

Compositional Static Race Detection at Scale, without False Positives

Ilya Sergey



Joint work with

Sam Blackshear, Peter O'Hearn, Nikos Gorogiannis

facebook

Key Messages

Static analyses for *concurrency* can be made *scalable* and *precise*.

Unsound (and incomplete) static analyses can be *principled*, satisfying meaningful theorems.

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

Practice

RACERD: Compositional Static Race Detection

SAM BLACKSHEAR, Facebook, USA

NIKOS GOROGIANNIS, Facebook, UK and Middlesex University London, UK

PETER W. O'HEARN, Facebook, UK and University College London, UK

ILYA SERGEY*, Yale-NUS College, Singapore and University College London, UK

Automatic static detection of data races is one of the most basic problems in reasoning about concurrency. We present RACERD—a static program analysis for detecting data races in Java programs which is fast, can scale to large code, and has proven effective in an industrial software engineering scenario. To our knowledge, RACERD is the first inter-procedural, compositional data race detector which has been empirically shown to have non-trivial precision and impact. Due to its compositionality, it can analyze code changes quickly, and this allows it to perform *continuous reasoning* about a large, rapidly changing codebase as part of deployment within a continuous integration ecosystem. In contrast to previous static race detectors, its design favors reporting high-confidence bugs over ensuring their absence. RACERD has been in deployment for over a year at Facebook, where it has flagged over 2500 issues that have been fixed by developers before reaching production. It has been important in enabling the development of new code as well as fixing old code: it helped support the conversion of part of the main Facebook Android app from a single-threaded to a multi-threaded architecture. In this paper we describe RACERD's design, implementation, deployment and impact.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Concurrent programming structures**;

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom

ACM Reference Format:

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RACERD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (November 2018), 28 pages. <https://doi.org/10.1145/3276514>

OOPSLA '18

Theory



A True Positives Theorem for a Static Race Detector

NIKOS GOROGIANNIS, Facebook, UK and Middlesex University London, UK

PETER W. O'HEARN, Facebook, UK and University College London, UK

ILYA SERGEY*, Yale-NUS College, Singapore and National University of Singapore, Singapore

RACERD is a static race detector that has been proven to be effective in engineering practice: it has seen thousands of data races fixed by developers before reaching production, and has supported the migration of Facebook's Android app rendering infrastructure from a single-threaded to a multi-threaded architecture. We prove a True Positives Theorem stating that, under certain assumptions, an idealized theoretical version of the analysis *never reports a false positive*. We also provide an empirical evaluation of an implementation of this analysis, versus the original RACERD.

The theorem was motivated in the first case by the desire to understand the observation from production that RACERD was providing remarkably accurate signal to developers, and then the theorem guided further analyzer design decisions. Technically, our result can be seen as saying that the analysis computes an under-approximation of an over-approximation, which is the reverse of the more usual (over of under) situation in static analysis. Until now, static analyzers that are effective in practice but unsound have often been regarded as ad hoc; in contrast, we suggest that, in the future, theorems of this variety might be generally useful in understanding, justifying and designing effective static analyses for bug catching.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Concurrent programming structures**;

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom, Abstract Interpretation

ACM Reference Format:

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (January 2019), 29 pages. <https://doi.org/10.1145/3290370>

POPL '19

Part I

RacerD:

Compositional Static Race Detection



Litho: A declarative UI framework for Android

GET STARTED

LEARN MORE

TUTORIAL

Litho Component

Fetch data

Talk to network

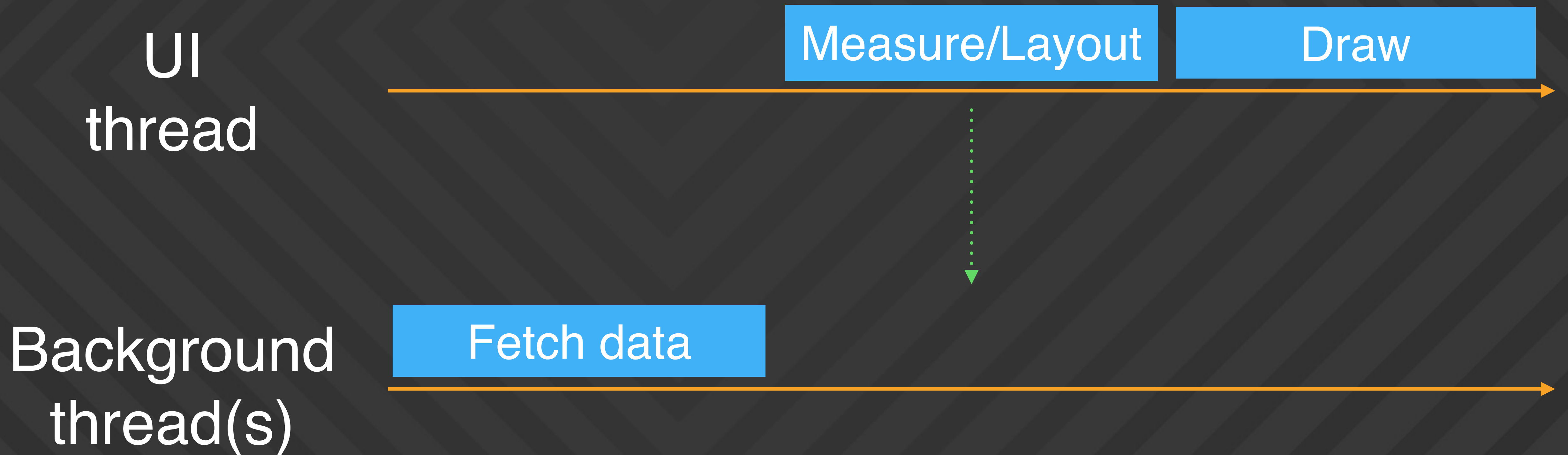
Measure/Layout

Determine size and position

Draw

Render and attach

Moving layout to background for better perf



BUT: to migrate, Measure/Layout step needs to be thread-safe. Otherwise...

Adding concurrency can introduce **data races**

Data race:

**two concurrent accesses to
the same memory location
where at least one is a write.**

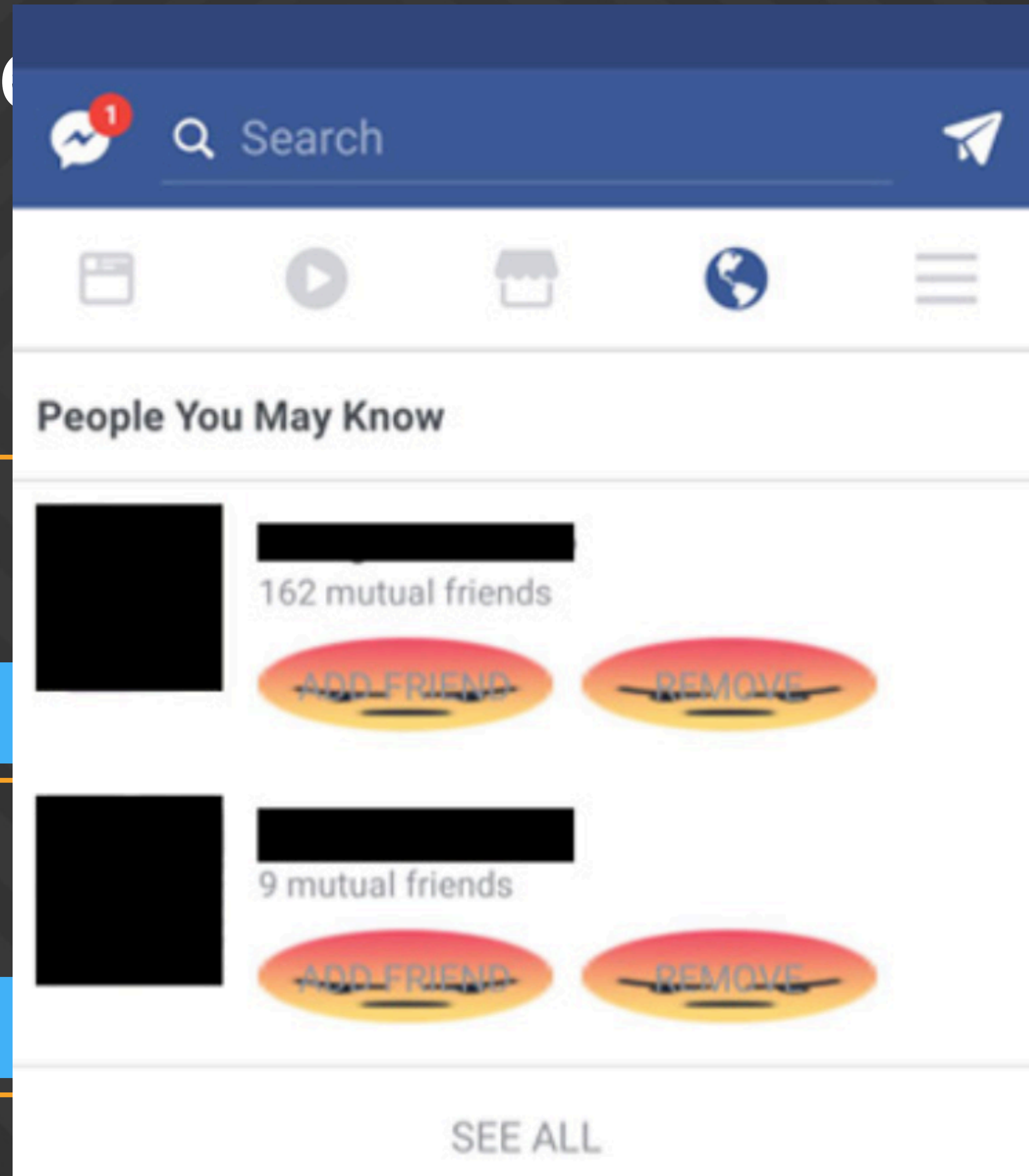
Concurrency

Data races

UI thread

Background thread 1

Background thread 2



Draw

cts

Conflicts

\$

SEE ALL

Adding concurrency to sequential code is scary

Problem 1: 1000s of existing components. Where should we add synchronization to avoid races?

Problem 2: Nondeterminism makes it hard to test for races. How do we prevent regressions?

Static race detector can show us where to add synchronization + prevent regressions at code review time.

Devs need static analysis for migration

Talking with [REDACTED], one of our managers - we realized that the timeline of background layout in feed might be closely tied to the timeline of static analysis - I'm wondering if you have your roadmap already fleshed out.

Stringent requirements for helpful analysis

Interprocedural

Scalable +
incremental

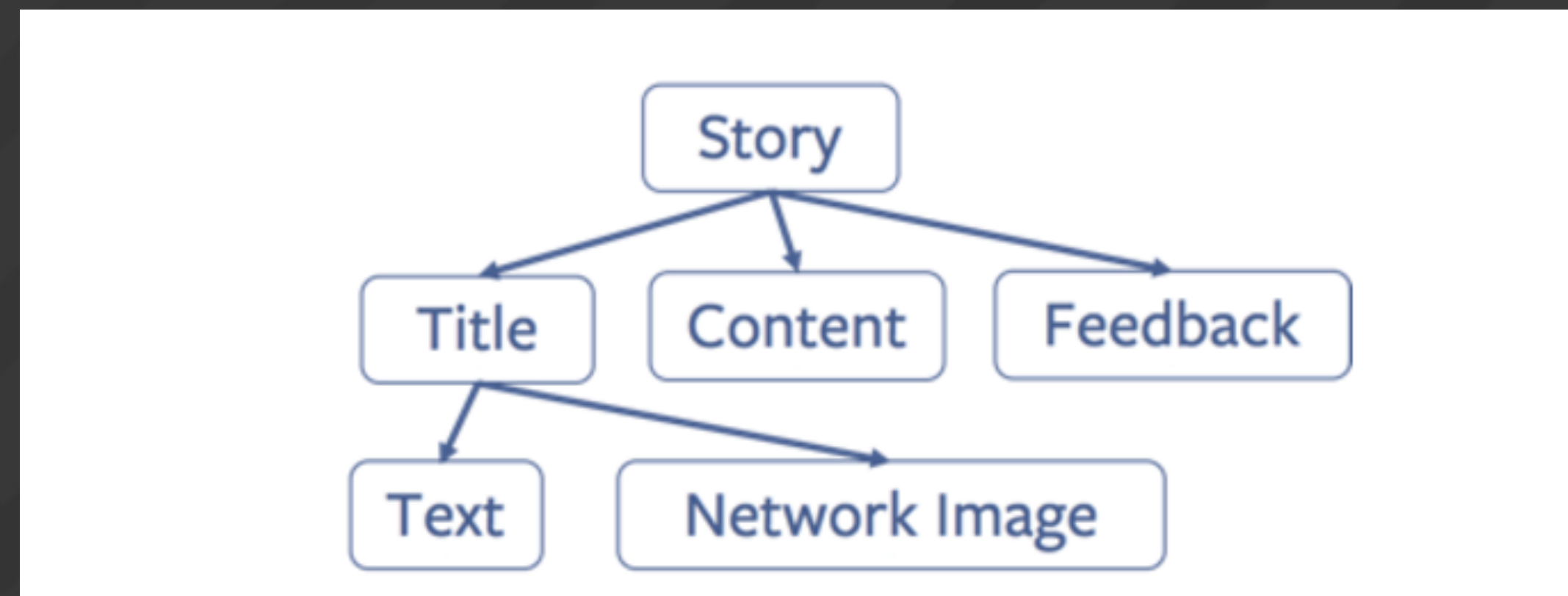
Low annotation
burden

High signal >>
catching all bugs

cc on @ThreadSafe

Will the eventual thread safe annotation be recursive? Will it check that dependencies, at least how they're used, are thread safe?

