Lecture 12

# Symbolic Testing II

Zvonimir Rakamarić
University of Utah

# Last Time

- Drawbacks of concrete testing
- Symbolic execution
- Solutions for the path explosion problem
  - Structural abstraction
  - Compositional symbolic execution

# Symbolic Execution

- Key idea: execution of programs using symbolic input values instead of concrete data
- Concrete vs symbolic
  - Concrete execution
    - Program takes only one path determined by input values
  - Symbolic execution
    - Program can take any feasible path – coverage!

# Symbolic Program State

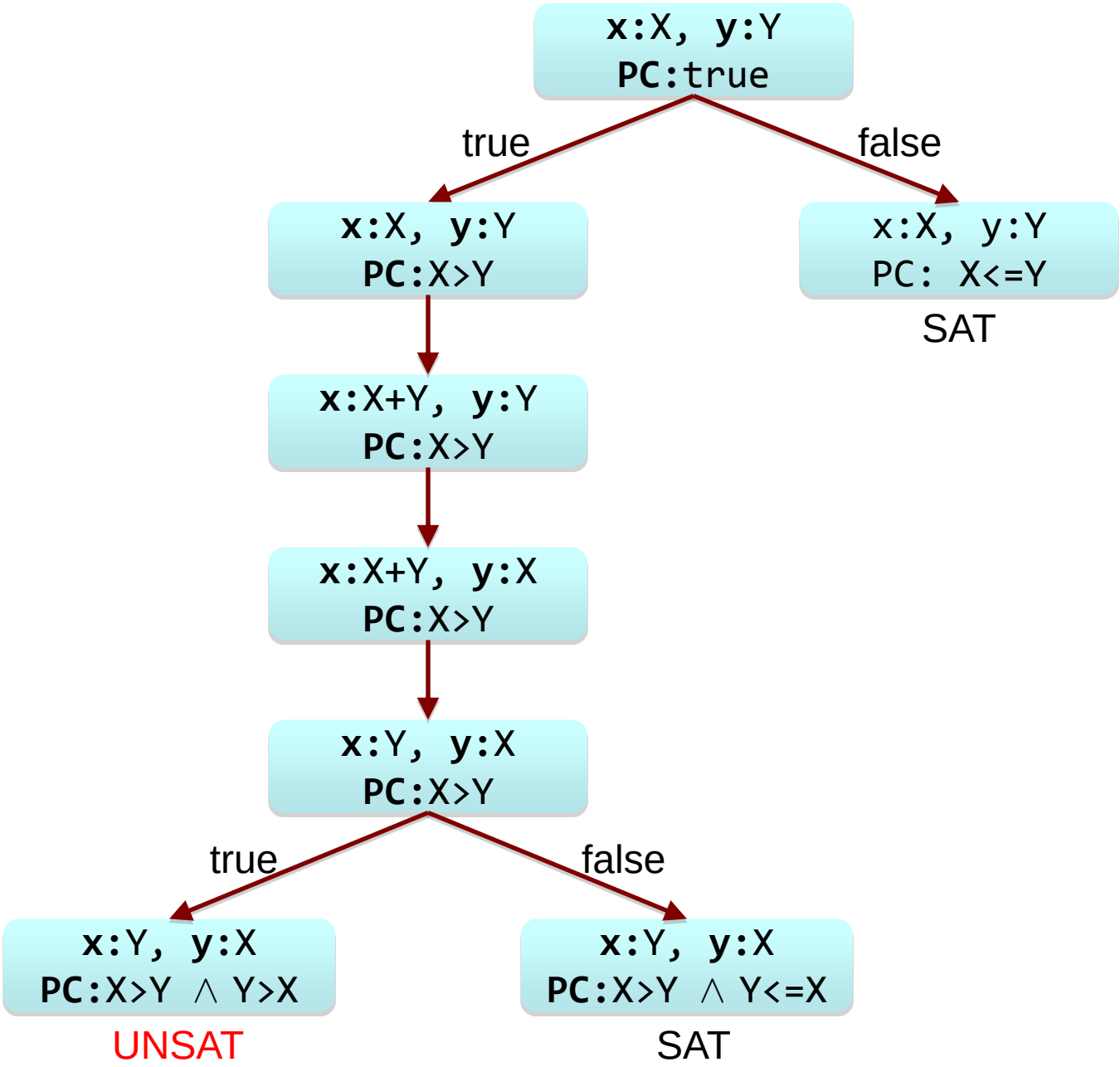
- Symbolic values of program variables
- Path condition (PC)
  - Logical formula over symbolic inputs
  - Accumulates constraints that inputs have to satisfy for the particular path to be executed
  - If a path is feasible its PC is satisfiable
- Program location

# Symbolic Execution Tree

- Characterizes execution paths constructed during symbolic execution
- Nodes are symbolic program states
- Edges are labeled with program transitions

# Example

```
1) int x, y;
2) if (x > y) {
3)    x = x + y;
4)    y = x - y;
5)    x = x - y;
6)    if (x > y)
7)       assert false;
8) }
```


x:X, y:Y
PC:true
true
false
x:X, y:Y
PC:X>Y
x:X, y:Y
PC: X<=Y
SAT
x:X+Y, y:Y
PC:X>Y
x:X+Y, y:X
PC:X>Y
x:Y, y:X
PC:X>Y
true
false
x:Y, y:X
PC:X>Y ∧ Y>X
UNSAT
x:Y, y:X
PC:X>Y ∧ Y<=X
SAT

# Further Limitations of Symbolic Execution

- Limited by the power of constraint solver
  - Cannot handle non-linear and very complex constraints
- Inherently white-box technique
  - Source code (or equivalent) is required for precise symbolic execution
  - Modeling libraries is a huge problem

# This Time

- Combining concrete and symbolic execution
- Many names referring to the same thing:
  - DART (directed automated random testing)
  - Concolic (concrete + symbolic) execution
  - Dynamic symbolic execution
- Learning interfaces of software components
  - Combining concolic execution with automata learning

# Concolic Execution

- Combination of concrete and symbolic execution to overcome the two weaknesses of classic symbolic execution
- Algorithm
  - Execute program concretely
  - Collect the symbolic path condition along the way
  - Negate a constraint on the path condition after the run and solve it to get a model
  - Execute again with the newly found concrete input values

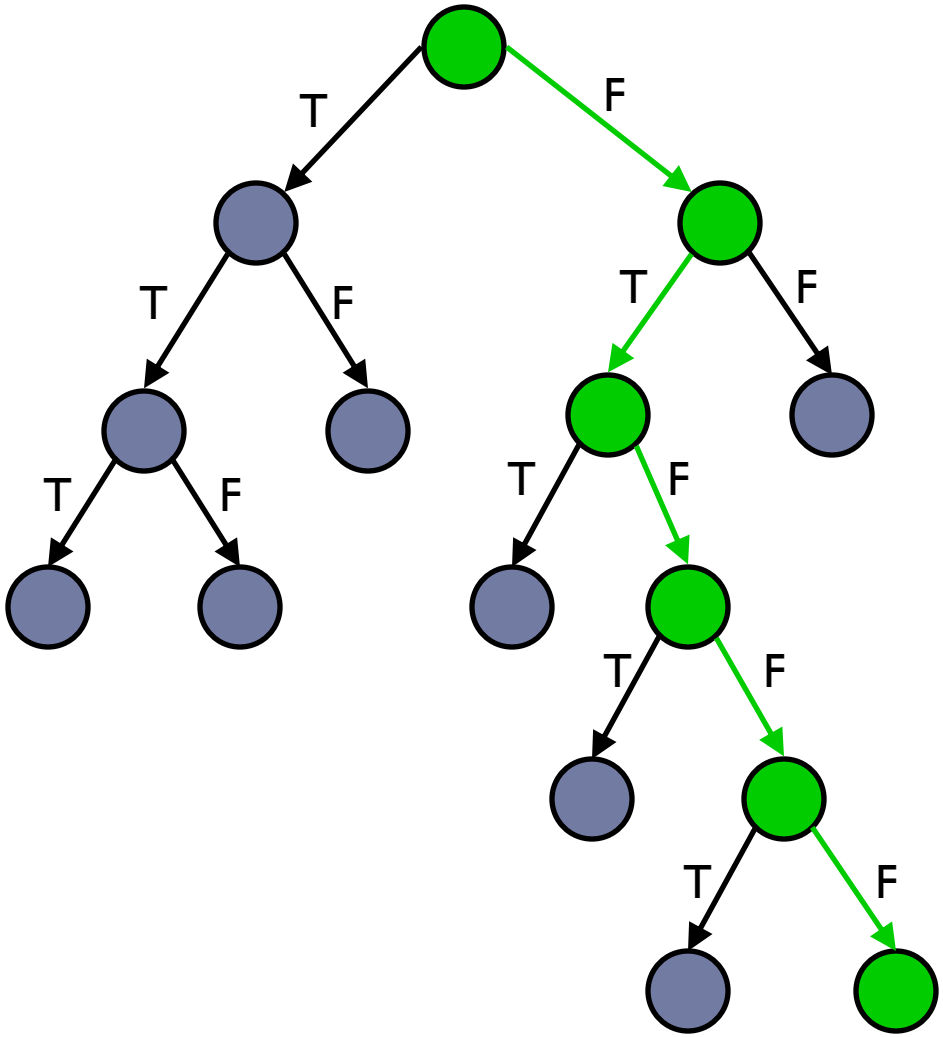
# Simple Example I

```
void foo(int x, int y) {
   z = 2*y;
   if (z == x) {
      if (x > y+10) {
         assert false;
      }
   }
}
```

