

Software Testing and Validation

A.A. 2022/2023

Corso di Laurea in Informatica

Finite Models of Software

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Models in Testing

- Model checking is based on models of the artifact, testing addresses the artifact
- However, some modeling is often required also for testing
 - models for the environment (i.e., what is providing inputs)
 - models for plant, when the software is a controller
 - in some cases, testing on the final product in its “natural” environment only may be also dangerous
 - e.g., testing of the controller for a flying aircraft
 - models of the software itself
 - UML diagrams
 - control flow diagrams et al. (will be defined in the following)
 - help in devising better tests
- May be already available from specifications, or a modeling phase may be needed



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Models Must Be...

- Compact, i.e., understandable
 - often, they are for human inspection
 - if models are for some automatic procedure, then they must be manipulable in the given computational resources
 - this is exactly the case for model checking!
- Predictive, i.e., not too simple
 - at least be able to detect what is “bad” and what is “good”
 - different models may be used for the same artifact, when testing different aspects
 - e.g., model to predict airflow w.r.t. efficient passenger loading and safe emergency exit



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Models Must Be...

- Semantically meaningful
 - given something went bad, we need to understand why
 - identify the part with the failure
- Sufficiently general
 - not too specilized on some characteristics
 - otherwise, not useful
 - e.g., a C program analyzer which only works for programs without pointers



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite Abstraction of Behaviour

- Given a program, a state is an assignment for all variables in the program
 - state space*: set of all possible states
- A behaviour is a sequence of states, interleaved by program statements being executed
- The number of behaviours for non-trivial programs is extremely huge
 - infinite if we do not consider machine limitations
 - e.g., integers need not to be represented on maximum 64 bits
- An *abstraction* is a function from states to (reduced) states
 - some detail are suppressed
 - e.g., some variables are not considered



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite Abstraction of Behaviour

- Two different states may be considered the same by an abstraction
 - e.g., they differ by some variable, which is abstracted out
- States sequence may be squeezed
- Non-determinism may be introduced
 - e.g., a choice was made by considering the value of some abstracted-out variable
- In model checking, this is done by hand for each system
 - here, instead, we will consider some standard models which are especially tailored for testing
 - in some cases, they may be automatically extracted from code

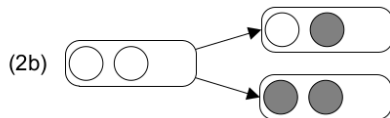
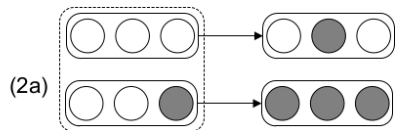
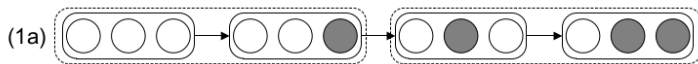


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite Abstraction of Behaviour



(Intraprocedural) Control Flow Graphs

- Model close to the actual program source code
 - finite by construction
- Often compilers are also able to build the control flow graph
 - e.g., `gcc -fdump-tree-cfg`
- Directed graph:
 - nodes are program statements or group of statements
 - more on this in the following
 - edges represent the possibility to go from a node to another
 - either by branch or by sequential execution



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Flow Graphs (CFGs)

- Nodes usually are a maximal group of statements with a single entry and single exit
 - *basic block*
 - e.g., sequential assignments are grouped together
- On the contrary, it may happen that a single statement is broken down
 - e.g., `if (++i > 3)` becomes `i++`; `if (i > 3)`
 - e.g., the `for` statement
 - e.g., short-circuit evaluation
 - it depends on the level of accuracy needed

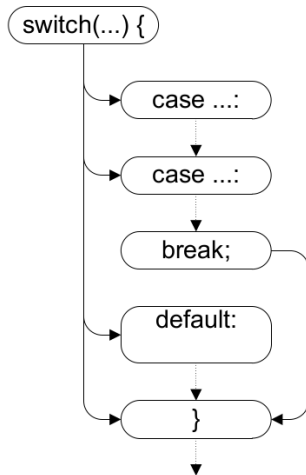
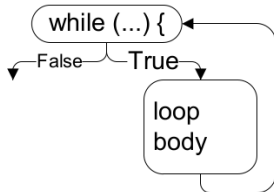
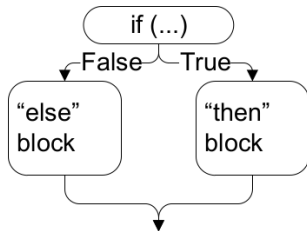


