

Foundations of Software Testing

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
<http://www.utdallas.edu/~ewong>

Speaker Biographical Sketch

- Professor & Director of International Outreach
Department of Computer Science
University of Texas at Dallas
- Guest Researcher
Computer Security Division
National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
 - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference
(*IEEE International Conference on Software Security and Reliability*)
(<http://paris.utdallas.edu/sere13>)

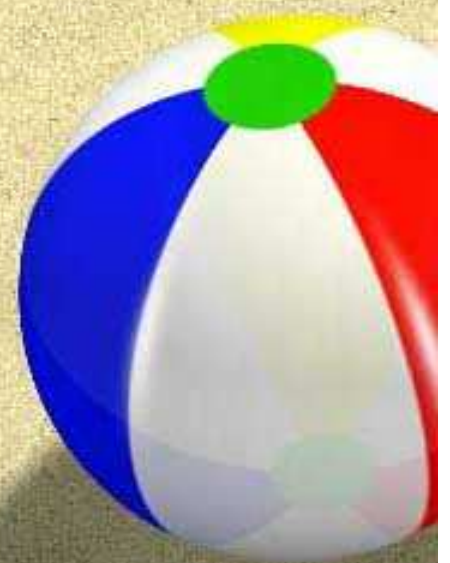




Learning Objectives

- Errors, testing, debugging, test process, CFG, correctness, reliability, oracles.
- Testing techniques

Errors, Faults, Failures





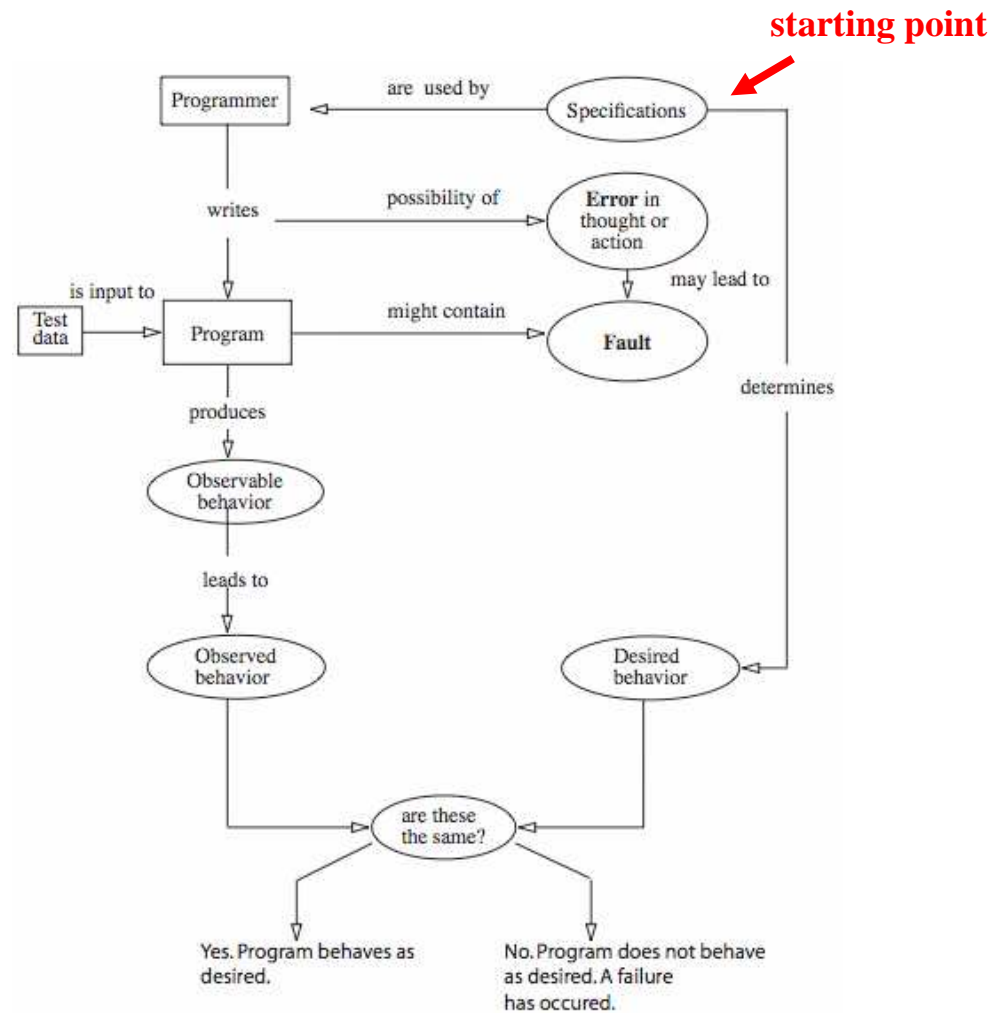
Error

- Errors are a part of our daily life.
- *Humans make errors* in their thoughts, actions, and in the products that might result from their actions.
- Errors occur wherever humans are involved in taking actions and making decisions.
- *These fundamental facts of human existence make testing an essential activity.*

