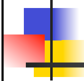
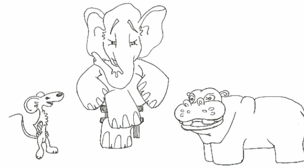
# Foundations of Software Testing

## Chapter 1: Preliminaries

Aditya P. Mathur
Purdue University





These slides are copyrighted. They are for use with the **Foundations of Software Testing** book by Aditya Mathur. Please use the slides but do not remove the copyright notice.

Last update: September 3, 2007

## Learning Objectives

- Errors, Testing, debugging, test process, CFG, correctness, reliability, oracles.
- Finite state machines
- Testing techniques

© Aditya P. Mathur 2005



## Errors, faults, failures

## Errors

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.*

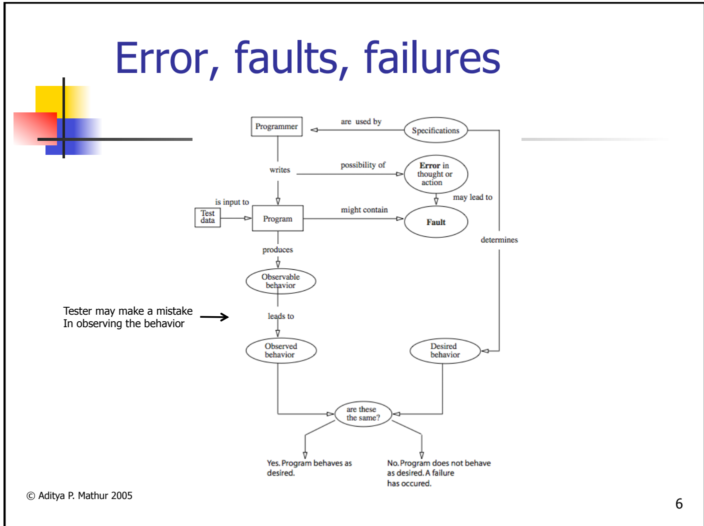
© Aditya P. Mathur 2005

# Errors: Examples

| Area | Error |
|---|---|
| Hearing | Spoken: He has a garage for repairing *foreign* cars. |
| | Heard: He has a garage for repairing *falling* 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: new_line, correct identifier: next_line. |
| | 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: *waple malnut*, intent: *maple walnut*. |
| | Spoken: *We need a new refrigerator*, intent: *We need a new washing machine*. |
| Sports | Incorrect call by the referee in a tennis match. |
| Writing | Written: What kind of *pans* did you use? |
| | Intent: What kind of *pants* did you use? |

© Aditya P. Mathur 2005





# Software quality

**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

© Aditya P. Mathur 2005

## Software quality (contd.)

**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. An example of inconsistency would be when a database application displays the date of birth of a person in the database in different formats ignoring user preference.

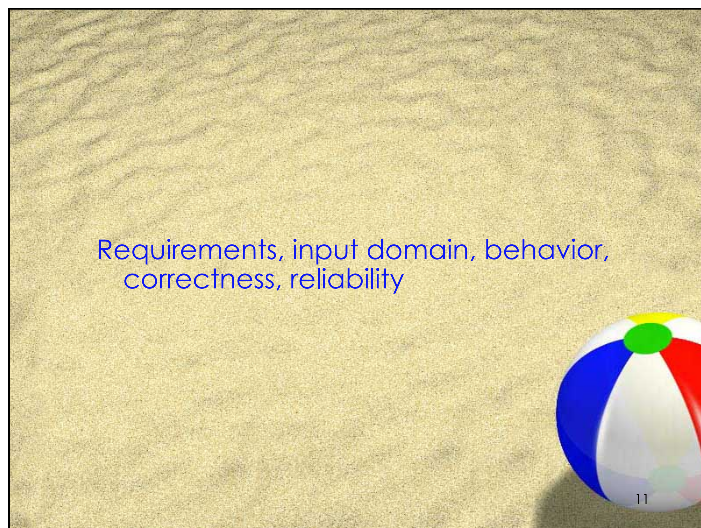
© Aditya P. Mathur 2005


## Software quality (contd.)

**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."

© Aditya P. Mathur 2005


## 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.

© Aditya P. Mathur 2005

## Requirements: Incompleteness

Suppose that 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.

