A True Positives Theorem for a Static Race Detector

Nikos Gorogiannis — facebook, Middlesex University London

Peter O'Hearn — facebook, UCL

Ilya Sergey — Yale NUS College, NUS National University of Singapore

Key Messages

Unsound (and incomplete) static analyses can be *principled*,
satisfying meaningful theorems
that help to understand their behaviour and guide their design

One can have an unsound but effective static analysis,
which has significant industrial impact,
and which is supported by a *meaningful theorem*.

Context

1. We had a demonstrably-effective industrial analysis:
   RacerD (OOPSLA'18); >3k fixes in Facebook Java codebase
2. No soundness theorem

Static Analyses for Bug Detection

Context

1. We had a demonstrably-effective industrial analysis: RacerD (OOPSLA'18); >3k fixes in Facebook Java
2. No soundness theorem
3. Architecture: compositional abstract interpreter
4. No heuristic alarm filtering

Just ad hoc?

Our reaction:
Semantics/theory should understand/explain, not lecture.

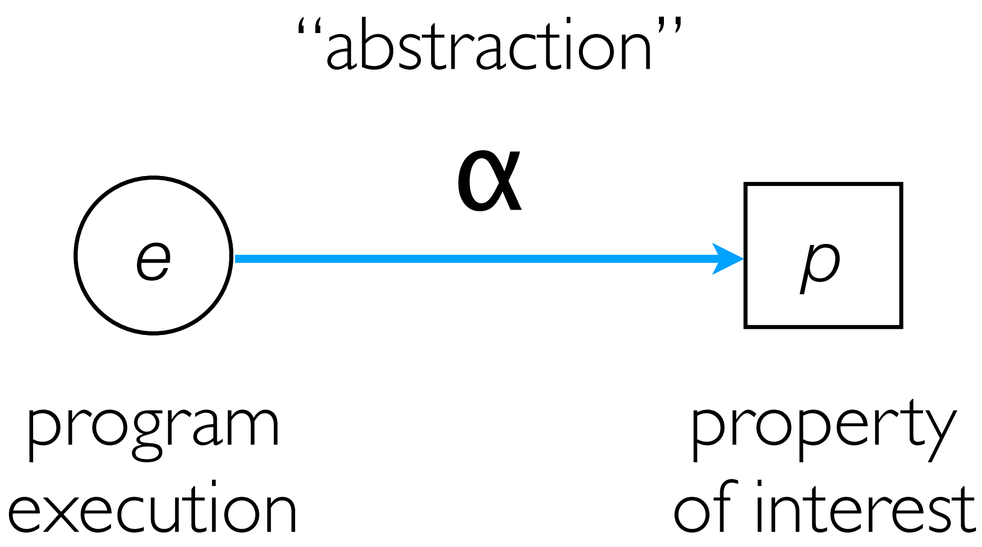
Conjecture

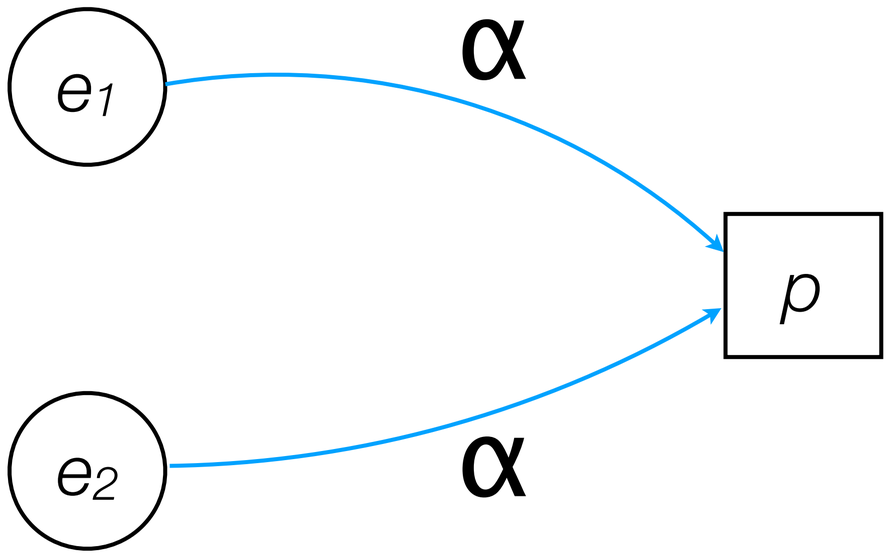
True Positives Theorem:

Under certain assumptions, the static bug detector reports *no false positives*.