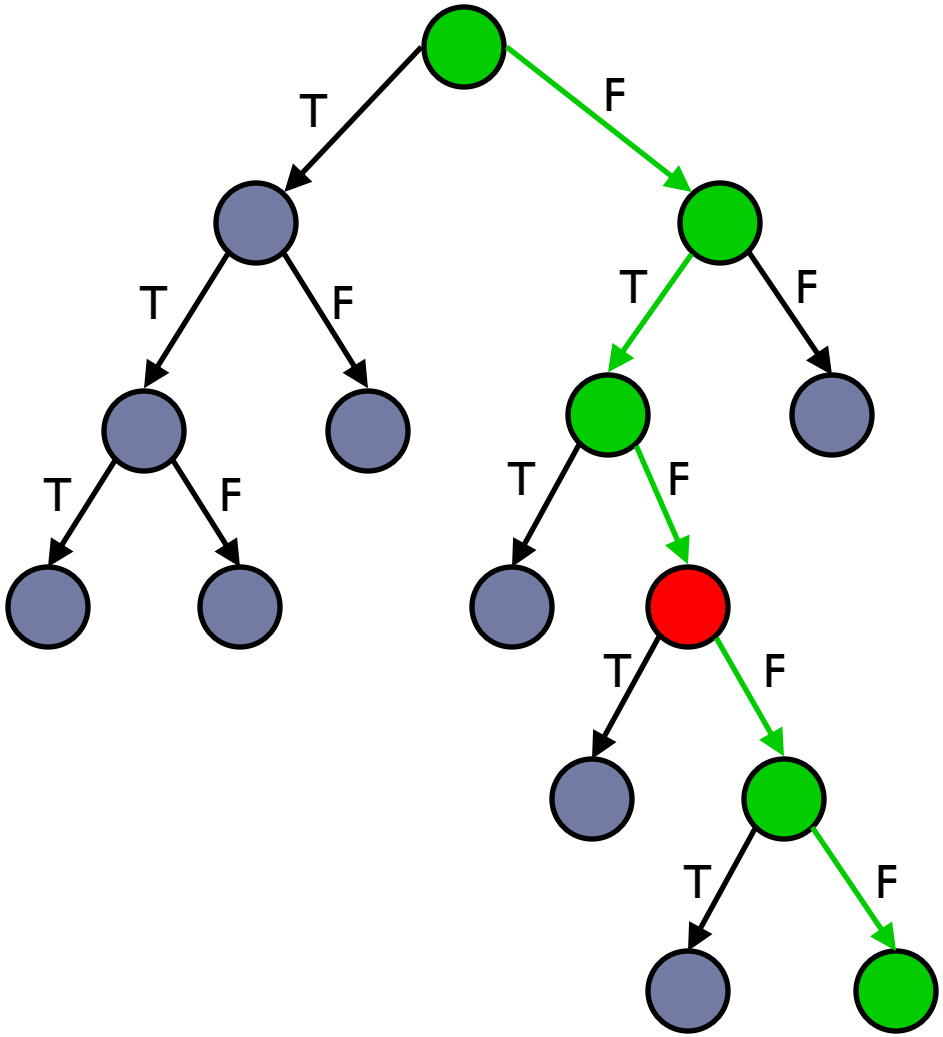
# Simple Example II

```
void foo(int x, int y) {
  z = double(y); // no source or complex
  if (z == x) {
    if (x > y+10) {
      assert false;
    }
  }
}
```

