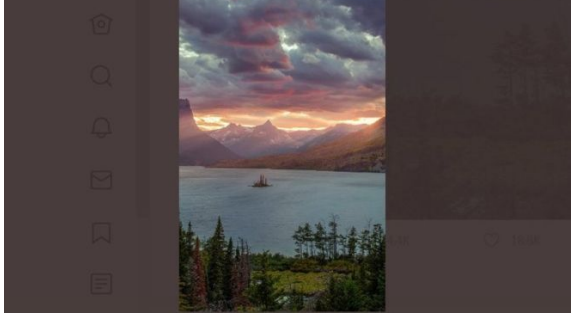
Lightweight Verification via Specialized Typecheckers

Martin Kellogg
University of Washington

Bugs in software

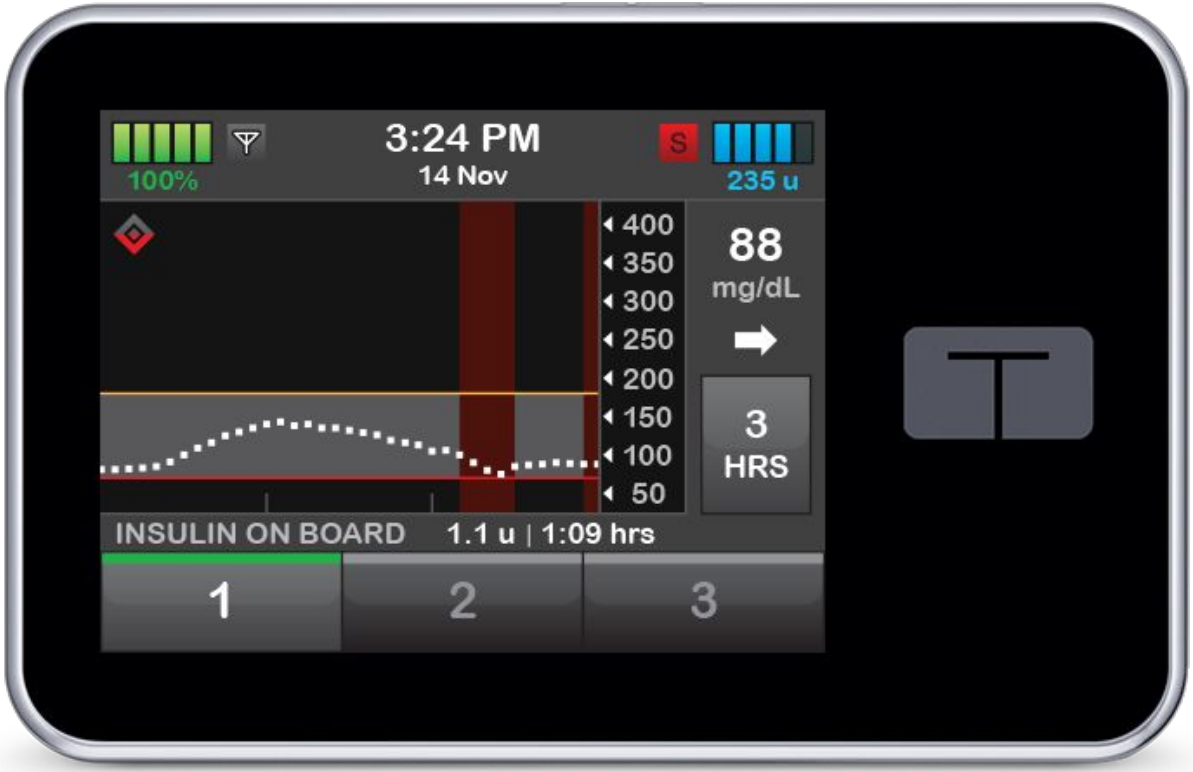


Hours	Seconds	Calculation Time (seconds)	Inaccuracy (seconds)	Approximate Shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0275	55
20^1	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100^2	360000	359999.6567	.3433	687

EQUIFAX

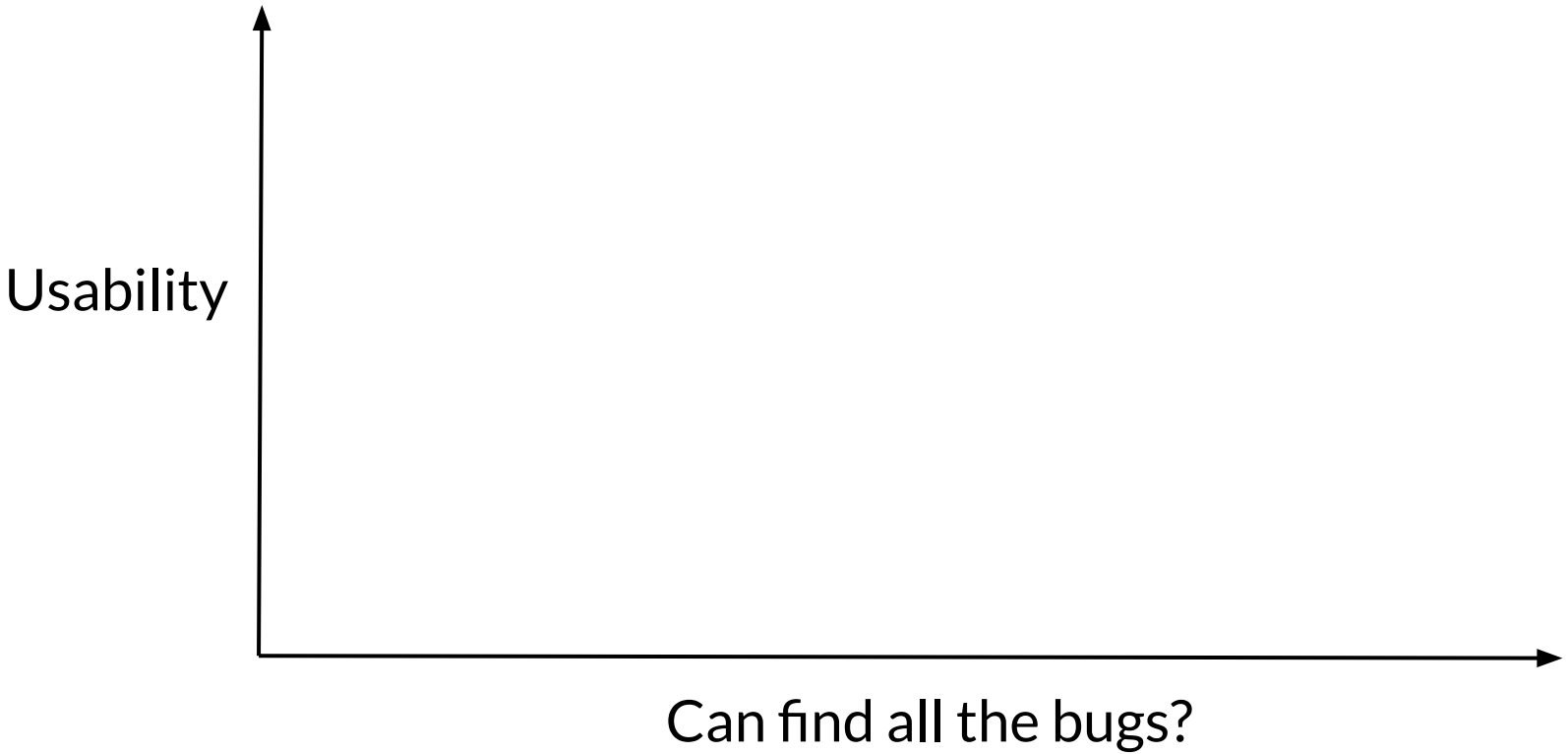
etc.

