Advanced Software Testing and Debugging (CS598)

Program Analysis Basics

Spring 2022

Lingming Zhang
Program analysis

Program analyzers aim to automatically analyze the behavior of computer programs regarding certain properties.
How do we analyze arbitrary programs?
Abstraction!

• Transform programs under analysis into structured code representations
  • Easier parsing
  • Easier modification
  • Easier generation
What code representations are used in a typical compiler pass?

Source code

Lexical Analysis

Parsing

Semantic Analysis

Optimization

Code Generation

Machine/byte code
Lexical analysis

- **Input:** source code text (sequence of chars)
- **Output:** sequence of tokens

```plaintext
while (y < z) {
    x = a + b;
    y += x;
}
```

T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_PLUS
T_PlusAssign
T_Identifier x
T_Semicolon
T_Semicolon
T_CloseBrace
Syntactic analysis

• **Input**: sequence of tokens from lexical analysis
• **Output**: abstract syntax tree (AST)

```
while (y < z) {
    x = a + b;
    y += x;
}
```
Semantic analysis

• **Input**: abstract syntax tree (AST)
• **Output**: annotated AST

```
while (y < z) {
    x = a + b;
    y += x;
}
```

```
while < y < z:
    block:
        y = void
        z = void
        x = int
        x = int
        y = int
        y = int
        x = int
        x = int
```

Type checking rules:

- **i is an integer literal**
  \[
  E \vdash i : \text{int}
  \]

- **x:T is in E**
  \[
  E \vdash x : T
  \]

- **E \vdash e_1 : \text{int}**
  \[
  E \vdash e_2 : \text{int}
  \]
  \[
  E \vdash e_1 + e_2 : \text{int}
  \]

- **E \vdash e_1 : \text{int}**
  \[
  E \vdash e_2 : \text{int}
  \]
  \[
  E \vdash e_1 \ast e_2 : \text{int}
  \]
Optimization

- **Input**: original code representation
- **Output**: optimized code representation

```
int a=1;
int b=1;
...
while (y < z) {
    x = a + b;
    y += x;
}
```

```
int a=1;
int b=1;
...
while (y < z) {
    y += 2;
}
```
Code generation

- **Input**: optimized code representation
- **Output**: final target code

```java
while (y < z) {
    x = a + b;
    y += x;
}
```
Topics

• Abstract syntax tree (AST)
• Control-flow graph (CFG)
• Control-flow-based code coverage
• Data-flow analysis
• Data-flow-based code coverage
How do we describe a programming language?

Example program:

```java
while (y < z) {
    x = a + b;
    y += x;
}
```

A grammar covering this program and similar ones:

- `Stmt → WhileStmt | AssignStmt | CompoundStmt`
- `WhileStmt → "while" "(" Exp ")" Stmt`
- `AssignStmt → ID "=" Exp ";"`
- `CompoundStmt → "{" StmtList "}"`
- `StmtList → ε | Stmt StmtList`
- `Exp → Less | Add | ID`
- `Less → Exp "<" Exp`
- `Add → Exp "+" Exp`
Context-free grammar

• A context-free grammar \( G = \langle \Sigma, N, P, S \rangle \), where
  • \( \Sigma \): alphabet (finite set of symbols, or terminals)
    • Often written in lowercase
  • \( N \): a finite, nonempty set of nonterminal symbols, \( N \cap \Sigma = \emptyset \)
    • Often at least the first letter in UPPERCASE
  • \( P \): the set of production rules, each with the form \( X \rightarrow Y_1 Y_2 \ldots Y_n \)
    • where \( X \in N \), \( n \geq 0 \), and \( Y_k \in N \cup \Sigma \)
  • \( S \): the start symbol (one of the nonterminals), i.e., \( S \in N \)

Grammar (P):
\[
\begin{align*}
E & \rightarrow E+E \\
E & \rightarrow E*E \\
E & \rightarrow (E) \\
E & \rightarrow id
\end{align*}
\]

Grammar (P):
\[
\begin{align*}
E & \rightarrow E+E \\
& \quad \mid E*E \\
& \quad \mid (E) \\
& \quad \mid id
\end{align*}
\]

