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

Maksymilian Mozolewski

May 10, 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

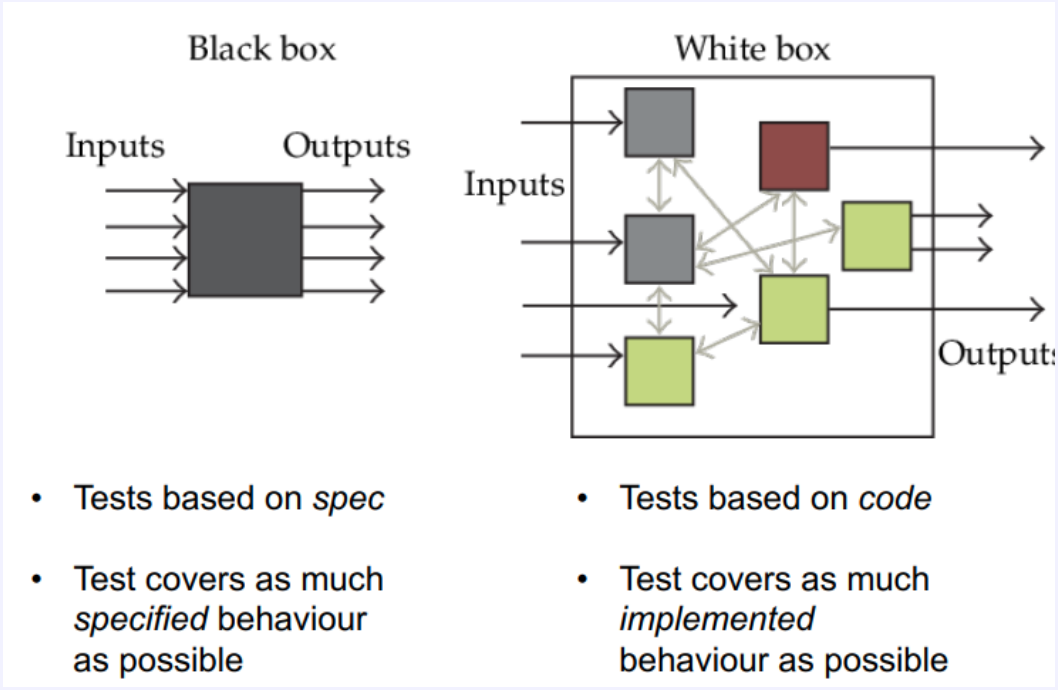
Functional Requirements:

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!)

