UNIVERSITÄT BERN

Introduction to Software Engineering

7. Verification

Mircea F. Lungu

Based on a lecture by Oscar Nierstrasz.

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Software Reliability

The **reliability** of a software system is a measure of how well it provides the services expected by its users, expressed in terms of software failures.

Failures vs. Faults

A **software failure** is an ***execution event*** where the software behaves in an unexpected or undesirable way.

A **software fault** is an ***erroneous portion of software*** which may cause failures to occur if it is run in a particular state, or with particular inputs.

Therac-25

Intel Pentium

NASA Mars Mission

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Common Sources of Software Faults ...

Concurrency is dangerous because *timing differences* can affect overall program behaviour in *hard-to-predict* ways.

—Minimize inter-process dependencies

Recursion can lead to *convoluted logic*, and may exhaust (stack) memory.

—Use recursion in a disciplined way, within a controlled scope

Floating point numbers are *inherently imprecise* and may lead to invalid comparisons.

—Fixed point numbers are safer for exact comparisons

What to do with Faults?

Fault tolerance

developing programs that will *operate despite the presence of faults*

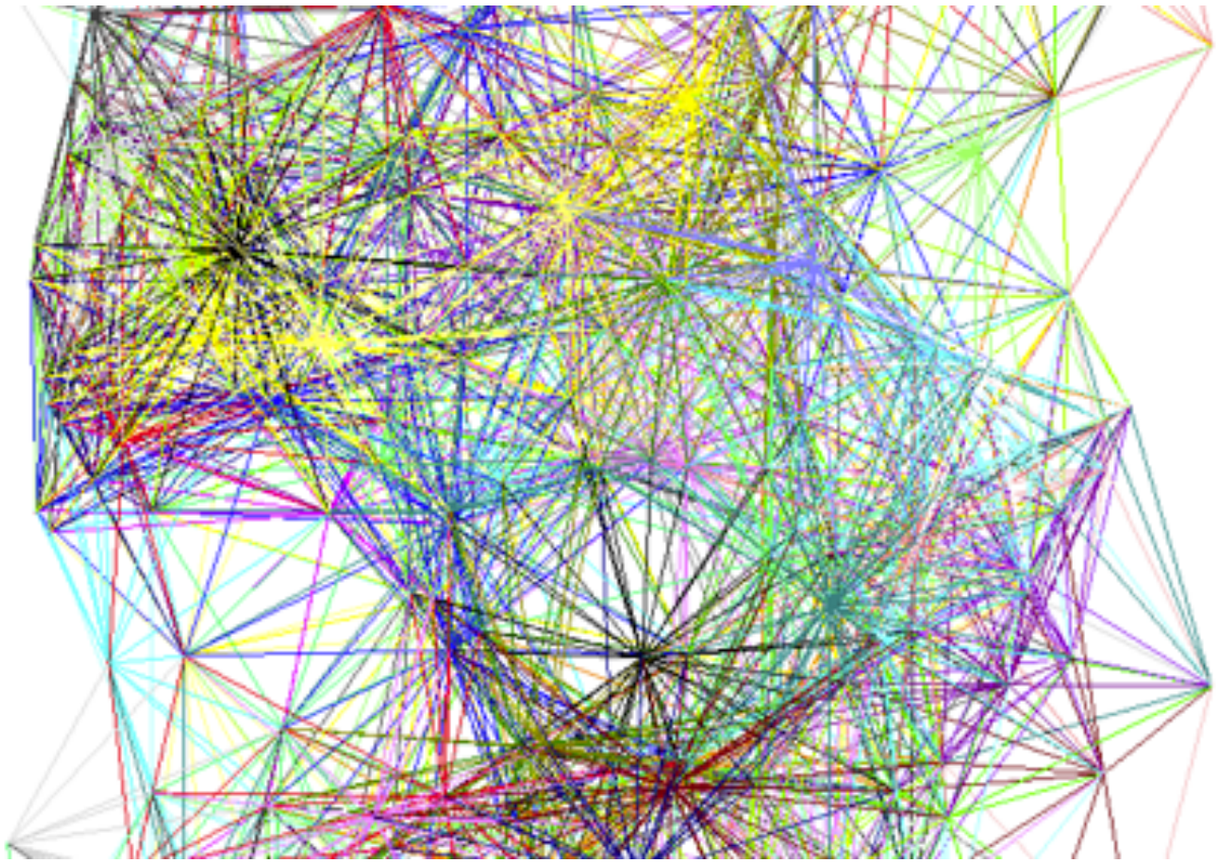
Fault avoidance

development techniques to *reduce the number of faults* in a system

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Approaches to Hardware Fault Tolerance



Distributed Systems
The Internet

Approaches to Software Fault Tolerance

N-version Programming. *Multiple versions* of the software system are implemented *independently by different teams*.

