Fault: An incorrect step, process, or data definition in a computer program. Fault triggers an error and/or failure. It is where the bug exists.

Failure: A deviation between the observed behaviour of a program, or a system, from its specification. (Output != expected output)

Error: An incorrect internal state that is the result of some fault. That is, executing the faulty code. Sometimes the faulty line of code may not execute then. It is not considered an error for that case. An error may not result in a failure -- it is possible that an internal state is incorrect but does not affect the output.

Controllability: the degree to which a tester can provide test inputs to the software.
Observability: degree to which a tester can observe the behaviour of a software artifact, such as its outputs and its effect on its environment.
Testability is composed of two measures*:* controllability and observability.

Equivalence partitioning tries to break up input domains into sets of ***valid*** and ***invalid*** inputs based on these **input** conditions.

Input condition: An input condition on the input domain is a predicate on the values of a single variable in the input domain. When we need to consider the values of more than one variable in the input domain, we refer to this as the combination of input conditions
A **precondition** is a condition on the input variables of a program that is *assumed to hold* when that program is executed. i.e. If the behaviour is undefined, then *we do not test for it*.
A **valid input** to a program is an element of the input domain that is the value expected from the program specification; that is, a non-error value according to the program specification.
An **invalid input** is an input to a program is an element of the input domain that is not expected or an error value that as given by the program specification. In other words, an invalid input is a value in the input domain that is not a valid input.

- A non-testable input is one that violates the *precondition* of a program.
- A testable invalid input is an invalid input that satisfies the precondition.

Input domain: the set of values that are accepted by the program as inputs, including all valid input domains and invalid input domain. i.e. all files can be read by the program, possible parameter sets suits the type, used global variables (only those used, appeared on the right hand side of equal sign).

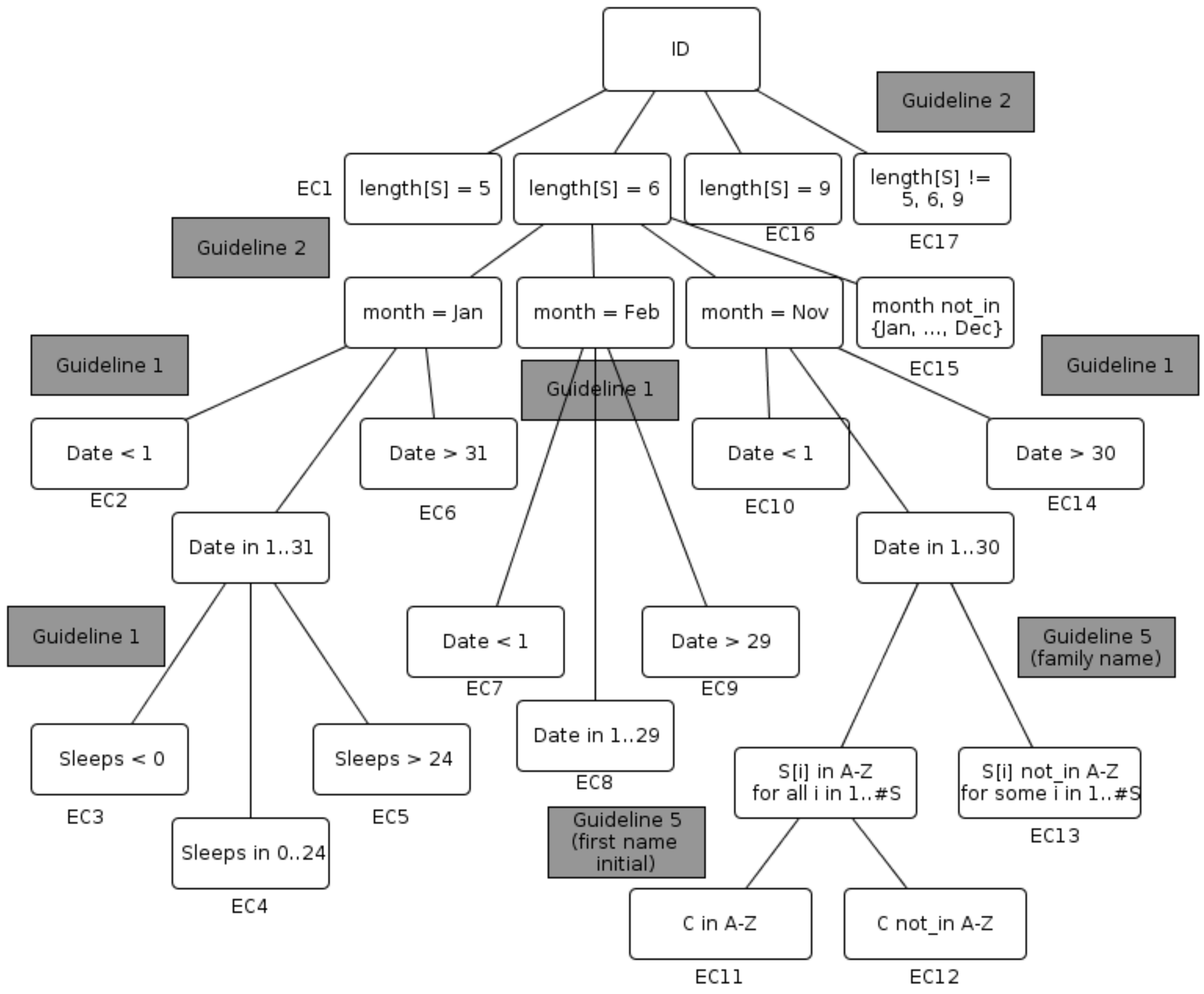
Domain testing refers to white box input partitioning
Equivalence partitioning refers to black box input partitioning

