Introduction to Models and Abstractions

Outline for Lecture

- Models for Software
  - Overview
  - Finite State Models
  - Kripke Structures
  - Graph Models
    - CFG
    - DFG
- Abstractions
  - Overview
  - Structural
  - Data Value
  - Property Specific

Overview of Models

- A model is some abstraction of a system
  - Operational Semantics
- Verification (formal or informal) is formulated in terms of some model of the system
- A model may be a “complete” or partial representation of a system in a “specification” language
- Ideally models are automatically generated from the system representation or the system representation can be generated from the model.

Finite State Machines

A finite state machine is *model of computation* consisting of a set of *states*, a *start state*, an input *alphabet*, and a *transition function* that maps input symbols and current states to a *next state*. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (*Mealy machine*) or states (*Moore machine*), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (*nondeterministic finite state machine*), one or more states designated as *accepting states* (*recognizer*), etc.

http://www.nist.gov/dads/HTML/finiteStateMachine.html

Mealy Machines

A Mealy machine is a finite state machine that generates an output based on its current state *and* an input. This means that the state diagram will include both an input and output signal for each transition edge. In contrast, the output of a Moore finite state machine depends only on the machine's current state; transitions have no input attached. However, for each Mealy machine there is an equivalent Moore machine whose states are the union of the Mealy machine's states and the Cartesian product of the Mealy machine's states and the input alphabet.

Moore Machines

A **Moore machine** is a finite state automaton where the outputs are determined by the current state alone (and do not depend directly on the input). The state diagram for a Moore machine will include an output signal for each state. Compare with a Mealy machine, which maps *transitions* in the machine to outputs

Simple Microwave Oven Controller

The oven has a momentary-action push button Run to start (apply the power) and a Timer that determines the cooking length. Cooking can be interrupted at any time by opening the oven door. After closing the door the cooking is continued. Cooking is terminated when the Timer elapses. When the door is open a lamp inside the oven is switched on, when the door is closed the lamp is off. During cooking the lamp is also switched on.. The solution should also take into account the possibility that the push button Run could get blocked continuously in the active position. In such a case cooking must not start again until it is deactivated when the cooking is terminated. In other words, each cooking cycle requires intentional activation of the Run button.

Simple Microwave Oven Controller

The control system has the following inputs:

- **Run** momentary-action push button - when activated starts cooking,
- **Timer -** while this runs keep on cooking,
- **Door** sensor - can be true (door closed) or false (door open). And the following outputs:
- **Power -** can be true (power on) or false (power off),
- **Lamp -** can be true (lamp on) or false (lamp off).

Simple Microwave Oven Controller

Moore model

While specifying Moore state machines the states dominate. We think in the following manner: if the input condition changes the state machine changes its state (if a specific transition condition is valid). Entering the new state, the state machine does some actions and waits for the reaction of the controlled system. In a Moore model the entry actions define effectively the state. For instance, we would think about the state Cooking: it is a state where the Timer runs and the state machines waits for the Timer OVER signal.

Moore Machine Model

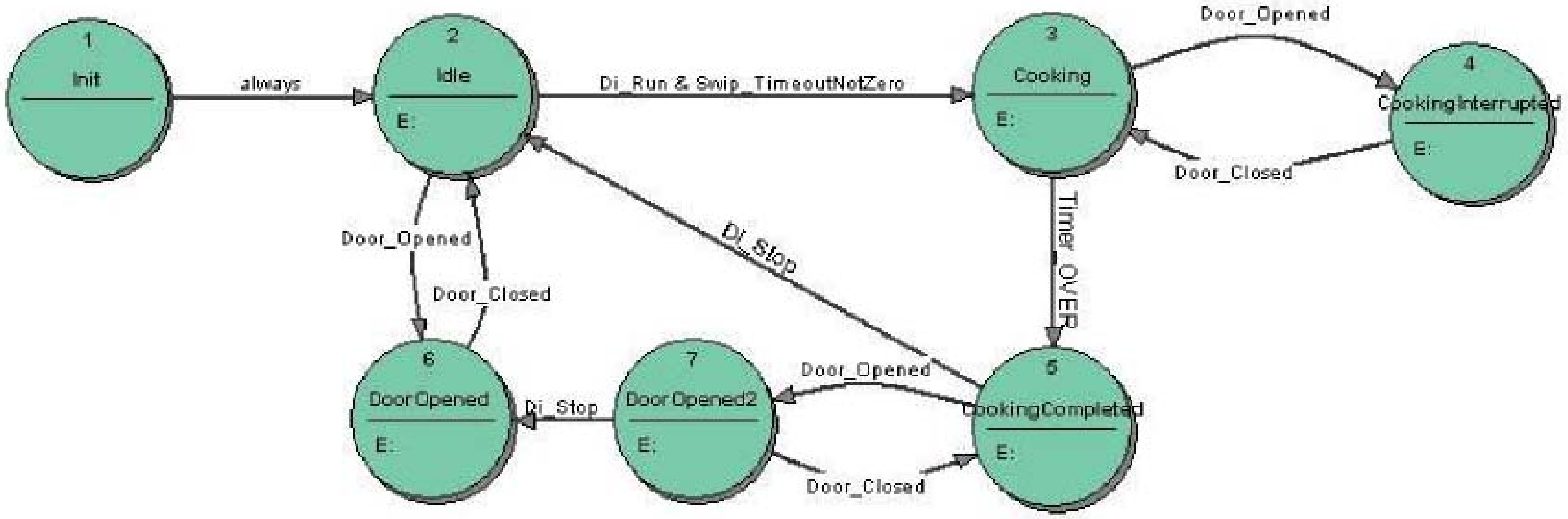
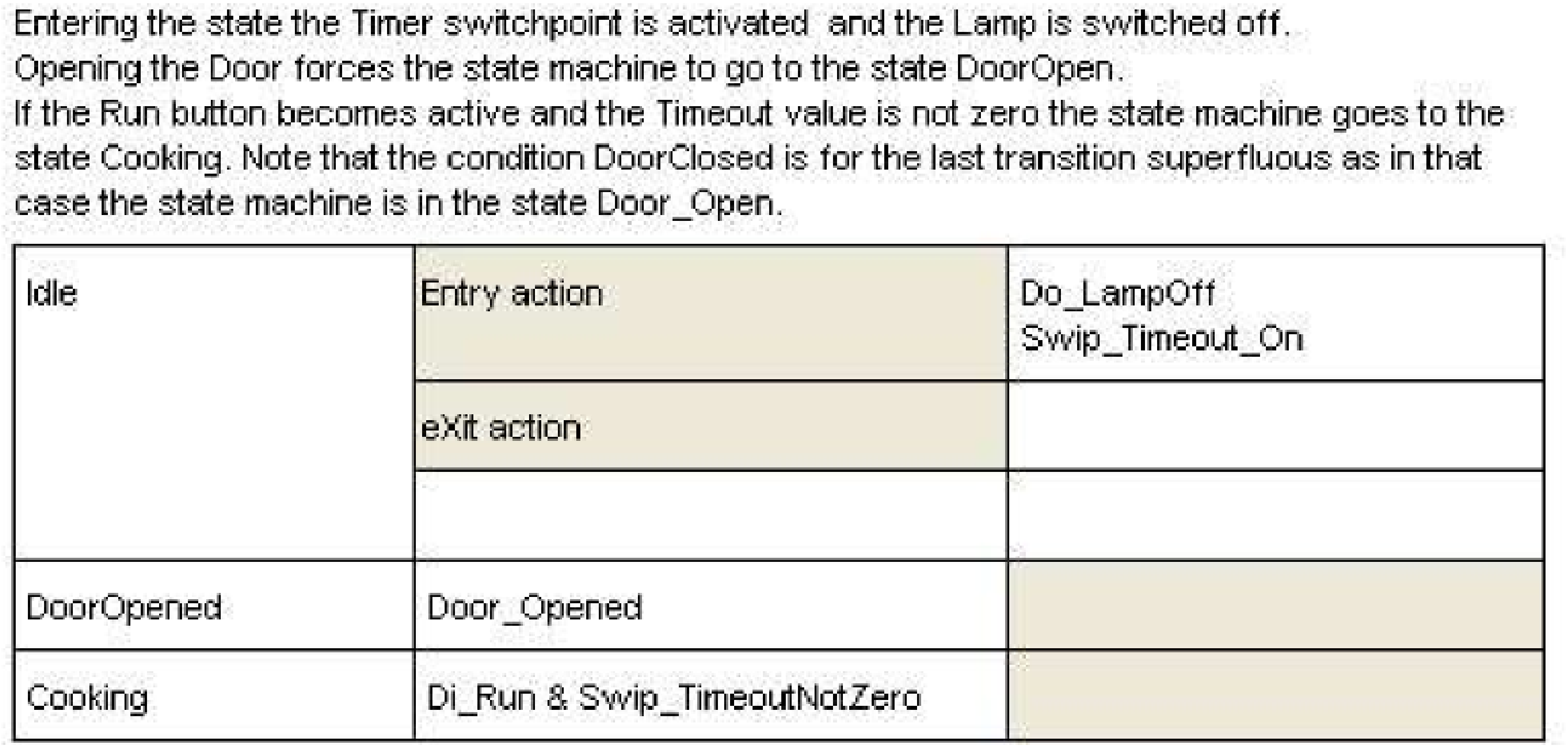


Figure 1 Moore model

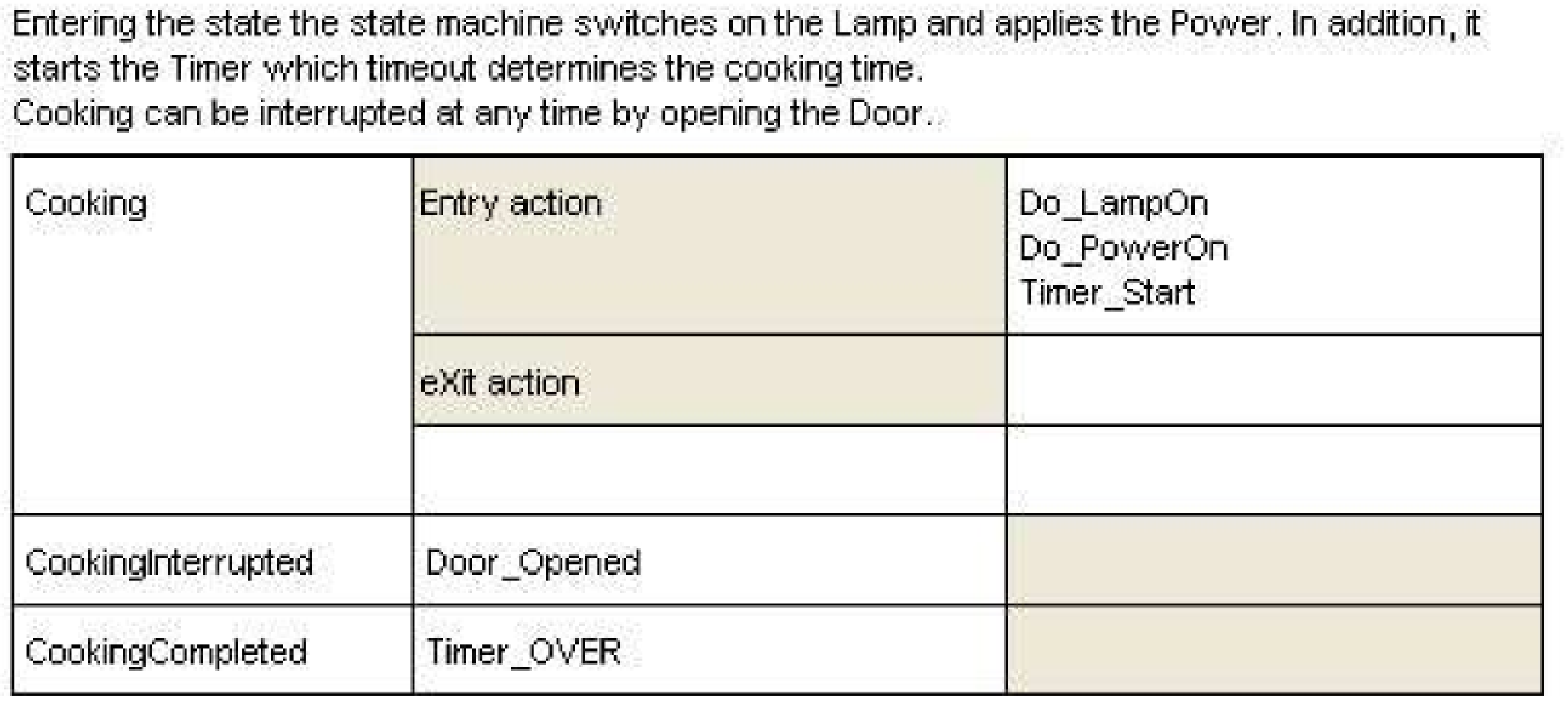
Moore Machine Model



Entering the state the Timer switchpoint is activated and the Lamp is switched off.
Opening the Door forces the state machine to go to the state DoorOpen.
If the Run button becomes active and the Timeout value is not zero the state machine goes to the state Cooking. Note that the condition DoorClosed is for the last transition superfluous as in that case the state machine is in the state Door_Open.

