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

Maksymilian Mozolewski

May 8, 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

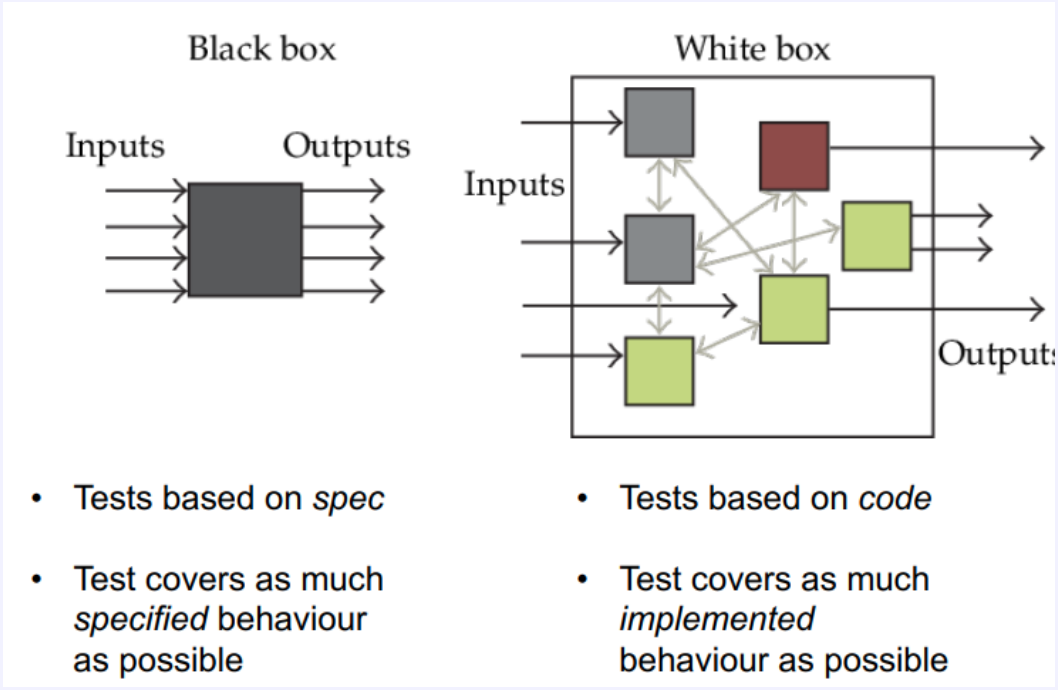
Functional Requirements:

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!)

