Test Adequacy Measurement and Enhancement

Professor Mark Gabel
Slides Courtesy of W. Eric Wong
Department of Computer Science
University of Texas at Dallas

Learning Objectives

- What is test adequacy? What is test enhancement? When should we measure test adequacy and how should we use it to enhance tests?
- Control-flow based test adequacy: statement/block coverage, decision coverage, etc.
- Data-flow base test adequacy: c-uses, p-uses, all-uses
- Strengths and limitations of code coverage-based measurement of test adequacy
- The “subsumes” & “probably better” relations among coverage criteria
- Tools for the measurement of code coverage:
  - γSuds /Bellcore/Telcordia/Eclipse/IntelliJ/VisualStudio
Test Adequacy

What is adequacy?

- It is a measure of the completeness of a test suite. Coverage Metrics are ways of implementing the more general concept of adequacy.

- Consider a program $P$ written to meet a set of functional requirements $R$. We notate such a $P$ and $R$ as $(P, R)$. Let $R$ contain $n$ requirements labeled $R_1, R_2, \ldots, R_n$.

- 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 $R$.

  Also, $P$ has been executed against each test in $T$ and has produced correct behavior.

- We now ask: Is $T$ good enough? This question can be stated differently as: Has $P$ been tested thoroughly?, or as: Is $T$ adequate?
Measurement of adequacy

- In the context of software testing, the terms “thorough,” “good enough,” and “adequate,” used in the questions above, have the same meaning.

- Adequacy is measured for a given test set designed to test $P$ to determine whether or not $P$ meets its requirements.

- This measurement is done against a given criterion $C$. A test set is considered adequate with respect to criterion $C$ when it satisfies $C$. The determination of whether or not a test set $T$ for program $P$ satisfies criterion $C$ depends on the criterion itself and is explained later.

Example I

Program `sumProduct` must meet the following requirements:

$R_1$ Input two integers, say $x$ and $y$, from the standard input device

$R_{2.1}$ Find and print to the standard output device the sum of $x$ and $y$ if $x < y$

$R_{2.2}$ Find and print to the standard output device the product of $x$ and $y$ if $x \geq y$

(board)
Example I (contd.)

Suppose now that the test adequacy criterion $C$ is specified as:

$C$: A test suite $T$ 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 $r$.

Obviously, $T$={($t$: $x<2$, $y=3$)} (which has $x < y$) is inadequate with respect to $C$ for program sumProduct. The test case $t$ in $T$ 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 white-box test adequacy criterion if the corresponding coverage domain $C_e$ depends solely on program $P$ under test.

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

We want to measure the adequacy of $T$. Given that $C_e$ has $n \geq 0$ elements, we say that $T$ covers $C_e$ if for each element $e'$ in $C_e$ there is at least one test case in $T$ that tests $e'$.

$T$ is considered adequate with respect to $C$ if it covers all elements in the coverage domain.
$T$ is considered inadequate with respect to $C$ if it covers $k$ elements of $C_e$ where $k<n$.

The fraction $k/n$ is a measure of the extent to which $T$ is adequate with respect to $C$. This fraction is also known as the coverage of $T$ with respect to $C$, $P$, and $R$.

Example II

Let us again consider the following criterion: “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 $r$.”

In this case the finite set of elements $C_e = \{R_1, R_{2.1}, R_{2.2}\}$.
$T = \{t: <x=2, y=3>\}$ covers $R_1$ and $R_{2.1}$ but not $R_{2.2}$.
Hence $T$ is not adequate with respect to $C$.
The coverage of $T$ with respect to $C$, $P$, and $R$ is $2/3$. 
Example III

Consider the following criterion: “A test $T$ for program $(P, R)$ is considered adequate if each path in $P$ is traversed at least once.”

Assume that $P$ has exactly two paths, one corresponding to condition $x<y$ and the other to $x \geq y$. We refer to these as $p_1$ and $p_2$, respectively. For the given adequacy criterion $C$ we obtain the coverage domain $C_e$ to be the set \{p_1, p_2\}.

Example III (contd.)

To measure the adequacy of $T$ of \texttt{sumProduct} against $C$, we execute $P$ against each test case in $T$.

As $T=\{t: <x=2, y=3>\}$ contains only one test for which $x<y$, only the path $p_1$ is executed. Thus, \textit{the coverage of $T$ with respect to $C$, $P$, and $R$ is 0.5 and hence $T$ is not adequate with respect to $C$.}

We can also say that $p_1$ is tested and $p_2$ is \textit{not tested}.
Example IV

```
sumProduct-1
1 begin
2 int x, y;
3 input (x, y);
4 sum=x+y;
5 output (sum);
6 end
```

This program is obviously incorrect as per the requirements of `sumProduct`.

There is only one path denoted as $p_1$. This path traverses all the statements.

Using the path-based coverage criterion $C$, we get coverage domain $C_e=\{p_1\}$. $T=\{t: \langle x=2, y=3\rangle\}$ is adequate w.r.t. $C$ but does not reveal the error.

Lesson

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

However, this does not diminish in any way the need for the measurement of test adequacy as increasing coverage might reveal an error!
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.

