# Software Verification

CSE 403 - Spring 1997

# Outline

- What's software verification?
- Testing
- Analysis

# Verification vs. Validation

- Boehm
  - *Verification*: did we build the system right?
  - *Validation*: did we build the right system?
- We'll focus on basic techniques for verification.
  - Basic properties of analysis and testing.
- Validation is a much harder activity, requiring interactions of various sorts between the customers and the developers.

# Testing vs. Analysis

- Two basic approaches to verification:
  - *Analysis*: It consists of evaluating the system and its specifications to posit particular properties (such as termination, relationships between inputs and outputs, etc.)
  - *Testing*: It uses sampling to increase confidence in the correctness properties of a system.

# Testing

- The basic idea of testing is to exercise a software entity by:
  - specified inputs,
  - capturing specific outputs, and
  - comparing the outputs to those that the specifications require.
- The primary objective is to find differences between the actual and intended results, allowing correction of those differences.

# Testing Questions

- Criteria: What cases should be tested?
- Oracles: Are the outputs correct?
  - Verification: with the spec
  - Validation: with the customer's needs
- Adequacy: When can testing be stopped?
  - Have enough failures been found and fixed?
  - Is the entity "covered" enough?

# Testing: A Weakness

- The heart of testing is characterized by Dijkstra's statement:
  - "Program testing can be used to show the presence of bugs, but never to show their absence."
- Unless you test all possible inputs, testing can never allow you to say, "This program works as specified."
- So, the challenge is to figure out how to use limited resources in an effective way.

# State of the Practice

- Software testing is expensive.
  - Boehm's early studies showed that as much as 50% of development costs were attributable to testing.
  - Korel has reconfirmed these numbers recently.
- But operational software still fails. [Boehm]
  - Commercial software: 3-10 failures per KLOC.
  - Industrial software: 1-3 failures per KLOC.

# Analysis: Better?

- Can analysis ever let you say, "This program works as specified?"
  - In some cases, perhaps.
- Proofs are Social Processes
  - "Proofs," be they of mathematical theorems or of the correctness of computer programs, are essentially a social, communicative device.
  - They take time for people to absorb and agree upon.
  - May turn out to be incorrect on occasion.
- This does not invalidate the usefulness of analysis.
  - But shows that it too have weaknesses.
- When do you want to analyze?

# Systematic Testing

- If testing is not systematic, you can learn only that those specific test cases succeed or fail.
- Consider probabilistic algorithms, which are a strange form of testing.
  - To check for primality, which is too expensive to do completely for very large numbers, select a random set of numbers between 2 and $N^{0.5}$.
    - If each of those is in fact relatively prime to N, confidence is increased that N is prime.
  - But if the same number is "randomly" picked each time, the benefit is zero.

# Failures & Faults

- A *failure* occurs when a program acts differently from the stated specification.
  - E.g., an incorrect value may be produced, the program might bomb, etc.
  - The consequences of a failure can range from mild to catastrophic.
- A *fault* is an incorrect intermediate state of a program.
  - Failures can only occur due to faults, but faults do not necessarily cause failures.

# Defects & Errors

- A *defect* is the piece of code that causes a fault.
  - In some standards, this is called a fault not a defect.
- An *error* is the conceptual cause of a defect.
  - What thought process caused the programmer to insert a defect?

# Test Cases & Sets

- A test case is a set of input data with the associated expected output data.
  - This might be a one-to-one or one-to-many relationship.
- A test set is a finite set of test cases.
- Test cases and sets either fail or succeed; a test set fails if any test case in the set fails.
  - Some definitions reverse success and failure.
  - This is not a minor point: it completely changes the point of view of testing.
- How to come up with test sets that increase our confidence in software?

# Is More Always Better?

- "The more test cases the better" is a false statement.
- Consider [GJM]
  - if x > y then
    max := x;
    else
    max := x
    endif
- Two test sets:
  - {(x=3, y=2), (x=2, y=3)}
  - {(x=3, y=2), (x=4, y=3), (x=5, y=1)}

# Another Example

- if x > 0 then
  write(x)
  elseif x < 0 then
  write("("); write(x); write(")");
  else
  write("-0-")
  endif
- Test sets:
  - {(x=3), (x=-5)}
  - {(x=3), (x=4), (x=0)}

# Wait a Minute!

- In both of these cases, we assumed what the programs should do.
- Testing must be done with respect to a specification.
- But testing is often done against what a student/programmer thinks the program should do.
  - This often makes testing easier, but it often leaves wrong programs.
- In introductory courses, this is clear even in the use of programming language constructs.
  - In asking a student what a statement does, he/she often responds saying what he/she wants it to do, not what it does.

