Software Testing and Validation

A.A. 2024/2025

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

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

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

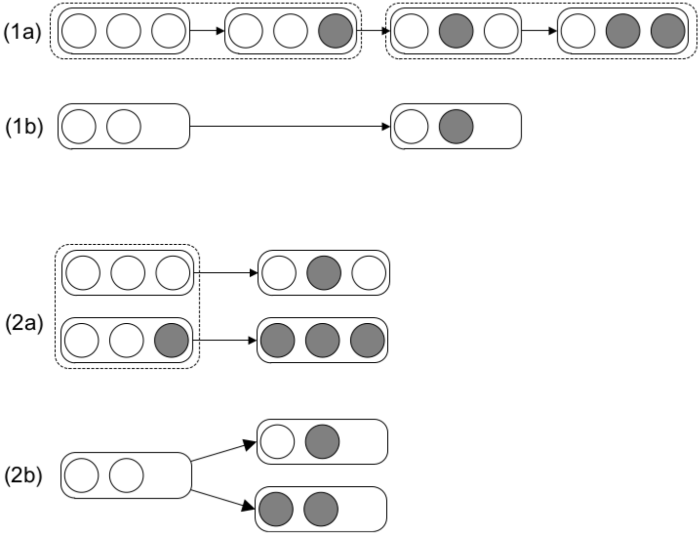
Finite Abstraction of Behaviour

- 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 details are suppressed
  - e.g., some variables are not considered

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 sequences may be squeezed
- Non-determinism may be introduced
  - e.g., when 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

Finite Abstraction of Behaviour



(Each circle is a binary variable...)

