Finite Models

Learning objectives

- Be able to explain the relationship between a finite state abstraction and the original system and describe some consequences of this when the model is used to approximate the original system.
- 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

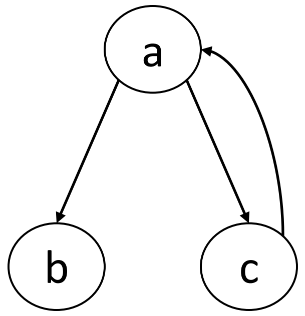
- **Compact**: representable and manipulable in a reasonably compact form
  - What is *reasonably compact* depends largely on how the model will be used
- **Predictive**: must represent some salient characteristics of the modeled artifact well enough to distinguish between *good* and *bad* outcomes of 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

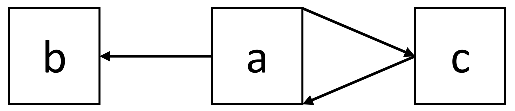
Models approximate/abstract

Graph Representations: directed graphs

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

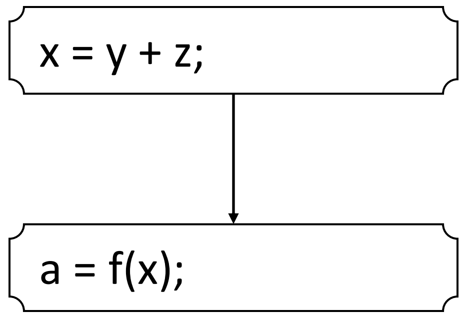
Nodes: {a, b, c}
Edges: {(a,b), (a, c), (c, a)}





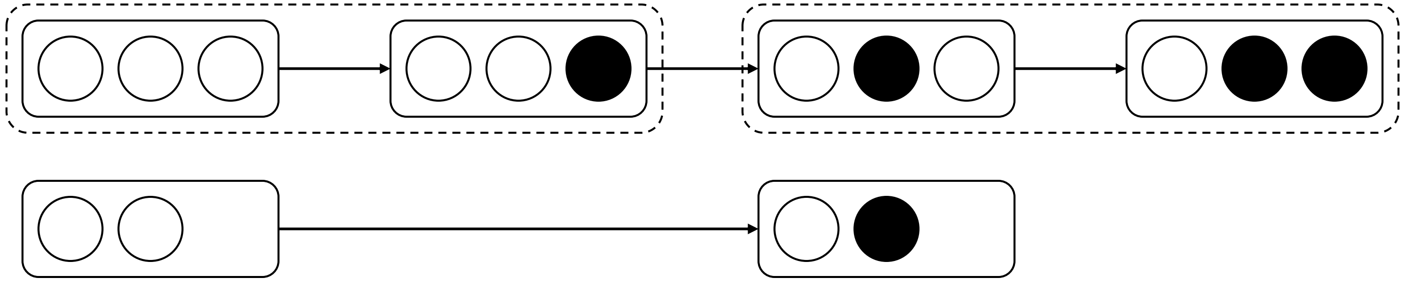
Graph Representations: labels and code

- 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:



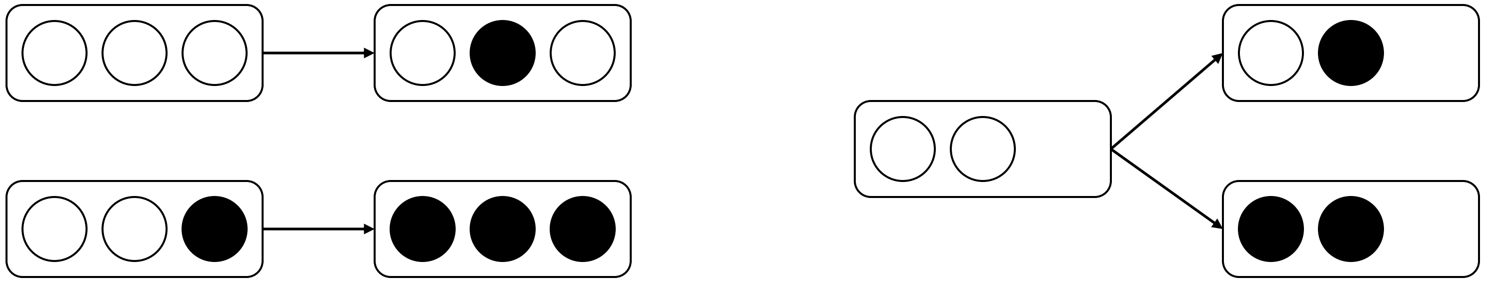
Finite Abstraction of Behavior

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

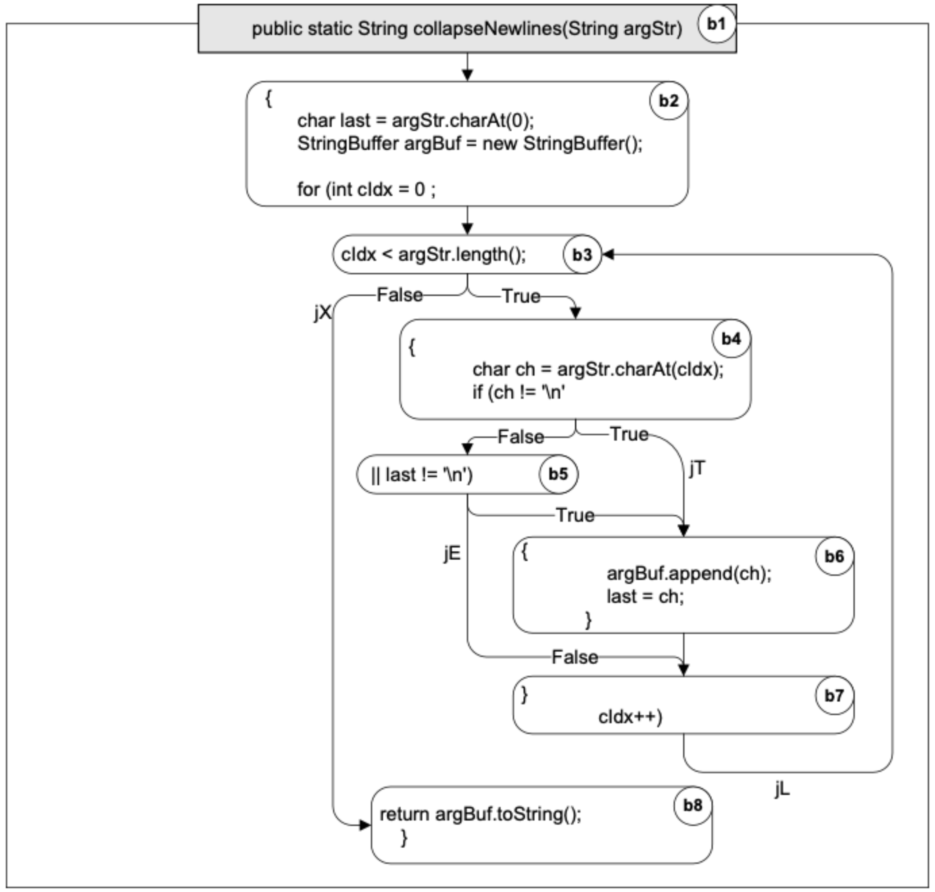


(Intraprocedural) Control Flow Graph

- 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();
   }
```

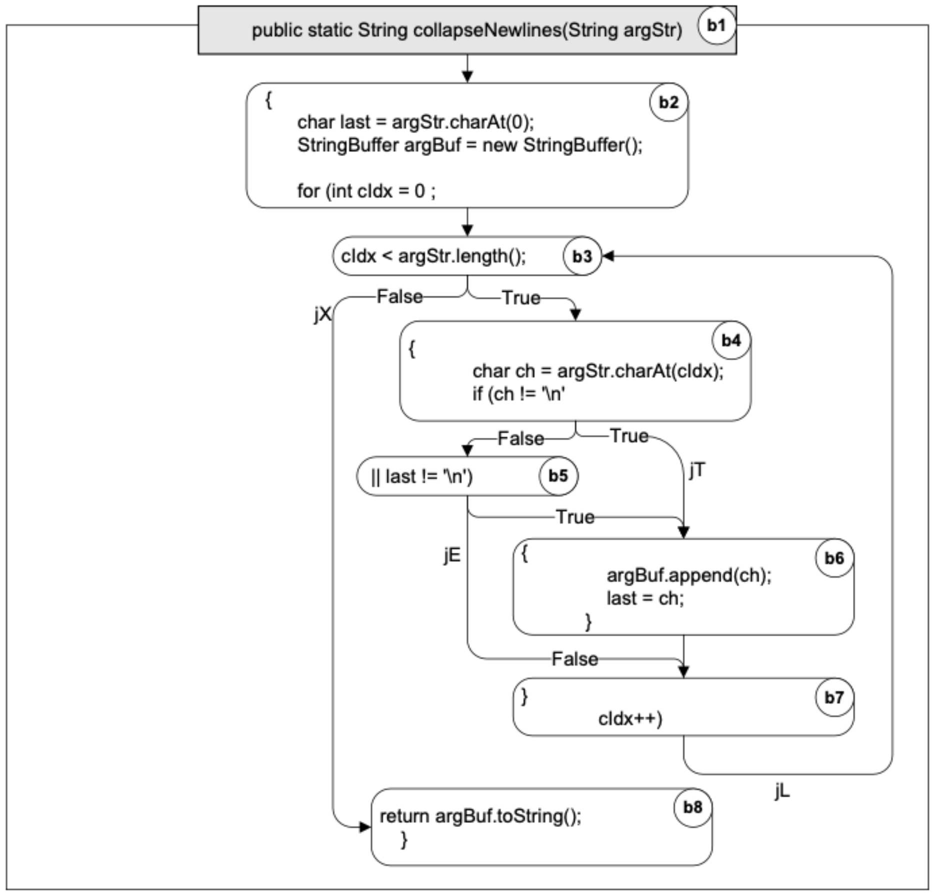


public static String collapseNewlines(String argStr)
b1
{
char last = argStr.charAt(0);
StringBuffer argBuf = new StringBuffer();
for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
jX
{
char ch = argStr.charAt(cIdx);
if (ch != '\n'
b4
False
True
jT
|| last != '\n')
b5
True
jE
{
argBuf.append(ch);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

Linear Code Sequence and Jump (LCSJ)

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

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


```
public static String collapseNewlines(String argStr)
{
  char last = argStr.charAt(0);
  StringBuffer argBuf = new StringBuffer();

