Symbolic execution

Pietro Braione

University of Milano-Bicocca

pietro.braione@unimib.it

Symbolic execution

- A technique for the analysis of the behavior of a program (Clarke, TSE 1976) (King, CACM 1976)
- Based on simulating a set of program executions
- Builds constraints that characterize
  - The inputs that execute program paths
  - The effects of the execution on the program state

Background

Background topics

- Feasible and infeasible program paths
- Constraints and satisfiability

Feasible and infeasible program paths

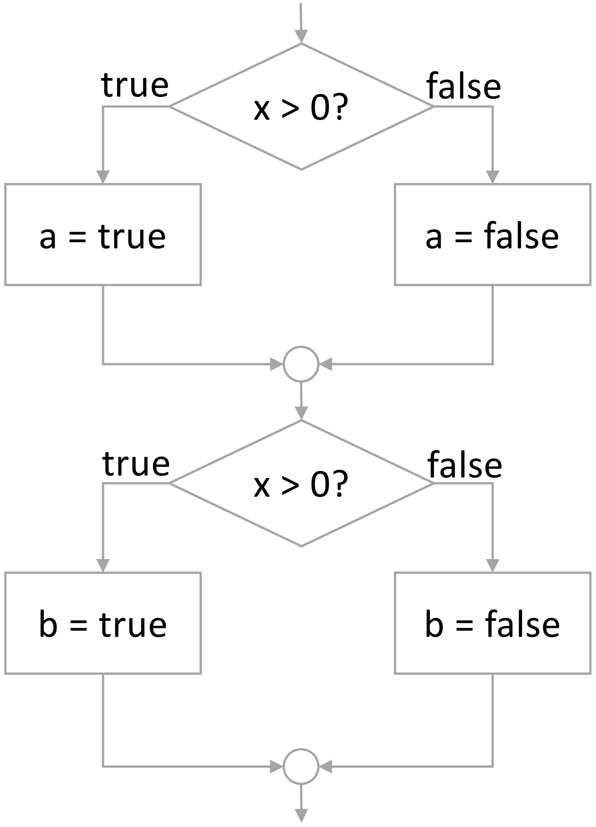
- A **program path** is a path in the interprocedural control flow graph of the program
- A program path is **feasible** iff there is at least one input that drives the execution of the program through it…
- ...otherwise the program path is **infeasible**

Feasible and infeasible paths: Example

```
public class IfExample {
    boolean a, b;
    public void m(int x) {
        if (x > 0) {
            a = true;
        } else {
            a = false;
        }
        if (x > 0) {
            b = true;
        } else {
            b = false;
        }
    }
}
```

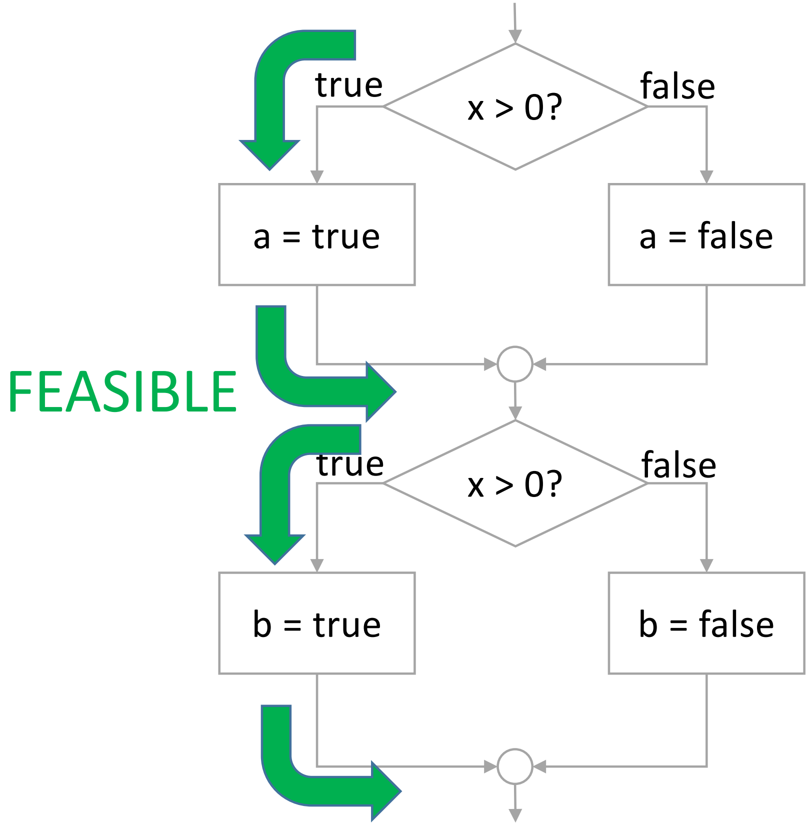
Feasible and infeasible paths: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



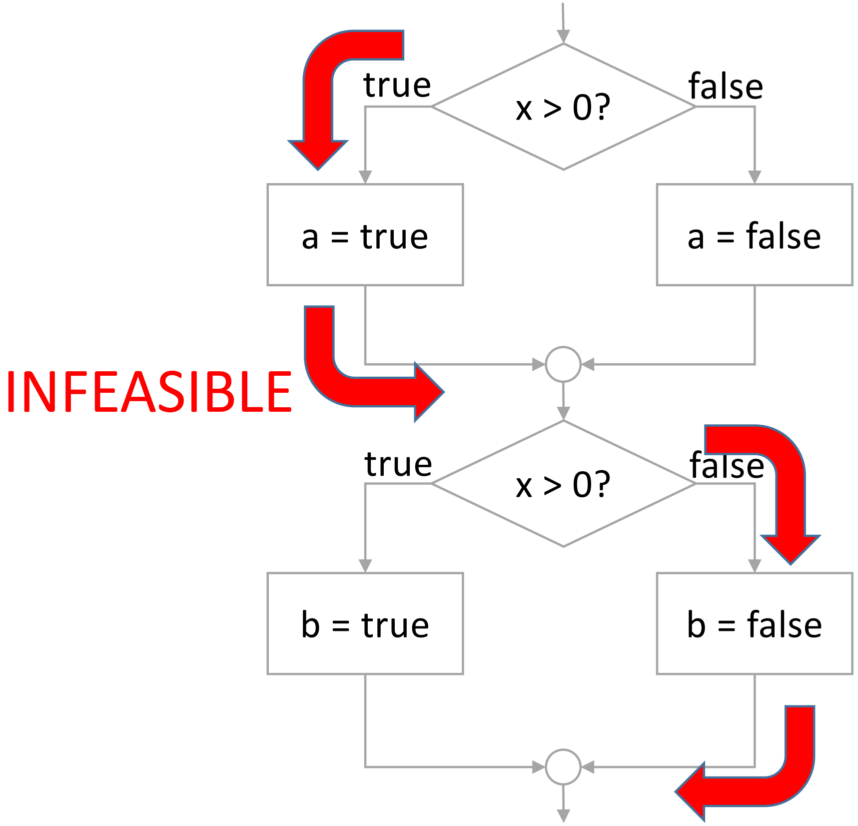
Feasible and infeasible paths: Example

```
public class IfExample {
    boolean a, b;
    public void m(int x) {
        if (x > 0) {
            a = true;
        } else {
            a = false;
        }
        if (x > 0) {
            b = true;
        } else {
            b = false;
        }
    }
}
```



Feasible and infeasible paths: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



Feasible and infeasible paths: Comments

- A program may have a finite or infinite number of feasible paths
    - If a program has no loops/recursion, it has a finite number of feasible paths
    - If a program has loops or recursion, it may have an infinite number of feasible paths
- A program may also have infinite-length feasible paths: This happens if the program diverges for some inputs
- Infeasible paths do not imply dead code, but it is true the opposite (dead code implies infeasible paths)
- In all real software a very large number of paths is infeasible

Constraints

- A **constraint** is a boolean predicate over **(free) variables**
- A **solution** for a constraint is an assignment to its free variables that evaluates the constraint to true
- A constraint that has solutions is said to be **satisfiable**, a constraint that is not satisfiable is said to be **contradictory**
- Example:
  - Let us consider the constraint X > Y && X + Y < 10
  - {X == 3, Y == 2} is a solution
  - {X == 6, Y == 5} is not a solution

Decision procedures and constraint solvers

- A **decision procedure** is an algorithm that can decide whether a constraint is satisfiable or not
- A **constraint solver**, in addition, emits a solution if the constraint is satisfiable
- Satisfiability/constraint solving is undecidable in general
- However it can become decidable if we restrict the set of possible constraints to suitable subsets (e.g. linear constraints)
- Nowadays effective solvers for some standard classes of constraint types exist (SMT solvers: Z3, CVC4, MathSAT…)

Symbolic execution of numeric programs

What is symbolic execution?

