Finite Models

資科系
林偉川

Learning objectives

- Understand goals and implications of finite state abstraction
- Learn how to model program control flow with graphs
- Learn how to model the software system structure with call graphs
- Learn how to model finite state behavior with finite state machines

Why model

- Analysis and test cannot wait until the actual artifact is constructed
- It is impractical to test the actual artifact as thoroughly as we wish whether that means subjecting it to all foreseeable hurricane and earthquake forces, or to all possible program states and inputs
- Models permit us to start analysis earlier and repeat it as a design evolves, and allow us to apply analytic methods that cover a much larger class of scenarios than we can explicitly test

Why model

- The fundamental concepts and tradeoffs in the design of models is necessary for a full understanding of those test and analysis techniques, and is a foundation for devising new techniques and models to solve domain-specific problem
- A model is a representation that is simpler than the artifact it represents but preserves some important attributes of the actual artifact
- Model of program execution not the models of other attributes such as the effort required to develop the SW or it usability

Properties of a good Models

- **Compact**: representable and manipulable in a reasonably compact form (scale down)
  - What is *reasonably compact* depends largely on how the model will be used
  - Models intended for human inspection and reasoning must be small enough to be comprehensible
  - Models intended for automated analysis may be far too large and complex for human comprehension but still be sufficiently small for computer processing

Properties of a good Models

- **Predictive**: must represent some salient characteristics of the modeled artifact well enough to distinguish between *good* and *bad* outcomes of analysis
  - no single model represents all characteristics well enough to be useful for all kinds of analysis

Properties of Models

- **Semantically meaningful**: it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
  - A finite element model of a building predicts collapse in a category five hurricane to know enough about the collapse to suggest revisions to the design
  - A model of an accounting system predicts a failure when used concurrently by several clients, we need a description of that failure sufficient to suggest possible revisions

Properties of Models

- **Sufficiently general**: models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application
- Sometimes we tolerate limits on design imposed by limitations of our modeling and analysis techniques ➔ conventional design V.S. novel design? (have confidence in analysis techniques for the former but not the latter)

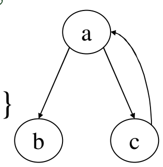
Properties of Models

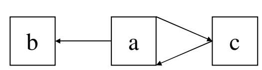
- Design models are intended to aid in making and evaluating design decision, they should share these characteristics with models constructed primarily for analysis. Ex. UML is designed for human communication, with less attention to semantic meaning and prediction
- Models are often used indirectly in evaluating an artifact or guide test case selection

Graph Representations: directed graphs

- Use directed graph to represent models of programs
- Directed graph:
  - N (set of nodes) nodes
  - E (relation on the set of nodes) edges
  - Edges represent some relation among the entities
- Program control flow using a directed graph model, an edge (a,b) would be interpreted as the statement " program region a can be directed followed by program region b in the program execution

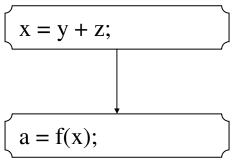
Nodes: {a, b, c}
Edges: {(a,b), (a, c), (c, a)}





Graph Representations: labels and code

- Node represent entities such as procedures or classes or regions of source code. Edge represent some relation among the entities
- 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:

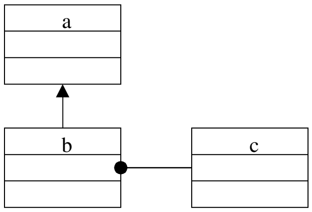


Multidimensional Graph Representations

- Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once (for clear communication)
  - 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)}



Multidimensional Graph Representations

```
class a12{}
class b extends a12{}
class c extends a12{}
class d extends c {  b  mm; }
class test {
public static void main(String args[]){
   a12 a1=new a12(); b b1=new b();  c c1=new c();
   d d1=new d();  a1=b1; a1=c1; a1=d1;
   b1=(b)a1;  c1=(c)a1;  d1=(d)a1; d1=(d)c1;
}}
```


Multidimensional Graph Representations

```
class a13{ public static void main(String [] ar){ new sub(); }}
class bb{ bb(boolean t){if (t)System.out.println("True");}}
class sub extends bb{
  class com1 {  com1() {}  }
  class com2 {  com2() {}  }
  sub(){  super (true); }
}
```

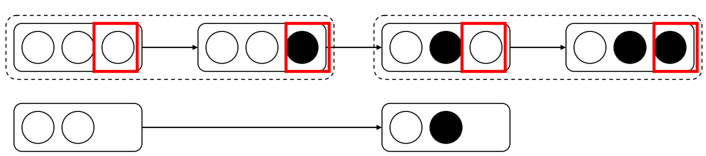
Finite Abstraction of Behavior

- Abstraction elides (omits) details of execution states and in so doing may cause an abstract model execution state to represent more than one concrete program execution state
- Program state is represented by three attributes, each with two possible values, drawn as light or dark circles
- Abstract model states retain the first two attributes and elide the third.

