CSIT 415: Software Engineering

Instructor: Dr. Hubert Johnson

Lecture: Software Testing cont.

Verification, Validation, Testing

- ***Verification***: Demonstration of consistency, completeness, and correctness of the sw artifacts at each stage of and between each stage of the software life-cycle.
  - Different types of verification: manual inspection, testing,
  - 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.

Software Testing

- Goal of testing
  - finding faults in the software
  - demonstrating specs met (for the test cases that has been used during testing)

- Not possible to *prove* 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 helpful

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

Testing Software is Difficult/Challenging

- 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/impractical
  - 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** *t* is an element of input domain $t \in D$
  - 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 Challenging/Impractical

```
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
- The power of the test set is not determined by the number of test cases
- But, if $T_1 \supseteq T_2$, then T_2 will detect every fault detected by T_1

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, naive exhaustive testing is pretty hopeless

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 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
- So, naive random testing is pretty hopeless as well

Types of Testing

- Black box vs. White box testing
  - Black box: Generating test cases based on functionality of the software
    - testing the internal structure of the program is hidden from the testing process
  - White box: Generating test cases based on the structure of the program
    - testing internal structure of the program is taken into account
- Module vs. Integration testing
  - Module testing: Testing the modules of a program in isolation
  - Integration testing: Testing an integrated set of modules

Black-Box Testing

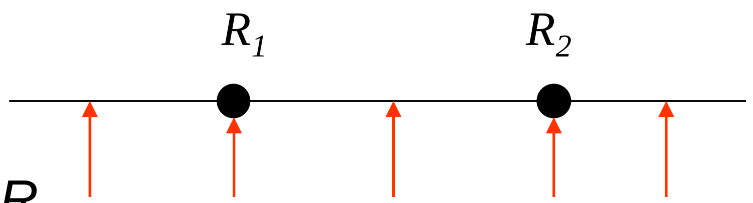
- Black box testing:
  - identify the functions which the software is expected to perform