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Flow Graphs



DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Flow Graphs

```
1  /**
2   * Remove/collapse multiple newline characters.
3   *
4   * @param String string to collapse newlines in.
5   * @return String
6   */
7  public static String collapseNewlines(String argStr)
8  {
9      char last = argStr.charAt(0);
10     StringBuffer argBuf = new StringBuffer();
11
12     for (int cldx = 0 ; cldx < argStr.length(); cldx++)
13     {
14         char ch = argStr.charAt(cldx);
15         if (ch != ' \n' || last != ' \n')
16         {
17             argBuf.append(ch);
18             last = ch;
19         }
20     }
21
22     return argBuf.toString();
23 }
```

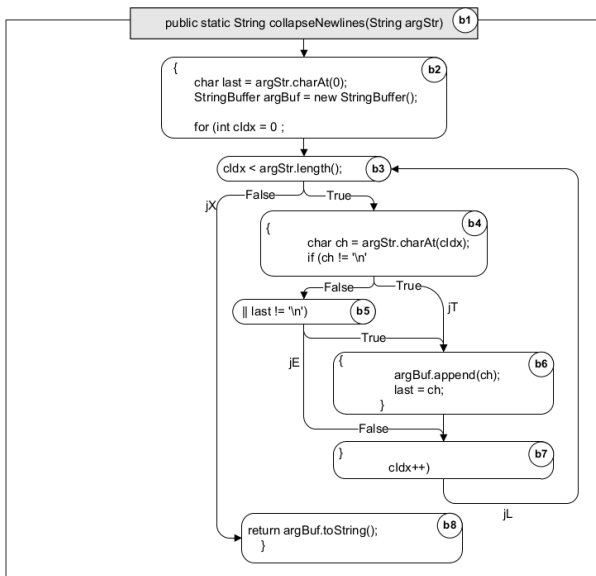


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Flow Graphs



Control Flow Graphs

- Let P be a (part of a) function or procedure for which testing must be performed
 - white-box testing: we know the code of P as a sequence $\mathcal{C}(P) = \langle l_1, \dots, l_k \rangle$ of statements
 - we assume P is written in some imperative language
 - we assume that complex statements in $\mathcal{C}(P)$ are already broken down in parts
 - short-circuited conditions, inline increments, function/procedure calls...
- Let $g = \langle i_1, \dots, i_m \rangle$ be a grouping for the statements of $\mathcal{C}(P)$
 - $1 \leq i_j < i_{j+1} \leq k$ for all $j = 1, \dots, m - 1$
 - e.g., for $g = \langle 3, 5, 10 \rangle$ we will consider three blocks:
 - the first 3 statements, then other two statements, and finally the remaining 5 statements



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Flow Graphs

- A CFG for a program P with granularity g is a graph $G = (V, E)$ s.t.
 - $V = \{\langle I_{g_{i-1}+1} \dots I_{g_i} \rangle \mid i = 1, \dots, |g|\}$
 - with $g_0 = 0$
 - nodes are basic blocks
 - $E = \{(u, v) \mid u, v \in V \wedge \text{control flow from last statement of } u \text{ and first of } v \text{ may take place}\}$
- Typically, nodes $v_i \in V$ are labeled with the corresponding basic block $\langle I_{g_{i-1}+1} \dots I_{g_i} \rangle$
- Typically, edges $(u, v) \in E$ may be labeled by a boolean value if flow from u to v is conditioned
 - last statement of u is an if or a while
 - and similar, e.g., for, until etc
- In some cases, some alphanumeric label is added to ease references



From CFG to LCSAJ

- Linear code sequences and jumps
 - maximal sequences of consecutives statements
 - may be directly derived from a CFG
 - very useful for testing coverage criteria

<i>From</i>	<i>Sequence of Basic Blocks</i>							<i>To</i>
entry	b1	b2	b3					jX
entry	b1	b2	b3	b4				jT
entry	b1	b2	b3	b4	b5			jE
entry	b1	b2	b3	b4	b5	b6	b7	jL
jX							b8	return
jL			b3	b4				jT
jL			b3	b4	b5			jE
jL			b3	b4	b5	b6	b7	jL



