ST Condensed Summary Notes For Quick In-Exam Strategic Fact Deployment

Maksymilian Mozolewski

May 16, 2021

Contents

ST

Introduction

Validation

Checking that the software specification adheres to the requirements of the customer. Are we building the right product?

Verification

Checking that the software adheres to the software specification. Are we building the product right ?

Software Specification

Defines the product being created, includes:

- **Functional requirements** - describe features of a product
- **Non-Functional requirements** - constraints on the product (e.g. security, reliability etc.)

Software bug

A software bug occurs when one of the following occurs:

- The software does not do something that the true specification says it should do.
- The software does something that the specification says it should not do.
- The software does something that the specification does not mention.
- The software does not do something that the product specification does not mention but should.
- The software is difficult to understand, hard to use, slow, etc..

Phases of Software Testing

- Test generation - Writing tests, Automatic test generation etc.
- Test Execution - Actually executing tests, build tools etc.
- Test Oracle - determining what the output for each test **should** be
- Test Adequacy - determining the effectiveness of the test suite.

Software Fault

A static defect in software

Software Error

An incorrect internal state that is the manifestation of some **fault**

Software Failure

External, incorrect behaviour with respect to the requirements, or other description of expected behaviour. (The actual moment the software fails, for some input space)

Unit Testing

Unit Testing

Looking for errors in a subsystem in isolation:

- Generally a class or object
- Junit in Java

Junit 4

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	`@Test`	`@Test`
Execute before all test methods in the current class	`@BeforeClass`	`@BeforeAll`
Execute after all test methods in the current class	`@AfterClass`	`@AfterAll`
Execute before each test method	`@Before`	`@BeforeEach`
Execute after each test method	`@After`	`@AfterEach`
Disable a test method / class	`@Ignore`	`@Disabled`

Unit Testing Tips

- Tests need to be **atomic** - the failure of a test should pinpoint the location of the fault
- Each test name should be clear, long and descriptive
- Assertions should always have clear messages to know what failed
- Write many small tests, not one big one ($\approx$ 1 assertion per test)
- Test for expected errors/exceptions too
- Choose descriptive assert method (not always assertTrue, i.e. prefer more specific methods)
- Choose representative test cases for equivalent input classes
- Avoid complex logic in test methods if possible
- Use helpers, @Before to reduce redundancy between tests
- Never rely on test execution order, or call other test fixtures (apart from helpers)
- Dont share state between tests

Functional Testing

Functional Testing (Black-box testing)

Deriving tets cases from program specifications. Functional refers to the source of information used in test case design, not to what is tested Also known as **specification-based** testing.
Functioal specification = description of intended program behaviour. We act as if we didn't know anything about the code (hence black-box)
Example:

User Goal:

Keep information private

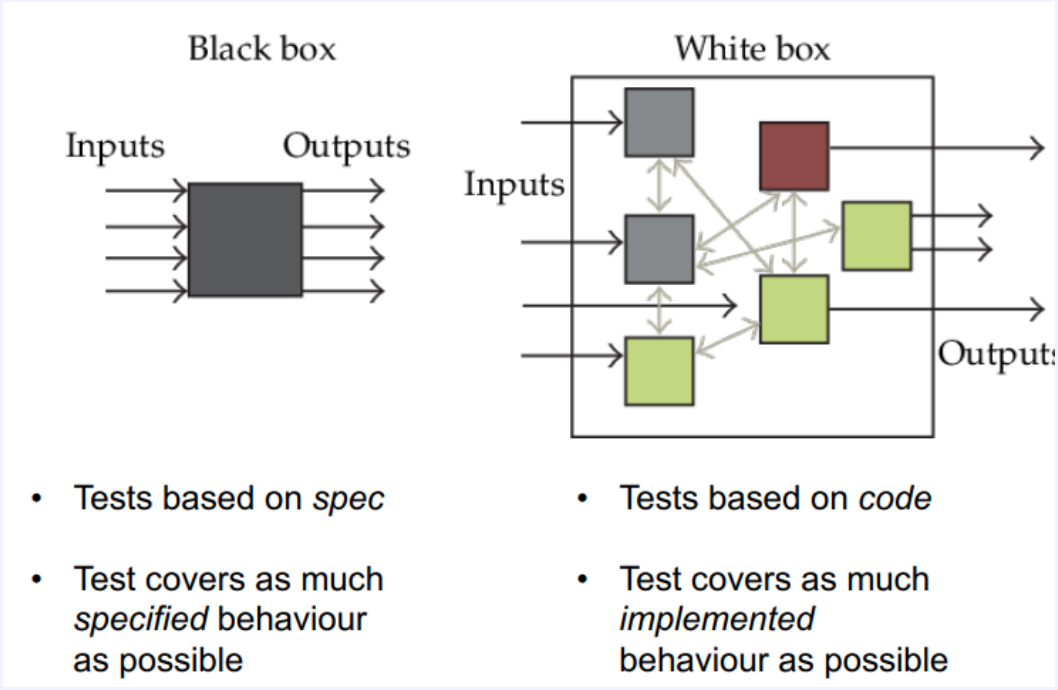
Functional Requirements:

Sec-010: Only authorized users shall access the system

Sec-020: The system administrator shall create and edit user accounts and access rights.

Sec-030: ....

Requirement ID	Requirement Description	Test Cases
Sec-010	Only authorized users shall access the system	Sec-010 Types of Test Cases 1. Attempt to log in with UserID not in the authorized user database 2. Attempt to log in with a correct UserID and incorrect user password 3. Attempt to log in from outside the firewall 4. ....



+ Often reveals ambiguities and inconsistency in specifications
+ Useful for assesing testability
+ Useful explanation of specification (test cases outline the specification)
+ does not require any code to write tests (TDD)
+ Ideal for missing logic (if we only test what's there - white box testing, we won't identify what's missing!)

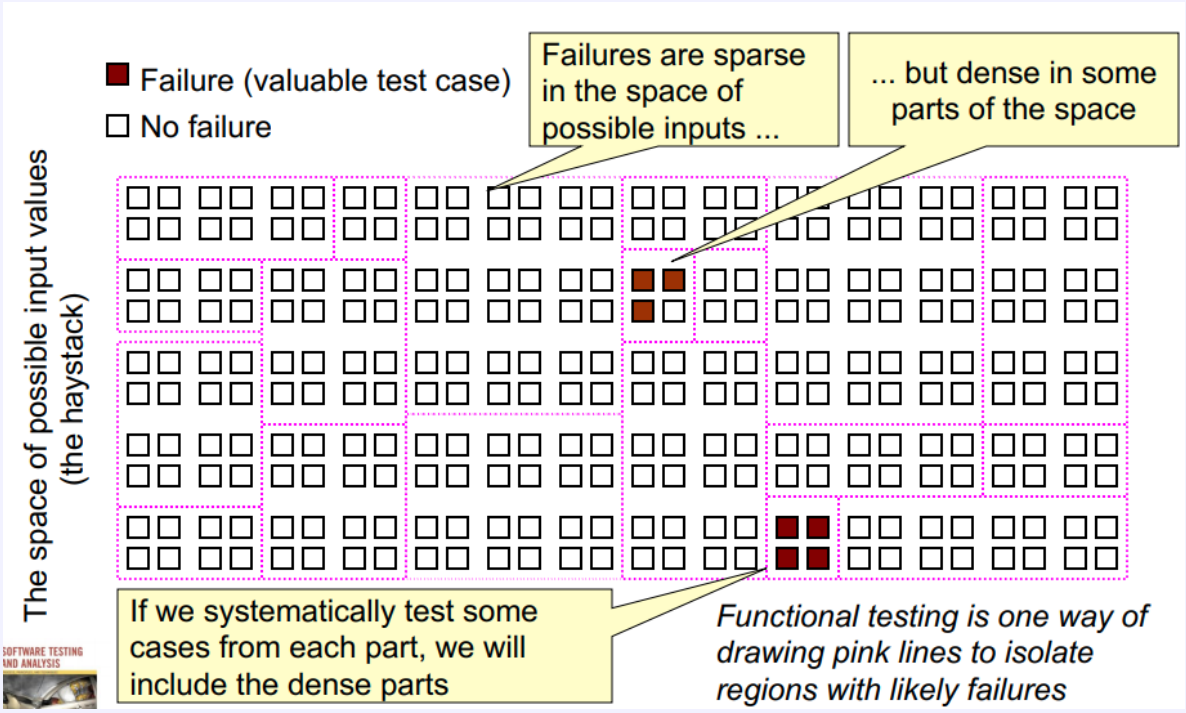
Random Testing

Pick possible inputs uniformly.

+ avoids designer bias (test designer can make logical mistakes and bad assumptions)
- treats all inputs as equally valuable
- Real faults are distributed non-uniformly (think finding roots, 3 important cases, rest is same class)
- input space is often extremely large

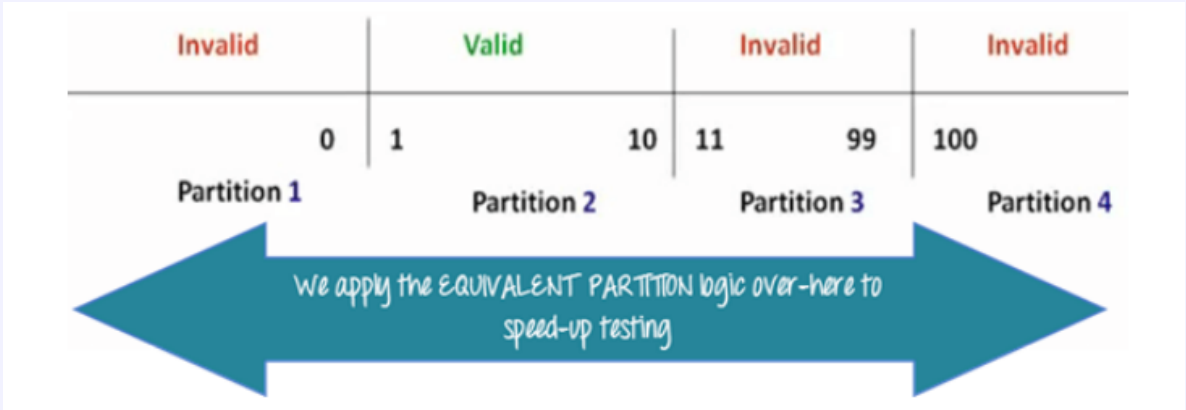
Systematic Functional Testing

Try to select inputs that are especially valuable (non-uniform selection)
Usually by choosing representatives of classes that are apt to fail often or not at all.
Functional Testing **IS** systematic testing



Partition Testing

Exploit some knowledge to choose input samples more likely to include "special" or trouble-prone regions of input space. Failures are sparce in the whole input space but we may find regions in which they are dense
Ideally want **Equivalance classes** of inputs - for example, the positive numbers might be equivalent with respect to the program, i.e. 1 and 2 will cause the same general behaviour.
Then we only really need one value from each partition, since if one condition/value in a partition passes, all others will also pass and vice-versa!

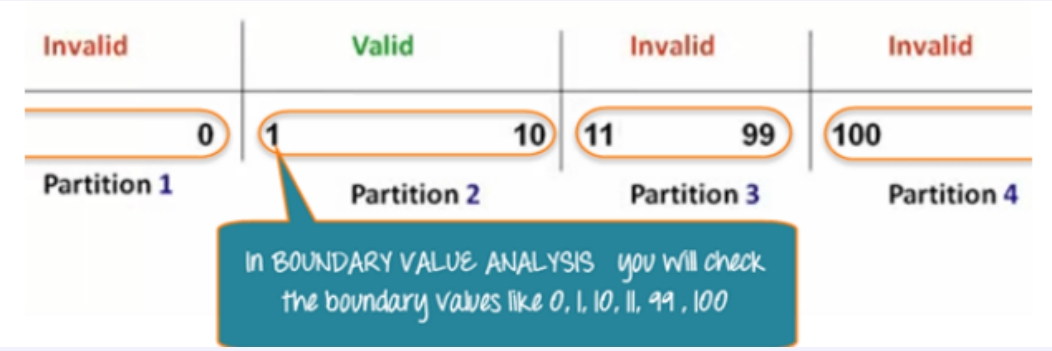


Quasi-Partition testing

Separate the input space into classes, whose union is the entire space, where the classes can overlap

Boundary Value Analysis

Testing of the inputs on the boundaries between equivalence partitions (both **valid and invalid**)



Functional vs Structural (White-box testing)

Functional testing applies at all granuality levels: Unit, Integration, System and Regression testing.
Structural testing applies to a relatively small part of the system: Unit and Integration testing.

Functional Testing Process

1. Decompose the specification into **Independently testable features** (think functions: Airport connection check etc.)
2. Select representative values of each input, or behaviours of a model (which part of the input space are interesting ?)
3. Form test specifications (input/output values, model behaviours etc)
4. Produce and execute actual tests

Combinatorial Testing

Combinatorial Testing

Identify distinct attributes that can be varied:

- Environment - characterized by combination of hardware and software (IE, Firefox, XP, OSX, 10GB Ram etc.)
- Input Parameters
- State Variables

Equivalence partitioning and **boundary value analysis** can be used to identify values of each attribute.
Systematically generate combinations of values for different attributes to be tested (combinations of attributes)
It's often impractical to test all of the combinations of distinct attributes, so we can pick certain subsets according to combination strategies.

