White Box Testing

Readings: Ron Patton Ch 7, Daniel Galin Ch 9

White Box testing

- Structural testing approaches
- Focus on system internal logic

```
...
while(z <w){
 if (y == 2){
        ...
  }
}
```

- need source code
- specification needed for expected results
- Advantage
  - test what is actually there
- Dis-advantage
  - non implemented functionality in specification may be missed

White Box testing

- Control flow oriented approaches
  - Based on the analysis of how control flows through a program
  - Examples of test coverage criteria
    - statement coverage (all-nodes)
    - branch coverage (all-edges)
    - path coverage (all-paths)
    - condition coverage
    - condition/branch coverage

White Box testing

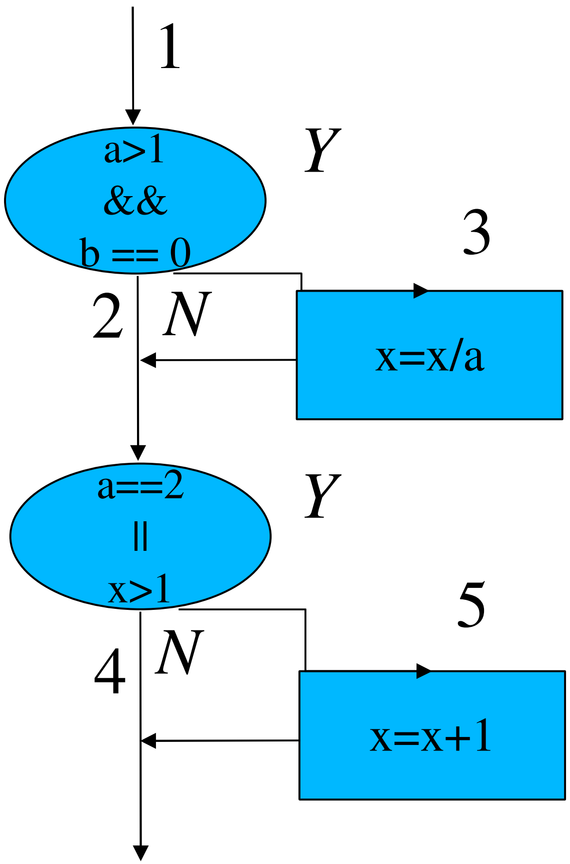
- Data flow oriented approaches
  - Based on the analysis of how data flows through a program
  - Examples of test coverage criteria
    - all-defs
    - all-c-uses
    - all-p-uses
    - all-uses
    - all-du-paths

Control Flow Coverage (1)

- Statement coverage
- Branch coverage
  - include statement coverage
  - minimum coverage - IEEE unit test standard
- Condition coverage
  - conditions in branches that evaluated to *true*, *false*
- Branch/Condition Coverage
  - combines branch and condition coverage
- Multiple Condition coverage
  - combination of condition in decisions
- Path coverage
  - 100% path coverage impossible in practice (loops)

Control Flow Coverage - example

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```

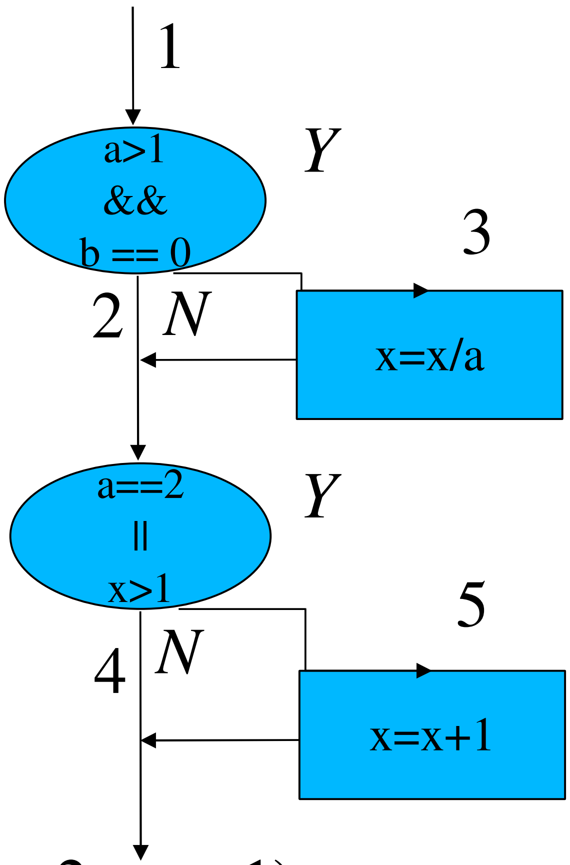


Statement coverage:

- test case for 1 – 3 – 5
- example: a = 2, b = 0, x = 3

Control Flow Coverage - example

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```

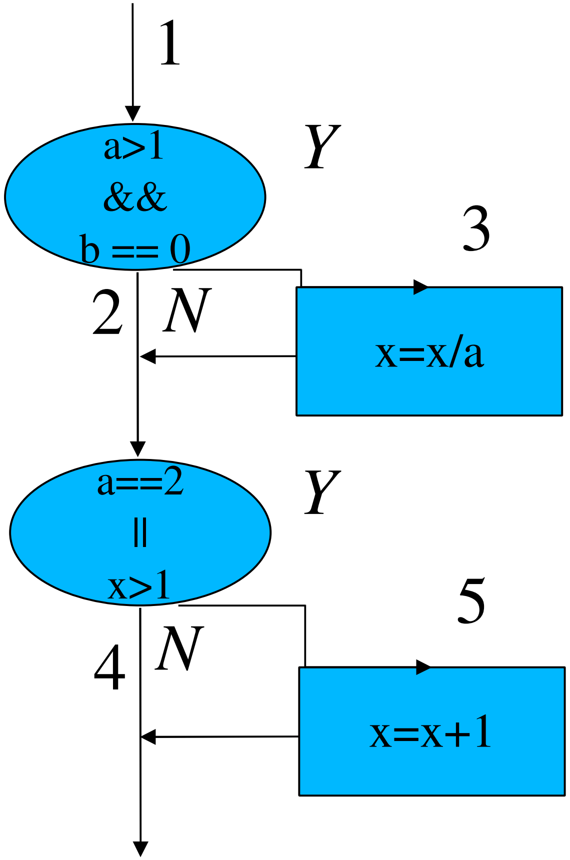


Branch coverage:

- test case for 1 – 2 – 5 (a = 2, b = 2, x = -1)
- test case for 1 – 3 – 4 (a = 3, b = 0, x = 1)

Control Flow Coverage - example

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```

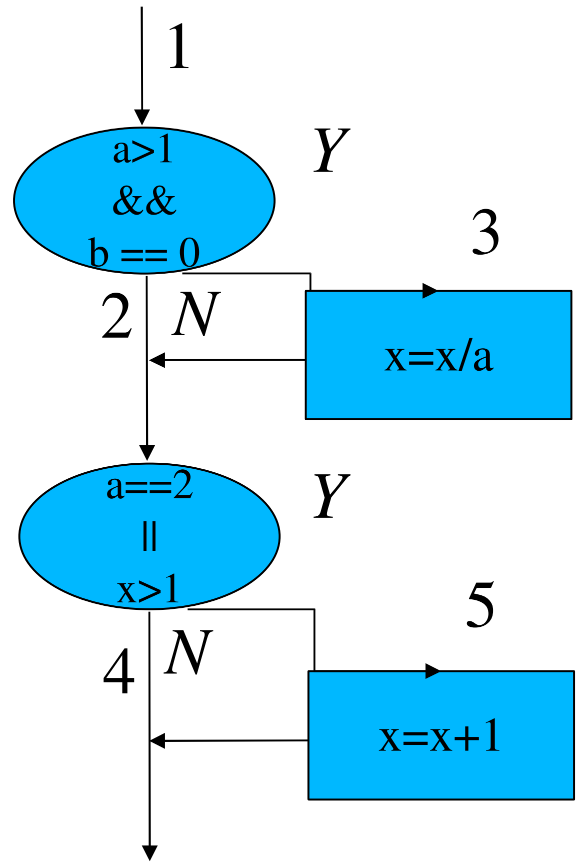


Condition coverage:

- conditions: a > 1, b == 0, a == 2, x > 1
- test case for a > 1, b == 0, a == 2, x > 1: a = 2, b = 0, x = 4
- test case for a <= 1, b <> 0, a <> 2, x <= 1: a = 1, b = 1, x = 1

Control Flow Coverage - example

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```

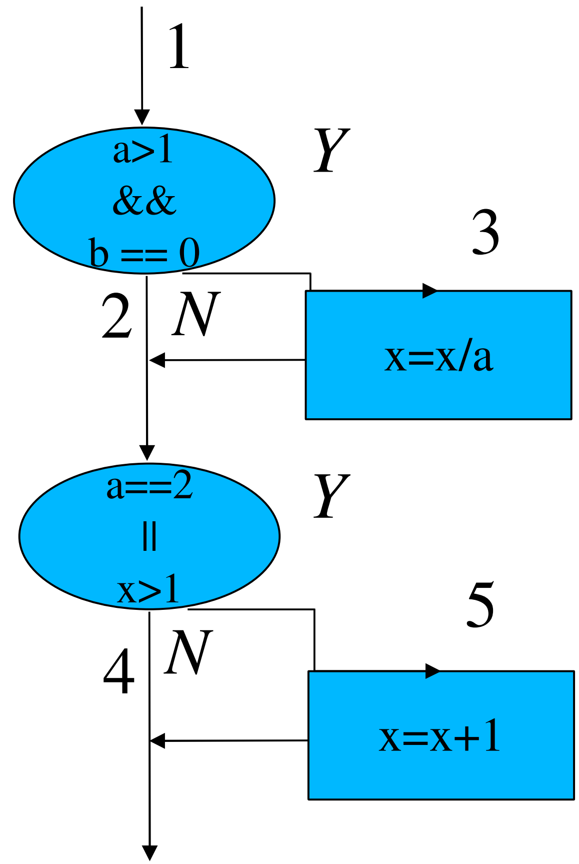


Multiple Condition coverage:

- combinations:
  - (a > 1, b == 0) (a > 1, b <> 0) (a <= 1, b == 0) (a <= 1, b <> 0)
  - (a == 2, x > 1) (a == 2, x <= 1) (a <> 2, x > 1) (a <> 2, x <= 1)

Control Flow Coverage - example

```
int proc(int a, int b, int x) {
  if ((a>1) && (b==0))
    x = x/a;
  if ((a==2)||(x>1))
    x = x+1;
  return x;
}
```



Path coverage:

- Paths
  - 1 – 2 – 4, 1 – 2 – 5, 1 – 3 – 4, 1 – 3 – 5

White box test process

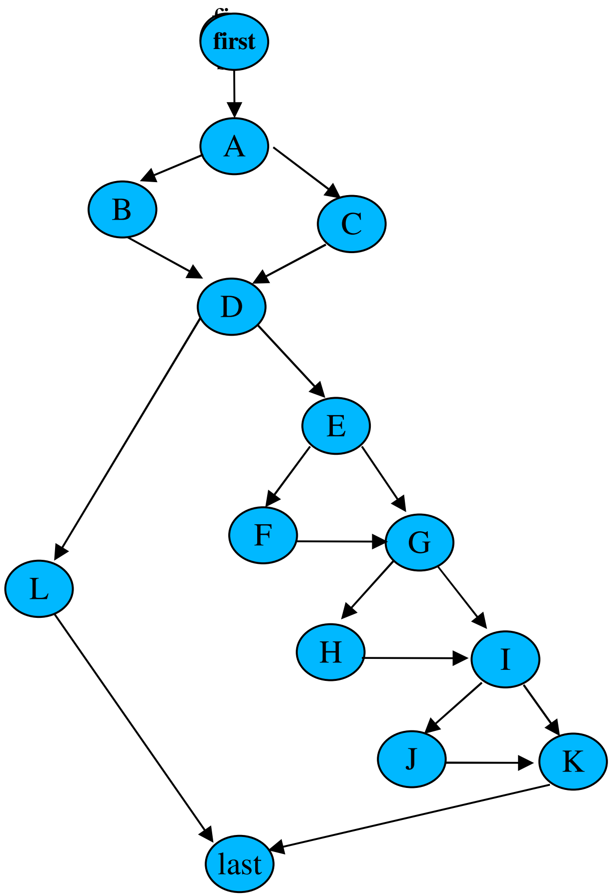
1. Set coverage goal
2. Derive flowchart from source code
3. Determine paths to obtain coverage goal
4. For each path
   - Sensitize path for input values
   - Use specification for expected output
   - Watch for unfeasible paths
5. Run test cases
6. Check coverage

Simplified flowchart

- More convenient for large units
- Obtained by collapsing decision-to decision path
- Nodes
  - Decision nodes: with more than one *outlink*
    - end with decision statement
  - Junction nodes: with more than one *inlink*
    - merge several paths