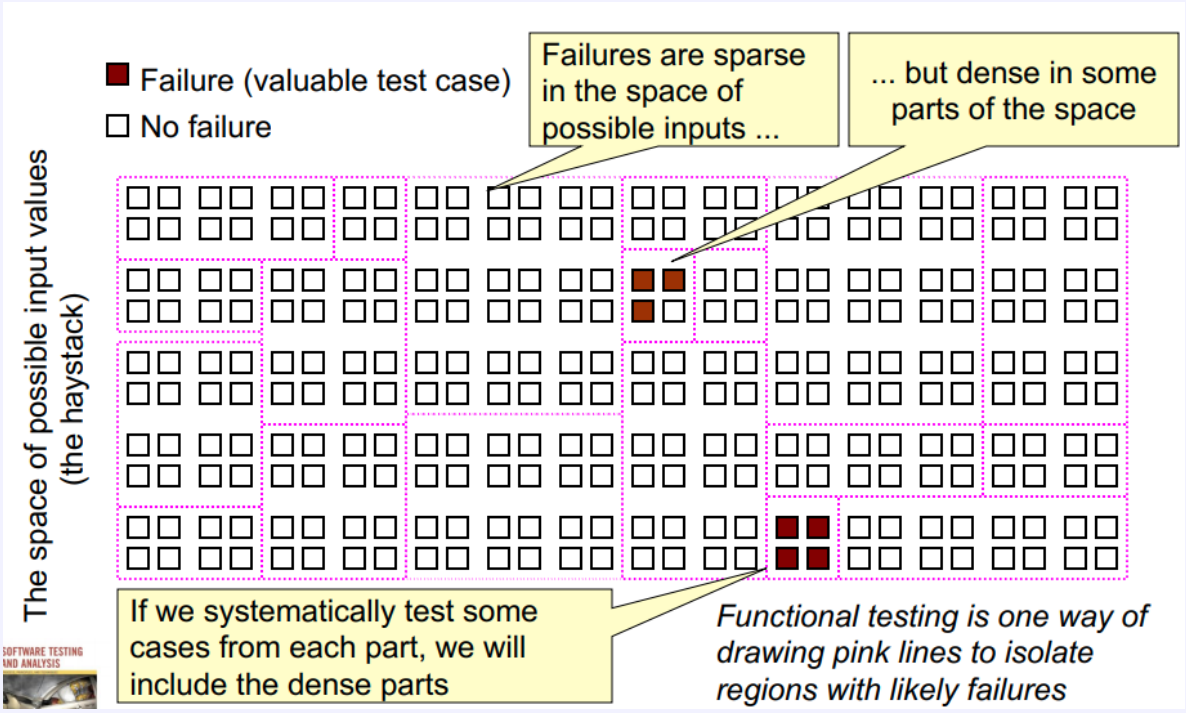
Random Testing

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

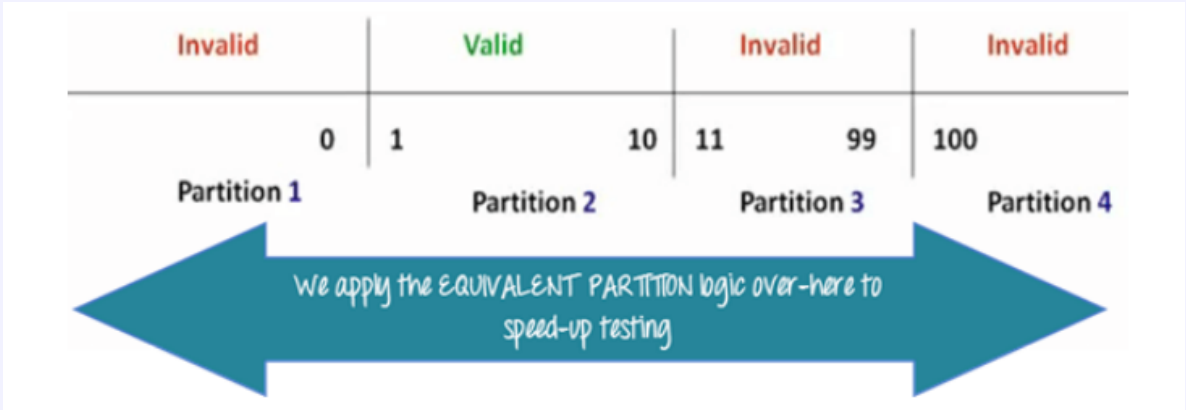
Systematic Functional Testing

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!

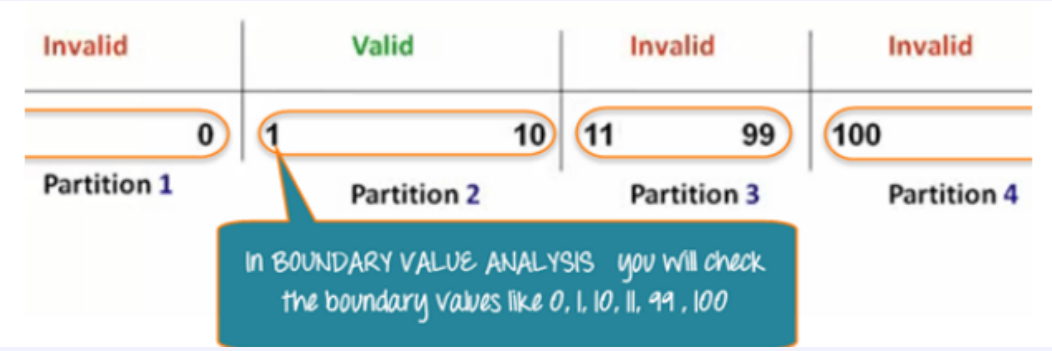


Quasi-Partition testing

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)

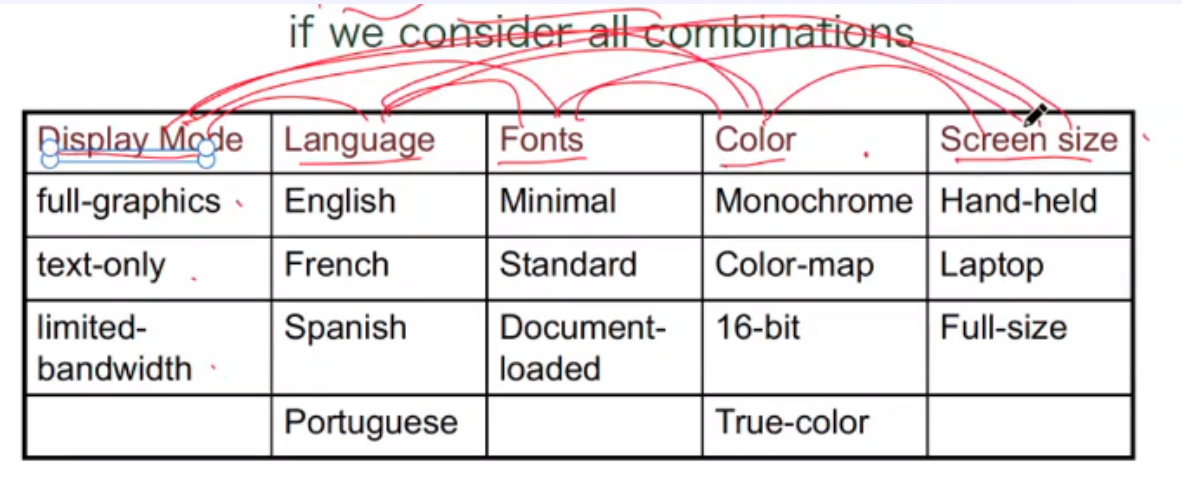
Constraints

- [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

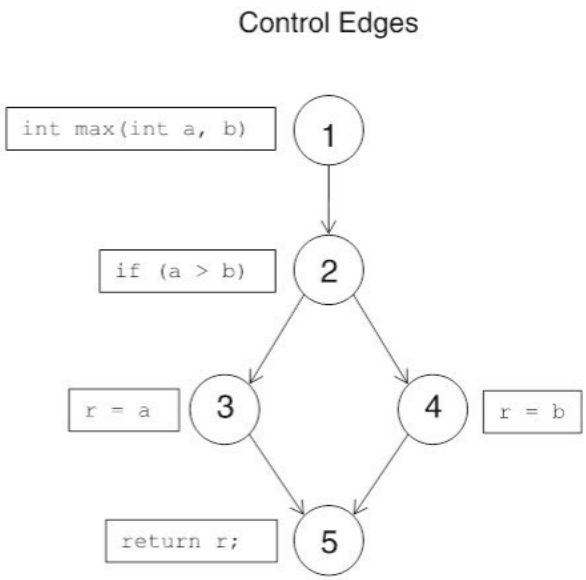
Properties of models

- 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

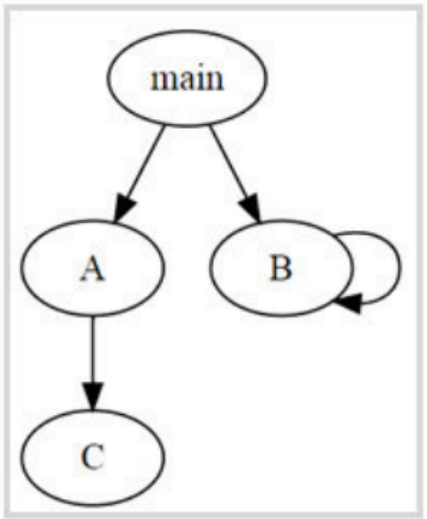
```
1  int main(int a, b) {
     int r;
2    if (a > b)
3      r = a;
     else
4    r = b;
5    return r;
   }
```