Key ideas

- Category-partition testing - identification of values that characterise the input space, done manually, the actual combinations of attributes are generated automatically.
- Pairwise testing - systematically test interactions among attributes of the program input space with a relatively small number of test cases
- Catalog-based testing - aggregate and synthesize the experience of test designers in a particular organization or applciation domain, to aid in identifying attribute values (automates the manual parts too)

Category Partitioning (manual)

1. Decompose the specification into ITF's (Independently testable features)
2. For each feature identify:
   - Parameters (i.e. Arriving flight, Departing flight)
   - Environmental elements (State of airport database)
3. For each parameter and environment element identify **elementary characteristics** or **categories** (arrival destination matches departure origin, origin and destination airports exist in the base, database is available)
4. Identify relevant values, For each characteristic (**category**) identify **classes of** values, e.g.:
   - normal values (1,2,3)
   - boundary values (0,-1)
   - special values (INF)
   - error values (' ',"","CHEESE")
5. Introduce constraints ("[error]" cases are tested once, [property if-property] run only in combination with if-property, [single] run once)

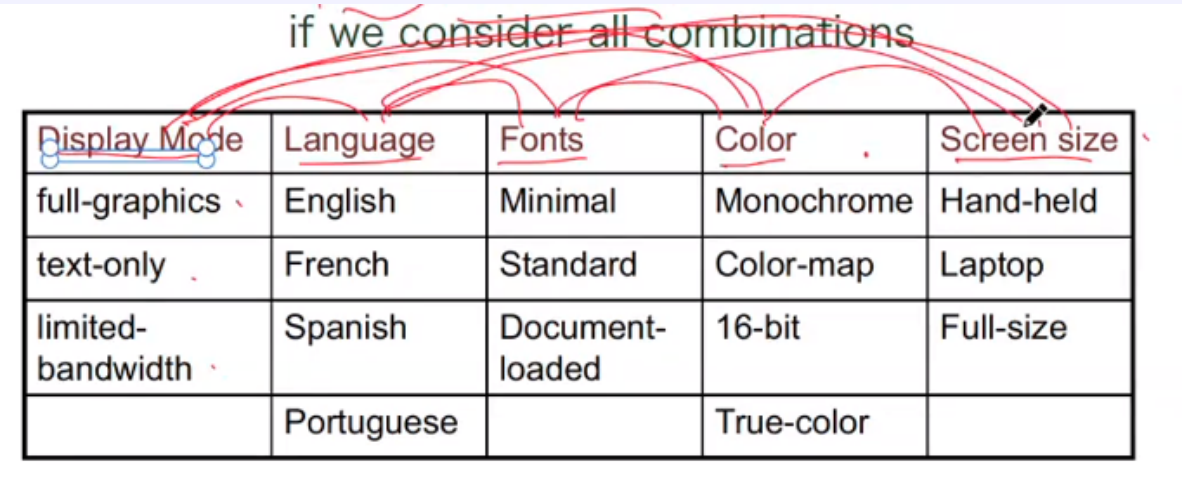
Constraints

- [error], value class which corresponds to erroneous values, only needs to be tested once, with all the other parameters and environments set to any valid values
- [property] [if-property] mark a value class as a property to be referenced by others
- [if-property] only include this property in combinations combined with value classes marked with the given property
- [single], same as error, but difference in semantics - rationale

Characteristic	Partition	Value Classes	Conditions/properties
Database ok?	**P1**	Database ok	
	P2	Database not ok	[error]
AF → OAC in DB?	**P3**	AF → OAC ∈ database	
	P4	AF → OAC ∉ database	[error]
AF → DAC in DB?	**P5**	AF → DAC ∈ database	
	P6	AF → DAC ∉ database	[error]
DF → OAC in DB?	**P7**	DF → OAC ∈ database	
	P8	DF → OAC ∉ database	[error]
DF → DAC in DB?	**P9**	DF → DAC ∈ database	
	P10	DF → DAC ∉ database	[error]
Flights meet?	**P11**	AF → DAC = DF → OAC	
	P12	AF → DAC ≠ DF → OAC	[error]
AF international?	**P13**	AF → OAC → AZ = AF → DAC → AZ	
	P14	AF → OAC → AZ ≠ AF → DAC → AZ	[property International]
DF international?	**P15**	DF → OAC → AZ = DF → DAC → AZ	
	P16	DF → OAC → AZ ≠ DF → DAC → AZ	[property International]
Connect time ok?	**P17**	CT = -1	[if International]
	P18	DF → SDT - AF → SAT < CT	
	P19	DF → SDT - AF → SAT = CT	[single]
	P20	DF → SDT - AF → SAT > CT	

Pairwise Combinatorial Testing

Generate combinations that efficently cover all pairs of classes. Most failures are triggered by single values or combinations of a few values. Covering pairs, reduces the number of test cases, but reveals most faults.



When designing test cases, we set all irrelevant values (not included in the pair) with valid values.
In this case we want test cases which cover, all combinations of values between Display Mode and Language, Display Mode and Fonts, Fonts and Color and so on and so on.
This would yield 136 pairs in total for this example $(3 \cdot 4 + 3 \cdot 3...)$

Catalog Based Testing

Deriving value classes requires human judgement. Gathering experience in a systematic collection can:

- Speed up the test design process
- Routinize many decisions, better focusing human effort
- Accelerate training and reduce human error
- Catalaogs **capture the experience of test designers** by listing important cases for each possible type of variable

Process:

1. Analyze the initial specification to identify simple elements:
   - Pre-conditions
   - Post-conditions
   - Definitions
   - Variables
   - Operations
2. Derive a first set of test case specifications from pre-conditions, post-conditions and definitions
3. Complete the set of test case specifications using test catalogs

• Boolean	
- True	in/out
- False	in/out
• Enumeration	
- Each enumerated value	in/out
- Some value outside the enumerated set	in
• Range L ... U	
- L-1	in
- L	in/out
- A value between L and U	in/out
- U	in/out
- U+1	in
• Numeric Constant C	
- C	in/out
- C -1	in
- C+1	in
- Any other constant compatible with C	in

Finite Models

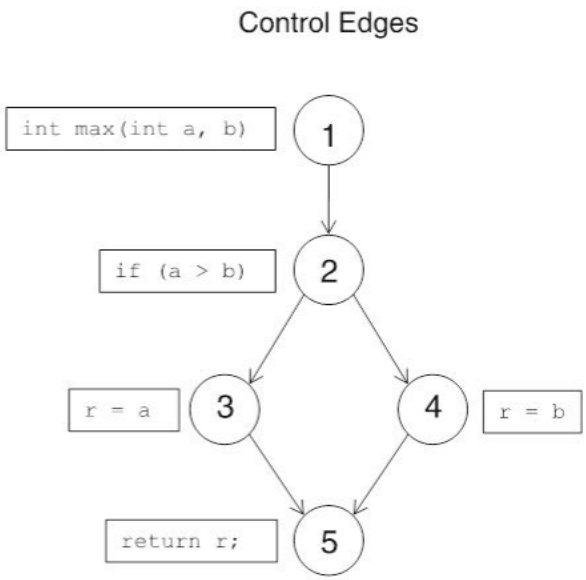
Properties of models

- Compact: representable and manipulable in a reasonably compact form
- Predictive: must represent some salient characteristics of the modelled artifact well enough to distinguish between good and bad outcomes of analysis
- Semantically meaningful: it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
- Sufficiently general: models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application

Intraprocedural Control Flow Graph

Nodes represent regions of source code (not necessarily lines), each block of code in a node has a single entry and exit point. Edges represent posibility that control flow transitions from end of one block to beginning of another

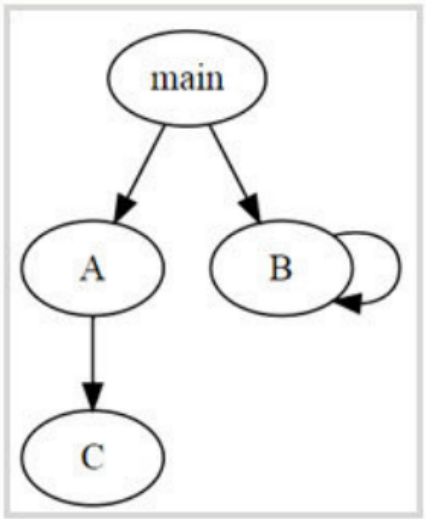
```
1  int main(int a, b) {
     int r;
2    if (a > b)
3      r = a;
     else
4    r = b;
5    return r;
   }
```



Call Graph

Similiar to control flow graph, but at higher level, nodes represent functions, and edges the possibility of a call from one call to another.

```
class Main {
  public static void main(String[] args) {
    A();
    B();
  }
  public static int A(){
    C();
  }
  public static void B(){
      B();
  }
}
```

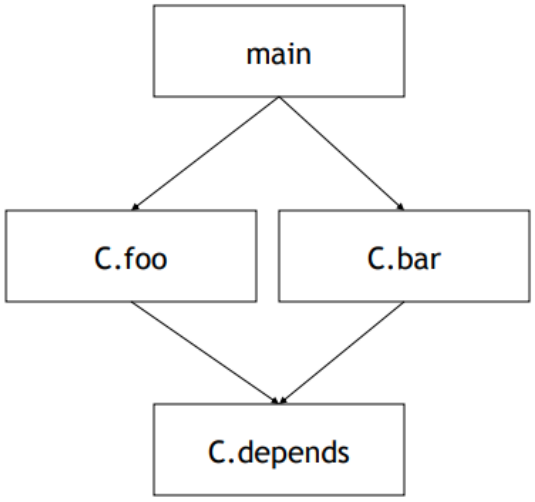


Note: the 'calls' relation represented by the edges, is overestimated via call graphs, some calls are simply impossible in real execution due to state

Context Insensitive Call Graphs

Each edge represents any call to a procedure with any context

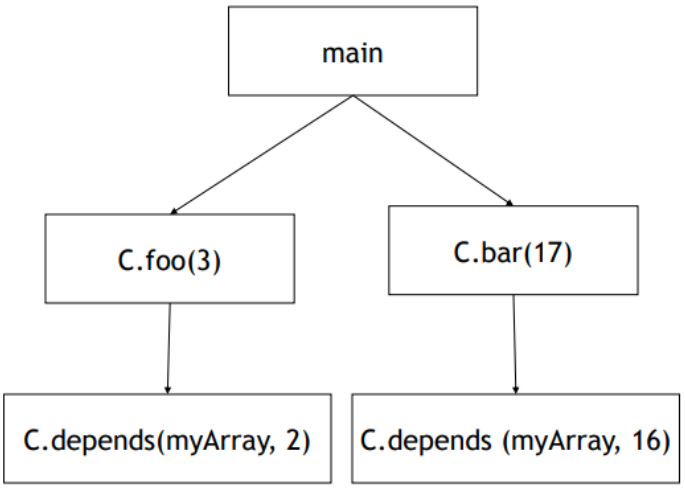
```
public class Context {
   public static void main(String
    args[]) {
    Context c = new Context();
    c.foo(3);
    c.bar(17);
   }

   void foo(int n) {
    int[]  myArray = new int[ n ];
    depends( myArray, 2) ;
   }

   void bar(int n) {
    int[]  myArray = new int[ n ];
    depends( myArray, 16) ;
   }

   void depends( int[] a, int n ) {
    a[n] = 42;
   }
}
```



Context Sensitive Call Graphs

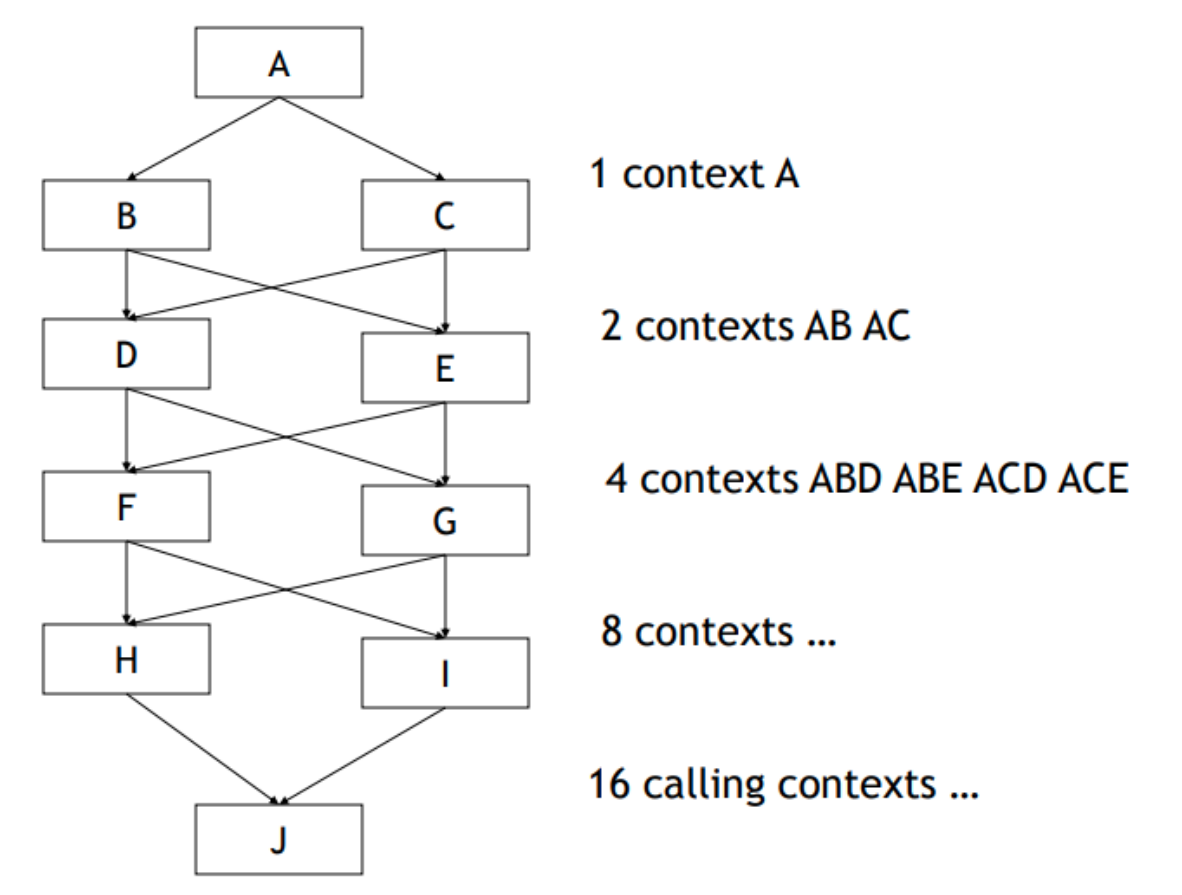
Each edge represents a unique **call stack** as well as function call (different calls have different edges)

```
public class Context {
   public static void main(String
    args[]) {
    Context c = new Context();
    c.foo(3);
    c.bar(17);
   }

   void foo(int n) {
    int[]  myArray = new int[ n ];
    depends( myArray, 2) ;
   }

   void bar(int n) {
    int[]  myArray = new int[ n ];
    depends( myArray, 16) ;
   }

   void depends( int[] a, int n ) {
    a[n] = 42;
   }
}
```



