# Characterizing Bugs by Example in Support of Static Analysis

Brianna Satinoff and Bryan Robbins

CMSC 631 Final Project

# The Truth about Bugs

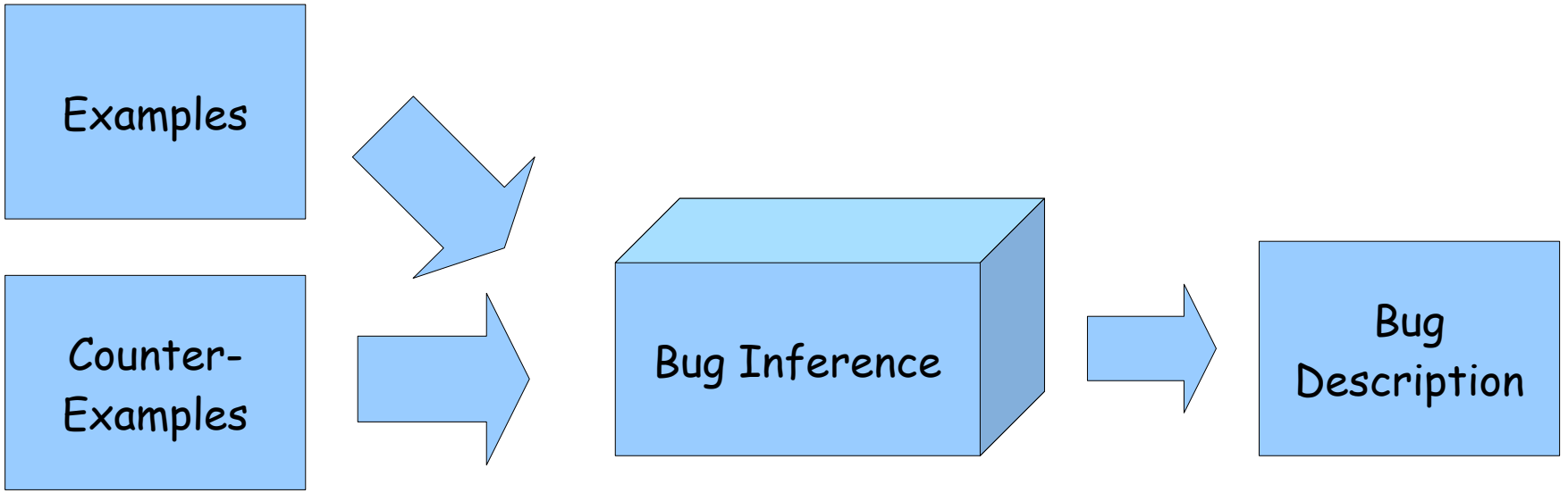
- We want to find and "kill" as many bugs as possible.
- Bugs are usually hard to **find**.
- Bugs are sometimes hard to even ***define***.
  - What about expert developers?

# Definition from Experts

- Expert differences
  - In order to work on increasingly complex problems, experts **proceduralize**
  - **Proceduralization** is the cognitive activity of combining two or more actions into one
- Result: Experts aren't going to be very good at describing the low-level internals of their bugs.