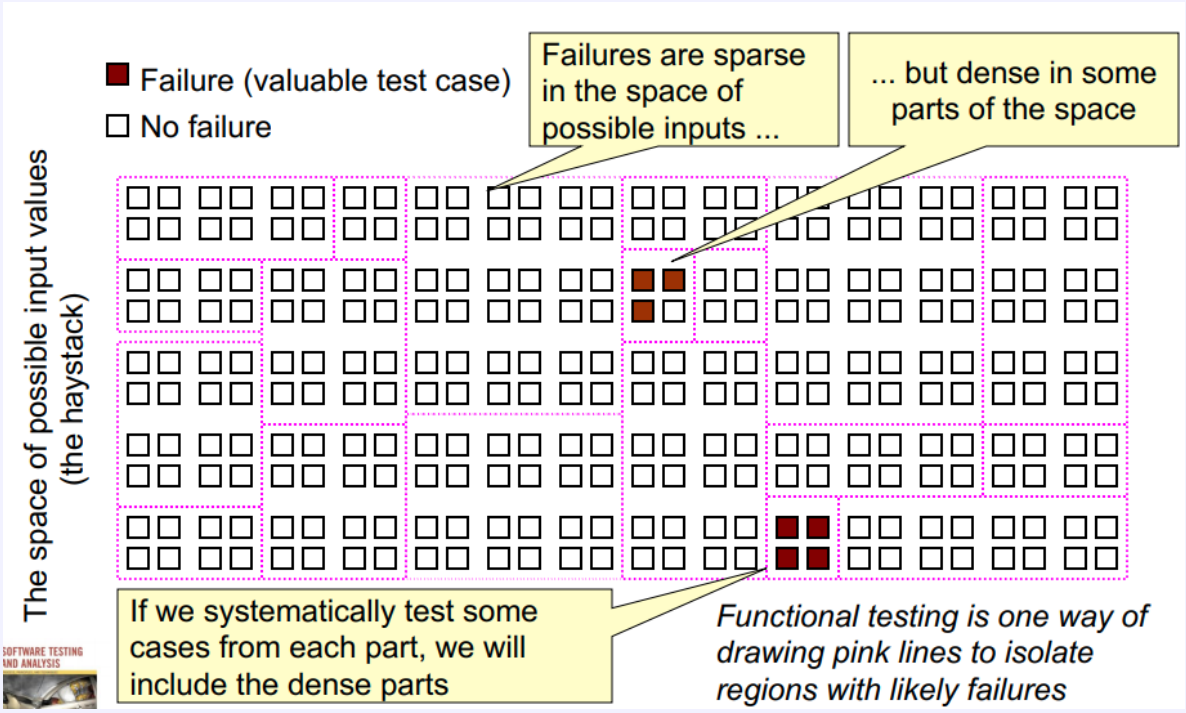
Random Testing

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

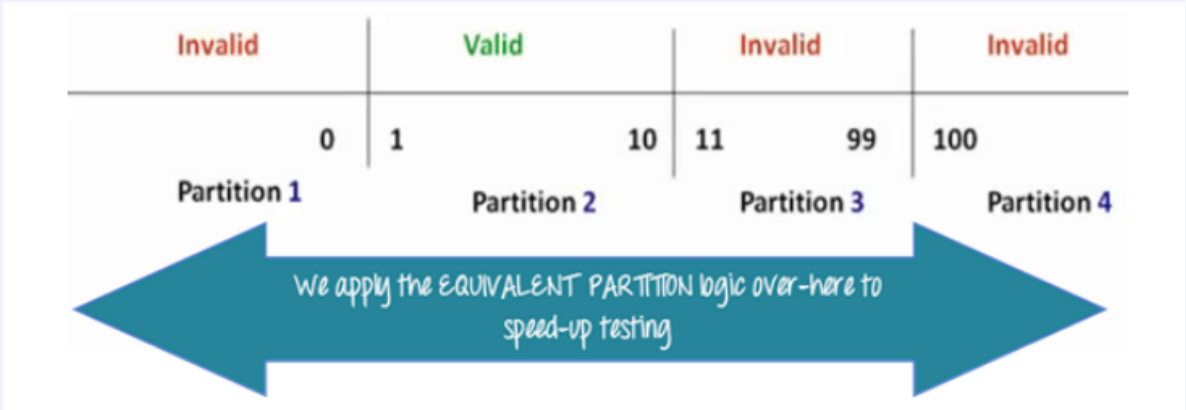
Systematic Functional Testing

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!

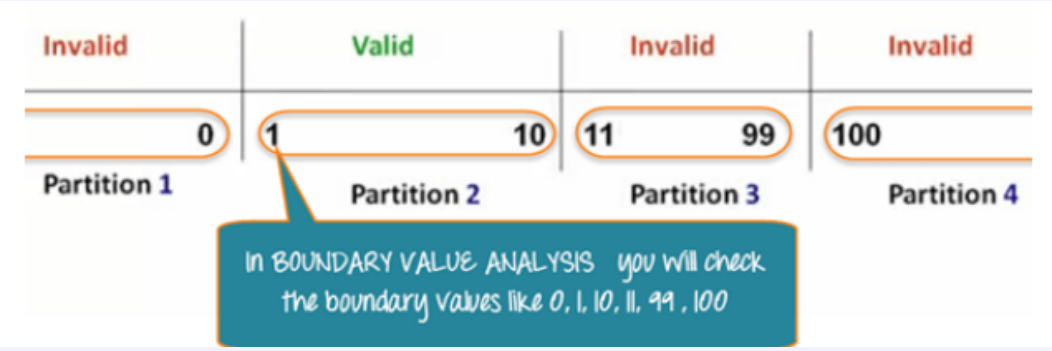


Quasi-Partition testing

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)