Like · Reply · Share · 2 · October 14, 2016 at 11:04pm



~~@GuardedBy("this")
Classname object;~~

RacerD: compositional static race detection

(1) Motivation: assist Litho migration + detect regressions

(2) RacerD deep dive: design, domains, reporting

(3) Evaluation: RacerD vs static/dynamic race detectors, RacerD @ FB

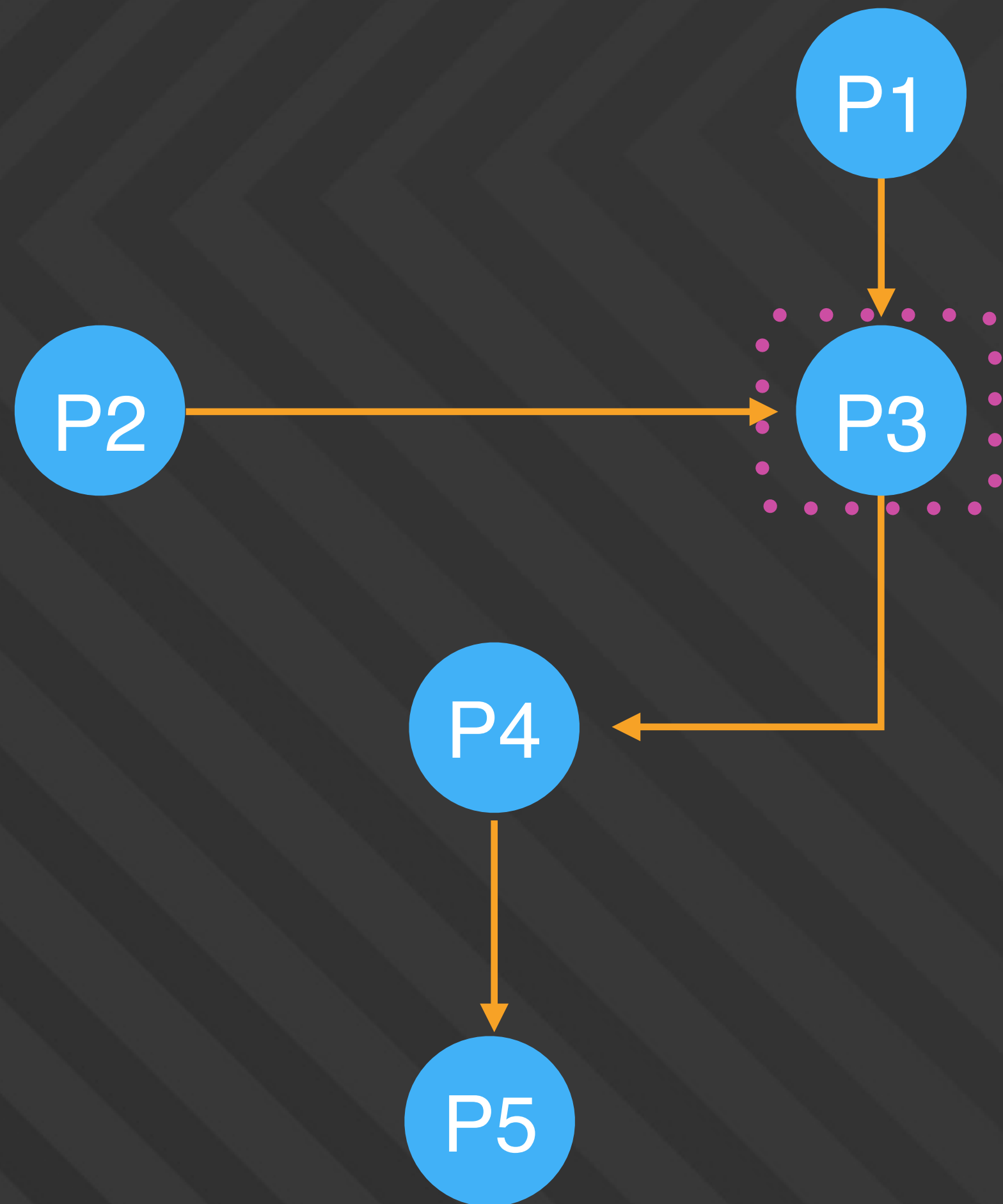
RacerD Design Principles

- Be *compositional*; don't do whole-program analysis
- Report races between *syntactically* identical access paths; don't attempt a general alias analysis
- Reason sequentially about memory accesses, locks, and threads; don't explore interleaving
- Occam's razor; *don't* use complex techniques (unless forced)

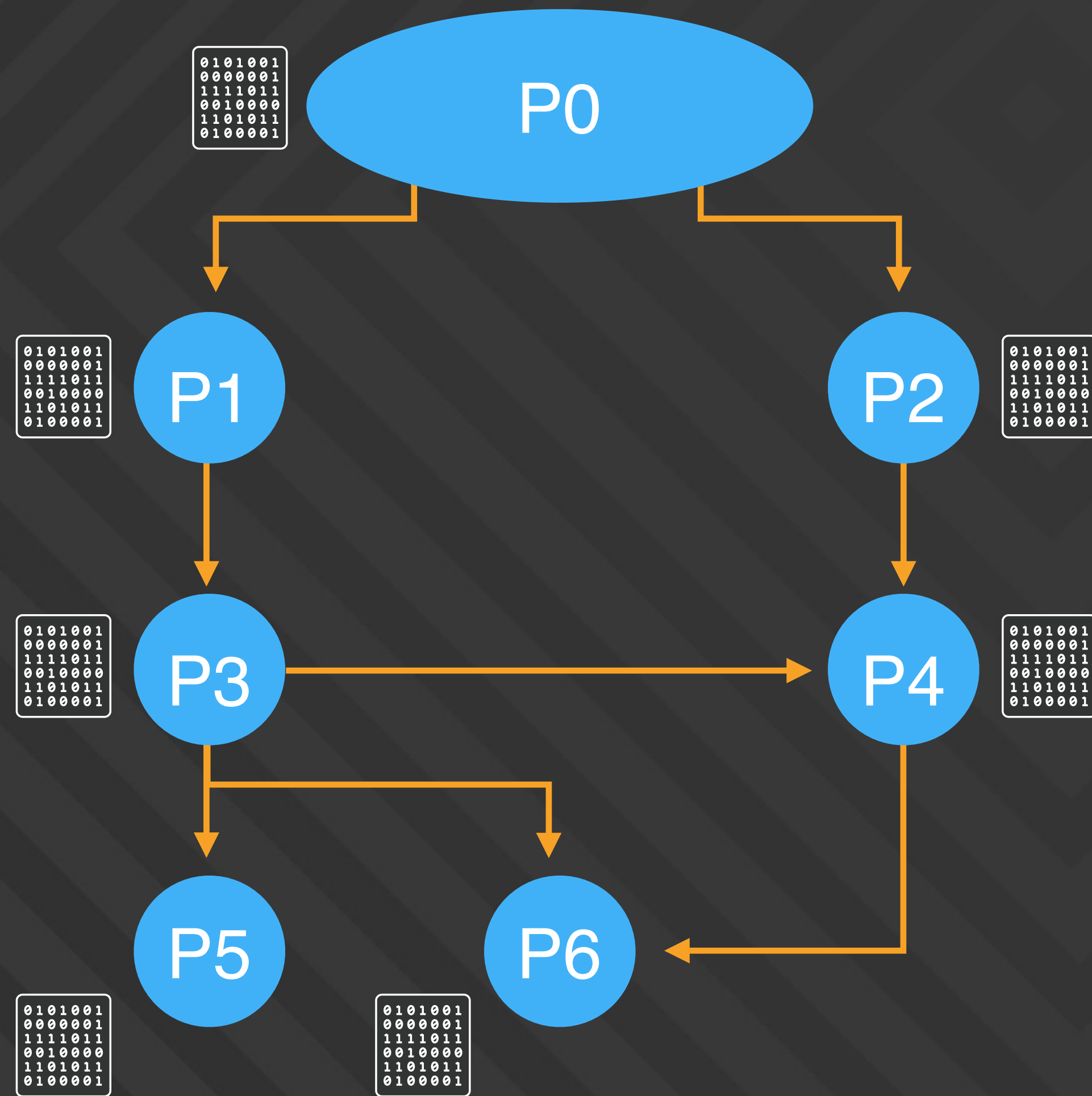
Background: compositional analysis

When analyzing P3:

- Will have summary for callee P4
- But don't know anything about callers P1, P2, or transitive callee P5
- Need to compute summary for P3 usable in any calling context



Background: compositional analysis



- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point

Scalable: analyze each procedure just once (without cycles)

Computing procedure summaries

```
Summary = { (access path, kind, locks) }
```

```
get      { (this.mCount, READ, 0) }
```

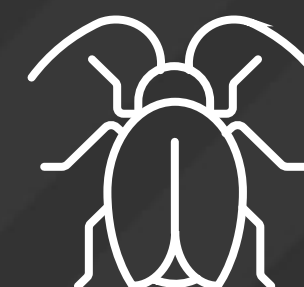
```
set      { (this.mCount, WRITE, 0) }
```

```
reset    { (this.mCount, WRITE, 1) }
```

```
class Counter {  
    private int mCount;  
  
    int get() {  
        return this.mCount;  
    }  
  
    private void set(int i) {  
        this.mCount = i;  
    }  
  
    synchronized void reset() {  
        set(0);  
    }  
  
    ...  
}
```



get and **reset** access same memory location
reset performs a write under synchronization
get uses no synchronization



RacerD abstract domains

(1) Can access touch the same memory?

(2) Can accesses happen concurrently?

access snapshots $A \subseteq \mathcal{A} = \{\langle \pi, k, \ell, t, o \rangle \mid k \in \{\text{rd}, \text{wr}\}\}$

(1) access paths $\pi \in \text{Path} = \text{Var} \times \text{Field}^*$

(2) lock count $\ell \in \mathcal{L} = \mathbb{N}^\top$

(2) concurrent threads $t \in \mathcal{T} = \text{NoThread} \sqsubset \text{AnyThreadButMain} \sqsubset \text{AnyThread}$

(1) ownership value $o \in \mathcal{O} = \text{OwnedIf}(\wp(\mathbb{N})) \sqsubset \text{Unowned}$

(1) ownership environment $E \subseteq \mathcal{E} = \text{Path} \rightarrow \mathcal{O}$

domain $d \in \mathcal{D} = \mathcal{L} \times \mathcal{T} \times \mathcal{E} \times \mathcal{A}$

Concurrent context inference

Accesses are **NoThread** unless we see:

Android thread assertions

Introduce **AnyThreadButMain**

```
void m() {  
    ThreadUtils.assertMainThread();  
    this.f = ...  
}
```

java.util.concurrent annots

```
@ThreadSafe class C { ...
```

Lock usage

```
synchronized void m() { this.f = ... }
```

Introduce **AnyThread**

concurrent threads $t \in \mathcal{T}$ = NoThread \sqsubset AnyThreadButMain \sqsubset AnyThread

Ownership identifies clearly safe accesses

```
Obj local = new Obj();  
local.f = ... // safe write
```

Freshly
allocated
ownership

```
Obj objFactory() {  
    return new Obj();  
}  
  
Obj local = objFactory();  
local.f = ... // safe write
```

Returning
ownership

concurrent threads $t \in \mathcal{T}$ = NoThread \square AnyThreadButMain \square AnyThread

Ownership can be conditional

```
private void writeF(Obj a) {  
    a.f = ...  
}
```

```
Obj o = new Obj();  
writeF(o); // safe
```

Safe if formal is owned by caller

```
Builder setX(X x) {  
    this.x = x;  
    return this;  
}
```

```
new Builder().setX(x).setY(y); // safe  
global.set(X).t = /; // not safe
```

Returns ownership if
formal is owned by caller

Ownership preconditions in summaries

```
private void writeF(Obj a) {  
    a.f = ...  
}  
summ: { a.f if ¬owned(a) }
```

```
Obj o = new Obj();  
owned(o)  
writeF(o);  
owned(o) ∧ { o.f if ¬owned(o) }  
owned(o) ∧ {}
```

Ownership post can depend on ownership pre

```
Builder setX(X x) {  
    this.x = x;  
    return this;  
}  
summ: { (this.x if ¬owned(this) ) ^  
        owned(ret) if owned(this)
```

```
owned(a)  
Builder b = a.setX(x);  
owned(a) ^ owned(b) if owned(a)  
          ^ { (a.x if ¬owned(a) )  
owned(a) ^ owned(b) ^ {}  
b.setY(y); // safe by similar reasoning
```

RacerD abstract domains

(1) Can access touch the same memory?