Simplified flowchart - example

```
public static void main (String[] args)
      throws IOException
{
 boolean isATriangle;
 System.out.print("Enter the sides: ");
 int a = IOEasy.readInt();
 int b = IOEasy.readInt();
 int c = IOEasy.readInt();
 System.out.println("A is " + a + ", B is " + b + ", C is " + c);
 if ((a < b+c) && (b < a+c) && (c < a+b)) (A)
   isATriangle = true; (B)
 else isATriangle = false; (C)
 if (isATriangle) { (D)
   if (((a==b) || (a==c) || (b==c)) &&!((a==b) && (a==c))) (E)
      System.out.println( "Triangle is Isosceles");  (F)
   if ((a==b) && (b==c)) (G)
      System.out.println( "Triangle is Equilateral"); (H)
   if ((a!=b) && (a!=c) && (b!=c)) (I)
      System.out.println("Triangle is Scalene"); (J)
 }
 else System.out.println("Not a Triangle");   (L)
}
}
```

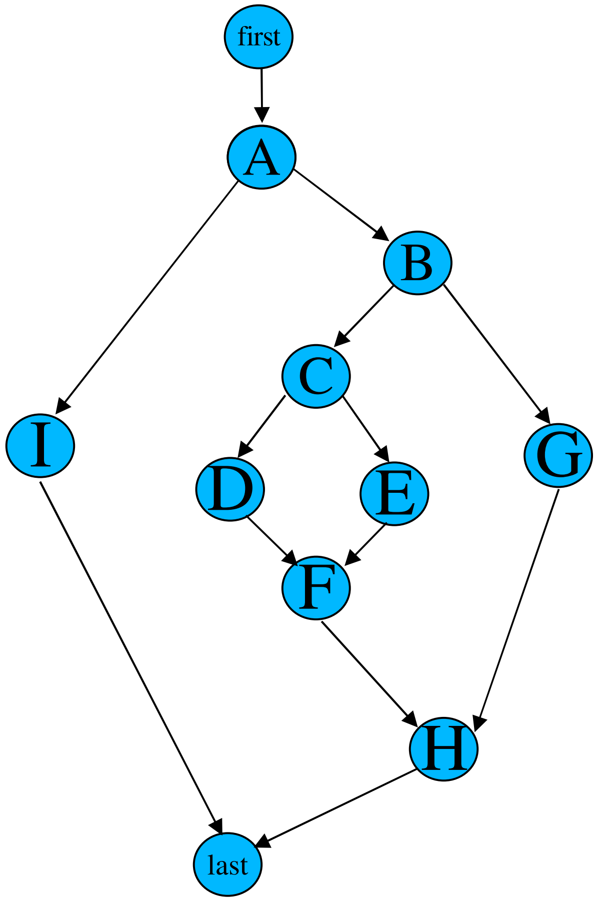


Path Sensitizing

- To find a set of input to force a selected path
- Backward strategy
  - from exit to entry
- Forward strategy
  - from entry to exit
- Problem with unfeasible paths
  - may call for re-writing of program

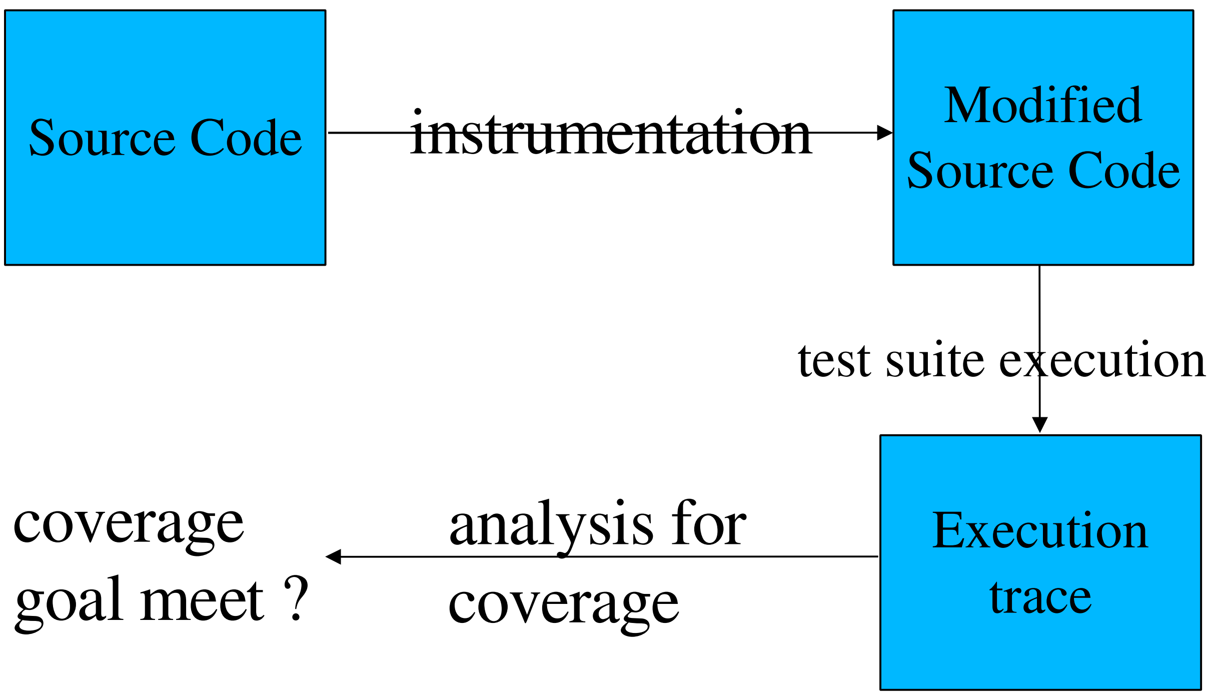
Better Triangle program

```
public static void main (String[] args)   throws IOException{
  System.out.print( "Enter the sides: ");
  int a = IOEasy.readInt();
  int b = IOEasy.readInt();
  int c = IOEasy.readInt();
  System.out.println("A is " + a + ", B is " + b +  ", C is " + c);
  if ((a < b+c) && (b < a+c) && (c < a+b)) { (A)
    if ((a!=b) && (a!=c) && (b!=c)) { (B)
       System.out.println("Triangle is Scalene"); (G)
    } else {
      if ((a==b) && (a==c)) (C)
              System.out.println("Triangle is Equilateral"); (E)
      else
              System.out.println("Triangle is Isosceles");   (D)
    } (F)
  }  else
    System.out.println("Not a Triangle"); (I)
  } (H)
}
```



Path Instrumentation

•To measure code coverage



Approaches

- link markers
- link counters
- symbolic debugger
- code coverage tool

Path Condition

- Conjunction of branch predicates required to hold for all the branches along a path
- Used to find
  - values for a path
  - unfeasible paths
- Determined using *symbolic evaluation*
  - variables take symbolic values (e.g.: $X_0, X_1, \ldots X_n$)

Path Condition – Symbolic values

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 1,3

(1) $X_1 = X_0^2$

$Y_1 = Y_0^2$

(3) $X_3 = X_1 + 1 = X_0^2 + 1$

$Y_3 = Y_1 + 1 = Y_0^2 + 1$

Path Condition – Symbolic values

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 1,4

(1) $X_1 = X_0^2$

$Y_1 = Y_0^2$

(4) $X_4 = X_1 - 1 = X_0^2 - 1$

$Y_4 = Y_1 - 1 = Y_0^2 - 1$

Path Condition – Symbolic values

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 2, 3

(2) $X_2 = X_0 + 1$

$Y_2 = Y_0 + 1$

(3) $X_3 = X_2 + 1 = X_0 + 2$

$Y_3 = Y_2 + 1 = Y_0 + 2$

Path Condition – Symbolic values

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 2, 4

(2) $X_2 = X_0 + 1$

$Y_2 = Y_0 + 1$

(4) $X_4 = X_2 - 1 = X_0$

$Y_4 = Y_2 - 1 = Y_0$

Path Condition determination (1)

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 1,3

(1) $X_1 = X_0^2$

$Y_1 = Y_0^2$

(3) $X_3 = X_1 + 1 = X_0^2 + 1$

$Y_3 = Y_1 + 1 = Y_0^2 + 1$

Path Condition

$$((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((X_1 < 1) \vee (Y_1 < 1))$$

$$\equiv ((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((X_0^2 < 1) \vee (Y_0^2 < 1))$$

$$\equiv ((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((-1 < X_0 < 1) \vee (-1 < Y_0 < 1))$$

Path Condition determination (2)

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 1,4

(1) $X_1 = X_0^2$

$Y_1 = Y_0^2$

(4) $X_4 = X_1 - 1 = X_0^2 - 1$

$Y_4 = Y_1 - 1 = Y_0^2 - 1$

Path Condition

$$((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((X_1 \ge 1) \wedge (Y_1 \ge 1))$$

$$\equiv ((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((X_0^2 \ge 1) \wedge (Y_0^2 \ge 1))$$

$$\equiv ((X_0 \le 0) \vee (Y_0 \le 0)) \wedge ((X_0 \le -1) \vee (X_0 \ge 1)) \wedge ((Y_0 \le -1) \vee (Y_0 \ge 1))$$

Path Condition determination (3)

```
if (X<=0) or (Y<=0) then
(1)  X := X**2
     Y := Y**2
  else
(2)  X := X+1
     Y := Y+1
  end_if_else
  if (X<1) or (Y<1) then
(3)  X := X+1
     Y := Y+1
  else
(4)  X := X-1
     Y := Y-1
  end_if_else
```

Path 2, 3

(2) $X_2 = X_0 + 1$

$Y_2 = Y_0 + 1$

(3) $X_3 = X_2 + 1 = X_0 + 2$

$Y_3 = Y_2 + 1 = Y_0 + 2$

Path Condition

$$((X_0 > 0) \wedge (Y_0 > 0)) \wedge ((X_2 < 1) \vee (Y_2 < 1))$$

$$\equiv ((X_0 > 0) \wedge (Y_0 > 0)) \wedge ((X_0 + 1 < 1) \vee (Y_0 + 1 < 1))$$

$$\equiv ((X_0 > 0) \wedge (Y_0 > 0)) \wedge ((X_0 < 0) \vee (Y_0 < 0))$$

Path conditions in presence of loops

```
(1) C := 0
    while (X>Y) do
(2)   X := X-Y
      C := C+1
    end_while
```

Symbolic values

(1) $X_1 = X_0, C_1 = 0$

(2,1) $X_{2,1} = X_1 - Y_1 = X_0 - Y_0$

$C_{2,1} = C_1 + 1 = 1$

(2,2) $X_{2,2} = X_{2,1} - Y_{2,1} = (X_0 - Y_0) - Y_0 = X_0 - 2 * Y_0$

$C_{2,2} = C_{2,1} + 1 = 2$

(2,3) $X_{2,3} = X_{2,2} - Y_{2,2} = (X_0 - 2 * Y_0) - Y_0 = X_0 - 3 * Y_0$

$C_{2,3} = C_{2,2} + 1 = 3$

(2,N) $X_{2,N} = X_0 - N * Y_0$

$C_{2,N} = N$

Path conditions in presence of loops

```
(1) C := 0
    while (X>Y) do
(2)   X := X-Y
      C := C+1
    end_while
```

(1) $X_1 = X_0, C_1 = 0$

(2,1) $X_{2,1} = X_1 - Y_1 = X_0 - Y_0$
$C_{2,1} = C_1 + 1 = 1$

(2,2) $X_{2,2} = X_{2,1} - Y_{2,1} = (X_0 - Y_0) - Y_0 = X_0 - 2 * Y_0$
$C_{2,2} = C_{2,1} + 1 = 2$

(2,3) $X_{2,3} = X_{2,2} - Y_{2,2} = (X_0 - 2 * Y_0) - Y_0 = X_0 - 3 * Y_0$
$C_{2,3} = C_{2,2} + 1 = 3$

(2,N) $X_{2,N} = X_0 - N * Y_0$
$C_{2,N} = N$

Path Condition for body executed 0 times $(X_1 \leq Y_1) \equiv (X_0 \leq Y_0)$

Path Condition for body executed 2 times

$$(X_1 > Y_1) \wedge (X_{2,1} \leq Y_{2,1})$$
$$\equiv (X_0 > Y_0) \wedge (X_0 - Y_0 \leq Y_0)$$
$$\equiv (X_0 > Y_0) \wedge (X_0 \leq 2 * Y_0)$$
$$\equiv (Y_0 < X_0 \leq 2 * Y_0)$$

Path conditions in presence of loops

```
(1) C := 0
    while (X>Y) do
(2)   X := X-Y
      C := C+1
    end_while
```