© Aditya P. Mathur 2005

## Requirements: Ambiguity

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

© Aditya P. Mathur 2005

## Input domain (Input space)

*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 till 32,767.

© Aditya P. Mathur 2005

## Input domain (Continued)

**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.

© Aditya P. Mathur 2005

## Input domain (Continued)

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.

© Aditya P. Mathur 2005

## 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.

© Aditya P. Mathur 2005

## 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.*

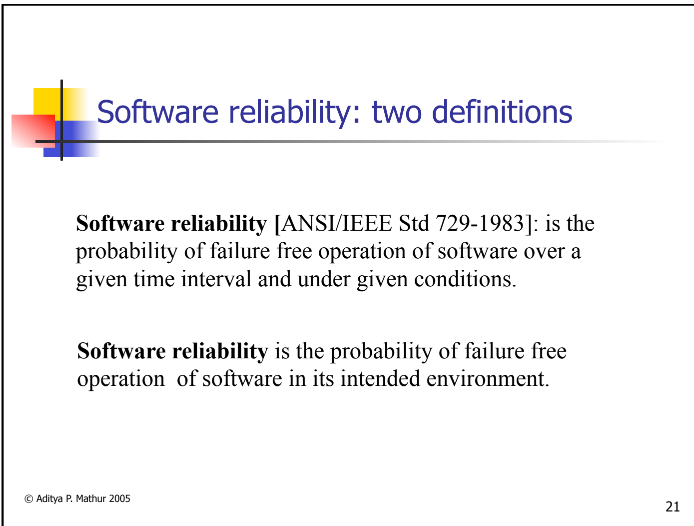
© Aditya P. Mathur 2005

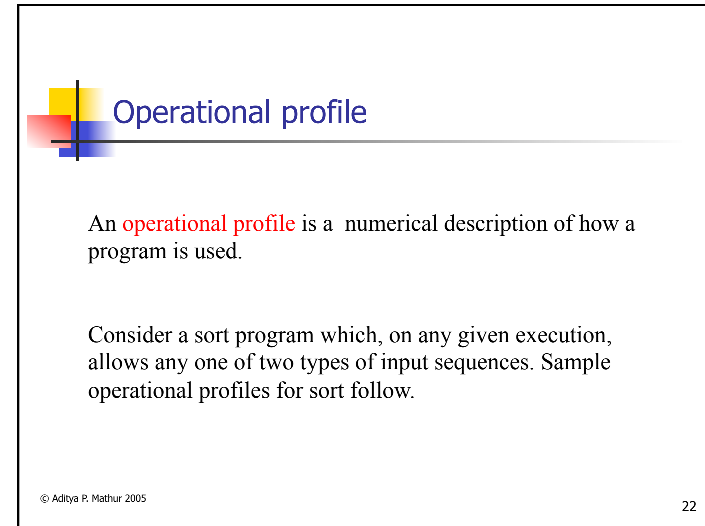
## Correctness and Testing

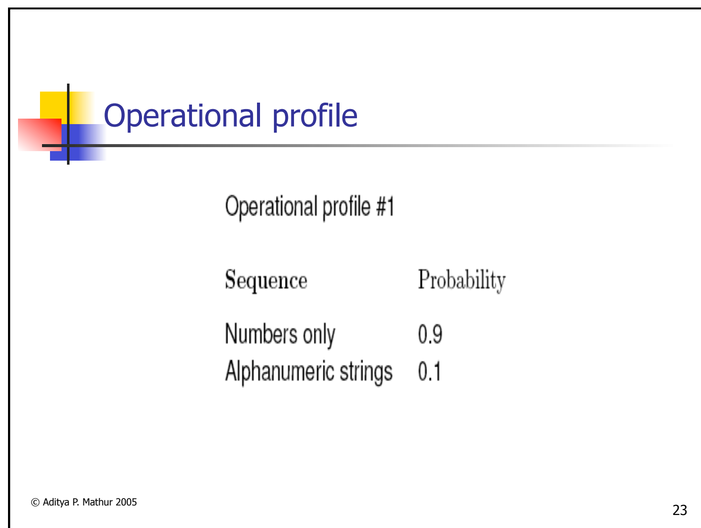
While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it. Thus testing does not demonstrate that a program is error free.