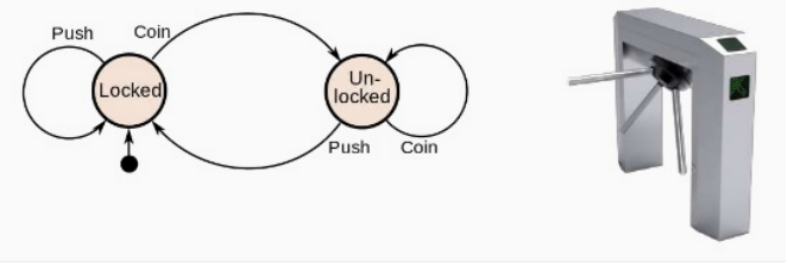
Context exponential growth

The call stacks grow exponentially making context sensitive call graphs very large:



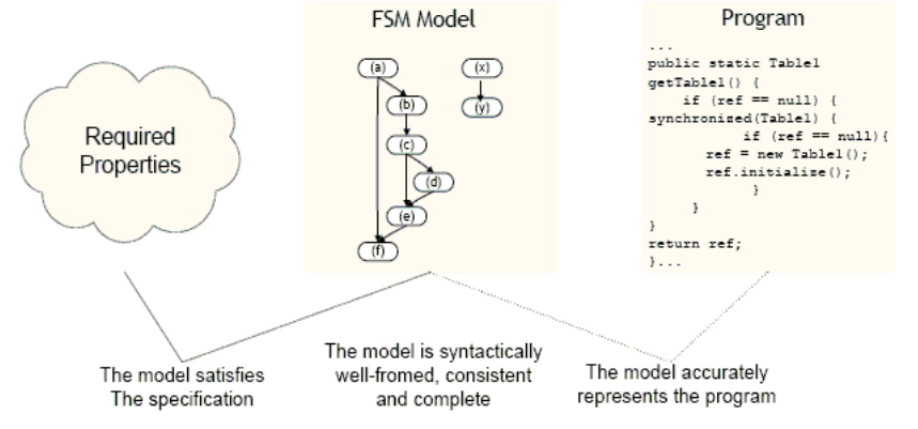
Finite State Machines

Each node is a state, and each edge is a possible transition, can be used to model high-level program behaviour



Models and System properties

Models might be easier to validate against program specifications, and if the program satisfies some model which satisfies the program specification, then those specifications are met by the program.



Structural Testing

Structural Testing (White-box testing)

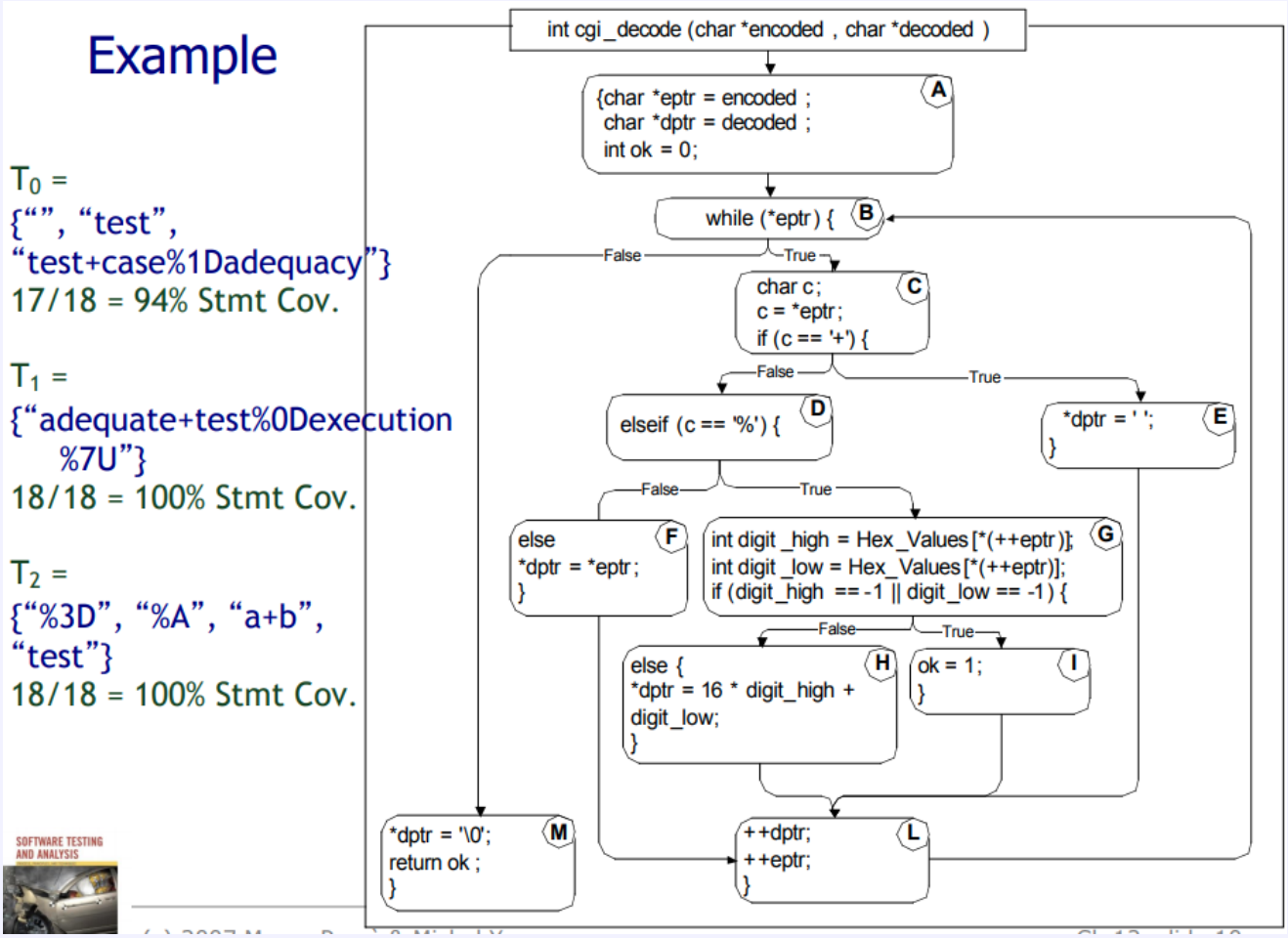
Type of testing based on the structure of the program itself, still testing product functionality against the specification but the throughness measures are changed.
Ansers the question "what is missing in our test suite?" - If a part of a program is not executed by any test case in the suite, faults in that part cannot be exposed - different types of structural testing define **part** differently.

- \- Executing all control flow elements does not guarantee fidning all faults - might depend on state
- \+ Increases confidence in thoroughness of testing, removes obvious inadequacies
- \+ Coverage criteria can be automated fully

Statement Testing

Adequacy criterion: each statement (or node in CFG) must be executed at least once

$$\frac{\textit{executed statements}}{\textit{statements}}$$



- can miss some cases (if branches without an else case for example)

Note: no essential differences between statement granuality, 100% CFG node coverage iff 100% statement coverage

Branch Testing

Adequacy criterion: each branch (edge in the CFG) must be executed at least once

$$\frac{\textit{executed branches}}{\textit{branches}}$$

In above example:

T_3 = {"", "+%0D+%4J"}
100% Stmt Cov. 88% Branch Cov. (7/8 branches)

T_2 = {"%3D", "%A", "a+b", "test"}
100% Stmt Cov. 100% Branch Cov. (8/8 branches)

+ subsumes statement testing

- can still miss some states (compound logical expressions such as $A|B$, can lead to both branches with only A being varied)

Basic condition testing

Adequacy criterion: each atomic condition must be executed once when it's true and once when it's false. i.e. $A|B$ requires that A,B be False and True in some test (not all combinations though).

$$\frac{\textit{truth values taken by all basic conditions}}{2 \cdot \textit{basic conditions}}$$

- does not imply branch coverage (not realy comparable)

Compound condition testing

Adequacy criterion: each compound condition must have all possible combinations of its atomic conditions executed.

$$\frac{\textit{combinations of compound conditions executed}}{\textit{compound conditions cases}}$$

+ subsumes branch testing

- 2^n cases arising from each condition with n atomic conditions

Modified condition/decision (MC/DC)

Adequacy criterion: all **important** combinations of conditions are tested, i.e. each basic condition shown to independently affect the outcome of each compound condition.

- For each basic condition C, two test cases are required
- In each case, the values of all the other basic conditions must stay the same, and only C must vary
- The whole condition must evaluate to true in one case, and false in another

+ n+1 test cases for n basic conditions (for minimal test suite)

+ subsumes statement, basic condition and branch testing

+ good balance of thoroughness and test size (widely used)

`(((a || b) && c) || d) && e`

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

Path adequacy

Adequacy criterion: each path must be executed at least once. I.e. a path is just a walk over the CFG (along the directed edges)

$$\frac{\textit{executed paths}}{\textit{paths}}$$

- number of paths is infinite if any loops are present in the CFG

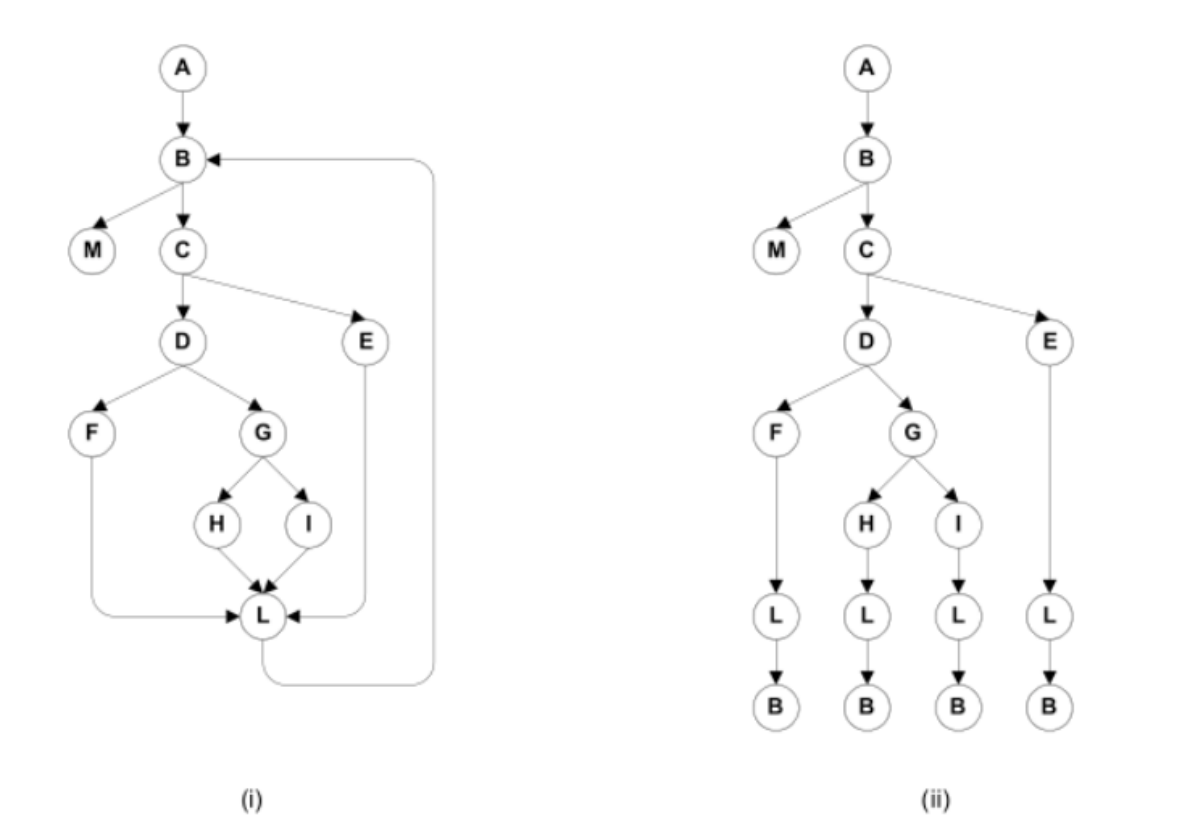
- usually impossible to satisfy

+ very thorough

Boundary interior path testing

Adequacy criterion: Requires the set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage.

$$\frac{paths}{unique\ paths\ reaching\ a\ leaf}$$



- Still number of leaf paths can grow exponentially (parallel if statements N statements = 2^N paths that must be traversed)
- some paths might not be reachable since some conditions are not independent
+ more managable