Idle	Entry action	Do_LampOff Swip_Timeout_On
	eXit action	
DoorOpened	Door_Opened	
Cooking	Di_Run & Swip_TimeoutNotZero	

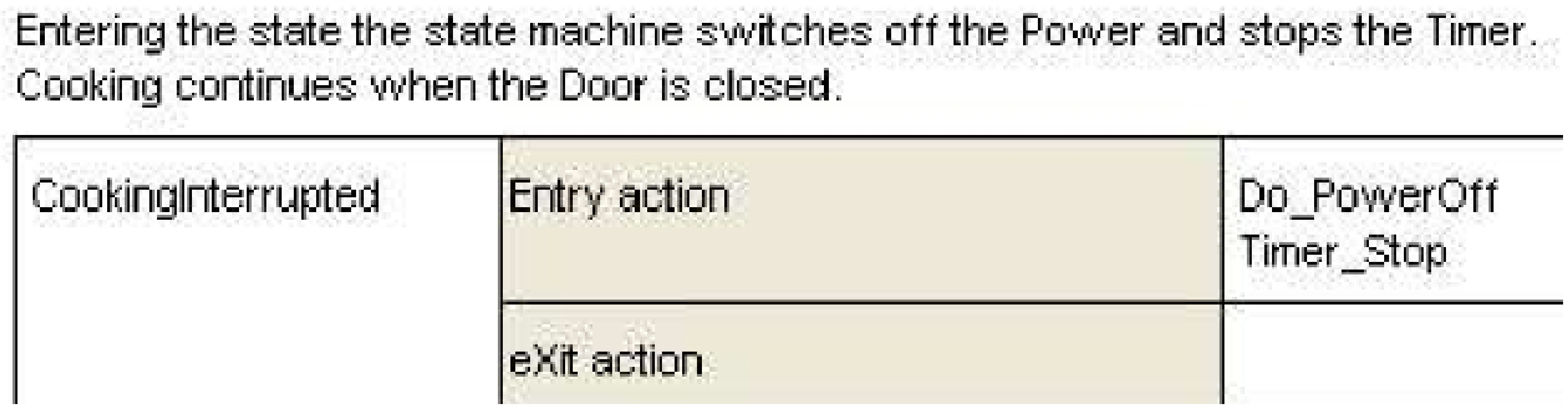
Moore Machine Model



Entering the state the state machine switches on the Lamp and applies the Power. In addition, it starts the Timer which timeout determines the cooking time.
Cooking can be interrupted at any time by opening the Door.

Cooking	Entry action	Do_LampOn Do_PowerOn Timer_Start
	eXit action	
CookingInterrupted	Door_Opened	
CookingCompleted	Timer_OVER	

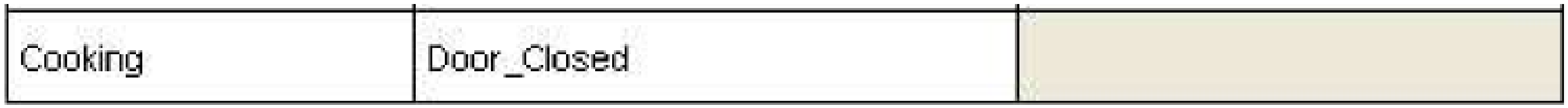
Moore Machine Model



Entering the state the state machine switches off the Power and stops the Timer. Cooking continues when the Door is closed.

CookingInterrupted	Entry action	Do_PowerOff Timer_Stop
	eXit action	

Moore Machine Model



Cooking	Door_Closed	

Note the necessity of having two states DoorOpened. The DoorOpened2 is required to keep the information about the Run button: until it stays active in the state CookingCompleted the state machine must not return to the state Idle.

Mealy Machine Model

The states: Idle, Cooking and CookingInterrupted illustrate its features. All activities are done as Input actions, which means that actions essential for a state must be performed in all states which have a transition to that state. The Timer must be now started in both states: Idle and CookingInterrupted.

Mealy Machine Model

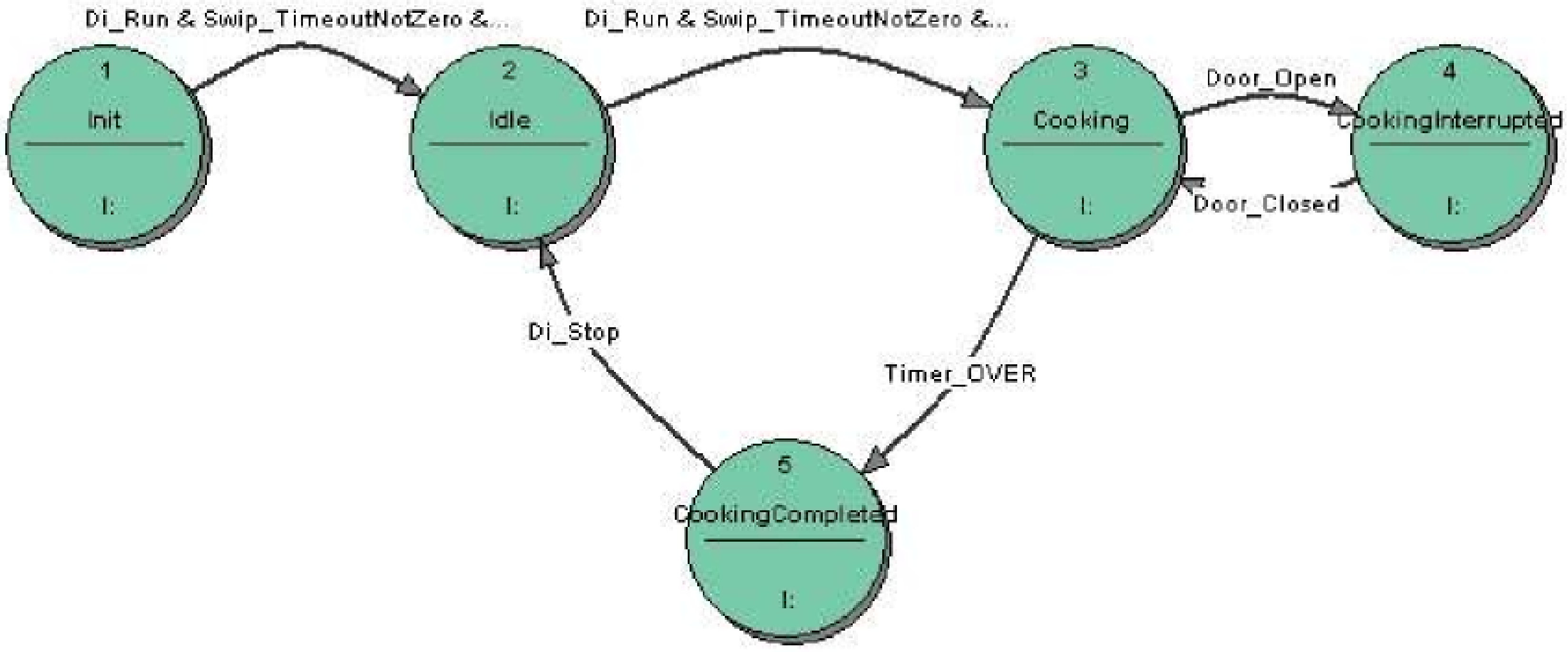
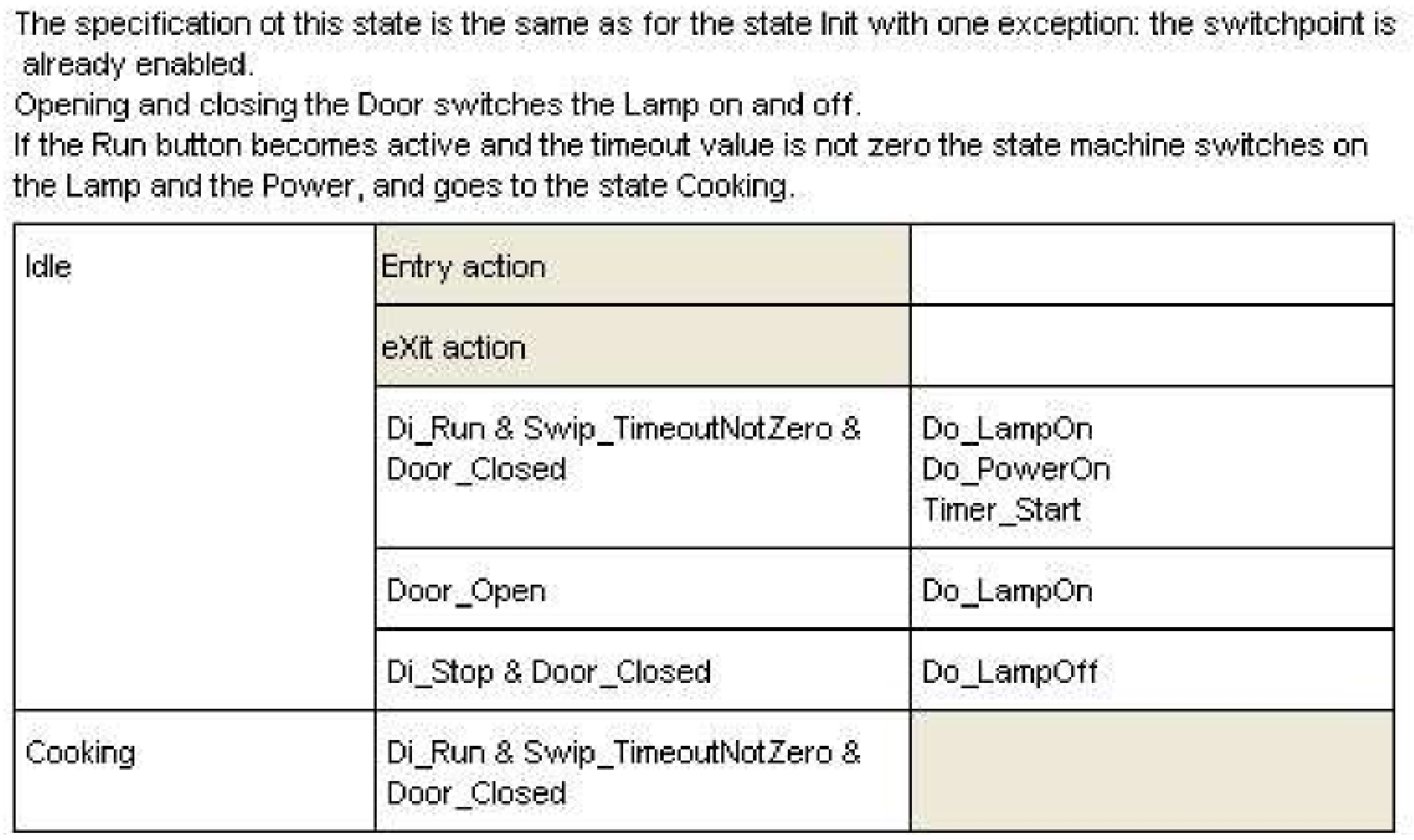


Figure 5 Mealy model

Mealy Machine Model

In a Mealy model the inputs dominate thinking during specification. If an input condition changes the state machine reacts to that change performing some action(s); it may also change the state. In a Mealy model the inputs in a given state determine less the present state but rather the next states. For instance, the Timer which decides about the state Cooking must be started in both neighboring states: Idle and CookingInterrupted. Therefore the meaning of states is more hazy: the state Cooking is determined by the Timer but the Timer control is spread over several states.

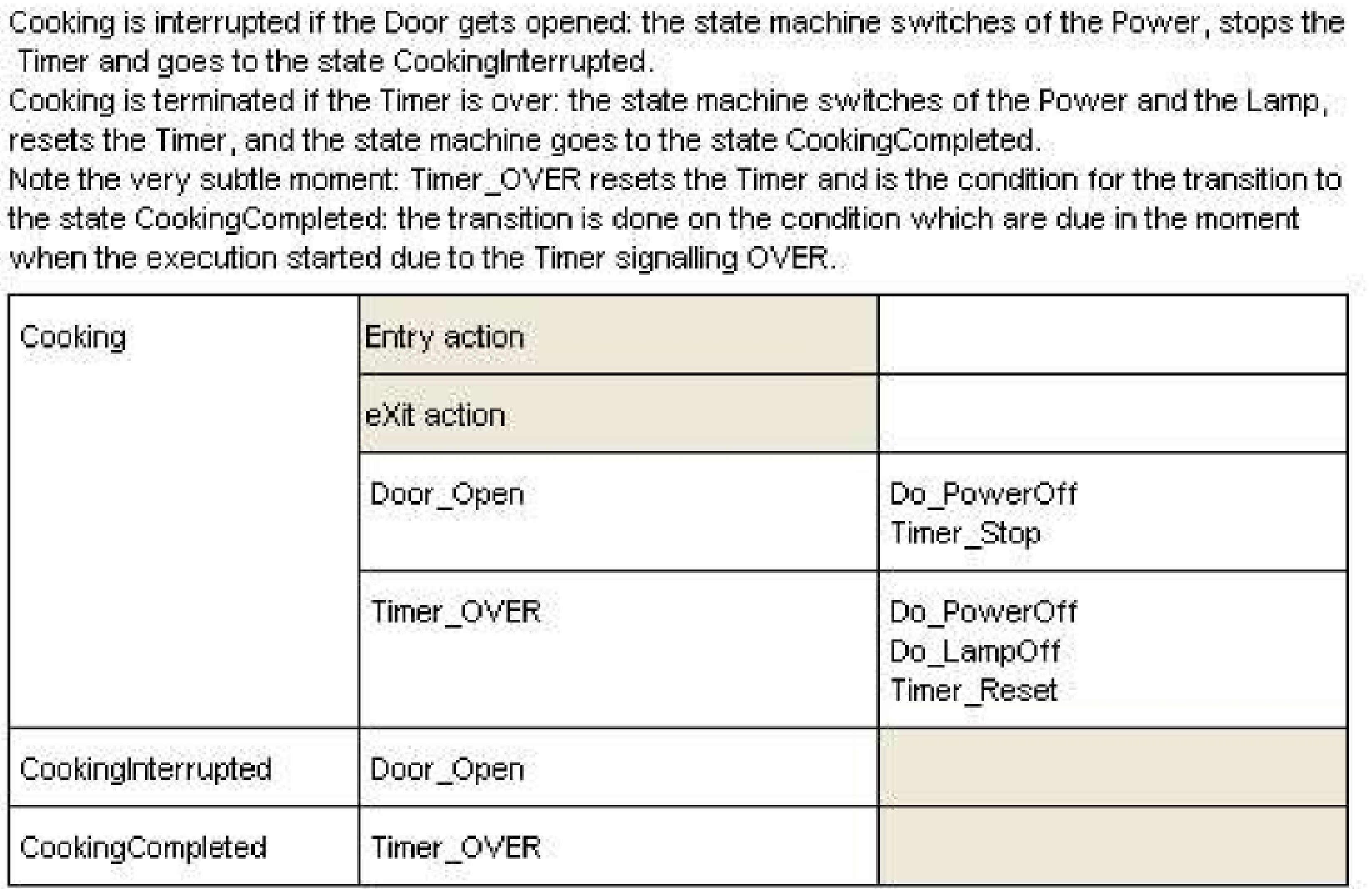
Mealy Machine Model



The specification of this state is the same as for the state Init with one exception: the switchpoint is already enabled.
Opening and closing the Door switches the Lamp on and off.
If the Run button becomes active and the timeout value is not zero the state machine switches on the Lamp and the Power, and goes to the state Cooking.

