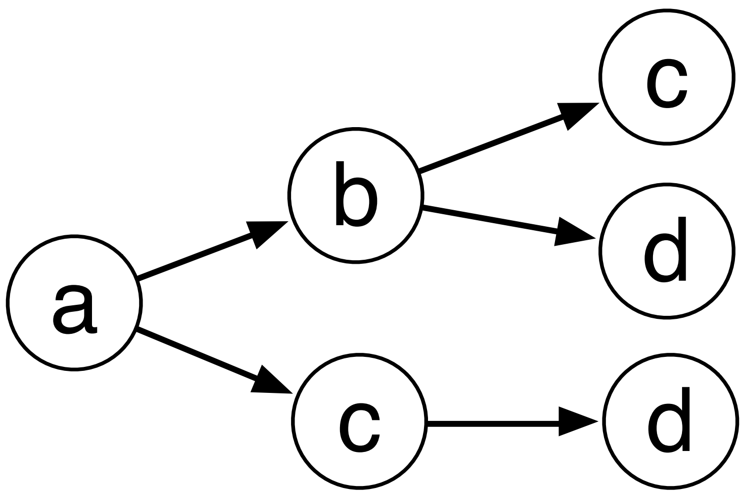
- Linear array of events
- Optimized representations
  - Map (event to a list of positions inside a trace)
  - Prefix tree (collapse identical prefixes)

a	a	a
b	b	c
c	d	d

Linear

a : 0 b : 1 c : 2 d :	a : 0 b : 1 c : d : 2	a : 0 b : c : 1 d : 2

Map



Prefix tree

Linear property instance checking

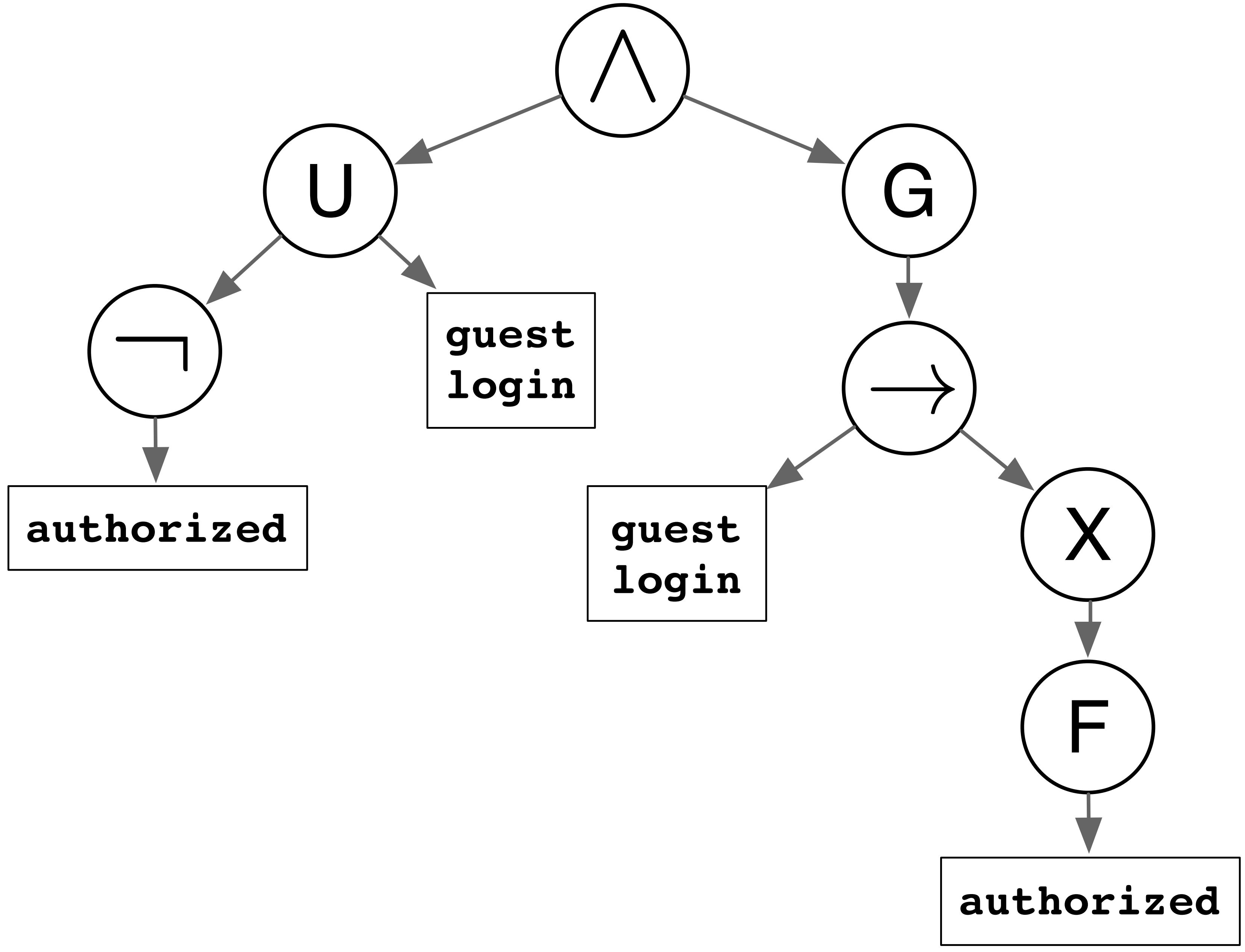
- LTL tree traversal and recursive trace traversal

$$(\neg \text{ authorized } U \text{ guest login}) \wedge G(\text{guest login} \rightarrow XF \text{ authorized})$$

Linear property instance checking

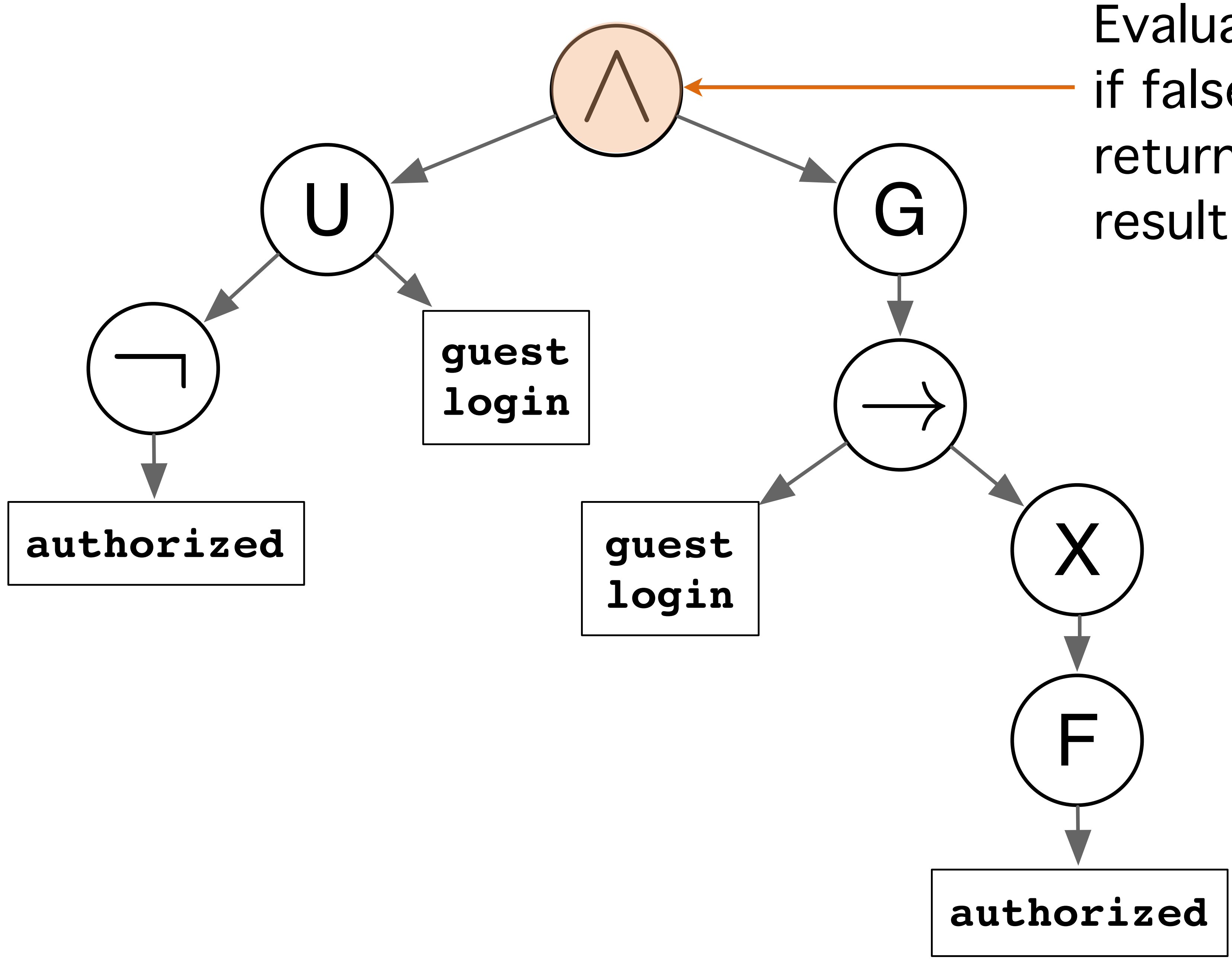
- LTL tree traversal and recursive trace traversal

$(\neg\ \text{authorized}\ U\ \text{guest login}) \wedge G(\text{guest login} \rightarrow XF\ \text{authorized})$