# Characteristic Tests

- The point is that it is important to pick "characteristic" values that achieve greater *coverage*.
  - That is, they are more significant test cases, in that they are likely to realize failures if they exist.
- One way to do this is to pick test cases that represent other possible test cases.
  - In the example, above, picking negative, positive, and zero values would be useful in partitioning the input.
  - The goal is to increase the "reasonable expectation" (without guarantee, of course) that the results of the test are representative of the full input domain.

# Expectation vs. Truth

- This point about reasonable expectation vs. truth is made clearly by this example [GJM].
  - If the input value is < 0, then an appropriate error message must be printed. If 0 <= n < 20, then the exact value of n! must be printed. If 20 <= n <=200, then an approximate value of n! must be printed in floating point format, e.g., using some approximate method of numerical calculus. 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.
- Why might the test set {-10, 5, 175} not be sufficient to build complete confidence in the program.
  - What would you do?

# Equivalence Classes

- In some cases, the classes of input values for which you want test sets do not overlap, thus defining equivalence classes.
  - That is, it doesn't matter (in theory) which element you test that is less than zero, between 0 and 19 inclusive, between 20 and 200 inclusive, and over 200.
- In other cases these sets might overlap; but this just allows (if desired) one test case to exercise several classes of input values.

# Another Example

- Another example might be to count boundary cases (those near the "edges") as one class and to count error cases as another.
  - ```
    p := 0.0;
    i := n;
    while (1 <= i) do
       p := (p * x) + a(i);
       i := i - 1;
    end
    ```
- Testing {-1, 0, 1, m-1, m, m+1} might be the boundary cases while the "out-of-range" cases might include {-100, -1, m+1, m+100}.
  - Note the overlap.
    ```

# Complete Coverage

- Building complete coverage is useful, even if it doesn't build complete confidence.
- But coverage is an ill-defined term....
- The extremes include:
  - Assume each class consists of a single element, which means that you have to do exhaustive testing to ensure that each class is covered. (**Infeasible**)
  - Assume that all input is in the same class, in which case you could test the system with complete coverage using a single test case. (**Useless**)

# What Then?

- The challenge, then, is to handle coverage effectively.
- There is no standard, well-accepted approach towards doing this, although there are lots of alternatives.
- One approach is to ensure that you define test sets that make sure that each statement in the program is exercised by at least one test case.
- The reason for this is obvious: if there is a statement that has never executed, then it might cause a failure if it can in fact ever be executed.
  - If it can never be executed, then it does not belong in the program....
  - Any exceptions?

# Statement Coverage

- This approach is called *statement coverage*.
- Consider the GCD example.
- Any test case executes the first three statements (including the loop test) and the last statement.
- Is the test set {(x=3, y=3), (x= 4, y=5)} good for coverage?
- How about {(x=4, y=5)}?