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();
  }
}
```

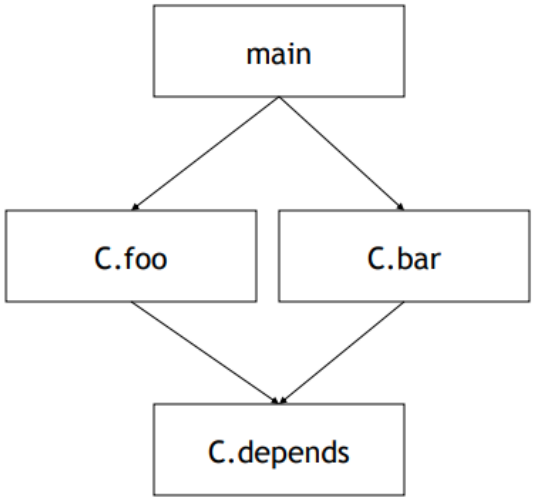


Note: the 'calls' relation represented by the edges, is overestimated via call graphs, some calls are simply impossible in real execution due to state

Context Insensitive Call Graphs

Each edge represents any call to a procedure with any context

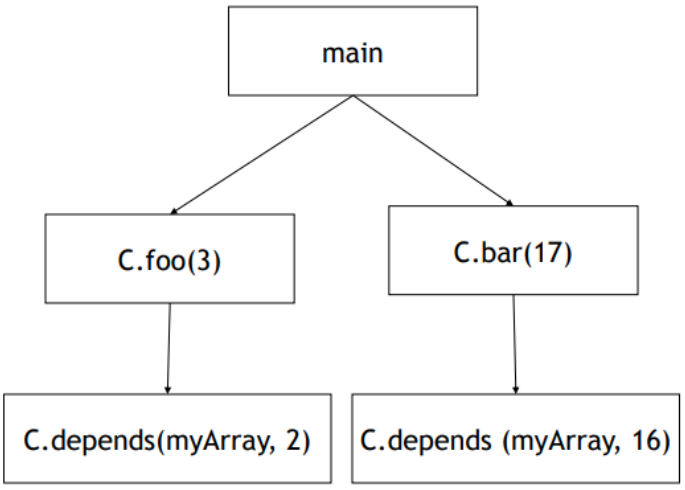
```
public class Context {
   public static void main(String
    args[]) {
    Context c = new Context();
    c.foo(3);
    c.bar(17);
   }

   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

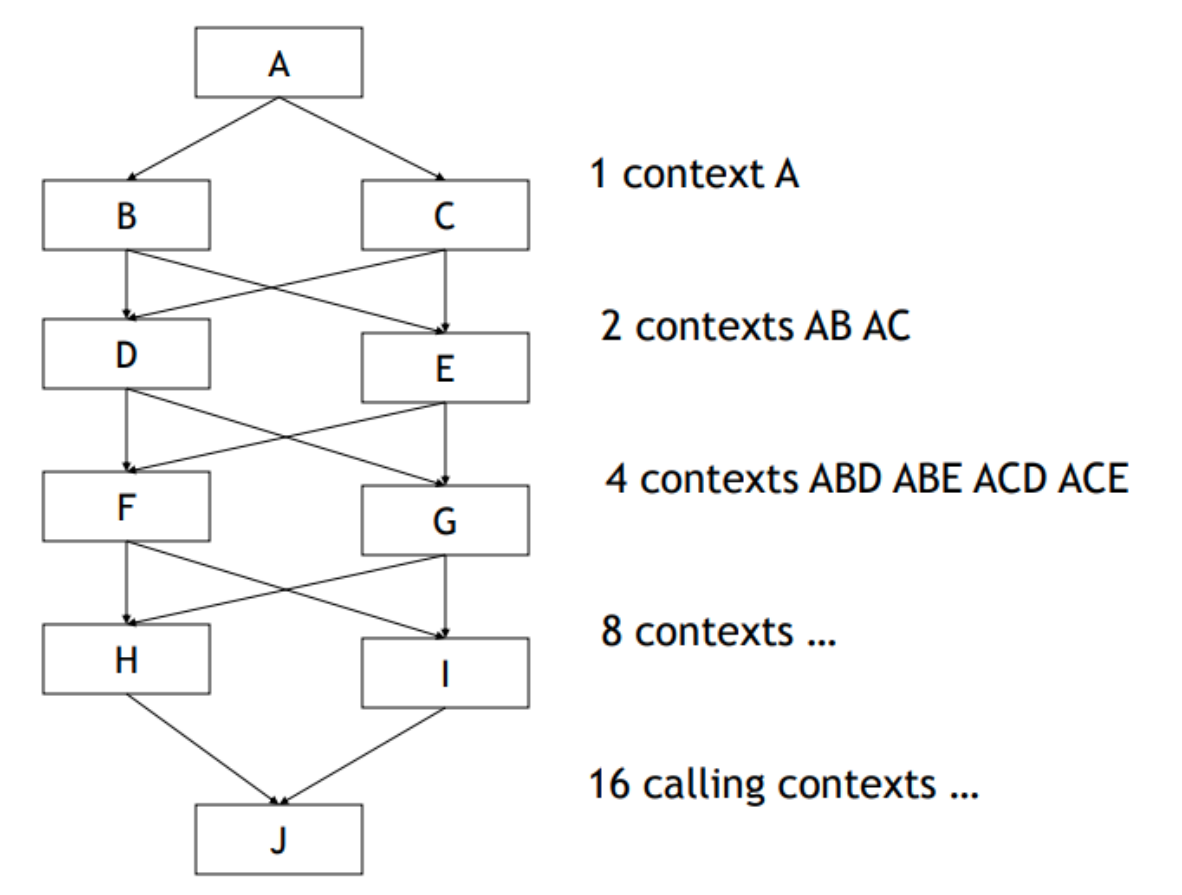
Each edge represents a unique **call stack** as well as function call (different calls have different edges)

```
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;
   }
}
```