Loop boundary adequacy

Adequacy criterion: for every loop:

- In at least one test case, the loop body is iterated zero times
- In at least one test case, the loop body is iterated once
- In at least one test case, the loop body is iterated more than once

LCSAJ Adequacy

Adequacy criterion: A test suite satisfies LCSAJ adequacy, if all the LCSAJ segments are executed in some test.

$$\frac{LCSAJ's\ executed}{allLCSAJ's}$$

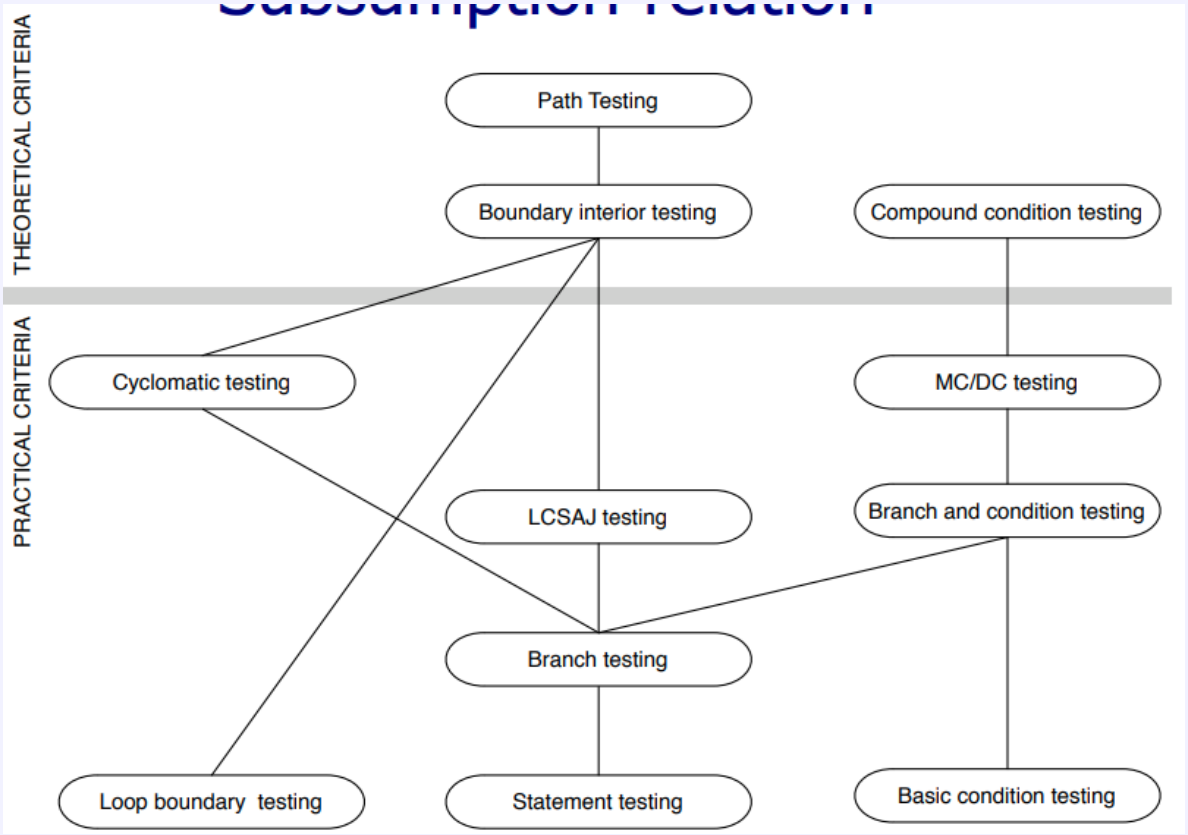
LCSAJ stands for Linear Code Sequence and Jump, i.e. a node in the CFG followed by a branch.
Various "thoroughness" levels:

- TER_1 = statement coverage
- TER_2 = branch coverage
- TER_{n+2} = coverage of n consecutive LCSAJs

Cyclomatic adequacy

Adequacy criterion: number of "linearly independent" paths (if you treat each node as an entry in a vector signifying the presence of an edge in the given path) is equal to cyclomatic number of the CFG.
The cyclomatic number can be calculates as: $e - n + 2$

Subsumption relation



Data Flow Models

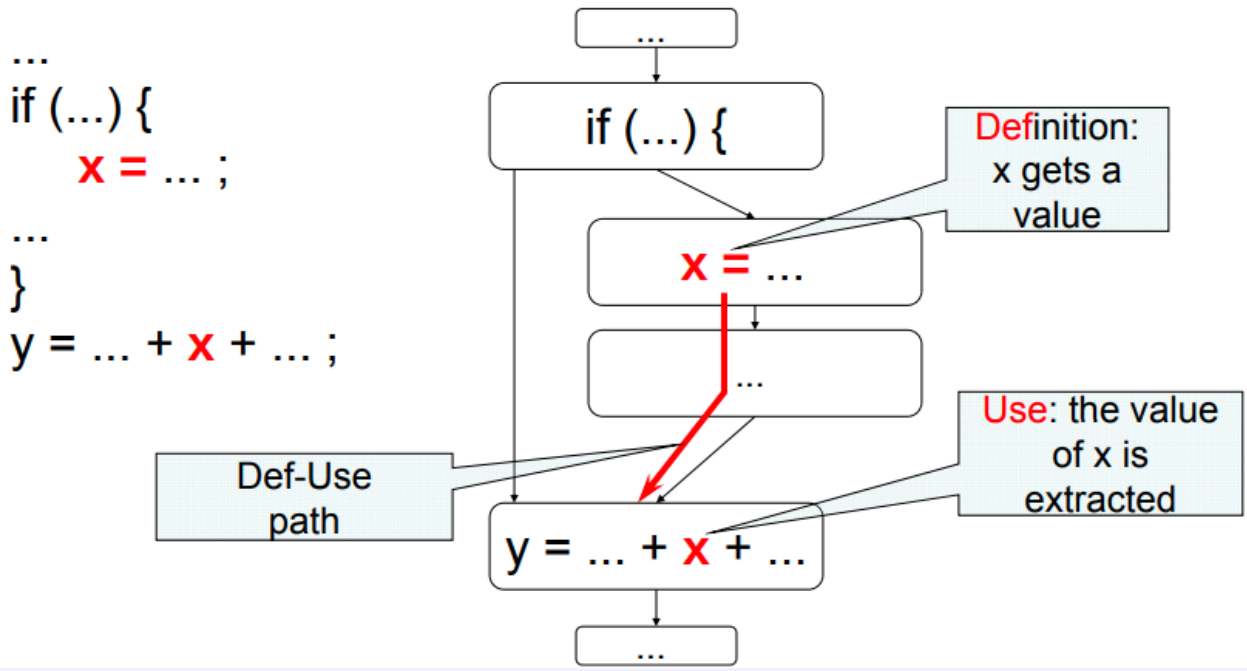
Def-Use Pairs

Associates a point in a program where a value is produced with a point where it is used.
Definition - where a variable gets a value assigned:

- Variable declaration
- Variable initialization
- Assignment
- Values received by a parameter

Use - extraction of a value from a variable:

- Expression
- Conditional statement
- Parameter passing
- Return

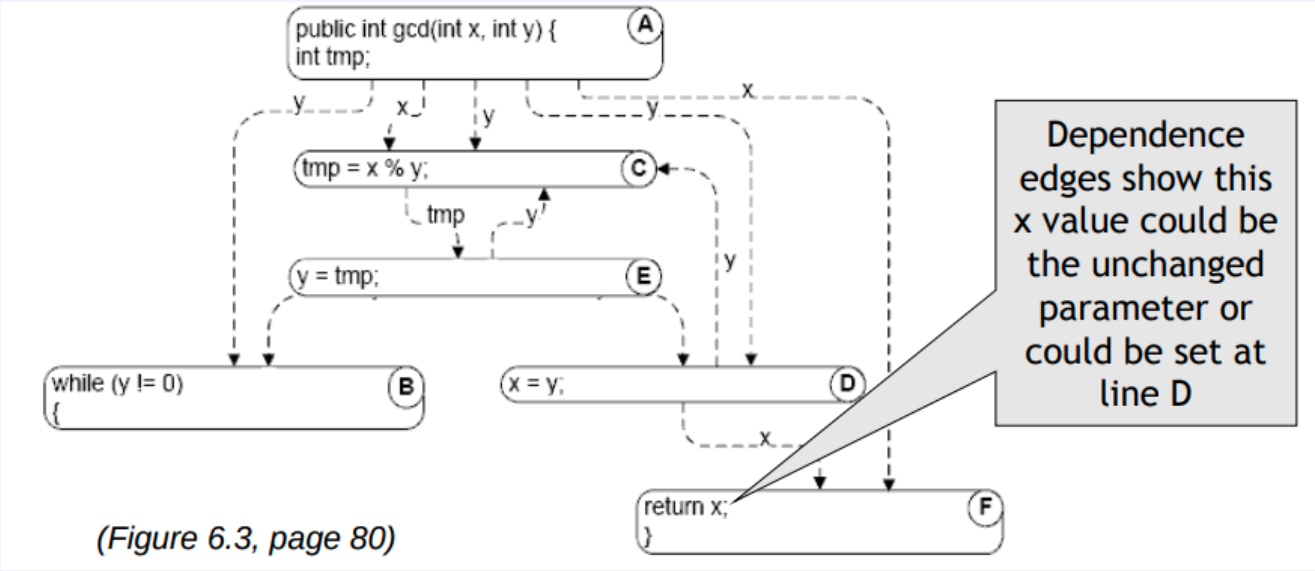


Definition-Clear path

A definition-clear path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between. (If such a definition occurs, it is said to **kill** the former definition)
Def-use pairs are formed iff there exist definition-clear paths between the definition and the use of the variable.

Data dependence graph

Nodes just like in a CFG, but edges represent def-use relationships:



Calculating def-use pairs

There is an association (d,u) between a definition of variable v at d and its use at u iff:

- There is at least one control flow path from d to u
- With no intervening definition of v
- v_d reaches u (v_d is a reaching definition at u)
- If a control flow path passes through another definition e of the same variable v, v_e kills v_d at that point

Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges. In practice, don't search all paths, summarize the reaching definitions at a node over all the paths reaching it.

DF Algorithm

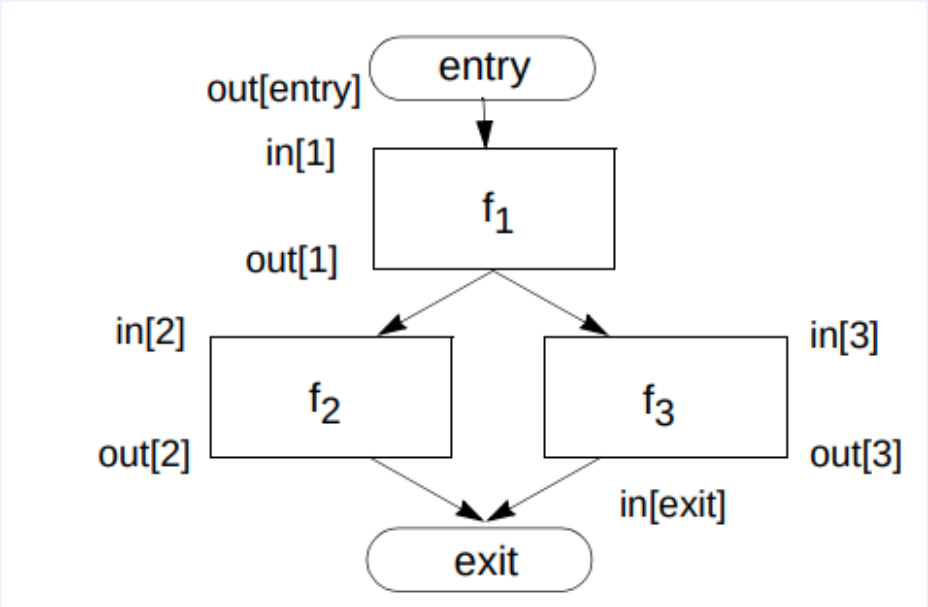
The data flow algorithm is a flexible tool which can be applied over any CFG's to find out some properties of the given graph.
The algorithm is based on the following set of equations, where b is some block of code (usually enough to carry out analysis on granuality of CFG node's or the edges of those nodes):

$$out_b = trans_b(in_b) \tag{1}$$
$$in_b = join_{p \in pred_b}(out_p) \tag{2}$$

Here $trans_b$ is the **transfer** function, of the block b. It works on the entry state in_b yielding the exit state out_b. The *join* operation combines the exit statuses of the predecessors p, of b, yielding the **entry** state of b.
I.e. join works out the state of some "data-based property" at the start of our block, by looking at the predecessor nodes, whereas the transfer function, works out how this property gets altered by the current block.
Each particular type of data-flow analysis has its own specific transfer function and join operation. Some require backward analysis which is very similar except the transfer function is applied to the exit state, and yields the entry state, while the join operation works on the entry states of the successors to yield the exit state, i.e.:

$$out_b = join_{s \in succ_b}(in_s) \tag{3}$$
$$in_b = trans_b(out_b) \tag{4}$$

The **entry point** (in forward flow, backward flow this would be the **exit point**) plays an important role, since it has no predecessors, its entry state is well defined at the start, if there are no cycles (no loops), solving the equations is straight forward, since we can simply work out the out states of nodes in **topological order** of the CFG.
If **cycles are present**, an iterative approach is required