Equivalence partitioning
Step 1 - Identify the initial equivalence classes
Guidelines
- Guideline 1: Ranges
  - if an input condition specifies a range of values, identify one valid equivalence class for the set of values in the range, and two invalid equivalence classes; one for the set of values below the range and one for the set of values above the range. e.g. valid equivalence range for x is 1..99, then invalid range will be x < 1 and x > 99.
- Guideline 2: Categorical inputs
  - If an input condition specifies a set of possible input values and each is handled differently, identify a valid equivalence class for each element of the set and one invalid equivalence class for the elements that are not in the set.
  - If the input is selected from a set of vehicle types, such as {BUS, TRUCK, TAXI, MOTORCYCLE}, then we require
    - Four valid equivalence classes; one for each element of the set {BUS, TRUCK, TAXI, MOTORCYCLE
    - One invalid equivalence class for an element “outside” of the set; such as ONFOOT or BIKE
- Guideline 3: Fixed number of inputs
  - If the input condition specifies the number (say N) of valid inputs, define one valid equivalence class for the correct number of inputs and two invalid equivalence classes – one for values <N and one for values >N.i.e. N, <N, >N
- Guideline 4: Zero-one-many
  - If the input condition specifies that an input is a collection of items, and the collection can be of varying size; for example, a list or set of elements; define one valid equivalence class for a collection of size 0, one valid equivalence class for a collection of size 1, and one valid equivalence class for a collection of size> 1. i.e. 0, 1, >1
- Guideline 5: "Must" requirement
  - If an input condition specifies a “must be” situation, identify one valid equivalence class and one invalid equivalence class.
- Guideline 6: Catch all
  - derive tests based on intuition or your understanding of the program and domain, where none of the other guidelines fit!

Step 2 - Eliminating overlapping equivalence classes

Step3 - Selecting test cases

Test Template Trees e.g.



Partition combinations

All combinations

The ***all combinations*** criterion specifies that every combination of each equivalence class between blocks must be used. This is analogous to the *cross product* of sets. e.g. we have equivalent classes x = 0, x ≠ 0, y = 0, y ≠ 0, z = 0, z = 1, z = -1, then we have 2 * 2 * 3 = 12 test cases

Each choice combination

The ***each choice*** criterion specifies that just one test case must be chosen from each equivalence class.

e.g.
$$x = 0 \quad \wedge \quad y = 0 \quad \wedge \quad z \textit{ is lowercase}$$
$$0 < x < 10 \quad \wedge \quad y > 0 \quad \wedge \quad z \textit{ is uppercase}$$
$$x \geq 10 \quad \wedge \quad y = 0 \quad \wedge \quad z \textit{ is a non-letter}$$

Pair-wise combinations

The *pair-wise combinations* criterion aims to combine values, but not an exhaustive enumeration of all possible combinations. As the name suggests, an equivalence class from each block must be paired with every other equivalence class from all other blocks.

$x = 0 \wedge y = 0$	$0 < x < 10 \wedge y = 0$	$x \geq 10 \wedge y = 0$	$y = 0 \wedge z \textit{ is lowercase}$
$x = 0 \wedge y > 0$	$0 < x < 10 \wedge y > 0$	$x \geq 10 \wedge y > 0$	$y = 0 \wedge z \textit{ is uppercase}$
$x = 0 \wedge z \textit{ is lowercase}$	$0 < x < 10 \wedge z \textit{ is lowercase}$	$x \geq 10 \wedge z \textit{ is lowercase}$	$y = 0 \wedge z \textit{ is a non} - \textit{letter}$
$x = 0 \wedge z \textit{ is uppercase}$	$0 < x < 10 \wedge z \textit{ is uppercase}$	$x \geq 10 \wedge z \textit{ is uppercase}$	$y > 0 \wedge z \textit{ is lowercase}$
$x = 0 \wedge z \textit{ is a non} - \textit{letter}$	$0 < x < 10 \wedge z \textit{ is a non-letter}$	$x \geq 10 \wedge z \textit{ is a non-letter}$	$y > 0 \wedge z \textit{ is uppercase}$
			$y > 0 \wedge z \textit{ is a non-letter}$

Boundary conditions are predicates that apply directly on, above, and beneath the boundaries of input equivalence classes and output equivalence classes.

A ***computational fault*** is a fault that occurs during a computation in a program; for example, an arithmetic computation or a string processing error.
A computation fault of average = (left + right) / 2 could be average = left + right / 2

A ***boundary shift*** is when a predicate in a branch statement is incorrect, effectively 'shifting' the boundary away from its intended place.
e.g. a boundary shift of if (length(list) > 10) then … could be if (length(list) >= 10) then …

A **path condition** is the condition that must be satisfied by the input data for that path to be executed.
A **domain** is the set of input data satisfying a path condition.
A **domain boundary** is the boundary of a domain. It typically corresponds to a simple predicate.

A **closed boundary** is a domain boundary where the points on the boundary belongs to the domain, and is defined using an operator that contains an equality. For example, x≤10 and y == 0 are both closed boundaries.
An **open boundary** is a domain boundary which is not closed, and is defined using a strict inequality. For example, y < 10 is an open boundary, because 10 does not fall withing the boundary.

An **on point** is a point on the boundary of an equivalence. For example, the value 10 is an on point for the boundary x ≤ 10, x < 10 and x == 10.
An **off point** is a point just off the boundary of an equivalence class. For example, if x is an integer, the value 9 and 11 are the off points for the equivalence class x ≤ 10 and x==10. If x is a floating point number, the off point would be 9.999 and 10.001.

Boundary value analysis guideline

1. If a boundary is a strict equality or non-equality, for example x = 10 or x ≠ 10, select the on point, 10, and both off points, 9 and 11.
2. For an open boundary, for example x < 10, select the on point 10, and the off point 9.
3. For a closed boundary, for example x ≤ 10, select the on point 10, and the off point 11.
4. For boundaries containing unordered types, that is, types such as Booleans or strings, choose one on point and one off point. An on point is any value that is in the equivalence class, while an off point is any value that is not. For example, if the equivalence class for a string variable is "contains no spaces", then test one string containing no spaces, and one string containing at least one space.
5. Analyse boundaries of equivalance classes; not individual equivalence classes. If we have the equivalences classes x ≤ 0 and x > 0, then 0 is the on point for both, so analyse that boundary not each class in isolation.