(2) Can accesses happen concurrently?

access snapshots $A \subseteq \mathcal{A} = \{\langle \pi, k, \ell, t, o \rangle \mid k \in \{\text{rd}, \text{wr}\}\}$

(1) access paths $\pi \in \text{Path} = \text{Var} \times \text{Field}^*$

(2) lock count $\ell \in \mathcal{L} = \mathbb{N}^\top$

(2) concurrent threads $t \in \mathcal{T} = \text{NoThread} \sqsubset \text{AnyThreadButMain} \sqsubset \text{AnyThread}$

(1) ownership value $o \in \mathcal{O} = \text{OwnedIf}(\wp(\mathbb{N})) \sqsubset \text{Unowned}$

(1) ownership environment $E \subseteq \mathcal{E} = \text{Path} \rightarrow \mathcal{O}$

domain $d \in \mathcal{D} = \mathcal{L} \times \mathcal{T} \times \mathcal{E} \times \mathcal{A}$

Reporting:

Can $\langle \pi_1, o_1, \ell_1, t_1, k_1 \rangle$ and $\langle \pi_2, o_2, \ell_2, t_2, k_2 \rangle$ race?

two **concurrent** accesses to
the **same memory location**
where **at least one is a write.**

$$(1) \pi_1 = \pi_2$$

$$(2) o_1 \neq \text{Owned} \wedge o_2 \neq \text{Owned}$$

$$(3) \ell_1 = \mathbf{0} \vee \ell_2 = \mathbf{0}$$

$$(4) t_1 \sqcup t_2 = \text{AnyThread}$$

$$(5) k_1 = \text{wr} \vee k_2 = \text{wr}$$

Applying summaries: see paper

$$\text{apply}(\ell_{\text{caller}}, \ell_{\text{callee}}) \triangleq \ell_{\text{caller}} +^{\top} \ell_{\text{callee}} \quad (\text{A})$$

$$\text{apply}(t_{\text{caller}}, t_{\text{callee}}) \triangleq \begin{cases} \text{AnyThreadButMain} & \text{if } t_{\text{callee}} = \text{AnyThreadButMain} \\ t_{\text{caller}} & \text{otherwise} \end{cases} \quad (\text{B})$$

$$E(e) \triangleq \begin{cases} o & \text{if } e = \pi \text{ and } (\pi, o) \in E \\ \text{Unowned} & \text{if } e = \pi \text{ and } \pi \notin \text{dom}(E) \\ \text{OwnedIf}(\emptyset) & \text{otherwise} \end{cases} \quad (\text{C})$$

$$\text{apply}(E_{\text{caller}}, \vec{e}, o_{\text{callee}}) \triangleq \begin{cases} \text{Unowned} & \text{if } o_{\text{callee}} = \text{Unowned} \\ \text{OwnedIf}(\emptyset) & \text{if } o_{\text{callee}} = \text{OwnedIf}(\emptyset) \\ \bigsqcup_{i \in N} E_{\text{caller}}(e_i) & \text{if } o_{\text{callee}} = \text{OwnedIf}(N) \end{cases} \quad (\text{D})$$

$$\text{apply}(E_{\text{caller}}, \vec{e}, \pi_{\text{ret}}, E_{\text{callee}}) \triangleq E_{\text{caller}}[\pi_{\text{ret}} \mapsto \text{apply}(E_{\text{caller}}, E_{\text{callee}}(\pi_{\text{ret}}), \vec{e})] \quad (\text{E})$$

$$\text{subst}(\pi, \vec{e}) \triangleq \begin{cases} y.f_1 \cdots f_n.g_1 \cdots g_m & \text{if } \begin{cases} \pi = x.g_1 \cdots g_m, \\ x \text{ is the } i\text{th formal, and} \\ e_i = y.f_1 \cdots f_n \end{cases} \\ \pi & \text{otherwise} \end{cases}$$

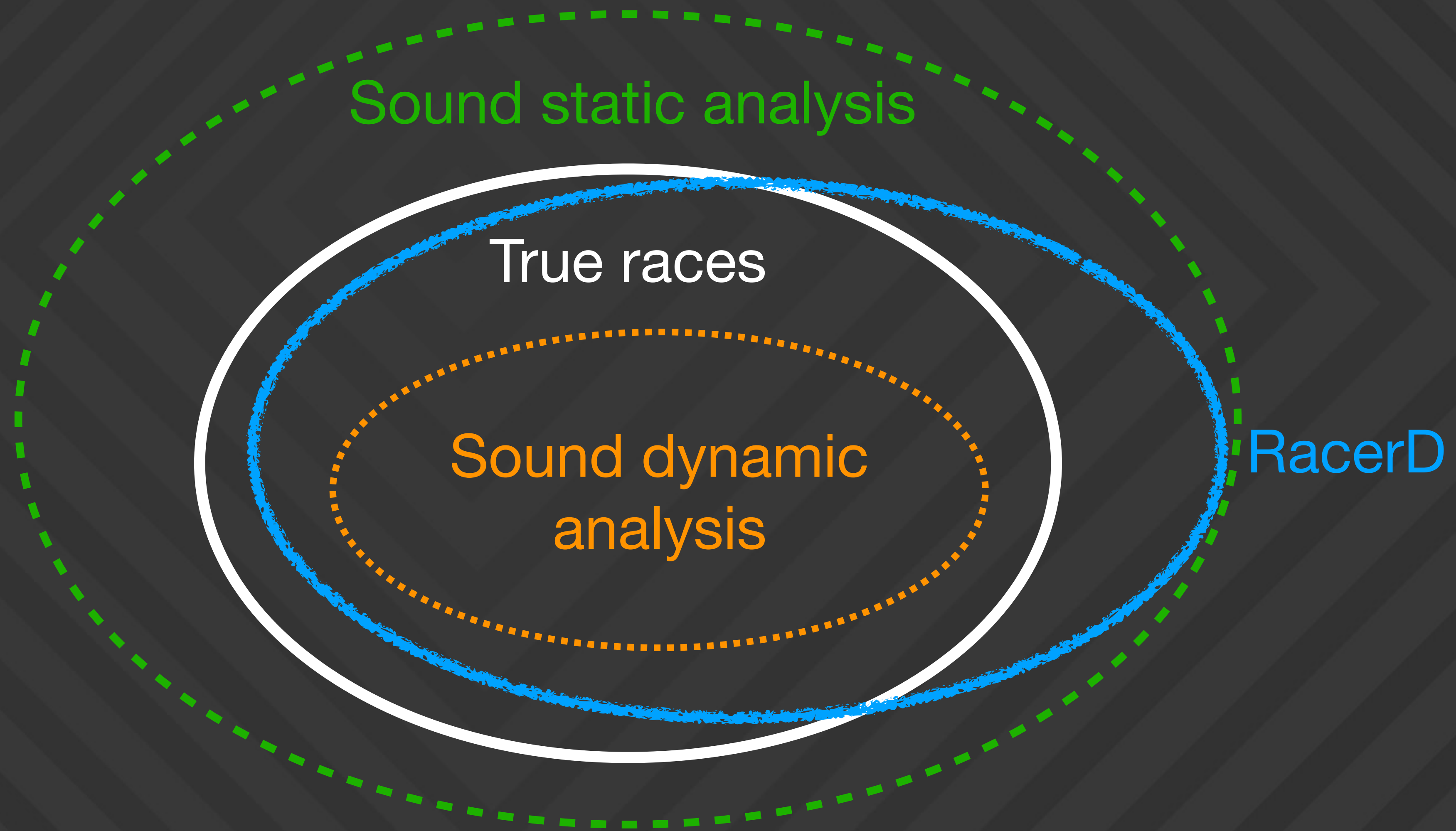
$$\text{apply}(A_{\text{caller}}, \ell_{\text{caller}}, t_{\text{caller}}, E_{\text{caller}}, \vec{e}, A_{\text{callee}}) \triangleq A_{\text{caller}} \sqcup \left\{ \left\langle \pi', k, \ell', t', o' \right\rangle \left| \begin{array}{l} \pi' = \text{subst}(\pi, \vec{e}), \\ \ell' = \text{apply}(\ell_{\text{caller}}, \ell), \\ t' = \text{apply}(t_{\text{caller}}, t), \\ o' = \text{apply}(E_{\text{caller}}, o, \vec{e}), \\ \langle \pi, k, \ell, t, o \rangle \in A_{\text{callee}} \end{array} \right. \right\} \quad (\text{F})$$

Fig. 5. Applying a callee summary at a call site $\pi_{\text{ret}} := \text{call } m(\vec{e})$.

RacerD: compositional static race detection

- (1) Motivation: assist Litho migration + detect regressions
- (2) RacerD deep dive: design, domains, reporting
- (3) Evaluation: RacerD vs static/dynamic race detectors, RacerD @ FB

RacerD vs Static and Dynamic Analysis tools



Want: more true races than dynamic,
fewer false positives than static

RacerD vs Chord (static)

- Ran RacerD on DaCapo benchmarks used by Chord (orig. PLDI'06) in PLDI'18 paper
- RacerD reports $>3X$ fewer alarms in all cases, $>10x$ fewer on half of the benchmarks
- RacerD found all true bugs reported by Chord (+ more!)
- Time: 0.6s — 25s (RacerD) vs 1m — 1h 47m (Chord)

RacerD vs Chord (static)

Program	# Files	# LOC	CHORD		RACERD		Alarm Overlap
			# Alarms	Runtime	# Alarms	Runtime	
hedc	133	11,767	152	4m 47s	49	1.5s	11%
ftp	140	12,050	522	5m 14s	39	1.5s	20%
weblech	12	1,309	30	11m 02s	9	0.6s	63%
jspider	214	7,413	257	1m 54s	13	1.4s	10%
avrrora	470	68,864	966	8m 49s	81	7s	18%
luindex	331	36,151	940	3m 26s	183	5s	64%
sunflow	170	21,960	958	42m 44s	43	2.8s	49%
xalan	975	175,784	1870	1h 47m	421	21.5s	38%

RacerD vs DroidRacer (dynamic)

- Ran RacerD on DaCapo benchmarks used by DroidRacer in PLDI'14 paper
- RacerD found more true bugs than DroidRacer on each benchmark
- In two cases, also reported fewer false positives
- Time: 12s — 3m (RacerD) vs “”few seconds to few hours (DroidTacer)

RacerD reports

fewer false positives than sound *static* tools and
finds *more* bugs than *dynamic* tools.

Finding data race regressions

Impact (1y)

~500

PROGRAMMERS
REACHED

~7K

REPORTS

~4K

FIXES



RacerD made migration faster/easier/safer

- 100s of Litho components safely migrated to background layout in <6 mo by 2 engineers
- 1000+ potential races fixed during migration
- Only 3 false negatives reported by engineers (all analyzer bugs that we fixed)

Engineer Comments

better. The thread safety violations are doubly useful - since these help catch nasty and hard to debug bugs that can commonly happen in our multi-thread UI stack on Android

Infer was really instrumental in ensuring thread safety in Litho code. This allowed us to ship Newsfeed layout on a background thread and get huge wins in terms of scroll performance in FB4a

Without Infer, multithreading in News Feed would not have been tenable

RacerD Limitations

1. Races on aliased, syntactically distinct access paths not caught
 - Tried alias analysis, too many FPs. Couldn't deploy.
2. Misses races on local objects that escape, "deep" ownership analysis
 - Tried escape analysis, too many FPs. Couldn't deploy.
3. Not fully underapproximate
 - Tried join-as-meet, but eliminates too many valuable reports. Couldn't deploy
4. Works for simple mutex locks, but not subtler constructs like R/W locks
5. No reasoning about weak memory, volatile, or other fine-grained concurrency
6. No *soundness/completeness* theorem

Helps real programmers write concurrent code anyway

Try RacerD

<https://fbinfer.com/docs/racerd.html>

or Google “Facebook RacerD”

Checkpoint

- (1) **RacerD**: scalable + low annotation static race detection designed around completeness rather than soundness
- (2) Detected 1000s of bugs in prod at FB + enabled Litho migration
- (3) Can we have a "True Positives Theorem"?

Part 2

A True Positives Theorem for a Static Race Detector

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.

RACERD: Compositional Static Race Detection



SAM BLACKSHEAR, Facebook, USA

NIKOS GOROGIANNIS, Facebook, UK and Middlesex University London, UK

PETER W. O'HEARN, Facebook, UK and University College London, UK

ILYA SERGEY*, Yale-NUS College, Singapore and University College London, UK

Automatic static detection of data races is one of the most basic problems in reasoning about concurrency. We present RACERD—a static program analysis for detecting data races in Java programs which is fast, can scale to large code, and has proven effective in an industrial software engineering scenario. To our knowledge, RACERD is the first inter-procedural, compositional data race detector which has been empirically shown to have non-trivial precision and impact. Due to its compositionality, it can analyze code changes quickly, and this allows it to perform *continuous reasoning* about a large, rapidly changing codebase as part of deployment within a continuous integration ecosystem. In contrast to previous static race detectors, its design favors reporting high-confidence bugs over ensuring their absence. RACERD has been in deployment for over a year at Facebook, where it has flagged over 2500 issues that have been fixed by developers before reaching production. It has been important in enabling the development of new code as well as fixing old code: it helped support the conversion of part of the main Facebook Android app from a single-threaded to a multi-threaded architecture. In this paper we describe RACERD's design, implementation, deployment and impact.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Concurrent programming structures**;

Additional Key Words and Phrases: Concurrency, Static Analysis, Race Freedom

ACM Reference Format:

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RACERD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (November 2018), 28 pages. <https://doi.org/10.1145/3276514>

In theory ...