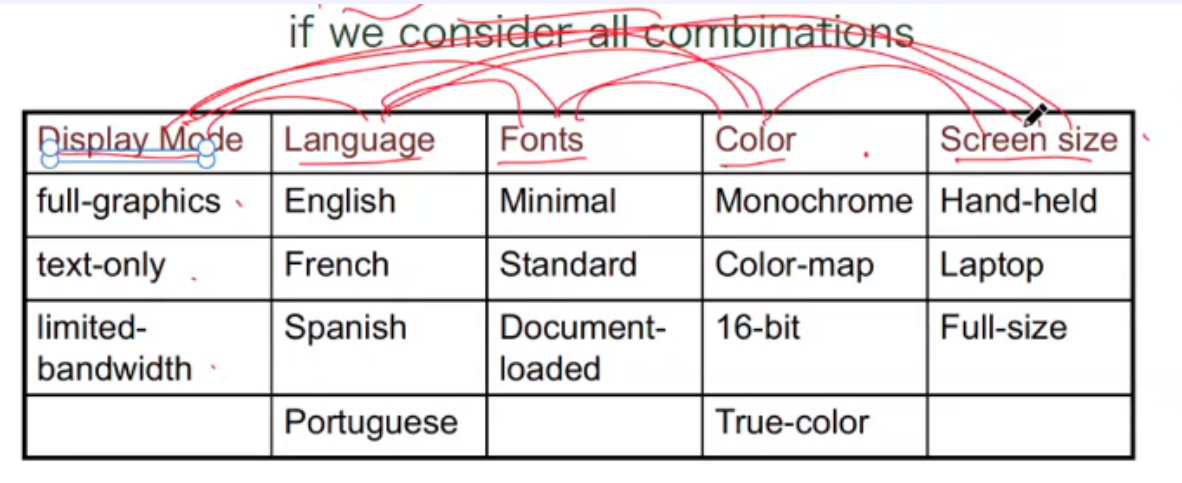
Constraints

- [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

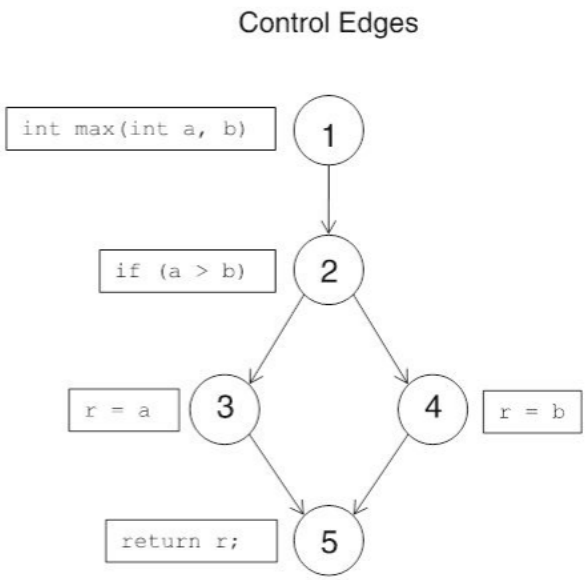
Properties of models

- 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

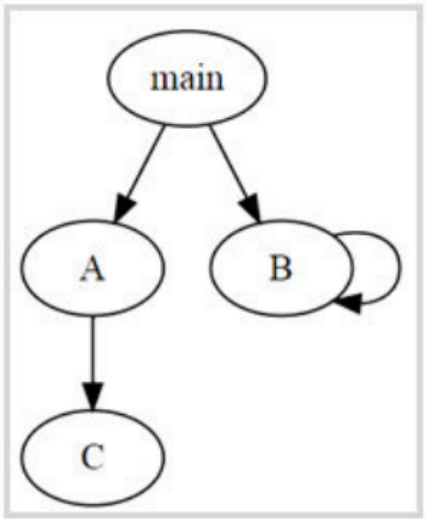
```
1  int main(int a, b) {
     int r;
2    if (a > b)
3      r = a;
     else
4    r = b;
5    return r;
   }
```