- Is the simulation of the effect of the execution of a program
- With **symbols** as inputs
- Along a program path

Symbolic inputs

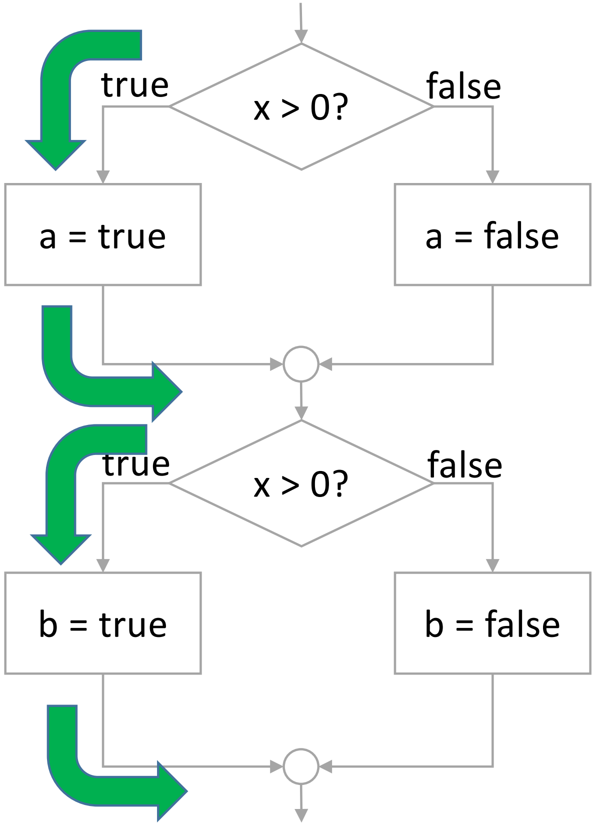
- To execute a program (e.g., when we test it) we must provide input values
- For example, we can invoke the example method m with x = 1
- A symbolic input value stands for the possible values that the input might assume when the program is executed
- For example, symbolic execution would simulate execution of m with x = P, where P is a symbol standing for an arbitrary int

Executing with symbolic inputs

- Symbolic execution keeps track of
    - The **(symbolic) state** of the program, i.e., the values of all the program variables
    - The **path condition**, i.e., a constraint on the symbolic inputs that ensures that the program execution goes through the selected path
- Assignment statements change the state of the program
- Conditionals update the path condition

Symbolic execution: Example

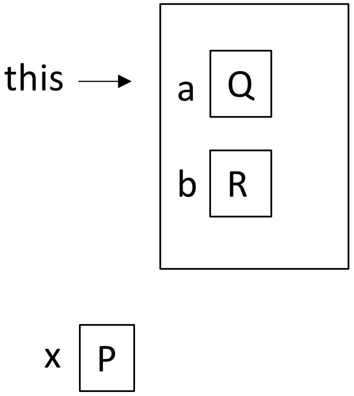
```
public class IfExample {
    boolean a, b;
    public void m(int x) {
        if (x > 0) {
            a = true;
        } else {
            a = false;
        }
        if (x > 0) {
            b = true;
        } else {
            b = false;
        }
    }
}
```



Symbolic execution: Example

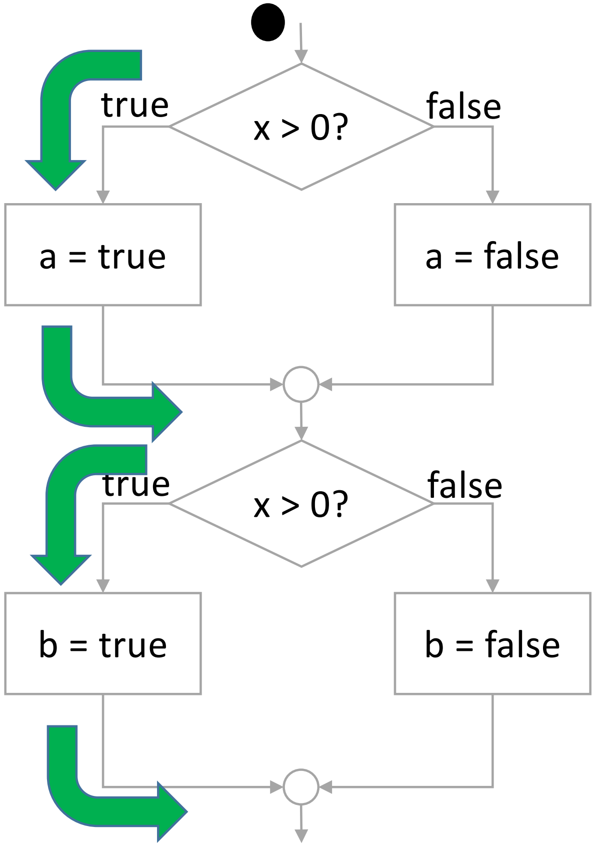
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

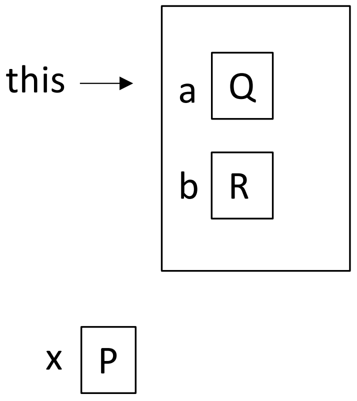
(true)



Symbolic execution: Example

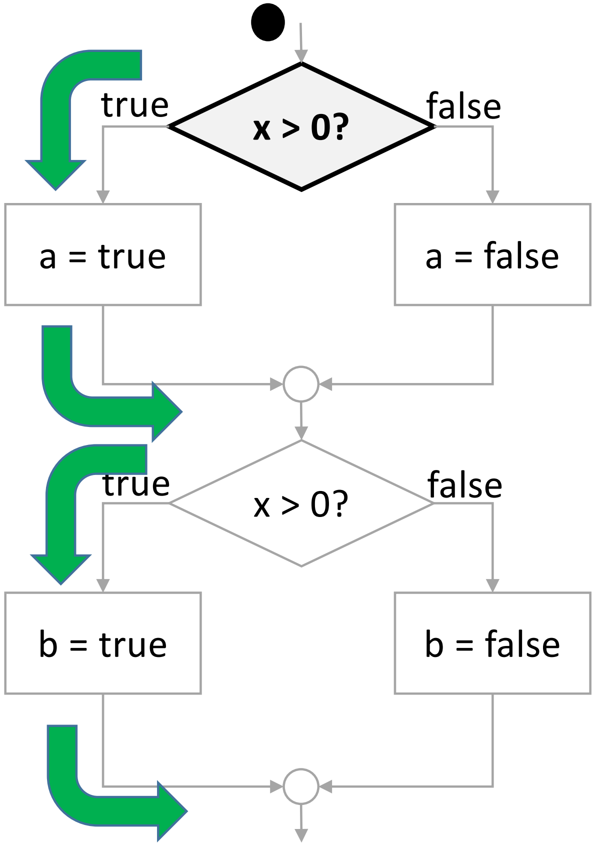
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state