```
read(x);
read(y);
while x <> y do
    if x > y then
        x := x - y;
    else
        y := y - x;
    endif
end;
gcd := x;
```

# Statement Coverage

- How do you ensure that a test set exercises all statements?
- This is a difficult thing to determine manually.
  - A tool to help determine if a test set does this is often useful, if not necessary.
- Producing new test cases that cover missed statements may not be trivial.

# What is a Statement?

- What is a statement?
- Consider the following. (Possible in many languages.)
  - max := if x > y then x else y
- Is this one statement? It is to the compiler.
- Thus, you have to be careful either about:
  - What you call a statement, or
  - What you expect the statement coverage criterion to tell you about the program's correctness.

# Minimal Statement Coverage?

- Another question is:
  - Should the size of the test set be minimal?
  - This goes back to the case of overlapping input domains.
- For statement coverage, this addresses test cases that touch some of the same statements (which is always the case.)
- Even if {(x=4, y=5)} does touch all statements, is it enough?

# A Weakness of Statement Coverage

- Consider
  - if x < 0 then
    x := -x;
    endif;
    z := x;
- If you select a test case with x negative, then all the statements are exercised...
  - But it seems clear that testing with other classes of x is important.

# Clarifying the Weakness

- This is easier to understand if the program is thought of as:
  - if x < 0 then
    x := -x;
    else
    null;
    endif;
    z := x;
- Ensuring that the null statement is executed requires an additional test case in the test set.

# Edge Coverage

- Another approach is based on ensuring that all control branches of a program are exercised.
- This is another way, for instance, of ensuring that both the true and the false cases of the conditional in the previous example are executed by some test case (without creating the null statement)
- Understanding this *edge coverage* criterion requires an understanding of the *control flow graph* of a program.

# Control Flow Graphs

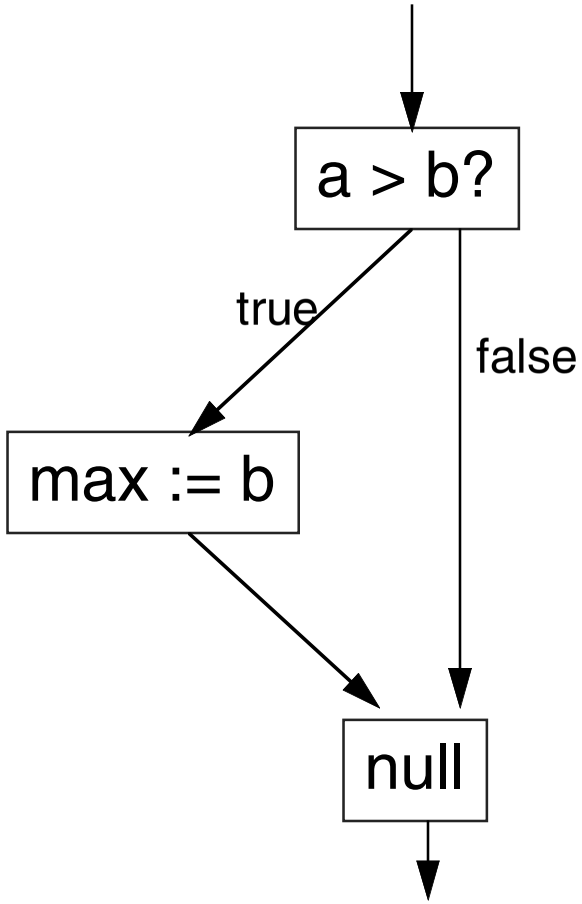
- The control flow graph (CFG) is a representation of a program that has nodes representing statements (and conditional tests) and arcs representing possible control transitions among these nodes.
  - The CFG is broadly useful in domains including compilation, optimization and others.
- The edge coverage criterion ensures that all arcs in the CFG for a program are exercised at least once.

# CFG: Sequential Statements and Conditionals

- Two sequentially executed statements are represented simply as two nodes with an arc from the first to the second (in the order they are to be executed.)
- An if statement is represented in the CFG as shown on the right.
- The conditional is required to have outgoing arcs labeled with all possibilities (true and false here)
  - When there is a single out-edge, it is generally not labeled and indicates that the arc is always taken after the statement is executed (that is, sequential composition.)

# CFG: Sequential Statements and Conditionals

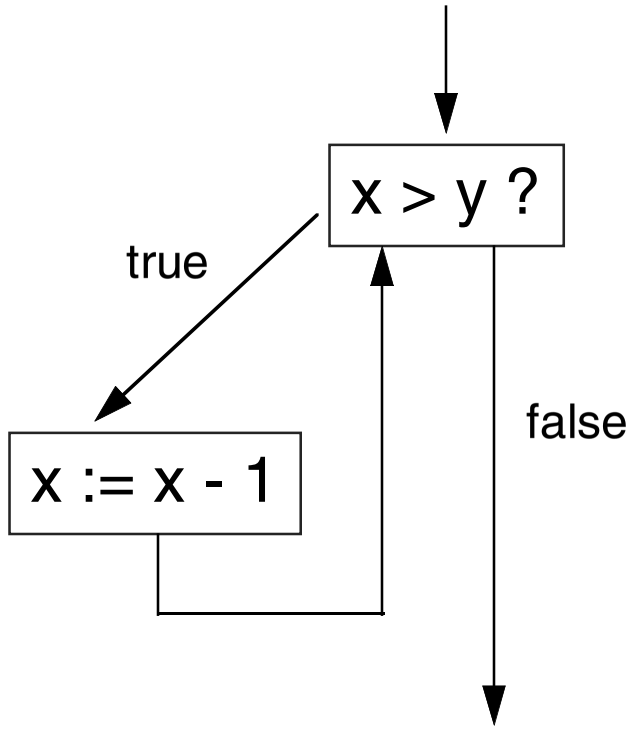
- if a > b then
  max := b
  endif



# CFG: Loops

- A loop is represented as a combination of sequential execution and conditional execution.