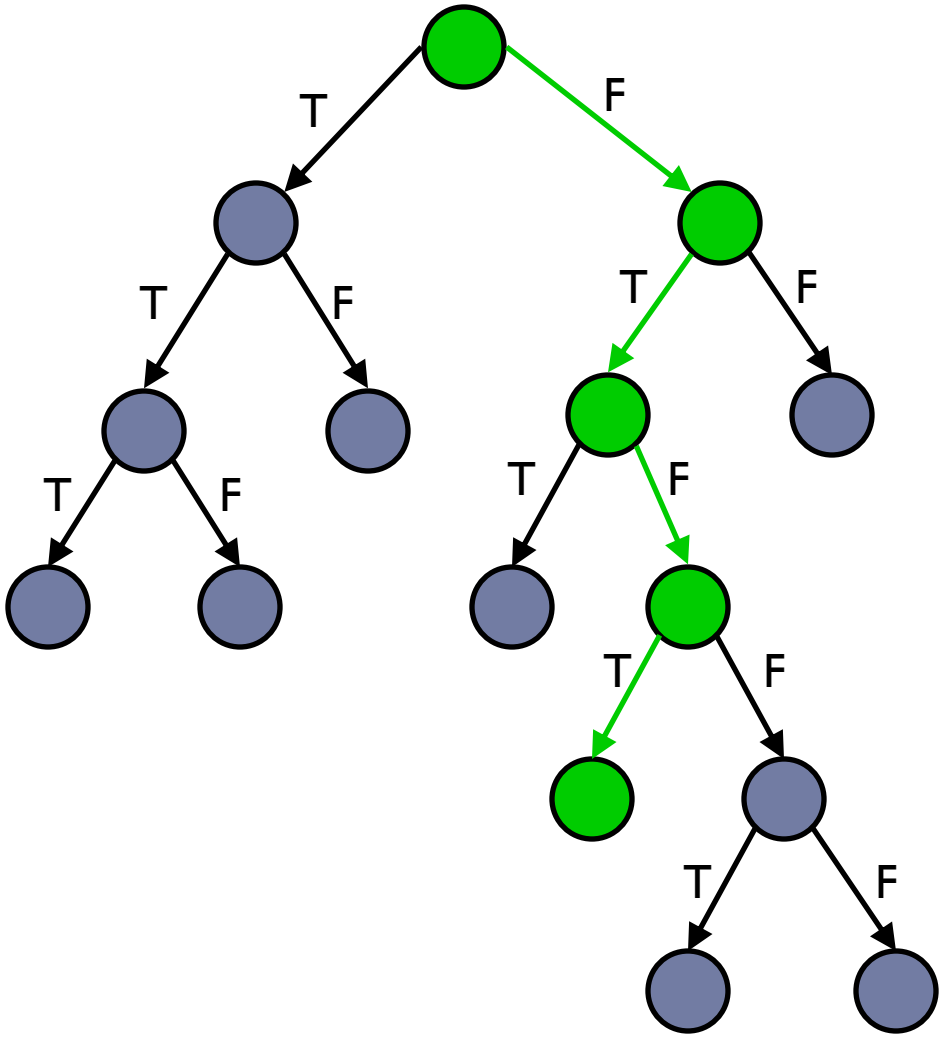
# High-Level Picture



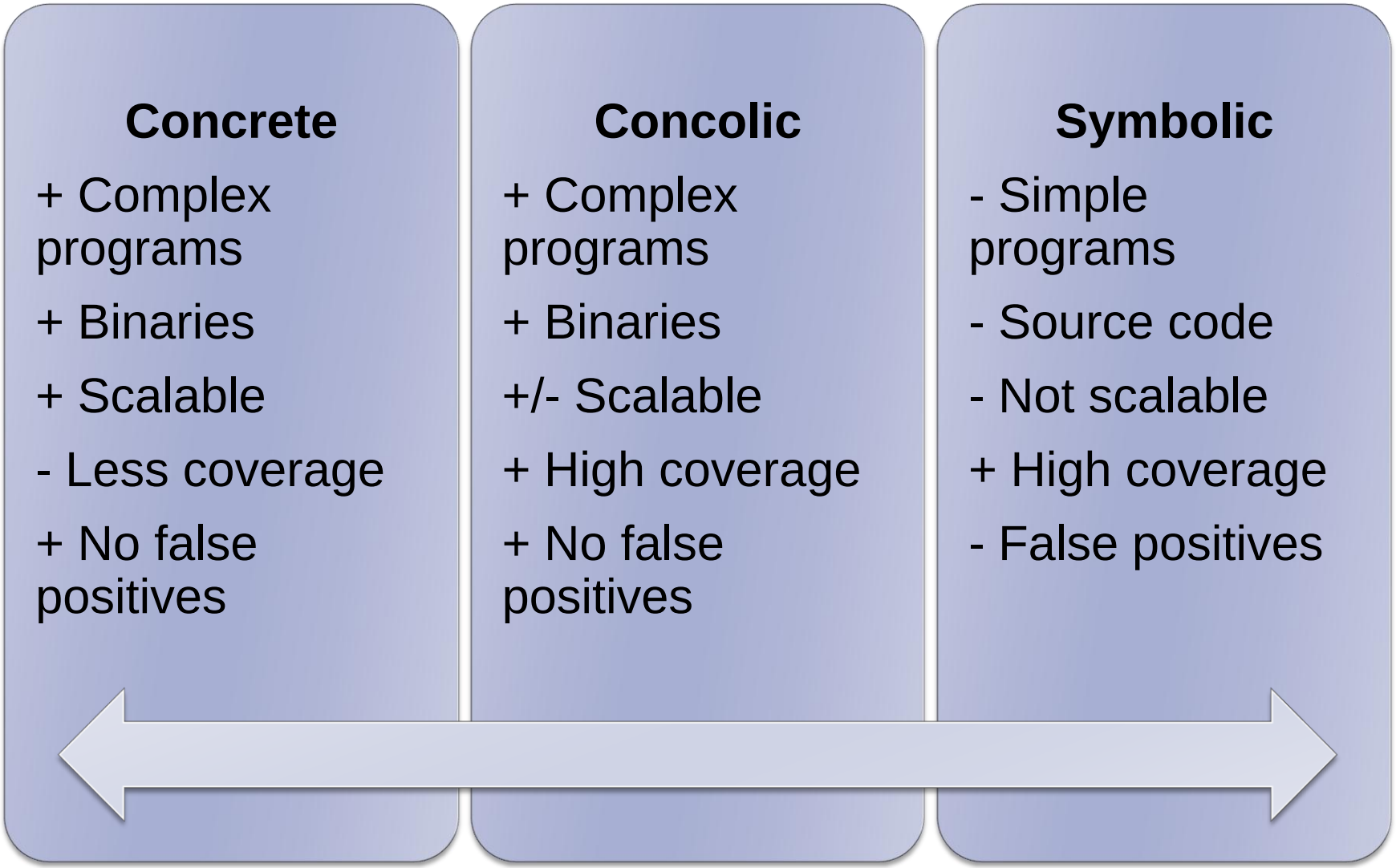
# High-Level Picture



# High-Level Picture



# Concolic Covering Middle Ground



# Recent Success Stories

- SAGE
  - Microsoft's internal tool for finding security bugs
  - White-box fuzzing
    - Concolic execution for finding bugs in file parsers (jpeg, docx, ppt,…)
  - Last line of defense
  - Big clusters continuously running SAGE
- KLEE
  - Open source concolic executor
  - Runs on top of LLVM
  - Has found lots of problems in open-source software

# Learning Interfaces of Software Components

# Motivation

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```

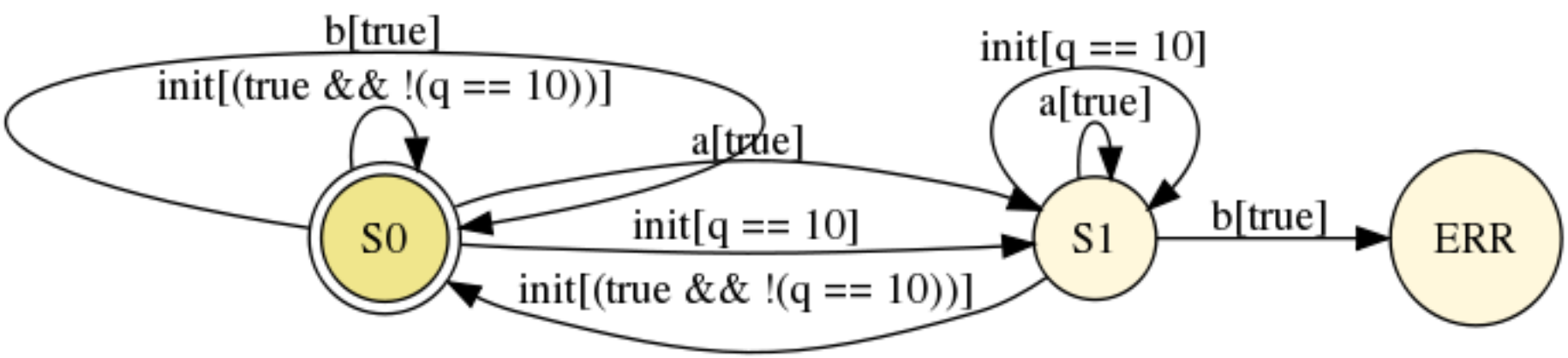
- **`init`** can be called unconditionally
- **`a`** can be called unconditionally
- **`b`** can be called after **`init`** only when **`y != 10`**

# Goal

- Learn *temporal interfaces* of software components
  - Legal and illegal sequences of method calls defined as an automaton

# Goal cont.

```
public class Example {
   ...
}
```




(All sequences to depth 2.)

# Active Automata Learning

- Algorithm is called L*
  - L* learns unknown regular language U (over alphabet $\Sigma$) and produces minimal DFA A such that L(A) = U
- L* learner communicates with a teacher using two types of queries
  - Membership queries: should word w be included in L(A)?
    - Expected answer: yes/no
  - Equivalence queries: here is a DFA A – is L(A) = U?
    - Expected answer: yes/no+counterexample

# Interface Learning with L*

- L* uses a teacher to answer the following queries
  - Membership queries
    - Whether or not a given sequence of method calls leads to an error or not in the implementation
  - Equivalence queries
    - Whether a conjectured DFA captures all the behaviors of the implementation

# Answering Membership Queries

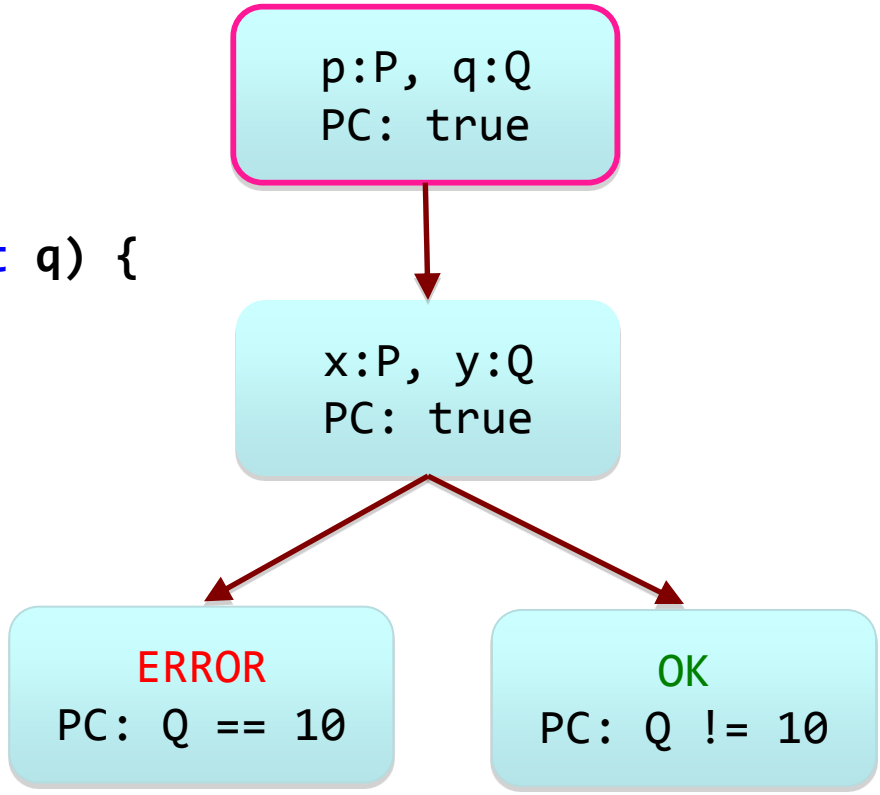
- L* uses a teacher to answer the following queries
  - Membership queries
    - Whether or not a given sequence of method calls leads to an error or not in the implementation
  - Equivalence queries
    - Whether a conjectured DFA captures all the behaviors of the implementation