Linear property instance checking

- LTL tree traversal and recursive trace traversal

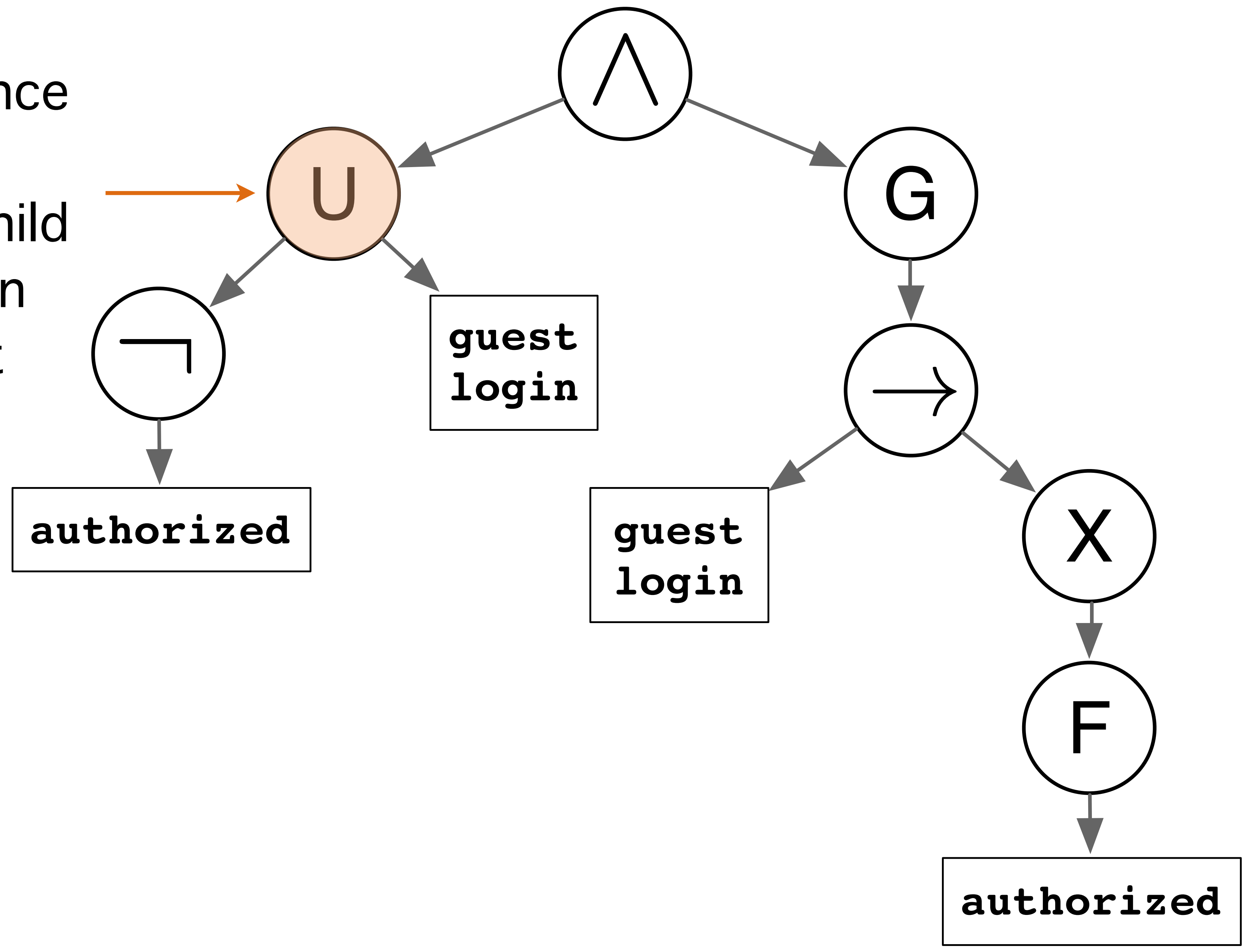


Evaluate left child, if false stop. Else, return right child result

Linear property instance checking

- LTL tree traversal and recursive trace traversal

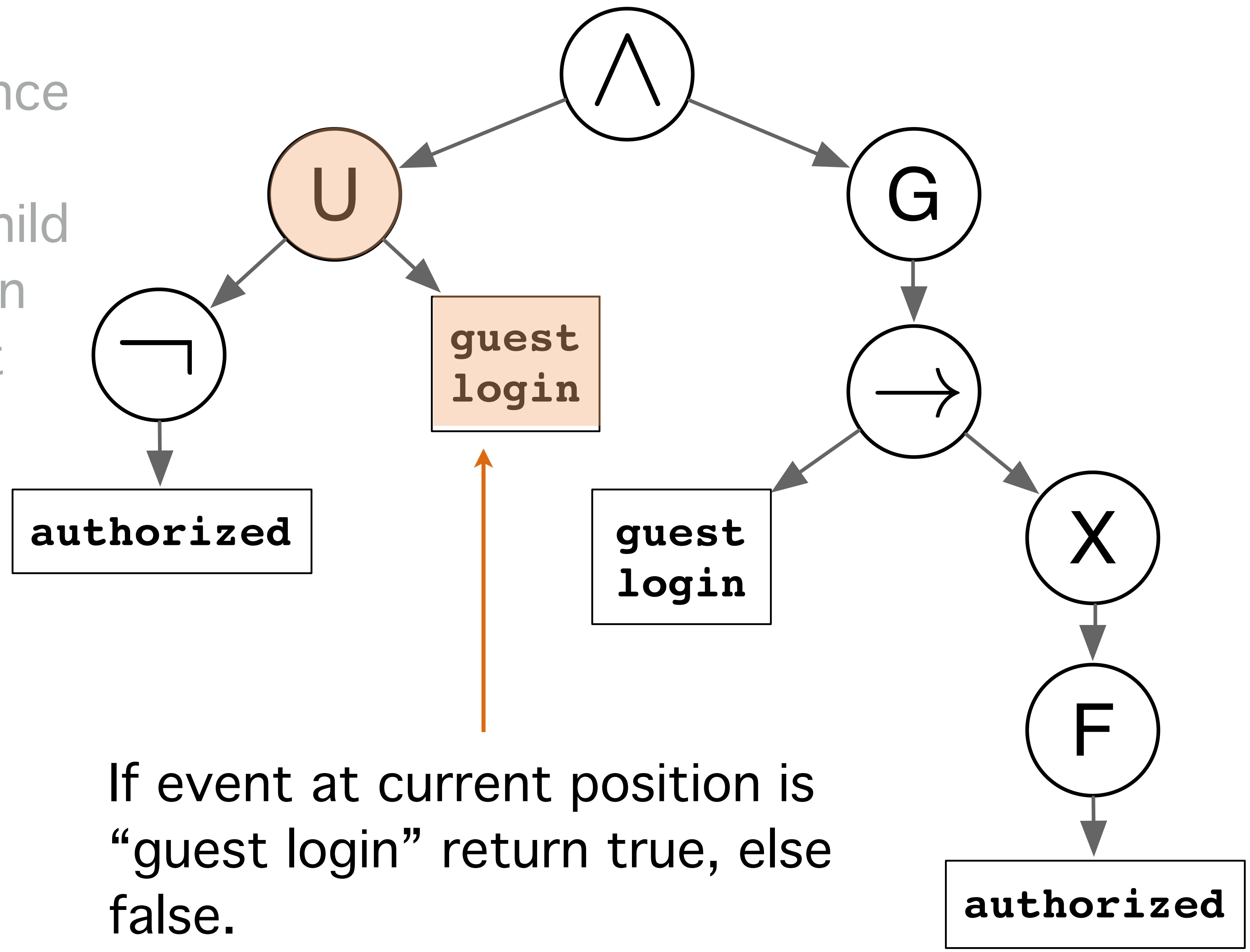
Find first instance of right child, evaluate left child at each position preceding right child



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

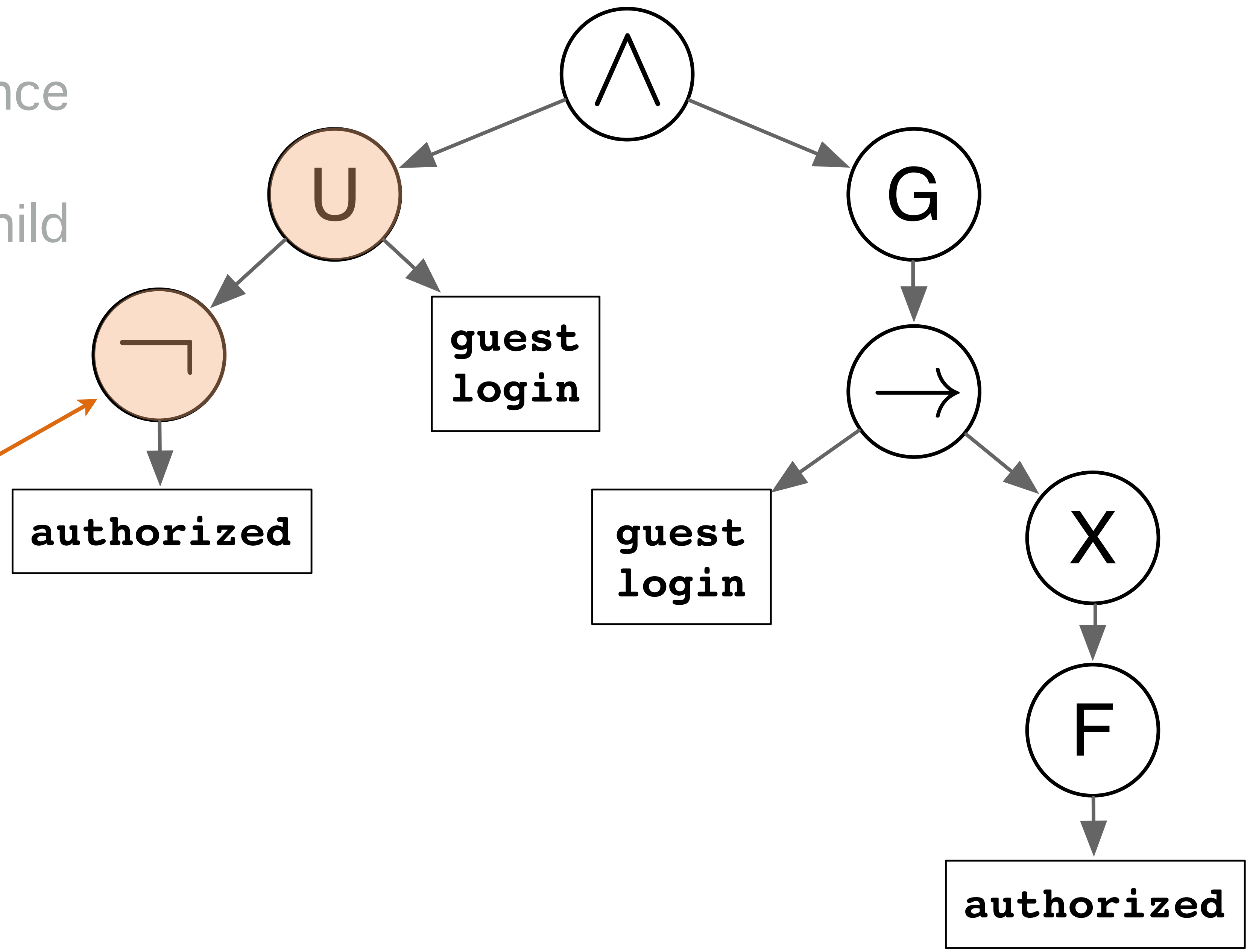


If event at current position is "guest login" return true, else false.