Idle	Entry action	
	eXit action	
	Di_Run & Swip_TimeoutNotZero & Door_Closed	Do_LampOn Do_PowerOn Timer_Start
	Door_Open	Do_LampOn
	Di_Stop & Door_Closed	Do_LampOff
Cooking	Di_Run & Swip_TimeoutNotZero & Door_Closed	

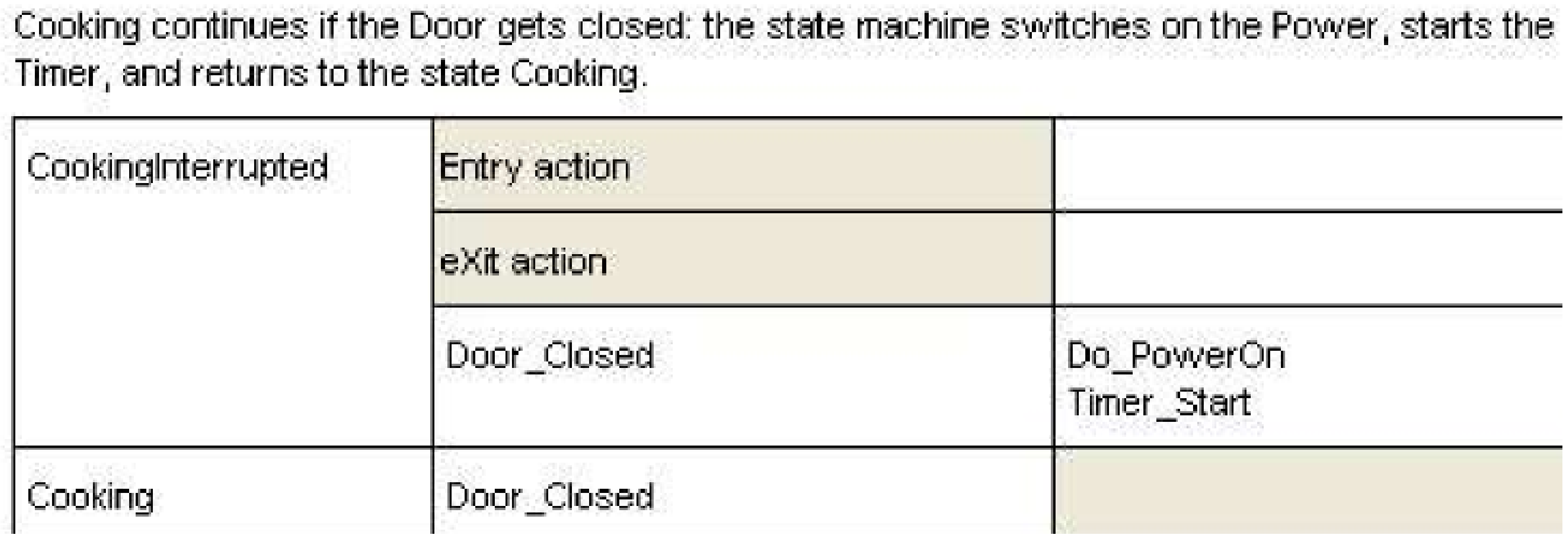
Mealy Machine Model



Cooking is interrupted if the Door gets opened: the state machine switches of the Power, stops the Timer and goes to the state CookingInterrupted.
Cooking is terminated if the Timer is over: the state machine switches of the Power and the Lamp, resets the Timer, and the state machine goes to the state CookingCompleted.
Note the very subtle moment: Timer_OVER resets the Timer and is the condition for the transition to the state CookingCompleted: the transition is done on the condition which are due in the moment when the execution started due to the Timer signalling OVER.

Cooking	Entry action	
	eXit action	
	Door_Open	Do_PowerOff Timer_Stop
	Timer_OVER	Do_PowerOff Do_LampOff Timer_Reset
CookingInterrupted	Door_Open	
CookingCompleted	Timer_OVER	

Mealy Machine Model



Cooking continues if the Door gets closed: the state machine switches on the Power, starts the Timer, and returns to the state Cooking.

CookingInterrupted	Entry action	
	eXit action	
	Door_Closed	Do_PowerOn Timer_Start
Cooking	Door_Closed	

Mixed Model

The states Cooking and CookingInterrupted correspond to the Moore model. The states Init and CookingCompleted is a mixture of both models where we use both: Entry and Input actions.

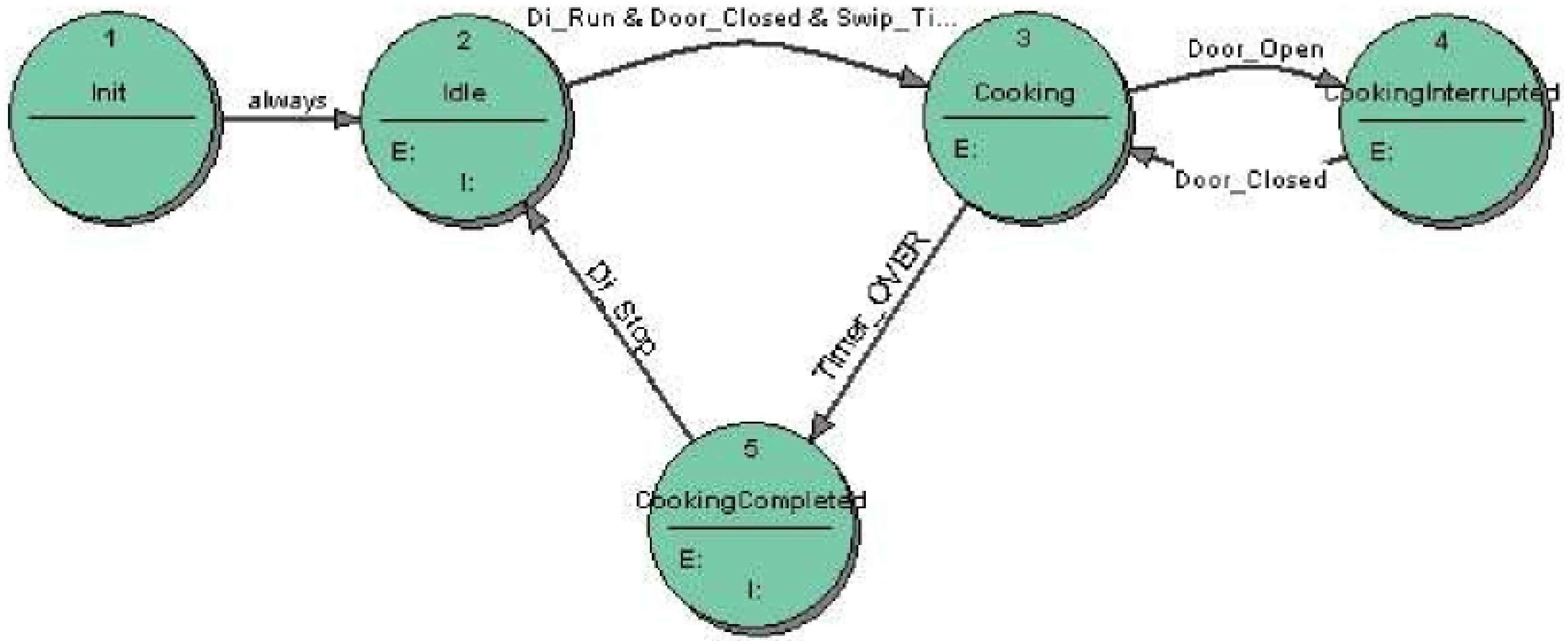


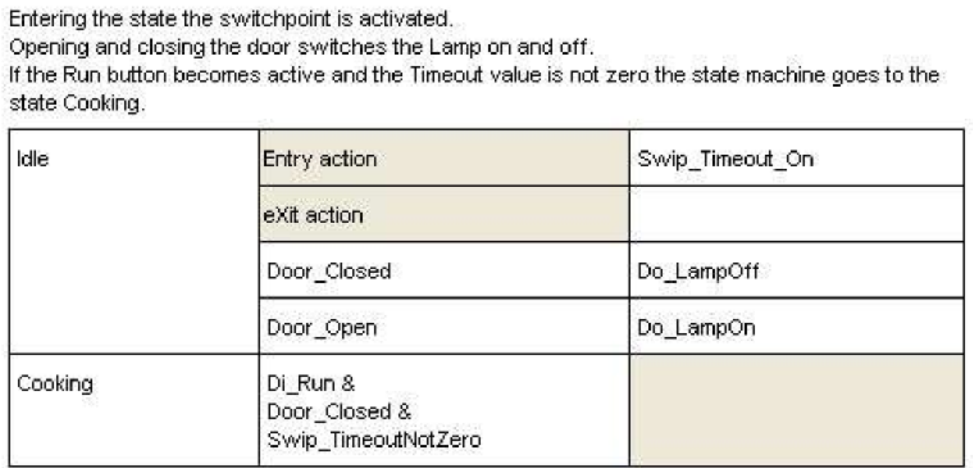
Figure 9 Mixed model

Mixed Model

In the example, we use essentially a Moore model completed by Input Actions in the states: Idle and CookingCompleted. That small change gives a reduction from 7 to 5 states. Considering the complexity both state machines: Moore and mixed model are similar, the mixed model having the advantage of fewer states.

On the other hand, though the Mealy model and the mixed model have the same number of states, they are very different. Comparing the Mealy model with the mixed model we notice an essential simplification in favour of the mixed model.

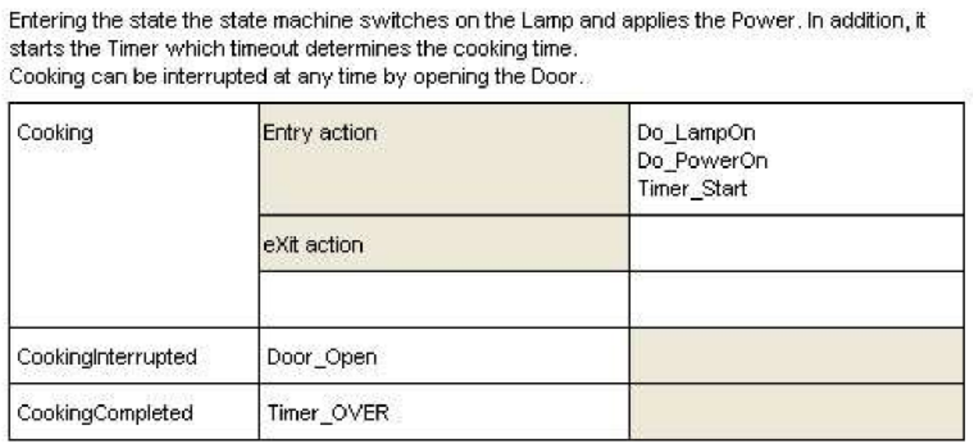
Mixed Model



Entering the state the switchpoint is activated.
Opening and closing the door switches the Lamp on and off.
If the Run button becomes active and the Timeout value is not zero the state machine goes to the state Cooking.

Idle	Entry action	Swip_Timeout_On
	eXit action	
	Door_Closed	Do_LampOff
	Door_Open	Do_LampOn
Cooking	Di_Run & Door_Closed & Swip_TimeoutNotZero	

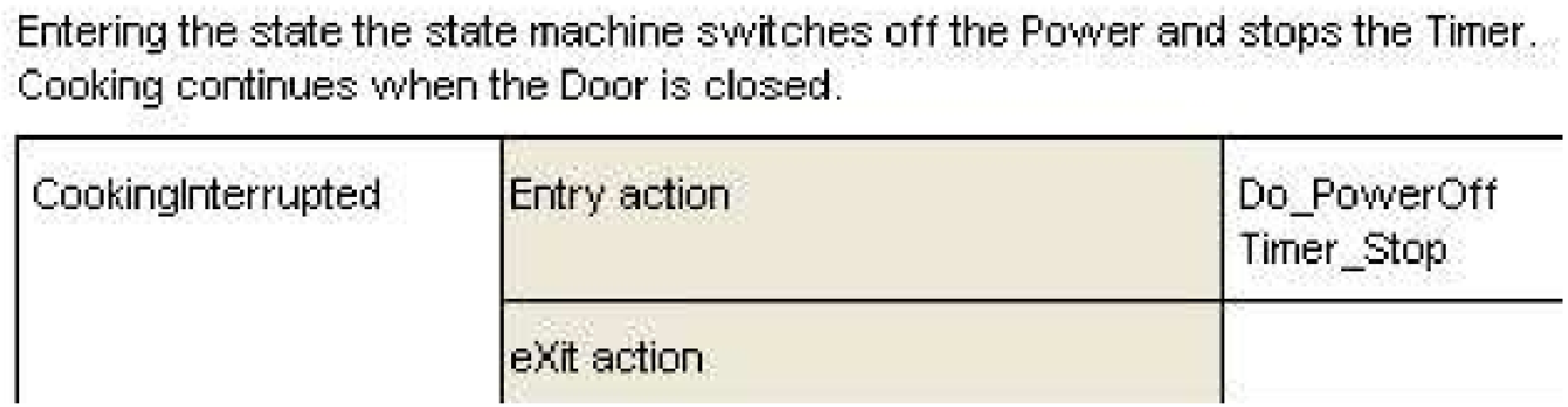
Mixed Model



Entering the state the state machine switches on the Lamp and applies the Power. In addition, it starts the Timer which timeout determines the cooking time.
Cooking can be interrupted at any time by opening the Door.