*Identification of this deficiency helps in the enhancement of the inadequate test set.* Enhancement in turn is also likely to test the program in ways it *has not been tested before* such as testing untested portion, or testing the features in a sequence different from the one used previously. *Testing the program differently than before raises the possibility of discovering any uncovered errors.*
This program is correct as per the requirements of \texttt{sumProduct}. It has two paths denoted by $p_1$ and $p_2$. (see Example III) $C_e=\{p_1, p_2\}$. $T=\{t: x=2, y=3\}$ is \textit{inadequate} w.r.t. the path-based coverage criterion $C$.

For \texttt{sumProduct2}, 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 $\{x=3, y=1\}$. Adding this test to $T$ and denoting the expanded test set by $T'$ we get:

$T' = \{t_1: x=3, y=4, t_2: x=3, y=1\}$

Executing \texttt{sumProduct-2 against the two tests} in $T'$ causes paths $p_1$ and $p_2$ are traversed. Thus $T'$ is \textit{adequate with respect to the path coverage criterion}. 
Test Enhancement: Example (contd.)

Consider a program intended to compute \(x^y\) given integers \(x\) and \(y\). For \(y<0\) the program skips the computation and outputs a suitable error message.

```plaintext
begin
  int x, y;
  int product, count;
  input (x, y);
  if(y \geq 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
```
Test Enhancement: Example (cont’d.)

Suppose that test set $T$ is considered adequate if it tests the exponentiation program for \textit{at least one zero and one non-zero value of each of the two inputs} $x$ and $y$.

The \textit{coverage domain} for $C$ can be determined using $C$ alone and without any inspection of the program. For $C$ we get $C_e=\{x=0, y=0, x\neq 0, y\neq 0\}$. Again, one can derive an adequate test set for the program by an examination of $C_e$. One such test set is

$$T=\{t_1: (x=0, y=1), \ t_2: (x=1, y=0)\}.$$

---

Test Enhancement: Example: Path Coverage

Criterion $C$ of the previous example is a \textit{black-box coverage criterion} as it \textit{does not require an examination of the program under test} for the measurement of adequacy.

Let us now consider the \textit{path coverage criterion} defined in an earlier example. An examination of the exponentiation program reveals that it has \textit{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.
Example: Path Coverage (contd.)

Given that \( y \) is any non-negative integer, the number of paths can be arbitrarily large. This simple analysis of paths in exponentiation reveals that 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:

\[ A \text{ test } T \text{ is considered adequate if it tests all paths. In case the program contains a loop, then it is adequate to traverse the loop body zero times and once. (boundary-interior) } \]

The modified path coverage criterion leads to \( C_e=\{p_1, p_2, p_3\} \). The elements of \( C_e \) are enumerated below with respect to flow graph for the exponentiation program.

\( p_1: [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9] \) (orange path)

\( p_2: [1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 9] \) (red path)

\( p_3: [1 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 9] \) (green path)
Example: Path Coverage (contd.)

We measure the adequacy of $T$ with respect to $C'$. As $T$ does not contain any test with $y < 0$, $p_3$ (the green path) remains uncovered. Thus, the coverage of $T$ with respect to $C'$ is $2/3 = 0.66$

$T = \{t_1: <x=0, y=1>, t_2: <x=1, y=0>\}$ (from slide 20)

Example: Path Coverage (contd.)

Any test case with $y < 0$ will cause $p_3$ to be traversed. Let us use $t_3: <x=5, y=-1>$. Test $t$ covers path $p_3$ and $P$ behaves correctly. We add $t_3$ to $T$. We have covered all feasible elements of $C_e$. The enhanced test set is:

$T = \{t_1: <x=0, y=1>, t_2: <x=1, y=0>, t_3: <x=5, y=-1>\}$
Repeat until “100%”. Is it as simple as this?

**General Process**

- Requirements (R)
- Program (P)
- Tests (T)

- Define Coverage Domain
- Test and Record

- Coverage Domain ($C_n$)
- Covered Elements

- Compare
- Metric

- Enhance Tests, Debug Program

---

**Infeasibility and Test Adequacy**

**No.** There are often elements of a coverage domain we cannot cover. 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.

There does not exist an algorithm that would analyze a given program and determine if a given element in the coverage domain is infeasible or not. Thus 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 indeed the element under consideration is covered.

“Line 20 can be covered by using inputs (3,25)”

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. Thus, an attempt to enhance a test set by executing a test \( t \) aimed at covering element \( e \) of program \( P \), might fail.

“Line 20 cannot be covered by any execution on any input (because)”

Question: what other mathematical or scientific processes are subject to this problem?

Adequacy and Infeasibility

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

While programmers might not be concerned with infeasible elements, testers attempting to obtain code coverage are. Prior to test enhancement, a tester usually does not know which elements of a coverage domain are infeasible. Unfortunately, it is only during an attempt to construct a test case to cover an element that one might realize the infeasibility of an element.
Adequacy and Infeasibility

Infeasibility is a central problem in measuring test adequacy. Infeasible elements of the coverage domain (e.g. unreachable code, impossible paths) cause inaccuracy in coverage metrics and are very time consuming to evaluate.

Silver lining: the process of analyzing elements for feasibility can be highly beneficial.

For example, if your project has truly dead code (i.e. infeasible elements of the ‘statement coverage’ domain), then there might be a design problem.