\( \Sigma \): +, * (, ), id
\( N \): E
\( S \): E
Context-free grammar

Example program:

```plaintext
while (y < z) {
    x = a + b;
    y += x;
}
```

A grammar covering this program and similar ones:

```plaintext
Stmt → WhileStmt | AssignStmt | CompoundStmt
WhileStmt → "while" "(" Exp ")" Stmt
AssignStmt → ID "=" Exp ";"
CompoundStmt → "{" StmtList "}"
StmtList → ε | Stmt StmtList
Exp → Less | Add | ID
Less → Exp "<" Exp
Add → Exp "+" Exp
Σ: ID, "while", "(" , "=" , ")", "{" , "}" , ";" , ε
N: Stmt, WhileStmt, ...
S: Stmt
```
Context-free grammar: generating strings

- $G$ defines a language $L(G)$ over the alphabet $\Sigma$

- $\Sigma^*$ is the set of all possible sequences of $\Sigma$ symbols

- $L(G)$ is the subset of $\Sigma^*$ that can be derived from the start symbol $S$, by following the production rules $P$
  - A **derivation** is such a sequence of productions applied

<table>
<thead>
<tr>
<th>Grammar:</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $\rightarrow$ E+E</td>
</tr>
<tr>
<td>$\mid$ E*E</td>
</tr>
<tr>
<td>$\mid$ (E)</td>
</tr>
<tr>
<td>$\mid$ id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $\rightarrow$ E+E</td>
</tr>
<tr>
<td>$\rightarrow$ E * E+E</td>
</tr>
<tr>
<td>$\rightarrow$ id * E + E</td>
</tr>
<tr>
<td>$\rightarrow$ id * id + E</td>
</tr>
<tr>
<td>$\rightarrow$ id * id + id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L(G)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>id * id</td>
</tr>
<tr>
<td>id * id + id</td>
</tr>
<tr>
<td>id * id + id *id</td>
</tr>
<tr>
<td>id + id + id + id</td>
</tr>
</tbody>
</table>

...
Context-free grammar: parsing strings

- Checking if input string (e.g., code) \( s \in L(G) \), i.e., checking for acceptance
- Algorithm: Find a derivation starting from the start symbol of \( G \) to \( s \)

Language grammar:
\[
E \rightarrow E+E \mid E^*E \mid (E) \mid id
\]

Source

Derivation

Parse tree
Abstract syntax tree (AST)

• Simplified syntactic representations derived from code parse tree
• Represents the abstract syntactic structure of a language construct
• Usually the interior and root nodes represent operators, and the children of each node represent the operands of that operator

ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees...
AST: more examples

Example program:

```java
while (y < z) {
    x = a + b;
    y += x;
}
```

Parse tree

```
Stmt
  └── WhileStmt
        └── Exp
                └── Less
                        └── Exp
                                └── Exp
                                        └── Stmt
```

AST

```
WhileStmt
  └── Less
      └── Stmt
          └── Exp
                  └── Exp
                          └── Exp
```

18
Example program:

```java
while (y < z) {
    x = a + b;
    y += x;
}
```

Parse tree

```
AssignStmt
  /
/: Stmt
  /
  : Exp
    /
    : Add
      /
      : Exp
        /
        : a
      /
      : Exp
        /
        : b
```

AST

```
AssignStmt
  /
/: Add
  /
  : Exp
    /
    : a
  /
  : Exp
    /
    : b
```

AST: more examples
Example program:

```java
while (y < z) {
    x = a + b;
    y += x;
}
```

Parse tree and AST representation:

```
Stmt
   AssignStmt
      Exp
         Add
            Exp
            Exp
       y = y + x;
```

AST:

```
AssignStmt
   y
   Add
      y
      x
```
AST: typical structures

- **AssignStmt**
  - **ID**
  - **Exp**
  - **Assignment**

- **Op**
  - **Exp**
  - **Exp**
  - **Binary operator**

- **Op**
  - **Exp**
  - **Unary operator**

- **WhileStmt**
  - **Exp**
  - **Stmt**
  - **Loop**

- **IfStmt**
  - **Exp**
  - **Stmt**
  - **Stmt**
  - **Conditional check**

- **CompoundStmt**
  - **Stmt**
  - **Stmt**
  - **Stmt**
  - **...**
  - **Compound statement**
## Mapping between parse tree and AST

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁+E₂</td>
<td>E.node = new Node(“+”, E₁.node, E₂.node)</td>
</tr>
<tr>
<td>E → E₁*E₂</td>
<td>E.node = new Node(“*”, E₁.node, E₂.node)</td>
</tr>
<tr>
<td>E → (E₁)</td>
<td>E.node = E₁.node</td>
</tr>
<tr>
<td>E → id</td>
<td>E.node = new Leaf(id, id.entry)</td>
</tr>
</tbody>
</table>

**Parse tree**

```
E
  +
  E
  *
  E
  id
```

**AST**

```
+  
  |   |
  E  id
  |
  | id
  |
  | id
```
AST applications

• AST provides a basic model of source code, supporting reading, modifying, and even generating source code in a systematic way
  • Compilers
  • Program analysis
  • Source code instrumentation
  • Automated program repair
  • Code generation and program synthesis
  • ...