  - create test data which will check whether these functions are performed by the software
  - no consideration is given how the program performs these functions, program is treated as a black-box: **black-box 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
  - derive test cases automatically from a formal specification of the functional requirements

Domain Testing

- Partition the input domain into equivalence classes (Remember this?}
- For some requirements specifications it is possible to define equivalence classes in the input domain
- 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

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



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 the factorial example, ranges for variable *n* are:
  - $[-\infty, 0]$, $[0,20]$, $[20,200]$, $[200, \infty]$
  - 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.

White-Box (Structural) Testing

- White box Testing
  - the test data is derived from the structure of the software

  - **white-box testing**: take internal structure of the software 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***: execute all statements at least once
  - ***Branch coverage***: execute all branches in the program at least once
  - ***Path coverage***: all execution paths should be executed at lest once
- The best case: 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)}

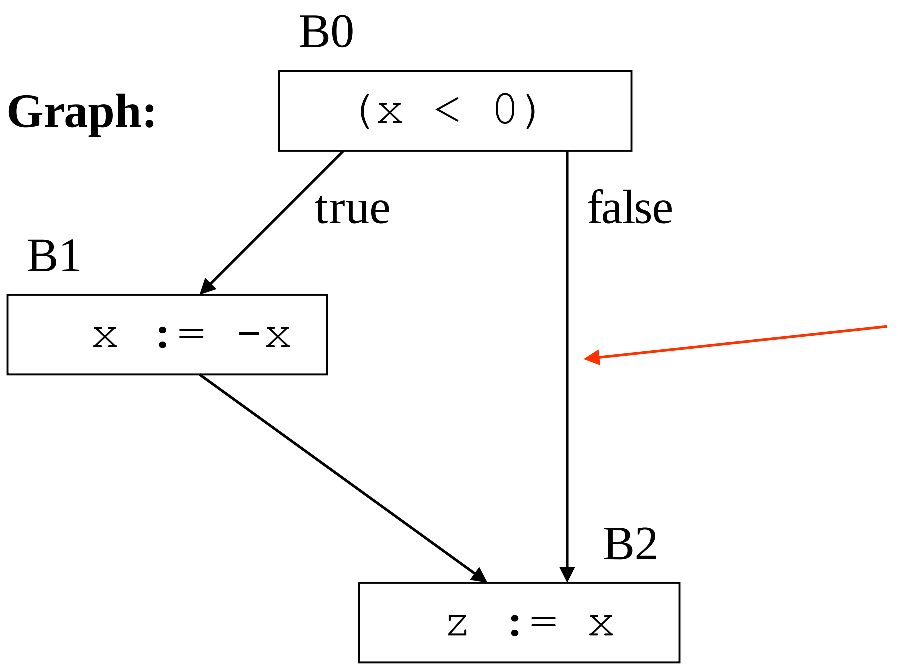
There is a difference between these two test sets though. Can you tell the difference?

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, but not branch coverage

Control Flow Graph:

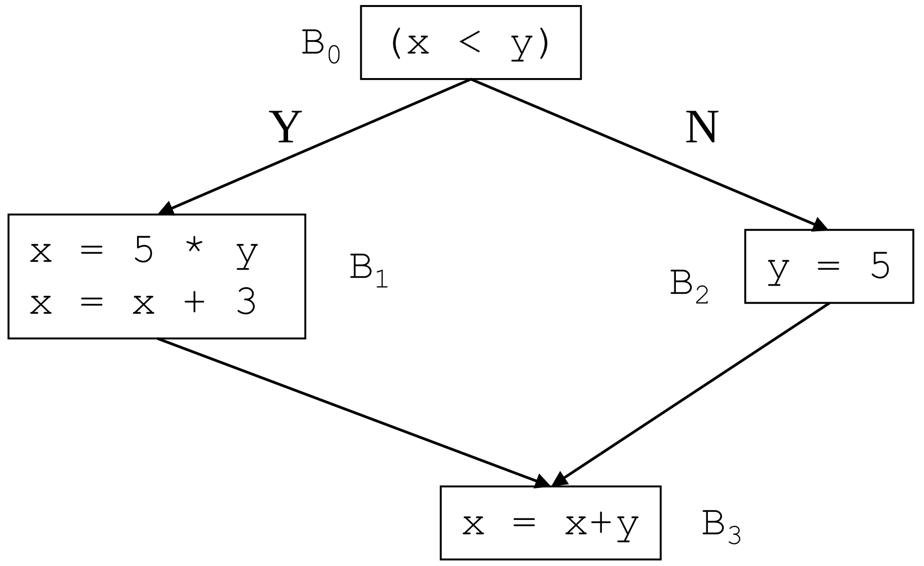


Test set {x=–1} does not execute this edge, hence, it does not give branch coverage

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

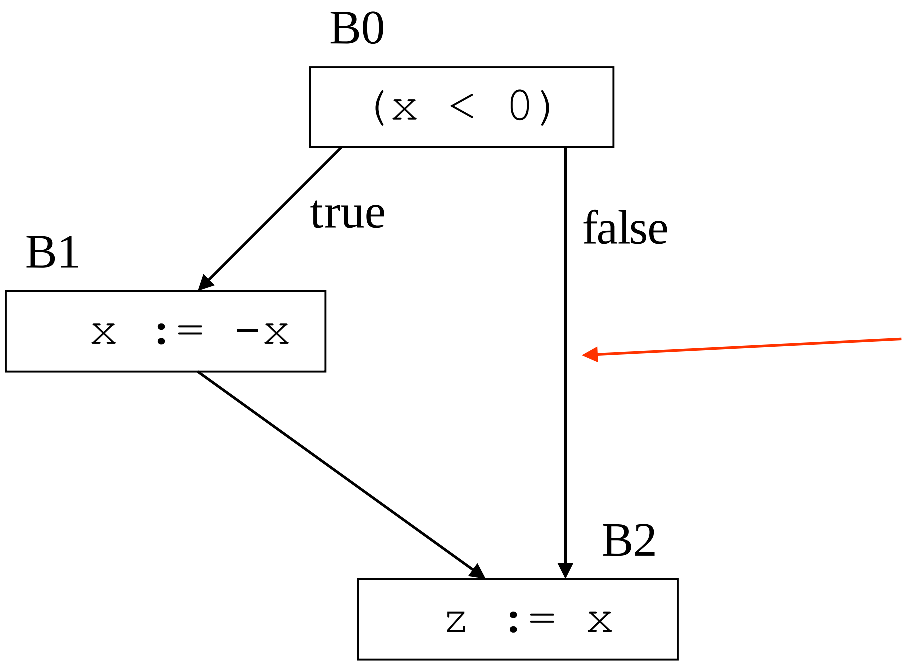
```
if (x < y) {
   x = 5 * y;
   x = x + 3;
}
else
   y = 5;
x = x+y;
```



- 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

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

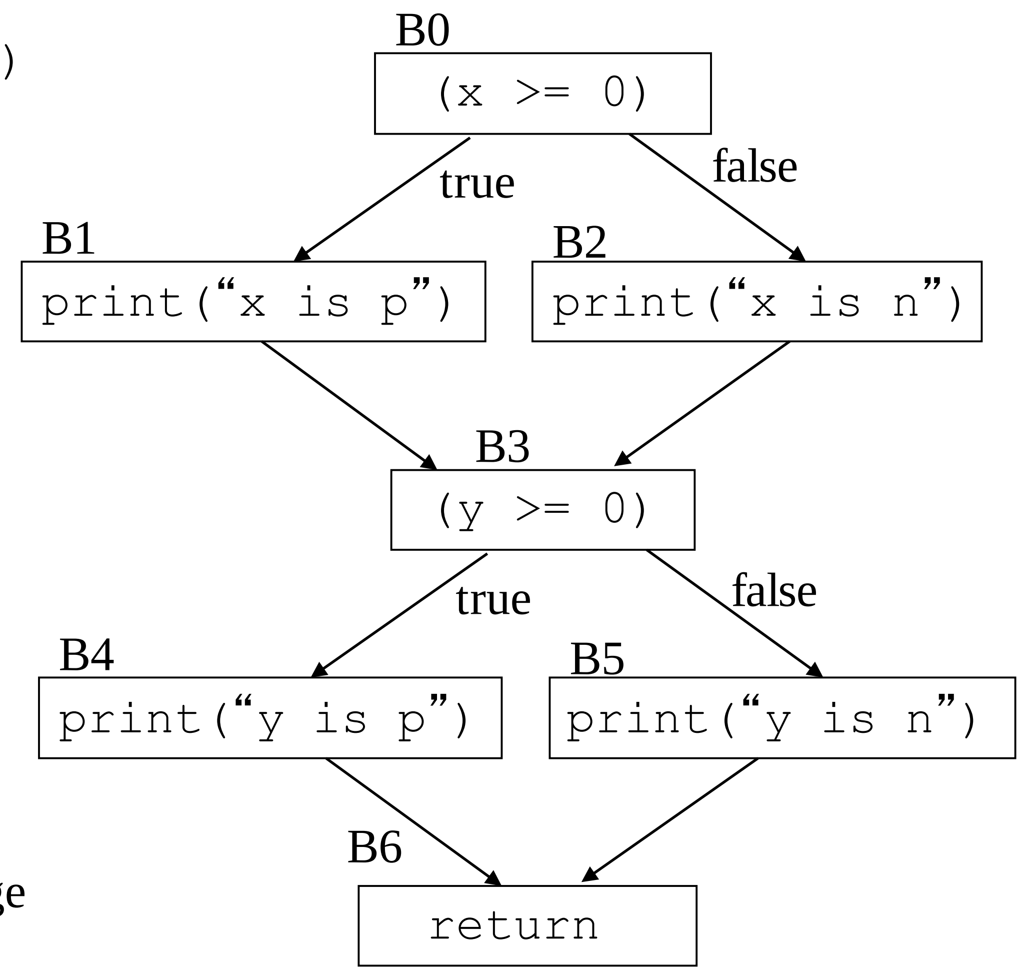
Test set {x= –1, x=2}gives both statement and branch coverage

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”);
}
```