Selecting boundary values for **n-dimensional boundaries**:

- N on points and 1 off point are needed for boundaries based on inequalities, with the on points as far apart as possible;
- N on points and 2 off points if are needed boundaries based on equalities and non-equalities.
- If the off point test case is a distance d from the boundary, then only boundary shifts of magnitude greater d can be detected.

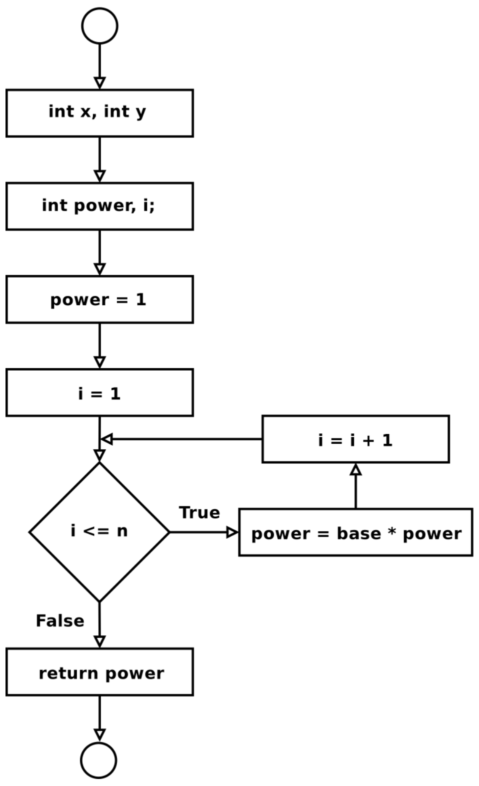
Example: triangle with side x, y, z, we can have

- (1, 2, 3) on point 1
- (100, 200, 300) on point 2
- (400, 2, 402) on point 3
- (1, 2.9999, 3) off point 1

Control flow diagram

- An **execution path**, or just a path, is a sequence of nodes in the control flow graph that starts at the entry node and ends at the exit node.
- A **branch**, or **decision**, is a point in the program where the flow of control can diverge. For example, if-then-else statements and switch statements cause branches in the control flow graph.
- A **condition** is a simple atomic predicate or simple relational expression occurring within a branch. Conditions do not contain and (in C &&), or (in C ||) and not (in C !) operators.
- A **feasible path** is a path where there is at least one input in the input domain that can force the program to execute the path. Otherwise the path is an infeasible path and no test case can force the program to execute that path.

e.g. If (a && b) statement; The a and b are conditions in the example, The (a && b) is branch.
Note here! The for loop is expressed as condition and branch!



Statement/node coverage: All the statements (in rhombus and squares) in control flow diagram should be executed at least once
Arc/branch/decision coverage: Every possible alternative in a branch (or decision) of the program should be exercised at least once. For if statements this means that the branch must be made to take on the values true and false, *even if the result of this branch does nothing*, for example, for an if statement with no else, the false case must still be executed.
Condition coverage: Each condition in a branch is made to evaluate to true and false at least once.
in the branch if (a and b), where a and b are both conditions, then we must execute a set of tests such as:

- a evaluates to true at least once and false at least once; and
- b evaluates to true at least once and false at least once.

Decision/condition coverage
Decision here can be think as branch. Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false. That is, a combination of branch coverage and condition coverage.

1. a = true b = false
2. a = true b = true
3. a = false b = true
4. a = false b = false

Think of the conditions 1 and 3. In condition coverage we made a is true once and b is true once, however that does not make the branch true at all.
Therefore we combine the two concepts together and call it decision/condition coverage. In this case can be achieved by testing 2 and 4.
Multiple condition coverage
All possible combinations of condition outcomes within each branch should be exercised at least once.
e.g. If (a && b && c) statement;
We just find all $2^3 = 8$ combination of all possibility of a, b and c
Path coverage: the path to get to the final answer. However the control flow containing loops will have infinite number of path, therefore not that useful

Coverage measurement
Coverage score = number of test objects / number of total test objectives

Static data-flow analysis
Define (d): it could be x = 0, x = y + 2, x++, scanf(x), it means give it a value. Note that it is not same as declaring a variable. int x is declare, x = 0 is define.
reference(r): using the variable while not changing it. e.g. print(x), $y = x^2$, note x value is not changed here. Sometimes called use a variable.
undefine(u): e.g. {int i = 0}, after going outside of the bracket scope, it is undefining variable i
In path of a variable, it should be defined first and be referenced somewhere and be undefined at the end. Sometimes called kill a variable.

A *u-r anomaly* occurs when an undefined variable is referenced. Most commonly *u-r* anomalies occur when a variable is referenced without it having been assigned a value first — that is, it is *uninitialised*. A common source of *u-r* anomalies arises when the wrong variable is referenced
A *d-u anomaly* occurs when a defined variable has not been referenced before it becomes undefined. This anomaly usually indicates that the wrong variable has been defined or undefined.
A *d-d anomaly* indicates that the same variable is defined twice causing a hole in the scope of the first definition of the variable. This anomaly usually occurs because of misspelling or because variables have been imported from another module.