this →
a Q
b R
x P

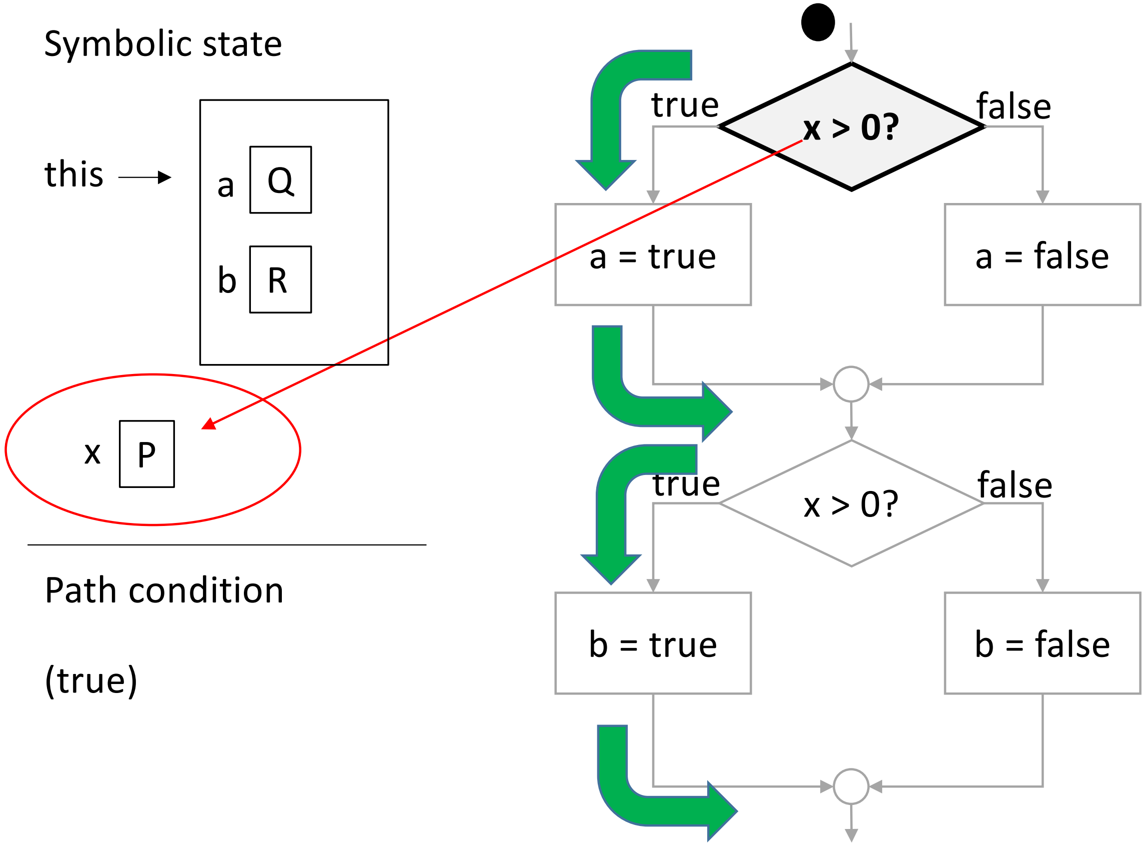
Path condition

(true)



Symbolic execution: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



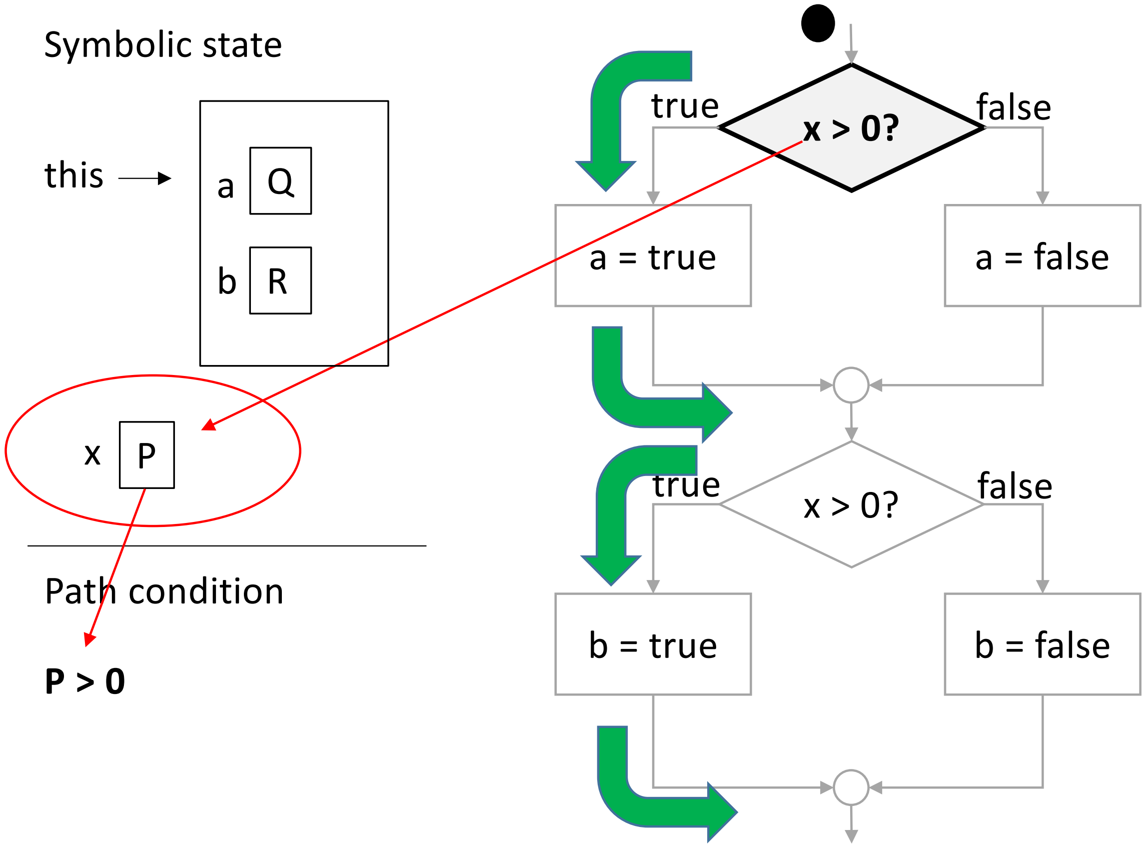
Symbolic state

Path condition

(true)

Symbolic execution: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



Symbolic state

this

a Q

b R

x P

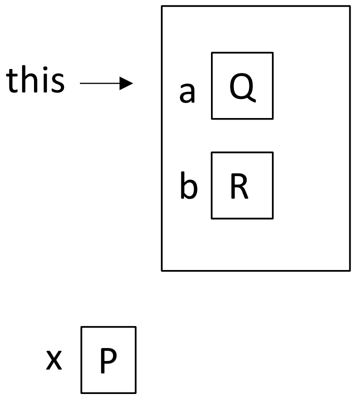
Path condition

P > 0

Symbolic execution: Example

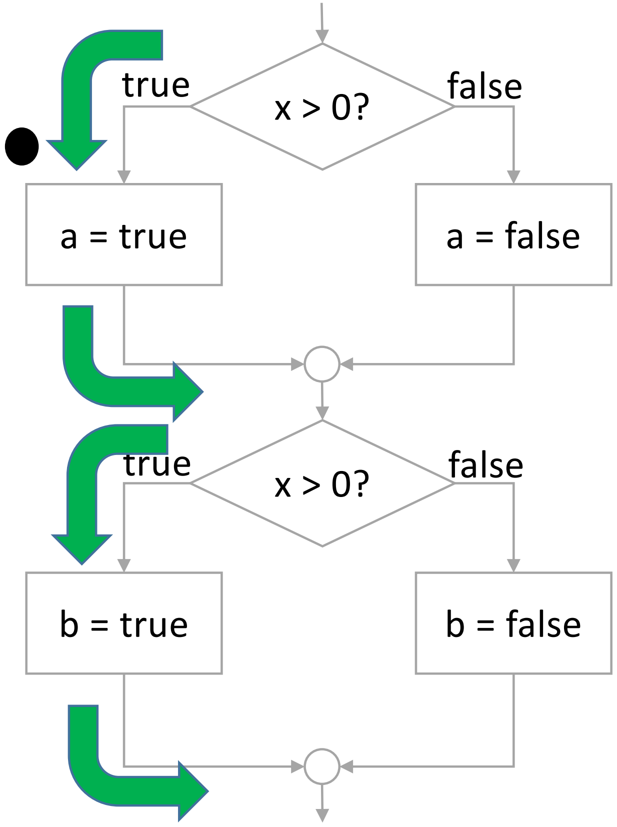
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

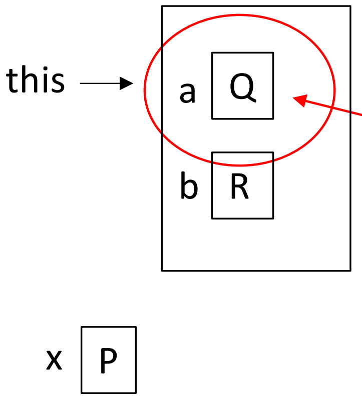
P > 0



Symbolic execution: Example

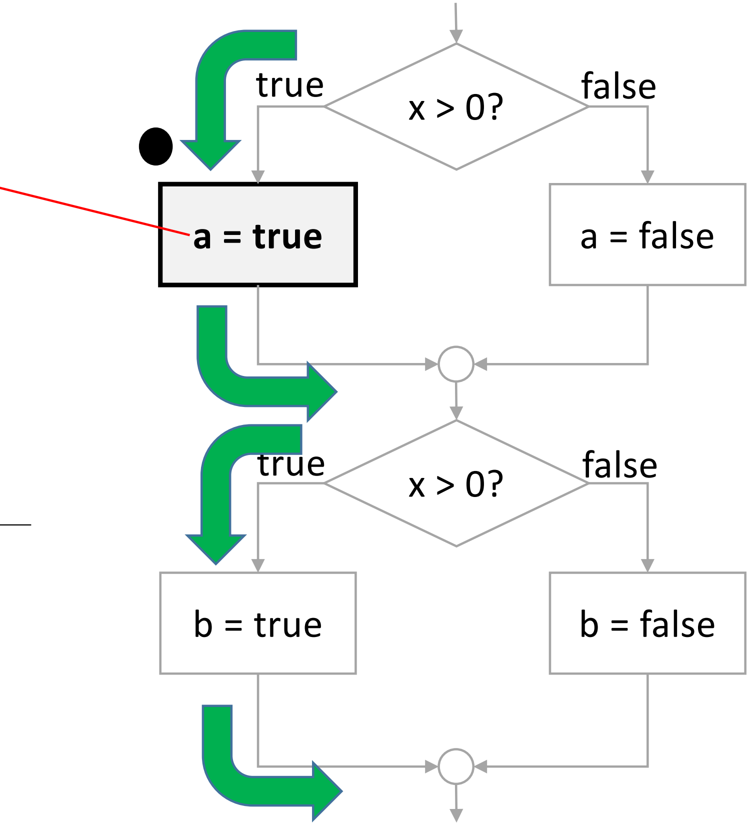
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