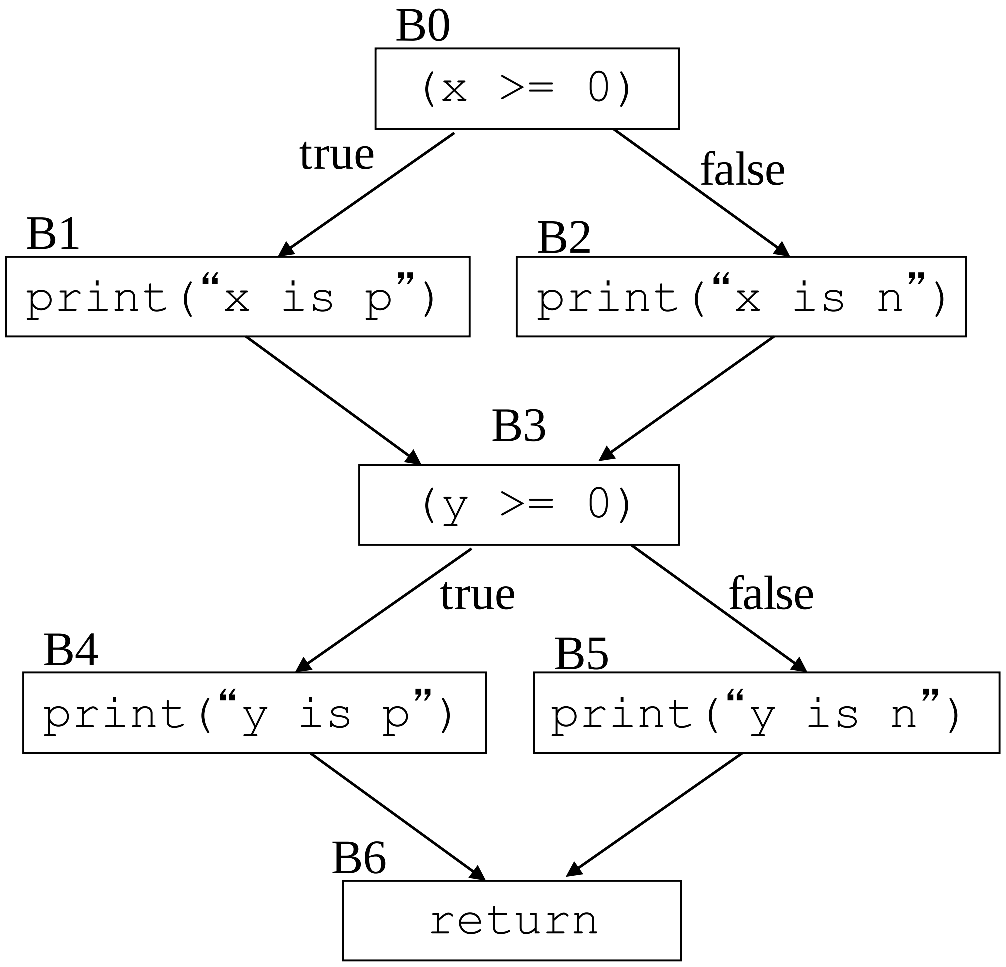
Test set:
T_2 = {(x=12,y= – 5), (x= –1,y=35)}
gives both branch and statement
coverage but it does not give path coverage