(1) $X_1 = X_0, C_1 = 0$

(2,1) $X_{2,1} = X_1 - Y_1 = X_0 - Y_0$

$C_{2,1} = C_1 + 1 = 1$

(2,2) $X_{2,2} = X_{2,1} - Y_{2,1} = (X_0 - Y_0) - Y_0 = X_0 - 2 * Y_0$

$C_{2,2} = C_{2,1} + 1 = 2$

(2,3) $X_{2,3} = X_{2,2} - Y_{2,2} = (X_0 - 2 * Y_0) - Y_0 = X_0 - 3 * Y_0$

$C_{2,3} = C_{2,2} + 1 = 3$

(2,N) $X_{2,N} = X_0 - N * Y_0$

$C_{2,N} = N$

Path Condition for body executed N times

$$(X_1 > Y_1) \wedge (X_{2,1} > Y_{2,1}) \wedge \ldots \wedge (X_{2,N-1} > Y_{2,N-1}) \wedge (X_{2,N} \leq Y_{2,N})$$

$$\equiv (N-1) * Y_0 < X_0 \leq N * Y_0$$

Loop Coverage (1)

- Single loop - execute loop
  - minimum-1
  - minimum
  - minimum+1
  - typical
  - maximum
  - maximum+1
- time when possible.

```
examp1(char[] val) {
  for (int i = 0; i < val.length; i++ {
     ...
  }
}
```

```
examp2(int x, int y) {
  i = x;
  while (i <= y) {
     ...
     i++;
  }
}
```

Loop Coverage (2)

- Nested loop
  - Start at innermost loop.
    - Set all the outer loops to their minimum values.
    - Set all other loops to *typical*
    - Test the *minimum*, *minimum+1*, *typical*, *maximum-1*, *maximum* for the innermost loop
  - Move up in loop nest
  - If outermost loop done do five cases for all loops in the nest simultaneously.

```
for (i = 0; i < x; i++) {
  for (j = 0; j < y; j++) {
     for (k = 0; k < z; k++) {
     }
  }
}
```

Data Flow testing

- Testing based on *data flow*
  - where data is defined
  - where data is modified
  - sequence of data related events

Data Flow testing - definitions

- Variable definition ***d(v,n)***
  - value is assigned to *v* at node *n* (e.g. LHS of assignment, input statement, parameter)
- Variable use
  - variable reference
  - ***c-use(v,n)*** – variable ***v*** used in a computation at node ***n*** (e.g. RHS of assignment, argument of procedure call, output statement)
  - ***p-use(v,m,n)*** – variable ***v*** used in predicate from node ***m*** to ***n*** (e.g as part of an expression used for a decision)
- Variable kill *k(v,n)*
  - variable de-allocated

Data Flow testing - definitions

```
   int main (void)
  {
1   char *line;
2   int x = 0, y;
3   line = malloc(256 * sizeof (*line));
4   fgets (line, 256, stdin);
5   scanf ("%d", &y);
6   if (y > x)
7       y = y - x;
8   else {
9       x = getvalue();
10      y = y - x;
11  }
12  printf ("%s%d", line,y);
13  free(line);
}
```

Line			
2		*d(x,2)*	
3		*d(line,3)*	
4		*d(line,4)*	
5		*d(y,5)*	
6		*u(y,6) p-use*	*u(x,6) p-use*
7	*d(y,7)*	*u(x,7) c-use*	*u(y,7) c-use*
9		*d(x,9)*	
10	*d(y,10)*	*u(x,10) c-use*	*u(y,10) c-use*
12		*u(y,12) c-use*	
13		*k(line,13)*	

Data Flow testing - definitions

- Each field in a *record r* is treated as an individual variable
  - definition of *r* is interpreted as def of each field in *r*,
  - reference to *r* is interpreted as use of each field in *r*
- In general arrays are considered globally
  - a definition of *a[e]* is interpreted as a c-use of variables in *e* followed by a def of *a*
  - a reference to *a[e]* is interpreted as a c-use of variables in *e* followed by a use of *a*

Dataflow actions checklist

For each successive pair of actions

- dd - suspicious
- dk - probably defect
- du - normal case
- kd - okay
- kk - probably defect (harmless)
- ku - a defect
- ud - okay
- uk - okay
- uu - okay

Dataflow actions checklist

First occurrence of action on a variable

- k - suspicious
- d - okay
- u - suspicious (may be global)

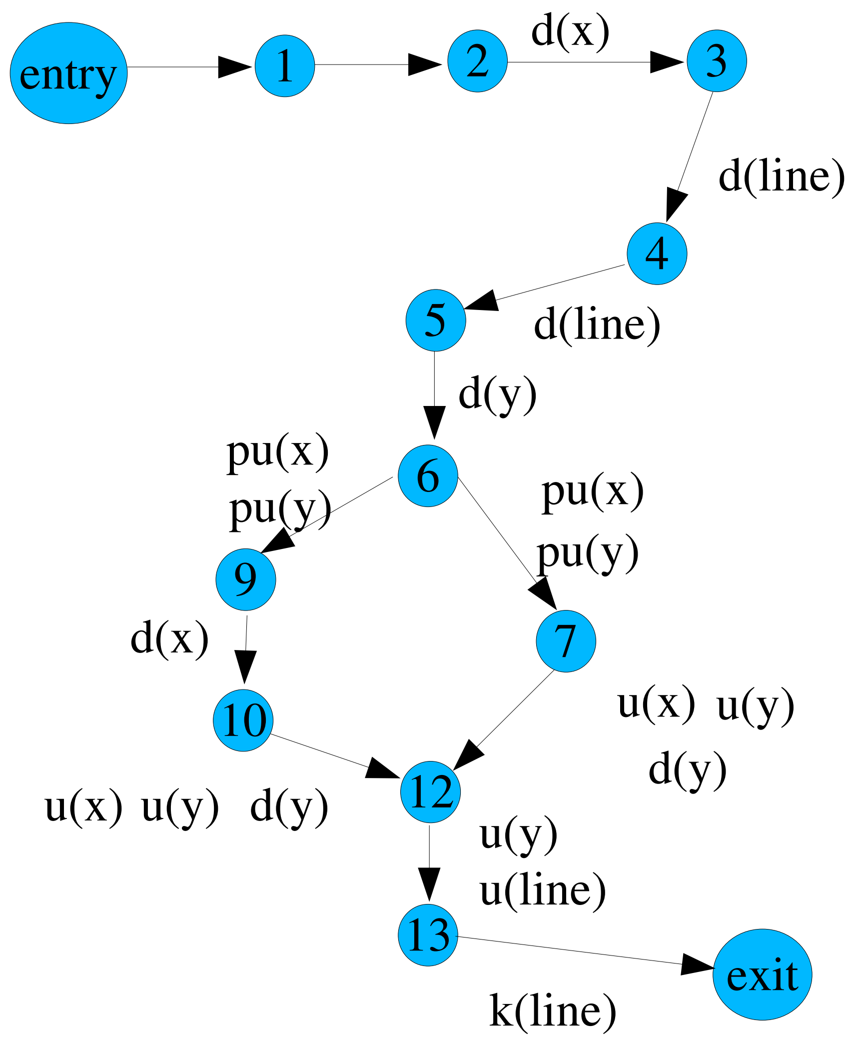
Dataflow actions checklist

Last occurrence of action on a variable

- k - okay
- d - suspicious (defined but never used after)
- u - okay (but may be deallocation forgotten)

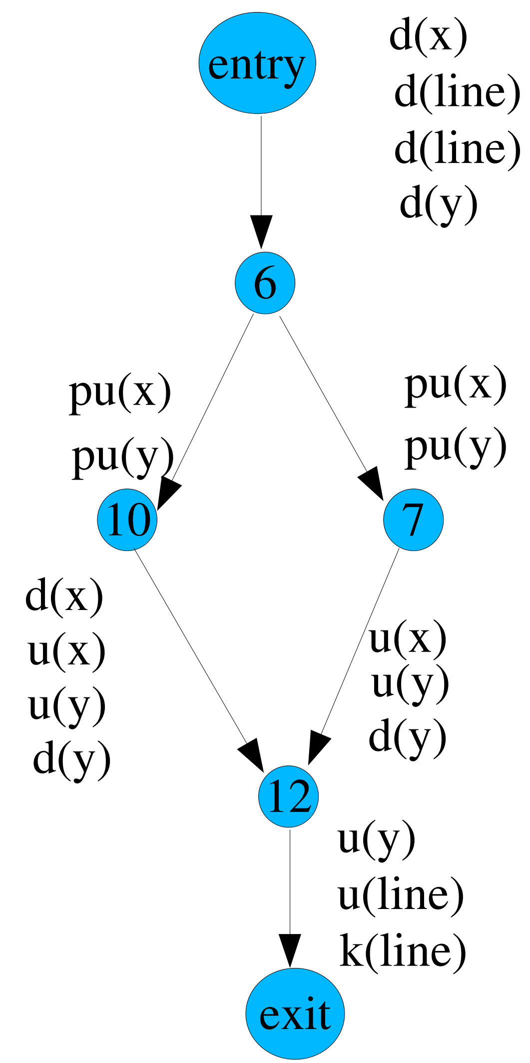
Data Flow testing – annotated flowchart

```
  int main (void)
 {
1   char *line;
2   int x = 0, y;
3   line = malloc(256 * sizeof (*line));
4   fgets (line, 256, stdin);
5   scanf ("%d", &y);
6   if (y > x)
7      y = y - x;
8   else {
9      x = getvalue();
10     y = y - x;
11 }
12 printf ("%s%d", line,y);
13 free(line);
}
```



Data Flow testing – annotated flowchart

```
int main (void)
{
  char *line;
  int x = 0, y;
  line = malloc(256 * sizeof (*line));
  fgets (line, 256, stdin);
  scanf ("%d", &y);
  if (y > x)
     y = y - x;
  else {
     x = getvalue();
     y = y - x;
  }
  printf ("%s%d", line,y);
  free(line);
}
```



Data Flow testing – definitions (1)

- *def-clear* path with respect to *v*
  - no redefinition of *v*
- *complete path*
  - initial node is *start node*, final node is *exit node*
- *du-pair* with respect *v* *(d,u)*
  - *d* node where *v* is defined
  - *u* node where *v* is used
  - def-clear path with respect to *v* from *d* to *u*

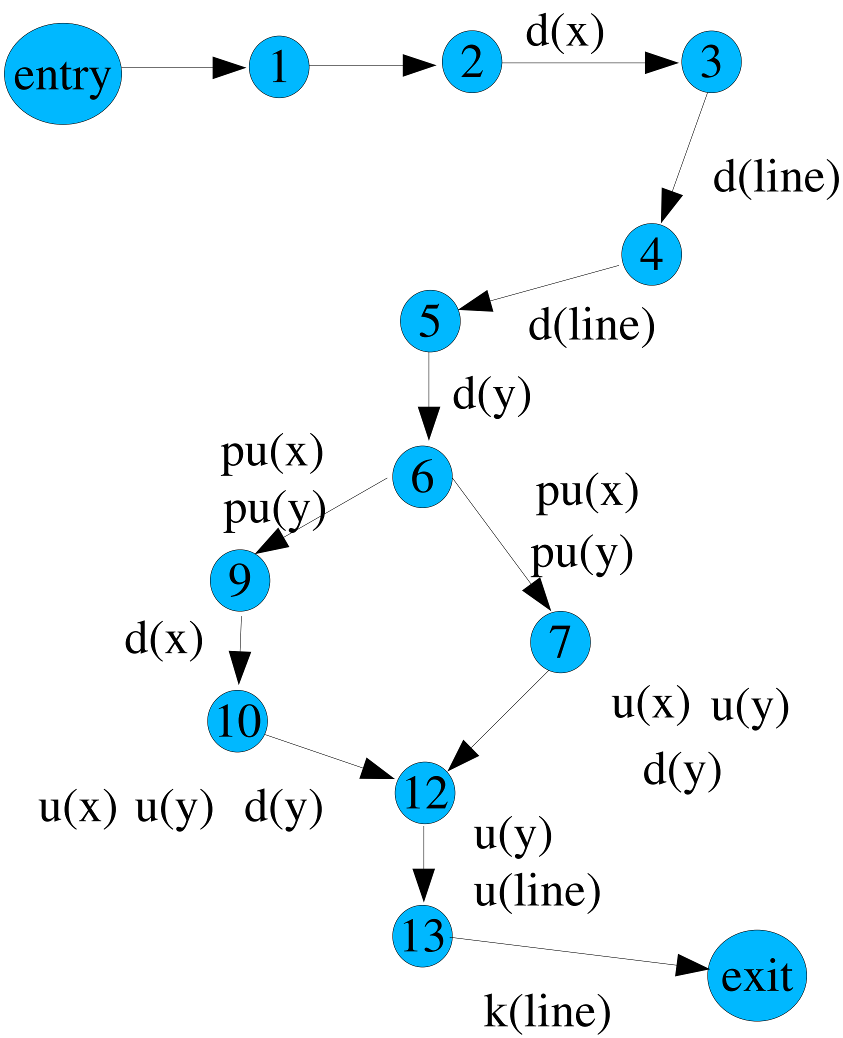
Data Flow testing – definitions (2)