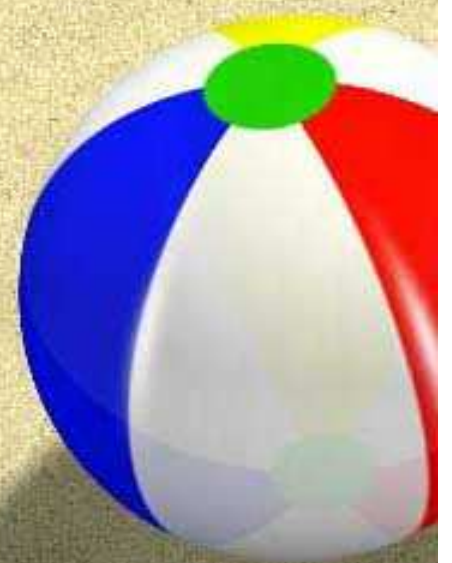
Errors: Examples

Area	Error
Hearing	Spoken: He has a garage for repairing <i>foreign</i> cars. Heard: He has a garage for repairing <i>falling</i> cars.
Medicine	Incorrect antibiotic prescribed.
Music performance	Incorrect note played.
Numerical analysis	Incorrect algorithm for matrix inversion.
Observation	Operator fails to recognize that a relief valve is stuck open.
Software	Operator used: \neq , correct operator: $>$. Identifier used: <code>new_line</code> , correct identifier: <code>next_line</code> . Expression used: $a \wedge (b \vee c)$, correct expression: $(a \wedge b) \vee c$. Data conversion from 64-bit floating point to 16-bit integer not protected (resulting in a software exception).
Speech	Spoken: <i>waple walnut</i> , intent: <i>maple walnut</i> . Spoken: <i>We need a new refrigerator</i> , intent: <i>We need a new washing machine</i> .
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of <i>pans</i> did you use? Intent: What kind of <i>pants</i> did you use?

Error, Faults, Failures



Software Quality



Software Quality (1)

- **Static quality attributes:** structured, maintainable, testable *code* as well as the availability of correct and complete *documentation*
- **Dynamic quality attributes:** software reliability, correctness, completeness, consistency, usability, and performance

Software Quality (2)

- **Reliability:** To be discussed
- **Correctness:** A program is considered correct if it behaves as desired on *all possible test inputs*.
 - valid input
 - invalid inputs
- Questions
 - Can we prove the correctness of a program by testing?
 - Correctness versus Reliability:

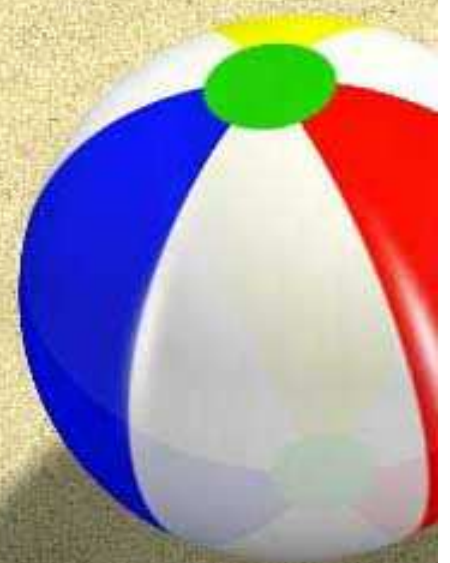
Software Quality (3)

- **Completeness** refers to the availability of all features listed in the requirements, or in the user manual. *An incomplete software is one that does not fully implement all features required.*
- **Consistency** refers to *adherence to a common set of conventions and assumptions*. For example, all buttons in the user interface might follow a common color coding convention.

Software Quality (4)

- **Usability** refers to *the ease with which an application can be used*. This is an area in itself and there exist techniques for usability testing. *Psychology* plays an important role in the design of techniques for usability testing.
- **Performance** refers to the time the application takes to perform a requested task. It is considered as a *non-functional requirement*. It is specified in terms such as “This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory.”

*Requirements, Input Domain, Behavior,
Correctness, Reliability*



Requirements, Behavior, Correctness

- Requirements leading to two different programs:
- **Requirement 1:** It is required to write a program that inputs two integers and outputs the **maximum** of these.
- **Requirement 2:** It is required to write a program that inputs a sequence of integers and outputs the **sorted version** of this sequence.

Requirements: Incompleteness

- Suppose that the program **max** is developed to satisfy Requirement 1. The expected output of **max** when the input integers are 13 and 19 can be easily determined to be 19.
- Suppose now that the tester wants to know if the two integers are to be input to the program *on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number*. The requirement as stated above *fails to provide an answer to this question*.

Requirements: Ambiguity

- Requirement 2 is ambiguous. It is not clear whether the input sequence is sorted in *ascending or descending order*. The behavior of the **sort** program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing **sort**.

Input Domain (1)

- The set of *all possible inputs* to a program P is known as the *input domain or input space*, of P .
- Using Requirement 1 above we find the input domain of **max** to be *the set of all pairs of integers* where each element in the pair integers is in the range -32,768 through 32,767.

Input Domain (2)

- **Modified Requirement 2:**

- It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted **in either ascending or descending order**. The order of the output sequence is determined by an input request character which should be “A” when an ascending sequence is desired, and “D” otherwise.
- While providing input to the program, *the request character is input first followed by the sequence* of integers to be sorted; the sequence is *terminated with a period*.

Input Domain (3)

- Based on the above modified requirement, the input domain for **sort** is **a set of pairs**. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period.

Valid/Invalid Inputs

- The modified requirement for *sort* mentions that the request characters can be “A” and “D,” but fails to answer the question *“What if the user types a different character ?”*
- When using **sort** it is certainly possible for the user to type a character other than “A” and “D.” *Any character other than “A” and “D” is considered as invalid input to sort.* The requirement for **sort** *does not specify* what action it should take when an invalid input is encountered.