Static Analyses for Program Validation

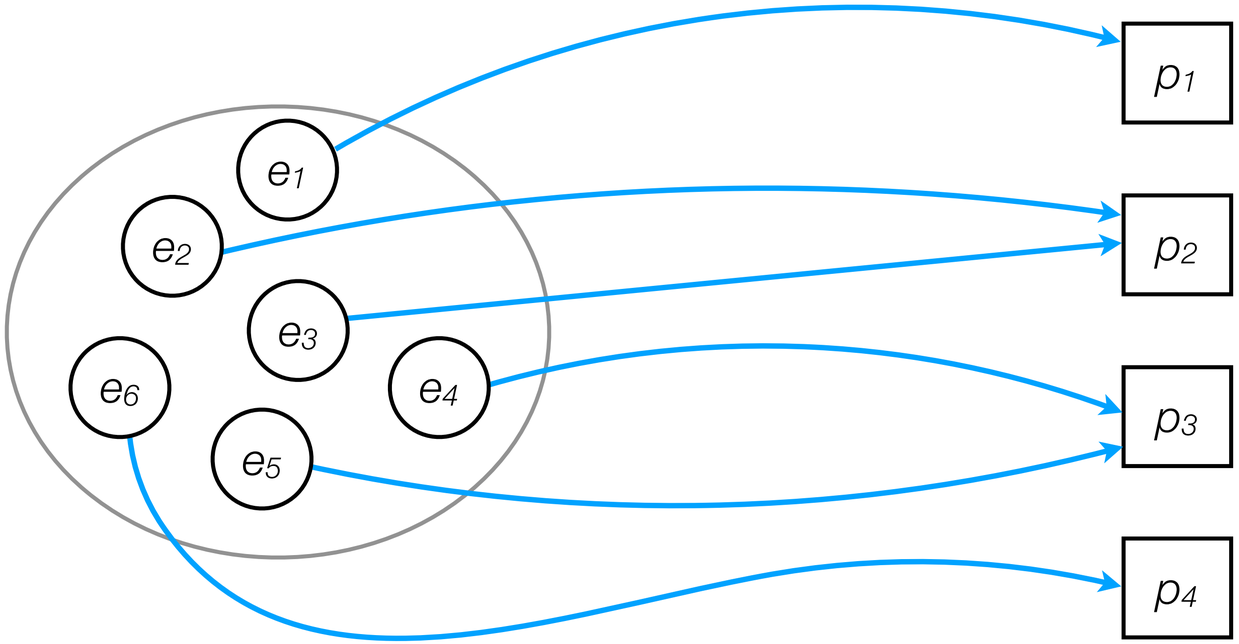
The Essence of Static Analysis



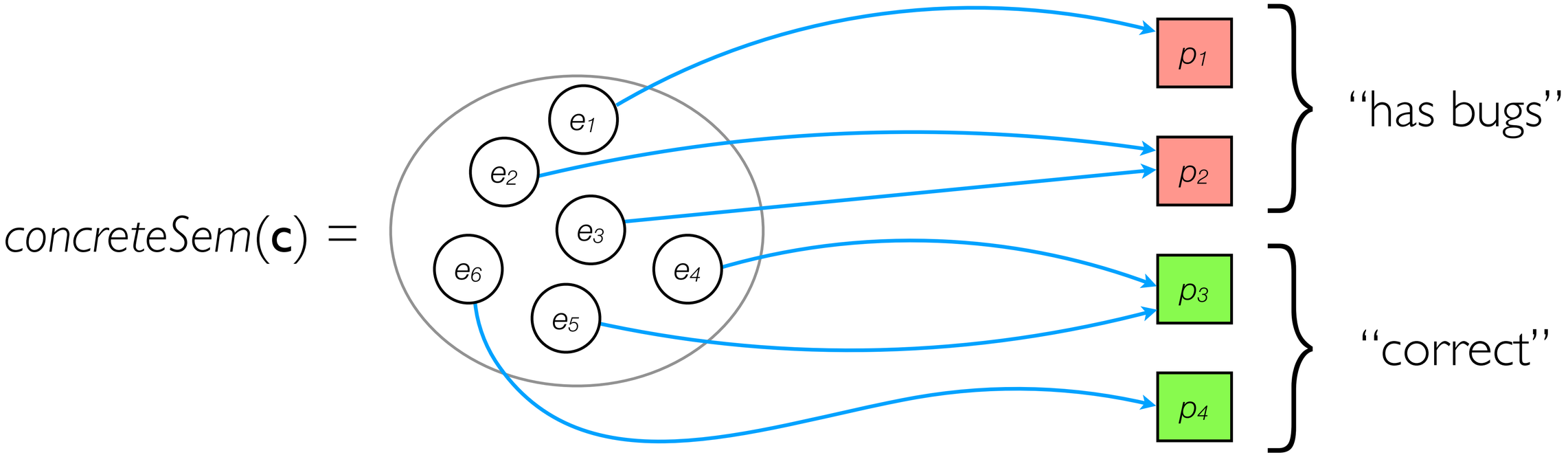

e_1
α
p
e_2
α

Static Analysis

$concreteSem(\mathbf{c}) =$

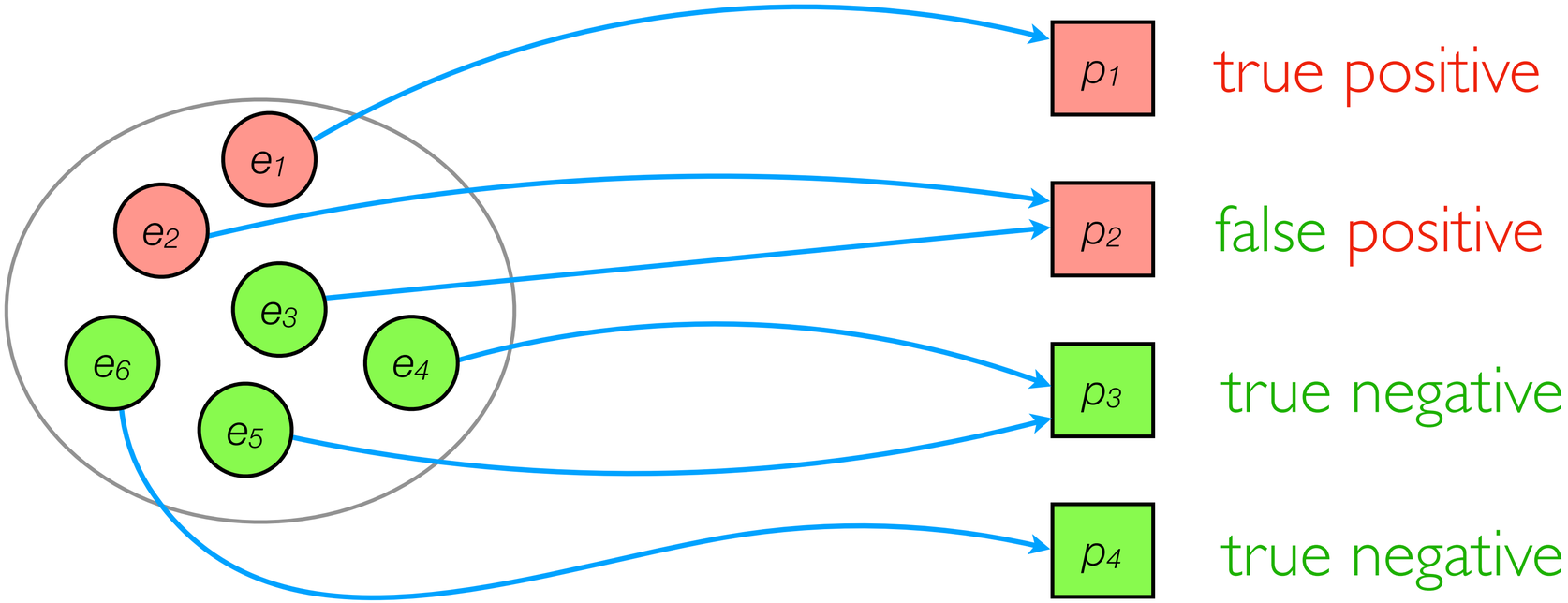


Static Analysis

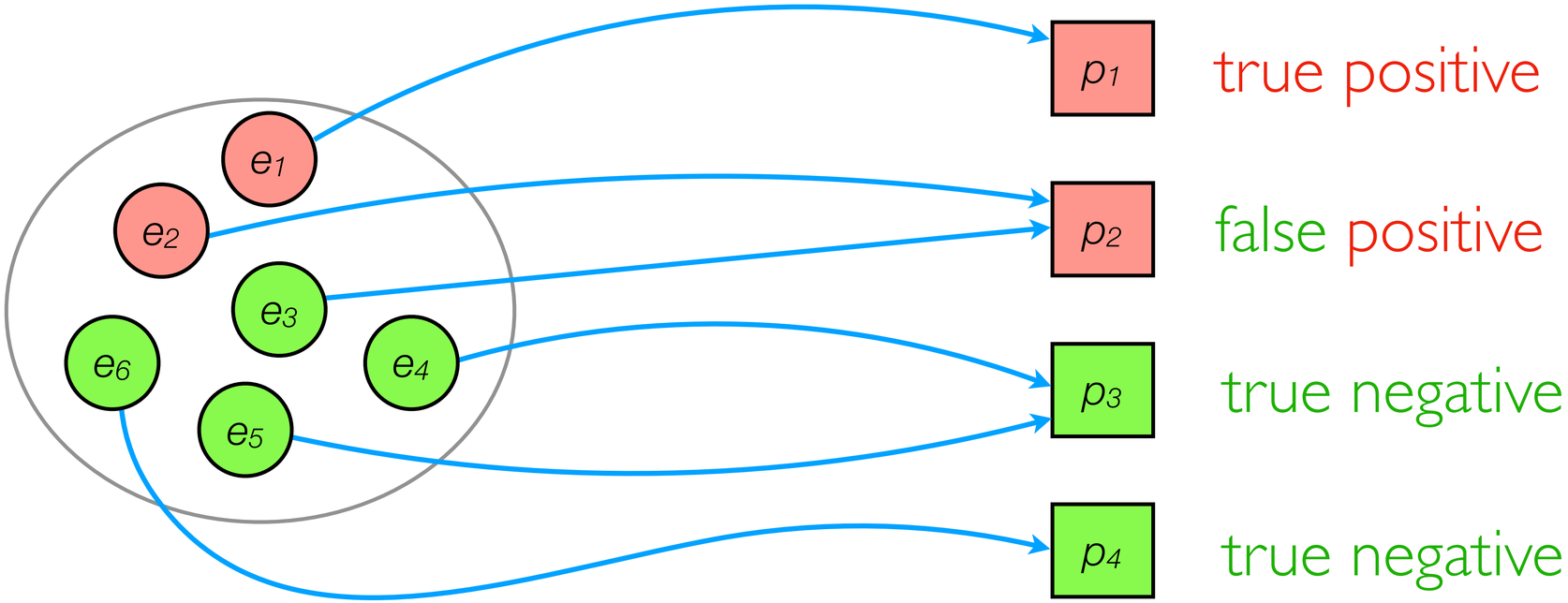


Verifier
or a
Bug Detector?