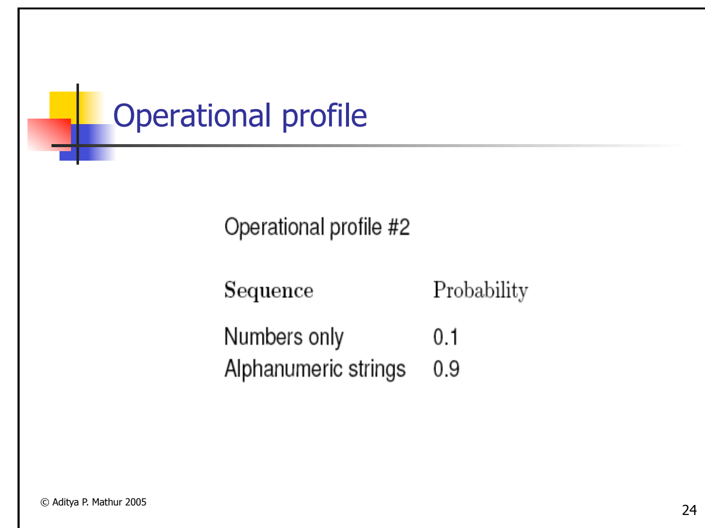
*Testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.*

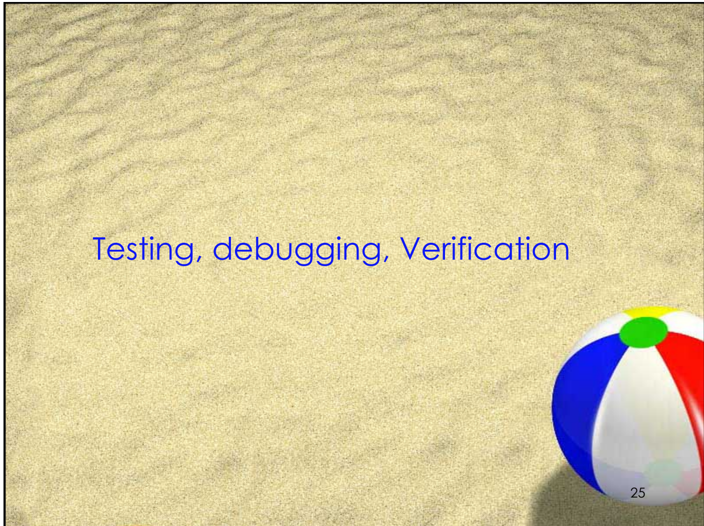
© Aditya P. Mathur 2005


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.
© Aditya P. Mathur 2005
21


Operational profile
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. Sample operational profiles for sort follow.
© Aditya P. Mathur 2005
22


Operational profile
Operational profile #1
Sequence Probability
Numbers only 0.9
Alphanumeric strings 0.1
© Aditya P. Mathur 2005
23


Operational profile
Operational profile #2
Sequence Probability
Numbers only 0.1
Alphanumeric strings 0.9
© Aditya P. Mathur 2005
24