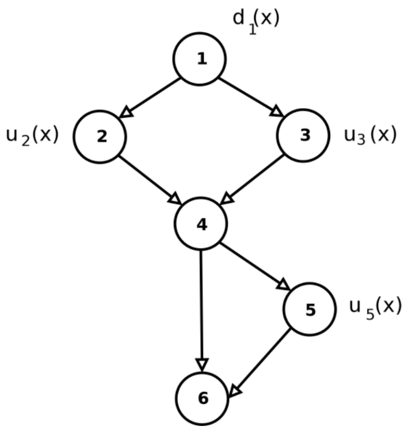
Dynamic data-flow analysis
A **C-use** of a variable is a *computation use* of a variable, for example y = x * 2;
A **P-use** of a variable is a *predicate use* , for example, if (x > 2)

- Let $d_n(x)$ denote a variable x that is assigned or initialised to a value at node (statement) n (Definition).
- Let $u_n(x)$ denote a variable x that is used, or referenced, at node (statement) n (Use).
- Let $c_n(x)$ denote a computational usage of the variable x at the node n (Computational Use).
- Let $p_n(x)$ denote a predicate usage of the variable x at the node n (Predicate Use).
- Let $k_n(x)$ denote a variable x that is killed, or undefined, at a node (statement) n (Kill).

e.g. node n "y = 3 * x", is annotated with a *computational* use of x, and a definition of y

- A definition clear path p with respect to a variable x is a sub-path of the control-flow graph where x is defined in the first node of the path p, and is not defined or killed in any of the remaining nodes in p.
- A loop-free path segment is a sub-path p of the control-flow graph in which each node is visited at most once.
- A definition $d_m(x)$ reaches a use $u_n(x)$ if and only if there is a sub-path p that is definition clear (with respect to x, and for which m is the head element, and n is the final element.

Dynamic data-flow coverage-based criteria



1.All-defs (definition): For the All-Defs criterion we require that there is some definition-free sub-path from *all definitions* of a variable to a single use of that variable. A test case that starts from def to end without repeated defs is sufficient. The paths 1, 2, 4, 6 or the path 1, 3, 4, 5 would be satisfactory.
2.All-uses (reference): The All-Uses criteria requires some definition-free sub-path from *all definitions* of a variable to *all uses* reached by that definition. Test cases need to cover all uses at least once. A test suite that traverses the paths 1, 2, 4, 5, 6 and 1, 3, 4, 6 are satisfactory under this criteria.
3.All-DU-paths (definition/use): The All-DU-Paths criterion requires that a test set traverse *all definition-free sub-paths* that are cycle-free or simple-cycles from *all definitions* to *all uses* reached by that definition, and every successor node of that use. Test cases should cover all nodes before use. Under this criteria, satisfactory paths are given by 1, 2, 4, 5, 6 and 1, 3, 4, 5, 6.

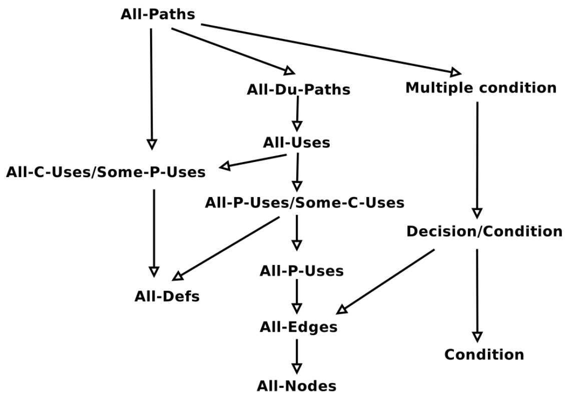
Mutants
Given a program, a ***mutant*** of that program is a copy of the program, but with *one* slight *syntactic* change.
Given a program, a test input for that program, and a mutant, we say that the test case kills the mutant if and only if the output of the program and the mutant differ for the test inputs.
• A mutant that has been killed is said to be dead.
• A mutant that has not been killed is said to be alive.
The *coupling effect* states that a test case that distinguishes a small fault in a program by identifying unexpected behaviour is so sensitive that it will distinguish more complex faults; that is, complex faults in programs are *coupled* with simple faults.
A *mutant operator* is a transformation rule that, given a program, generates a mutant for that program.
if the statement if (x < y) occurs in a program, the following seven mutants will be created:
• if (x =< y) • if (x >= y) • if (x != y) • if (false)
• if (x > y) • if (x == y) • if (true)
if the statement x = 5 occurs in a program, the following three mutants will be created:
• x = abs(5) • x = -abs(5) • x = failOnZero(5)
fail-on-zero throws an exception if the expression evaluates to zero.

Java mutant rules
• Arithmetic Operator Replacement: Replace each occurrence of an arithmetic operator +, -, *, /, **, and % with each of the other operators, and also replace this with the left operand and right operand (for example, replace x + y with x and with y).
• Conditional Operator Replacement: Replace each occurrence of a logical operator &&, ||, &, |, and ^ with each of the other operators, and also replace this entire expression with the left operand and the right operand.
• Shift Operator Replacement: Replace each occurrence of the shift operators <<, >>, and >>> with each of the other operators, and also replace the entire expression with the left operand.
• Logical Operator Replacement: Replace each occurrence of a bitwise logical operator &, |, and ^ with each of the other operators, and also replace the entire expression with the left and right operands.
• Assignment Operator Replacement: Replace each occurrence of the assignment operators +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, and >>>= with each of the other operators.
• Unary Operator Insertion: Insert each unary operator +, -, !, and ~ before each expression of the correct type. For example, replace if (x = 5) with if (!(x = 5)).
• Unary Operator Deletion: Delete each occurrence of a unary operator.
• Scalar Variable Replacement: Replace each reference to a variable in a program by every other variable of the same type that is in the same scope.
• Bomb Statement Replacement: Replace every statement with a call to a special Bomb() method, which throws an exception. This is to enforce statement coverage, and in fact, only one call to Bomb() is required in every program block.

mutation score = mutants killed / total mutants

Given a program and a mutation of that program, the mutant is said to be an *equivalent mutant* if, for every input, the program and the mutant produce the same output. e.g s[j++] = s[I]; and s[j++] = s[abs(i)]; where i always between 0 and six of array, this mutant is equivalent to the original program, and cannot be killed.