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();
  }
}
```

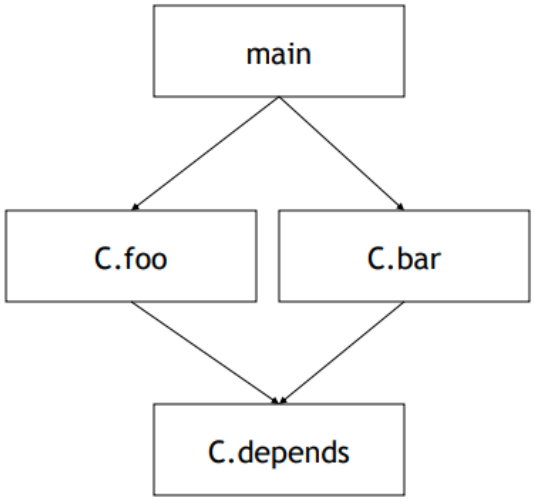


Note: the 'calls' relation represented by the edges, is overestimated via call graphs, some calls are simply impossible in real execution due to state

Context Insensitive Call Graphs

Each edge represents any call to a procedure with any context

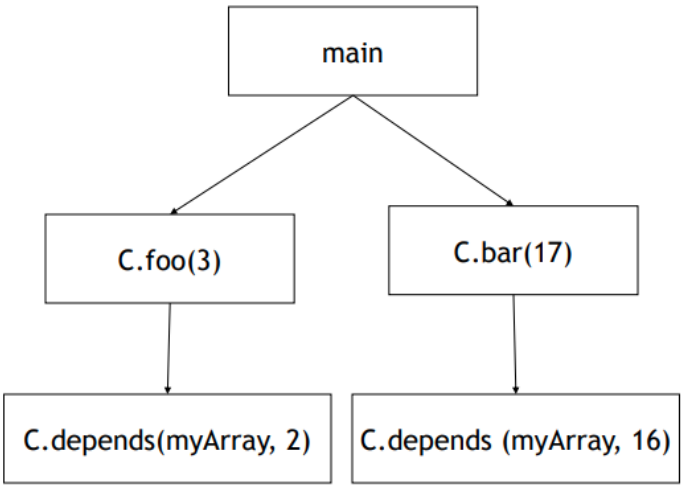
```
public class Context {
   public static void main(String
    args[]) {
    Context c = new Context();
    c.foo(3);
    c.bar(17);
   }

   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

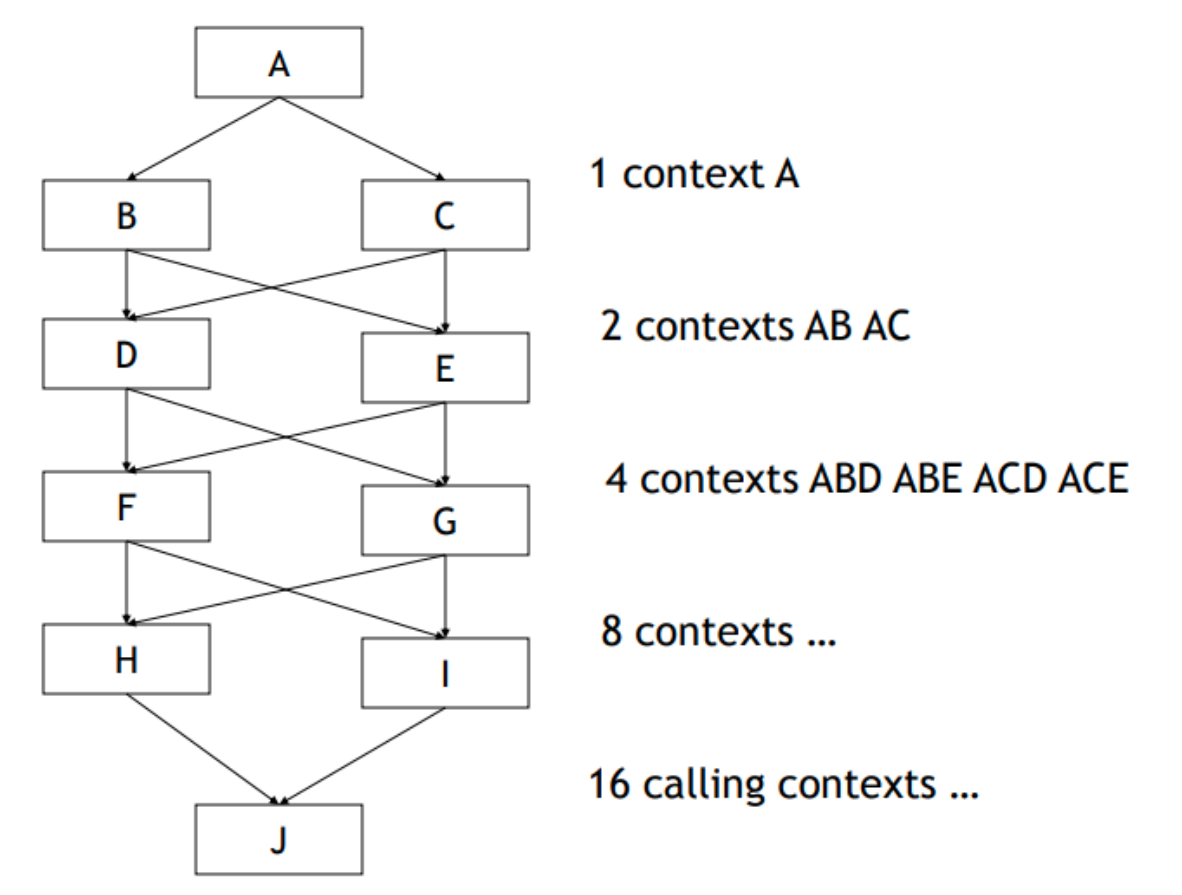
Each edge represents a unique **call stack** as well as function call (different calls have different edges)

```
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;
   }
}
```