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

- The number of paths may be exponential based on the number of conditional branches
  - testing cost may be expensive

- Note: Every path in the control flow graphs may not be executable
  - It’s 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 is meant 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)

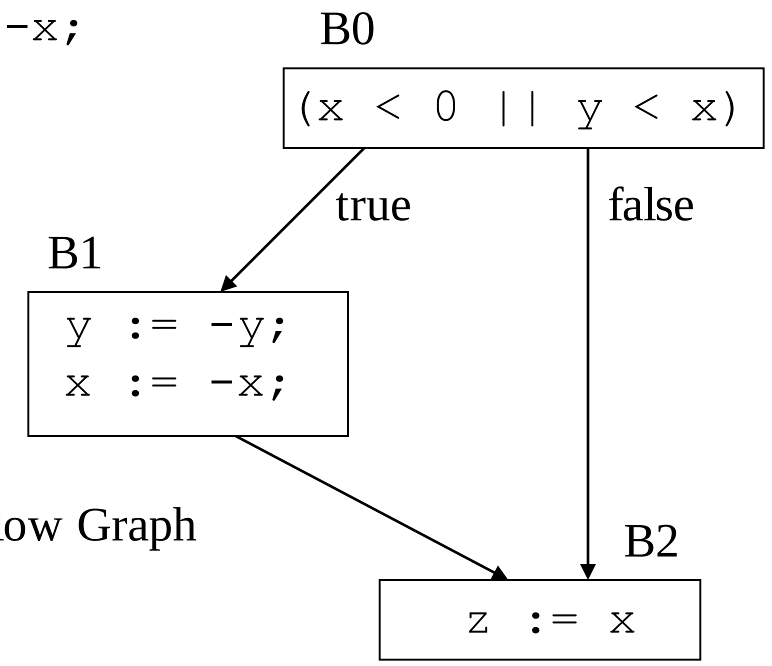
Condition Coverage

- In branch coverage we ensure that we execute every branch at least once
  - For conditional branches, this means, we execute the TRUE branch at least once and the FALSE branch at least once
- Conditions for conditional branches can be compound Boolean expressions
  - An expression consisting 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;
}
```

Give a test set T =