Correctness vs. Reliability

- Though correctness of a program is desirable, it is almost never the **objective of testing**.
- To establish correctness via testing would *imply testing a program on all elements in the input domain*. In most cases that are encountered in practice, this is **impossible** to accomplish.
- *Thus correctness is established via mathematical proofs of programs.*

Correctness and Testing

- Correctness attempts to establish that the program is *error free*.
- *Testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.*

Software Reliability: Two Definitions

- **Software reliability** [ANSI/IEEE Std 729-1983]: is the *probability of failure free operation* of software over a given time interval and under *given conditions*.
- **Software reliability** is the probability of failure free operation of software in *its intended environment*.

Operational Profile (1)

- An *operational profile* is a numerical description of how a program is used.
- Consider a sort program which, on any given execution, allows any one of two types of input sequences (numbers only or alphanumeric strings). Sample operational profiles for sort follow.

Operational Profile (2)

- Operational profile #1

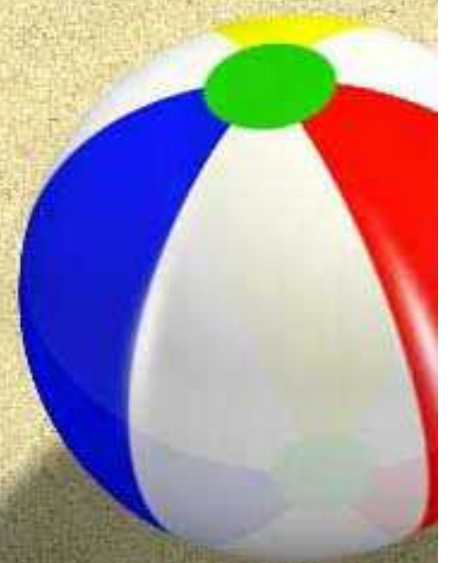
Sequence	Probability
Numbers only	0.9
Alphanumeric strings	0.1

Operational Profile (3)

- Operational profile #2

Sequence	Probability
Numbers only	0.1
Alphanumeric strings	0.9

Testing, Debugging, Verification

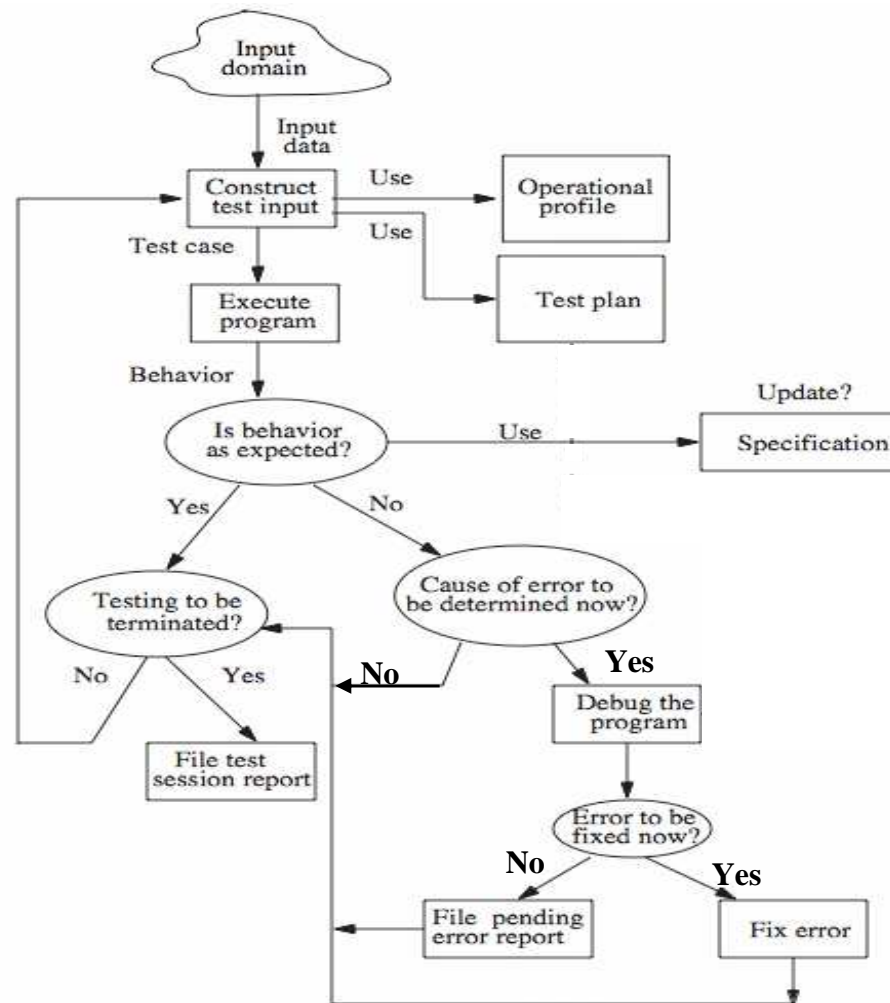




Testing and Debugging

- When testing reveals a bug, the process used to determine the cause of this bug and to remove it, is known as **debugging**.

A Test/Debugging Cycle



Test Plan (1)

- A test cycle is often guided by a **test plan**.
- Example: The **sort** program is to be tested to meet the requirements given earlier. Specifically, the following needs to be done.
 - Execute **sort** on at least two input sequences, one with “A” and the other with “D” as the request character.

Test Plan (2)

- Execute the program on an *empty input* sequence. Special case!
- Test the program for *robustness against erroneous inputs* such as “R” typed in as the request character.
- *All failures* of the test program *should be recorded* in a suitable file using the Company Failure Report Form.

