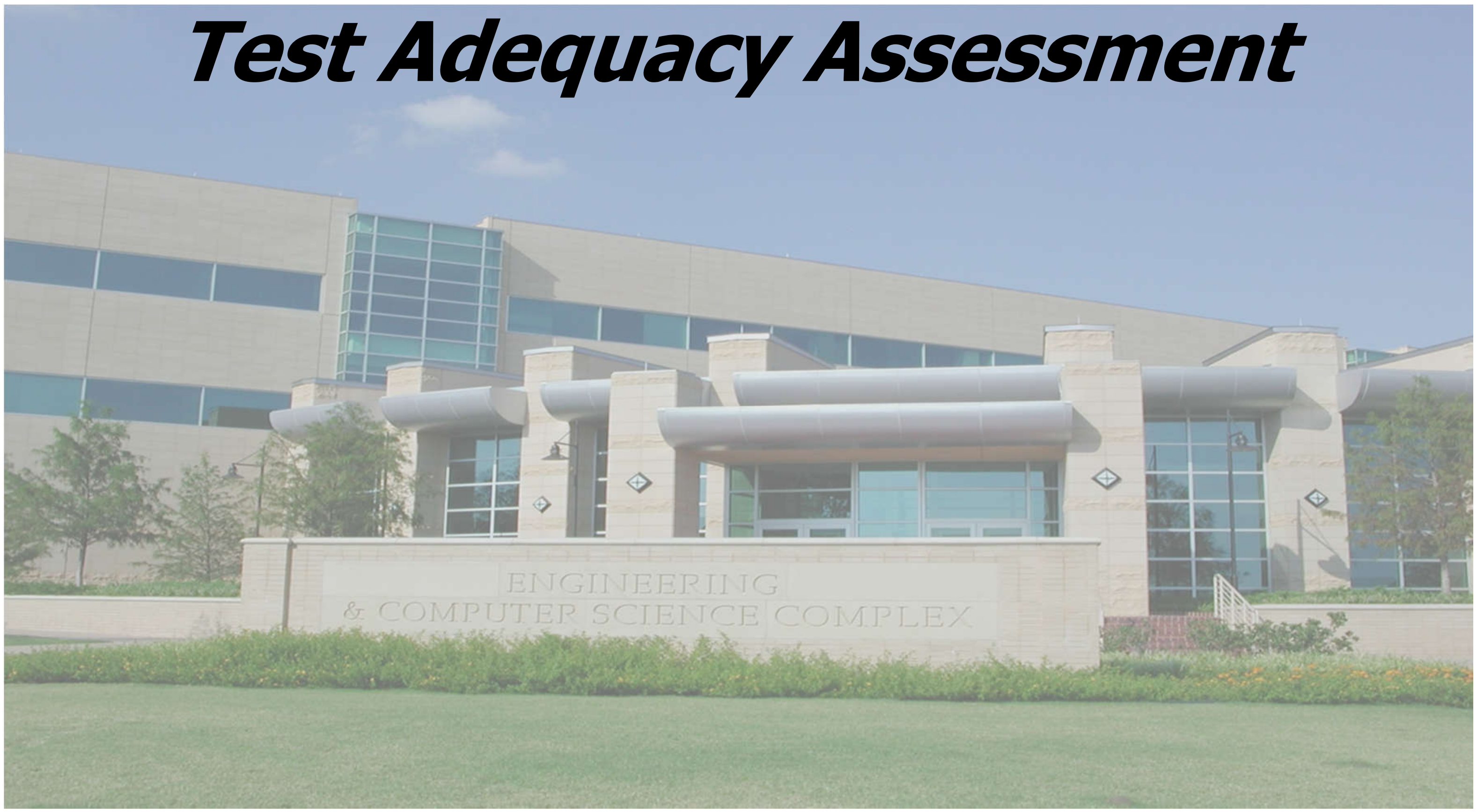
Test Adequacy Assessment



Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

---

Jonsson School of Engineering and Computer Science

Test Adequacy Assessment Topics

<u>Part III. Test Adequacy Assessment and Enhancement</u>

7. Test Adequacy Assessment Using Control Flow and Data Flow

- **Basic**
- **Adequacy criteria based on control flow**
- **Concepts from data flow**
- **Adequacy criteria based on data flow**
- **Control flow versus data flow**
- **The “subsumes” relation**
- **Structural and functional testing**
- **Scalability of coverage measurement**
- **Tools**

8. Test Adequacy Assessment Using Program Mutation

What Is Adequacy?