 will achieve statement, branch and path coverage, however T will not achieve condition coverage.



Control Flow Graph

Give a test set
T =
which will not achieve condition coverage and does not achieve branch 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 :

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

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

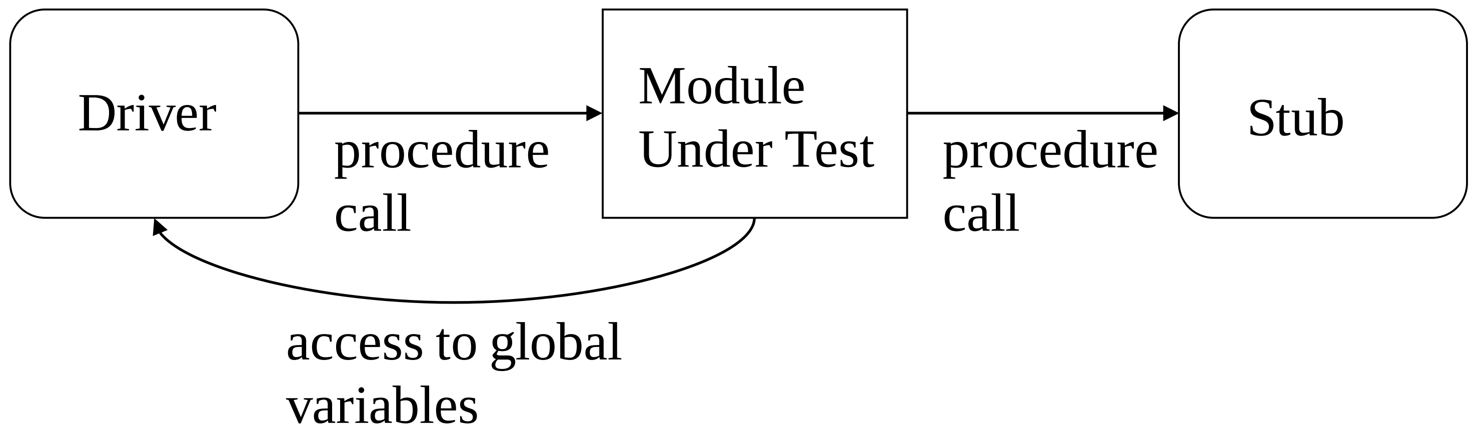
- Modules in a program are not isolated, they interact with each other. Possible interactions:
  - calling procedures in other modules
  - receiving procedure calls from other modules
  - sharing variables

- 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
  - Mock objects: Create an object that mimics only the behavior needed for testing
-

Drivers and Stubs