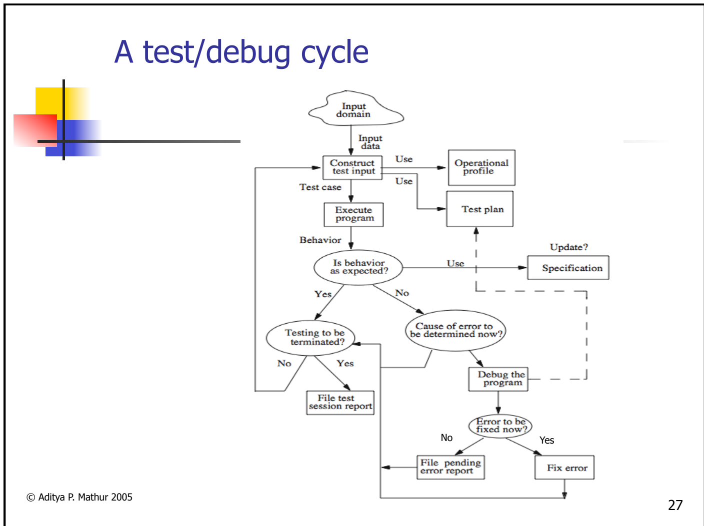

## Testing and debugging

Testing is the process of determining if a program has any errors.

When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as debugging.

© Aditya P. Mathur 2005






## Test plan

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 request characters.

© Aditya P. Mathur 2005

## Test plan (contd.)

- Execute the program on an empty input sequence.
- 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.

© Aditya P. Mathur 2005


## 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

© Aditya P. Mathur 2005


## Program behavior

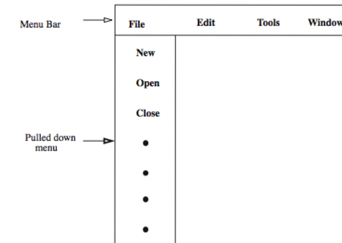
Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.

A state diagram specifies program states and how the program changes its state on an input sequence. inputs.

© Aditya P. Mathur 2005

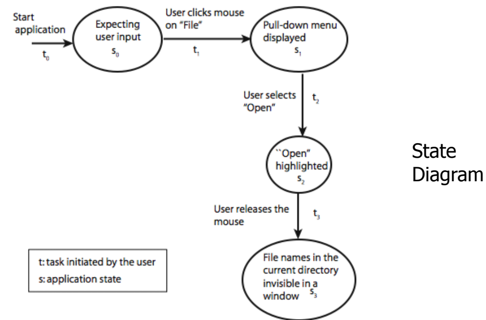

## Program behavior: Example

Consider a menu driven application.




© Aditya P. Mathur 2005

## Program behavior: Example (contd.)




State Diagram

© Aditya P. Mathur 2005



## Behavior: observation and analysis

In the first step one observes the behavior.

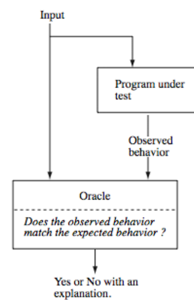
In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle.

© Aditya P. Mathur 2005



## Oracle: Example




© Aditya P. Mathur 2005



## 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.

© Aditya P. Mathur 2005

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.

© Aditya P. Mathur 2005

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.

© Aditya P. Mathur 2005

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 mistake 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.*