Topics

• Abstract syntax tree (AST)
• Control-flow graph (CFG)
• Control-flow-based code coverage
• Data-flow analysis
• Data-flow-based code coverage
Basic block

- A **basic block** is a sequence of straight-line code that can be entered only at the beginning and exited only at the end

Building basic blocks:
1. Identify leaders:
   - The first instruction in a procedure, or
   - The target of any branch, or
   - An instruction immediately following a branch
2. Gobble all subsequent instructions until the next leader

\[ x = a \]
\[ y = b \times 2 \]
\[ \text{if}(x > y) \]
Basic block example

Program

```java
while (x < y) {
    y = f(x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```

Leaders

1 while (x < y) {
2     y = f(x, y);
3     if (y == 0) {
4         break;
5     } else if (y < 0) {
6         y = y*2;
7         continue;
8     }
9     x = x + 1;
10 }
11 print (y);

Basic blocks

1: while (x<y)
2: y=f(x,y)
3: if(y==0)
4: break
5: else if(y<0)
6: y=y*2
7: continue
9: x=x+1
11: print(y)

Building basic blocks:
1. Identify leaders :
   • The first instruction in a procedure, or
   • The target of any branch, or
   • An instruction immediately following a branch
2. Gobble all subsequent instructions until the next leader
Control-flow graph (CFG)

- A control-flow graph (CFG) is a rooted directed graph $G = (N, E)$
  - $N$ is the set of basic blocks
  - $E$ is the flow of control between basic blocks

Building CFG:
1. Each CFG node represents a basic block
2. There is an edge from node $i$ to $j$ if
   - Last statement of block $i$ branches to the first statement of $j$, or
   - Block $i$ is immediately followed in program order by block $j$ (fall through)

That said, as long as the execution of node $i$ could be followed by node $j$, connect them!
while \(x < y\) {
\[y = f(x, y);\]
if \(y == 0\) {
break;
} else if \(y < 0\) {
\[y = y \times 2;\]
continue;
} x = x + 1;
print (y);
Topics

• Abstract syntax tree (AST)
• Control-flow graph (CFG)
• Control-flow-based code coverage
• Data-flow analysis
• Data-flow-based code coverage
Control-flow-based code coverage

• Given the CFG, define a coverage target and write tests to achieve it
  • Higher coverage=> more code portions tested=> potentially better tests!

• A practical way to measure test quality!

• Typical control-flow-based code coverage
  • Statement coverage
  • Branch coverage (aka decision coverage)
  • Path coverage
  • Condition coverage
  • Modified condition/decision coverage (MCDC)
  • …
Statement coverage

• **Target:** covering all CFG nodes

**Test1:** 1-11  
**Test2:** 1-2-3-4-11  
**Test3:** 1-2-3-5-9-1-11

Are they covering all statements?  
NO, statement coverage: 7/9, statements 6 and 7 never covered!
Branch coverage (decision coverage)

• **Target**: covering all CFG edges
• Equivalent to covering all branches of the predicate nodes
  • True and false branches of each `if` node
  • The two branches corresponding to the condition of a loop
  • All alternatives in a `switch` node
• Is branch coverage equivalent to statement coverage?

