Dataflow Testing

Chapter 10
Dataflow Testing

- Testing All-Nodes and All-Edges in a control flow graph may miss significant test cases.

- Testing All-Paths in a control flow graph is often too time-consuming.

- Can we select a subset of these paths that will reveal the most faults?

- Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.
Concordance

- What is a concordance?
What is a concordance?

- An alphabetical list of the words (esp. the important ones) present in a text, usually with citations of the passages concerned
- Used to help find particular passages
- Also used to analyze books to establish authorship
  - A concordance to the Bible

What is the a concordance wrt to program text?
- What is the analogue?
Data flow analysis is in part based on concordance analysis such as that shown below.

- Result is a variable cross-reference table

<table>
<thead>
<tr>
<th>Defined</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>25, 51, 123</td>
</tr>
<tr>
<td>beta</td>
<td>18, 123, 124, 51, 123, 124</td>
</tr>
<tr>
<td>gamma</td>
<td>51, 25, 51, 124</td>
</tr>
</tbody>
</table>
Dataflow Analysis

- Can reveal interesting bugs
  - A variable that is defined but never used
  - A variable that is used but never defined
  - A variable that is defined twice before it is used
  - Sending a modifier message to an object more than once between accesses
  - Deallocating a variable before it is used
    - Container problem
      - Deallocating container loses references to items in the container, memory leak
Bugs can be found from a cross-reference table using **static analysis**.

Paths from the definition of a variable to its use are more likely to contain bugs.
Definitions

- A node \( n \) in the program graph is a **defining** node for variable \( v \) – \( \text{DEF}(v, n) \) – if the value of \( v \) is defined at the statement fragment in that node
  - **Input, assignment, procedure calls**

- A node in the program graph is a **usage** node for variable \( v \) – \( \text{USE}(v, n) \) – if the value of \( v \) is used at the statement fragment in that node
  - **Output, assignment, conditionals**
A usage node is a predicate use, **P-use**, if variable \( v \) appears in a predicate expression

- **Always in nodes with outdegree \( \geq 2 \)**

A usage node is a computation use, **C-use**, if variable \( v \) appears in a computation

- **Always in nodes with outdegree \( \leq 1 \)**
A node in the program is a kill node for a variable \( v \) – \( \text{KILL}(v, n) \) – if the variable is deallocated at the statement fragment in that node.
Example 2 – Billing program

calculateBill (usage : INTEGER) : INTEGER

double bill = 0;

if usage > 0 then bill = 40 fi

if usage > 100

then if usage ≤ 200

then bill = bill + (usage – 100) * 0.5

else bill = bill + 50 + (usage – 200) * 0.1

if bill ≥ 100 then bill = bill * 0.9 fi

fi

fi

return bill

end
What is a du-path?

Definition-Use path
What is a du-path?

A definition-use path, du-path, with respect to a variable \( v \) is a path whose first node is a defining node for \( v \), and its last node is a usage node for \( v \).
Definition clear path

- What is a dc-path?
What is a dc-path?

- A du-path with no other defining node for $v$ is a definition-clear path
Example 1 – Max program

```java
1  int max = 0;
2  int j = s.nextInt();
3  while (j > 0)
4    if (j > max) {
5      max = j;
6    }
7    j = s.nextInt();
8  }
9  System.out.println(max);
```
Max program – analysis

```java
int max = 0;
int j = s.nextInt();
while (j > 0)
    System.out.println(max);
    max = j;
    if (j > max)
        j = s.nextInt();
```

Legend
A..F Segment name
\(d\) defining node for \(j\)
\(u\) use node for \(j\)

dc-paths \(j\)
A B
A B C
A B C D
E B
E B C
E B C D

dc-paths \(max\)
A B F
A B C
D E B C
D E B F
Based on these definitions we can define a set of coverage metrics for a set of test cases.

We have already seen:
- All-Nodes
- All-Edges
- All-Paths

Data flow has additional test metrics for a set $T$ of paths in a program graph:
- All assume that all paths in $T$ are feasible.
All-Defs Criterion

- The set $T$ satisfies the All-Def criterion
  - For every variable $v$, $T$ contains a dc-path from every defining node for $v$ to at least one usage node for $v$
  - Not all use nodes need to be reached

$$\forall v \in V(P), nd \in prog\_graph(P) \setminus DEF(v, nd)$$
$$\exists nu \in prog\_graph(P) \setminus USE(v, nu)$$
$$dc\_path(nd, nu) \in T$$
The set $T$ satisfies the All-Uses criterion iff

- For every variable $v$, $T$ contains dc-paths that start at every defining node for $v$, and terminate at every usage node for $v$
  - Not $\text{DEF}(v, n) \times \text{USE}(v, n)$ – not possible to have a dc-path from every defining node to every usage node

\[
(\forall v \in V(P), nu \in \text{prog\_graph}(P) \setminus \text{USE}(v, nu) \quad \exists nd \in \text{prog\_graph}(P) \setminus \text{DEF}(v, nd) \quad \exists \text{dc\_path}(nd, nu) \in T)
\]