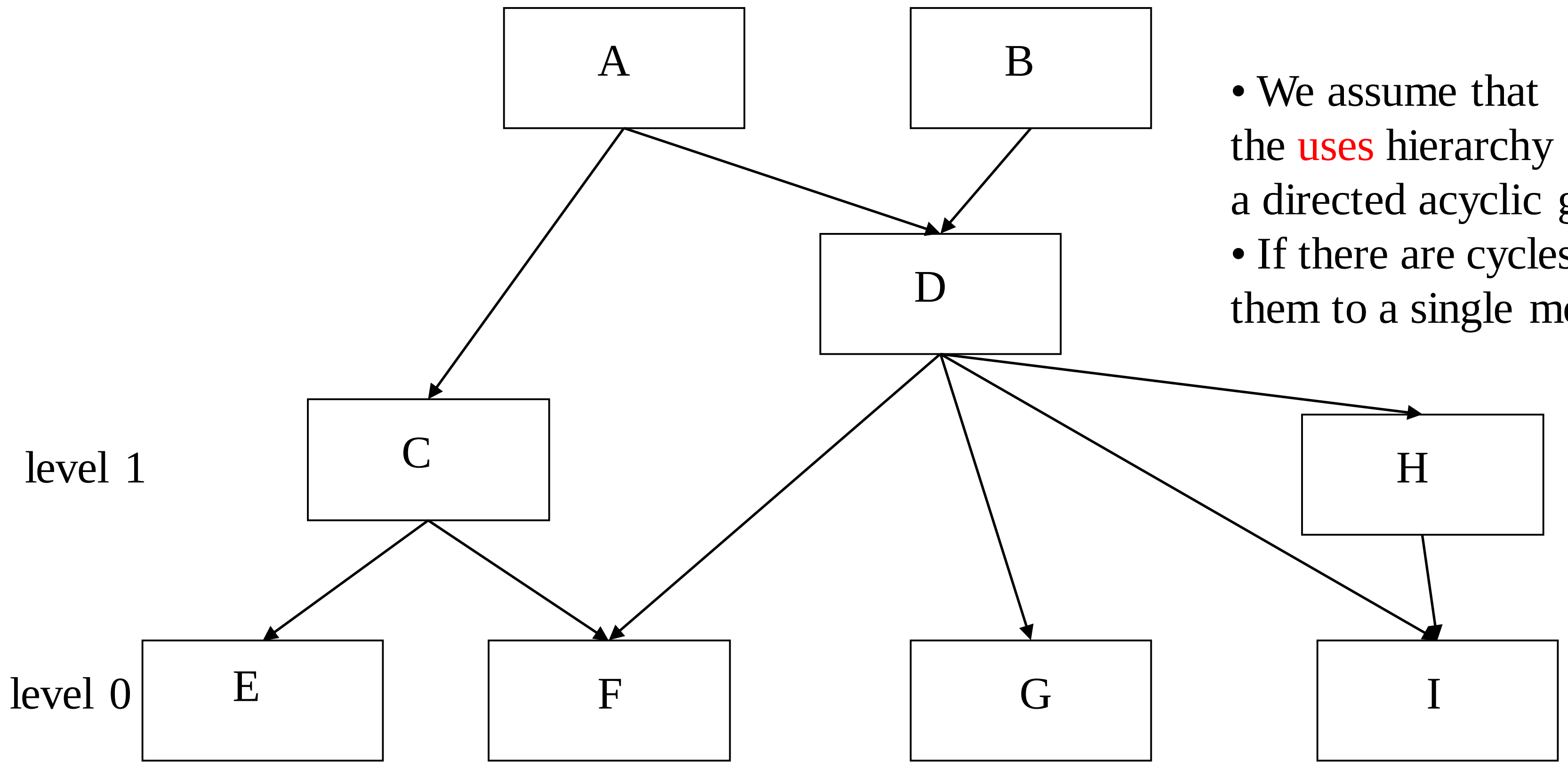


- Driver and Stub should have the same interface as the modules they replace
- Driver and Stub should be simpler than the modules they replace

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



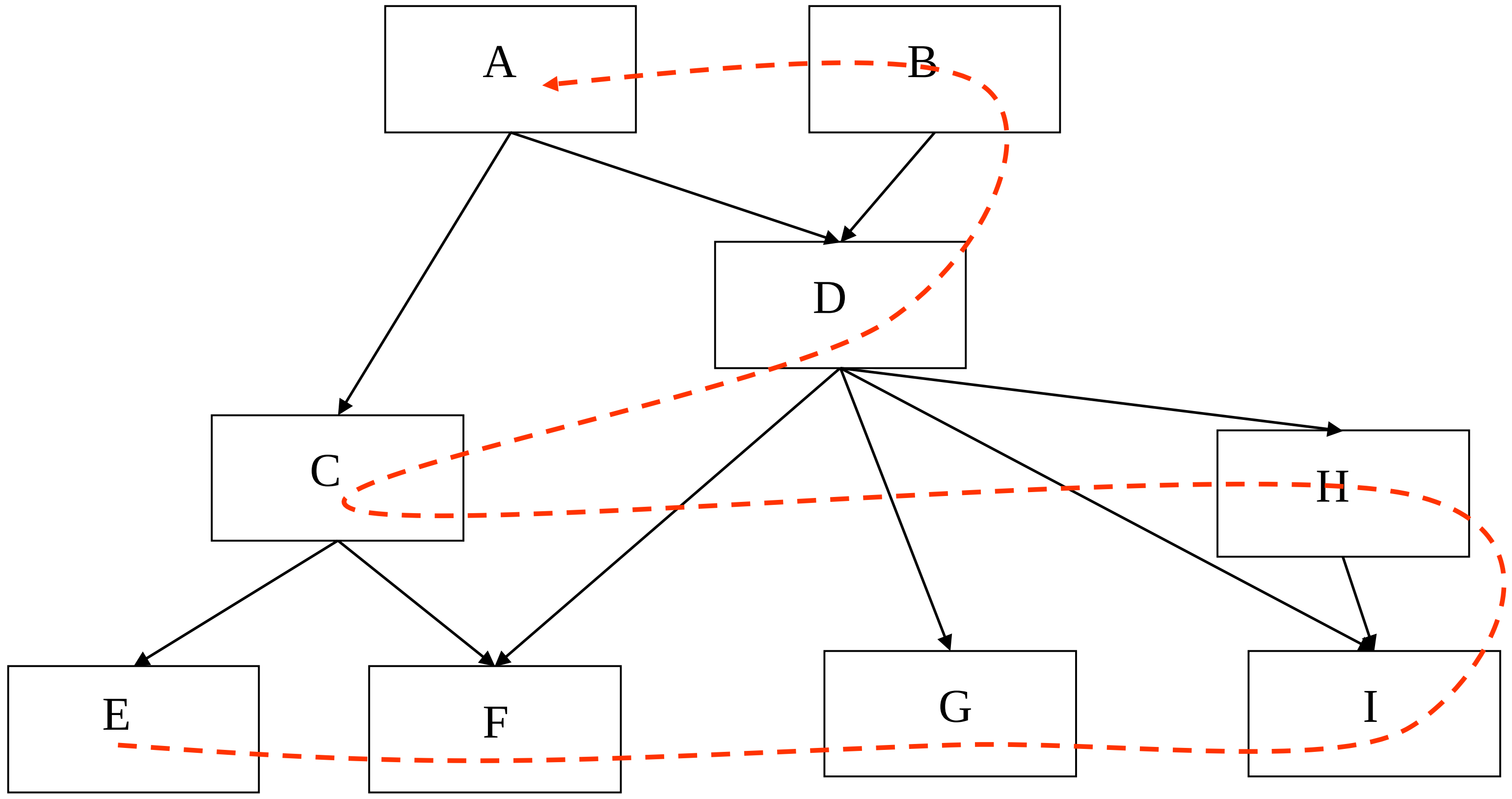
- We assume that the <span>uses</span> hierarchy is a directed acyclic graph.
- If there are cycles merge them to a single module

- 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



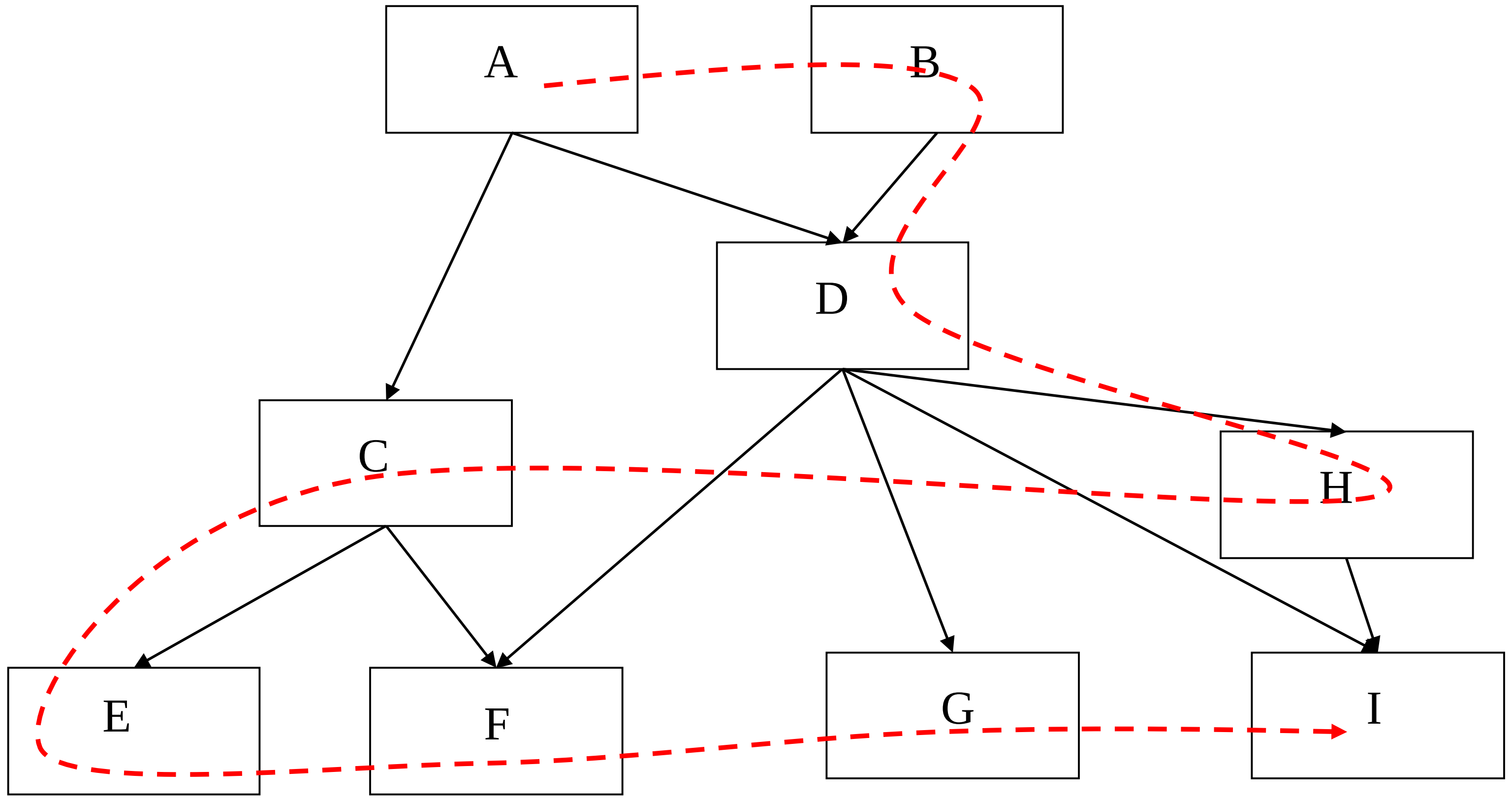
Top-down Integration

- 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 highly recommended to have a test plan

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

Test Driven Development

- A style of programming that has become popular with agile software development approaches such as extreme programming