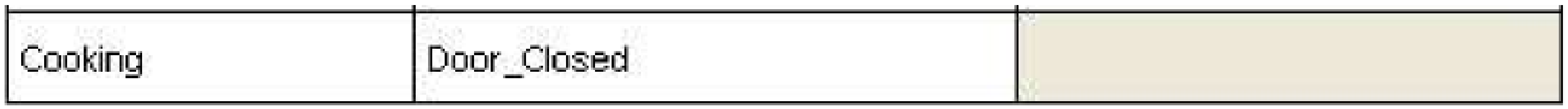
Cooking	Entry action	Do_LampOn Do_PowerOn Timer_Start
	eXit action	
CookingInterrupted	Door_Open	
CookingCompleted	Timer_OVER	

Mixed Model



Entering the state the state machine switches off the Power and stops the Timer.
Cooking continues when the Door is closed.

CookingInterrupted	Entry action	Do_PowerOff Timer_Stop
	eXit action	



Cooking	Door_Closed	

Mixed Model

Mixed Model

- Exit Actions expand the possible specification modes but are not supported by many FSM systems.
- A Moore model is very easy to code, the transition may be often implemented just by constants as initialized tables. The Mealy model opens the Pandora box: the program becomes so complex that we lose the state machine in the confusing code.

Example Problem from PY

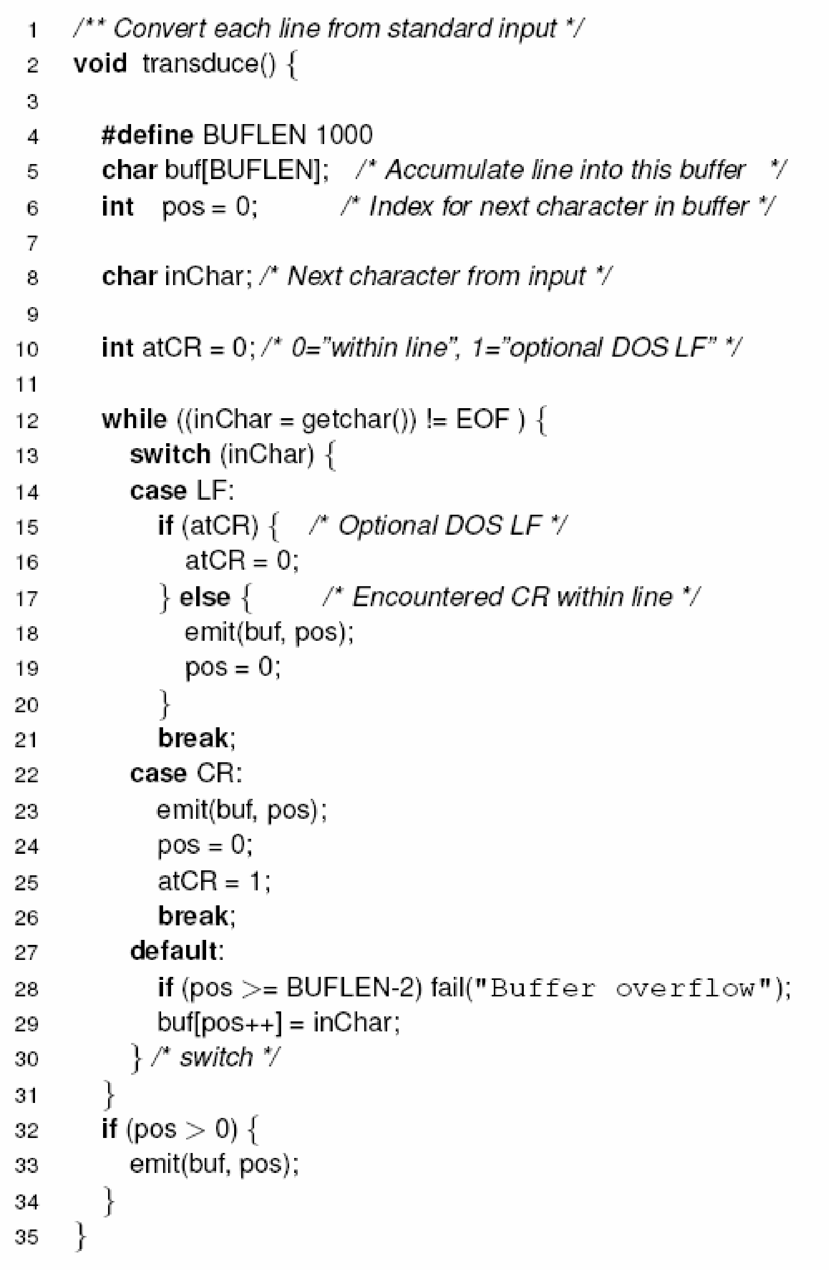
Consider the problem of converting among DOS, Unix and Mac line end conventions in the form of a Mealy machine. An event is reading a character or encountering an "end of file marker." The possible input characters are divided into four catagories: carriage return, line feed, end of file and everything else. The states are defined by program control points with some associated variables. The specification is to read in lines in one format and emit lines in another format.

The actions are:

1. A character is input
2. If the character is an EOF then the program empties its buffer and terminates.
3. If the character is a CR then the program emits the buffer and sets the number of characters in the buffer to zero.
4. For any other character, the character is appended to the current position in the buffer and the current position is incremented.

Example Problem from PY

The obvious states of the program are: the buffer is empty, the machine is processing a buffer, the machine is removing a LF and the program is terminating.



```
/** 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);
   }
}
```

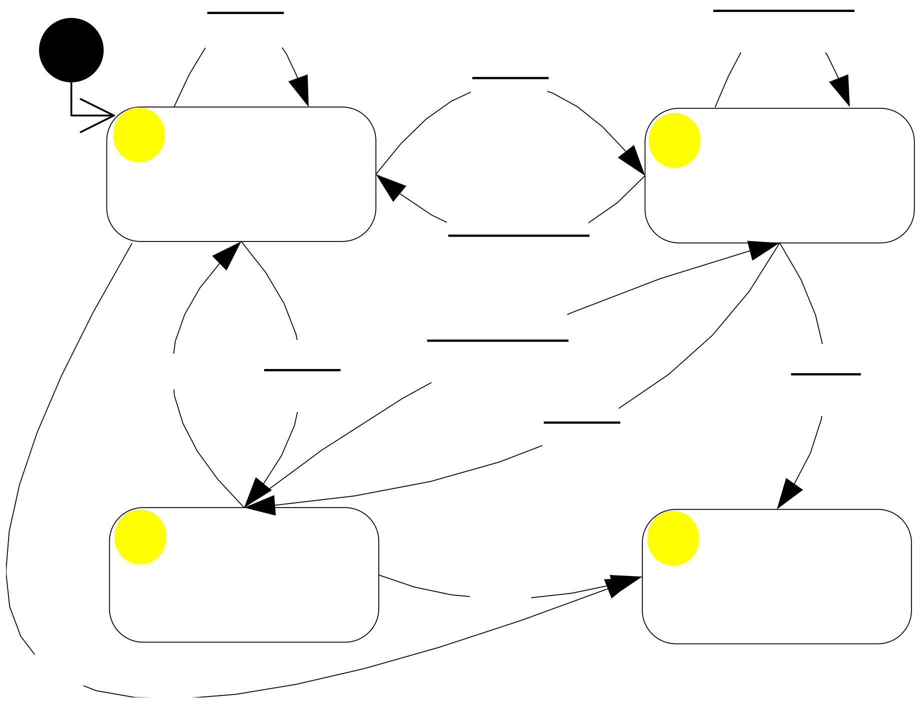
Program (more or less) corresponding to the specification.

Environment for program is a procedure which supplies characters and a procedure which absorbs characters.

Finite State Machine Representations

- finite set of states (nodes)
- set of transitions among states (edges)

Graph representation (Mealy machine)



Tabular representation

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

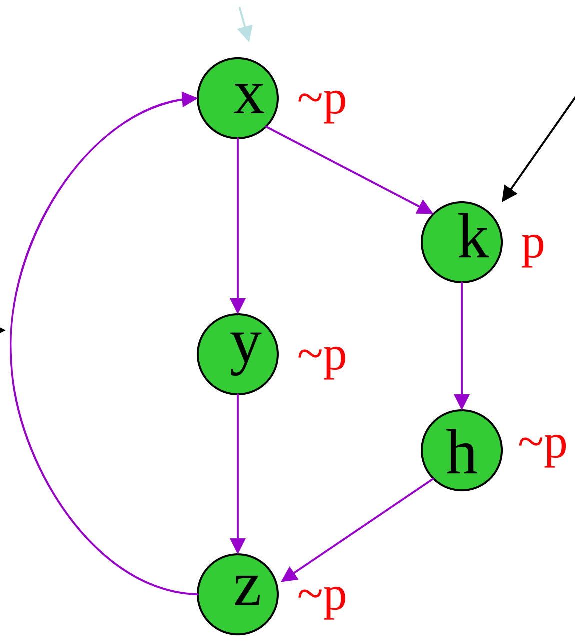
Abstraction Function

```
 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  }
```

Abstract state	*Concrete state*		
	Lines	atCR	pos
e (Empty buffer)	3 – 13	0	0
w (Within line)	13	0	> 0
l (Looking for LF)	13	1	0
d (Done)	36	–	–

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

Labeled State Graph
Kripke Structure



Each state represents all variable values and location counters

Each transition represents an execution step in the system

The labels represent predicates in each state e.g. (x = 5)

$K = (\{p, \sim p\}, \{x, y, z, k, h\}, R, \{x\}, L)$

Kripke Structures

- Kripke structures can be generated by the execution of a finite state machine (or set of interacting state machines.)
- Kripke structure is a finite state machine
- Extension to infinite state systems as (Buchi automata) will be taken up later under model checking
- A Kripke structure which captures the complete set of states of a program represents all possible executions of the program from a given initial state (or environment).

Kripke Structures

- Kripke structures can be represented as guarded equational systems (model of execution must be specified) and visa versa.

Microwave Oven Specification

This simple oven has a single control button. When the oven door is closed and the user presses the button, the oven will cook (that is, energize the power tube) for 1 minute.

If the control button is pressed and the door is open the power tube is not energized.

When the oven times out (cooks until the timer expires), it turns off both the power tube.

The user can stop the cooking by opening the door. Once the door is opened, the timer resets to zero.

Simple Microwave Oven

- K (or M) = (S, S_0, R, L)
- S = (S1, S2, S3, S4)
- S_0 = S1 is the initial state
- R = ({S1, S2} {S2, S1}, {S1, S4}, {S4, S2}, {S2,
- S3}, {S3, S2}, {S3, S3}
- L (S1) = {¬close, ¬ start, ¬ cooking} L (S2) =
- {close, ¬ start, ¬ cooking} L (S3) = {close, start,
- cooking} L (S4) = {¬close, start, ¬ cooking}

Kripke Structure (State Model) for Microwave Oven

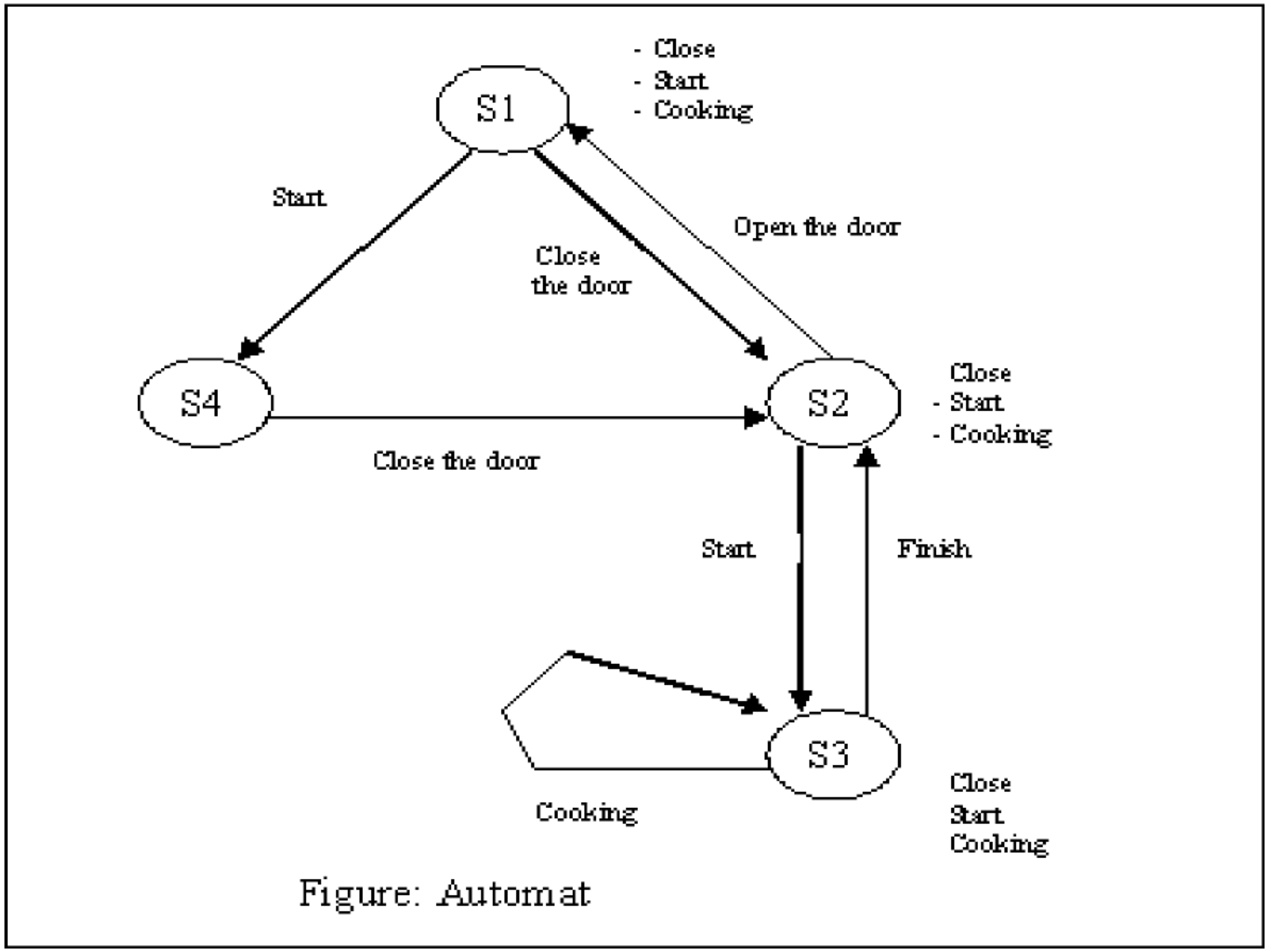


Figure: Automat

Environment for Microwave Oven

- Entity to open and close door (send open and close events)
- Timer which receives start events and sends finish events.

