Data-flow Testing

1: `read(x, y)`

2: `x := x + 2;`

3: `y := 2;`

4: `y := y * 2;`

5: `x := x + 2;`

6: `x := y + 2;`

7: `x := y + 2;`

8: `x := x + y + 2;`
Data-flow Testing

1: read(x, y)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: read(x, y)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: \text{read}(x, y)

2: \text{ } x := x + 2;

3: \text{ } y := 2;

4: \text{ } y := y * 2;

5: \text{ } x := x + 2;

6: \text{ } x := y + 2;

7: \text{ } x := y + 2;

8: \text{ } x := x + y + 2;
Data-flow Testing

1: read(x, y)

2: x := x + 2;
3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: read(x, y)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: read(x, y)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: \texttt{read}(x, y)

2: \texttt{x := x + 2;}

3: \texttt{y := 2};

4: \texttt{y := y * 2;}

5: \texttt{x := x + 2;}

6: \texttt{x := y + 2;}

7: \texttt{x := y + 2;}

8: \texttt{x := x + y + 2;}

Data-flow Testing

read(x, y)

1: read(x, y)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: \text{read}(x, y)

2: \text{\textcolor{red}{x := x + 2;}}

3: y := 2;

4: \text{\textcolor{red}{y := y * 2;}}

5: x := x * 2;

6: \text{\textcolor{red}{x := y + 2;}}

7: \text{\textcolor{red}{x := y + 2;}}

8: \text{\textcolor{red}{x := x + y + 2;}}
Data-flow Testing

1: read(x, y)

2: x := x + 2; y := 2;
3: y := y * 2; x := x + y + 2;
4: y := y * 2;
5: x := x + 2;
6: x := y + 2;
7: x := y + 2;
8: x := x + y + 2;
Data-flow Testing

1: \texttt{read}(x, y)

2: \texttt{x := x + 2;}

3: \texttt{y := 2;}

4: \texttt{y := y * 2;}

5: \texttt{x := x + 2;}

6: \texttt{x := y + 2;}

7: \texttt{x := y + 2;}

8: \texttt{x := x + y + 2;}
Data-flow Testing

read(x, y)
x := x + 2; y := 2;
x := x + 2;
x := y + 2;

1:
2:
x := x + 2;
3:
y := 2;
4:
y := y * 2;
5:
x := x + 2;
6:
x := y + 2;
7:
x := y + 2;
8:
x := x + y + 2;
Data-flow Testing

1: \( \text{read}(x, y) \)

2: \( x := x + 2; \)

3: \( y = 2; \)

4: \( y := y * 2; \)

5: \( x := x + 2; \)

6: \( x := y + 2; \)

7: \( x := x + 2; \)

8: \( x := x + y + 2; \)
Data-flow Testing

1: \textit{read}(x, y)

2: \textit{x} := x + 2;

3: \textit{y} := 2;

4: y := y \times 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + 2;

8: x := x + y + 2;
Data-flow Testing

1: read\((x, y)\)
2: \(x := x + 2;\)
3: \(y = 2;\)
4: \(y := y * 2;\)
5: \(x := x + 2;\)
6: \(x := y + 2;\)
7: \(x := y + 2;\)
8: \(x := x + y + 2;\)
Data-flow Testing

1: \text{read}(x, y)

2: \text{\texttt{x := x + 2;}}

3: \text{\texttt{y := 2;}}

4: \text{\texttt{y := y * 2;}}

5: \text{\texttt{x := x + 2;}}

6: \text{\texttt{x := y + 2;}}

7: \text{\texttt{x := y + 2;}}

8: \text{\texttt{x := x + y + 2;}}
Data-flow Testing

1: \texttt{read}(x, y)

2: \texttt{x := x + 2; y := 2;}

3: \texttt{y := y + 2;}

4: \texttt{y := y * 2;}

5: \texttt{x := x + 2;}

6: \texttt{x := y + 2;}

7: \texttt{x := y + 2;}

8: \texttt{x := x + y + 2;}

9: \texttt{...}
Data-flow Testing

1: \texttt{read}(x, y)

2: \texttt{x} := \texttt{x} + 2;

3: \texttt{y} := 2;

4: \texttt{y} := \texttt{y} * 2;

5: \texttt{x} := \texttt{x} + 2;

6: \texttt{x} := \texttt{y} + 2;

7: \texttt{x} := \texttt{x} + 2;

8: \texttt{x} := \texttt{x} + 2;
Data-flow Testing

1: read(x, y)

2: x := x + 2; y := 2;

3: x := y + 2;

4: y := y * 2;

5: x := x + y + 2;

6: x := y + 2;

7: x := x + 2;