Bugs in software

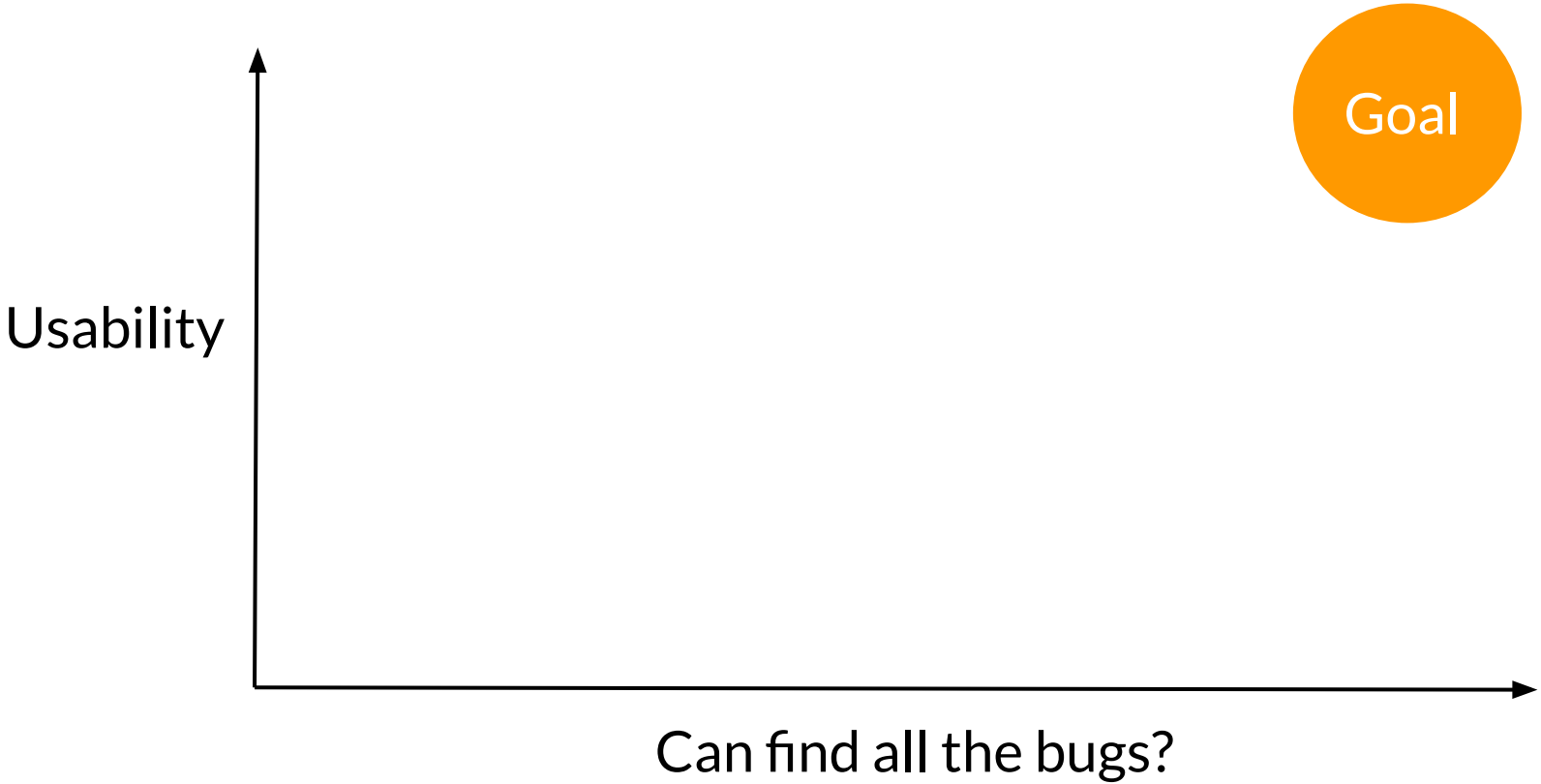


Goal: every developer uses verification

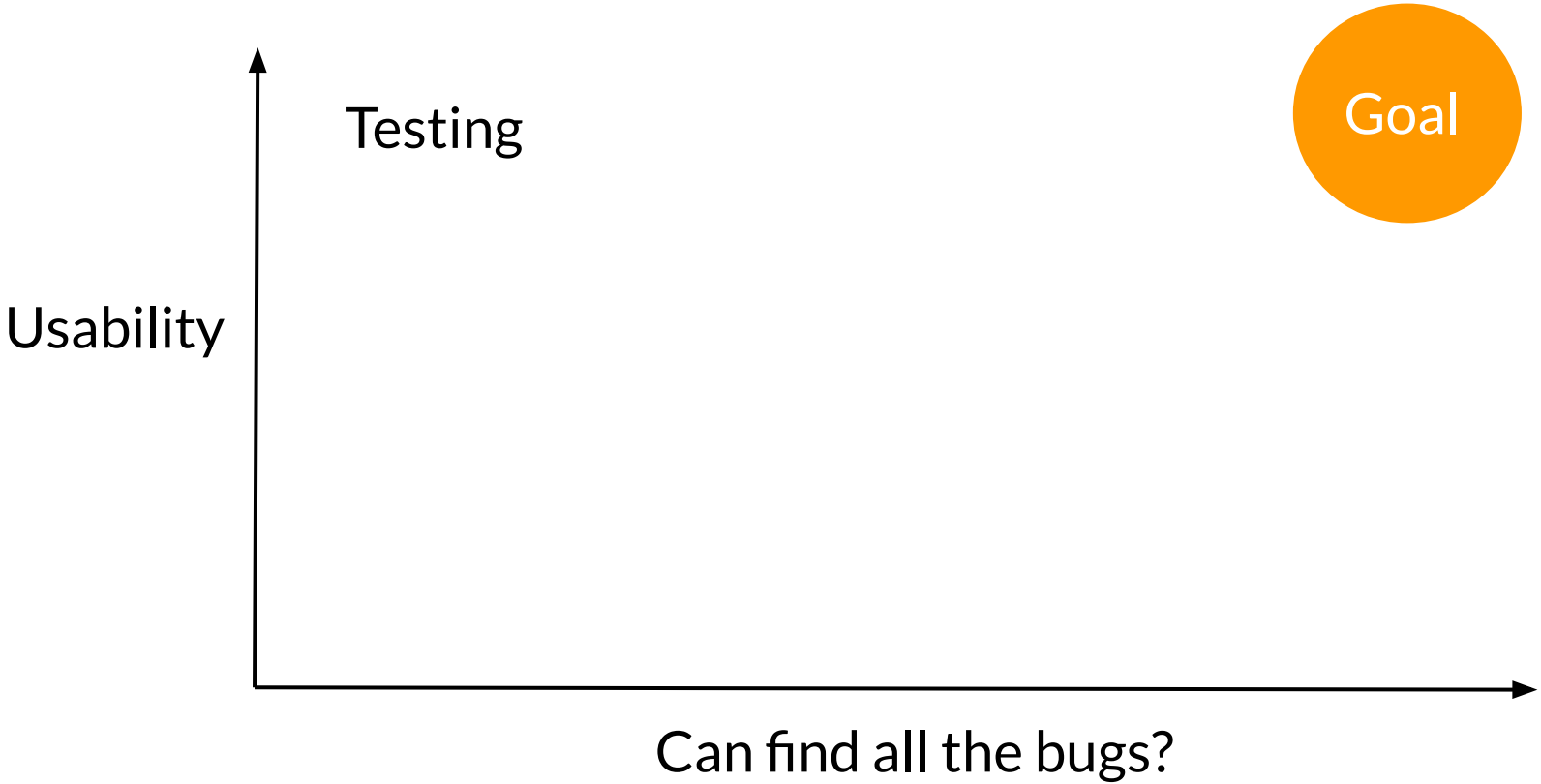
Preventing bugs: a gross oversimplification



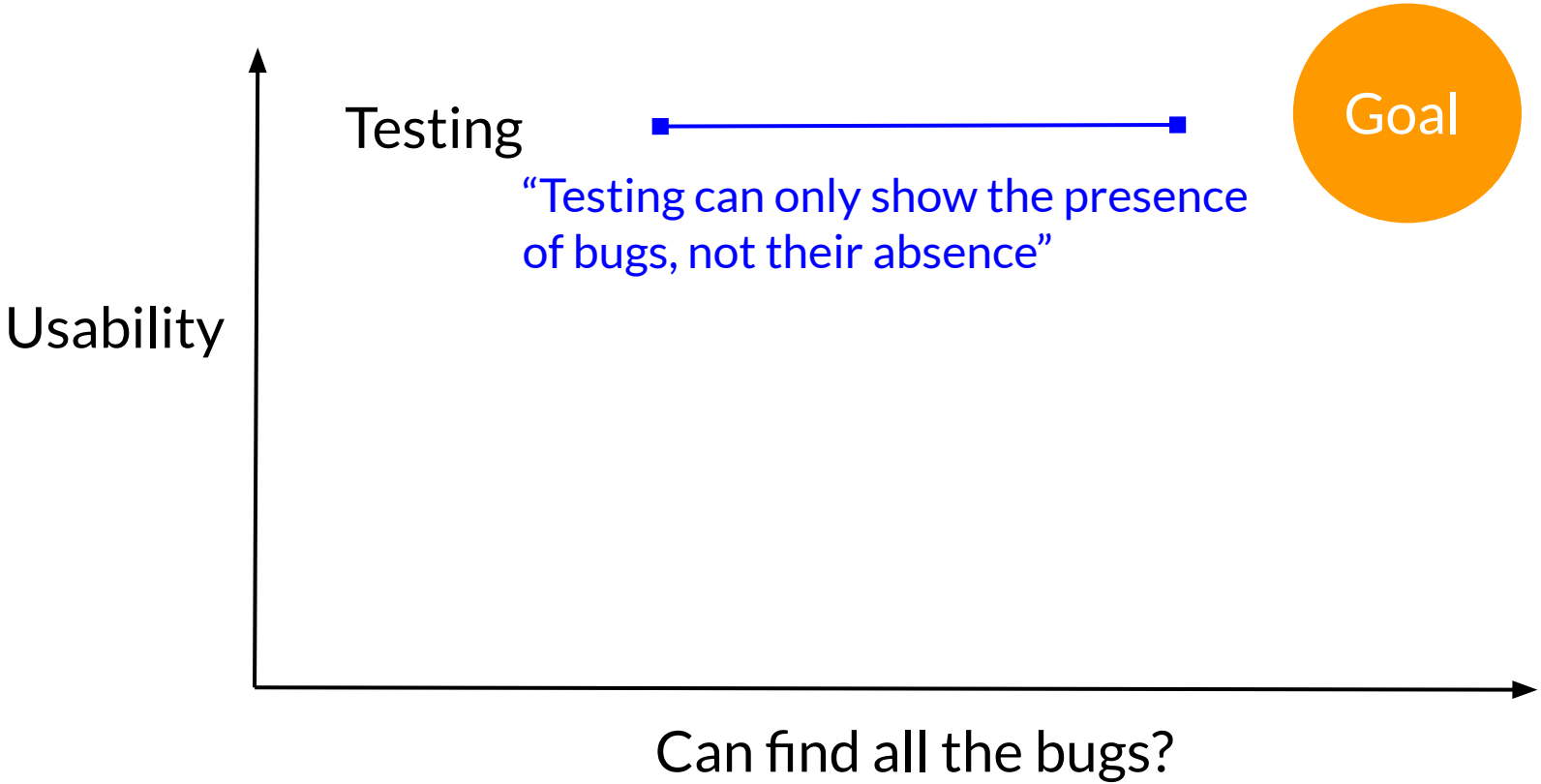
Preventing bugs: a gross oversimplification



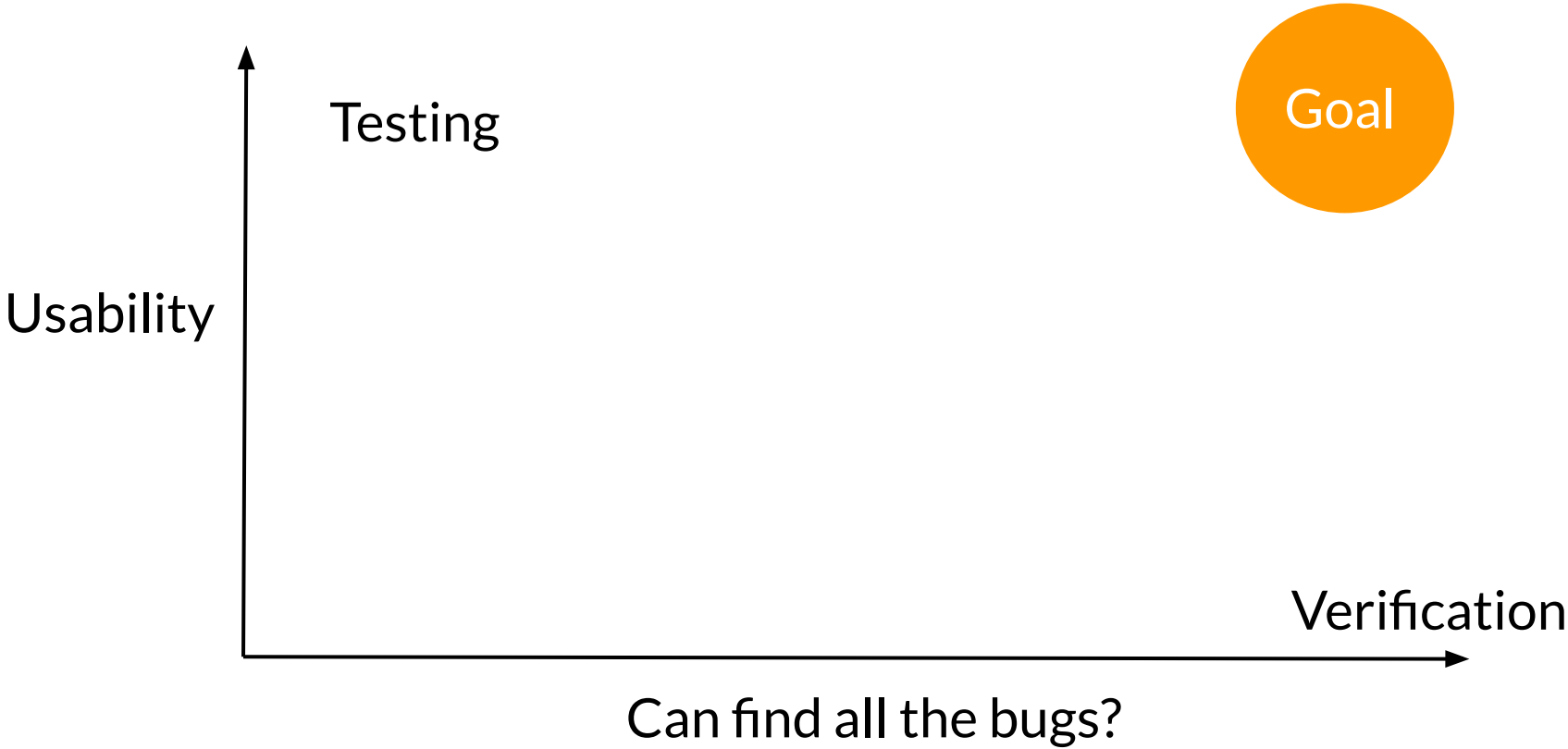
Preventing bugs: a gross oversimplification



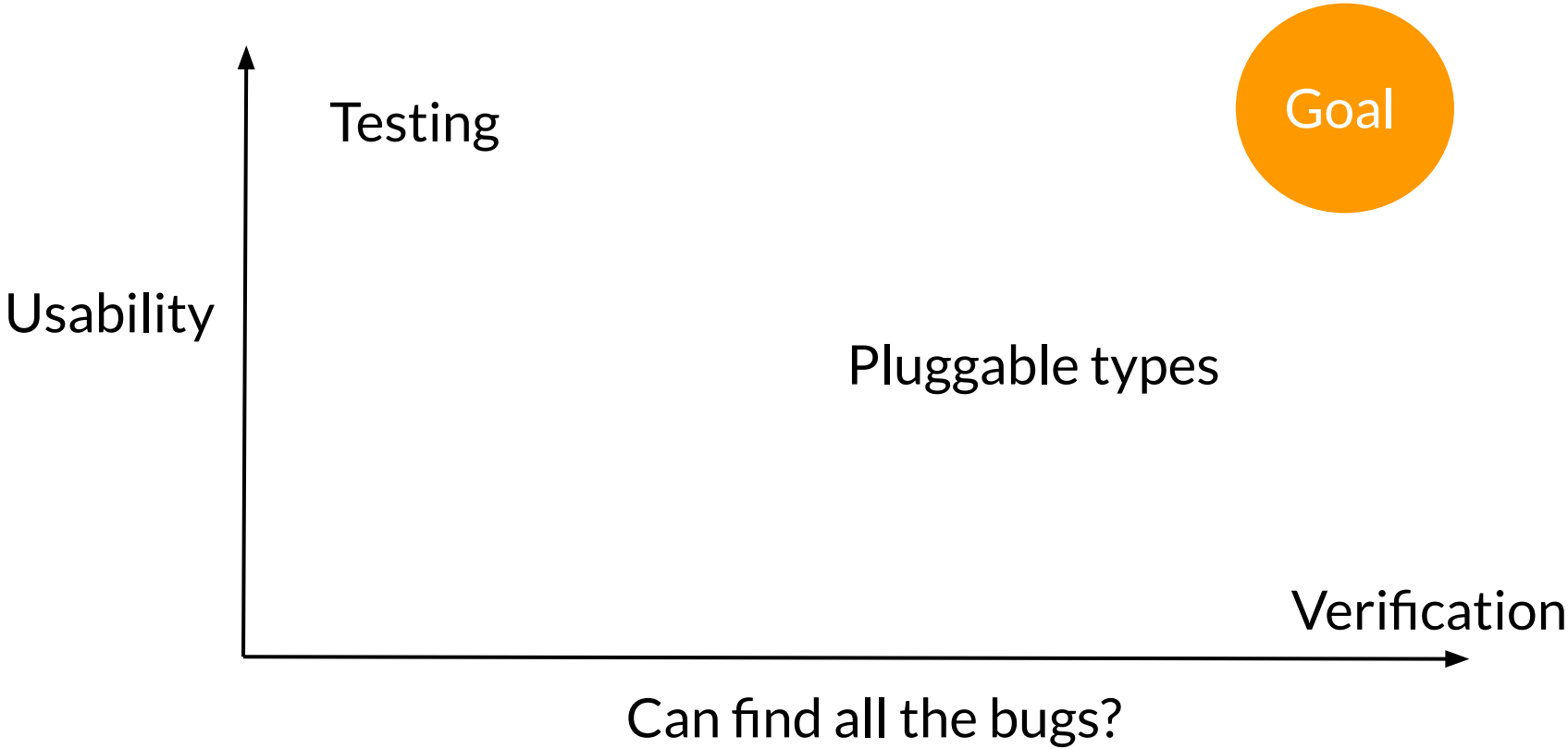
Preventing bugs: a gross oversimplification