Reaching definition analysis

Reaching definition analysis is a type of forward analysis which calculates for each program block, a set of definitions which may potentially reach this program point
The DF equations for this analysis are:

$$out_b = gen(b) \cup (in_b - kill(b)) \tag{5}$$

$$in_b = \bigcup_{p \in pred_b} (out_p) \tag{6}$$

in other words:

$$trans_b = gen(b) \cup (in_b - kill(b)) \tag{7}$$

$$join_b = \bigcup \tag{8}$$

where:

$$gen(b) = set\ of\ definitions/modifiations\ locally\ available\ in\ block$$

$$kill(b) = set\ of\ definitions,\ killed\ by\ definitions\ in\ the\ block\ (not\ locally\ available,\ but\ in\ the\ rest\ of\ the\ program)$$

Available expression analysis

A forward analysis, which determines the set of available expressions, i.e. those which need to be recomputed (in compiler design) at a program point. An expression is available if the operands of the expression are not modified on any path from the occurence of that expression to the program point. This analysis uses the following DF equations:

$$out_b = gen(b) \cup (in_b - kill(b)) \tag{9}$$

$$in_b = \bigcap p \in pred_b(out_p) \tag{10}$$

in other words:

$$trans_b = gen(b) \cup (in_b - kill(b)) \tag{11}$$

$$join_b = \bigcap \tag{12}$$

where:

$$gen(b) = set\ of\ expressions\ computed\ at\ b$$

$$kill(b) = set\ of\ expressions\ which\ whose\ variables\ are\ modified\ at\ b$$

Liveness analysis

An example of **backward** analysis, which finds the set of live variables, i.e. those which hold a value which may be needed in later parts of a program at point p.
in the example below:
b = 3; c = 5; a = f(b * c);
the set of live variables after line 2 is {b,c}, since they are used in the next line, and after line 1, this set is only {b}
the DF equation used in this analysis are as follows:

$$out(b) = \begin{cases} \bigcap s \in succ_b(in_s) & if\ b \neq exit \\ \emptyset & otherwise \end{cases} \tag{13}$$

$$in(b) = gen(b) \cup (out(b) - kill(b)) \tag{14}$$

where:

$$gen(b) = set\ of\ variables\ that\ are\ used\ in\ b,\ before\ any\ assignment$$

$$kill(b) = set\ of\ variables\ that\ are\ modified\ in\ s$$

Iterative solution of dataflow equations

Cycles in a CFG complicate dataflow analyses, and necessitate a more complex approach. We can deal with cycles with the following iterative algorithm:

1. Approximate the *in* state of **each** block (for $\bigcap$ transfer functions, first guess is the union of all "gen" sets, for analyses using $\bigcup$ first guess is the empty set, i.e. the most **conservative** option).
2. Compute the *out* states bu applying the transfer function on the *in* states.
3. Now update the *in*-states are updated with the join operations
4. Repeat steps 2-3 untill reaching a **fixpoint**, i.e. the point of convergence in which the in-states do not change (and so the out-states)

The order in which we update the nodes matters, usually apply the most **natural** order, i.e. in forward analysis, you'd use **reverse postorder** - visit a node before any of its successors, unless when the successor is reached by a back edge (think loop start nodes). Or in the case of backward analysis, you'd want to use **postorder** i.e. visit a node after all its successor nodes have been visited (DFS)

Classification of analyses

DF algorithm analyses can be classified based on:

- if a node's set depends on that of its predecessors/successors (**Forward or backward**)
- if a node's set contains a value iff it is coming from any/all of its inputs (**any/all path**)

-	Any-path ($\bigcup$)	All-paths ($\bigcap$)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

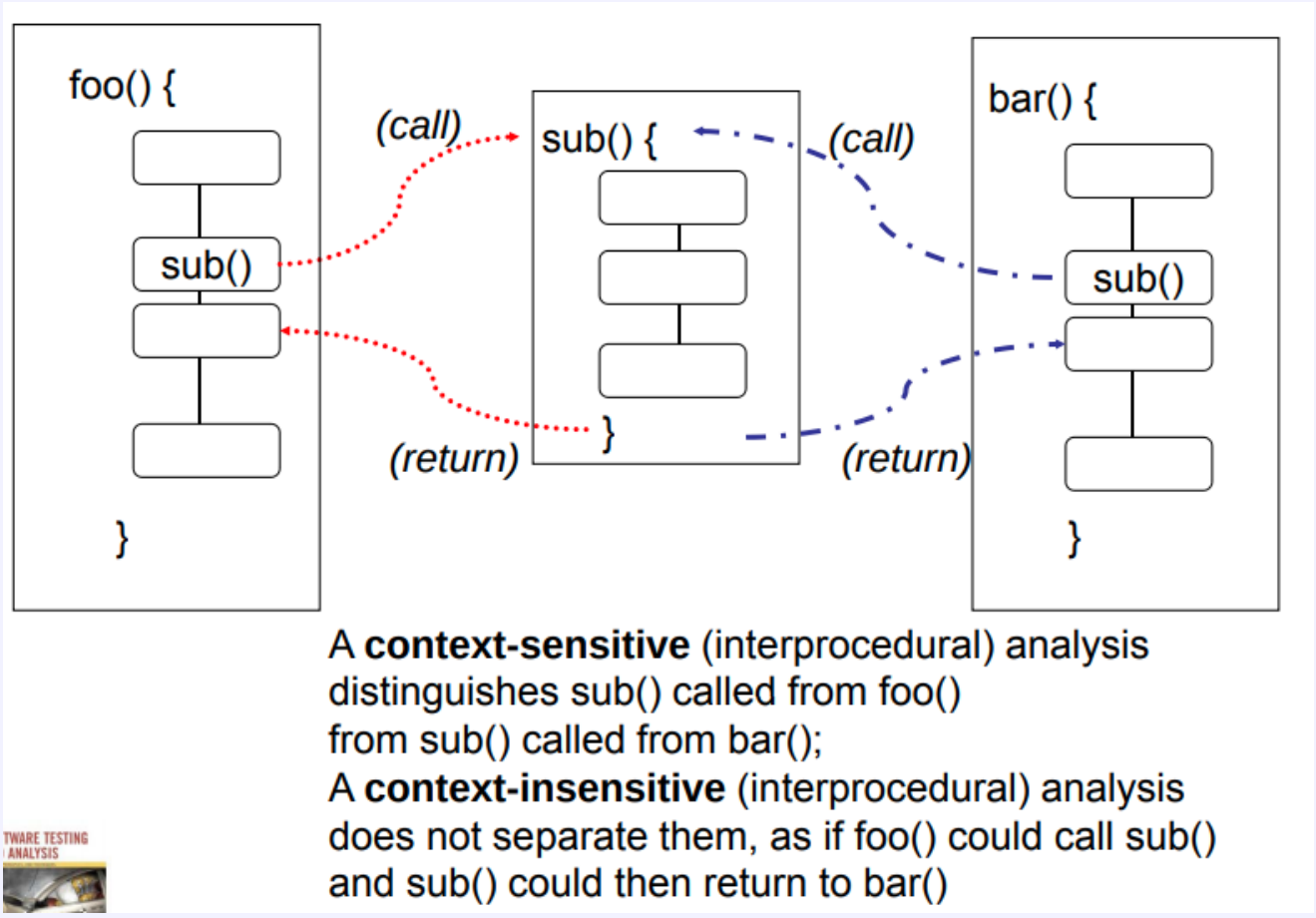
Scope of DF analysis

- Intra-procedural - within a single method or procedure
- Inter-procedural - across severall methods or classes

Cost/precision trade-offs for inter-procedural analysis are critical, and difficult. Two main trade-offs:

- Context-sensitivity
- Flow-sensitivity

Context sensitivity



Flow sensitivity

Reach, Avail etc. were all **flow-sensitive**:

- They considered ordering and control flow decisions
- Within a single procedure or method, this is fairly cheap: $O(n^3)$ for n CFG nodes
- Many inter-procedural flow analyses are **flow-insensitive**
- $O(n^3)$ would not be acceptable for all the statements in a program
- Often flow-insensitive is good enough (i.e. type checking)

Summary

DF algorithms:

+ Can be implemented by efficient iterative algorithms
+ Widely applicable (not just for "classic data flow" properties)
- Unable to distinguish feasible from infeasible paths
- Analysies spanning whole programs (e.g. alias analysis) must trade off precision against computational cost

Data Flow Testing

Def-Use pair testing

Adequacy criteria: Each DU **pair** is exercised by at least one test case

Def-Use path testing

Adequacy criteria: Each DU **clear path** (simple and non looping) is exercised by at least one test case

Def testing

Adequacy criteria: Each Definition has at least one test case which exercises a DU pair containing that definition

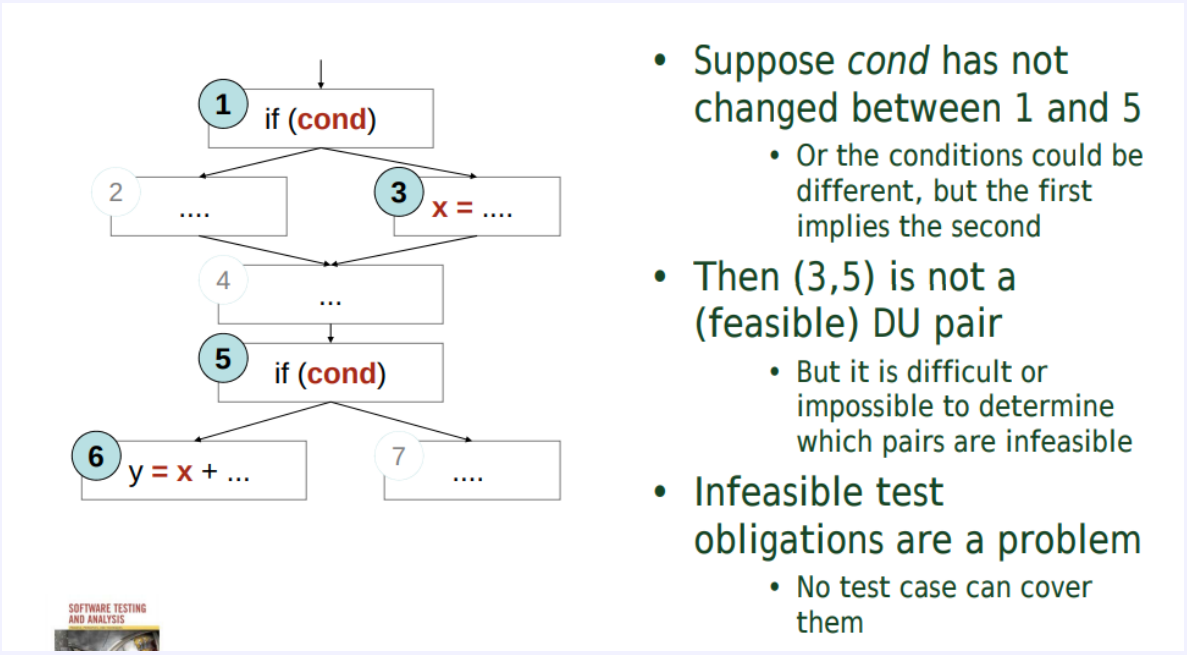
DF with complex structures

Problem of aliases - which references are (always or sometimes) the same?
Arrays and pointers are critical for data flow analysis:

- Under-estimation of aliases may fail to include some DU paris
- Over-estimation, on the other hand, may introduce unfeasible test obligations

For testing, it may be preferable to accept under-estimation

Infeasibility



- In practice, reasonable coverage, is often but not always achievable
- All DU paths is more often impractical

Test Selection and Adequacy

Test suite adequacy

A real way of measuring effective testing is impossible, provably undecidable.
What we can do is define **inadequacy** criteria:

- If the specification describes different treatment in two cases, but the test suite does not check that the two cases are in fact treated differently, we may conclude that the test suite is inadequate to guard against faults in the program logic
- If no test in the test suite executes a particular program statement, the test suite is inadequate to guard against faults in that statement.
- If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite
- If a test suite satisfies all the obligation by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidecnce of its thorougness

Unsatisfiability

Sometimes no test suite can satisfy a criterion for a given program (defensive programming - can't happen checks for example)
We can either exclude unsatisfiable obligations from the criterion or measure the extent to which a test suite approaches an adequacy criterion (percentage).
An adequacy criterion is satisfied or not, acoverage measure is the fraction of satisfied obligations

Comparing criteria

We can try to distinguish stronger from weaker adequacy criteria.
Best way to do that is by describing conditions under which one adequacy criterion is provably stronger than another - i.e. gives stronger guarantees

Subsumption

Test adequacy criterion A subsumes B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.
Example: branch coverage subsumes statement coverage

Uses of adequacy criteria

We shouldn't blindly rely on coverage criteria as infallible measure of success of a test suite. Ideally use it to guide improvement to a test suite, not as the primary motivation behind test design!

Mutation Testing

Mutation testing

Adequacy criteria: A test suite satisfies the criteria, iff every generated mutant is killed by the suite.
I.e. a mutant is a program with a small random modification. A mutant is killed if the test suite fails on incorrect behaviour introduced by the mutant. Mutants might produce equivalent behaviour, which is not useful from a testing perspective. Want mutants which should mess up the program in weird ways

+ measures quality of test cases nicely
+ provides tester with a clear target (mutants to kill)
- computationally intensive, a lot of mutants
- equivalent mutants are a practical problem - determining which mutants are equivalent is un-decidable!
- really only useful at unit testing level

Mutation operators

predefined program modification rules corresponding to a fault model.
Ideally we would like mutation operators which are representative of all realistic types of faults that could occur in practice (be made by a real programmer).
In general operator set is large, and can capture all syntactic variations in a program.

Mutation types

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

Mutation coverage

- Complete coverage equals the killing of all non-equivalent mutants (or random sample, not necessarily all possible modifications)
- The amount of coverage is called the **mutation score**
- The number of mutants depends on definition of mutation operators and structure of software
- Random sampling used since this set of mutants can be large.