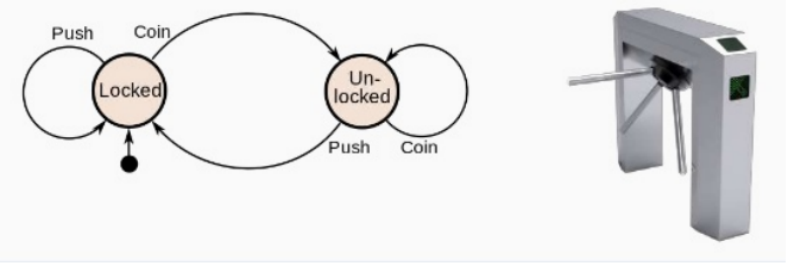
Context exponential growth

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



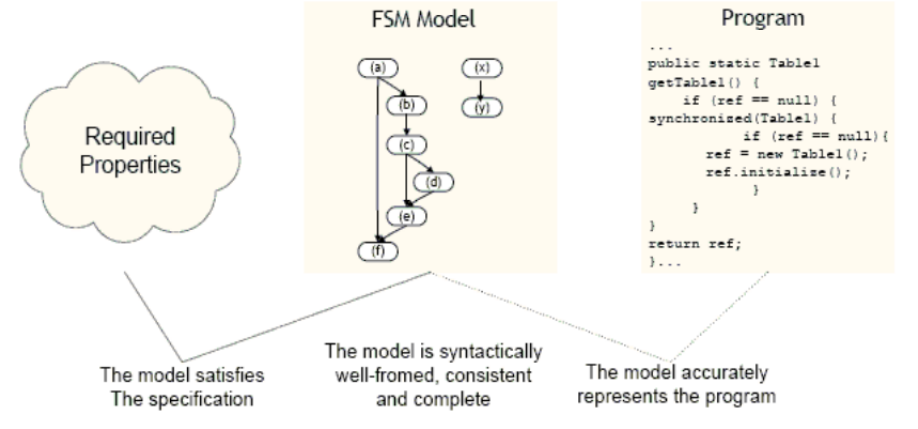
Finite State Machines

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)

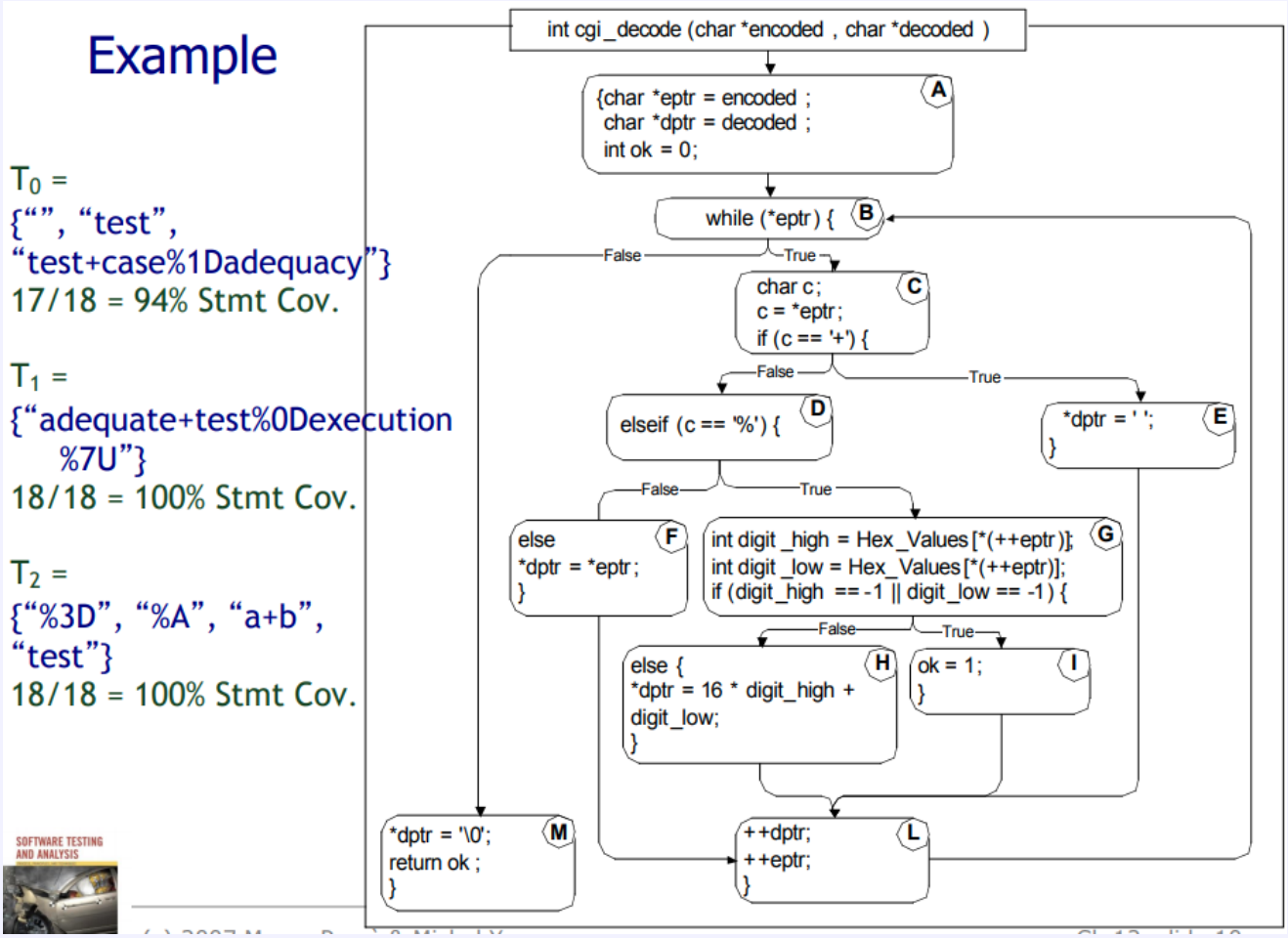
Type of testing based on the structure of the program itself, still testing product functionality against the specification but the throughness measures are changed.
Ansers the question "what is missing in our test suite?" - If a part of a program is not executed by any test case in the suite, faults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