Test Case/Data

- A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.
- A **test set** is a collection of zero or more test cases.
- Sample test case for **sort**:
 - Test data: <“A” 12, -29, 32 >
 - Expected output: - 29, 12, 32



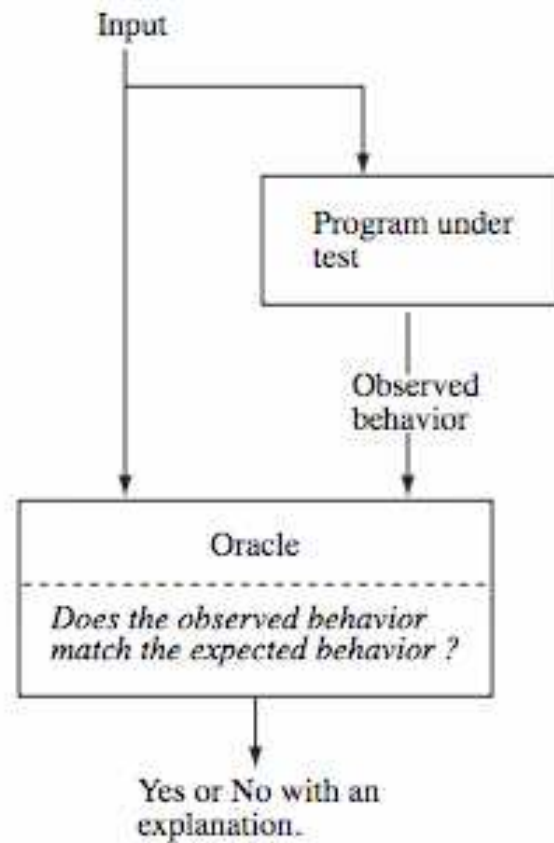
Program Behavior

- Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.
- A **state diagram** specifies program states and how the program changes its state on an input sequence.

Behavior: Observation and Analysis

- In the first step one **observes** the behavior.
- In the second step one **analyzes** the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.
- The entity that performs the task of checking the correctness of the observed behavior is known as an *oracle*.

Oracle: Example



Oracle: Programs

- Oracles can also *be programs designed to check the behavior of other programs.*
- For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix.

Oracle: Construction

- Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, *requires the determination of input-output relationship.*
- In general, the construction of automated oracles is a complex undertaking.

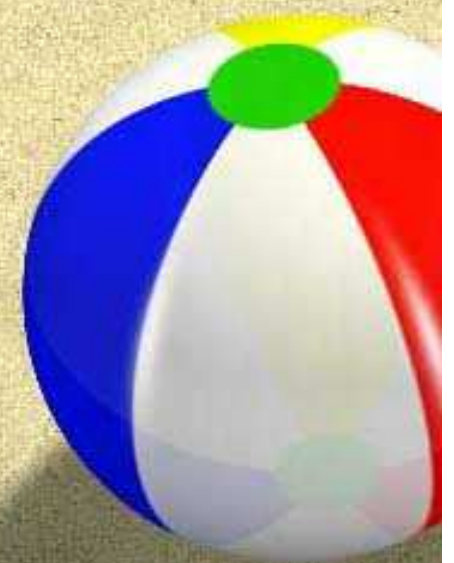
Testing and Verification (1)

- **Program verification** aims at proving the correctness of programs by showing that it contains no errors. This is very different from **testing** that aims at uncovering errors in a program.
- Program verification and testing are best considered *as complementary techniques*. In practice, one can shed program verification, but not testing.

Testing and Verification (2)

- Testing is *not a perfect technique* in that a program might contain errors despite the success of a set of tests.
- Verification might appear to be perfect technique as it promises to verify that a program is free from errors. However, *the person who verified a program might have made mistakes in the verification process*; there might be an *incorrect assumption* on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on.
- *Verified and published programs have been shown to be incorrect.*

*Program Representation:
Control Flow Graphs*



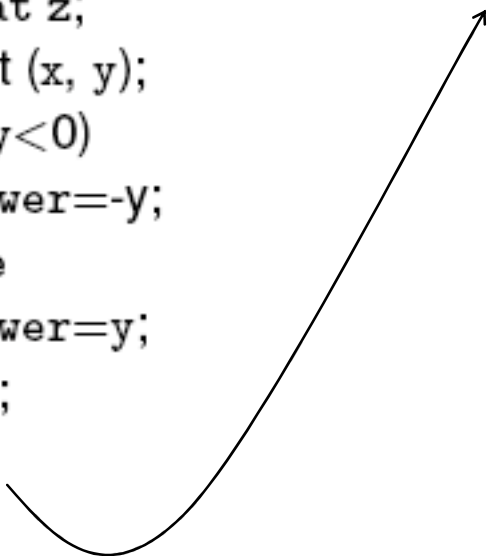
Program Representation: Basic Blocks

- A **basic block** in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus *a block has unique entry and exit points.*
- Control always enters a basic block at its entry point and exits from its exit point. There is *no possibility of exit or a halt at any point inside the basic block except at its exit point.* The entry and exit points of a basic block coincide when the block contains only one statement.