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition of LCSAJ

- Let $G = (V, E, L_1, L_2)$ be a labeled CFG
 - $L_1 : V \rightarrow \mathcal{L}_V, L_2 : E \rightarrow \mathcal{L}_E$ are two labeling functions for nodes (basic blocks) and edges, respectively
- The ICSAJ associated to G is
$$\mathcal{I}(G) \subseteq \{ \langle l_1, l_2, l_3 \rangle \mid l_1, l_3 \in \mathcal{L}_E, l_2 \in \mathcal{L}_V^* \} \text{ s.t.:}$$
 - l_1 is connected to the first statement of l_2
 - l_3 is connected from the last statement of l_2
 - l_2 only contain consecutive statements in $\mathcal{C}(P)$



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Call Graphs

- CFG is typically intraprocedural; call graphs are interprocedural
 - simply a graph where nodes are defined functions
 - there is an edge from f to g iff f *may* call g
 - thus, they may contain calls which are actually never made
 - sometimes arguments are made explicit
 - number of paths inside a call graph may be exponential, even without recursion



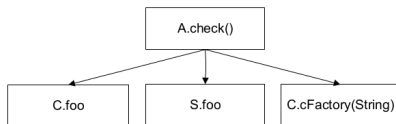
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Call Graphs

```
1 public class C {
2
3     public static C cFactory(String kind) {
4         if (kind == "C") return new C();
5         if (kind == "S") return new S();
6         return null;
7     }
8
9     void foo() {
10         System.out.println("You called the parent's method");
11     }
12
13     public static void main(String args[]) {
14         (new A()).check();
15     }
16 }
17
18 class S extends C {
19     void foo() {
20         System.out.println("You called the child's method");
21     }
22 }
23
24 class A {
25     void check() {
26         C myC = C.cFactory("S");
27         myC.foo();
28     }
29 }
```



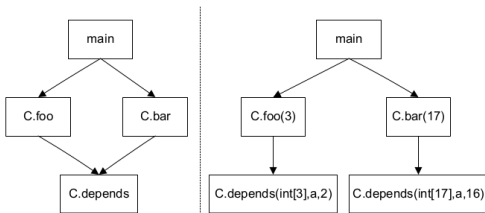
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Call Graphs

```
1 public class Context {  
2     public static void main(String args[]) {  
3         Context c = new Context();  
4         c.foo(3);  
5         c.bar(17);  
6     }  
7  
8     void foo(int n) {  
9         int[] myArray = new int[ n ];  
10        depends( myArray, 2 );  
11    }  
12  
13    void bar(int n) {  
14        int[] myArray = new int[ n ];  
15        depends( myArray, 16 );  
16    }  
17  
18    void depends( int[] a, int n ) {  
19        a[n] = 42;  
20    }  
21 }
```

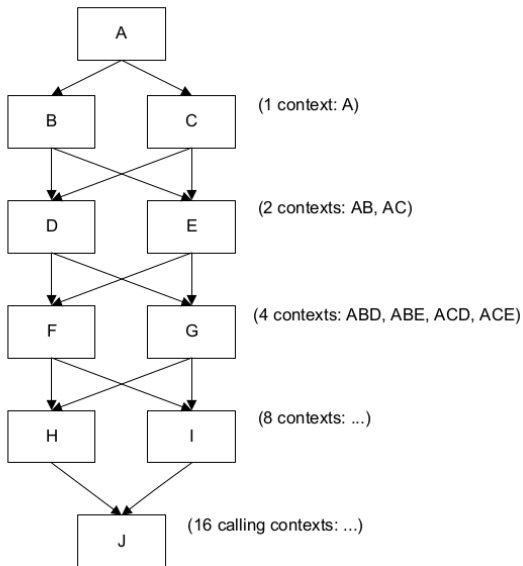


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Call Graphs



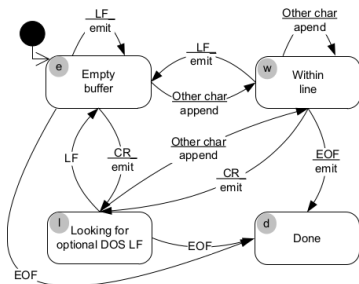
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite State Machines

- Here we will focus on Mealy Machines
 - a graph where nodes are “modalities” of a given software
 - edges are labeled with input/output



	LF	CR	EOF	other
e	e / emit	l / emit	d / -	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / -		d / -	w / append