```java
if (x < y) {
    x++;  
}
return x;
```

**Test1**: x=1, y=2

Statement coverage: $3/3$
Branch coverage: $1/2$
Path coverage

- **Target**: covering all possible paths on CFG
- Is path coverage equivalent to branch coverage?

The number of paths could be infinite (loops) or exponential (branches)!
Control-flow-based coverage: summary

- Path coverage strictly subsumes branch coverage
- Branch coverage in turn strictly subsumes statement coverage
Deadlines

• Jan 29 (11:59pm)
  • Presentation choice submission (submission on course webpage)

• Jan 31 (11:59pm)
  • First paper review (submission on Canvas assignments)

• Feb 17 (in class)
  • Proposal presentation
Topics

• Abstract syntax tree (AST)
• Control-flow graph (CFG)
• Control-flow-based code coverage
• Data-flow analysis
• Data-flow-based code coverage
Data-flow analysis

• A framework for proving facts (e.g., reaching definitions) about programs
• Operates on control-flow graphs (CFGs), typically
• Works best on properties about how program computes
• Based on all paths through program
  • Including infeasible paths
Variable definition/use

• A program variable is **defined** whenever its value is modified:
  • On the left-hand side of an assignment statement: \( y = 17 \)
  • In an input statement: \texttt{read(y)}
  • As a call-by-reference parameter in a subroutine call: \texttt{update(x, &y)}

• A program variable is **used** whenever its value is read:
  • \textbf{P-use} (predicate-use): use in the predicate of a branch statement
  • \textbf{C-use} (computation-use): all other uses

\begin{verbatim}
if ( x > 0 ){
  print(y);
}
\end{verbatim}
A typical analysis: reaching definitions

- A definition (statement) $d$ of a variable $v$ reaches CFG node $n$ if there is a path from $d$ to $n$ such that $v$ is not redefined along that path.

- Reaching definitions applications:
  - Build use/def chains
  - Constant propagation
  - Loop invariant code motion

Is this the only def of $x$ reaching $n$? Can we replace $y = x \times 2$ with $y = 10$?

Any other reaching definitions of $x/y$ in the loop? Can we move “a = x + y” out of the loop?
Reaching definitions: example

<table>
<thead>
<tr>
<th>n</th>
<th>IN[n]</th>
<th>OUT[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>∅</td>
<td>{d1}</td>
</tr>
<tr>
<td>B2</td>
<td>{d1}</td>
<td>{d1}</td>
</tr>
<tr>
<td>B3</td>
<td>{d1}</td>
<td>{d2}</td>
</tr>
<tr>
<td>B4</td>
<td>{d1, d2}</td>
<td>{d3}</td>
</tr>
<tr>
<td>B5</td>
<td>{d1, d3}</td>
<td>{d1, d3, d4}</td>
</tr>
</tbody>
</table>

**IN[n]:** set of facts (reaching definitions) at entry of node n

**OUT[n]:** set of facts (reaching definitions) at exit of node n

Constant propagation can be applied to B5 as j is always 1!
Reaching definitions: transfer functions

\[ d: x = y + z \]

\[ \text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n'] \]

\[ \text{IN}[n] = \text{OUT}[n_1] \cup \text{OUT}[n_2] \cup \text{OUT}[n_3] \]

\[ \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \]

\[ \text{KILL}[n] = \text{Def}[x] - \{d\}, \text{ where } \text{Def}[x] : \text{set of all definitions of } x \]

\[ \text{GEN}[n] = \{d\} \]
Reaching definitions algorithm

for (each node n):
    \( \text{IN}[n] = \text{OUT}[n] = \emptyset \)

for (each node n):
    \( \text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n'] \)
    \( \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \)

Any issues?
Reaching definitions: example

The IN set for B4 is incorrect (should be \{d1,d2\})!
Reaching definitions algorithm: revised

for (each node n):
   \( \text{IN}[n] = \text{OUT}[n] = \emptyset \)

repeat:
for (each node n):
   \( \text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n'] \)
   \( \text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \)

until fixed point: \( \text{IN}[n] \) and \( \text{OUT}[n] \) stop changing for all \( n \)
**Reaching definitions: revisit the example**

<table>
<thead>
<tr>
<th>n</th>
<th>GEN[n]</th>
<th>Kill[n]</th>
<th>IN[n]</th>
<th>OUT[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1}</td>
<td>{d2, d3}</td>
<td>∅</td>
<td>{d1}</td>
</tr>
<tr>
<td>B2</td>
<td>∅</td>
<td>∅</td>
<td>{d1}</td>
<td>{d1}</td>
</tr>
<tr>
<td>B3</td>
<td>{d2}</td>
<td>{d1, d3}</td>
<td>{d1}</td>
<td>{d2}</td>
</tr>
<tr>
<td>B4</td>
<td>{d3}</td>
<td>{d1, d2}</td>
<td>{d1, d2}</td>
<td>{d3}</td>
</tr>
<tr>
<td>B5</td>
<td>{d4}</td>
<td>∅</td>
<td>{d1, d3}</td>
<td>{d1, d3, d4}</td>
</tr>
</tbody>
</table>

**Definitions**

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

- IN[n] = a set of reaching definitions before n
- OUT[n] = a set of reaching definitions after n
- KILL[n] = a set of definitions killed by definitions in node n
- GEN[n] = a set of locally available definitions in node n
Does it always terminate?

The two operations of reaching definitions analysis are monotonic

- IN and OUT sets never shrink, only grow
- Largest they can be is set of all definitions in program, i.e., finite

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

IN and OUT will stop changing after some iteration
Other classical dataflow analyses

• **Live Variables Analysis**: for dead code elimination
  • Determine for each program point which variables could be *live* at the point’s exit
  • A variable is **live** if there is a path to a use of the variable that doesn’t redefine the variable

• **Available Expressions Analysis**: for avoiding recomputing expressions
  • Determine, for each program point, which expressions must already have been computed, and not later modified, on all paths to the program point

• **Very Busy Expressions Analysis**: for reducing code size
  • An expression is very *busy* at a program point $p$ if, no matter what path is taken from $p$, the expression is used before any of its variables is redefined
Live variables: transfer functions

KILL[n] = a set of variables defined in node n
GEN[n] = a set of variables used in node n

\[
\text{OUT}[n] = \bigcup_{n' \in \text{successors}(n)} \text{IN}[n']
\]

\[
\text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]
Live variables analysis: example

\[ \text{OUT}[n] = \bigcup_{n' \in \text{successors}(n)} \text{IN}[n'] \]

\[ \text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \]

<table>
<thead>
<tr>
<th>n</th>
<th>GEN[n]</th>
<th>Kill[n]</th>
<th>IN[n]</th>
<th>OUT[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(\emptyset)</td>
<td>{j}</td>
<td>(\emptyset)</td>
<td>{j}</td>
</tr>
<tr>
<td>B2</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B3</td>
<td>(\emptyset)</td>
<td>{j}</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>B4</td>
<td>(\emptyset)</td>
<td>{j}</td>
<td>(\emptyset)</td>
<td>{j}</td>
</tr>
<tr>
<td>B5</td>
<td>{j}</td>
<td>{m}</td>
<td>{j}</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

IN[n] = a set of live variables before n
OUT[n] = a set of live variables after n
KILL[n] = a set of variables defined in node n
GEN[n] = a set of variables used in node n
Reaching definitions vs. live variables

**Reaching definitions**

- **Facts:** set of definitions
- **Direction:** forward
- **Join operator:** $\cup$
- **Transfer functions:**
  - $\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$
  - $\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$

**Live variables**

- **Facts:** set of variables
- **Direction:** backward
- **Join operator:** $\cup$
- **Transfer functions:**
  - $\text{OUT}[n] = \bigcup_{n' \in \text{successors}(n)} \text{IN}[n']$
  - $\text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$
Classifying all four dataflow analyses

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching Definitions</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live Variables</td>
<td>Very Busy Expressions</td>
</tr>
</tbody>
</table>

- Forward = Data flow from in to out
- Backward = Data flow from out to in
- Must = At join point, property must hold on all paths that are joined
- May = At join point, property may hold on some paths that are joined
Topics

• Abstract syntax tree (AST)
• Control-flow graph (CFG)
• Control-flow-based code coverage
• Data-flow analysis
• Data-flow-based code coverage
Dataflow-based code coverage

• Why another family of code coverage?

• A family of dataflow criteria is then defined, each providing a different degree of **data** coverage
  • Existing control-flow coverage criteria only consider the execution paths (**structure**)
  • In the program paths, which variables are defined and then used should also be covered (**data**)

Are **test1** and **test2** always identical?

Although the paths are the same, different tests may have different variable values defined/used!
Def-clear path

• A path \( \langle d, n_1, \ldots, n_m, u \rangle \) is a **def-clear path** from \( d \) to \( u \) with respect to \( v \) if it has no variable re-definition of \( v \) on the path
  • I.e., the definition of \( v \) at \( d \) can reach \( u \)
DU-pair

• A **DU-pair** with respect to a variable \( v \) is a pair \((d, u)\) such that
  
  • \( d \) is a node defining \( v \)
  
  • \( u \) is a node or edge using \( v \)
    
    • When it is a p-use of \( v \), \( u \) is an outgoing edge of the predicate statement
  
  • There is a def-clear path with respect to \( v \) from \( d \) to \( u \)
DU-path

• A path \( \langle n_1, \ldots, n_j, n_k \rangle \) is a **DU-path** for variable \( v \) if \( n_1 \) contains a definition of \( v \) and either
  • \( n_k \) is a c-use of \( v \) and \( \langle n_1, \ldots, n_j, n_k \rangle \) is a def-clear **simple** path for \( v \) (all nodes, except possibly \( n_1 \) and \( n_k \), are distinct), or
  • \( \langle n_j, n_k \rangle \) is a p-use of \( v \) and \( \langle n_1, \ldots, n_j \rangle \) is a def-clear **loop-free** path for \( v \) (all nodes are distinct)