Assumptions

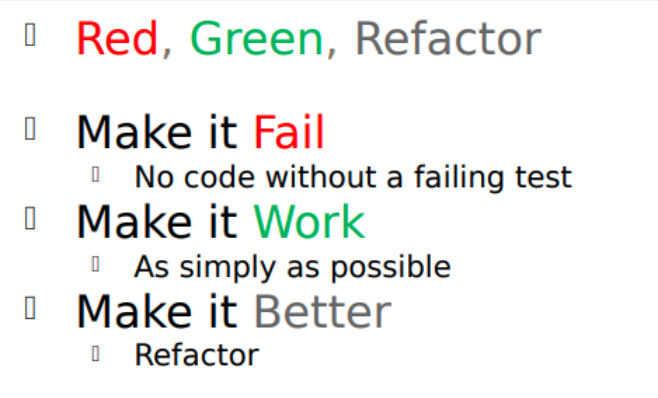
- Competent programmer assumption - they write programs which are nearly correct
- Coupling effect assumption - test cases that distinguish all programs differing from a correct one by only simple errors are so sensitive that they also implicitly distinguish more complex errors

Some empirical evidence of the above

Test Driven Developement

TDD

Never write a single line of code unless you have a failing automated test.



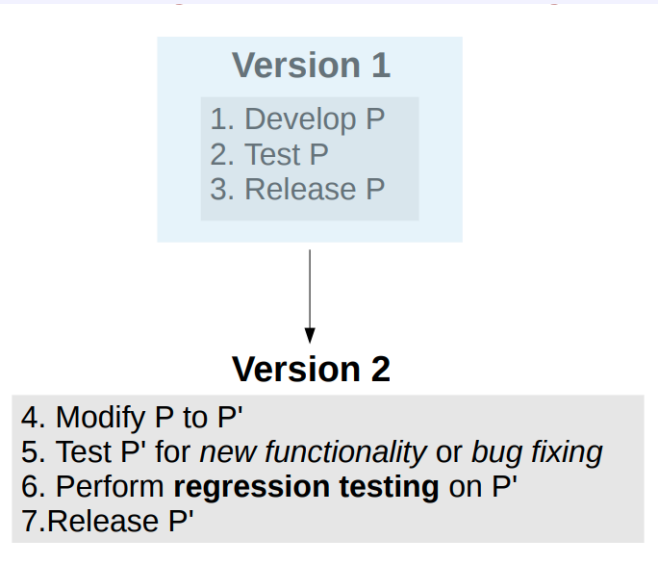
Summary

+ Confidence in change - can change code without shitting your pants
+ Documents requirements via tests
+ Discovers usability issues or problems with requirements early
+ Regression testing = stable software = quality software
+ Major quality improvement for minor time investment
- Programmers like to code, not test
- Test writing is consuming
- Test completeness is difficult to judge
- May not always be suitable
• Only really relevant to Functional testing

Regression Testing

Regression testing

Testing which tries to identify **regressions** in code, i.e. breaking changes introduced by modifications made to code.



Tests must be re-run after any change:

- Adding new features
- Changing or adapting software to new conditions
- Fixing bugs

Otherwise code may break.

- takes a long time, after each change new tests pile up
- difficult to tell which test cases should be removed or replaced/added
- it often takes a while to run all the tests

Test case maintenance

Some maintenance is inevitable, Some maintenance should be avoided. Test suites should be modular.
Obsolete - test cases which are obsolete are no longer valid and should be removed **Redundant** - test cases which do not contribute significantly to the test suite (perhaps with coverage) may be removed since they're inlikely to find faults missed by similar test cases

Test optimisation

- Re-test all - always running all tests in the test suite. Good when you want 100% certainty the new version works on all tests developed for previous
- Test selection - Only select subset of test cases whose execution is relevant to changes - a test case cannot find a fault in code it doesn't execute - only execute test cases that execute changed or new code
- Test set minimisation - Identifying and removing redundant test cases to reduce test suite's size
- Test set prioritisation - Sort test cases in order of increasing cost per additional coverage - run n tests in that order untill max cost is reached

Control-flow and Data-flow Test Selection

- Re-run test cases only if they include changed elements
- Elements may be modified control flow nodes and edges, or DU pairs in data flow
- Record elements touched by each test case in database to automate this process

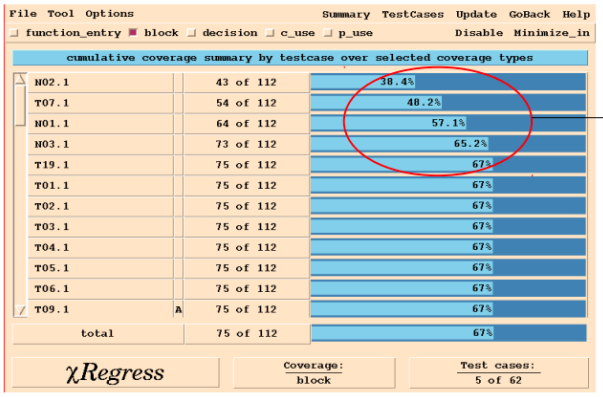
Specification based test selection

- Pick test cases that test new and changed functionality
- No guarantee of independence - a test case that isn't for changed or added feature X might find a bug in feature X anyway
- Typical approach: execute all tests, but start with those that related to changed and added features by specification

Test set minimization

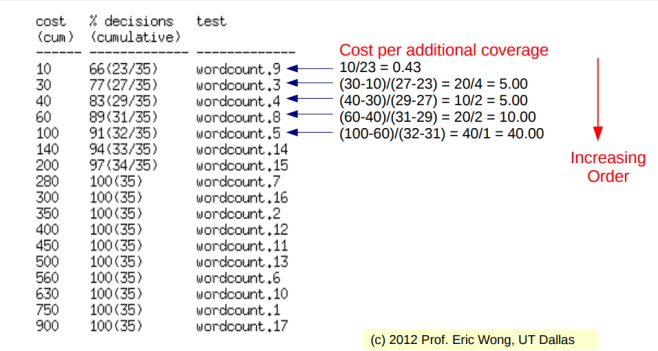
- Maximize coverage with minimum number of test cases
- The minimization algorithm can be exponential in time
- Does not occur in our experience
- Stop after a pre-defined number of iterations
- Obtan an approximate solution by using a greedy heuristic

Sort test cases in order of increasing cost per additional coverage



Only 5 of the 62 test cases are included in the minimized subset which has the same block coverage as the original test set.

Test set prioritisation



- Execute all test cases, eventually
- Execute some sooner than others
- Possibly prioritize using RR scheduling, Run tests which detected more faults in the past first, or based on the structure of the tests or code

Security Testing

Security testing

- Normal testing aims to ensure program meets customer requirements in terms of features and functionality
- Security testing aims to ensure the program fulfills security requirements - more interested in misuse cases, weird "corner" cases

Common security testing approaches

- Test for known vulnerability types
- Attempt directed or random search of program state space to uncover the corner cases

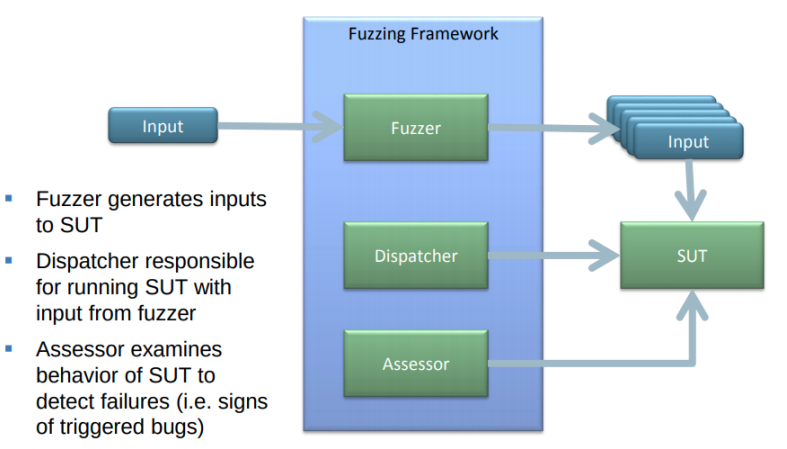
Penetration testing

- Manually try to "break" software
- Relies on human intuition and experience
- Typically involves looking for known common problems
- Can uncover problems that are impossible or difficult to find using automated methods - results completely dependent on skill of tester

Fuzz testing

idea: send semi-valid input to a program and observe behaviour.

- Black-box testing - System treated as a "black box"
- The only feedback is the output and/or externally observable behaviour
- First proposed in a 1990 paper where completely random data was sent to 85 common Unix utilities in 6 different systems. 24-33% crashed.
- Remember: crash implies memory protection errors, often signs of exploitable flaws in the program



- Simplest method: completely random imput - wont work well in practice - input deviates too much from expected format, rejected early in processing
- Mutation based fuzzing
- Generation based fuzzing

Mutation based fuzzing

Start with valid seed input, and mutate it:

- Flip some bits, change value of some bytes
- Programs that have highly structured input, e.g. XML, may require ”smarter” mutations

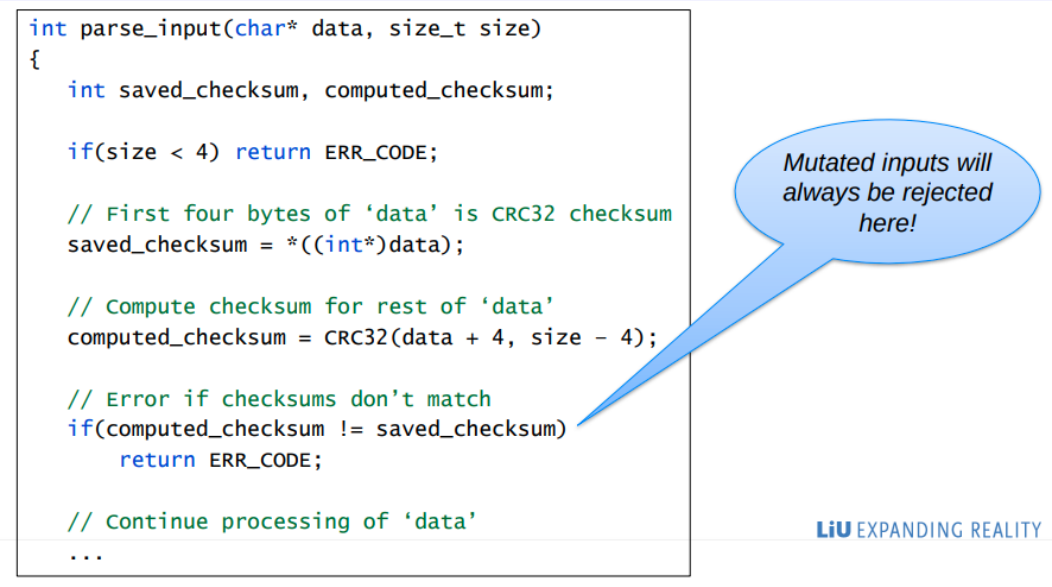
How do we select appropriate input?:

- If official test suites are available, these can be used
- Generally mostly used for programs that take files as input
- Trickier to do when interpretation of inputs depends on program state, e.g. network protocol parsers

\+ easy to get started - little knowledge required of specific input format needed

\- typically yields low code coverage, inputs tend to deviate too much from expected format - rejected early by sanity checks

\- hard to reach ”deeper” parts of program by random guessing (i.e. unlikely to guess magic constants correctly in switch cases etc.)



Generation based fuzzing

Idea: use a specification of the input format (a grammar or similar) to automatically generate semi-valid inputs. Usually combined with various fuzzing heuristics that are known to trigger certain vulnerability types:

- Very long strings, empty strings
- Strings with format specifiers, ”extreme” format strings
- Very large or small values
- Negative values where positives are expected

\+ Input is much closer to the expected, much better coverage

\+ can includ emodels of protocol state machines to send messages in the sequence expected by SUT

\- Requires input format to be known

\- May take considerable time to write the input format grammar/specification.

