These lectures provide an overview of data flow testing. At the end of this section you should be able to:

1. Describe the main data flow test techniques
2. Apply the data flow testing
3. Identify the strengths and weaknesses of these techniques

**Data Flow Testing**

These notes are based upon notes by Prof. Robert Hierons, whom I gratefully acknowledge.

**Definition**

These lectures provide an overview of data flow testing. At the end of this section you should be able to:

1. Describe the main data flow test techniques
2. Apply the data flow testing
3. Identify the strengths and weaknesses of these techniques

**Reading**

The material for this section is covered in slides and these notes. The techniques are also described, with examples, in the draft Unit Test British Standard which may be found at: http://www.testingstandards.co.uk/

This is an excellent source of information on testing and I strongly recommend it to you as a source of additional material to back up the notes and slides I provide.

**Data flow testing**

Unless we test all paths, we might not have tested the ways in which data can flow in the program. This section will describe data-flow and test techniques based around this. Naturally, data-flow test techniques are intended to complement control-flow techniques, not replace them – ideally both types are used. In control flow testing we seek to cover all branches or all nodes. In data flow testing, we are concerned with the flow of data in the program.

When we consider data-flow in a program we are interested in how values assigned to variables in statements can affect other parts of the program. We are thus interested in definitions and uses of variables. Data-flow techniques are thus based around the following definitions:

- A node $n$ of the CFG is a **definition** for a variable $x$ if $n$ represents an assignment to $x$ or an input of a value into $x$.
- A node $n$ of the CFG is a **computation-use** (c-use) for a variable $x$ if $n$ represents a statement that references $x$ (either $n$ is an assignment with $x$ on its right hand side, $n$ is a function/procedure call with $x$ as an actual parameter, or $n$ is an output statement that references $x$).
- A pair $n_1$-$n_2$ of nodes that form an edge of the CFG is a **predicate-use** (p-use) for a variable $x$ if $n_1$ represents a control predicate that references $x$. It is often convenient to associate the predicate use with the predicate node $n_1$.
- A structure of the CFG is an **use** for a variable $x$ if it is either a c-use or a p-use of $x$. Note this may be a node (c-use) or edge (p-use).

Consider the following program:

```java
1. input(x);
2. y=1;
```

### Notes

- **These notes are based upon notes by Prof. Robert Hierons, whom I gratefully acknowledge.**

### Example

```java
3. if (x>0)
4. x=x+1;
5. else x=-x;
6. x=x+1;
7. while (x>0) {
8. y=y*x;
9. x=x-1;
}
10. output(y);
```

Here 1, 4, 5, 6 and 9 are all definitions of $x$; 2 and 8 are definitions of $y$; 4, 5, 6, 8, and 9 are all c-uses of $x$; 3-4, 3-5, 7-8 and 7-10 are p-uses (as edges) of $x$. That is, 3 and 7 are p-uses (as nodes) of $x$. 8 and 10 are c-uses of $y$.

We are interested in how definitions of a variable may affect uses of this variable. We are thus interested in paths, from a definition of a variable $x$ to a use of $x$, in which the value of $x$ is not reassigned. This is captured by the following definition.

- A path $P$, containing the sequence $n_1, \ldots, n_k$ of nodes, in the CFG of a program $p$ is **definition clear** (def-clear) for variable $x$ if nodes $n_1, \ldots, n_k$ are not assignments to $x$.

Note that the path $P$ typically will not be from the start to the end of program $p$, but will be from some definition to some use. We call this a def path (a “definition-use” path).

In the example, 234678 is def-clear for $y$, 1234 is def-clear for $x$, but 12346 is not def-clear for $x$ since node 4 is an assignment to $x$. We can also consider paths that go from one iteration of the loop to another. For example, 978 and 9789 are def-clear for $x$ and 8978 is def-clear for $y$. In fact, a path from a definition to a use may appear to go ‘backwards’ in a loop. For example, there is a def-clear path for $x$ from the definition of $x$ at 9 to the use of $x$ at 8: the path 978 (this is a path in the CFG since execution can follow this route).

Note that the above definition allows the start and end nodes of path $P$ to be assignments to $x$: it is only the intermediate nodes that do not assign to $x$. The significance of this definition is that if we find a node $n_i$ that is a definition of $x$, a node $n_k$ that is a use of $x$, and a path $P$ from $n_i$ to $n_k$ that is definition clear for $x$, then any test that takes us through path $P$ will have tested a case in which the value assigned at $n_i$ is used at $n_k$.

By considering the data flow we can identify faults without executing the system: we might identify data flow anomalies. An example of such an anomaly is where there is a feasible path to a use of a variable $x$ that contains no definition for $x$. Many compilers will identify such anomalies. I give a more detailed list of anomalies in the lecture slides for this part of the course.

We are now ready to define some data-flow test criteria.
The all definitions criterion
Here we simply insist that every definition of a variable reaches a use of that variable for some test case. Clearly this is an extremely weak criterion. The slides which I used in the lecture show how weak this is.

The all p-uses criterion
This criterion looks at p-uses and the definitions which may propagate onto them. Specifically, given a p-use \( n_1 \) for a variable \( x \) we insist that:

- Suppose for definition of \( x \) at node \( n' \) there is a feasible def-clear path for \( x \) from \( n' \) to \( n_1 \). Then some test case must follow a def-clear path for \( x \) from \( n' \) to \( n_1 \).

Thus, we test all combinations of def/p-use for which there is a feasible def-clear path for the appropriate variable. However, we need not test using all such def-clear paths.

I give examples in the slides and tutorial.

All uses
The all-uses criterion takes the all p-uses criterion a step further: whenever a definition can propagate onto a use we must test this combination. Specifically, given a use \( m \) for a variable \( x \) we insist that:

- Suppose a definition of \( x \) at node \( n' \), if there is some feasible def-clear path for \( x \) from \( n' \) to \( m \). Then some test case must follow a def-clear path for \( x \) that starts at \( n' \) and ends at \( m \).

I give examples in the slides and tutorials.

All du-paths
The all-du-paths criterion is an extension of the all uses criterion. Rather than insisting that (at least) one feasible def-clear path from a definition to a use is executed, we insist that all such paths are executed. In the example, the set of paths achieves this. However, in general this criterion will require significantly more effort.

The du-paths criterion may be defined in the following way.

Given a use \( m \) for a variable \( x \) and a definition \( n' \) for \( x \) we insist that:

- all feasible def-clear paths for \( x \) that start at \( n' \) and end at \( m \) are executed in testing.

In the slides I give examples of these criteria and some other variations on the theme of dataflow testing. You will find the BCS testing standard http://www.testingstandards.co.uk/ to be a very useful source of additional notes. In addition to this, if you want to go back to the original definitions of data flow testing, I cite the 1988 paper by Frankl and Weyuker at the end of the slides. This was a very seminal paper by two women who have made an very strong and consistent contribution to a number of problems in software testing and who continue to work in this area today.

You should be aware that, as the idea of data flow testing has matured, there are some slight variations on the theme and some mild variations in the precise definitions of data flow testing criteria. Naturally, those you will be examined on will be those you have been taught in this course, so, when reading other sources, be aware that there may be slight changes of definition. The primary example is with the definition of a p-use. In these notes and the slides which accompany them, I have simplified this by considering a p-use to be a node, rather than an edge. The definitions (which you might see) which require a p-use to be an edge lead to a more demanding test criterion.