10. Data flow testing

Usage
Definitions, c-uses and p-uses
Defining test cases
Hierarchy of data flow coverage
Annotations and time sequence pairs

Data flow testing

- A structural testing technique
- Aims to execute sub-paths from points where each variable is defined to points where it is referenced
- These sub-paths are called definition-use pairs, or du-pairs (du-paths, du-chains)
- Data flow testing is centred on variables (data)

Data flow testing – rationale

- Most failures involve execution of an incorrect definition
  - Incorrect assignment or input statement
  - Incorrect path is taken, which leads to incorrect definition (predicate is faulty)
  - Definition is missing (i.e. null definition)
- Data flow testing follows the sequences of events related to a given data item with the objective to detect incorrect sequences
- It explores the effect of using the value produced by every and each computation

Weyuker E (1993) More experience with data flow testing. TSE 19

Example

SolveQuadratic

Quadratic equation:

$Ax^2 + Bx + C = 0$

It can have up to two real (i.e. not complex) solutions.

First test to see whether the real solutions exist, and how many:

if $B^2 – 2AC > 0$ there are two solutions
if $B^2 – 2AC = 0$ there is one solutions
if $B^2 – 2AC < 0$ there are no real solutions

If solutions exist, compute:

$x1 = -B + \mathrm{sqrt}(B^2 – 2AC)/2A$
$x2 = -B - \mathrm{sqrt}(B^2 – 2AC)/2A$

Example

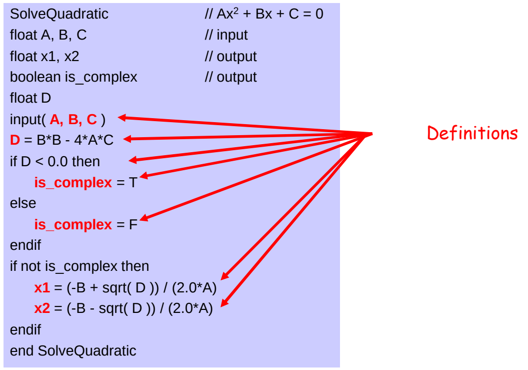
```
SolveQuadratic                  // Ax^2 + Bx + C = 0
float A, B, C                   // input
float x1, x2                    // output
boolean is_complex               // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Terminology

```
SolveQuadratic                  // Ax^2 + Bx + C = 0
float A, B, C                   // input
float x1, x2                    // output
boolean is_complex               // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

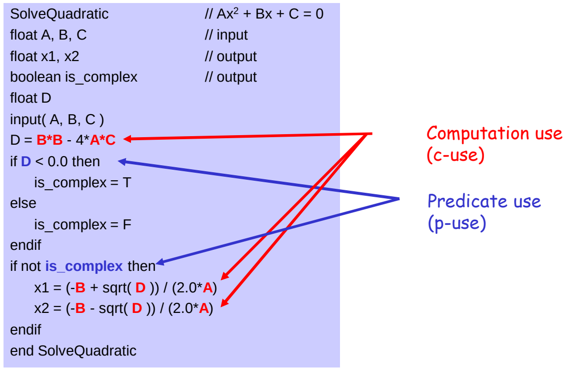
} Declarations

Terminology



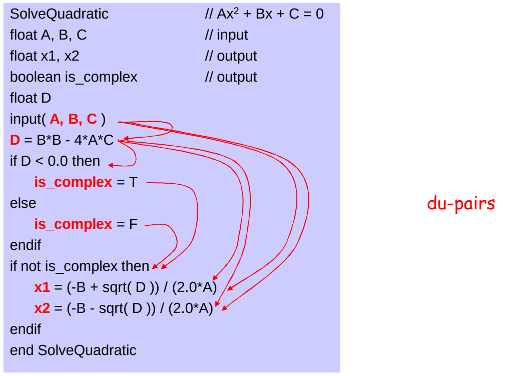
```
SolveQuadratic          // Ax^2 + Bx + C = 0
float A, B, C           // input
float x1, x2            // output
boolean is_complex      // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Terminology