```
while(w<10)
    w+=v
return w
```

Def-clear paths
1-2-3
1-2-3-2-3
1-2-3-2-3-2-3
1-2-3-2-3-2-3-2-3
...

DU paths
1-2-3
1-2-3-2-3
1-2-3-2-3-2-3
1-2-3-2-3-2-3-2-3
...

Diagram:
- B1: read(v,w)
- B2: while(w<10)
- B3: w+=v
- B4: return w
Typical dataflow-based coverage

• Identify all DU pairs and construct test cases that cover these pairs
  • Variations with different “strength”

All-DU-Paths

All-Uses

All-Defs
Typical dataflow-based coverage: definitions

- **All-DU-paths**: for every du-pair \((d, u)\) of every variable \(v\), cover all possible def-clear DU paths from \(d\) to \(u\)

- **All-Uses**: for every du-pair \((d, u)\) of every variable \(v\), cover at least one def-clear path from \(d\) to \(u\)

- **All-Defs**: for each definition \(d\) of each variable \(v\), cover at least one du-pair for \(d\)
Typical dataflow-based coverage: example

With respect to variable v

(w should be analyzed similarly)
Typical dataflow-based coverage: example

With respect to variable $v$
($w$ should be analyzed similarly)
Typical dataflow-based coverage: example

With respect to variable $v$ ($w$ should be analyzed similarly)
Typical dataflow-based coverage: example

input(v, w)
if(w>1)
v=v+7
if(v>10)
W+=v+w
output(v,w)

<table>
<thead>
<tr>
<th>du-pair</th>
<th>path(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>&lt;1,2&gt;</td>
</tr>
<tr>
<td>(1,4)</td>
<td>&lt;1,3,4&gt;</td>
</tr>
<tr>
<td>(1,5)</td>
<td>&lt;1,3,4,5&gt;</td>
</tr>
<tr>
<td>(1,&lt;3,4&gt;)</td>
<td>&lt;1,3,4&gt;</td>
</tr>
<tr>
<td>(1,&lt;3,5&gt;)</td>
<td>&lt;1,3,5&gt;</td>
</tr>
<tr>
<td>(2,4)</td>
<td>&lt;2,3,4&gt;</td>
</tr>
<tr>
<td>(2,5)</td>
<td>&lt;2,3,4,5&gt;</td>
</tr>
<tr>
<td>(2,&lt;3,4&gt;)</td>
<td>&lt;2,3,4&gt;</td>
</tr>
<tr>
<td>(2,&lt;3,5&gt;)</td>
<td>&lt;2,3,5&gt;</td>
</tr>
</tbody>
</table>

With respect to variable v
(w should be analyzed similarly)
Typical dataflow-based coverage: example

```
if (v > 10)
  w += v + w
output(v, w)
```

```
B1 input(v, w)
  if(w > 1)

B2 v = v + 7

B3 if(v > 10)

B4 w += v + w

B5 output(v, w)
```

<table>
<thead>
<tr>
<th>du-pair</th>
<th>path(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>&lt;1,2&gt;</td>
</tr>
<tr>
<td>(1,4)</td>
<td>&lt;1,3,4&gt;</td>
</tr>
<tr>
<td>(1,5)</td>
<td>&lt;1,3,4,5&gt;</td>
</tr>
<tr>
<td>(1,&lt;3,4&gt;)</td>
<td>&lt;1,3,4&gt;</td>
</tr>
<tr>
<td>(1,&lt;3,5&gt;)</td>
<td>&lt;1,3,5&gt;</td>
</tr>
<tr>
<td>(2,4)</td>
<td>&lt;2,3,4&gt;</td>
</tr>
<tr>
<td>(2,5)</td>
<td>&lt;2,3,4,5&gt;</td>
</tr>
<tr>
<td>(2,&lt;3,4&gt;)</td>
<td>&lt;2,3,4&gt;</td>
</tr>
<tr>
<td>(2,&lt;3,5&gt;)</td>
<td>&lt;2,3,5&gt;</td>
</tr>
</tbody>
</table>

With respect to variable v
(w should be analyzed similarly)
Typical dataflow-based coverage: example

```
if (v > 10)
    w += v + w
```

```
input(v, w)
if (w > 1)
    B1
    v = v + 7
B2
if (v > 10)
    B3
    v = v + 7
    B4
    w += v + w
B5
output(v, w)
```

<table>
<thead>
<tr>
<th>du-pair</th>
<th>path(s)</th>
<th>All-Defs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>&lt;1,2&gt;</td>
<td>x</td>
</tr>
<tr>
<td>(1,4)</td>
<td>&lt;1,3,4&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,5)</td>
<td>&lt;1,3,4,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,4&gt;)</td>
<td>&lt;1,3,4&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,5&gt;)</td>
<td>&lt;1,3,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(2,4)</td>
<td>&lt;2,3,4&gt;</td>
<td>x</td>
</tr>
<tr>
<td>(2,5)</td>
<td>&lt;2,3,4,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(2,&lt;3,4&gt;)</td>
<td>&lt;2,3,4&gt;</td>
<td></td>
</tr>
<tr>
<td>(2,&lt;3,5&gt;)</td>
<td>&lt;2,3,5&gt;</td>
<td></td>
</tr>
</tbody>
</table>

With respect to variable v
(w should be analyzed similarly)
Typical dataflow-based coverage: example

<table>
<thead>
<tr>
<th>du-pair</th>
<th>path(s)</th>
<th>All-Defs</th>
<th>All-Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>&lt;1,2&gt;</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>(1,4)</td>
<td>&lt;1,3,4&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(1,5)</td>
<td>&lt;1,3,4,5&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,4&gt;)</td>
<td>&lt;1,3,4&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,5&gt;)</td>
<td>&lt;1,3,5&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(2,4)</td>
<td>&lt;2,3,4&gt;</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>(2,5)</td>
<td>&lt;2,3,4,5&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(2,&lt;3,4&gt;)</td>
<td>&lt;2,3,4&gt;</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(2,&lt;3,5&gt;)</td>
<td>&lt;2,3,5&gt;</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

With respect to variable \( v \)

(\( w \) should be analyzed similarly)
Typical dataflow-based coverage: example

With respect to variable \( v \)

\((w\text{ should be analyzed similarly})\)
Typical dataflow-based coverage: example

```
if(v>10)
  w+=v+w
output(v, w)
if(w>1)
  B1
  B2 v=v+7
  B3 if(v>10)
  B4 W+=v+w
  B5 output(v,w)
```

<table>
<thead>
<tr>
<th>du-pair</th>
<th>path(s)</th>
<th>Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>&lt;1,2&gt;</td>
<td>X</td>
</tr>
<tr>
<td>(1,4)</td>
<td>&lt;1,3,4&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,5)</td>
<td>&lt;1,3,4,5&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;1,3,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,4&gt;)</td>
<td>&lt;1,3,4&gt;</td>
<td></td>
</tr>
<tr>
<td>(1,&lt;3,5&gt;)</td>
<td>&lt;1,3,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(2,4)</td>
<td>&lt;2,3,4&gt;</td>
<td>X</td>
</tr>
<tr>
<td>(2,5)</td>
<td>&lt;2,3,4,5&gt;</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>&lt;2,3,5&gt;</td>
<td></td>
</tr>
<tr>
<td>(2,&lt;3,4&gt;)</td>
<td>&lt;2,3,4&gt;</td>
<td>X</td>
</tr>
<tr>
<td>(2,&lt;3,5&gt;)</td>
<td>&lt;2,3,5&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Test1: 1-2-3-4-5

Only All-Defs, needs more tests!

With respect to variable v
(w should be analyzed similarly)
More dataflow coverage

• **All-P-Uses/Some-C-Uses**: for each definition $d$ of each variable $v$, cover at least one def-clear path from $d$ to *any* p-use of $v$
  • If no p-use of $v$, at least one def-clear path to *one* c-use of $v$ must be covered

• **All-C-Uses/Some-P-Uses**: for each definition $d$ of each variable $v$, cover at least one def-clear path from $d$ to *any* c-use of $v$
  • If no c-use of $v$, at least one def-clear path to *one* p-use of $v$ must be covered

• **All-P-Uses**: for each definition $d$ of each variable $v$, cover at least one def-clear path from $d$ to *any* p-use of $v$

• **All-C-Uses**: for each definition $d$ of each variable $v$, cover at least one def-clear path from $d$ to *any* c-use of $v$
Coverage subsumption graph

Path Coverage

Branch Coverage

Statement Coverage

All-DU-Paths

All-Uses

All-P-Uses/
Some-C-Uses

All-P-Uses

Mutation Testing

All-Defs

All-C-Uses
Mutation Testing (aka Mutation Analysis)

- Mutation testing changes programs to generate **mutants**, each of which is the same as the original program but with an **artificial bug** injected.
- A mutant is **killed** when its artificial bug can be detected by the tests.

**Mutation operators**: the patterns used to create artificial bugs

Original Program $P$

```java
if (mTimeZoneForced) {
    calendar.getTimeInMillis();
    calendar = (Calendar) calendar.clone();
    calendar.setTimeZone(mTimeZone);
}
```

$M_1$

```java
if (!mTimeZoneForced) {
    calendar.getTimeInMillis();
    calendar = (Calendar) calendar.clone();
    calendar.setTimeZone(mTimeZone);
}
```

$M_2$

```java
if (mTimeZoneForced) {
    calendar.getTimeInMillis();
    calendar = (Calendar) calendar.clone();
    calendar.setTimeZone(mTimeZone);
}
```

$M_3$
Does Mutation Testing work?

“Mutation testing is more powerful than statement or branch coverage.”
Walsh, PhD thesis, State University of NY at Binghampton, 1985

“Mutation testing is superior to data-flow coverage criteria.”

“Generated mutants are similar to real faults.”
Andrews, Briand, Labiche, ICSE 2005

“Mutants can substitute real faults in software testing experimentation.”
Just, Jalali, Inozemtseva, Ernst, Holmes, and Fraser, FSE 2014
Interprocedural analysis

• So far, all the program analyses we covered are **intraprocedural**
  • Analyzing each function (a.k.a, method/procedure) separately

• However, real-world programs usually involve the connection of a large number of functions, thus we need **interprocedural** analysis:
  • **Call-graph analysis**: analyzing the potential invocation relationship between different functions [Tip et al.]
  • **Interprocedural CFG**: connecting intraprocedural CFGs with call-graph
  • **Interprocedural dataflow analysis**: analyzing dataflow across functions [Reps et al.]
  • **Taint analysis**: tracking how private information flows through the program and if it is leaked to public observers [Arzt et al.]

- Tip et al., Scalable Propagation-Based Call Graph Construction Algorithms, 2000, OOPSLA
- Reps et al., Precise Interprocedural Dataflow Analysis via Graph Reachability, 1987, POPL
Do I need to implement such basic program analyses from scratch?

• Java
  • ASM (https://asm.ow2.io/): A lightweight bytecode-level analysis and manipulation framework
  • Soot (https://github.com/soot-oss/soot): An Intermediate Representation (IR) level analysis and manipulation framework
  • Wala (https://github.com/wala/WALA): An IR-level analysis and manipulation (via Shrike) framework for Java and JavaScript
  • Eclipse JDT (https://www.eclipse.org/jdt/): A source-level code analysis and manipulation framework
  • PITest (https://pitest.org/): State-of-the-art mutation testing engine

• C/C++
  • LLVM (http://llvm.org/): Highly customizable and modular compiler framework
Further readings

• Aho et al., Compilers: Principles, Techniques, and Tools (2nd Edition)
• Rapps and Weyuker. Selecting Software Test Data Using Data Flow Information. 1985, TSE
• Ferrante et al., The program dependence graph and its use in optimization, 1987, TOPLAS
• Horwitz et al., Interprocedural slicing using dependence graphs, 1988, PLDI
• Reps et al., Precise Interprocedural Dataflow Analysis via Graph Reachability, 1995, POPL
• Jia and Harman, An analysis and survey of the development of mutation testing, 2010, TSE
Thanks and stay safe!