Data Flow Testing

Mark New

University of Wales Swansea
Saturday 2\textsuperscript{nd} December, 2006
Overview

- Background
- Data flow testing
- Define/Use testing
- Slice-based testing
Background
Data Flow Testing

- Structural testing
- A form of path testing?
- Focus on variables
- Most programs work with data
  - Variables receive values
  - Values are then used/referenced in calculations
  - (maybe used when setting other variables)
Data Flow Testing Cont'd

• Start with a program graph (next slide)
• 2 forms:
  1. Define/Use testing
  2. “Program slice” testing
• Early data flow testing centred on three faults:
  • Variable defined but never used/referenced
  • Variable used but never defined
  • Variable defined twice before use
• Define/reference anomalies – static analysis
program Example()

var staffDiscount, totalPrice, finalPrice, discount, price

staffDiscount = 0.1
totalPrice = 0

input(price)
while(price != -1) do
  totalPrice = totalPrice + price
  input(price)
od

print("Total price: "+totalPrice)

if(totalPrice > 15.00) then
  discount = (staffDiscount * totalPrice) + 0.50
else
  discount = staffDiscount * totalPrice
fi

print("Discount: "+discount)
finalPrice = totalPrice - discount
print("Final price: "+finalPrice)
endprogram
Define/Use Testing
Define/Use Testing

- First formalised by Rapps/Weyuker in early '80s
- A way to examine points where faults may occur
- Uses statement fragments (or statements)
- For structured program P
  - Program graph: $G(P)$
    - Single entry & exit nodes; no edges from node to itself
- Set of program variables: $V$
- Set of all paths in P: $PATHS(P)$
Defining and Usage Nodes

• Defining node (e.g. input $x$, $v = 2$, etc.):
  $\text{DEF}(v, n)$: Node $n$ in $G(P)$ is a defining node of var $v$ in $V$ iff value of $v$ is **defined** at $n$.

• Usage node (e.g. output $x$, $a = 2+v$, etc.):
  $\text{USE}(v, n)$: Node $n$ in $G(P)$ is a usage node of var $v$ in $V$ iff value of $v$ is **used** at $n$. 
Du- and Dc-Paths

- **Definition-use (du) path (wrt. variable v)**
- A path in PATHS(P) such that
- for some v in V
- There exist DEF(v, m), USE(v, n) nodes s.t.
- m and n are *initial and final nodes* of the path respectively.
Du- and Dc-Paths

- Definition-clear (dc) path (wrt. variable v)
- A *du-path* in PATHS(P) where
- the initial node of the path is the *only defining node* of v (in the path).
Example

• For price variable in example

2 define nodes
DEF(price, 5)
DEF(price, 8)

2 use nodes
USE(price, 6)
USE(price, 7)

Du-paths:
<5, 6>
<5, 6, 7>
<8, 9, 6>
<8, 9, 6, 7>
All are definition-clear.
Definitions

• USE – five types:
  – P-use – predicate (decision) (e.g. \( if(x=5) \))
  – C-use – computation (e.g. \( b=3+d \))
  – O-use – output (e.g. \( output(x) \))
  – L-use – location (pointers, etc.)
  – I-use – Iteration (internal counters, loop indices)

• DEF – two types:
  – I-def – input
  – A-def – assignment
Def/Use Test Coverage Metrics

- Du-paths allow you to define a set of test coverage metrics
- Rapps-Weyuker data flow metrics
- Defined in early 1980s
- Relationship: “subsumption” between metrics
The Metrics

- **All-Paths, All-Edges** and **All-Nodes** are equivalent to Miller's metrics (Path Testing)
- For the others, assume that define & usage nodes have been defined for all variables
- Du-paths identified wrt. each variable
- \( T = \) a set of paths in \( G(P) \)
- **DEF nodes** X **USE nodes** to define du-paths
  - Can result in infeasible paths.
Metrics cont'd

- T satisfies **All-Defs** for P iff for every var \( v \) in \( V \), T contains dc-paths from every DEF of \( v \) to a USE of \( v \).

- T satisfies **All-P-Uses** for P iff T contains dc-paths from every DEF of \( v \) to every P-use of \( v \).

- T satisfies **All P-Uses/Some C-Uses** for P iff for every var \( v \) in \( V \), T contains dc-paths from every DEF of \( v \) to every P-use of \( v \) – if a def of \( v \) has no P-uses, dc-path leads to at least 1 C-use.

- **All-C-Uses/Some-P-Uses** - vice-versa!
Metrics cont'd

• T satisfies **All-Uses** for P iff for every var v in V, T contains dc-paths from every DEF of v to every USE of v and to the successor node of each USE(v, n).

• T satisfies **All-DU-Paths** for P iff for every var v in V, T contains dc-paths from every DEF of v to every USE of v and to the successor node of each USE(v, n)
  - And paths are either single loop traversals or loop free.
“Subsumption” of Metrics

- Arrows show relationship
- e.g. All-Paths “stronger” than All-DU-Paths
- All-Defs “not comparable” to All-Edges/Nodes
- Typically accepted minimum metric: All-Edges
- All-Paths often infeasible
Slice-Based Testing
What is a slice?

• Given a program P, program graph G(P) and set of variables (in P) V
• Slice on V at statement (fragment) n – S(V, n)
• S(V, n) is the set of node numbers of all statements in P prior to n that contribute to the values of variables in V at n.
• Exclude all non-executable statements
• Also exclude O-use, L-use, I-use nodes from slices
Slice: Example

- Variable price in example program
  - $S\left(\text{price}, 5\right) = \{5\}$
  - $S\left(\text{price}, 6\right) = \{5, 6, 8, 9\}$
  - $S\left(\text{price}, 7\right) = \{5, 6, 8, 9\}$
  - $S\left(\text{price}, 8\right) = \{8\}$
Use of Slices

- Slice composition (code slices, test, merge)
- Relative complements of slices
  - e.g. $S(a, 35)$ is a subset of $S(b, 48)$ ($b$ uses $a$)
  - Problem with $b$ at line 48?
  - If there is no problem with $a$ at line 35, then...
  - ...problem is in $S(b, 48) - S(a, 35)$
  - Otherwise problem could be in either part.
- When slice for DEF for var = slice for USE for var, then path is definition-clear.
Summary

- Data flow testing
  - Looking at variable usage to find faults
- Define/Use
  - DEF, USE, Du-paths, Dc-paths
  - Rapps/Weyuker metrics
- Program slice testing