Consider a program P written to meet a set R of functional requirements.

- **notate P and R as (P,R)**
- **R contains n requirements labeled R_1, R_2, …, R_n**

Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R.

- **P has been executed against each test in T and has produced correct behavior**
- **foundations of test completeness defined by Goodenough and Gerhart**

Has P been tested thoroughly?
Is T good enough? Is T adequate?

- ***What does "adequate" mean?***

Measuring Adequacy

Adequacy is measured for a given test set T designed to test program P to determine whether P meets its requirements against a given criterion C.

T is considered <u>adequate</u> with respect to criterion C when it satisfies C.

- **determination of whether test set T for program P satisfies criterion C depends on the criterion itself**

Mathur, Example 7.1

Program sumProduct must meet the following requirements

R_1: Input two integers x and y

$R_{2.1}$: Find and print the sum of x and y if $x < y$

$R_{2.2}$: Find and print the product of x and y if $x \geq y$

Suppose the test adequacy criterion C is specified as:

C : A test T for program (P, R) is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r.

T = {t: <x=2, y=3>} is inadequate with respect to C for program sumProduct.

- **T tests R_1 and $R_{2.1}$, but not $R_{2.2}$**

Black Box and White Box Criteria

For each adequacy criterion C, we derive a finite set known as the coverage domain and denoted as C_e.

A criterion C is a <u>black-box</u> test adequacy criterion if the corresponding coverage domain C_e depends solely on requirements R for the program P under test.

A criterion C is a <u>white-box</u> test adequacy criterion if the corresponding coverage domain C_e depends solely on program P under test.

Measuring Coverage

Given that C_e has $n \geq 0$ elements, we say that T <u>covers</u> C_e if for each element e' in C_e there is at least one test case in T that tests e'.

Test set T is considered <u>adequate</u> with respect to adequacy criterion C if it covers all elements in the coverage domain.

T is considered <u>inadequate</u> with respect to C if it covers k elements of C_e where k<n.

The fraction k/n is known as the <u>coverage</u> of T with respect to C, P, and R.

Mathur, Example 7.2

Consider P, T, and C of Example 7.1

C_e = {R1, R2.1, R2.2}

T covers R1 and R2.1 but not R2.2

- **T is not adequate with respect to C**

Coverage of T with respect to C is 2/3 = 0.67

Mathur, Example 7.3

Consider the path coverage criterion:

- **A test set T for program (P, R) is considered adequate if each path in P is traversed at least once.**

Assume that P has exactly two paths

- **p_1 corresponding to $x < y$**
- **p_2 corresponding to $x \geq y$**

For the given adequacy criterion C we obtain the coverage domain C_e to be the set $\{p_1, p_2\}$.

To measure the adequacy of T for sumProduct against C, we execute P against each test case in T.

T contains only one test for which x < y

- **only path p_1 is executed**
- **coverage of T with respect to C is 0.5**
- **T is not adequate with respect to C**

We can also say that p_1 is tested and p_2 is not tested.

Code-Based Coverage Domain

In Example 7.3, we assumed that P contains exactly two paths.

This assumption is based on a knowledge of the requirements.

When the coverage domain must contain elements from the code, these elements must be derived by analyzing the code and not only by an examination of its requirements.

- **white-box testing**

Mathur, Example 7.4

Program P7.1 is obviously incorrect wrt the requirements of sumProduct.

There is only one path

- **denote as p_1**
- **p_1 traverses all the statements**

Using the path-based coverage criterion C, we get coverage domain $C_e = \{p_1\}$

T = {t: <x=2, y=3> } is adequate wrt C but does not reveal the error.