(Break to example)

Error Detection and Test Enhancement

The purpose of test enhancement is to determine test cases that test the untested ‘parts’ of a program. 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.
Example

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 \(y \geq 0\).
- 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 \(x\) and factorial 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.

Example (contd.)

Consider this program written to meet the above requirements.

```
begin
1 int x, y;
2 int product, request;
3 #define exp=1
4 #define fact=2
5 #define out=3
6
7 get_request (request); // Get user request (one of three possibilities).
8 product=1; // Initialize product.
9 // Set up the loop to accept and execute requests.
10 while (request != exit)
11
12 if (request==exp)
13 input (x, y); count=y;
14 while (count > 0){
15 product=product * x; count=count-1;
16 }
17 // End of processing the “exponentiation” request.
18
19 else if (request==fact)
20 input (x); count=x;
21 product=product * x; count=joint-1;
22 }
23 // End of processing the “factorial” request.
24
25 output(product); // Output the value of exponential or factorial and re-enter the loop.
26 input (request);
27 // Get user request once again and jump to loop begin.
28 wait
```

In-class exercise.

Generate a set of test cases for this program.

(Or simply find the bug)
Consider this program written to meet the above requirements.

```c
begin
int x, y;
int product, request;
define exp-1
#define fact-2
#define out-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 \neq end) 

// Process the “exponentiation” request.
if(request == 1)
input (x, y); count=y;
while (count > 0) 
    product=product \times count; count=count-1;
}
End of processing the “exponentiation” request.

// Process “factorial” request.
else if(request == 2)
input (x); count=x; product=x;
while (count > 0) 
    product=product \times count; count=count-1;
}
End of processing the “factorial” request.

// Output the value of exponential or factorial and re-enter the loop.
output (product); // Output user request once again and jump to loop begin.
```

Example (contd.)

Suppose now that the following test set has been developed to test whether or not our program meets its requirements (black-box criterion).

\[ T = \{<\text{request}=1, x=2, y=3>, <\text{request}=2, x=4>, <\text{request}=3>\} \]

For the first two requests (exponential followed by factorial), the program correctly outputs 8 and 24. The program exits when executed against the third request. This program’s behavior is correct and hence one might conclude that the program is correct.

However, it will not be difficult for you to believe that this conclusion is incorrect.
Example (contd.)

Let us now evaluate $T$ against the path coverage criterion.

Construct the control flow graph of the example program and identify the paths not covered by $T$.

The coverage domain consists of all paths that traverse each of the three loops zero, once and twice in the same or different executions of the program.

Example (contd.)

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, and then the code to compute the exponential starting at line 13.

$p$ is traversed when the program is launched and the first input request is to compute the factorial of a number, followed by a request to compute the exponential. It is easy to verify that the sequence of requests in $T$ (on slide 34) does not exercise $p$. Therefore, $T$ is inadequate with respect to the path coverage criterion.
Example (contd.)

To cover $p$ we construct the following test:

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

When the values in $T'$ are input to our example program in the sequence given, the program correctly outputs 24 as the \textit{factorial} of 4 but incorrectly outputs 192 (24 * 2 * 2 * 2) as the value of $2^3$.

This happens because $T'$ traverses our “tricky” path \textit{which makes the computation of the exponentiation begin without initializing product}. In fact the code at line 14 begins with the value of \textit{product} set to 24.

Example (contd.)

In our effort to increase the path coverage we constructed $T'$. \textit{Execution of the test program on $T'$ did cover a path that was not covered earlier and revealed an error in the program.}

This example has illustrated a \textit{benefit of test enhancement based on code coverage}.

Question: why were we wrong the first time?

\textit{Why did analyzing the code help?}
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 the variable request. Should $T$ (or $T'$) be considered a single test or a sequence of three tests?

$$T'=\{\langle \text{request}=2, x=4 \rangle, \langle \text{request}=1, x=2, y=3 \rangle, \langle \text{request}=3 \rangle\}$$

Multiple executions (contd.)

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, it as follows:

$$T = \{ t_1 : \langle \langle \text{request}=1, x=2, y=3 \rangle \rightarrow \langle \text{request}=2, x=4 \rangle \rightarrow \langle \text{request}=3 \rangle \rangle \}$$

$$T' = \{ t_2 : \langle \langle \text{request}=2, x=4 \rangle \rightarrow \langle \text{request}=1, x=2, y=3 \rangle \rightarrow \langle \text{request}=3 \rangle \rangle \}$$

$$T'' = T \cup T'$$
Statement and block coverage

Declarations and basic blocks

Any program written in a procedural language consists of a sequence of statements. Some of these statements are declarative, such as the `#define` and `int` statements in C, while others are executable, such as the `assignment`, `if`, and `while` statements in C and Java.

Recall that a basic block is a sequence of consecutive statements that has exactly one entry point and one exit point. For any procedural language, adequacy with respect to the statement coverage and block coverage criteria are defined next.

Notation: $(P, R)$ denotes program $P$ subject to requirements $R$. 
Statement coverage

The statement coverage of $T$ with respect to $(P, R)$ is computed as $S_c/(S_e-S_i)$, where $S_c$ is the number of statements covered, $S_i$ is the number of unreachable statements, and $S_e$ is the total number of executable statements in the program, i.e., the size of the coverage domain.