Basic Blocks: Example (1)

- Example: Computing x raised to y

```
1  begin
2    int x, y, power;
3    float z;
4    input (x, y);
5    if (y<0)
6      power=-y;
7    else
8      power=y;
9    z=1;
10   while (power!=0){
11     z=z*x;
12     power=power-1;
13   }
14   if (y<0)
15     z=1/z;
16   output(z);
17 end
```



Basic Blocks: Example (2)

- Basic blocks

Block	Lines	Entry point	Exit point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Control Flow Graph (CFG) (1)

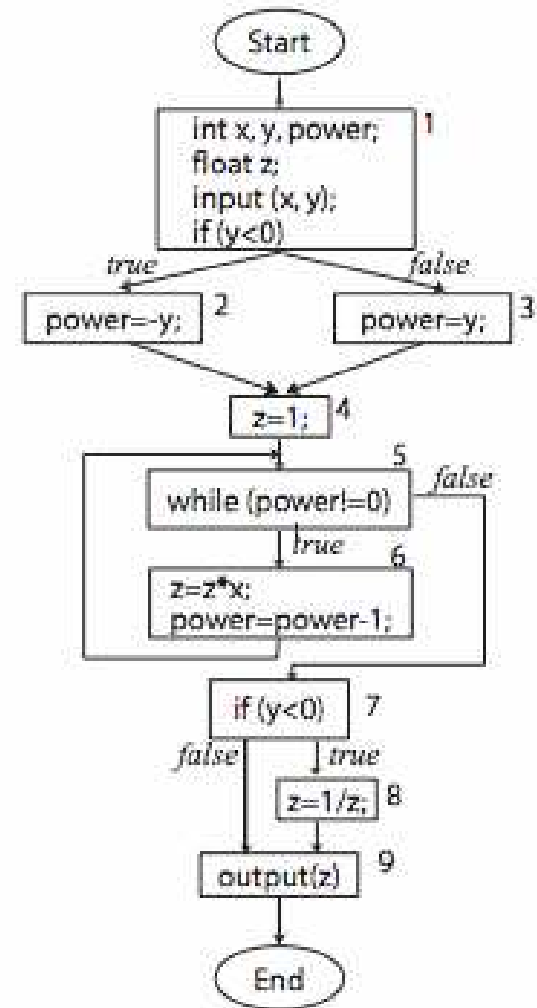
- A **control flow graph** (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N . We often write $G = (N, E)$ to denote a flow graph G with nodes given by N and edges by E .

Control Flow Graph (CFG) (2)

- In a flow graph of a program, *each basic block becomes a node* and *edges are used to indicate the flow of control between blocks*.
- Blocks and nodes are labeled such that block b_i corresponds to node n_i . An edge (i, j) connecting basic blocks b_i and b_j implies that *control can go from block b_i to block b_j* .
- We also assume that there is a node labeled **Start** in N that has no incoming edge, and another node labeled **End**, also in N , that has no outgoing edge.

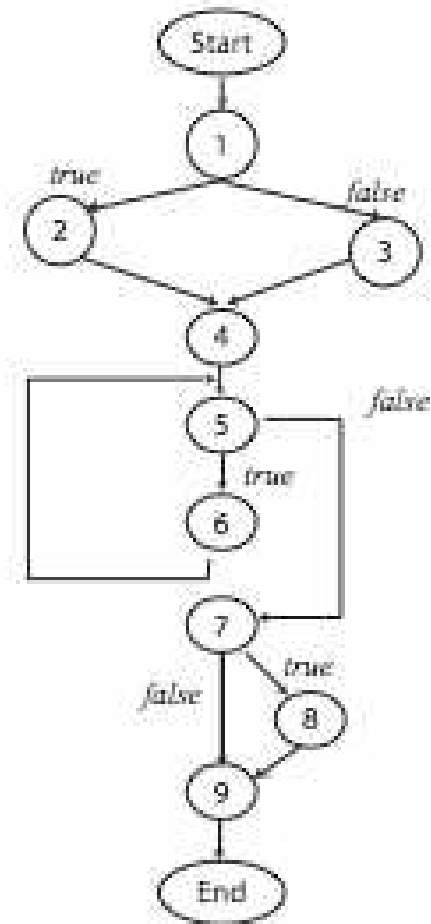
CFG Example (1)

- $N = \{\text{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{End}\}$
- $E = \{(\text{Start}, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, \text{End})\}$



CFG Example (2)

- Same CFG with statements removed.
- $N = \{\text{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{End}\}$
- $E = \{(\text{Start}, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, \text{End})\}$

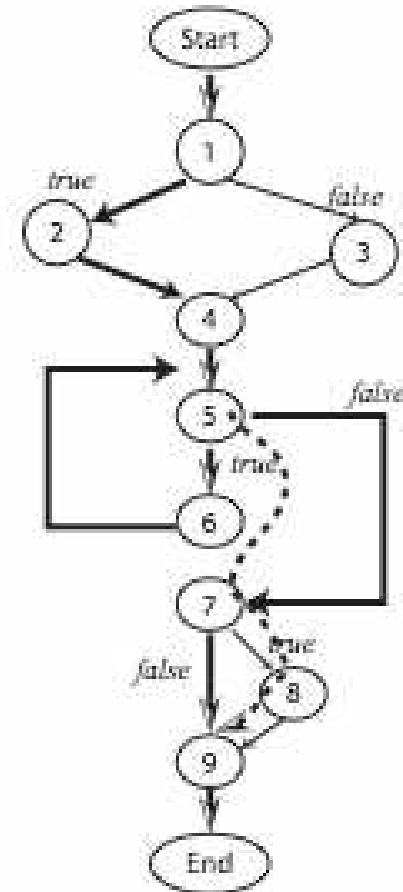


Paths (1)

- Consider a flow graph $G = (N, E)$. *A sequence of k edges*, $k > 0$, (e_1, e_2, \dots, e_k) , denotes *a path of length k* through the flow graph if the following sequence condition holds.
- Given that n_p, n_q, n_r , and n_s are nodes belonging to N , and $0 < i < k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$