<u>Program P7.1</u>

```
begin
   int x, y;
   input (x, y);
   sum = x + y;
   output (sum);
end
```

Program P7.2 is correct as per the requirements of sumProduct.

It has two paths denoted by p_1 and p_2.

$C_e = \{p_1, p_2\}$

T = {t: <x=2, y=3>}
is <u>inadequate</u> wrt the path-based coverage criterion C.

<u>Program P7.2</u>

```
begin
   int (x, y);
   input (x, y);
   if (x < y)
   then
      output (x + y);
   else
      output (x * y);
end
```

Take Away

An adequate test set might not reveal even the most obvious error in a program.

This does not diminish the need for measuring test adequacy.

- **increasing coverage might reveal an error**
- **it does indicate that test adequacy is not a complete answer to the question of whether our testing is good enough…**

Test Enhancement

While a test set adequate with respect to some criterion does not guarantee an error-free program, an inadequate test set is a cause for worry.

- **Inadequacy with respect to any criterion often implies deficiency.**

Enhancement is also likely to test the program in ways it has not been tested before.

- **testing untested portions**
- **testing features in a sequence different from the ones used previously**
- **raises the possibility of discovering new defects**

Mathur, Example 7.5

For Program P7.2, to make T adequate with respect to the path coverage criterion we need to add a test that covers p_2.

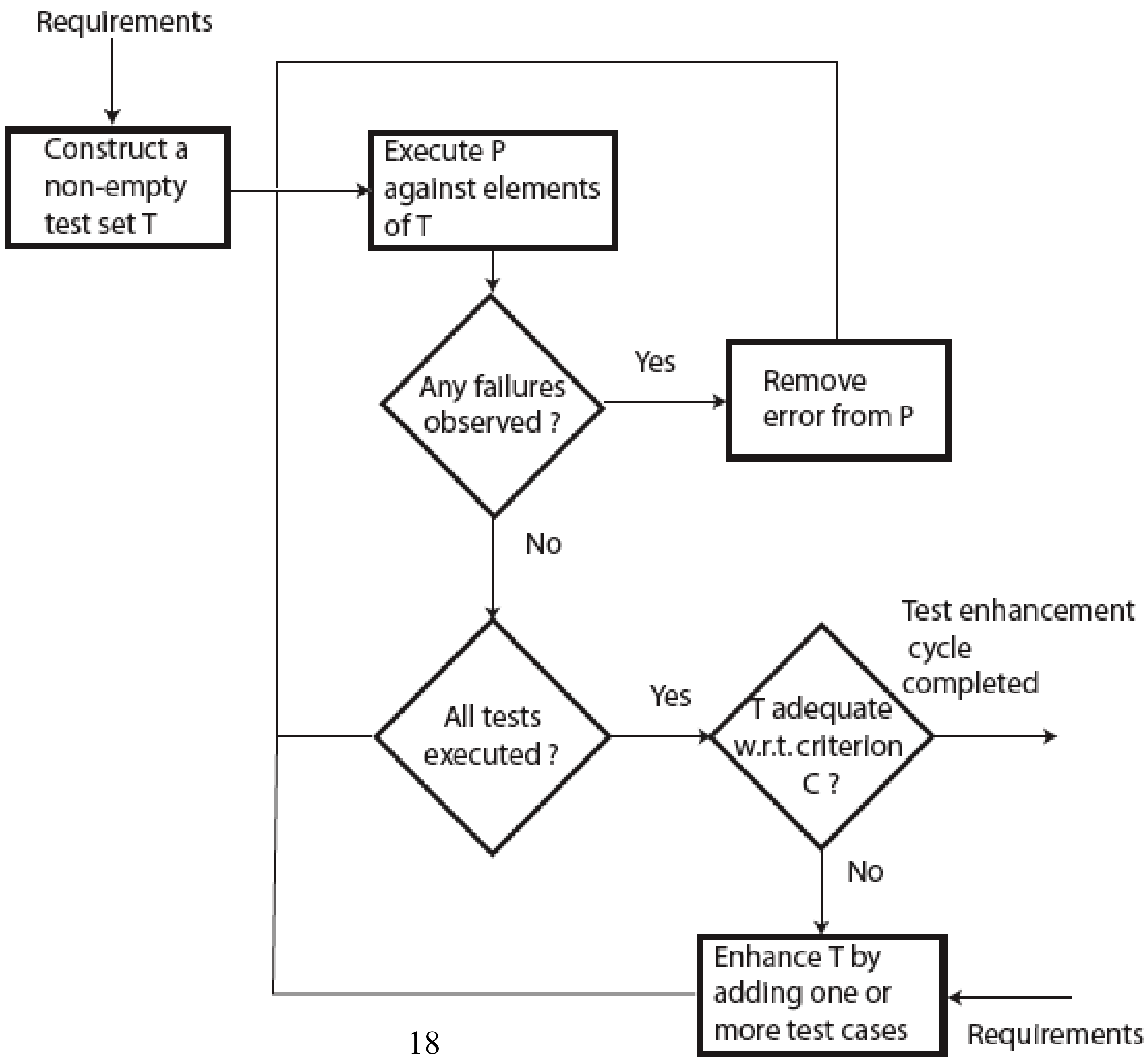
- **one test that does so is {<x=3>, y=1>}**