```
while x > y do
    x := x - 1
end
```

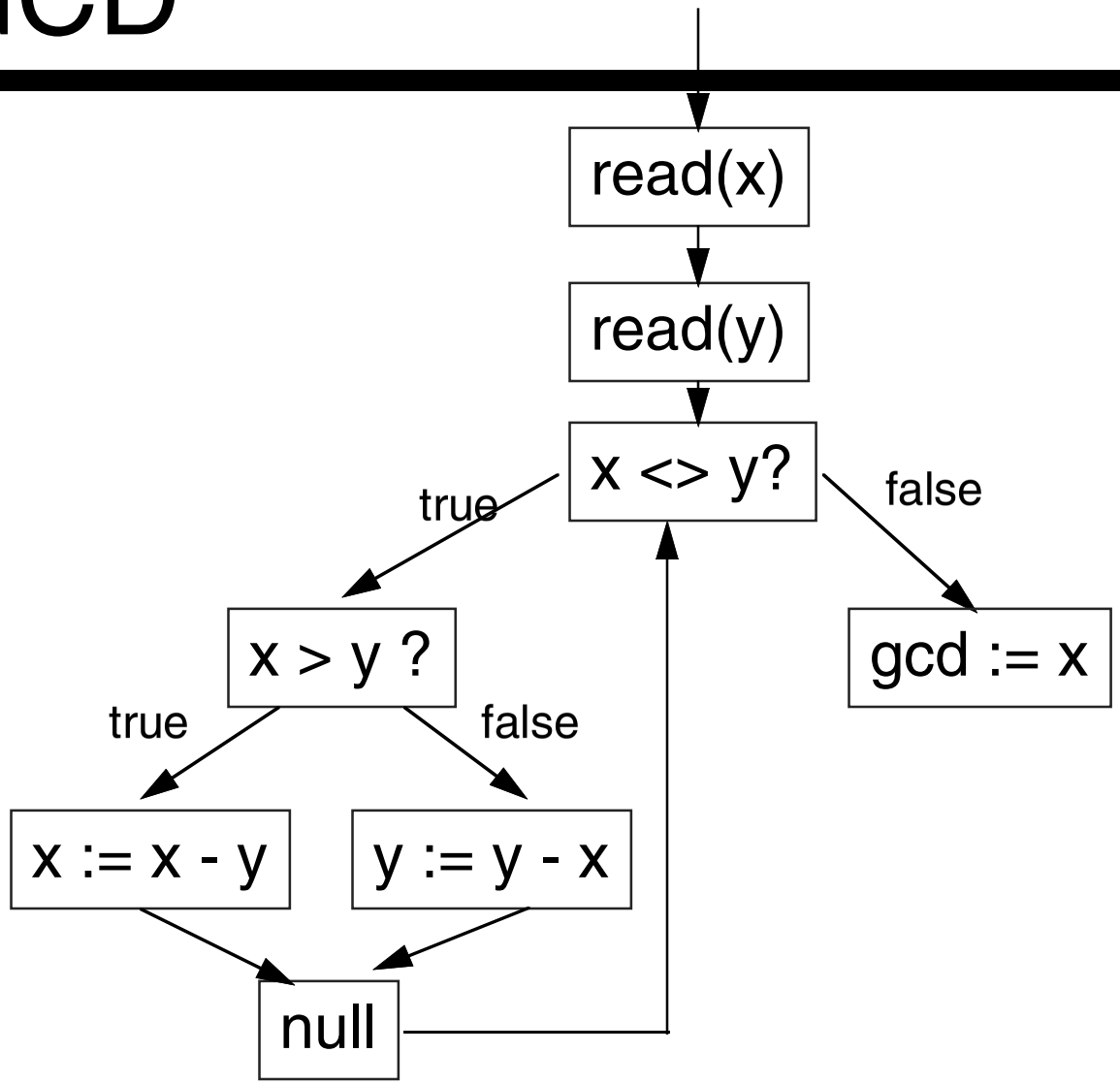


# CFG: Other Constructs

- How about?
  - switch
  - conditional expression
  - for loop?

# CFG for GCD

- The definition of statement may confuse statement coverage for this example.
- But edge coverage is clear.



# Compound Conditionals

- Checking every edge may not give you everything you want.
- This is especially the case with compound conditionals.
- The problem is that compound conditionals can be interpreted in several ways, and the interpretation affects the coverage.
- If a compound **and**, for instance is considered as a single condition, there is no guarantee by edge coverage that all combinations of the condition are exercised.

# Short-Circuit Conditionals

- Edge coverage is further confused by the issue of whether or not the conditions are short-circuit conditions or not.
  - If they are not, then both "sides" will be executed, but the resulting alternatives may not all be exercised.
  - If they are short-circuit, then even the second "side" might never be executed.
- An alternative is to consider these compound conditions as nested conditionals.

# Examples

- if a > b and c > d then
  s1
  endif
- if x = y or y = z then
  s2
  else
  s3
  endif
- if p andif q then
  s4
  elseif s orif t then
  s5
  endif

# Condition Coverage

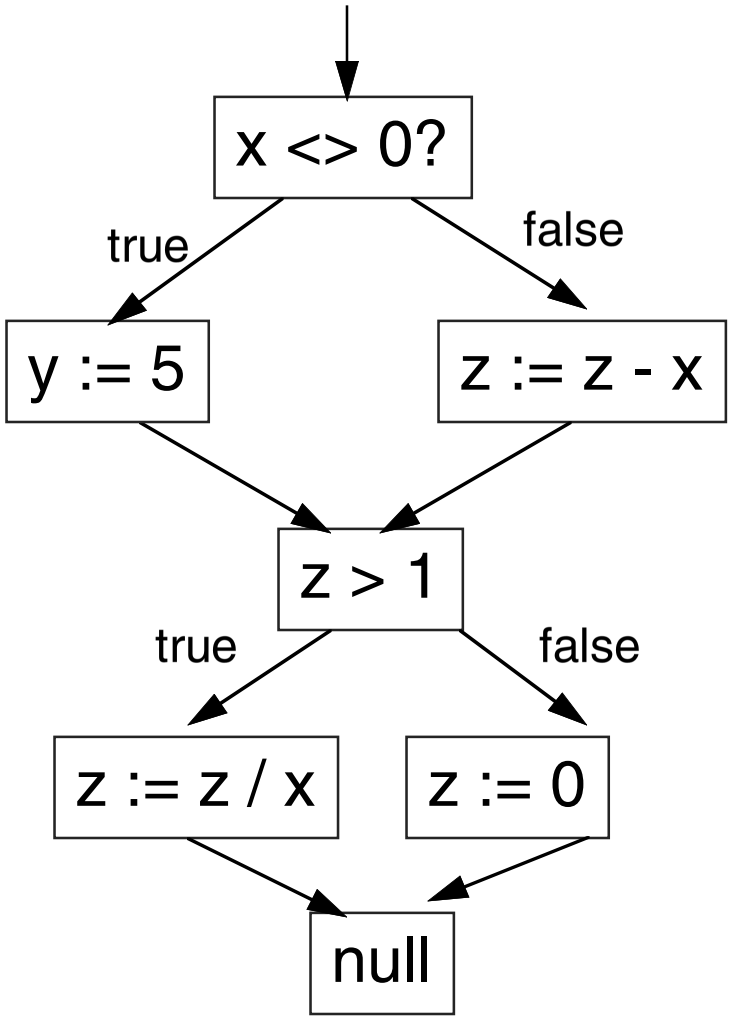
- A test criterion that guarantees that all (combinations of) conditions are tested is called *condition coverage*.
  - All control branches must be tested as well.
- The way to do this is to expand each of the compound and short-circuit conditions.

# Condition Coverage Weakness

- What kinds of defects are missed by condition coverage?
- Consider the example on the right.
  - The test set {(x=0, z=1), (x=1, z=3)} covers all control edges and all conditions.
  - But it still misses the potential for zero-divide (which is probably a defect.)
- Why does it miss this?
  - Because there are paths---sequences of control edges---that are not exercised.