Mutual Exclusion Example

- Two process mutual exclusion with shared semaphore
- Each process has three states
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- Semaphore can be available (S_0) or taken (S_1)
- Initially both processes are in the Non-critical state and the semaphore is available --- $N_1\ N_2\ S_0$

$$\begin{array}{lcl} N_1 & \rightarrow & T_1 \\ T_1 \wedge S_0 & \rightarrow & C_1 \wedge S_1 \\ C_1 & \rightarrow & N_1 \wedge S_0 \end{array} \quad \Big|\Big| \quad \begin{array}{lcl} N_2 & \rightarrow & T_2 \\ T_2 \wedge S_0 & \rightarrow & C_2 \wedge S_1 \\ C_2 & \rightarrow & N_2 \wedge S_0 \end{array}$$

Mutual Exclusion Example

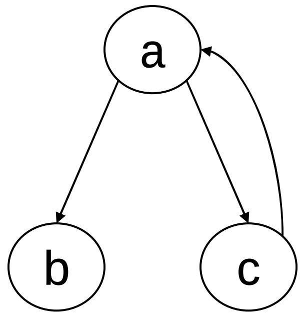
$$\begin{tikzcd}
 & & N_1N_2S_0 \arrow[dl] \arrow[dr] & & \\
 & T_1N_2S_0 \arrow[dl] \arrow[dr] & & N_1T_2S_0 \arrow[dl] \arrow[dr] & \\
C_1N_2S_1 \arrow[uurr, bend left] \arrow[dr] & & T_1T_2S_0 \arrow[dl] \arrow[dr] & & N_1C_2S_1 \arrow[uull, bend right] \arrow[dl] \\
 & C_1T_2S_1 \arrow[uurr, bend right] & & T_1C_2S_1 \arrow[uull, bend left] &
\end{tikzcd}$$

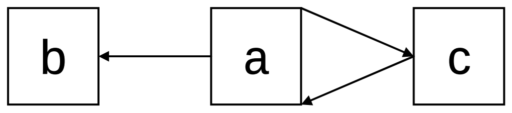
Execution model is asynchronous interleaved with atomic actions.

Graph Representations: directed graphs

- Directed graph:
  - N (set of nodes)
  - E (relation on the set of nodes) edges

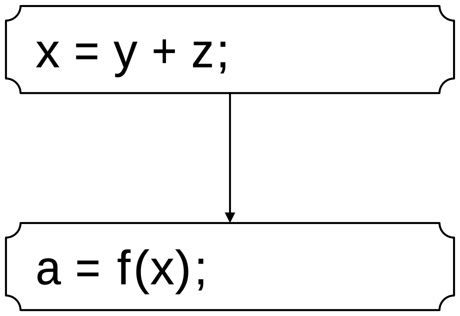
Nodes: {a, b, c}
Edges: {(a,b), (a, c), (c, a)}





Graph Representations: labels and code

- We can label nodes with the names or descriptions of the entities they represent.
  - If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way:



Multidimensional Graph Representations

- Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once
  - class B extends (is a subclass of) class A
  - class B has a field that is an object of type C

extends relation

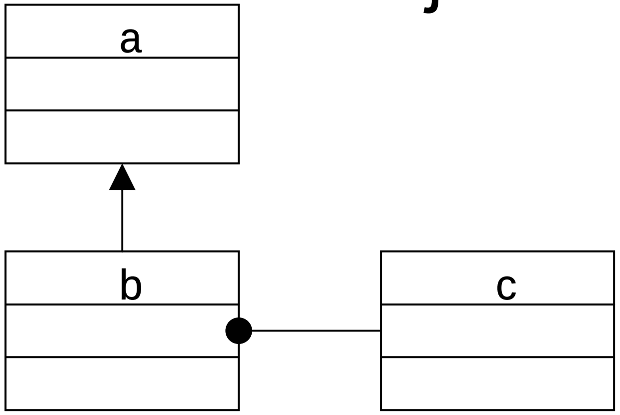
NODES = {A, B, C}

EDGES = {(A,B)}

includes relation

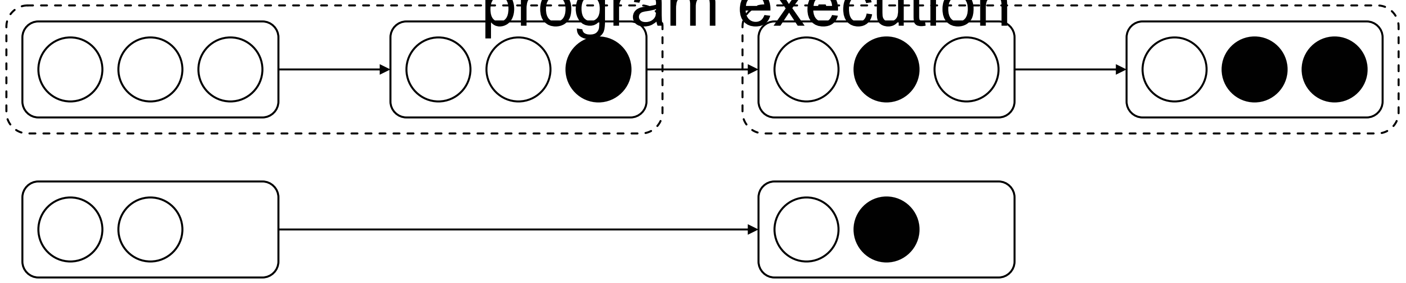
NODES = {A, B, C}

EDGES = {(B,C)}



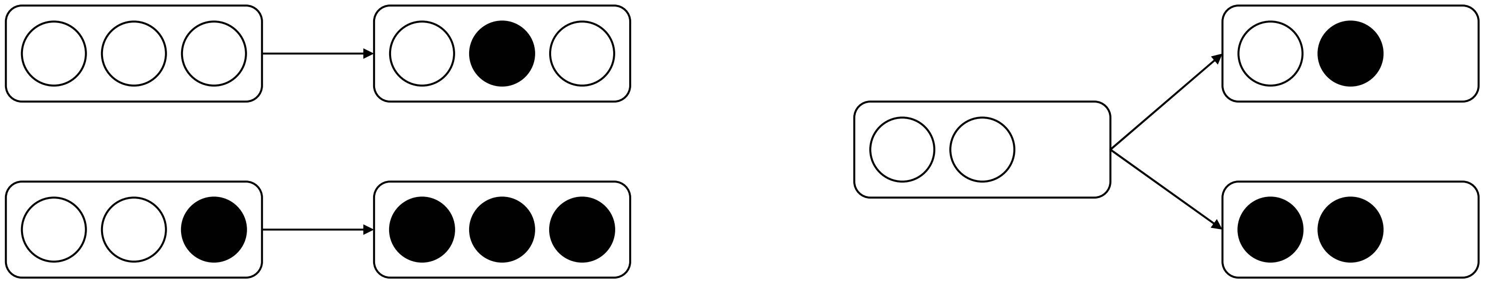
Finite Abstraction of Behavior

an abstraction function suppresses some details of program execution



⇒

it lumps together execution states that differ with respect to the suppressed details but are otherwise identical

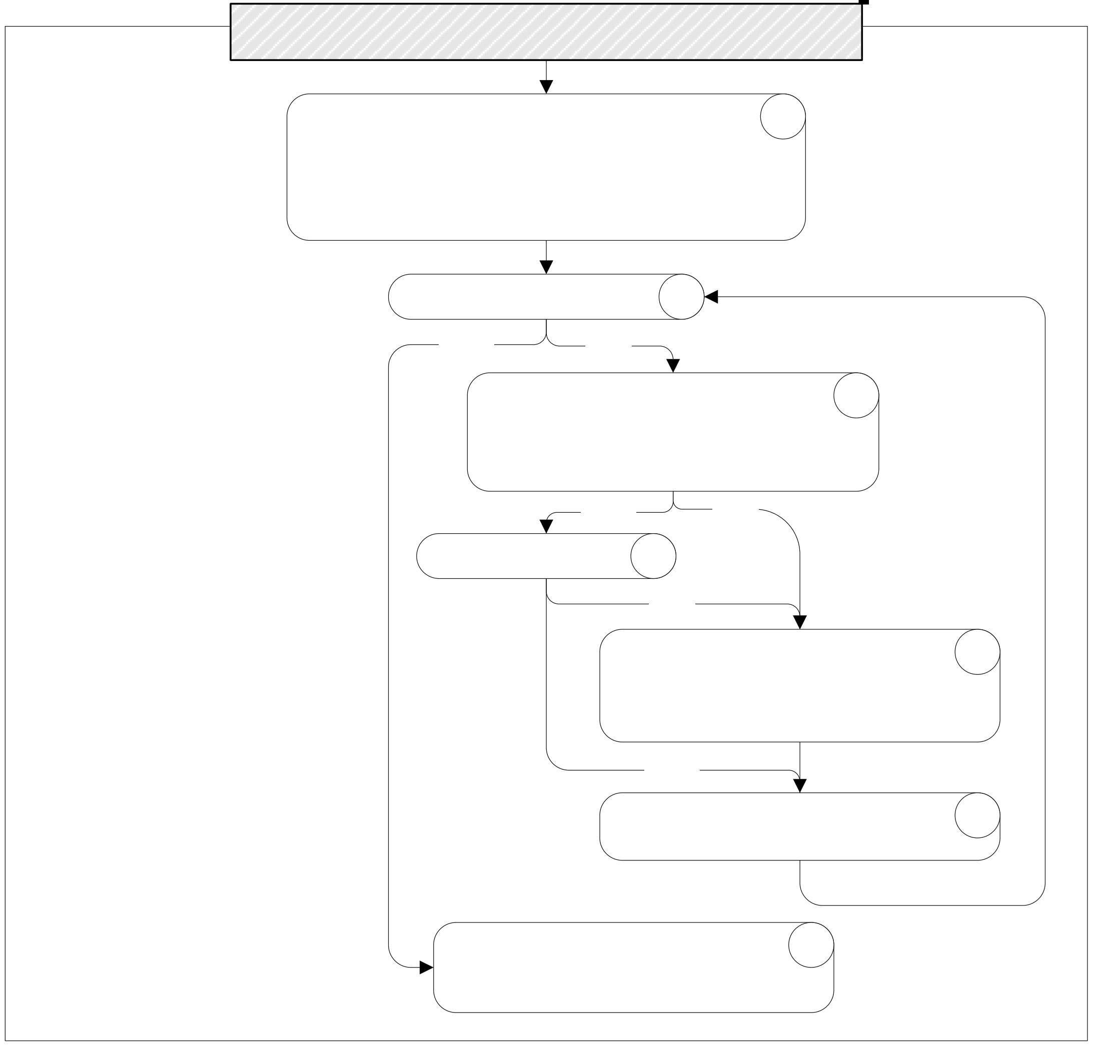


(Intraprocedural) Control Flow Graph

- nodes = regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Often statements are grouped in single regions to get a compact model
  - Sometime single statements are broken into more than one node to model control flow within the statement
- directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another

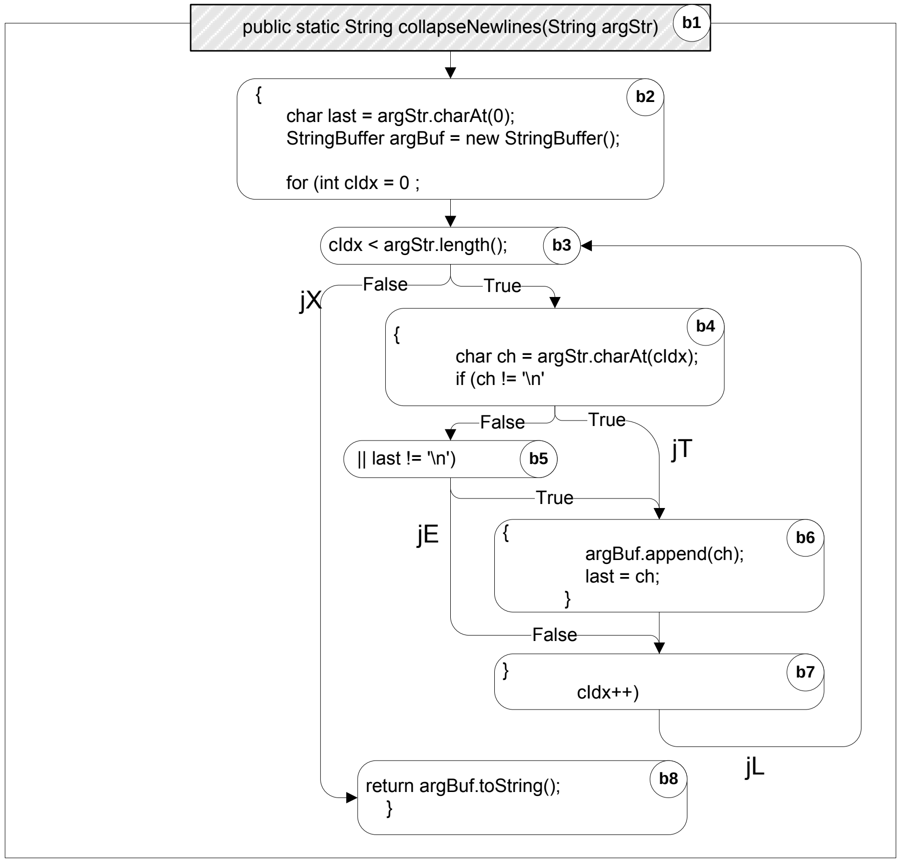
Example of Control Flow Graph

```
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();
  }
```