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite State Machines

```
1  /* Convert each line from standard input */
2  void transduce() {
3
4      #define BUFLen 1000
5      char buf[BUFLen]; /* Accumulate line into this buffer */
6      int  pos = 0;      /* Index for next character in buffer */
7
8      char inChar; /* Next character from input */
9
10     int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12     while ((inChar = getchar()) != EOF ) {
13         switch (inChar) {
14             case LF:
15                 if (atCR) { /* Optional DOS LF */
16                     atCR = 0;
17                 } else { /* Encountered CR within line */
18                     emit(buf, pos);
19                     pos = 0;
20                 }
21                 break;
22             case CR:
23                 emit(buf, pos);
24                 pos = 0;
25                 atCR = 1;
26                 break;
27             default:
28                 if (pos >= BUFLen-2) fail("Buffer overflow");
29                 buf[pos++] = inChar;
30         } /* switch */
31     }
32     if (pos > 0) {
33         emit(buf, pos);
34     }
35 }
```

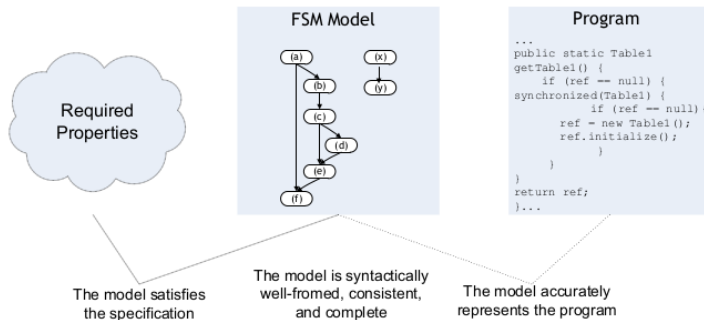


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Finite State Machines



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Mealy Machine Formal Definition

- A Mealy machine is a 6-tuple $\mathcal{M} = (S, S_0, \Sigma, \Lambda, T, G)$ consisting of the following:
 - a finite set of states S
 - a start state (also called initial state) $S_0 \in S$
 - a finite set called the input alphabet Σ
 - a finite set called the output alphabet Λ
 - a (deterministic!) transition function $T : S \times \Sigma \rightarrow S$ mapping pairs of a state and an input symbol to the corresponding next state
 - an output function $G : S \times \Sigma \rightarrow \Lambda$ mapping pairs of a state and an input symbol to the corresponding output symbol.
- Given an input $w \in \Sigma^*$, \mathcal{M} outputs $o \in \Lambda^*$, $|o| = |w|$ s.t.
 - $\forall i = 1, \dots, |w|. s_i = T(s_{i-1}, w_i) \wedge o_i = G(s_{i-1}, w_i)$
 - $s_0 = S_0$



Data Flow Models

- CFGs, FSMs etc are a good way to represent *control flow*
- What about *data flow*?
- Again, ideas are borrowed from compilers theory
 - data flow is used to detect errors for type checking, or also opportunities for optimization
 - also used in software engineering tout court, for refactoring or reverse engineering
- As for testing, useful for:
 - select test cases based on dependence information
 - detect anomalous patterns that indicate probable programming errors, e.g. usage of uninitialized values



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

- Definition of a variable: either its declaration or a write access
 - for languages like Python, only write access...
 - write access may be:
 - left part of an assignment
 - parameter initialization in function calls
 - other special cases such as ++ construct in C-like languages
- Use of a variable: a read access
 - right part of an assignment
 - variable passed in function calls
 - variable used without being modified
- The same line of code may be both definition and use
 - typically, nearly all lines either define and/or use at least one variable



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

```
1  public int gcd(int x, int y) {  
2      int tmp;  
3      while (y != 0) {  
4          tmp = x % y;  
5          x = y;  
6          y = tmp;  
7      }  
8      return x;  
9  }
```

/ A: def x,y */*
/ def tmp */*
/ B: use y */*
/ C: use x,y, def tmp */*
/ D: use y, def x */*
/ E: use tmp, def y */*

/ F: use x */*



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

- For a given definition, there may be many uses, and viceversa
 - of course, for a fixed variable
- A definition-use pair combines a given use with the *closest* definition
 - w.r.t. some possible execution (*path*) of the code
- Other definitions behind the closest one are *killed*



