272: Software Engineering
Winter 2024

Instructor: Tevfik Bultan

Lecture: Testing Overview, Foundations

Verification, Validation, Testing

- ***Verification***: Demonstration of consistency, completeness, and correctness of the software artifacts at each stage of and between each stage of the software life-cycle.
  - Different types of verification: manual inspection, testing, formal methods
  - Verification answers the question: ***Am I building the product right?***
- ***Validation***: The process of evaluating software at the end of the software development to ensure compliance with respect to the customer needs and requirements.
  - Validation can be accomplished by verifying the artifacts produced at each stage of the software development life cycle
  - Validation answers the question: ***Am I building the right product?***
- ***Testing***: Examination of the behavior of a program by executing the program on sample data sets.
  - Testing is a verification technique used at the implementation stage.

Verification/Testing Through Software Life-Cycle

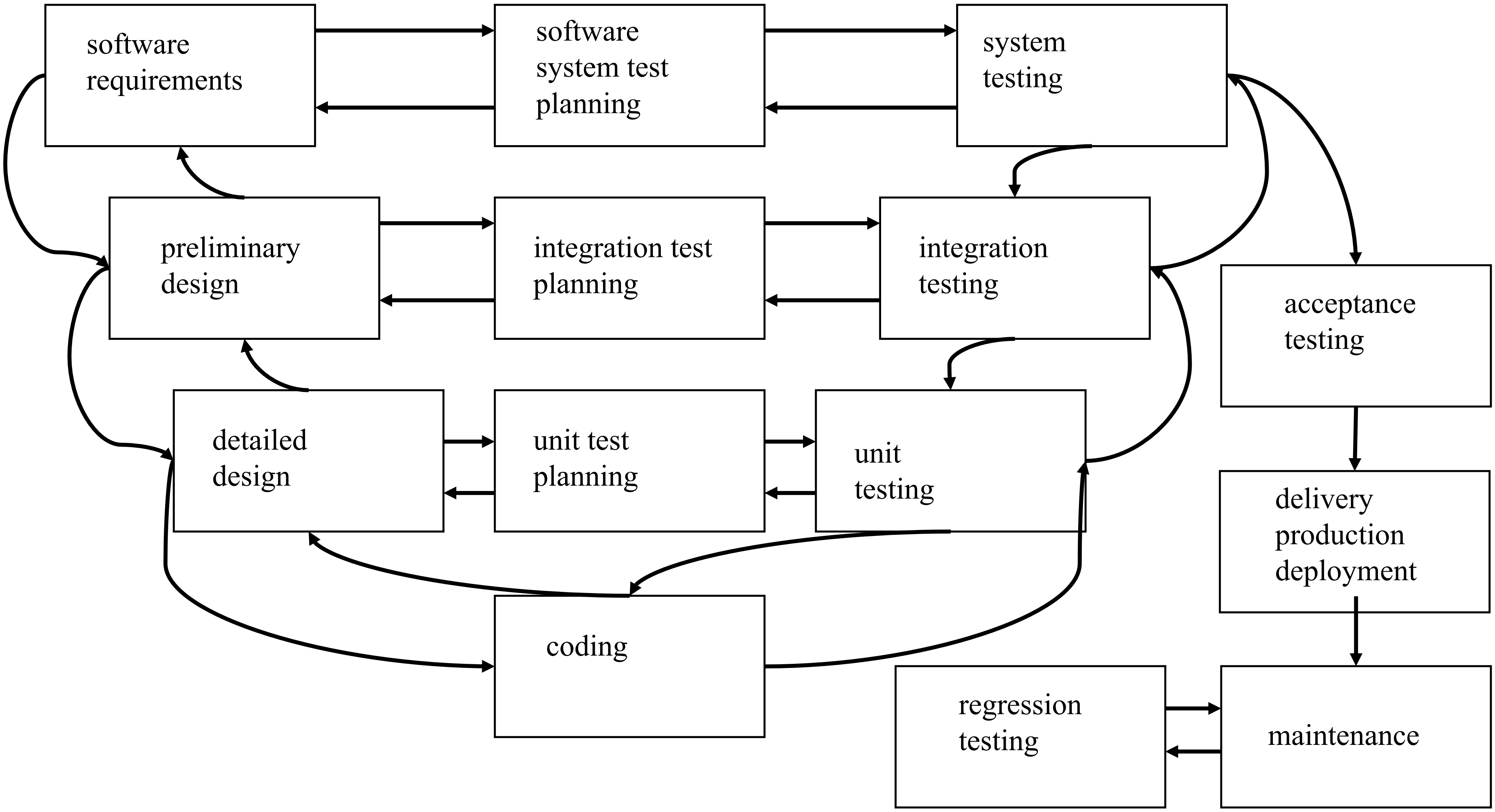
- Every phase of software life-cycle requires verification techniques to find errors (violating correctness), omissions (violating completeness), contradictions (violating consistency)
  - Requirements analysis and specification
    - use cases, scenarios of expected system use, help in establishing completeness, and can be used to generate test cases later on in the implementation stage
    - formal requirements specifications (for example: Statecharts, UML+OCL) can be checked for properties such as consistency and completeness automatically
    - As I mentioned earlier, late discovery of requirements errors is very costly

Verification/Testing Through Software Life-Cycle

- Design
  - Pre, post-conditions, class invariants can be used for verification at the detailed design stage
  - Design walk-throughs, design inspections, and design review

- Implementation
  - Program testing is one of the main verification tools at this stage
  - Code walk-throughs, code inspections, code review, audits
  - Dynamic analysis tools such as dynamic monitoring of assertions and dynamic design by contract monitoring
  - Static verification tools like ESC/Java Dafny

Relationships between different phases of software life-cycle and testing



software requirements

software system test planning

system testing

preliminary design

integration test planning

integration testing

acceptance testing

detailed design

unit test planning

unit testing

delivery production deployment

coding

regression testing

maintenance

Manual Verification:
Reviews, Walkthroughs, Inspections, Audits

Manual Verification

- When we have an executable program we can use testing methods for verification

- Can we find a way to check requirements specifications, design specifications, and source code?

- Manual verification techniques help in these situations:
  - ***Walkthroughs***, ***Inspections***, ***Reviews***, ***Audits***
  - Sometimes all of these techniques together are called Reviews

- These tasks are done typically in meetings, manually

- They are useful when no automated technique is available

General Characteristics

Each Review, Walk-through, Inspection and Audit should have the following three phases:

1. The planning phase
    - stating the purpose of the review
    - arranging the participants
    - ensuring that review materials are provided for their inspection well prior to the conduct of the review
    - making arrangements for the location and support required
    - preparing an agenda

General Characteristics

2. The meeting conduct phase
    - follow the agenda in a disciplined manner
    - identify problems and assign action for their resolution not try to fix them during the review
    - a moderator or review leader maintains control
    - a recorder has to be assigned to transcribe the proceedings for preparing a record of the meeting and post-meeting action list

3. The post-meeting phase
    - flexible depending on the actions required
    - actions are followed to completion by management and reported in the next review

Reviews

- **Review:** A process or meeting during which a work product, or a set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code review, design review, and requirements review.
- Characteristics for a review:
  - Review should generate a written report on status of the product reviewed—a report that is available to everyone involved in the project, including management;
  - Review requires active and open participation of everyone in the review group;
  - Review requires full responsibility of all participants for the quality of the review—that is, for the quality of the information in the written report.

Walk-through

- **Walk-through:** A manual static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems.
- Walk-throughs are a form of manual simulation
- Two variations
  - led by a reader or presenter who could be the person responsible for the product
  - led by a moderator independent of the person responsible for the product
- In a code walkthrough, you go over the code statement-by-statement explaining what each statement does

Walkthroughs

- Objectives
  - detect, identify, and describe software element defects
  - examine alternatives and stylistic issues
  - provide a mechanism that enables the authors to collect valuable feedback on their work, yet allows them to retain the decision-making authority for any changes

- Planning
  - identify the walkthrough team
  - select a place and schedule a meeting
  - distribute all necessary input materials to the participants, allowing for adequate preparation time

- Participants review the input material during the preparation phase

Walkthroughs

- During the meeting
  - author makes an overview presentation of the software element
  - the author walks through the specific software element so that member of the walkthrough team may ask questions or raise issues about the software element, and/or make notes documenting concerns
  - the recorder writes down comments and decision of inclusion in the walk-through report
- Output: the walkthrough-report contains
  - identification of the walkthrough team
  - identification of the software elements examined
  - the statement of objectives that were to be handled during the walkthrough meeting
  - A list of noted deficiencies, omissions, contradictions, and suggestions for improvement

