CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 8.1 Static Program Analysis

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

Alternatives to Testing

- So "program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra's Law), what can we do?
- Testing is limited to finite concrete cases;
  - Can we check unbounded symbolic cases?
- Yes! *

*Some restrictions apply:

- Can show absence, but cannot show presence;
- Sometimes cannot show either;
- How much time do you have?
- …

Outline of this lesson

1. The impractical goal of **program verification**.
2. What lies between testing and verification?
   a. Partial verification
   b. Optional type systems
      - (Should be familiar: TypeScript anyone?)
   c. Bug finders
      - (Also familiar: es-lint)

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - List challenges preventing full verification;
  - Define false/true negatives/positives;
  - Variously describe the effects of false positives and negatives;
  - Explain when to integrate static analysis into builds.

Verification Checks Code Against Specification

A Specification: What the system is supposed to do.

B Code: What the system actually does.

- Difficulties:
  - Need a **full formal** specification;
  - Even proving termination is undecidable, let alone proving adherence to a specification.

A Full Formal Specification

- … precisely defines exactly the behavior that the system should have:
  - What the outputs are in terms of the inputs;
  - What behaviors the system should have.
- Wait a minute! That's called a program.
- Yes, a full formal specification is essentially a program, perhaps expressed at a higher level.
  - … with all the complexity that entails,
  - … including bugs!

Inefficiently (w/o algorithms)

We can't avoid Human fallibility.

Specification Must Be Modular

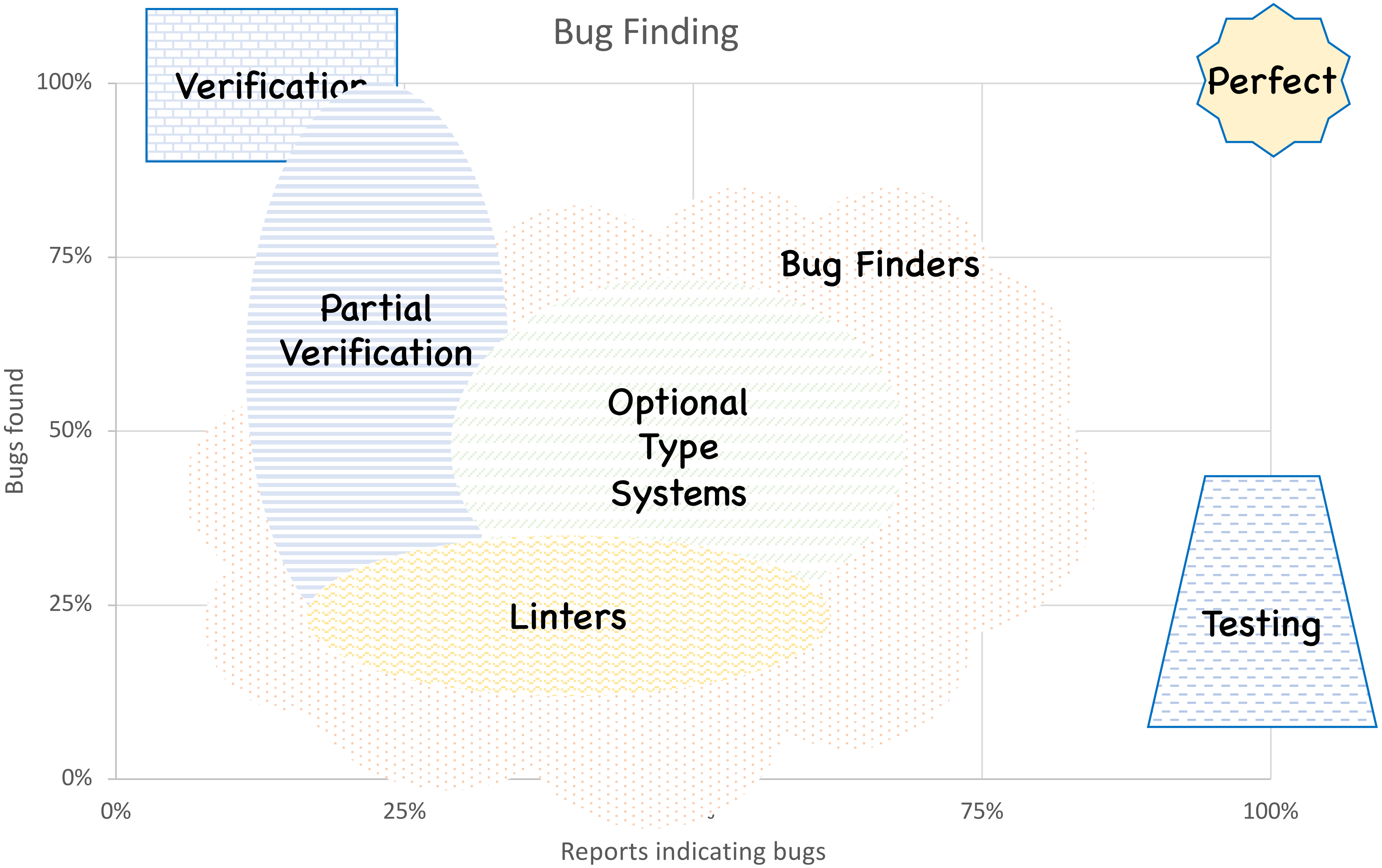
- Without modularity
  - Specification is incomprehensible
    - It is likely *inadequate* (i.e., doesn't specify what we want).
  - Specification is unprovable
    - Proof checking is usually exponential or worse
      - Must break down into usable pieces.
- Specification has to be maintained as code is:
  - Every function/class/module needs specification.
  - Even every loop needs its own specification.