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

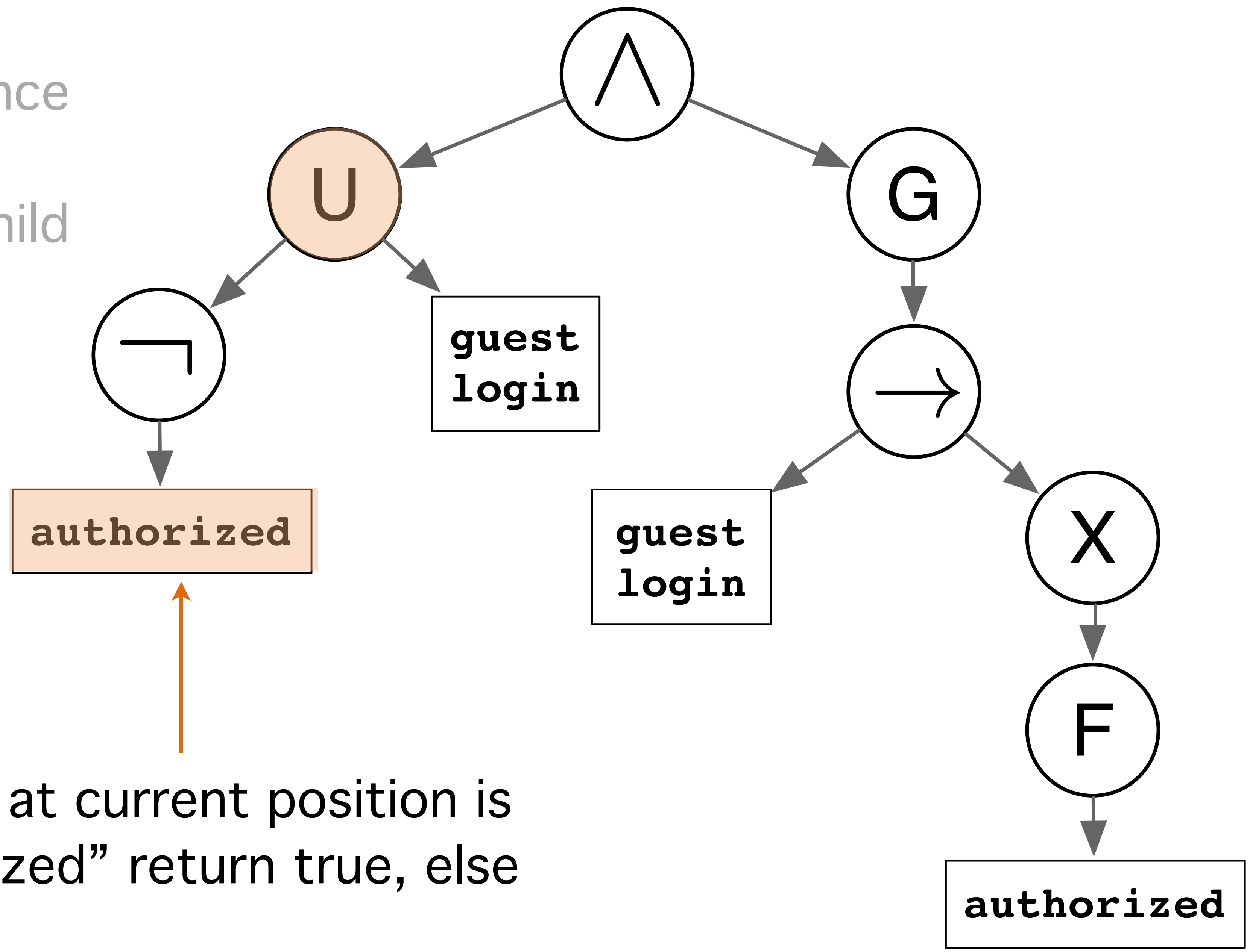


Return negation of child

Linear property instance checking

- LTL tree traversal and recursive trace traversal

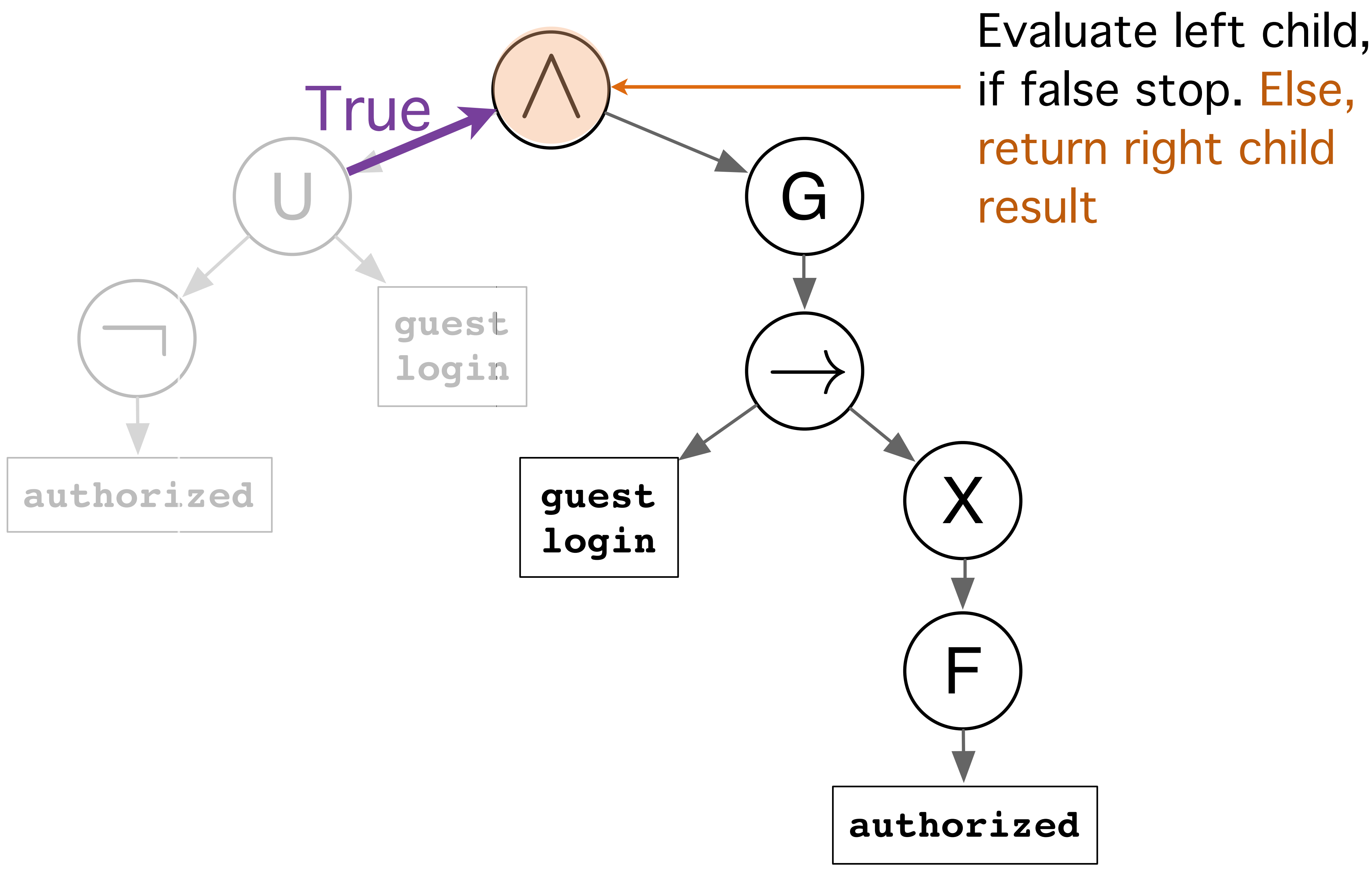
Find first instance of right child, evaluate left child on each event that precedes right child



If event at current position is "authorized" return true, else false.

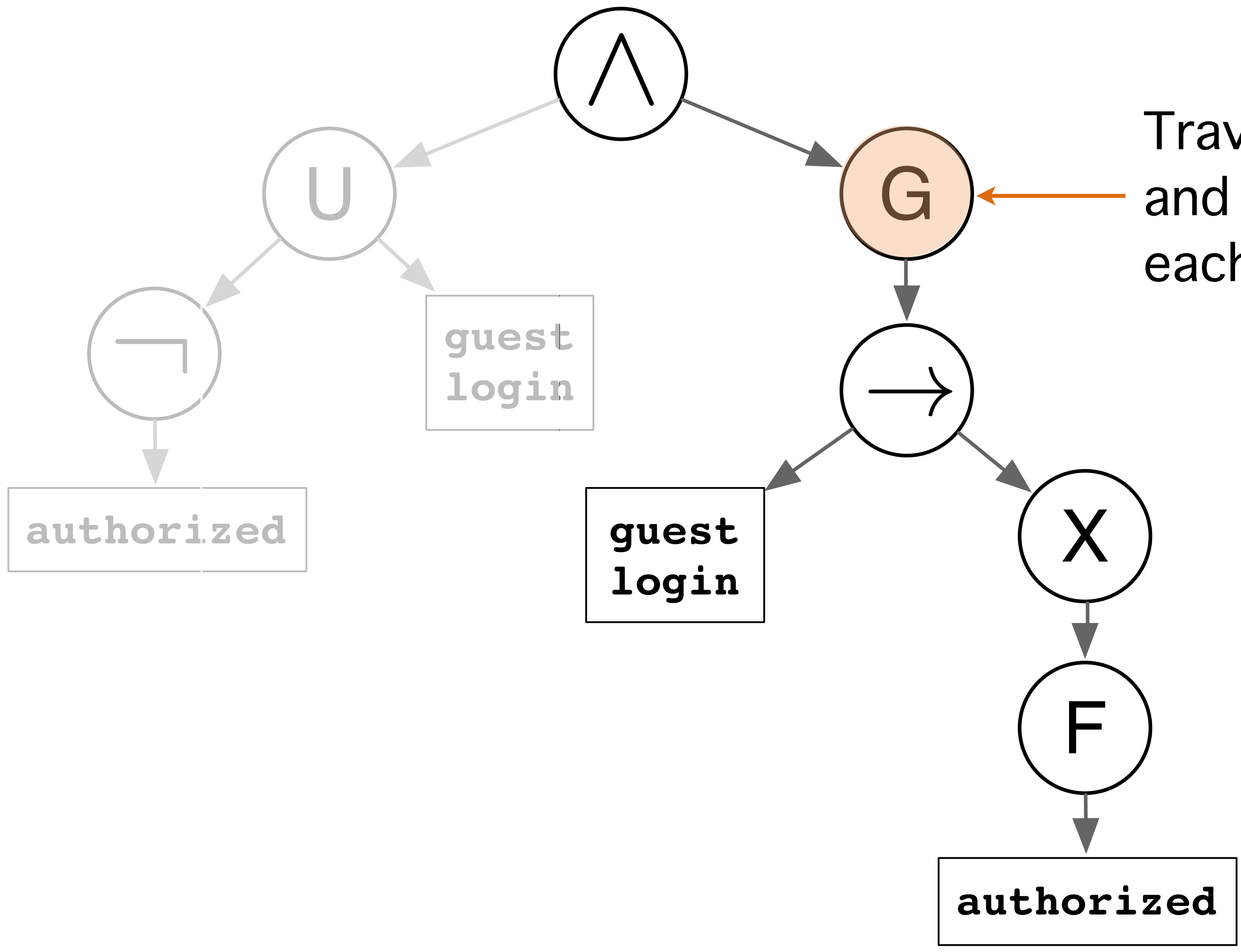
Linear property instance checking

- LTL tree traversal and recursive trace traversal



Linear property instance checking

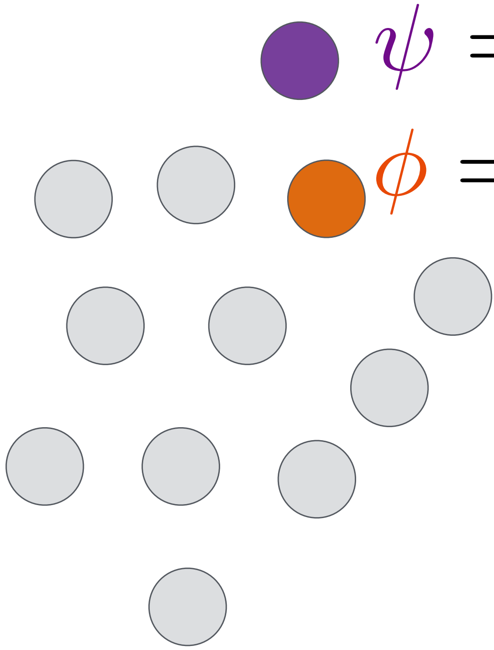
- LTL tree traversal and recursive trace traversal



Traverse all events and check child at each one

Key optimization: checking memoization

- Many property instances have a similar structure

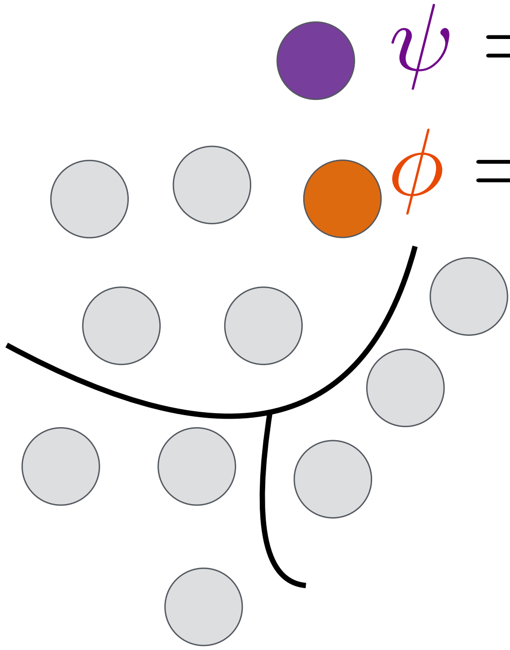


$$\psi = G(c \land \neg e \rightarrow ((a \rightarrow (\neg e \; U \; (b \land \neg e)))W \; e))$$

$$\phi = G(d \land \neg e \rightarrow ((a \rightarrow (\neg e \; U \; (b \land \neg e)))W \; e))$$

Key optimization: checking memoization

- Many property instances have a similar structure



$$\psi = G(c \wedge \neg e \rightarrow ((a \rightarrow (\neg e \; U \; (b \wedge \neg e)))W \;\; e))$$

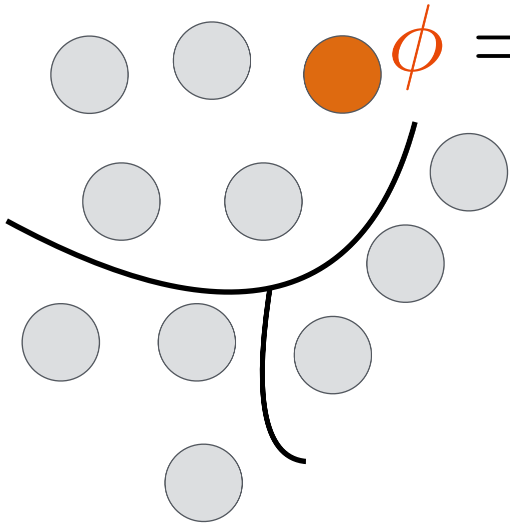
$$\phi = G(d \wedge \neg e \rightarrow ((a \rightarrow (\neg e \; U \; (b \wedge \neg e)))W \;\; e))$$

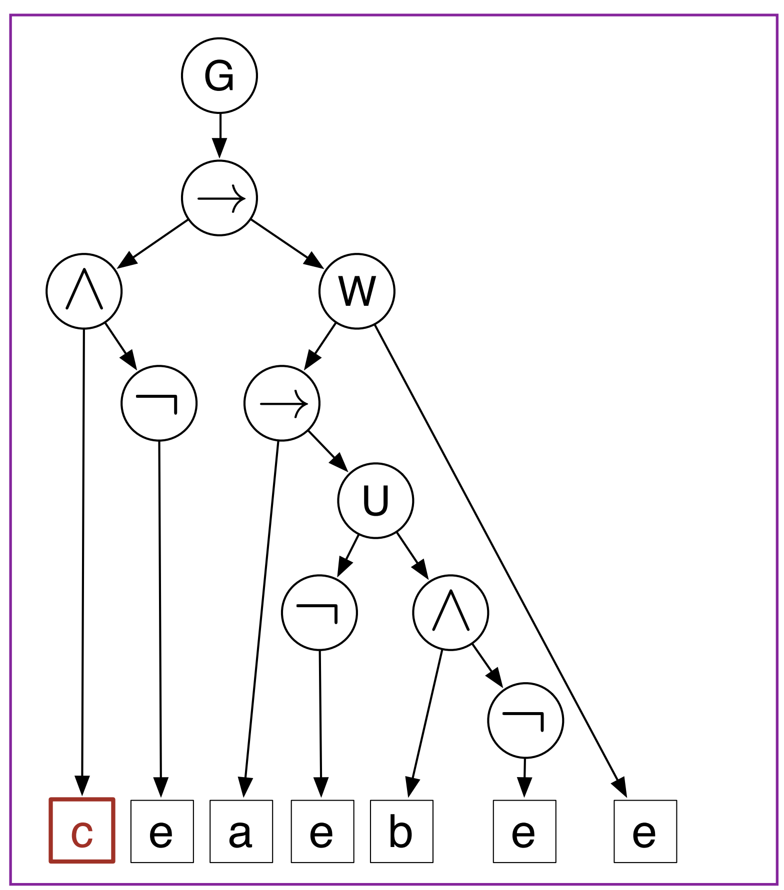
Checking memoization

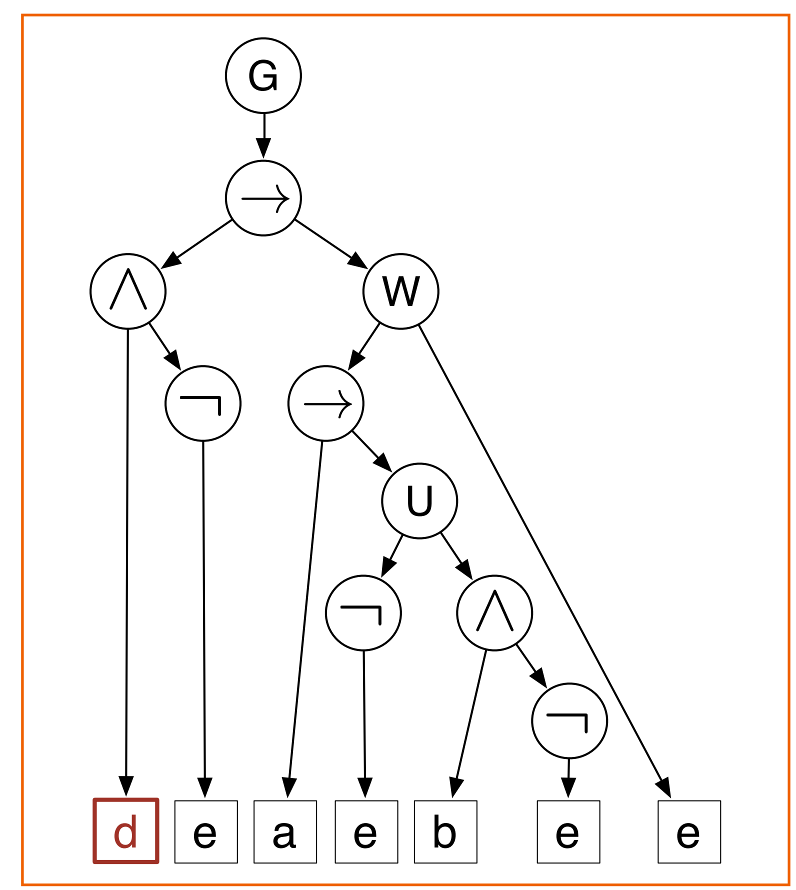
- Many property instances have a similar structure

$$\psi = G(c \wedge \neg e \rightarrow ((a \rightarrow (\neg e\ U\ (b \wedge \neg e)))W\ e))$$

$$\phi = G(d \wedge \neg e \rightarrow ((a \rightarrow (\neg e\ U\ (b \wedge \neg e)))W\ e))$$