Finite Abstraction of Behavior

an abstraction function suppresses some details of program execution

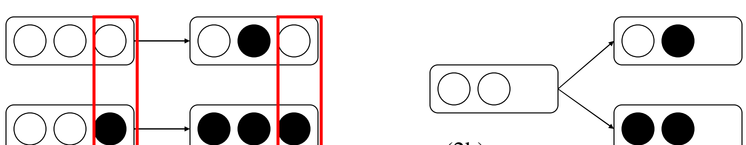
(1a) (1b)



⇒

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

(2a)



(2b)

Finite Abstraction of Behavior

- The relation between (1a) and (1b) illustrates coarsening (rough) of the execution model, since the first and the third program execution steps modify only the omitted attribute
- The relation between (2a) and (2b) illustrates introduction of non-determinism, because program execution states with different successor states have been merged

Finite abstractions of behavior

- A single program execution can be viewed as a sequence of states alternating with actions. The possible behaviors of a program are a set of such sequences
- The most trivial programs the set of possible execution sequences is infinite
- The whole set of states and transitions is called the state space of the program. Models of program execution are abstractions of that space
- States in the state space of program execution are related to states in a finite model of execution by an abstraction function

Finite abstractions of behavior

- An abstraction function hides some details of program execution, it combines together execution states that differ with respect to the hidden details but are not identical
- Two effects of abstraction are shown : the execution model is coarsened (sequences of transitions are collapsed into fewer execution steps), and non-determinism is introduced (because information required to make deterministic choice is sacrificed)

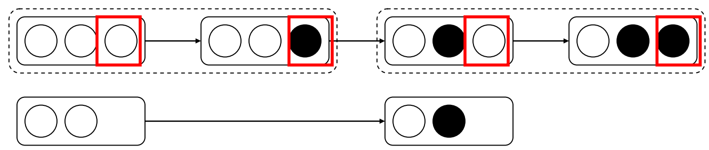
Finite abstractions of behavior

- Finite models of program execution are imperfect. Collapsing the potentially infinite states of actual execution into a finite number of representative model states involves omitting some information ➔ the omitting information may be hoped to be irrelevant to the property one wishes to verify, this is not completely true
- 2 (a) and 2 (b) illustrates how abstraction can cause a set of deterministic transitions to be modeled by a nondeterministic choice among transitions and making the analysis imprecise ➔ "false alarm" in the analysis of models

Finite Abstraction of Behavior

an abstraction function suppresses some details of program execution

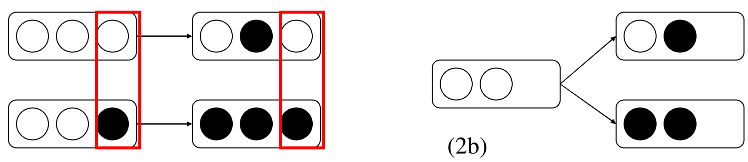
(1a) (1b)



⇒

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

(2a) (2b)