Inspection

- **Inspection:** A manual static analysis technique that relies on visual examination of development products to detect errors, violations of developing standards, and other problems. Types include code inspection; design inspections.
- Inspections are used to manually check for common errors
- A method of rapidly evaluating material by confining attention to a few selected aspects, one at a time.
- In an inspection, the inspector uses a rigid set of guidelines or a checklist to assess the degree of compliance with the checklist or guidelines
- In a code inspection you have a checklist that looks for errors such as uninitialized variables, division by zero etc. and check each item in the checklist one by one

Inspections

- Objective is to detect defects in the product by comparison with a checklist that typifies the types of defects that are common to the type of product being inspected.
- There is a moderator and inspectors, the developer of the product and a recorder
- Planning:
  - moderator makes arrangements: the materials to be inspected, the checklists to be used, selecting a place and scheduling the meeting
- The moderator controls the meeting by walking through the code
- A checklist is used to identify the defects

Inspections

- A typical inspection checklist for code inspections may include:
  - wrongful use of data: unintialized variables, array index out of bounds, dangling pointers
  - faults in declarations: use of undeclared variables, declaration of the same name in nested blocks
  - faults in computations: division by zero, overflow, wrong use of variables of different types in one and the same expression, faults caused by an erroneous conception of operator priorities
  - faults in relational expressions: using an incorrect operator, an erroneous conception of priorities of Boolean operators
  - faults in control flow: infinite loops, a loop that gets executed n+1 or n-1 times rather than n
  - faults in interfaces: incorrect number of parameters, parameters of the wrong types, inconsistent use of global variables

Audits

- **Audit:** An independent examination of a work product or a set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria.
- Similar to inspections but
  - More interactive than inspections
  - Less structured than inspections
- You can consider your project demo an audit

Manual Verification Software Life-Cycle

Software Requirements Analysis

Requirements Walkthroughs

Requirements Specification Inspection

Software Requirements Review

Preliminary Design

Design Walkthroughs

Preliminary Design Review

Detailed Design

Design Specification Inspection

Critical Design Review

Coding and Unit Test

Code Walkthroughs

Code Inspections

System Test and Integration

User Manual Inspection

Functional and Physical Configuration Audits

Software Testing

Software Testing

- Correctness
  - software should match its specifications
  - software should meet its functional requirements

- Testing is necessary because we cannot guarantee correctness in the software development process

- Testing: techniques of checking software correctness by executing the software on some data sets

Software Testing

- Goal of testing
  - finding faults in the software
  - demonstrating that there are no faults in the software

- It is not possible to ***prove*** that there are no faults in the software using testing

- Testing should help locate errors, not just detect their presence
  - a “yes/no” answer to the question “is the program correct?” is not very helpful

- Testing should be repeatable
  - could be difficult for distributed or concurrent software
  - effect of the environment, uninitialized variables

Testing Software is Hard

- If you are testing a bridge's ability to sustain weight, and you test it with 1000 tons you can infer that it will sustain weight ≤ 1000 tons

- This kind of reasoning does not work for software systems
  - software systems are not linear nor continuous

- Exhaustively testing all possible input output combinations is too expensive
  - the number of test cases increase exponentially with the number of input/output variables

Some Definitions

- Let *P* be a program and let *D* denote its input domain
- A **test case** *d* is an element of input domain $d \in D$
    - a test case gives a valuation for all the input variables of the program
- A **test set** *T* is a finite set of test cases, i.e., a subset of *D*, $T \subseteq D$
- The basic difficulty in testing is finding a test set that will uncover the faults in the program
- Exhaustive testing corresponds to setting $T = D$

Exhaustive Testing is Hard

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

- Number of possible test cases (assuming 32 bit integers)
  - $2^{32} \times 2^{32} = 2^{64}$
- Do bigger test sets help?
  - Test set
    {(x=3,y=2), (x=2,y=3)}
    will detect the error
  - Test set
    {(x=3,y=2),(x=4,y=3),(x=5,y=1)}
    will not detect the error although it has more test cases
- It is not the number of test cases
- But, if $T_1 \supseteq T_2$, then T_1 will detect every fault detected by T_2

Exhaustive Testing

- Assume that the input for the `max` procedure was an integer array of size *n*
  - Number of test cases: $2^{32 \times n}$

- Assume that the size of the input array is not bounded
  - Number of test cases: ∞

- The point is, exhaustive testing is infeasible in many cases

Random Testing

- Use a random number generator to generate test cases

- Derive estimates for the reliability of the software using some probabilistic analysis

- Coverage is a problem

Generating Test Cases Randomly

```
bool isEqual(int x, int y)
{
  if (x = y)
    z := false;
  else
    z := false;
  return z;
}
```

- If we pick test cases randomly it is unlikely that we will pick a case where x and y have the same value
- If x and y can take 2^{32} different values, there are 2^{64} possible test cases. In 2^{32} of them x and y are equal
  - probability of picking a case where x is equal to y is 2^{-32}
- It is not a good idea to pick the test cases randomly (with uniform distribution) in this case

Testing

- Testing can be categorized in different ways:
  - ***Functional vs. Structural testing***
    - Functional testing: Generating test cases based on the functionality of the software
    - Structural testing: Generating test cases based on the structure of the program
  - ***Black box vs. White box testing***
    - Black box testing is same as functional testing. Program is treated as a black box, its internal structure is hidden from the testing process.
    - White box testing is same as structural testing. In white box testing internal structure of the program is taken into account
  - ***Module (Unit) vs. Integration testing***
    - Module testing: Testing the modules of a program in isolation
    - Integration testing: Testing an integrated set of modules

Functional Testing, Black-Box Testing

- Functional testing:
  - identify the the functions which software is expected to perform
  - create test data which will check whether these functions are performed by the software
  - no consideration is given how the program performs these functions, program is treated as a black-box: **black-box testing**
  - need an ***oracle***: oracle states precisely what the outcome of a program execution will be for a particular test case. This may not always be possible, oracle may give a range of plausible values
- A systematic approach to functional testing: requirements based testing
  - driving test cases automatically from a formal specification of the functional requirements

Domain Testing

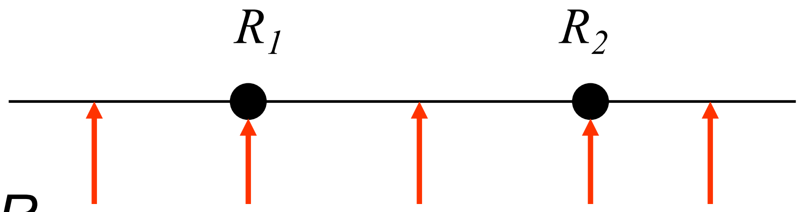
- Partition the input domain to equivalence classes
- For some requirements specifications it is possible to define equivalence classes in the input domain
- Here is an example: A factorial function specification:
  - *If the input value n is less than 0 then an appropriate error message must be printed. If* $0 \leq n < 20$*, then the exact value n! must be printed. If* $20 \leq n \leq 200$*, then an approximate value of n! must be printed in floating point format using some approximate numerical method. The admissible error is 0.1% of the exact value. Finally, if* $n > 200$*, the input can be rejected by printing an appropriate error message.*
- Possible equivalence classes: $D_1 = \{n<0\}$, $D_2 = \{0 \leq n < 20\}$, $D_3 = \{20 \leq n \leq 200\}$, $D_4 = \{n > 200\}$
- Choose one test case per equivalence class to test

Equivalence Classes

- If the equivalence classes are disjoint, then they define a partition of the input domain
- If the equivalence classes are not disjoint, then we can try to minimize the number of test cases while choosing representatives from different equivalence classes
- Example: D_1 = {x is even}, D_2 = {x is odd}, D_3 = {$x \leq 0$}, D_4={$x > 0$}
  - Test set {x=48, x= –23} covers all the equivalence classes
- On one extreme we can make each equivalence class have only one element which turns into exhaustive testing
- The other extreme is choosing the whole input domain D as an equivalence class which would mean that we will use only one test case

Testing Boundary Conditions

- For each range [R_1, R_2] listed in either the input or output specifications, choose five cases:
  - Values less than R_1
  - Values equal to R_1
  - Values greater than R_1 but less than R_2
  - Values equal to R_2
  - Values greater than R_2



- For unordered sets select two values
  - 1) in, 2) not in
- For equality select 2 values
  - 1) equal, 2) not equal
- For sets, lists select two cases
  - 1) empty, 2) not empty

Testing Boundary Conditions

- For the factorial example, ranges for variable *n* are:
  - [−∞, 0], [0,20], [20,200], [200, ∞]
  - A possible test set:
    - {n = -5, n=0, n=11, n=20, n= 25, n=200, n= 3000}
  - If we know the maximum and minimum values that *n* can take we can also add those *n*=MIN, *n*=MAX to the test set.