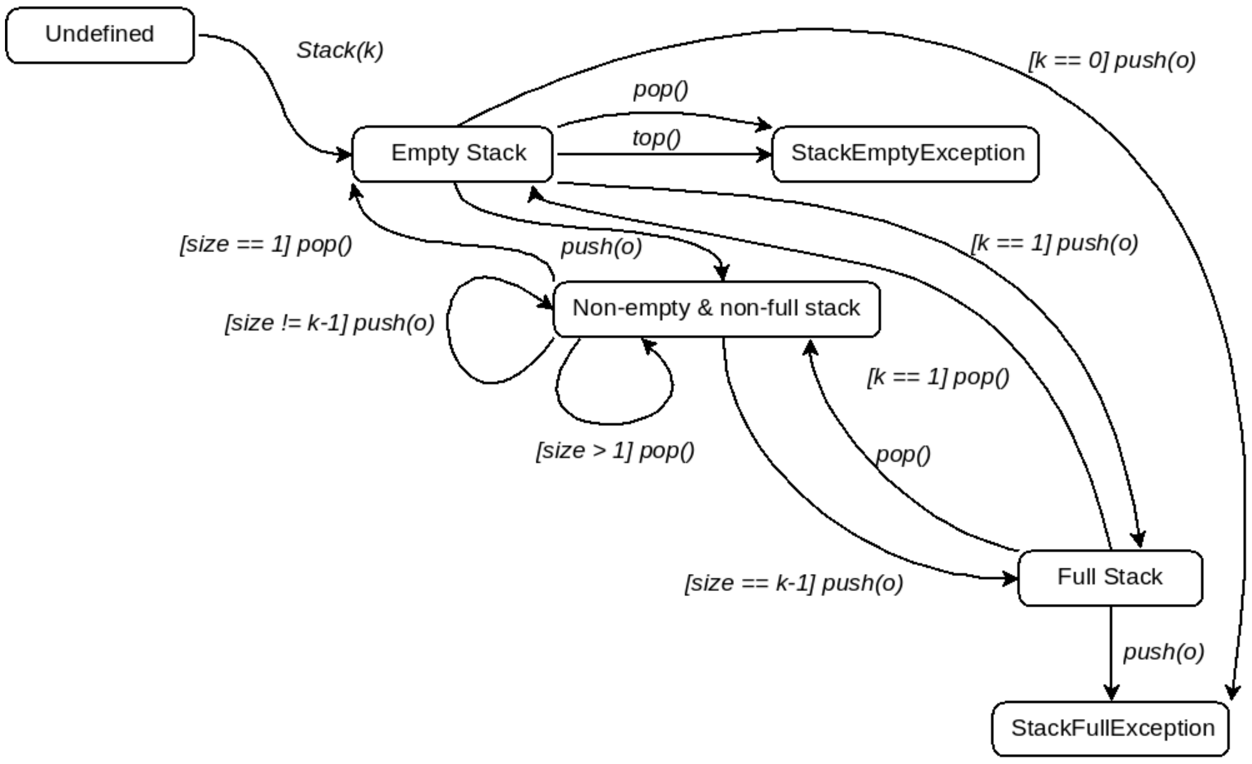
Criterion A *subsume*s criterion B if and only if for any control-flow graph P:
P satisfies A ⟹ P satisfies B
Criteria A is equivalent to criteria B if and only if A subsumes B, and B subsumes A.
The subsumes relation is transitive, therefore, if A subsumes B, and B subsumes C, then A subsumes C.



A *finite state automaton* (FSA) or *finite state machine* is a model of behaviour consisting of a finite set of states with actions that move the automata from one state to another.

Testing state-based programs
Test using finite state automata (FSA),
1. Identify all possible states
2. Identify all possible operations for each state
3. Identify the source and target states of every operation in the model

E.g. FSA for a Stack



Test with FSA:
- **State coverage*: Each state in the FSA must be reached by the traversal. (impossible for FSA with loops)
- **Transition coverage**: Each transition in the FSA must be traversed. This subsumes state coverage.
- **Path coverage**: Each path in the FSA must be traversed. This subsumes state coverage, but is impossible to achieve for a FSA with cyclic paths.

e.g. test the stack object, we can draw a FSA graph, and test its push, pop and top method according to the graph

Intrusive testing
adding things in code to test internal state.
Problem: breaks encapsulation

Non-intrusive testing
The idea is to test each of the transitions in the testing automaton implicitly by using other operations in the module to examine the results of a transition.
Initialisers: The operations that initialise the module, such as the Stack constructor
Transformers: Operations to change the state of the module, such as pop operation in the stack example.
Observers: Operations to observe the module state, such as the isEmpty and top operations in the stack example
The technique for testing involves deriving test sequences, using initialisers and transformers, to move the automaton to the required state. At each state, observe the value of the module using the observer operations. The only case where the observer operations are not used is in the exception states, because the execution of the module is considered to have terminated.
All observer operations could be used to observe the state when an FSA state is being visited for the first time.

OOP Testing inheritance
What functions should be tested in sub-classes?
The functions that changed, and the functions that uses the changed functions.
Can tests for a parent class be reused for a child class?
Clearly, new tests that are necessary will be those for which the *requirements* of the overridden methods change.
Building and Testing Inheritance Hierarchies
flattening the inheritance hierarchy
In effect each subclass is tested as if all inherited features were newly defined in the class under test — so we make absolutely no assumptions about the methods and attributes inherited from parent classes. Test cases for parent classes may be re-used after analysis but many test cases will be redundant and many test cases will need to be newly defined.
Incremental inheritance-based testing
Step 1: First test each base class by:
- Testing each method using a suitable test case selection technique; and
- Testing interactions among methods by using the techniques discussed earlier in this chapter.

Step 2: Then, consider all sub-classes that inherit or use (via composition or association) only those classes that have already been tested.
A child inherits the parent's test suite which is used as a basis for test planning. We only develop new test cases for those entities that are directly, or indirectly, changed.
Incremental inheritance-based testing does reduce the size of test suites, but there is an overhead in the analysis of what tests need to be changed. It certainly reduces the number of test cases that need to be selected over a flattened hierarchy. If the test suite is structured correctly, test cases can be reused via inheritance as well, which provides the same benefits for conceptualisation and maintenance for test suite implementations as it does for product implementations.