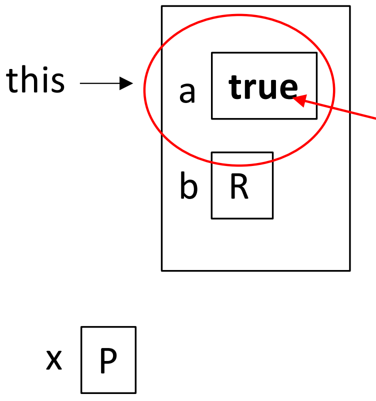
P > 0



Symbolic execution: Example

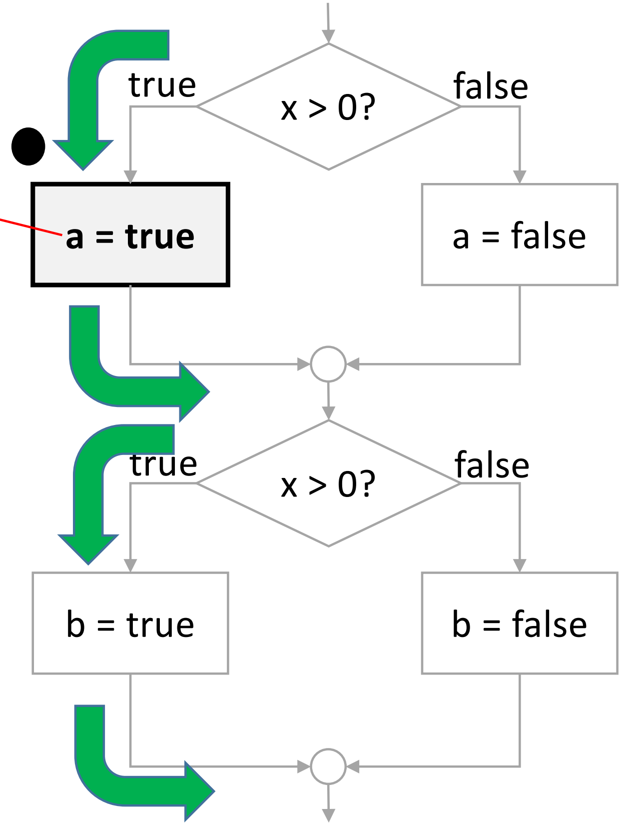
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

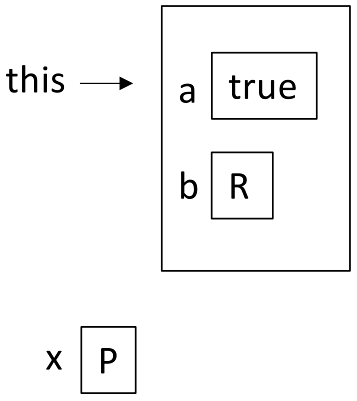
P > 0



Symbolic execution: Example

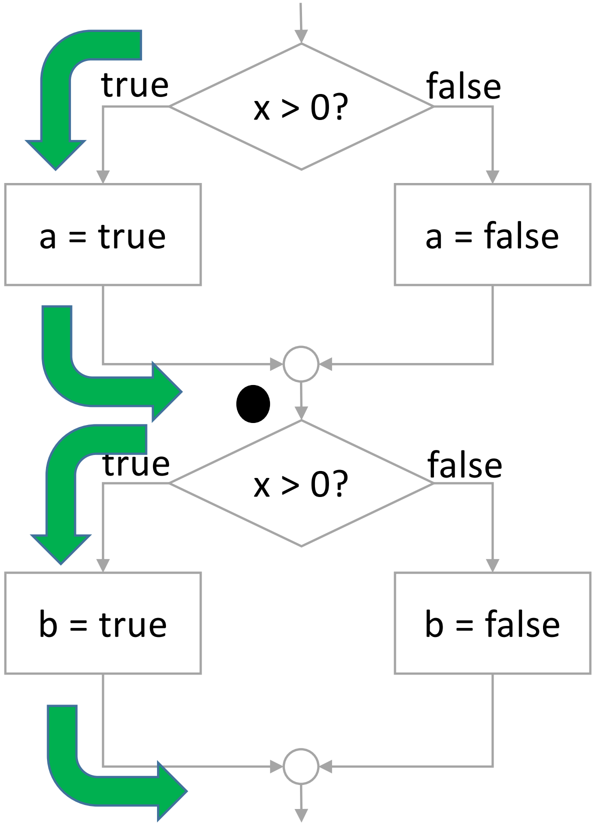
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state


this
a
true
b
R
x
P

Path condition

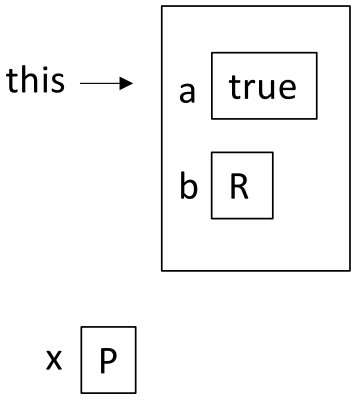
P > 0



Symbolic execution: Example

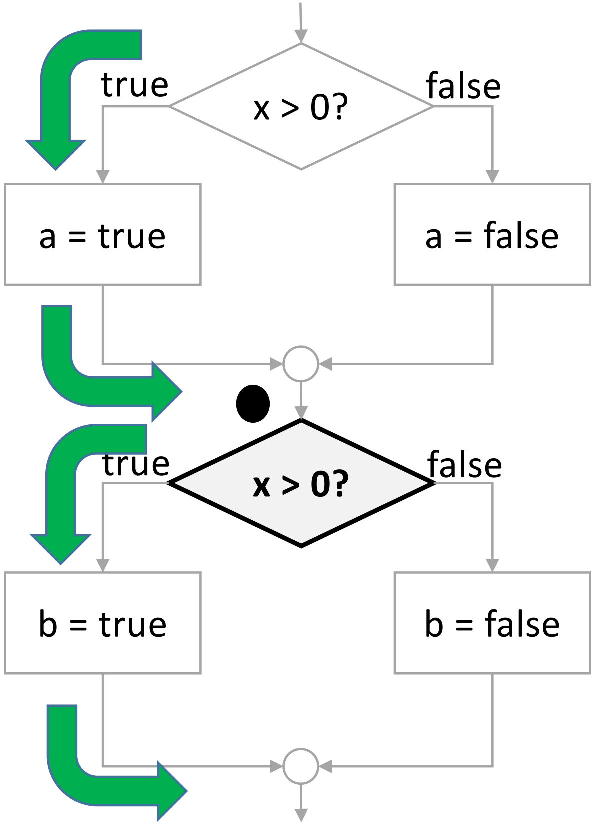
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