$T$ is considered adequate with respect to the statement coverage criterion if the statement coverage of $T$ with respect to $(P, R)$ is 1.

Block coverage

The block coverage of $T$ with respect to $(P, R)$ is computed as $B_c/(B_e-B_i)$, where $B_c$ is the number of basic blocks covered, $B_i$ is the number of unreachable blocks, and $B_e$ is the total number of executable blocks in the program, i.e., the size of the block coverage domain.

$T$ is considered adequate with respect to the block coverage criterion if the statement coverage of $T$ with respect to $(P, R)$ is 1.
Example: statement coverage

Coverage domain: $S_e = \{4, 5, 6, 7, 8, 9, 12, 13\}$
Let $T_1 = \{t_1: <x= -1, y=-1>, t_2: <x=1, y=1>\}$

Statements covered:
$t_1$: 4, 5, 6, 7, 8 and 13
$t_2$: 4, 5, 6, 12, and 13.

$S_e = 7, S_i = 1, S_e = 8$. The statement coverage for $T$ is $7/(8 - 1) = 1$. Hence we conclude that $T_1$ is adequate for $(P, R)$ with respect to the statement coverage criterion. Note: 9 is unreachable.

Question: do coverage tools know what statements are reachable?

Example: block coverage

Coverage domain: $B_e = \{1, 2, 3, 4, 5\}$

Let $T_2 = \{t_1: <x= -1 \ y=-1>, t_2: <x= -3 \ y=-1>, t_3: <x= -1 \ y=-3>\}$

Blocks covered:
$t_1$: Blocks 1, 2, 5
$t_2, t_3$: same coverage as of $t_1$.

$B_e = 5, B_c = 3, B_i = 1$.
Block coverage for $T_2 = 3/(5 - 1) = 0.75$.
Hence $T_2$ is not adequate for $(P, R)$ with respect to the block coverage criterion.
Example: block coverage (contd.)

T₁ is adequate w.r.t. block coverage criterion.
Verify this statement!

Also, if test t₂ in T₁ is added to T₂, we obtain a test set adequate with respect to the block coverage criterion for the program under consideration.
Verify this statement!

Question: how do statement coverage and block coverage differ?

Condition and decision coverage
Conditions

Any expression that evaluates to true or false constitutes a condition. Such an expression is also known as a predicate.

Given that A, B, and D are Boolean variables, and x and y are integers, A, x > y, A OR B, A AND (x<y), (A AND B) are sample conditions.

Note that in the C programming language, x and x+y are valid conditions, and the constants 1 and 0 correspond to, respectively, true and false.

Simple and compound conditions

A simple condition does not use any Boolean operators (connectives) except for the not operator. It is made up of variables and at most one relational operator from the set { <, ≤, >, ≥, ==, ≠ }.

Simple conditions are also referred to as atomic or elementary conditions because they cannot be parsed any further into two or more conditions.

A compound condition is made up of two or more simple conditions joined by one or more Boolean connectives.
Conditions as decisions

Any condition can serve as a decision in an appropriate context within a program. Most high level languages provide if, while, and switch, statements to serve as contexts for decisions.

Outcomes of a decision

A decision can have three possible outcomes, true, false, and undefined.

In some cases the evaluation of a condition might fail in which case the corresponding decision’s outcome is undefined.
Coupled conditions

How many simple conditions are there in the compound condition: C=(A AND B) OR (C AND A)? The first occurrence of A is said to be coupled to its second occurrence.

Does C contain three or four simple conditions? Both answers are correct depending on one's point of view. Indeed, there are three distinct conditions A, B, and C. The answer is four when one is interested in the number of occurrences of simple conditions in a compound condition.

Conditions within assignments

Strictly speaking, a condition becomes a decision only when it is used in the appropriate context such as within an if statement.

At line 4, x<y does not constitute a decision and neither does A*B.

```
1   A = x < y; // A simple condition assigned to a Boolean variable A.
2   X = P or Q; // A compound condition assigned to a Boolean variable x.
3   z = y + z * s; // The condition will be true if z = 1 and false otherwise.
4   A = x < y; z = A + B; // A is used in a subsequent expression for x but not as a decision.
```
Decision coverage

A decision is considered covered if the flow of control has been diverted to all possible destinations that correspond to this decision, i.e., all outcomes of the decision have been taken.

This implies that, for example, the expression in the if or a while statement has evaluated to true in some execution of the program under test and to false in the same or another execution.

Decision coverage: switch statement

A decision implied by the switch statement is considered covered if during one or more executions of the program under test the flow of control has been diverted to all possible destinations.
Decision coverage: Example
(Why does decision coverage matter?)

```
1 begin
2 int x, z;
3 input (x);
4 if(x<0)
5   x = -x;
6   z = foo-1(x);
7 output(z);
8 end
```

Requirement:
This program inputs an integer $x$, and if $x < 0$, transforms it into a positive value before invoking \texttt{foo-1} to compute the output $z$.

This program is supposed to compute $z$ using \texttt{foo-2} when $x \geq 0$.

The code on the left has an error.

There should have been an \texttt{else} clause before this statement.

---

Decision coverage: Example (contd.)