T' = {t_1: <x=3, y=4>, t_2: <x=3, y=1>}

Executing Program P7.2 against the two tests in T' traverses paths p_1 and p_2.

- **T' is <u>adequate</u> with respect to the path coverage criterion**

Test Enhancement Procedure



Mathur, Example 7.6

Program P7.3

```
begin
   int x, y;
   int product, count;
   input (x, y);
   if (y ≥ 0) {
      product = 1; count = y;
      while (count > 0) {
         product = product * x;
         count = count – 1;
      }
      output (product);
   }
   else
      output ("Input does not match its specification.");
end
```

Consider Program P7.3 intended to compute x^y given integers x and y.

For y < 0, the program skips the computation and outputs a suitable error message.

Suppose that test T is considered adequate if it tests Program P7.3 for at least one zero and one non-zero value of each of the two inputs x and y.

The coverage domain for C can be determined using C alone and without any inspection of the program.

$C_e = \{x=0, y=0, x \neq 0, y \neq 0\}$

$T = \{t_1: <x=0, y=1>, t_2: <x=1, y=0>\}$

Mathur, Example 7.7

Criterion C of Example 7.6 is a black-box coverage criterion

- **it does not require an examination of the program under test for measuring test adequacy**

Consider the path coverage criterion.

Examination of Program P7.3 reveals that it has an indeterminate number of paths due to the while loop.

- **the number of paths depends on the value of y and hence that of count**

Given that y is any non-negative integer, the number of paths can be arbitrarily large.

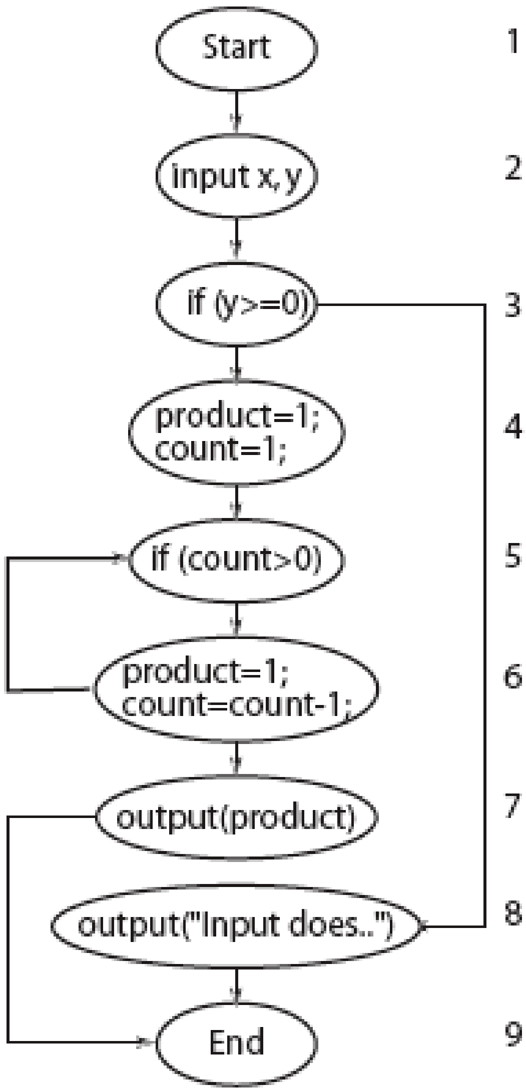
- **for the path coverage criterion, we cannot determine the coverage domain**