8: x := x + y + 2;
Data-flow Testing

1: `read(x, y)`

2: `x := x + 2;`

3: `y := 2;`

4: `y := y * 2;`

5: `x := x + 2;`

6: `x := y + 2;`

7: `x := x + 2;`

8: `x := x + 2;`
Data-flow Testing

1: `read(x, y)`

2: `x := x + 2; y := 2;`

3: `x := y + 2;`

4: `y := y * 2;`

5: `x := x + 2;`

6: `x := y + 2;`

7: `x := y + 2;`

8: `x := x + y + 2;`
A set $P$ of execution paths satisfies the all-definitions criterion iff
- for all definition occurrences of a variable $x$ such that
  - there is a use of $x$, which is feasibly reachable from that definition,
  - there is at least one path $p$ in $P$ such that
    - $p$ includes a subpath through which the definition of $x$ reaches some use occurrence of $x$
A set $P$ of execution paths satisfies the all-definitions criterion iff

- for all definition occurrences of a variable $x$ such that
  - there is a use of $x$, which is feasibly reachable from that definition,
  - there is at least one path $p$ in $P$ such that
    - $p$ includes a subpath through which the definition of $x$ reaches some use occurrence of $x$
A set $P$ of execution paths satisfies the all-definitions criterion iff:

- For all definition occurrences of a variable $x$ such that there is a use of $x$, which is feasibly reachable from that definition,
- There is at least one path $p$ in $P$ such that $p$ includes a subpath through which the definition of $x$ reaches some use occurrence of $x$
All Uses Criterion

• A set $P$ of execution paths satisfies the all-uses criterion iff
  - for all definition occurrences of a variable $x$ and all use occurrences of $x$,
    • that the definition feasibly reaches,
  - there is at least one path $p$ in $P$ such that
    • $p$ includes a subpath through which that definition reaches the use.

1: $\text{read}(x, y, z)$
2: $x := x + 2$
3: $y := 2$
4: $y := y \times 2$
5: $x := x + 2$
6: $x := y + 2$
7: $x := y + z + 2$
8: $x := x + y + 2$
A set $P$ of execution paths satisfies the all-uses criterion iff:

- for all definition occurrences of a variable $x$ and all use occurrences of $x$,
  - that the definition feasibly reaches,
  - there is at least one path $p$ in $P$ such that
    - $p$ includes a subpath through which that definition reaches the use.
All Uses Criterion

• A set $P$ of execution paths satisfies the all-uses criterion iff
  - for all definition occurrences of a variable $x$ and all use occurrences of $x$,
    • that the definition feasibly reaches,
  - there is at least one path $p$ in $P$ such that
    • $p$ includes a subpath through which that definition reaches the use.
A set $P$ of execution paths satisfies the all-uses criterion iff:

- for all definition occurrences of a variable $x$ and all use occurrences of $x$,
  - that the definition feasibly reaches,
  - there is at least one path $p$ in $P$ such that
    - $p$ includes a subpath through which that definition reaches the use.
1: read(x, y, z)

2: x := x + 2;

3: y := 2;

4: y := y * 2;

5: x := x + 2;

6: x := y + 2;

7: x := y + z + 2;

8: x := x + y + 2;
read(x, y, z)

1: read(x, y, z)
2: x := x + 2;
3: y := 2;
4: y := y * 2;
5: x := x + 2;
6: x := y + 2;
7: x := y + z + 2;
8: x := x + y + 2;
All Uses Criterion

1: `read(x, y, z)`

2: `x := x + 2;`

3: `y := 2;`

4: `y := y * 2;`

5: `x := x + 2;`

6: `x := y + 2;`

7: `x := y + z + 2;`

8: `x := x + y + 2;`
All DU-paths criterion

- A set $P$ of execution paths satisfies the all-DU paths criterion iff
  - for all definitions of a variable $x$ and all paths $q$ through which that definition reaches a use of $x$,
  - there is at least one path $p$ in $P$ such that
    - $q$ is a subpath of $p$ and $q$ is cycle-free
An Applicable Family of Data Flow Testing Criteria

• Assumptions about the program
  - **No**
    • goto statements
    • with
    • variant records
    • Functions having ‘var’ parameters
      - By reference
    • Procedural or functional parameters
    • Conformant arrays
  - Every boolean expression that determines the flow of control has at least one occurrence of a variable or a call to the function ‘eof’ or ‘eoln’
Program Structure

- Program consists of 'blocks'
- Block
  - Sequence of statements
    - Whenever the first statement is executed, the remaining statements in the block are executed in the given order
- Can be represented by a flow graph
Classifying each variable occurrence

- **Definition**
  - Value is stored in a memory location

- **Use**
  - Value is fetched from a memory location

- **Undefinition**
  - Value and location becomes unbound

- **C-use**
  - Use in a computation or output statement
  - Associated with each node

- **P-use**
  - Use in a predicate
  - Associated with each edge
Simple Statements

Assignment statement: \( v := expr; \)

Node \( i \) has c-uses of each variable in \( expr \) followed by a definition of \( v \).
Simple Statements

Input/Output statements:

read(v1,...,vn);
readln(v1,...,vn);
read(f,v1,...,vn);
readln(f,v1,...,vn);

Node i has definitions of v1,...,vn.
If the file variable f is present then node i also has a c-use followed by a definition of ft.

write(e1,...,en);
writeln(e1,...,en);
write(f,e1,...,en);
writeln(f,e1,...,en);

Node i has c-uses of each variable occurring in e1,...,en.
If the file variable f is present then node i also has a definition followed by a c-use of ft.
Simple Statements

Procedure call: $P(e_1,...,e_n)$;

Node $j$ has $c$-uses of each variable occurring in the expressions $e_1,...,e_n$.
These are followed by definitions of each actual parameter which corresponds to a `var` formal parameter.