Proof Must Be Modular Too

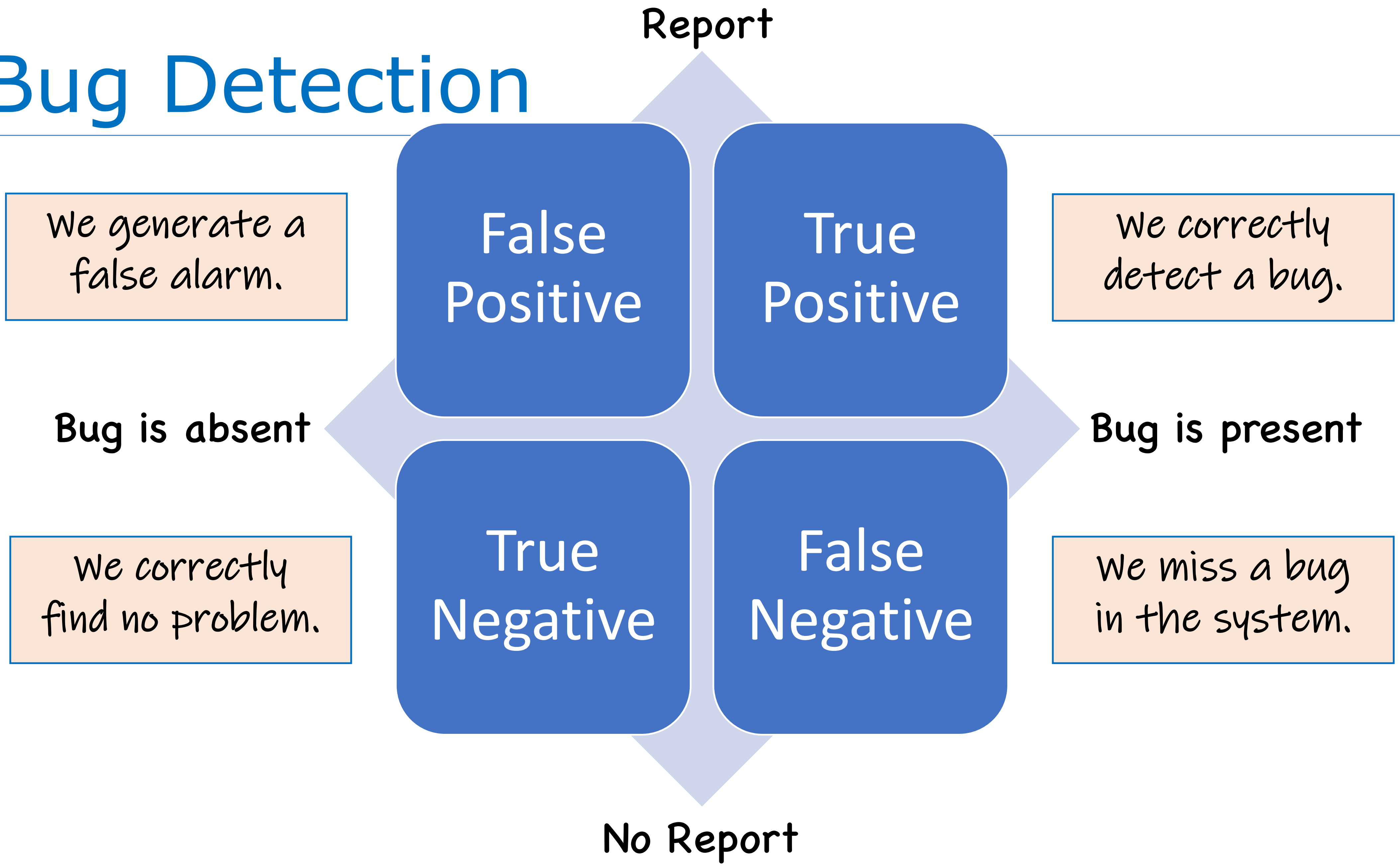
- A verification proof is usually very complex, needing lemmas written by hand.
  - Typically written and stored along with the specification.
- Engineering proofs is a highly specialized skill:
  - Hint: harder than coding → More $$$
  - Proof must be updated
    - **every time** program **or** specification changes!
- Usually too expensive unless safety critical **and** mandated by regulation.