© Aditya P. Mathur 2005

## 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.

© Aditya P. Mathur 2005



## Basic blocks: Example

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
```

© Aditya P. Mathur 2005



## Basic blocks: Example (contd.)

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 |

© Aditya P. Mathur 2005



## Control Flow Graph (CFG)

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.

© Aditya P. Mathur 2005

## Control Flow Graph (CFG)

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.

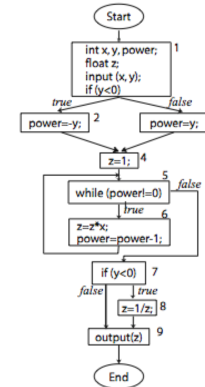
© Aditya P. Mathur 2005



## CFG Example

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (
End)}




© Aditya P. Mathur 2005

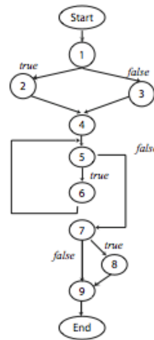


## CFG Example

Same CFG with statements removed.

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (
End)}




© Aditya P. Mathur 2005



## Paths

Consider a flow graph G= (N, E). A sequence of k edges, k>0, (e_1, e_2, … 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 $n_q = n_r$. }

© Aditya P. Mathur 2005

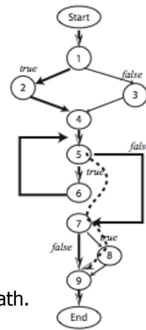
## Paths: sample paths through the exponentiation flow graph

Two feasible and complete paths:

$p_1$= ( Start, 1, 2, 4, 5, 6, 5, 7, 9, End)
$p_2$= (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)

Specified unambiguously using edges:

$p_1$= ( (Start, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, End))




Bold edges: complete path.

Dashed edges: subpath.

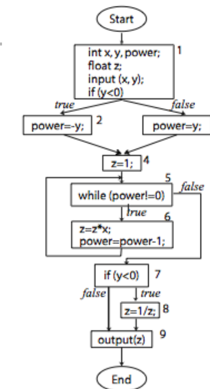
© Aditya P. Mathur 2005



## Paths: feasible paths

A path p through a flow graph for program P is considered feasible if there exists at least one test case which when input to P causes p to be traversed.

$p_1$= ( Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)
$p_2$= (Start, 1, 1, 2, 4, 5, 7, 9, , End)




© Aditya P. Mathur 2005



## 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 increases the number of distinct paths by at least one.

Depending on their location, conditions can have a multiplicative effect on the number of paths.

© Aditya P. Mathur 2005



## 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 directly from the requirements document serving as the source. In more advanced test processes, requirements serve as a source for the development of formal models.

© Aditya P. Mathur 2005


## Test generation strategies

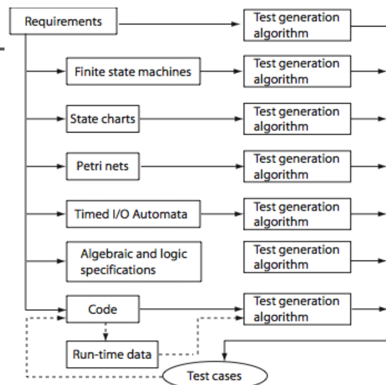
Model based: require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri nets, etc.

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.

© Aditya P. Mathur 2005


## Test generation strategies (Summary)




© Aditya P. Mathur 2005


## Strings and languages

## Strings

Strings play an important role in testing. A string serves as a test input. Examples: 1011; AaBc; "Hello world".