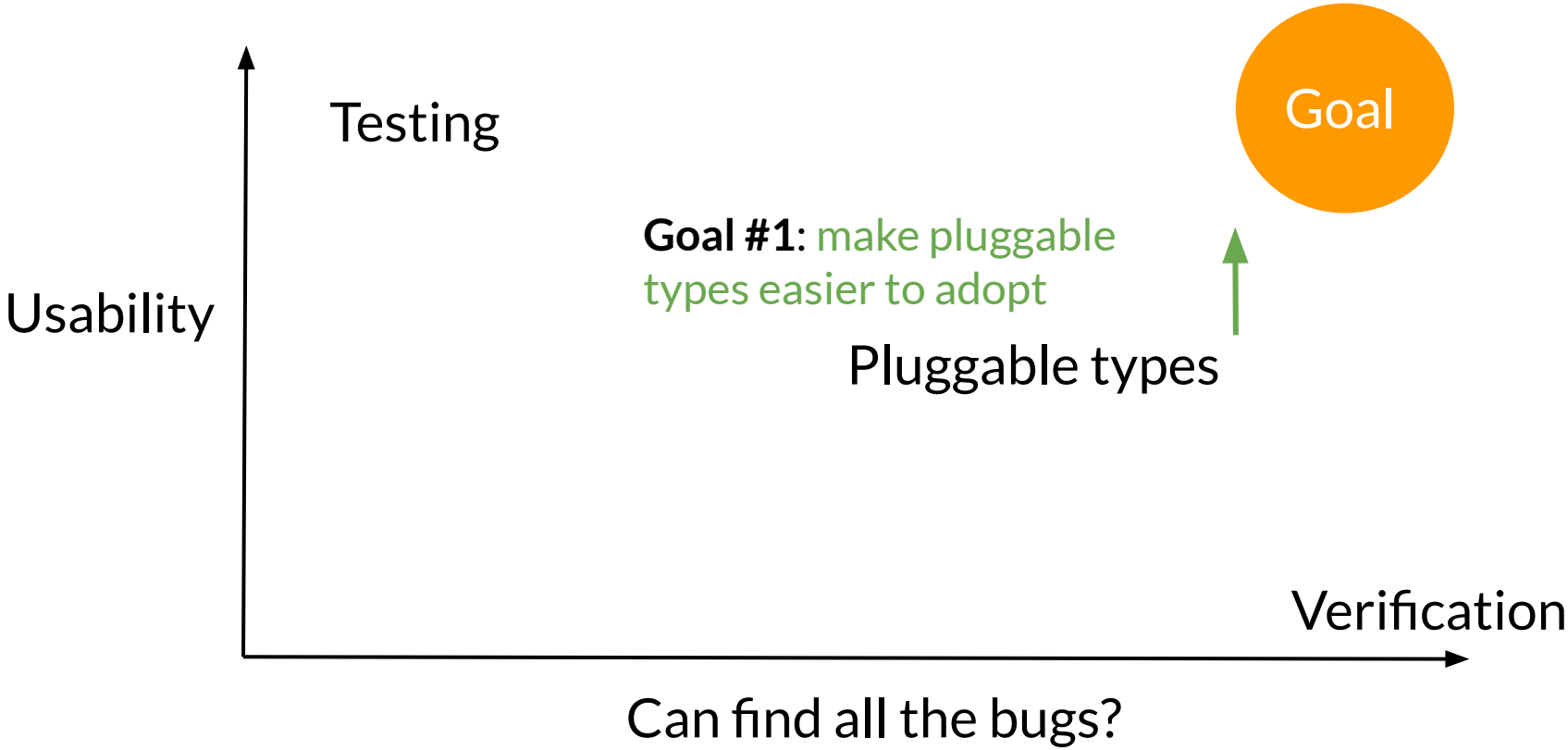
Preventing bugs: a gross oversimplification



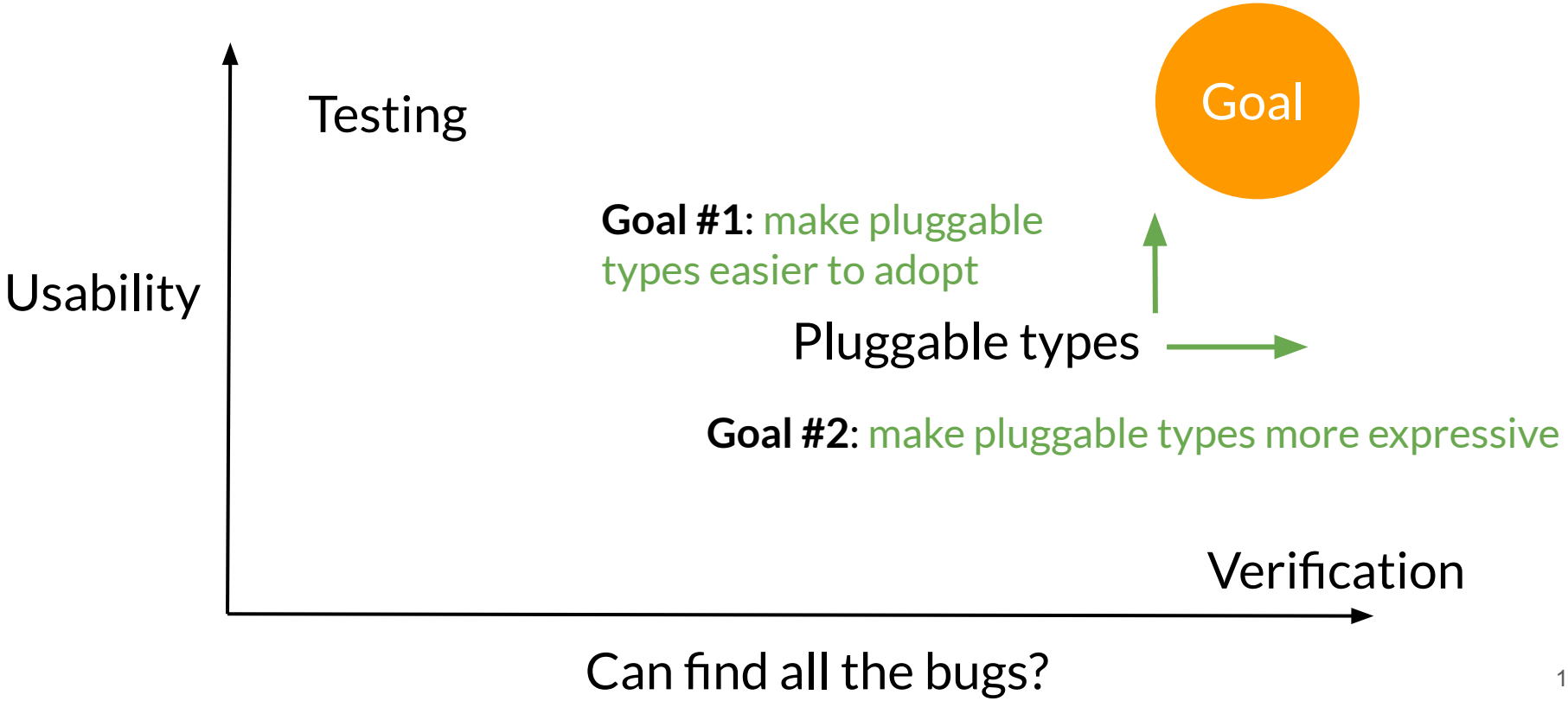
Preventing bugs: a gross oversimplification



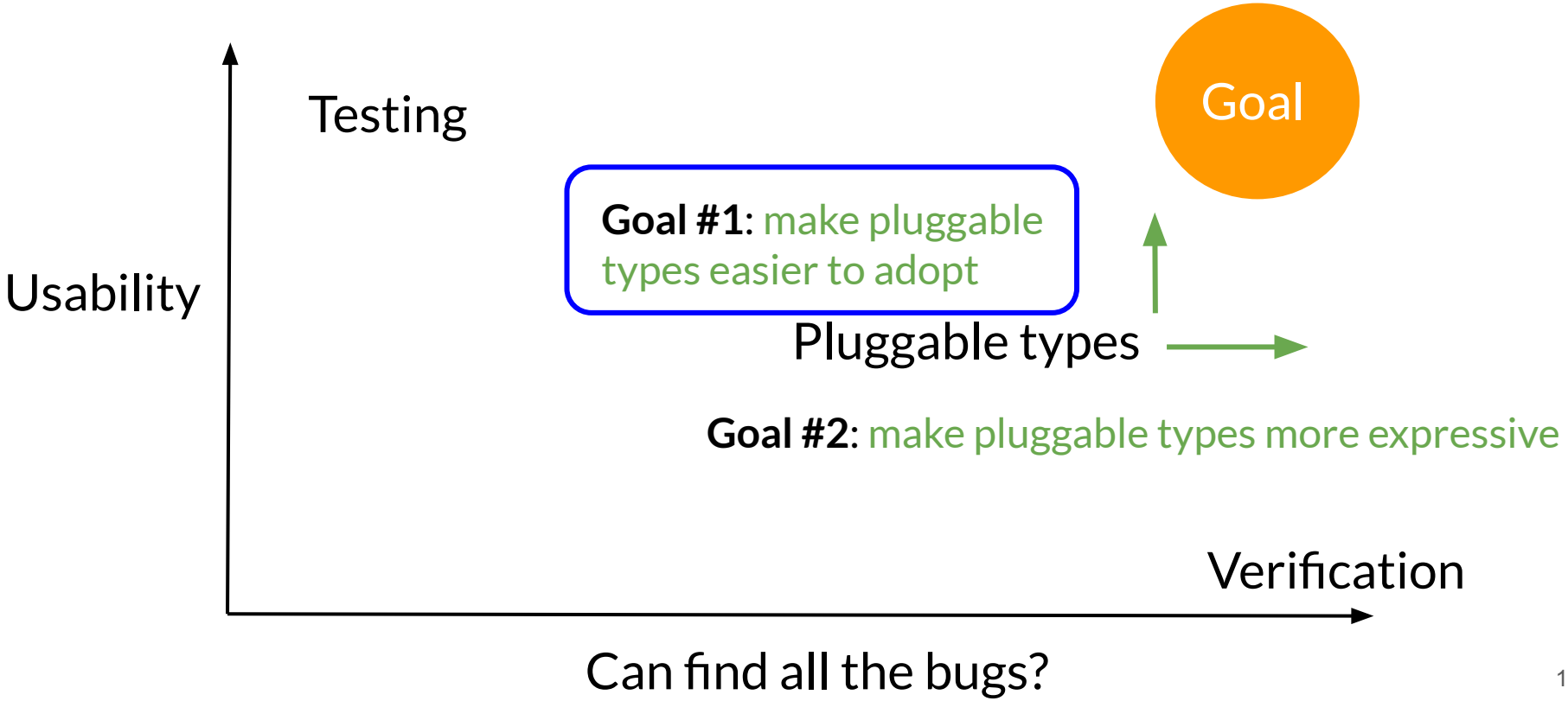
Preventing bugs: a gross oversimplification



Preventing bugs: a gross oversimplification



Preventing bugs: a gross oversimplification



A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.