- Basic idea: Write the test cases before writing the code
  - Test first, code second

- Divide the implementation to small chunks
  - First write the test that tests the next functionality
  - Check if the test fails (it should, since the functionality is not implemented yet)
  - Then, write the code to implement the functionality
  - Run all the tests and make sure that the code passes all the tests

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

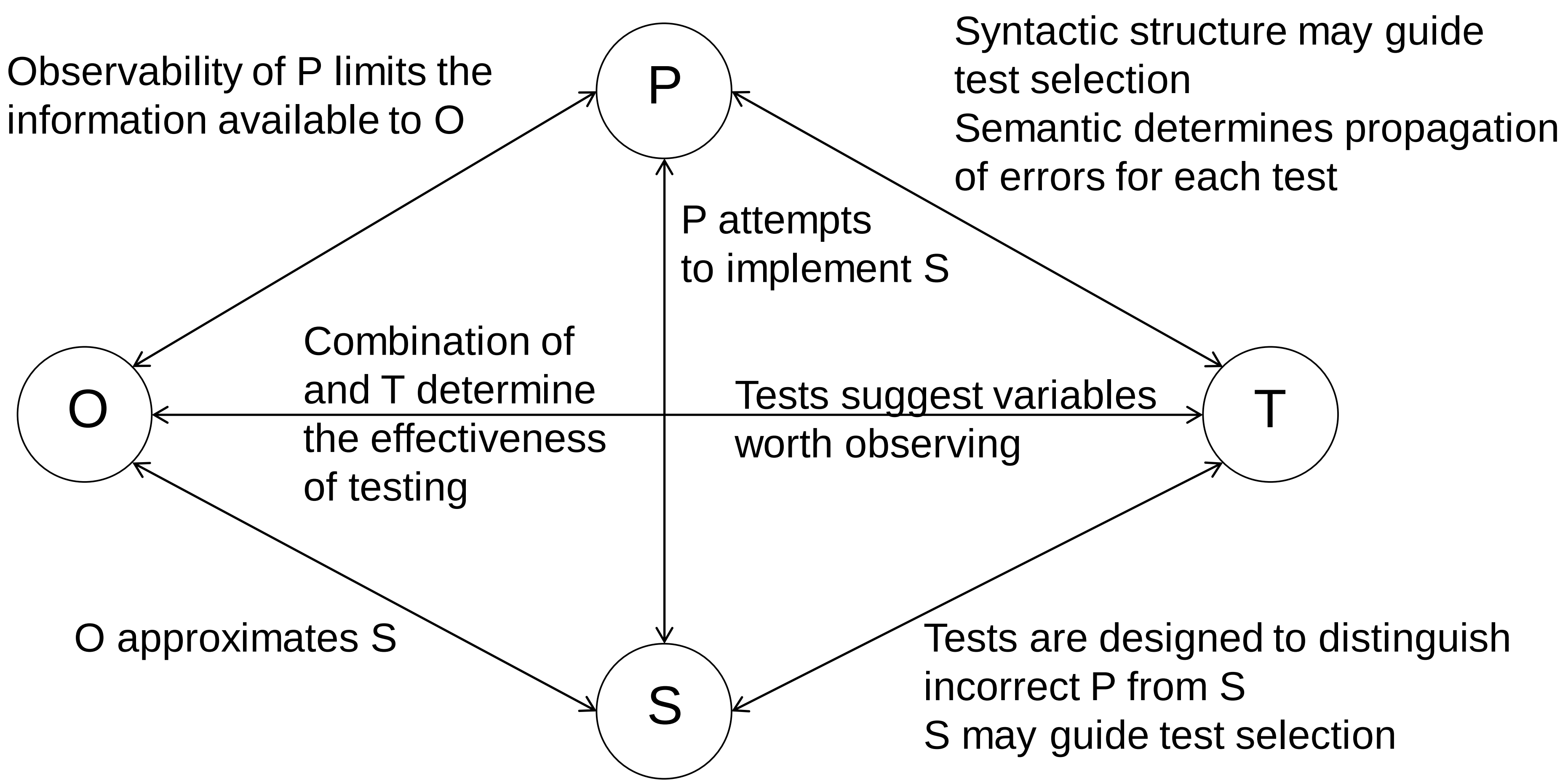
Formalizing Testing

- 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 $t \in T$ 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
P
Syntactic structure may guide test selection
Semantic determines propagation of errors for each test
P attempts to implement S
Combination of and T determine the effectiveness of testing
O
Tests suggest variables worth observing
T
O approximates S
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 specificaitons
  - P is a set of programs
  - T is a set of tests
  - O is a set of oracles
  - $corr \subseteq P \times S$
  - $corr_t \subseteq T \times P \times S$

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 h holds for program p when running test t

for all $p \in P$, for all $s \in S$, corr(p,s) $\Rightarrow$ for all $t \in T$ $corr_t$(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, $o_{1\ \text{has}}$ 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

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:

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

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