Nodes $i$ and $k$ are included to assure that the procedure call has its own node.
Repetitive Statements

while statement: while B do S;

Let h be the entry node to subgraph S. Edges (i,h) and (i,j) have p-uses of each variable in the boolean expression B.
Repetitive Statements

for statement:
for v := e1 to e2 do S;
for v := e1 downto e2 do S;

Let tmp be a new variable.
Let f and g be the entry
and exit nodes, respectively,
of S. Node h has c-uses of
each variable in e1,
followed by a definition of v and
c-uses of each variable in e2
followed by a definition of tmp.
Edges (i,f) and (i,j) have
p-uses of v and tmp. Node g has
a c-use followed by a def of v.
Repetitive Statements

repeat statement:
 repeat $S_1;...;S_n$ until $B$;

Let $j$ be the entry node of $S_1$, and let $k$ be the exit node of $S_n$. Edges $(k,j)$ and $(k,i)$ have p-uses of each variable in the boolean expression $B$. 
Conditional Statements

if-then-else statement

if B then S1;
if B then S1 else S2;

Let k and j be the entry nodes of S1 and S2, respectively.
Edges (i,j) and (i,k) have p-uses of each variable in the boolean expression B.
if there is no “else” part then subgraph S2 has a single node corresponding to an empty block.
Conditional Statements

case $e_1$ of
  $label$-list1 : $S_1$;
  
  
  $label$-listn : $S_n$
end;

Let $j_1, \ldots, j_n$ be the entry nodes of $S_1, \ldots, S_n$, respectively. Edges $(i, j_1), \ldots, (i, j_n)$ have $p$-uses of each variable in the expression $e_1$. 
Entry and exit nodes

- **Entry node**
  - Has the definition of
    - Each parameter
    - Each non-local variable that is used in the program
    - Input buffer input↑

- **Exit node has**
  - An undefined of each local variable
  - A c-use of each variable parameter
  - A c-use of each non-local variable
  - A c-use of the input buffer input↑
Arrays

- It is impossible to determine the particular array element which is being used or defined in an occurrence of an array variable
  - $A[i+j]$
- Definition of $a[expr]$
  - A c-use of each variable in $expr$
  - Followed by a definition of $a$
- Use of $a[expr]$
  - c-uses of all the variables in $expr$
  - Followed by a use of $a$
Pointers

• Impossible to determine statically the memory location to which a pointer points
• Syntactic treatment
• If $p$ is a pointer variable
  - Definition of $p$
    • C-use of $p$
    • Followed by a definition of $p$
  - Use of $p$
    • C-use of $p$
    • Followed by a c-use of $p$
Records & Files

• Records
  - Each field is treated as an individual variable
  - Any unqualified occurrence of a record is treated as an occurrence of each field

• File variables
  - Considering the effect on the file buffer
Global Definition

• **Global c-use**
  - A c-use of x in node i is global if x has been assigned in some block other than i

• **Def-clear path wrt x**
  - A path \((i, n_1, n_2, …, n_m, j)\) containing no definitions or undefinitions of x in nodes \(n_1, n_2, …, n_m\)

• **Global definition of x**
  - A node i has a global definition of a variable x if
    - it has a definition of x and
    - there is a def-clear path wrt x from node i to some node containing
      - a global c-use or
      - edge containing a p-use of x
Restricted Programs Class

- Satisfying the following properties
  - **NSUP**
    - No-syntactic-undefined-p-use Property
      - For every p-use of a variable x on an edge (i,j), in P, there is some path from the start node to edge (i,j), which contains a global definition of x
  - **NSL**
    - Non-straight-line property
      - P has at least one conditional or repetitive statement
        » At least one node in P’s flow-graph has more than one successor
        » At least one variable has a p-use in P
Def-use graph