A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.
- State-of-the-practice is **manual audits** of source code

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.
- State-of-the-practice is **manual audits** of source code

Developers hate doing this work

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.
- State-of-the-practice is **manual audits** of source code
- **Insight:** specialized checkers can replace manual audits

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

A new domain: compliance

- Certificates that a company follows a ruleset
  - PCI DSS for credit card transactions
  - HIPAA for healthcare information
  - FedRAMP for US government cloud vendors
  - SOC for information security vendors
  - etc.
- State-of-the-practice is **manual audits** of source code
- **Insight:** specialized checkers can replace manual audits

Developers love this, because it saves work

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

Specialized compliance checkers, industry

Run on 76M NCNB LoC

Verified	37,315 pkgs
True pos.	173 pkgs
False pos.	1 pkg

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

Specialized compliance checkers, industry

Only 23 annotations

Run on 76M NCNB LoC

Verified	37,315 pkgs
True pos.	173 pkgs
False pos.	1 pkg

Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

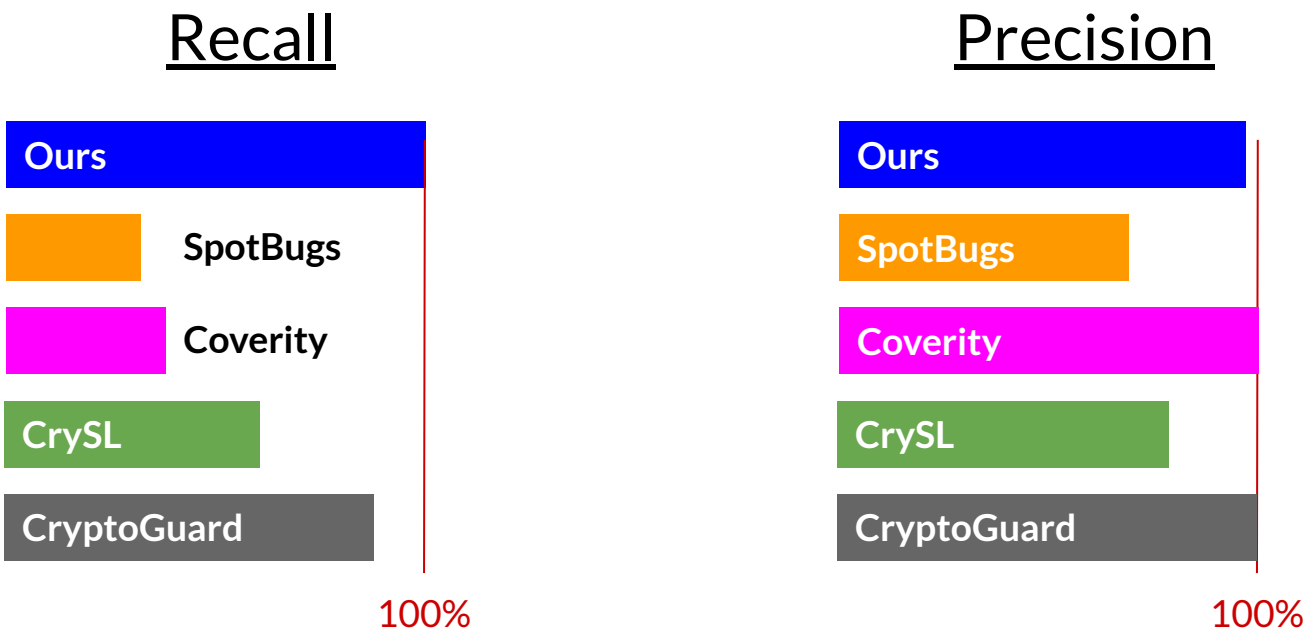
Specialized compliance checkers, industry

Run on 76M NCNB LoC

Verified	37,315 pkgs
True pos.	173 pkgs
False pos.	1 pkg

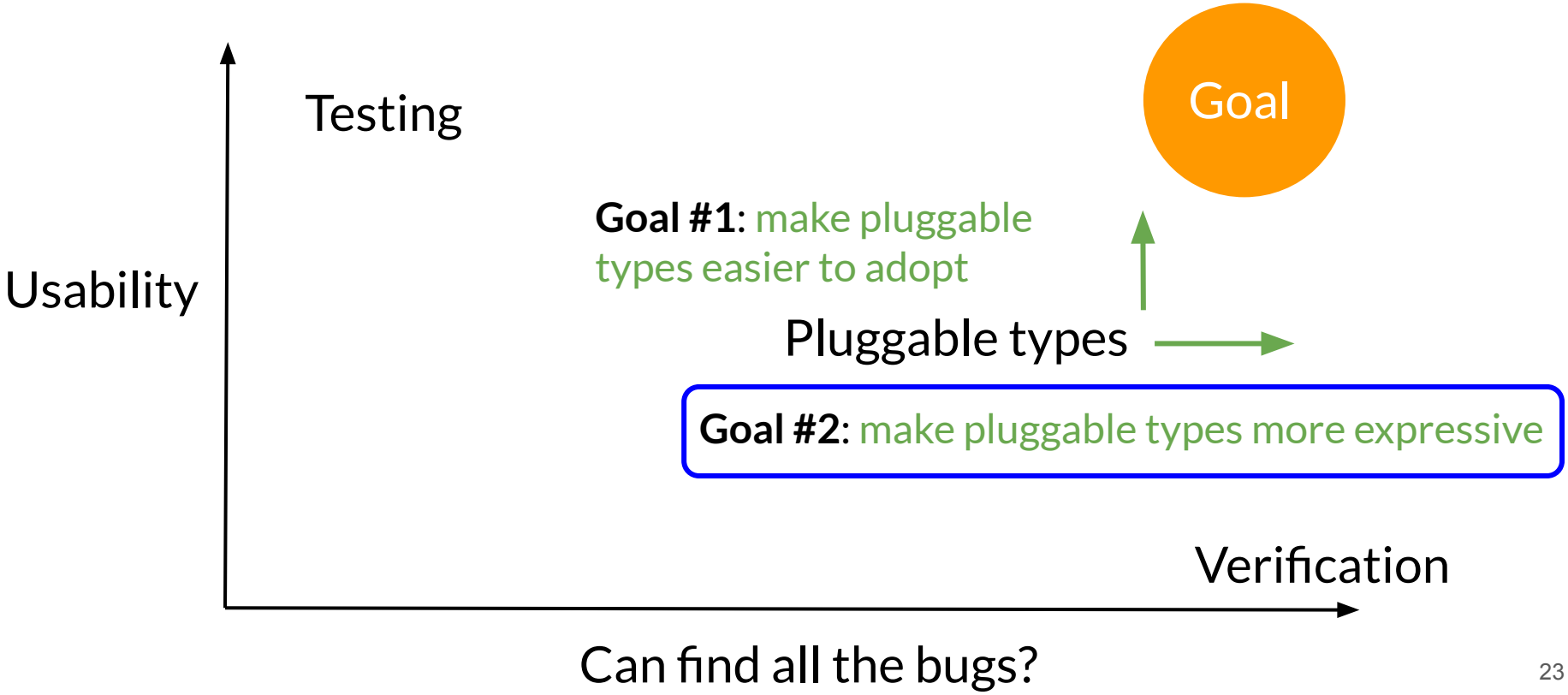
- Auditors accepted output of typecheckers as evidence during a **real audit**
- Checkers **integrated** into build process

Types vs. other approaches



Kellogg, Schaef, Tasiran, Ernst. *Continuous Compliance*. ASE 2020.

Preventing bugs: a gross oversimplification



Harder problem: array indexing

```
T[] a = …;

int i = …;

... a[i] …
```

We need to show that:

- `i` is an index for a

Harder problem: array indexing

```
T[] a = …;

int i = …;

... a[i] …
```

We need to show that:

- ~~`i` is an index for `a`~~
- `i ≥ 0`
- `i < a.length`

Harder problem: array indexing

```
T[] a = …;

int i = …;

... a[i] …
```

Insight: treat array indexing as a **collection of problems**

We need to show that:

- ~~`i` is an index for `a`~~
- `i ≥ 0`
- `i < a.length`

Kellogg, Dort, Millstein, Ernst. *Lightweight Verification of Array Indexing*. ISSTA 2018

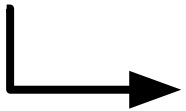
Harder problem: array indexing

```
T[] a = …;

int i = …;

... a[i] …
```

We need to show that:

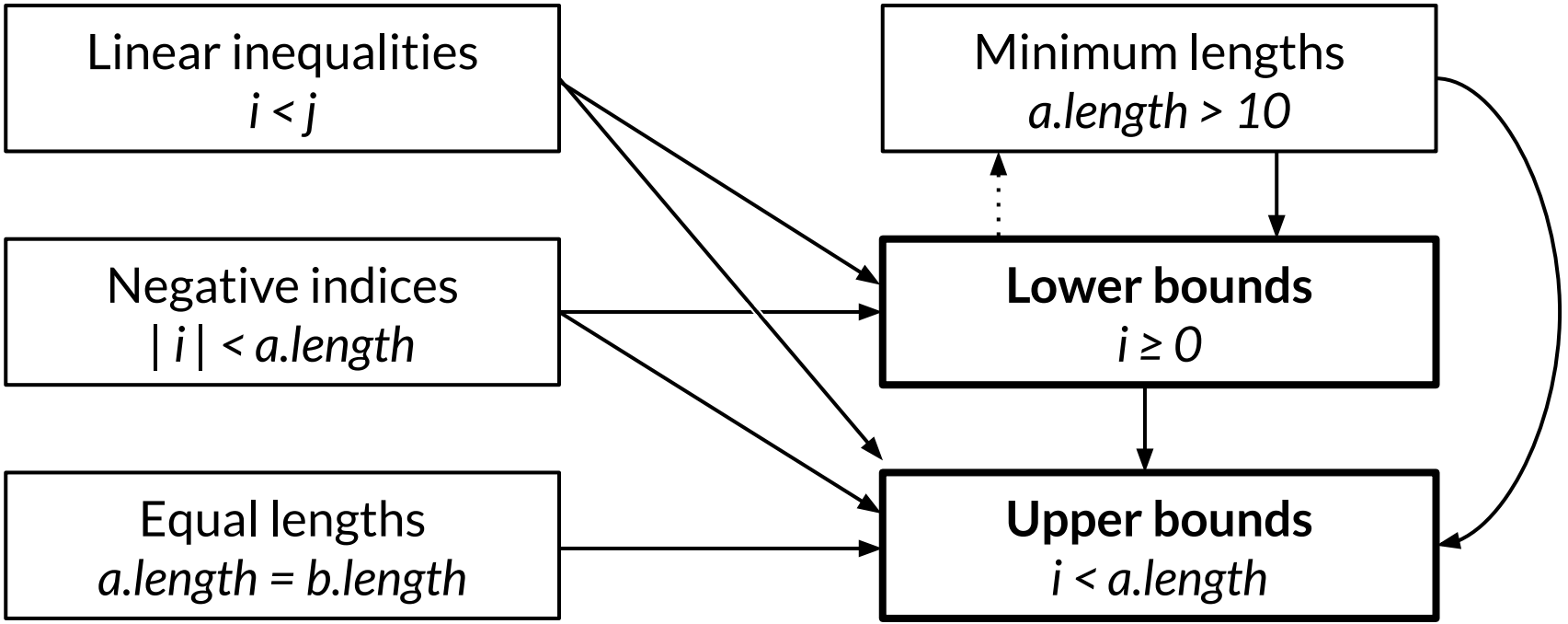
- ~~`i` is an index for `a`~~
- `i ≥ 0`
- `i < a.length`

Insight: treat array indexing as a **collection of problems**



build many analyses instead of just one

Cooperating specialized checkers: array indexing



Kellogg, Dort, Millstein, Ernst. *Lightweight Verification of Array Indexing*. ISSTA 2018

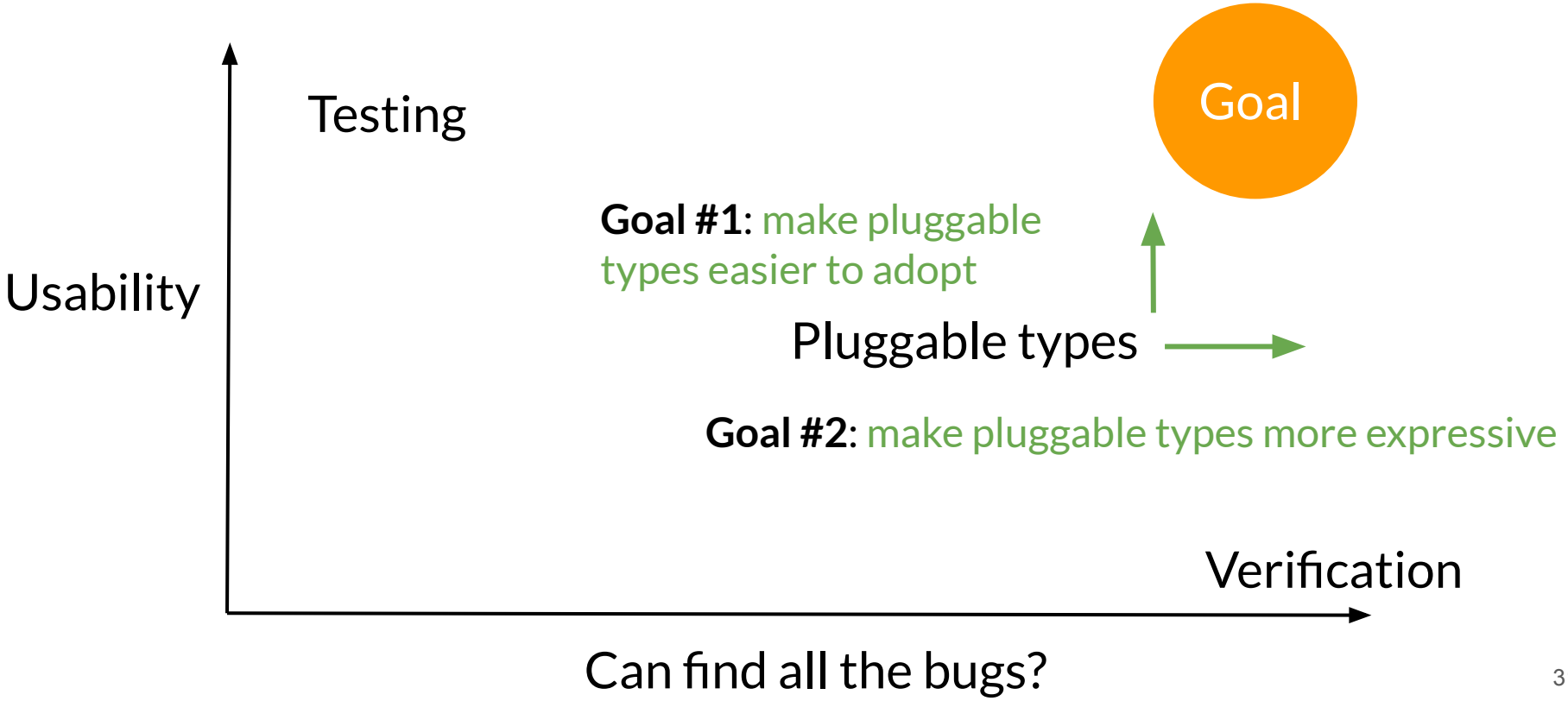
Summary of results

- Found bugs in industrial codebases (Google Guava)