# Running Example

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```

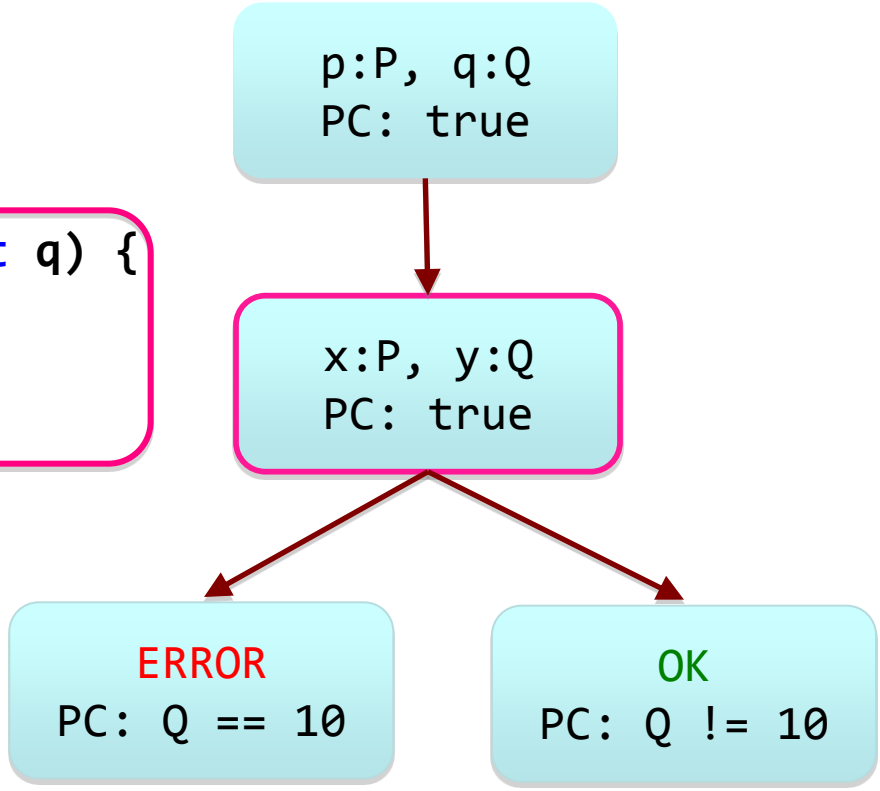
# Executing query <init;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



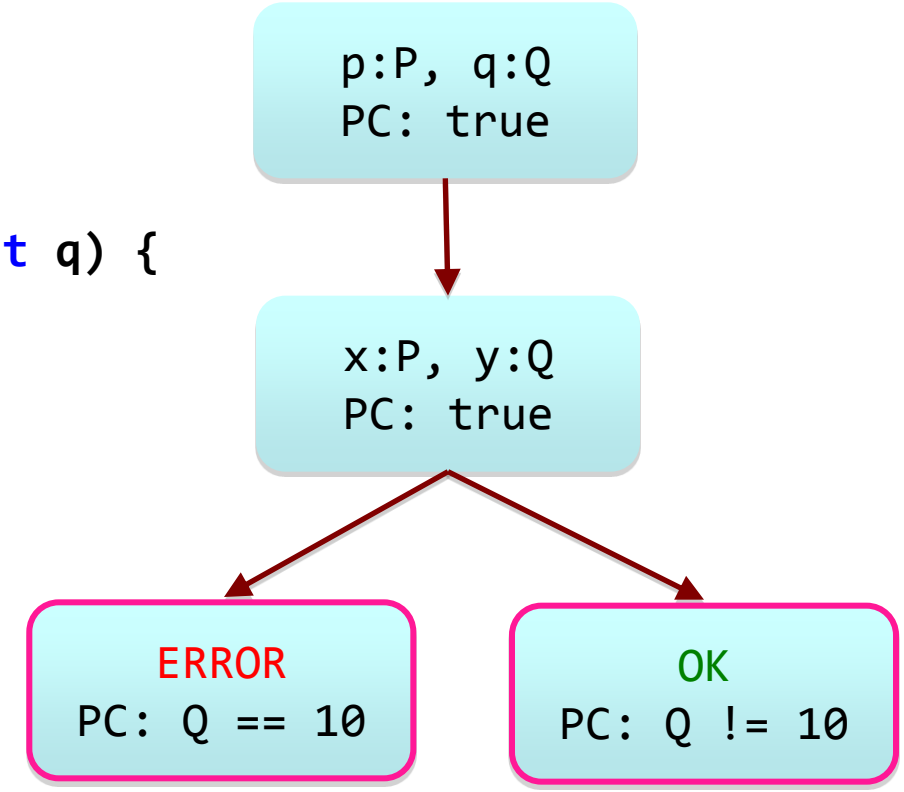
# Executing query <init;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



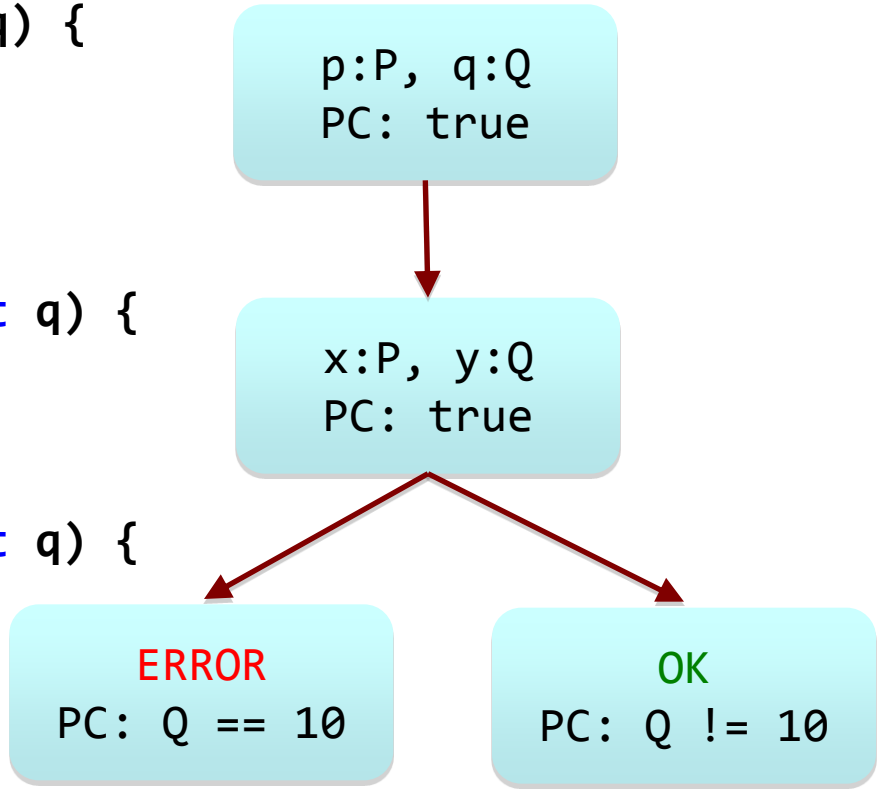
# Executing query <init;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



# Refinement: Split init

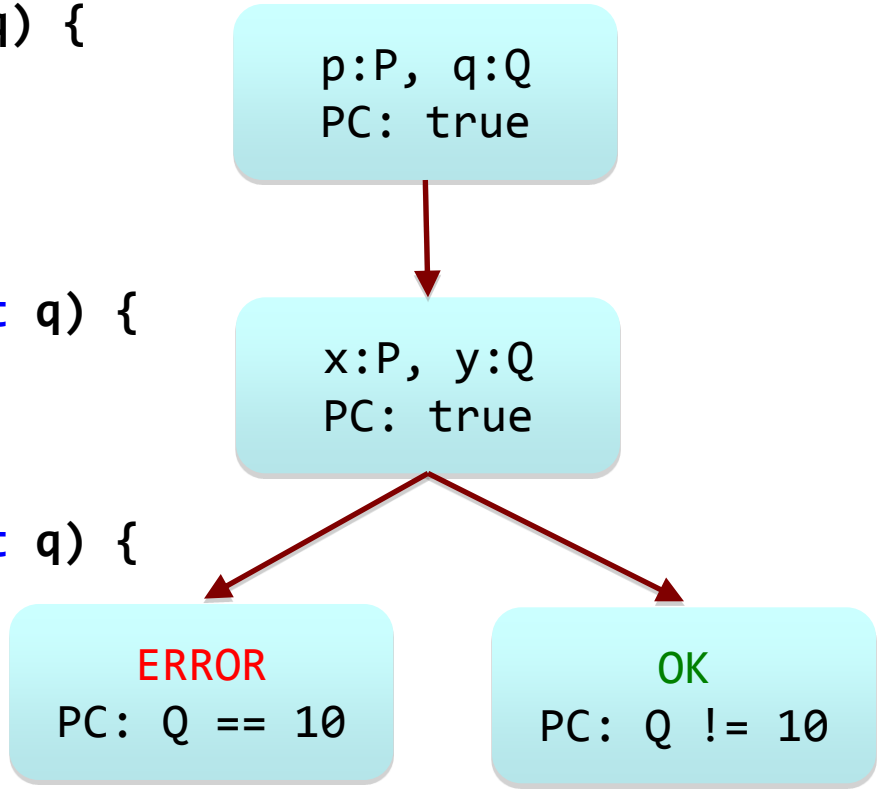
```
public static void init(int p, int q) {
  x = p;
  y = q;
}

public static void init_0(int p, int q) {
  assume q != 10;
  init(p, q);
}
public static void init_1(int p, int q) {
  assume q == 10;
  init(p, q);
}
```




```
init_0 :=
init[q != 10]

init_1 :=
init[q == 10]
```

# Restart Learning

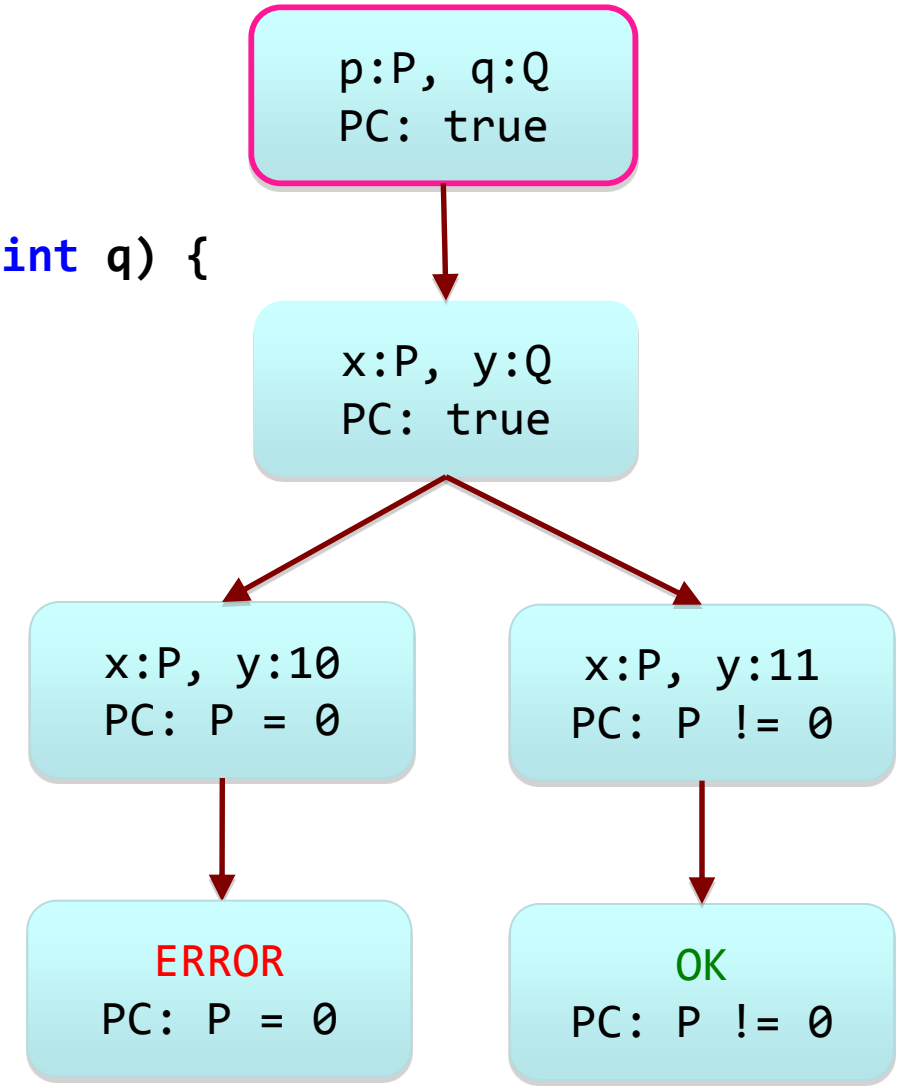
```
public static void init(int p, int q) {
  x = p;
  y = q;
}

public static void init_0(int p, int q) {
  assume q != 10;
  init(p, q);
}
public static void init_1(int p, int q) {
  assume q == 10;
  init(p, q);
}
```




new learner alphabet:
**{init_0, init_1, a, b}**
learning restarts, re-using results from previous iterations

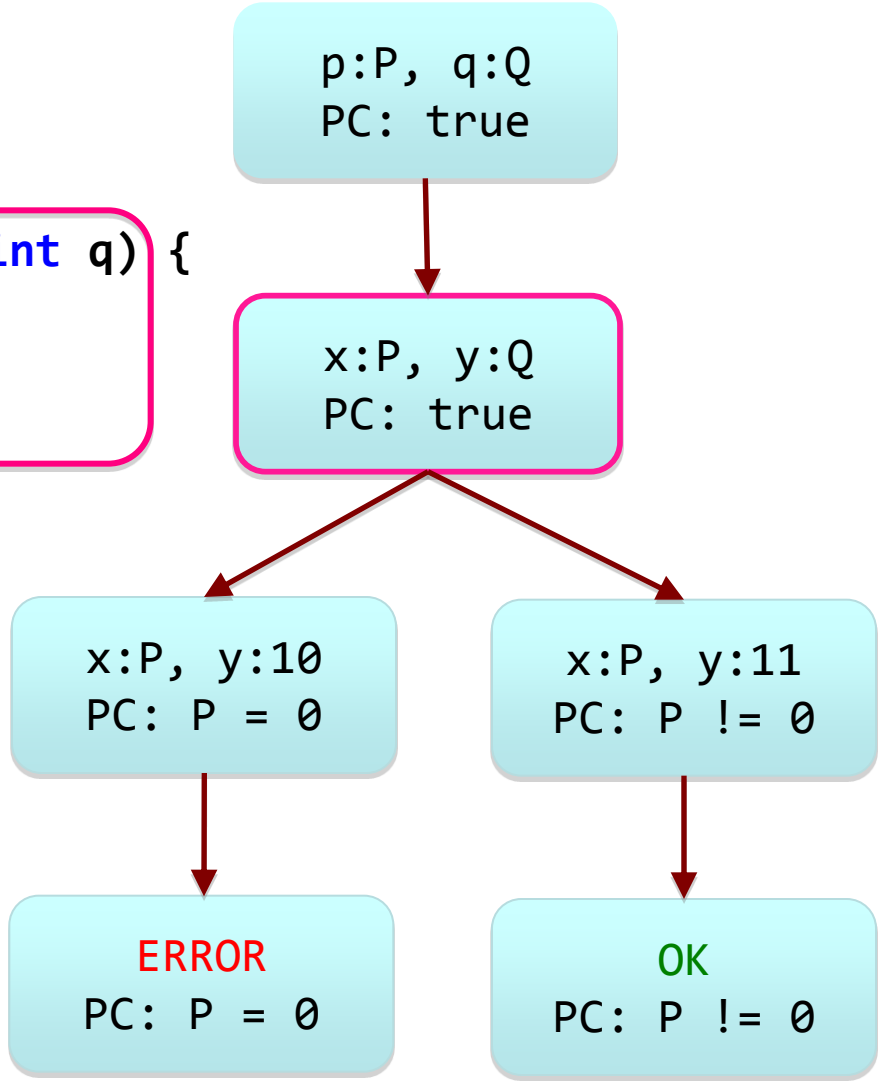
# Executing query <init_0;a;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init_0(int p, int q) {
    assume q != 10;
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



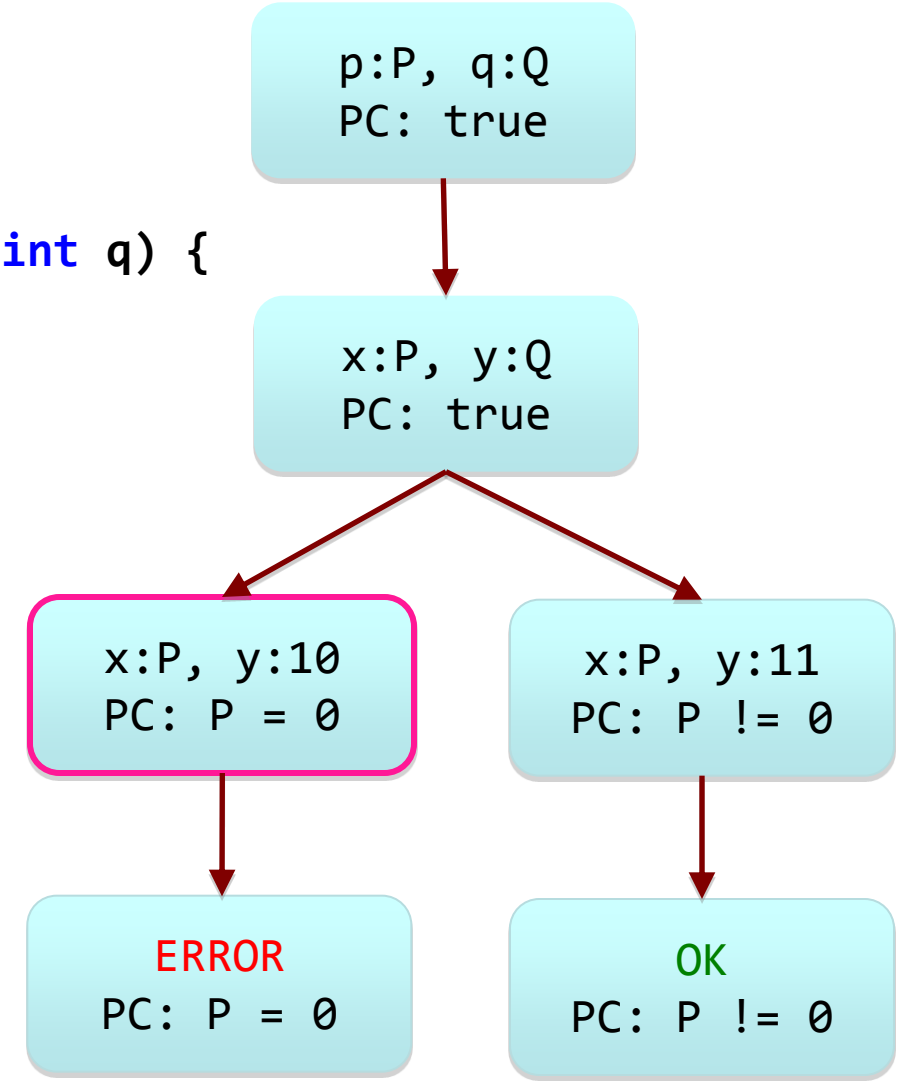
# Executing query <init_0;a;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init_0(int p, int q) {
    assume q != 10;
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



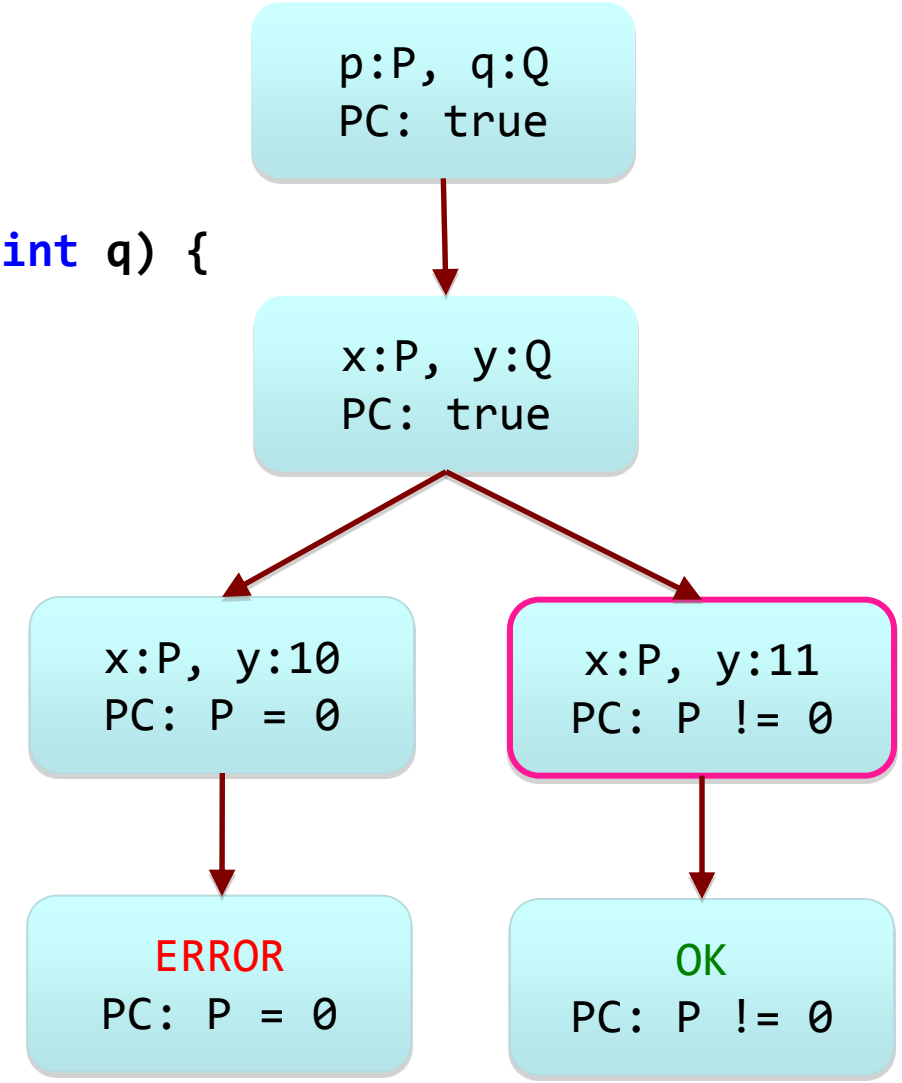
# Executing query <init_0;a;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init_0(int p, int q) {
    assume q != 10;
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



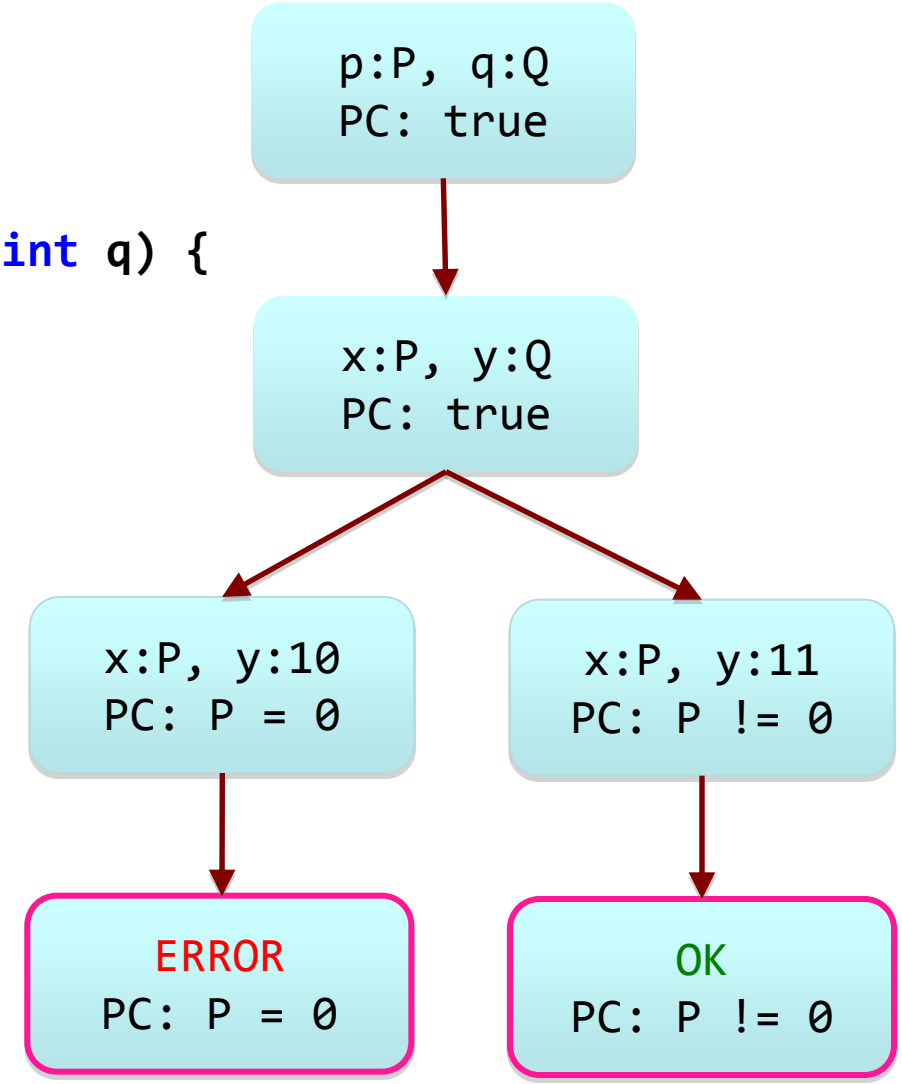
# Executing query <init_0;a;b>

```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init_0(int p, int q) {
    assume q != 10;
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



# Executing query <init_0;a;b>

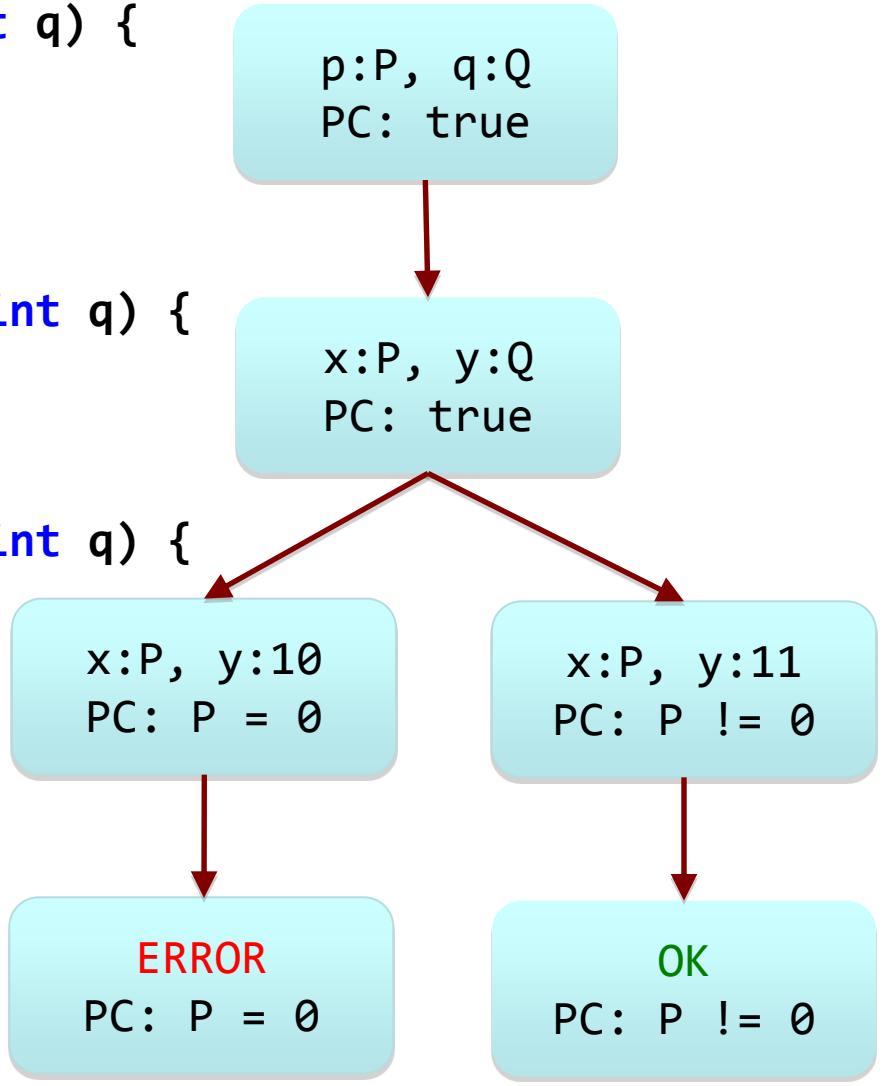
```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init_0(int p, int q) {
    assume q != 10;
    x = p;
    y = q;
  }
  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