• Obtained from the flow graph
• Associate with each node the sets
  – \( C\text{-use}(i) \)
    • Variables which have global c-uses in block-\( i \)
  – \( \text{Def}(I) \)
    • Variables which have global definitions in block-\( i \)
• Associate with each edge (\( i,j \))
  – \( P\text{-use}(i,j) \)
    • Variables which have p-uses on edge (\( i,j \))
• Define sets of nodes
  – \( \text{dcu}(x,i) \)
    • Nodes \( j \) such that \( x \in c\text{-use}(j) \) and there is a def-clear paths with respect to \( x \) from \( i \) to \( j \)
  – \( \text{dpu}(x,i) \)
    • Edges (\( j,k \)) such that \( x \in p\text{-use}(j,k) \) and there is a def-clear path with respect to \( x \) from \( i \) to (\( j,k \))
Definitions for def-use graph

\[ V = \text{the set of variables} \]
\[ N = \text{the set of nodes} \]
\[ E = \text{the set of edges} \]
\[ \text{def}(i) = \{ x \in V \mid x \text{ has a global definition in block } i \} \]
\[ \text{c-use}(i) = \{ x \in V \mid x \text{ has a global c-use in block } i \} \]
\[ \text{p-use}(i,j) = \{ x \in V \mid x \text{ has a p-use in edge } (i,j) \} \]
\[ \text{dcu}(x,i) = \{ j \in N \mid x \in \text{c-use}(j) \text{ and there is a def-clear path wrt } x \text{ from } i \text{ to } j \} \]
\[ \text{dpu}(x,i) = \{ (j,k) \in E \mid x \in \text{p-use}(j,k) \text{ and there is a def-clear path wrt } x \text{ from } i \text{ to } (j,k) \} \]
Explanation

• If \( x \in \text{def}(i) \) and \( j \in \text{dcu}(x,i) \), then
  - \( x \) has a global definition in node \( i \) and
  - A c-use in node \( j \), and
  - There is a definition clear path with respect to \( x \) from node \( i \) to node \( j \)

• Hence
  - It may be possible for control to reach node \( j \) with the variable \( x \) having the value which was assigned to it in node \( i \)
More definitions

- **Definition-c-use association**
  - Triple \((i,j,x)\) where \(i\) is a node containing a global definition of \(x\) and \(j \in \text{dcu}(x,i)\)

- **Definition-p-use association**
  - Triple \((i,(j,k),x)\) where \(i\) is a node containing a global definition of \(x\) and \((j,k) \in \text{dpu}(x,i)\)

- A path \((n_1,n_2, \ldots, n_j,n_k)\) is a du-path wrt \(x\) if \(n_1\) has a global definition of \(x\) and either
  - \(n_k\) has a global c-use of \(x\) and \((n_1, \ldots, n_j, n_k)\) is a def-clear simple path wrt \(x\), and
  - \((n_j, n_k)\) has a p-use of \(x\) and \((n_1, \ldots, n_j)\) is a def-clear loop-free path wrt \(x\)

- An association is a definition-c-use association, a definition-p-use association, or a du-path
Yet more definitions

- **Complete path**
  - Path from the entry node to the exit node

- **Covering**
  - A complete path $\pi$ covers a definition-c-use association $(i, j, x)$ if it has a definition clear subpath wrt $x$ from $i$ to $j$
  - A complete path $\pi$ covers a definition-p-use association $(i, (j, k), x)$ if it has a definition clear subpath wrt $x$ from $i$ to $(j, k)$
  - $\pi$ covers a du-path $\pi'$ if $\pi'$ is a subpath of $\pi$
  - The set $\Pi$ of paths covers an association if some element of the set does
  - A test set $T$ covers an association if the elements of $T$ cause the execution of the set of paths $\Pi$, and $\Pi$ covers the association
Finally, the criteria

• Intuitively
  - The family of DF testing criteria is based on requiring that
    • the test data execute definition-clear paths from each node containing a global definition of a variable to specified nodes containing
      - global c-uses and
      - edges containing p-uses of that variable
  - For each variable definition, the criteria require that
    • All/some definition-clear paths wrt that variable from the node containing the definition to all/some of the uses/c-uses/p-uses reachable by some such paths be executed
All-defs criterion

- If variable $x$ has a global definition in node $i$, the all-defs criterion requires the test data to exercise some path which goes from $i$ to some node or edge at which the value assigned to $x$ in node $i$ is used.
All-uses criterion

- If variable \( x \) has a global definition in node \( i \), the all-uses criterion requires the test data to exercise at least one path which goes from \( i \) to each node and edge at which the value assigned to \( x \) in node \( i \) is used.
All-DU-paths criterion

- If variable $x$ has a global definition in node $i$, the all-DU-paths criterion requires the test data to exercise all paths which go from $i$ to each node and edge at which the value assigned to $x$ in node $i$ is used.
Other DF testing criteria

- All-p-uses
- All-c-uses
- All-p-uses/some-c-uses
- All-c-uses/some-p-uses
## Definitions of DF criteria