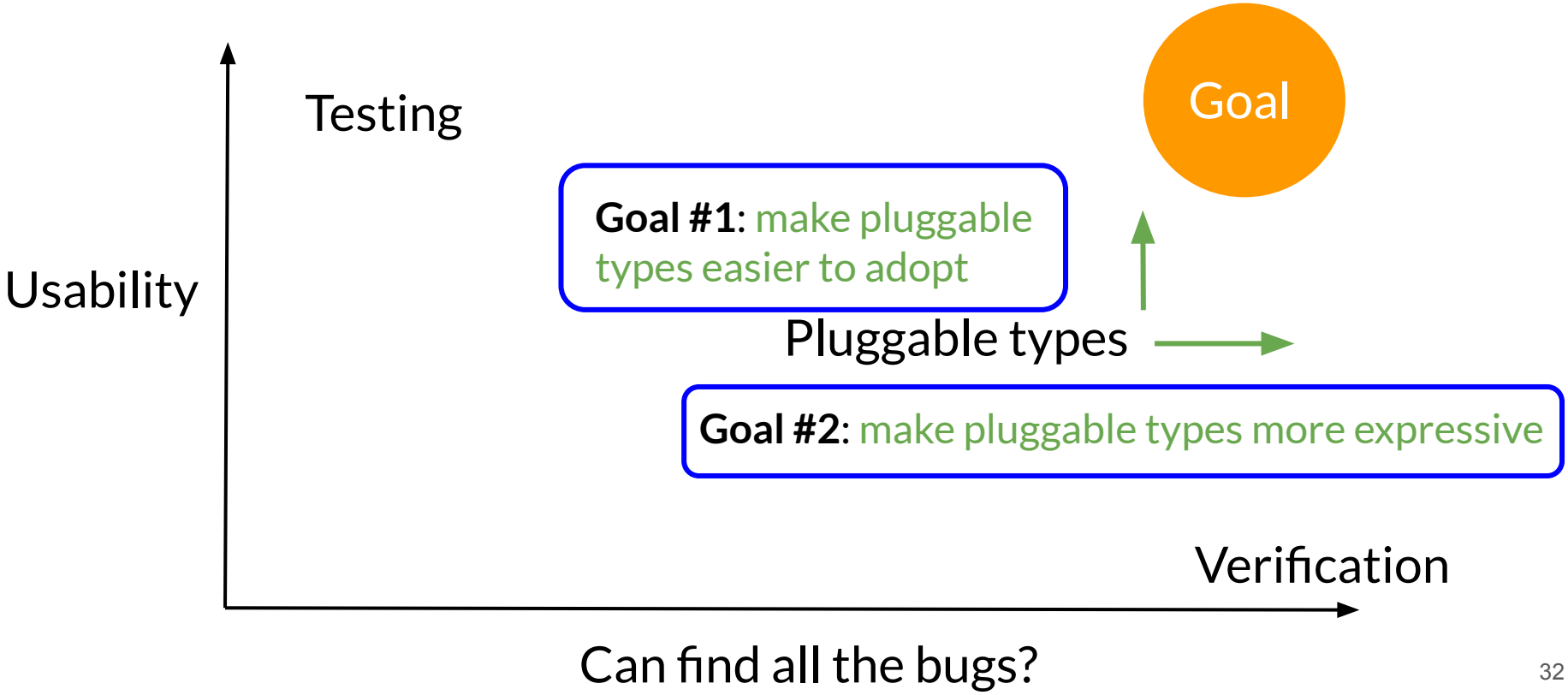
Summary of results

- Found bugs in industrial codebases (Google Guava)
- vs prior verification approaches (KeY, Clousot):
  - **more sound** in microbenchmarks
  - **equally precise** on large codebases
  - **more scalable** - 10 min vs 3 hrs to check 100k LoC

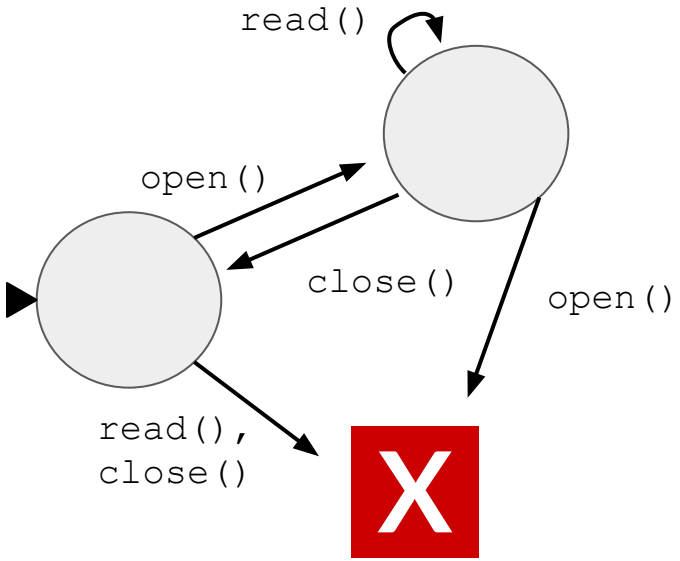
Preventing bugs: a gross oversimplification



Preventing bugs: a gross oversimplification

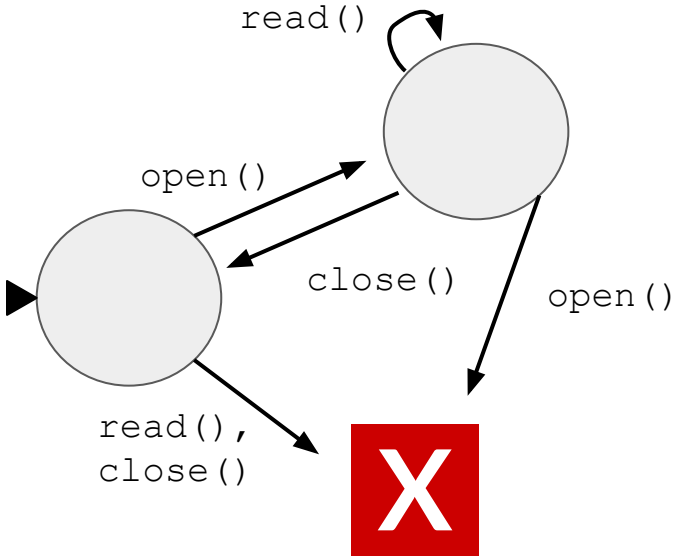


Typestate analysis



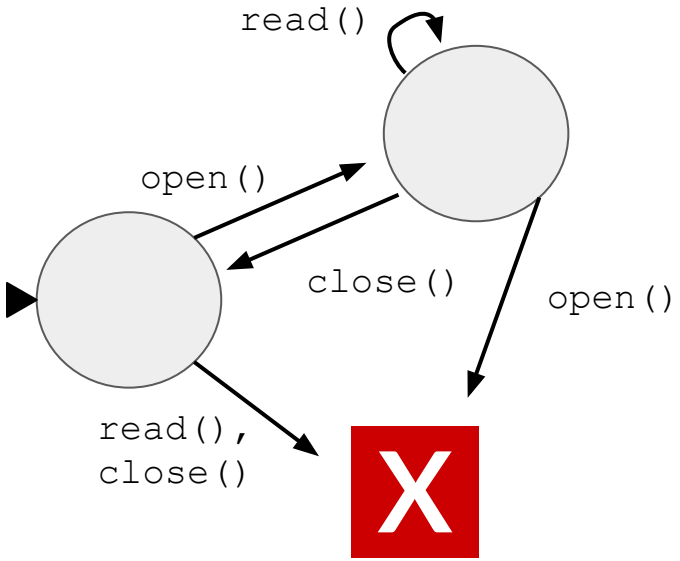

```
File f = …;
f.open();
File f2 = f;
f.close();
f2.read();
```

Typestate analysis




```
File f = …;
f.open();
File f2 = f;
f.close();
f2.read();
```
X

Typestate analysis




```
File f = …;
f.open();
File f2 = f;
f.close();
f2.read();
```

Aliasing!

X

The builder pattern

```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(generateRandom(32))
            .build();
```

The builder pattern

```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(generateRandom(32))
            .build();
```

The builder pattern

```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(generateRandom(32))
            .build();
```

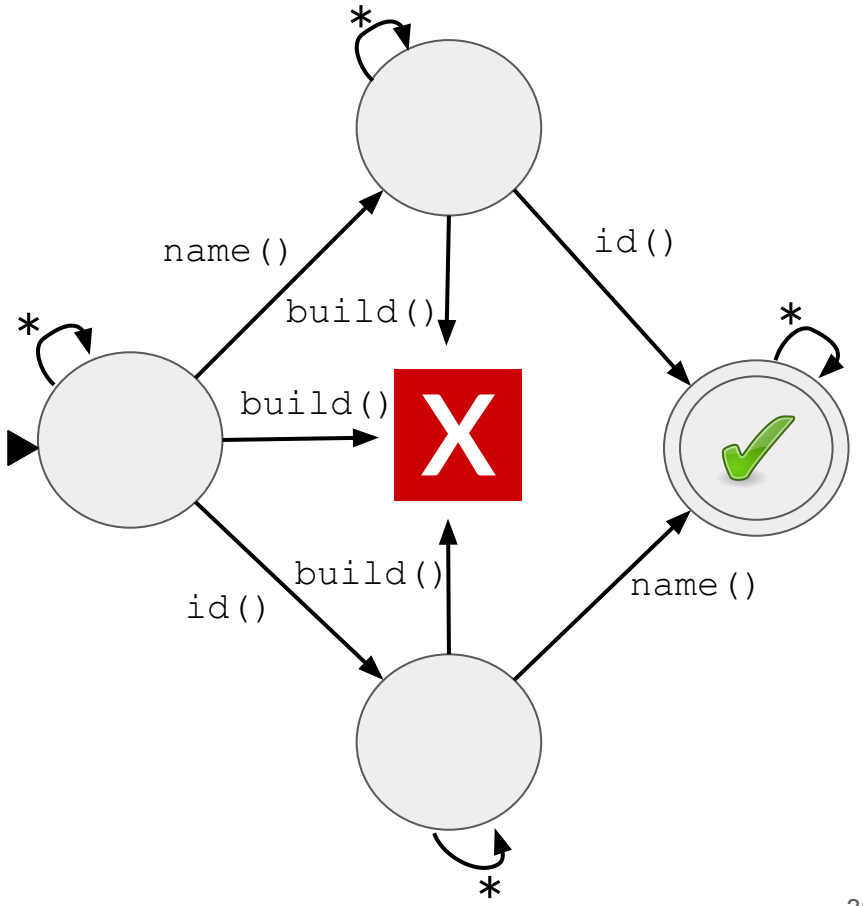
```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .build();
```

Kellogg, Ran, Sridharan, Schaef, Ernst. *Verifying Object Construction*. ICSE 2020.

The builder pattern

```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(generateRandom(32))
            .build();
```

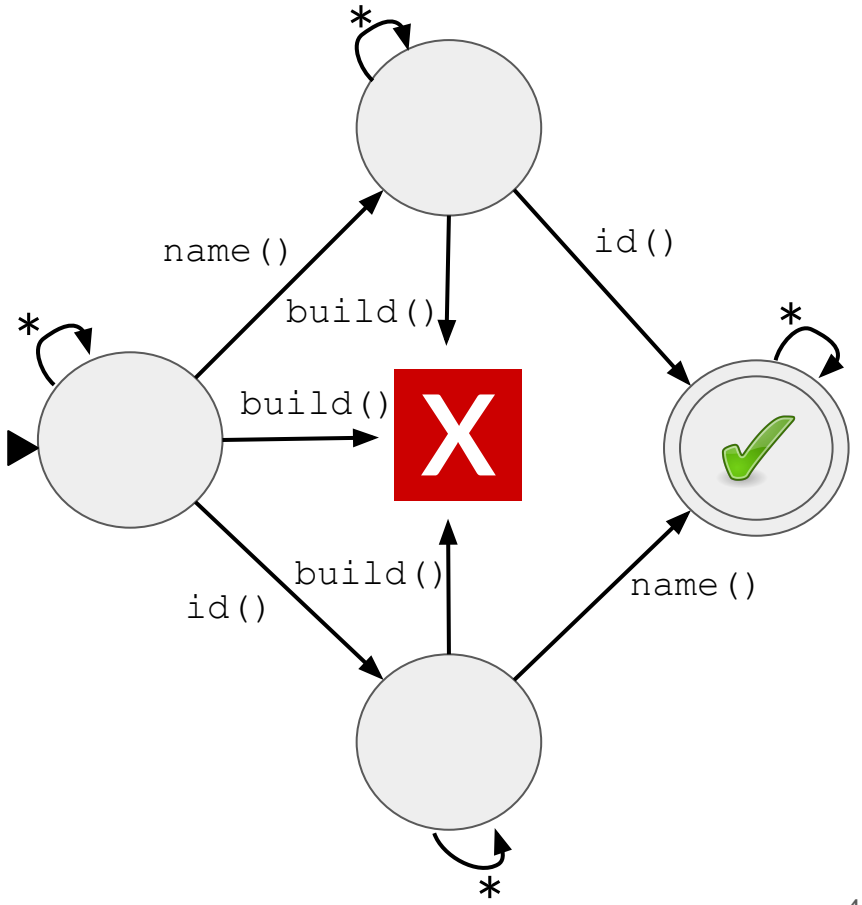
```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .build();
```



Kellogg, Ran, Sridharan, Schaef, Ernst. *Verifying Object Construction*. ICSE 2020.

