Test-Adequacy

Test-suites Assessment Using Control Flow and Data Flow

Andrea Polini

Advanced Topics on Software Engineering – Software Testing
MSc in Computer Science
University of Camerino

What is test adequacy?

It is necessary to know if the system has been tested thoroughly. The question is:

Is test suite T good enough?

Correspondingly this requires to define an adequacy criterion to make the assessment

Two different classes of criteria - to combine

- Black-box: based on models and requirements
- White-box: based on code

Example

Consider a program P developed to satisfy a set of requirements (P,R) (simplified version)

- **R1**: Input two integers, x, y, from the standard input device
- **R2**: Find and print to the standard output the sum if $x < y$
- **R3**: Find and print to the standard output the product of the two numbers if $x \geq y$

- **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

What is test adequacy?

It is necessary to know if the system has been tested thoroughly. The question is:

Is test suite T good enough?

Correspondingly this requires to define an adequacy criterion to make the assessment

Two different classes of criteria - to combine

- Black-box: based on models and requirements
- White-box: based on code

Example

Consider a program P developed to satisfy a set of requirements (P,R) (simplified version)

- **R1**: Input two integers, x, y, from the standard input device
- **R2**: Find and print to the standard output the sum if $x < y$
- **R3**: Find and print to the standard output the product of the two numbers if $x \geq y$

- **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

What is test adequacy?

It is necessary to know if the system has been tested thoroughly. The question is:

Is test suite T good enough?

Correspondingly this requires to define an adequacy criterion to make the assessment

Two different classes of criteria - to combine

- Black-box: based on models and requirements
- White-box: based on code

Example

Consider a program P developed to satisfy a set of requirements (P,R) (simplified version)

- **R1**: Input two integers, x, y, from the standard input device
- **R2**: Find and print to the standard output the sum if $x < y$
- **R3**: Find and print to the standard output the product of the two numbers if $x \geq y$

- **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

Adequacy criteria push the improvements of test sets

Adequancy criteria are mainly meant as indicators to consider to improve a test suite.

1. Measure adequacy of T with respect to C. If T is adequante go to 3
2. For each uncovered element $e \in C_e$ do the following until e is covered or is determined to be infeasible
    1. construct a test t that covers e or will likely cover e
    2. execute P against t till no fault is identified
    3. if e is covered then t is added to T otherwise the tester can still decide to add it or to ignore it
3. The procedure ends

Example

A program computing x^y:

```
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");
}
```

Criteria

C1: A test set is considered adequate if it tests the program for at least one zero and one nonzero value of each of the two inputs x and y

C2: A test set 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.

Infeasibility

It is clearly possible that some criteria could be infeasible given P structure since some of the path are not possible
Checking infeasibility is not an easy task that in general cannot be decided. Is the tester that should decide if a path is feasible or not

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

Single or multiple executions

Warning

When a software keeps state among different runs it could be necessary to bring the system in a given state before being able to observe a failure

Criteria based on control flow

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 set of statements covered, S_i the set of unreachable statements, and S_e the set of statements in the program, that is 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 set of blocks covered, B_i the set of unreachable blocks, and B_e the blocks in the program, that is the block coverage domain. T is considered adequate with respect to the block coverage criterion if the block coverage of T with respect to (P,R) is 1.

Conditions and decisions

- Conditions can be classified as simple or compound
- Conditions are generally used to define decision points
- A decision is covered if the flow has been diverted to all possible destinations

Decision Coverage

The decision coverage of T with respect to (P,R) is computed as $|D_c|/(|D_e| - |D_i|)$ where D_c is the set of decisions covered, D_i the set of unfeasible decision, and D_e the set of decision in the program, that is the decision coverage domain. T is considered adequate with respect to the decision coverage criterion if the decision coverage of T with respect to (P,R) is 1.
To be considered are peculiarities related to the `switch` statements

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 set of simple conditions covered, D_i the set of unfeasible simple conditions, and C_e is the set of simple conditions in the program, that is the condition coverage domain. T is considered adequate with respect to the decision coverage criterion if the decision coverage of T with respect to (P,R) is 1.

Condition vs. decision coverage

Condition coverage does not guarantee decision coverage and viceversa

Condition/decision coverage

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 variable as defined as before. T is considered adequate with respect to the condition/decision coverage criterion if the condition/decision coverage of T with respect to (P,R) is 1.