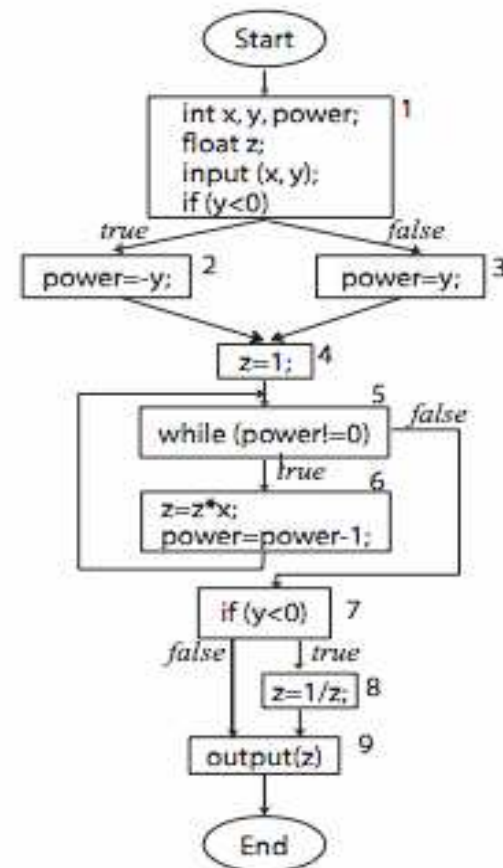
Paths (2)

- Two *feasible* and *complete* paths:
- $p_1 = (\text{Start}, 1, 2, 4, 5, 6, 5, 7, 9, \text{End})$
- $p_2 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 9, \text{End})$
- Specified unambiguously using edges:
- $p_1 = ((\text{Start}, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, \text{End}))$
- Bold edges: *complete path*
- Dashed edges: *subpath*



Paths: Feasible Paths

- A path p through a flow graph for program \mathcal{P} is considered *feasible* if there exists *at least one test case* which when input to \mathcal{P} causes p to be traversed.
- $p_1 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 8, 9, \text{End})$
- $p_2 = (\text{Start}, 1, 1, 2, 4, 5, 7, 9, \text{End})$



Paths: Infeasible Paths (Defensive Code)

```
#include <io.h>

void isOdd(int i) {

    switch (i % 2) {
    case 0:
        printf("The input value is even!\n");
        break;
    case 1:
        printf("The input value is odd!\n");
        break;
    default:
        printf("No such value\n");
        break;
    }
}

void main() {

    int i = 0;
    scanf("%d", &i);
    isOdd(i);
}
```

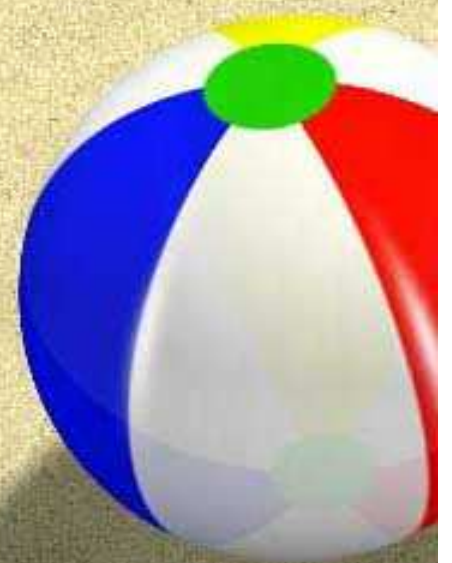
Paths: Infeasible Paths (Program Bug)

- `if ($k < 2$)`
 {
 `if ($k > 3$)` [should be: $k > -3$]
 `$x = x \times k$;`
 }
- `for ($i = 0$; $i < 0$; $i++$)` [should be: $i < 10$]
 {
 `total = total + value[i];`
 }

Number of Paths

- There can be *many distinct paths* through a program.
A program with no condition contains exactly one path that begins at node Start and terminates at node End.
- Each additional condition in the program can increase the number of distinct paths by *at least one*.
- Depending on their location, conditions can have a multiplicative effect on the number of paths.
 - two nested if-then-else
 - while loop
 - for loop