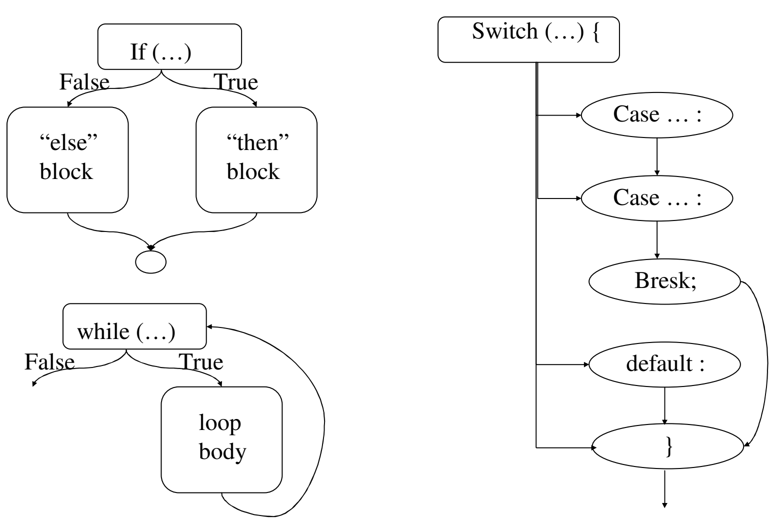
(Intra-procedural) Control Flow Graph

- To construct models whose states are closely related to locations in program source code ➔ associate an abstract state with a whole region (a set of locations) in a program
- Program source code is finite, so a model that associates a finite amount of information with each of a finite number of program points or regions will be finite
- Control flow of a single procedure or method can be represented as an intra-procedural control flow graph, abbreviated as control flow graph (CFG)

(Intra-procedural) Control Flow Graph

- The intra-procedural control flow graph is a directed graph in which :
  - 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 either through sequential execution or by a branch

Typical control flow constructs in a CFG



Control Flow Graph

- A CFG model retains some information about the program counter and omits other information about program execution. Information that determines the outcome of conditional branches is omitted, the CFG represents not only possible program paths but also some paths that cannot be executed
- The node in a CFG could represent individual program statements, even individual machine operations, but it is desirable to make the graph model as compact and simple as possible

Control Flow Graph

- Nodes in a CFG model of a program represent not a single point but rather a basic block, a maximal program region with a single entry and single exit point
- A basic block unites adjacent, sequential statements of source code, but in some cases a single syntactic program statement is broken across basic blocks to model control flow within the statement

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();
}
```


Control Flow Graph

- A sequence of two statements within the loop has been collapsed into a single block, but the for statement and the complex predicate in the if statement have been broken across basic blocks to model their internal flow of control
- A directed edge leads from the start node to the node representing the first executable block, and a directed edge from each procedure exit (each return statement and the last sequential block in the program) to the distinguished end node