Linear Code Sequence and Jump (LCSJ)

Essentially subpaths of the control flow graph from one branch to another


public static String collapseNewlines(String argStr)
b1
{
char last = argStr.charAt(0);
StringBuffer argBuf = new StringBuffer();
for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
jX
{
char ch = argStr.charAt(cIdx);
if (ch != '\n'
b4
False
True
jT
\|\| last != '\n')
b5
True
jE
{
argBuf.append(ch);
last = ch;
}
b6
False
}
cIdx++)
b7
jL
return argBuf.toString();
}
b8

From	Sequence of basic blocs	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	ret
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL

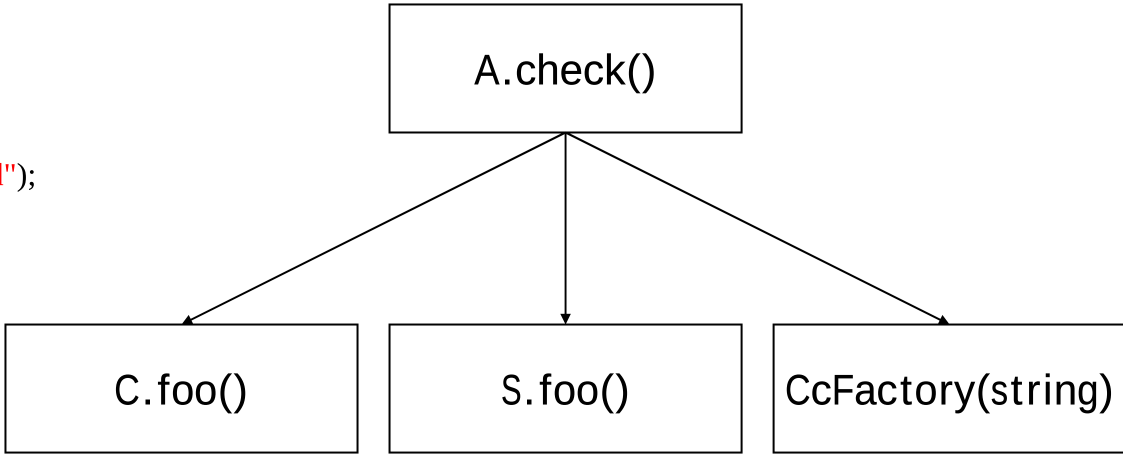
Interprocedural control flow graph

- Call graphs
  - Nodes represent procedures
    - Methods
    - C functions
    - ...
  - Edges represent *calls* relation

Overestimating the *calls* relation

The static call graph includes calls through dynamic bindings that never occur in execution.

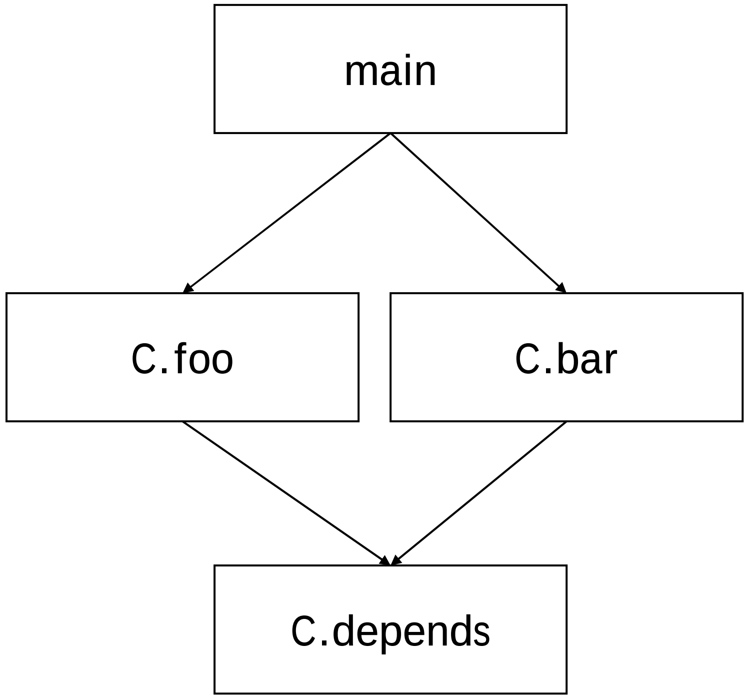
```
public class C {
   public static C cFactory(String kind) {
      if (kind == "C") return new C();
      if (kind == "S") return new S();
      return null;
   }
   void foo() {
      System.out.println("You called the parent's method");
   }
   public static void main(String args[]) {
      (new 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();
   }
}
```



$$\begin{tikzcd}
 & \text{A.check()} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{C.foo()} & \text{S.foo()} & \text{CcFactory(string)}
\end{tikzcd}$$

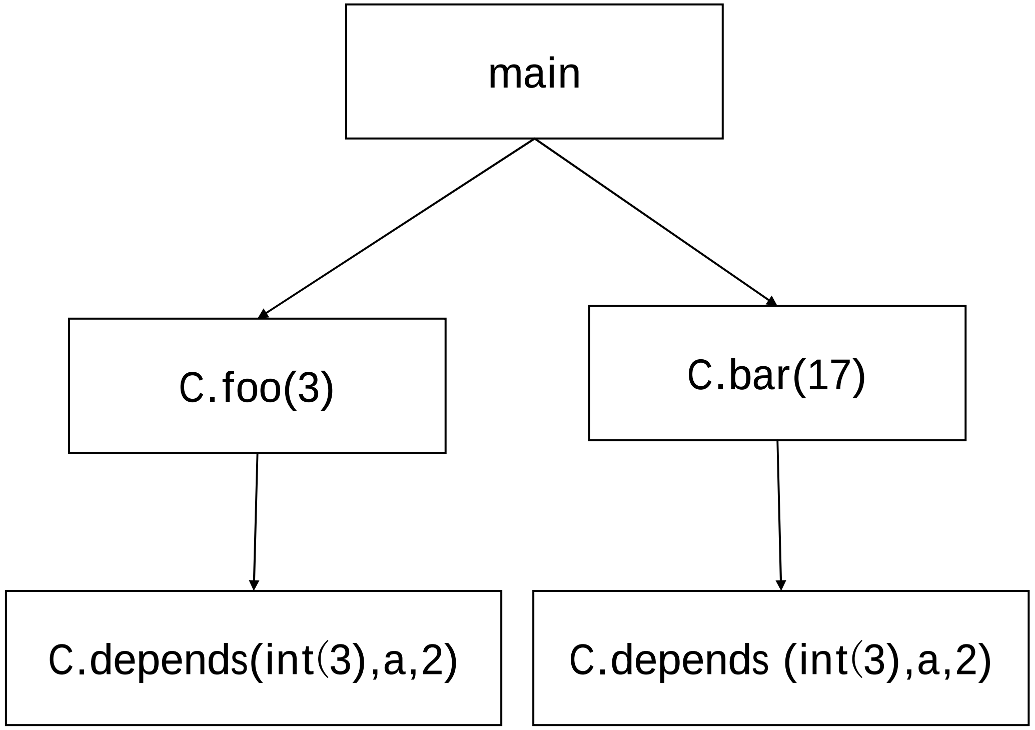
Contex Insensitive Call graphs

```
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;
   }
}
```

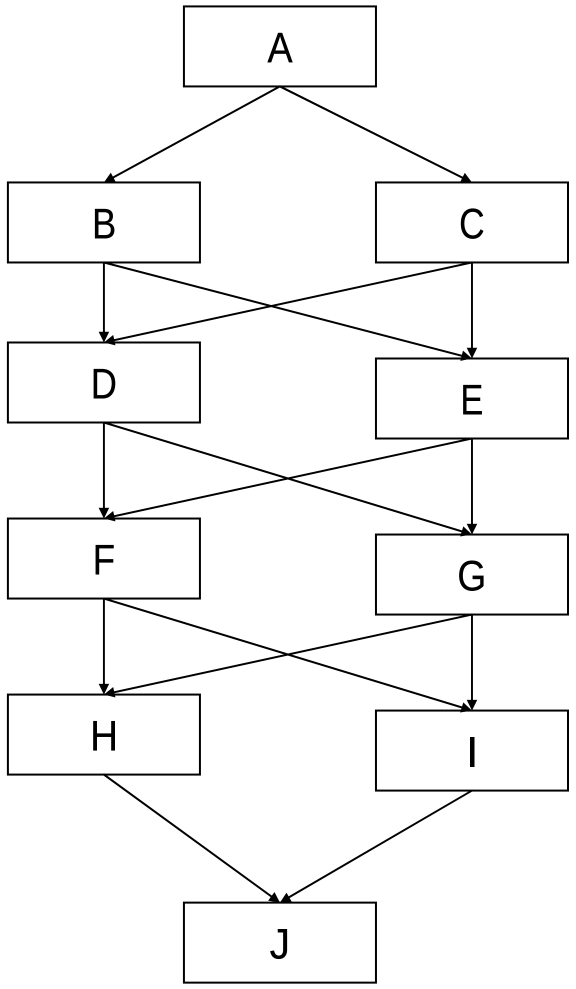


Contex Sensitive Call graphs

```
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;
    }
}
```



Context Sensitive CFG exponential growth



1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

8 contexts …

16 calling contexts …

Dependence and Data Flow Models

Why Data Flow Models?

- Models from Chapter 5 emphasized control
    - Control flow graph, call graph, finite state machines
- We also need to reason about dependence
    - Where does this value of x come from?
    - What would be affected by changing this?
    - ...
- Many program analyses and test design techniques use data flow information
    - Often in combination with control flow
        - Example: "Taint" analysis to prevent SQL injection attacks
        - Example: Dataflow test criteria (Ch.13)

Learning objectives

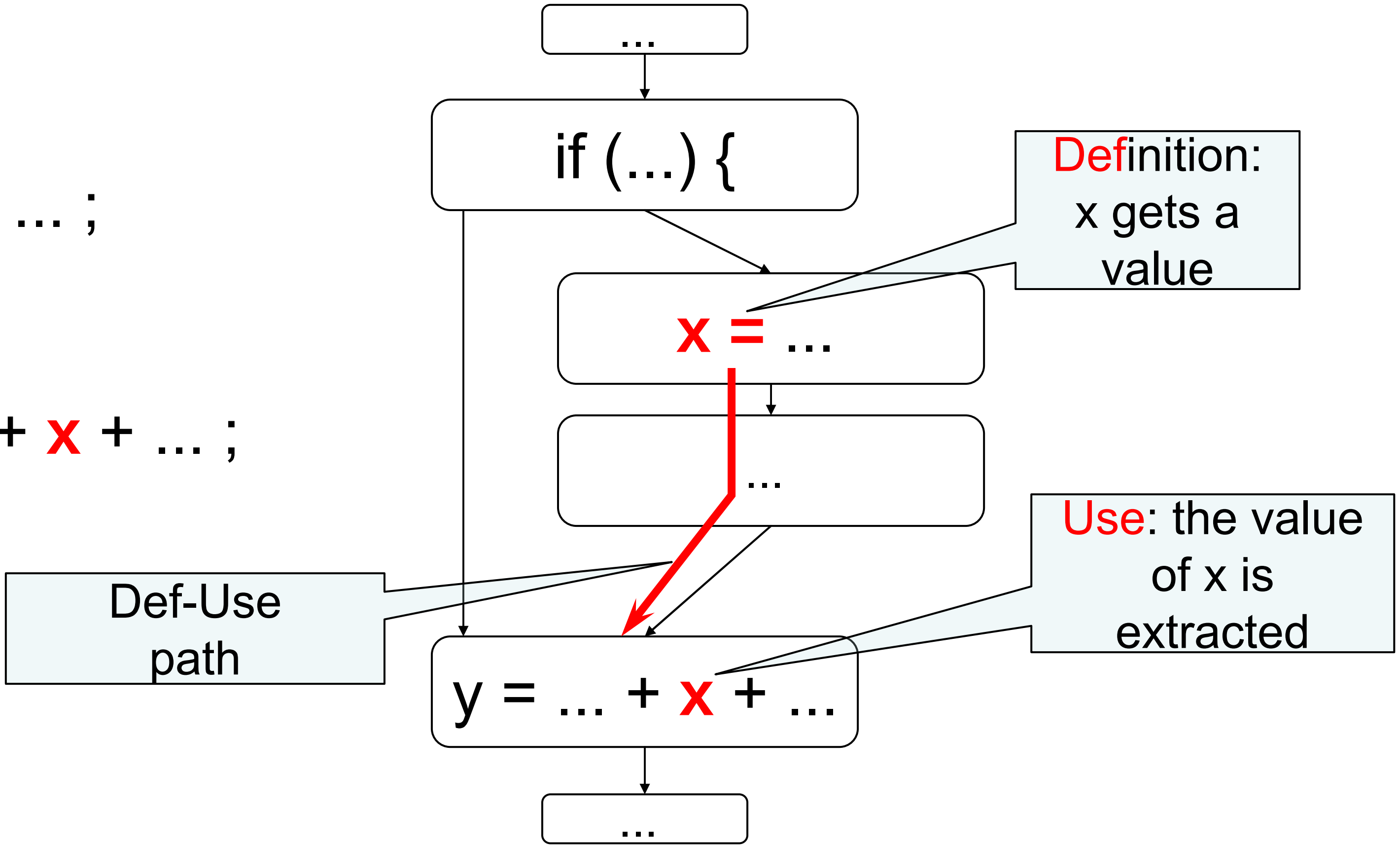
- Understand basics of data-flow models and the related concepts (def-use pairs, dominators…)
- Understand some analyses that can be performed with the data-flow model of a program
  - The data flow analyses to build models
  - Analyses that use the data flow models
- Understand basic trade-offs in modeling data flow
  - variations and limitations of data-flow models and analyses, differing in precision and cost