```
SolveQuadratic          // Ax^2 + Bx + C = 0
float A, B, C           // input
float x1, x2            // output
boolean is_complex      // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Terminology



```
SolveQuadratic          // Ax^2 + Bx + C = 0
float A, B, C           // input
float x1, x2            // output
boolean is_complex      // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Terminology

- **Variable definition**

  Occurrences of a variable where a variable is given a new value (assignment, input by the user, input from a file, etc.)

  Variable DECLARATION is NOT its definition !!!

- **Variable uses**

  Occurences of a variable where a variable is not given a new value (variable DECLARATION is NOT its use)

Terminology

- **Variable uses**

  Occurences of a variable where a variable is not given a new value

  - **p-uses (predicate uses)**

    Occur in the predicate portion of a decision statement such as if-then-else, while-do etc.

  - **c-uses (computation uses)**

    All others, including variable occurrences in the right hand side of an assignment statement, or an output statement

- **du-path**

  A sub-path from a variable definition to its use

Data Flow testing

- Checks the correctness of the du-paths in a Control Flow Graph of a program
- Test case definitions based on four groups of coverage
  - All definitions
  - All c-uses
  - All p-uses
  - All du-paths

Data Flow testing: key steps

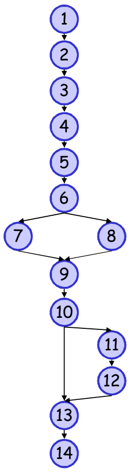
Given a code (program or pseudo-code)

1. Number the lines
2. List the variables
3. List occurrences & assign a category to each variable
4. Identify du-pairs and their use (p- or c-)
5. Define test cases, depending on the required coverage

Data flow testing

1: number the lines

```
1    SolveQuadratic
2    float A, B, C, D, x1, x2
3    boolean is_complex
4    input( A, B, C )
5    D = B*B - 4*A*C
6    if D < 0.0
7         then is_complex = T
8         else is_complex = F
9    endif
10   if not is_complex
11        then x1 = (-B + sqrt( D )) / (2.0*A)
12                x2 = (-B - sqrt( D )) / (2.0*A)
13   endif
14   end SolveQuadratic
```



Data flow testing

2: list the variables

```
1    SolveQuadratic
2    float A, B, C, D, x1, x2
3    boolean is_complex
4    input( A, B, C )
5    D = B*B - 4*A*C
6    if D < 0.0
7         then is_complex = T
8         else is_complex = F
9    endif
10   if not is_complex
11        then x1 = (-B + sqrt( D )) / (2.0*A)
12                x2 = (-B - sqrt( D )) / (2.0*A)
13   endif
14   end SolveQuadratic
```

A, B, C
x1, x2
D
is_complex

Data flow testing

3:list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