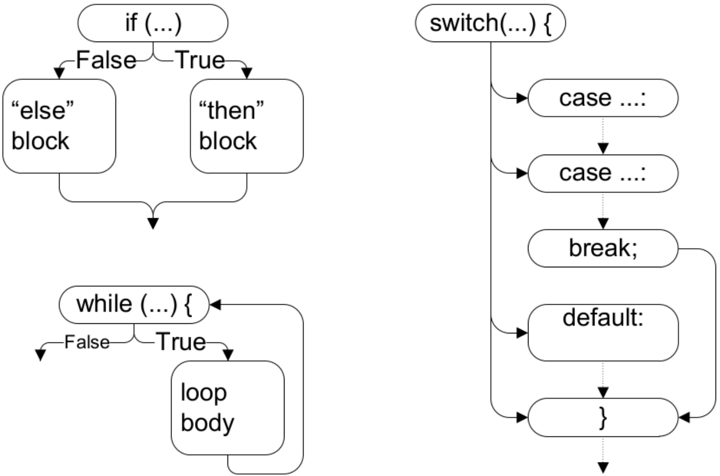
(Intraprocedural) Control Flow Graphs

- 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

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

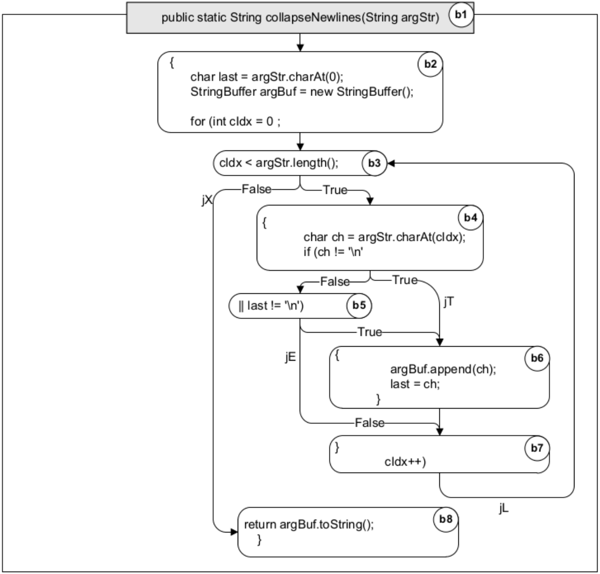
Control Flow Graphs



Control Flow Graphs

```java
/**
 * Remove/collapse multiple newline characters.
 *
 * @param String string to collapse newlines in.
 * @return String
 */
public static String collapseNewlines(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```

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 $\mathcal{C}(P) = \langle I_1, \ldots, I_k \rangle$ 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...
  - in the previous example (`collapseNewlines`), $k = 11$

Control Flow Graphs

- Let $g = \langle i_1, \ldots, i_m \rangle$ be a grouping for the statements of $\mathcal{C}(P)$
  - $1 \leq i_j < i_{j+1} \leq k$ for all $j = 1, \ldots, m-1$
  - 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
  - we will call g *granularity* for a given $\mathcal{C}(P)$
- Of course, granularities must comply with code
  - no flow branches (`if`, `while`, etc) inside a block $I_{i_j+1} \ldots I_{i_{j+1}}$
- Usually, maximal granularities are chosen
  - from a flow branch (or starting point) to another flow branch (or ending point)

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 and $|V| = |g|$
  - $E = \{(u, v) \mid u, v \in V \wedge$ control flow from last statement of u and first of v 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

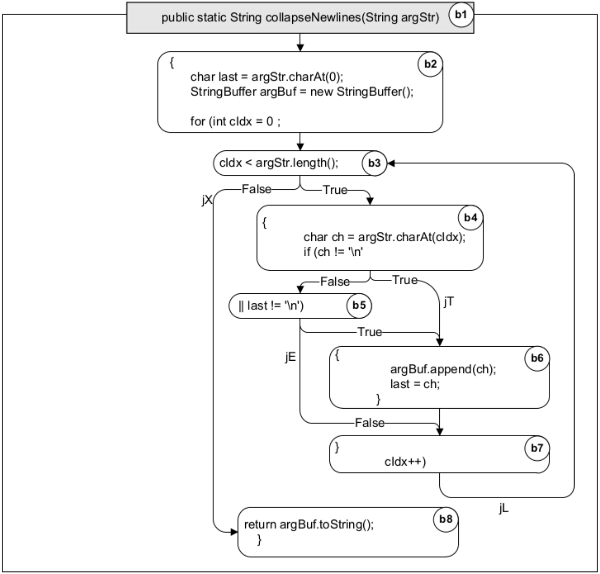
From CFG to LCSAJ

- Linear code sequences and jumps
  - maximal sequences of consecutives statements
  - may be directly derived from a CFG
- In a nutshell: all sequences of consecutive basic blocks (both in the CFG and in the source code)
  - while a basic block cannot contain branches, LCSAJ can
  - conditional branches create overlapping LCSAJs
  - basic blocks cannot overlap
  - typically, there are 4x more LCSAJs than basic blocks
- Let $G = (V, E, L_1, L_2)$ be a labeled CFG
  - $L_1 : V \to \mathcal{L}_V, L_2 : E \to \mathcal{L}_E$ are two bijective labeling functions for nodes (basic blocks) and edges, respectively
  - no really need of having the labeling function: it simply makes the LCSAJ more readable

Control Flow Graphs

```
/**
 * Remove/collapse multiple newline characters.
 *
 * @param String string to collapse newlines in.
 * @return String
 */
public static String collapseNewlines(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```

Control Flow Graphs



From CFG to LCSAJ

From	*Sequence of Basic Blocks*								*To*
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