Def-Use Pairs (1)

- A **def-use (du) pair** associates a point in a program where a value is produced with a point where it is used
- **Definition**: where a variable gets a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter
- **Use**: extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

Def-Use Pairs

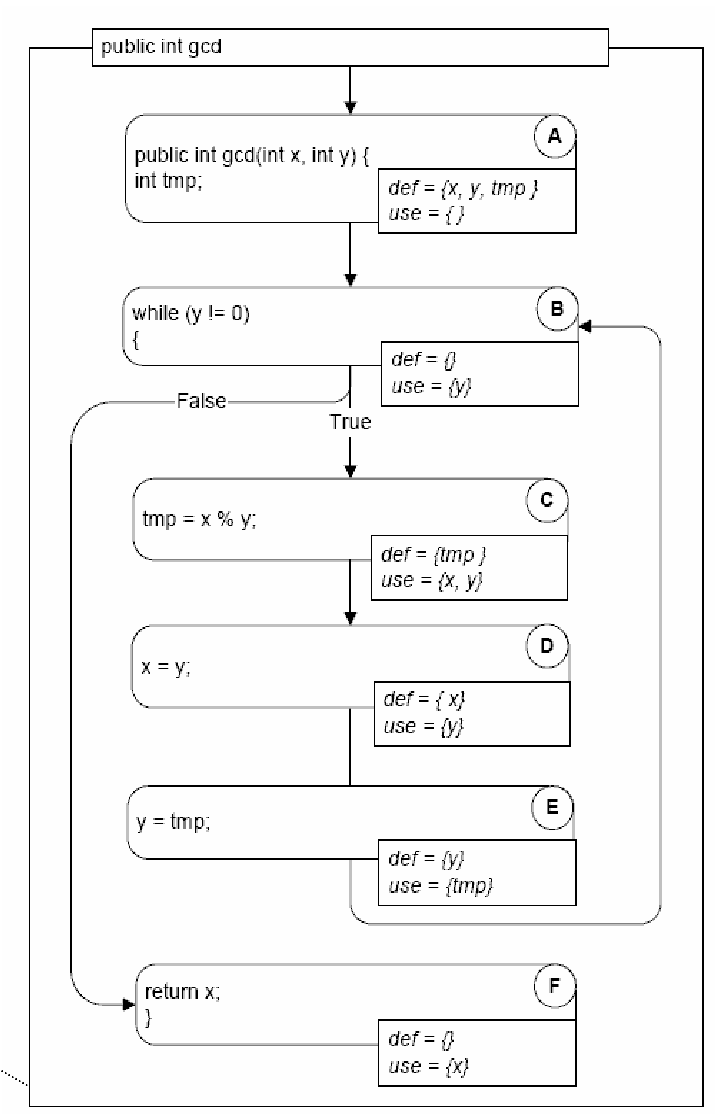
```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



Def-Use Pairs (3)

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

Figure 6.2, page 79



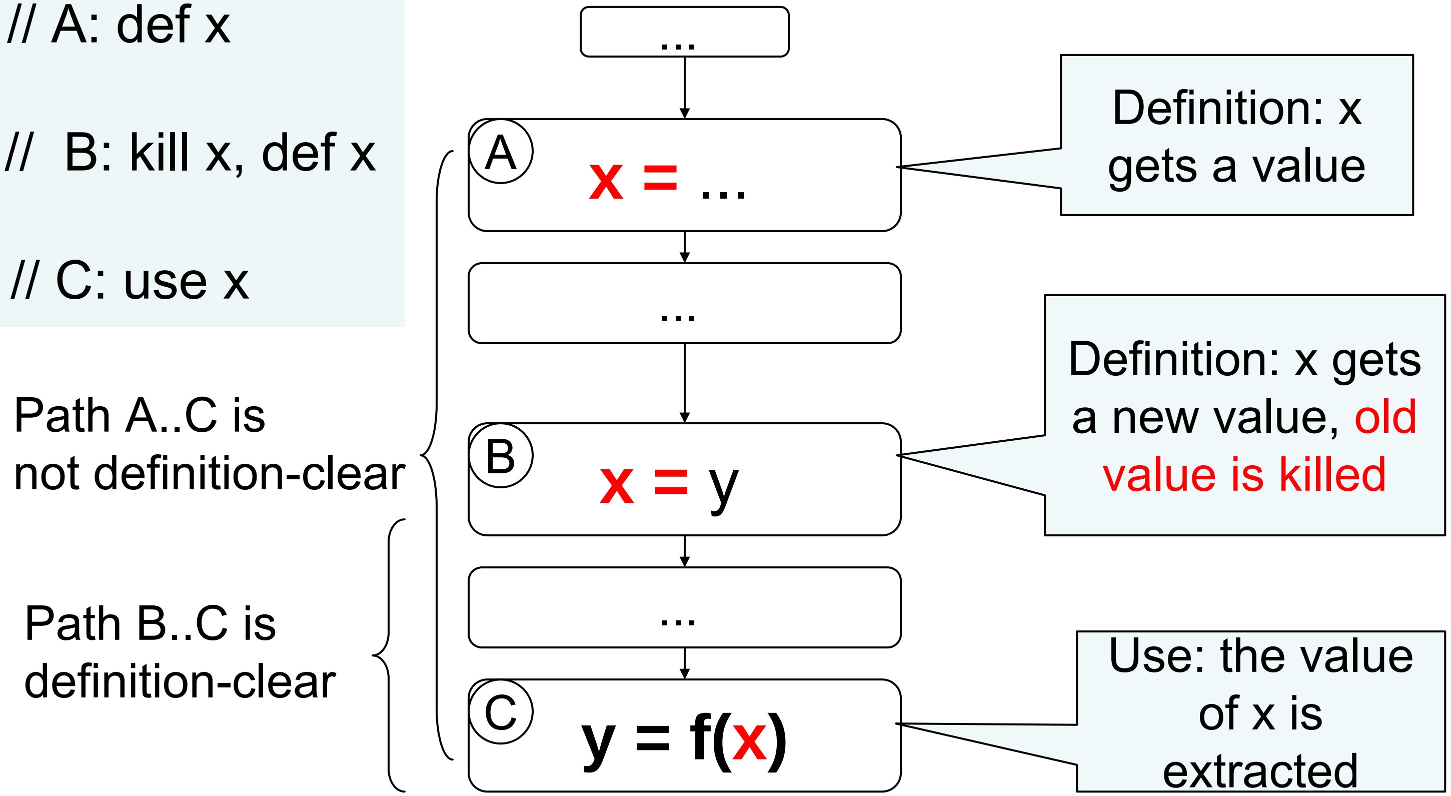
Def-Use Pairs (3)

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

**There is an over-simplification here, which we will repair later.*

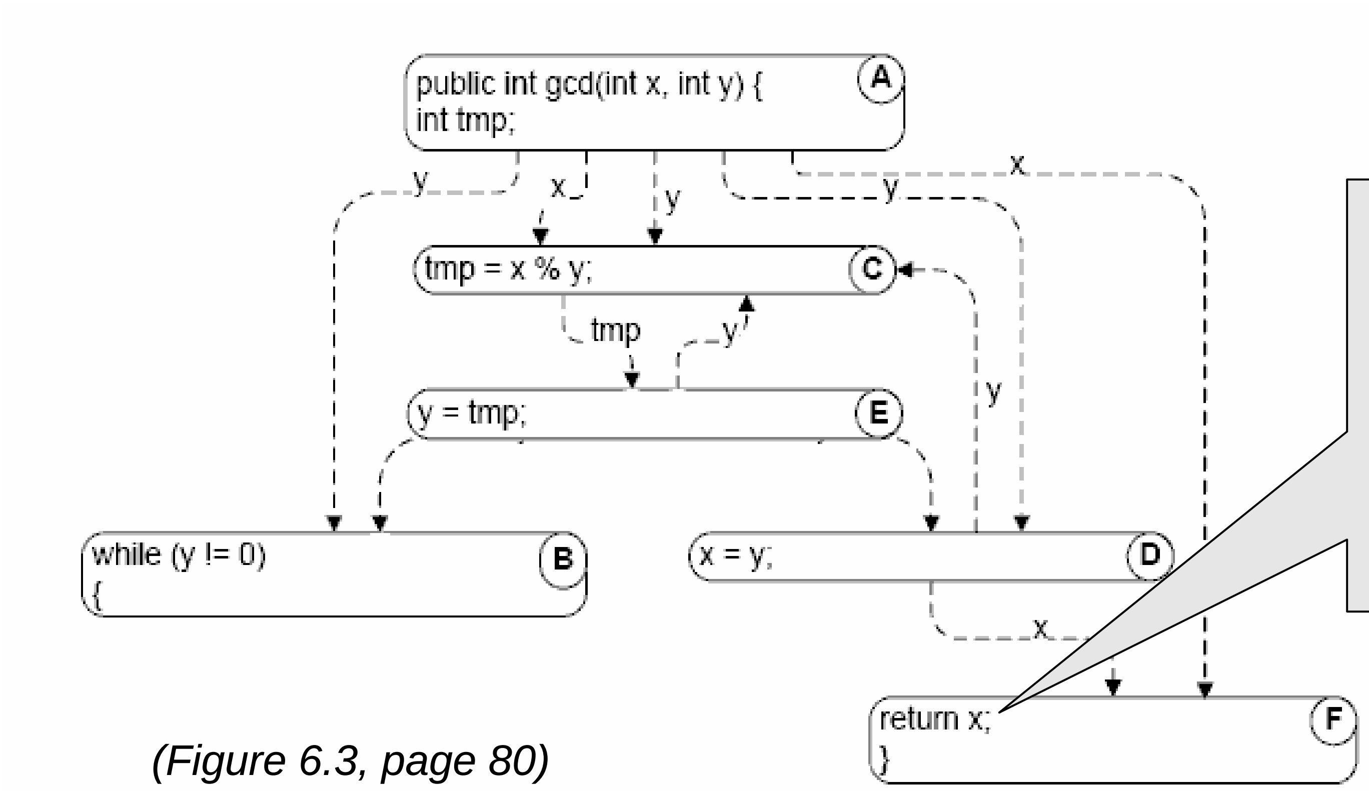
Definition-Clear or Killing