```
1    SolveQuadratic
2    float A, B, C, D, x1, x2
3    boolean is_complex
4    input( A, B, C )
5    D = B*B - 4*A*C
6    if D < 0.0
7         then is_complex = T
8         else is_complex = F
9    endif
10   if not is_complex
11        then x1 = (-B + sqrt( D )) / (2.0*A
12                x2 = (-B - sqrt( D )) / (2.0*A)
13   endif
14   end SolveQuadratic
```

Data flow testing

3:list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4			
5	D		
6			D
7			
8			
9			
10			
11			
12			
13			
14			

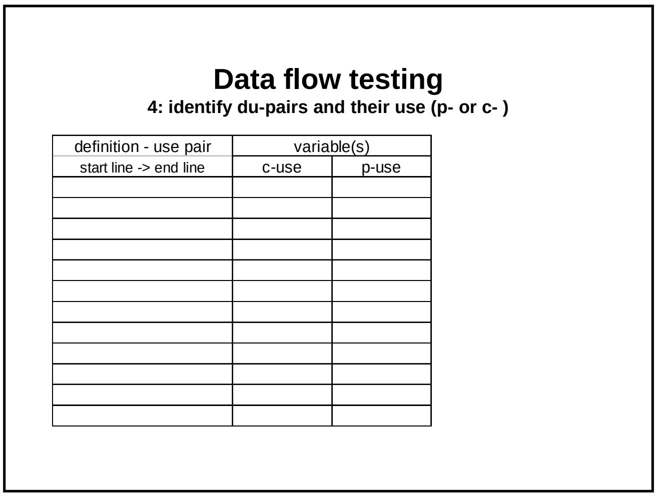
```
1    SolveQuadratic
2    float A, B, C, D, x1, x2
3    boolean is_complex
4    input( A, B, C )
5    D = B*B - 4*A*C
6    if D < 0.0
7         then is_complex = T
8         else is_complex = F
9    endif
10   if not is_complex
11        then x1 = (-B + sqrt( D )) / (2.0*A
12                x2 = (-B - sqrt( D )) / (2.0*A)
13   endif
14   end SolveQuadratic
```

Data flow testing

3: list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4	A, B, C		
5	D	A, B, C	
6			D
7	is_complex		
8	is_complex		
9			
10			is_complex
11	x1	A, B, D	
12	x2	A, B, D	
13			
14			

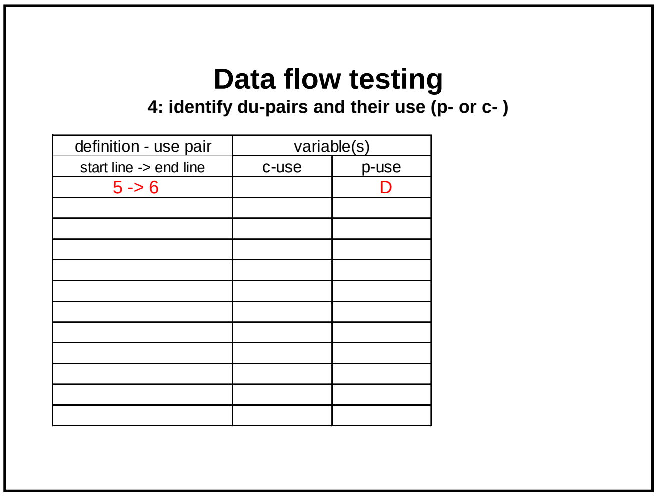
```
1    SolveQuadratic
2    float A, B, C, D, x1, x2
3    boolean is_complex
4    input( A, B, C )
5    D = B*B - 4*A*C
6    if D < 0.0
7         then is_complex = T
8         else is_complex = F
9    endif
10   if not is_complex
11        then x1 = (-B + sqrt( D )) / (2.0*A
12                x2 = (-B - sqrt( D )) / (2.0*A)
13   endif
14   end SolveQuadratic
```



Data flow testing

4: identify du-pairs and their use (p- or c-)

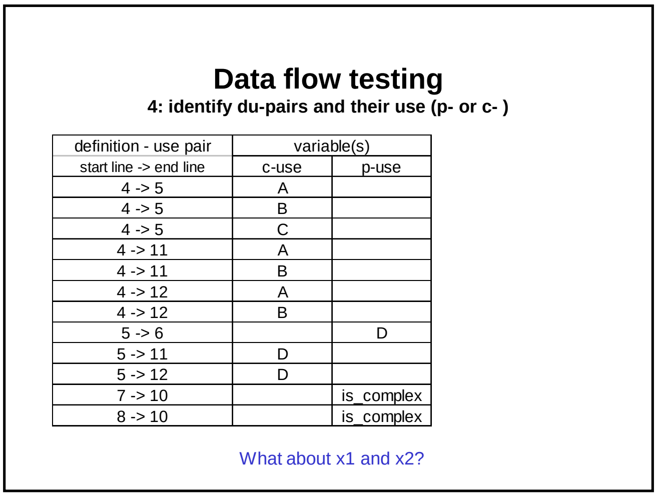
definition - use pair	variable(s)	
start line -> end line	c-use	p-use



Data flow testing

4: identify du-pairs and their use (p- or c-)

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
5 -> 6		D

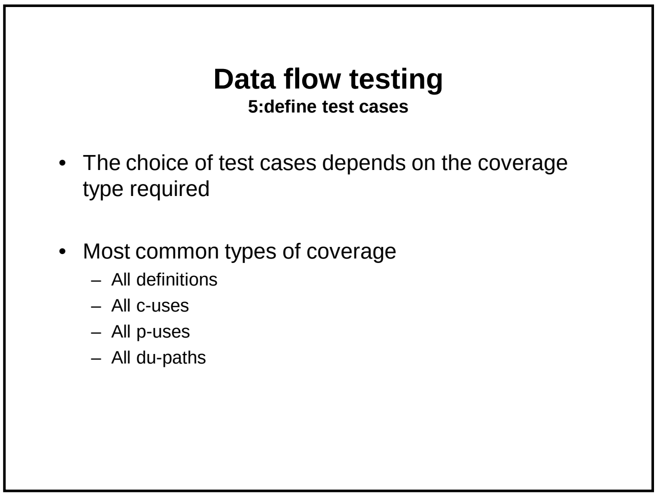


Data flow testing

4: identify du-pairs and their use (p- or c-)

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

What about x1 and x2?

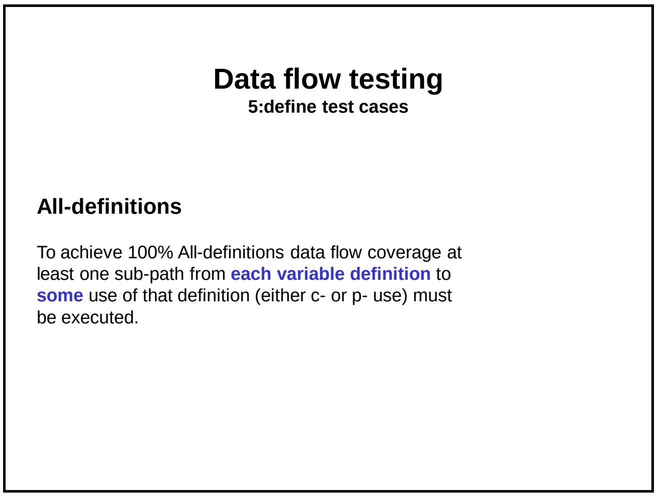


Data flow testing

5:define test cases

- The choice of test cases depends on the coverage type required

- Most common types of coverage
  - All definitions
  - All c-uses
  - All p-uses
  - All du-paths

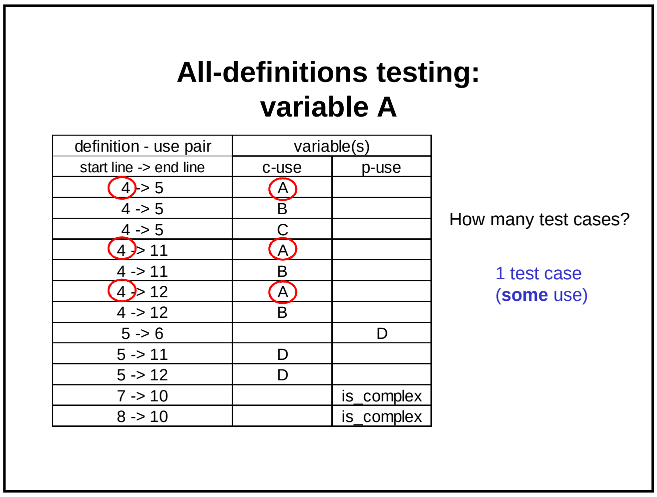


Data flow testing

5:define test cases

All-definitions

To achieve 100% All-definitions data flow coverage at least one sub-path from **each variable definition** to **some** use of that definition (either c- or p- use) must be executed.



All-definitions testing: variable A