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



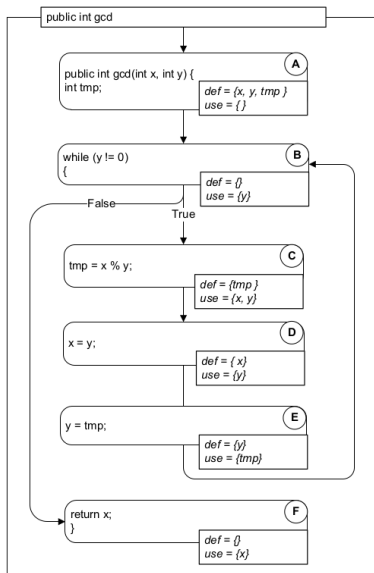
DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

- Consider an execution path $\pi = s_1, \dots, s_k$:
 - s_i are *statements* and s_i, s_{i+1} may be contiguous in π iff the control flow may go from s_i to s_{i+1}
 - e.g., from the previous code: 1,2,3,8,9 and 1,2,3,4,5,6,7,3,4,5,6,7,8,9
- Consider an execution path $\pi = s_1, \dots, s_k$:
 - if $\exists k. \text{use}(v) \in s_k$, let $L = \{\ell < k \mid \text{def}(v) \in s_\ell\}$
 - $(d, u) = (\max L, k)$ is a definition-use pair
 - v_d reaches u or v_d is a *reaching definition* of u
 - s_ℓ is a *killed definition* if $\ell \in L \wedge \ell \neq \max L$
 - the sub-path $s_\ell \dots s_k$ is *definition-clear*



Definition-Use Pairs



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

- Use-definition pairs defines a *direct data dependence*, can be used to build the *data dependence graph*
 - there is an edge (s, t) with label v iff (s, t) is a definition-use pair for variable v (for some path)
- *Granularity* on nodes may be tuned according to needs:
 - single expressions (especially for compilers)
 - statements (figure below)
 - basic blocks
 - etc



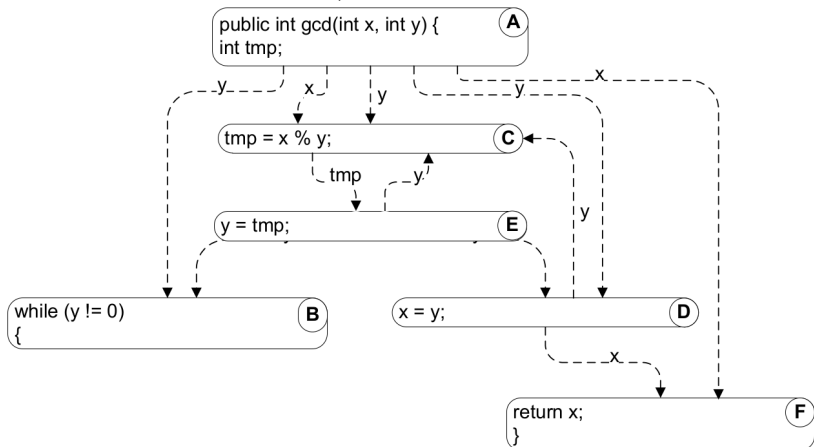
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Definition-Use Pairs

One error and one omission, where?



Algorithm to Generate All Definition-Use Pairs

Algorithm *Reaching definitions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$
 $\text{pred}(n) = \{m \in \text{nodes} \mid (m, n) \in \text{edges}\}$
 $\text{succ}(m) = \{n \in \text{nodes} \mid (m, n) \in \text{edges}\}$
 $\text{gen}(n) = \{v_n\}$ if variable v is defined at n , otherwise $\{\}$
 $\text{kill}(n) =$ all other definitions of v if v is defined at n , otherwise $\{\}$

Output: $\text{Reach}(n)$ = the reaching definitions at node n

```
for  $n \in \text{nodes}$  loop
     $\text{ReachOut}(n) = \{\}$  ;
end loop;
workList = nodes ;
while (workList  $\neq \{\}$ ) loop
    // Take a node from worklist (e.g., pop from stack or queue)
     $n =$  any node in workList ;
    workList = workList  $\setminus \{n\}$  ;

    oldVal =  $\text{ReachOut}(n)$  ;

    // Apply flow equations, propagating values from predecessors
     $\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$ ;
     $\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$  ;
    if (  $\text{ReachOut}(n) \neq \text{oldVal}$  ) then
        // Propagate changed value to successor nodes
        workList = workList  $\cup \text{succ}(n)$ 
    end if;
end loop;
```