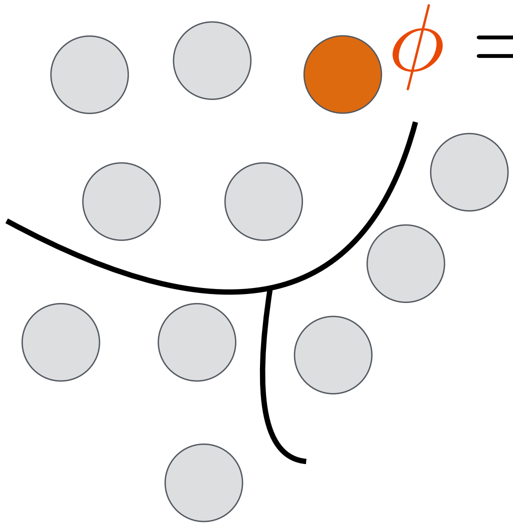


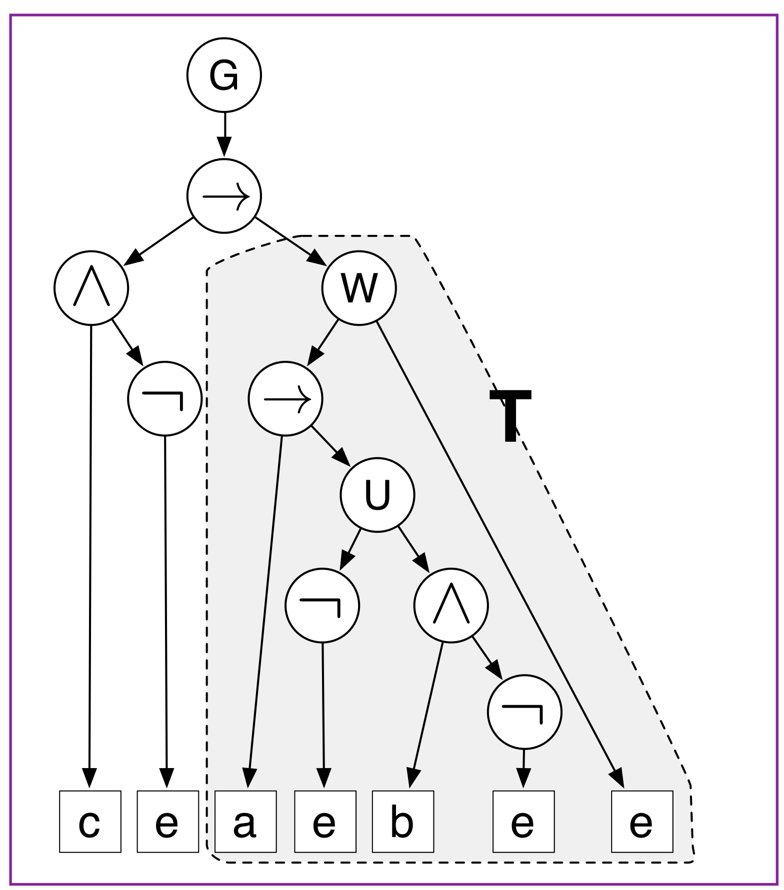
Checking memoization

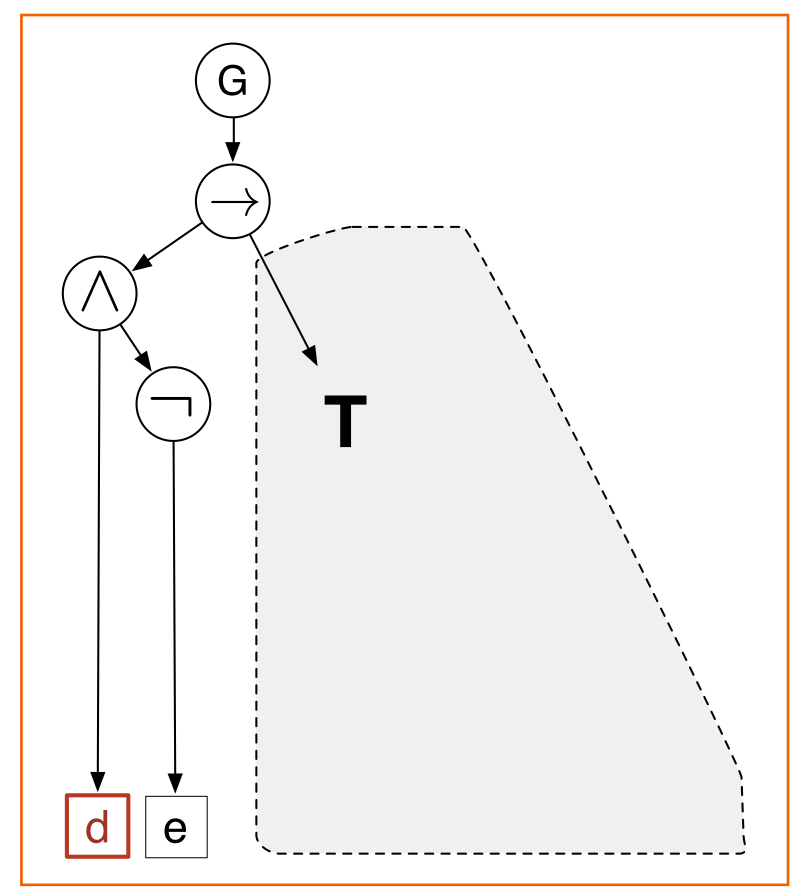
- Many property instances have a similar structure

$$\psi = G(c \wedge \neg e \rightarrow ((a \rightarrow (\neg e\ U\ (b \wedge \neg e)))W\ e))$$

$$\phi = G(d \wedge \neg e \rightarrow ((a \rightarrow (\neg e\ U\ (b \wedge \neg e)))W\ e))$$





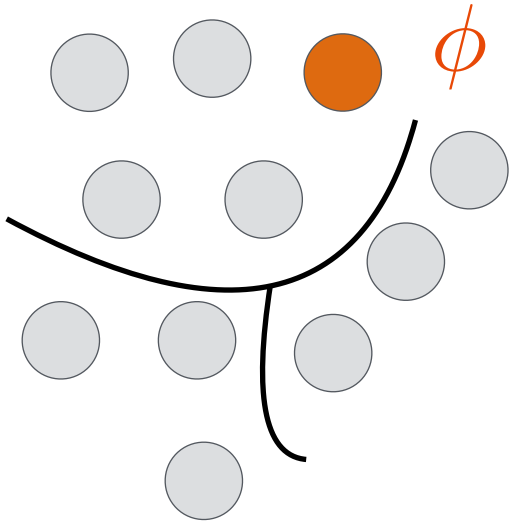


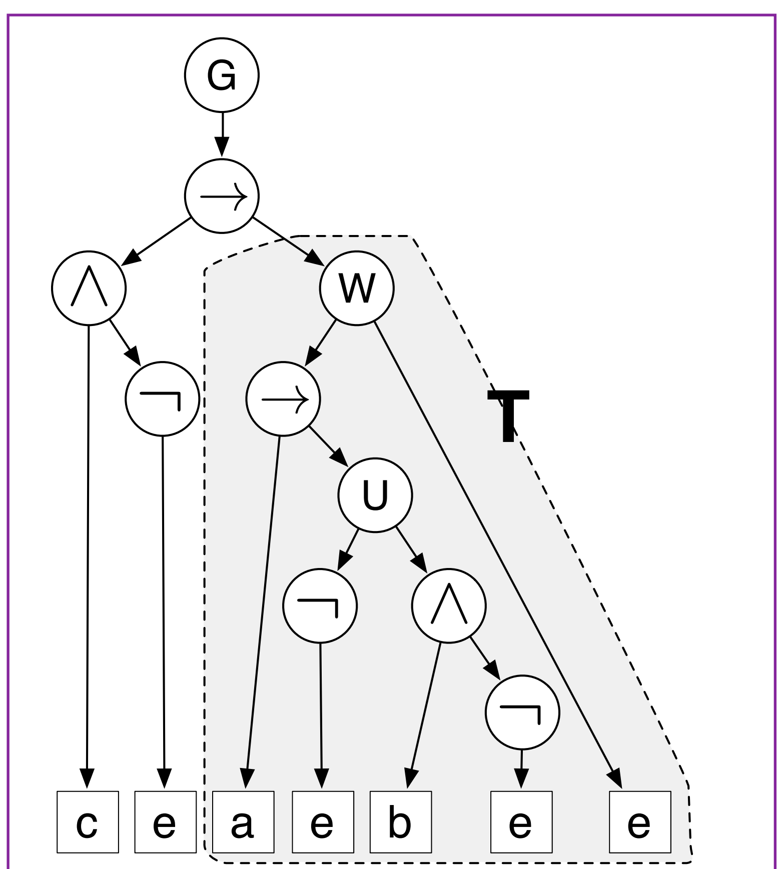
Checking memoization

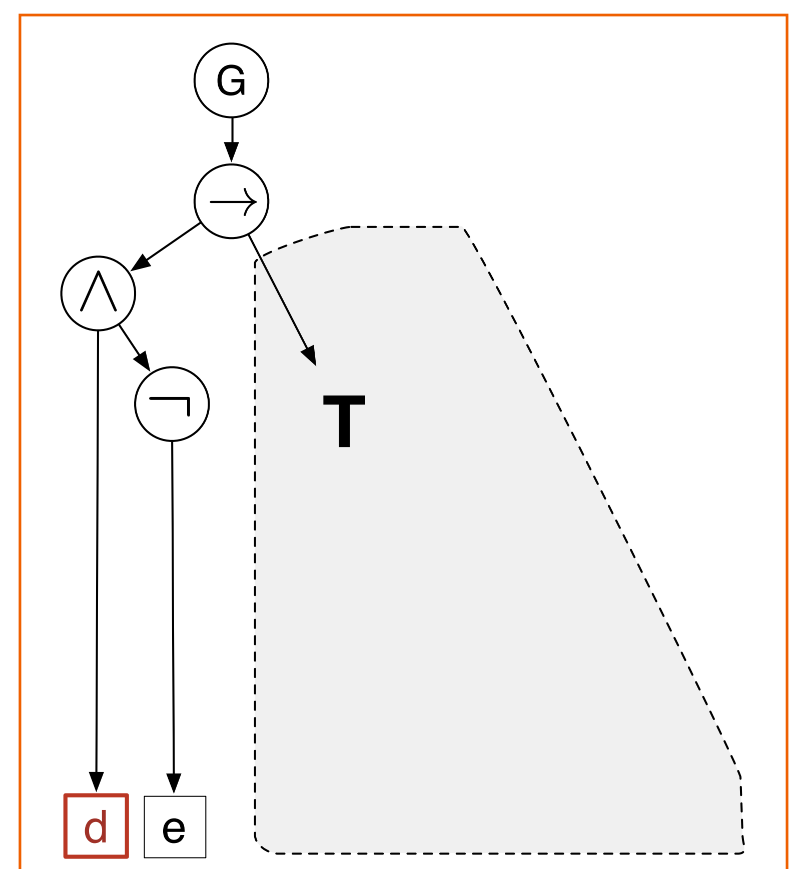
- Many property instances have a similar structure

$\psi = G(c \wedge \neg e \rightarrow \mathbf{T})$

$\phi = G(d \wedge \neg e \rightarrow \mathbf{T})$

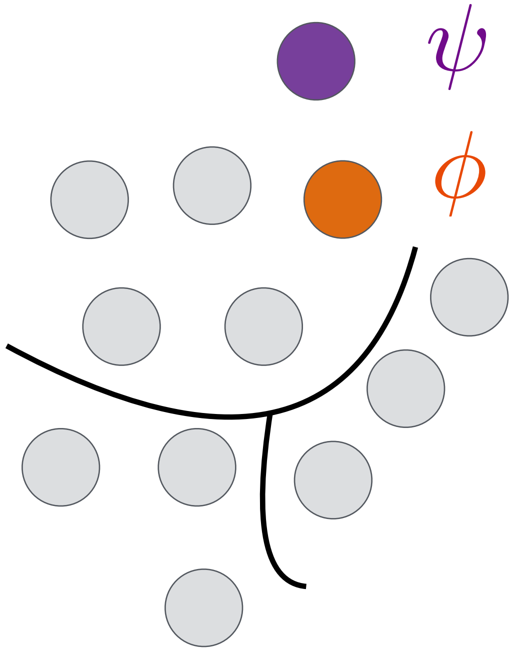






Checking memoization

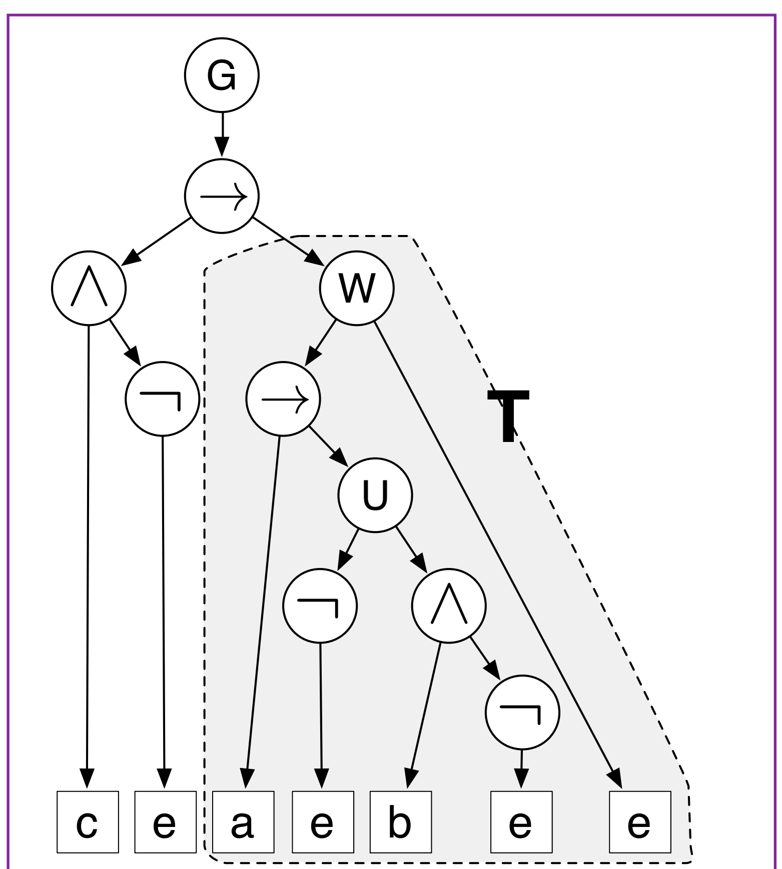
- Many property instances have a similar structure

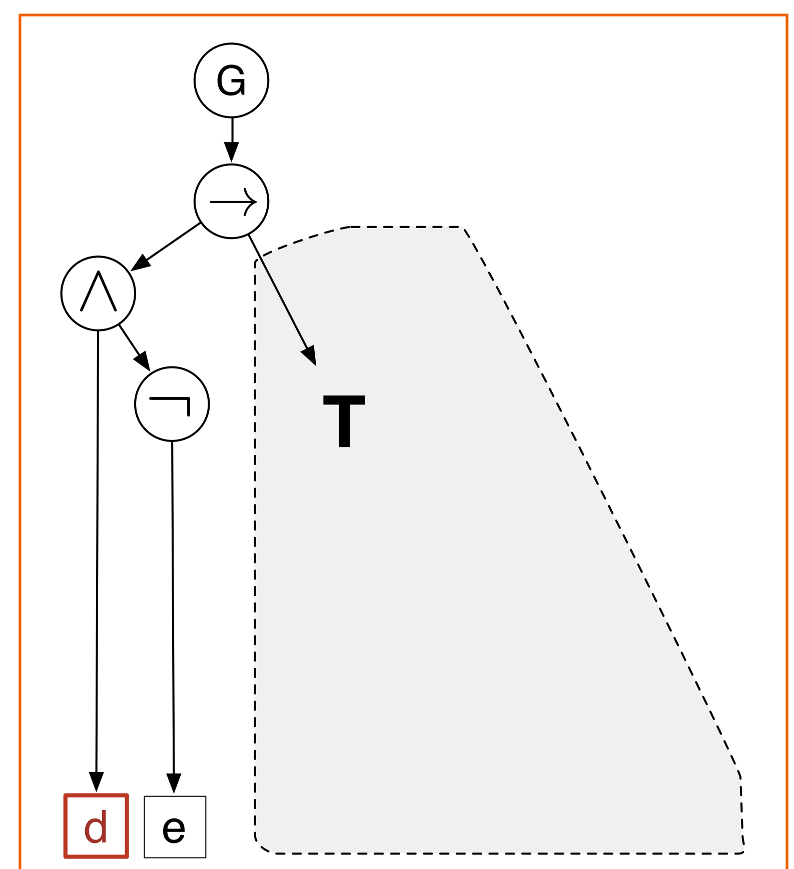


$\psi = G(c \wedge \neg e \rightarrow$ **T** $)$

$\phi = G(d \wedge \neg e \rightarrow$ **T** $)$

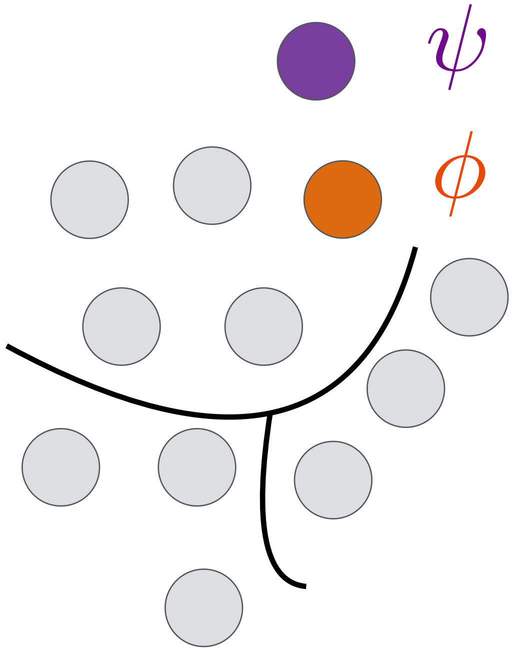
- Can only re-use results if evaluated at same point in the trace
- Memory vs. compute trade-off





Checking memoization

- Many property instances have a similar structure



$$\psi = G(c \wedge \neg e \rightarrow \boxed{\mathbf{T}})$$

$$\phi = G(d \wedge \neg e \rightarrow \boxed{\mathbf{T}})$$

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: aaaab ✖ 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 ✓ 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]

  – Perracotta[2]

Name	Regex	LTL
Always Followed by		G(x→XFy)
Never Followed by		G(x→XG!y)
Always Precedes		(!y W x)
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.

Expressiveness of Property Types

- Texada can express properties from prior work

Name	Regex	LTL
Always Followed by		G(x→XFy)
CauseFirst	(xx*yy*)*	(!y W x) & G(x→XFy)
OneEffect	y*(xx*y)*	G((x→XFy) & (y→X(!y W x)))

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

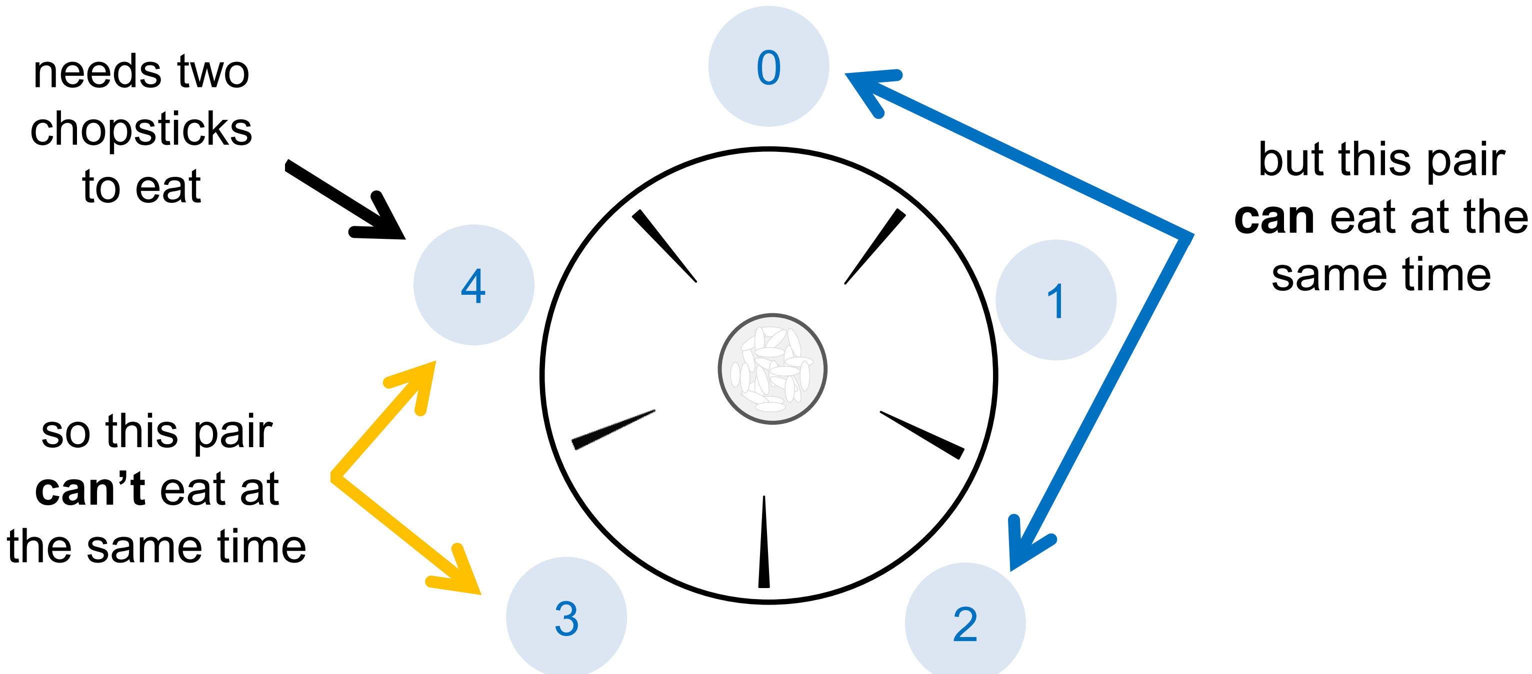
– *Patterns in Property Specifications for Finite-State Verification* [Dwyer et al. ICSE'99]

[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

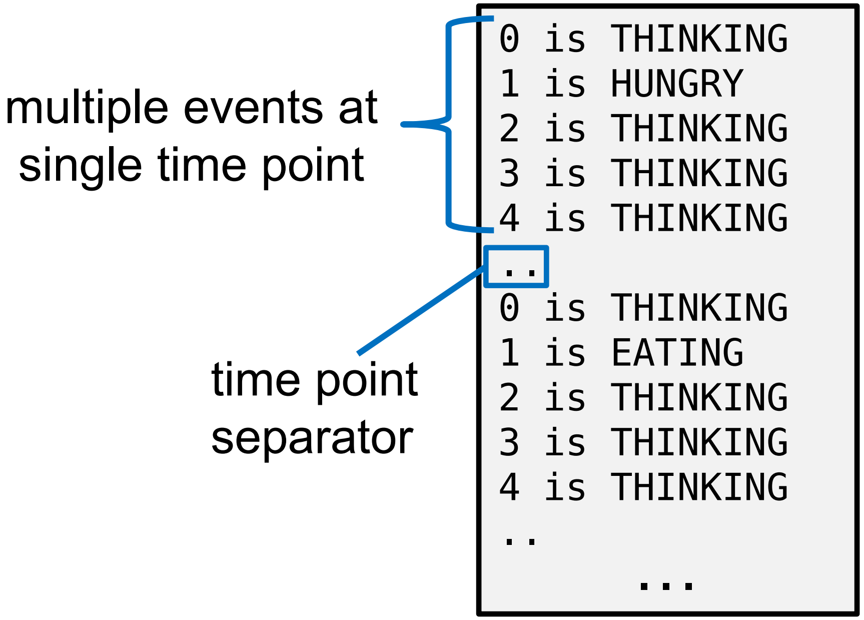
- Classic concurrency problem: philosophers sit around a table, thinking, hungry, or eating.



- These specs could not be checked with previous temporal spec miners!