Example

Consider a program that takes in input two integers `x` and `y`, and returns an integer `z` according to the following table:

x<0	**y<0**	**output(z)**
`true`	`true`	`foo1(x,y)`
`true`	`false`	`foo2(x,y)`
`false`	`true`	`foo2(x,y)`
`false`	`false`	`foo1(x,y)`

Apply the test suite $T = \{t_1 :< x = -3, y = -2 >, t_2 :< x = -4, y = 2 >\}$ to the program below

```
begin
  int x,y,z;
  input(x,y);
  if (x<0 and y<0)
    z=foo1(x,y);
  else
    z=foo2(x,y);
  output(z);
end
```

Multiple Condition Coverage

This criterion aims at assessing the software with all possible combinations of simple conditions constituting a compound condition

Multiple condition coverage

The multiple condition coverage of T with respect to (P,R) is computed as $|C_c|/(|C_e| - |C_i|)$ where $|C_c|$ denotes the set of combinations covered, $|C_i|$ denotes the set 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 multiple-condition coverage of T with respect to (P,R) is 1.

Let's consider a code composed of n decisions each one including K_i with $i \in [1 \cdots n]$ simple conditions. In case all of them are feasible which is the total number of possible combinations?

Example

Consider a program that takes in input three integers `A`, `B` and `C`, and returns a value `S` according to the following table:

A<B	**A>C**	**S**
`true`	`true`	`f1(A,B,C)`
`true`	`false`	`f2(A,B,C)`
`false`	`true`	`f3(A,B,C)`
`false`	`false`	`f4(A,B,C)`

Apply the test suite $T = \{t_1 :< A = 2, B = 3, C = 1 >, t_2 :< A = 2, B = 1, C = 3 >\}$ to the program below

```
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
```

Modified Condition/Decision Coverage – MC/DC

- Combinations necessary to satisfy the Multiple Condition Coverage is generally too big.
- MC/DC allows a coverage of all decisions and all conditions avoiding the exponential explosion
- To derive the test set the idea is to identify those tuples which can cover the two criteria without requiring a complete combinations of values.

Let's consider the compound condition $(C_1 \wedge C_2) \vee C_3$

Modified Condition/Decision Coverage – MC/DC

- Combinations necessary to satisfy the Multiple Condition Coverage is generally too big.
- MC/DC allows a coverage of all decisions and all conditions avoiding the exponential explosion
- To derive the test set the idea is to identify those tuples which can cover the two criteria without requiring a complete combinations of values.

Let's consider the compound condition $(C_1 \wedge C_2) \vee C_3$

Modified Condition/Decision Coverage – MC/DC

- Combinations necessary to satisfy the Multiple Condition Coverage is generally too big.
- MC/DC allows a coverage of all decisions and all conditions avoiding the exponential explosion
- To derive the test set the idea is to identify those tuples which can cover the two criteria without requiring a complete combinations of values.

Let's consider the compound condition $(C_1 \wedge C_2) \vee C_3$

Modified Condition/Decision Coverage – MC/DC

- Combinations necessary to satisfy the Multiple Condition Coverage is generally too big.
- MC/DC allows a coverage of all decisions and all conditions avoiding the exponential explosion
- To derive the test set the idea is to identify those tuples which can cover the two criteria without requiring a complete combinations of values.

Let's consider the compound condition $(C_1 \wedge C_2) \vee C_3$

Definition of MC/DC coverage

The MC/DC criterion requires that:

- Each block in P has been covered
- Each simple condition in P has taken both `true` and `false` value
- Each decision in P has taken all possible outcomes
- Each simple condition within a compound condition C in P has been shown to independently affect the outcome of C (limited to the simple condition when it occurs more than once).

Measure

Measure the 4 different factors separately and for MC:

- $MC_c = \frac{\Sigma_{i=1}^{N} e_i}{\Sigma_{i=1}^{N}(n_i - f_i)}$
  where n_i number of simple conditions, e_i single conditions for which independent effects have been shown, f_i number of infeasible conditions.