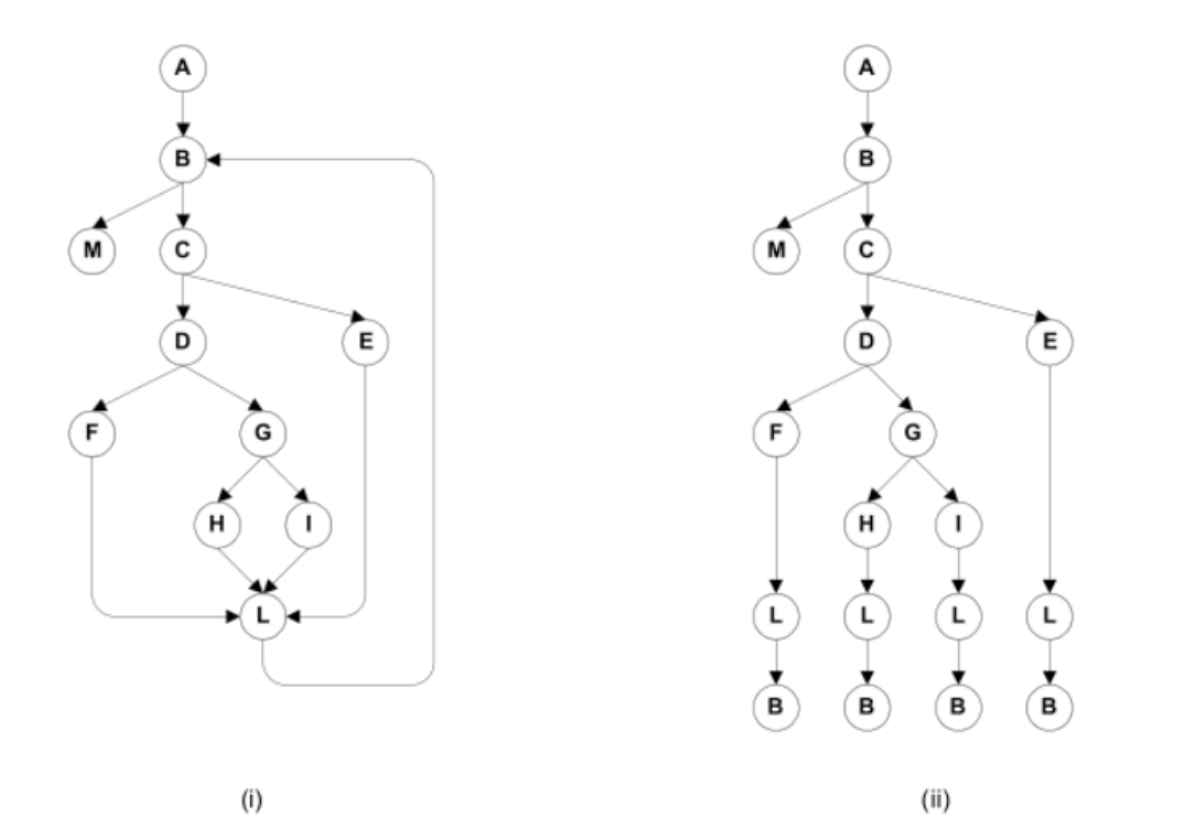
- usually impossible to satisfy

+ 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.