- The procedure should draw a start node identified with the procedure body, and to leave the end node

Control Flow Graph

- The intra-procedural CFG may be used to define thoroughness criteria for testing.
- Some criteria are defined by reference to Linear Code Sequences And Jumps (LCSAJs), which are essentially sub-paths of the CFG from one branch to another
- The java exception handling happened by the API "String.charAt()" would terminate the program if the argument is an empty string
- This could be represented in the CFG as a directed edge to an exit node

Control Flow Graph

- However, if one includes such implicit control flow edges for every possible exception, the CFG becomes very large
- It may not be simple to determine which of the implicit control flow edges can be executed → we can assume that the cIdx of "String.charAt(cIdx)" used in the for loop would within bounds, but we cannot expect an automated tool for extracting CFG to perform such inferences.
- Whether to include some or all implicit control flow edges in a CFG representation therefore involves a trade-off between possibly omitting some execution paths or representing many spurious (false) paths

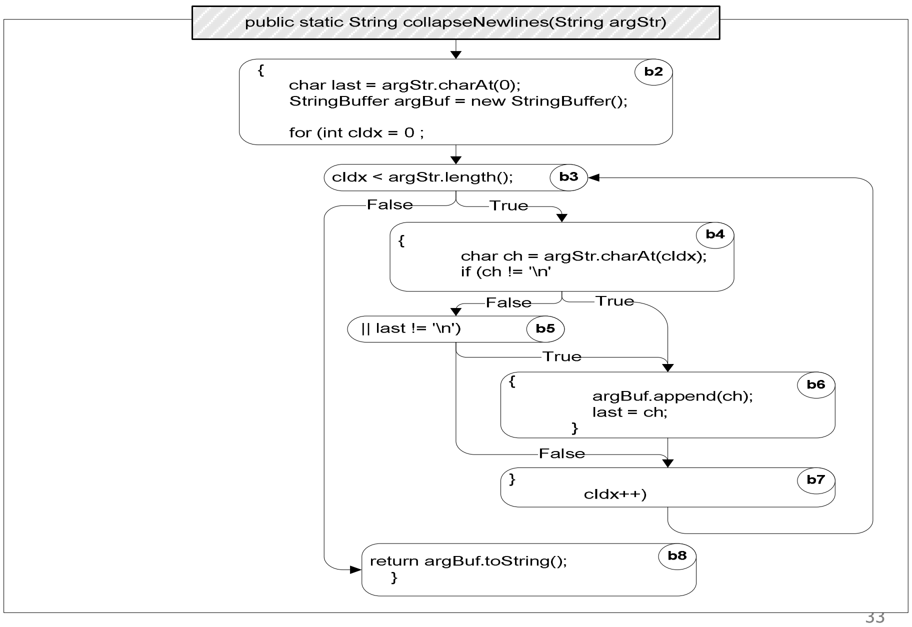
Control Flow Graph

- Which is preferable depends on the uses where the CFG representation will be put
- The representation of explicit control flow may differ depending on the uses where a model is put
- For statement has been broken into its constituent parts (initialization, comparison, and increment for the next iteration), each of which appears at a different point in the control flow

Control Flow Graph

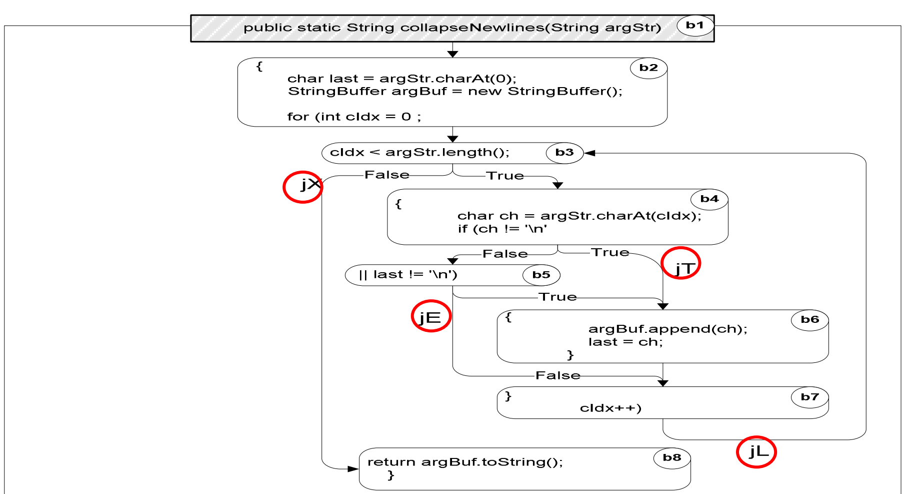
- A complex conditional expression in Java or C is executed by "short-circuit" evaluation➔ I>0 && I<10 can be broken across two basic blocks
- If this fine level of execution detail is not relevant to an analysis, we may choose to ignore short-circuit evaluation and treat the entire conditional expression as if it were fully evaluated

Example of Control Flow Graph



Linear Code Sequence and Jump (LCSAJ)

Essentially sub-paths of the control flow graph from one branch to another



Linear Code Sequence and Jump (LCSAJ)

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

How about the GCD() ?