Consider the test set $T = \{ t_1; x = -5 \}$.
It is adequate with respect to \texttt{statement and block} coverage criteria, but does not reveal the error.

Another test set $T' = \{ t_1; x = -5 \ \ t_2; x = 3 \}$ does reveal the error. It covers the decision whereas $T$ does not. Check!

This example illustrates how and why decision coverage might help in revealing an error that is not revealed by a test set adequate with respect to \texttt{statement and block coverage}.

There should have been an \texttt{else} clause before this statement.
Decision coverage: Computation

The decision coverage of \( T \) with respect to \((P, R)\) is computed as \( \frac{D_c}{D_e - D_i} \), where \( D_c \) is the number of decisions covered, \( D_i \) is the number of infeasible decisions, and \( D_e \) is the total number of decisions in the program, i.e., the size of the decision coverage domain.

\( T \) is considered adequate with respect to the decisions coverage criterion if the decision coverage of \( T \) with respect to \((P, R)\) is 1.

Alternative formulation: \( \frac{D_p + 2 \times D_c}{2 \times (D_e - D_i)} \)

Decision coverage: domain

The domain of decision coverage consists of all decisions in the program under test.
Condition coverage

A decision can be composed of a simple condition such as \(x < 0\), or of a more complex condition, such as \(((x < 0 \text{ AND } y < 0) \text{ OR } (p \geq q))\).

AND, OR, XOR are the logical operators that connect two or more simple conditions to form a compound condition.

A simple condition is considered covered if it evaluates to true and false in one or more executions of the program in which it occurs.

A compound condition is considered covered if each simple condition it is comprised of is also covered.

Decision and condition coverage

Decision coverage is concerned with the coverage of decisions regardless of whether or not a decision corresponds to a simple or a compound condition. Thus in the statement

```java
1    if (x < 0 and y < 0) {
2      z=foo(x,y);
```

there is only one decision that leads control to line 2 if the compound condition inside the if evaluates to true.

However, a compound condition might evaluate to true or false in one of several ways.
With this evaluation characteristic in view, compilers often generate code that uses short circuit evaluation of compound conditions.

Here is a possible translation:

```plaintext
1   if (x < 0 and y < 0) {
2       z=foo(x,y);
}

The condition at line 1 evaluates to \textbf{false} when \(x \geq 0\) regardless of the value of \(y\).

Another condition, such as \((x<0 \text{ OR } y<0)\), evaluates to \textbf{true} regardless of the value of \(y\), when \(x<0\).

With this evaluation characteristic in view, compilers often generate code that uses \textbf{short circuit} evaluation of compound conditions.

We now see two decisions, one corresponding to each simple condition in the \textbf{if} statement.
Condition coverage

The condition coverage of T with respect to (P, R) is computed as 
\[ C_c / (C_e - C_i) \], where 
- \( C_c \) is the number of simple conditions covered,
- \( C_i \) is the number of infeasible simple conditions, and
- \( C_e \) is the total number of simple conditions in the program.

T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

An alternate formula where each simple condition contributes 2, 1, or 0 to \( C_c \) depending on whether it is covered, partially covered, or not covered, respectively. is:

\[
\frac{\text{\# covered}}{2 \times (\text{\# total} - \text{\# infeasible})}
\]

Condition coverage: Example

Partial specifications for computing z:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Output (z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; 0</td>
<td>y &lt; 0</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

This program has a bug based on the specification.

```
1 begin
2 int x, y, z;
3 input (x, y);
4 if (x < 0 and y < 0)
5   z = foo1(x, y);
6 else
7   z = foo2(x, y);
8 output(z);
9 end
```
Consider the test set:

\[ T = \{ t_1 : < x = -3, y = -2 >, t_2 : < x = 4, y = -2 > \} \]

Check that \( T \) is adequate with respect to the \textit{statement, block, and decision} coverage criteria and the program behaves correctly against \( t_1 \) and \( t_2 \).

\[ C_c = 1, \quad C_e = 2, \quad C_i = 0. \] Hence condition coverage for \( T = 0.5 \).

Add the following test case to \( T \):

\[ t_3 : < x = 3, y = 4 > \]

Check that the enhanced test set \( T \) is adequate with respect to the \textit{condition coverage criterion} and possibly reveals an error in the program.

\( \rightarrow \) The program shows \( z = \text{foo2}(x,y) \)
\( \rightarrow \) But the specifications says \( z = \text{foo1}(x,y) \)

\textbf{Under what conditions will a possible error at line 7 be revealed by} \( t_3 \)?
\( \rightarrow ? \)
Condition/decision coverage

When a decision is composed of a compound condition, decision coverage does not imply that each simple condition within a compound condition has taken both values true and false. (slide 68)

Condition coverage ensures that each component simple condition within a condition has taken both values true and false.

**Question:** Does the condition coverage require each decision to take all its outcomes?

Condition/decision coverage: Example

Consider the following program and two test sets.