A test oracle is: a program, a process or a body of data that determines if actual output from a program has failed or not.

Active oracle: A program that, given an input for a program-under-test, can generate the expected output of that input.
Passive oracle: A program that, given an input for a program-under-test, and the actual output produced by that program-under-test, verifies whether the actual output is correct.

Metamorphic Oracles
By 'metamorphic', we mean that, given two different inputs that are related by some property p, if we run them both through our program, their outputs will be related by some other property q. The metamorphic relation is the relationship between p and q.
For example, if we want to test a function that sorts a list of numbers, then we can make use of the fact that, given a list, any permutation of that list will result in the same output of the sort function (assuming a non-stable sort).
To do metamorphic testing, we generate an input for a program, execute this input, and then generate another input whose output will be related to the first.
Formally, metamorphic testing follows the following pattern to test a function f:
1. Given input i, generate another input j, such that property p(i, j) holds between i and j.
2. Obtain outputs oi=f(i) and oj=f(j).
3. Check whether $q(o_i, o_j)$ holds between o_i and o_j, raising an error if it fails.

```
 e.g. int [] permutatedList = permutate(list);
    output1 = sort(list);
    output2 = sort(permutatedList);
    assertEquals(output1, output2);
```

An **alternate implementation** of a program, which can be executed to get the expected output. An alternate implementation is likely to have the same faults as the program-under-test, and some faults would not be detected via testing as a result.

H*euristic oracles* provide *approximate* results for inputs, and tests that "fail" must be scrutinised closely to check whether they are a true or false positive. E.g. shortest path on map may be the straight line, but sometimes the straight line is impossible.

Golden program is an executable specification, a previous versions of the same program, or test in parallel with a trusted system that delivers the same functionality.

Property-based testing combines the idea of (usually passive) test oracles with techniques such as random testing to run a large amount of test inputs, and check their output against certain properties.
The general template of a property-based test is as follows:

```
for all (x, y, z, ...)
such that condition(x, y, z) holds
property(x, y, z, ..., f(x, y, z, ...)) is true.
```

where x, y, z, ... are input variables, and f is a function that we are testing. condition is a predicate that returns true if some property is true for x, y, z, ..., while property is a test oracle that returns true if the some property holds true between the inputs in x, y, z, ... and the output of the program f(x, y, z, ...).
We can then write a test driver that generates random inputs for our variables x, y, z, etc., filters out those that do not hold for the condition (alternatively we can generate inputs that only hold for the condition), run those tests on f, observe the output, and then check that the property holds.
Metamorphic testing is one form of property-based testing. In metamorphic testing, condition is the relation, p(i,j), on the input, and property is the relation, q(oi,oj), on the outputs.

The Big-bang Integration Strategy
The *Big-Bang* method involves coding all of the modules in a sub-system, or indeed the whole system, separately and then combining them all at once. Each module is typically unit tested first, before integrating it into the system. The modules can be implemented in any order so programmers can work in parallel.
The problem is that if the integrated systems fails a test then it is often difficult to determine which module or interface caused the failure.
The other problem with a Big-Bang integration strategy is that the number of input combinations becomes extremely large — testing all of the possible input combinations is often impossible and consequently we there may be latent faults in the integrated sub-system

Top-Down Integration Strategy
The *top-down* method involves implementing and integrating the modules on which no other modules depend, first. The modules on which the these *top-level* modules depend are implemented and integrated next and so on until the whole system is complete.
The advantages over a Big-Bang approach are that the machine demand is spread throughout integration phase. Also, if a module fails a test then it is easier to isolate the faults leading to those failures. Because testing is done incrementally, is is more straightforward to explore the input for program faults.
The top-down integration strategy requires *stubs* to be written. A stub is an implementation used to stand in for some other program. A stub may simulate the behaviour of an existing implementation, or be a temporary substitute for a yet-to-be-developed implementation.

The Bottom-Up Integration Strategy
The *bottom-up* method is essentially the opposite of the top-down. Lowest-level modules are implemented first, then modules at the next level up are implemented forming subsystems and so on until the whole system is complete and integrated.
This is a common method for integration of object-oriented programs, starting with the testing and integration of base classes, and then integrating the classes that depend on the base classes and so on.
The bottom-up integration strategy requires *drivers* to be written. A driver is a piece of code use to supply input data to another piece of code. Typically, the piece modules being tested need to have input data supplied to them via their interfaces. The driver program typically calls the modules under test supplying the input data for the tests as it does so.

Threads-Based or Iterative Integration Strategy
The *threads-based* integration method attempts to gain all of the advantages of the top-down and bottom-up methods while avoiding their weaknesses. The idea is to is to select a minimal set of modules that perform a program function or program capability, called a **thread**, and then integrate and test this set of modules using either the top-down or bottom-up strategy.
Ideally, a thread should include some I/O and some processing where the modules are selected from all levels of the module hierarchy. Once a thread is tested and built other modules can be added to start building up a complete the system.
This idea is common in agile methodologies. Threads are focussed on user requirements, or user stories. Given a coherent piece of functionality from a set of user requirements, we build that small piece of functionality to deliver some value to the user. However, this is NOT an integration strategy on its own: we will still have to bring together parts of this as we go, using top-down and bottom-up strategies.

Unit testing
Unit could be functions sharing one state or a single function. Units, if they are functions or classes, are tested for correctness, completeness and consistency. Unit testing measures the attributes of units. They can be tested for performance but almost never for reliability; units are far too small in size to have the statistical properties required for reliability modelling.

Integration Testing
Integration testing tests collections of modules working together. Which collection of modules are integrated and tested depends on the integration strategy. The aim is to test the interfaces between modules to confirm that the modules interface together properly. Integration testing also aims to test that subsystems meet their requirements.