$\wedge$

all\_defs\_criterion
The set $T$ satisfies the All-P-uses/Some-C-uses criterion iff

- For every variable $v$ in the program $P$, $T$ contains a dc-path from every defining node of $v$ to every P-use node for $v$
  - If a definition of $v$ has no P-uses, a dc-path leads to at least one C-use node for $v$

$$(\forall v \in V(P), nu \in \text{prog\_graph}(P) \mid P\_use(v,nu)$$
$$\bullet \exists nd \in \text{prog\_graph}(P) \mid \text{DEF}(v,nd) \bullet \text{dc\_path}(nd,nu) \in T)$$
$$\wedge$$
$$\text{all\_defs\_criterion}$$
All-C-uses / Some-P-uses

- The test set $T$ satisfies the All-C-uses/Some-P-uses criterion iff
  - For every variable $v$ in the program $P$, $T$ contains a dc-path from every defining node of $v$ to every C-use of $v$
  - If a definition of $v$ has no C-uses, a dc-path leads to at least one P-use

$$(\forall v \in V(P), \text{nu} \in \text{prog\_graph}(P) \setminus \text{C\_use}(v, \text{nu}))$
  $$
  \begin{equation}
  \exists \text{nd} \in \text{prog\_graph}(P) \setminus \text{DEF}(v, \text{nd}) \cdot \text{dc\_path}(\text{nd}, \text{nu}) \in T
  \end{equation}$$

\text{all\_defs\_criterion}$$
miles_per_gallon ( miles, gallons, price : INTEGER )

if gallons = 0 then
   // Watch for division by zero!!
   Print("You have " + gallons + "gallons of gas")
else if miles/gallons > 25
   then print( "Excellent car. Your mpg is " + miles/gallon)
   else print( "You must be going broke. Your mpg is " + miles/gallon + " cost " + gallons * price)
fi
end
Miles-per-gallon Program – 2

- We want du- and dc-paths
- What do you do next?
### Mile-per-gallon Program – Segmented

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>gasguzzler (miles, gallons, price : INTEGER)</strong></td>
<td>A</td>
</tr>
<tr>
<td><strong>if gallons = 0 then</strong></td>
<td>B</td>
</tr>
<tr>
<td>// Watch for division by zero!!</td>
<td>C</td>
</tr>
<tr>
<td>Print(&quot;You have (^{+}) gallons (^{+}) &quot; &quot;gallons of gas&quot;))</td>
<td>C</td>
</tr>
<tr>
<td><strong>else if miles/gallons &gt; 25</strong></td>
<td>D</td>
</tr>
<tr>
<td><strong>then print( &quot;Excellent car. Your mpg is &quot;</strong></td>
<td>E</td>
</tr>
<tr>
<td>+ miles/gallon)</td>
<td>E</td>
</tr>
<tr>
<td><strong>else print( &quot;You must be going broke. Your mpg is &quot;</strong></td>
<td>F</td>
</tr>
<tr>
<td>+ miles/gallon + &quot; cost &quot; + gallons * price)</td>
<td>F</td>
</tr>
<tr>
<td><strong>fi</strong></td>
<td>G</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td>G</td>
</tr>
</tbody>
</table>
Miles-per-gallon Program – 3

- We want du- and dc-paths
- What do you do next?
What do you do now?
Def miles, gallons

P-use gallons

P-use miles, gallons

Possible C-use miles, gallons
But not common practice

DFT–28
We want du- and dc-paths

What do you do next?
Example du-paths

For each variable in the miles_per_gallon program create the test paths for the following dataflow path sets

- All-Defs (AD)
- All-C-uses (ACU)
- All-P-uses (APU)
- All-C-uses/Some-P-uses (ACU+P)
- All-P-uses/Some-C-uses (APU+C)
- All-uses
MPG – DU-Paths for Miles

- **All-Defs**
  - Each definition of each variable for at least one use of the definition
    - A B D

- **All-C-uses**
  - At least one path of each variable to each c-use of the definition
    - A B D E
    - A B D F
    - A B D
All-P-uses

- At last one path of each variable to each p-use of the definition
  - A B D

All-C-uses/Some-P-uses

- At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered use p-use
  - A B D E   A B D F   A B D
All-P-uses/Some-C-uses

- At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered by p-use, then use c-use
  - A B D

All-uses

- At least one path of each variable definition to each p-use and each c-use of the definition
  - A B D
  - A B D E
  - A B D F
MPG – DU-Paths for Gallons

- **All-Defs**
  - Each definition of each variable for at least one use of the definition
    - A B

- **All-C-uses**
  - At least one path of each variable to each c-use of the definition
    - A B C  A B D E  A B D F  A B D
All-P-uses