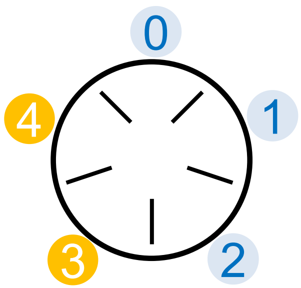
Multi-Propositional Traces

- 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



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)
```

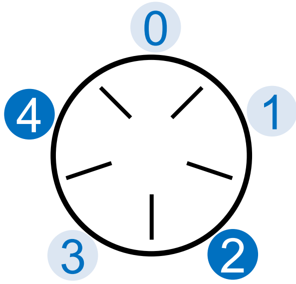
- Texada output for G($x \rightarrow !y$) includes

```
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

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

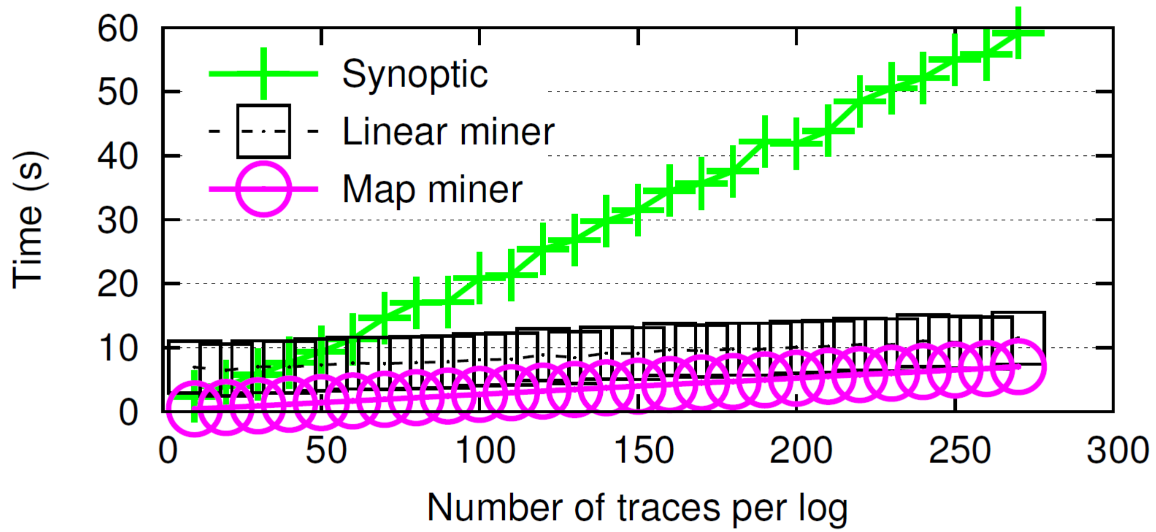
- T

```
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

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:

Unique events (10K events/trace, 20 traces/log)	**Perracotta**	**Texada (map miner)**
120	0.85 s	2.42 s
160	0.97 s	4.07 s
260	1.42 s	10.21 s

- 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:

Unique events (10K events/trace, 20	Perracotta	Texada (map miner)

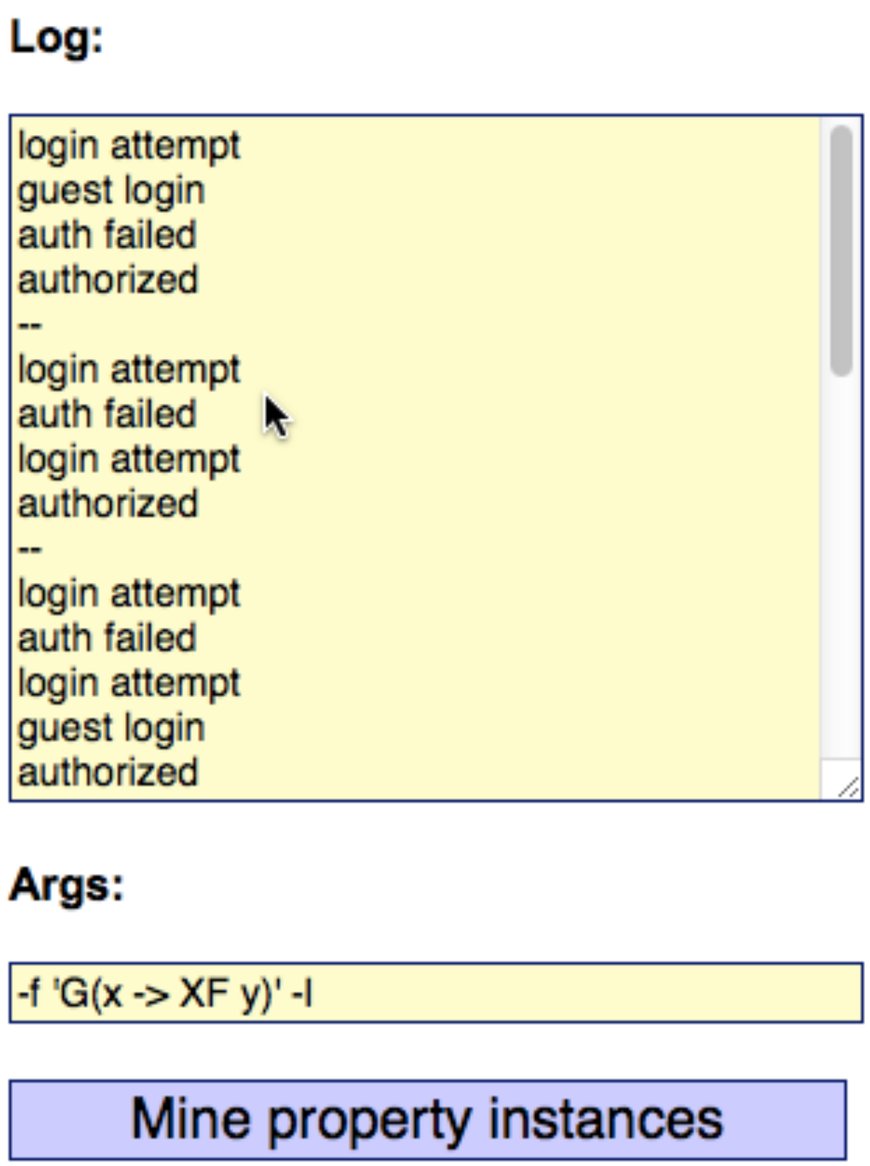
- Texada can mine a wide variety of properties
- Texada can mine concurrent sys. properties
- Texada has reasonable performance

- 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 demo

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

Online tool:
http://bestchai.bitbucket.org/texada/



In this talk

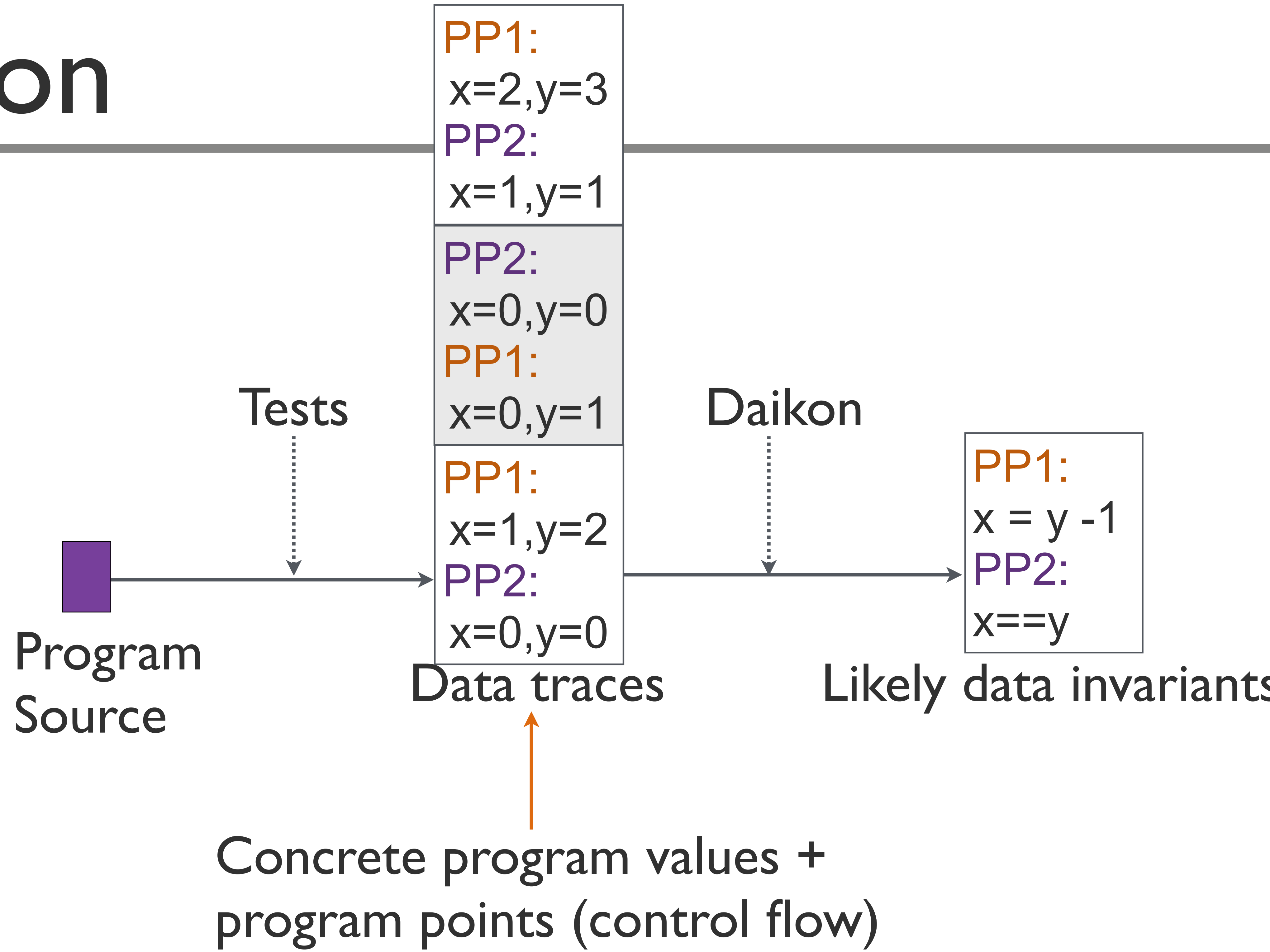
- Overview linear temporal logic (LTL)
- Texada: a tool to mine general LTL properties
- Overview Daikon: a data property miner
- Quarry: a tool that combines Daikon and Texada to mine data-temporal properties

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


PP1:
x=2,y=3
PP2:
x=1,y=1
PP2:
x=0,y=0
PP1:
x=0,y=1
PP1:
x=1,y=2
PP2:
x=0,y=0
Tests
Daikon
PP1:
x = y -1
PP2:
x==y
Program Source
Data traces
Likely data invariants
Concrete program values + program points (control flow)

Daikon applied to a queue

- Likely invariants
  - size <= capacity
  - isFull one of {true, false}

vars : {size, capacity, isFull}

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

Temporal invariants (Texada)