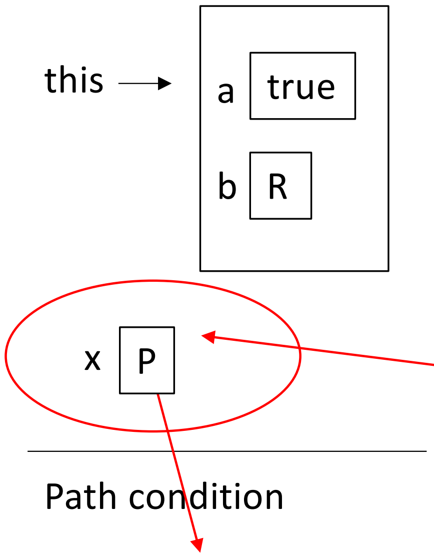
Path condition

P > 0



Symbolic execution: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



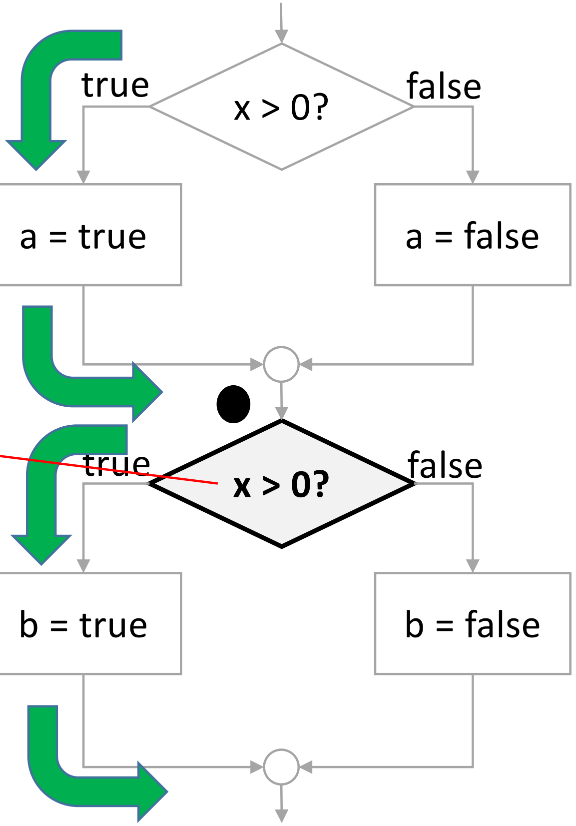
Symbolic state

this → a: true; b: R

x: P

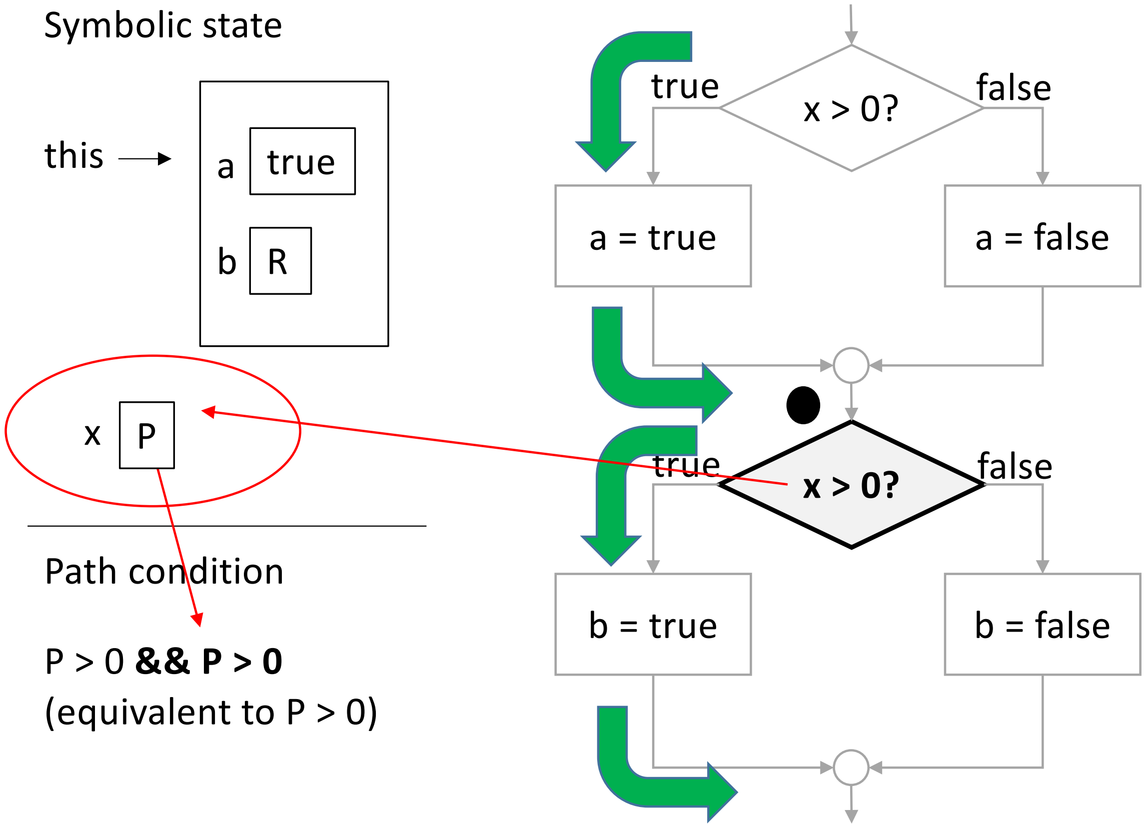
Path condition

P > 0 **&& P > 0**



Symbolic execution: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



Symbolic state

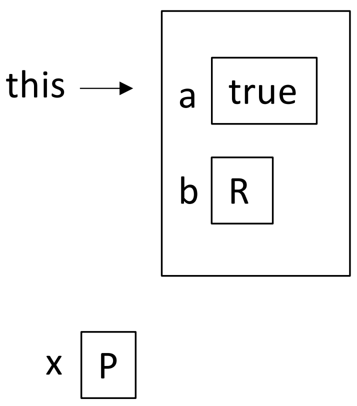
Path condition

P > 0 **&& P > 0**
(equivalent to P > 0)

Symbolic execution: Example

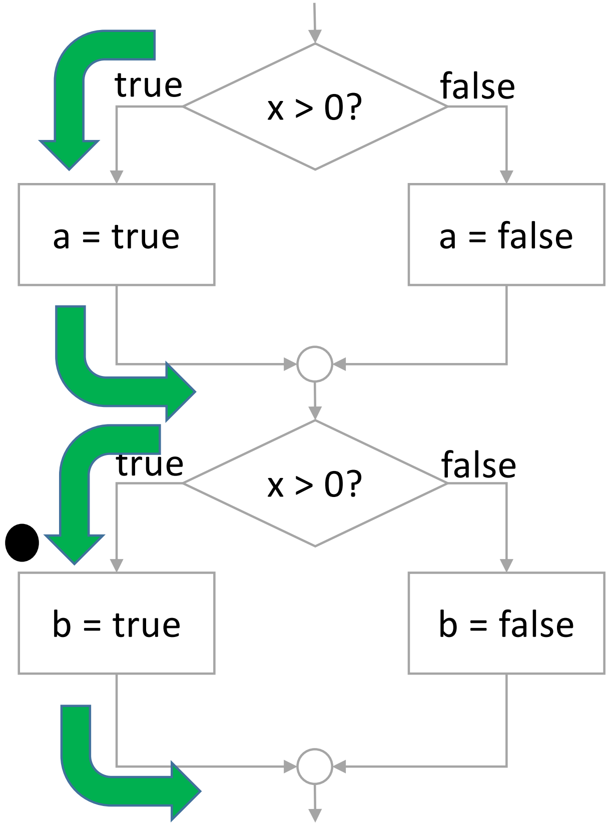
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

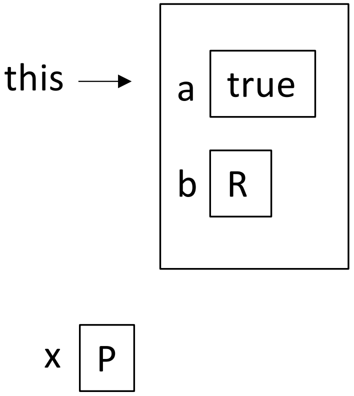
P > 0



Symbolic execution: Example

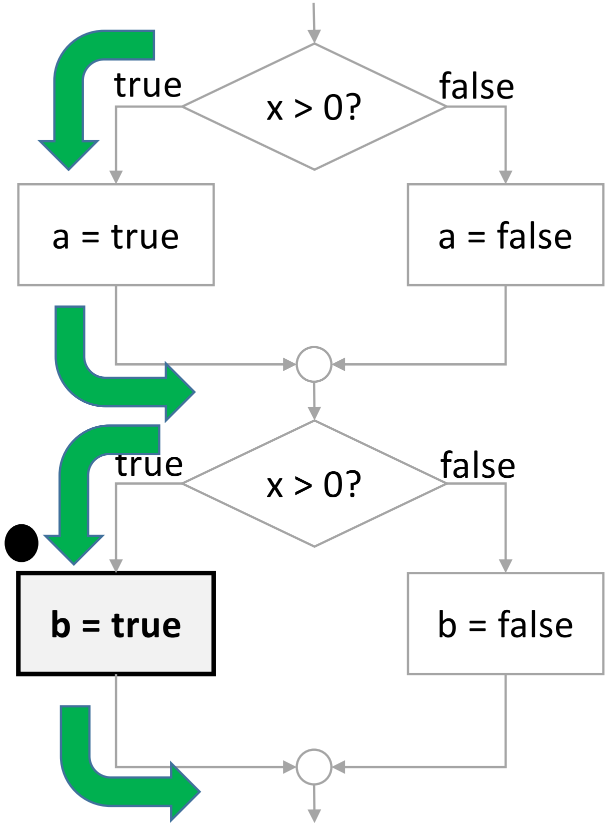
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