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

How many test cases?

1 test case
(**some** use)

All-definitions testing: variable is_complex

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

How many test cases?

2 test cases (at least one use for each definition)

Data flow testing: All-definitions test cases

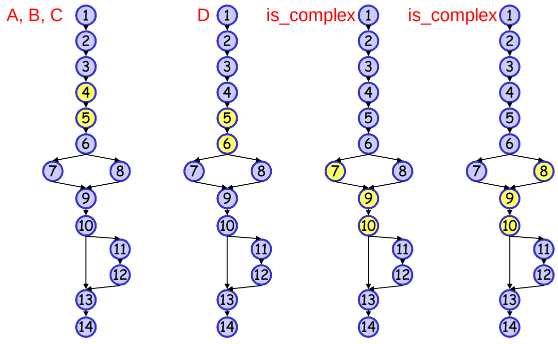
			Inputs			Expected outcome		
variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2

Data flow testing: All-definitions test cases

			Inputs			Expected outcome		
variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2
A	4 -> 5	4-5	1	1	1	T	n/a	n/a
B	4 -> 5	4-5	1	1	1	T	n/a	n/a
C	4 -> 5	4-5	1	1	1	T	n/a	n/a
D	5 -> 6	5-6	1	1	1	T	n/a	n/a
is_complex	7 -> 10	7-10	1	1	1	T	n/a	n/a
is_complex	8 -> 10	8-10	1	2	1	F	-1	-1

These are just sample inputs

			Inputs			Expected outcome		
variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2
A, B, C	4 -> 5	4-5	1	1	1	T	n/a	n/a
D	5 -> 6	5-6	1	1	1	T	n/a	n/a
is_complex	7 -> 10	7-10	1	1	1	T	n/a	n/a
is_complex	8 -> 10	8-10	1	2	1	F	-1	-1



Data flow testing

5:define test cases

All-c-uses

To achieve 100% All-c-uses data flow coverage at least one sub-path from **each variable definition** to **every** c-use of that definition must be executed.

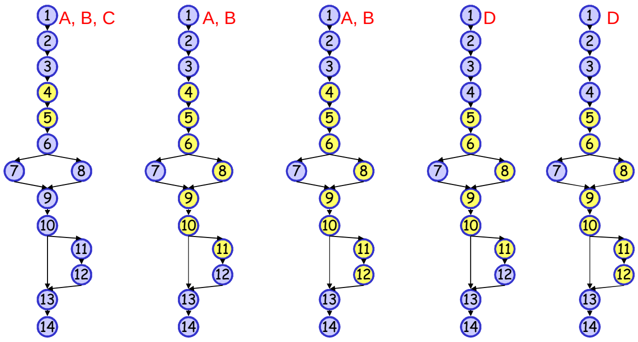
All-c-uses testing: variable A

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

How many test cases?

3 test cases (every c-use)

variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2
A, B, C	4 -> 5	4-5	1	1	1	T	n/a	n/a
A, B	4 -> 11	4-5-6-8-9-10-11	1	2	1	F	-1	-1
A, B	4 -> 12	4-5-6-8-9-10-11-12	1	2	1	F	-1	-1
D	5 -> 11	5-6-8-9-10-11	1	2	1	F	-1	-1
D	5 -> 12	5-6-8-9-10-11-12	1	2	1	F	-1	-1



Data flow testing

5:define test cases

All-p-uses

To achieve 100% All-p-uses data flow coverage **at least one** sub-path from **each** variable definition to **every** p-use of that definition must be executed.

All-uses

To achieve 100% All-uses data flow coverage **at least one** sub-path from **each** variable definition to **every** use of that definition (both p- and c- use) must be executed.

All-du-paths

To achieve 100% All-du-paths data flow coverage **every** simple sub-path from **each** variable definition to **every** use of that definition must be executed.

All-p-uses testing: variable is_complex

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

How many test cases?

2 test cases (from each definition to every p-use)

Data flowgraph

- A graphical tool to explore the sequences of events related to a variable of interest
