Introduction: Recap

Testing, Quality Assurance, and Maintenance
Winter 2018

Prof. Arie Gurfinkel
Ultimate Goal: Static Program Analysis

Reasoning statically about behavior of a program without executing it
- compile-time analysis
- exhaustive, considers all possible executions under all possible environments and inputs

The *algorithmic* discovery of *properties* of program by *inspection* of the *source text*

Manna and Pnueli

Also known as static analysis, program verification, formal methods, etc.
Undecidability

A problem is undecidable if there does not exists a Turing machine that can solve it

• i.e., not solvable by a computer program

The halting problem

• does a program $P$ terminates on input $I$
• proved undecidable by Alan Turing in 1936

Rice’s Theorem

• for any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property
• in practice, this means that there is no machine that can always decide whether the language of a given Turing machine has a particular nontrivial property
• [https://en.wikipedia.org/wiki/Rice%27s_theorem](https://en.wikipedia.org/wiki/Rice%27s_theorem)
Living with Undecidability

“Algorithms” that occasionally diverge

Limit programs that can be analyzed
  • finite-state, loop-free

Partial (unsound) verification
  • analyze only some executions up-to a fixed number of steps

Incomplete verification / Abstraction
  • analyze a superset of program executions

Programmer Assistance
  • annotations, pre-, post-conditions, inductive invariants

testing
sym exec
automated verification
deductive verification
(User) Effort vs (Verification) Assurance

Effort

Assurance/Coverage

Testing

Symbolic Execution

Automated Verification

Deductive Verification
Key Challenges

Testing
  • Coverage

Symbolic Execution and Automated Verification
  • Scalability

Deductive Verification
  • Usability

Common Challenge
  • Specification / Oracle
Topics Covered in the Course

Foundations
• syntax, semantics, abstract syntax trees, visitors, control flow graphs

Testing
• coverage: structural, dataflow, and logic

Symbolic Execution
• using SMT solvers, constraints, path conditions, exploration strategies
• building a (toy) symbolic execution engine

Deductive Verification
• Hoare Logic, weakest pre-condition calculus, verification condition generation
• verifying algorithm using Dafny, building a small verification engine

Automated Verification
• (basics of) software model checking
A little about me

2007, PhD University of Toronto

2006-2016, Principle Researcher at Software Engineering Institute, Carnegie Mellon University

Sep 2016, Associate Professor, University of Waterloo
Interests and Tools

Interests

• Software Model Checking, Program Verification, Decision Procedures, Abstract Interpretation, SMT, Horn Clauses, …

Active Tools

• SeaHorn – Algorithmic Logic-Based Verification framework for C
• AVY – Hardware Model Checker with Interpolating PDR
• SPACER – Horn Clause Solver based on Z3 GPDR
• for more, see http://arieg.bitbucket.org/tools.html

Current Work

• parametric symbolic reachability – verifying safety properties of parametric systems
• automated verification of C
• …
Fault, Error, and Failure

Testing, Quality Assurance, and Maintenance
Winter 2018

Prof. Arie Gurfinkel

based on slides by Prof. Lin Tan and others
Terminology, IEEE 610.12-1990

**Fault** -- often referred to as **Bug** [Avizienis’00]
- A static defect in software (incorrect lines of code)

**Error**
- An incorrect internal state (unobserved)

**Failure**
- External, incorrect behaviour with respect to the expected behaviour (observed)

Not used consistently in literature!
What is this?

A fault?

An error?

A failure?

We need to describe specified and desired behaviour first!
Erroneous State ("Error")
Design Fault
Mechanical Fault
Example: Fault, Error, Failure

```java
public static int numZero (int[] x) {
    //Effects: if x==null throw NullPointerException
    //         else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i]==0) {
            count++;
        }
    }
    return count;
}
```

State of the program:  x, i, count, PC

Fix: for(int i=0; i<x.length; i++)

x = [2,7,0], fault executed, error, no failure
x = [0,7,2], fault executed, error, failure

State of the program:  x, i, count, PC
Exercise: The Program

/* Effect: if x==null throw NullPointerException. Otherwise, return the index of the last element in the array ‘x’ that equals integer ’y’. Return -1 if no such element exists. */

public int findLast (int[] x, int y) {
    for (int i=x.length-1; i>0; i--) {
        if (x[i] == y) { return i; }
    }
    return -1;
}

/* test 1: x=[2,3,5], y=2;
   expect: findLast(x,y) == 0
   test 2: x=[2,3,5,2], y=2;
   expect: findLast(x,y) == 3 */
Exercise: The Problem

Read this faulty program, which includes a test case that results in failure. Answer the following questions.