```
public int GCD(int a, int b) {
    int m;

    while ((m=a%b) != 0)   {
        a=b; b=m
     } // end while
     return b;
}
```


36

How about the Prime() ?

```
public int prime(int a) {
    int i,m;

    i=2; m=1;
    while ((i<=a-1)   {
        if (a%i == 0) { m=0; break }
        i++;
     } // end while
     return m;
}
```


37

How about the binarySearch() ?

```
public int binarySearch(int sortedArray[ ], int searchValue) {
    int bottom = 0;   int top = sortedArray.length - 1;
    int middle, locationOfsearchValue;
    boolean found = flase;
    locationOfsearchValue = -1;  /* the location of searchValue in the sortedArray */
                                 /* location = -1 means that searchValue is not found */
    while ( bottom <= top && !found)   {
        middle = (top + bottom)/2;
        if (searchValue == sortedArray[ middle ])   {
            found = true;  locationOfsearchValue = middle;
        }
        else if (searchValue < sortedArray[ middle ]) top = middle - 1;
        else bottom = middle + 1;
      } // end while
      return locationOfsearchValue;
}
```


38

Testing Methods

- Two general software testing methods:
  - White-box testing: (logic-driven)
    - Design tests to exercise internal structures of the software to make sure they operates according to specifications and designs
  - Black-box testing: (data-driven or input/output-driven)
    - Design tests to exercise each function of the software and check its errors.
  - White-box and black-box testing approaches can uncover different class of errors and are complement each other

White-Box Testing

- White-box testing
  - Also known as glass-box testing or structural testing
  - Has the knowledge of the program's structures
  - A test case design method that uses the control structure of the procedural design to derive test cases
  - Focus on the control structures, logical paths, logical conditions, data flows, internal data structures, and loops.
  - W. Hetzel describes white-box testing as "testing in the small"

White-Box Testing

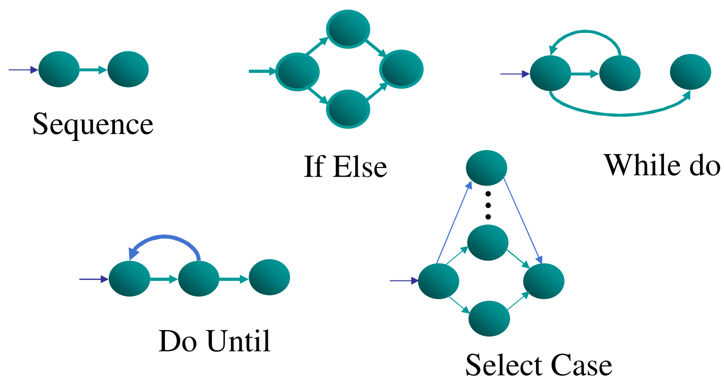
- Using white-box testing methods, we can derive test cases that
  - Guarantee that all independent paths within a module have been exercised at least once.
  - Exercise all logical decisions on their true and false sides.
  - Execute all loops at their boundaries and within their operational bounds.
  - Exercise internal data structures to assure their validity.

Basis Path Testing

- Basic path testing (a white-box testing technique):
  - First proposed by Tom McCabe.
  - Can be used to derive a logical complexity measure for a procedure design.
  - Used as a guide for defining a basis set of execution path.
  - Guarantee to execute every statement in the program at least one time.

Basis Path Testing

- The basic structured-constructs in a flow graph :



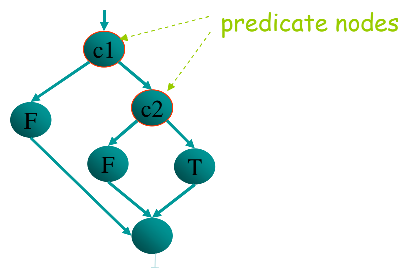
Basis Path Testing

- Flow graph notation (control flow graph)
  - Node represents one or more procedural statements
    - A sequence of process boxes and a decision diamond can map into a single node
    - A **predicate node** is a node with two or more edges emanating (exit) from it
  - Edge (or link) represents flow of control
  - Region: areas bounded by edges and nodes
    - When counting regions, include the area outside the graph as a region

Basis Path Testing

- Compound condition
  - Occurs when one or more Boolean operators (OR, AND, NAND, NOR) is present in a conditional statement
  - A separate node is created for each of the conditions *C1* and *C2* in the statement *IF C1 AND C2*

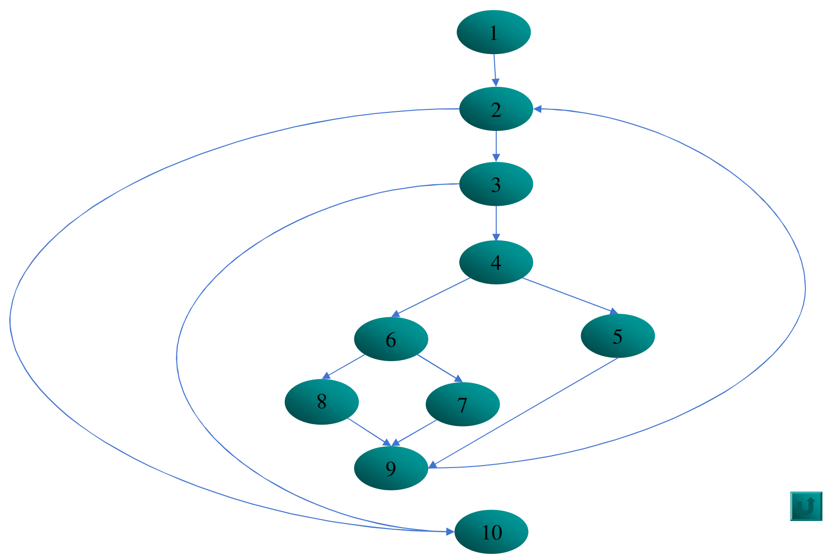
```
if (c1 AND c2) then
    print T;
else
    print F;
end if;
```



binarySearch() Example