Test Generation



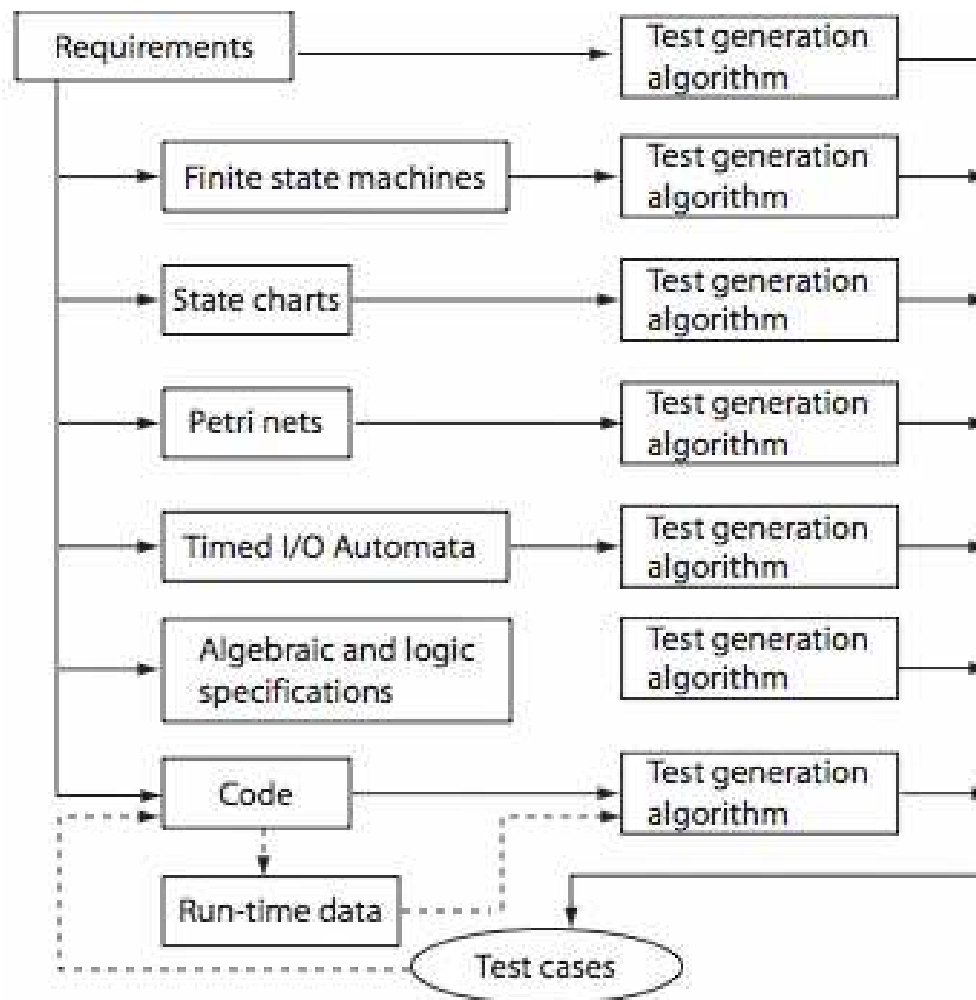
Test Generation

- Any form of test generation uses a source document. In the most informal of test methods, *the source document resides in the mind of the tester* who generates tests based on a knowledge of the requirements.
- In most commercial environments, the process is a bit more formal. The tests are generated *using a mix of formal and informal methods* from the *requirements document* serving as the source. *In more advanced test processes, requirements serve as a source for the development of test plans.*

Test Generation Strategies (1)

- **Model based**: require that a subset of the requirements be *modeled using a formal notation (usually graphical)*. Models: Finite State Machines, Timed automata, Petri net, SDL, UML, etc.
- **Specification based**: require that a subset of the requirements be *modeled using a formal mathematical notation*. Examples: B, Z, and Larch.
- **Code based**: generate tests *directly from the code*.

Test Generation Strategies (2)



Types of Software Testing



Types of Testing

- One possible classification is based on the following four classifiers:
 - C1: Source of test generation.
 - C2: Lifecycle phase in which testing takes place
 - C3: Goal of a specific testing activity
 - C4: Characteristics of the artifact under test

C1: Source of Test Generation

Artifact	Technique	Example
Requirements (informal)	Black-box	Ad-hoc testing Boundary value analysis Category partition Classification trees Cause-effect graphs Equivalence partitioning Partition testing Predicate testing Random testing
Code	White-box	Adequacy assessment Coverage testing Data-flow testing Domain testing Mutation testing Path testing Structural testing Test minimization using coverage
Requirements and code	Black-box and White-box	
Formal model: Graphical or mathematical specification	Model-based Specification	Statechart testing FSM testing Pairwise testing Syntax testing
Component interface	Interface testing	Interface mutation Pairwise testing

C2: Lifecycle Phase in Which Testing Takes Place

Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Post system, pre-release	Beta-testing

C3: Goal of Specific Testing Activity

Goal	Technique	Example
Advertised features	Functional testing	
Security	Security testing	
Invalid inputs	Robustness testing	
Vulnerabilities	Vulnerability testing	
Errors in GUI	GUI testing	Capture/plaback Event sequence graphs Complete Interaction Sequence
Operational correctness	Operational testing	Transactional-flow
Reliability assessment	Reliability testing	
Resistance to penetration	Penetration testing	
System performance	Performance testing	Stress testing
Customer acceptability	Acceptance testing	
Business compatibility	Compatibility testing	Interface testing Installation testing
Peripherals compatibility	Configuration testing	