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

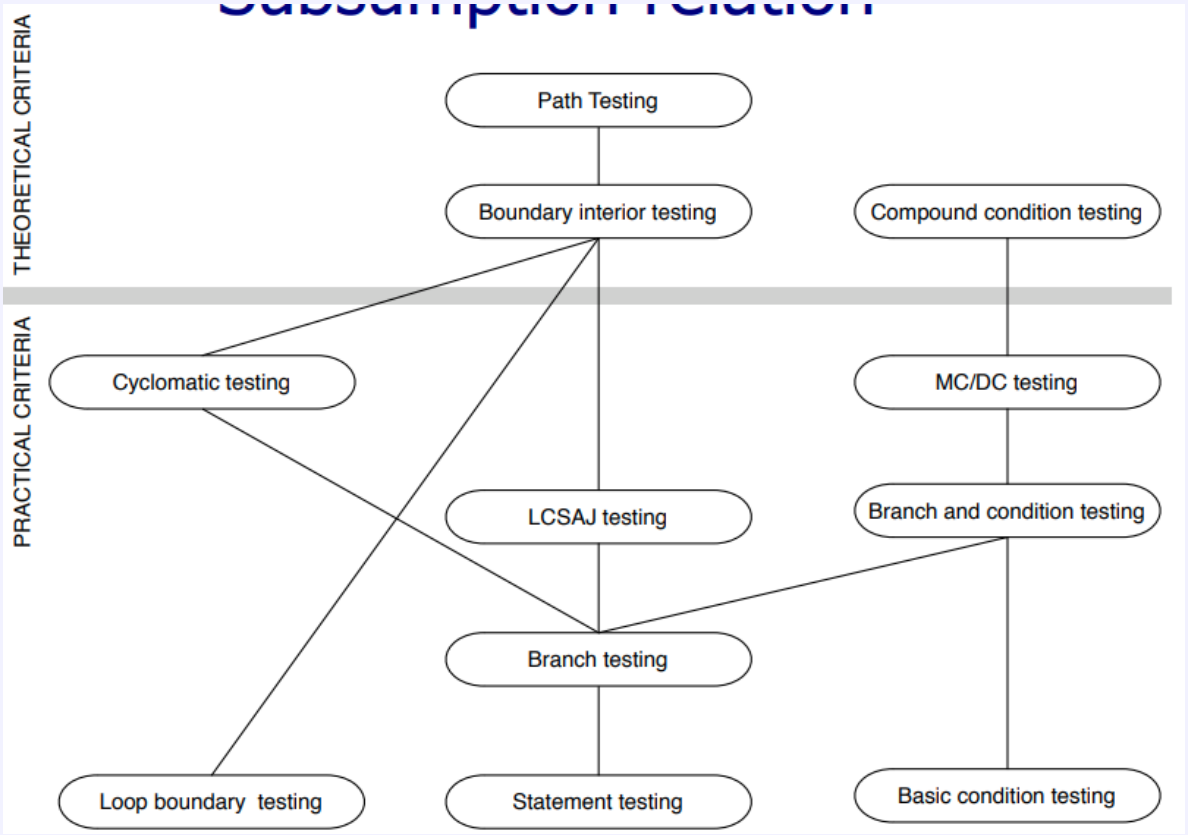
LCSAJ stands for Linear Code Sequence and Jump, i.e. a node in the CFG followed by a branch.
Various "thoroughness" levels:

- 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

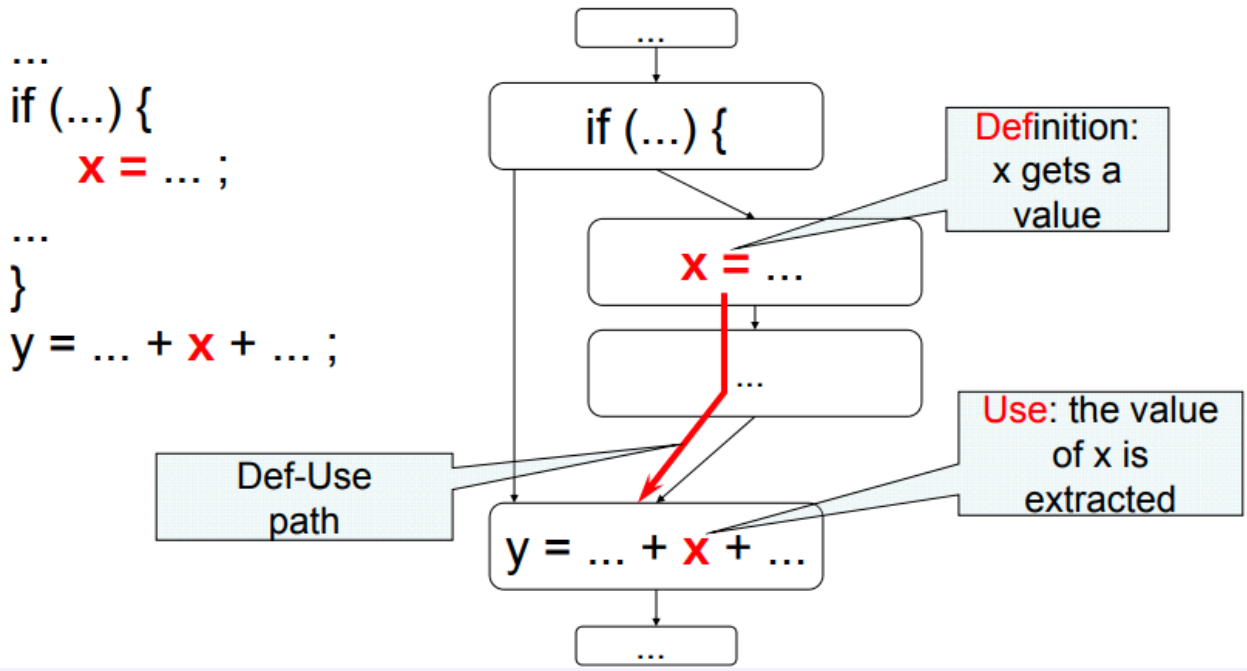
Def-Use Pairs

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

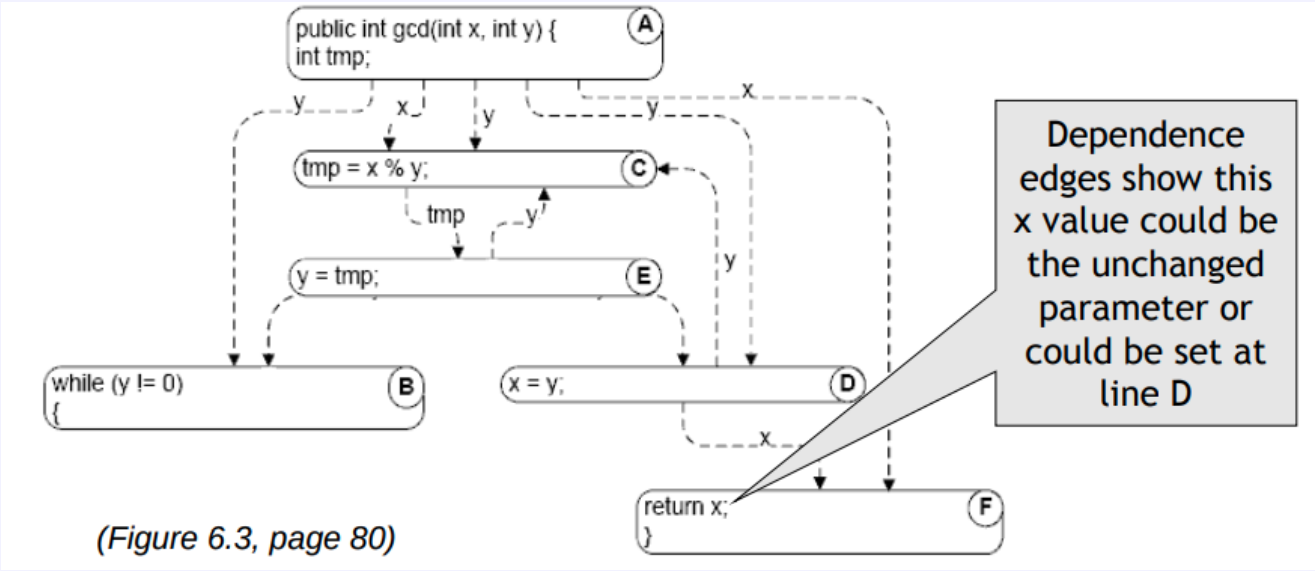


Definition-Clear path

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

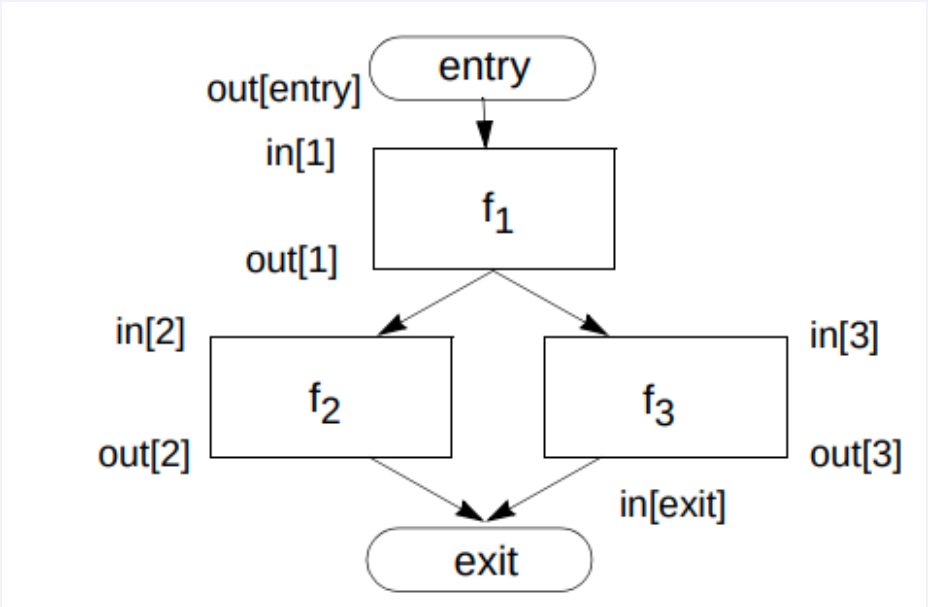
The data flow algorithm is a flexible tool which can be applied over any CFG's to find out some properties of the given graph.
The algorithm is based on the following set of equations, where b is some block of code (usually enough to carry out analysis on granuality of CFG node's or the edges of those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

