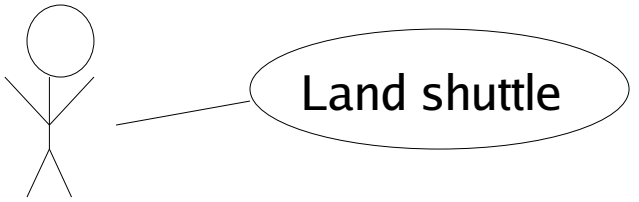
Identifying State Transitions and their Functions in Source Code

Neil Walkinshaw, Kirill Bogdanov and Mike Holcombe

Background

- Software modelled as a Finite State Machine (FSM) can be rigorously tested
  - Often incomplete or ambiguous

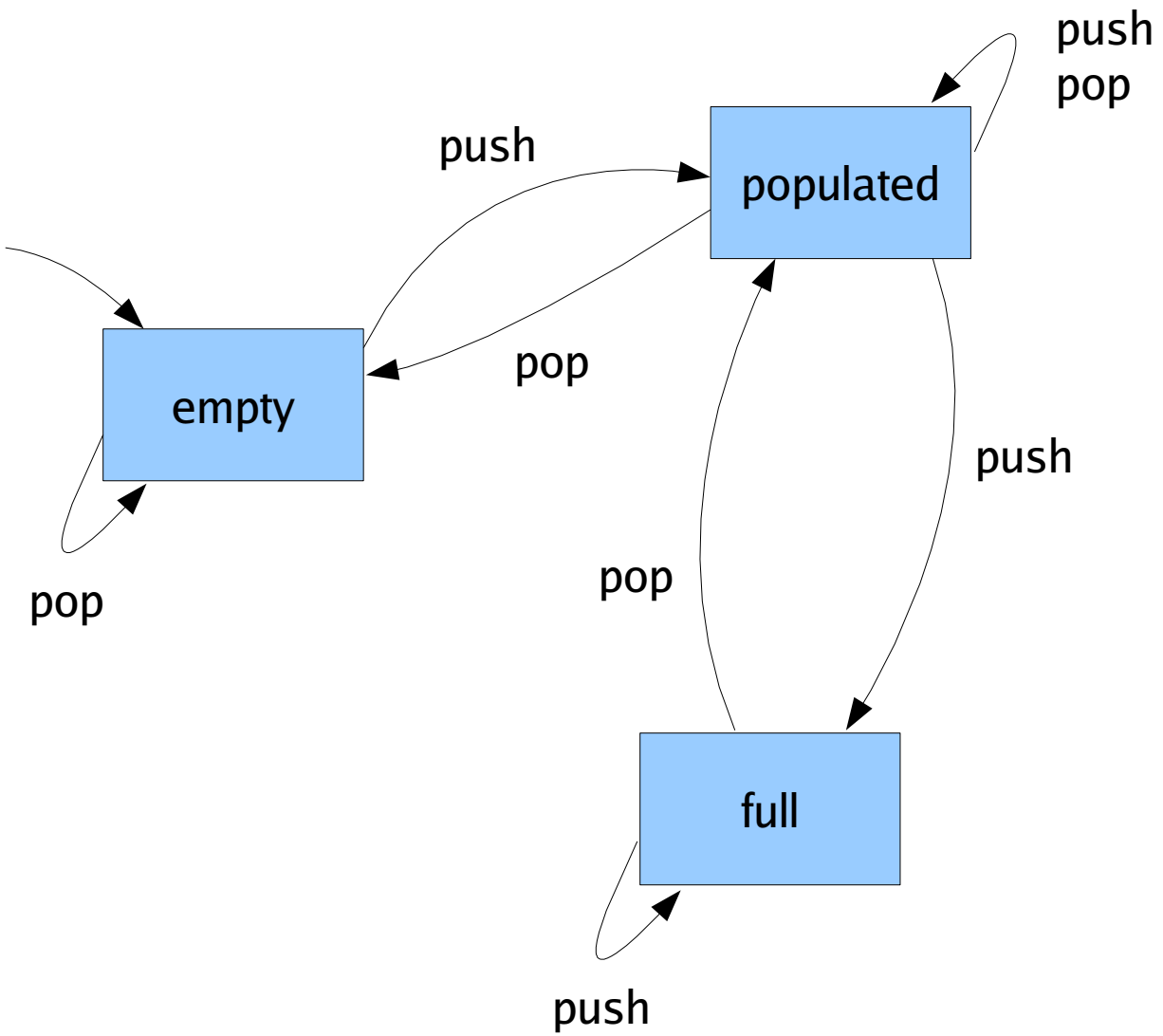


  - As software evolves, specifications are rarely kept up to date
- Need to reverse-engineer FSMs
  - Several techniques exist
  - Most of them work by observing program executions
  - Produce low level state machines, transitions simply labelled with trigger
  - **Fail to accurately describe state transition behaviour**
  - **Dynamic analysis is unsound – unsuitable as basis for testing**

Reverse Engineering State Transition Functions

- State transition functions
  - Program behaviour that leads from state A to B
  - Key to X-Machines and Abstract State Machines
- Reverse engineering state transition functions
  - Can map the start and end of state transitions to the syntax of the source code
  - Refer to these as transition points
  - A state transition function is executed between a pair of transition points

Transition Functions

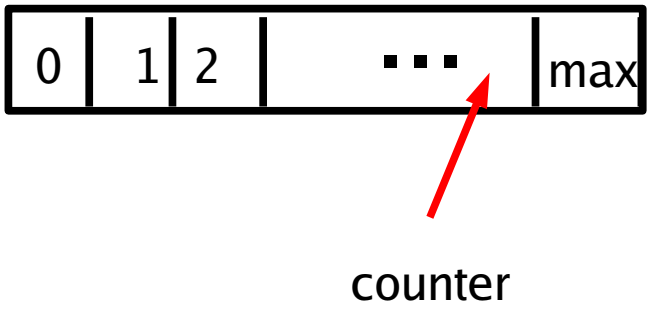


Transition Functions

```
public Stack init(int max){
        this.max = max;
        counter=-1;
        stack = new Object[max];
}
```

```
public void push(Object o){
        if(counter < 0){
                counter = 0;
        }
        if(counter < max){
                stack[counter]=o;
                counter++;
        }
}
```

```
public Object pop(){
        Object ret = null;
        counter--;
        if(counter>=0){
                ret = stack[counter];
        }
        return ret;
}
```

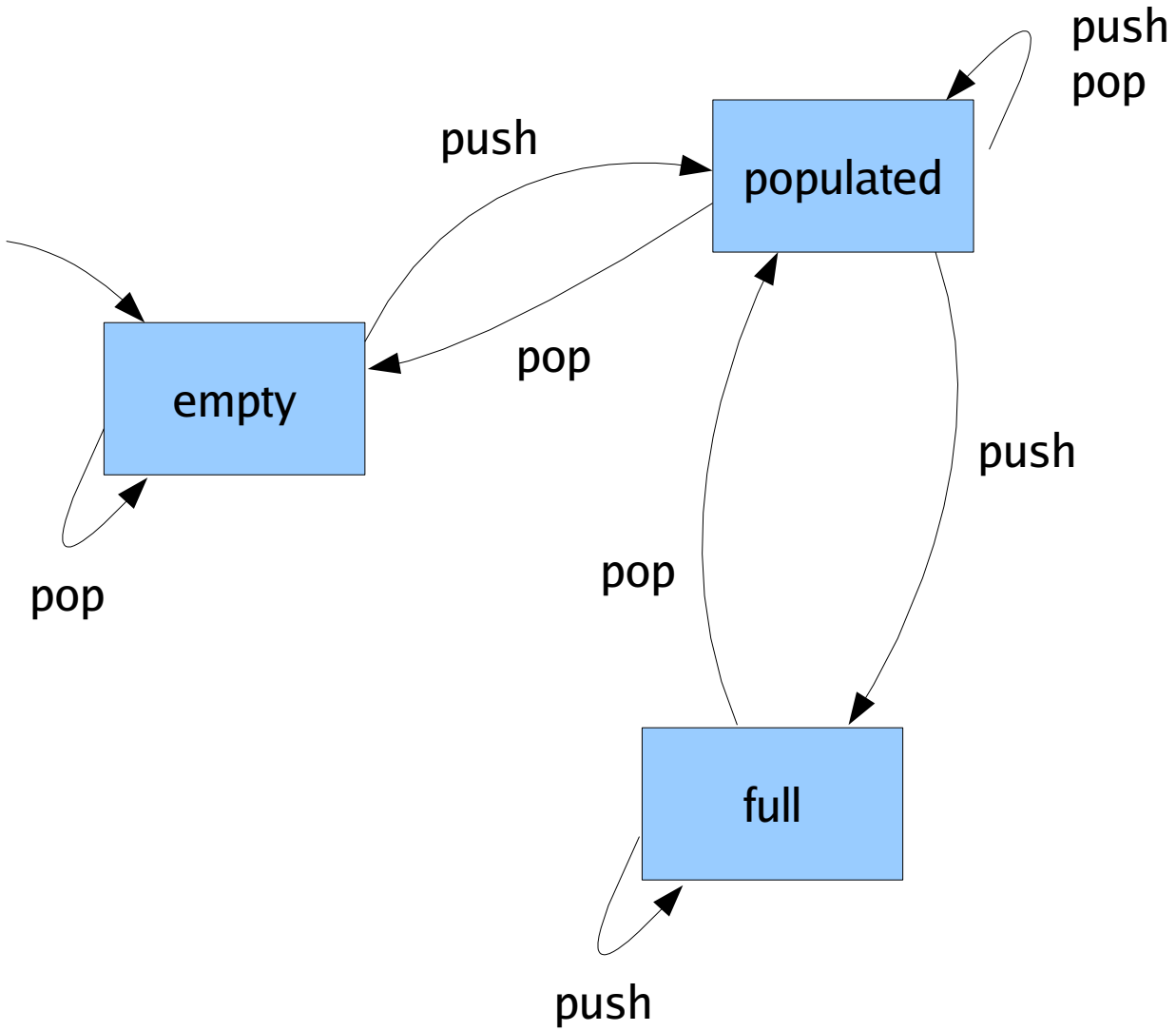


Transition Functions

```
public Stack init(int max){
        this.max = max;
        counter=-1;
        stack = new Object[max];
}
```

```
public void push(Object o){
        if(counter < 0){
                counter = 0;
        }
        if(counter < max){
                stack[counter]=o;
                counter++;
        }
}
```

```
public Object pop(){
        Object ret = null;
        counter--;
        if(counter>=0){
                ret = stack[counter];
        }
        return ret;
}
```

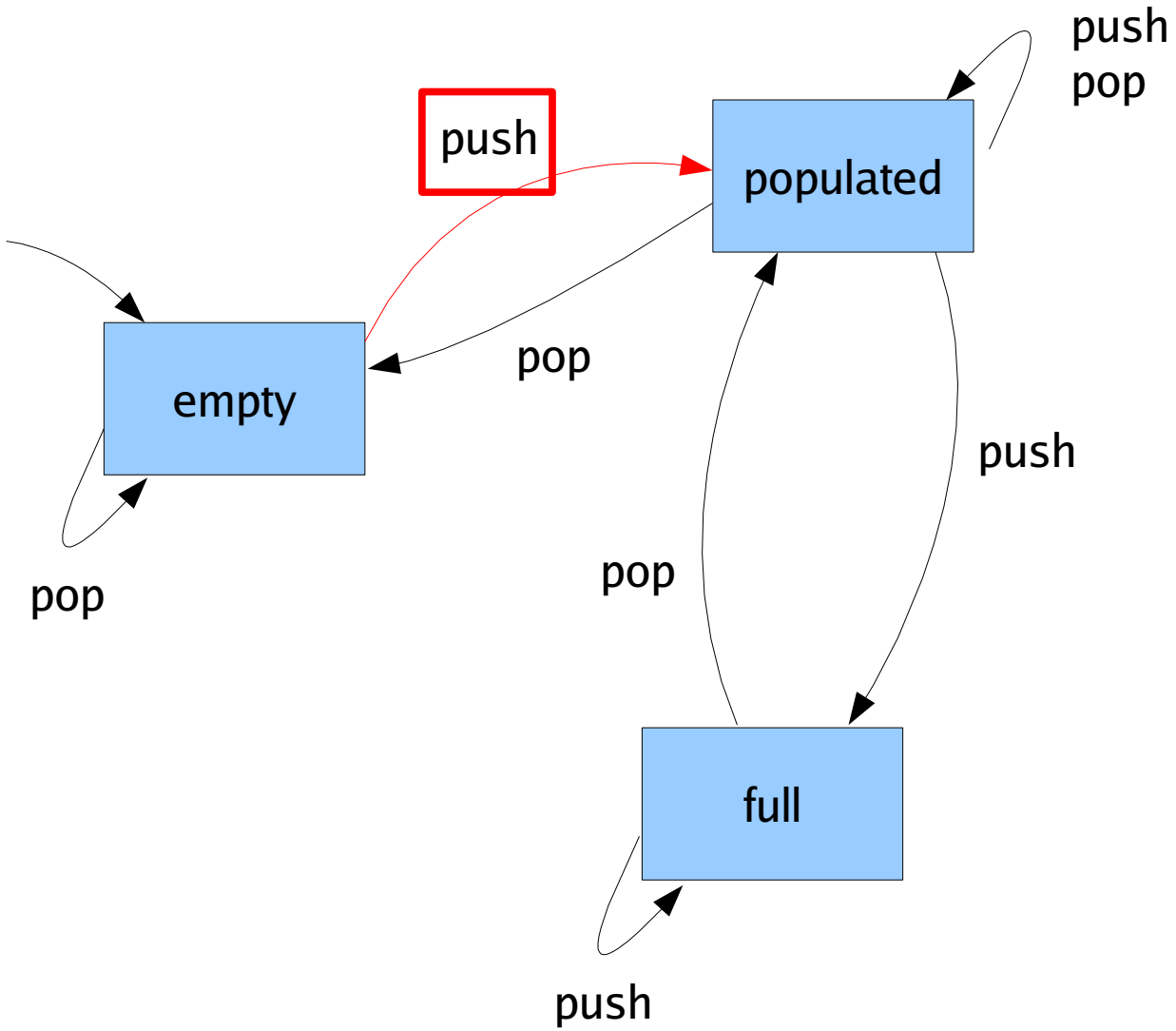


Transition Functions

```
public Stack init(int max){
        this.max = max;
        counter=-1;
        stack = new Object[max];
}
```

```
public void push(Object o){
        if(counter < 0){
                counter = 0;
        }
        if(counter < max){
                stack[counter]=o;
                counter++;
        }
}
```

```
public Object pop(){
        Object ret = null;
        counter--;
        if(counter>=0){
                ret = stack[counter];
        }
        return ret;
}
```

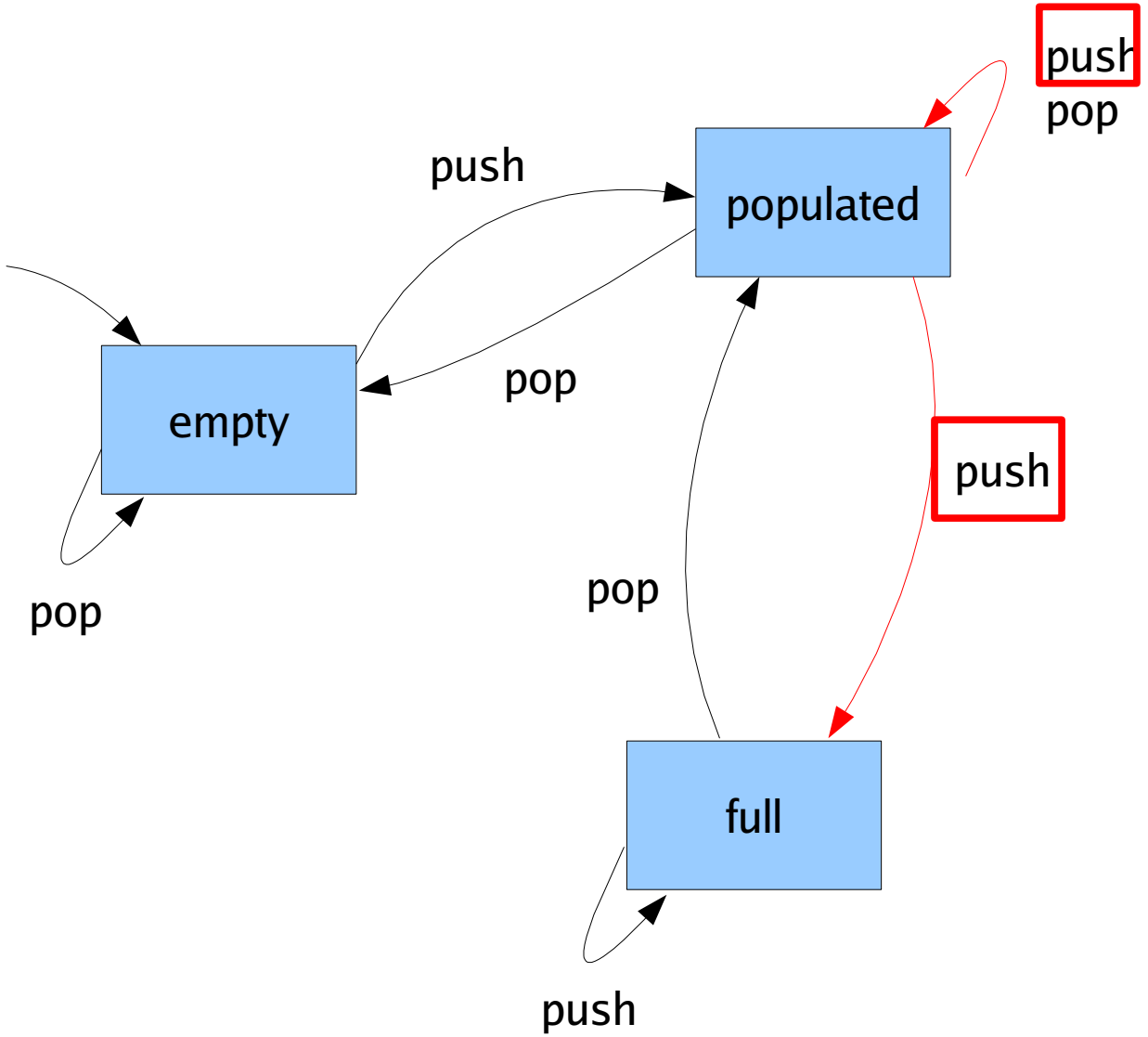


Transition Functions

```
public Stack init(int max){
        this.max = max;
        counter=-1;
        stack = new Object[max];
}
```

```
public void push(Object o){
        if(counter < 0){
                counter = 0;
        }
        if(counter < max){
                stack[counter]=o;
                counter++;
        }
}
```

```
public Object pop(){
        Object ret = null;
        counter--;
        if(counter>=0){
                ret = stack[counter];
        }
        return ret;
}
```

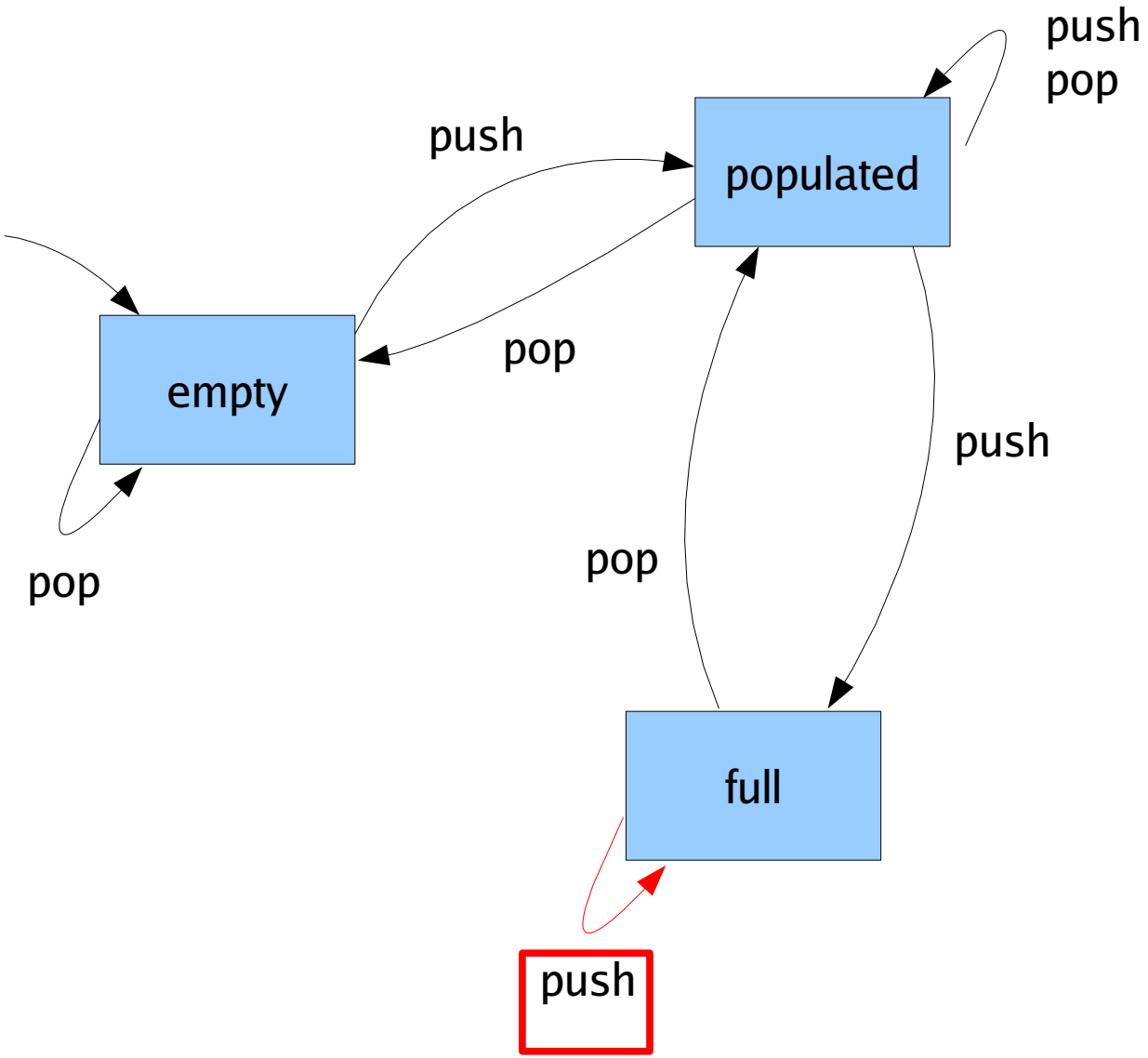


Transition Functions

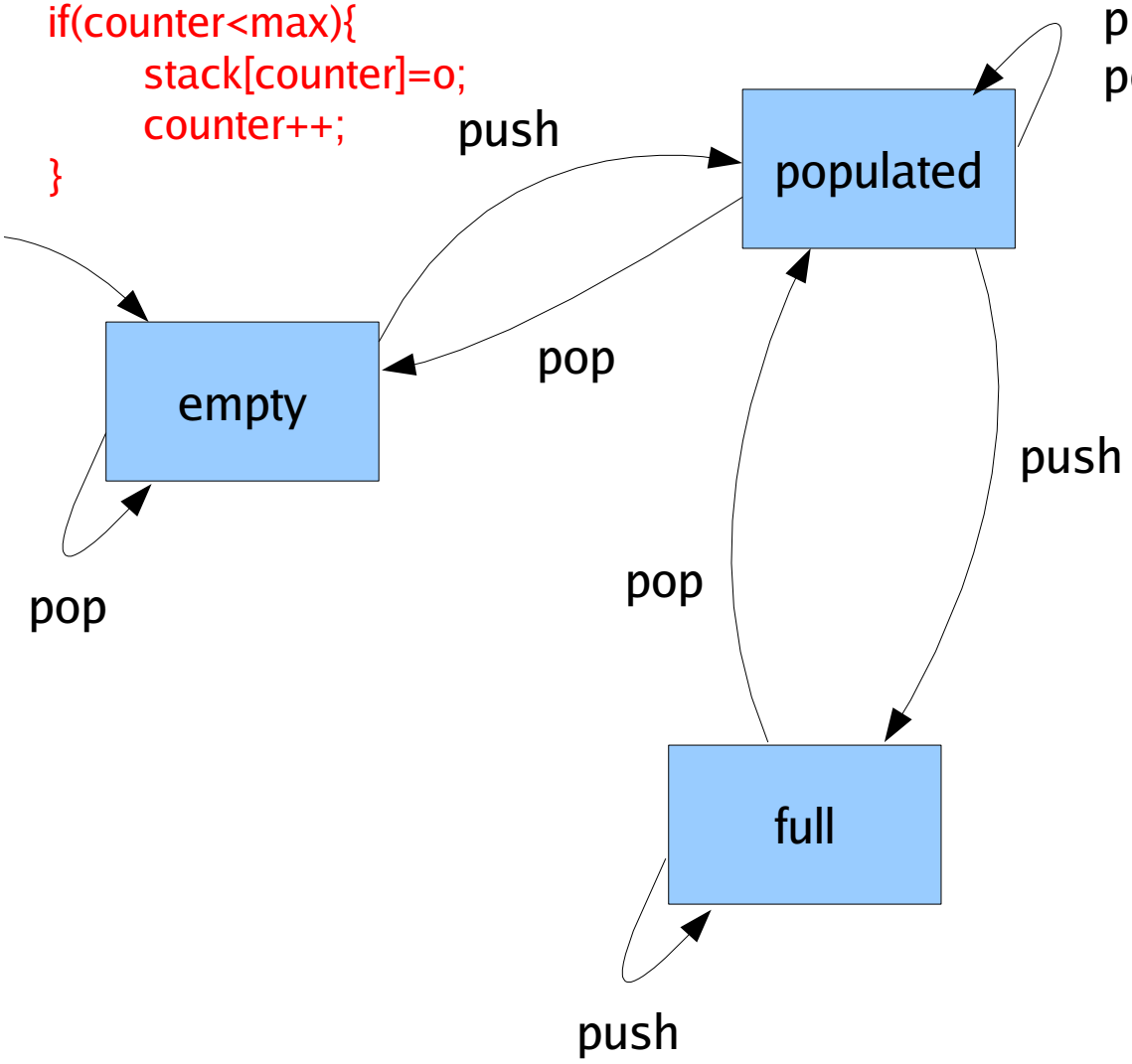
```
public Stack init(int max){
        this.max = max;
        counter=-1;
        stack = new Object[max];
}
```

```
public void push(Object o){
        if(counter < 0){
                counter = 0;
        }
        if(counter < max){
                stack[counter]=o;
                counter++;
        }
}
```

```
public Object pop(){
        Object ret = null;
        counter--;
        if(counter>=0){
                ret = stack[counter];
        }
        return ret;
}
```



```
public void push(Object o){
    if(counter < 0){
        counter = 0;
    }
    if(counter<max){
        stack[counter]=o;
        counter++;
    }
}
```




```
public void push(Object o){
    if(counter < 0){ }
    if(counter < max){
        stack[counter]=o;
        counter++;
    }
}
```

```
public void push(Object o){
    if(counter < 0){ }
    if(counter < max){
        stack[counter]=o;
        counter++;
    }
}
```

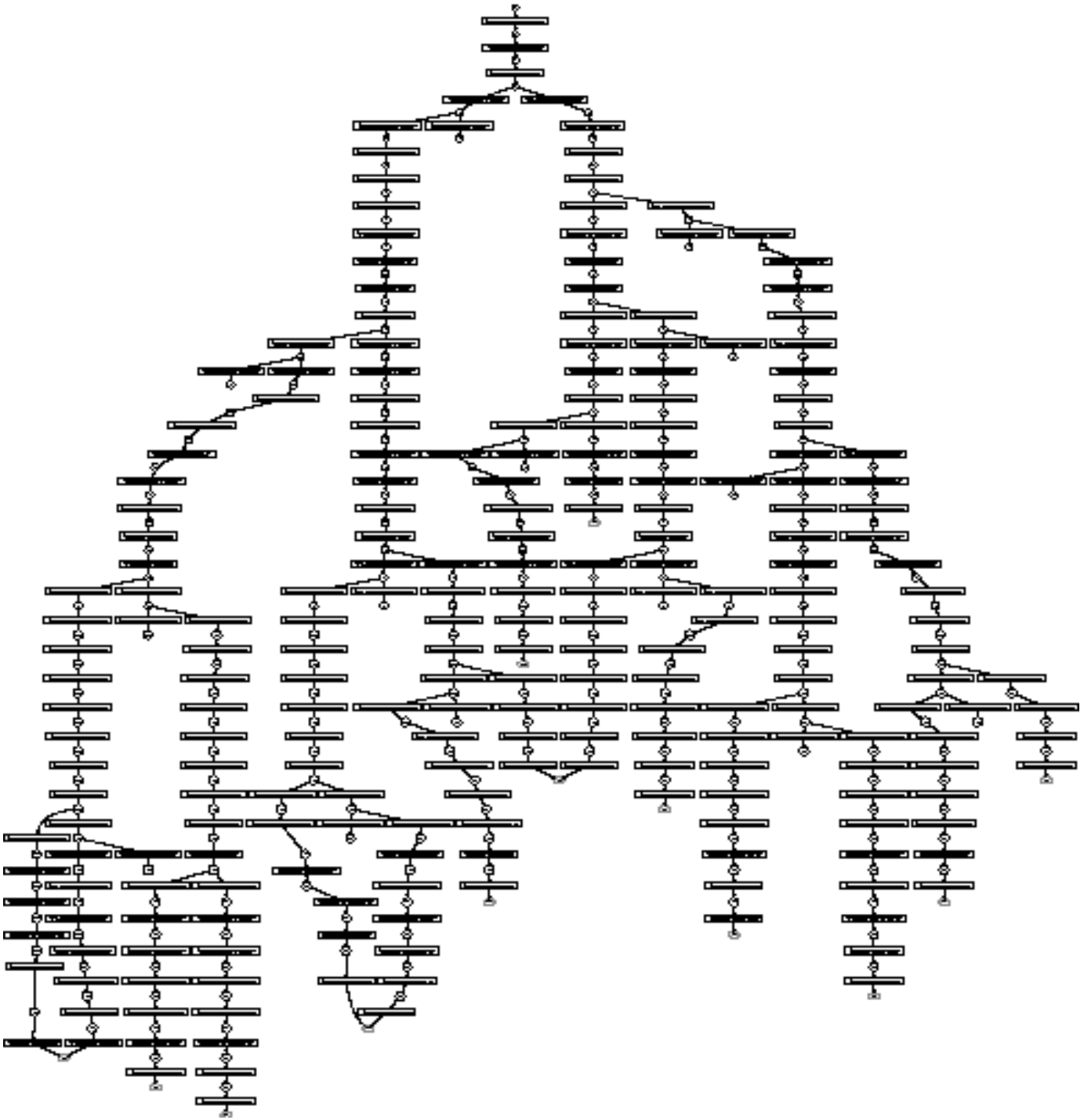
```
public void push(Object o){
    if(counter < 0){}
    else if(counter < max){}
}
```

Use Symbolic Execution

- Constructs execution tree, where every leaf node represents feasible path of execution
  - For every branch statement, executes both branches
  - Accumulates path conditions that must be satisfied for execution of every block of code
  - Implemented as extensions to a number of model checkers (Java Pathfinder, Bogor etc.)

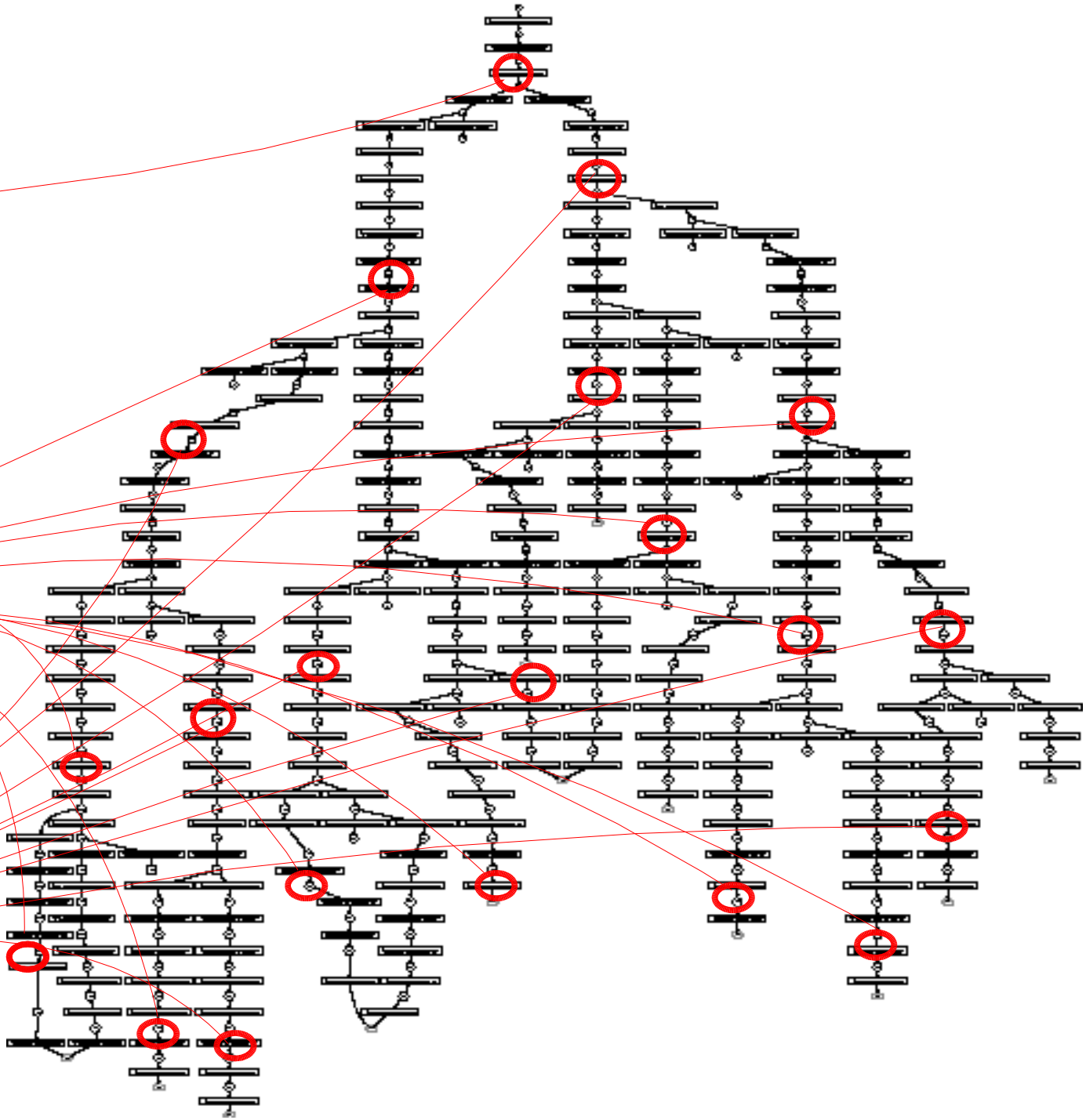
```
1: public Stack init(int max){
2:     this.max = max;
3:     this.counter=-1;
4: }

5: public void push(Object o){
6:     if(counter < 0){
7:             counter = 0;
8:             stack[counter]=i;
9:     }
10:    else if(counter < max){
11:            stack[counter]=o;
12:            counter++;
13:    }
14: }

15: public Object pop(){
16:    Object ret = null;
17:    if(counter>0){
18:            ret = stack[counter];
19:    }
20:    counter--;
21:    return ret;
22: }
```

Symbolic execution tree



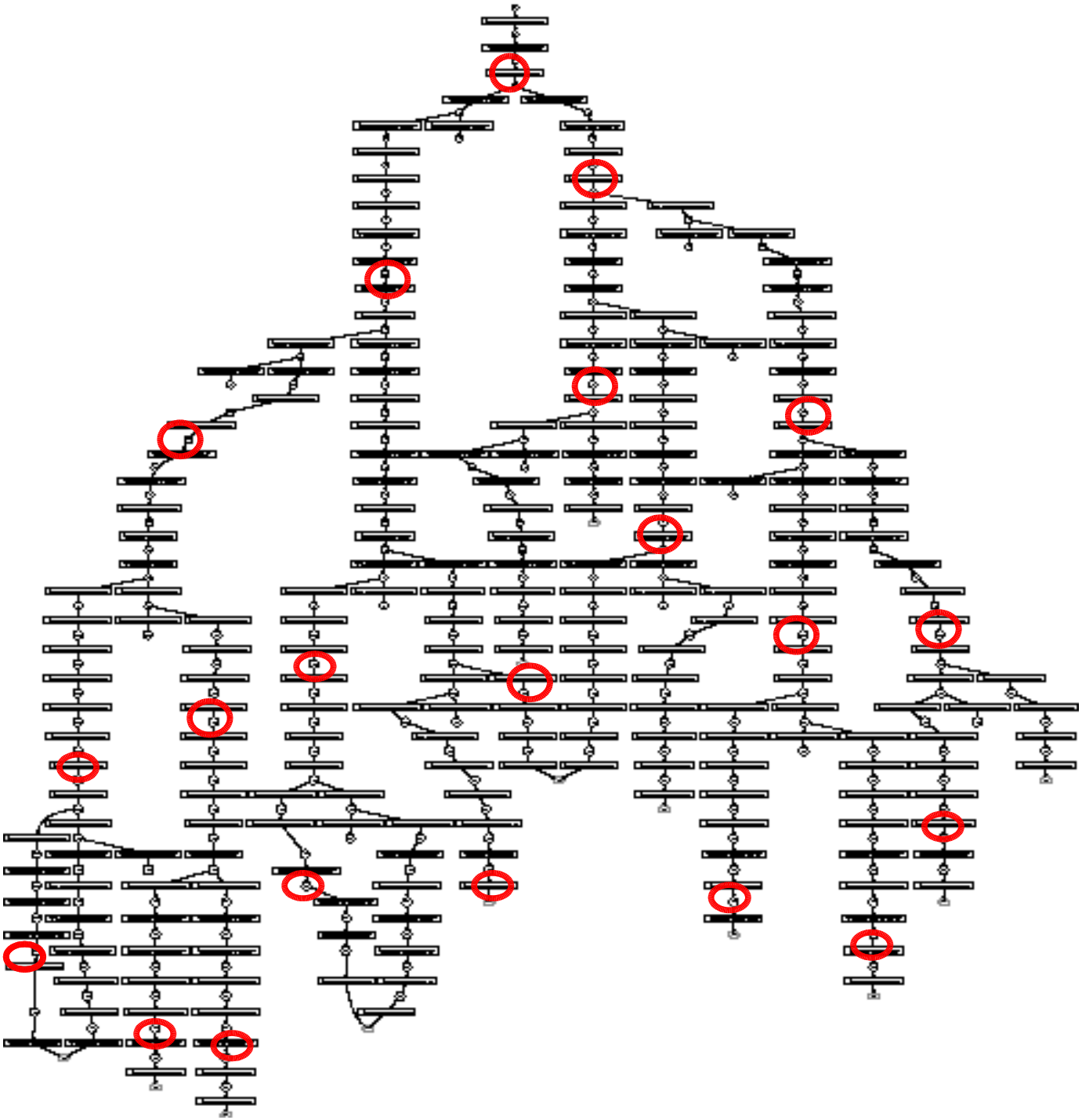
Symbolic execution tree

```
1: public Stack init(int max){
2:     this.max = max;
3:     this.counter=-1;
4: }

5: public void push(Object o){
6:     if(counter < 0){
7:             counter = 0;
8:             stack[counter]=i;
9:     }
10:    else if(counter < max){
11:            stack[counter]=o;
12:            counter++;
13:    }
14: }

15: public Object pop(){
16:    Object ret = null;
17:    if(counter>0){
18:            ret = stack[counter];
19:    }
20:    counter--;
21:    return ret;
22: }
```

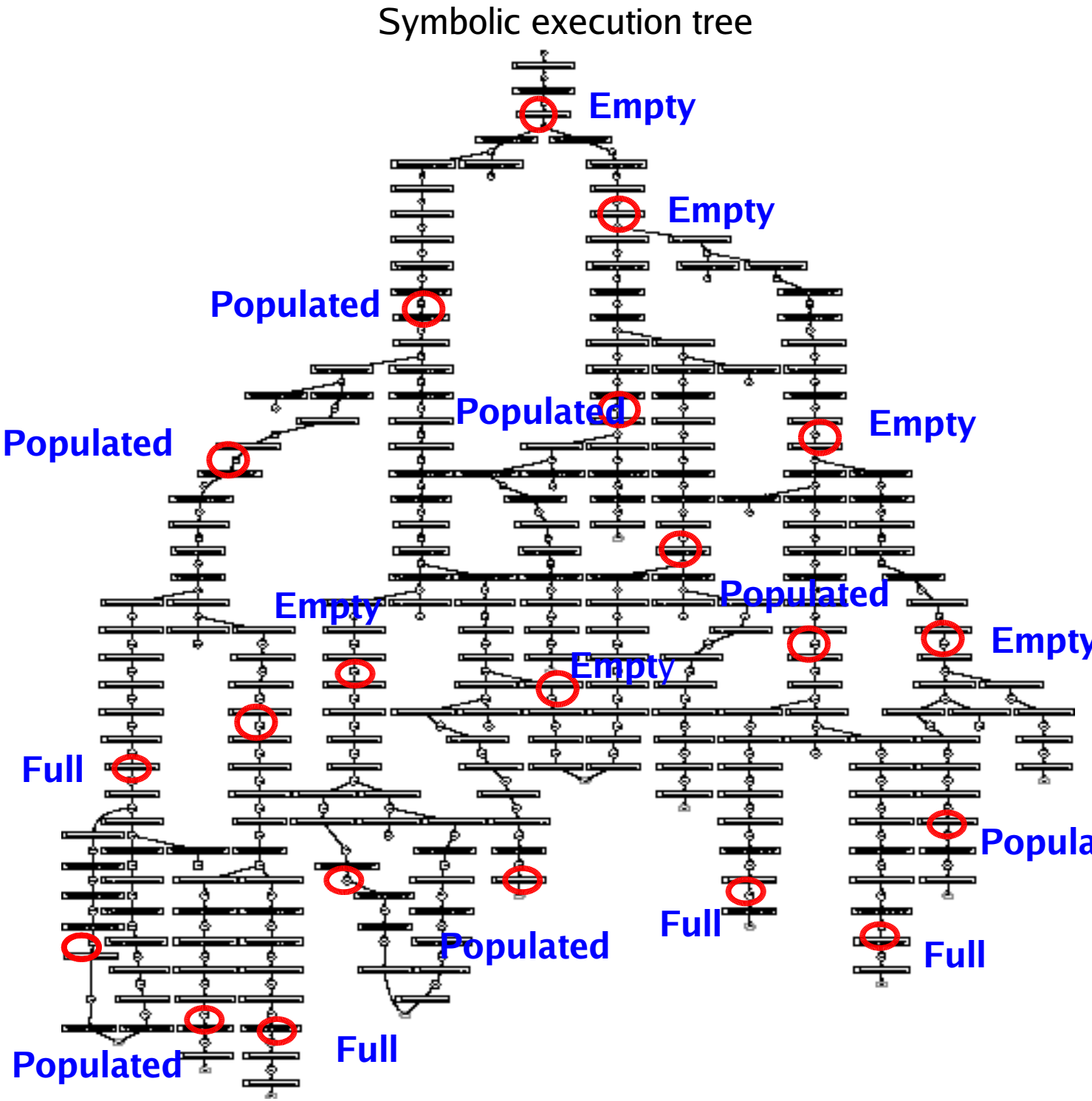


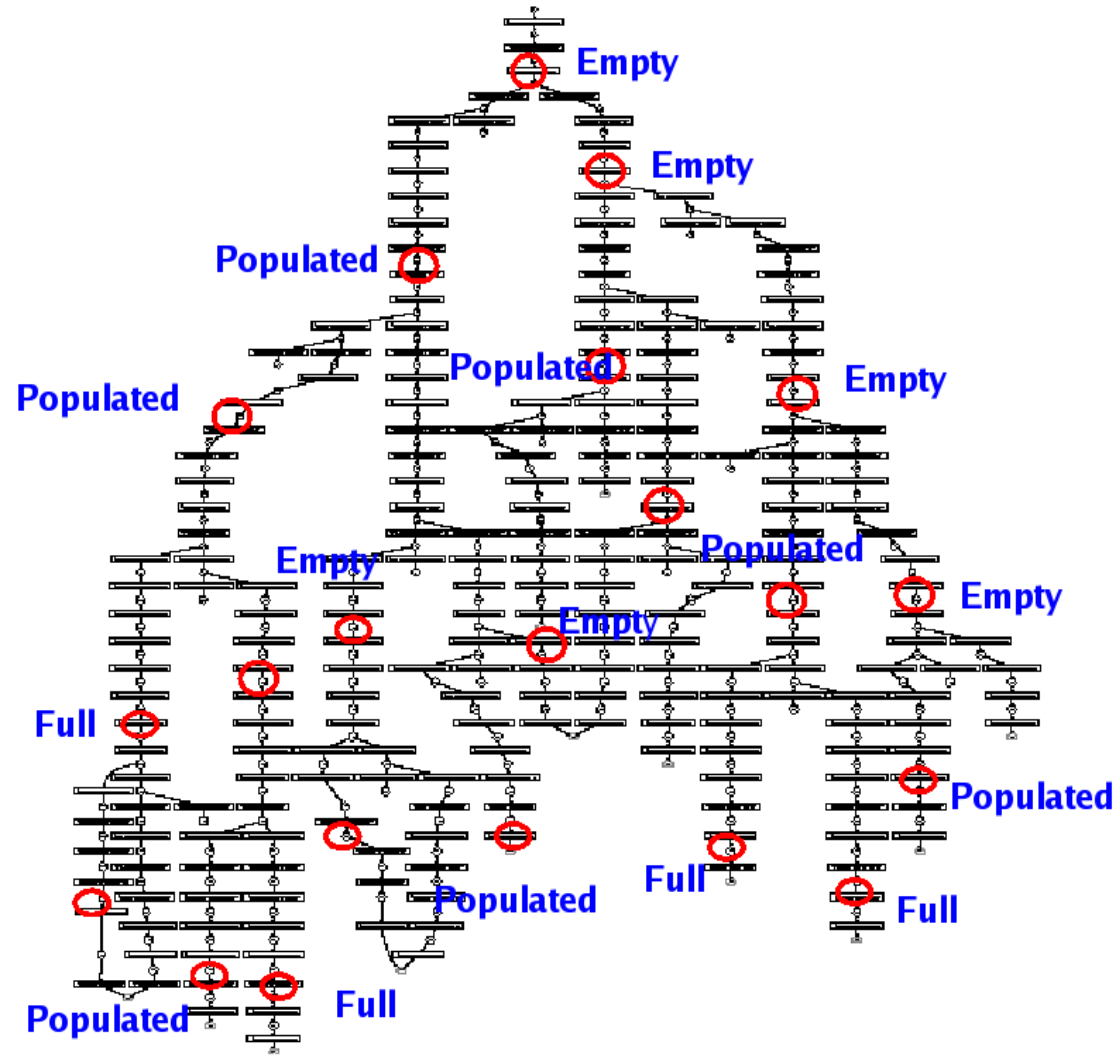
```
1: public Stack init(int max){
2:     this.max = max;
3:     this.counter=-1;
4: }

5: public void push(Object o){
6:     if(counter < 0){
7:             counter = 0;
8:             stack[counter]=i;
9:     }
10:    else if(counter < max){
11:            stack[counter]=o;
12:            counter++;
13:    }
14: }

15: public Object pop(){
16:    Object ret = null;
17:    if(counter>0){
18:            ret = stack[counter];
19:    }
20:    counter--;
21:    return ret;
22: }
```

Symbolic execution tree

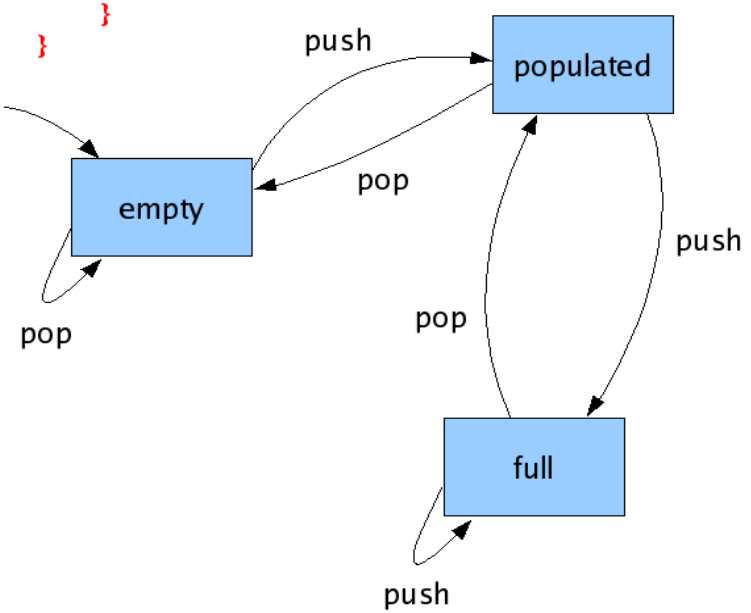


```
1: public Stack init(int max){
2:     this.max = max;
3:     this.counter=-1;
4: }

5: public void push(Object o){
6:     if(counter < 0){
7:             counter = 0;
8:             stack[counter]=i;
9:     }
10:    else if(counter < max){
11:            stack[counter]=o;
12:            counter++;
13:    }
14: }

15: public Object pop(){
16:    Object ret = null;
17:    if(counter>0){
18:            ret = stack[counter];
19:    }
20:    counter--;
21:    return ret;
22: }
```



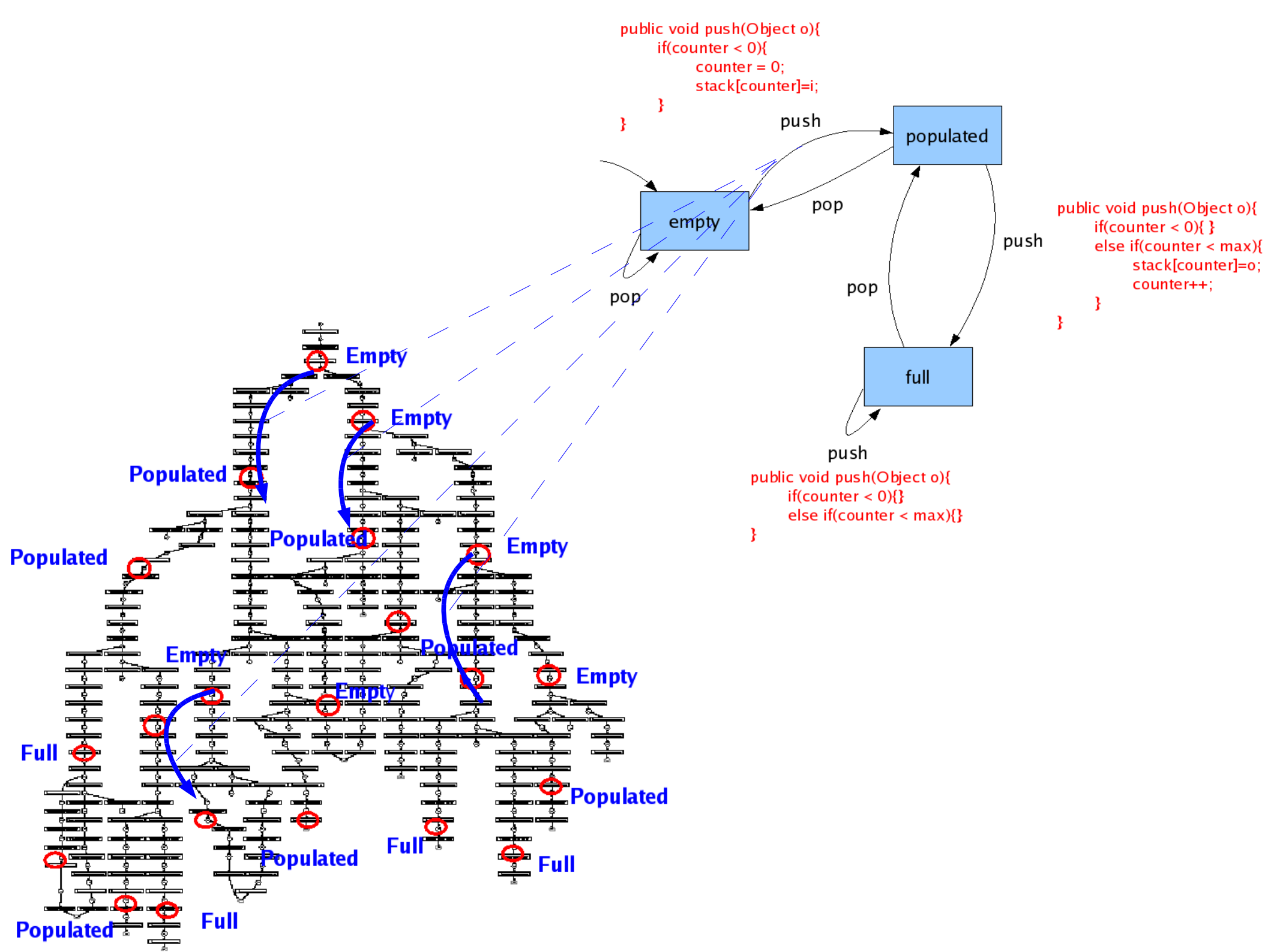



```
public void push(Object o){
    if(counter < 0){
        counter = 0;
        stack[counter]=i;
    }
}
```

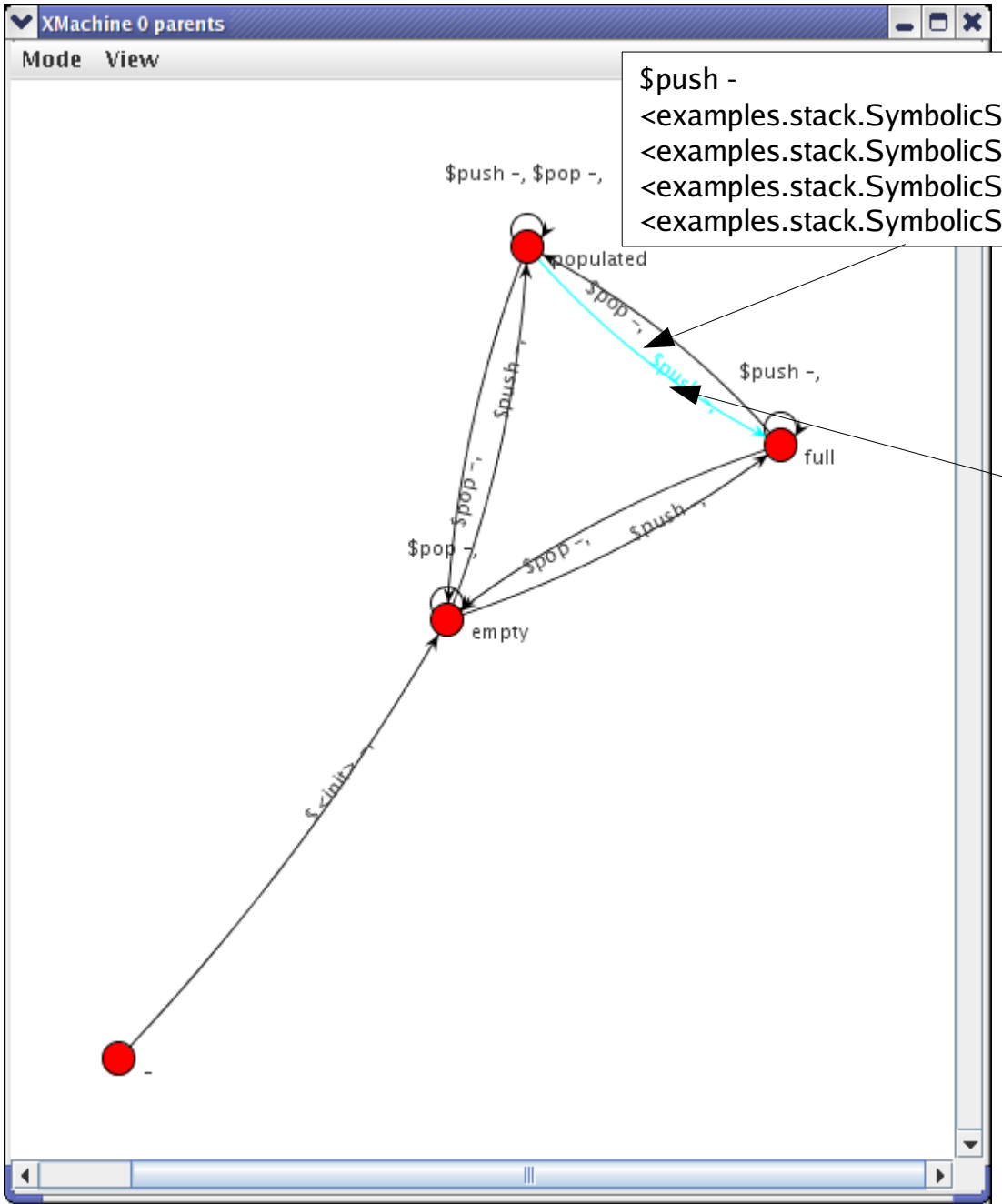



```
public void push(Object o){
    if(counter < 0){ }
    else if(counter < max){
        stack[counter]=o;
        counter++;
    }
}
```

```
public void push(Object o){
    if(counter < 0){}
    else if(counter < max){}
}
```


public void push(Object o){
if(counter < 0){
counter = 0;
stack[counter]=i;
}
}
push
populated
pop
empty
pop
push
public void push(Object o){
if(counter < 0){ }
else if(counter < max){
stack[counter]=o;
counter++;
}
}
pop
full
push
public void push(Object o){
if(counter < 0){}
else if(counter < max){}
}
Empty
Empty
Populated
Populated
Populated
Empty
Empty
Empty
Populated
Empty
Full
Populated
Full
Full
Populated
Populated
Full

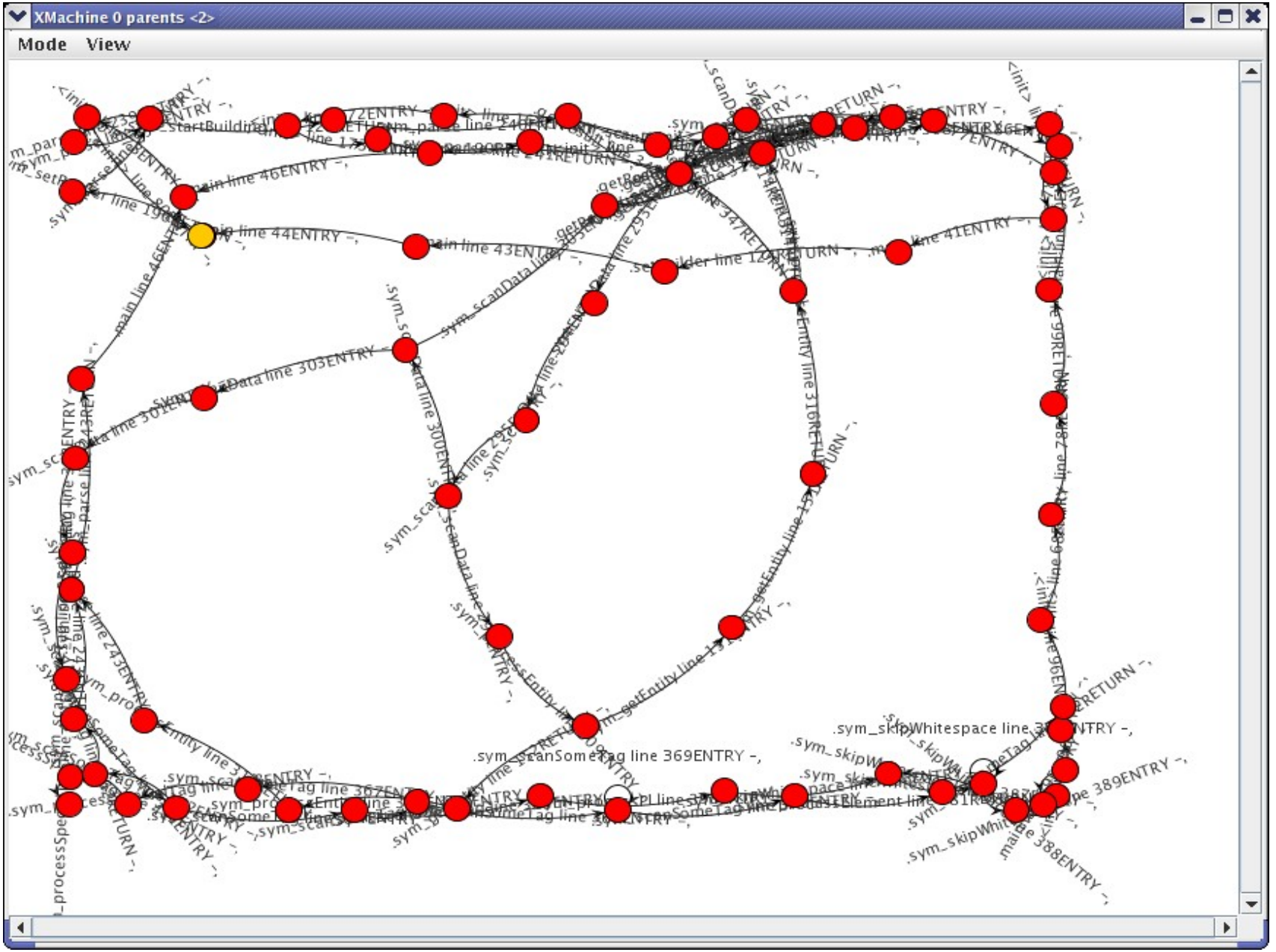
Demo Backup



$push -
<examples.stack.SymbolicStack2: void push(gov.nasa.jpf.symbolic.integer.Expression): 26 - 26
<examples.stack.SymbolicStack2: void push(gov.nasa.jpf.symbolic.integer.Expression): 18 - 18
<examples.stack.SymbolicStack2: void push(gov.nasa.jpf.symbolic.integer.Expression): 23 - 24
<examples.stack.SymbolicStack2: void push(gov.nasa.jpf.symbolic.integer.Expression): 22 - 22

0 >= (stackSize - 1) &&
0 >= 0 &&
0 < stack.length &&
stack.length > 0 &&
stackSize <= 3 &&
stackSize >= 0

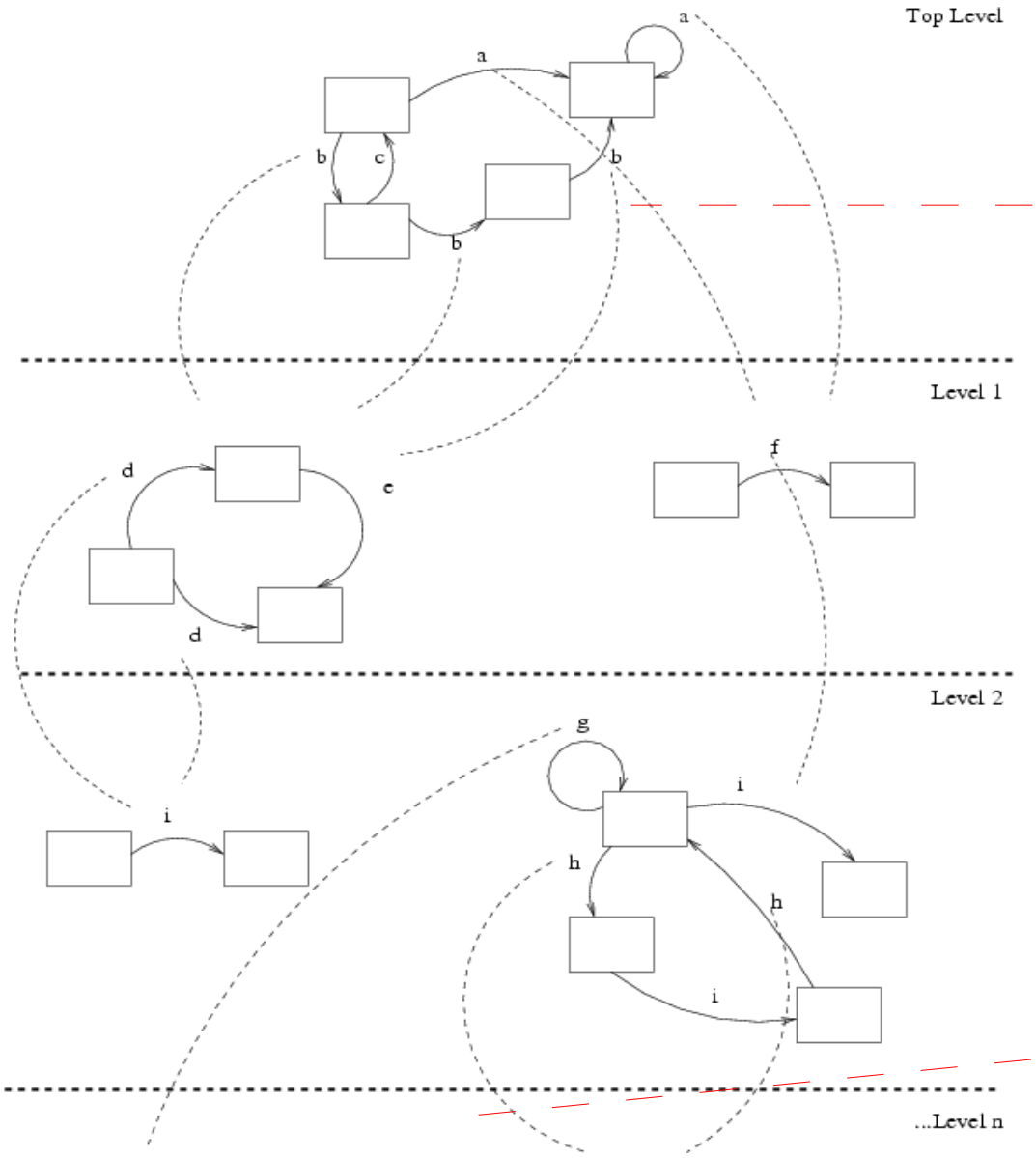
Demo Backup 2



Future Work

- Investigate symbolic execution of loops
  - Currently simply limit search depth to a level k
- Looking at different state representations (data and control)
  - Effect of state representation on test set size
- Constructing a hierarchy of state machines
  - See next slide

Hierarchy of X-Machines



X-Machine Functions represent user-level functions / features

X-Machines represent atomic / library functions