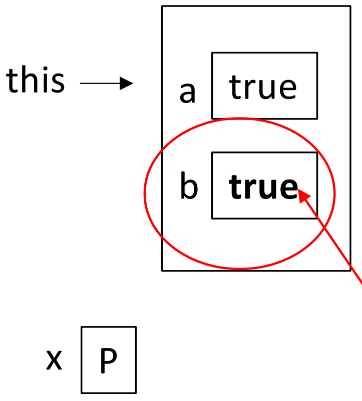
P > 0



Symbolic execution: Example

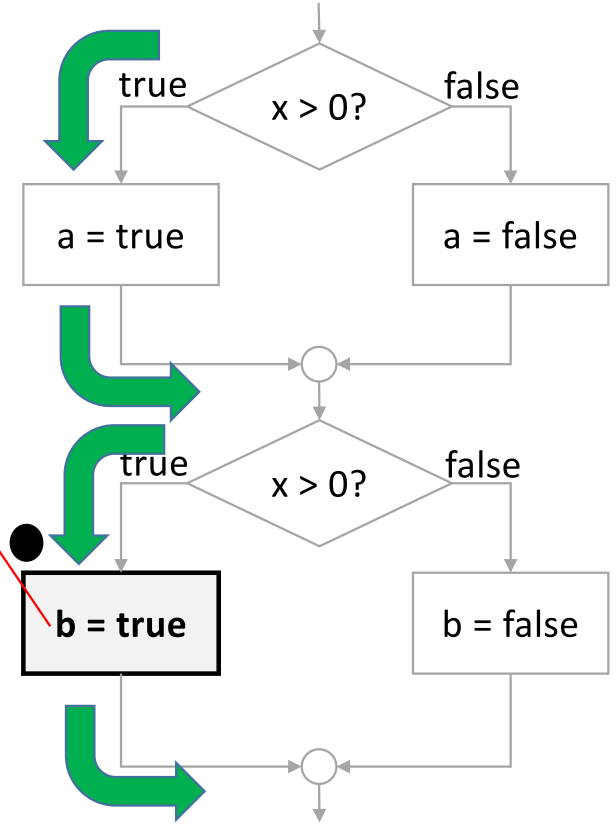
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

$P > 0$



Symbolic execution: Example

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

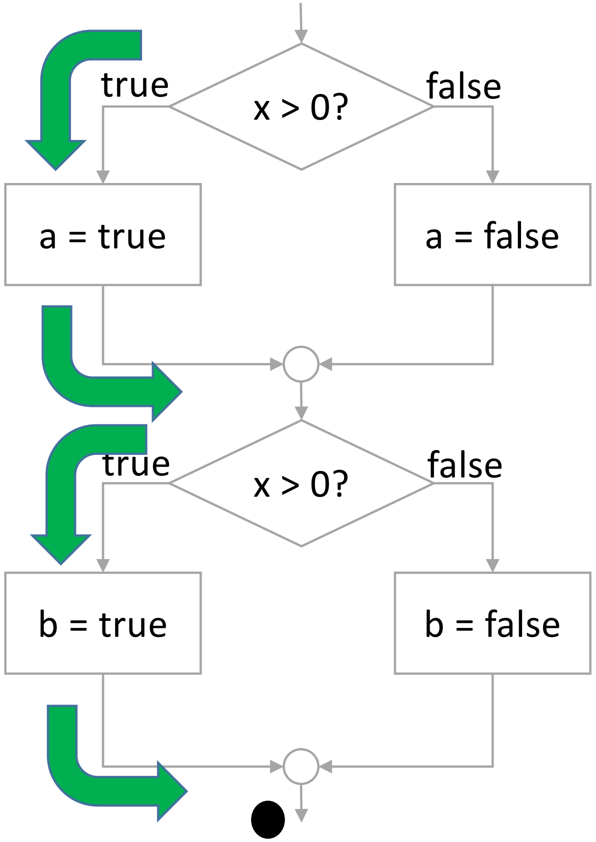
Symbolic state

this → a true, b true

x P

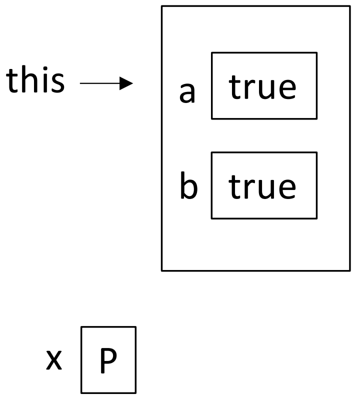
Path condition

P > 0



Symbolic execution: Example

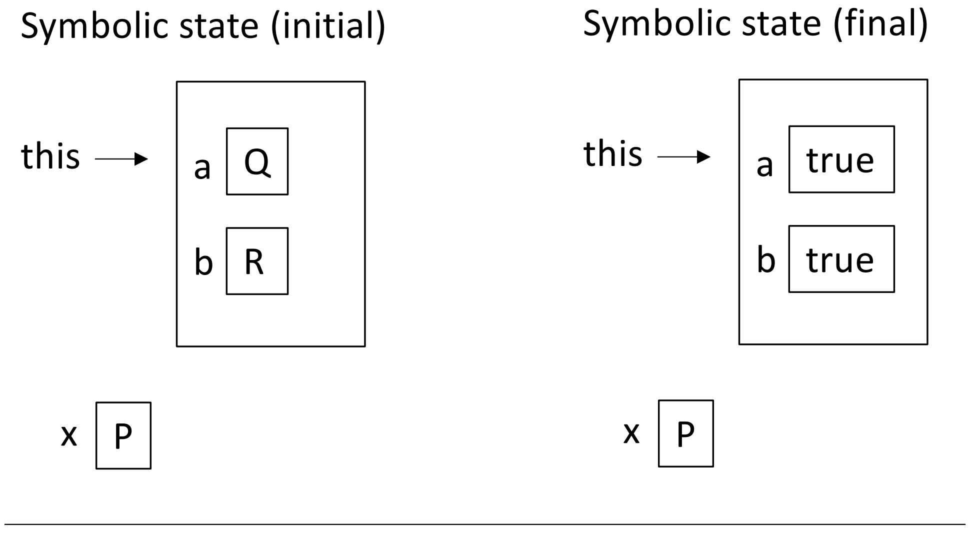
Symbolic state (final)



Path condition

P > 0

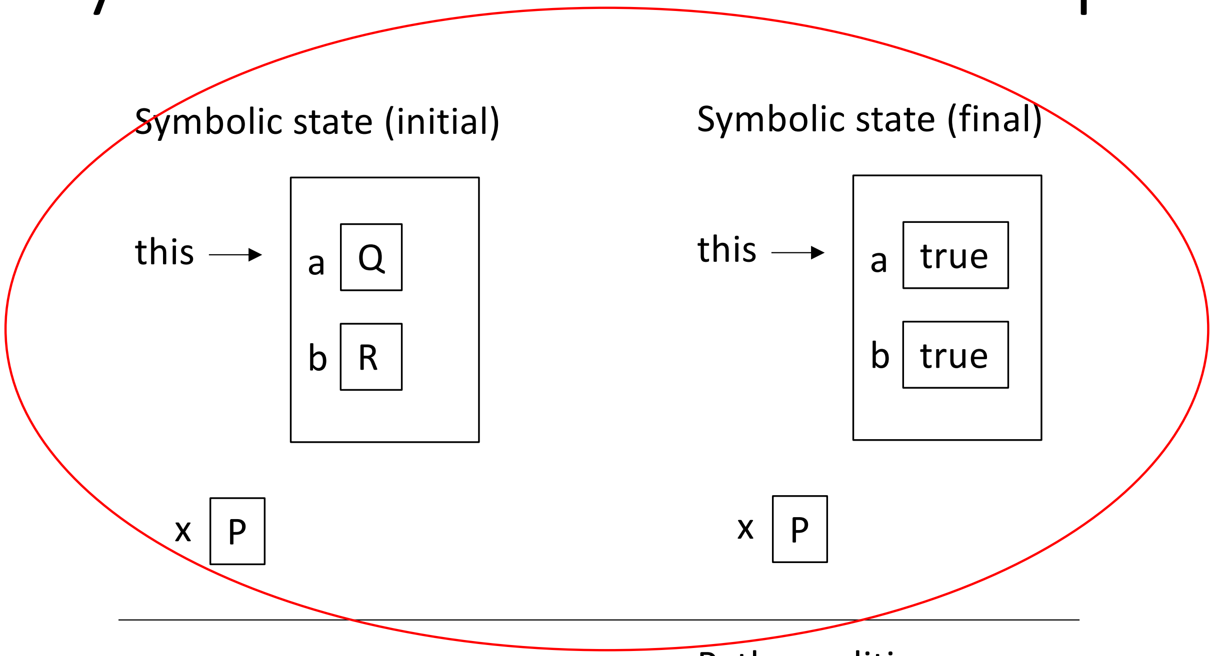
Symbolic execution: Example



Path condition

$P > 0$

Symbolic execution: Example



Symbolic state (initial)

this → a Q, b R

x P

Symbolic state (final)