```
x = ...      // A: def x
q = ...
x = y;       //  B: kill x, def x
z = ...
y = f(x);  // C: use x
```


...
A
x = ...
...
B
x = y
...
C
y = f(x)
Definition: x gets a value
Definition: x gets a new value, old value is killed
Use: the value of x is extracted
Path A..C is not definition-clear
Path B..C is definition-clear

(Direct) Data Dependence Graph

- A direct data dependence graph is:
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (du) pairs, labelled with the variable name

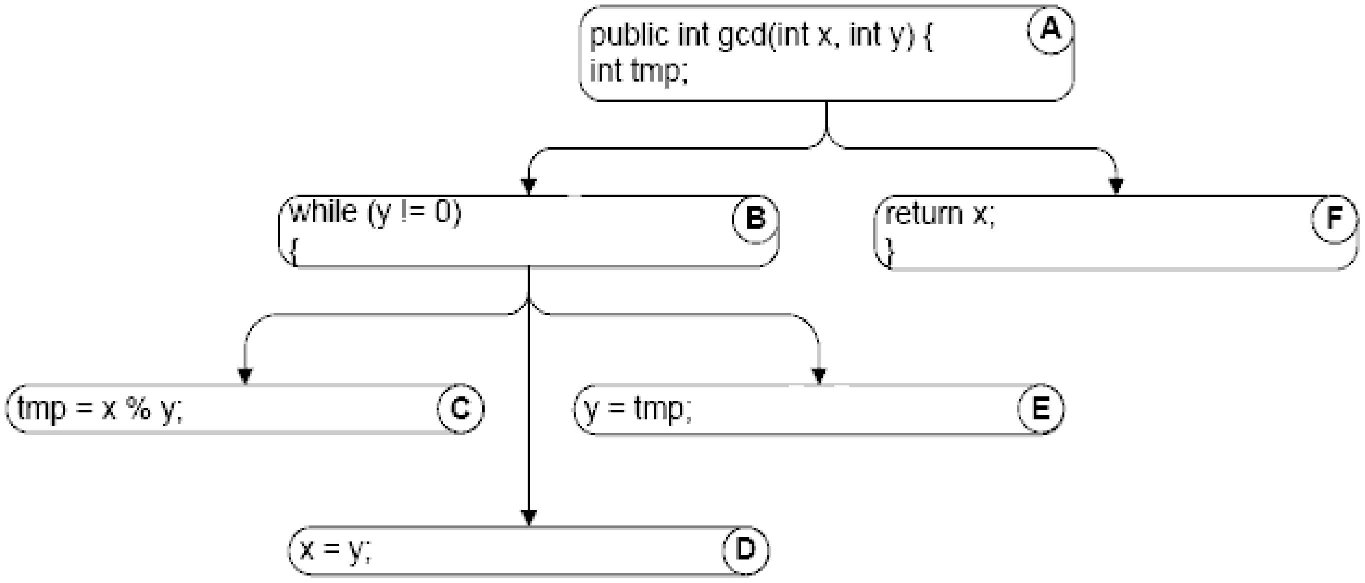


(Figure 6.3, page 80)

Dependence edges show this x value could be the unchanged parameter or could be set at line D

Control dependence (1)

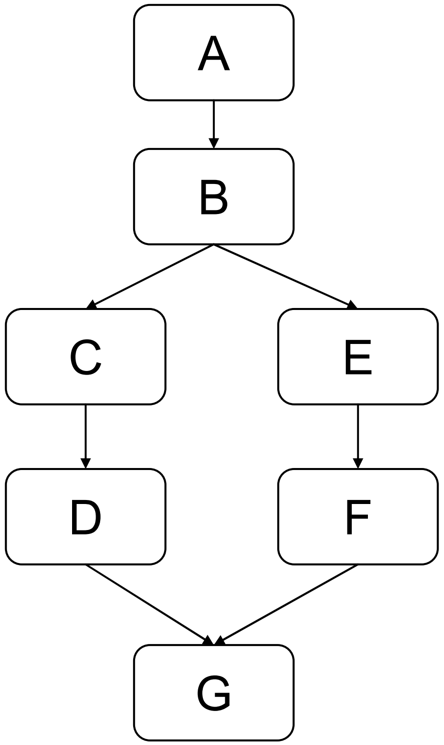
- Data dependence: Where did these values come from?
- Control dependence: Which statement controls whether this statement executes?
  - Nodes: as in the CFG
  - Edges: unlabelled, from entry/branching points to controlled blocks



Dominators

- **Pre-dominators** in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise.
- Node M dominates node N if every path from the root to N passes through M.
  - A node will typically have many dominators, but except for the root, there is a unique **immediate dominator** of node N which is closest to N on any path from the root, and which is in turn dominated by all the other dominators of N.
  - Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.
- **Post-dominators**: Calculated in the reverse of the control flow graph, using a special "exit" node as the root.

Dominators (example)

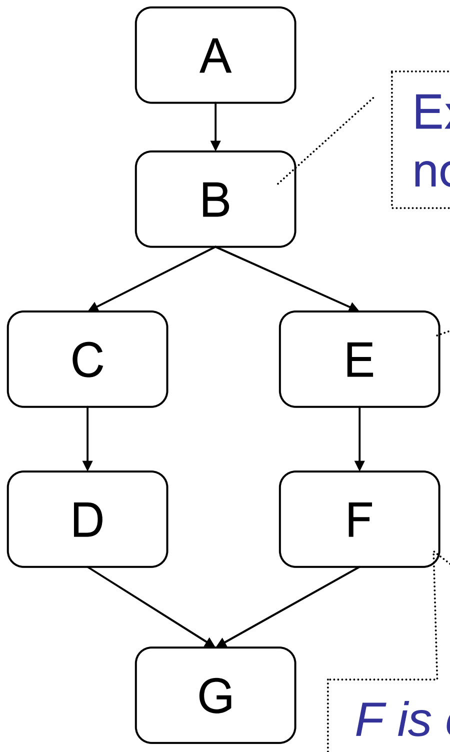


- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

Control dependence (2)

- We can use post-dominators to give a more precise definition of control dependence:
  - Consider again a node N that is reached on some but not all execution paths.
  - There must be some node C with the following property:
    - C has at least two successors in the control flow graph (i.e., it represents a control flow decision);
    - C is not post-dominated by N
    - there is a successor of C in the control flow graph that is post-dominated by N.
  - When these conditions are true, we say node N is control-dependent on node C.
    - Intuitively: C was the last decision that controlled whether N executed

Control Dependence



Execution of F is not inevitable at B

Execution of F is inevitable at E

F is control-dependent on B, the last point at which its execution was not inevitable

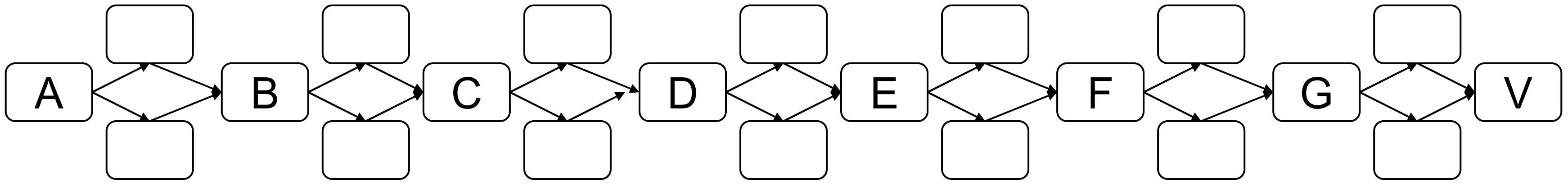
Data Flow Analysis

Computing data flow information

Calculating def-use pairs

- Definition-use pairs can be defined in terms of paths in the program control flow graph:
  - There is an association (d,u) between a definition of variable v at d and a use of variable v at u iff
    - there is at least one control flow path from d to u
    - with no intervening definition of v.
  - v_d **reaches** u (v_d is a **reaching definition** at u).
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point.
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

Exponential paths (even without loops)



2 paths from A to B

4 from A to C

8 from A to D

16 from A to E

...

128 paths from A to V

Tracing each path is not efficient, and we can do much better.

DF Algorithm

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node.
- Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p.
  - If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is generated at p.
  - If a definition v_p of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from p to n.

Equations of node E (y = tmp)

Calculate reaching definitions at E in terms of its immediate predecessor D

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

$$\text{Reach}(E) = \text{ReachOut}(D)$$

$$\text{ReachOut}(E) = (\text{Reach}(E) \setminus \{y_A\}) \cup \{y_E\}$$

Equations of node B (while (y != 0))

This line has two predecessors: Before the loop, end of the loop

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

- Reach(B) = ReachOut(A) $\cup$ ReachOut(E)
- ReachOut(A) = gen(A) = $\{x_A, y_A, tmp_A\}$
- ReachOut(E) = (Reach(E) \ $\{y_A\}$) $\cup$ $\{y_E\}$

General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}\ (n)) \cup \text{gen}(n)$$

$\text{gen}(n) = \{ v_n \mid v$ is defined or modified at $n \}$

$\text{kill}(n) = \{ v_x \mid v$ is defined or modified at x, $x \neq n \}$

Avail equations

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

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

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

Live variable equations

$$\text{Live}(n) = \bigcup_{m \in \text{succ}(n)} \text{LiveOut}(m)$$

$$\text{LiveOut}(n) = (\text{Live}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v \mid v \text{ is used at } n \}$$

$$\text{kill}(n) = \{ v \mid v \text{ is modified at } n \}$$

Classification of analyses

- Forward/backward: a node's set depends on that of its predecessors/successors
- Any-path/all-path: a node's set contains a value iff it is coming from any/all of its inputs

	Any-path ($\cup$)	All-paths ($\cap$)
Forward (pred)	Reach	Avail
Backward (succ)	Live	"inevitable"

Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
  - For “any path” problems, first guess is “nothing” (empty set) at each node
  - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a “fixed point” solution where every new calculation produces the same value as the previous guess.

Worklist Algorithm for Data Flow

See figures 6.6, 6.7 on pages 84, 86 of Pezzè & Young

One way to iterate to a fixed point solution.

General idea:

- Initially all nodes are on the work list, and have default values
  - Default for “any-path” problem is the empty set, default for “all-path” problem is the set of all possibilities (union of all gen sets)
- While the work list is not empty
  - Pick any node n on work list; remove it from the list
  - Apply the data flow equations for that node to get new values
  - If the new value is changed (from the old value at that node), then
    - Add successors (for forward analysis) or predecessors (for backward analysis) on the work list
- Eventually the work list will be empty (because new computed values = old values for each node) and the algorithm stops.

Cooking your own: From Execution to Conservative Flow Analysis

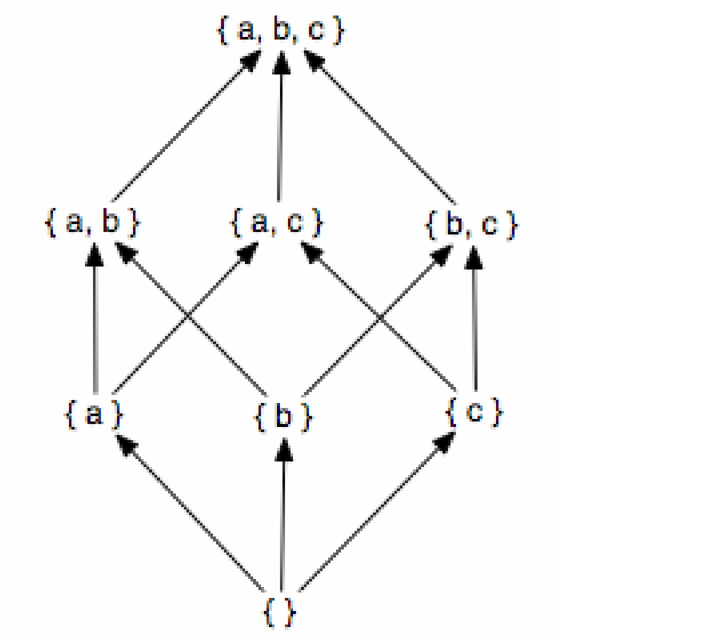
- We can use the same data flow algorithms to approximate other dynamic properties
  - Gen set will be “facts that become true here”
  - Kill set will be “facts that are no longer true here”
  - Flow equations will describe propagation
- Example: Taintedness (in web form processing)
  - “Taint”: a user-supplied value (e.g., from web form) that has not been validated
  - Gen: we get this value from an untrusted source here
  - Kill: we validated to make sure the value is proper

Cooking your own analysis (2)

- Flow equations must be monotonic
  - Initialize to the bottom element of a lattice of approximations
  - Each new value that changes must move up the lattice
- Typically: Powerset lattice
  - Bottom is empty set, top is universe
  - Or empty at top for all-paths analysis

Monotonic: y > x implies f(y) ≥ f(x)

(where f is application of the flow equations on values from successor or predecessor nodes, and ">" is movement up the lattice)



Data flow analysis with arrays and pointers

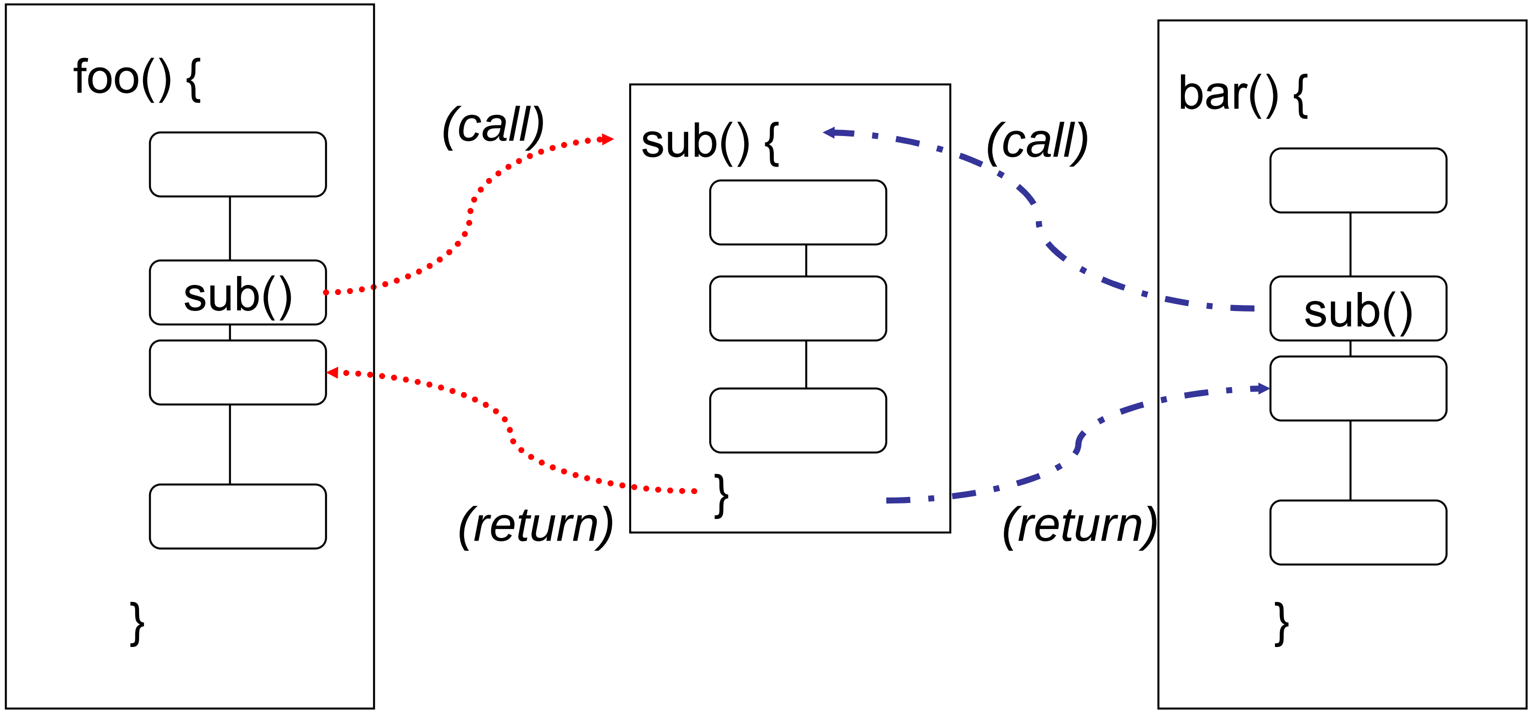
- Arrays and pointers introduce uncertainty:
  Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (**aliasing**)
- The uncertainty is accomodated depending to the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa

Scope of Data Flow Analysis

- Intraprocedural
  - Within a single method or procedure
    - as described so far
- Interprocedural
  - Across several methods (and classes) or procedures

- Cost/Precision trade-offs for interprocedural analysis are critical, and difficult
  - context sensitivity
  - flow-sensitivity

Context Sensitivity



A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()

Flow Sensitivity

- Reach, Avail, etc. were flow-sensitive, intraprocedural analyses
  - They considered ordering and control flow decisions
  - Within a single procedure or method, this is (fairly) cheap — $O(n^3)$ for n CFG nodes
- Many interprocedural flow analyses are flow-insensitive
  - $O(n^3)$ would not be acceptable for all the statements in a program!
    - Though $O(n^3)$ on each individual procedure might be ok
  - Often flow-insensitive analysis is good enough ... consider type checking as an example

Summary

- Data flow models detect patterns on CFGs:
  - Nodes initiating the pattern
  - Nodes terminating it
  - Nodes that may interrupt it
- Often, but not always, about flow of information (dependence)
- Pros:
  - Can be implemented by efficient iterative algorithms
  - Widely applicable (not just for classic “data flow” properties)
- Limitations:
  - Unable to distinguish feasible from infeasible paths
  - Analyses spanning whole programs (e.g., alias analysis) must trade off precision against computational cost