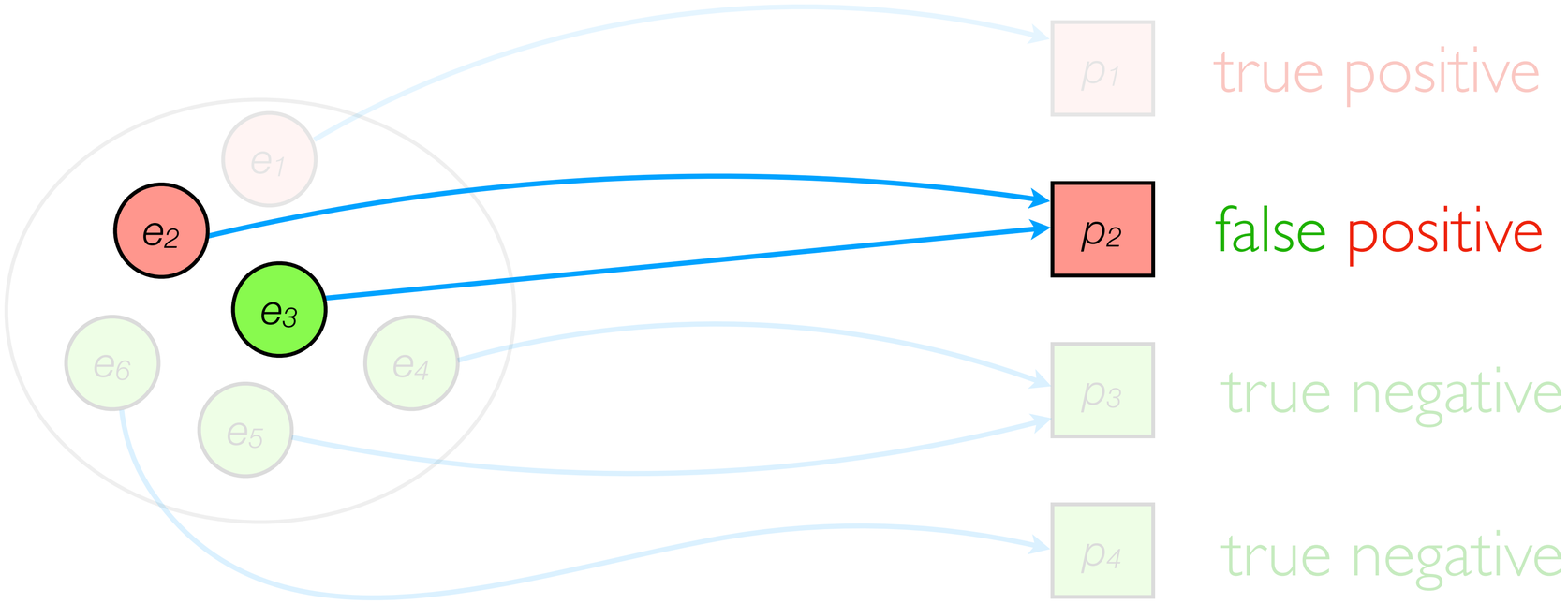
Program Verifier



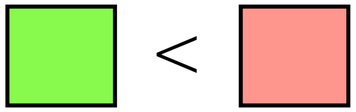
Sound Program Verifier



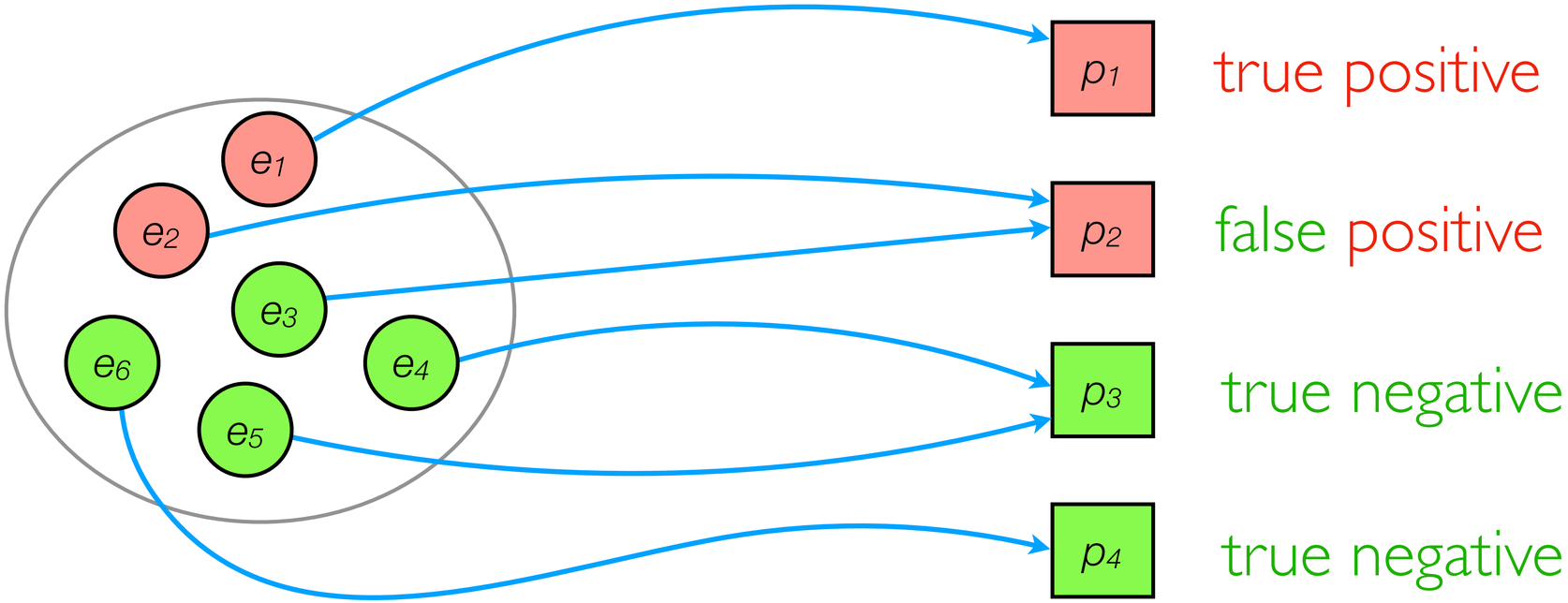
Sound Program Verifier



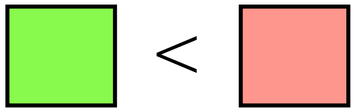
abstract over-approximation



Sound Program Verifier