A collection of strings also forms a language. For example, a set of all strings consisting of zeros and ones is the language of binary numbers. In this section we provide a brief introduction to strings and languages.

© Aditya P. Mathur 2005

## Alphabet

A collection of symbols is known as an alphabet. We use an upper case letter such as X and Y to denote alphabets.

Though alphabets can be infinite, we are concerned only with finite alphabets. For example, X={0, 1} is an alphabet consisting of two symbols 0 and 1. Another alphabet is Y={dog, cat, horse, lion}that consists of four symbols ``dog", ``cat", ``horse", and ``lion".

© Aditya P. Mathur 2005

## Strings over an Alphabet

A string over an alphabet X is any sequence of zero or more symbols that belong to X. For example, 0110 is a string over the alphabet {0, 1}. Also, dog cat dog dog lion is a string over the alphabet {dog, cat, horse, lion}.

We will use lower case letters such as p, q, r to denote strings. The length of a string is the number of symbols in that string. Given a string s, we denote its length by |s|. Thus |1011|=4 and |dog cat dog|=3. A string of length 0, also known as an empty string, is denoted by ε.

*Note that ε denotes an empty string.*

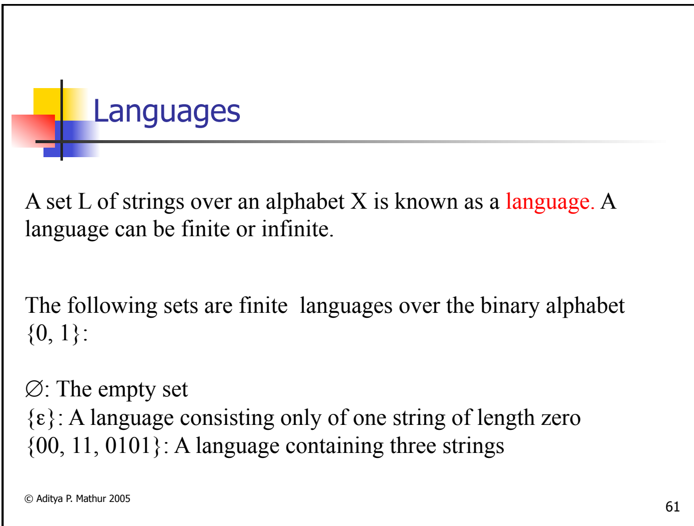
© Aditya P. Mathur 2005

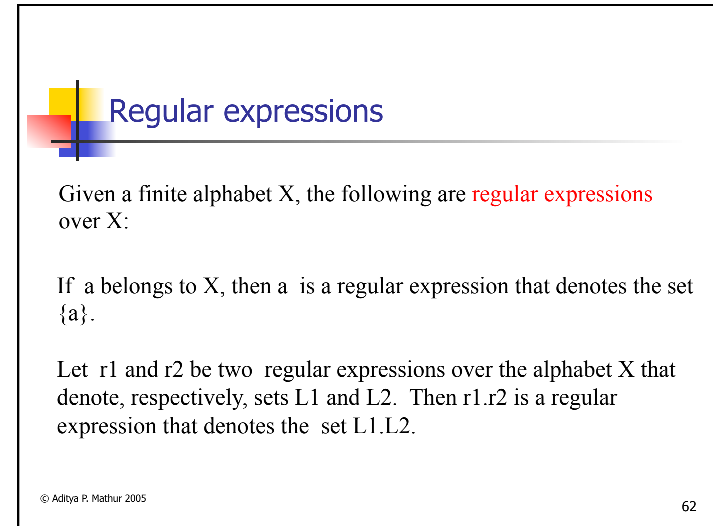
## String concatenation

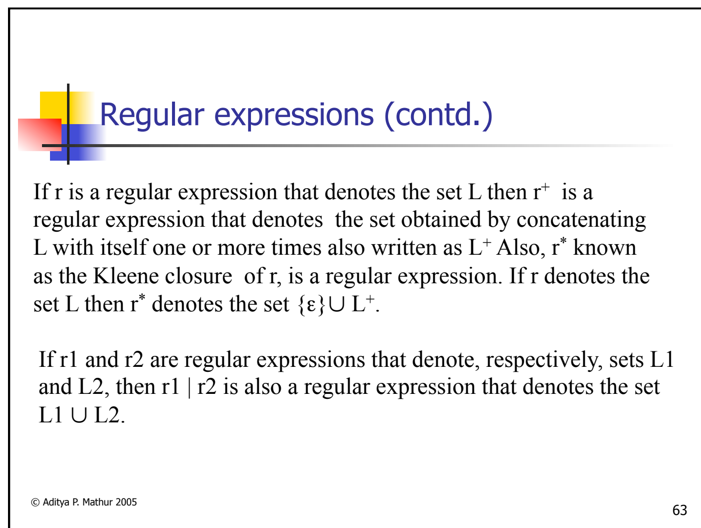
Let s1 and s2 be two strings over alphabet X. We write s1.s2 to denote the concatenation of strings s1 and s2.

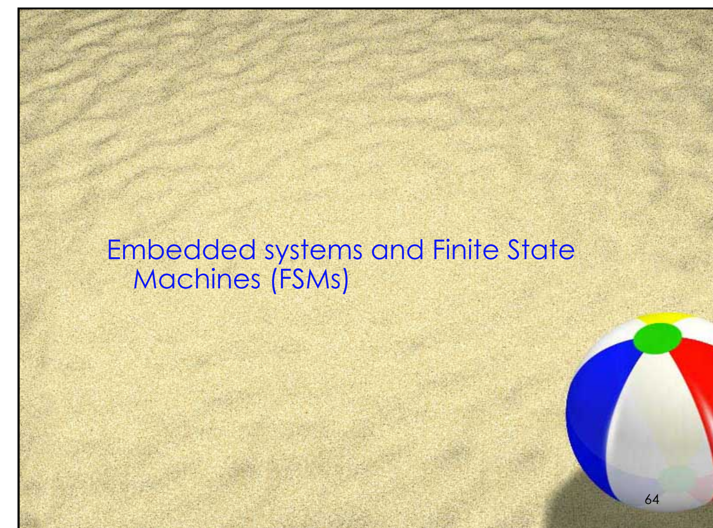
For example, given the alphabet X={0, 1}, and two strings 011 and 101 over X, we obtain 011.101=011101. It is easy to see that |s1.s2|=|s1|+|s2|. Also, for any string s, we have s. ε =s and ε.s=s.

© Aditya P. Mathur 2005


Languages
A set L of strings over an alphabet X is known as a language. A language can be finite or infinite.
The following sets are finite languages over the binary alphabet {0, 1}:
∅: The empty set
{ε}: A language consisting only of one string of length zero
{00, 11, 0101}: A language containing three strings
© Aditya P. Mathur 2005
61


Regular expressions
Given a finite alphabet X, the following are regular expressions over X:
If a belongs to X, then a is a regular expression that denotes the set {a}.
Let r1 and r2 be two regular expressions over the alphabet X that denote, respectively, sets L1 and L2. Then r1.r2 is a regular expression that denotes the set L1.L2.
© Aditya P. Mathur 2005
62


Regular expressions (contd.)
If r is a regular expression that denotes the set L then r+ is a regular expression that denotes the set obtained by concatenating L with itself one or more times also written as L+ Also, r* known as the Kleene closure of r, is a regular expression. If r denotes the set L then r* denotes the set {ε}∪ L+.