Verification Doesn't Prove Presence of Bugs

- Verification fails if
  - Missing lemma for unit behavior, or
  - Cannot verify loop invariant, or
  - Functional specification missing a piece, or
  - Run out of time trying to construct proof, or
  - Specification is wrong.
- Constructing the proof can easily take as long as constructing the software, if not much more.
  - Just because there is no proof does not mean the software has a fault.


Bug Finding
Verification
Perfect
100%
75%
50%
25%
0%
Bugs found
Partial Verification
Bug Finders
Optional Type Systems
Linters
Testing
0%
25%
75%
100%
Reports indicating bugs

Bug Detection



Static Program Analysis

- Bug finders and partial verification use static program analysis
  - Reads in the program (just like a compiler);
  - Analyze to determine properties:
    - E.g., are all open resources eventually closed?
  - ”static” means ”without running the program”.
- All non-trivial properties are undecidable
  - Approximations are always necessary: make a choice
    - E.g., miss some closes of open resources, or
    - Miss some open resources not being closed.

Compromises with Static Analysis

- Getting precise results may take **time**:
  - Many algorithms are exponential in precision measures.
- Getting precise results may require **whole** program:
  - If parts of the program loaded at runtime:
    - Analysis results may be very imprecise, or (worse)
    - Incorrect, if they assume the whole program is available.
- Getting precise results may require intervention:
  - Code may need to be **annotated** with information:
    - E.g., this method may return an open resource.

Effects of Analysis Imprecision

FALSE NEGATIVES

- The static analysis misses something ”bad” in program:
  - Bug not found.
- Can give a false sense of security.
- Can be reduced, but at the cost of false positives!

FALSE POSITIVES

- The static analysis reports a problem that doesn’t exist:
  - There is no bug.
- Real bugs can be swamped by a flood of spurious reports.
- Programmer time is wasted chasing down false leads.

Google defined "Effective False Positive"

- A report from static analysis is effectively false,
  - If it is ignored by developers;
  - Whether or not it represents a true bug.
- Even if the report is technically correct
  - It may refer to something considered unimportant:
    - E.g., who cares if all the files aren't closed, if the program is about to exit anyway.
    - E.g., yes, there is a race condition between two logging statements, but that's not important.
- Even if the report is technically wrong
  - Developers may see potential problem, and fix.

Criteria For Automated Program Analysis

- Efficient and Easy
  - Should not require whole program or annotations.
- Rarely spurious
  - No more than 10% effectively false positive.
- Actionable
  - Should point out things easy to fix.
- Effective
  - Problems should be perceived as important.

Automatically applied during Code Review.

Source: Software Engineering at Google, [Chapter 20]

Review: Learning Objectives for this Lesson

- You should now be able to:
  - List challenges preventing full verification;
  - Define false/true negatives/positives;
  - Variously describe the effects of false positives and negatives;
  - Explain when to integrate static analysis into builds.

Next steps...

- In our next lesson, we'll talk about Code Smells and Refactoring.