Structural Testing, White-Box Testing

- Structural Testing
  - the test data is derived from the structure of the software

  - **white-box testing**: the internal structure of the software is taken into account to derive the test cases

- One of the basic questions in testing:
  - when should we stop adding new test cases to our test set?
  - Coverage metrics are used to address this question

Coverage Metrics

- Coverage metrics
  - ***Statement coverage***: all statements in the programs should be executed at least once
  - ***Branch coverage***: all branches in the program should be executed at least once
  - ***Path coverage***: all execution paths in the program should be executed at lest once
- The best case would be to execute all paths through the code, but there are some problems with this:
  - the number of paths increases fast with the number of branches in the program
  - the number of executions of a loop may depend on the input variables and hence may not be possible to determine
  - most of the paths can be infeasible

Statement Coverage

- Choose a test set *T* such that by executing program *P* for each test case in *T*, each basic statement of *P* is executed at least once
- Executing a statement once and observing that it behaves correctly is not a guarantee for correctness, but it is an heuristic
  - this goes for all testing efforts since in general checking correctness is undecidable

```
bool isEqual(int x, int y)
{
  if (x = y)
    z := false;
  else
    z := false;
  return z;
}

int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

Statement Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
    print(“x is positive”);
  else
    print(“x is negative”);
  if (y >= 0)
    print(“y is positive”);
  else
     print(“y is negative”);
}
```

Following test set will give us statement coverage:
$T_1 = \{(x=12,y=5), (x=-1,y=35), (x=115,y=-13),(x=-91,y=-2)\}$

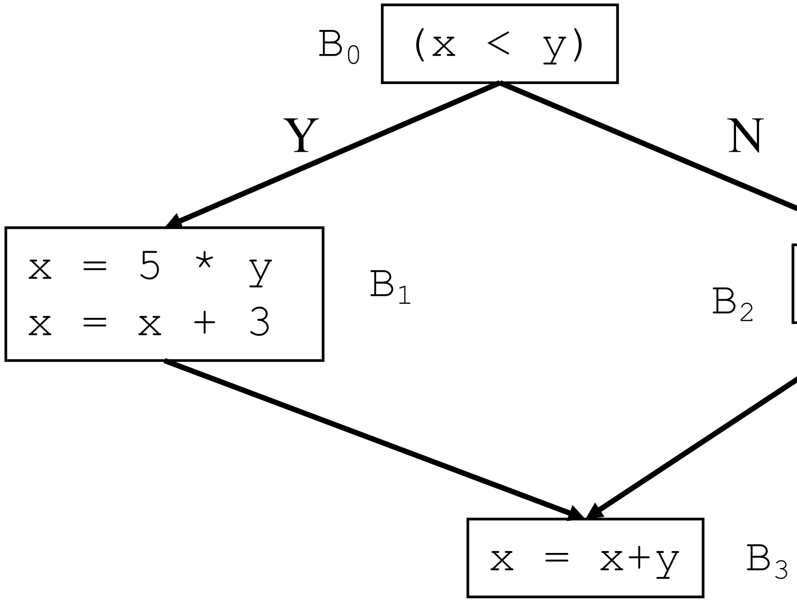
There are smaller test cases which will give us statement coverage too:
$T_2 = \{(x=12,y=-5), (x=-1,y=35)\}$

There is a difference between these two test sets though

Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A ***basic block*** is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

```
if (x < y) {
   x = 5 * y;
   x = x + 3;
}
else
   y = 5;
x = x+y;
```



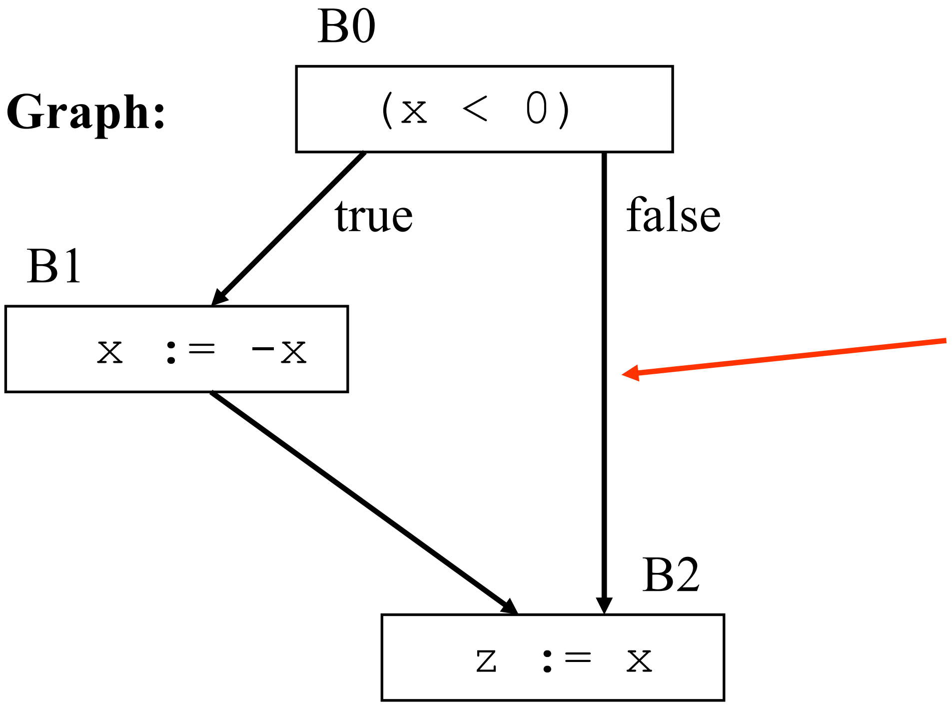
- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Statement vs. Branch Coverage

```
assignAbsolute(int x)
{
  if (x < 0)
    x := -x;
  z := x;
}
```

Consider this program segment, the test set T = {x=−1} will give statement coverage, however not branch coverage

Control Flow Graph:

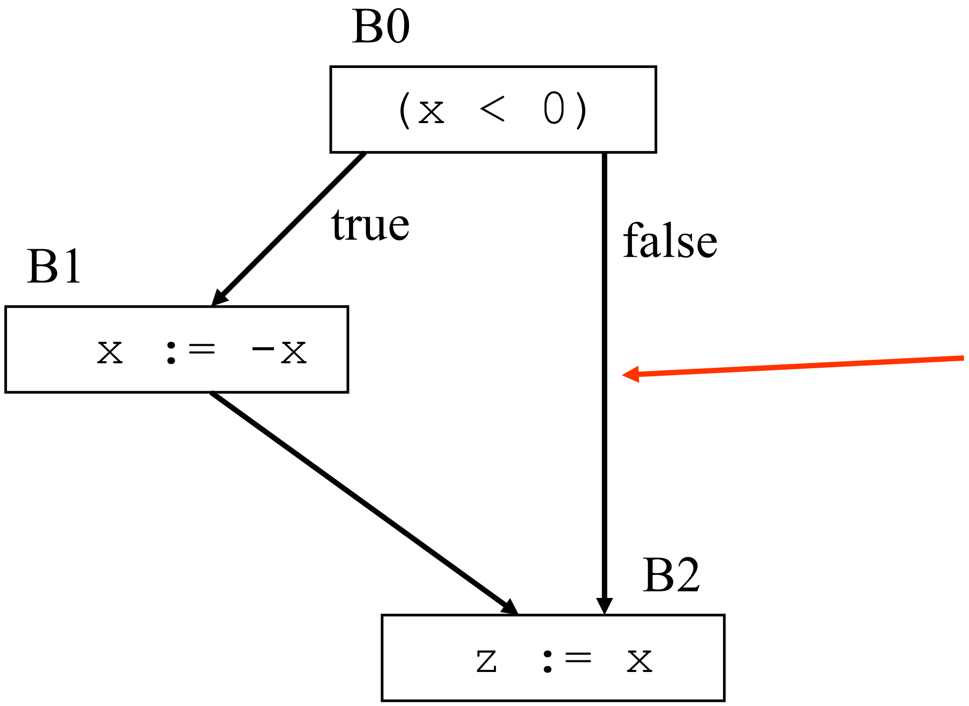


Test set {x=−1} does not execute this edge, hence, it does not give branch coverage

Branch Coverage

- Construct the control flow graph

- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, each edge of *P*'s control flow graph is traversed at least once



Test set {x=−1} does not execute this edge, hence, it does not give branch coverage