Fuzzing: Dispatcher

Repsponsible for running the SUT, on each input generated by fuzzer

- Must provide suitable environment for SUT
- SUT may modify environment (might require a VM)

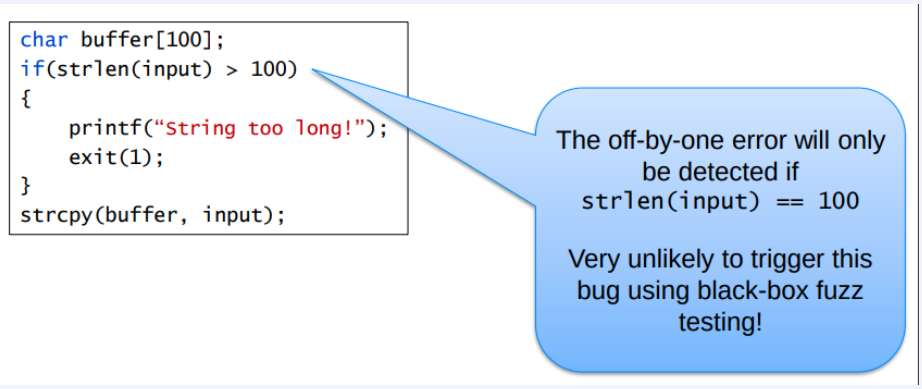
Fuzzing: The Assessor

Must automatically assess observed SUT behaviour to determine if a fault was triggered:

- For C/C++ programs: monitor for memory access violations, e.g. out of bounds reads or writes
- Simplest method: check if crashed
- Problem: SUT may catch signals/exception to gracefully handle e.g. seg faults which makes it difficult to tell if a fault has occured
- solution is to attach a programmable debugger to SUT
- This can catch signals/exceptions prior to being delivered to application
- Can help manual diagnosis of detected faults by recording stack traces, value of registers etc.
- Not all result in failures though
- An out-of-bound read/write or use-after-free may e.g. not result in a memory access violation.
- Solution: Use a dynamic-analysis tool that can monitor what goes on "under-the-hood" (this is very slow)

Limitations of Fuzz testing

- Many programs have an infinite input space, and state space - combinatorial explosion!
- Conceptually simple idea, but many practical challenges
- Difficult to create a trully generic fuzzing framework that can cater for all possible input formats - might often require a custom fuzzer for each SUT
- Semi-randomly generated inputs are very unlikely to trigger certain faults
- mutation based fuzzing can typically onl find "low hanging fruit" - shallow bugs which are easy to find
- Generation-based fuzzers are much better but also require much more effort



Concolic testing

idea: combine concrete and symbolic execution

- Execute program for real on some input, record path taken
- Encode path as query to SMT solver and negate one branch condition on the path
- ask the solver to find new satisfying input that will give a different path

Reported bugs are always accompanied by an input that triggers the bug - reported bugs are always real bugs

Challenges of concolic testing

Number of paths increases exponentially with number of branches:

- Most real-world programs have an infinite state space
- Number of loop iterations may depend on size of input
- Not possible to explore all paths - need strategy to explore interesting parts of the program
- DFS will easilly get stuck in one part of the program (a loop)
- BFS will take a very long time to reach deep states
- Need to try smarter ways of exploring program state space

Generational search (”whitebox fuzzing”)

- Run program on concrete seed input and record path constraint
- For each branch condition:
  - Negate condition and keep the earlier conditions unchanged.
  - Have SMT solver generate new satisfying input and run program with that input
  - Assign input a score based on how much it improves coverage
  - Store input in a global worklist sorted on score
- Pick input at head of worklist and repeat

Coverage based heuristics avoid getting ”stuck” like DFS. If one test case executes the exact same code as in a previous test case, its score will be 0, so it will probably never be used again. Gives sparse but more diverse search of the paths of the program.

- Implements a form of greedy search heuristic
- For example, loops may only be executed only once
- Only generates input ”close” to the seed input
- Will try to explore weird corner cases in code exercised by seed input
- Choice of seed input important for good coverage

\+ Proven to work well in practice

\+ Source code not needed, since can be done at machine code level,

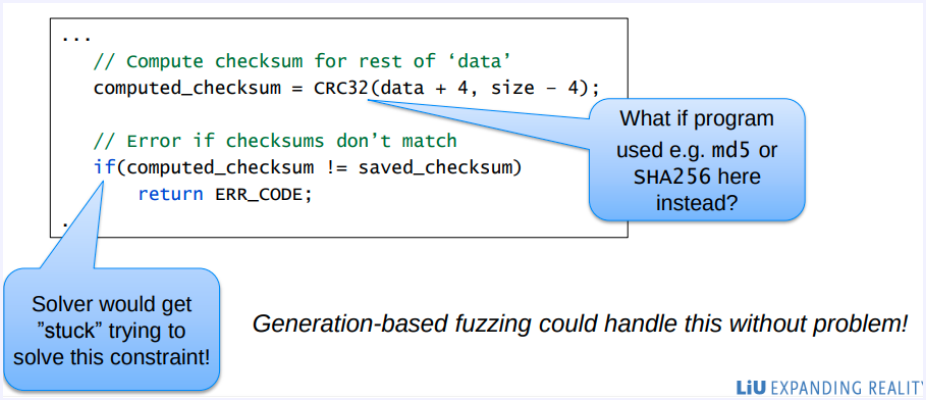
\- Sacrifices some coverage due to additional approximations needed when working on machine code

\- Solving SAT/SMT problems is NP-complete

\- may take unacceptably long time to solve certain constraints

\- Solving SMT queries with non-linear arithmetic (multiplication, division, modulo etc) is undecidable in general - but decidable in fixed-precision data types which are typically used

\- any usage of cryptographical functions usually ends in failure of symbolic execution



Greybox fuzzing

Probability of hitting a "deep" level of the code decreases exponentially with depth of code for mutation based fuzzing
Similarly, the time required for solving an SMT query is high, and increases exponentially with depth of path constraint
Black-box fuzzing is too "dumb" and whitebox fuzzing may be "too smart"
idea: find spot in-between.
Instead of recording full path constraint, record light-weight coverage information to guide fuzzing. Use evolutionary algorithms to learn input format

- AFL is considered the current state-of-the-art in fuzzing
- Performs regular mutation based-fuzzing (with several strategies) and measures code coverage
- Every generated input that resulted in any new coverage is saved and later re-fuzzed
- This extremely simple algorithm allows AFL to learn how to reach deeper parts of the program
- Also highly optimized for speed - can reach several thousand test cases per second
- Often beats smarter and slower methods like whitebox fuzzing
- Found hundreds of serious vulnerabilities in open-source programs

Integration and Component Based Testing

Integration testing

Integration testing serves as a process check:

- If module faults are revealed in integration testing, they signal inadequate unit testing
- If integration faults occur in interfaces between correctly implemented modules, the errors can be traced to module breakdown and interface specifications

	Module test	Integration test	System test
Specification:	Module interface	Interface specs, module breakdown	Requirements specification
Visible structure:	Coding details	Modular structure (software architecture)	— none —
Scaffolding required:	Some	Often extensive	Some
Looking for faults in:	Modules	Interactions, compatibility	System functionality

SOFTWARE TESTING AND ANALYSIS

Integration Faults

- Inconsistent interpretation of parameters of values (mixed units)
- Violation of value domains, capacity, or size limits (buffer overflows)
- Side effects on parameters or resources (conflicts on temp files)
- Omitted or misunderstood functionality (inconsistent interpretation of web hits)
- Nonfunctional properties (Unanticipated performance issues)
- Dynamic mismatches

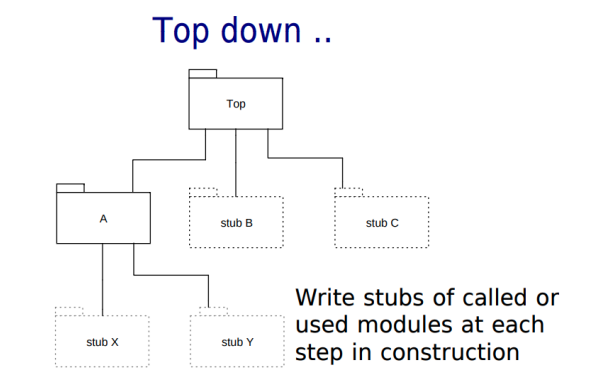
Big bang integration test

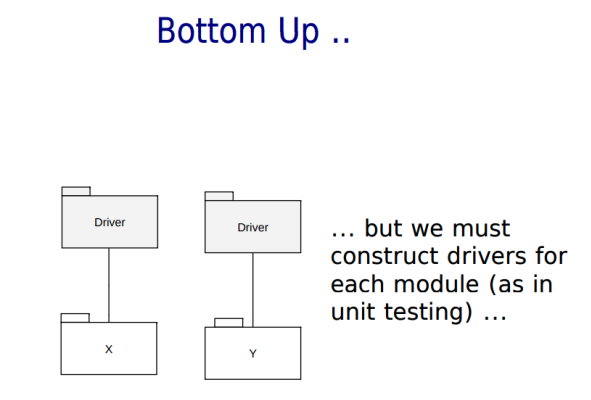
Test only after integrating all modules:

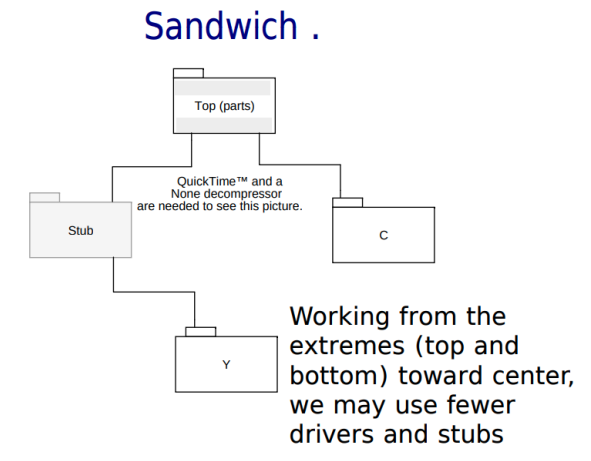
+ Does not require scaffolding
- Minimum observability, diagnosability, efficacy, feedback
- High cost of repair

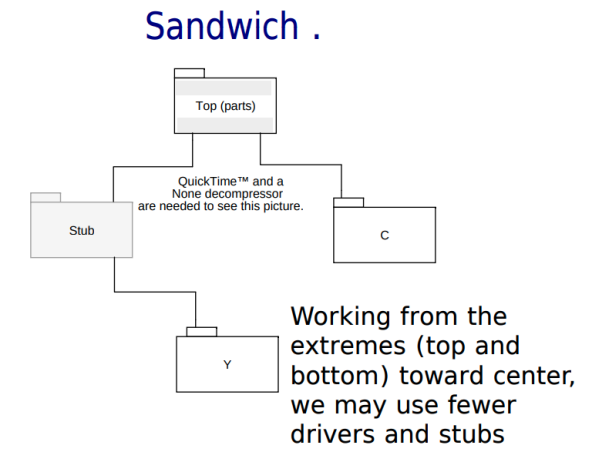
Structural and functional strategies

Structural orientation: modules constructed, integrated and tested based on a hierarchical project structure (top-down, bottom-up, sandwich etc)

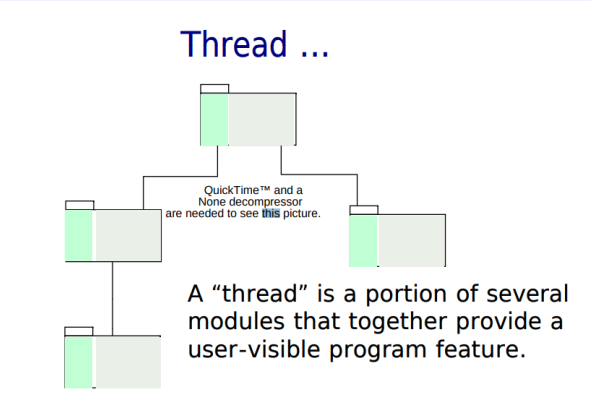


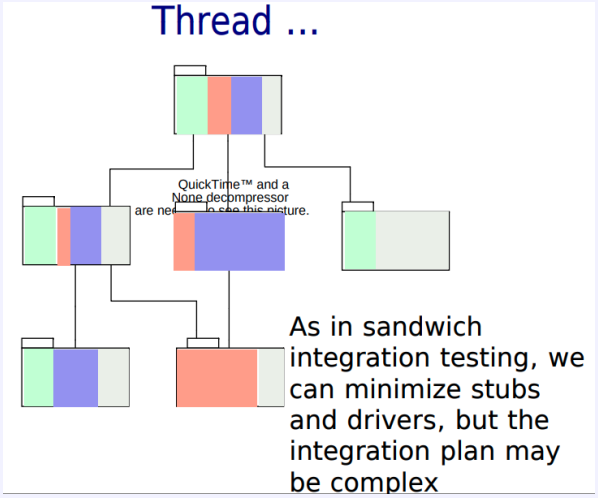






Functional orientation: modules integrated according to application characteristics or features (threads, critical module)





Critical Modules

Strategy: start with riskiest modules:

- Risk assesment necessary first step
- May include technical risks (is X feasible), process risks (is schedule for X realistic) or other risks
- May resemble thread or sandwich process in tactics for flexible build order
- Key point is risk-oriented process (integration as risk-reduction activity, deliver bad news early)

Choosing a strategy

- Functional strategies require more planning
- but also provide better process visibility, especiall in complex systems
- Possible to combine: top-down, bottom-up or sandwich are reasonable for relatively small components and subsystems
- Combination of threads and critical modules integration testing are often preferred for larger subsystems

Component

Reusable unit of deployment and composition;

- Deployed and integrated multiple times
- Integrated by different teams (usually)
- Characterised by an interface or contract
- Often larger grain than objects or packages (a database system, or geometry module)

Component interface contracts

- API is distinct from implementation
- DOM interface for XML is distinct from possible implementations
- Interface includes everything that must be known to use the component
- More than just method signatures, exceptions etc.
- May include non-functional characteristics like performance, capacity and security
- May include dependence on other components

Challenges in testing components

- The component builder's challenge - impossible to know all the ways a component may be used
- Difficult to recognize and specify all potentially important properties and dependencies
- The component user's challenge - no visibility "inside" the component, often difficult to judge suitability for a particular use and context

Testing a Component: Producer view

- First: thorough unit and subsystem testing
- Includes thorough functional testing based on API
- Reusable component requires at least twice the effort in design, implementation and testing as a subsystem constructed for a single use
- Second: thorough acceptance testing
- Based on scenarios of expected use
- Includes stress and capacity testing

Testing a component: User view

- Not primarily to find faults in the component
- Major question: is the component suitable for this application ?
- Primary risk is not fitting the applciation context
- Unanticipated dependence or interactions with environment
- Performance or capcaity limits
- Missing functionality, misunderstood API
- Risk high when using component for first time
- Reducing risk: trial integration early
- Often worthwile to build driver ot test model scenarios, long before actual integration

System and Acceptance Testing

System and acceptance testing

	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
	Verification	*Validation*	Verification

- Comprehensive
- Based on specification of observable behaviour
- Independent of design and implementation
- Avoid repeating software design errors in system test design
- should be performed by a different organisation