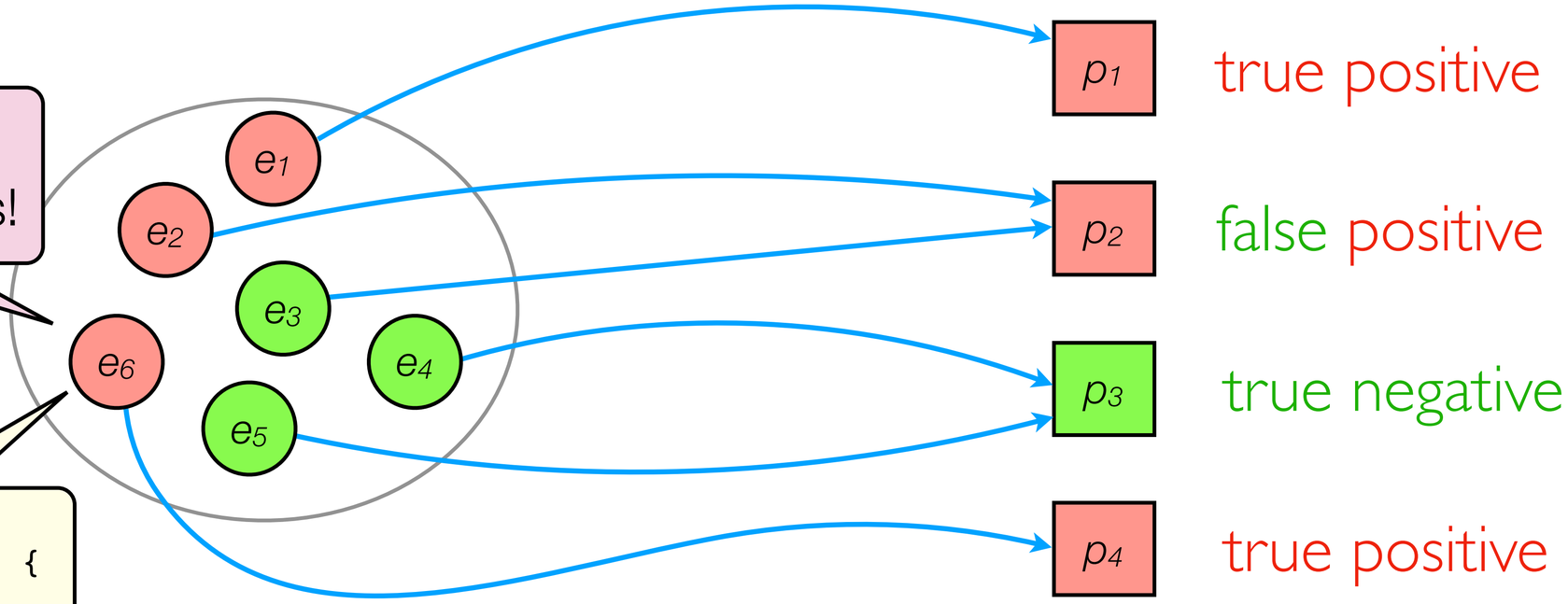


abstract over-approximation



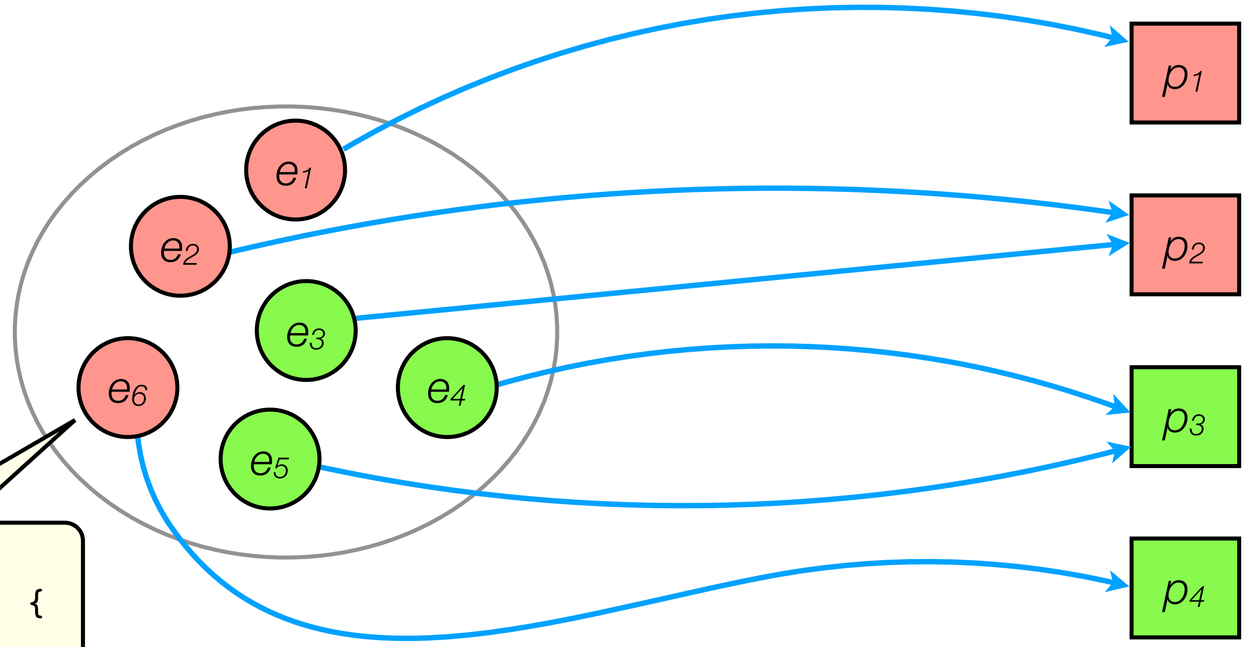
Sound Program Verifier

Developer:
Go away, that never happens!




```
if (n == VERY_UNLIKELY_VALUE) {
  bug.explode();
} else {
  // do nothing
}
```

Unsound Program “Verifier”