<table>
<thead>
<tr>
<th>CRITERION</th>
<th>ASSOCIATIONS REQUIRED</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-defs</td>
<td>Some ((i,j,x)) s.t. (j \in dcu(x,i)) or some ((i,(j,k),x)) s.t. ((j,k) \in dpu(x,i)).</td>
</tr>
<tr>
<td>All-c-uses</td>
<td>All ((i,j,x)) s.t. (j \in dcu(x,i)).</td>
</tr>
<tr>
<td>All-p-uses</td>
<td>All ((i,(j,k),x)) s.t. ((j,k) \in dpu(x,i)).</td>
</tr>
<tr>
<td>All-p-uses/some-c-uses</td>
<td>All ((i,(j,k),x)) s.t. ((j,k) \in dpu(x,i)). In addition, if (dpu(x,i)=\emptyset) then some ((i,j,x)) s.t. (j \in dcu(x,i)). Note that since (i) has a global definition of (x), (dpu(x,i)=\emptyset \Rightarrow dcu(x,i) \neq \emptyset).</td>
</tr>
<tr>
<td>All-c-uses/some-p-uses</td>
<td>All ((i,j,x)) s.t. (j \in dcu(x,i)). In addition, if (dcu(x,i)=\emptyset) then some ((i,(j,k),x)) s.t. ((j,k) \in dcu(x,i)). Note that since (i) has a global definition of (x), (dcu(x,i)=\emptyset \Rightarrow dpu(x,i) \neq \emptyset).</td>
</tr>
<tr>
<td>All-uses</td>
<td>All ((i,j,x)) s.t. (j \in dcu(x,i)) and all ((i,(j,k),x)) s.t. ((j,k) \in dpu(x,i)).</td>
</tr>
<tr>
<td>All-du-paths</td>
<td>All du-paths from (i) to (j) with respect to (x) for each (j \in dcu(x,i)) and all du-paths from (i) to ((j,k)) with respect to (x) for each ((j,k) \in dpu(x,i)).</td>
</tr>
</tbody>
</table>
“includes”

• **Criterion** $C_1$ *includes* criterion $C_2$ *iff*
  - For every subprogram, any test set that satisfies $C_1$ also satisfies $C_2$

• $C_1$ *strictly includes* $C_2$, *iff*
  - denoted $C_1 \Rightarrow C_2$,
  - $C_1$ includes $C_2$ and for some subprogram $P$ there is a test set that satisfies $C_2$ but does not satisfy $C_1$
Includes relationship

ALL-PATHS
  └── ALL-DU-PATHS
      └── ALL-USES
          ├── ALL-C-USES/SOME-P-USES
          │    └── ALL-C-USES
          │          └── ALL-DEFs
          │            └── ALL-P-USES
          │                └── ALL-EDGES
          │                        └── ALL-NODES
          └── ALL-P-USES/SOME-C-USES
                └── ALL-P-USES
Applicability

• It may be the case that no test set for program P satisfies criterion C
  - Infeasible paths
• Tailor the DF criteria so that they are applicable
• Assumptions
  - All aliases are known
  - All side effects are known
  - No element of the test set causes the program to crash
    • Execution of entry node to exit node
Executable/Feasible Paths

- **Recall**
  - Complete path
    - Path from the entry node to the exit node

- **Executable/feasible complete path**
  - A complete path that is executed on some assignment of values to input variables

- **Executable/feasible path**
  - A subpath of an executable complete path
Recall Definition

• **Definition-c-use association**
  - Triple \((i,j,x)\) where \(i\) is a node containing a global definition of \(x\) and \(j \in \text{dcu}(x,i)\)

• **Definition-p-use association**
  - Triple \((i,(j,k),x)\) where \(i\) is a node containing a global definition of \(x\) and \((j,k) \in \text{dpu}(x,i)\)

• A path \((n_1,n_2, ..., n_j,n_k)\) is a du-path wrt \(x\) if \(n_1\) has a global definition of \(x\) and either
  - \(n_k\) has a global c-use of \(x\) and \((n_1, ..., n_j, n_k)\) is a def-clear simple path wrt \(x\), and
  - \((n_j, n_k)\) has a p-use of \(x\) and \((n_1, .., n_j)\) is a def-clear loop-free path wrt \(x\)

• An association is a definition-c-use association, a definition-p-use association, or a du-path
Executable Associations

• Definition
  - An association is executable if there is some executable complete path that covers it; otherwise it is unexecutable

• $\text{fdcu}(x,i) \in \text{dcu}(x,i)$
  - Nodes $j$ such that $x \in \text{c-use}(j)$ and there is an executable definition clear path wrt $x$ from $i$ to $j$

• $\text{fdpu}(x,i) \in \text{dpu}(x,i)$
  - Edges $(j,k)$ such that $x \in \text{p-use}(j,k)$ and there is an executable definition clear path wrt $x$ from $i$ to $(j,k)$
Equivalently

- \( f_{dcu}(x,i) = \{ j \in dcu(x,i) \mid \text{the association } (i,j,k) \text{ is executable} \} \)
- \( f_{dpu}(x,i) = \{ (j,k) \in dpu(x,i) \mid \text{the association } (i,(j,k),x) \text{ is executable} \} \)

Intuitively

- new criterion \( C^* \) for each DF criterion \( C \)
- By selecting the required associations from \( f_{dcu}(x,i) \) and \( f_{dpu}(x,i) \) instead of from \( dcu(x,i) \) and \( dpu(x,i) \)
### Feasible Data-flow Criteria (FDF)

<table>
<thead>
<tr>
<th>CRITERION</th>
<th>REQUIRED ASSOCIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(all-defs)*</td>
<td>if ( \text{fdcu}(x,i) \cup \text{fdpu}(x,i) \neq \emptyset ) then some ((i,j,x)) s.t. (j \in \text{fdcu}(x,i)) or some ((i,j,k,x)) s.t. ((j,k) \in \text{fdpu}(x,i)).</td>
</tr>
<tr>
<td>(all-c-uses)*</td>
<td>all ((i,j,x)) s.t. (j \in \text{fdcu}(x,i)).</td>
</tr>
<tr>
<td>(all-p-uses)*</td>
<td>all ((i,j,k,x)) s.t. ((j,k) \in \text{fdpu}(x,i)).</td>
</tr>
<tr>
<td>(all-p-uses/some-c-uses)*</td>
<td>all ((i,j,k,x)) s.t. ((j,k) \in \text{fdpu}(x,i)). In addition, if (\text{fdpu}(x,i) = \emptyset) and (\text{fdcu}(x,i) \neq \emptyset) then some ((i,j,x)) s.t. (j \in \text{fdcu}(x,i)).</td>
</tr>
<tr>
<td>(all-c-uses/some-p-uses)*</td>
<td>all ((i,j,x)) s.t. (j \in \text{fdcu}(x,i)). In addition, if (\text{fdcu}(x,i) = \emptyset) and (\text{fdpu}(x,i) \neq \emptyset) then some ((i,j,k,x)) s.t. ((j,k) \in \text{fdpu}(x,i)).</td>
</tr>
<tr>
<td>(all-uses)*</td>
<td>all ((i,j,x)) s.t. (j \in \text{fdcu}(x,i)) and all ((i,j,k,x)) s.t. ((j,k) \in \text{fdpu}(x,i)).</td>
</tr>
<tr>
<td>(all-du-paths)*</td>
<td>all executable du-paths with respect to (x) from (i) to (j) s.t. (j \in \text{dcu}(x,i)) and all executable du-paths with respect to (x) from (i) to ((j,k)) for each ((j,k) \in \text{dpu}(x,i)).</td>
</tr>
</tbody>
</table>
Includes Relationships

(ALL-PATHS)*

(ALL-DU-PATHS)*  (ALL-EDGES)*

(ALL-USES)*  (ALL-NODES)*

(ALL-C-USES/SOME-P-USES)*  (ALL-P-USES/SOME-C-USES)*

(ALL-C-USES)*  (ALL-DEFs)*  (ALL-P-USES)*
Why the different relationships

Example Program

1. read(x, y);
   i := 1;

2. i := 1;

3. i > 2
   i <= 2

4. writeln('hello');
   i := i + 1;

5. y < 0
   y >= 0

6. writeln(x);
# The Program's DU-paths

<table>
<thead>
<tr>
<th>Path</th>
<th>With respect to</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>input↑</td>
<td>yes</td>
</tr>
<tr>
<td>(2,3,4)</td>
<td>i</td>
<td>yes</td>
</tr>
<tr>
<td>(2,3,5)</td>
<td>i</td>
<td>no</td>
</tr>
<tr>
<td>(4,3,4)</td>
<td>i</td>
<td>yes</td>
</tr>
<tr>
<td>(4,3,5)</td>
<td>i</td>
<td>yes</td>
</tr>
<tr>
<td>(2,3,5,6,7,9,10)</td>
<td>input↑</td>
<td>no</td>
</tr>
<tr>
<td>(2,3,5,6,8,9,10)</td>
<td>input↑</td>
<td>no</td>
</tr>
</tbody>
</table>
Why the different relationships

Let \( x = X \) (any integer)
And \( y = Y < 0 \)

Path executed is
\{1,2,3,4,3,4,3,5,6,7,9,10\}

Are all DU-paths shown earlier covered?
YES

But the associations \( (2, (6,8), y) \) and \( (2,8,x) \) are not!
And they are executable by a test case that causes the execution of \{1,2,3,4,3,4,3,5,6,8,9,10\}
Hence \((\text{all-du-paths})* \nsat \nsat ((\text{all-p-uses})*, (\text{all-u-uses})*, etc.)\)
Interprocedural DF Testing

- Most DF testing methodologies deal with dependencies that exist within a procedure (i.e., *intraprocedural*)
- Data dependencies also exist among procedures
- Requires analysis of the flow of data across procedure boundaries
- Calls and Returns
  - Direct dependencies (single call/return)
  - Indirect dependencies (multiple calls/returns)
module Main
declare
    S: an array 1...N of integer;
    I, MAX, MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(1,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F, L: integer;
    MX: reference integer;
declare M1, M2, MD: integer;
begin
    if F+1=L then PairMax(S[F],S[L],MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F,MD,M1);
        GetMax(MD+1,L,M2);
        PairMax(M1,M2,MX);
    endif;
end;

procedure PairMax;
input I,J,K: reference integer;
begi
    if I>J then K := I
    else K := J;
end;
module Main
declare
    S: an array 1...N of integer;
    I, MAX, MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(1,N,MAX);
    write(MAX);
end;

procedure GetMax:
input
    F, L: integer;
    MX: reference integer;
declare
    M1, M2, MD: integer;
begin
    if F + 1 = L then PairMax(S[F], S[L], MX)
    else begin
        MD := (F + L) DIV 2;
        GetMax(F, MD, M1);
        GetMax(MD + 1, L, M2);
        PairMax(M1, M2, MX);
    end;
end;

procedure PairMax:
input
    I, J, K: reference integer;
begin
    if I > J then K := I
    else K := J;
end;
module Main
declare
   S: an array 1...N of integer;
   I,MAX,MN: integer;
begin
   for I := 1 to N do read(S[I]);
   GetMax(I,N,MAX);
   write(MAX);
end;

procedure GetMax;
input
   F,L: integer;
   MX: reference integer;
declare M1,M2,MD: integer;
begin
   if F+1=L then PairMax(S[F],S[L],MX)
   else begin
      MD := (F+L) DIV 2;
      GetMax(F,MD,M1);
      GetMax(MD+1,L,M2);
      PairMax(M1,M2, MX);
   endif;
end;

procedure PairMax;
input L,J,K: reference integer;
begin
   if I>J then K := I
   else K := J;
end;

param(F,L,MX)

F+1=L?

T

F

B1

B2

B4

B5

B6

B7

B8

MD:=(F+L)/2

GetMax(F,MD,M1)

GetMax(MD+1,L,M2)

PairMax(M1,M2, MX)

return(MX)
module Main
declare
    S: an array 1...N of integer;
    I, MAX, MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(1,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F, L: integer;
    MX: reference integer;
declare M1, M2, MD: integer;
begin
    if F+1 = L then PairMax(S[F],S[L],MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F,MD,M1);
        GetMax(MD+1,L,M2);
        PairMax(M1,M2,MAX);
    endif;
end;

procedure PairMax;
input I, J, K: reference integer;
begin
    if I > J then K := I
    else K := J;
end;
module Main
declare
    S: an array 1...N of integer;
    I,MAX,MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(I,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F,L: integer;
    MX: reference integer;
declare M1,M2,MD: integer;
begin
    if F+1=L then PairMax(S[F],S[L],MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F,MD,M1);
        GetMax(MD+1,L,M2);
        PairMax(M1,M2, MX);
    end;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
    if I>J then K := I
    else K := J;
end;

B1 param(F,L, MX)
B2 F+1=L?
T
B3 PairMax(S[F],S[L], MX)
B4 MD := (F+L)/2
F
B5 GetMax(F,MD, M1)
B6 GetMax(MD+1,L, M2)
B7 PairMax(M1,M2, MX)
B8 return(MX)
module Main
declare
    S: an array 1..N of integer;
    I,MAX,MN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(I,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F,L: integer;
    MX: reference integer;
declare M1,M2,MD: integer;
begin
    if F+1=L then PairMax(S[F],S[L],MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F,MD,M1);
        GetMax(MD+1,L,M2);
        PairMax(M1,M2,MX);
    end;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
    if I>J then K := I
    else K := J;
end;
module Main
declare
  S: an array 1...N of integer;
  I, MAX, MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(1,N,MAX);
  write(MAX);
end;

procedure GetMax;
  input F,L: integer;
  MX: reference integer;
  declare M1,M2, MD: integer;
begin
  if F+1=L then PairMax(S[F],S[L],MX)
  else begin
    MD := (F+L) DIV 2;
    GetMax(F,MD,M1);
    GetMax(MD+1,L,M2);
    PairMax(M1,M2,MAX);
  end;
end;

procedure PairMax;
  input I,J,K: reference integer;
begin
  if I>J then K := I
  else K := J;
end;

param(F,L,MAX)
F+1=L?
T
B3
PairMax(S[F],S[L],MX)
B4
MD:=(F+L)/2
B5
GetMax(F,MD,M1)
B6
GetMax(MD+1,L,M2)
B7
PairMax(M1,M2,MAX)
return(MAX)
module Main
declare
    S: an array 1..N of integer;
    I, MAX, MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(1,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F, L: integer;
    MX: reference integer;
declare M1, M2, MD: integer;
begin
    if F+1=L then PairMax(S[F], S[L], MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F, MD, M1);
        GetMax(MD+1, L, M2);
        PairMax(M1, M2, MX);
    end;
end;

procedure PairMax;
input I, J, K: reference integer;
begin
    if I>J then K := I
    else K := J;
end;