The usual approach in such cases is to simplify C and reformulate it as follows:

C': A test T is considered adequate if it tests all paths.

Modified Path Coverage Criterion

In case the program contains a loop, then it is adequate to traverse the loop body zero times and once.



The modified path coverage criterion leads to

$C'_e = \{p_1, p_2, p_3\}$

p_1: [1→2→3→4→5→7→9]

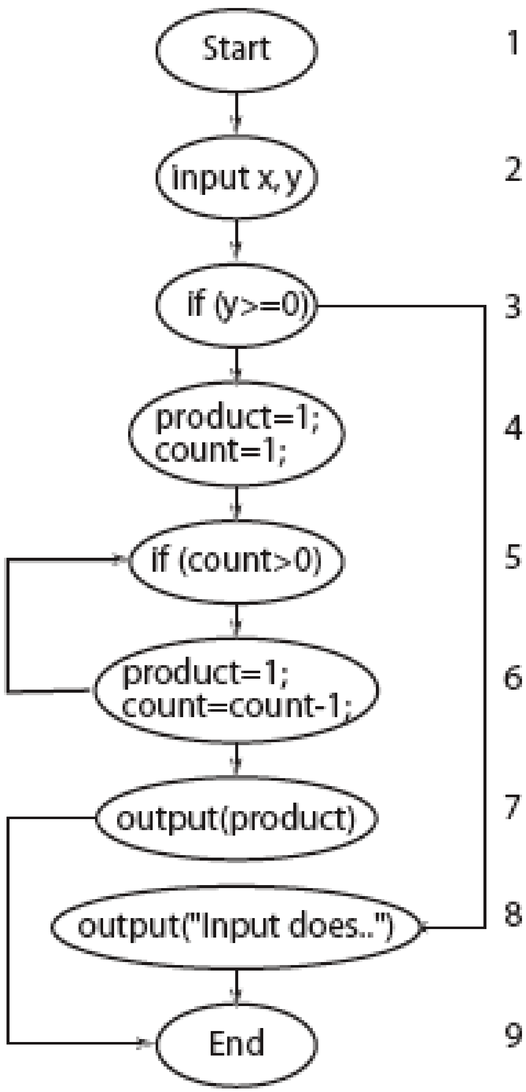
- corresponds to y ≥ 0 and loop traversed 0 times

p_2: [1→2→3→4→5→6→5→7→9]

- corresponds to y ≥ 0 and loop traversed 1 time

p_3: [1→2→3→8→9]

- corresponds to y < 0 and control reaches output without entering loop



We measure the adequacy of T with respect to C'.

p_1: [1→2→3→4→5→7→9]
p_2: [1→2→3→4→5→6→5→7→9]
p_3: [1→2→3→8→9]

T does not contain any test with y < 0

- p_3 remains uncovered

Coverage of T with respect to C' is 2/3 = 0.67

Any test case with $y < 0$ will cause p_3 to be traversed.

- **t: <x=5, y=-1>**

Test t covers path p_3 and P7.3 behaves correctly.

- **add test case t to test set T**
- **the loop in the enhancement terminates as we have covered all feasible elements of C'_e**

The enhanced test set T' is:

- **T' = {<x=0, y=1>, <x=1, y=0>, <x=5, y=-1>}**

Infeasibility

An element of the coverage domain is infeasible if it cannot be covered by any test in the input domain of the program under test.