- runs all the versions in *parallel*
- *compares* their results using a voting system, and
- *rejects* inconsistent outputs

(n > 3 && n % 2 !=0)

Defensive Programming

Use the *type system* to ensure that variables do not get assigned invalid values.

Use *assertions* to detect failures and raise exceptions. Explicitly state and check all invariants for abstract data types, and pre- and post-conditions of procedures as assertions.

Use exception handlers to recover from failures.

Example: Fault Recovery

- Change logs (rollback and replay)
  - Smalltalk image and changes

- Transactional Memory (software and hardware)
  - ACID (Atomicy, Consistency, Isolation, Durability)

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- ➡ Avoidance via Verification
  - Testing
    - —White box
    - —Black box
    - —Regression

Fault Avoidance Can Benefit From ...

1. *Formal proofs* that certain properties hold
2. *Code reviews* during the development process
3. *System testing* to expose faults and assess reliability

Fault Avoidance Can Benefit From ...

4. *A more zen approach* to writing software

Formal Verification

Mathematically-based. Use mathematical reasoning to demonstrate that program meets specification

—e.g., that invariants are not violated

—e.g., model-checking tools

Static Verification

Program Inspections:

> Small team systematically checks program code

> Inspection checklist often drives this activity

—e.g., “Are all invariants, pre- and post-conditions checked?” ...

Static Program Analyzers:

> Complements compiler to check for common errors

—e.g., variable use before initialization

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Testing vs. Correctness

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

— Edsger Dijkstra, The Humble Programmer, ACM Turing lecture, 1972

Tests are designed to *reveal the presence of defects* in the system.

Testing In Practice

Testing **in practice** can only be representative.

Test data are inputs devised to test the system.

Test cases are input/output specifications for a particular function being tested.

The Testing Process

1. Unit testing:
   - Individual (stand-alone) *components* are tested to ensure that they operate correctly.
2. Module testing:
   - A collection of *related components* (a module) is tested as a group.
3. Sub-system testing:
   - The phase tests a *set of modules* integrated as a sub-system. Since the most common problems in large systems arise from sub-system interface mismatches, this phase focuses on testing these interfaces.

The Testing Process ...

4. System testing:
   — This phase concentrates on (i) detecting errors resulting from unexpected interactions between sub-systems, and (ii) validating that the complete systems fulfils functional and non-functional requirements.
5. Acceptance testing (alpha/beta testing):
   — The system is tested with *real* rather than simulated data.

Bottom-up Testing

- *Start by testing units* and modules
- *Test drivers* must be written to exercise lower-level components
- Works well for *reusable components* to be shared with other projects

Bottom-up testing will not uncover *architectural faults*

Top-down Testing

- *Start with sub-systems*, where modules are represented by "stubs" / mocks
- Similarly test modules, representing functions as stubs
- *Coding and testing* are carried out as a *single activity*
- Design errors can be detected early on, avoiding expensive redesign
- Always have a running (if limited) system!