```
create()
enqueue(5)
enqueue(3)
enqueue(8)
enqueue(16)
dequeue()
```

enqueue() is always followed by dequeue()

Relate events through time.

But: **data values** may interact **through time**

Daikon applied to a queue

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

What if we consider non-global scope?

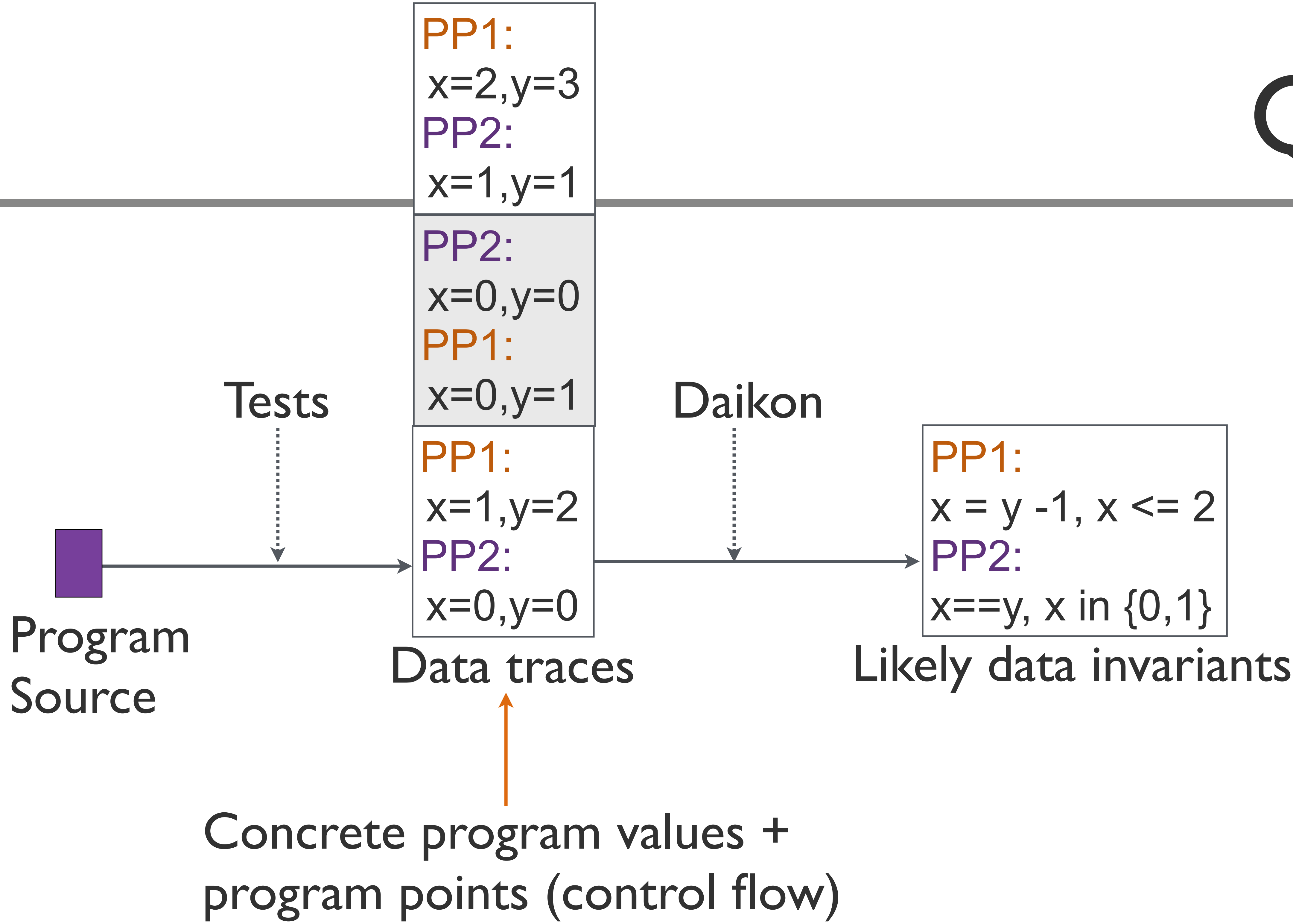
Daikon applied to a queue

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

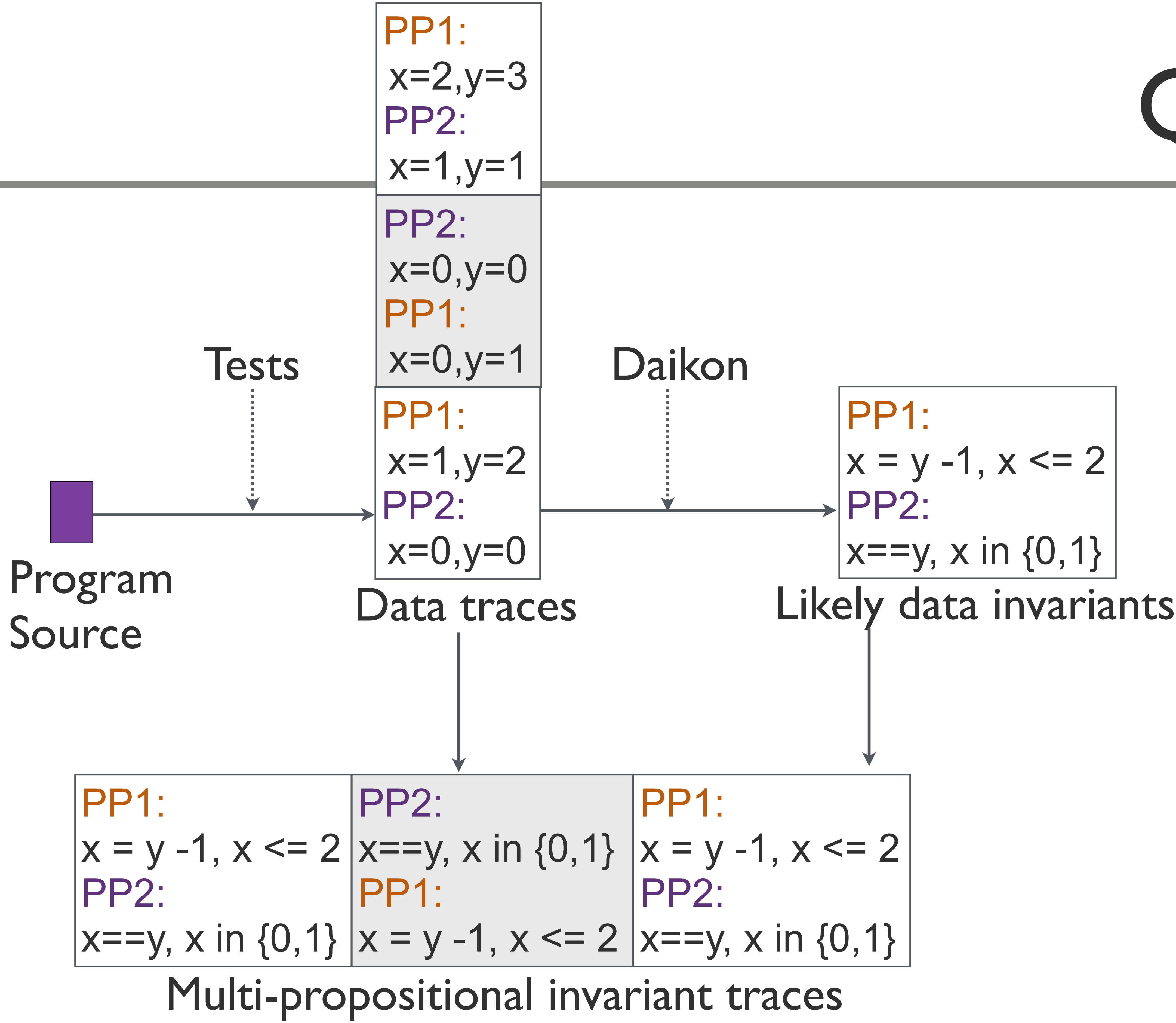
What if we consider non-global scope?

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

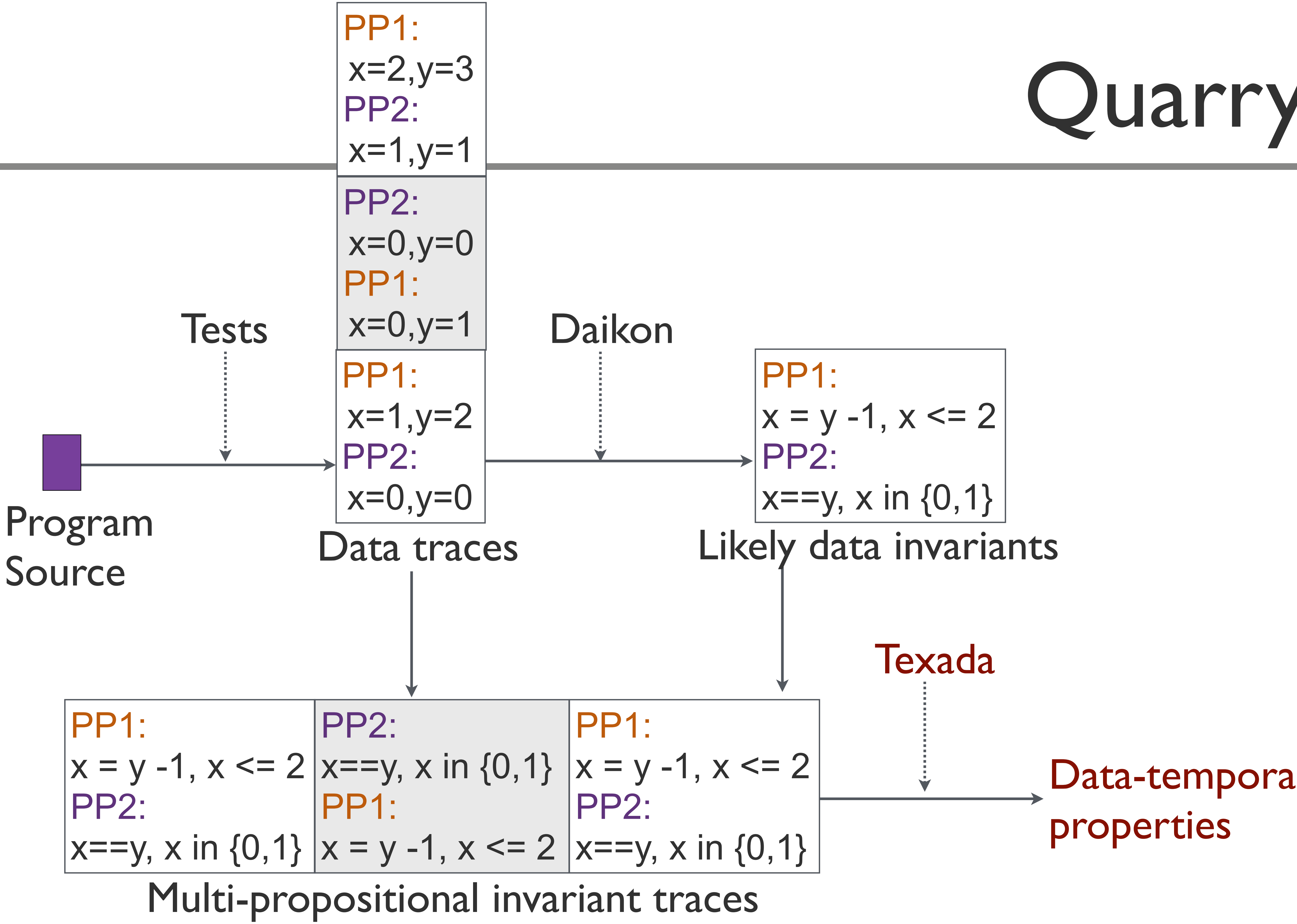
Quarry


PP1:
x=2,y=3
PP2:
x=1,y=1
PP2:
x=0,y=0
PP1:
x=0,y=1
Tests
Daikon
PP1:
x=1,y=2
PP2:
x=0,y=0
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
Program Source
Data traces
Likely data invariants
Concrete program values +
program points (control flow)

Quarry


PP1:
x=2,y=3
PP2:
x=1,y=1
PP2:
x=0,y=0
PP1:
x=0,y=1
PP1:
x=1,y=2
PP2:
x=0,y=0
Tests
Daikon
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
Program Source
Data traces
Likely data invariants
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
PP2:
x==y, x in {0,1}
PP1:
x = y -1, x <= 2
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
Multi-propositional invariant traces

Quarry


PP1:
x=2,y=3
PP2:
x=1,y=1
PP2:
x=0,y=0
PP1:
x=0,y=1
Tests
PP1:
x=1,y=2
PP2:
x=0,y=0
Daikon
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
Program Source
Data traces
Likely data invariants
Texada
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
PP2:
x==y, x in {0,1}
PP1:
x = y -1, x <= 2
PP1:
x = y -1, x <= 2
PP2:
x==y, x in {0,1}
Data-temporal properties
Multi-propositional invariant traces