Actual parameters at the call site that are bound to formal reference parameters in called procedures
module Main
declare
  S: an array 1..N of integer;
  I, MAX, MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(1,N,MAX);
  write(MAX);
end;

procedure GetMax;
input
  F, L: integer;
  MX: reference integer;
declare M1, M2, MD: integer;
begin
  if F + 1 = L then PairMax(S[F], S[L], MX)
  else begin
    MD := (F + L) DIV 2;
    GetMax(F, MD, M1);
    GetMax(MD + 1, L, M2);
    PairMax(M1, M2, MX);
  end;
end;

procedure PairMax;
input I, J, K: reference integer;
begin
  if I > J then K := I
  else K := J;
end;

begin
  param(F, L, MX)
  F + 1 = L?
  T
  F
  B3
  B4
  MD := (F + L) / 2
  GetMax(F, MD, M1)
  B5
  GetMax(MD + 1, L, M2)
  B6
  B8
  return(MX)
module Main
declare
    S: an array 1...N of integer;
    I, MAX, MIN: integer;
begin
    for I := 1 to N do read(S[I]);
    GetMax(1,N,MAX);
    write(MAX);
end;

procedure GetMax;
input
    F, L: integer;
    MX: reference integer;
declare M1, M2, MD: integer;
begin
    if F+1=L then PairMax(S[F],S[L],MX)
    else begin
        MD := (F+L) DIV 2;
        GetMax(F,MD,M1);
        GetMax(MD+1,L,M2);
        PairMax(M1,M2, MX);
    end;
end;

procedure PairMax;
input I, J, K: reference integer;
begin
    if I>J then K := I
    else K := J;
end;

param(F, L, MX)

B1
F+1=L?

B2
T
F

B3 PairMax(S[F],S[L],MX)

B4 MD:=(F+L)/2

B5 GetMax(F,MD,M1)

B6 GetMax(MD+1,L,M2)

B7 PairMax(M1,M2, MX)

B8 return(MX)
module Main
declare
   S: an array 1...N of integer;
   I,MAX,MIN: integer;
begin
   for I := 1 to N do read(S[I]);
   GetMax(1,N,MAX);
   write(MAX);
end;

procedure GetMax;
input
   F,L: integer;
   MX: reference integer;
declare M1,M2,MD: integer;
begin
   if F+1=L then PairMax(S[F],S[L],MX)
   else begin
      MD := (F+L) DIV 2;
      GetMax(F,MD,M1);
      GetMax(MD+1,L,M2);
      PairMax(M1,M2,MX);
   endif;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
   if I>J then K := I
   else K := J;
end;
A test case

\[ S = \{3, 5, 1, 6\} \]

\[ F = 1 \]

\[ L = 4 \]
A test case

\[ S = \{3, 5, 1, 6\} \]

\[ F = 1 \]

\[ L = 4 \]
A test case

$S = \{3, 5, 1, 6\}$

$F = 1$

$L = 4$

All def-use pairs are covered
module Main
declare
  S: an array 1..N of integer;
  I,MAX,MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(I,N,MAX);
  write(MAX);
end;

procedure GetMax;
input
  F,L: integer;
  MX: reference integer;
declare M1,M2,MD: integer;
begin
  if F+1=L then PairMax(S[F],S[L],MX)
  else begin
    MD := (F+L) DIV 2;
    GetMax(F,MD,M1);
    GetMax(MD+1,L,M2);
    PairMax(M1,M2,MAX);
  end;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
  if I>J then K := I
  else K := J;
end;
Any missed def-uses?

module Main
declare
  S: an array 1..N of integer;
  I, MAX, MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(1, N, MAX);
  write(MAX);
end;

procedure GetMax;
input
  F, L: integer;
  MX: reference integer;
declare M1, M2, MD: integer;
begin
  if F + 1 = L then PairMax(S[F], S[L], MX)
  else begin
    MD := (F + L) DIV 2;
    GetMax(F, MD, M1);
    GetMax(MD + 1, L, M2);
    PairMax(M1, M2, MX);
  end;
end;

procedure PairMax;
input
  I, J, K: reference integer;
begin
  if I > J then K := I
  else K := J;
end;
module Main
declare
  S: an array 1..N of integer;
  I,MAX,MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(1,N,MAX);
  write(MAX);
end;

procedure GetMax;
input
  F,L: integer;
  MX: reference integer;
declare M1,M2,MD: integer;
begin
  if F+1=L then PairMax(S[F],S[L],MX)
  else begin
    MD := (F+L) DIV 2;
    GetMax(F,MD,M1);
    GetMax(MD+1,L,M2);
    PairMax(M1,M2,MD);
  endif;
end;

procedure PairMax;
input
  I,J,K: reference integer;
begin
  if I>J then K := I
  else K := J;
end;