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



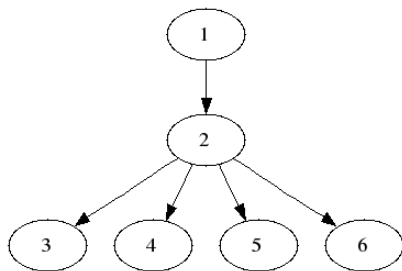
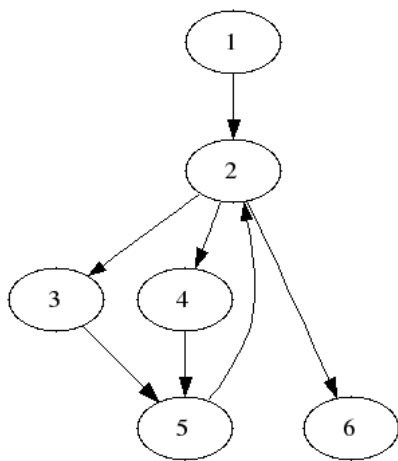
DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Dependence

- *Control dependence graph*
 - nodes are statements, but again granularity may change
 - to define edges, the notion of *dominators* is needed
 - a node n is dominated by node m iff, for all paths π from the root to n , m is also in π
 - the (unique) *immediate dominator* of n is the closest dominator of n
 - i.e., with the minimum path to reach n
 - also stated as: the dominator of n which does not dominate any other dominator of n
 - *dominator tree*: there is an edge (s, t) iff s is the immediate dominator of t
 - for all reachable nodes there is exactly one immediate dominator
 - *post-dominators*: same definition, but in the reverse graph
 - an exit node must be present



Dominators



Control Dependence

- Back to the control dependence graph: given nodes s, t , we have that (s, t) is an edge iff t is *control dependent* on s
- To define when t is control dependent on s , the following holds:
 - t is reached on all execution paths
 - then, t is control dependent on the root only
 - it may actually be the root itself
 - t is reached on some but not all execution paths; then for s the following must hold:
 - the outgoing degree of s in the CFG is at least 2
 - one of the successors of s in the CFG is post-dominated by t
 - s is not post-dominated by t



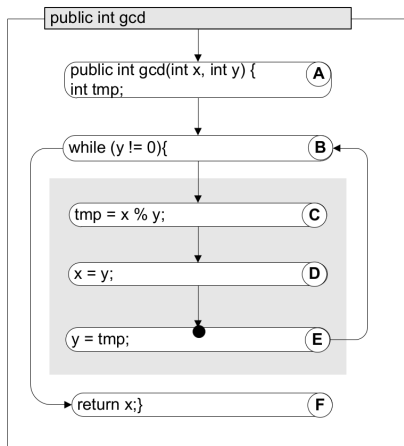
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Dependence

Proof that B is control dependent on E



Gray region: nodes
post-dominated by
 E

Node B has
successors both
within and outside
the gray region
→ E is control-
dependent on B



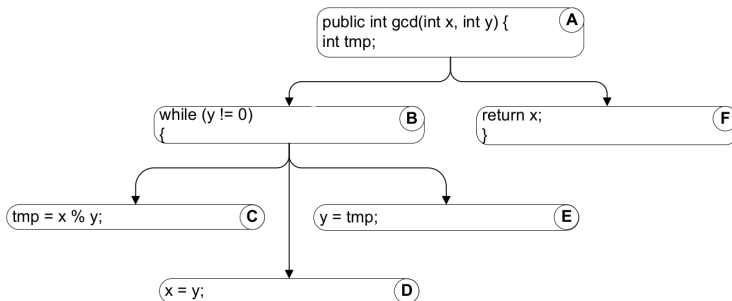
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Control Dependence

Full control graph



Available Expressions

- Other uses of the control flow graph: available expressions
 - again, mutated from compilers: when a given expression can be evaluated just once and stored for later use
- An expression E is:
 - *generated* when its value is computed
 - *killed* when at least one of the variables involved changes its value
 - not necessarily by assignments, could be a side effect of a function call...
 - *available* at some point p iff, for paths π from start to p , E is generated but not killed in π
- Algorithm is very similar to the reaching definitions algorithm:
 - for available expressions, is a forward all-paths analysis
 - for reaching definitions, is a forward any-path analysis



Algorithm to Generate All Available Expressions

Algorithm *Available expressions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$, with a distinguished root node *start*.