No algorithm exists that would analyze a given program and determine if a given element in the coverage domain is infeasible or not.

It is usually the tester who determines whether or not an element of the coverage domain is infeasible.

Demonstrating Feasibility

Feasibility can be demonstrated by executing the program under test against a test case and showing that the element under consideration is covered.

Infeasibility cannot be demonstrated by program execution against a finite number of test cases.

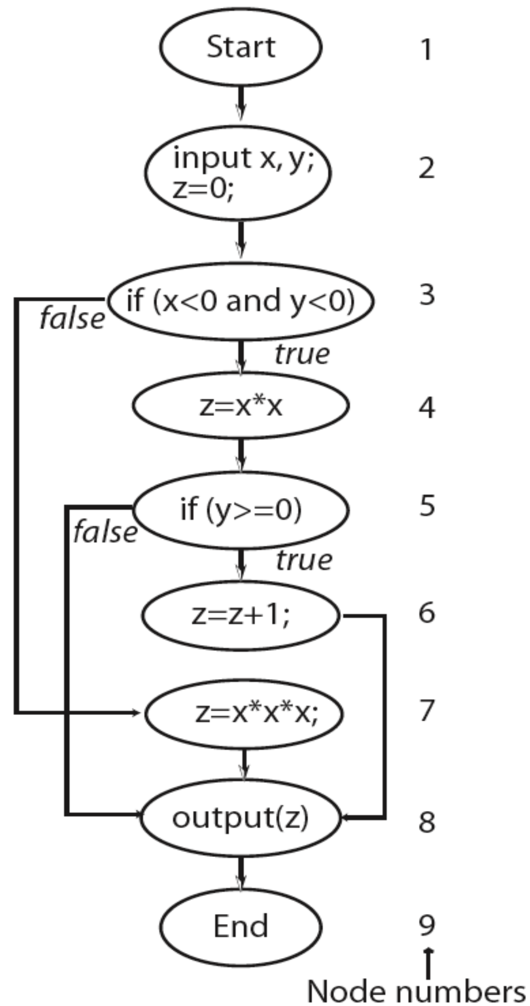
- **In some cases simple arguments can be constructed to show that a given element is infeasible.**
- **For more complex programs the problem of determining infeasibility could be difficult.**

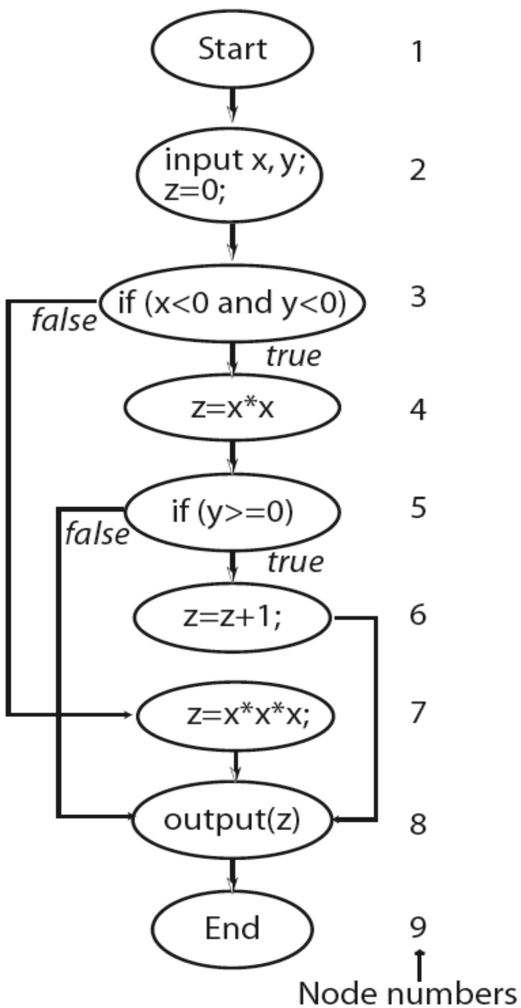
An attempt to enhance a test set by executing a test t aimed at covering element e of program P may fail.