Incremental system testing

- System test are often used to measure progress
- System test suite covers all features and scenarios of use
- As project progresses, the system passes more and more system tests
- Assumes a "threaded" incremental build plan: features exposed at top level as they are developed

Global properties

- Performance, latency reliability etc.
- Early and incremental testing is still necessary but provide only estimates
- A major focus of system testing
- The only opportunity to verify global properties against actual system specifications
- Especially to find unanticipated effects, e.g. an unexpected performance bottleneck

Context-dependent properties

- Beyond system-global: some properties depend on the system context and use
- Example: performance properties depend on environment and configuration
- Example: privacy depends both on system and how it is used
- Example: security depends on threat profiles (and threats change)
- Testing is just one part of the approach

Stress testing

- Often requires extensive simulation of the execution environment
- What happens when parameters are pushed ? (10,100 times more ?)
- Often requires more resources (human and machine), than typical test cases

Capacity testing

Capacity Testing

- **When:** systems that are intended to cope with high volumes of data should have their limits tested and we should consider how they fail when capacity is exceeded
- **What/How:** usually we will construct a harness that is capable of generating a very large volume of simulated data that will test the capacity of the system or use existing records
- **Why:** we are concerned to ensure that the system is fit for purpose say ensuring that a medical records system can cope with records for all people in the UK (for example)
- **Strengths:** provides some confidence the system is capable of handling high capacity
- **Weaknesses:** simulated data can be unrepresentative; can be difficult to create representative tests; can take a long time to run

Security Testing

- **When:** most systems that are open to the outside world and have a function that should not be disrupted require some kind of security test. Usually we are concerned to thwart malicious users.
- **What/How:** there are a range of approaches. One is to use league tables of bugs/errors to check and review the code (e.g. SANS top twenty-five security-related programming errors). We might also form a team that attempts to break/break into the system.
- **Why:** some systems are essential and need to keep running, e.g. the telephone system, some systems need to be secure to maintain reputation.
- **Strengths:** this is the best approach we have most of the effort should go into design and the use of known secure components.
- **Weaknesses:** we only cover known ways in using checklists and we do not take account of novelty using a team to try to break does introduce this.

Performance Testing

- **When:** many systems are required to meet performance targets laid down in a service level agreement (e.g. does your ISP give you 2Mb/s download?).
- **What/How:** there are two approaches - modelling/simulation, and direct test in a simulated environment (or in the real environment).
- **Why:** often a company charges for a particular level of service - this may be disputed if the company fails to deliver. E.g. the VISA payments system guarantees 5s authorisation time delivers faster and has low variance. Customers would be unhappy with less.
- **Strengths:** can provide good evidence of the performance of the system, modelling can identify bottlenecks and problems.
- **Weaknesses:** issues with how representative tests are.

Compliance testing

Compliance Testing

- **When:** we are selling into a regulated market and to sell we need to show compliance. E.g. if we have a C compiler we should be able to show it correctly compiles ANSI C.
- **What/How:** often there will be standardised test sets that constitute good coverage of the behaviour of the system (e.g. a set of C programs, and the results of running them).
- **Why:** we can identify the problem areas and create tests to check that set of conditions.
- **Strengths:** regulation shares the cost of tests across many organisations so we can develop a very capable test set.
- **Weaknesses:** there is a tendency for software producers to orient towards the compliance test set and do much worse on things outside the compliance test set.

Documentation testing

Documentation Testing

- **When:** most systems that have documentation should have it tested and should be tested against the real system. Some systems embed test cases in the documentation and using the doc tests is an essential part of a new release.
- **What/How:** test set is maintained that verifies the doc set matches the system behaviour. Could also just get someone to do the tutorial and point out the errors.
- **Why:** the user gets really confused if the system does not conform to the documentation.
- **Strengths:** ensures consistency.
- **Weaknesses:** not particularly good on checking consistency of narrative rather than examples.

Estimating dependability

Measuring quality, not searching for faults.

- **reliability** - survival Probability
- **Availability** - fraction of time a system meets specs
- **Failsafe** - system fails to a known safe state
- **Dependability** - system does the right thing at right time

Statistical sampling

- Need valid operational profile (model)
- sometimes from older version of system
- sometimes from operational environment
- sensitivity testing reveals parameters which are most important
- Clear definition of what is being measured
- Many random samples

System reliability

- The reliability, $R_F(t)$ of a system is the probability that no fault of the class F occurs (i.e. system survives) during time t.

$$R_F(t) = P(t_{init} \leq t < t_f \forall f \in F)$$

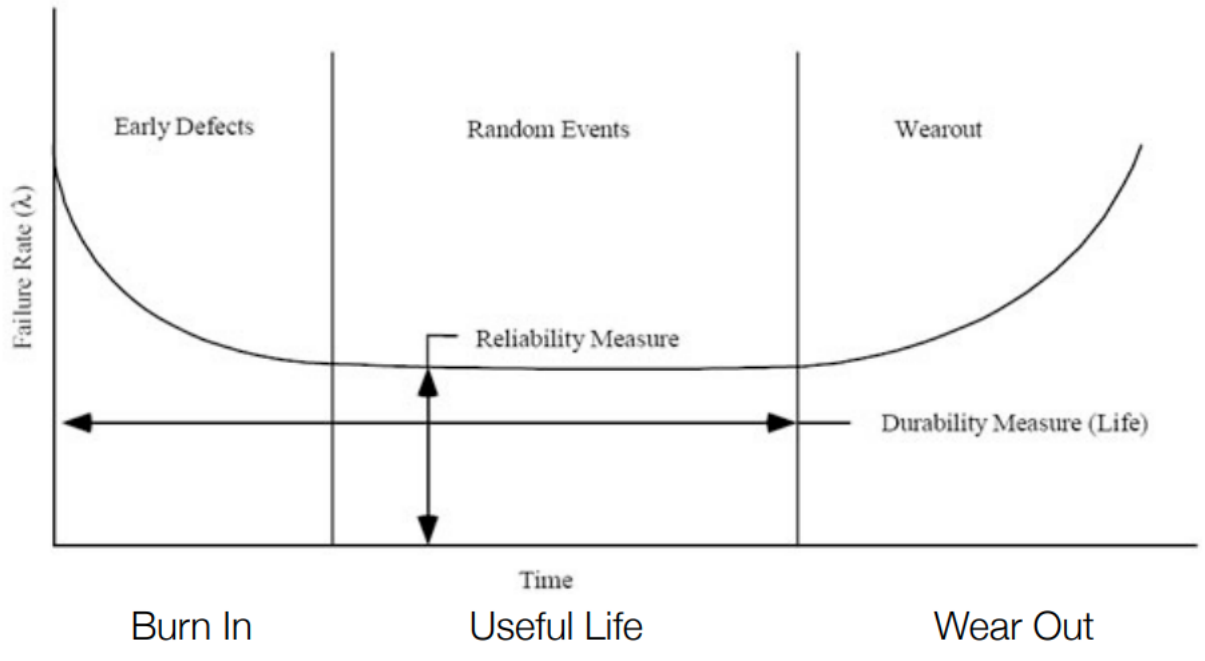
where t_{init} is time of introduction of the system to service,
t_f is time of occurrence of the first failure f drawn from F.

- Failure Probability, $Q_F(t)$ is complementary to $R_F(t)$

$$R_F(t) + Q_F(t) = 1$$

- We can take off the F subscript from $R_F(t)$ and $Q_F(t)$

- When the lifetime of a system is exponentially distributed, the reliability of the system is: $R(t) = e^{-\lambda t}$ where the parameter λ is called the failure rate



Mean time to failure (MTTF)

- **MTTF**: Mean Time to Failure or Expected Life
- **MTTF**: Mean Time To (first) Failure is defined as the expected value of t_f

$$MTTF = E(t_f) = \int_0^\infty R(t)dt = \frac{1}{\lambda}$$

where λ is the failure rate.

- **MTTF** of a system is the expected time of the first failure in a sample of identical initially perfect systems.
- **MTTR**: Mean Time To Repair is defined as the expected time for repair.
- **MTBF**: Mean Time Between Failure

Serial system reliability

- Serially Connected Components
- $R_k(t)$ is the reliability of a single component k: $R_k(t) = e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.
- The overall system reliability $R_{ser}(t)$

$$R_{ser}(t) = R_1(t) \times R_2(t) \times R_3(t) \times \ldots \times R_n(t)$$

$$R_{ser}(t) = \prod_{i=1}^{n} R_i(t)$$

- No redundancy: Overall system reliability depends on the proper working of each component $R_{ser}(t) = e^{-t(\sum_{i=1}^{n} \lambda_i)}$
- Serial failure rate $\lambda_{ser} = \sum_{i=1}^{n} \lambda_i$

Parallel system reliability

- Parallel Connected Components
- $Q_k(t)$ is $1 - R_k(t)$: $Q_k(t) = 1 - e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.

$$Q_{par}(t) = \prod_{i=1}^{n} Q_i(t)$$

- Overall system reliability: $R_{par}(t) = 1 - \prod_{i=1}^{n}(1 - R_i(t))$

Process-based measures

- Less rigorous than statistical
- Alpha - real users, controlled enviormnent
- Beta -real users (real environment)

Usability testing

Usability Testing

- **When:** where the system has a significant user interface and it is important to avoid user error — e.g. this could be a critical application e.g. cockpit design in an aircraft or a consumer product that we want to be an enjoyable system to use or we might be considering efficiency (e.g. call-centre software).
- **What/How:** we could construct a simulator in the case of embedded systems or we could just have many users try the system in a controlled environment. We need to structure the test with clear objectives (e.g. to reduce decision time,...) and have good means of collecting and analysing data.
- **Why:** there may be safety issues, we may want to produce something more useable than competitors' products...
- **Strengths:** in well-defined contexts this can provide very good feedback – often underpinned by some theory e.g. estimates of cognitive load.
- **Weaknesses:** some usability requirements are hard to express and to test, it is possible to test extensively and then not know what to do with the data.

Reliability Testing

- **When:** we may want to guarantee some system will only fail very infrequently (e.g. nuclear power control software we might claim no more than one failure in 10,000 hours of operation). This is particularly important in telecommunications.
- **What/How:** we need to create a representative test set and gather enough information to support a statistical claim (system structured modelling supports demonstrating how overall failure rate relates to component failure rate).
- **Why:** we often need to make guarantees about reliability in order to satisfy a regulator or we might know that the market leader has a certain reliability that the market expects.
- **Strengths:** if the test data is representative this can make accurate predictions.
- **Weaknesses:** we need a lot of data for high-reliability systems, it is easy to be optimistic.

Availability/Reparability Testing

- **When:** we are interested in avoiding long down times we are interested in how often failure occurs and how long it takes to get going again. Usually this is in the context of a service supplier and this is a Key Performance Indicator.
- **What/How:** similar to reliability testing – but here we might seed errors or cause component failures and see how long they take to fix or how soon the system can return once a component is repaired.
- **Why:** in providing a critical service we may not want long interruptions (e.g. 999 service).
- **Strengths:** similar to reliability.
- **Weaknesses:** similar to reliability – in the field it may be much faster to fix common problems because of learning.

Concurrency Testing

Cocnurrency bug types

- Data race - occurs when two conflicting accesses to one shared variable are executed without proper synchronisation
- Deadlock - occurs when two or more operations circularly wait for each other to release an acquired resource
- Atomicity violation bugs - bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region
- Order violation bugs - bugs that don't follow the programmer's intended order

Preventing data races

- Using Locks or Atomic operations on shared variables.

Orig. Code with data race

```
public class Counter {
   int counter;

   public void increment() {
      counter++;
   }
}
```

Data race on *counter shared variable.*
counter++ is a combination of 3 operations: reading the value, incrementing and writing the updated value.

Using Locks to prevent data race

```
public class SafeCounterWithLock {
   private volatile int counter;

   public synchronized void increment() {
      counter++;
   }
}
```

Volatile keyword for reference visibility among threads.
The **synchronized** keyword acts as a lock and ensures that only one thread can enter the method at one time.

7 / 13

Using Atomics to prevent data race

The most commonly used atomic variable classes in Java are **AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference**. These classes represent an int, long, boolean and object reference respectively which can be atomically updated. Methods exposed by these classes are *get(), set(), lazySet(), compareAndSet().*

```
public class SafeCounterWithAtomic {
   private final AtomicInteger counter =
                  new AtomicInteger(0);
   public int getValue() {
      return counter.get();
   }
   public void increment() {
         while(true) {
         int existingValue = getValue();
         int newValue = existingValue + 1;
         if(counter.compareAndSet
                  (existingValue, newValue)) {
            return;
         }
      }
   }
}
```

Deadlock example

```
thread 1                thread 2

void f1()               void f2()
{                       {
get(A);                 get(B);
get(B);                 get(A);
release(B);             release(A);
release(A);             release(B);
}                       }
```

Deadlock prevention

- **Lock Ordering:** Deadlock occurs when multiple threads need the same locks but obtain them in different order. If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur.
- **Lock Timeout:** Another deadlock prevention mechanism is to put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up. If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry. The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking.

Atomicity violation example

Case 1

```
thread-1                thread-2
deposit(int val){       deposit(int val){
   int tmp = bal;          int tmp = bal;
   tmp = tmp + val;        tmp = tmp + val;
   bal = tmp;              bal = tmp;
}                       }
```

Case 2

```
thread-1                thread-2
deposit(int val){       deposit(int val){
   synchronized(o){        synchronized(o){
     int tmp = bal;          int tmp = bal;
     tmp = tmp + val;        tmp = tmp + val;
   }                       }
   synchronized(o){        synchronized(o){
     bal = tmp;              bal = tmp;
   }                       }
}                       }
```