$\text{pred}(n) = \{m \in \text{nodes} \mid (m, n) \in \text{edges}\}$

$\text{succ}(m) = \{n \in \text{nodes} \mid (m, n) \in \text{edges}\}$

$\text{gen}(n)$ = all expressions e computed at node n

$\text{kill}(n)$ = expressions e computed anywhere, whose value is changed at n ;

$\text{kill}(\text{start})$ is the set of all e .

Output: $\text{Avail}(n)$ = the available expressions at node n

for $n \in \text{nodes}$ **loop**

$\text{AvailOut}(n)$ = set of all e defined anywhere ;

end loop;

$\text{workList} = \text{nodes}$;

while ($\text{workList} \neq \{\}$) **loop**

// Take a node from worklist (e.g., pop from stack or queue)

n = any node in workList ;

$\text{workList} = \text{workList} \setminus \{n\}$;

$\text{oldVal} = \text{AvailOut}(n)$;

// Apply flow equations, propagating values from predecessors

$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$;

$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$;

if ($\text{AvailOut}(n) \neq \text{oldVal}$) **then**

// Propagate changes to successors

$\text{workList} = \text{workList} \cup \text{succ}(n)$

end if;

end loop;



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

From Available Expressions to General Properties

- G occurs on all execution paths leading to U , and there is no intervening occurrence of K between the last occurrence of G and U
- For any G, K, U , we can use the algorithm above
- E.g., it can be used for the Java initialized variables problem
 - before any use, a variable must be initialized
 - the kill set is not empty only in the starting node, as we cannot uninitialized a variable
 - also a warning in C



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Example for Uninitialized Variables

```
static void questionable() {  
    int k;  
    for (int i=0; i < 10; ++i) {  
        if (someCondition(i)) {  
            k = 0;  
        } else {  
            k += i;  
        }  
    }  
    System.out.println(k);  
}
```

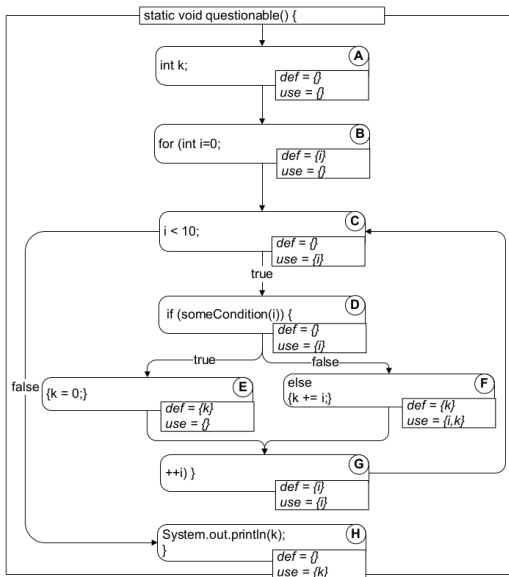


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Example for Uninitialized Variables

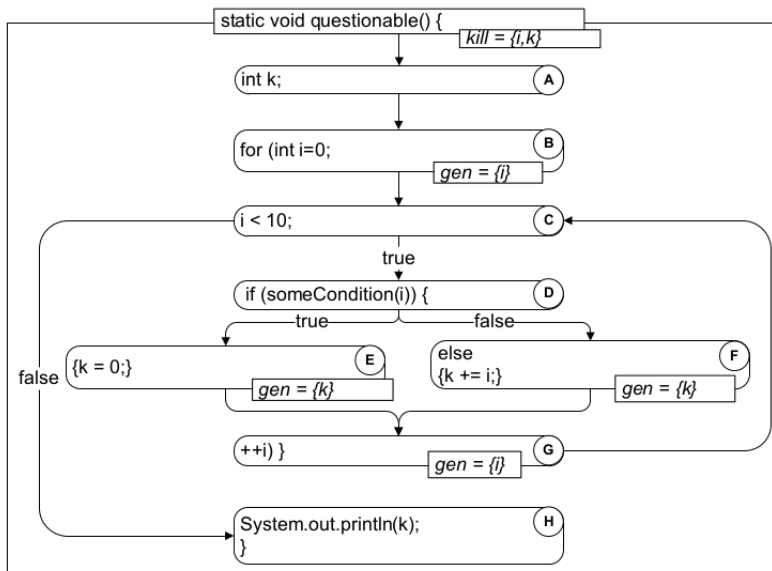


UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Example for Uninitialized Variables