BUT: may be impractical for stubs to simulate complex components

Roadmap

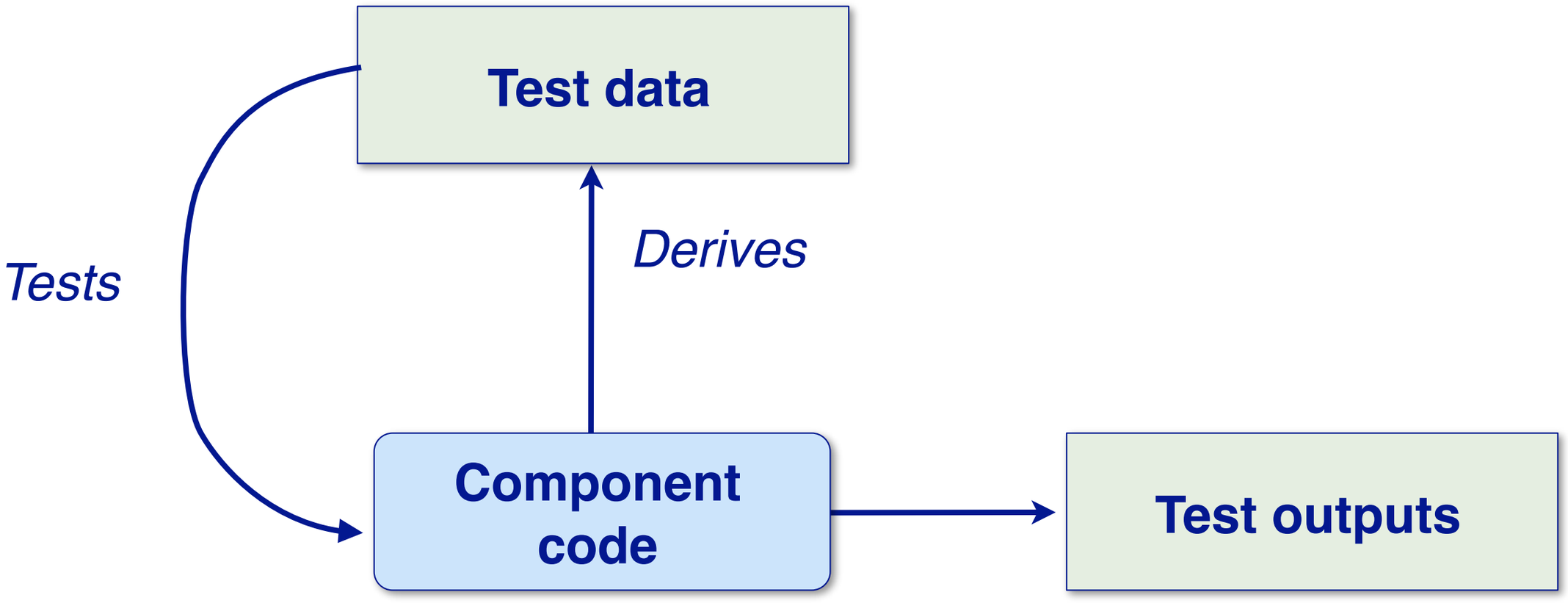
- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

```
class BinSearch {
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// the key is -1 if the element is not found
    public static void search (int key, int [] elemArray, Result r)
    {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;
        r.found = false; r.index = -1;                                  (1)
        while ( bottom <= top)                                          (2)
        {
            mid = (top + bottom) / 2;
            if (elemArray [mid] == key)                                 (3)
            {
                r.index = mid;                                          (8)
                r.found = true;
                return ;                                                -> (9)
            } // if part
            else
            {
                if (elemArray [mid] < key)                              (4)
                        bottom = mid + 1;                               (5)
                else
                        top = mid -i;                                   (6)
            }                                                           (7)
        } //while loop
    } //search                                                          (9)
} //BinSearch
```

Structural (white box) Testing

Structural testing treats a component as a *"white box"* or "glass box" whose *structure can be examined to generate test cases.*

Derive test cases to *maximize coverage* of that structure, yet *minimize the number of test cases*.



Program flow graphs

- Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node

- The number of tests to test all control statements equals the *cyclomatic complexity*

Cyclomatic complexity = Number of edges - Number of nodes +2

Path Testing

Test cases should be chosen to cover all *independent paths* through a routine:

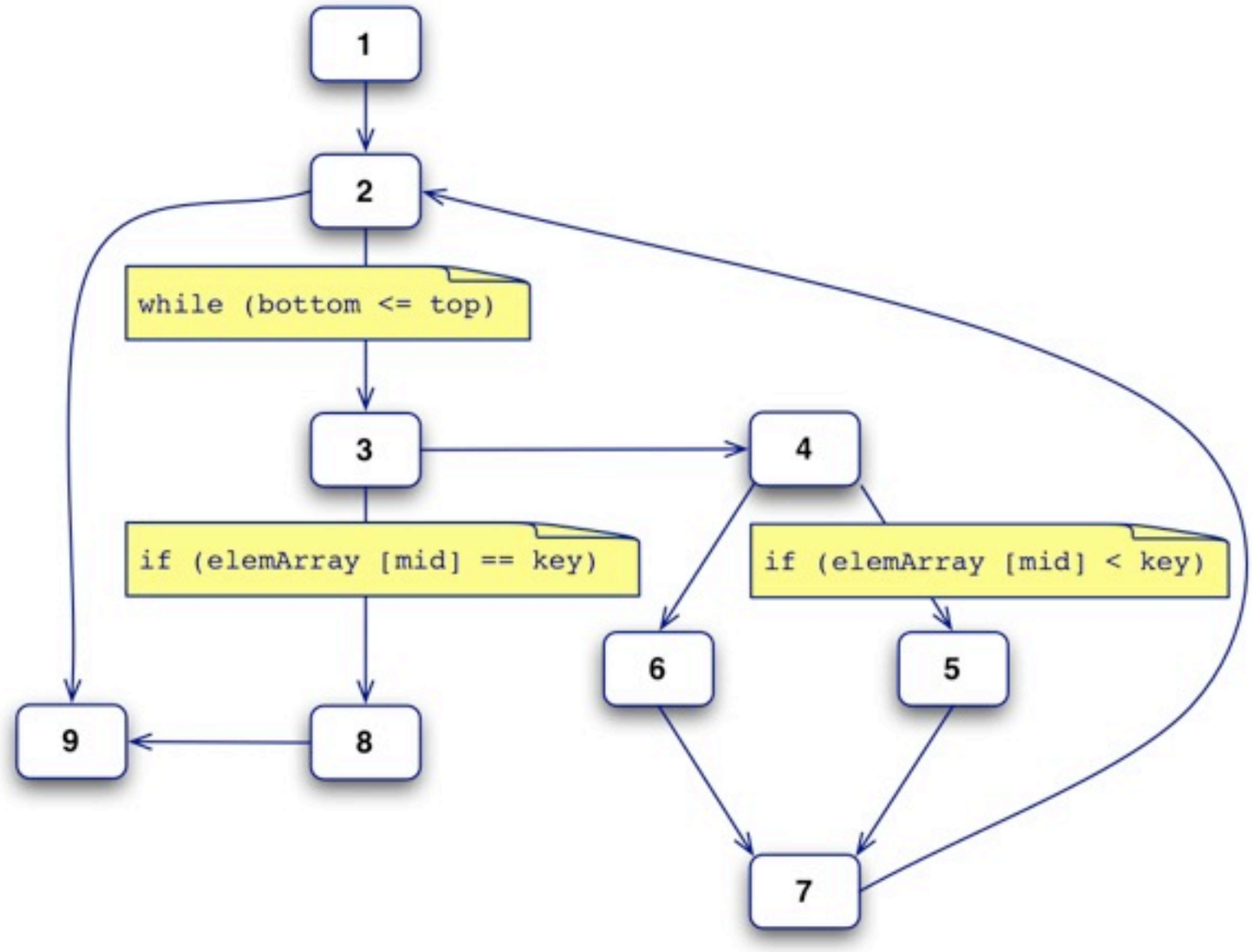
—1, 2, 9

—1, 2, 3, 8, 9

—1, 2, 3, 4, 5, 7, 2, 9

—1, 2, 3, 4, 6, 7, 2, 9

(Each path traverses at least one new edge)



Coverage criteria

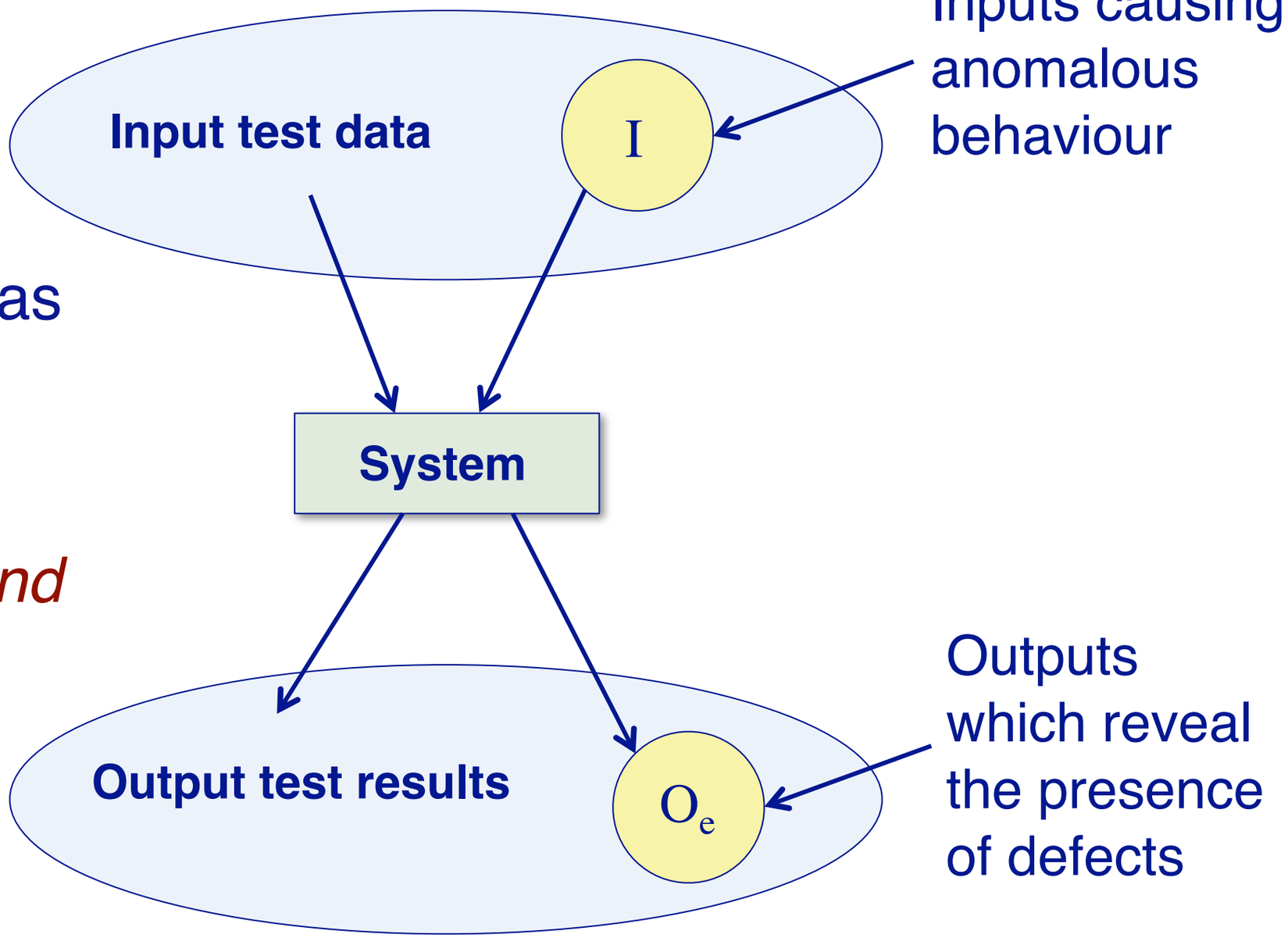
- *every statement* at least once
- *all portions of control flow* at least once
- *all possible values of compound conditions* at least once
- *all portions of data flow* at least once
- for *all loops* L, with n allowable passes:
  - I. skip the loop;
  - II.1 pass through the loop
  - III.2 passes
  - IV.m passes where 2 < m < n
  - V.n-1, n, n+1 passes