From CFG to LCSAJ

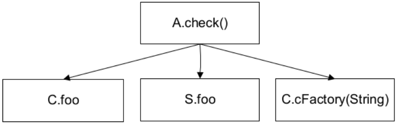
- Let $G = (V, E, L_1, L_2)$ be a labeled CFG
- The LCSAJ associated to G is $\mathcal{I}(G) = \{\langle l_1, \ell_2, l_3 \rangle \mid l_1, l_3 \in \mathcal{L}_E, \ell_2 \in \mathcal{L}_V^*\}$ s.t.:
  - l_1 arrives to the first statement of ℓ_2
    - that is: if $L_2^{-1}(l_1) = (u, v)$, then ℓ_2 begins with $L_1(v)$
  - l_3 exits from the last statement of ℓ_2
    - that is: if $L_2^{-1}(l_3) = (u, v)$, then ℓ_2 ends with $L_1(u)$
  - ℓ_2 only contain *maximal consecutive* basic blocks of $\mathcal{C}(P)$
    - that is, $\ell_2 = v_1 \ldots v_n$ implies that:
    - v_i and v_{i+1} are consecutive basic blocks both in G and in the source code for all $i = 1, \ldots, n-1$
    - v_n has no other consecutive basic block
    - i.e., either it is followed by a control flow jump or it is the end of the unit
    - v_1 is either the beginning of the unit or the destination of a control flow jump

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

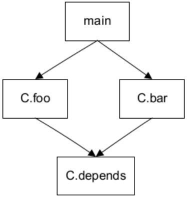
Call Graphs

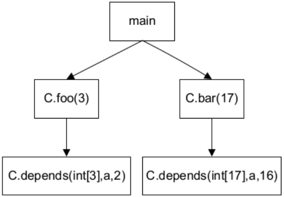
```
public class C {

    public static C cFactory(String kind) {
        if (kind == "C") return new C();
        if (kind == "S") return new S();
        return null;
    }

    void foo() {
        System.out.println("You called the parent's method");
    }

    public static void main(String args[]) {
        (new A()).check();
    }
}

class S extends C {
    void foo() {
        System.out.println("You called the child's method");
    }
}

class A {
    void check() {
        C myC = C.cFactory("S");
        myC.foo();
    }
}
```



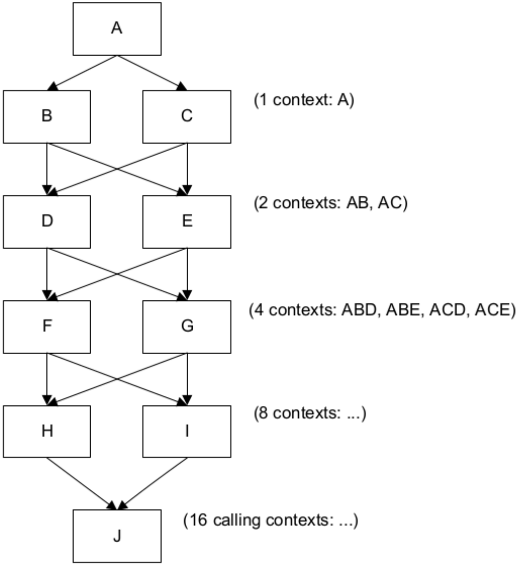
Call Graphs

```java
public class Context {
    public static void main(String args[]) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }

    void foo(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 2) ;
    }

    void bar(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 16) ;
    }

    void depends( int[] a, int n ) {
        a[n] = 42;
    }
}
```





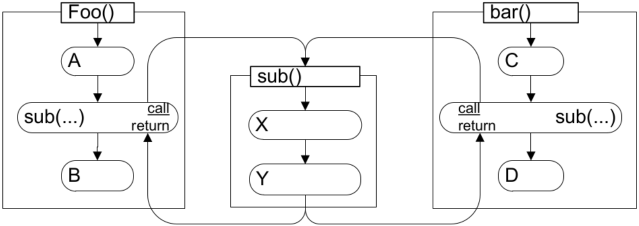
Call Graphs



A
B
C
D
E
F
G
H
I
J
(1 context: A)
(2 contexts: AB, AC)
(4 contexts: ABD, ABE, ACD, ACE)
(8 contexts: ...)
(16 calling contexts: ...)

