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
          - size of an array parameter is not known to the called function until run-time
  - 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);
- readin(v1,...,vn);
- read(f,v1,...,vn);
- readin(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 f.

- write(e1,...,en);
- writein(e1,...,en);
- write(f,e1,...,en);
- writein(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 f.

Simple Statements

Procedure call: P(e1,...,en);

Node j has c-uses of each variable occurring in
the expressions e1,...,en.
These are followed by definitions of each actual parameter which corresponds to a 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,b) 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
followed by a definition of tmp.
Edges (i,b) and (i,j) have
p-uses of v and tmp. Node g has
a c-use followed by a def of v.
R**epe**tt**i**ve** Statements

**Repeat statement:**
repeat statement: repeat S1;...;Sn until B;

Let j be the entry node of S1, and let k be the exit node of Sn.
Edges (k,j) and (j,i) have p-uses of each variable in the boolean expression B.

C**ond**ition**al** Statements

**If-then-else statement**
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.

C**ond**ition**al** Statements

**Case statement**
case c1 of
label-list1 : S1;
...
label-listn : Sn
end;

Let j1,...,jn be the entry nodes of S1,...,Sn, respectively.
Edges (i,j1),...,(i,jn) have p-uses of each variable in the expression c1.

E**ntry** and e**xit** nodes

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

- **Exit node**
  - An undefinition 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^*
- Ignore definitions and uses 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

Simplifying Assumptions

- No interprocedural dataflow analysis
- Ignore pointers
- Array reference simplification
- No aliasing/side-effects
- Consequences
  - Perhaps "less than perfect" test data
**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 “from node i to node j” and “from node i to edge (n_m, j)”**
  - A path (i, n_i, n_j, ..., n_m) containing no definitions or undefinitions of x in nodes n_i, n_j, ..., 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-use(i)
    - Variables which have global c-uses in block i
  - Def(i)
    - Variables which have global definitions in block i
- **Associate with each edge (i, j)**
  - P-use(i, j)
    - Variables which have p-uses on edge (i, j)
- **Define sets of nodes**
  - dcu(x, i)
    - Nodes j such that x ∈ c-use(j) and there is a def-clear path with respect to x from i to j
  - dpu(x, i)
    - Edges (j, k) such that x ∈ p-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 = the set of variables
- N = the set of nodes
- E = the set of edges
def(v) = \{ x ∈ V | x has a global definition in block i \}
c-use(i) = \{ x ∈ V | x has a global c-use in block i \}
p-use(i) = \{ x ∈ V | x has a p-use in edge (i, j) \}
dcu(x, i) = \{ j ∈ N | x ∈ c-use(j) and there is a def-clear path wrt x from i to j \}
dpu(x, j) = \{ (j, k) ∈ E | x ∈ p-use(j, k) and there is a def-clear path wrt x from i 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 \( (n_1,n_2, \ldots, n_k) \) is a du-path wrt \( x \) if \( n_k \) has a global definition of \( x \) and either
    - \( n_k \) has a global c-use of \( x \) and \( (n_1, \ldots, n_k) \) 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 \( \pi \) 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 \( \pi \) 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-def</td>
<td>Some ${\text{ok}, x_i }$ or some ${\text{ok}, x_i, x_j}$</td>
</tr>
<tr>
<td>All-p-subs</td>
<td>${x_1, x_2, \ldots, x_n}$</td>
</tr>
<tr>
<td>All-p-calls/some-calls</td>
<td>${x_1, x_2, \ldots, x_n}$</td>
</tr>
<tr>
<td>All-calls/some-calls</td>
<td>${x_1, x_2, \ldots, x_n}$</td>
</tr>
<tr>
<td>All-uses</td>
<td>${x_1, x_2, \ldots, x_n}$</td>
</tr>
<tr>
<td>All-du-paths</td>
<td>${(x_i, y_j) \mid i \neq j}$</td>
</tr>
</tbody>
</table>

#### "includes" relationship

- **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-P-USES/SOME-C-USES
  |   |   | ALL-C-USES
  |   | ALL-DEF
  |   |   | ALL-P-USES
  | ALL-KNOTS
  |   | ALL-NODES
```

### 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 \(j \in dcu(x, i)\)
- **Definition-p-use association**
  - Triple \((i, (j, k), x)\) where \(i\) is a node containing a global definition of \(x\) and \((j, k) \in dpu(x, i)\)
  - A path \((n_1, n_2, \ldots, n_{k-1}, n_k)\) 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 \((n_1, \ldots, n_{k-1}, n_k)\) is a def-clear simple path wrt \(x\), and
    - \((n_{k-1}, n_k)\) has a p-use of \(x\) and \((n_1, \ldots, n_k)\) 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
  - \(fdcu(x, i) \in dcu(x, i)\)
    - Nodes \(j\) such that \(x \in c-use(j)\) and there is an executable definition clear path wrt \(x\) from \(i\) to \(j\)
  - \(fdpu(x, i) \in dpu(x, i)\)
    - Edges \((j, k)\) such that \(x \in p-use(j, k)\) and there is an executable definition clear path wrt \(x\) from \(i\) to \((j, k)\)

Equivalently

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

Intuitively

- new criterion \(C^*\) for each DF criterion \(C\)
- By selecting the required associations from \(fdcu(x, i)\) and \(fdpu(x, i)\) 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-uses)*</td>
<td>all (i,j) s.t. ( j\in D(i) )</td>
</tr>
<tr>
<td>(all-p-uses)*</td>
<td>all ( R(i,j) ) s.t. ( j\in P(i) )</td>
</tr>
<tr>
<td>(all-p-uses/some-c-uses)*</td>
<td>all ( R(i,j) ) s.t. ( j\in P(i) ) &amp; phen ( P(i) ) \neq \emptyset )</td>
</tr>
<tr>
<td>(all-c-uses/some-p-uses)*</td>
<td>all ( R(i,j) ) s.t. ( j\in P(i) ) &amp; phen ( P(i) ) \neq \emptyset ) &amp; ( i\in D(j) )</td>
</tr>
<tr>
<td>(all-use)*</td>
<td>all ( R(i,j) ) s.t. ( j\in P(i) )</td>
</tr>
<tr>
<td>(all-do-paths)*</td>
<td>all executable do-paths with respect to ( x ) from 1 to ( j ) s.t. ( \text{phen}(i) ) \neq \emptyset and all executable do-paths with respect to ( x ) from 1 to ( j ) for each ( (j,i) \in D(i) )</td>
</tr>
</tbody>
</table>

Includes Relationships

- (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-DFNs)*
- (ALL-P-USES)*

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).

Recursive procedure

First element of array

Last element

Actual parameters at the call site that are bound to formal reference parameters in called procedures.

Recursive procedure

First element of array

Last element

Actual parameters at the call site that are bound to formal reference parameters in called procedures.

Recursive procedure

First element of array

Last element

Actual parameters at the call site that are bound to formal reference parameters in called procedures.
The Def-uses

S = {3, 5, 1, 6}
F = 1
L = 4

All def-use pairs are covered

A test case

S = (3, 5, 1, 6)
F = 1
L = 4

Execute and check

All def-use pairs are covered

Any missed def-uses?

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

procedure GetMinMax:
input
P.L: integer;
MV: reference integer;
declare M1,M2,M3: integer;
begin
  if i = 1 then PairMax(M1,LMAX,MIN)
  else begin
    MD = (M1,LMAX,MIN);
    GetMinMax(M1,M2,M3);
    PairMax(MD,M1,M2,M3)
  end;
end.

procedure PairMax:
input
MD: reference integer;
begin
  if i = 1 then K := 1
  else K := 2;
end.