Unsound for verification
(misses bugs)

Unsound for testing
(reports non-bugs)

yet, over 6 months of use
and 2500 bugs reported

Only **3** known races missed

Less than 20% false positives

Can we *provably* make it 0%?

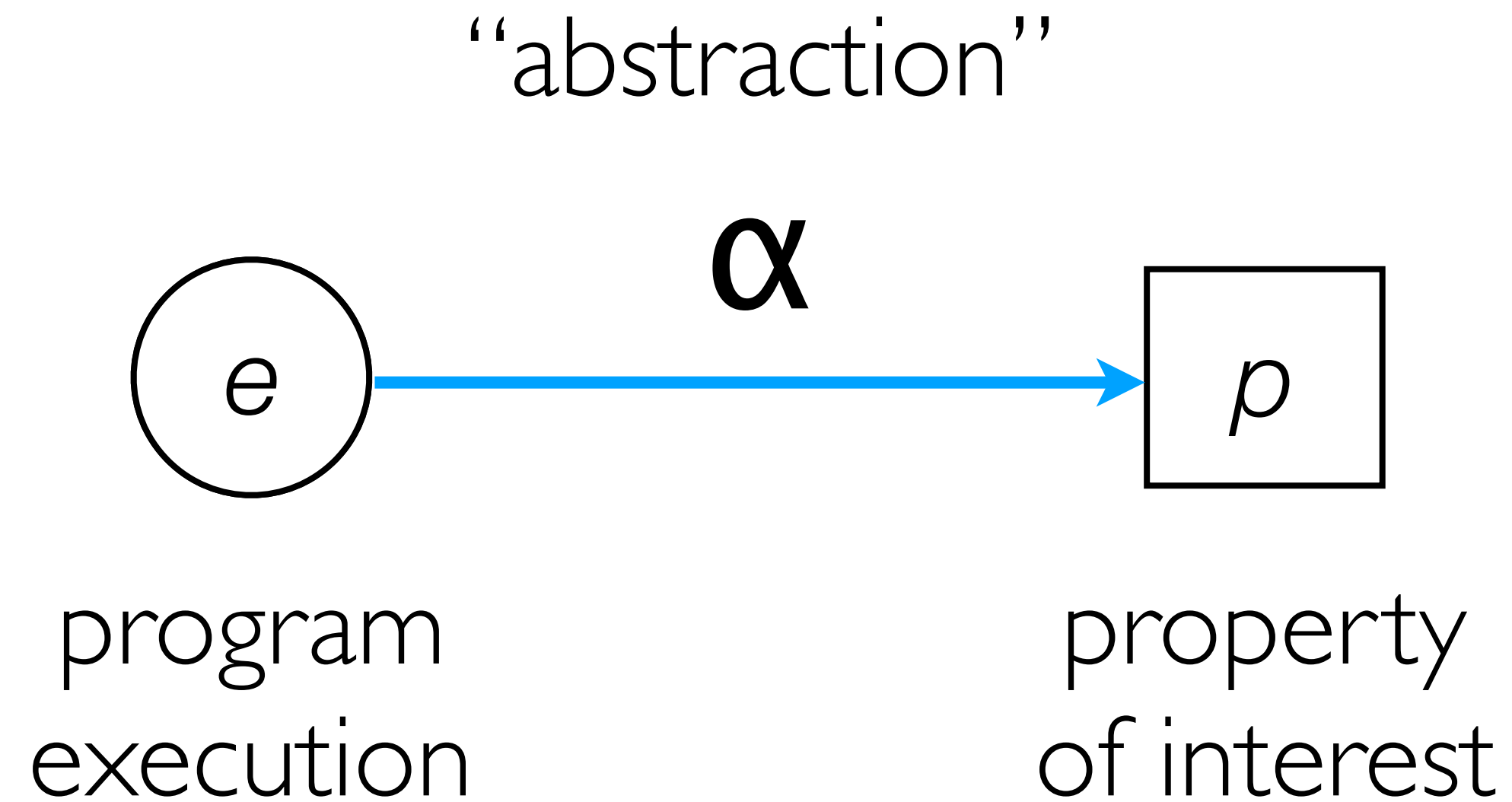
Conjecture

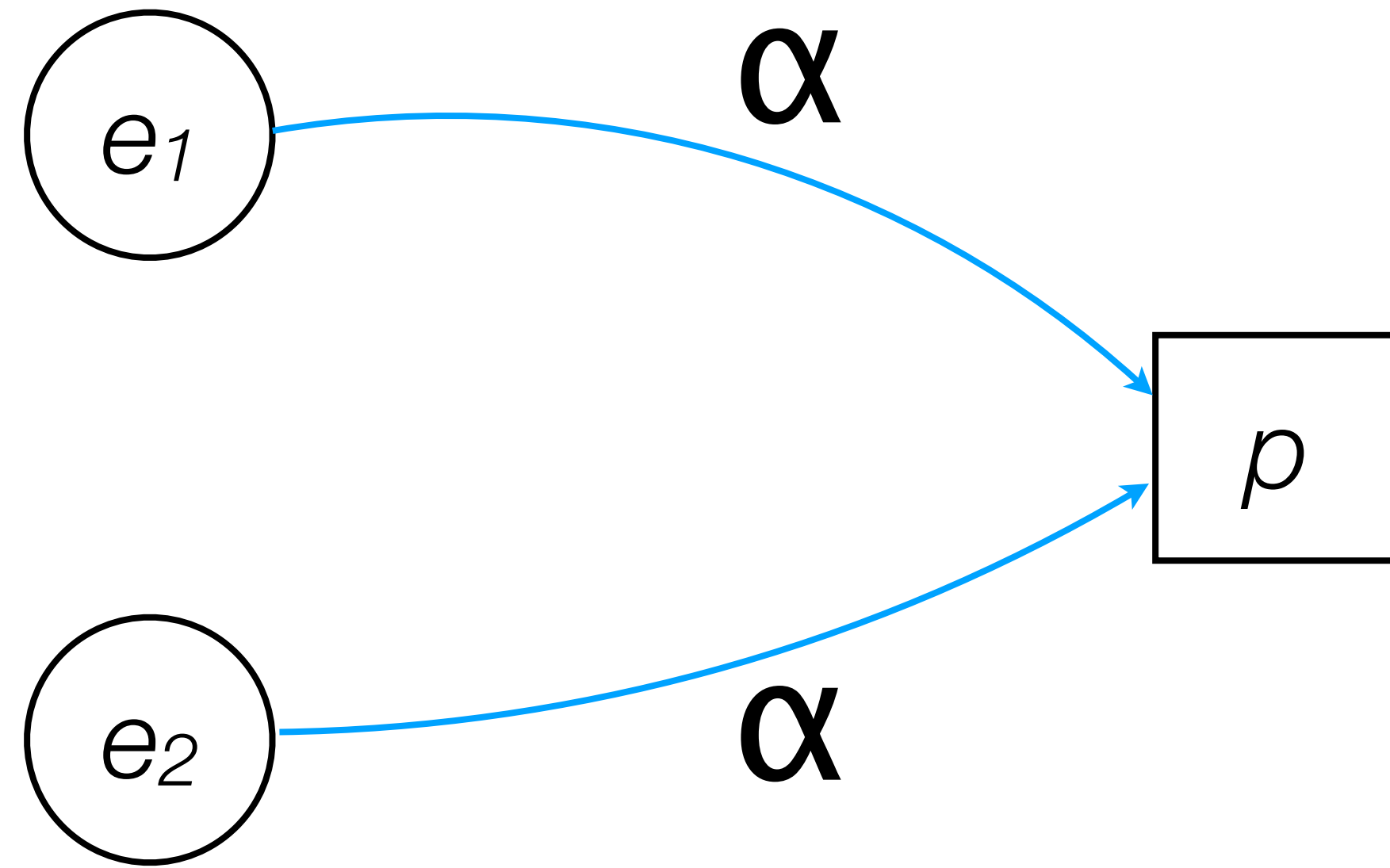
True Positives Theorem:

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

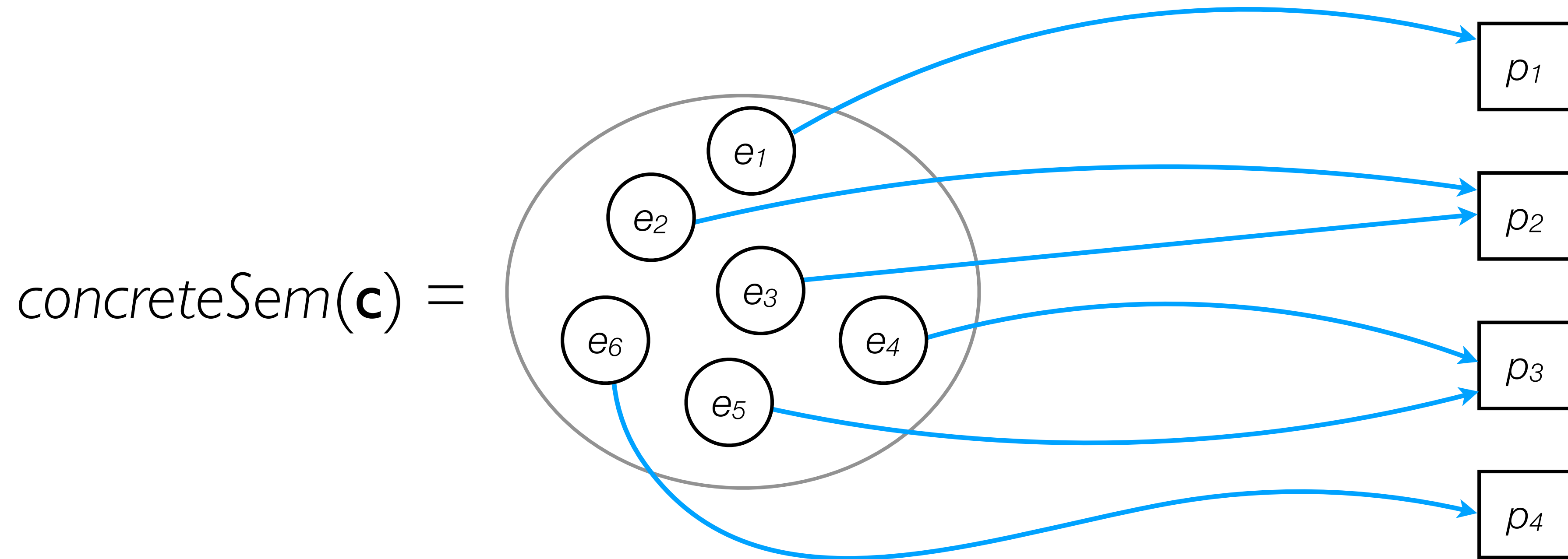
Static Analyses for Program Validation

The Essence of Static Analysis

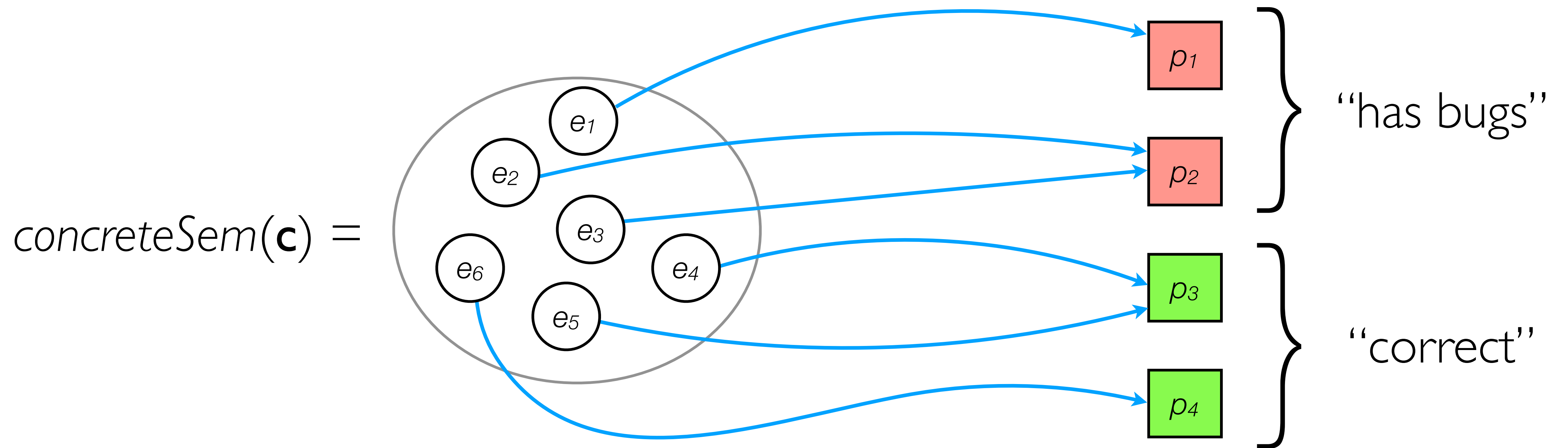




Static Analysis

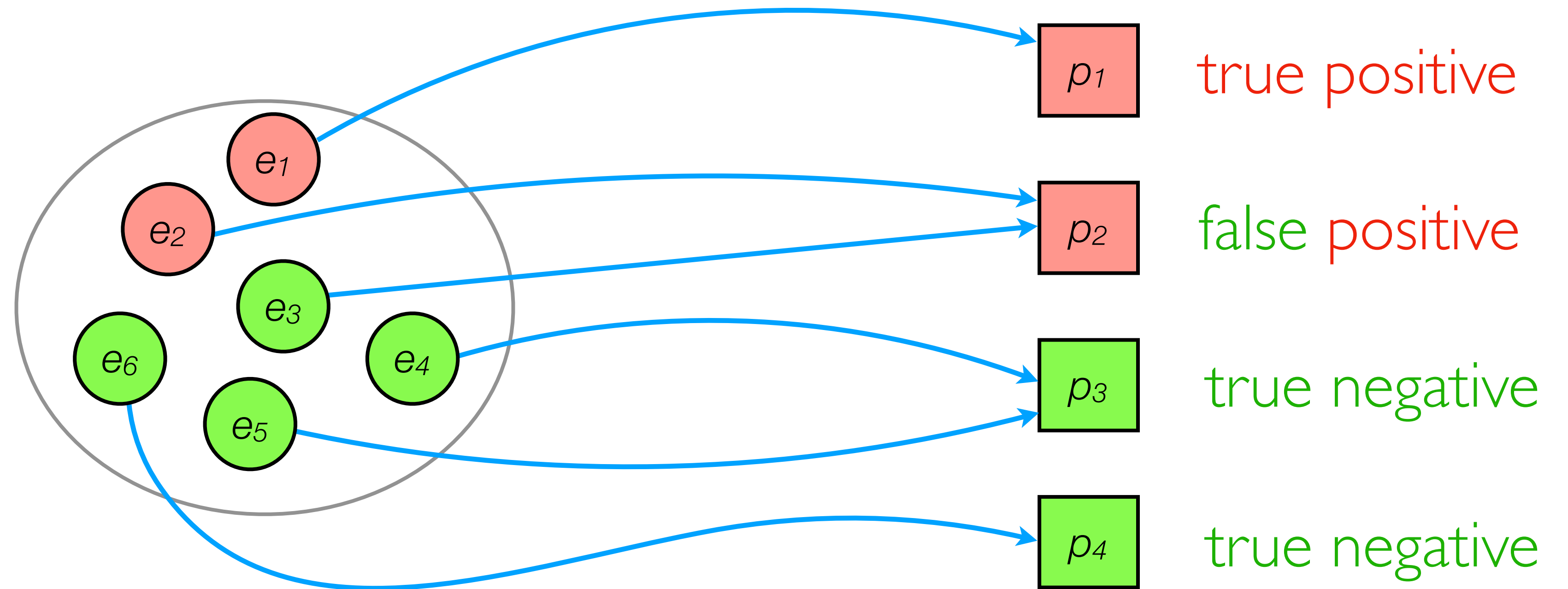


Static Analysis

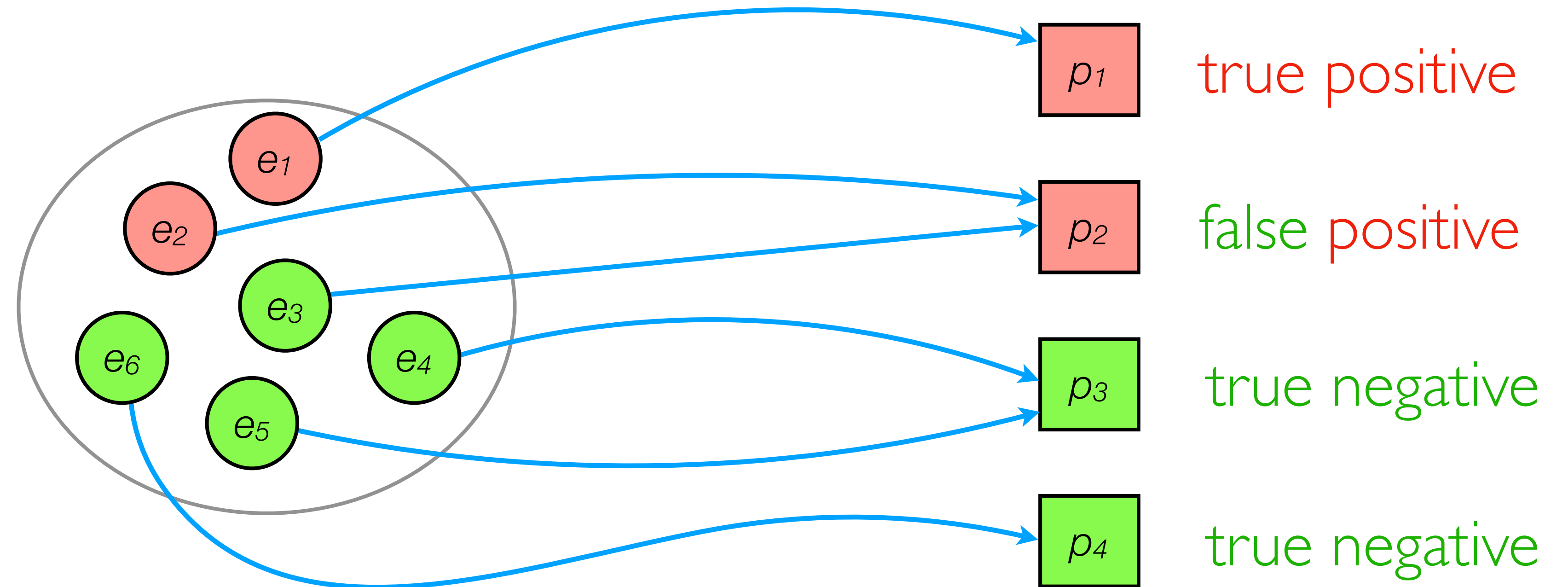


Verifier
or a
Bug Detector?