Quarry applied to a queue

- G(size <= capacity)
- (isFull == false) U (size == capacity)
- G(this.back <= 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` 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

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

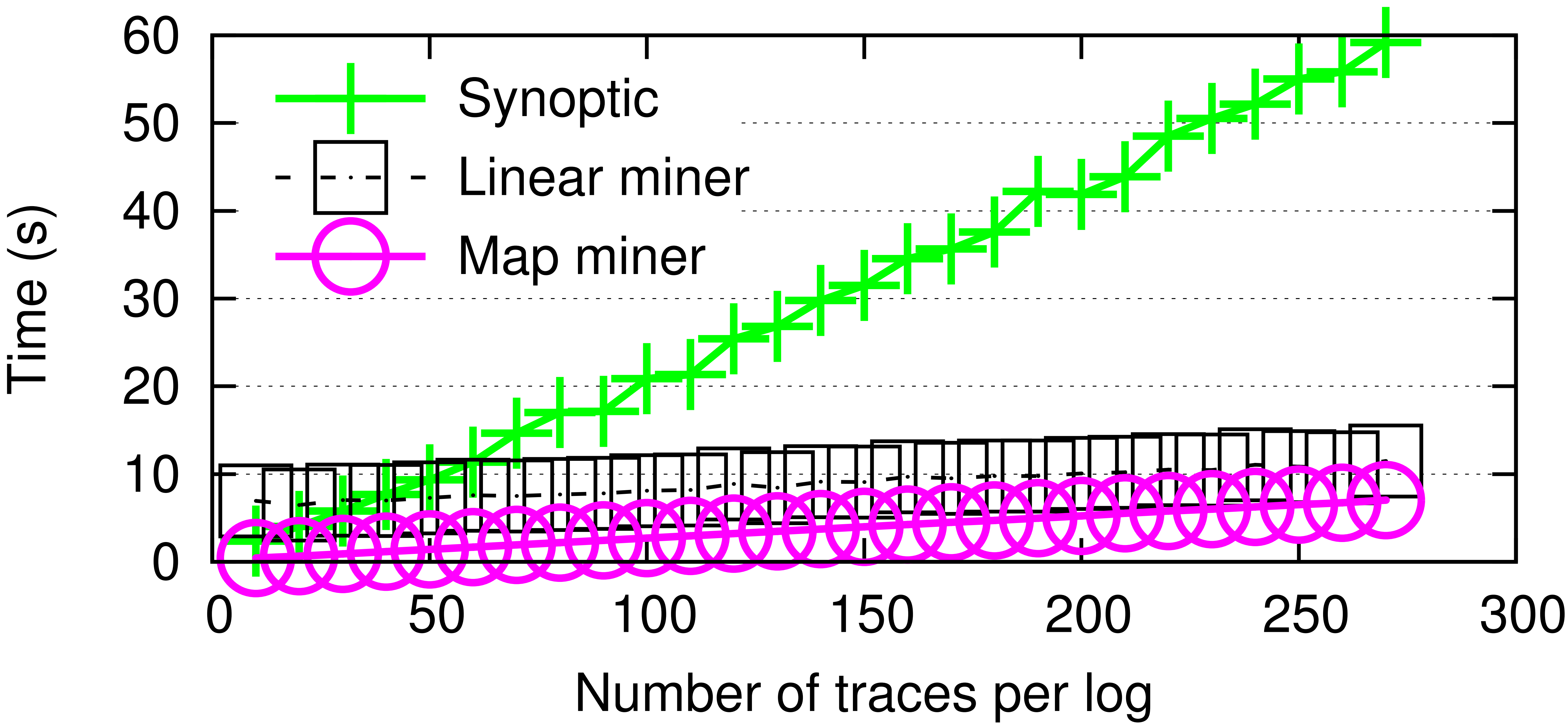
- Compared performance of Texada against Synoptic's miner on three property types
  - x always followed by y : $G(x \to XF\ y)$
  - x never followed by y : $G(x \to G(\neg y))$
  - x always precedes y : $F\ y \to (\neg y\ U\ x)$
  - x immediately followed by y : $G(x \to Xy)$

An optimized Java miner for these property types

- Synthetic logs, uniformly randomly distributed events
- Average tool runtime over 5 executions on log input

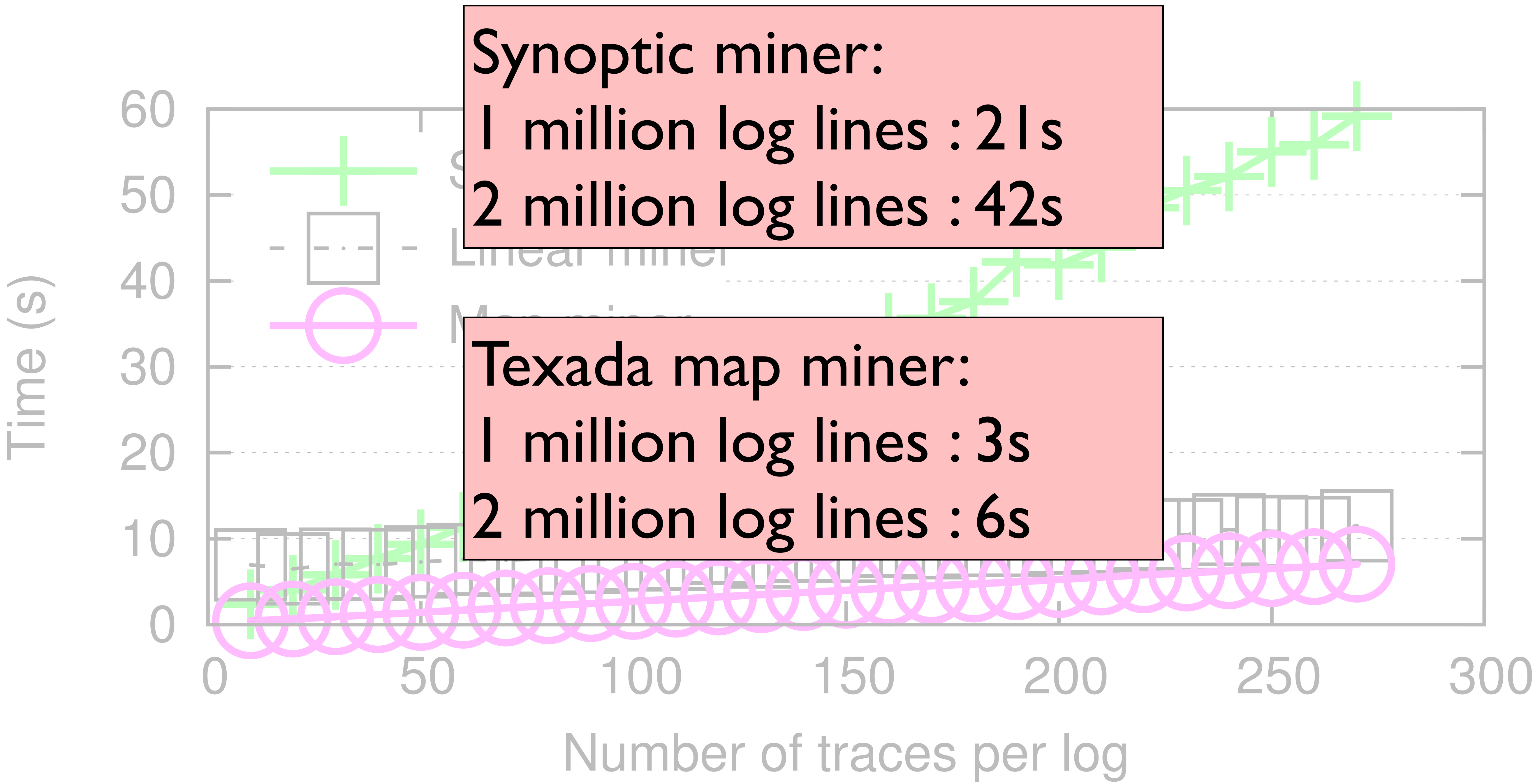
Eval: vary number of traces

- 10K events/trace, 50 event types



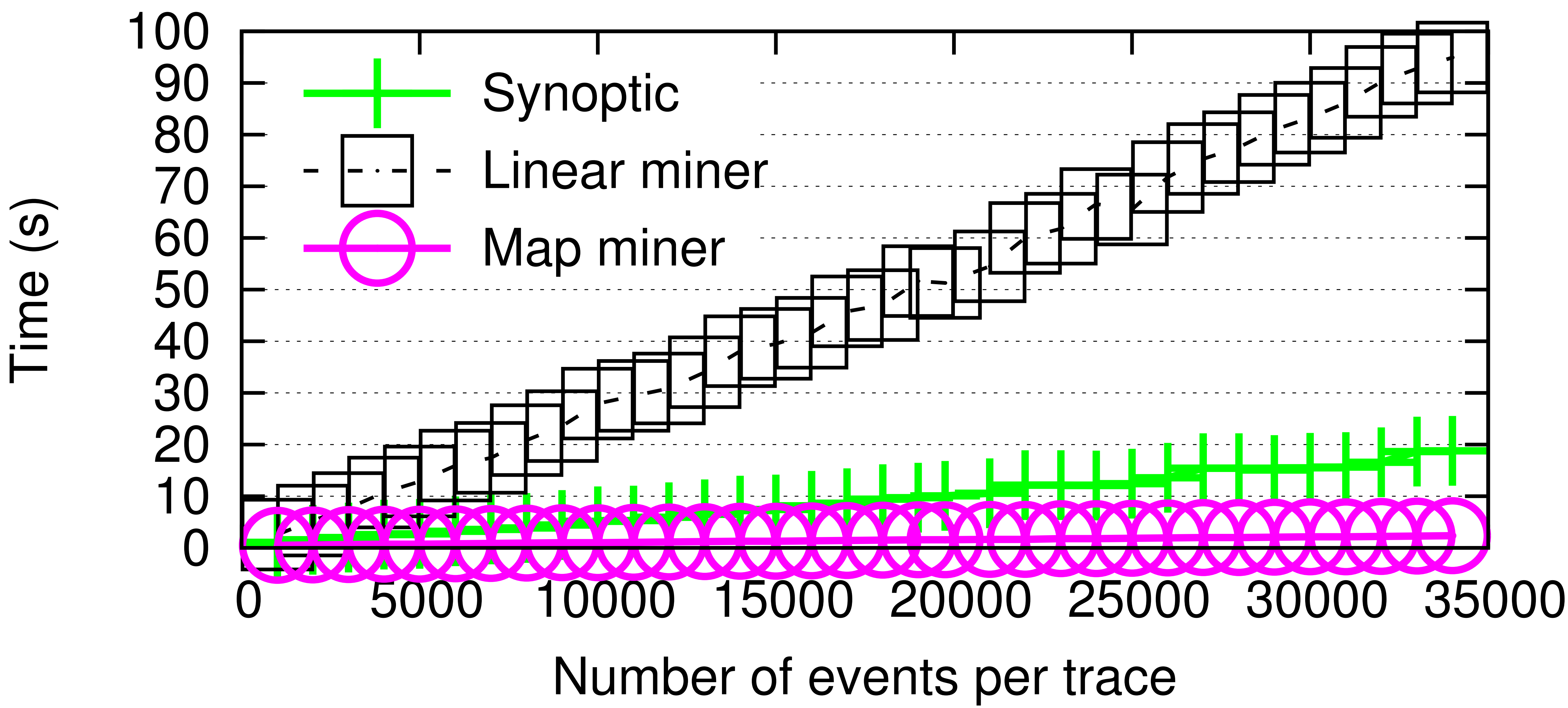
Eval: vary number of traces

- 10K events/trace, 50 event types



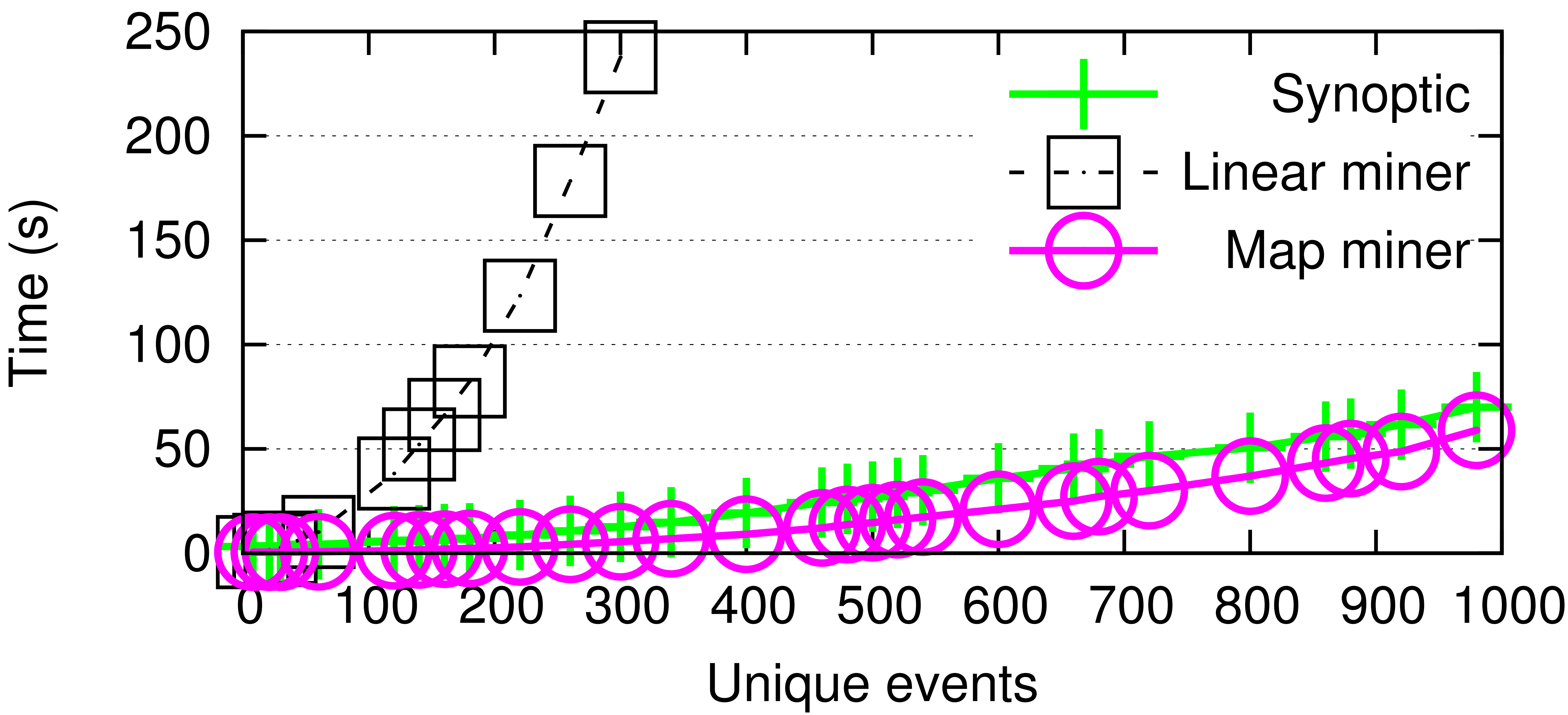
Eval: vary trace length

- 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 [Ghezzi et al. ICSE 2014] [Ohmann et al. ASE 2014]
  - 12K events, 13 event types
  - Use a subset of the property types from [Dwyer et al. ICSE 1999]
  - Texada's runtime < 1s

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.

$$G((contacts \wedge \neg homepage \wedge F\, homepage) \rightarrow (\neg search \; U \; homepage))$$

$$G((\neg homepage \wedge news\_page \wedge F\, homepage) \rightarrow (\neg search \; U \; homepage))$$

Support/confidence in LTL mining

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

Default settings → (supp. 0, conf. 1)

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