- *simple* path
  - all nodes except possibly *first* and *last* are distincts
- *loop free* path
  - all nodes distincts
- *definition-use path* (*du-path*) with respect to *v*

  path <n1,n2,...,nj,nk> such that

  - d(v,n1) and either:
    - c-use of v at node nk, and
    - def-clear simple path
  - or
    - p-use of v on edge nj -> nk, and
    - <n1,n2,...nj> def-clear loop-free path

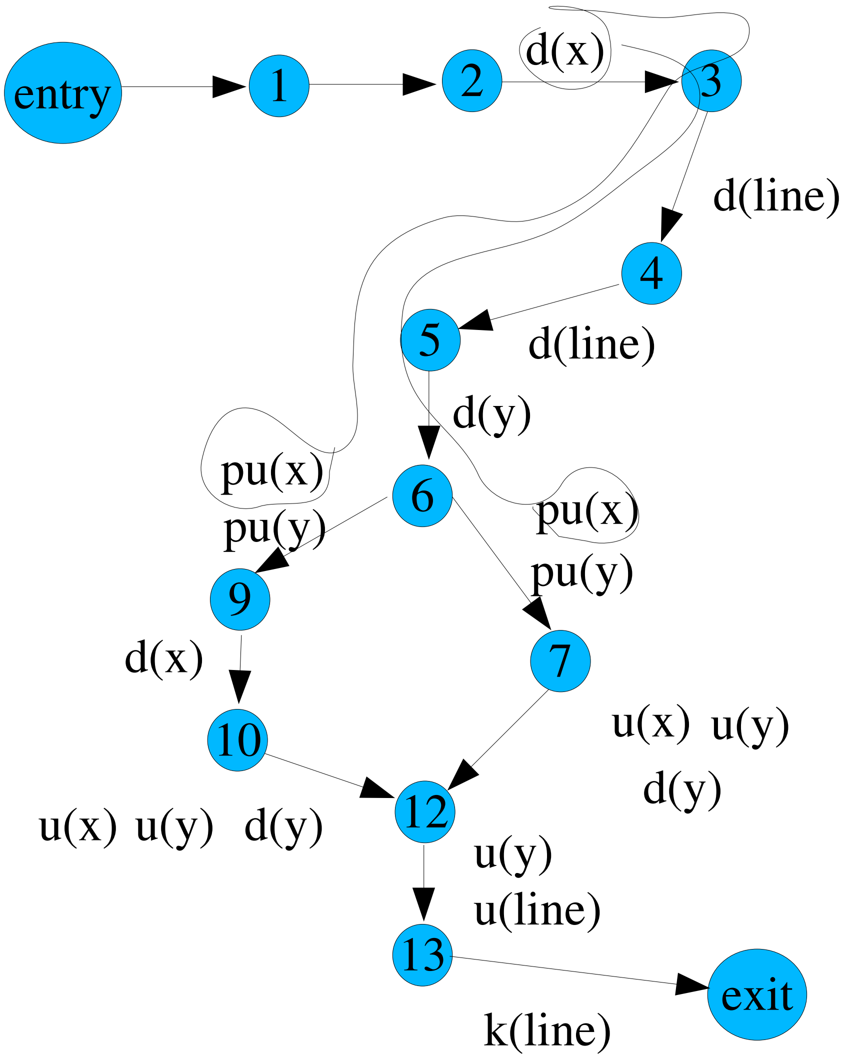
Define/Use information



Define/Use information

Variable	Define	Use	Comments
x	2		
		6-7, 6-9	p-use
		7	c-use
	9		READ
		10	c-use
line	3		mem-alloc
	4		READ
		12	c-use
y	5		
		6 -7, 6-9	p-use
		7	c-use
	7		
		10	c-use
	10		
		12	c-use