If r1 and r2 are regular expressions that denote, respectively, sets L1 and L2, then r1 | r2 is also a regular expression that denotes the set L1 ∪ L2.
© Aditya P. Mathur 2005
63


Embedded systems and Finite State Machines (FSMs)
64

## Embedded systems

Many real-life devices have computers embedded in them. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses.

Such devices are also known as embedded systems. An embedded system can be as simple as a child's musical keyboard or as complex as the flight controller in an aircraft. In any case, an embedded system contains one or more computers for processing inputs.

© Aditya P. Mathur 2005



## Specifying embedded systems

An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another.

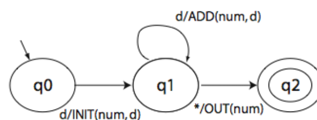
The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by a finite state machine (FSM).

© Aditya P. Mathur 2005



## FSM: Actions with state transitions

Machine to convert a sequence of decimal digits to an integer:




(a) Notice ADD, INIT, ADD,OUT actions.

(b) INIT: Initialize num. ADD: Add to num. OUT: Output num.

© Aditya P. Mathur 2005



## FSM: Formal definition

An FSM is a quintuple: $(X, Y, Q, q_0, \delta, O)$, where:,

X is a finite set of input symbols also known as the input alphabet.

Y is a finite set of output symbols also known as the output alphabet,

Q is a finite set states,

© Aditya P. Mathur 2005

## FSM: Formal definition (contd.)

$q_0$ in Q is the initial state,

δ: Q x X→ Q is a next-state or state transition function, and

O: Q x X→ Y is an output function.

In some variants of FSM more than one state could be specified as an initial state. Also, sometimes it is convenient to add $F \subseteq Q$ as a set of final or accepting states while specifying an FSM.

© Aditya P. Mathur 2005



## State diagram representation of FSM

A state diagram is a directed graph that contains nodes representing states and edges representing state transitions and output functions.

Each node is labeled with the state it represents. Each directed edge in a state diagram connects two states. Each edge is labeled i/o where i denotes an input symbol that belongs to the input alphabet X and o denotes an output symbol that belongs to the output alphabet O. i is also known as the input portion of the edge and o its output portion.

© Aditya P. Mathur 2005



## Properties of FSM

Completely specified: An FSM M is said to be completely specified if from each state in M there exists a transition for each input symbol.

Strongly connected: An FSM M is considered strongly connected if for each pair of states $(q_i\, q_j)$ there exists an input sequence that takes M from state $q_i$ to $q_j$.

© Aditya P. Mathur 2005



## Properties of FSM: Equivalence

V-equivalence: Let $M_1=(X, Y, Q_1, m^1_0, T_1, O_1)$ and $M_2=(X, Y, Q_2, m^2_0, T_2, O_2)$ be two FSMs. Let V denote a set of non-empty strings over the input alphabet X i.e. $V \subseteq X^+$.