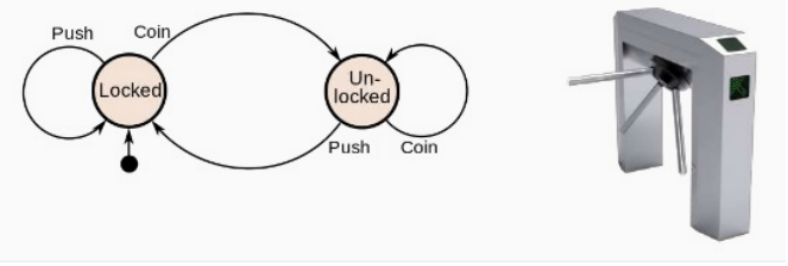
Context exponential growth

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



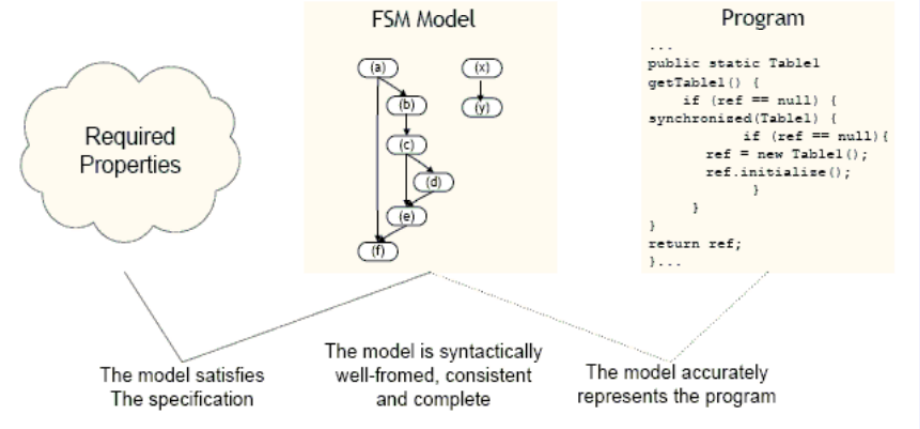
Finite State Machines

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

Data Flow Models

Data Flow Testing

Test Selection and Adequacy

Mutation Testing

Test Driven Developement

Regression Testing

Security Testing

Integration and Component Based Testing

System and Acceptance Testing

Concurrency Testing