Finite Models
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
Properties of Models

- **Compact**: representable and manipulable in a reasonably compact form
  - What is *reasonably compact* depends largely on how the model will be used
- **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
- **Semantically meaningful**: it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
- **Sufficiently general**: models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application
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:

```
x = y + z;
a = f(x);
```
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
public static String collapseNewlines(String argStr) {
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();
    for (int cldx = 0; cldx < argStr.length(); cldx++) {
        char ch = argStr.charAt(cldx);
        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) {
    char last = argStr.charAt(0);
    String argBuf = new StringBuffer();
    for (int cIdx = 0; cIdx < argStr.length(); cIdx++) {
        char ch = argStr.charAt(cIdx);
        if (ch != '
' || last != '
') {
            argBuf.append(ch);
            last = ch;
        }
        return argBuf.toString();
    }
    return argBuf.toString();
}
```

<table>
<thead>
<tr>
<th>From</th>
<th>Sequence of basic blocs</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry b1 b2 b3</td>
<td>jX</td>
<td></td>
</tr>
<tr>
<td>Entry b1 b2 b3 b4</td>
<td>jT</td>
<td></td>
</tr>
<tr>
<td>Entry b1 b2 b3 b4 b5</td>
<td>jE</td>
<td></td>
</tr>
<tr>
<td>Entry b1 b2 b3 b4 b5 b6 b7</td>
<td>jL</td>
<td></td>
</tr>
<tr>
<td>jX</td>
<td>b8</td>
<td>ret</td>
</tr>
<tr>
<td>jL</td>
<td>b3 b4</td>
<td>jT</td>
</tr>
<tr>
<td>jL</td>
<td>b3 b4 b5</td>
<td>jE</td>
</tr>
<tr>
<td>jL</td>
<td>b3 b4 b5 b6 b7</td>
<td>jL</td>
</tr>
</tbody>
</table>
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.

```java
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();
    }
}
```
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

```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;
    }
}
```

(c) 2007 Mauro Pezzè & Michal Young
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

<table>
<thead>
<tr>
<th></th>
<th>LF</th>
<th>CR</th>
<th>EOF</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>e/emit</td>
<td>e/emit</td>
<td>d/-</td>
<td>w/append</td>
</tr>
<tr>
<td>w</td>
<td>e/emit</td>
<td>e/emit</td>
<td>d/emit</td>
<td>w/append</td>
</tr>
<tr>
<td>l</td>
<td>e/-</td>
<td>d/-</td>
<td>w/append</td>
<td></td>
</tr>
</tbody>
</table>
Using Models to Reason about System Properties

The model satisfies The specification
The model is syntactically well-formed, consistent and complete
The model accurately represents the program
Abstraction Function

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

#define BUFSIZE 1000
char buf[BUFSIZE];    /* 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 >= BUFSIZE-2) fail("Buffer overflow");
        buf[pos++] = inChar;
    } /* switch */
    if (pos > 0) {
        emit(buf, pos);
    }

}
```

---

(c) 2007 Mauro Pezzè & Michal Young

---

(c) 2007 Mauro Pezzè & Michal Young

Ch 5, slide 18
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