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

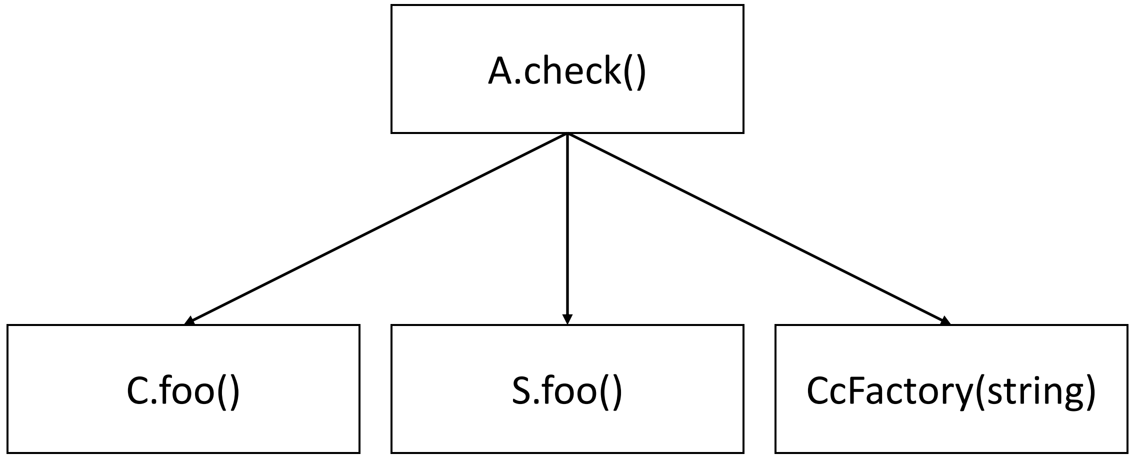
Interprocedural control flow graph

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

Overestimating the *calls* relation

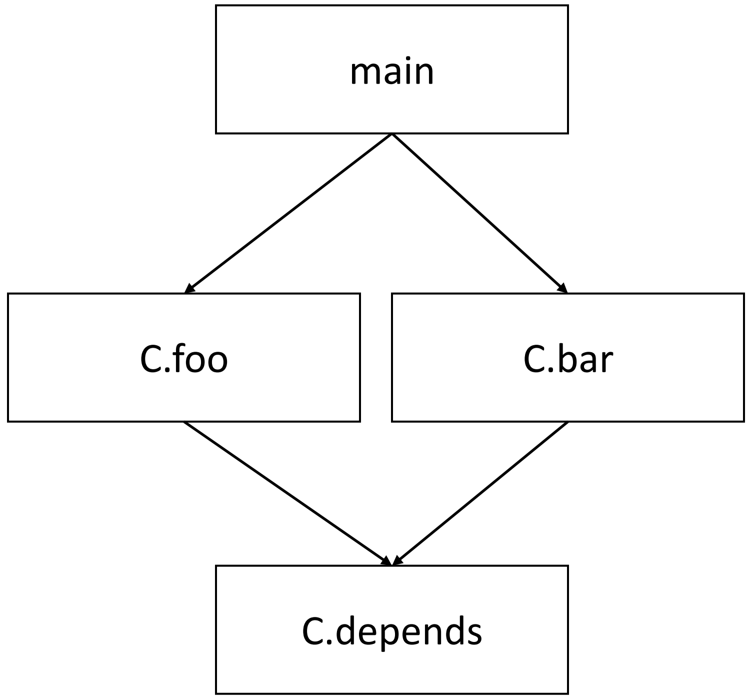
The static call graph includes calls through dynamic bindings that never occur in execution.

```
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();
   }
}
```



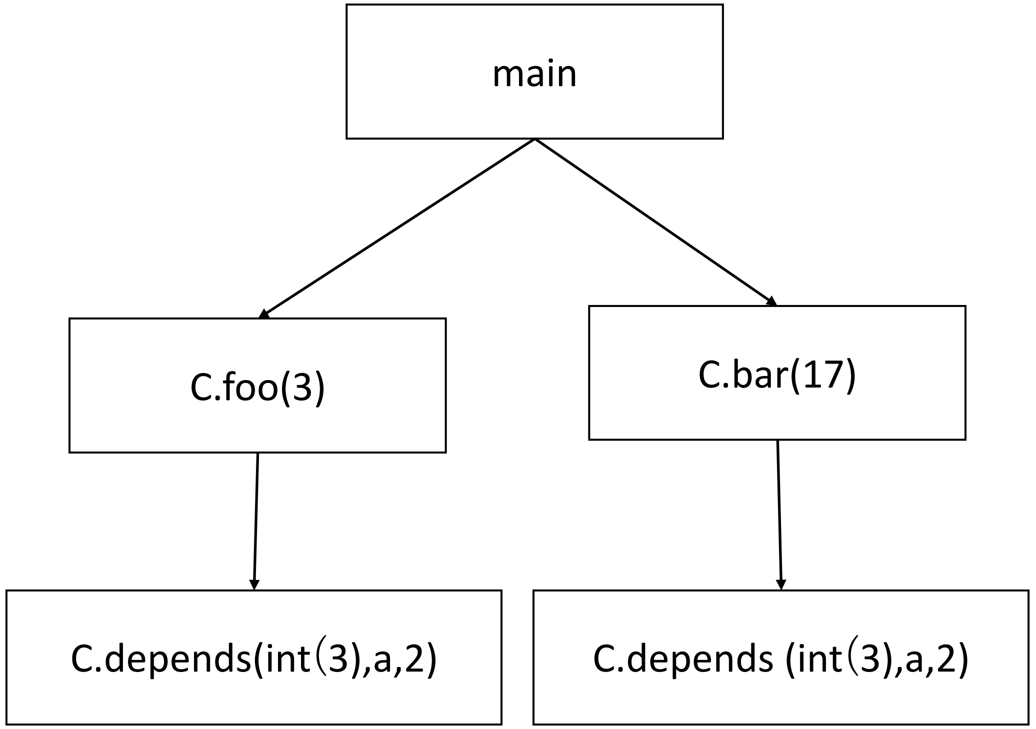
Contex Insensitive Call graphs

```
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;
   }
}
```