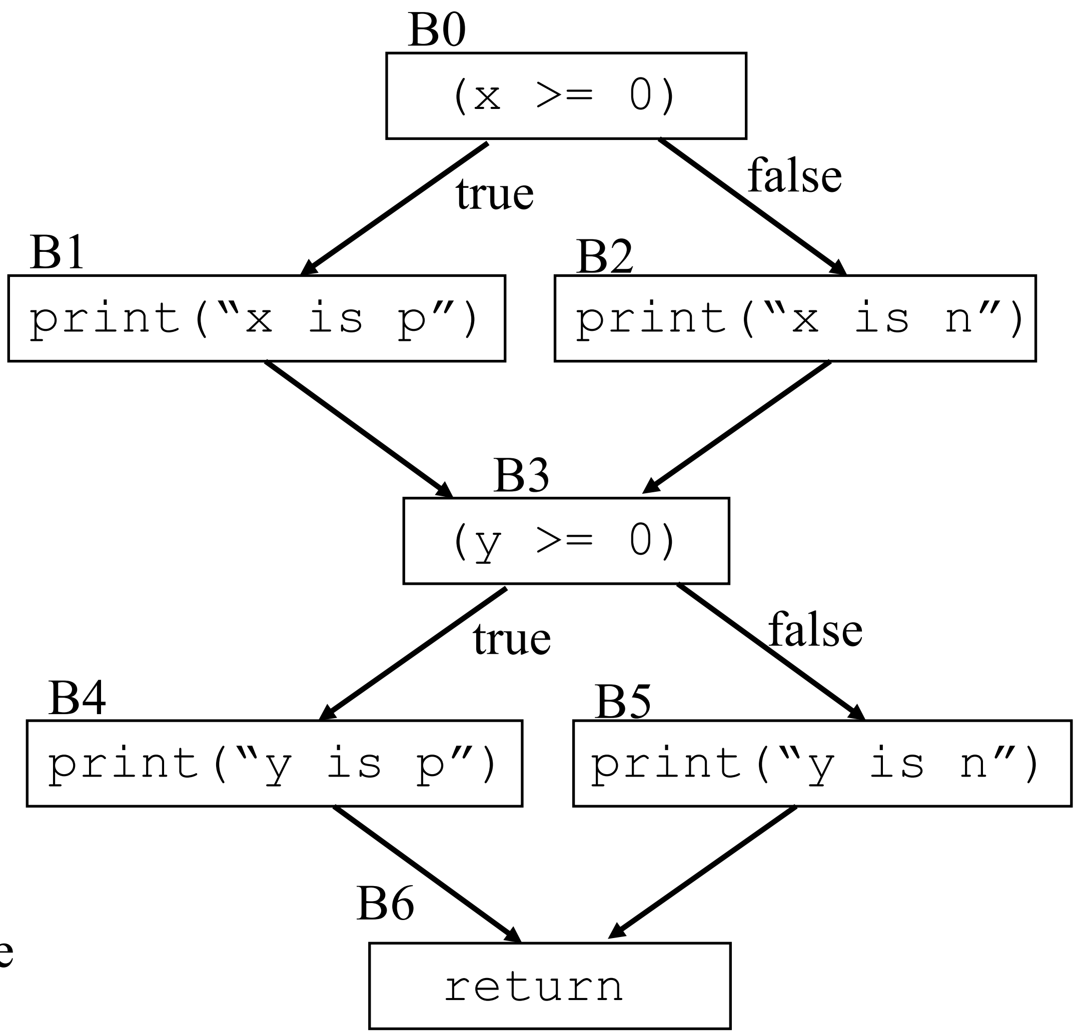
Test set {x=−1, x=2}gives both statement and branch coverage

Path Coverage

- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, all paths leading from the initial to the final node of P's control flow graph are traversed

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print(“x is positive”);
  else
    print(“x is negative”);
  if (y >= 0)
    print(“y is positive”);
  else
     print(“y is negative”);
}
```



Test set:
$T_2 = \{(x=12, y=-5), (x=-1, y=35)\}$
gives both branch and statement
coverage but it does not give path coverage

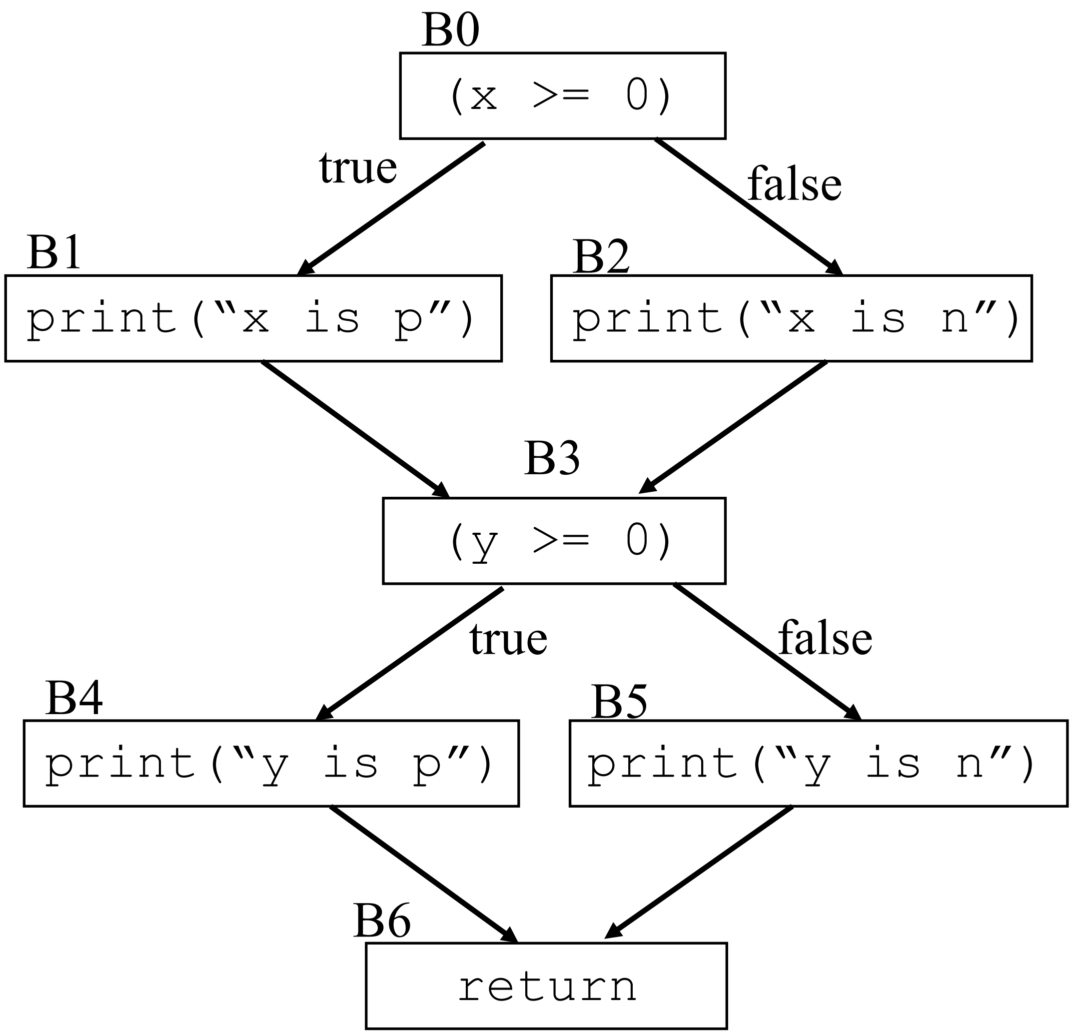
Set of all execution paths: {(B0,B1,B3,B4,B6), (B0,B1,B3,B5,B6), (B0,B2,B3,B4,B6), (B0,B2,B3,B5,B6)}

Test set T_2 executes only paths: (B0,B1,B3,B5,B6) and (B0,B2,B3,B4,B6)

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```

Test set:
T_1 = {(x=12,y=5), (x=−1,y=35), (x=115,y=−13),(x=−91,y=−2)} gives both branch, statement and path coverage



Path Coverage

- Number of paths is exponential in the number of conditional branches
  - testing cost may be expensive

- Note that every path in the control flow graphs may not be executable
  - It is possible that there are paths which will never be executed due to dependencies between branch conditions

- In the presence of cycles in the control flow graph (for example loops) we need to clarify what we mean by path coverage
  - Given a cycle in the control flow graph we can go over the cycle arbitrary number of times, which will create an infinite set of paths
  - Redefine path coverage as: each cycle must be executed 0, 1, ..., k times where k is a constant (k could be 1 or 2)

Condition Coverage

- In the branch coverage we make sure that we execute every branch at least once
  - For conditional branches, this means that, we execute the TRUE branch at least once and the FALSE branch at least once
- Conditions for conditional branches can be compound boolean expressions
  - A compound boolean expression consists of a combination of boolean terms combined with logical connectives AND, OR, and NOT
- Condition coverage:
  - Select a test set T such that by executing program P for each test case d in T,

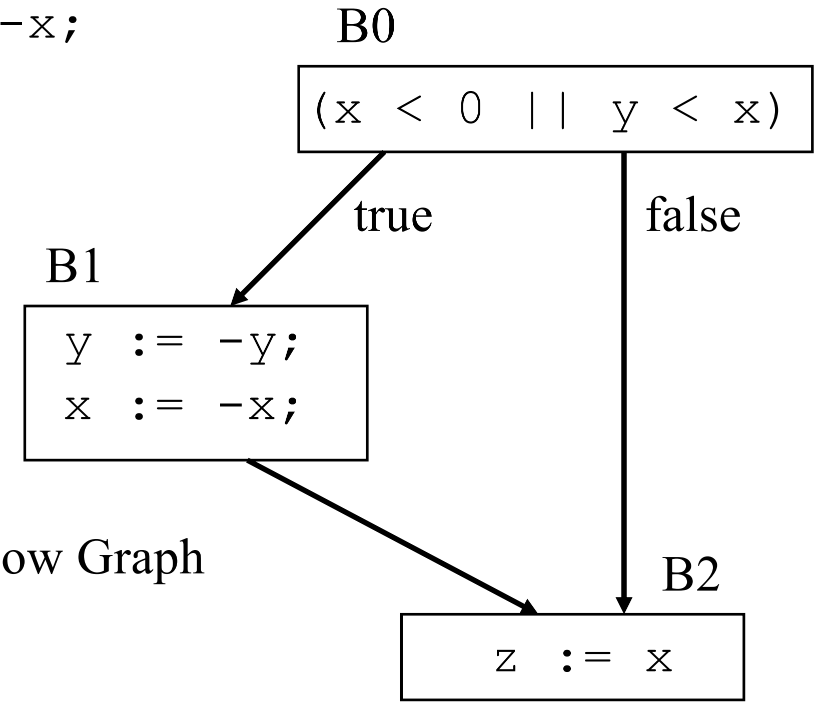
    (1) *each edge of P's control flow graph is traversed at least once and*

    (2) *each boolean term that appears in a branch condition takes the value TRUE at least once and the value FALSE at least once*
- Condition coverage is a refinement of branch coverage (part (1) is same as the branch coverage)

Condition Coverage