Mathur, Example 7.8

<u>Program P7.4</u>

```
begin
    int x, y;
    int z;
    input (x, y); z = 0;
    if (x < 0 and y < 0) {
        z = x * x;
        if (y ≥ 0) z = z + 1;
    }
    else
        z = x * x * x;
    output (z);
end
```





Program P7.4 inputs two integers x and y and computes z.

$C_e = \{p_1, p_2, p_3\}$

p_1: [1→2→3→4→5→6→8→9]

- **corresponds to x < 0, y < 0, and y ≥ 0**

p_2: [1→2→3→4→5→8→9]

- **corresponds to x < 0, y < 0 (y ≥ 0 is false)**

p_3: [1→2→3→7→8→9]

- **corresponds to x < 0 and y < 0 being false**

p_1 is infeasible and cannot be traversed by any test case

- when control reaches node 5, condition $y \geq 0$ is false
- control can never reach node 6

Any test adequate with respect to the path coverage criterion will only cover p_2 and p_3.

Infeasible

A <u>constraint</u> C is considered infeasible for predicate p_r if there exists no input values for the variables in p_r that satisfy C.

An element of the coverage domain is infeasible if it cannot be covered by any test in the input domain of the program under test.

- **feasibility can be demonstrated by executing the program against a test case and showing that it is covered**
- **infeasibility cannot be demonstrated with a finite number of test cases**

An Infeasibile Decision

Dependence of one decision on another might lead to an infeasible combination.

Consider, for example, the following sequence of statements.

```
int A, B, C;
input (A, B, C);
if (A > 10 and B > 30) {
    S1 = f1 (A, B, C);
    if (A < 5 and B > 10) {  ← infeasible decision
        S2 = f2 (A, B, C);         A > 10, A < 5
    }
}
```

Replace line 3 with

if (A>10 and foo())

What if there is a side-effect of foo that affects the value of A?

- **the condition at line 5 might then be feasible!**

```
1   int A, B, C;
2   input (A, B, C);
3   if (A > 10 and foo()) {
4       S1 = f1 (A, B, C);
5       if (A < 5 and B > 10) {  ← feasible decision?
6           S2 = f2 (A, B, C);
7       }
8   }
```

Infeasibility vs Reachability

Infeasibility is different from reachability.

A decision might be reachable but not feasible... is vice versa true?

The second decision is not reachable due an error at line 3.

It may, however, be feasible.

- **If a decision cannot be reached, does that not make it infeasible by definition?**

```
1    int A, B, C;
2    input (A, B, C);
3    if (A > A + 1) {
4        S1 = f1 (A, B, C);
5        if (A > 5 and B > 10) {
6            S1 = f2 (A, B, C);
7        }
8    }
```

Feasibility and Reachability

Feasibility is typically used when talking about conditions and decisions.

- **decision coverage**
- **condition coverage**

Unreachable <u>code</u> is usually defined as a special type of infeasible code.

Reachability is typically used when talking about executing some component of a program.

- **statement coverage**
- **block coverage**

Adequacy and Infeasibility

In the presence of one or more infeasible elements in the coverage domain, a test is considered adequate when all <u>feasible</u> elements in the domain have been covered.

Programmers may not be concerned with infeasible elements…

Testers attempting to obtain code coverage are…

It is only during the attempt to construct a test case to cover an element that one may realize the infeasibility of an element.

Error Detection and Test Enhancement

The purpose of test enhancement is to determine test cases that

- **test the untested parts of a program**
- **exercise the program using uncovered portions of the input domain**

Even the most carefully designed tests based exclusively on requirements can be enhanced.

The more complex the set of requirements, the more likely it is that a test set designed using requirements is inadequate with respect to even the simplest of various test adequacy criteria.

Mathur, Example 7.9

A program to meet the following requirements is to be developed.

R_1: Upon start the program offers the following three options to the user:

- Compute x^y for integers x and y ≥ 0.
- Compute the factorial of integer $x \geq 0$.
- Exit.