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

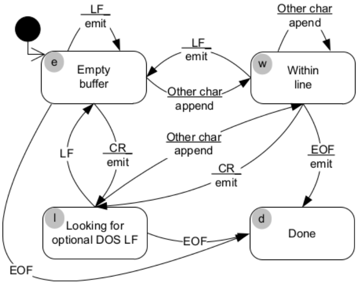


Interprocedural Analysis

- 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

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

Finite State Machines

```
/** Convert each line from standard input */
void transduce() {

  #define BUFLEN 1000
  char buf[BUFLEN];   /* Accumulate line into this buffer   */
  int   pos = 0;         /* Index for next character in buffer */

  char inChar; /* Next character from input */

  int atCR = 0; /* 0="within line", 1="optional DOS LF" */

  while ((inChar = getchar()) != EOF ) {
    switch (inChar) {
    case LF:
      if (atCR) {   /* Optional DOS LF */
        atCR = 0;
      } else {        /* Encountered CR within line */
        emit(buf, pos);
        pos = 0;
      }
      break;
    case CR:
      emit(buf, pos);
      pos = 0;
      atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    } /* switch */
  }
  if (pos > 0) {
    emit(buf, pos);
  }
}
```

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 $T : S \times \Sigma \to S$ mapping pairs of a state and an input symbol to the corresponding next state
  - an output function $G : S \times \Sigma \to \Lambda$ 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.
  - $\forall i = 1, \ldots, |w|.\ s_i = T(s_{i-1}, w_i) \wedge o_i = G(s_{i-1}, w_i)$
  - $s_0 = S_0$

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

Definition-Use Pairs

- Definition of a variable: either its declaration or a write access
  - for languages like Python, mostly 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

Definition-Use Pairs

```
public int gcd(int x, int y) {       /* A: def x,y   */
    int tmp;                         /*      def tmp  */
    while (y != 0) {                 /* B: use y      */
        tmp = x % y;                 /* C: use x,y, def tmp */
        x = y;                       /* D: use y, def x  */
        y = tmp;                     /* E: use tmp, def y */
    }
    return x;                        /* F: use x */
}
```

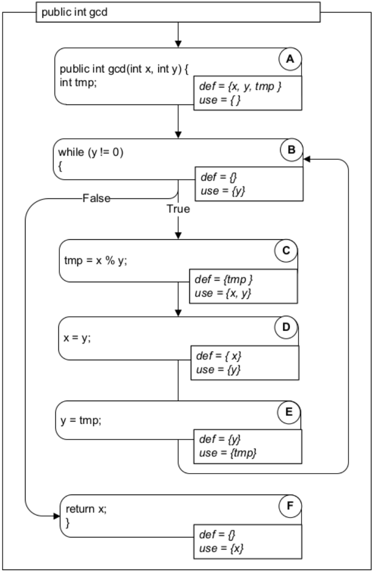
Definition-Use Pairs

- 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*

Definition-Use Pairs

- Consider an execution path $\pi = s_1, \ldots, s_k$:
  - 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 $\pi = s_1, \ldots, s_k$:
  - if $\exists k.\ \mathrm{use}(v) \in s_k$, let $L = \{\ell < k \mid \mathrm{def}(v) \in s_\ell\}$
  - $(d, u) = (\max L, k)$ 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 $s_\ell \ldots s_k$ is *definition-clear*