```
something(int x)
{
   if (x < 0 ||  y < x)
   {
      y := -y;
      x := -x;
   }
   z := x;
}
```

$T = \{(x=-1, y=1), (x=1, y=1)\}$ will achieve statement, branch and path coverage, however T will not achieve condition coverage because the boolean term `(y < x)` never evaluates to true. This test set satisfies part (1) but does not satisfy part (2).



Control Flow Graph

$T = \{(x=-1, y=1), (x=1, y=0)\}$ will not achieve condition coverage either. This test set satisfies part (2) but does not satisfy part (1). It does not achieve branch coverage since both test cases take the true branch, and, hence, it does not achieve condition coverage by definition.

$T = \{(x=-1, y=-2), \{(x=1, y=1)\}$ achieves condition coverage.

Multiple Condition Coverage

- Multiple Condition Coverage requires that all possible combination of truth assignments for the boolean terms in each branch condition should happen at least once
- For example for the previous example we had:

$$\underbrace{x < 0}_{\text{term1}} \;\&\&\; \underbrace{y < x}_{\text{term2}}$$

- Test set {(x=−1, y=−2), (x=1, y=1)}, achieves condition coverage:
  - test case (x=−1, y=−2) makes term1=true, term2=true, and the whole expression evaluates to true (i.e., we take the true branch)
  - test case (x=1, y=1) makes term1=false, term2=false, and the whole expression evaluates to false (i.e., we take the false branch)
- However, test set {(x=−1, y= −2), (x=1, y=1)} does not achieve multiple condition coverage since we did not observe the following truth assignments
  - term1=true, term2=false
  - term1=false, term2=true

Types of Testing

- ***Unit (Module) testing***
  - testing of a single module in an isolated environment
- ***Integration testing***
  - testing parts of the system by combining the modules
- ***System testing***
  - testing of the system as a whole after the integration phase
- ***Acceptance testing***
  - testing the system as a whole to find out if it satisfies the requirements specifications

Unit Testing

- Involves testing a single isolated module

- Note that unit testing allows us to isolate the errors to a single module
  - we know that if we find an error during unit testing it is in the module we are testing

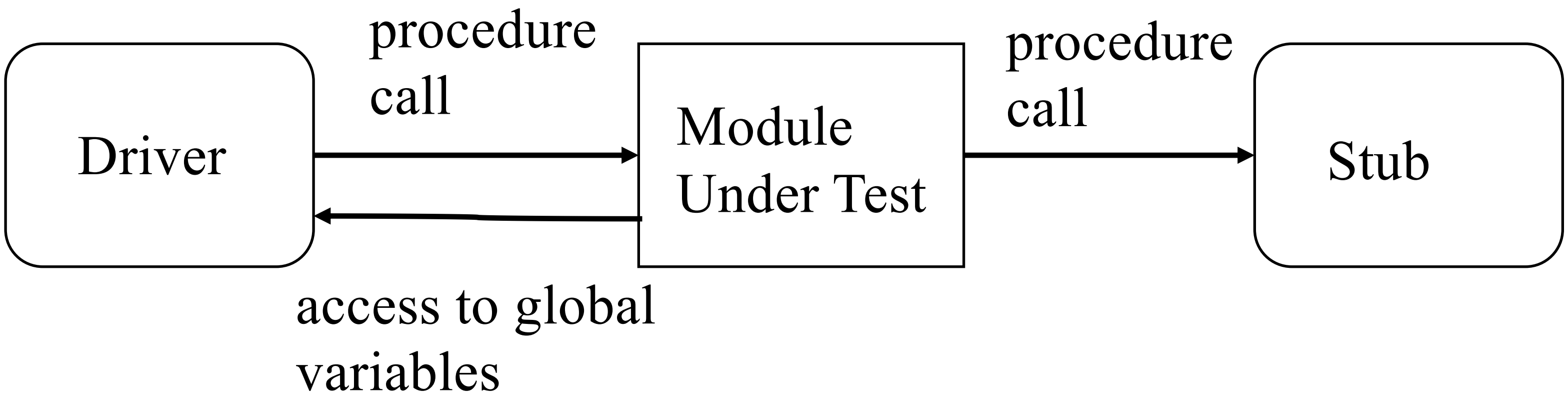
- Modules in a program are not isolated, they interact with each other. Possible interactions:
  - calling procedures in other modules
  - receiving procedure calls from other modules
  - sharing variables

- For unit testing we need to isolate the module we want to test, we do this using two things
  - drivers and stubs

Drivers and Stubs

- **Driver:** A program that calls the interface procedures of the module being tested and reports the results
  - A driver simulates a module that calls the module currently being tested
- **Stub:** A program that has the same interface as a module that is being used by the module being tested, but is simpler.
  - A stub simulates a module called by the module currently being tested

Drivers and Stubs



- Driver and Stub should have the same interface as the modules they replace

- Driver and Stub should be simpler than the modules they replace

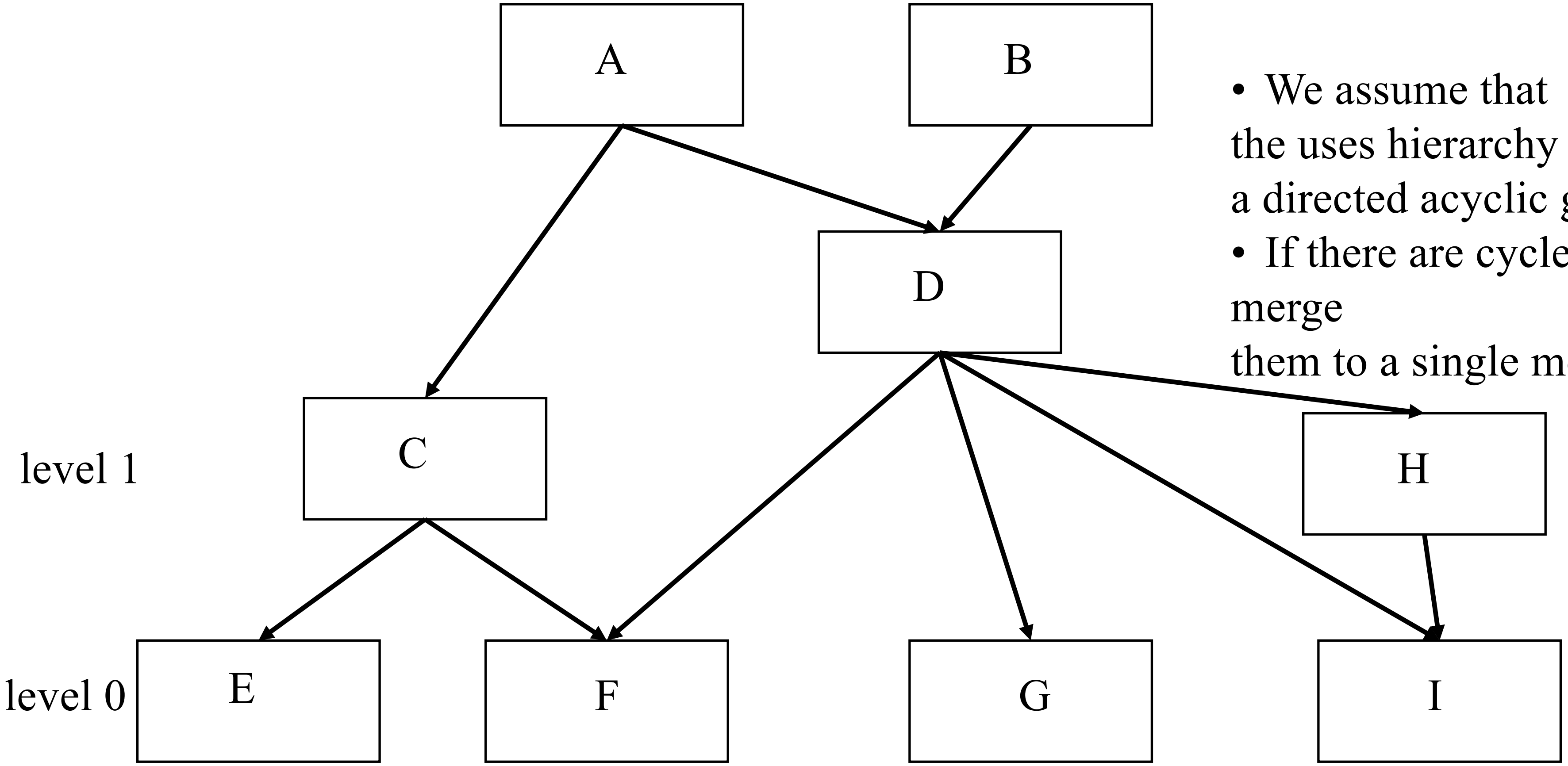
Integration Testing

- Integration testing: Integrated collection of modules tested as a group or partial system

- Integration plan specifies the order in which to combine modules into partial systems

- Different approaches to integration testing
  - Bottom-up
  - Top-down
  - Big-bang
  - Sandwich

Module Structure



- We assume that the uses hierarchy is a directed acyclic graph.
- If there are cycles merge them to a single module

- A uses C and D; B uses D; C uses E and F; D uses F, G, H and I; H uses I
- Modules A and B are at level 3; Module D is at level 2 Modules C and H are at level 1; Modules E, F, G, I are at level 0
- level 0 components do not use any other components