The builder pattern

Key insight:
No loops in this FSM!
(except self loops)

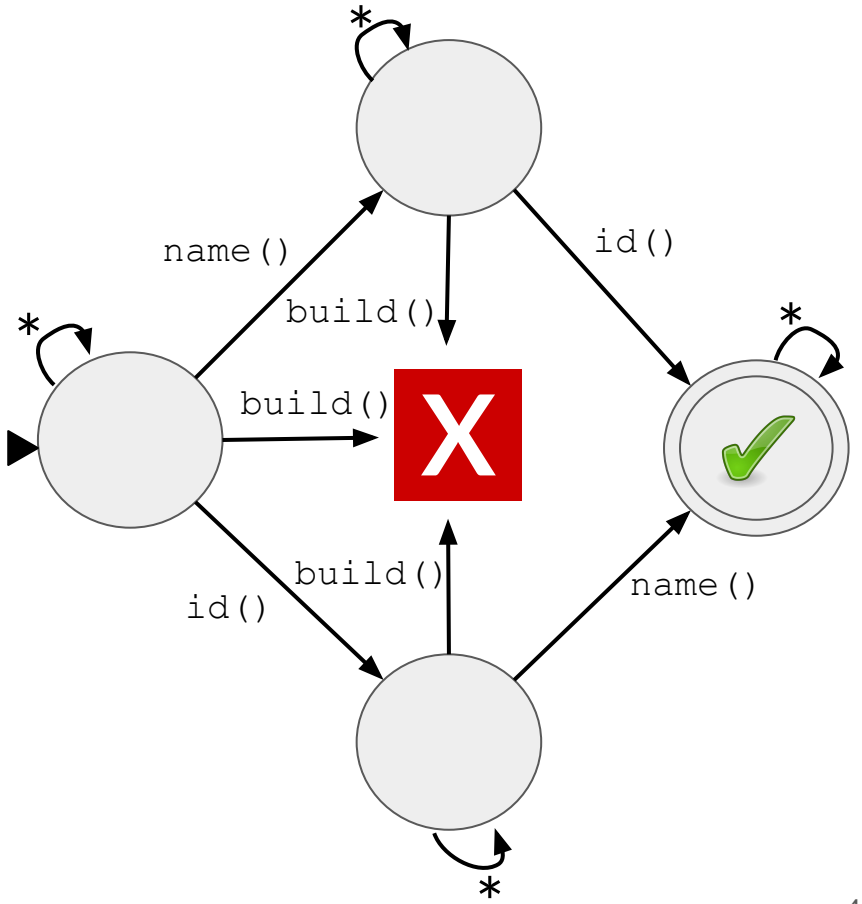


Kellogg, Ran, Sridharan, Schaef, Ernst. *Verifying Object Construction*. ICSE 2020.

The builder pattern

“accumulation analysis”

Key insight:
No loops in this FSM!
(except self loops)



Kellogg, Ran, Sridharan, Schaef, Ernst. *Verifying Object Construction*. ICSE 2020.

Accumulation analysis

*A typestate analysis whose state representation is a **monotonically-increasing set**.*

Kellogg, Ran, Sridharan, Schaef, Ernst. *Verifying Object Construction*. ICSE 2020.

Accumulation analysis

*A typestate analysis whose state representation is a **monotonically-increasing set**.*

Advantages:

- Does **not** require alias analysis for soundness
- **Modular**

User study

Task: add a new required field to a builder

Results:

- +50% **success rate**
- ~50% **faster**

Accumulation for resource leaks

```
try {
    Socket s = new Socket(address, port);
    ...
    s.close();
} catch (IOException e) {

}
```

Accumulation for resource leaks

```
try {
    Socket s = new Socket(address, port);
    ...

} catch (IOException e) {


}
```

Missing call to close()

Accumulation for resource leaks

3 stage checker:

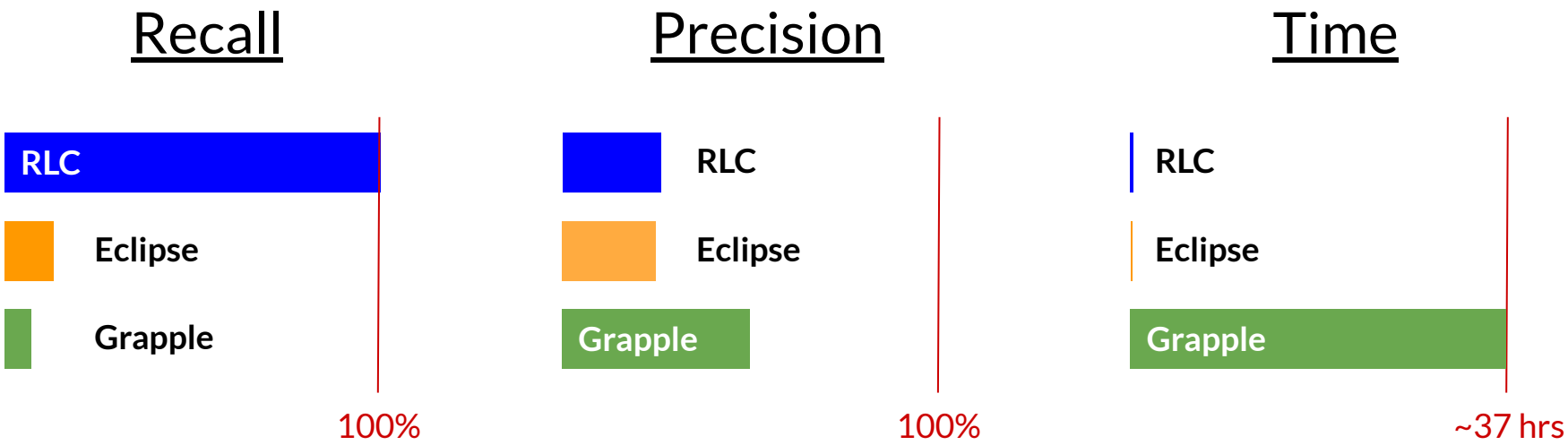
1. taint-tracker over-approximates methods that **need to be called**
2. accumulation under-approximates methods that **have been called**
3. dataflow analysis **compares** the two at “going out-of-scope” points

Accumulation for resource leaks: results

For full results, come to our talk on 26 August, 4pm Athens time ;)

Kellogg, Shadab, Sridharan, Ernst. *Lightweight and Modular Resource Leak Verification*. ESEC/FSE 2021.

Accumulation for resource leaks: results



Accumulation: future plans

Accumulation: future plans

Big question: How much of typestate is accumulation?

How much of typestate is accumulation?

What we know for sure is accumulation:

- **builders**
- **resource leaks**

How much of typestate is accumulation?

Plan #1: survey the literature

How much of typestate is accumulation?

Plan #1: survey the literature

Example: Dwyer, Avrunin, and Corbitt (ICSE 1999) split finite-state properties into 8 patterns

How much of typestate is accumulation?

Plan #1: survey the literature

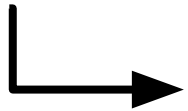
Example: Dwyer, Avrunin, and Corbitt (ICSE 1999) split finite-state properties into 8 patterns

↳ 5/8 can be expressed as accumulation

How much of typestate is accumulation?

Plan #1: survey the literature

Example: Dwyer, Avrunin, and Corbitt (ICSE 1999) split finite-state properties into 8 patterns



5/8 can be expressed as accumulation

60% of specifications they found in the wild!

How much of typestate is accumulation?

Plan #1: survey the literature

Plan #2: look for real problems solved with typestate

How much of typestate is accumulation?

Plan #1: survey the literature

Plan #2: look for real problems solved with typestate

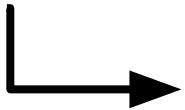
Example: Qi & Myers, POPL 2009 introduced “masked types” for safe object initialization

How much of typestate is accumulation?

Plan #1: survey the literature

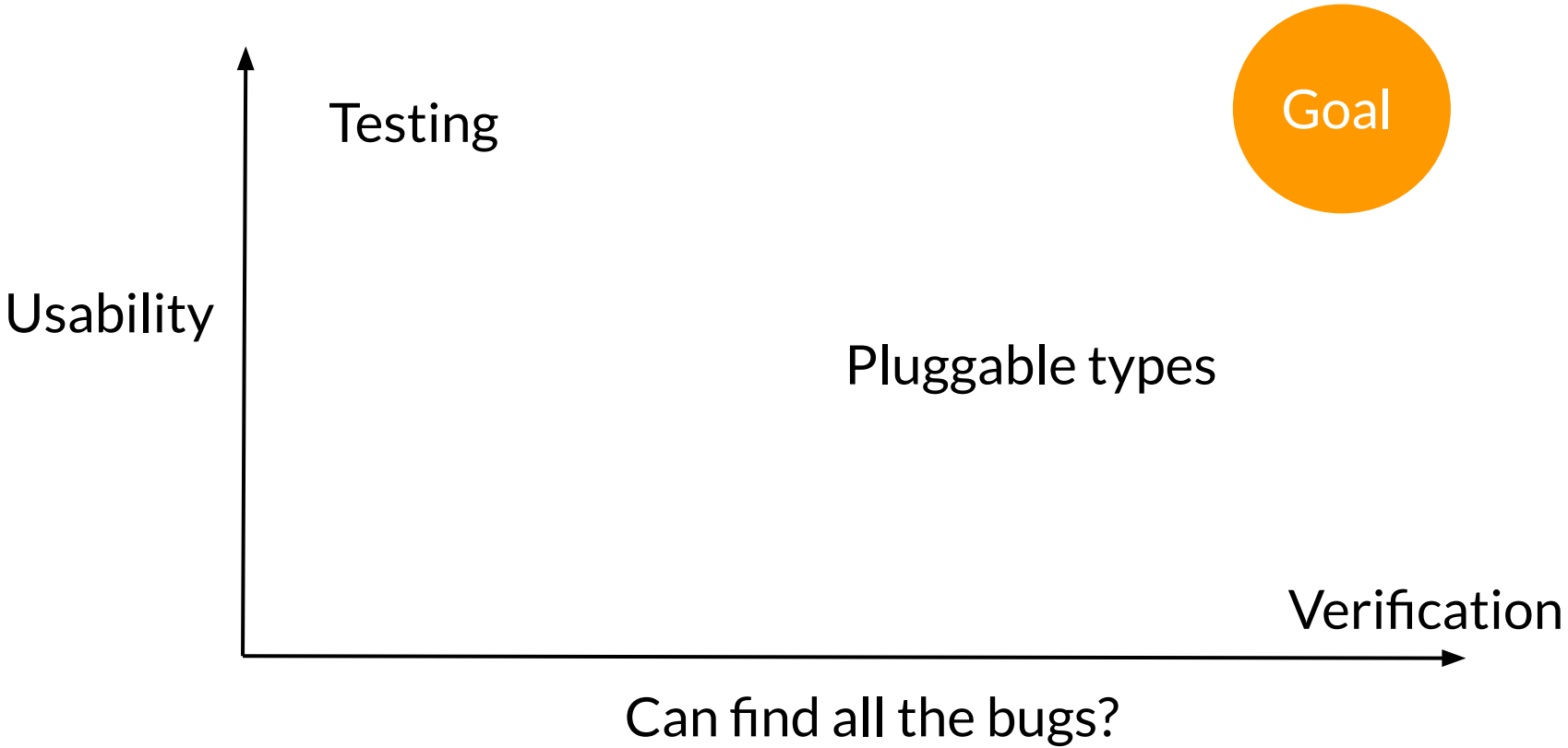
Plan #2: look for real problems solved with typestate

Example: Qi & Myers, POPL 2009 introduced “masked types” for safe object initialization