Definition-Use Pairs

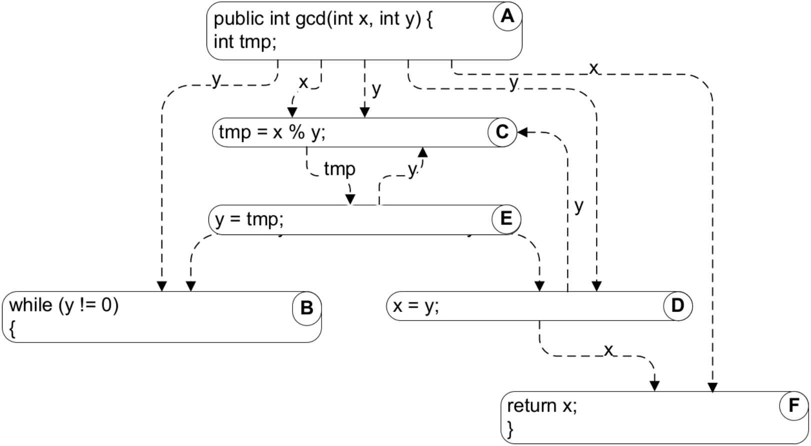


Definition-Use Pairs

- 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

Definition-Use Pairs

Not completely correct



Algorithm to Generate All Reaching Definitions

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 $\{\}$

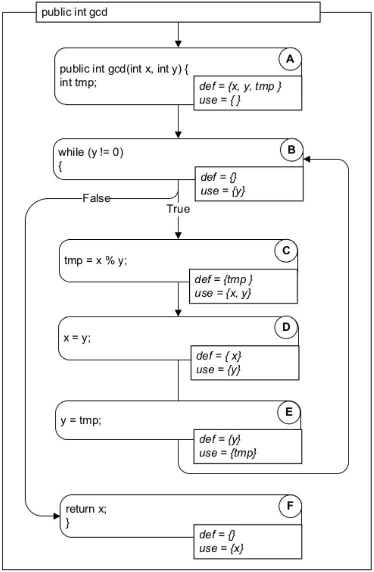
Output: $\text{Reach}(n) =$ the reaching definitions at node n

for $n \in$ nodes **loop**
    $\text{ReachOut}(n) = \{\}$;
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 ;
    $\text{workList} = \text{workList} \setminus \{n\}$;

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

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

All Reaching Definitions



A$\rightarrow \emptyset$

B$\rightarrow \{x_A, x_D, t_A, t_C, y_E, y_A\}$

C$\rightarrow \{x_A, x_D, t_A, t_C, y_E, y_A\}$

D$\rightarrow \{x_A, x_D, t_C, y_E, y_A\}$

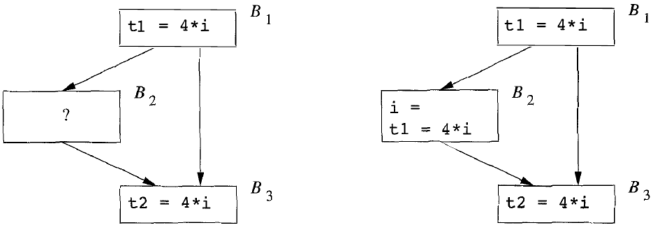
E$\rightarrow \{x_D, y_A, y_E, t_C\}$

F$\rightarrow \{x_A, x_D, t_A, t_C, y_A, y_E\}$

Available Expressions

- Other uses of the control flow graph: available expressions
  - again, mutuated from compilers: when a given expression can be evaluated just once and stored for later use
  - testing: available expressions should be always tested
- 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 all paths π from start to p, E is generated but not subsequently killed in π
- Algorithm is very similar to the reaching definitions one:
  - for available expressions, is a forward all-paths analysis
  - for reaching definitions, is a forward any-path analysis

Available Expressions



Available Expressions

Statement	Available Expressions
	$\emptyset$
`a = b + c`	
	$\{b+c\}$
`b = a - d`	
	$\{a-d\}$
`c = b + c`	
	$\{a-d\}$
`d = a - d`	
	$\emptyset$

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

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

```
for n ∈ nodes loop
    AvailOut(n) = set of all e defined anywhere ;
end loop;
workList = nodes ;
while (workList ≠ {}) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    n = any node in workList ;
    workList = workList \ {n} ;
    oldVal = AvailOut(n) ;
    // Apply flow equations, propagating values from predecessors
    Avail(n) = ⋂_{m∈pred(n)} AvailOut(m);
    AvailOut(n) = (Avail(n) \ kill(n)) ∪ gen(n) ;
    if ( AvailOut(n) ≠ oldVal ) then
        // Propagate changes to successors
        workList = workList ∪ succ(n)
    end if;
end loop;
```