true positive

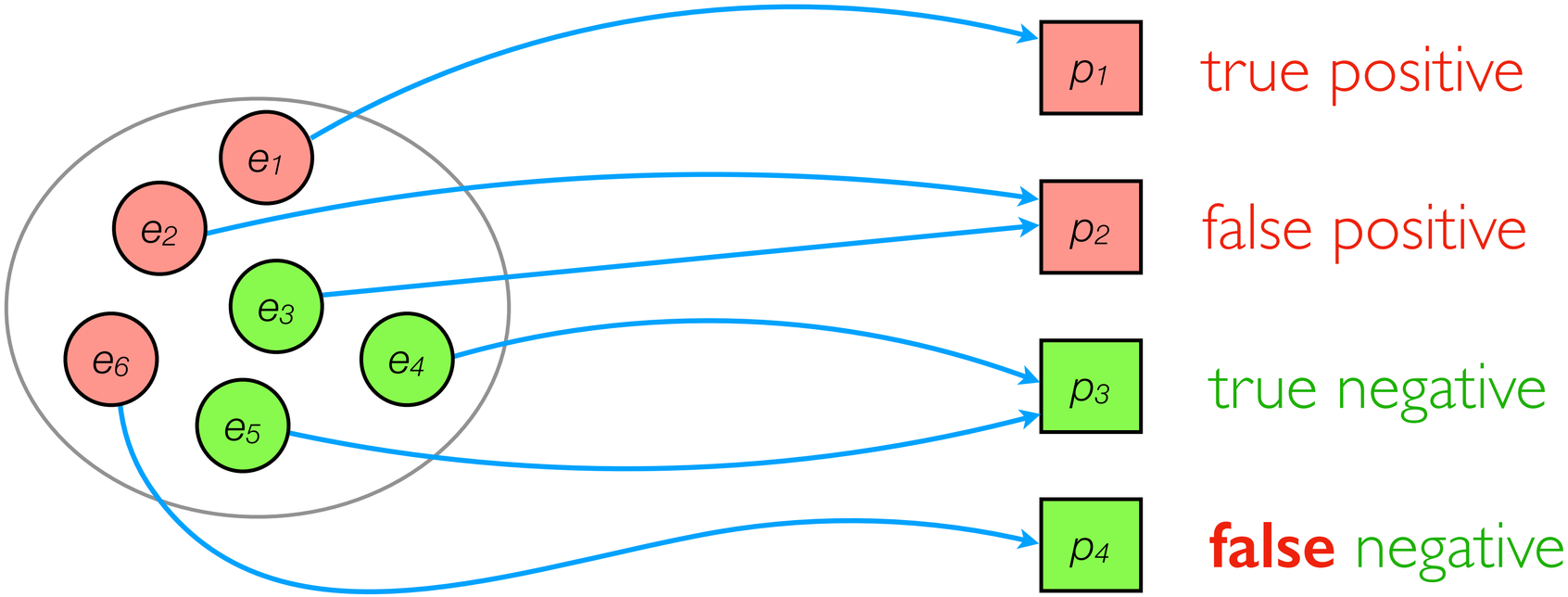
false positive

true negative

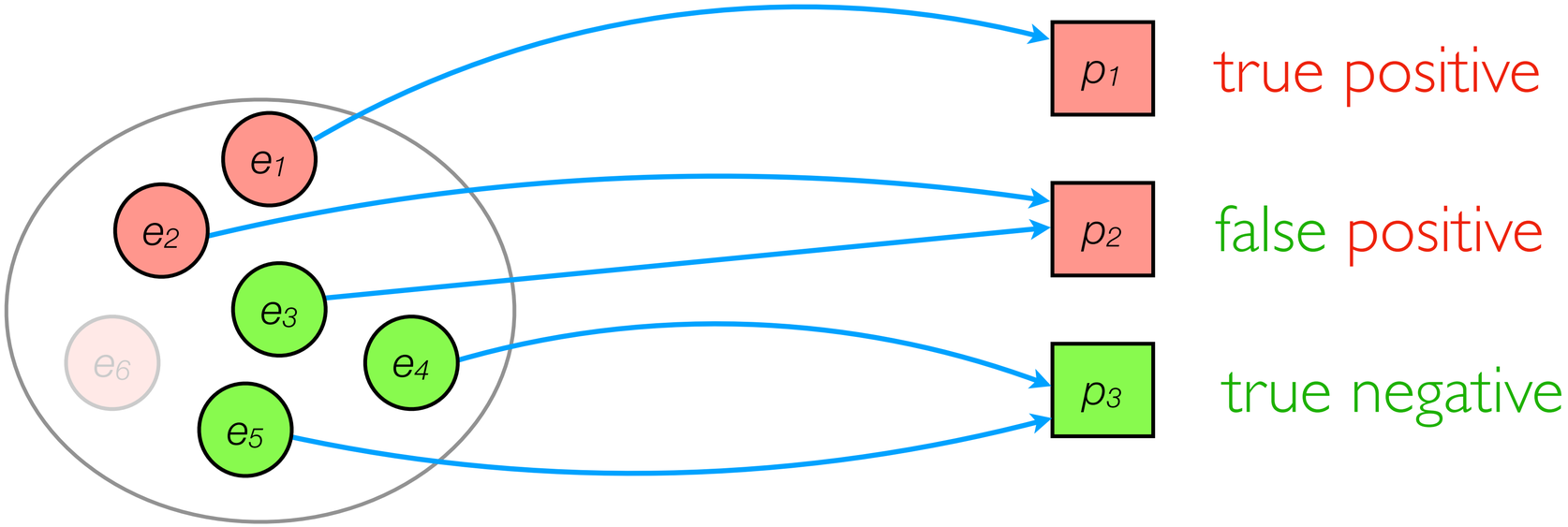
false negative

```
if (n == VERY_UNLIKELY_VALUE) {
  bug.explode();
} else {
  // do nothing
}
```