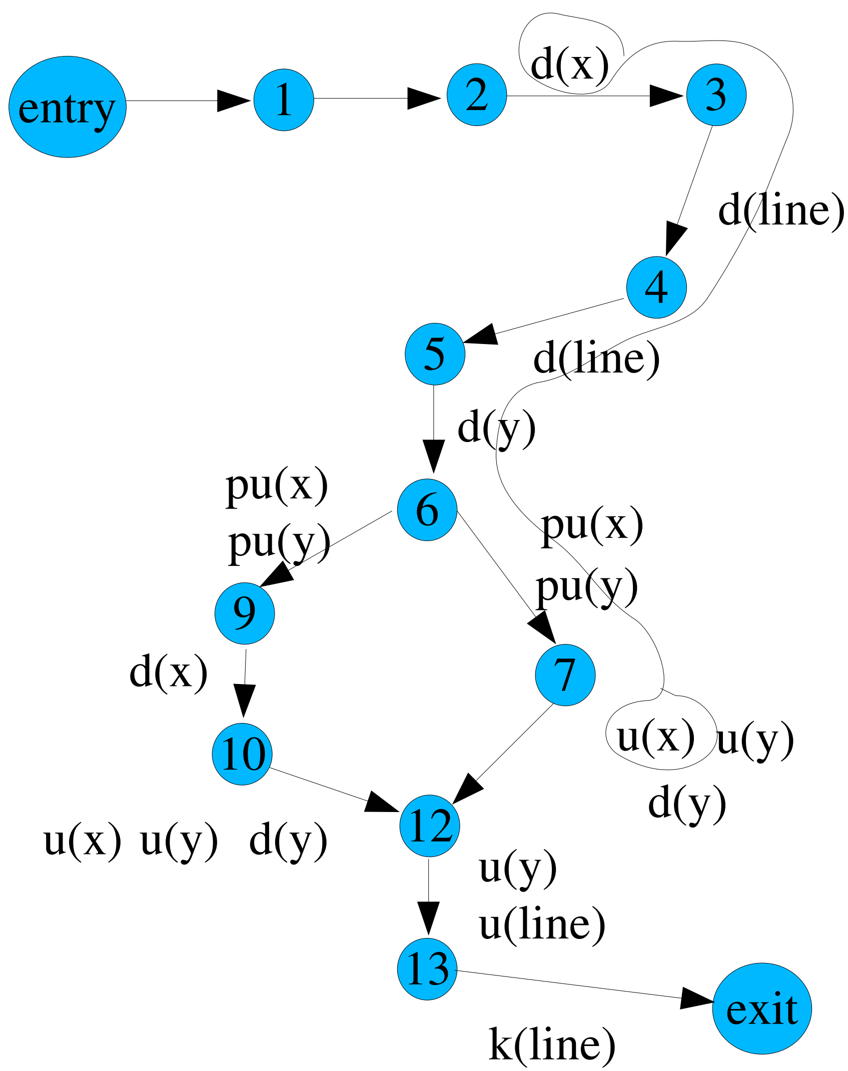
Example du-paths – variable *x*



2 – 3 – 4 – 5 - 6 - 7

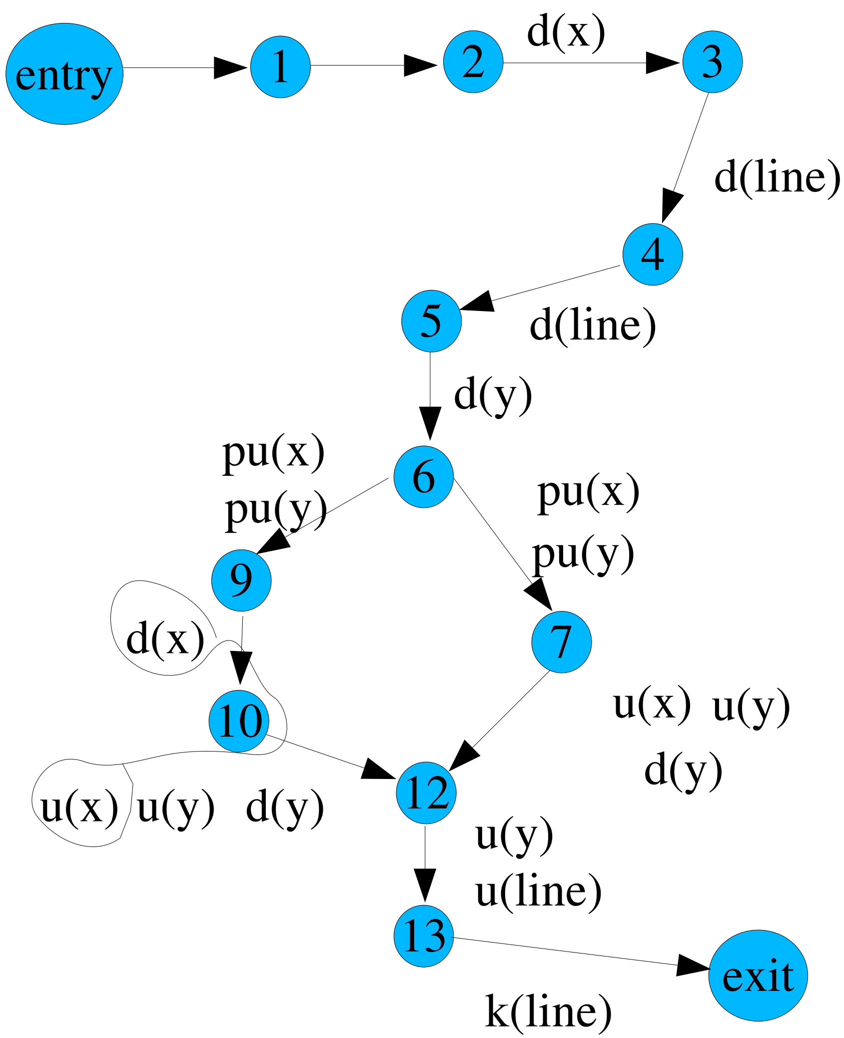
2 – 3 – 4 – 5 - 6 - 9

Example du-paths – variable *x*



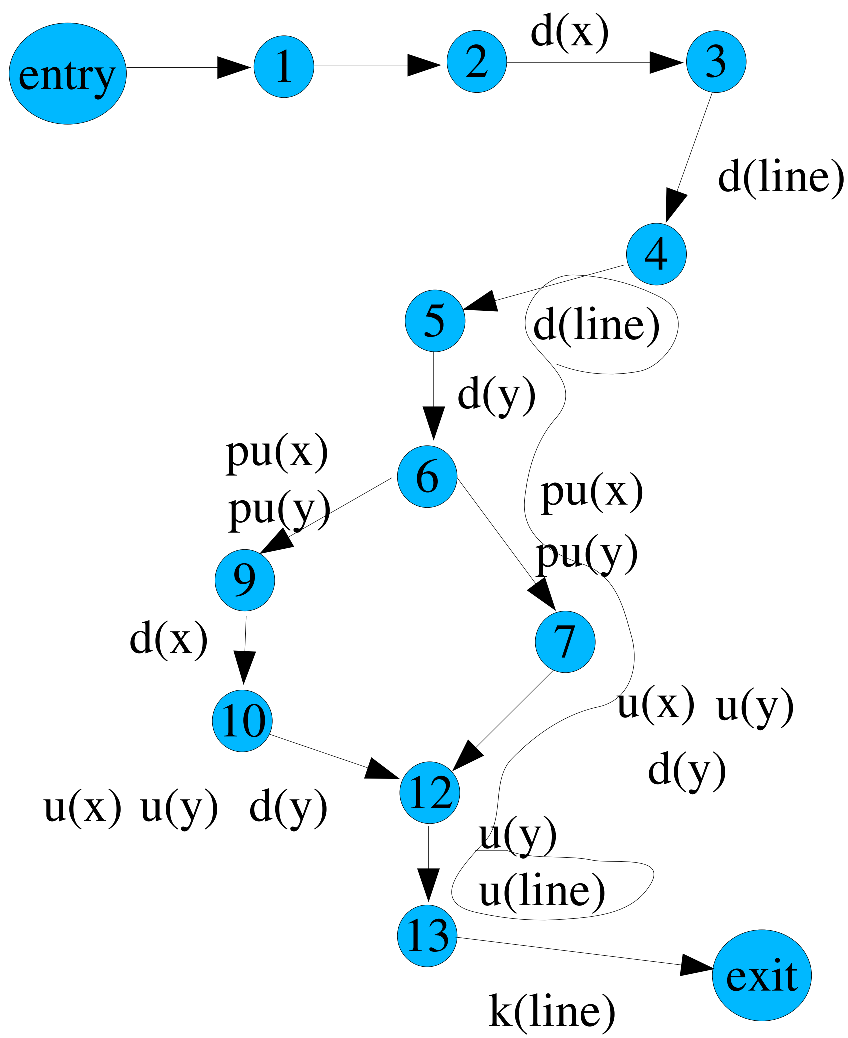
2 – 3 – 4 – 5 - 7

Example du-paths – variable *x*



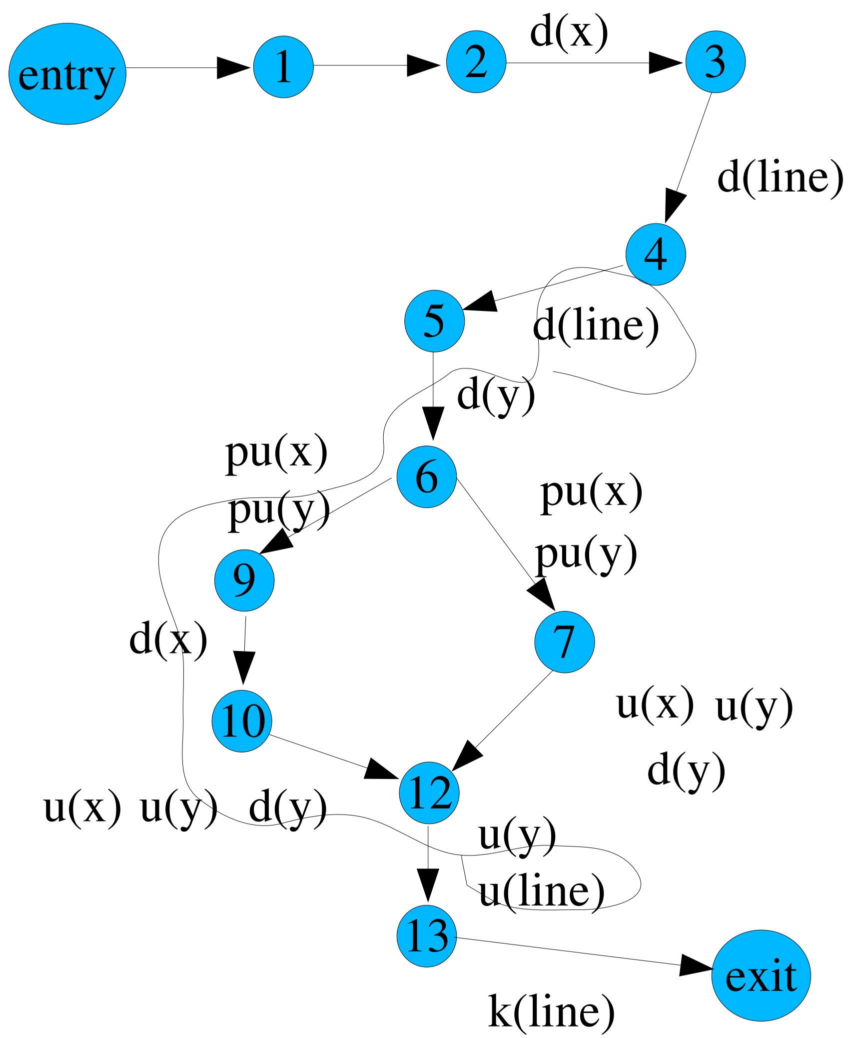
9 - 10

Example du-paths – variable *line*



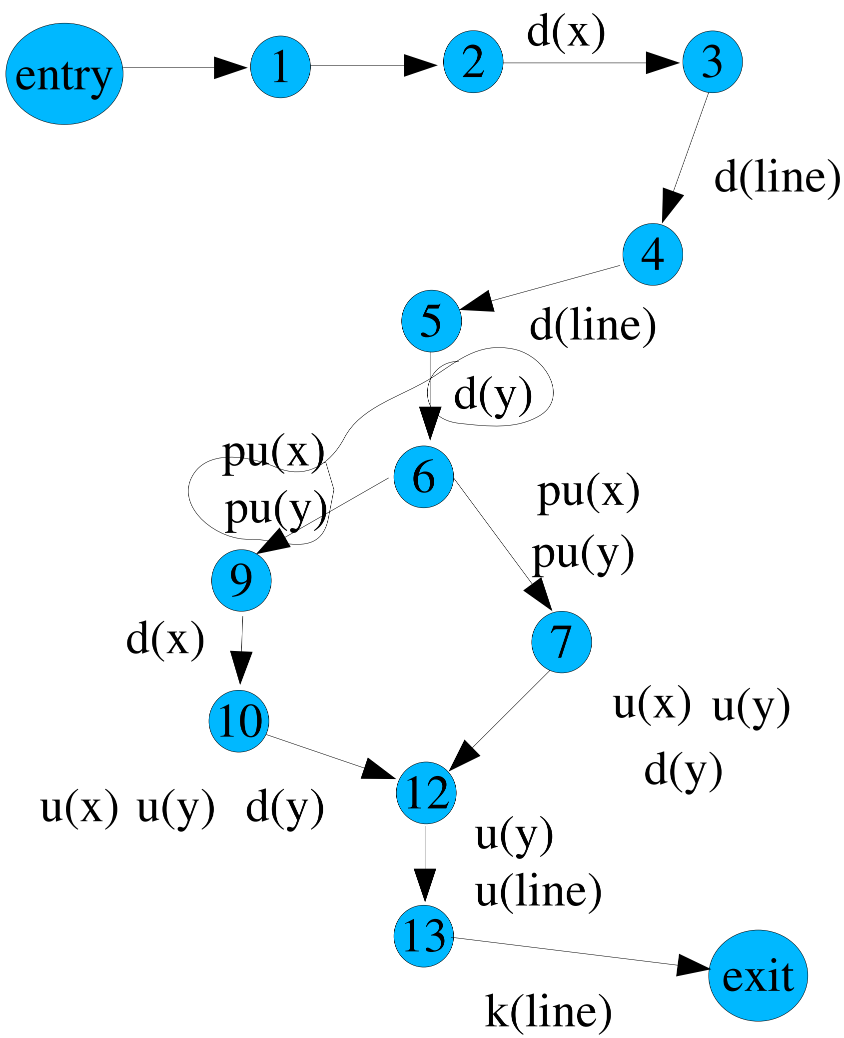
4 – 5 – 6 – 7 - 12

Example du-paths – variable *line*



4 – 5 – 6 – 9 - 10 - 12

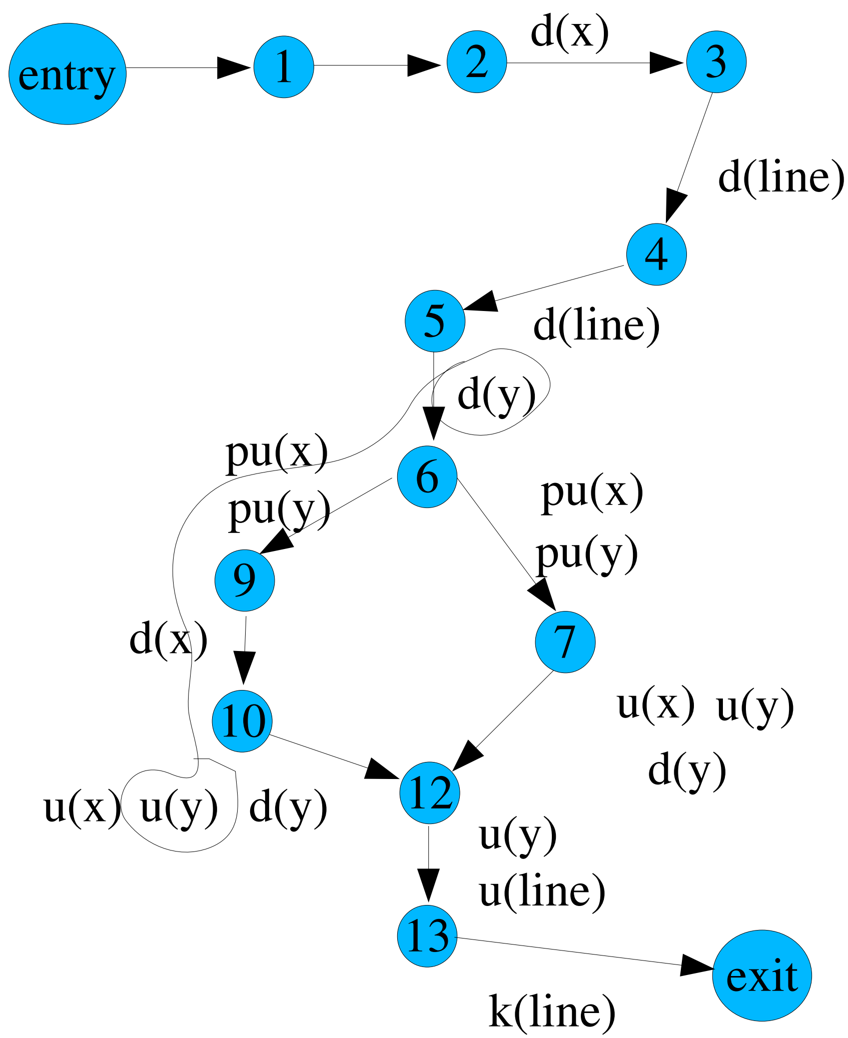
Example du-paths – variable *y*



5 – 6 - 7

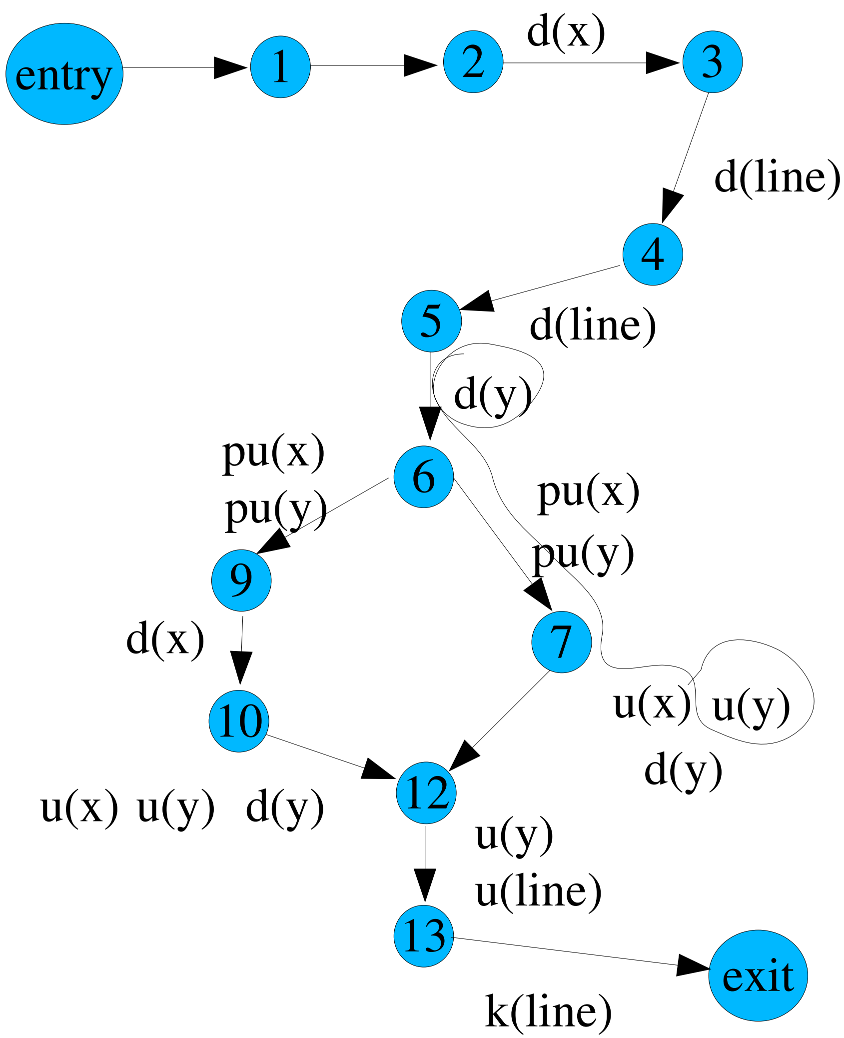