System Testing
System testing test the entire system against the requirements, use cases or scenarios from requirements specification and design goals.

Regression Testing
Regression testing refers to running the entire test suite of a system again after a change is made.

Exploratory Testing
Exploratory testing is a form of *unstructured* testing. It is done typically by experienced testers, preferably with some domain knowledge about the application. All they do is explore the software, using their experience from years of software engineering and what they have learnt about the piece of software during their exploratory testing to find faults.

Penetration testing (or *pentesting*) is the process of attacking a piece of software with the purpose of finding security vulnerabilities.

A **vulnerability** is a security hole in the hardware or software (including operating system) of a system that provides the possibility to attack that system; e.g., gaining unauthorised access. Vulnerabilities can be weak passwords that are easy to guess, buffer overflows that permit access to memory outside of the running process, or unescaped SQL commands that provide unauthorised access to data in a database.

SQL injection:
code: SELECT * FROM really_personal_details WHERE email = '$EMAIL_ADDRESS'
Input: myemail@somedomain.com' OR '1=1
then: SELECT * FROM really_personal_details WHERE email = 'myemail@somedomain.com' OR '1=1'

Buffer overflows are another common security vulnerability. These occur when data is written into a buffer (in memory) that is too small to handle the size of the data.

Fuzz testing (or *fuzzing*) is a (semi-)automated approach for penetration testing that involves the randomisation of input data to locate vulnerabilities.

Random fuzzing
tests are chosen according to some probability distribution (possible uniform) to permit a large amount of inputs to be generated in a fast and unbiased way.
Advantage:
- It is often (but not always) cheap and easy to generate random tests, and cheap to run many such tests automatically.
- Is is *unbiased*, unlike tests selected by humans. This is useful for pentesting because the cases that are missed during programming are often due to lack of human understanding, and random testing may search these out.

Disadvantages
- A prohibitively large number of test inputs may need to be generated in order to be confident that the input domain has been adequately covered.
- The distribution of random inputs simply misses the program faults.
- It is highly unlikely to achieve good coverage. e.g. chance hit the following fault is 1 in 2^{32}

```
void good_bad(char s[4]) {
  int count = 0;
  if (s[0] == 'b') count++;
  if (s[1] == 'a') count++;
  if (s[2] == 'd') count++;
  if (s[3] == '!') count++;
  if (count > 3) abort();  //fault
}
```

Mutation-based fuzzing
takes valid test inputs, and mutates small parts of the input, generating (possibly invalid) test inputs.
Advantage:
- It generally achieves higher code coverage than random testing. It takes shorter time to find the previous good_bad vulnerability.

Disadvantages
- The success is highly dependent on the valid inputs that are mutated.
- It still suffers from low code coverage due to unlikely cases (but not to the extent of random testing).

The inputs for mutation-based fuzzing is called seed.

Generation-based fuzzers (or *intelligent fuzzers*)
typically generate their own input from such existing models, rather than mutating existing input (although many tools combine the two approaches). It has knowledge about the input format and can generate random input within required structure.
Advantages
- Knowledge the input protocol means that valid sequences of inputs can be generated that explore parts of the program, thus generally giving higher coverage.

Disadvantages
- Compared to random testing and mutation fuzzing, it requires some knowledge about the input protocol.
- The setup time is generally much higher, due to the requirement of knowing the input protocol; although in some cases, the grammar may already be known (e.g., XML, RFC).

Memory debugger
a tool for finding memory leaks and buffer overflows.
Generally, memory debuggers monitor looking for four issues:
1. Uninitialised memory: references made to memory blocks that are uninitialised at the time of reference.
2. Freed memory: reads and writes to/from memory blocks that have been freed.
3. Memory overflows: writes to memory blocks past the end of the block being written to.
4. Memory leaks: memory that is allocated but no longer able to be referenced.

Performance cost is expensive: The overhead of keeping track of the allocated memory, plus the checks made

Undefined Behaviour
Buffer overflows are a form of *undefined behaviour*: something that a program does that causes its future behaviour to be unknown. It might continue working or it might do something totally unpredictable, such as executing attacker-supplied code in the case of a successful remote code execution attack.
e.g. Dividing by 0, Dereferencing a NULL pointer, Overflowing a signed integer, Underflowing a signed integer

Code Coverage-guided fuzzing

```
1. $interesting \leftarrow initial\_test\_seeds$
2. $seen \leftarrow \emptyset$
3. while true do
4.     Choose an input i from interesting
5.     $input \leftarrow MUTATE(i)$
6.     $path \leftarrow EXECUTE(input)$
7.     if $\{path\} \not\subset seen$ then
8.         $interesting \leftarrow interesting \cup \{input\}$
9.         $seen \leftarrow seen \cup \{path\}$
```

Advantages
- Good coverage can be achieved even with little to no knowledge of the input format
- Low set-up time
- The technique can be widely applied

Disadvantages
- Coverage can still be limited in programs that perform multi-byte equality tests (e.g. that compare two 4-byte integers), since such comparisons provide no incremental feedback to the fuzzer to allow it to discover the needed input to pass the test.
- Performance overhead is higher than with fuzzing techniques that use no feedback loop.
- For programs whose input format is known, generation-fuzzing and similar techniques are likely to perform better since they can avoid the drawbacks of coverage-guided fuzzing.

Constraint solving
x > 0, y < 0, x + y > 100 is solvable constraint
x < 0, y < 0, x + y > 0 is not solvable constraint
For checking constraint entailment, we ask the solver if constraint C *entails* constraint D, written C⊢D. e.g. x > 0 ⊢ x ≥ 0. That is, any value of x that satisfies x > 0 also satisfied x ≥ 0. However, the inverse, x ≥ 0 ⊢ x > 0, does not hold, because x = 0 is a solution for the former but not the latter.

Symbolic state
Constraint can be used to to assign symbolic value. e.g. x < 0 or x = 15
At any point during execution of a program, the program will have a *state*, which is a mapping from all variables to their concrete values at that time. As a program executes, calculations (instructions) are executed on the variables in that state, and their values are updated.