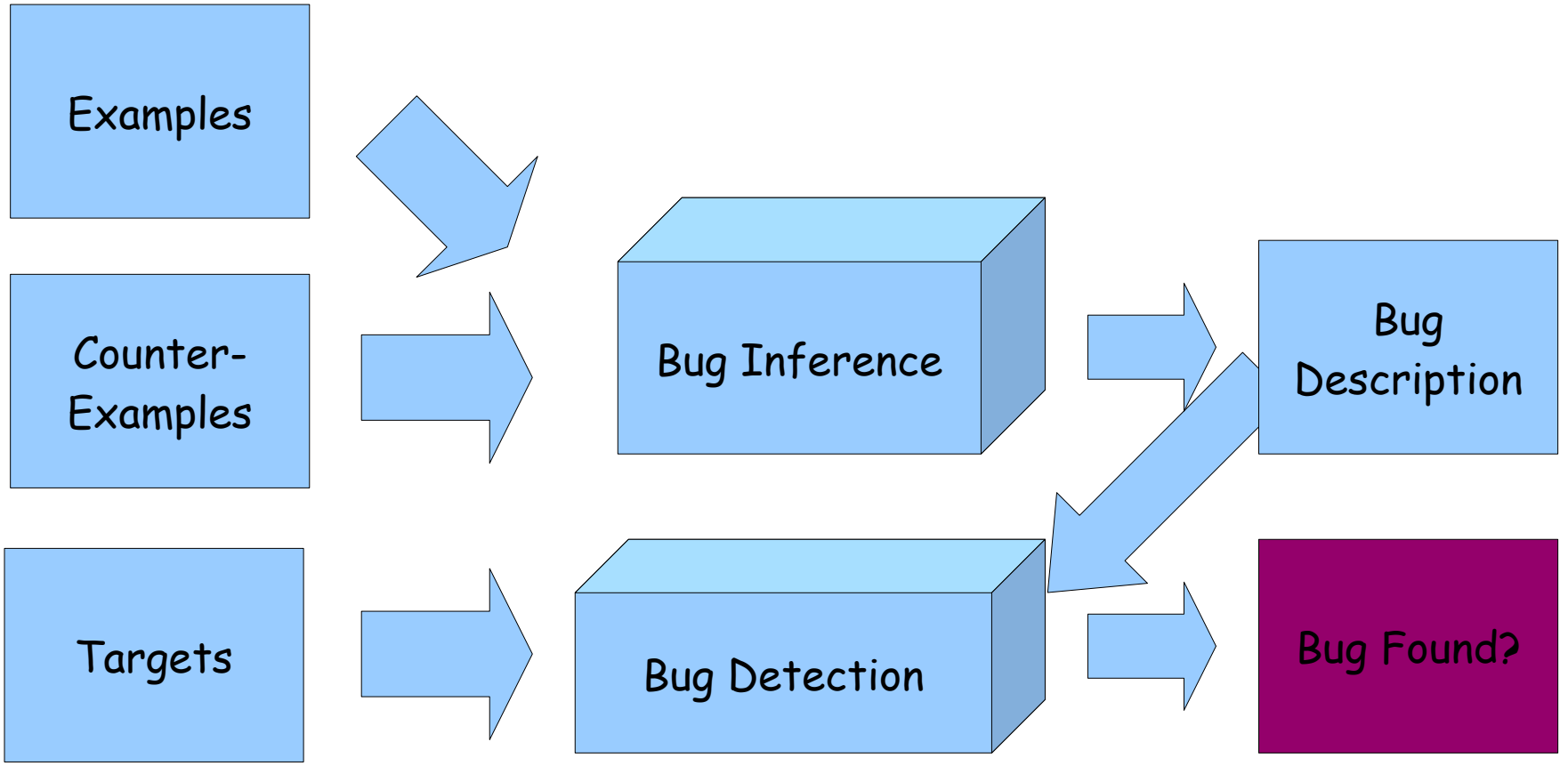
# Bugs by Example

- What if we allowed experts to "stay on their level" and define bugs with examples?



# Bugs by Example

- What if we allowed experts to "stay on their level" and define bugs with examples?



# Contributions

- Develop an expandable Bug Description Language that can support static analysis tools
- Allow for bug characterization at a higher level – by example

- Bug Descriptions come from program characteristics

# Characterizing Programs

- Characterize programs based on byte code
  - We use ASM, "a Java bytecode manipulation and analysis framework", http://asm.ow2.org/
- Describe bugs in terms of these same characteristics
  - Combine characteristics recursively and with logical operators

# Example: Java Bytecode

```
public String employeeName()
{
   return name;
}

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn
```

Example from:
http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/

# Program Characteristics

- Class characteristics
  - Abstract/final, superclass, interfaces
- Method characteristics
  - Name, return type, abstract/static/final
- Field characteristics
  - Type, static/final
- Local variable characteristics
  - Type, is argument?, is strictly local?, is either?

# Program Characteristics

- Detect the following:
  - Local Variable/Field read
  - Local Variable/Field written
  - Method called (class name + method name)

- ASM can do many more things …

# Describing Bugs

- Our approach to bug description is to use Java Objects that define constraints

- Combine constraints recursively and with Boolean operators

# Example: Unwritten Field

FieldConstraint: Access = Private

NotConstraint

MethodConstraint: Any Method

FieldAccessConstraint: Written

"There exists a private field such that there does not exist a method where that field is written"

# Example: Unwritten Field

```
String className = "test.TestClass";

FieldConstraint fc = new FieldConstraint(0);

fc.setAccess(Access.PRIVATE);

NotConstraint not = new NotConstraint();

fc.setInnerConstraint(not);

MethodConstraint mc = new MethodConstraint(1);

not.setInnerConstraint(mc);

FieldAccessTest test = new FieldAccessTest(0, 1,
ReadWrite.WRITE);

mc.setInnerConstraint(test);

Evaluator eval = new Evaluator(className, fc);

eval.evaluate(new PrintInfo(not, "There was an
unwritten field"));
```

# Algorithm: Bug Detection

- Input: Bug Description, Target file
- Output: True if given bug found, else False

- Basic idea
  - Use ASM to analyze the Target file
  - Check if each bug constraint is met

# Example: Unwritten Field

FieldConstraint: Access = Private

    NotConstraint

        MethodConstraint: Any Method

            FieldAccessConstraint: Written

For each private field

    For each method

        Check for Field Written

Bug found if this is false for all methods

# Algorithm: Bug Inference

- Input: Positive (i.e. Bug Present) and Negative Examples
- Output: Bug Description
- Overall algorithm: Top-down, brute-force
  - Try each possible bug description (with lots of pruning, of course)
  - Run the bug detection on each example
  - If it's TRUE for all positive examples and FALSE for all negative ones, keep it

# Bug Inference: Pruning

- How do we make brute-force tractable?
- Don't go down "impossible paths"
- Restrict method name and type constraints
- Limit where Boolean operators can go
  - Based on looking at FindBugs bug list

# FindBugs Examples

- Class defines equals() but not hashCode()
- The hasNext() method calls next()
- Class derives from Thread but doesn't override run()
- Method invokes inefficient new String(String) constructor
- clone() method doesn't call super.clone()

# In Progress

- Evaluation
  - Selected 40 inferable bugs from FindBugs
  - See if bugs can be inferred from contrived examples
  - Ranking possible bugs
- Better pruning
  - Could try bottom-up inference approach

# Conclusions

- We have shown it is possible to:
  - Describe bugs in a general way
  - Infer bug characteristics from code examples
- We are still investigating:
  - The robustness of bug inference
  - The scalability of bug inference