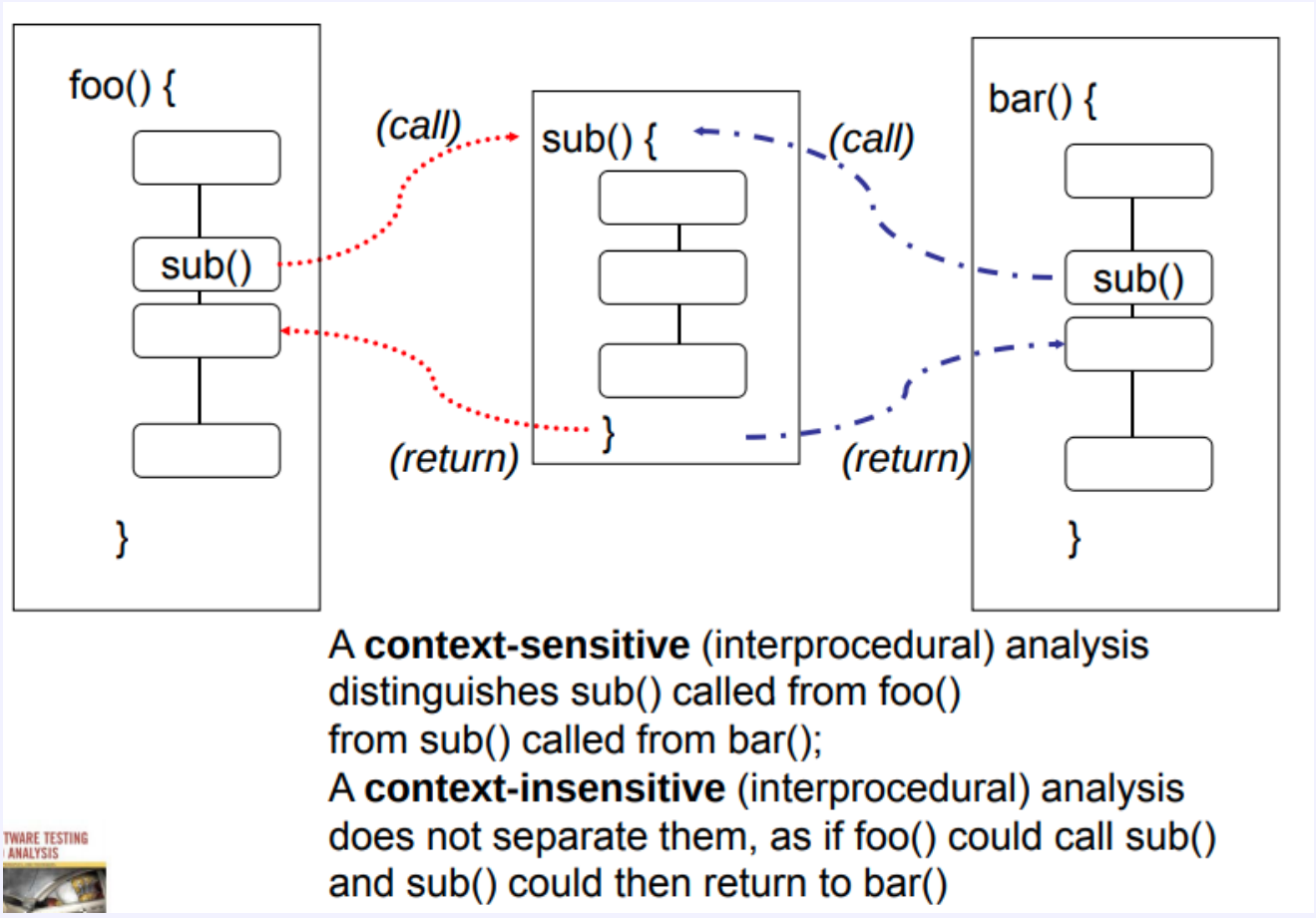
Scope of DF analysis

- 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

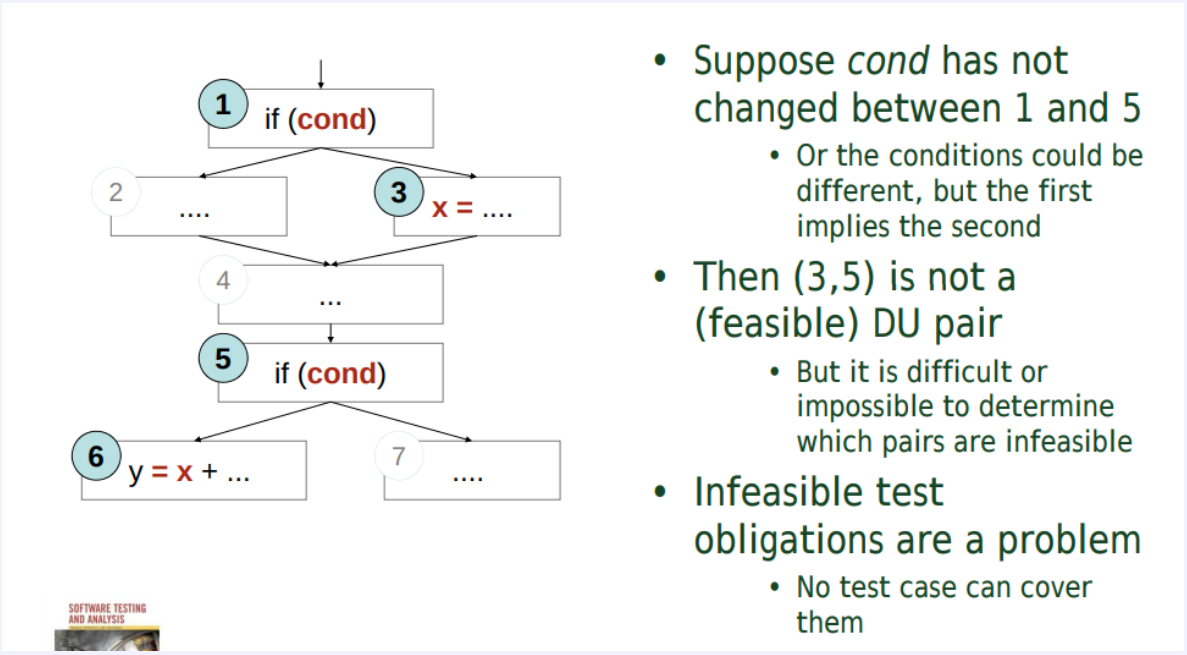
DF with complex structures

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 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

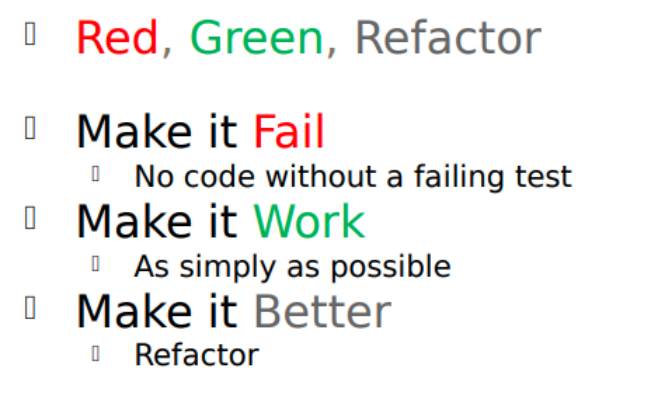
- Competent programmer assumption - they write programs which are nearly correct
- Coupling effect assumption - test cases that distinguish all programs differing from a correct one by only simple errors are so sensitive that they also implicitly distinguish more complex errors

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

Security Testing

Integration and Component Based Testing

System and Acceptance Testing

Concurrency Testing