$R_{1.1}$: If the “Compute x^y” option is selected then the user is asked to supply the values of x and y, x^y is computed and displayed. The user may now select any of the three options once again.

$R_{1.2}$: If the “Compute factorial x” option is selected then the user is asked to supply the value of x andfactorial of x is computed and displayed. The user may now select any of the three options once again.

$R_{1.3}$: If the “Exit” option is selected the program displays a goodbye message and exits.

Program P7.5 was written to meet the above requirements.

```
begin
    int x, y;
    int product, request;
    #define exp=1
    #define fact=2
    #define exit=3

    get_request (request); // Get user request (one of three possibilities).
    product=1; // Initialize product.

  // Set up the loop to accept and execute requests.

    while (request ≠ exit) {
```

```
// Process the “exponentiation” request.
  if(request == 1){
    input (x, y); count=y;
    while (count > 0){
      product=product * x; count=count-1;
    }
  } // End of processing the “exponentiation” request.
// Process “factorial” request.
  else if(request == 2){
    input (x); count=x;
    while (count >0){
      product=product * count; count=count-1;
    }
  } // End of processing the “factorial” request.
```

```
  // Process "exit" request.
    else if(request == 3){
      output( "Thanks for using this program. Bye!"); break; // Exit the loop.
    } // End of if.
    output(product); // Output the value of exponential or factorial and re-enter the loop.
    get_request (request); // Get user request once again and jump to loop begin.
    }
end
```

Suppose now that T containing three tests has been developed to test whether or not our program meets its requirements.

T = {<request=1, x=2, y=3>, <request=2, x=4>, <request=3>}

For the first two of the three requests the program correctly outputs 8 and 24, respectively.

The program exits when executed against the last request.

This program behavior is correct and hence one might conclude that the program is correct.

Do you believe this conclusion is correct?

Evaluate T against the path coverage criterion.

Go back to program P7.5 and identify the paths not covered by T.

The coverage domain consists of all paths that traverse each of the three loops <u>zero</u> and <u>once</u> in the same or different executions of the program.

We continue with a “tricky” uncovered path.

Consider the path p that

- begins execution at line 1
- reaches the outermost while at line 10
- then the first if at line 12
- followed by the statements that compute the factorial starting at line 20
- then the code to compute the exponential starting at line 13

p is traversed when

- the first input request is to compute the factorial of a number
- followed by a request to compute the exponential

T does not exercise p, therefore T is inadequate with respect to the path coverage criterion.

To cover p we construct the following test:

T' = {<request=2, x=4>, <request=1, x=2, y=3>,
<request=3>}

When the values in T' are input to program P7.5 in the sequence given, the program

- **correctly outputs 24 as the factorial of 4**
- **incorrectly outputs 192 as the value of 2^3**

T' traverses our "tricky" path which makes the computation of the exponentiation begin without initializing product.

- **the code at line 14 begins with the value of product set to 24**

Multiple Executions

In the previous example we constructed two test sets T and T'.

Notice that both T and T' contain three tests, one for each value of variable request.

Should T (or T') be considered a single test or a sequence of three tests?

We assumed that all three tests, one for each value of request, are input in a sequence during a single execution of the test program.

Hence we consider T as a test set containing one test case and write it as follows:

T = {t_1: <<request=1,x=2,y=3>
→ <request=2,x=4>
→ <request=3>>}

<< >> groups all the values in the test case
→ indicates the sequencing of variable values

T’ = {t_2: <<request=2,x=4>
→ <request=1,x=2,y=3>
→ <request=3>>}

Combining T and T’ we get

T’’ = {t_1: <<request=1,x=2,y=3>
→ <request=2,x=4>
→ <request=3>>,
t_2: <<request=2,x=4>
→ <request=1,x=2,y=3>
→ <request=3>>}

Summary – Things to Remember

Adequate, inadequate test coverage

Measuring test coverage

Adequate test coverage does not mean finding all defects

Feasible vs infeasible

Test enhancement
- **to 100% coverage**
- **adding new test criteria**

Questions and Answers