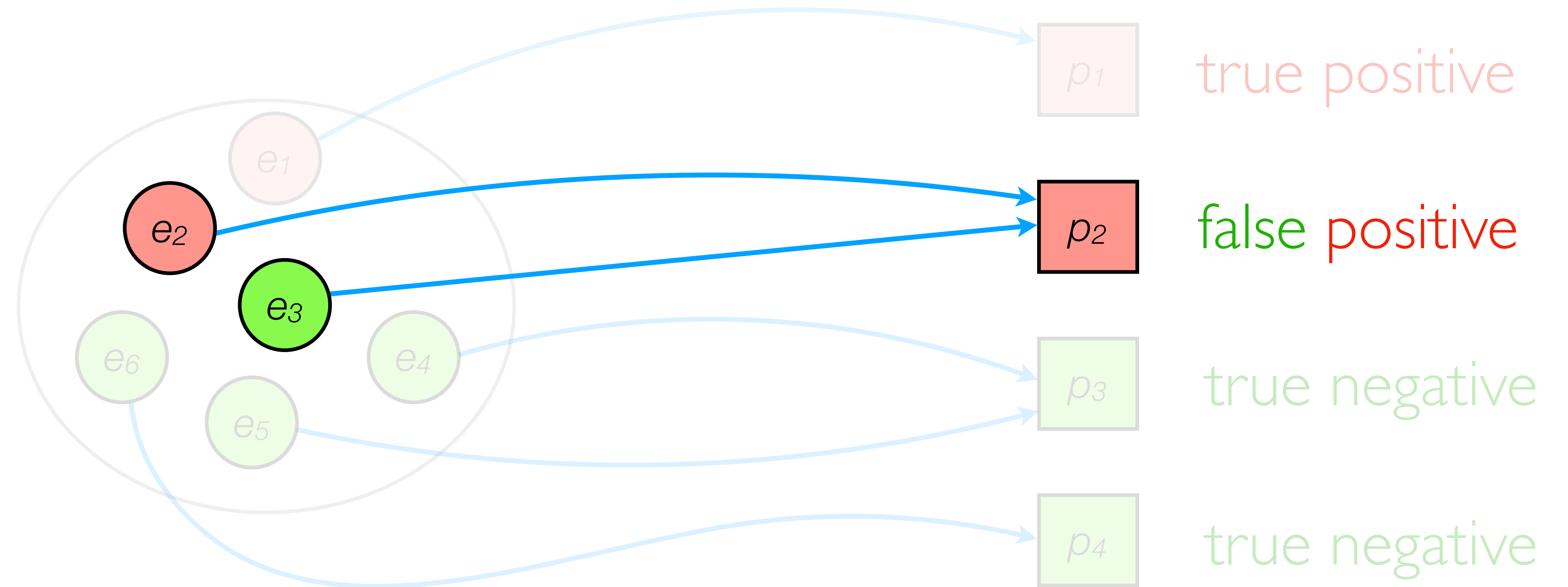
Program Verifier



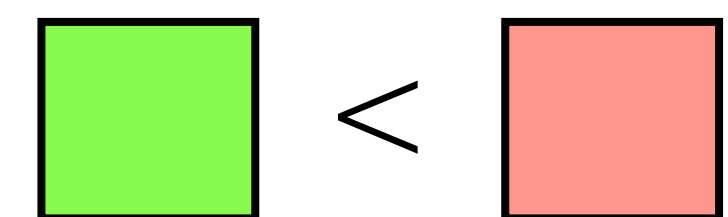
Sound Program Verifier



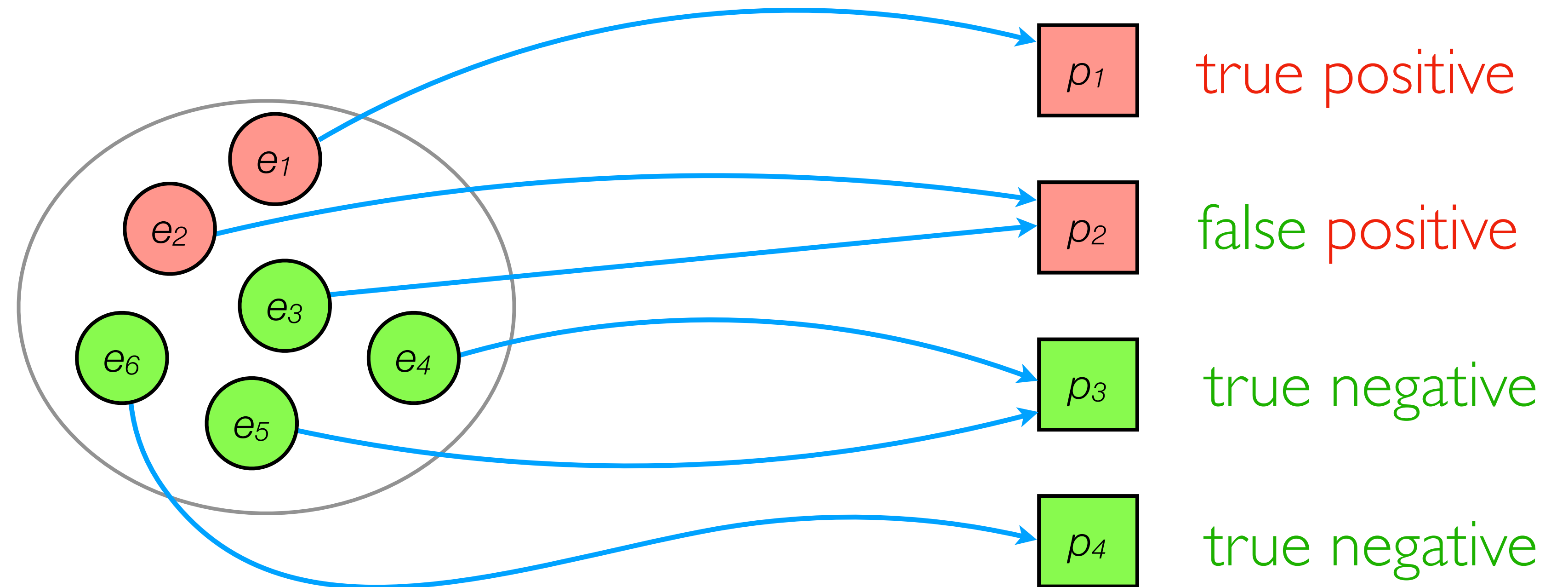
Sound Program Verifier



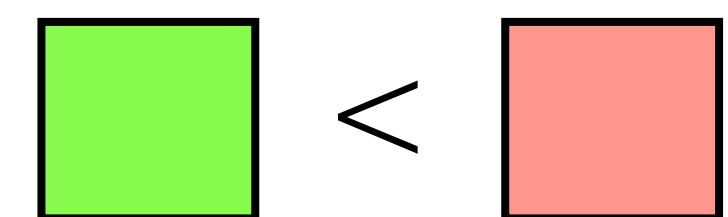
abstract over-approximation



Sound Program Verifier



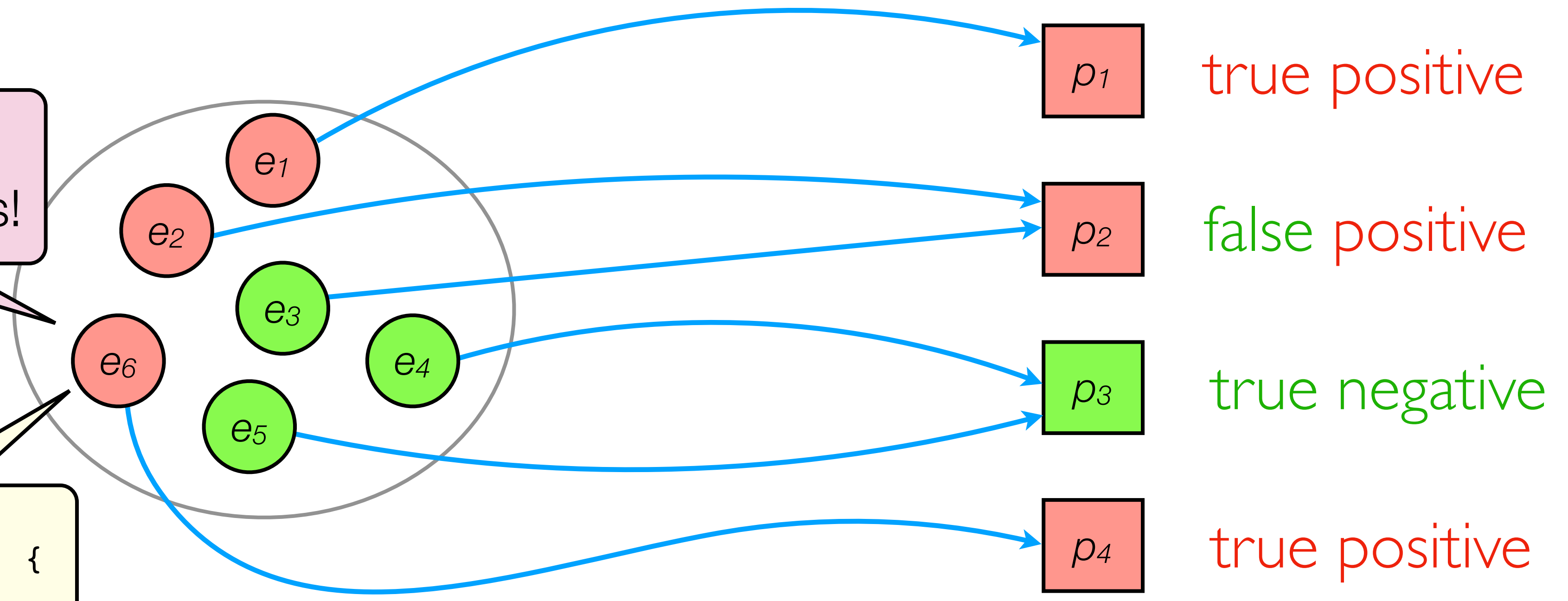
abstract over-approximation



Sound Program Verifier

Developer:
Go away, that never happens!

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



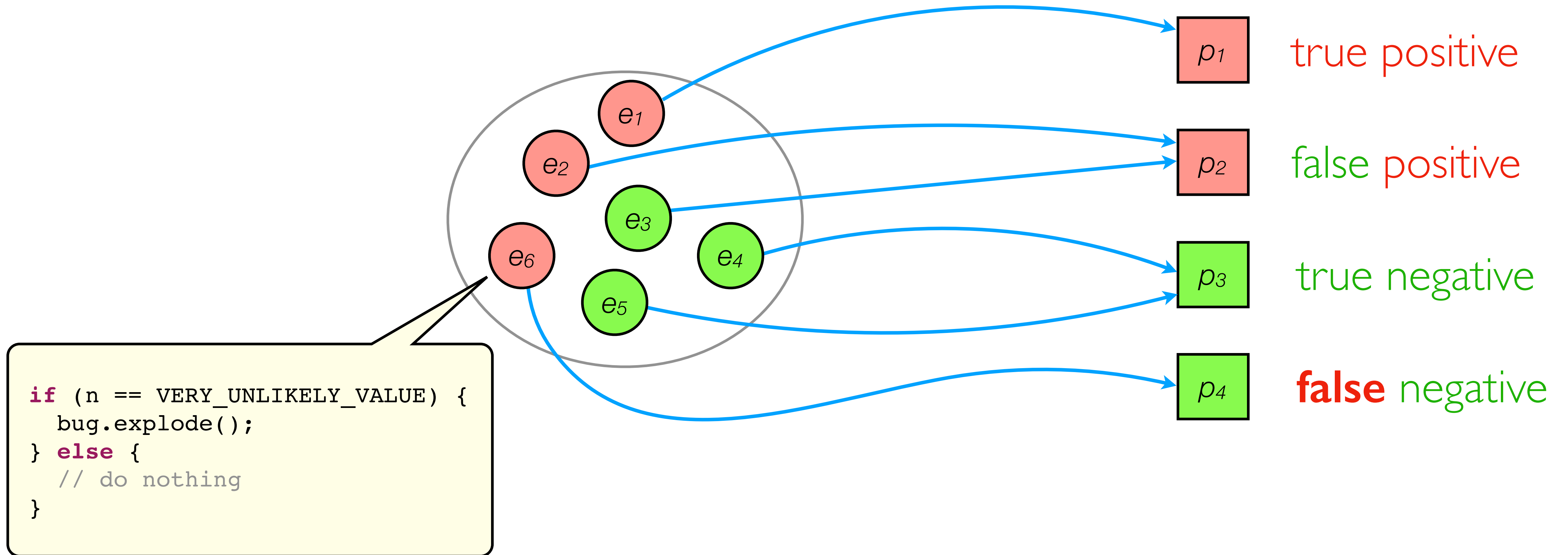
true positive

false positive

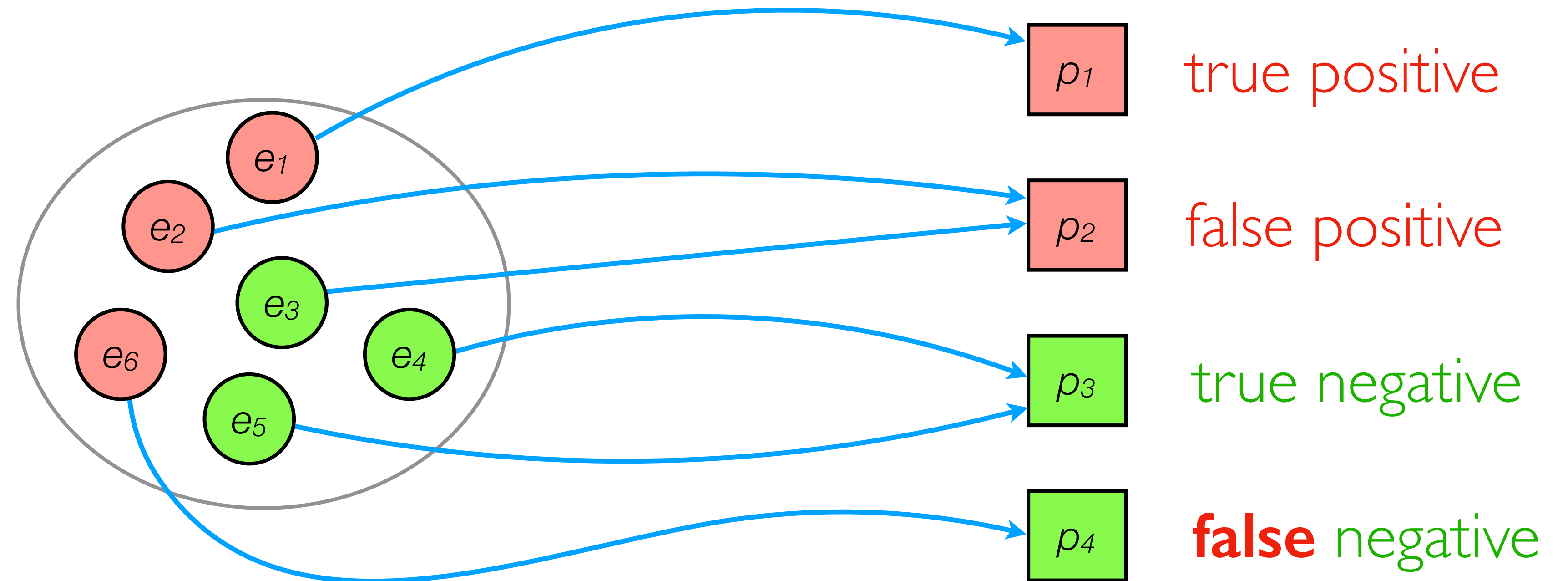
true negative

true positive

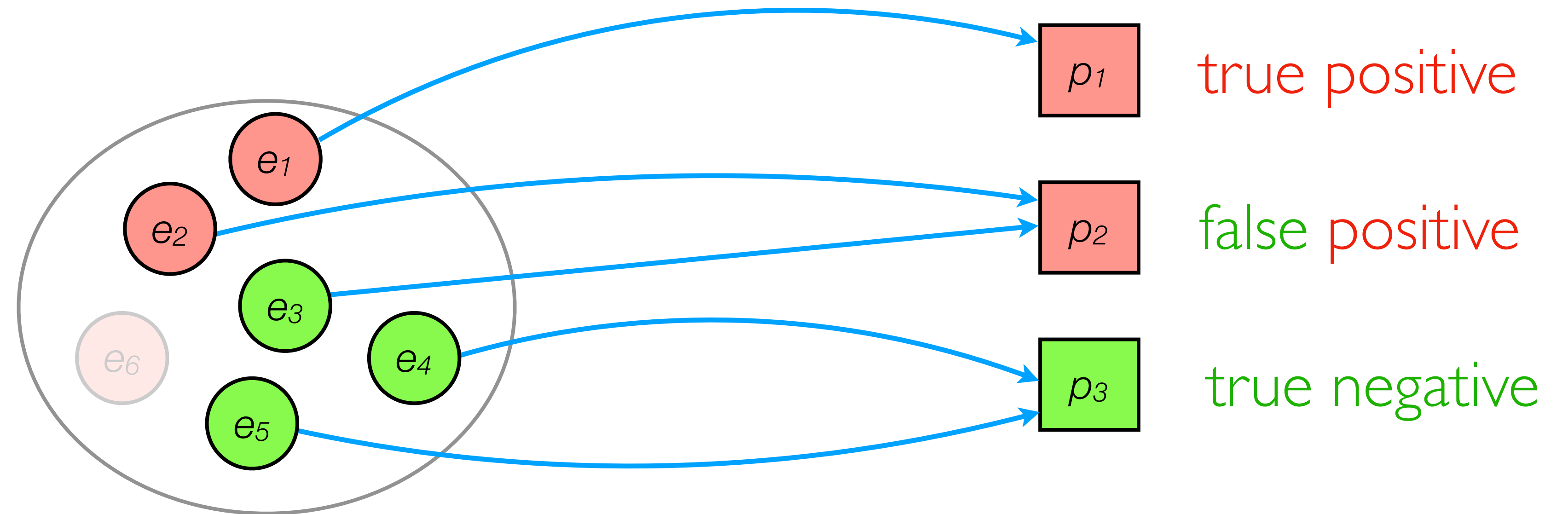
Unsound Program “Verifier”



“Sound” Program Verifier

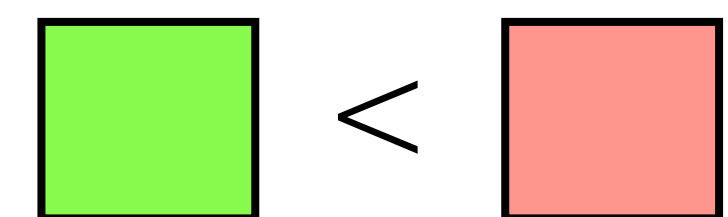


“Sound” Program Verifier



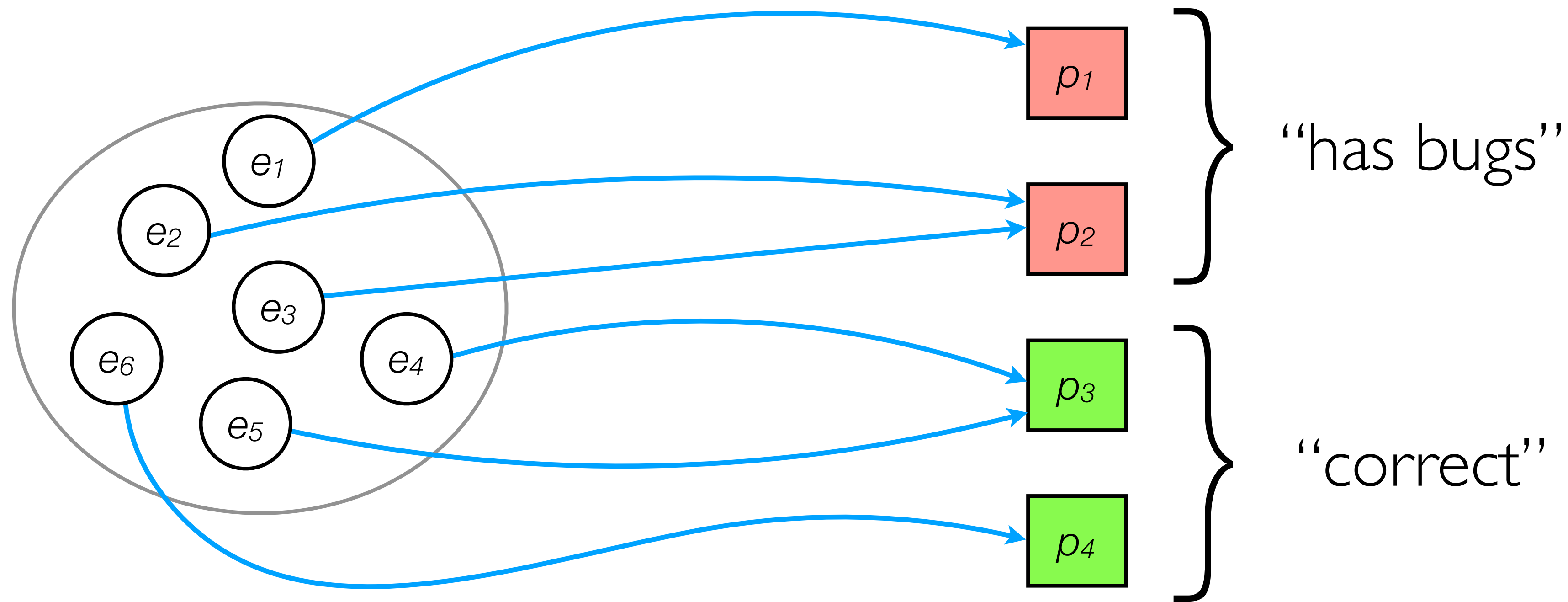
concrete under-approximation

abstract over-approximation

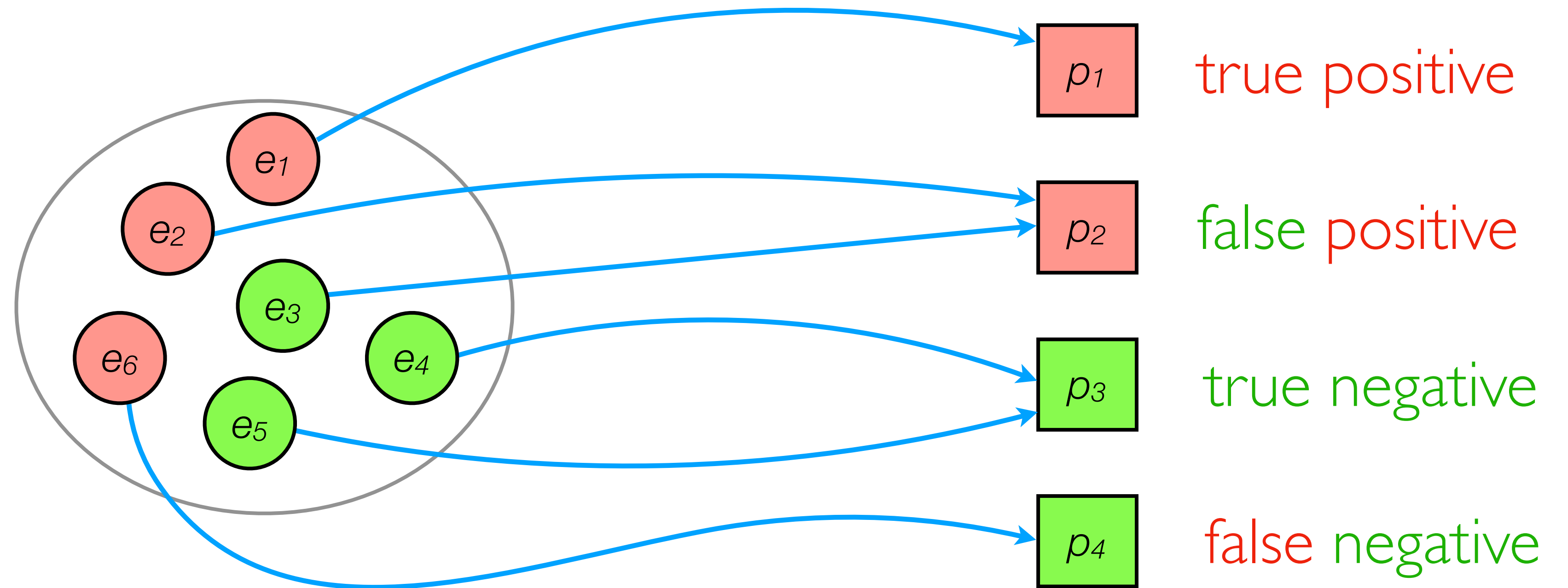


Sound Static Verifiers

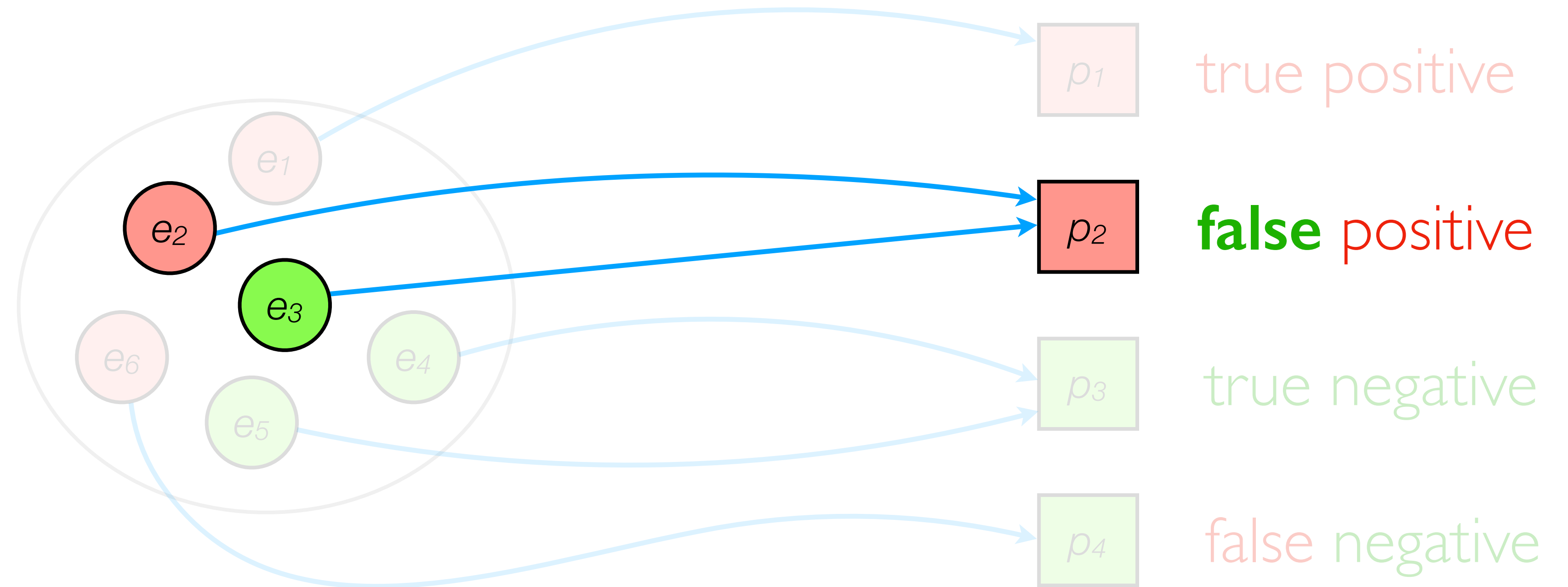
- **False negatives** (bugs missed) are **bad**
- **False positives** (non-bugs reported) are **okay**
- Constructed as **over-approximation** (of **under-approximation**)
- **Soundness Theorem:**
Under certain assumptions about the programs, the analyser has *no false negatives*.



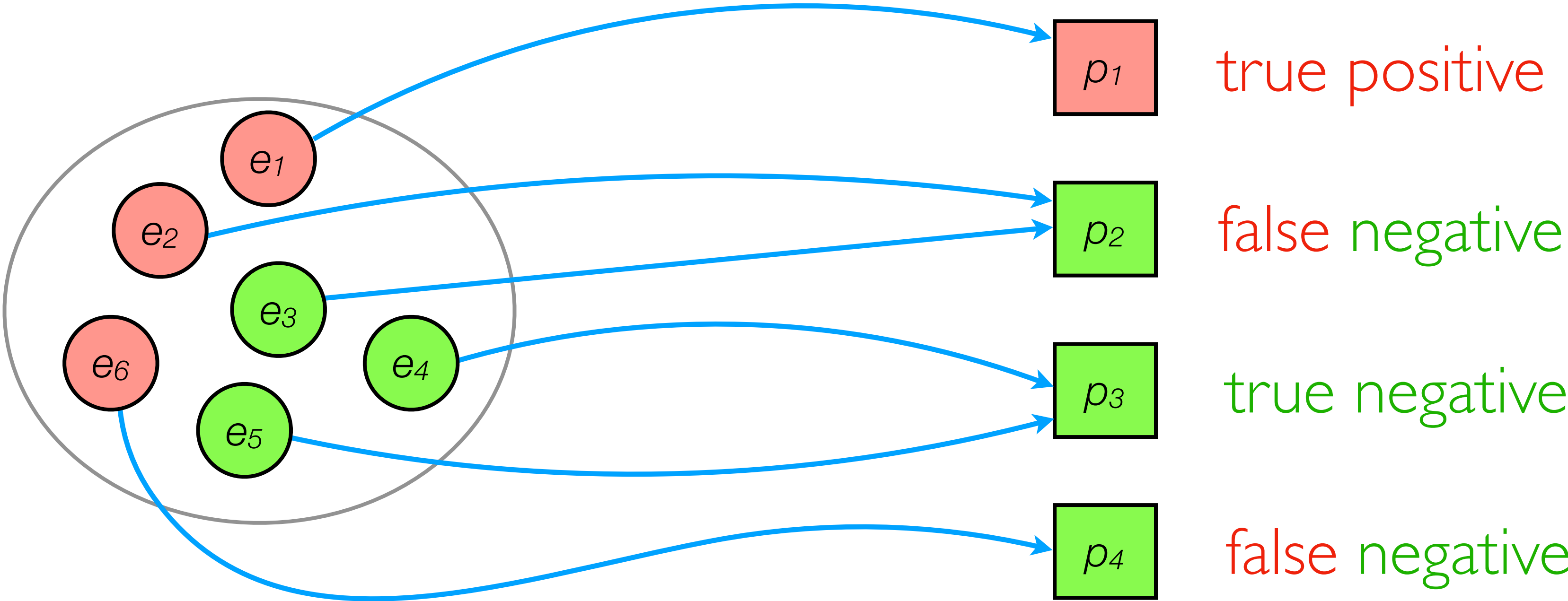
Static Bug Finder



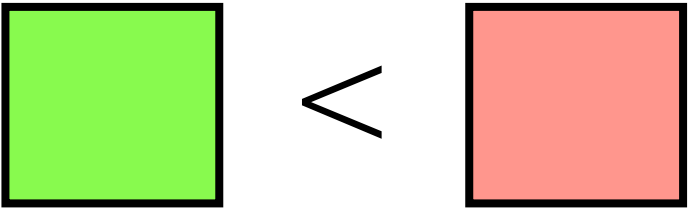
Unsound Static Bug Finder



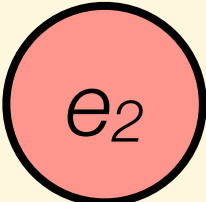
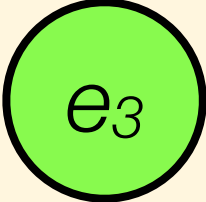
Sound (but imprecise) Static Bug Finder



abstract *under-approximation*



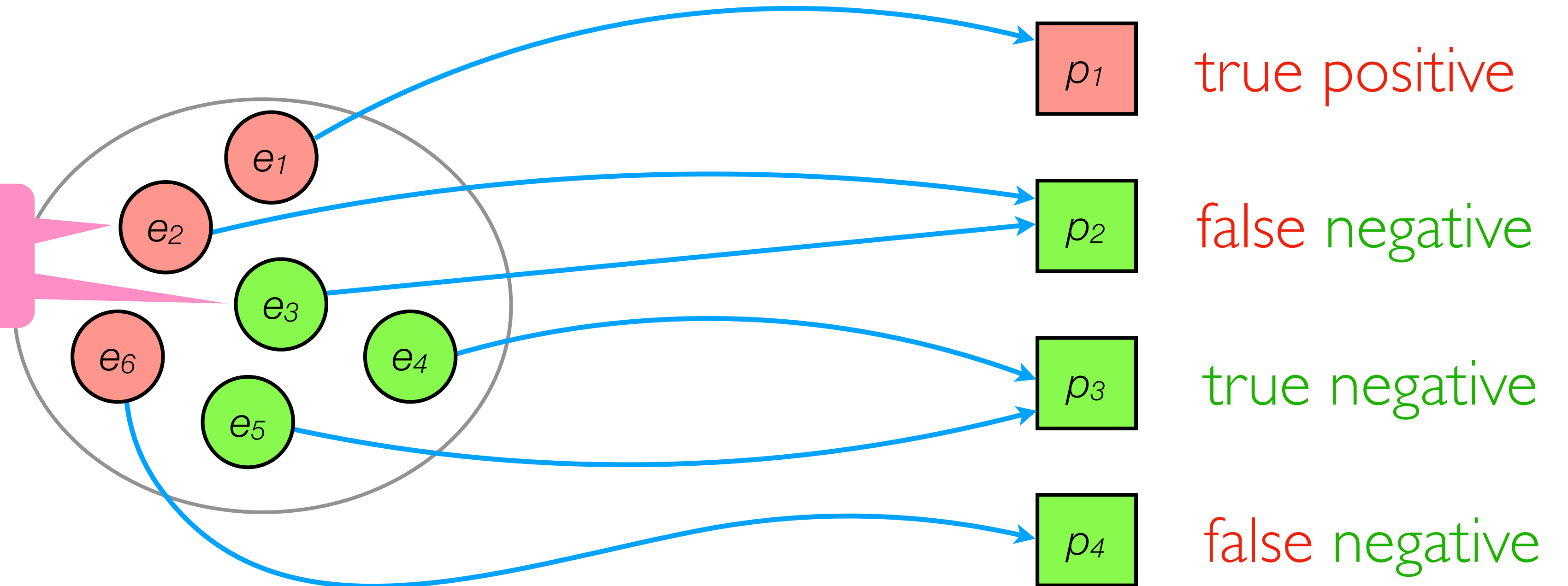
Loss of Precision in Static Bug Finders

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

Idea: *over-approximate* in concrete semantics!

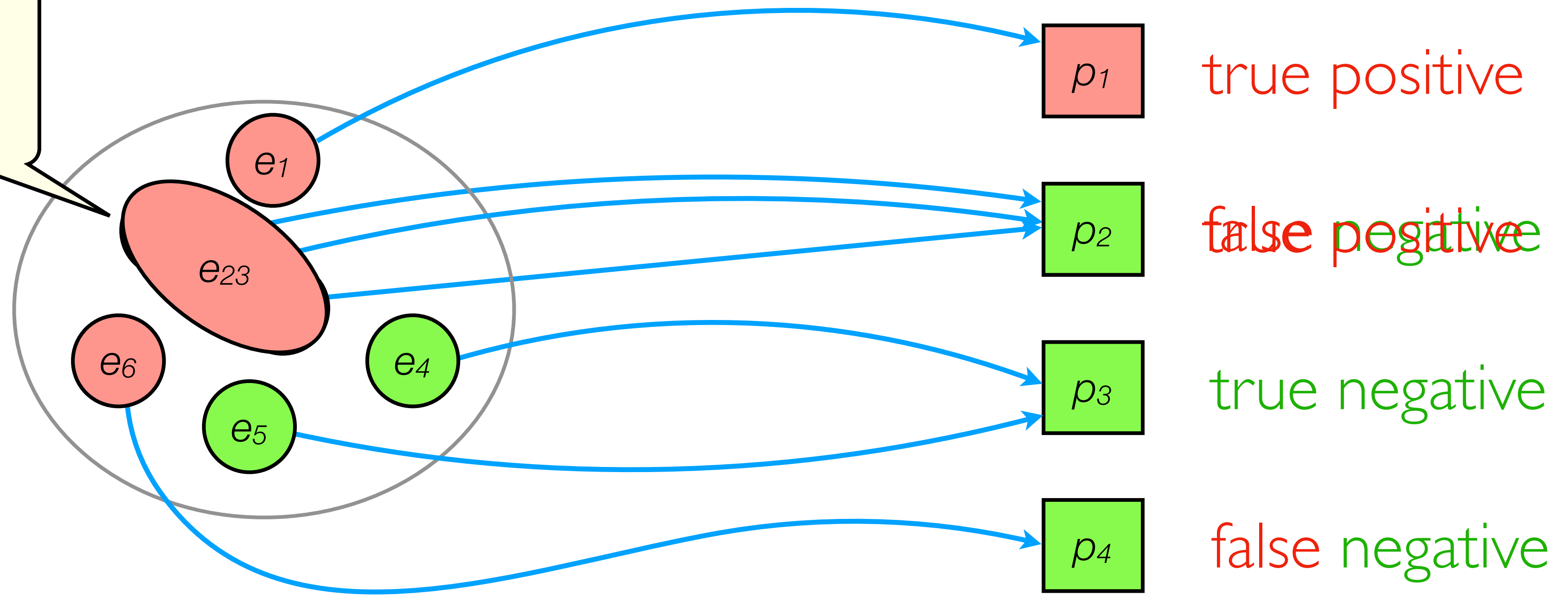
Sound (but Imprecise) Static Bug Finder

Let's merge these executions into one that subsumes both!



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

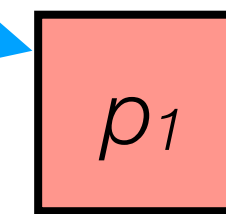
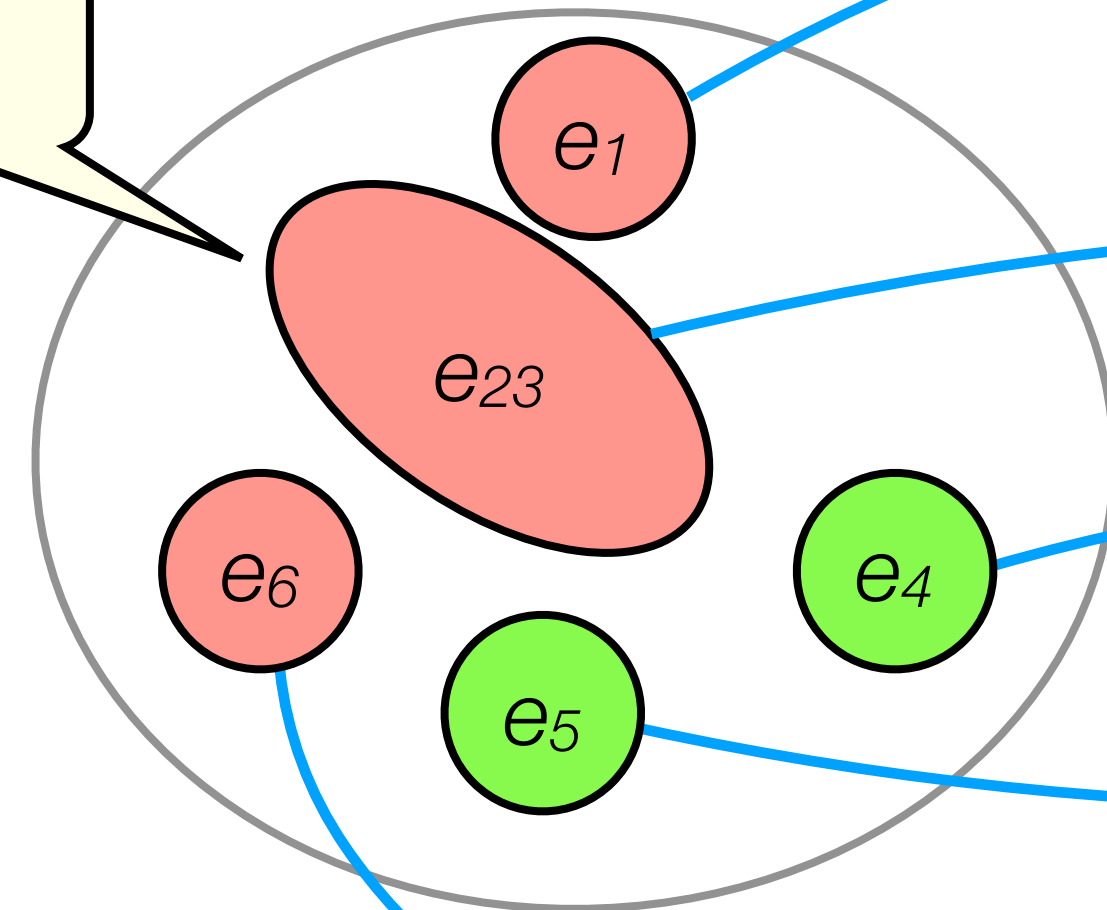
overApproxConcreteSem(c) =



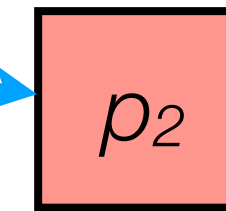
Sound Static Bug Finder

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

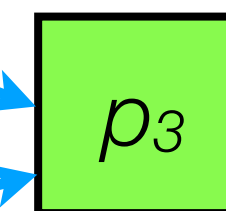
$\text{overApproxConcreteSem}(c) =$



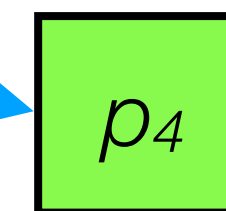
true positive



true positive



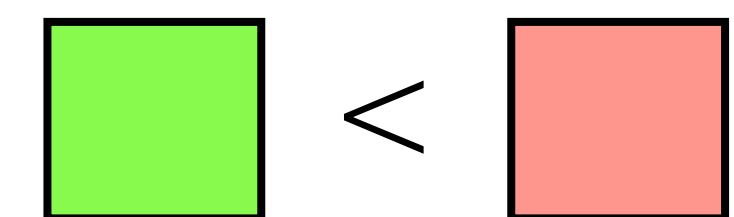
true negative



false negative

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:

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

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

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

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
- **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

```
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;  
    }  
}
```

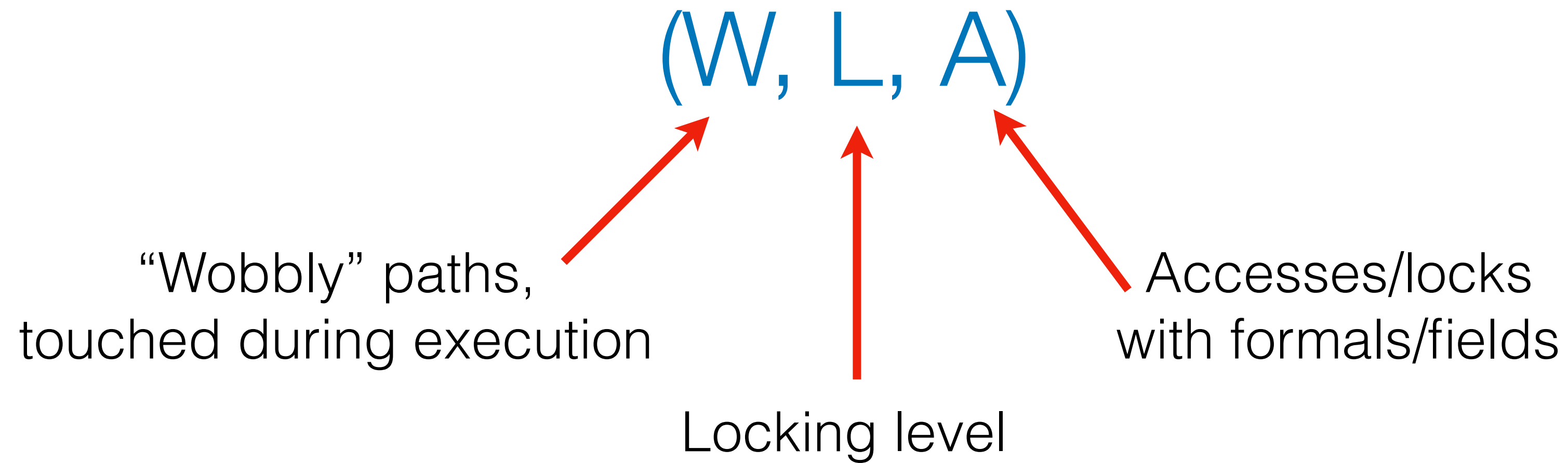
A False Race

```
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



- $\text{asem}(\text{meps}(\mathbf{b})) = (\{\mathbf{b.f}\}, 0, \{\mathbf{R}(\mathbf{b.f}, 1)\})$
- $\text{asem}(\text{reps}(\mathbf{b})) = (\{\mathbf{b.f}\}, 0, \{\mathbf{W}(\mathbf{b.f}, 0)\})$
- $\text{asem}(\text{beps}(\mathbf{b})) = (\{\mathbf{b}, \mathbf{b.f}\}, 0, \{\mathbf{W}(\mathbf{b}, 0), \mathbf{W}(\mathbf{b.f}, 0)\})$

```
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;  
    }  
}
```

Analysing Summaries for Races

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

$\text{meps}(b) \parallel \text{reps}(b) \Rightarrow$ **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

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

$\text{meps}(b) \quad || \quad \text{beps}(b) \Rightarrow$ Maybe don't race,
don't 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;  
    }  
}
```


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 $\pm\%$	D Reps	DX Reps	Reps $\pm\%$
avro	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 $\pm\%$	D Reps	DX Reps	Reps $\pm\%$
avro	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 $\pm\%$	D Reps	DX Reps	Reps $\pm\%$
avro	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%

What else is in the POPL'19 paper

- Formal definitions of races, concrete and abstract semantics
- Proof of the TP Theorem for RacerDX
 - Proofs of analysis completeness (wrt. (W, L, A) -abstraction)
 - Proof of the bug detection completeness
- Discussion and comparison with existing static race analyses, e.g., Chord (Naik et al., PLDI'06)

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
- 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 design.

Thanks!