At least one path of each variable definition to each p-use of the definition

- A B
- A B D

All-C-uses/Some-P-uses

At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered by c-use, then use p-use

- A B C
- A B D E
- A B D F
- A B D
All-P-uses/Some-C-uses

At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered use c-use

- A B
- A B D

All-uses

At least one path of each variable definition to each p-use and each c-use of the definition

- A B
- A B C
- A B D
- A B D E
- A B D F
MPG – DU-Paths for Price

- **All-Defs**
  - Each definition of each variable for at least one use of the definition
    - A B D F

- **All-C-uses**
  - At least one path of each variable to each c-use of the definition
    - A B D F
凡事-P-uses

- At least one path of each variable definition to each p-use of the definition
  - None

凡事-C-uses/Some-P-uses

- At least one path of each variable definition to each c-use of the variable. If any variable definitions are not covered use p-use
  - A B D F
All-P-uses/Some-C-uses

- At least one path of each variable definition to each p-use of the variable. If any variable definitions are not covered use c-use
  - A B D F

All-uses

- At least one path of each variable definition to each p-use and each c-use of the definition
  - A B D F
Rapps-Weyuker data flow hierarchy

- All-Paths
  - All-DU-Paths
    - All-Uses
      - All-C-uses
        - Some-P-uses
      - All-P-uses
        - Some-C-uses
  - All-Defs
  - All-P-uses
    - All-Edges
      - All-Nodes
### Potential Anomalies – static analysis

#### Data flow node combinations for a variable

**Allowed? – Potential Bug? – Serious defect?**

<table>
<thead>
<tr>
<th>Anomalies</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ d</td>
<td>first define</td>
</tr>
<tr>
<td>du</td>
<td>define-use</td>
</tr>
<tr>
<td>dk</td>
<td>define-kill</td>
</tr>
<tr>
<td>~ u</td>
<td>first use</td>
</tr>
<tr>
<td>ud</td>
<td>use-define</td>
</tr>
<tr>
<td>uk</td>
<td>use-kill</td>
</tr>
<tr>
<td>~ k</td>
<td>first kill</td>
</tr>
<tr>
<td>ku</td>
<td>kill-use</td>
</tr>
</tbody>
</table>
### Data flow node combinations for a variable

<table>
<thead>
<tr>
<th>Anomalies</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>kd</td>
<td>kill-define</td>
</tr>
<tr>
<td>dd</td>
<td>define-define</td>
</tr>
<tr>
<td>uu</td>
<td>use-use</td>
</tr>
<tr>
<td>kk</td>
<td>kill-kill</td>
</tr>
<tr>
<td>d ~</td>
<td>define last</td>
</tr>
<tr>
<td>u ~</td>
<td>use last</td>
</tr>
<tr>
<td>k ~</td>
<td>kill last</td>
</tr>
</tbody>
</table>

**Allowed? – Potential Bug? – Serious defect?**
<table>
<thead>
<tr>
<th>Anomalies</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ d</td>
<td>first define – Allowed – normal case</td>
</tr>
<tr>
<td>du</td>
<td>define-use – Allowed – normal case</td>
</tr>
<tr>
<td>dk</td>
<td>define-kill – Potential bug</td>
</tr>
<tr>
<td>~ u</td>
<td>first use – Potential bug</td>
</tr>
<tr>
<td>ud</td>
<td>use-define – Allowed – redefine</td>
</tr>
<tr>
<td>uk</td>
<td>use-kill – Allowed – normal case</td>
</tr>
<tr>
<td>~ k</td>
<td>first kill – Serious defect</td>
</tr>
<tr>
<td>ku</td>
<td>kill-use – Serious defect</td>
</tr>
</tbody>
</table>
## Potential Anomalies – static analysis – 4

<table>
<thead>
<tr>
<th>Anomalies</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>kd</td>
<td>kill-define Allowed - redefined</td>
</tr>
<tr>
<td>dd</td>
<td>define-define Potential bug</td>
</tr>
<tr>
<td>uu</td>
<td>use-use Allowed - normal case</td>
</tr>
<tr>
<td>kk</td>
<td>kill-kill Serious defect</td>
</tr>
<tr>
<td>d ~</td>
<td>define last Potential bug</td>
</tr>
<tr>
<td>u ~</td>
<td>use last Allowed - normal case</td>
</tr>
<tr>
<td>k ~</td>
<td>kill last Allowed - normal case</td>
</tr>
</tbody>
</table>
Data flow guidelines

- When is dataflow analysis good to use?
When is dataflow analysis good to use?

- Data flow testing is good for computationally/control intensive programs
  - If P-use of variables are computed, then P-use data flow testing is good
- Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.
Aliasing of variables causes serious problems!

Working things out by hand for anything but small methods is hopeless

Compiler-based tools help in determining coverage values