Combining Static and Dynamic Reasoning for Bug Detection

Yannis Smaragdakis and Christoph Csallner

Elnatan Reisner – April 17, 2008

Motivation

- Both testing and static analysis are useful
- But both have drawbacks
- Testing
  - Focuses on finding bugs
  - Without guidance, many tests will be useless
- Static analysis
  - Focuses on certifying correctness
  - Without guidance, many false warnings
- How can we combine these two approaches?

Outline

- Thoughts on static vs. dynamic
- Language- and user-level
- Tools: ESC/Java and JCrasher
- Making static checking ‘sound for incorrectness’
- Improving this to user-level soundness
- Experimental case study

Thoughts on static vs. dynamic

- Somewhat arbitrary distinction
- More relevant dichotomy:
  - Try to prove program correct:
    - ‘Sound’ here means no undetected errors
    - ‘If I say there’s no bug, there’s no bug.’
  - Try to find bugs
    - ‘Sound’ here means no false alarms
    - ‘If I say there’s a bug, there’s a bug.’
- Why prove programs incorrect?

Language- vs. user-level

- But what's a bug?
- Language-level: it is possible for the code to exhibit the behavior
  - Here's a 'bug' that is not language-level sound:

```
public int get0() {return 0;}

public int meth() {
  int[] a = new int[1];
  return a[get0()];
}
```

- User-level: a user might actually encounter the bug

ESC/Java

- ‘Extended Static Checker’
- Modular checking – context-insensitive
- Annotations (similar to JML) aid analysis
  - Specify preconditions, postconditions, etc.
- Uses Simplify theorem prover to ensure
  - No null dereference
  - Annotations obeyed
- Generates warnings (and *counterexample contexts*) when it finds potential bugs
  - But it is unsound in both senses

JCrasher

- Generates JUnit tests which crash program
  - Random inputs based on type information
  - Tests public methods
- Heuristically classifies exceptions as invalid test or actual bug
- Warnings are language-level sound (for incorrectness)
  - The program actually crashed!

Check ‘n‘ Crash: language-level soundness

- Idea: Combine ESC/Java and JCrasher
- Procedure
  - Run ESC/Java
  - Use warnings to find potential crashing inputs
  - Use JCrasher to create and run JUnit test cases
- Result: Focused testing + language-level soundness

DSD-Crasher: user-level soundness

- Heuristic for generating ‘normal’ input
  - Run Daikon on an existing test suite
    - Finds conditions that code exhibits in all observed executions
  - Use generated invariants as preconditions to filter ESC/Java warnings
- Limiting inputs to ‘normal’ cases eliminates user-level unsound bug reports
- **D**ynamic-**S**tatic-**D**ynamic

Experimental case study

- Compared JCrasher, Check ‘n‘ Crash, and DSD-Crasher on Groovy (a scripting language)

	Runtime [min:s]	ESC/Java warnings	Generated test cases	Reports confirmed by test cases
JCrasher	1:40	n/a	100,000	0.6
Check 'n' Crash	2:17	51	439	7.0
DSD-Crasher	10:31	47	434	4.0

- More bugs found with fewer test cases
  - But longer total running time

Conclusions

- Combining static and dynamic techniques can find more bugs with fewer tests
- Questions:
  - What is the time tradeoff in general?
  - Is user-level soundness the right goal?
    - Security exploits use abnormal inputs