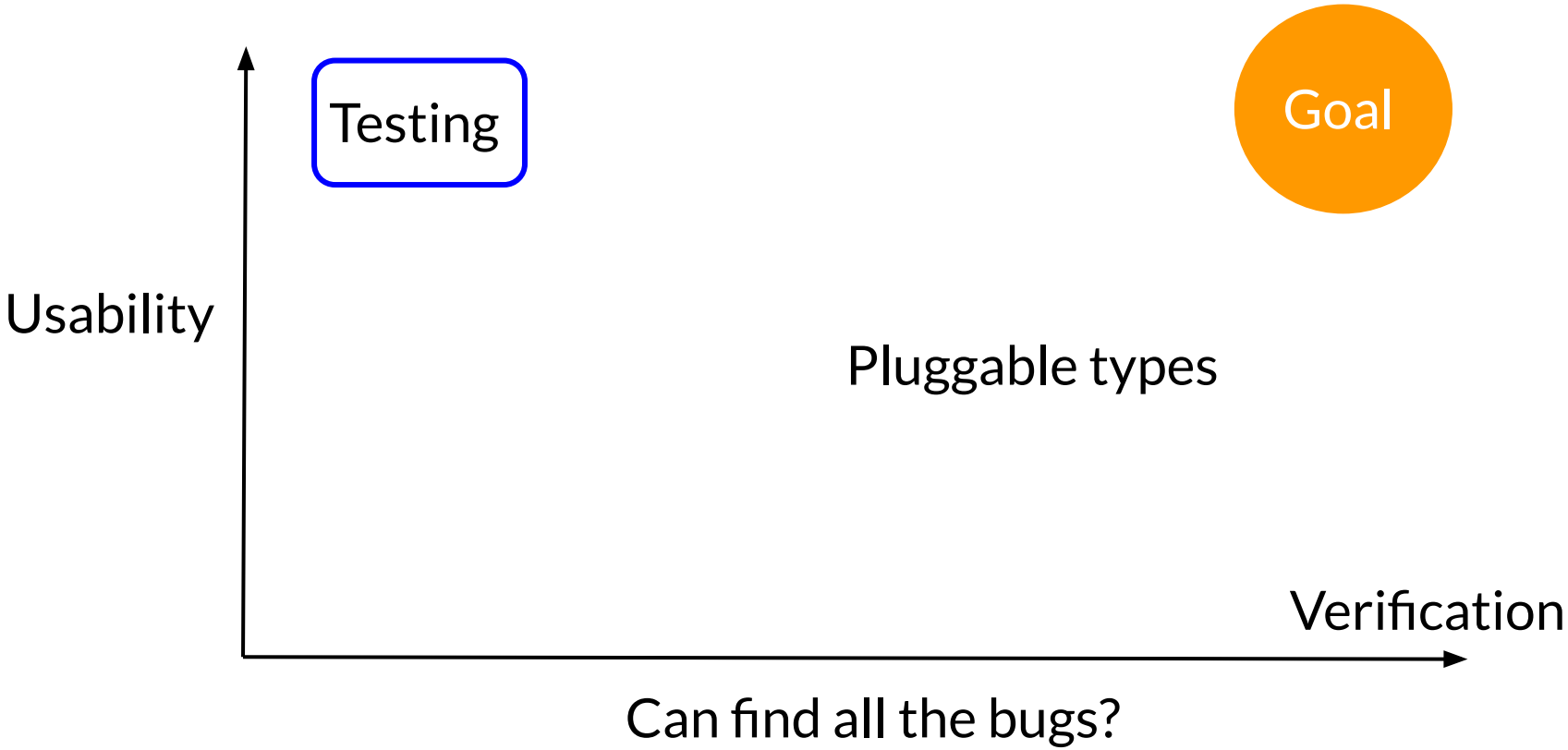


Masked types are an accumulation analysis that accumulates **fields** rather than method calls

Related work



Related work

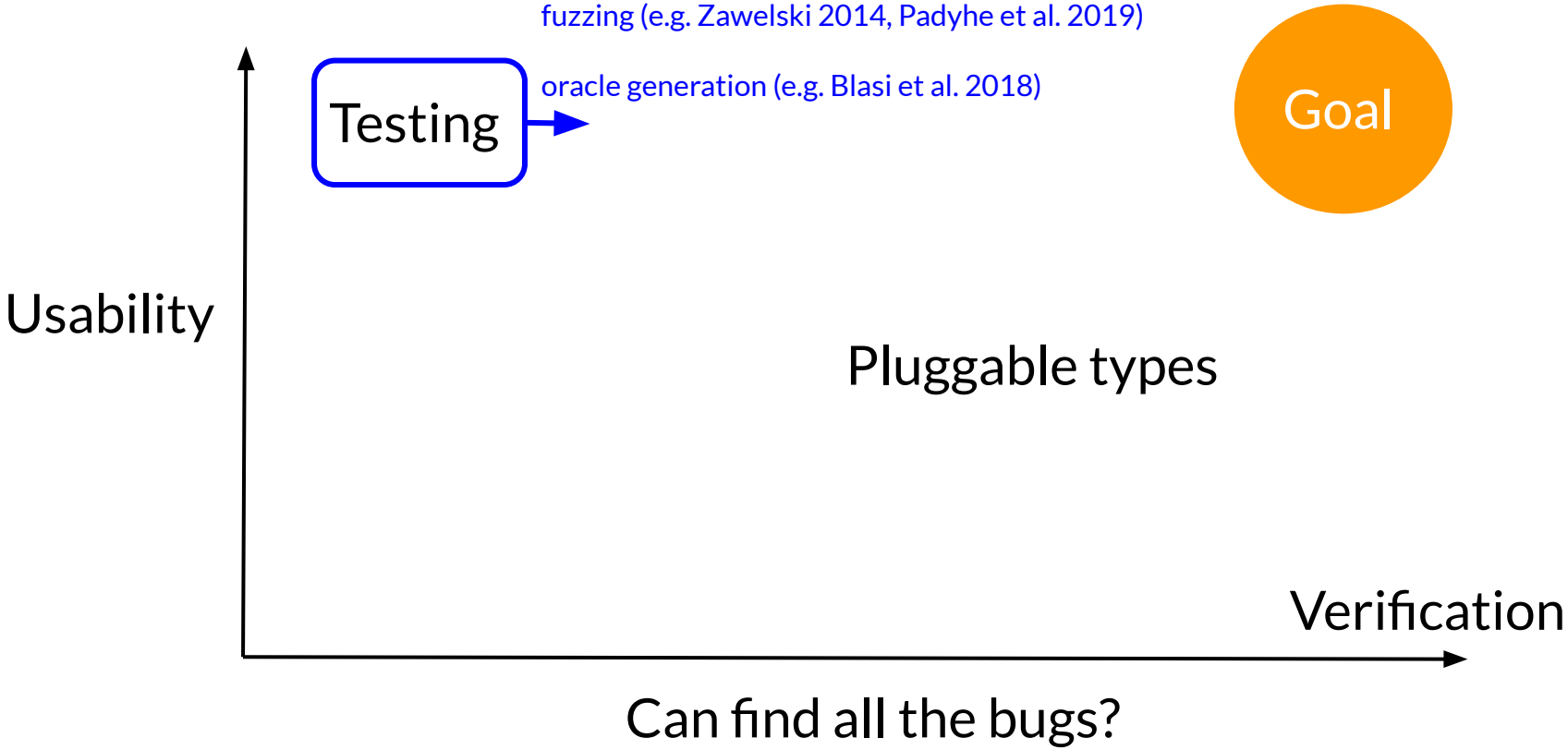


Related work

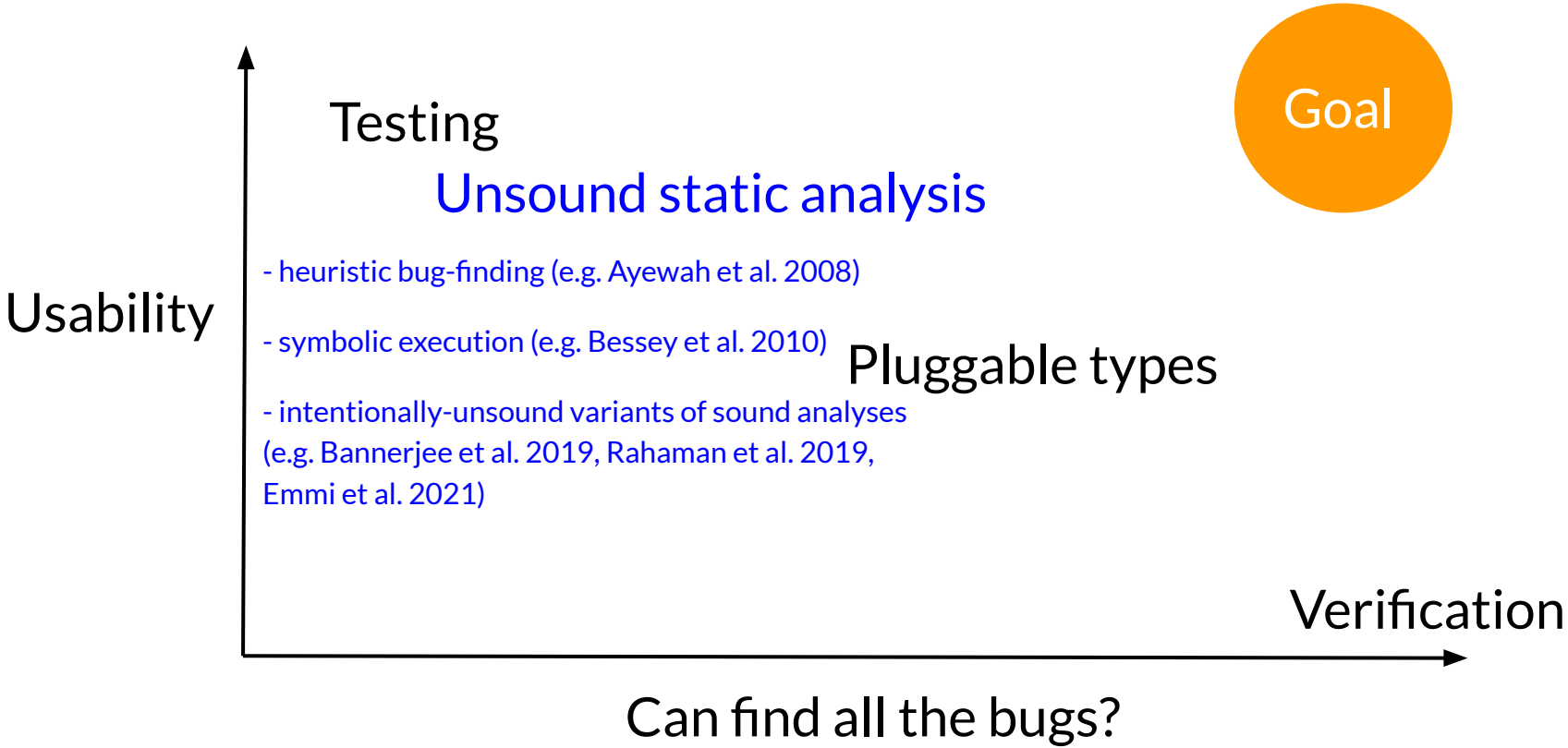
continuous integration (Fowler & Foemmel 2006)

fuzzing (e.g. Zawelski 2014, Padyhe et al. 2019)

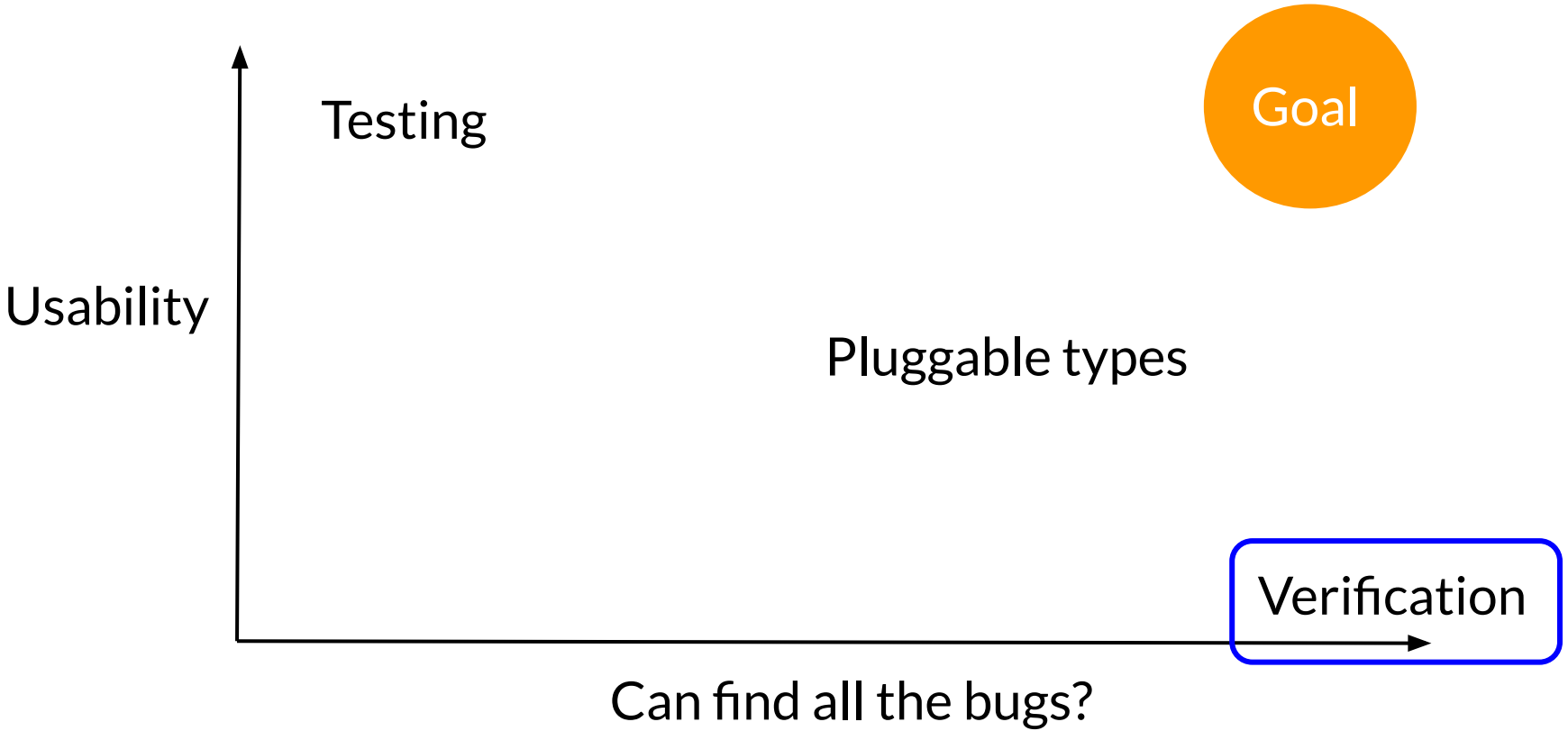
oracle generation (e.g. Blasi et al. 2018)