“Sound” Program Verifier

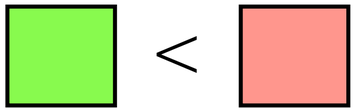


“Sound” Program Verifier



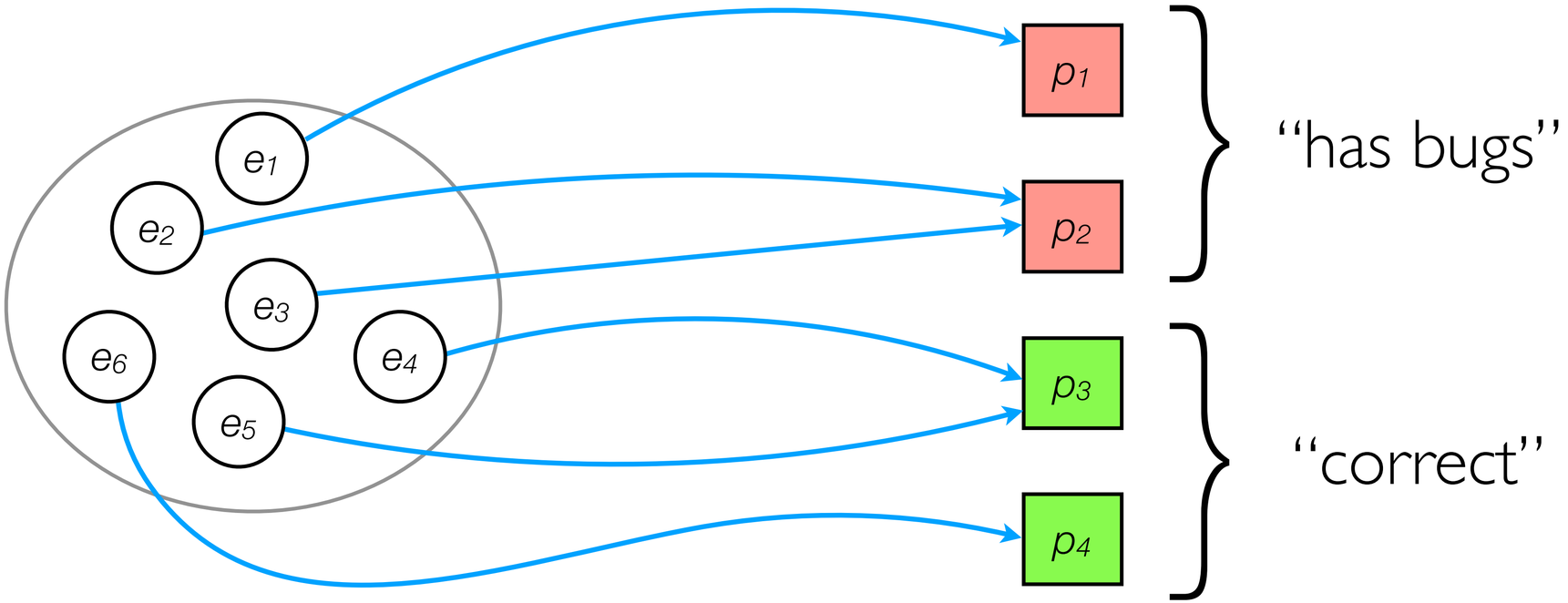
concrete under-approximation

abstract over-approximation

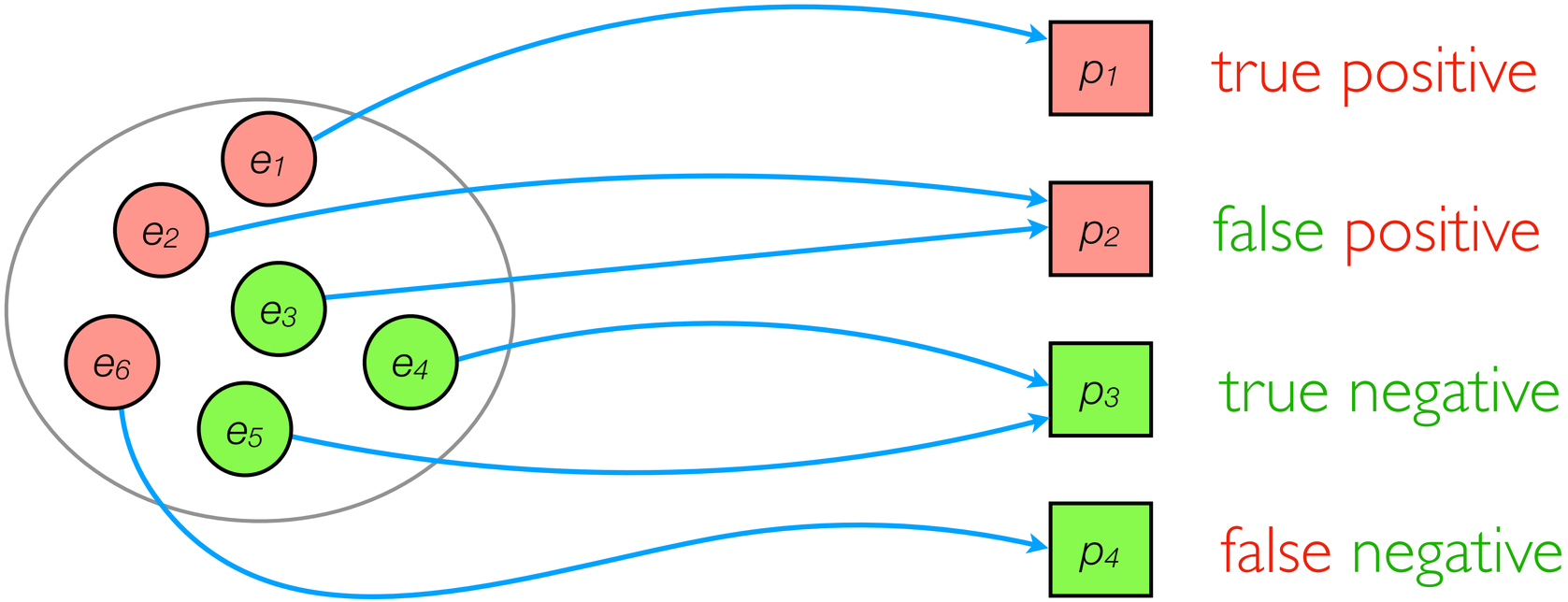


Sound Static Verifiers

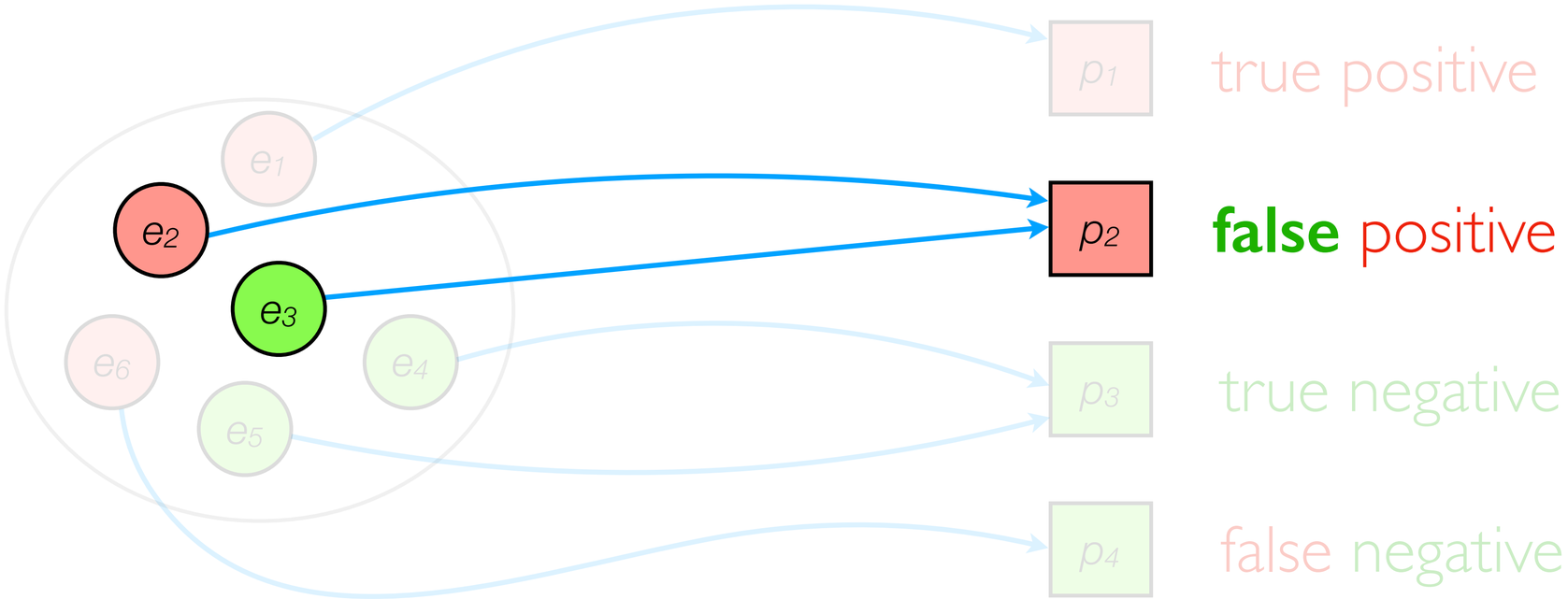
- False negatives (bugs missed) are ***bad***
- False positives (non-bugs reported) are ***okay***
- Constructed as ***over-approximation*** (of ***under-approximation***)
- **Soundness Theorem**:
  Under certain assumptions about the programs, the analyser has *no false negatives*.


e1
e2
e3
e4
e5
e6
p1
p2
p3
p4
"has bugs"
"correct"