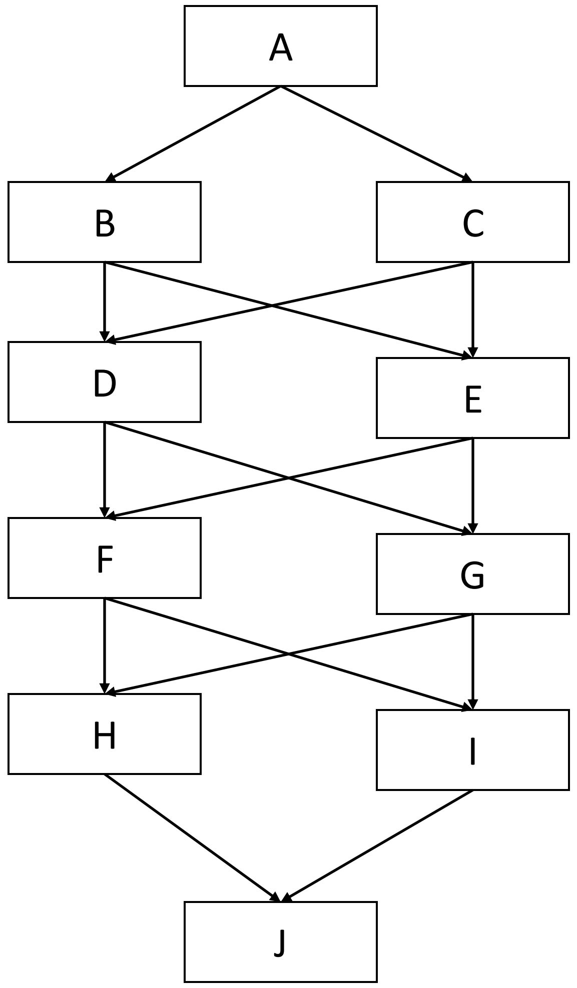

$$\begin{tikzcd}
 & \text{main} \arrow[dl] \arrow[dr] & \\
\text{C.foo} \arrow[dr] & & \text{C.bar} \arrow[dl] \\
 & \text{C.depends} &
\end{tikzcd}$$

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

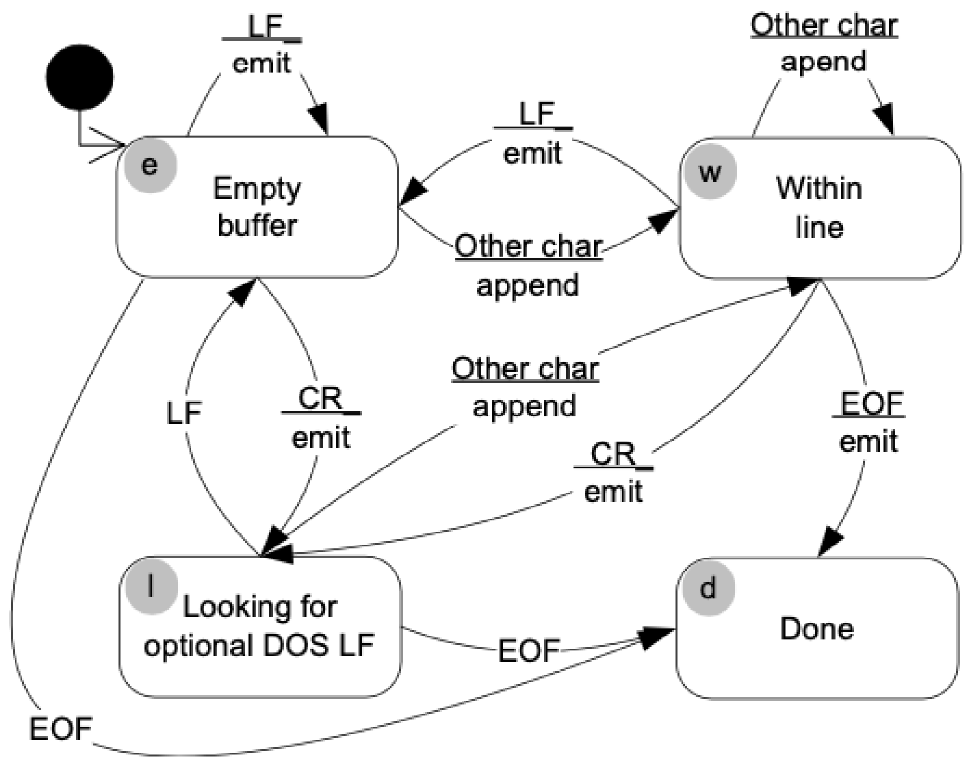
4 contexts ABD ABE ACD ACE

8 contexts …

16 calling contexts …

Finite state machines

Graph representation (Mealy machine)

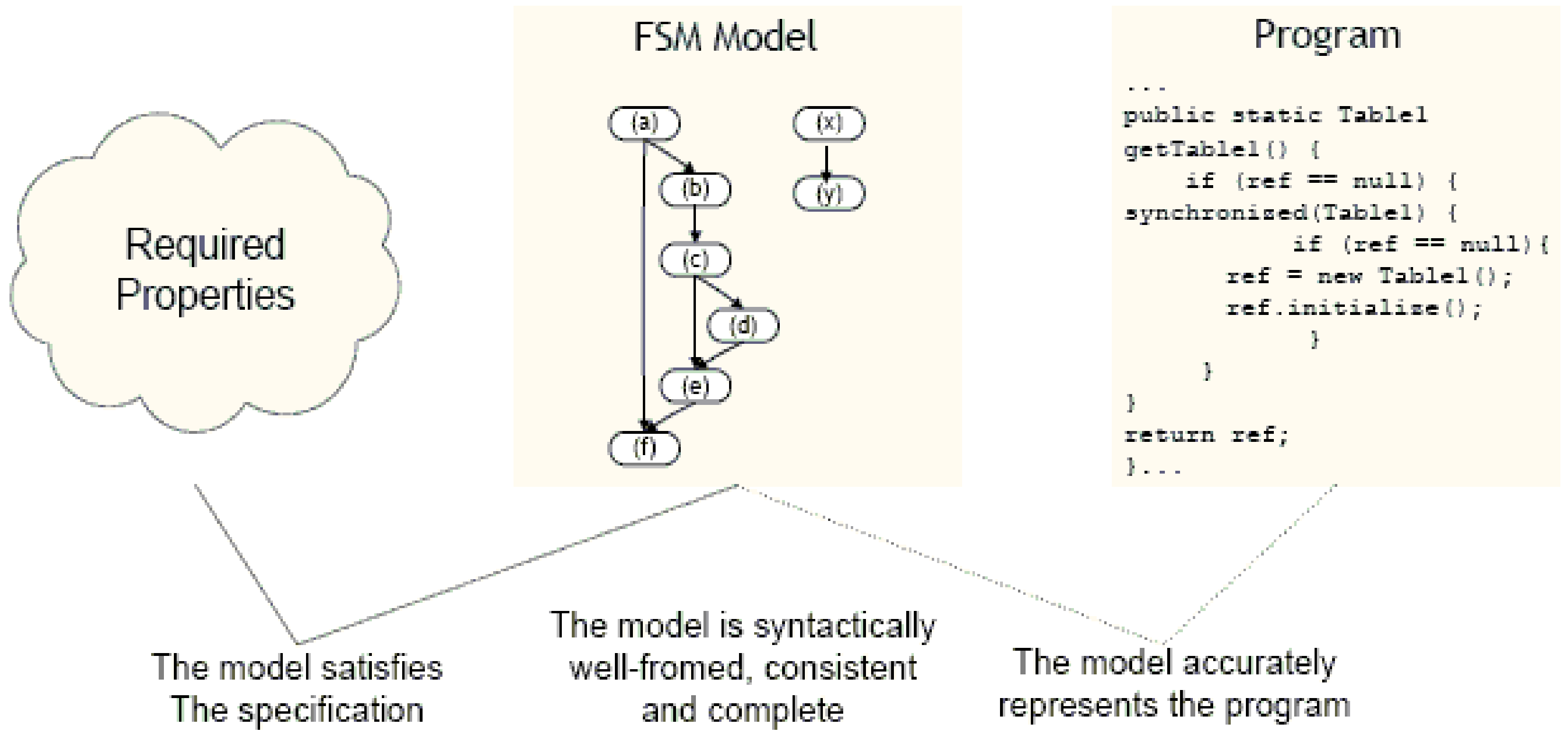


Tabular representation

	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

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

Using Models to Reason about System Properties


Required Properties
FSM Model
(a)
(b)
(c)
(d)
(e)
(f)
(x)
(y)
Program
The model satisfies The specification
The model is syntactically well-fromed, consistent and complete
The model accurately represents the program

```
...
public static Table1
getTable1() {
    if (ref == null) {
synchronized(Table1) {
                if (ref == null){
          ref = new Table1();
          ref.initialize();
                }
        }
}
return ref;
}...
```

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

Summary

- Models are simpler than the artifact they describe to be understandable and analyzable
- Must be sufficiently detailed to be useful in a particular context
- Flow Graphs are built from software
- FSM can be built before software to document intended behavior or explore designs
- FSM can be the basis for tools such as statecharts: https://statecharts.dev/