```
public int binarySearch(int sortedArray[ ], int searchValue) {
(1)   int bottom = 0, top = sortedArray.length - 1;
      int middle, locationOfsearchValue;
      boolean found = flase;
      locationOfsearchValue = -1;   /* the location of searchValue in the sortedArray    */
                                     /* location = -1 means that searchValue is not found */
(2)(3) while ( bottom <= top && !found)
      {
(4)      middle = (top + bottom)/2;
         if (searchValue == sortedArray[ middle ])
         {
(5)          found = true;
             locationOfsearchValue = middle;
         }
(6)      else if (searchValue < sortedArray[ middle ])
(7)            top = middle - 1;
(8)      else
               bottom = middle + 1;
(9)   } // end while
(10)  return locationOfsearchValue;
}
```

The CFG of Function binarySearch()



Cyclomatic Complexity

- Cyclomatic complexity is a software metric
  - provides a quantitative measure of the global complexity of a program.
  - When this metric is used in the context of the basis path testing
    - the value of cyclomatic complexity defines the number of independent paths in the basis set of a program
    - the value of cyclomatic complexity defines an upper bound of number of tests (i.e., paths) that must be designed and exercised to guarantee coverage of all program statements

Cyclomatic Complexity

- Independent path
  - An independent path is any path of the program that introduce at least one new set of procedural statements or a new condition
  - An independent path must move along at least one edge that has not been traversed before the path is defined
    - Examples: consider the CFG of binarySearch()
      - Path 1: 1-2-10
      - Path 2: 1-2-3-4-6-8-9-2-10
      - Path 3: 1-2-3-4-6-8-9-2-3-10

      } independent paths
      - Path 4: 1-2-3-4-6-8-9-2-3-4-6-8-9-2-10 (not an independent path)

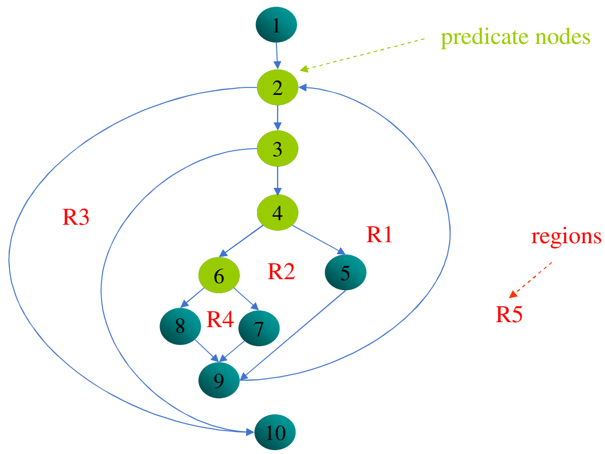
Cyclomatic Complexity

- Three ways to compute cyclomatic complexity:
  - The number of regions of the flow graph correspond to the cyclomatic complexity.
  - Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E - N + 2 (13-10+2=5)

    where E is the number of flow graph edges and N is the number of flow graph nodes.
  - Cyclomatic complexity, V(G) = P + 1

    where P is the number of predicate nodes contained in the flow graph G.

Cyclomatic Complexity of Function binarySearch()



Deriving Basis Test Cases

- The following steps can be applied to derive the basis set:
  1. Using the design or code as a foundation, draw the corresponding flow graph.
  2. Determine the cyclomatic complexity of the flow graph.
     - **V(G) = 5 regions**
     - **V(G) = 13 edges – 10 nodes + 2 = 5**
     - **V(G) = 4 predicate nodes + 1 = 5**

Deriving Basis Test Cases

3. Determine a basis set of linearly independent paths.
   - Path 1: 1-2-10
   - Path 2: 1-2-3-10
   - Path 3: 1-2-3-4-5-9-2- …
   - Path 4: 1-2-3-4-6-7-9-2-…
   - Path 5: 1-2-3-4-6-8-9-2-…
4. Prepare test cases that force the execution of each path in the basis set
   - Path 1 test case:
     - Inputs: sortedArray = { }, searchValue = 2
     - Expected results: locationOfSearchValue = -1

Deriving Basis Test Cases

- Path 2 test case: cannot be tested stand-alone!
  - Inputs: sortedArray = {2, 4, 6}, searchValue = 8
  - Expected results: locationOfSearchValue = -1
- Path 3 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 6
  - Expected results: locationOfSearchValue = 2
- Path 4 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 4