- level *i* components use at least one component on level *i*-1 and no component at a level higher than *i*-1

Bottom-Up Integration

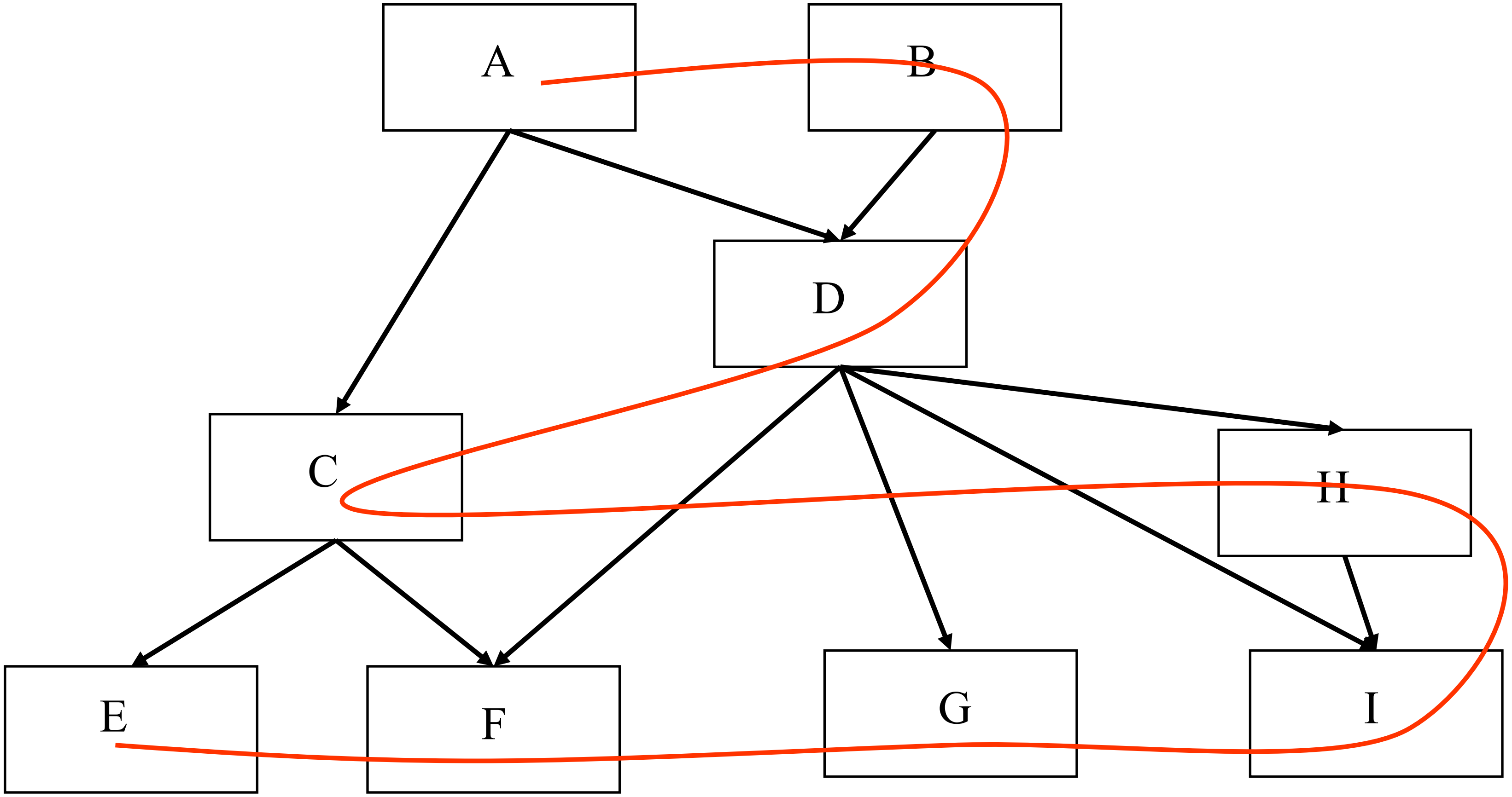
- Only terminal modules (i.e., the modules that do not call other modules) are tested in isolation

- Modules at lower levels are tested using the previously tested higher level modules

- Non-terminal modules are not tested in isolation

- ***Requires a module driver for each module*** to feed the test case input to the interface of the module being tested
  - However, ***stubs are not needed*** since we are starting with the terminal modules and use already tested modules when testing modules in the lower levels

Bottom-up Integration



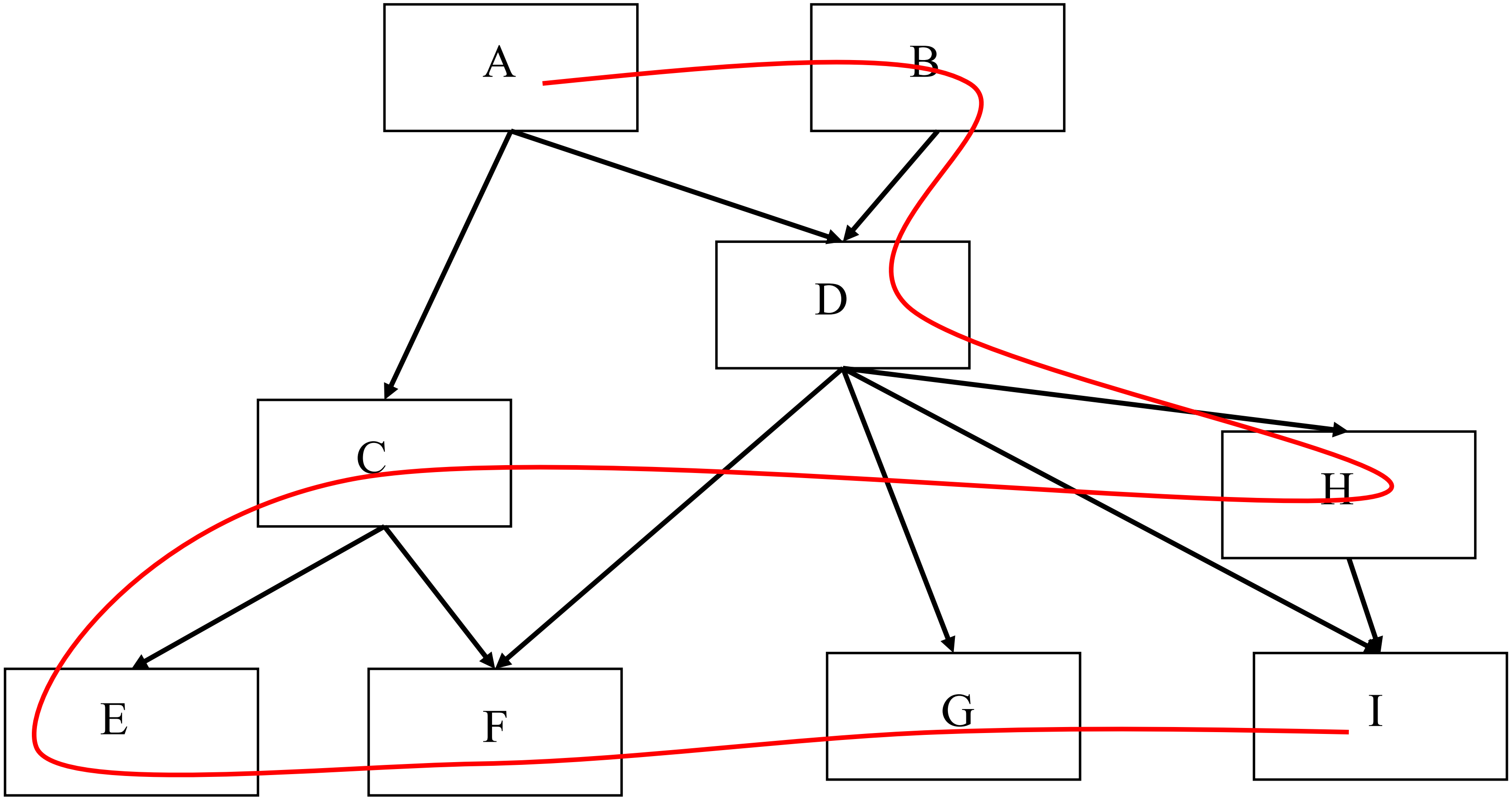
Top-down Integration

- Only modules tested in isolation are the modules which are at the highest level

- After a module is tested, the modules directly called by that module are merged with the already tested module and the combination is tested

- ***Requires stub modules*** to simulate the functions of the missing modules that may be called
  - However, ***drivers are not needed*** since we are starting with the modules which is not used by any other module and use already tested modules when testing modules in the higher levels

Top-down Integration



Other Approaches to Integration

- Sandwich Integration
  - Compromise between bottom-up and top-down testing
  - Simultaneously begin bottom-up and top-down testing and meet at a predetermined point in the middle

- Big Bang Integration
  - Every module is unit tested in isolation
  - After all of the modules are tested they are all integrated together at once and tested
  - No driver or stub is needed
  - However, in this approach, it may be hard to isolate the bugs!

System Testing, Acceptance Testing

- System and Acceptance testing follows the integration phase
  - testing the system as a whole

- Test cases can be constructed based on the the requirements specifications
  - main purpose is to assure that the system meets its requirements

- Manual testing
  - Somebody uses the software on a bunch of scenarios and records the results
  - Use cases and use case scenarios in the requirements specification would be very helpful here
  - manual testing is sometimes unavoidable: usability testing

System Testing, Acceptance Testing

- Alpha testing is performed within the development organization

- Beta testing is performed by a select group of friendly customers

- Stress testing
  - push system to extreme situations and see if it fails
  - large number of data, high input rate, low input rate, etc.

Regression testing

- You should preserve all the test cases for a program
- During the maintenance phase, when a change is made to the program, the test cases that have been saved are used to do **regression testing**
  - figuring out if a change made to the program introduced any faults
- Regression testing is crucial during maintenance
  - It is a good idea to automate regression testing so that all test cases are run after each modification to the software
- When you find a bug in your program you should write a test case that exhibits the bug
  - Then using regression testing you can make sure that the old bugs do not reappear

Test Plan

- Testing is a complicated task
  - it is a good idea to have a test plan

- A test plan should specify
  - Unit tests
  - Integration plan
  - System tests
  - Regression tests

Automation and Research in Software Testing

Automated Testing Techniques

Fuzzing / fuzz testing: An automated software testing technique that provides unexpected (or invalid) inputs to the program to find bugs

- Can be seen as an advanced version of random testing
- Especially useful in finding security vulnerabilities