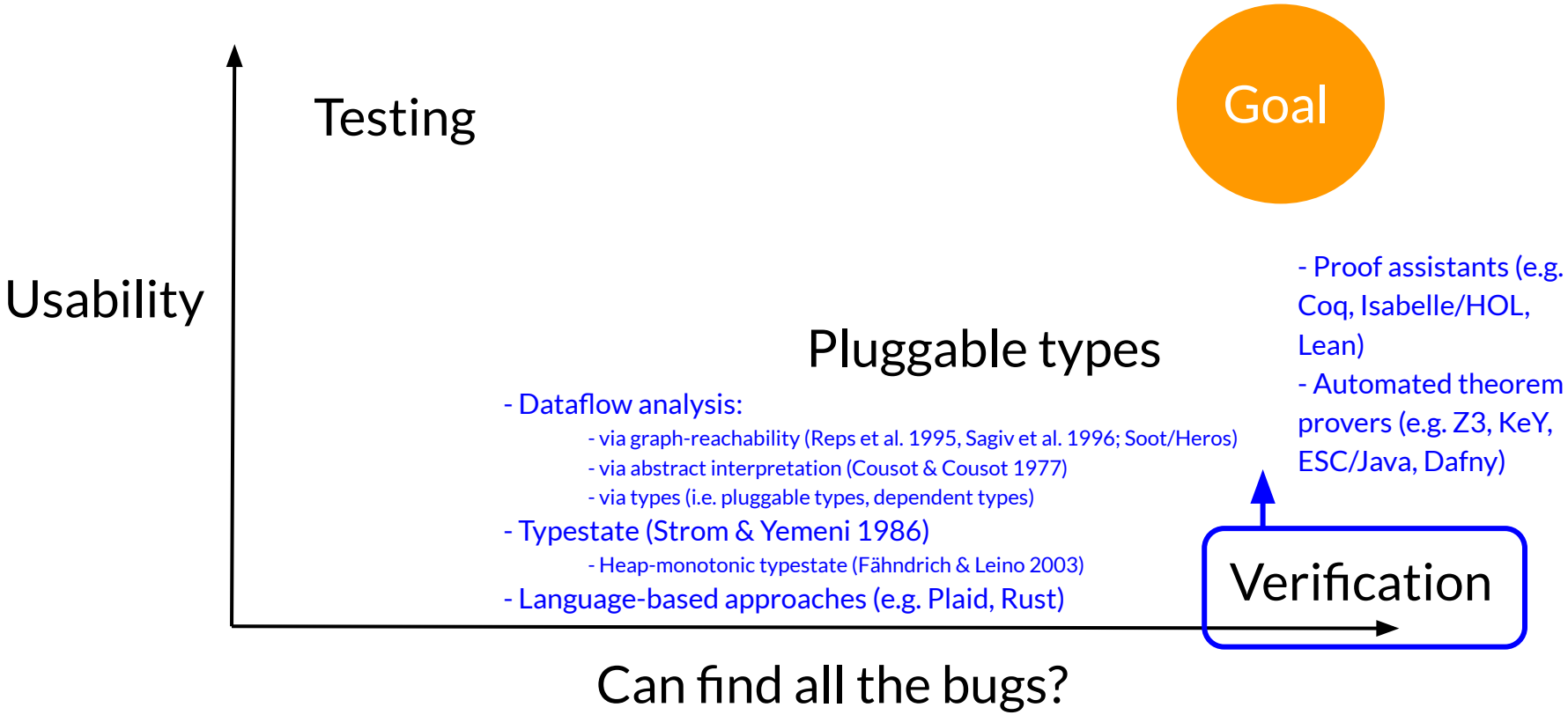
Related work



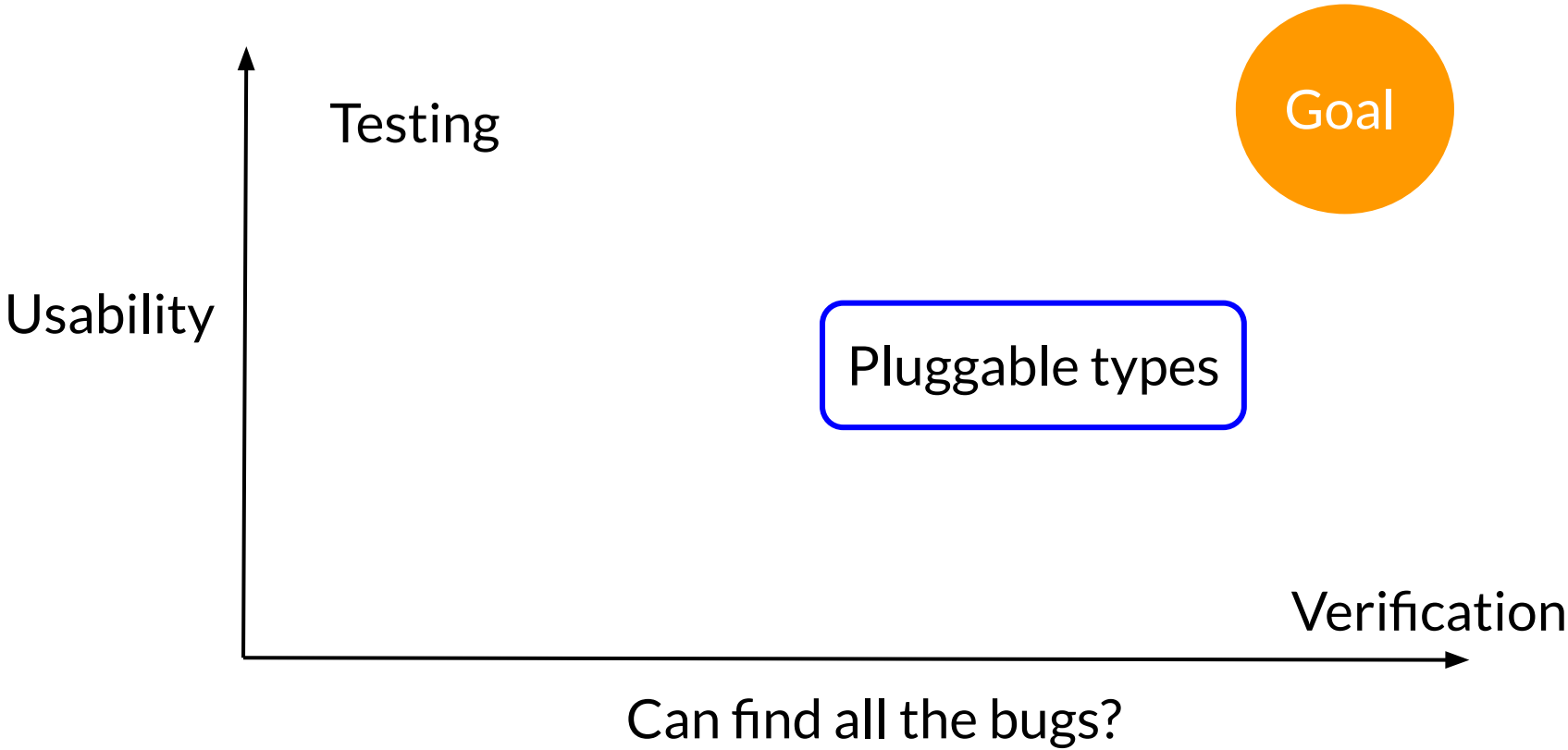
Related work



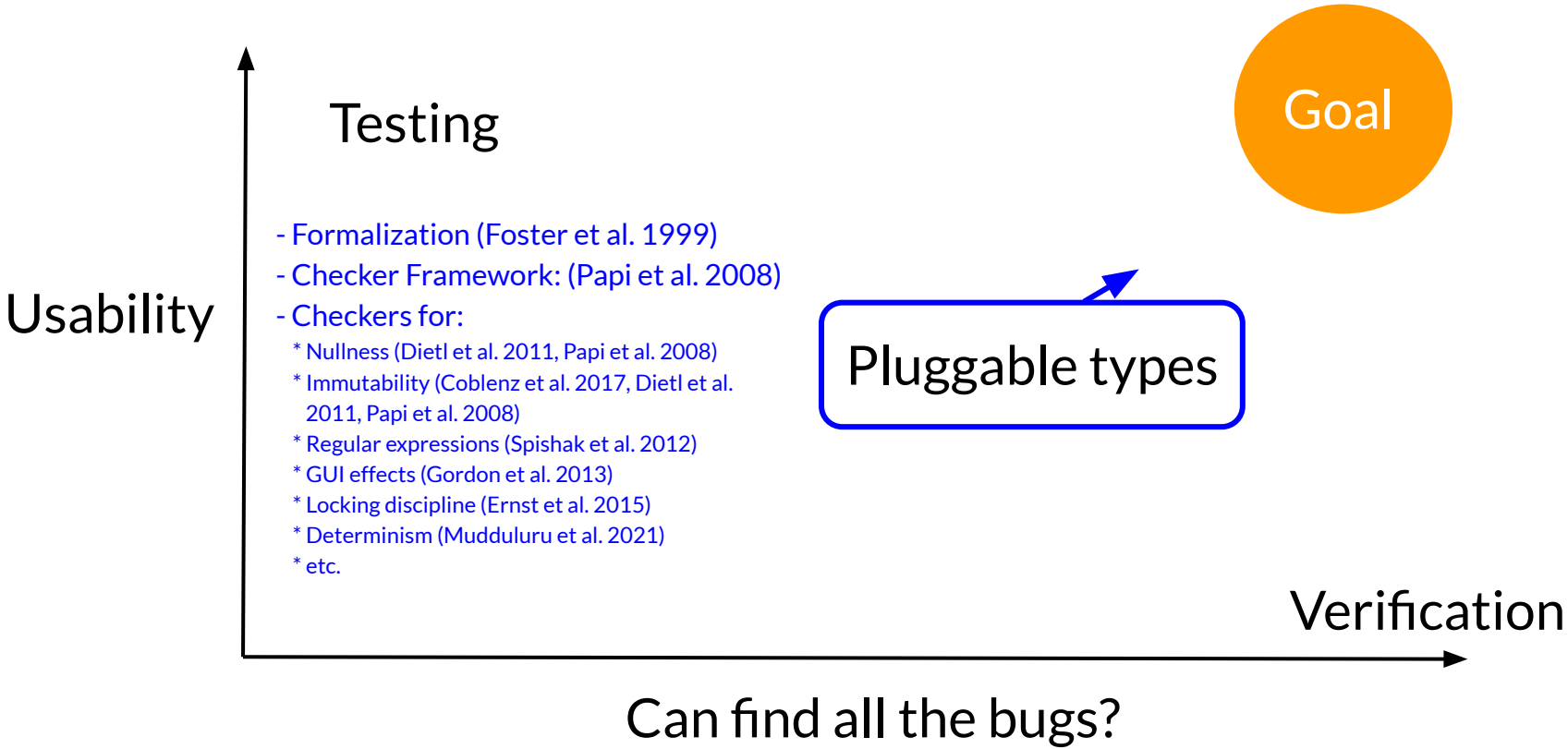
Related work



Related work



Related work



Conclusion

- **Goal:** every developer uses verification

Conclusion

- **Goal:** every developer uses verification

Our contributions:

- Pluggable types are a **powerful and useable** kind of verification
- Using types in **new domains** makes devs want to do verification
- **Cooperating type systems** can solve hard problems
- **Accumulation** can often replace typestate