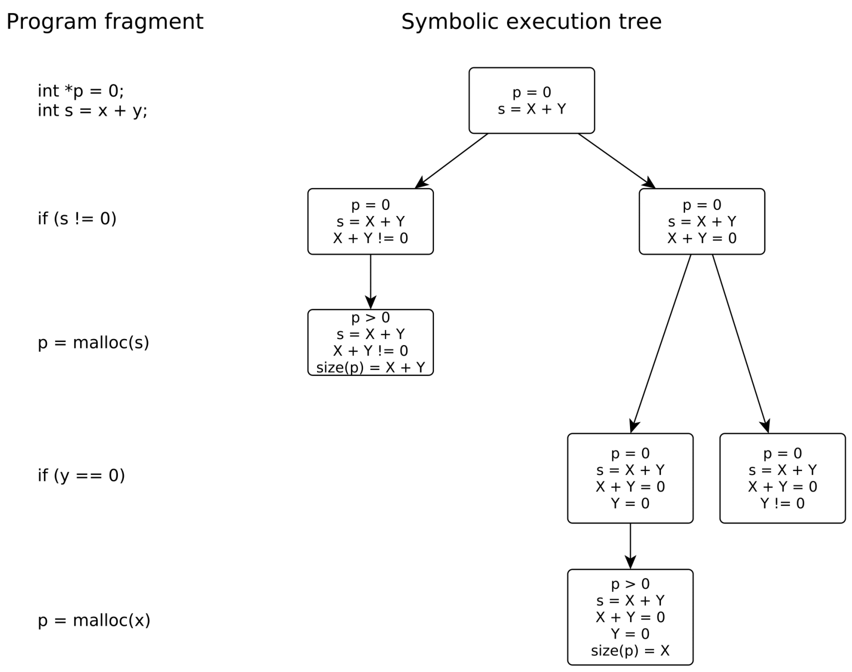
Symbolic execution

```
1.  void swap(int x, int y)
2.  {
3.    if (x < y) {
4.      x = x + y;
5.      y = x - y;
6.      x = x - y;
7.    }
8.    assert(x >= y);
9.  }
```

Line	Program fragment	Variable values		Notes
		x	y	
1.	`void swap(int x, int y)`	4	10	
3.	`if (x < y)`	4	10	(branch true)
4.	`x = x + y`	14	10	
5.	`y = x - y`	14	4	
6.	`x = x - y`	10	4	
8.	`assert(x >= y)`	10	4	(assertion true)

Symbolic execution tree

```
1. int* func(int x; int y)
2. {
3.   int* p = 0;
4.   int s = x + y;
5.   if (s != 0) {
6.     p = malloc(s);
7.   }
8.   else if (y == 0) {
9.     p = malloc(x);
10.  }
11.  return p;
12. }
```



Symbolic Execution limitations
Bad performance: cost is 80 times that of concrete execution
Path explosion: If we want our symbolic execution tool to explore all possible paths through a program, the resulting number of paths explodes exponentially as the number of branches in the program increases. Exhaustively symbolically executing every path through some real program is not feasible.
Loops: programs with loops has infinite paths
Unsolvable paths: A final issue is with unsolvable paths: those paths that cannot be symbolically executed, effectively halting exploration of the path.

Dynamic symbolic execution
Essentially, a DSE tool runs a concrete input on a program, and simultaneously performs symbolic execution on the path that this input executes. It then uses the path constraint generated by the symbolic execution to generate a new test.
Say for the following program starting with random input good

```
void good_bad(char s[4]) {
  int count = 0;
  if (s[0] == 'b') count++;
  if (s[1] == 'a') count++;
  if (s[2] == 'd') count++;
  if (s[3] == '!') count++;
  if (count > 3) abort();  //fault
}
```

Each time it will generate a new test by negating one of the branches in the path constraint. We can systematically repeat this for all 16 combinations of branches, symbolically executing each time, which will provide us with complete path coverage. One of the 16 combinations will be the
constraint s0= b, s1= a, s2= d, s3= !, which generates the input “bad!”, and will execute the fault.
However, the actual number of new input generated depends on the implementation of the function foo and the heuristics applied by the selected DSE tool.

x == hash(s) is symbolically executable, DSE can solve it by random input s and random input x, execute it, then negate by x = hash(s) from result just got
hash(x) == hash(y) is not symbolically executable, DSE cannot solve it

Advantages over random testing

- It is not so random comparing to random testing. While the first test case is random, using the program’s control-flow to explore the input space provides a much better coverage of the program structure than with random testing.

Advantages over symbolic execution

- when an unsolvable path (e.g. unsolvable constraint or external function call) is encountered, DSE uses the concrete values for those variables that cannot be solved, and adds these to the symbolic state. This effectively under-approximates the symbolic values of those variables, but it allows execution to continue.
- It opens up to new search strategies to deal with the path explosion problem.

Limitations
DSE still suffers from all of the same limitations as symbolic execution: path explosion, expensive constraint solving, loops, and unsolvable paths. However, the use of concrete information during execution helps to mitigate all of these at some level, significantly improving the scalability.
However, DSE suffers from a problem that is not found in symbolic execution: *divergence*. Because a concrete input is used to drive which path is executed next, a DSE algorithm does not fork at each branch, but instead just collects the symbolic values of the path being executed. However, if a new test is generated and the path it executes is different to the path we expected, we say that a *divergence* has occurred. This is caused by the limitations in the constraint solvers used.
Divergence e.g. start with x = 0, y = 1 -> x = 2, y = 1, then It will miss abort when x = 3 y = 1

```
void divergent(int x, int y)
{
  int s[4];
  s[0] = x;
  s[1] = 0;
  s[2] = 1;
  s[3] = 2;
  if (s[x] = s[y] + 2) {
    abort(); //error
  }
}
```