MC/DC comparisons

MC/DC vs. Condition

- Missing conditions
- Incorrect boolean operator
- Mixed type

MC/DC vs. Multiple time comparison

n	**Multiple Condition**	**MC/DC**	**Multiple Condition**	**MC/DC**
1	2	2	2ms	2ms
4	16	5	16ms	5ms
8	256	9	256ms	9ms
16	65536	17	65.6s	17ms
32	4294967296	33	49.5 days	33ms

MC/DC and Lazy evaluation

MC/DC comparisons

MC/DC vs. Condition

- Missing conditions
- Incorrect boolean operator
- Mixed type

MC/DC vs. Multiple time comparison

n	**Multiple Condition**	**MC/DC**	**Multiple Condition**	**MC/DC**
1	2	2	2ms	2ms
4	16	5	16ms	5ms
8	256	9	256ms	9ms
16	65536	17	65.6s	17ms
32	4294967296	33	49.5 days	33ms

MC/DC and Lazy evaluation

Example

Consider a program conceived to satisfy the following requirements:

R_1: Given coordinate position `x`, `y`, and `z`, and a direction value *d*, the program must invoke one of the three functions `fire-1`, `fire-2`, and `fire-3` as per conditions below:

- $R_{1,1}$: Invoke `fire-1` when (`x<y`) `and` (`z*z>y`) `and` (prev=”East”) where *prev* and *current* denote, respectively, the previous and current values of *d*.
- $R_{1,2}$: Invoke `fire-2` when (`x<y`) `and` (`z*z≤y`) `or` (current=”South”)
- $R_{1,3}$: Invoke `fire-3` when none of the two conditions above is `true`

R_2: The invocation described above must continue until an input Boolean variable becomes `true`

- ▶ let’s generate test satisfying the conditions and let’s analyze the covered decision on a possible implementation of the system

Code

```
begin
float x,y,z; direction d; string prev,current; bool done;
input(done); current ='North';
while(!done) {
  input(d); prev=current;current=f(d); input(x,y,z);
  if ((x<y) and (z*z>y) and (prev=='East'))
    fire-1(x,y);
  else if ((x<y) and (z*z <= y) or (current == 'South'))
      fire-2(x,y);
    else
      fire-3(x,y); input(done);
}
output('Firing completed');
end
```

- generate tests to meet the requirements (4 tests generated)

Test	**Req.**	`done`	`d`	`x`	`y`	`z`
t_1	$R_{1,2}$	false	East	10	15	3
t_2	$R_{1,1}$	false	South	10	15	4
t_3	$R_{1,3}$	false	North	10	15	5
t_4	R_2	true				

- Which kind of coverage criteria are satisfied by the test set?
- Cover $x < y$ to get condition coverage?
- What about Multiple Condition Coverage?
- What about MC/DC?

- generate tests to meet the requirements (4 tests generated)

Test	**Req.**	`done`	`d`	`x`	`y`	`z`
t_1	$R_{1,2}$	false	East	10	15	3
t_2	$R_{1,1}$	false	South	10	15	4
t_3	$R_{1,3}$	false	North	10	15	5
t_4	R_2	true				

- Which kind of coverage criteria are satisfied by the test set?
- Cover $x < y$ to get condition coverage?
- What about Multiple Condition Coverage?
- What about MC/DC?

- generate tests to meet the requirements (4 tests generated)

Test	**Req.**	`done`	`d`	`x`	`y`	`z`
t_1	$R_{1,2}$	false	East	10	15	3
t_2	$R_{1,1}$	false	South	10	15	4
t_3	$R_{1,3}$	false	North	10	15	5
t_4	R_2	true				

- Which kind of coverage criteria are satisfied by the test set?
- Cover $x < y$ to get condition coverage?
- What about Multiple Condition Coverage?
- What about MC/DC?

- generate tests to meet the requirements (4 tests generated)

Test	**Req.**	`done`	`d`	`x`	`y`	`z`
t_1	$R_{1,2}$	false	East	10	15	3
t_2	$R_{1,1}$	false	South	10	15	4
t_3	$R_{1,3}$	false	North	10	15	5
t_4	R_2	true				

- Which kind of coverage criteria are satisfied by the test set?
- Cover $x < y$ to get condition coverage?
- What about Multiple Condition Coverage?
- What about MC/DC?

Tracing test cases to requirements

Enhancing a test set we should understand *what portions of the requirements are tested when the program under test is executed against the newly added test case?*

- Trace back test to requirements is useful when they need to be modified

Data Flow concepts

- Criteria considered so far are based on the control flow
- it is possible to conceive adequacy criteria based on data flow characteristics

Consider the following program:

```
begin
  int x,y; float z;
  input(x,y);
  z=0;
  if (x!=0) z=z+y;
    else z=z-y;
  if (y!=0) z=z/x // Should be (y!=0 and x!=0)
    else z=z*x;
  output(z);
end
```

An MC/DC test set could not reveal the error while a test set based on definition and usage of variables would have been able

Data Flow concepts

- Criteria considered so far are based on the control flow
- it is possible to conceive adequacy criteria based on data flow characteristics

Consider the following program:

```
begin
  int x,y; float z;
  input(x,y);
  z=0;
  if (x!=0) z=z+y;
    else z=z-y;
  if (y!=0) z=z/x // Should be (y!=0 and x!=0)
    else z=z*x;
  output(z);
end
```

An MC/DC test set could not reveal the error while a test set based on definition and usage of variables would have been able

Data flow criteria

- Data flow criteria based on two main concepts:
  - Definitions – points in which a variable is defined (e.g. assignements, input statements)
  - Uses – points in which a variable is accessed
    - computational usage - c-use
    - predicate usage - p-use
- Which are the effect of parameter passing (by value or by reference)?

```
input(x,y); z=0;
z = x+1
A[x-1]=B[2];
foo(x*x);
output(z);
if (z>0) output(x);
if (A[x+1]>0) output(x);
```

Global, Local and Pointers

Variables can be defined in a block, used and redefined (killed) within the same block. Effects can also be available outside the block:

- `p = y+z; x = p+1; p = z*z;`

Definition and use of variables can be referred to:

- local
- global

Pointers

The use of pointers makes the data flow analysis rather complex:

```
z=&x;
y=z+1;
*z=25;
y=*z+1;
```

Data Flow Graph

A data-flow graph of a program (aka def-use graph) captures the flow of definitions across the basic blocks constituting the program. The graph can be constructed in the following way:

1. Construct def_i, $c - use_i$, $p - use_i$ for each basic block i in P
2. Associate each node i in N with def_i, $c - use_i$, $p - use_i$
3. For each node i that has a non empty $p - use$ set and ends in condition C, associate edges (i, j) and (i, k) with C and $!C$, respectively.

Data flow graph

Build the DFG for the following piece of code:

```
begin
  int x,y,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
```

Example

Let's build a def-use graph for the following program:

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

Which are the blocks?

Example

Let's build a def-use graph for the following program:

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

Which are the blocks?

Definitions

def-clear paths

A def-clear path for a variable x is a path from a definition of the variable to a usage without further definitions in the intermediate nodes of the path

live definition

A definition at node *i* is live at node *j* if there is no intermediate definition in a path from *i* to *j*

Def-use pairs

A def-use pair for a variable 'X' refers to a definition *d* and a usage *u* on a def-clear path

For each variable definition $d_i(x)$ there is:

- a **dcu**($d_i(x)$) set, that is constituted by all nodes *j* in any def-clear path from node *i* such that $u_j(x)$ in relation to a c-use
- a **dpu**($d_i(x)$) set, that is constituted by all sets of edges leaving a node *j* for which there is a def-clear path from *i* and $u_i(x)$ in relation to a p-use

Let's fill the table for the previous DFG

Variable(v)	Defined at node(n)	dcu(v,n)	dpu(v,n)

Def-use pairs

A def-use pair for a variable 'X' refers to a definition *d* and a usage *u* on a def-clear path

For each variable definition $d_i(x)$ there is:

- a **dcu**($d_i(x)$) set, that is constituted by all nodes *j* in any def-clear path from node *i* such that $u_j(x)$ in relation to a c-use
- a **dpu**($d_i(x)$) set, that is constituted by all sets of edges leaving a node *j* for which there is a def-clear path from *i* and $u_i(x)$ in relation to a p-use

Let's fill the table for the previous DFG

Variable(v)	**Defined at node**(n)	**dcu**(v,n)	**dpu**(v,n)

Def-use chains

A def-use chain (aka k-dr interaction) is constituted by path including a sequence of alternating def-use pairs. It is also possible to consider different variables E.g. consider the def-use chain for y and z and the sequence of nodes in which they are considered

Def-use optimization

Some def-use can subsume other relations. E.g. z defined in node 1 subsumes y defined in node 1

Let's fill the table for the previous DFG

Variable(v)	**Defined at node**(n)	**dcu**(v,n)	**dpu**(v,n)

Adequacy criteria for data-flow

Given the total number of c-uses (CU) and p-uses (PU) for all variable definitions we can define different coverage criteria for data-flow.

$$CU = \Sigma_{i=1}^{n} \Sigma_{j=1}^{d_i} |\mathbf{dcu}(v_i, n_j)|$$
$$PU = \Sigma_{i=1}^{n} \Sigma_{j=1}^{d_i} |\mathbf{dpu}(v_i, n_j)|$$

where $v = \{v_1, v_2, \ldots, v_n\}$ is the set of variables in a program and $n = \{n_1, n_2, \ldots, n_k\}$ is the set of blocks in the same program

Coverage

C-use coverage

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

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

where CU_c is the number of c-uses covered and CU_f 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.

P-use coverage

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

$$\frac{PU_c}{PU - PU_f}$$

where PU_c is the number of p-uses covered and PU_f 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.

Coverage

C-use coverage

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

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

where CU_c is the number of c-uses covered and CU_f 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.

P-use coverage

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

$$\frac{PU_c}{PU - PU_f}$$

where PU_c is the number of p-uses covered and PU_f 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.

Coverage's

All-uses coverage

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

$$\frac{CU_c + PU_c}{(CU + PU) - (CU_f + PU_f)}$$

where CU_c and PU_c are the number of c-uses and p-uses covered respectively. CU_f and PU_f are the number of infeasible c-uses and p-uses respectively. T is considered adequate with respect to the all-uses coverage criterion if its all-uses coverage is 1.

k-dr chain coverage

For a given $K \geq 2$ the kdr(k) coverage of T with respect to (P,R) is computed as:

$$\frac{C_c^k}{C^k - C_f^k}$$

where C_c^k is the number of k-dr interactions covered, C^k is the number of elements in K-dr(k), and C_f^k the number of infeasible interactions in k.dr(k). T is considered adequate with respect to the kdr(k)coverage criterion if its k-dr(k) coverage is 1.

Coverage's

All-uses coverage

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

$$\frac{CU_c + PU_c}{(CU + PU) - (CU_f + PU_f)}$$

where CU_c and PU_c are the number of c-uses and p-uses covered respectively. CU_f and PU_f are the number of infeasible c-uses and p-uses respectively. T is considered adequate with respect to the all-uses coverage criterion if its all-uses coverage is 1.

k-dr chain coverage

For a given $K \geq 2$ the kdr(k) coverage of T with respect to (P,R) is computed as:

$$\frac{C_c^k}{C^k - C_f^k}$$

where C_c^k is the number of k-dr interactions covered, C^k is the number of elements in K-dr(k), and C_f^k the number of infeasible interactions in k.dr(k). T is considered adequate with respect to the kdr(k)coverage criterion if its k-dr(k) coverage is 1.

Control flow vs. Data Flow

The subsumes relation

A coverage criterion C1 subsumes a coverage criterion C2 iff whenever the satisfaction of C1 implies the satisfaction of C2

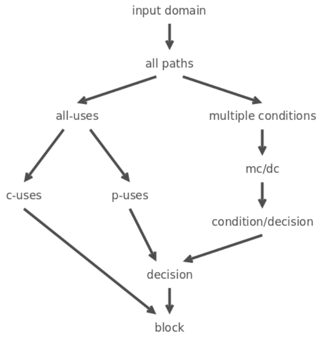

Figure: The subsumes relationship among the studied coverage criterion

Mutation analysis - Ch. 8

Sketch of the idea

Mutation is a powerful strategy to assess the quality of test suites. The approach is based on the generation of program mutants and on the score got by a test suite in “killing” them.

Regression testing - Ch. 9

Sketch of the idea

Definition of strategies to select subset of test cases in a test suite in order to test a system that has undergone a modification in order to reduce the costs of testing obviously getting enough confidence on the quality of the software.