this → a true, b true

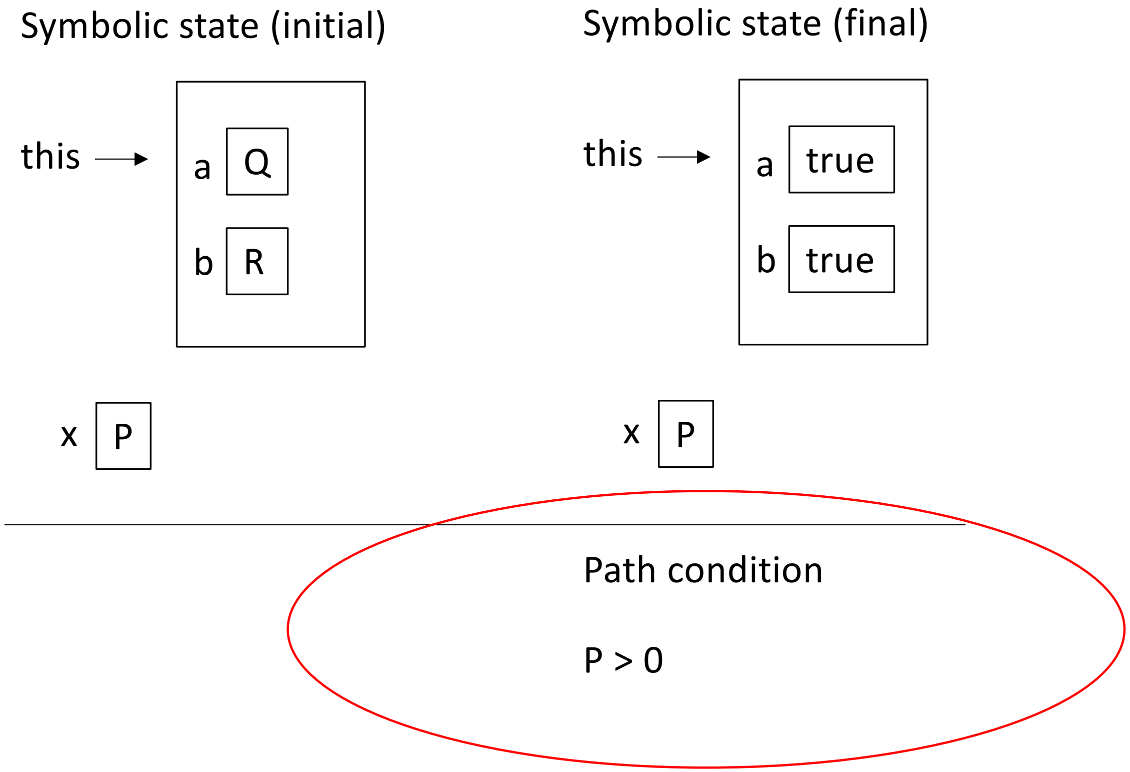
x P

Path condition

P > 0

- The values of this.a and this.b are set to true
- The value of x is unchanged
- The initial values of this.a and this.b are lost

Symbolic execution: Example



Symbolic state (initial)

this → a Q, b R

x P

Symbolic state (final)

this → a true, b true

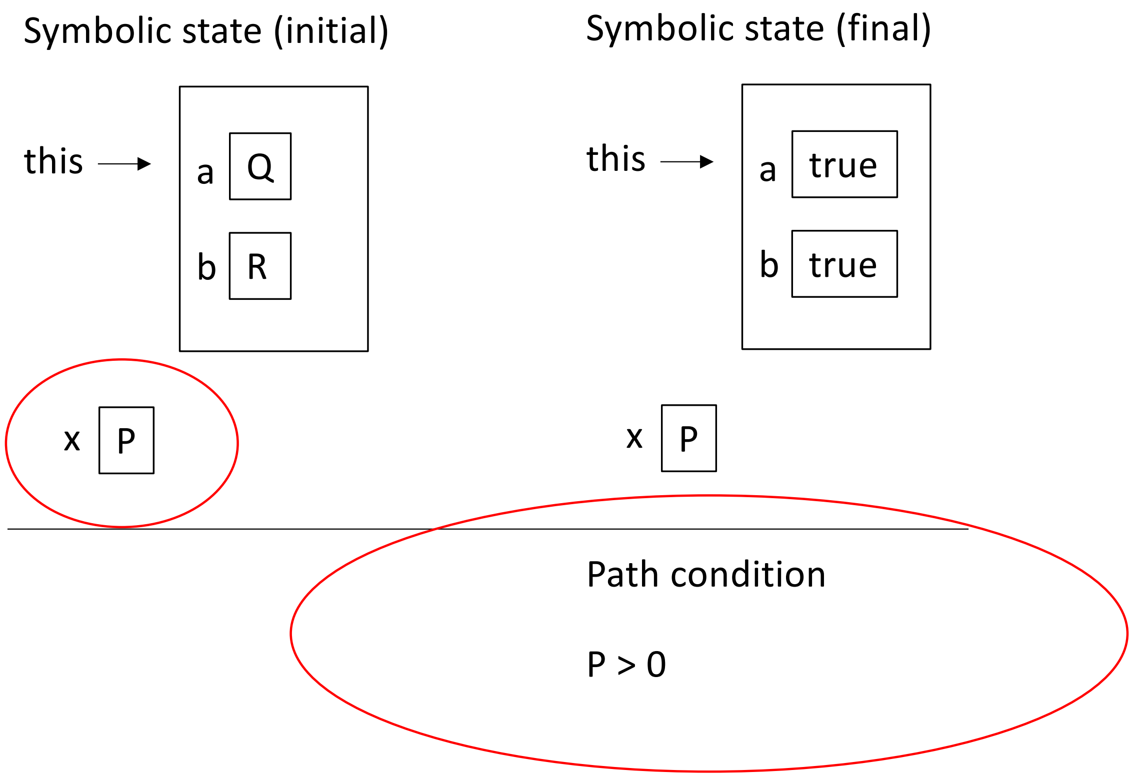
x P

Path condition

P > 0

- The program execution follows the green path iff P > 0

Symbolic execution: Example



Symbolic state (initial)

this →

a Q

b R

x P

Symbolic state (final)

this →

a true

b true

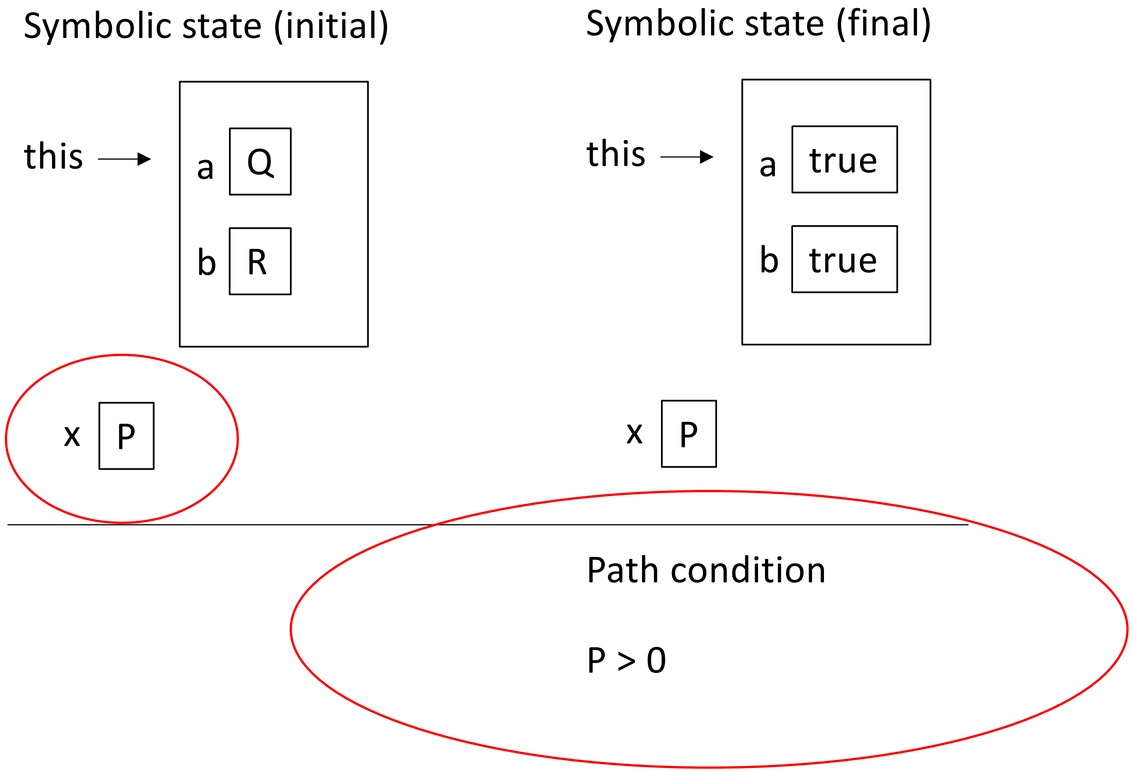
x P

Path condition

P > 0

- The program execution follows the green path iff P > 0
- But P is the initial value of x, thus…