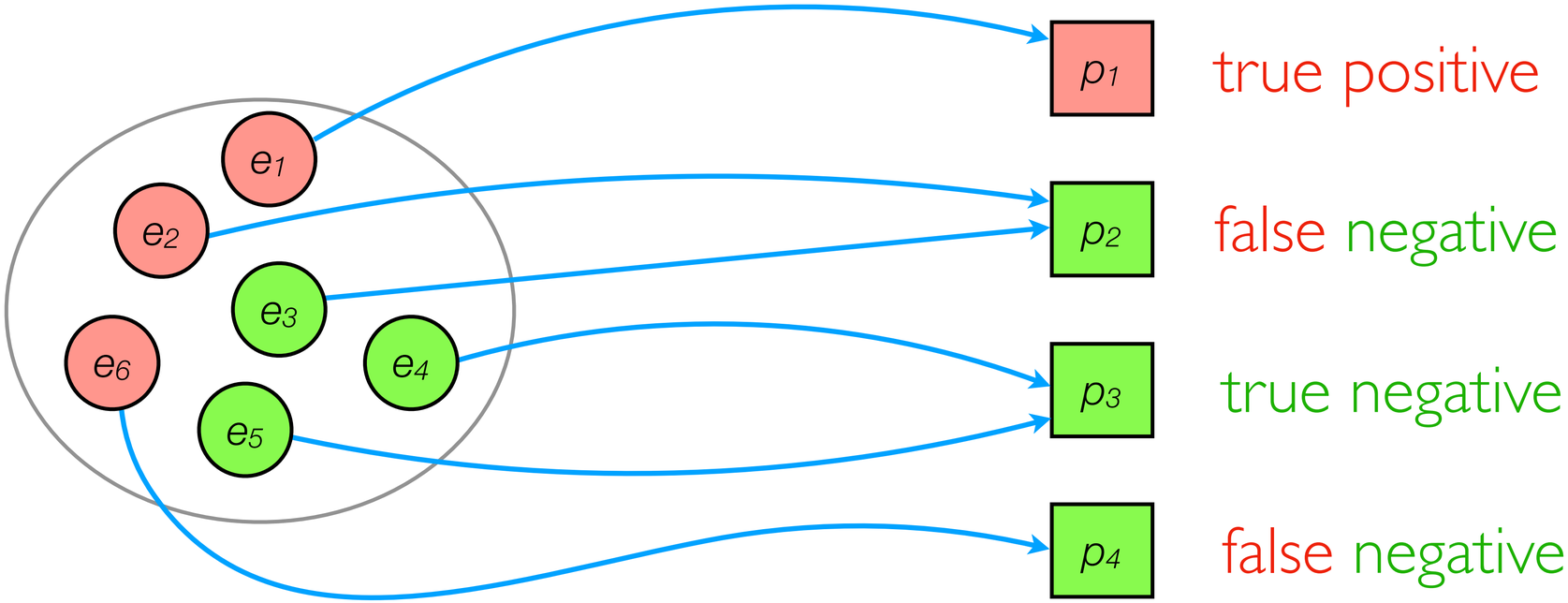
Static Bug Finder



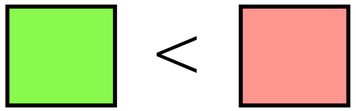
Unsound Static Bug Finder



Sound (but imprecise) Static Bug Finder



abstract *under-approximation*

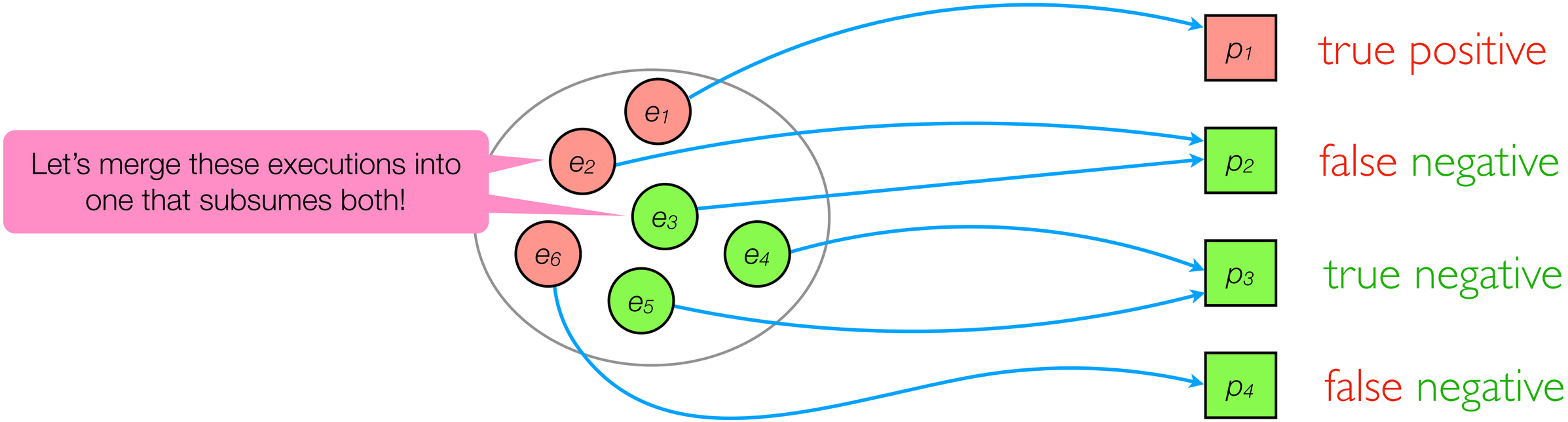


Loss of Precision in Static Bug Finders

```
if (n != VERY_UNLIKELY_VALUE) {
    e2    // bug happens here
} else {
    e3    // normal execution
}
```

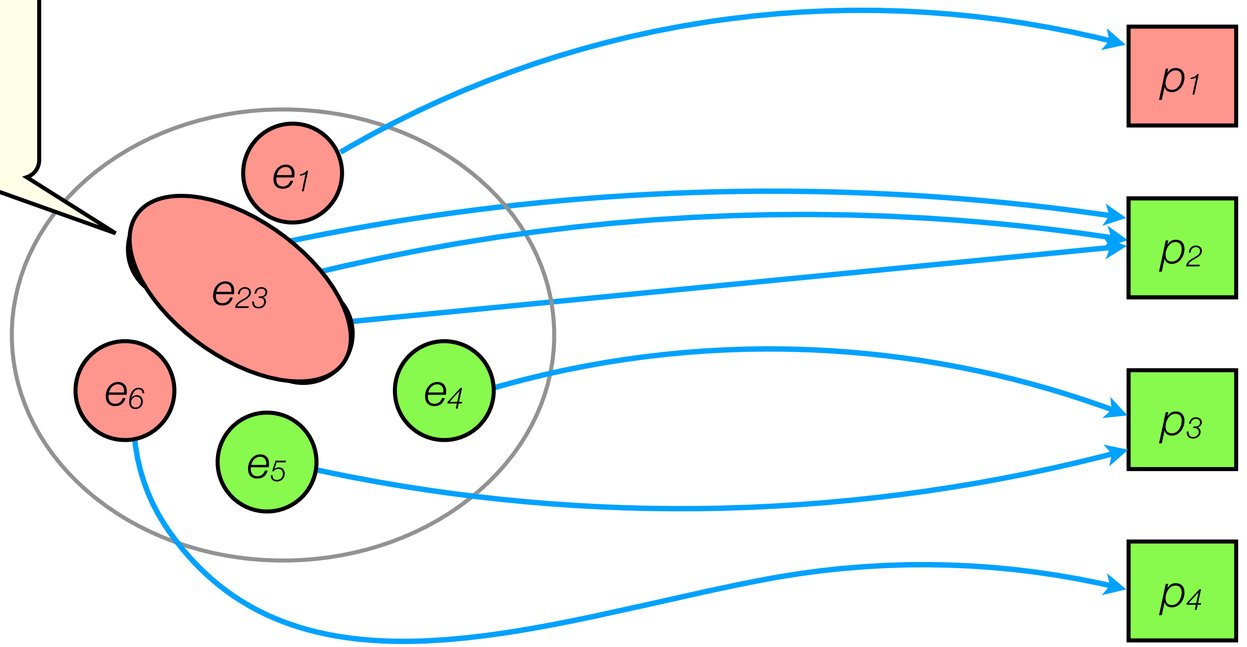
Idea: *over-approximate* in concrete semantics!

Sound (but Imprecise) Static Bug Finder



```
if (*) {
    // bug happens here
} else {
    // normal execution
}
```