```plaintext
1 begin
2 int x, y, z;
3 input (x, y);
4 if(x<0 and y<0)
5 z=foo-1(x,y);
6 else
7 z=foo-2(x,y);
8 output(z);
9 end
```

In class exercise:
Is \( T_1 \) is adequate with respect to decision coverage?
Is \( T_1 \) adequate with respect to condition coverage?
How about \( T_2 \)?
Condition/decision coverage: Definition

The condition/decision coverage of T with respect to (P, R) is computed as 
\((C_c+D_c)/((C_e-C_i)+(D_e-D_i))\), where 

- \(C_c\) is the number of simple conditions covered, 
- \(D_c\) is the number of decisions covered, 
- \(C_e\) and \(D_e\) are the number of simple conditions and decisions respectively, 
- \(C_i\) and \(D_i\) are the number of infeasible simple conditions and decisions, respectively.

Condition/decision coverage: Example

In class exercise: Is T adequate with respect to the condition/decision coverage criterion?
Multiple Condition Coverage

Consider *a compound condition with two or more simple conditions*. Using condition coverage on some compound condition C implies that each simple condition within C needs to be evaluated to true and false.

However, does it imply that all combinations of the values of the individual simple conditions in C have been exercised?
Multiple condition coverage: Example

Consider D=(A<B) OR (A>C) composed of two simple conditions A<B and A>C. The four possible combinations of the outcomes of these two simple conditions are enumerated in the table. Consider T:

<table>
<thead>
<tr>
<th></th>
<th>A &lt; B</th>
<th>A &gt; C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

\[ T = \{ t_1 : < A = 2 \quad B = 3 \quad C = 1 > \} \]

Check: Is T 100% w.r.t. the decision coverage?
Check: Is T 100% w.r.t. the condition coverage?
Check: Does T cover all four combinations?
Check: Does T' cover all four combinations?

\[ T' = \{ t_1 : < A = 2 \quad B = 3 \quad C = 1 > \}
\]

Multiple condition coverage: Definition

Suppose that the program under test contains a total of n decisions. Assume also that each decision contains \( k_1, k_2, ..., k_n \) simple conditions. Each decision has several combinations of values of its constituent simple conditions.

For example, decision \( i \) will have a total of \( 2^{k_i} \) combinations. Thus the total number of combinations to be covered is

\[ \sum_{i=1}^{n} 2^{k_i} \]
Multiple condition coverage: Definition (contd.)

The multiple condition coverage of T with respect to \((P, R)\) is computed as \(C_c/(C_e-C_i)\), where \(C_c\) is the number of combinations covered, \(C_i\) is the number of infeasible simple combinations, and \(C_e\) is the total number of combinations in the program.

T is considered adequate with respect to the multiple condition coverage criterion if the condition coverage of T with respect to \((P, R)\) is 1.

Practically speaking, feasibility is a major problem here.

Multiple condition coverage: Example

Consider the following program with specifications in the table.

```plaintext
begin
int A, B, C, S=0;
input (A, B, C);
if(A<B and A>C) S=f1(A, B, C);
if(A<B and A<C) S=f2(A, B, C);
if(A>B and A>C) S=f4(A, B, C);
output(S);
end
```

<table>
<thead>
<tr>
<th>A &lt; B</th>
<th>A &gt; C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>(f1(P, Q, R))</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>(f2(P, Q, R))</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>(f3(P, Q, R))</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>(f4(P, Q, R))</td>
</tr>
</tbody>
</table>

There is an obvious error in the program, computation of \(S\) for one of the four combinations, line 3 in the table, has been left out.
Multiple condition coverage: Example (contd.)

1 begin
2 int A, B, C, S=0;
3 input (A, B, C);
4 if(A>B and A>C) S=f1(A, B, C);
5 if(A>B and A≤C) S=f2(A, B, C);
6 if(A≥B and A≤C) S=f4(A, B, C);
7 output(S);
8 end

Is T adequate wrt decision coverage?
Multiple condition coverage?
Does it reveal the error?

\[ T = \begin{cases} 
    t_1 : & A = 2, B = 3, C = 1 \\
    t_2 : & A = 2, B = 1, C = 3 \\
    t_3 : & A = 2, B = 3, C = 5 
\end{cases} \]

Multiple condition coverage: Example (contd.)

1 begin
2 int A, B, C, S=0;
3 input (A, B, C);
4 if(A>B and A>C) S=f1(A, B, C);
5 if(A>B and A≤C) S=f2(A, B, C);
6 if(A≥B and A≤C) S=f4(A, B, C);
7 output(S);
8 end

Is T’100% with respect to decision coverage?

\[ T' = \begin{cases} 
    t_1 : & A = 2, B = 3, C = 1 \\
    t_2 : & A = 2, B = 1, C = 3 \\
    t_3 : & A = 2, B = 3, C = 5 
\end{cases} \]

Does T’ reveal the error?
Multiple condition coverage: Example (contd.)

In class exercise:
Is T’ 100% w.r.t. simple condition coverage?
Is T’ 100% w.r.t. multiple condition coverage?

Now add a test to T’ to cover the uncovered combinations.
Does your test reveal the error?
If yes, then under what conditions?

Dataflow Coverage
Basic concepts

We will now examine some test adequacy criteria based on the flow of “data” in a program. This is in contrast to criteria based on “flow of control” that we have examined so far.

Test adequacy criteria based on the flow of data are useful in improving tests that are adequate with respect to control-flow based criteria. Let us look at an example.

Example: Test enhancement using data flow