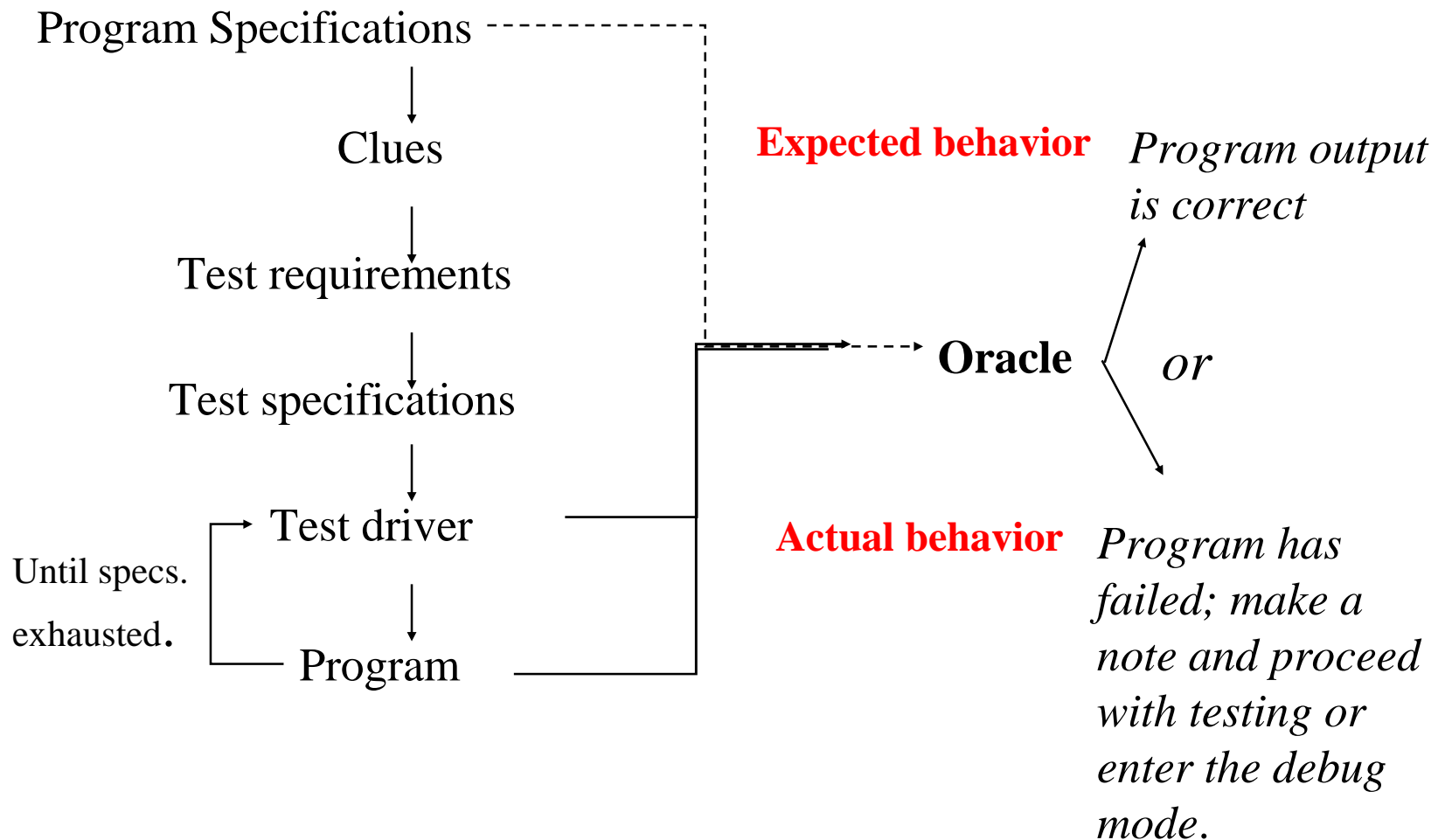
Functional Testing

- Testing either an *element* of or the *complete* product to determine whether it will function as planned.
- The *system testing* of an *integrated, black-box system* against its operational (i.e., functional) requirements.
- Testing the *advertised features* of a system for correct operation.
- Geared towards verifying that a product/application *conforms to all functional specifications*.
- Entail the following tasks: *test generation* from requirements or some model of the requirements, *test execution*, and *test assessment*.

C4: Artifact Under Test

Characteristics	Technique
Application component	Component testing
Client and server	Client-server testing
Compiler	Compiler testing
Design	Design testing
Code	Code testing
Database system	Transaction-flow testing
OO software	OO testing
Operating system	Operating system testing
Real-time software	Real-time testing
Requirements	Requirement testing
Software	Software testing
Web service	Web service testing

Test Methodology



Test Requirements Checklist

- Obtaining *clues* and deriving test requirements can become a tedious task.
- To keep it from overwhelming us it is a good idea to *make a checklist of clues*.
- This checklist is then transformed into *a checklist of test requirements* by going through each clue and deriving test requirements from it.

Test Specifications (1)

- A test requirement indicates “*how*” to test a program.
But *it does not provide exact values of inputs*.
- A test requirement is used to *derive test specification*, which is the *exact specification of values of input and environment variables*.

Test Specifications (2)

- There may *not be a one-to-one correspondence* between test requirements and test specifications.
- A test requirement checklist might contain 50 entries. These might result in only 22 test specifications.
- The fewer the tests the better but only if these tests are of *good quality*
 - We will discuss *test quality* when discussing *test assessment*.

Test Requirements to Specifications (1)

- The test requirements **checklist** guides the process of deriving test specifications.
- Initially all entries in the checklist are unmarked or set to 0.
- Each time a test is generated from a requirement it is marked or the count incremented by 1.

Test Requirements to Specifications (2)

- Thus, at any point in time, one *could assess the progress* made towards the generation of test specifications.
- One could also determine how many tests have been generated using any test requirement.

Test Requirements to Specifications (3)

- Once a test requirement has been marked or its count is more than 0 we say that it has been satisfied.
- *Some rules of thumb* to use while designing tests:
 - Try to satisfy multiple requirements using only one test.
 - Try to satisfy each requirement by more than one test.
 - Satisfy all test requirements.
 - Avoid reuse of same values of a variable in different tests.
- *In testing, variety helps!*

Test Requirements to Specifications (4)

- Though we try to combine several test requirements to generate one test case, this is **not advisable when considering error conditions.**

Test Requirements to Specifications (5)

- For example, consider the following:
 - speed_dial, an interval
 - ❑ speed_dial < 0, error
 - ❑ speed_dial > 120, error
 - zones, an interval
 - ❑ zones < 5, error
 - ❑ zones > 10, error

Test Requirements to Specifications (6)

- One test specification obtained by combining the two requirements above is:
 - speed_dial = -1
 - Zone = 3
- Now, if the code to handle these error conditions is:

Test Requirements to Specifications (7)

- **if** (speed_dial<0 || speed_dial>120)
 error_exit (“Incorrect speed_dial”)

if (zone<6 || zone>10)
 error_exit (“Incorrect zone”);

Bug!

- For our test, the program will exit before it reaches the second *if* statement. Thus, it will miss detecting the error in coding for *zone*.



Summary

*We have dealt with some of the most
basic concepts in software testing.*