# Refinement: Split init_0

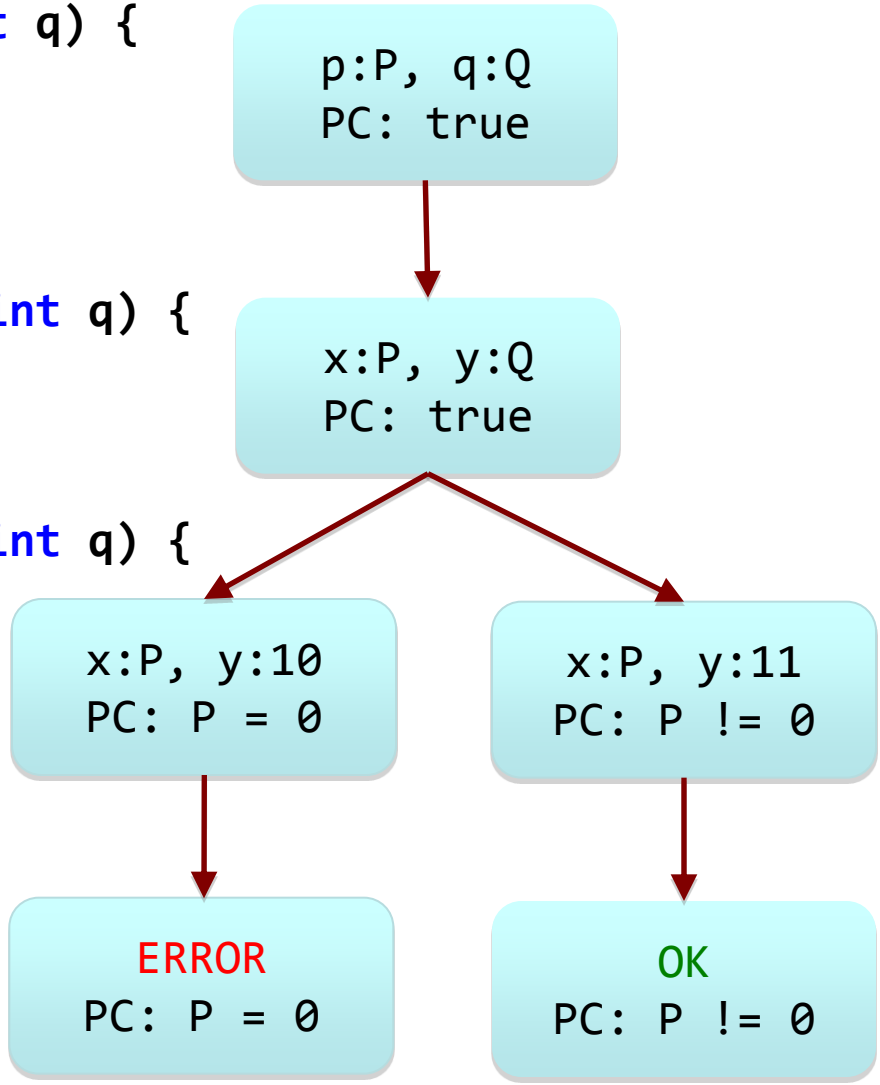
```
public static void init_0(int p, int q) {
  assume q != 10;
  x = p;
  y = q;
}
public static void init_0_0(int p, int q) {
  assume p == 0 && q != 10;
  init(p, q);
}
public static void init_0_1(int p, int q) {
  assume p != 0 && q != 10;
  init(p, q);
}
```

```
init_0_0 :=
init[q != 10 && p == 0]

init_0_1 :=
init[q != 10 && p != 0]
```



# Restart Learning

```
public static void init_0(int p, int q) {
  assume q != 10;
  x = p;
  y = q;
}
public static void init_0_0(int p, int q) {
  assume p == 0 && q != 10;
  init(p, q);
}
public static void init_0_1(int p, int q) {
  assume p != 0 && q != 10;
  init(p, q);
}
```



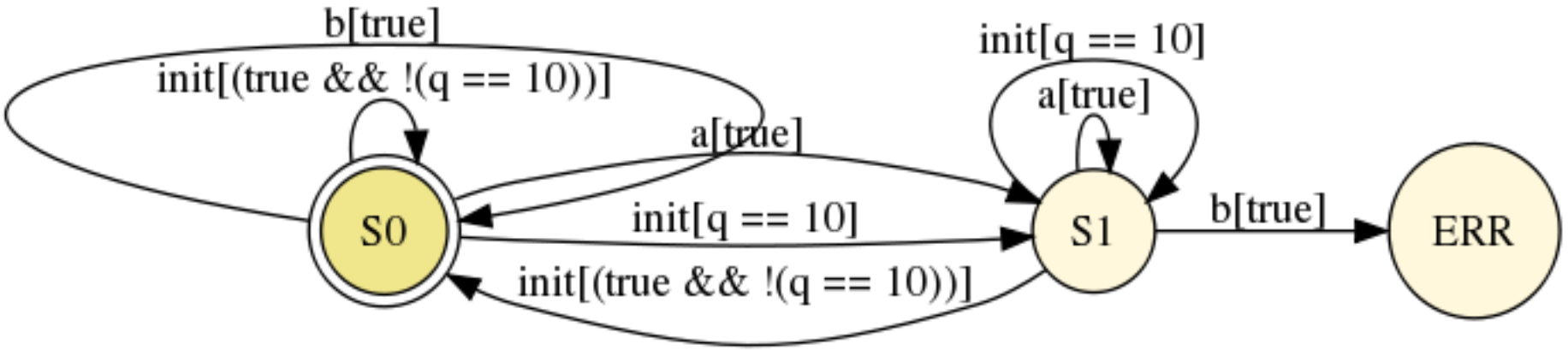

new learner alphabet:
**{init_0_0, init_0_1, init_1, a, b}**
learning restarts

# Answering Equivalence Queries

- L* uses a teacher to answer the following queries
  - Membership queries
    - Whether or not a given sequence of method calls leads to an error or not in the implementation
  - Equivalence queries
    - Whether a conjectured DFA captures all the behaviors of the implementation

# Unbounded Loops in Conjectures

- Component have no loops, but conjectures do!




(All sequences to depth 2.)

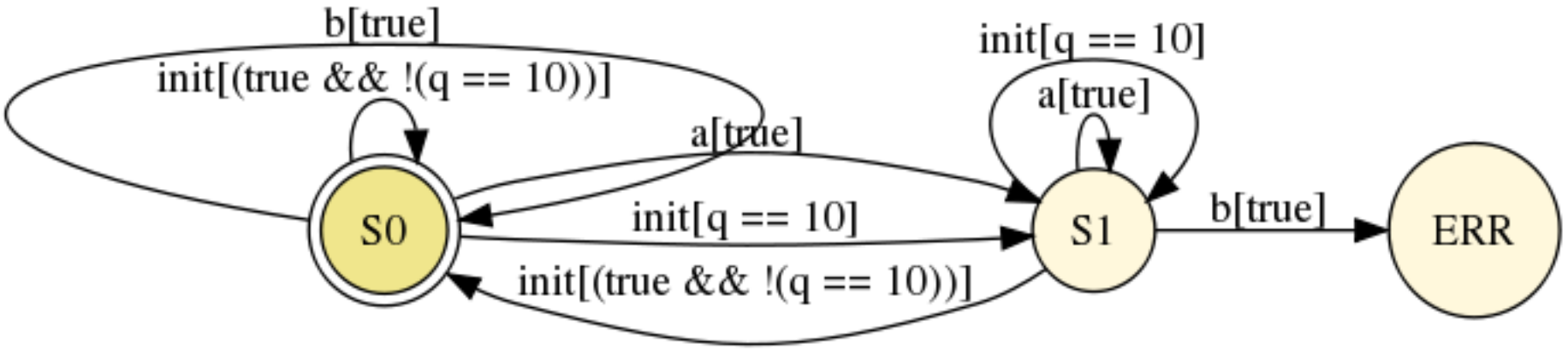
- We unroll unbounded loops in conjectures a bounded number of times