  - Expected results: locationOfSearchValue = 1
- Path 5 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 10
  - Expected results: locationOfSearchValue = 4

Deriving Basis Test Cases

- Each test cases is executed and compared to its expected results.
- Once all test cases have been exercised, we can be sure that all statements are executed at least once
- Note: some independent paths cannot be tested stand-alone because the input data required to traverse the paths cannot be achieved
  - In binarySearch(), the initial value of variable *found* is `FALSE`, hence path 2 can only be tested as part of path 3, 4, and 5 tests

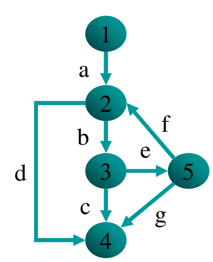
Graph Matrices

- A graph matrix
  - A tabular representation of a flow graph
  - A square matrix with a size equal to the number of nodes on the flow graph ➔ row
  - Matrix entries correspond to the edges between nodes
  - Adding link weight to each edge to represent
    - The connection between nodes
    - The probability of the edge to be executed
    - The resource (e.g., processing time or memory) required for traversing the edge

Graph Matrices

- A connection matrix
  - A graph matrix with the link weight is 1 (representing a connection exists) or 0 (representing a connection does not exist)
  - Each row of the matrix with two or more entries represents a predicate node
  - Provide another method for computing the cyclomatic complexity of a flow graph

Graph Matrices



Flow Graph

Node \ Connected to node

	1	2	3	4	5
1		a			
2			b	d	
3				c	e
4					
5		f		g	

Graph Matrix

	1	2	3	4	5
1		1			
2			1	1	
3				1	1
4					
5		1		1	

Connection Matrix

Connections

1 - 1 = 0

2 - 1 = 1

2 - 1 = 1

2 - 1 = 1

----

3 + 1 = 4

Cyclomatic complexity

Call graphs

- Intra-procedural control flow graph represents possible execution paths through a single procedure or method
- Inter-procedural control flow can also be represented as a directed graph named call graphs
  - Nodes represent procedures
    - Methods
    - C functions
    - ...
  - Edges represent *calls* relation

Call graphs

- Call graph present many more design issues and tradeoffs than intra-procedural control flow graph ➔ there are many variations on the basic call graph representation
- In OO language, method calls are made through object references and may be bound to methods in different subclasses depending on the current binding of the object
- A call graph for programs in an OO language might represent the call relation to each of the possible methods where a call might be dynamically bound

Call graphs

- The call graph will represent only a call to the method in the declared class of an object, but it will be part of a richer representation that include inheritance relation
- Construct an abstract model of execution in the course of analysis will involve interpreting this richer structure
- Sometime it may overestimation of the call relation due to dynamic dispatch. The static call graph includes calls through dynamic binding that never occur in execution

Call graphs

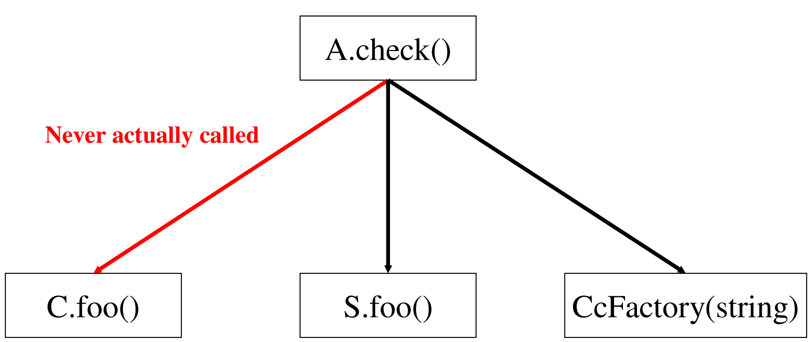
- If a call graph model represents different behaviors of a procedure depending on where the procedure is call ➔ context sensitive
- Context sensitive analysis can be more precise than context-insensitive analysis when the model includes some additional information that is shared or passed among procedures
- Information not only about the immediate calling context, but about the entire chain of procedure calls may be needed

Overestimating the *calls* relation