5 – 6 - 9

Example du-paths – variable *y*



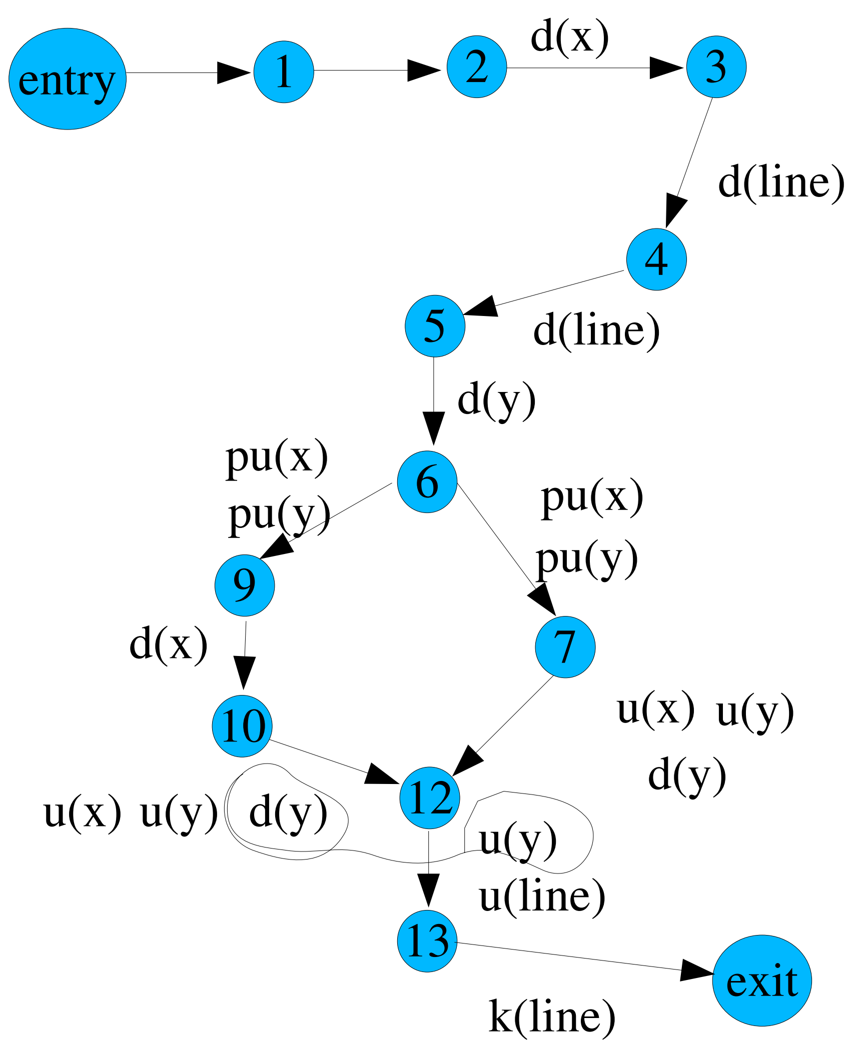
5 – 6 – 9 - 10

Example du-paths – variable *y*



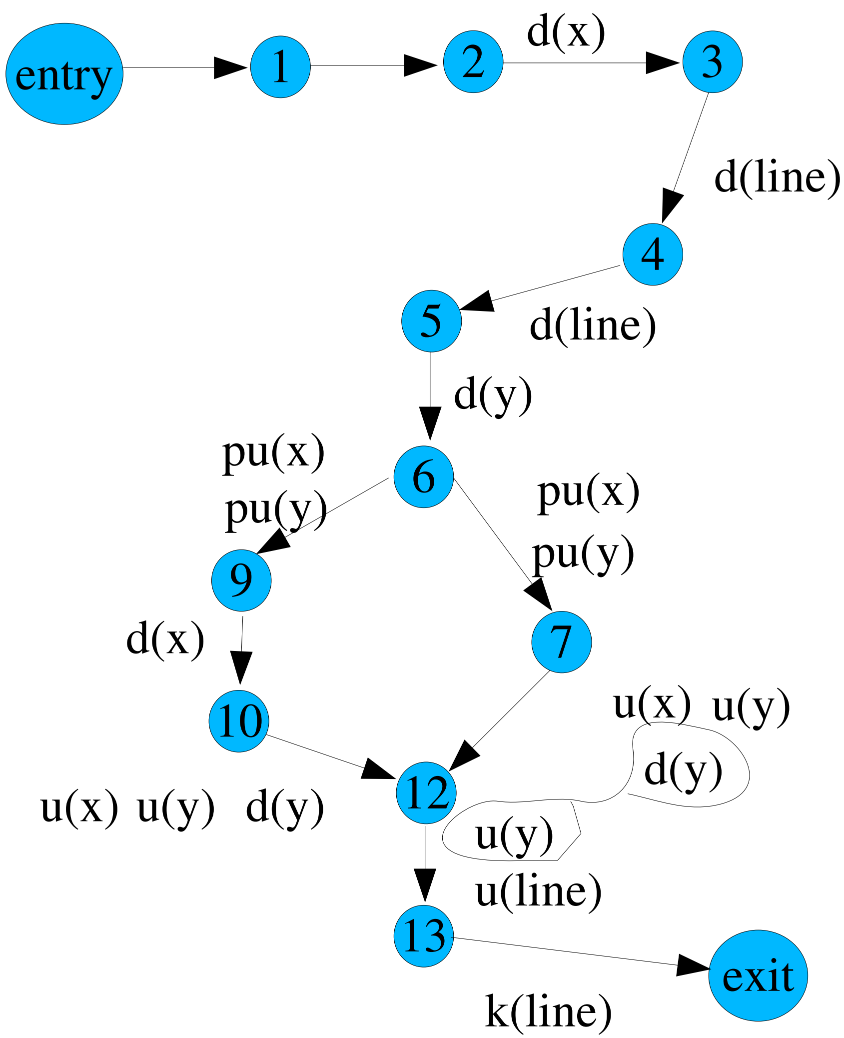
5 – 6 – 7

Example du-paths – variable *y*



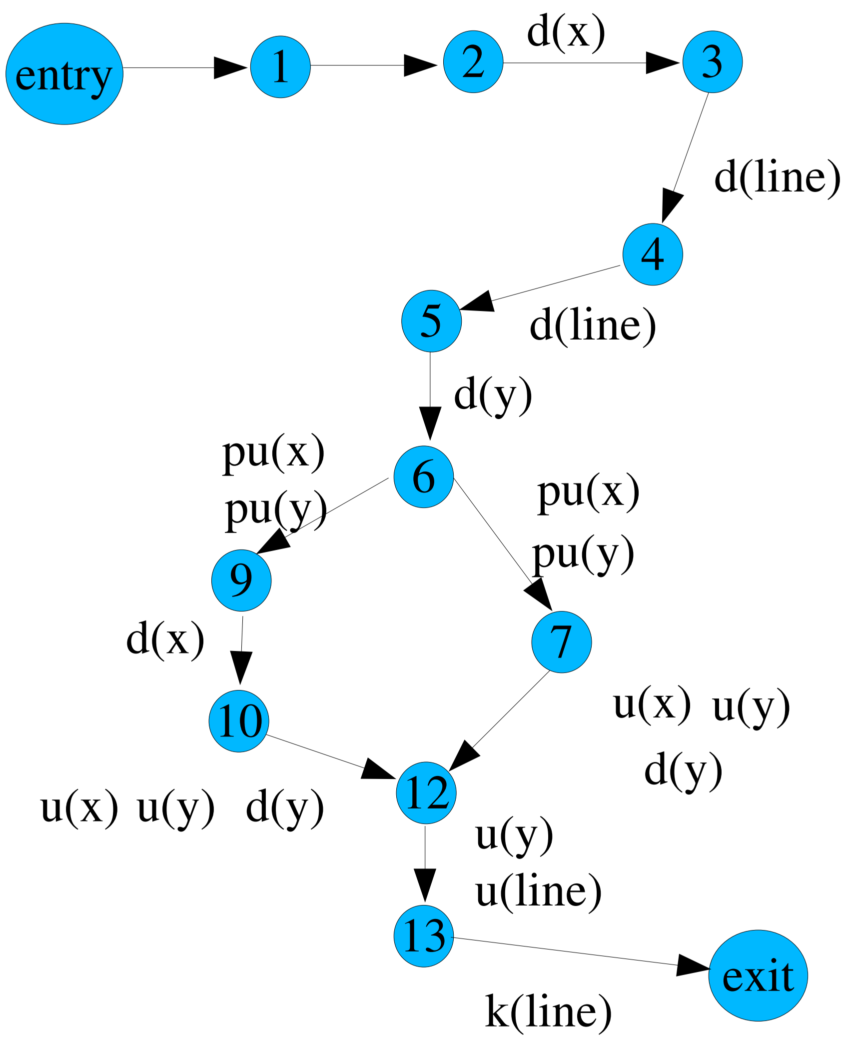
10 – 12

Example du-paths – variable *y*



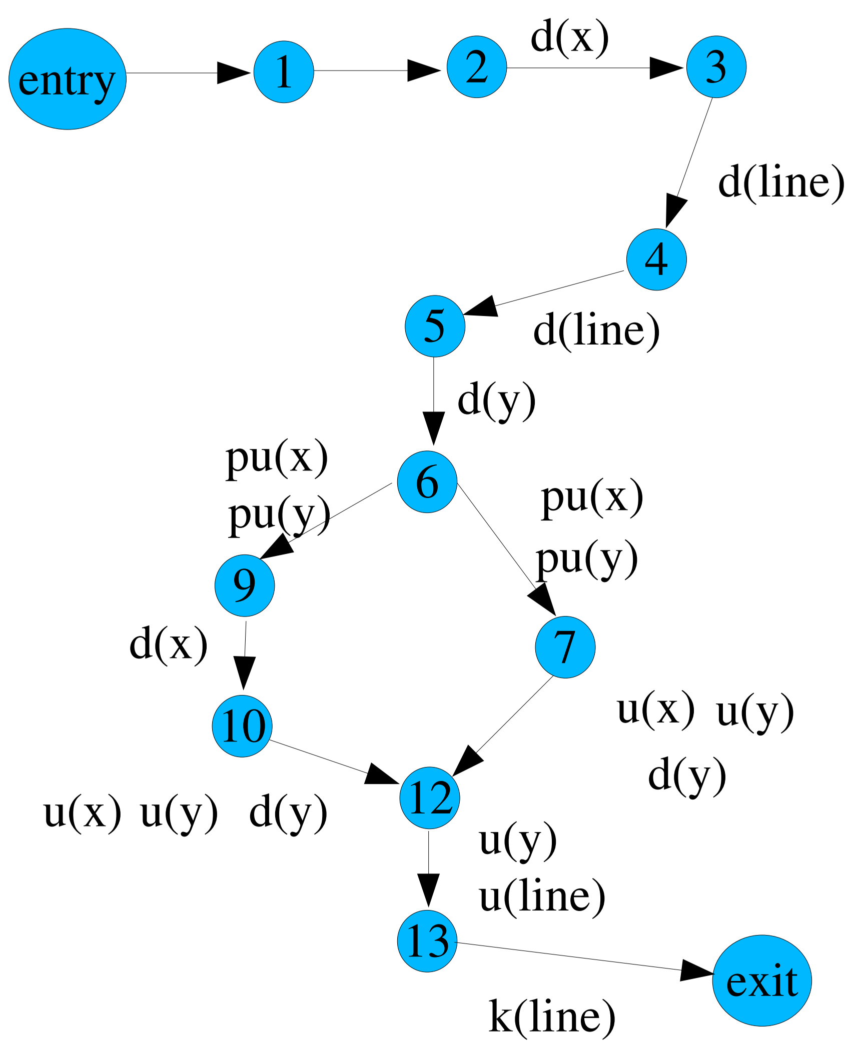
7 – 12

Example du-paths



Variable	du-path
x	2 – 3 – 4 – 5 – 6 – 7
	2 – 3 – 4 – 5 – 6 – 9
	9 – 10
y	5 – 6 – 7
	5 – 6 – 9
	5 – 6 – 9 – 10
	7 – 12
	10 – 12
line	4 – 5 – 6 – 7 – 12
	4 – 5 – 6 – 9 – 10 – 12

Data Flow Testing strategies



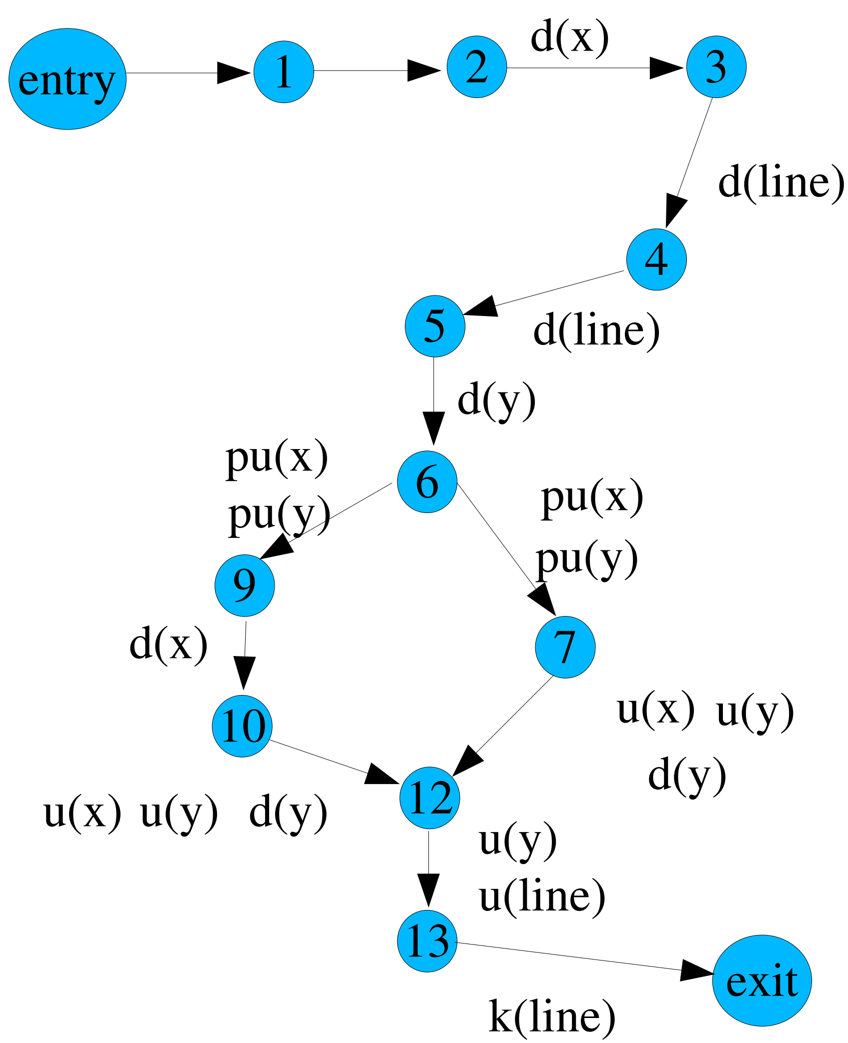
All-Defs:

- at least 1 *def-clear* path from every variable definition to at least 1 use of that variable definition

Example: tests covering

- 2 – 3 – 4 – 5 – 6 - 7 (*x*)
- 9 – 10 (*x*)
- 4 – 5 – 6 – 7 – 12 (*line*)
- 5 – 6 – 7 (*y*)
- 7 – 12 (*y*)
- 10 – 12 (*y*)

Data Flow Testing strategies



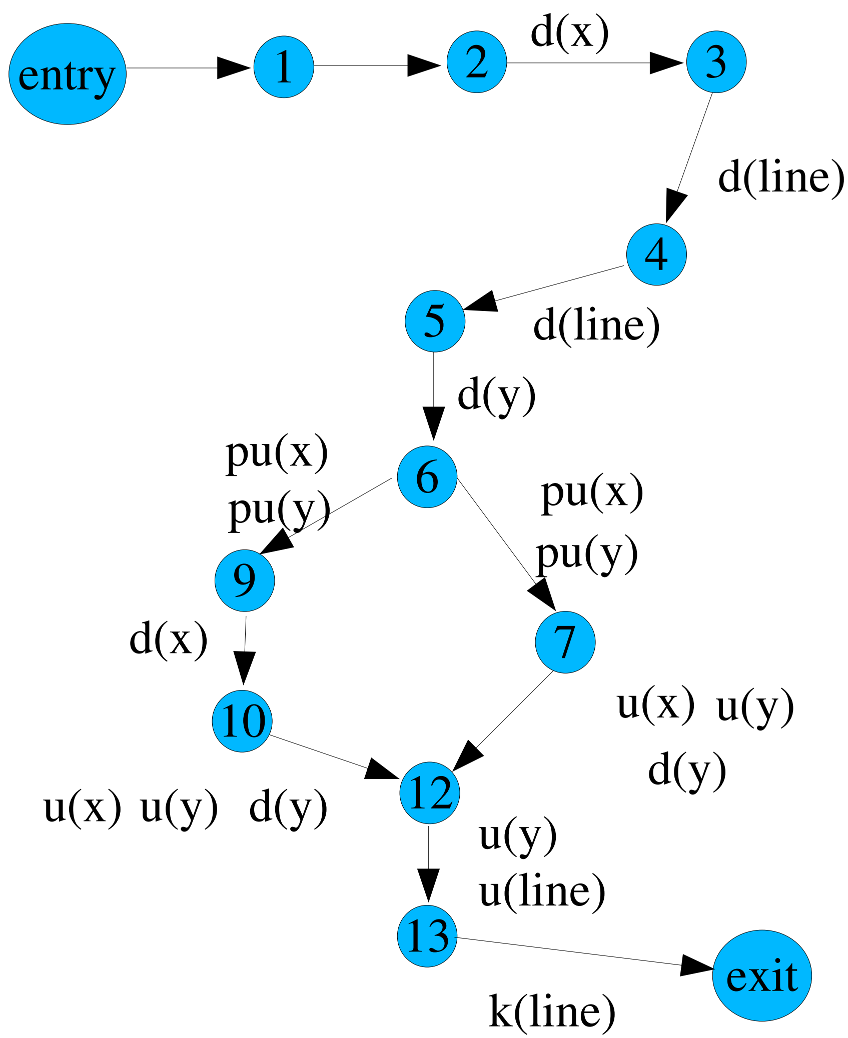
All-Uses:

- at least 1 *def-clear* path from every variable definition to every use of that variable definition

Example: tests covering

- 2 – 3 – 4 – 5 – 6 – 7 (*x*)
- 2 – 3 – 4 – 5 – 6 – 9 (*x*)
- 9 – 10 (*x*)
- 4 – 5 – 6 – 7 – 12 (*line*)
- 5 – 6 – 7 (*y*)
- 5 – 6 – 9 (*y*)
- 5 – 6 – 9 – 10 (*y*)
- 7 – 12 (*y*)
- 10 – 12 (*y*)

Data Flow Testing strategies



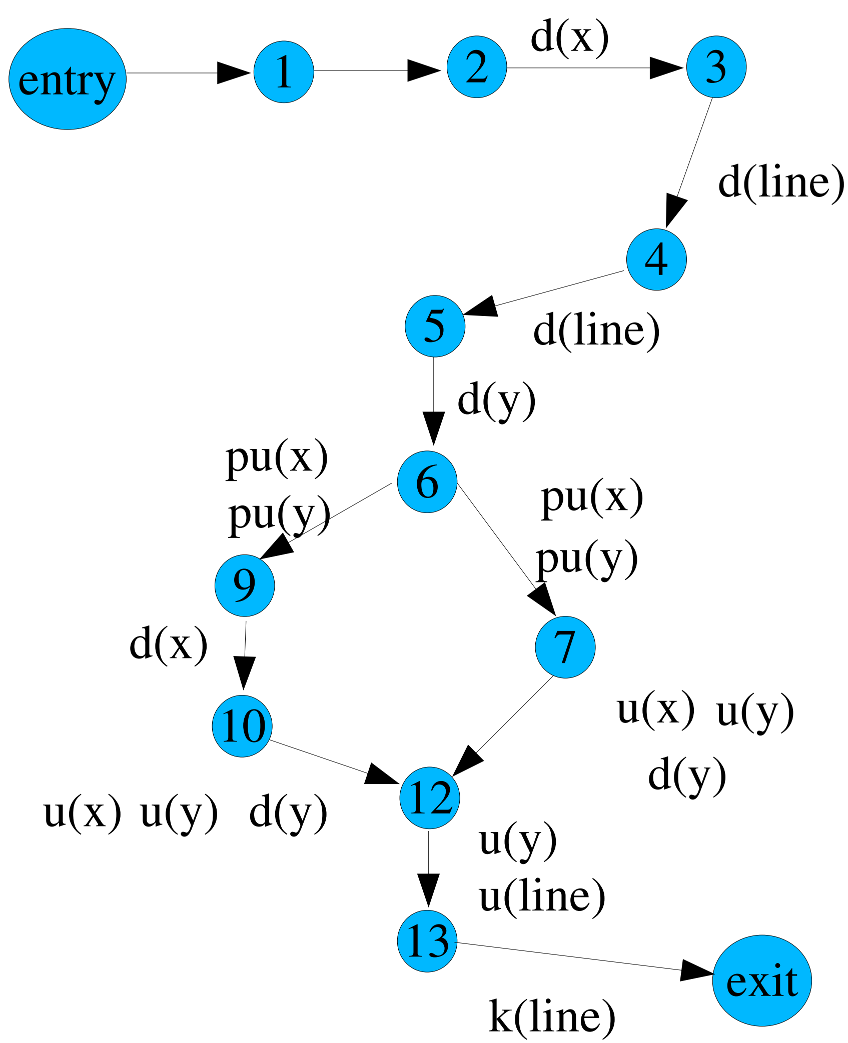
All-P-Uses/Some C-Uses:

- at least 1 *def-clear* path from every definition of *v* to every *p-use* of *v*, if none to a *c-use*

Example: tests covering

- 2 – 3 – 4 – 5 - 6 – 7 (*x*)
- 2 – 3 – 4 – 5 - 6 – 9 (*x*)
- 9 – 10 (*x*)
- 4 – 5 – 6 – 7 – 12 (*line*)
- 5 – 6 - 7 (*y*)
- 5 – 6 - 9 (*y*)
- 7 – 12 (*y*)
- 10 – 12 (*y*)

Data Flow Testing strategies



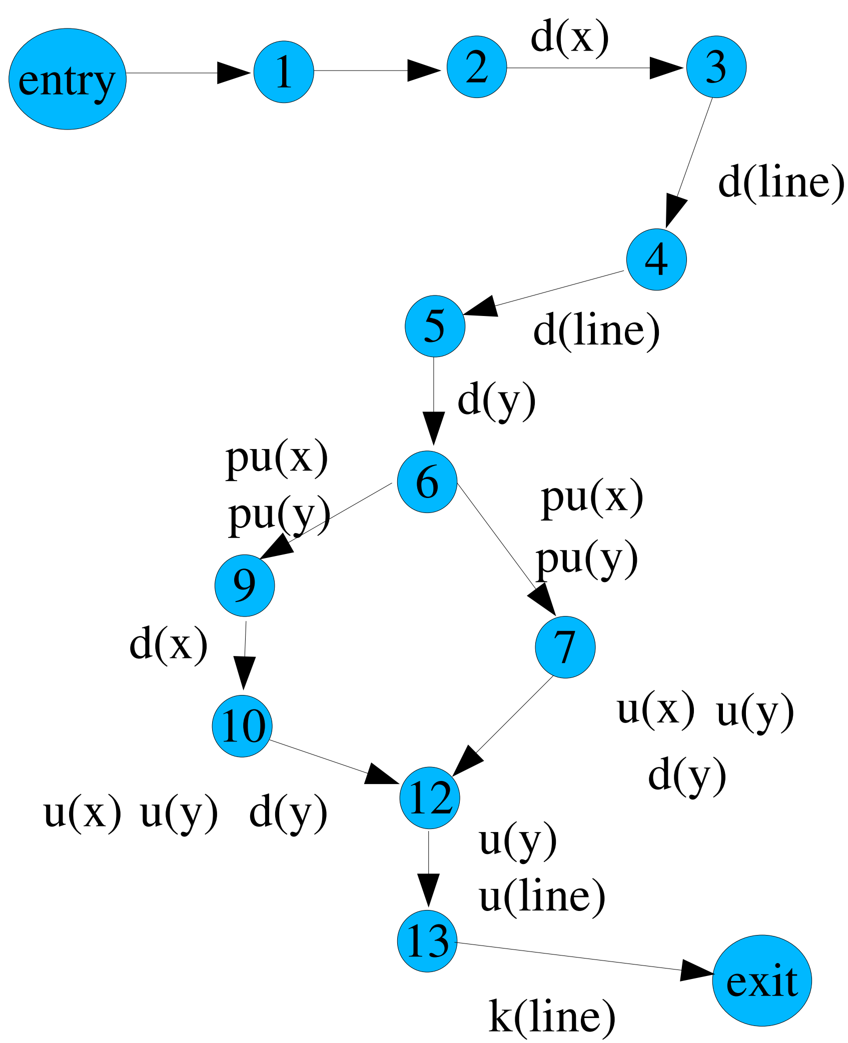
All-C-Uses/Some P-Uses:

- at least 1 *def-clear* path from every definition of *v* to every *c-use* of *v*, if none to a *p-use*

Example: test covering

- 2 – 3 – 4 – 5 - 6 – 7 (*x*)
- 9 – 10 (*x*)
- 4 – 5 – 6 – 7 – 12 (*line*)
- 5 – 6 – 7 (*y*)
- 5 – 6 – 9 – 10 (*y*)
- 7 – 12 (*y*)
- 10 – 12 (*y*)

Data Flow Testing strategies



All-du-paths:

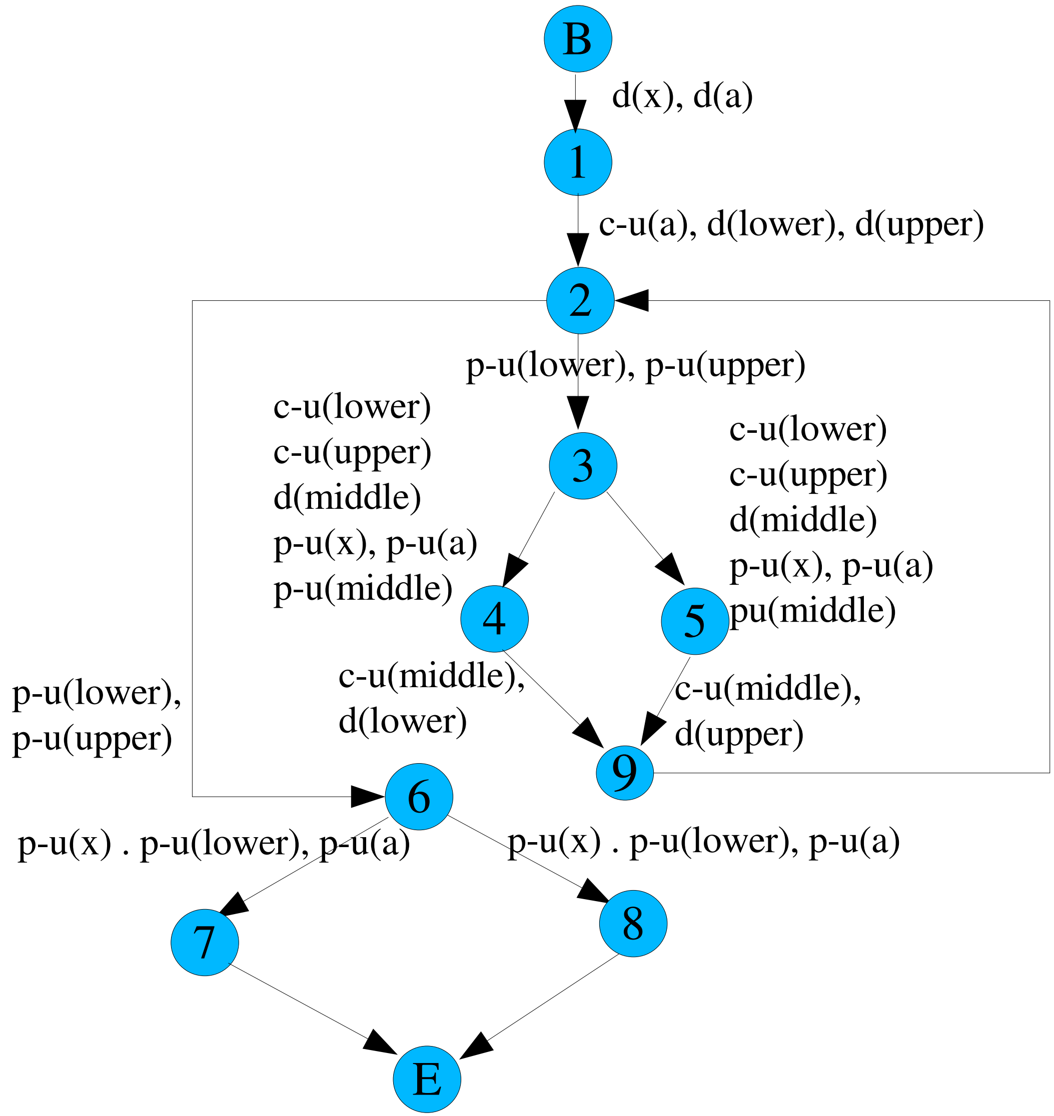
- all du-paths covered

Example: test covering

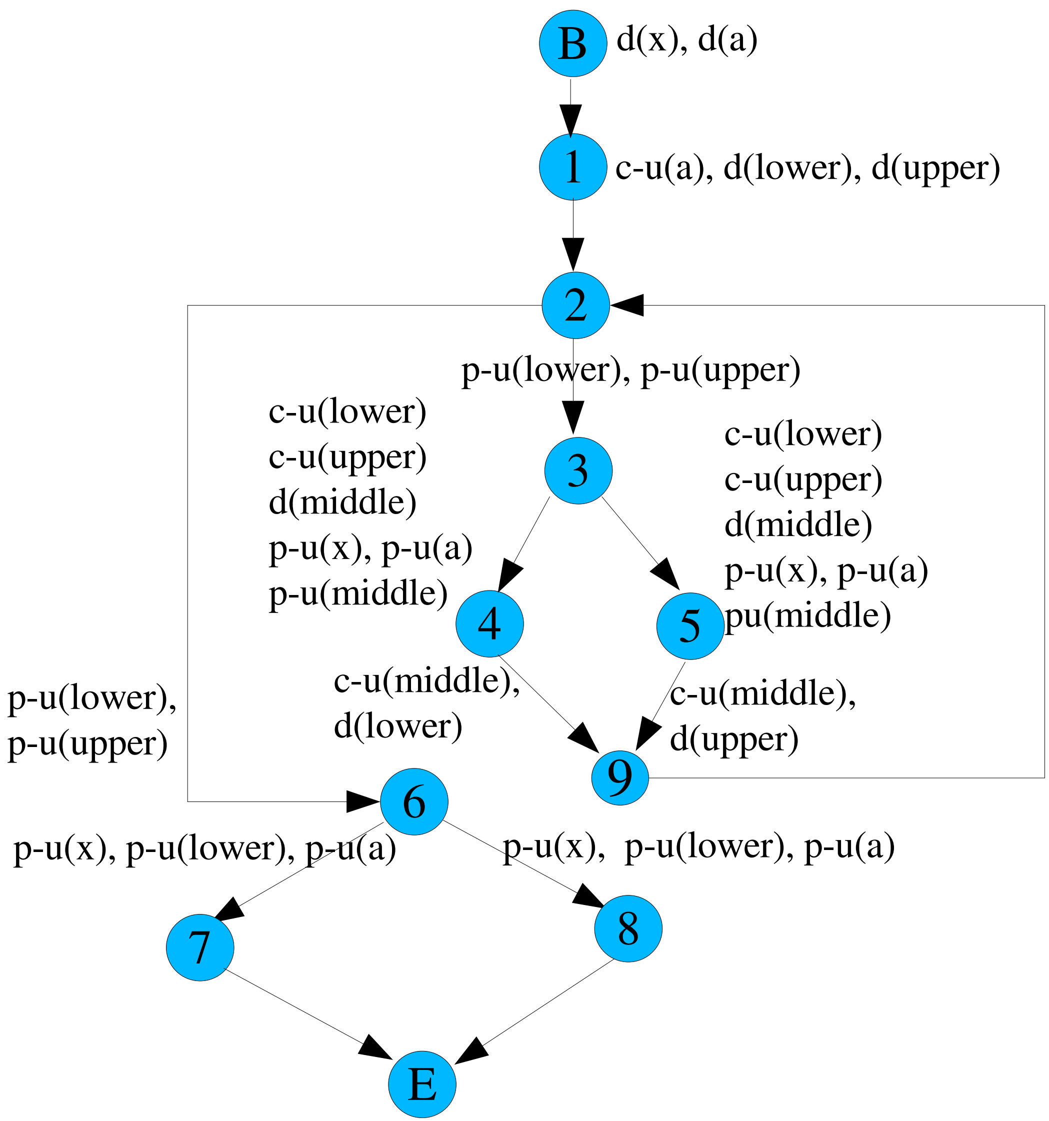
Variable	du-path
x	2 – 3 – 4 – 5 – 6 – 7 2 – 3 – 4 – 5 – 6 – 9 9 – 10
y	5 – 6 – 7 5 – 6 – 9 5 – 6 – 9 – 10 7 – 12 10 – 12
line	4 – 5 – 6 – 7 – 12 4 – 5 – 6 – 9 – 10 – 12

Example

```
boolean search(int x, int a[]) {
    int lower, upper, middle;
1   lower = 1; upper = a.length;
2   while (lower < upper) {
3      middle = (lower + upper)/2;
       if (x > a[middle])
4         lower = middle + 1;
5      else upper = middle;
    }
6 if (a[lower] == x)
7      return true;
8   else return false;
}
```



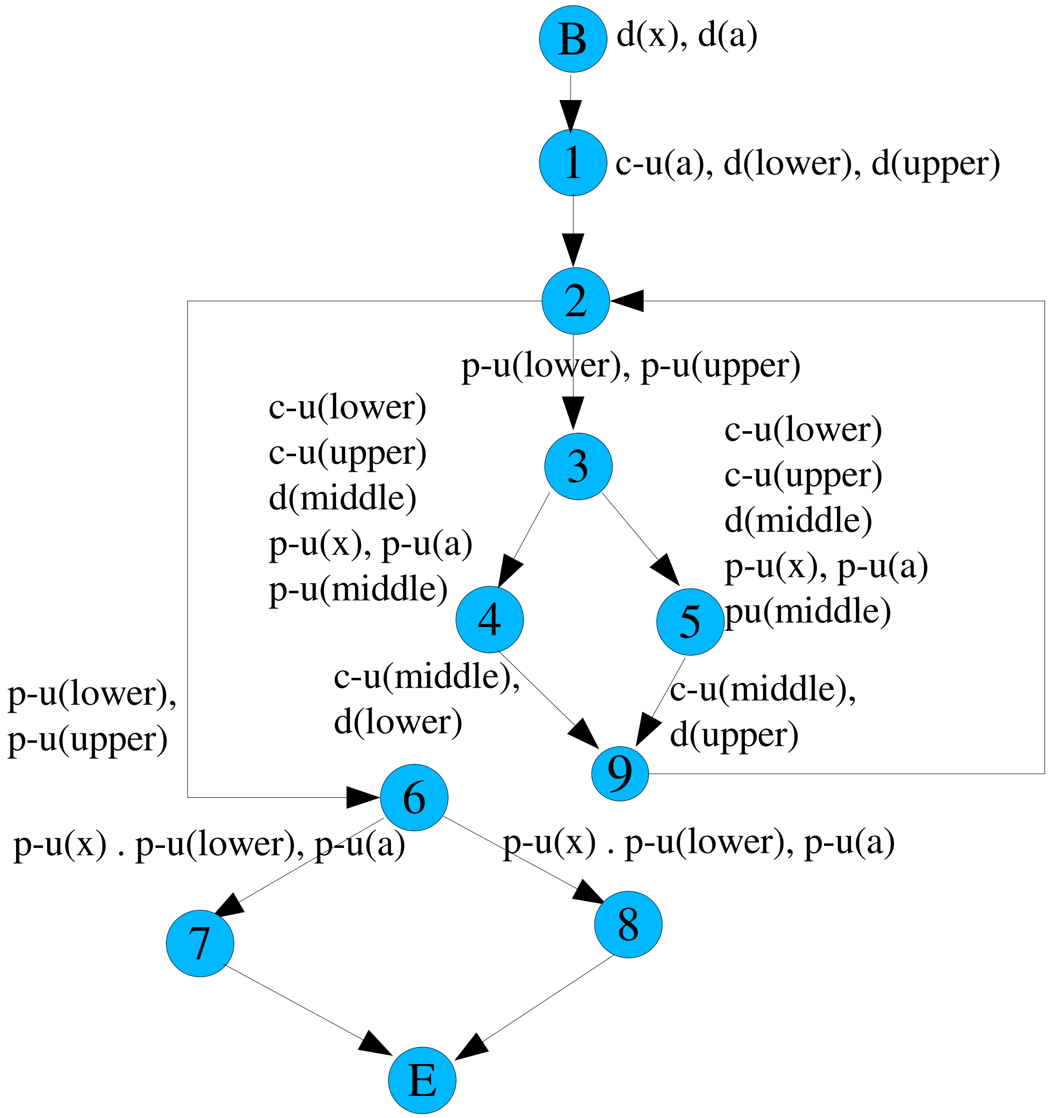
Example



Definitions/uses

Variable	Define	Use	Comments
x	B	3-4, 3-5	p-use
		6 -7, 6-8	p-use
a	B	1	c-use
		3-4, 3-5	p-use
		6-7, 6-8	p-use
lower	1	2-3, 2-6	p-use
		3	c-use
		6 -7, 6-8	p-use
	4		
upper	1	2-3, 2-6	p-use
		3	
	5		p-use
middle	3	3-4, 3-5	p-use
		4	c-use
		5	c-use

Example



Du-paths

- x
  - B-1-2-3-4
  - B-1-2-3-5
  - B-1-2-6-7
  - B-1-2-6-8
- a
  - B-1
  - B-1-2-6-7
  - B-1-2-6-8
  - B-1-2-3-4
  - B-1-2-3-5
- lower
  - 1-2-3
  - 1-2-6
  - 1-2-6-7
  - 1-2-6-8
  - 4-9-2-3
  - 4-9-2-6
- upper
  - 1-2-3
  - 1-2-6
  - 5-9-2-3
  - 5-9-2-6
- middle
  - 3-4
  - 3-5

Example

Du-paths

- x
  - B-1-2-3-4
  - B-1-2-3-5
  - B-1-2-6-7
  - B-1-2-6-8
- a
  - B-1
  - B-1-2-6-7
  - B-1-2-6-8
  - B-1-2-3-4
  - B-1-2-3-5
- lower
  - 1-2-3
  - 1-2-6
  - 1-2-6-7
  - 1-2-6-8
  - 4-9-2-3
  - 4-9-2-6
- upper
  - 1-2-3
  - 1-2-6
  - 5-9-2-3
  - 5-9-2-6
- middle
  - 3-4
  - 3-5

Examples of paths satisfying all du-paths:

- B-1-2-3-4-9-2-3-5-9-2-3-5-9-2-6-7-E
- B-1-2-3-5-9-2-3-4-9-2-6-8-E
- B-1-2-6-7-E
- B-1-2-6-8-E

Hierarchy of coverage criteria