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Functional (black box) testing

Functional testing treats a component as a *"black box"* whose behaviour can be determined only by studying its *inputs and outputs*.



Equivalence partitioning

```
public static void search(int key, int [] elemArray, Result r)
    { … }
```

Check input partitions:

- Do the inputs fulfil the *pre-conditions*?
  - is the array sorted, non-empty ...
- Is the key in the array?
  - leads to (at least) 2x2 equivalence classes

Check boundary conditions:

- Is the array of length 1?
- Is the key at the start or end of the array?
  - leads to further subdivisions (not all combinations make sense)

Test Cases and Test Data

Generate test data that cover all *meaningful* equivalence partitions.

Test Cases	***Test Data***
Array length 0	key = 17, elements = { }
Array not sorted	key = 17, elements = { 33, 20, 17, 18 }
Array size 1, key in array	key = 17, elements = { 17 }
Array size 1, key not in array	key = 0, elements = { 17 }
Array size > 1, key is first element	key = 17, elements = { 17, 18, 20, 33 }
Array size > 1, key is last element	key = 33, elements = { 17, 18, 20, 33 }
Array size > 1, key is in middle	key = 20, elements = { 17, 18, 20, 33 }
Array size > 1, key not in array	key = 50, elements = { 17, 18, 20, 33 }
...	

Coverage Criteria

Test cases are derived from the *external specification* of the component and should cover:

- **all exceptions**
- **all data ranges** (incl. invalid) generating different classes of output
- **all boundary values**

Test cases can be derived from a component's *interface*, by assuming that the component will behave similarly for all members of an *equivalence partition* ...

Roadmap

- Why Verification Matters
- Sources of Faults
- Tolerance
- Avoidance via Verification
- Testing
  - —White box
  - —Black box
  - —Regression

Regression testing

"Testing *old capabilities* is more important than testing new capabilities."

Regression testing means testing that everything that used to work *still works* after changes are made to the system!

What you should know

- Failures, Faults, and Reliability
- Test cases vs. test data
- How to do black-box testing
- How to do white-box testing
- What is regression testing and why it matters
- How to design an OO system that is testable

Can you answer the following questions?

- When would you combine black-box testing with white-box testing?
- Is it acceptable to deliver a system that is not 100% reliable?
- What is the goal of path testing?
- Why is regression testing important?
- How can we increase the chance that a system avoids failures?

Further Reading / Watching

The Clean Code Talks -- Unit Testing (Google Tech Talk)

Software Engineering, Chapter 7, I. Sommerville, 9th Edn., 2011.

creative commons

COMMONS DEED

Attribution-ShareAlike 3.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/