- Can help to detect data flow anomalies
- Uses annotations on the Control Flow Graph (CFG)
- CFG remains the same for every variable, but annotations change

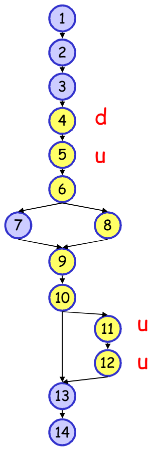
Data flowgraph - annotations

- **d** – defined, created, initialised, read, etc.
  - Always on the LHS of the expression
- **k** – killed, undefined, freed, released, etc
- **u** – used for something (i.e. c-use and p-use)
  - c - always on the RHS of the expression
  - p – used in a predicate or as a control variable of loop

Data flowgraph - annotations

- Examples
  - **v = expression**
    c-use of all the variables in the expression
    definition(s) of v
  - **method call: F(p1, p2, …, pn)**
    definition of each formal parameter inside the method
  - **if B then S1 else S2**
    p-use of each variable in boolean expression B
    S1 and S2 classified depending on their composition

Annotated Flow Graph for variable A in the SolveQuadratic program



Time-sequence pairs:
d-u
u-u
u-u
All correct

Examples of time sequence pairs

- Normal
  - du: a “du-pair”
  - uu: a sequence of uses
  - uk: use then kill
  - kd: kill then redefine
- Suspect
  - ud: usually normal, re-assignment after use
  - kk: kill twice, harmless but probably a bug
  - dk: define then kill, without use, probably a bug
  - dd: define twice, probably a bug
  - d: defined but never used
- Bug
  - ku: kill then use
  - u: used before defined
  - k: not defined but killed

Data flow testing

- There exist tools for assisting with data flow testing
- Test cases are usually defined by a tester

Next lecture

Software testing tools

Further reading

- Additional material on the web
  http://www.cs.bham.ac.uk/~exc/Teaching/STesting
  - BCS Standard for Software Component Testing
- http://www.rspa.com/reflib/TestingTactics.html
  - A large collection of articles on testing
  - Look out for “structured testing” and “white box testing”

Homework

- Define test cases for All-p-uses and All-uses for the SolveQuadratic program
- Define test cases for All-p-uses for program TRIANGLE (defined in the Path Testing lecture)
- Given test case definitions for
  - All-definitions
  - All-c-uses
  - All-p-uses
  - All-uses

  Specify additional test cases to achieve All-du-paths coverage
- Specify the use annotations (definition, c-use, p-use) for the following control flow statements
  - read(v1, …, vn)
  - write(v1, …, vn)
  - while B do S
  - for(v=e1 to e2 step e3)

  (e.g. for the assignment statement, v = expression, all the variables in the ‘expression’ have label ‘c-use’ and v has label ‘definition’)