overApproxConcreteSem(c) =



true positive

false positive

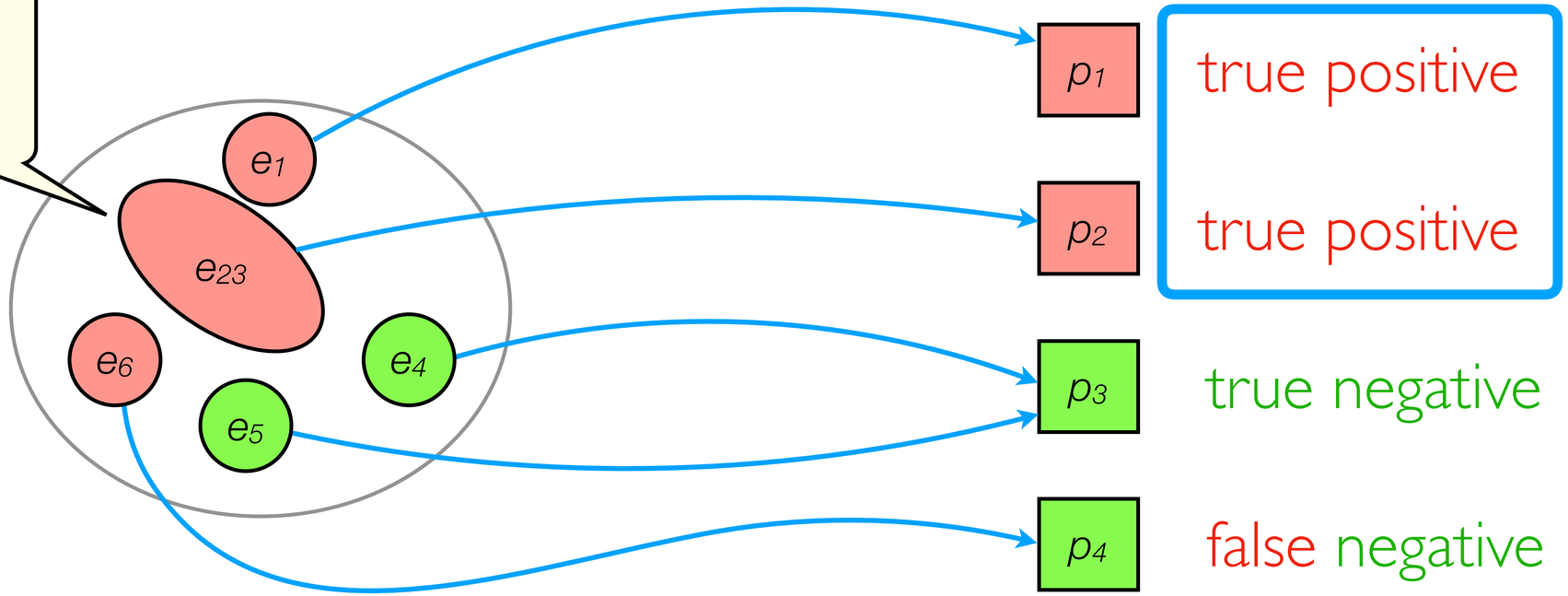
true negative

false negative

Sound Static Bug Finder

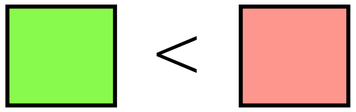
```
if (*) {
    // bug happens here
} else {
    // normal execution
}
```

overApproxConcreteSem(c) =



concrete *over-approximation*

abstract *under-approximation*



Towards Sound Static Bug Finders

(this work)

- False negatives (bugs missed) are ***okay***
- False positives (non-bugs reported) are ***bad***
- Constructed as ***under-approximation*** of ***over-approximation***
- **Soundness (True Positives) Theorem**:
  Under certain assumptions about the programs, the analyser has ***no false positives***.

A Recipe for True Positives Theorem

1. ***Over-approximate*** semantic elements to make up for "difficult" dynamic execution aspects

   Example: replace conditions and loops with their *non-deterministic* versions

2. Pick abstraction **α** for over-approximated executions that ***provably identifies*** "buggy" behaviours:

   $$\forall \text{ e: execution}, \textbf{\textit{hasBug}}(\alpha(\text{e})) \Rightarrow \text{execution e has a bug}$$

3. Design an abstract semantics ***asem***, so it is *complete* wrt. **α** and *over-approximated* concrete semantics:

   $$\forall \text{ c : program}, \textbf{\textit{asem}}(\text{c}) = \alpha(\textbf{overApproxConcreteSem}(\text{c}))$$

4. Together, ***asem*** and ***hasBug*** provide a *TP-sound* static bug finder.

Case Study: **RacerDX**

- A provably TP-Sound version of Facebook's **RacerD** concurrency analyser (Blackshear et al., OOPSLA'18)

- **Buggy executions:** *data races* in *lock-based concurrent* programs

- **Syntactic assumptions**:
  Java programs with well-scoped locking (`synchronised`), no recursion, reflection, dynamic class loading; global variables are ignored.

- **Concrete over-approximation**:
  Loops and conditionals are *non-deterministic.*

A True Race

```java
class Bloop {
  public int f = 1;
}
```

```java
class Burble {

  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
}
```

A False Race

```
class Bloop {
  public int f = 1;
}
```

```
class Burble {

  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
}
```

Path prefix **b** is *"unstable"* (*"wobbly"*),
as it's reassigned, hence race is evaded.

Complete Abstraction for Race Detection

(W, L, A)

"Wobbly" paths, touched during execution

Locking level

Accesses/locks with formals/fields

```
class Burble {

  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
}
```

- asem(`meps(b)`) = ({b.f}, 0, {R(b.f, 1)})
- asem(`reps(b)`) = ({b.f}, 0, {W(b.f, 0)})
- asem(`beps(b)`) = ({b, b.f}, 0, {W(b, 0), W(b.f, 0)})

Analysing Summaries for Races

- asem(`meps(b)`) = ({b.f}, 0, {R(b.f, 1)})
- asem(`reps(b)`) = ({b.f}, 0, {W(b.f, 0)})
- asem(`beps(b)`) = ({b, b.f}, 0, {W(b, 0), W(b.f, 0)})

`meps(b) || reps(b)` ⇒ Can race,

report a bug!

```
class Burble {

  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
}
```

Analysing Summaries for Races

- asem(`meps(b)`) = ({b.f}, 0, {R(b.f, 1)})
- asem(`reps(b)`) = ({b.f}, 0, {W(b.f, 0)})
- asem(`beps(b)`) = ({b, b.f}, 0, {W(b, 0), W(b.f, 0)})

```
class Burble {

  public void meps(Bloop b) {
    synchronized (this) {
      System.out.println(b.f);
    }
  }

  public void reps(Bloop b) {
    b.f = 42;
  }

  public void beps(Bloop b) {
    b = new Bloop();
    b.f = 239;
  }
}
```

`meps(b) || beps(b)` ⇒ Maybe don't race,

don't report a bug

Formal Result

RacerDX enjoys the True Positives Theorem *wrt*. Data Race Detection

(Details in the paper)

Evaluation

What is the price to pay for having the TP Theorem?

(Reporting *no bugs whatsoever* is TP-Sound)

RacerD *vs* RacerDX

Target	LOC	D CPU	DX CPU	CPU ±%	D Reps	DX Reps	Reps ±%
avrora	76k	103	102	0.4%	143	92	36%
Chronicle-Map	45k	196	196	0.1%	2	2	0%
jvm-tools	33k	106	109	-3.6%	30	26	13%
RxJava	273k	76	69	9.2%	166	134	19%
sunflow	25k	44	44	-1.4%	97	42	57%
xalan-j	175k	144	137	5.0%	326	295	10%

RacerD *vs* RacerDX

Target	LOC	D CPU	DX CPU	CPU ±%	D Reps	DX Reps	Reps ±%
avrora	76k	103	102	0.4%	143	92	36%
Chronicle-Map	45k	196	196	0.1%	2	2	0%
jvm-tools	33k	106	109	-3.6%	30	26	13%
RxJava	273k	76	69	9.2%	166	134	19%
sunflow	25k	44	44	-1.4%	97	42	57%
xalan-j	175k	144	137	5.0%	326	295	10%

RacerD *vs* RacerDX

Target	LOC	D CPU	DX CPU	CPU ±%	D Reps	DX Reps	Reps ±%
avrora	76k	103	102	0.4%	143	92	36%
Chronicle-Map	45k	196	196	0.1%	2	2	0%
jvm-tools	33k	106	109	-3.6%	30	26	13%
RxJava	273k	76	69	9.2%	166	134	19%
sunflow	25k	44	44	-1.4%	97	42	57%
xalan-j	175k	144	137	5.0%	326	295	10%

To Take Away: Theory

- A **True Positive-**Sound static bug finder never reports false positives. It can be designed as an under-approximation of an over-approximation
- An abstraction **α** for TP-Sound static bug detection can be *very simple*, but it has to be complete (i.e., sufficient) to report bugs.

To Take Away: Practice

- **RacerDX** is TP-Sound race detector, whose precision and performance are comparable with **Facebook's RacerD** (Blackshear et al., OOPSLA'18)

- If **RacerDX** had been deployed initially rather than **RacerD**, it would have found 1000s of bugs, far outstripping all *reported impact* in previous concurrency analyses (counterfactual reasoning)

- Until now, static analysers for bug catching that are effective in practice but unsound have often been regarded as ***ad hoc***; in the future, they can be ***principled, satisfying theorems*** to inform and guide their designs.

Thanks!