• (a) Identify the fault, and fix the fault.
• (b) If possible, identify a test case that does not execute the fault.
• (c) If possible, identify a test case that executes the fault, but does not result in an error state.
• (d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter.
• (e) For the given test case ‘test1’, identify the first error state. Be sure to describe the complete state.
States

State 0:
- \( x = [2,3,5] \)
- \( y = 2 \)
- \( i = \text{undefined} \)
- \( \text{PC} = \text{findLast(...)} \)

State 1:
- \( x = [2,3,5] \)
- \( y = 2 \)
- \( i = \text{undefined} \)
- \( \text{PC} = \text{before } i = x.\text{length}-1; \)

State 2:
- \( x = [2,3,5] \)
- \( y = 2 \)
- \( i = 2 \)
- \( \text{PC} = \text{after } i = x.\text{length}-1; \)

State 3:
- \( x = [2,3,5] \)
- \( y = 2 \)
- \( i = 2 \)
- \( \text{PC} = i > 0; \)
States

- **State 3:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 2$
  - PC = $i > 0$

- **State 4:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 2$
  - PC = if $(x[i] == y)$

- **State 5:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 1$
  - PC = $i--$

- **State 6:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 1$
  - PC = $i > 0$

- **State 7:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 1$
  - PC = if $(x[i] == y)$

- **State 8:**
  - $x = [2, 3, 5]$
  - $y = 2$
  - $i = 0$
  - PC = $i--$
States

Incorrect Program

• State 8:
  • $x = [2,3,5]$
  • $y = 2$
  • $i = 0$
  • $PC = i--;$

• State 9:
  • $x = [2,3,5]$
  • $y = 2$
  • $i = 0$
  • $PC = i>0;$

• State 10:
  • $x = [2,3,5]$
  • $y = 2$
  • $i = 0$ (undefined)
  • $PC = return -1;$

Correct Program

• State 10:
  • $x = [2,3,5]$
  • $y = 2$
  • $i = 0$
  • $PC = if (x[i]==y);$
Exercise: Solutions (1/2)

(a) The for-loop should include the 0 index:
• for (int i=x.length-1; i >= 0; i--)

(b) The null value for x will result in a NullPointerException before the loop test is evaluated, hence no execution of the fault.
• Input: x = null; y = 3
• Expected Output: NullPointerException
• Actual Output: NullPointerException

(c) For any input where y appears in a position that is not position 0, there is no error. Also, if x is empty, there is no error.
• Input: x = [2, 3, 5]; y = 3;
• Expected Output: 1
• Actual Output: 1
Exercise: Solutions (2/2)

(d) For an input where y is not in x, the missing path (i.e. an incorrect PC on the final loop that is not taken, normally i = 2, 1, 0, but this one has only i = 2, 1, ) is an error, but there is no failure.

• Input: x = [2, 3, 5]; y = 7;
• Expected Output: -1
• Actual Output: -1

(e) Note that the key aspect of the error state is that the PC is outside the loop (following the false evaluation of the 0>0 test. In a correct program, the PC should be at the if-test, with index i==0.

• Input: x = [2, 3, 5]; y = 2;
• Expected Output: 0
• Actual Output: -1
• First Error State:
  – x = [2, 3, 5]
  – y = 2;
  – i = 0 (or undefined);
  – PC = return -1;
RIP Model

Three conditions must be present for an error to be observed (i.e., failure to happen):

• **Reachability**: the location or locations in the program that contain the fault must be reached.

• **Infection**: After executing the location, the state of the program must be incorrect.

• **Propagation**: The infected state must propagate to cause some output of the program to be incorrect.
HOW DO WE DEAL WITH FAULTS, ERRORS, AND FAILURES?
Addressing Faults at Different Stages

- **Fault Avoidance**
  - Better Design,
  - Better PL, ...

- **Fault Detection**
  - Testing,
  - Debugging, ...

- **Fault Tolerance**
  - Redundancy,
  - Isolation, ...
Declaring the Bug as a Feature
Modular Redundancy: Fault Tolerance
Patching: Fixing the Fault
Testing: Fault Detection
Testing vs. Debugging

**Testing**: Evaluating software by observing its execution

**Debugging**: The process of finding a fault given a failure

**Testing is hard:**
- Often, only specific inputs will trigger the fault into creating a failure.

**Debugging is hard:**
- Given a failure, it is often difficult to know the fault.
Testing is hard

if ( x - 100 <= 0 )
  if ( y - 100 <= 0 )
    if ( x + y - 200 == 0 )
      crash();

Only input $x=100$ & $y=100$ triggers the crash
If $x$ and $y$ are 32-bit integers, what is the probability of a crash?

- $1 / 2^{64}$