```
if x <> 0 then
    y := 5;
else
    z := z - x;
endif;
if z > 1 then
    z := z / x;
else
    z := 0;
endif;
```

# CFG for the Previous Example



# Path Coverage Criterion

- The previous example leads to the *path coverage criterion*.
  - It states that you need a test set that ensures that all paths along the control flow graph are exercised.
  - In the previous example, path coverage would ensure that the error causing a zero-divide was caught.
  - So, what would be a test set we come up with?
- The bad news is that ensuring that you exercise all paths is, in general, difficult or impossible.
  - Loops, of course, introduce the central problem.

# Mimicking Path Coverage on Loops

- There are some criteria focused on loops that try to mimic the path coverage criterion.
  - Make sure every loop is skipped (executed zero, or at least the minimum number of times.)
  - Make sure that it is executed the “maximum” number of times.
  - Make sure it is executed an “average” number of times.

# Maximum & Average

- Maximum and average times a loop is executed can often be determined using analytical methods.
  - E.g. Quicksort.
- But sometimes they cannot be determined.
  - ```
    while x <> 1 do
        if odd(x) then
            x := 2 * x + 1;
        else
            x := x / 2;
        endif
    end loop
    ```
    ```

# Discussion

- These are all examples of white-box testing, where the specification and the program are available for testing.
- There are many more approaches and open questions.
  - How to handle combinatorial blowups (say, in the number of paths in a 1,000,000 line program)?
  - How to apply these notions to object-oriented programming?
  - ... many more...

# Test Data Adequacy Criteria

- Weyuker has defined test data adequacy criteria, which are intended "to determine whether or not testing can be terminated."
  - These criteria are considered with respect to specific approaches, such as statement or edge coverage.
- There are eight properties defined with which to assess program-based (white-box) testing approaches.
  - The goal of these properties is to understand the fundamental characteristics of various testing approaches.
  - Specific approaches may fail to meet some of the properties; they may still be useful, pinpointing their deficiencies.

# Properties 1 and 2

- **Applicability Property**:
  - For every program, there exists an adequate test set.
- **Non-Exhaustive Applicability Property**:
  - There is a program P and test set T such that P is adequately tested by T, and T is not an exhaustive test set.
- If the first property required test sets unbounded in size, it wouldn't be useful.

# Properties 3 and 4

- **Monotonicity Property**:
  - If T is adequate for P, and T is a subset of T', the T' is adequate for P.
    - Adding test cases to an adequate test set doesn't help.
    - Applicability and monotonicity together imply that exhaustive test sets are always adequate.
- **Inadequate Empty Set Property**:
  - The empty set is not an adequate test case for any program.

# Property 5

- **Antiextensionality Property**:
  - There are program P and Q such that P and Q are semantically-equivalent (always produce the same outputs for the same inputs). T is adequate for P, but T is not adequate for Q.
    - This captures the notion of black-box testing.
- What if programs are syntactically highly similar?
  - In the authors' term, the program have the same *shape* if you can produce one from the other by replacing relational operators, replacing constants, and replacing arithmetic operators.

# Property 6

- **General Multiple Change Property**:
  - There are programs P and Q which are the same shape, and a test set T that is adequate for P, but T is not adequate for Q.
    - Even syntactically close programs may have to be tested with different test sets.

# Property 7

- **Antidecomposition Property**:
  - There exists a program P and component Q such that T is adequate for P, T′ is the set of vectors of values that variables can assume on entry to Q for some t in T, and T′ is not adequate for Q.
- Even if a program has been adequately tested, it does not imply that is components have been adequately tested.
  - Perhaps the component is used in a restricted way in the program.
  - This means, for instance, that you would have to test them again for use in another context.

# Property 8

- **Anticomposition Property**:
  - There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q.
- This says that you can't test program pieces separately and then consider them adequately tested when you compose the pieces.

# Discussion

- Many well-known approaches don't satisfy one or more of these properties.
- For example, statement and edge coverage approaches do not even satisfy the first property (applicability).
  - This is unsatisfactory, since one doesn't know after trying it if one simply needs a different test set or if no adequate test set exists.
- They also fail to satisfy both the anticomposition and antidecomposition properties.

# Discussion

- Weyuker defines three additional properties that are defined to eliminate some intuitively unsuitable testing criterion from being considered successful by meeting the first eight.
  - Finding criteria that in fact satisfy even the first eight seems perhaps more critical.
- In any case, this work lays a careful theoretical foundation for evaluating different testing criteria.

# Functional vs. Structural Testing

- Functional testing
  - Test cases selected based on functional specifications.
  - Cannot determine if a program works without considering what it is intended to do.
  - Views the component as a black box.
- Structural testing
  - Test cases selected based on structure and implementation.
  - Cannot determine if a program works without considering what it actually does.
  - Views the component as a white box.
- Both are needed.