Algorithm to Generate All Reaching Definitions

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 $\{\}$

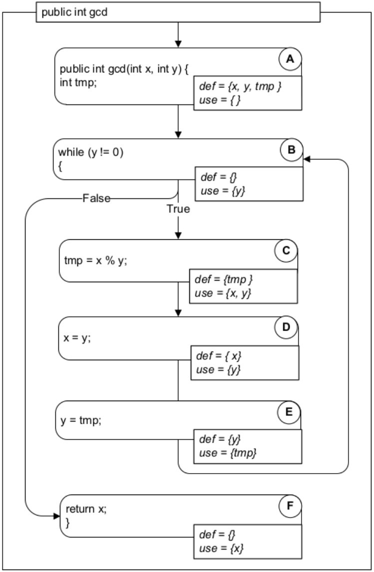
Output: $\text{Reach}(n) =$ the reaching definitions at node n

for $n \in$ nodes **loop**
    $\text{ReachOut}(n) = \{\}$;
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 ;
    $\text{workList} = \text{workList} \setminus \{n\}$;

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

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

All Available Expressions



$A \rightarrow \emptyset$

$B \rightarrow \emptyset$

$C \rightarrow \emptyset$

$D \rightarrow \emptyset$

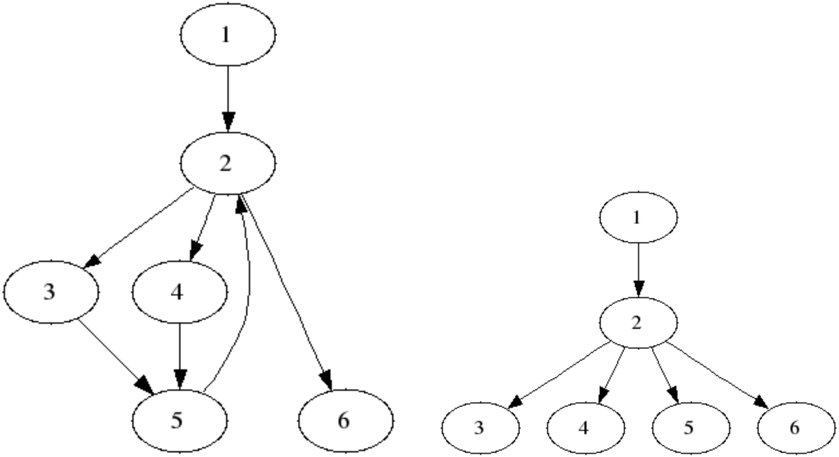
$E \rightarrow \emptyset$

$F \rightarrow \emptyset$

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

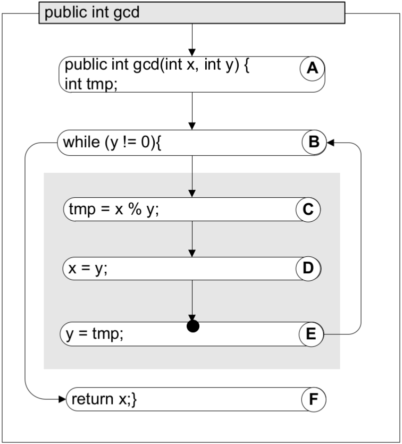


Control Dependence

- 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

Control Dependence

Proof that B is control dependent on E

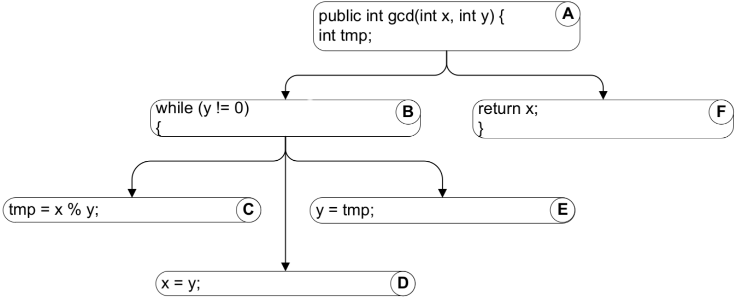


Gray region: nodes post-dominated by E

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

Control Dependence

Full control dependence graph



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 assigment
  - e.g., `b = a; a[2] = 42; i = b[2];` is a definition-use pair (or triple?) in Java

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();
```