Live Variables: Backward Analysis

- A variable v is *live* at point p iff, on some path π from p to the end, a read on v occurs before a write on v
 - `b = 5 /* b is live, c is not */; c = 3 /* now, both b and c are live */; a = f(b, c)`
- From the definition above, we need a backward-any path algorithm
- General property: after D occurs, there is at least one execution path on which G occurs with no intervening occurrence of K
- Example: useless definitions, i.e., assign define a variable which is never used afterwards
 - important for languages where declarations are not used, e.g., Python and Perl
 - not an error, but probably the variable is misspelled



UNIVERSITÀ
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Example for Useless Variables

```
class SampleForm(FormData):
```

```
    """ Used with Python cgi module  
        to hold and validate data  
        from HTML form """
```

```
    fieldnames = ('name', 'email', 'comment')
```

```
    # Trivial example of validation. The bug would be  
    # harder to see in a real validation method.
```

```
    def validate(self):
```

```
        valid = 1;
```

```
        if self.name == " " : valid = 0
```

```
        if self.email == " " : valid = 0
```

```
        if self.comment == " " : valid = 0
```

```
        return valid
```

Backward all-Paths Analysis

- Live variable analysis is a backward any-path analysis
- What about the missing one: backward all-paths analysis?
- After D occurs, G always occurs with no intervening occurrence of K
 - D inevitably leads to G before K
- Examples:
 - interrupts are re-enabled after interrupt routine is executed
 - files are closed on process termination
 - etc



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Generalizing Data Flow Analysis

- The examples above “simulates” the execution of a piece of code, considering its CFG
- All possible executions are considered, but only a restricted info is kept
 - “summary state”
 - e.g., a bit to denote if a variable has been defined
 - actually having all executions with full information entails all possible values for variables are considered
- By considering different kinds of summary states, we have different types of analyses
 - example: “taint mode” in the Perl language
 - blocks some sensitive operations depending on variables defined outside the Perl script itself
 - opening a file is a sensitive operation
 - thus, e.g., you cannot define a variable v by reading some input and then open a file having v as a filename



Data Flow Analysis with Arrays and Pointers

- Easy to perform data flow analysis on single variables
- When considering pointers and/or arrays, many difficulties arise
- Difficulty 1: definition-use on an array referenced by variables
 - e.g.: `a[i] = 1; k = a[j];` is a definition-use pair iff `i == j`
 - too difficult to determine if such a condition is always true, always false, or sometimes true and sometimes false
- Difficulty 2: aliases obtained by full array assignment
 - e.g., `b = a; a[2] = 42; i = b[2];` is a definition-use pair (or triple?) in Java



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Difficulty 3: Arguments Passing

```
fromCust == toCust? fromHome == fromWork? toHome ==  
toWork?
```

```
public void transfer (CustInfo fromCust, CustInfo toCust) {
```

```
    PhoneNum fromHome = fromCust.gethomePhone();
```

```
    PhoneNum fromWork = fromCust.getworkPhone();
```

```
    PhoneNum toHome = toCust.gethomePhone();
```

```
    PhoneNum toWork = toCust.getworkPhone();
```



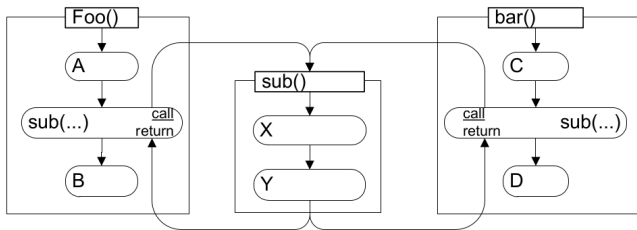
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Interprocedural Analysis

- Calls between different functions/methods, important, e.g., for the previous slide
- Simply following calls and returns in a CFG-like way is not practical: too many spurious paths
 - $(A, X, Y, B), (C, X, Y, D)$ are ok
 - $(A, X, Y, D), (C, X, Y, B)$ are impossible



Interprocedural Analysis

- To solve the problem, context is needed
 - if sub is called by A, it must return in B
- Number of contexts is exponential
 - may be ok for a small group of functions, e.g., a not-too-big single Java class
- Some special cases exist
 - the info needed to analyze the calling procedure must be small
 - e.g., proportional to the number of called procedures
 - the information about the called procedure must be context-independent
 - example: declaration of exception throwing in Java



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica