Modeling Software

CSCE 747 - Lecture 3 - 01/19/2016

Models and Software Analysis

- Before and while building products, engineers analyze models to address design questions.
- Software is no different.
- Models address two problems:
  - Analysis and testing cannot wait until a product is finished.
  - The finished product is often too complex to analyze “as-is”.

Today's Goals

- Building behavioral models.
    - Directed graphs.
    - Control-Flow graphs.
    - Call graphs.
    - Finite state machines.
- Properties of a good model.

Behavior Modeling

- **Abstraction** - simplifying a problem by identifying important aspects, focusing on those, and pretending other details don’t exist.

- The key to solving *many* computing problems.
  - Solve a simpler version, then apply to the big problem.
- A **model** is a simplified representation of an artifact, focusing on one facet of that artifact.
  - The model ignores *all* other elements of that artifact.

Models

- A **model** is a simplified representation of an artifact, focusing on one facet of that artifact.
  - The model ignores *all* other elements of that artifact.
- By abstracting away unnecessary details, extremely powerful analyses can be performed.
  - Proofs of correctness, security analysis, deadlock detection, automated verification.
- Model must preserve enough of the artifact that results hold.

Model Properties

To be useful, a model must be:

- Compact
  - Models must be simplified enough to be analyzed.
  - “How simple” depends on how it will be used.
- Predictive
  - Represent the real system well enough to distinguish between good and bad outcomes of analyses.
  - No single model usually represents all characteristics of the system well enough for all types of analysis.

Model Properties

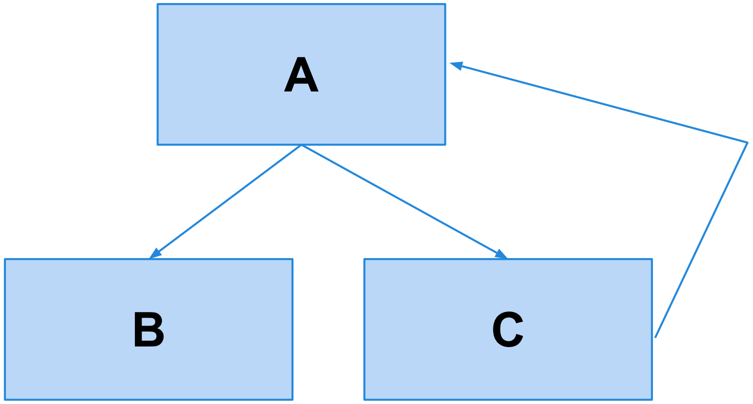
To be useful, a model must be:

- Meaningful
  - Must provide more information than success and failure. Must allow diagnoses of the causes of failure.
- Sufficiently General
  - Models must be practical for use in the domain of interest.
  - An analysis of C programs is not useful if it only works for programs without pointers.

Directed Graphs

A directed graph is composed of a set of *nodes* N and a relation E on the set (a set of ordered pairs, called *edges*).

- Nodes represent program entities.
- Edges represent relations between entities.
  - i.e., flow of execution.



Finite Abstraction

- A program execution can be viewed as a sequence of states alternating with actions.
- Software “behavior” is a sequence of state-action-state transitions.
- The set of all possible behavior sequences is often infinite.
    - Called the “state space” of the program.
    - Models of execution are abstractions of the program’s state space.

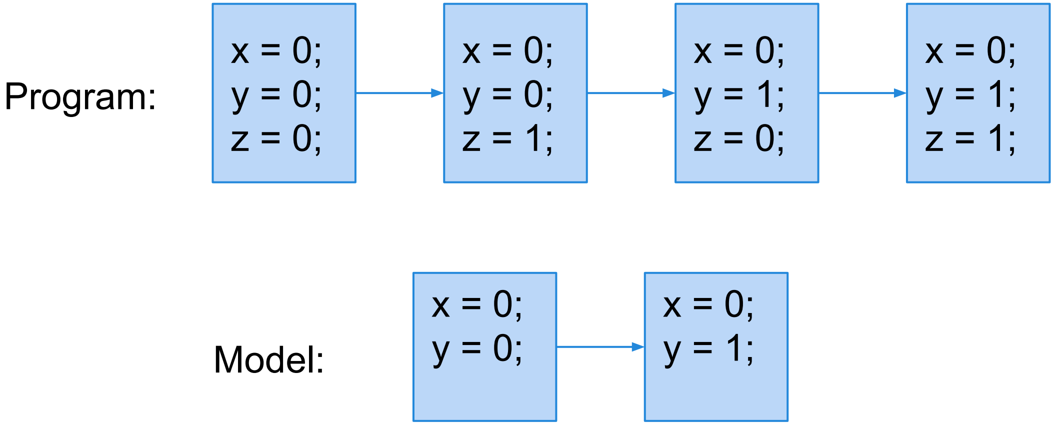
Abstraction Functions

- We can link a state in the real space of execution to a model state through an *abstraction function*.
  - The abstraction function translates the real program to a model by stripping away details.
  - The abstraction function lumps together states that only differ through details abstracted from the model. This has two effects:
    - Sequences of transitions are collapsed into fewer execution steps.
    - Nondeterminism can be introduced.

Abstraction Functions

This has two effects:

- Sequences of transitions are collapsed into fewer execution steps.



Program:

x = 0;
y = 0;
z = 0;

x = 0;
y = 0;
z = 1;

x = 0;
y = 1;
z = 0;

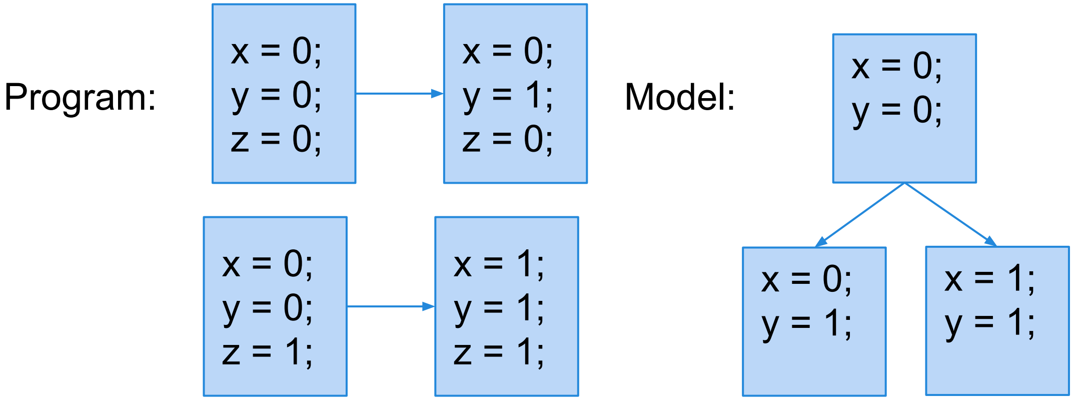
x = 0;
y = 1;
z = 1;

Model:

x = 0;
y = 0;

x = 0;
y = 1;

- Nondeterminism can be introduced.



Program:

x = 0;
y = 0;
z = 0;

x = 0;
y = 1;
z = 0;

x = 0;
y = 0;
z = 1;

x = 1;
y = 1;
z = 1;

Model:

x = 0;
y = 0;

x = 0;
y = 1;

x = 1;
y = 1;

Types of Models

- Two main “views” of program behavior:
  - **Source Code-Based**
    - Visualization of paths of execution (where states are code locations)
    - Often used to guide test generation.
  - **Behavior-Based**
    - Mapping of functionality to a series of abstract program states. Not directly linked to code statements.
- Models are also used to analyze required development effort, usability, etc.

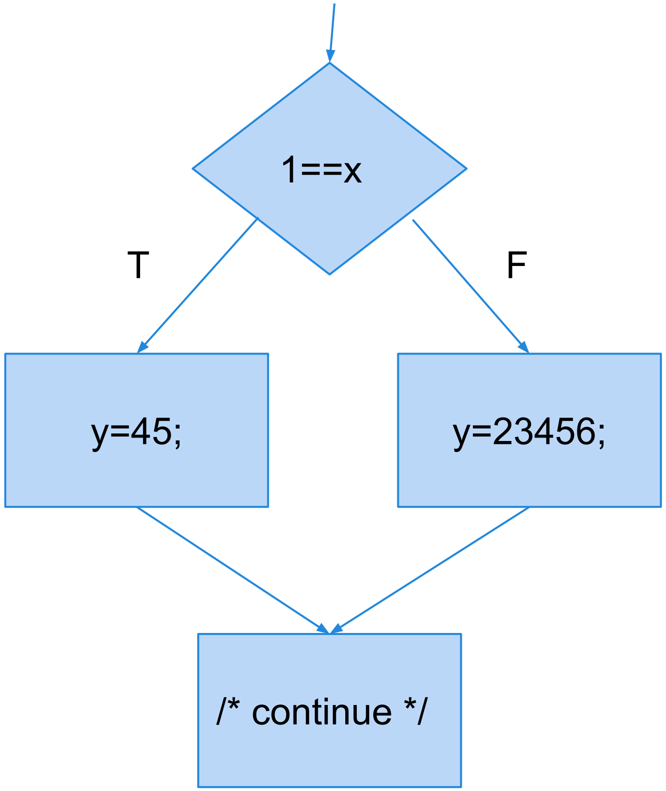
Code-Based Models

Control-Flow Graphs

- A directed graph representing the flow of control through the program.
  - Nodes represent sequential blocks of program commands.
  - Edges connect nodes in the sequence they are executed. Multiple edges indicate conditional statements (loops, if statements, switches).
    - The graph abstracts concrete execution details (variable values), so it depicts paths that are defined, but impossible to actually execute.

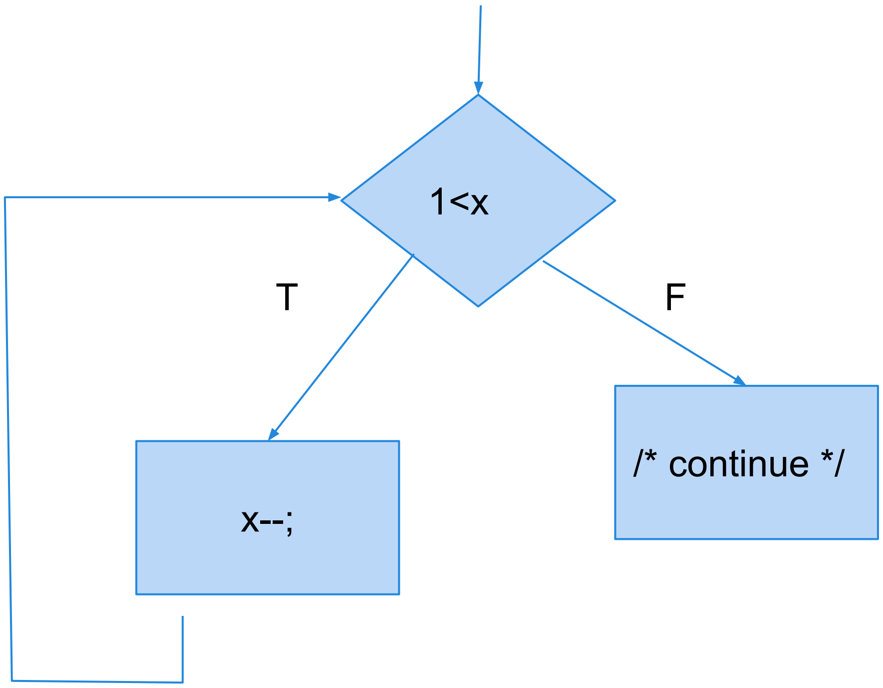
If-then-else

```
if (1==x) {
      y=45;
}
else {
      y=23456;
}
/* continue */
```



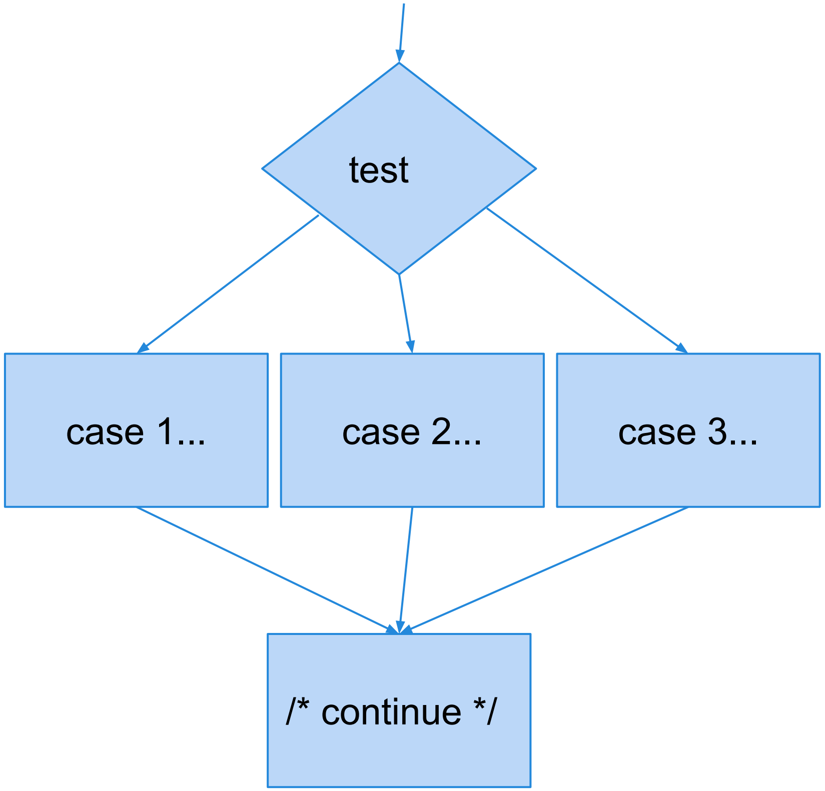
Loop

```
while (1<x) {
      x--;
}
/* continue */
```



Case

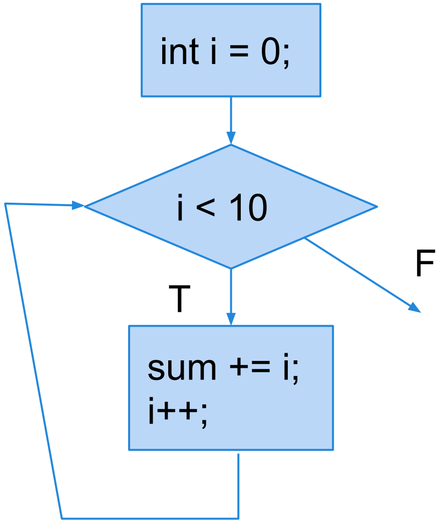
```
switch (test) {
      case 1 : ...
      case 2 : ...
      case 3 : ...
}
/* continue */
```



Basic Blocks

- Nodes represent basic blocks - a set of sequentially executed instructions with a single entry and exit point.
- Typically a set of adjacent statements, but a statement might be broken up into multiple blocks to model control flow in the statement.

```
for(int i=0; i < 10; i++){
    sum += i;
}
```

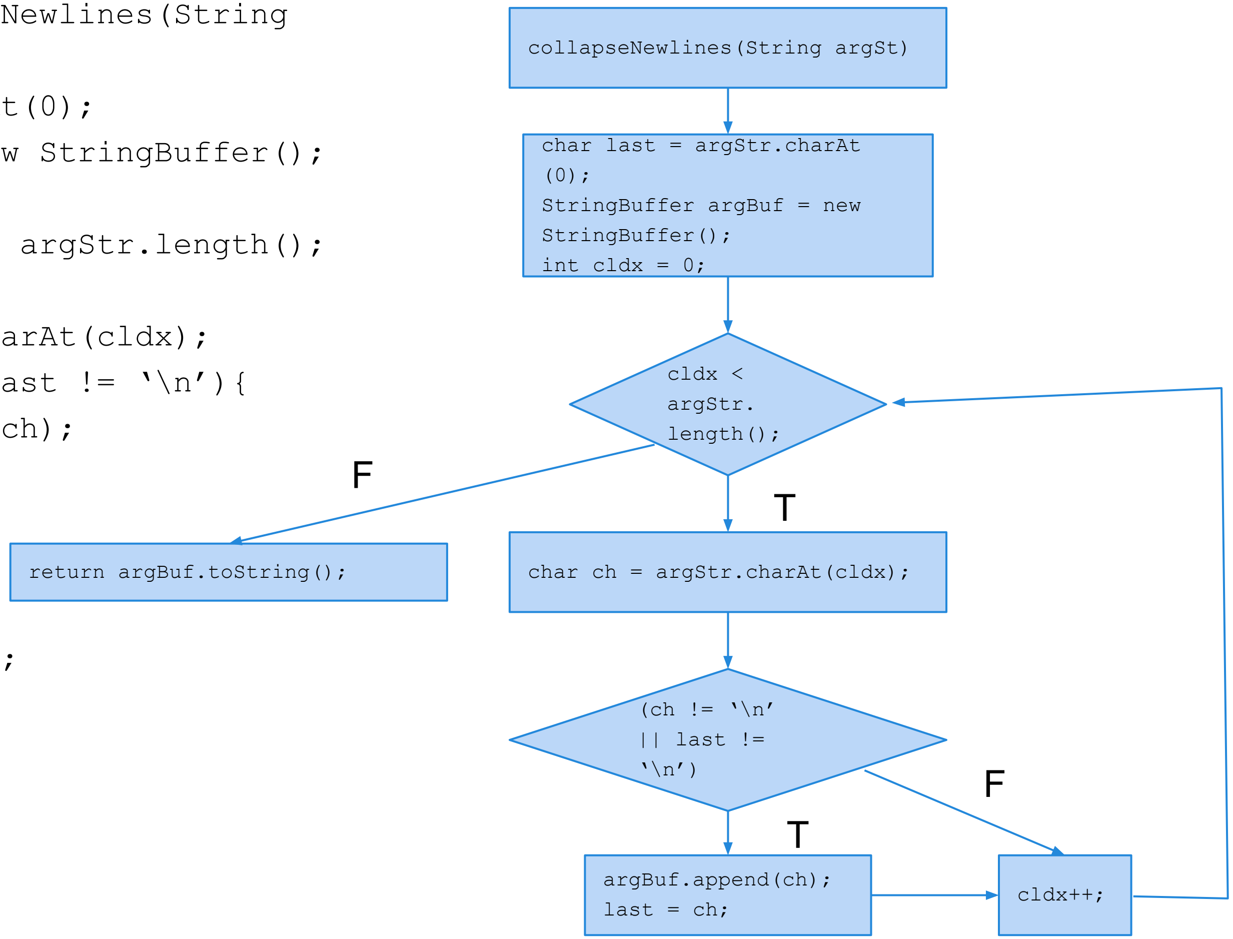


Control Flow Graph Example

```
public static String collapseNewlines(String
argSt){
     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();
}
```



```
collapseNewlines(String argSt)
```

```
char last = argStr.charAt
(0);
StringBuffer argBuf = new
StringBuffer();
int cldx = 0;
```

```
cldx <
argStr.
length();
```

```
return argBuf.toString();
```

```
char ch = argStr.charAt(cldx);
```

```
(ch != '\n'
|| last !=
'\n')
```

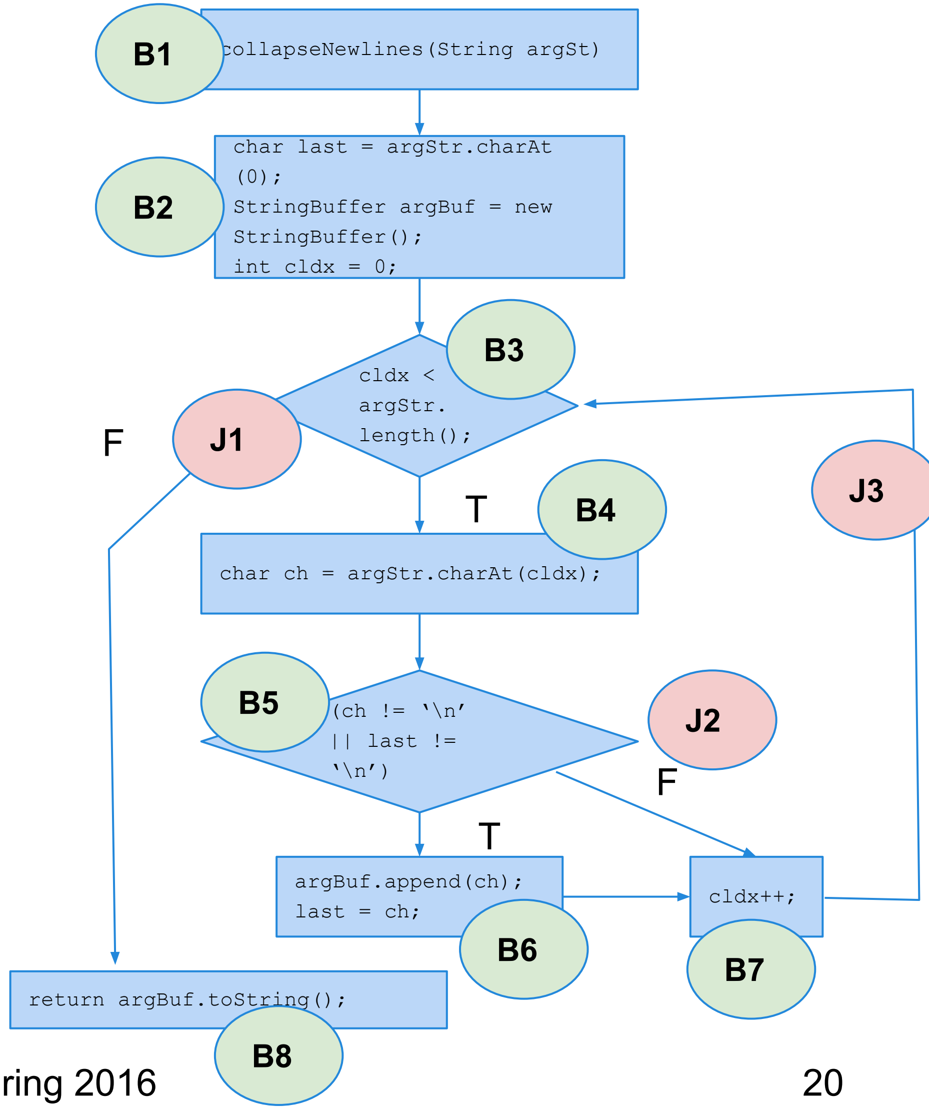
```
argBuf.append(ch);
last = ch;
```

```
cldx++;
```

Linear Code Sequences and Jumps

- Often, we want to reason about the subpaths that execution can take.
- A subpath from one branch of control to another is called a LCSAJ.
- The LCSAJs for this example:

From	To	Sequence of Basic Blocks
entry	j1	b1, b2, b3
entry	j2	b1, b2, b3, b4, b5
entry	j3	b1, b2, b3, b4, b5, b6, b7
j1	return	b8
j2	j3	b7
j3	j2	b3, b4, b5
j3	j3	b3, b4, b5, b6, b7



```
collapseNewlines(String argSt)
```

```
char last = argStr.charAt
(0);
StringBuffer argBuf = new
StringBuffer();
int cldx = 0;
```

```
cldx <
argStr.
length();
```

```
char ch = argStr.charAt(cldx);
```

```
(ch != '\n'
|| last !=
'\n')
```

```
argBuf.append(ch);
last = ch;
```

```
cldx++;
```

```
return argBuf.toString();
```

Activity 1 - Control-Flow Graph

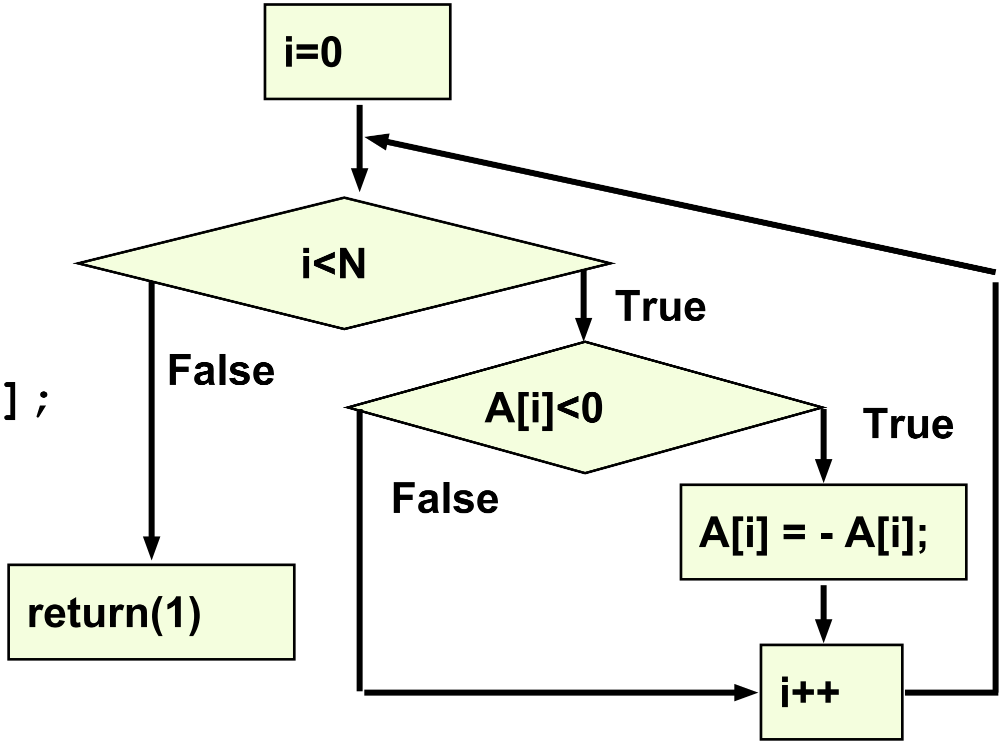
Draw a control-flow graph for the following code:

```
int abs(int A[], int N)
{
    int i=0;
   while (i< N)
   {
       if (A[i]<0)
          A[i] = - A[i];
       i++;
   }
   return(1);
}
```

Activity 1 - Solution

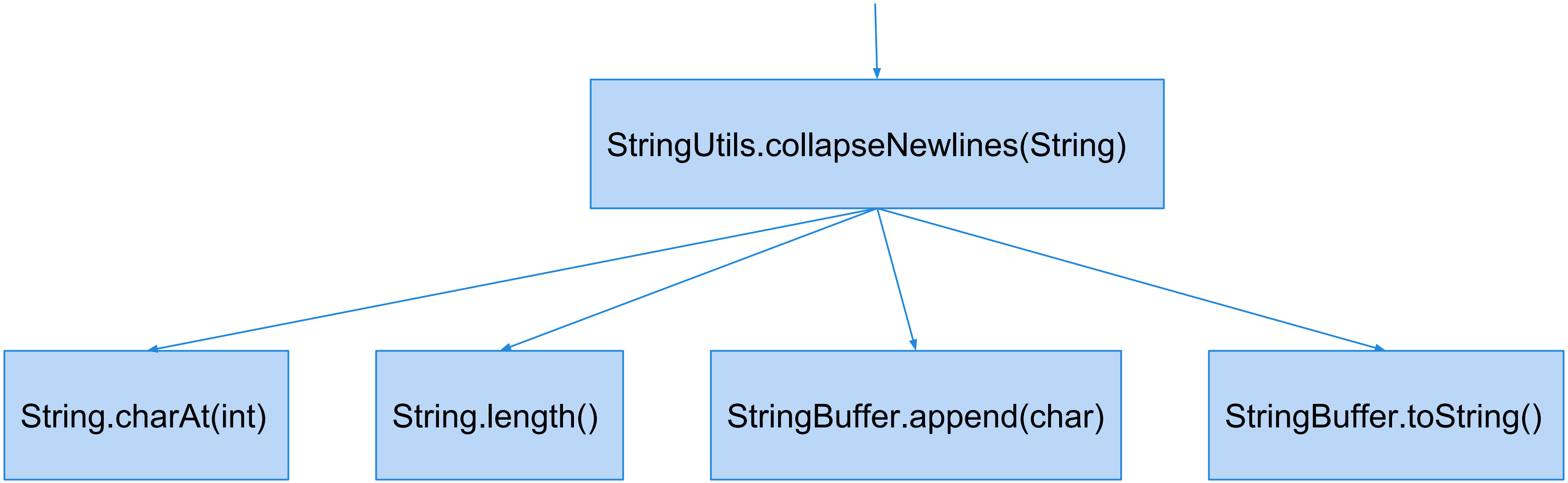
Draw a control-flow graph for the following code:

```
int abs(int A[], int N)
{
     int i=0;
    while (i< N)
    {
        if (A[i]<0)
           A[i] = - A[i];
        i++;
    }
    return(1);
}
```



Call Graphs

Directed graph representing *interprocedural* control-flow, where nodes represent procedures and edges represent "calls" relation.


StringUtils.collapseNewlines(String)
String.charAt(int)
String.length()
StringBuffer.append(char)
StringBuffer.toString()

Polymorphism and Call Graphs

- In OO languages, subclasses inherit a data type, methods, and variables from a parent
- Subclasses can override behavior of inherited methods. You cannot be sure which class is assigned to a variable at runtime.
- In the call graph, you can either model all subclasses that could be invoked, or just the declared class.
  - Latter is easier, but risks omitting execution paths.

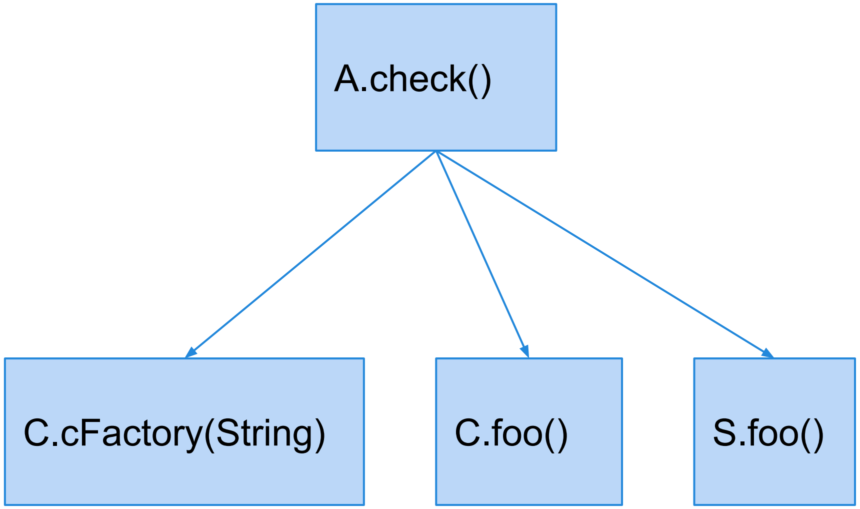
Call Graphs

```
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(“Hello.”);
      }

      public static void main(String args[]){
            (new A()).check();
      }
}

class S extends C{
      void foo(){
            System.out.println(“World.”);
      }
}
```

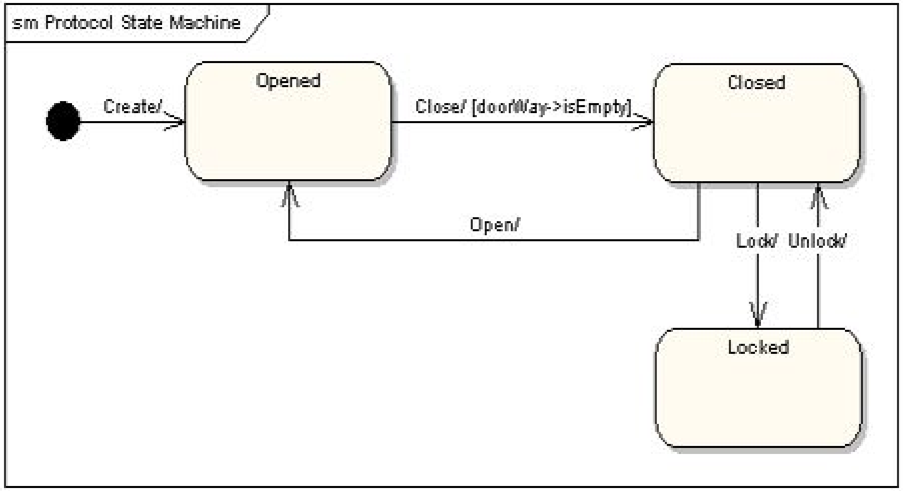


```
class A{
      void check(){
            C myC = C.cFactory(“S”);
            myC.foo();
      }
}
```

Behavioral Models

Finite State Machines

- A common method of modeling behavior of a system.
- A directed graph: nodes represent states, edges represent transitions.
- Not a substitute for a program, but a way to explore and understand a program.
  - ○ Can even build a model for each function.



Some Terminology

- **Event -** Something that happens at a point in time.
  - Operator presses a self-test button on the device.
  - The alarm goes off.
- **Condition** - Describes a property that can be true or false and has duration.
  - The fuel level is high.
  - The alarm is on.
- **State** - An abstract description of the current value of an entity's attributes.
  - The controller is in the "self-test" state after the self-test button has been pressed, and leaves it when the rest button has been pressed.
  - The tank is in the "too-low" state when the fuel level is below the set threshold for N seconds.

States, Transitions, and Guards

- **State** - An abstract description of the current value of an entity's attributes.
- States change in response to events.
    - A state change is called a **transition**.
- When multiple responses to an event (transitions triggered by that event) are possible, the choice is guided by the current conditions.
    - These conditions are also called the **guards** on a transition.

State Transitions

Transitions are labeled in the form:

```
event [guard] / activity
```

- `event`: The event that triggered the transition.
- `guard`: Conditions that must be true to choose this transition.
- `activity`: Behavior exhibited by the object when this transition is taken.
- All three are optional.
    - Missing Activity: No output from this transition.
    - Missing Guard: Always take this transition if the event occurs.
    - Missing Event: Take this transition immediately.

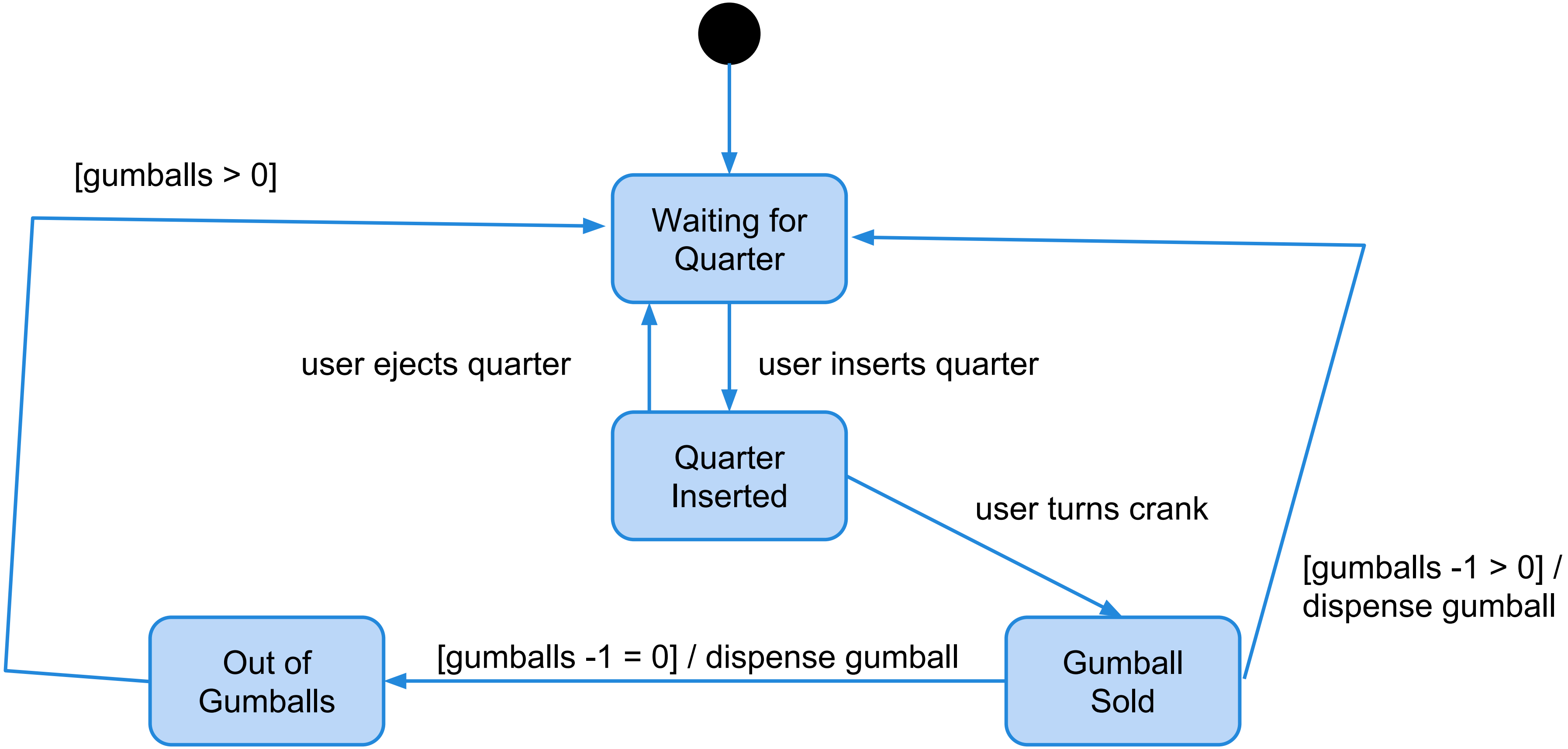
State Transition Examples

Transitions are labeled in the form:

```
event [guard] / activity
```

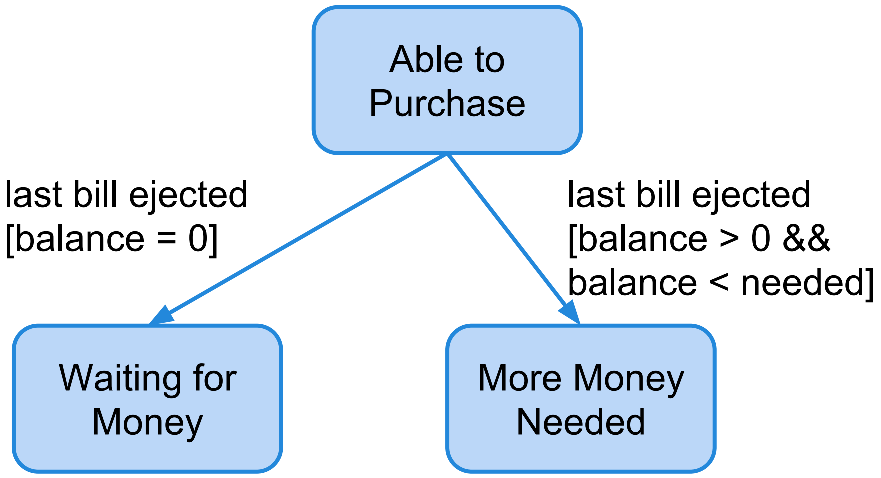
- The controller is in the “self-test” state after the self-test button has been pressed, and leaves it when the rest button has been pressed.
  - Pressing self-test button is an **`event`.**
- The tank is in the “too-low” state when the fuel level is below the set threshold for N seconds.
  - Fuel level below threshold for N seconds is a **`guard`**.

Example: Gumball Machine



More on Transitions

Guards must be mutually exclusive

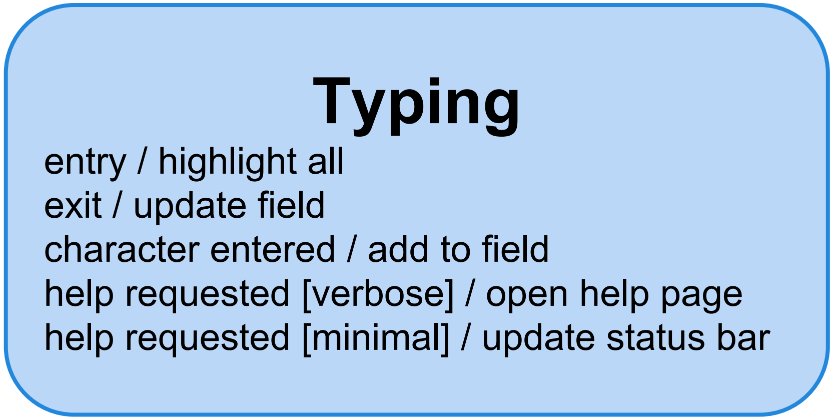

Able to Purchase
last bill ejected [balance = 0]
last bill ejected [balance > 0 && balance < needed]
Waiting for Money
More Money Needed

If an event occurs and no transition is valid, then the event is ignored.

last bill ejected [balance > 0 && balance >= needed]

Internal Activities

States can react to events and conditions without transitioning using internal activities.



Typing

entry / highlight all
exit / update field
character entered / add to field
help requested [verbose] / open help page
help requested [minimal] / update status bar

Special events: **entry** and **exit**.

Other activities occur until a transition occurs.

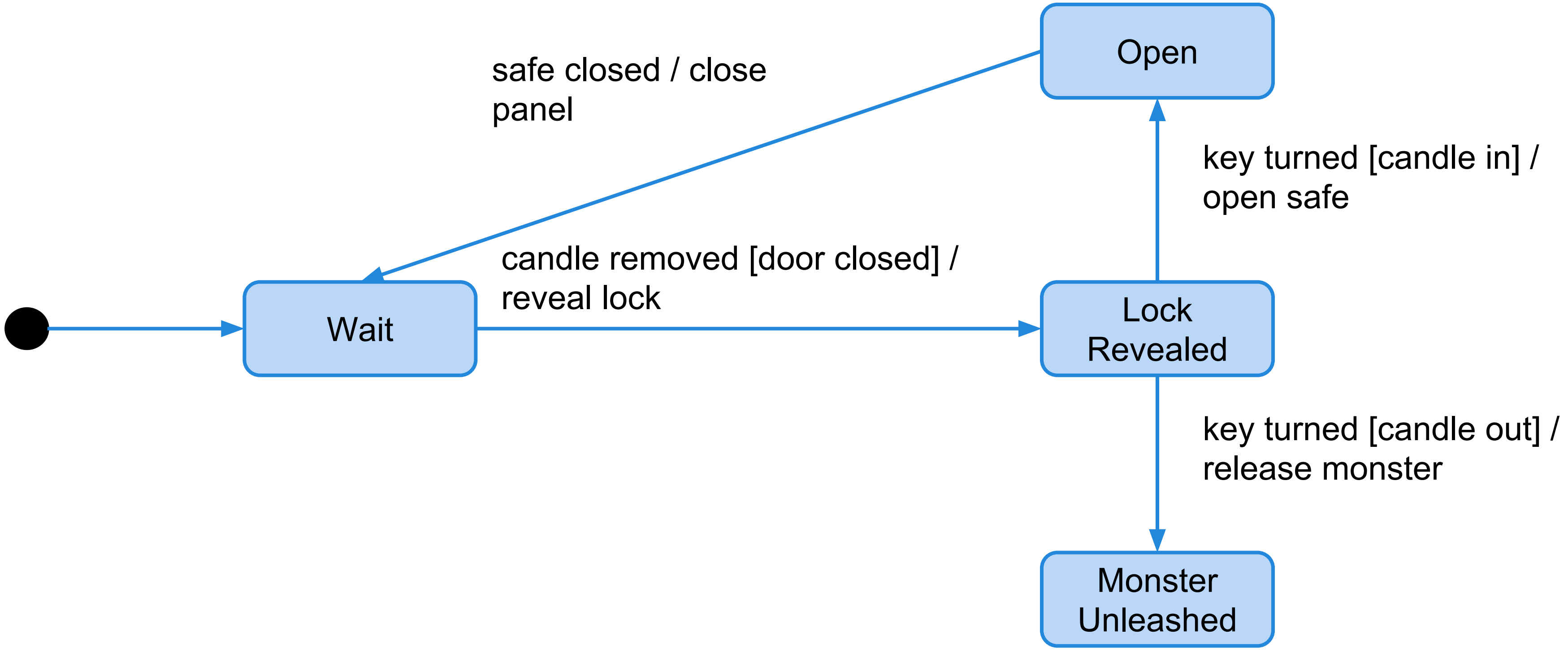
Similar to a **self-transition**, but entry and exit will not be re-triggered without using an actual self-transition.

Activity - Secret Panel Controller

You must design a state machine for the controller of a secret panel in Dracula’s castle.

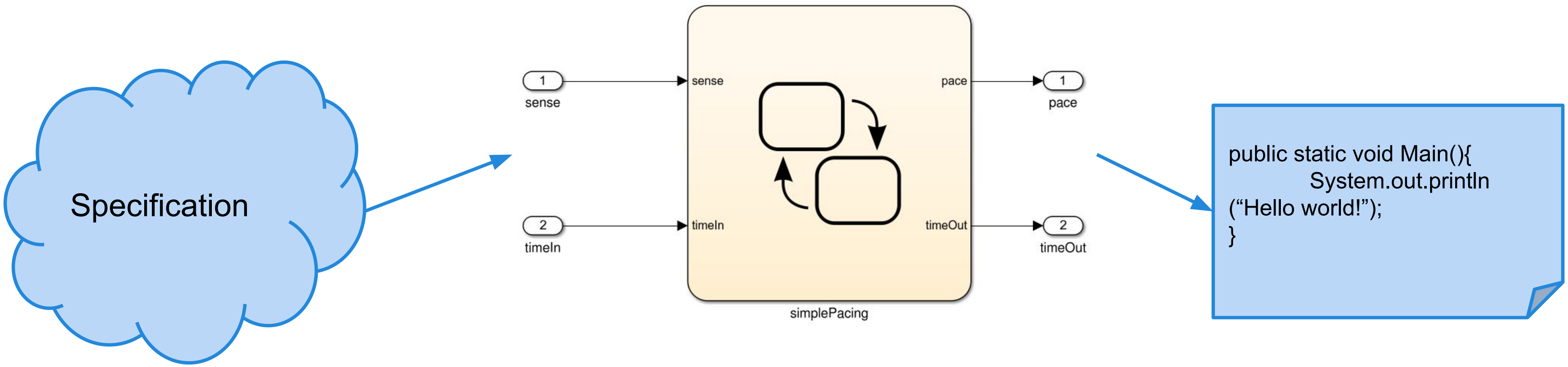
Dracula wants to keep his valuables in a safe that’s hard to find. So, to reveal the lock to the safe, Dracula must remove a strategic candle from its holder. This will reveal the lock only if the door is closed. Once Dracula can see the lock, he can insert his key to open the safe. For extra safety, the safe can only be opened if he replaces the candle first. If someone attempts to open the safe without replacing the candle, a monster is unleashed.

Activity Solution


safe closed / close panel
Open
key turned [candle in] / open safe
candle removed [door closed] / reveal lock
Wait
Lock Revealed
key turned [candle out] / release monster
Monster Unleashed

What Can We Do With This Model?

Now that we have a model, we can reason about our requirements and specifications.



```
public static void Main(){
            System.out.println
("Hello world!");
}
```

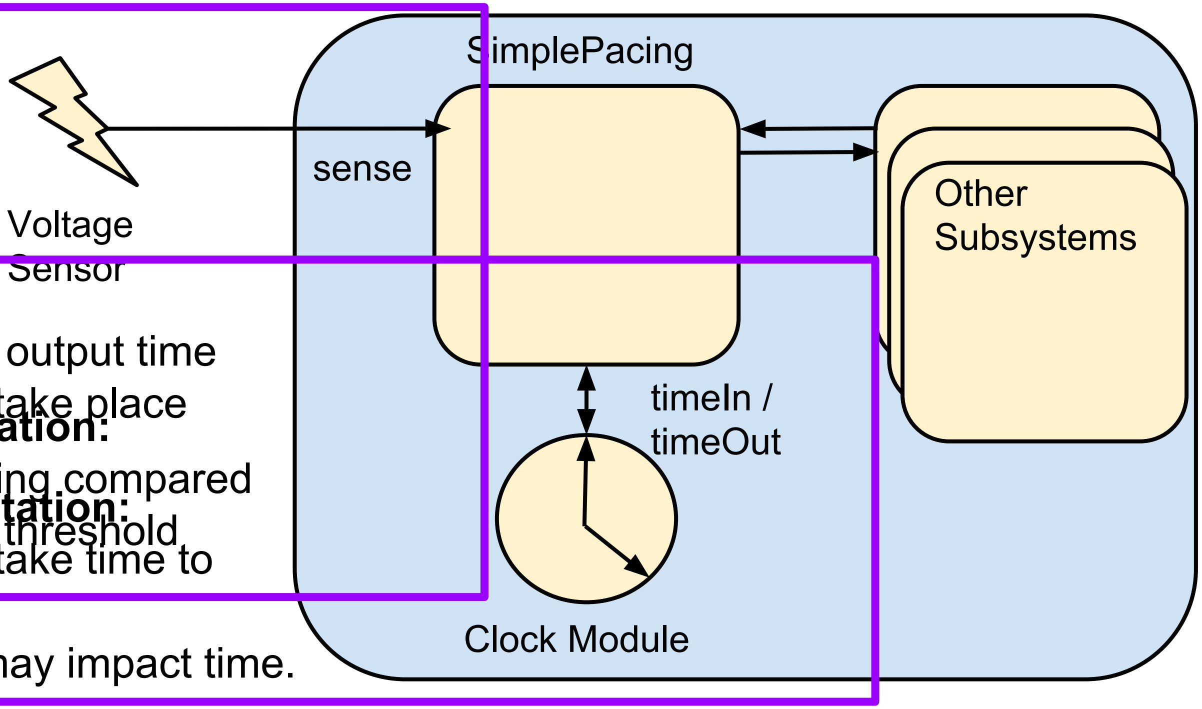
If the model satisfies the specification...

And If the model is well-formed, consistent, and complete.

And If the model accurately represents the program.

Challenge - Does the Model Match the Program?

Models require abstraction. Useful for requirements analysis, but may not reflect operating conditions.



In the model:

- Binary input

In the implementation:

- Voltage reading compared to calculated threshold

In the model:

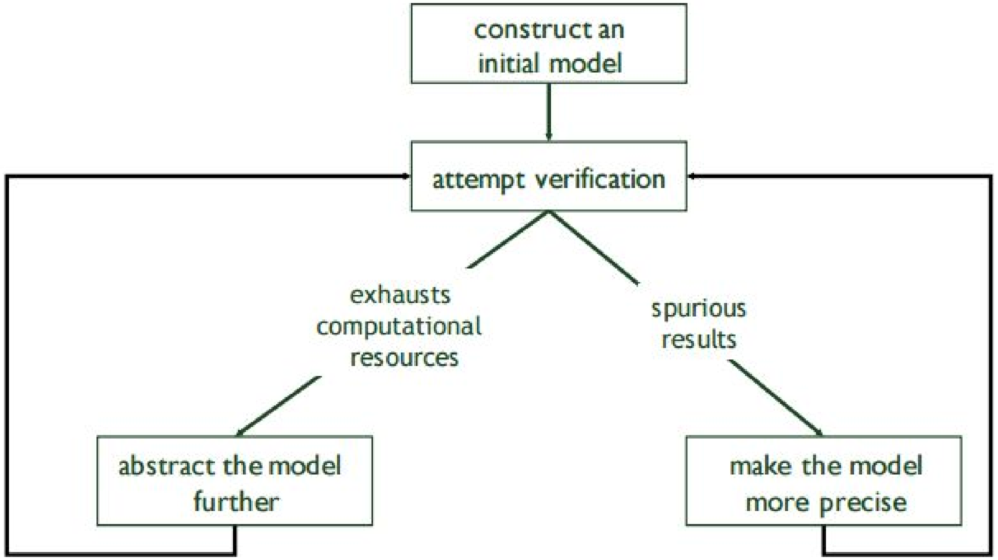
- input time = output time
- Operations take place instantly.

In the implementation:

- Operations take time to compute.
- Clock drift may impact time.

Model Refinement

- Models have to balance precision with efficiency.
- Abstractions that are too simple may introduce spurious failure paths that may not be in the real system.
- Models that are too complex may render model checking infeasible due to resource exhaustion.


construct an initial model
attempt verification
exhausts computational resources
spurious results
abstract the model further
make the model more precise

We Have Learned

- Often, the source code of the software is too complex to analyze in detail.
- Instead, we must create abstract models of the facets of a program we want to examine.
- Models can be based on source code and execution paths or on specifications of functional behavior.
- Models can be used by sophisticated verification techniques to prove that the program obeys the specifications.

Next Time

- Functional Testing
  - Building tests using the requirement specification.
  - Reading: Chapter 10

- Homework:
  - Team Selections due Thursday (11:59 PM)
    - e-mail me with your team roster (or to get placed)
  - Reading assignment:
    - James Whittaker. *The 10-Minute Test Plan*.
    - Due January 26 (11:59 PM)

Reading Assignment

- James Whittaker. *The 10-Minute Test Plan*.
- Individual assignment.
- Read the paper and turn in a one-page write-up:
  - Summary of the paper.
  - Your opinion on the work.
    - Is it applicable to real-world software?
    - Is it a useful approach?
    - Where does it fall short?
  - Your thoughts on how this could be improved and extended.