```plaintext
begin
int x, y; float z;
input (x, y);
z=0;
if (x=0)
z=z+y;
else z=x*y;
if (y=0) ← This condition should be (y!=0 and x!=0)
z=z/x;
else z=x*y;
output(z);
end
```

Question: Does the following test set reveal the error?

<table>
<thead>
<tr>
<th>Test</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>t₂</td>
<td>1</td>
<td>1</td>
<td>-1.0</td>
</tr>
</tbody>
</table>
Neither of the two tests force the use of $z$ defined on line 6, at line 9. To do so one requires a test that causes conditions at lines 5 and 8 to be true (i.e., need to satisfy $x = 0$ and $y \neq 0$).

The test which we have does not force the execution of this path and hence the divide by zero error is not revealed.

Example (contd.)

Verify that the following test set which covers all def-use pairs of $z$ and reveals the error.

<table>
<thead>
<tr>
<th>Test</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>def-use pairs covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>(4,6), (6,10), (10, 11)</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>(4,7), (7,9), (9, 11)</td>
</tr>
<tr>
<td>$t_3$</td>
<td>0</td>
<td>1</td>
<td></td>
<td>(4,6), (6,9)</td>
</tr>
<tr>
<td>$t_4$</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>(4,7), (7,10), (10, 11)</td>
</tr>
</tbody>
</table>

* In the pair $(t_i, t_j)$, $z$ is defined in $t_i$ and used in line $t_j$.
Definitions and uses

A program written in an imperative language, such as C and Java, contains variables. Variables are defined by assigning values to them and are used in expressions.

Statement \( x = y + z \) defines variable \( x \) and uses variables \( y \) and \( z \)

Statement `scanf("%d %d", &x, &y)` defines variables \( x \) and \( y \)

Statement `printf("Output: %d \n", x+y)` uses variables \( x \) and \( y \)

Definitions and uses (contd.)

A parameter \( x \) passed as `call-by-value` to a function, is considered as a use of (or a reference to) \( x \)

A parameter \( x \) passed as `call-by-reference`, can serve as a definition and use of \( x \)
Definitions and uses: Pointers

Consider the following sequence of statements that use pointers.

\[
\begin{align*}
    z &= \& x; \\
    y &= z + 1; \\
    *z &= 25; \\
    y &= *z + 1;
\end{align*}
\]

The first of the above statements defines a pointer variable \( z \)
the second defines \( y \) and uses \( z \)
the third defines \( x \) through the pointer variable \( z \), and
the last defines \( y \) and uses \( x \) accessed through the pointer variable \( z \).

Definitions and uses: Arrays

Arrays are also tricky. Consider the following declaration and two statements in C:

\[
\begin{align*}
    \text{int } & \text{A[10]; } \\
    \text{A[1]} &= x + y;
\end{align*}
\]

The first statement defines variable \( A \).
The second statement defines \( A \) and uses \( i, x \), and \( y \).

Alternate: second statement defines \( A[i] \) and not the entire array \( A \).
The choice of whether to consider the entire array \( A \) as defined or the specific element depends upon how stringent is the requirement for coverage analysis.
C-use

Uses of a variable that occurs within an expression as part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions, are classified as c-use, where the “c” in c-use stands for computational.

How many c-uses of x can you find in the following statements?

\[
\begin{align*}
z &= x+1; \\
x[1] &= b[2]; \\
\text{foo}(x*x) \\
\text{output}(x);
\end{align*}
\]

Answer = ?

p-use

The occurrence of a variable in an expression used as a condition in a branch statement such as an if and a while, is considered as a p-use. The “p” in p-use stands for predicate.

How many p-uses of z and x can you find in the following statements?

\[
\begin{align*}
\text{if}(z>0)\{\text{output}(x)\}; \\
\text{while}(z>x)\{\ldots\};
\end{align*}
\]

Answer = ?
p-use: possible confusion

Consider the statement:

```c
if(A[x+1]>0){output (x)};
```

The use of $A$ is clearly a p-use.

Is the use of $x$ in the subscript, a c-use or a p-use? Discuss.

---

C-uses within a basic block

Consider the basic block

```c
p, y + z, x = p + 1;
```

While there are two definitions of $p$ in this block, only the second definition will propagate to the next block. The first definition of $p$ is considered local to the block while the second definition is global.

Note that $y$ and $z$ are global uses; their definitions flow into this block from some other block.
Data flow graph

A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) and uses across basic blocks in a program.

It is similar to a control flow graph of a program in that the nodes, edges, and all paths in the control flow graph are preserved in the data flow graph. An example follows.

Data flow graph: Example

Given a program, find its basic blocks, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).

Attach defs, c-use and p-use to each node in the graph.
Label each edge with the condition which when true causes the edge to be taken.

We use $d_i(x)$ to refer to the definition of variable $x$ at node $i$. Similarly, $u_i(x)$ refers to the use of variable $x$ at node $i$. 
Data flow graph: Example (contd.)

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a def-clear path for x.

Path 2-5 is def-clear for variable z defined at node 2 and used at node 5.

Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.
Def-use pairs

Definition of a variable at line $l_1$ and its use at line $l_2$ constitute a def-use pair. $l_1$ and $l_2$ can be the same.

dcu ($d_i(x)$) denotes the set of all nodes where $d_i(x)$ is live and c-used.

dpu ($d_i(x)$) denotes the set of all edges $(k, l)$ such that there is a def-clear path from node $i$ to edge $(k, l)$ and $x$ is p-used at node $k$.

We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes $i$ to node $j$ is executed.

If $u_j(x)$ is a p-use then all edges of the kind $(j, k)$ must also be taken during some executions.

Def-clear path (another example) (1)

Find def-clear paths for defs and uses of $x$ and $z$. Which definitions are live at node 4?
Def-clear path (another example) (2)