```
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();    }
}
```

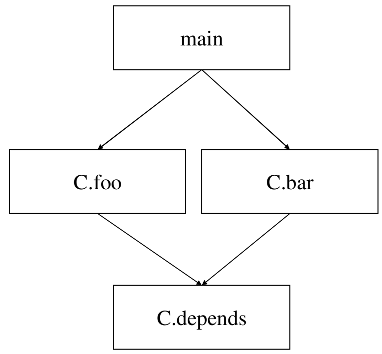

Overestimating the *calls* relation

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



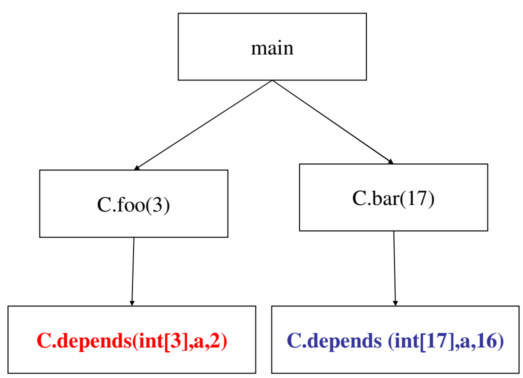
Contex Insensitive Call graphs(no parameter)

```
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(with parameter)

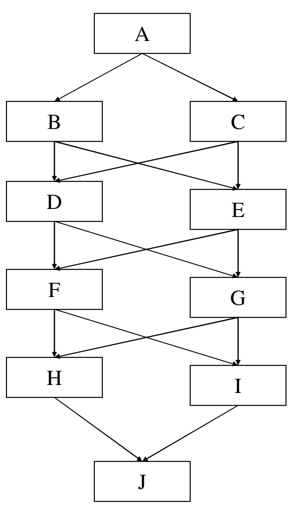
```
public class Context {
   public static void main(String args[]) {
    Context c = new Context();
    c.foo(3); c.bar(17);
   }
   void foo(int n) {
    int[]  myArray = new int[n];
    depends( myArray, 2) ;
   }
   void bar(int n) {
    int[]  myArray = new int[n];
    depends( myArray, 16) ;
   }
   void depends( int[] a, int n ) {a[n] = 42; }
}
```



Context Sensitive CFG exponential growth

- The cost of context-sensitive analysis depends on the number of paths from the root (main program) to each lowest level procedure
- The number of paths can be exponentially larger than the number of procedures

Context Sensitive CFG exponential growth



1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

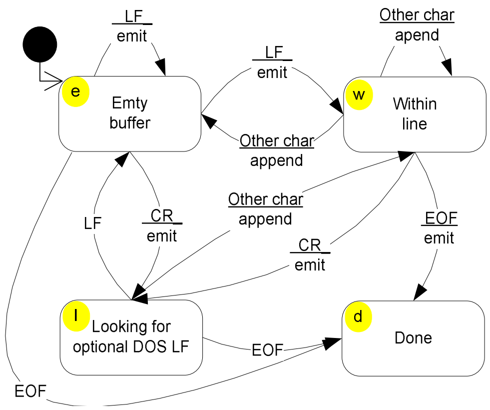
8 contexts …

16 calling contexts …

Finite state machines

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

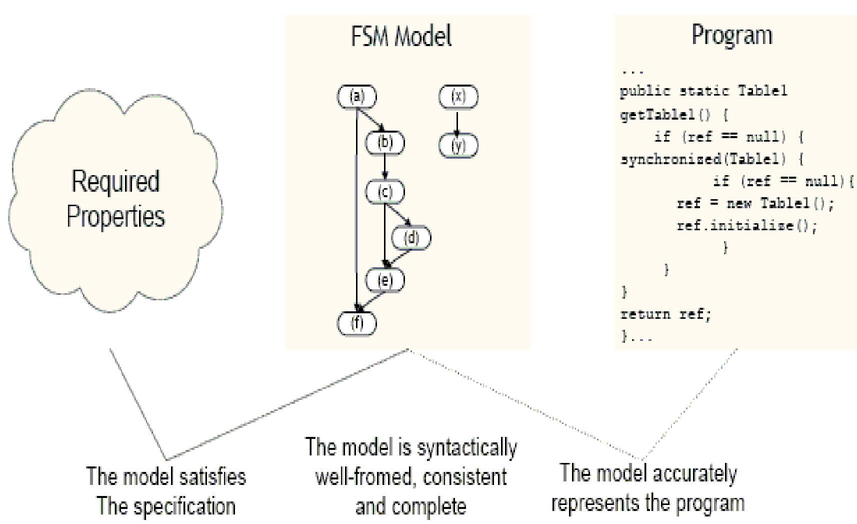
Graph representation (Mealy machine)



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

Using Models to Reason about System Properties



```
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  }
```

Abstraction Function

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

Summary

- Models must be much simpler than the artifact they describe to be understandable and analyzable
- Must also be sufficiently detailed to be useful
- CFG are built from software
- FSM can be built before software to document intended behavior