# Answering Equivalence Queries

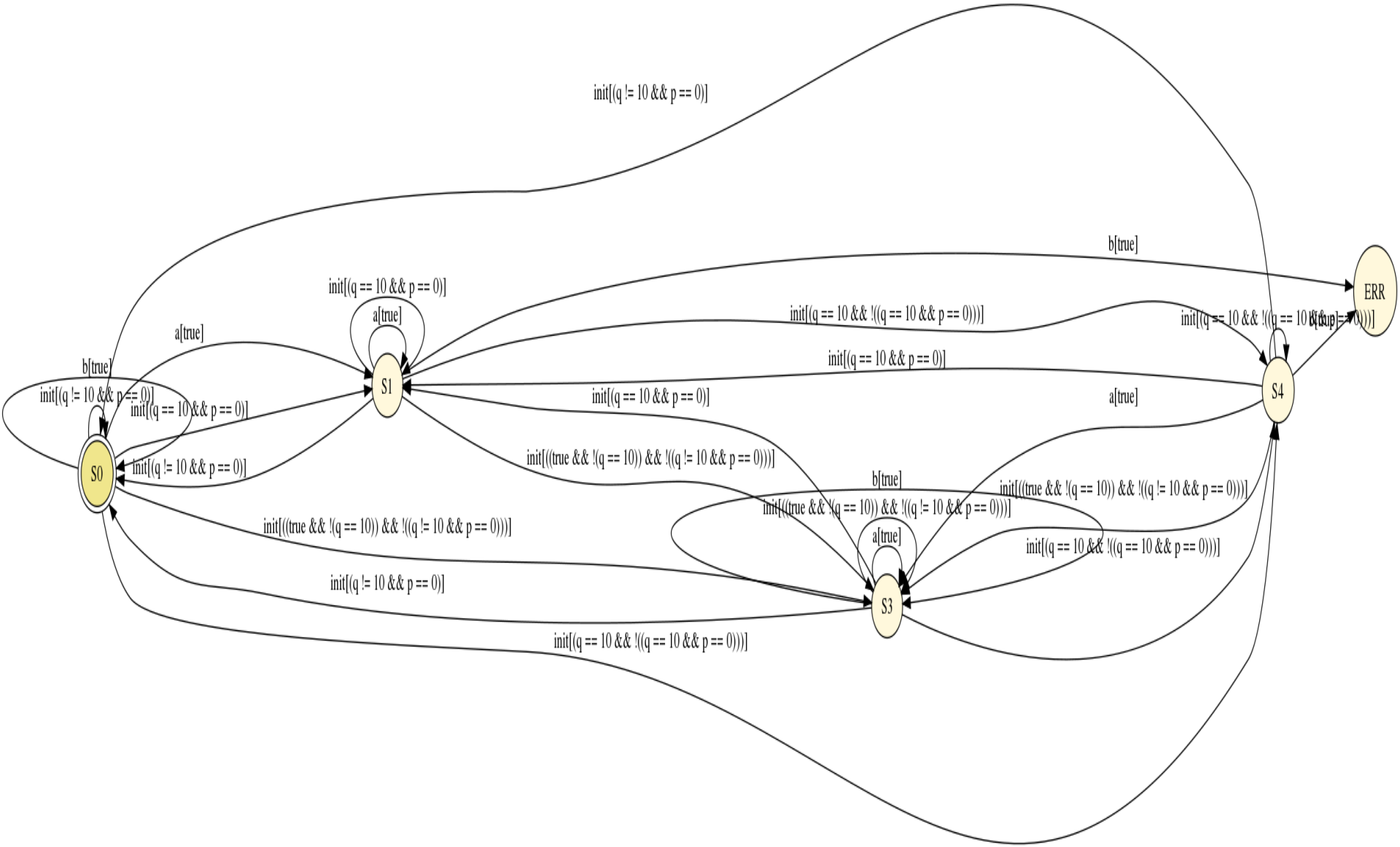
- Walk the conjectured automaton and extract,
  - all legal method sequences to a given depth k
  - all illegal and unknown method sequences
    - for each illegal or unknown sequence of depth n, extract the legal sequence of depth n - 1
- We then use membership queries to check the outcome of each sequence
  - If a sequence is misclassified by the learner, we have a counter-example for L*

# Running Example: Depth is 2

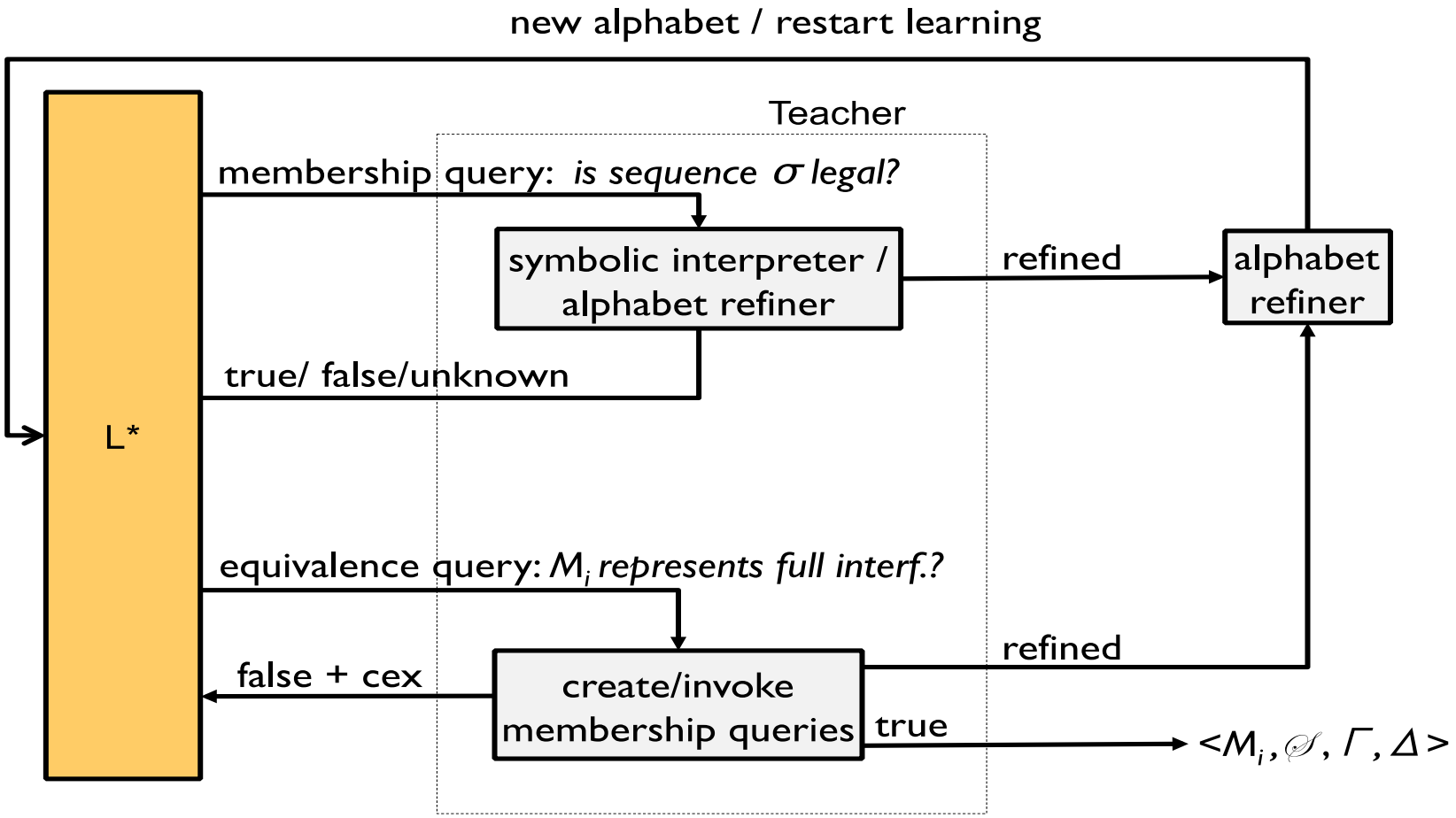
```
public class Example {
  private static int x = 0;
  private static int y = 0;

  public static void init(int p, int q) {
    x = p;
    y = q;
  }

  public static void a() {
    if (x == 0)
      y = 10;
    else
      y = 11;
  }
  public static void b() {
    if (y == 10)
      assert false;
  }
}
```



# Running Example: Depth is 3



# Architecture of PSYCO



# Next Time

- Checking concurrent programs