```
begin
  float x, y, z = 0.0,
  int count = 1
  input x, y, count:
  if (x < 0) {
    y = y + 1;
  } else {
    x = 1/x;
  }
  y = x * y;
  count = count + 1
  while (count > 0)
    output (z);
end
```

Infeasible! Why?

Def-use pairs: Minimal set (1)

Def-use pairs are items to be covered during testing. However, in some cases, coverage of a def-use pair implies coverage of another def-use pair. Analysis of the data flow graph can reveal a minimal set of def-use pairs whose coverage implies coverage of all def-use pairs.

Exercise: Analyze the def-use graph shown on slide 102 to determine

1. which def-uses are infeasible, and
2. a minimal set of def-uses to be covered

- corresponding to “set covering”
- in theory, this is NP-hard
- Suda/ATAC provides a good “approximate” solution
  (will be further explained when we discuss “regression testing”)
Def-use pairs: Minimal set (2)

What will be also covered if we have a test case which covers $(d_4(z), u_4(z))$?

How about $(d_4(z), u_4(z))$?

---

C-use coverage

C-use coverage:
The c-use coverage of $T$ with respect to $(P, R)$ is computed as

$$\frac{CU_c}{CU - C_U}$$

where $CU_c$ is the number of c-uses covered and $\bar{C}_U$ the number of infeasible c-uses. $T$ is considered adequate with respect to the c-use coverage criterion if its c-use coverage is 1.
C-use coverage: path traversed

Path \((\text{Start}, \ldots, q, k, \ldots, z, \ldots \text{End})\) covers the c-use at node \(z\) of \(x\) defined at node \(q\) given that \((k \ldots, z)\) is def clear with respect to \(x\)

Exercise: Find the c-use coverage when program P14.16 (refer to slide 101) is executed against the following test:

\(t_1: <x=5, y=-1, \text{count}=1>\)

p-use coverage

P-use coverage:

The p-use coverage of \(T\) with respect to \((P, R)\) is computed as

\[
P_T = \frac{PU}{PU - PU_I}
\]

where \(PU\) is the number of p-uses covered and \(PU_I\) the number of infeasible p-uses.

\(T\) is considered adequate with respect to the p-use coverage criterion if its p-use coverage is 1.
Exercise: Find the p-use coverage when program P14.16 (refer to slide 101) is executed against test \( t_2: <x=-2, y=-1, \text{count}=3> \)

All-uses coverage

The all-uses coverage of \( T \) with respect to \( (P, R) \) is computed as

\[
\frac{(CU_1 + PU_1)}{(CU + PU) - (CU_1 + PU_1)}
\]

where \( CU \) is the total c-uses, \( CU_1 \) is the number of c-uses covered, \( PU_1 \) is the number of p-uses of \( P \), \( CU_1 \) is the number of infeasible c-uses and \( PU_1 \) is the number of infeasible p-uses. \( T \) is considered adequate with respect to the all-uses coverage criterion if its c-use coverage is 1.

Exercise: Is \( T=\{t_1, t_2\} \) adequate w.r.t. to all-uses coverage for P14.16?
Infeasible p- and c-uses

Coverage of a c- or a p-use requires a path to be traversed through the program. However, if this path is infeasible, then some c- and p-uses that require this path to be traversed might also be infeasible.

Infeasible uses are often difficult to determine without some hint from a test tool.

Infeasible c-use: Example

Consider the c-use at node 4 of $z$ defined at node 5.

Show that this c-use is infeasible.
Subsumes relation

**Subsumes**: Given a test set T that is *adequate* with respect to criterion $C_1$, what can we conclude *about the adequacy* of T with respect to another criterion $C_2$?

**Effectiveness**: Given a test set T that is *adequate* with respect to criterion C, what can we expect *regarding its effectiveness in revealing errors*?
Summary

We have introduced the notion of test adequacy and enhancement.

We considered two types of adequacy criteria: one based on control flow and the other on data flow.

Control flow based: statement, decision, condition, multiple condition, ...

Data flow based: c-use, p-uses, all-uses,....

Summary (contd.)

Use of any of the criteria discussed here requires a test tool that measures coverage during testing and displays it in a user-friendly manner. Suds is one such set of tools. Several other commercial tools are available.

In general, people believe that code coverage is useful at unit-level.

Incremental assessment of code coverage and enhancement of tests can allow the application of coverage-based testing to large programs.
Summary (contd.)

Even though coverage is not guaranteed to reveal all program errors, it is the perhaps the most effective way to assess the amount of code that has been tested and what remains untested.

Tests derived using black-box approaches can almost always be enhanced using one or more of the assessment criteria discussed.