Symbolic execution: Symbolically executes programs on undefined inputs (called symbols) and uses logic solvers to identify feasible path constraints (constraints on the inputs that correspond to a particular execution path)

- Can be used at function/method level, not scalable to large programs
- Can be seen as a verification technique and help prove properties about programs
- Can be combined with fuzzing to generate hybrid testing strategies

Automated Testing Techniques

Differential testing: Feeding the same input to two programs which are expected to have same functionality in order to detect variations in their behaviors.

- Addresses the oracle generation problem

Metamorphic testing: Given a program (and its expected functionality) find transformations on input with corresponding known transformations on output. Given a successful test case, transform the input and see the generated output by the program is equal to the transformed output.

- Addresses the oracle generation problem

Mutation testing / mutation analysis: Modify (mutate) program in small ways and see if the test suite detects (kills) the mutants

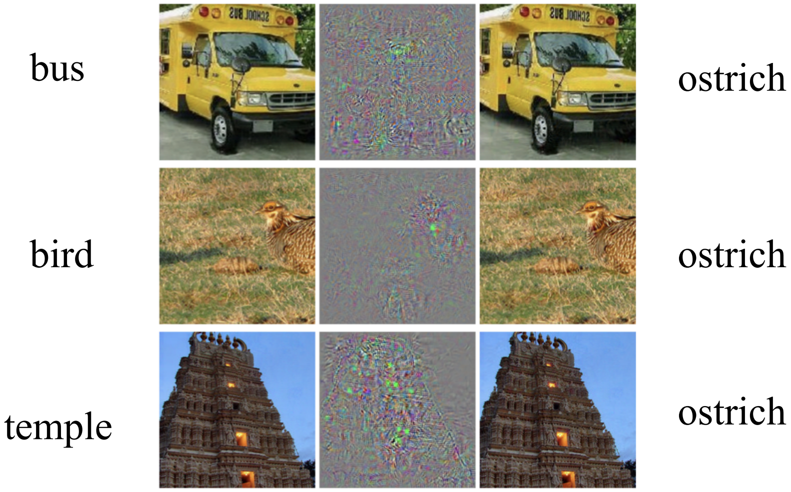
- Can be used to assess quality of a test suite

Mutation Analysis

- Mutation analysis is used to figure out the quality of a test set
- Mutation analysis creates **mutants of a program** by making changes to the program (change a condition, change an assignment, etc.)
- Each mutant program and the original program are executed using the test set
- If a mutant and the original program give different results for a test case then the test set detected that the mutant is different from the original program, hence the mutant is said to be dead
- If test set does not detect the difference between the original program and some mutants, these mutants are said to be live
- We want the test set to kill as many mutants as possible
  - Mutant programs can be equivalent to the original program, hence no test set can kill them

Testing in the Age of AI: Robustness

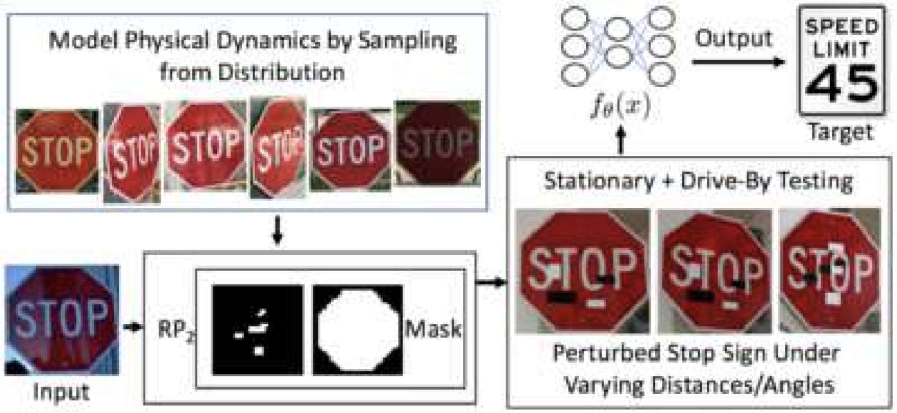
robustness failures



Christian Szegedy et al. ICLR 2014

Adding small perturbations to images that are not visible to humans can change the classification generated by Neural Network based classifiers (NNs)

robustness failures can be exploited in real world

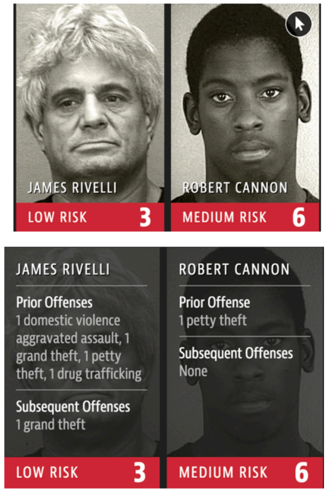


Kevin Eykholt et al. CVPR 2017

Testing in the Age of AI: Fairness

fairness failures

in recidivism risk assessment recommendations



hiring


REUTERS

Insight - Amazon scraps secret AI recruiting tool that showed bias against women

AI systems learn the unfair bias in the historic data and make recommendations based on that bias

Testing in the Age of AI

As the use of AI components in software increases, software testing techniques will need to be adapted to address weaknesses and vulnerabilities of AI components

- For example, software components that use Machine Learning techniques such as Deep Neural Networks need to be tested for robustness and fairness

Meanwhile, use of AI techniques in software development is also increasing.

- Copilot and ChatGPT provide code generation capabilities
- How should we test software generated using AI techniques?
- Can we trust software generated by AI techniques?

Currently, these are among the most significant research questions in software testing area

Formalizing Testing

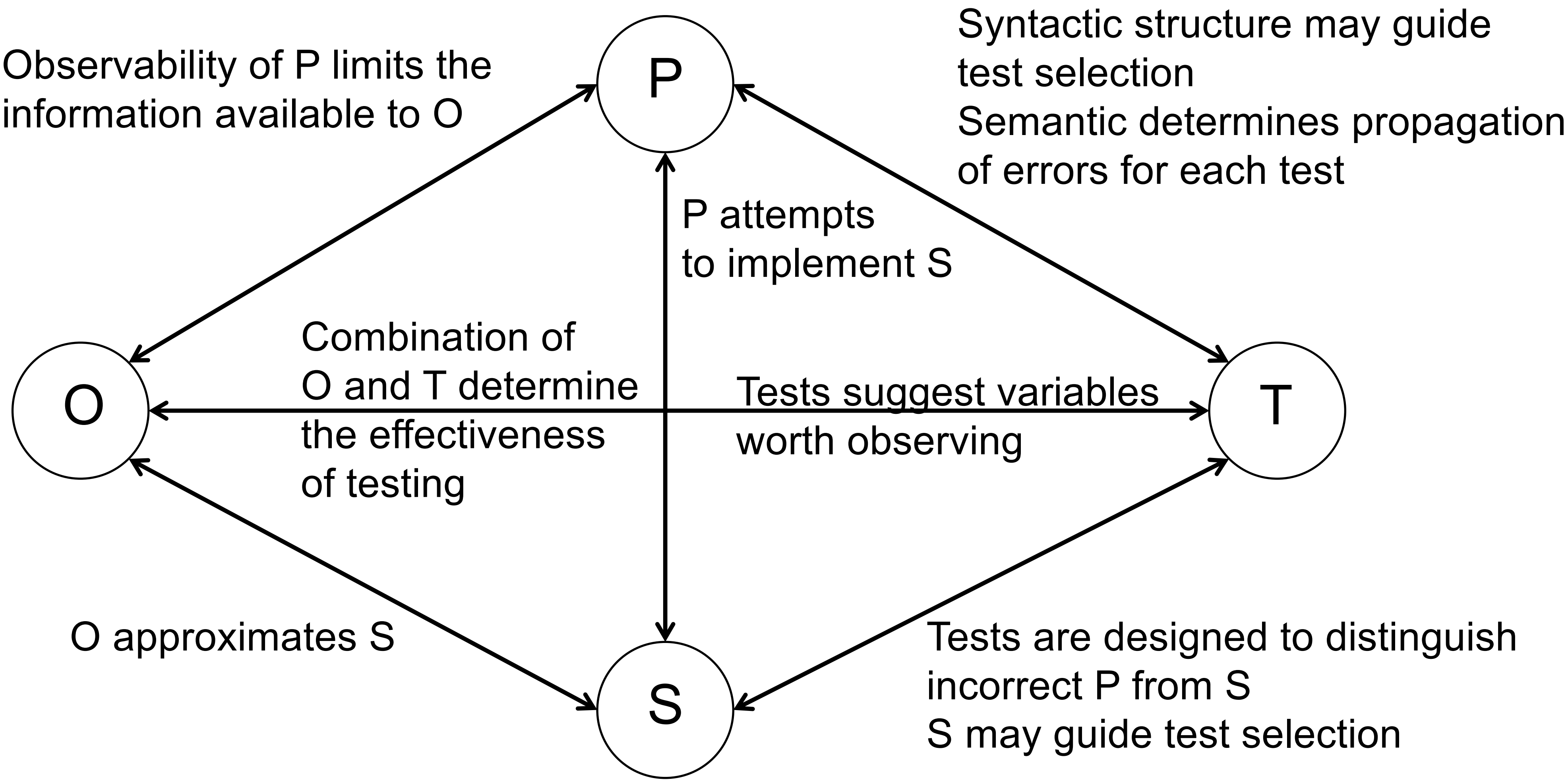
- The terminology used for testing is not always consistent

- The paper titled “Programs, Tests, and Oracles: The Foundations of Testing Revisited” tries to clarify some of the concepts about testing
  - It particularly focuses on the formalization of oracles

Formalizing Testing

- Basic concepts in testing:
  - P, Programs: This is the code, the implementation that we wish to test
  - T, Tests: T is a set of tests. Each test t ∈ T defines all the inputs to the program, so that given a test t, we can run the program p using t
