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)\}

\begin{center}
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (b) at (-1,-1) {b};
  \node (c) at (1,-1) {c};
  \draw (a) -- (b);
  \draw (a) -- (c);
\end{tikzpicture}
\end{center}
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 $C$

$$
\begin{align*}
\text{extends relation} \\
\text{NODES} &= \{A, B, C\} \\
\text{EDGES} &= \{(A,B)\}
\end{align*}
$$

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
  - Sometimes 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
Interprocedural control flow graph

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

Example of Control Flow Graph

```java
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 != ' ' && ch != '
') {
            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

Overestimating the calls relation

The static call graph includes calls through dynamic bindings that never occur in execution.
Contex Insensitive Call graphs

```java
public class Context {
    public static void main(String[] args) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
        c.depends(arg0);
    }

    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);
        c.depends(arg0);
    }

    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 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)

<table>
<thead>
<tr>
<th>Event</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

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

Abstraction Function

<table>
<thead>
<tr>
<th>Abstract state</th>
<th>Concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lines</td>
</tr>
<tr>
<td>c (Empty buffer)</td>
<td>3–13</td>
</tr>
<tr>
<td>w (Within line)</td>
<td>13</td>
</tr>
<tr>
<td>l (Looking for LF)</td>
<td>13</td>
</tr>
<tr>
<td>d (Done)</td>
<td>36</td>
</tr>
</tbody>
</table>

```
/** Convert each line from standard input */

void translate() {

    int BUFFERLEN = 1000;

    char inputBuffer[BUFFERLEN];
    // Accumulate line into this buffer

    int pos = 0;

    char inputChar; /* Next character in buffer */

    int atCR = 0; /* At 'within-line' * LF * optional DOS LF */

    while (inputChar = getchar()) != EOF) {

        switch (inputChar) {

        case LF:
            if (atCR) { /* Optional DOS LF */
                atCR--;
            } else { /* Encountered CR within line */
                atCR = 0;
                pos = 0;
            }
            break;

        case CR:
            atCR = 0;
            pos = 0;
            atCR = 1;
            break;

        default:
            if (pos --> BUFFERLEN) { /* Buffer overflow */
                outputBuffer[outputPos] = inputChar;
                outputPos++;
                atCR = 0;
            } else { /* Write line */
                outputBuffer[outputPos] = inputChar;
                outputPos++;
                atCR = 0;
            }
            break;

        if (pos = 0) {
            return;
        }
    }
    return;

    // EOF

    if (atCR) {
        outputBuffer[outputPos] = (char) 0x0A;
        outputPos++;
    }
    return;
```

(c) 2007 Mauro Pezzè & Michal Young