Let $q_i$ and $q_j$, $i \neq j$, be the states of machines $M_1$ and $M_2$, respectively. $q_i$ and $q_j$ are considered V-equivalent if $O_1(q_i, s)=O_2(q_j, s)$ for all s in V.

© Aditya P. Mathur 2005

## Properties of FSM: Distinguishable

Stated differently, states $q_i$ and $q_j$ are considered V-equivalent if $M_1$ and $M_2$ , when excited in states $q_i$ and $q_j$, respectively, yield identical output sequences.

States $q_i$ and $q_j$ are said to be equivalent if $O_1(q_i, r)=O_2(q_j, r)$ for any set V. If $q_i$ and $q_j$ are not equivalent then they are said to be distinguishable. Thus machines $M_1$ and $M_2$ could be the same machine.
* This definition of equivalence also applies to states within a machine.

© Aditya P. Mathur 2005

## Properties of FSM: Machine Equivalence

Machine equivalence: Machines $M_1$ and $M_2$ are said to be equivalent if (a) for each state σ in M1 there exists a state σ ' in $M_2$ such that σ and σ ' are equivalent and (b) for each state σ in $M_2$ there exists a state σ ' in $M_1$ such that σ and σ ' are equivalent.

Machines that are not equivalent are considered distinguishable.

Minimal machine: An FSM M is considered minimal if the number of states in M is less than or equal to any other FSM equivalent to M.

© Aditya P. Mathur 2005

## Properties of FSM: k-equivalence

k-equivalence: Let $M_1=(X, Y, Q_1, m^1{}_0, T_1, O_1)$ and $M_2=(X, Y, Q_2, m^2{}_0, T_2, O_2)$ be two FSMs.

States $q_i \varepsilon Q_1$ and $q_j \varepsilon Q_2$ are considered k-equivalent if, when excited by any input of length k, yield identical output sequences.

© Aditya P. Mathur 2005

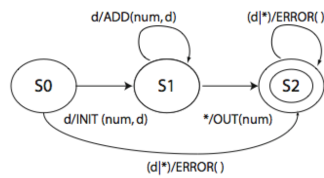
## Properties of FSM: k-equivalence (contd.)

States that are not k-equivalent are considered k-distinguishable.

It is also easy to see that if two states are k-distinguishable for any k>0 then they are also distinguishable for any n≥ k. If $M_1$ and $M_2$ are not k-distinguishable then they are said to be k-equivalent.

© Aditya P. Mathur 2005

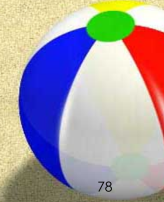
## Example: Completely specified machine




© Aditya P. Mathur 2005



## 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

© Aditya P. Mathur 2005



## C1: Source of test generation

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

© Aditya P. Mathur 2005

## 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 |

© Aditya P. Mathur 2005



## 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<br>Event sequence graphs<br>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<br>Installation testing |
| Peripherals compatibility | Configuration testing | |

© Aditya P. Mathur 2005



## 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 |

© Aditya P. Mathur 2005