  - S, Specifications: These are the specifications that characterize the correct behavior of the program; they may not be written down
  - O, Oracle: Oracle is used to determine if a test case passes or fails

Formalizing Testing



Observability of P limits the information available to O

Syntactic structure may guide test selection
Semantic determines propagation of errors for each test

P attempts to implement S

Combination of O and T determine the effectiveness of testing

Tests suggest variables worth observing

O approximates S

Tests are designed to distinguish incorrect P from S
S may guide test selection

Formalizing Testing

- A testing system consists of (P, S, T, O, corr, corr_t)
  - S is a set of specifications
  - P is a set of programs
  - T is a set of tests
  - O is a set of oracles
  - $\text{corr} \subseteq P \times S$
  - $\text{corr}_t \subseteq T \times P \times S$

corr(p, s) is returns true if the program p is correct with respect to s

corr_t(t, p, s) is true if and only if the specification s holds for program p when running test t

for all $p \in P$, for all $s \in S$, $\text{corr}(p,s) \Rightarrow$ for all $t \in T$ $\text{corr}_t(t, p, s)$

- These functions are not known and are just theoretical concepts used for defining properties of oracles

Formalizing Oracles

- An oracle $o \in O$ identifies which tests pass and which tests fail

$o(t, p)$ means that the test t passes for program p based on oracle o

- An oracle is complete with respect to p and s for t if:

  $corr_t(t, p, s) \Rightarrow o(t, p)$

- An oracle is sound with respect to p and s for t if:

  $o(t, p) \Rightarrow corr_t(t, p, s)$

- An oracle is perfect with respect to p and s if :

  for all t, $o(t, p)$ if and only if $corr_t(t, p, s)$

- Most oracles used in testing techniques are complete. However, in practice oracles are rarely sound.

Oracle Comparisons

- Given a test set TS, oracle o_1 has greater power than oracle o_2 (denoted as $o_1 \geq_{TS} o_2$) for program p and specification s if:

  for all $t \in TS$, $o_1(t, p) \Rightarrow o_2(t, p)$

- Assuming that the oracles are both complete, a more powerful oracle can catch more errors

- In some cases an oracle o_1 can be more powerful than another oracle o_2 for all possible test sets. In such cases, o_1 has power universally greater than o_2 (denoted as $o_1 \geq o_2$)

Test Adequacy

- Based on this formal framework, test and oracle adequacy can be defined as predicates:

- Test adequacy criterion: $T_C \subseteq P \times S \times 2^T$

- Oracle adequacy criterion: $O_C \subseteq P \times S \times O$

- Complete adequacy criterion: $TO_C \subseteq P \times S \times 2^T \times O$

- Complete adequacy criterion underlines the fact that the adequacy of testing must take into account both the tests and the oracles
  - Effectiveness of testing depends on both the tests and the oracles

Comparison of testing criteria

- A testing criterion C_1 is at least as powerful as a testing criterion C_2 with respect to a complete oracle o if:

For all p in P, s in S, T_1 in C_1, T_2, in C_2

If there exists a t_2 in T_2 where $\neg o(p, t_2)$ then

there exists a t_1 in T_1 where $\neg o(p, t_1)$

i.e., if test sets satisfying C_2 are guaranteed to find a fault for p when using oracle o, then so are all test sets satisfying C_1

Test Adequacy for Mutation Testing

- If we consider the method used to distinguish the mutants M from the program p as an oracle, we can formulate the mutation testing approaches using complete adequacy criterion

- For the set of mutants M, mutation adequacy Mut_M is satisfied for program p, specification s, test set TS, and oracle o if:

$Mut_M(p, s, TS, o) \Rightarrow$ for all $m \in M$, there exists a $t \in TS$: $\neg o(t, m)$

In other words, for each mutant $m \in M$, there exists a test t such that the oracle o signals a fault.