Symbolic execution: Example



Symbolic state (initial)

this → a Q, b R

x P

Symbolic state (final)

this → a true, b true

x P

Path condition

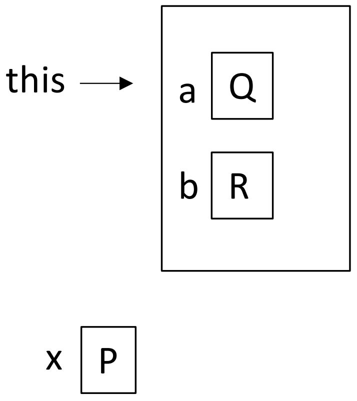
P > 0

- The program execution follows the green path iff P > 0
- But P is the initial value of x, thus…
- …the program execution follows the green path iff we pass a positive x input

Symbolic execution: Infeasible path

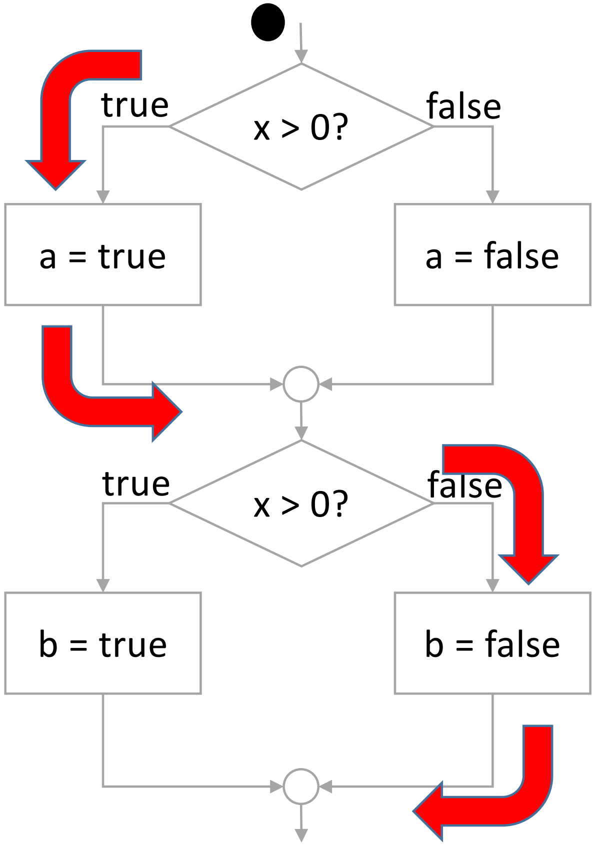
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



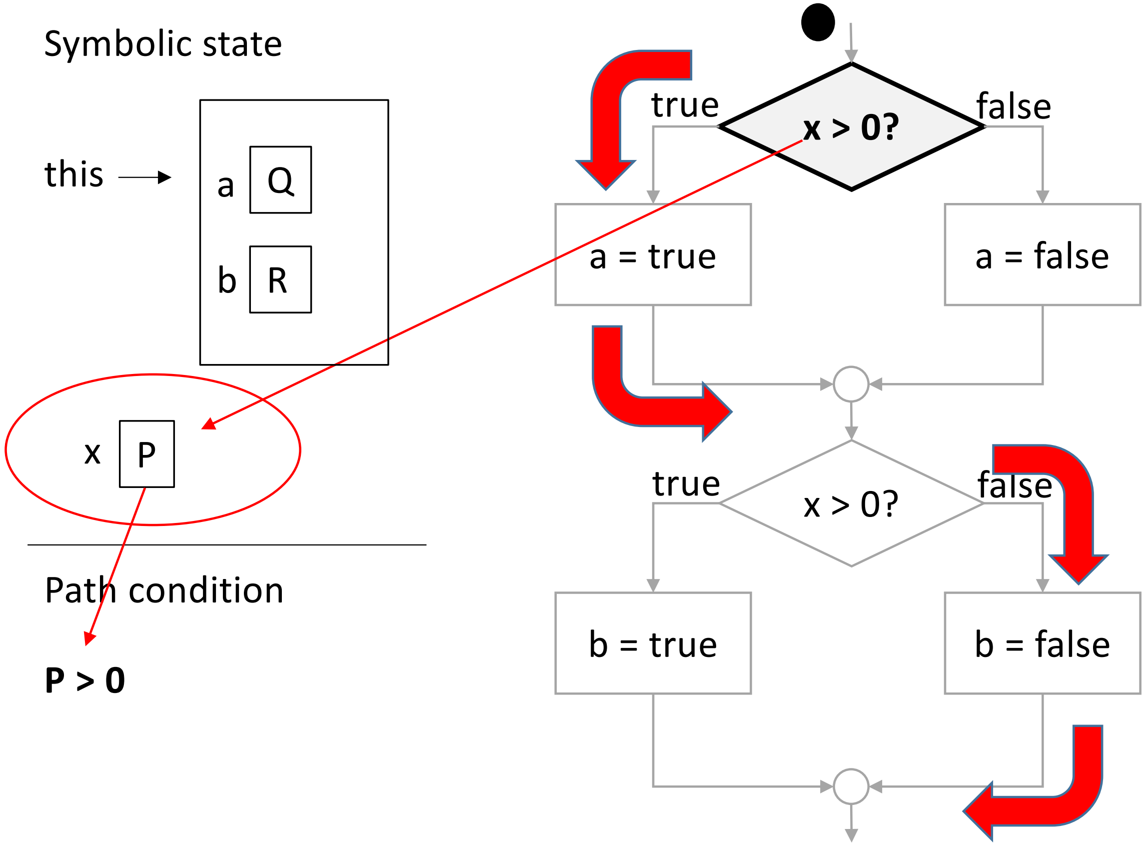
Path condition

(true)



Symbolic execution: Infeasible path

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```


Symbolic state

this

a Q

b R

x P

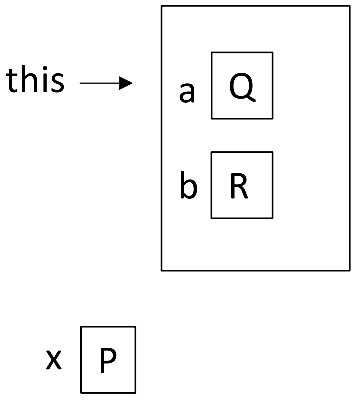
Path condition

P > 0

Symbolic execution: Infeasible path

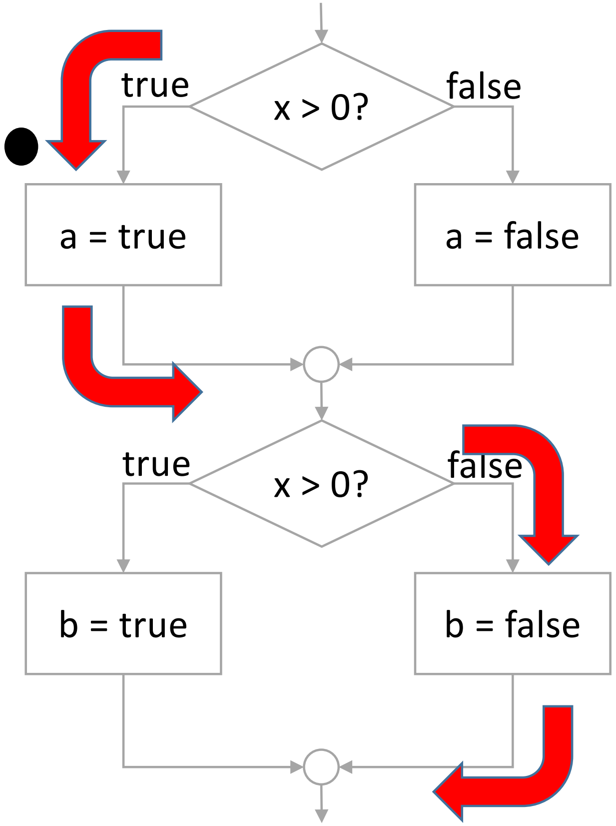
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

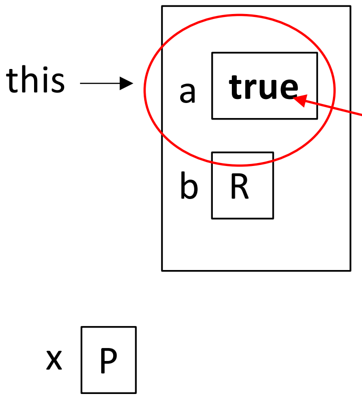
P > 0



Symbolic execution: Infeasible path

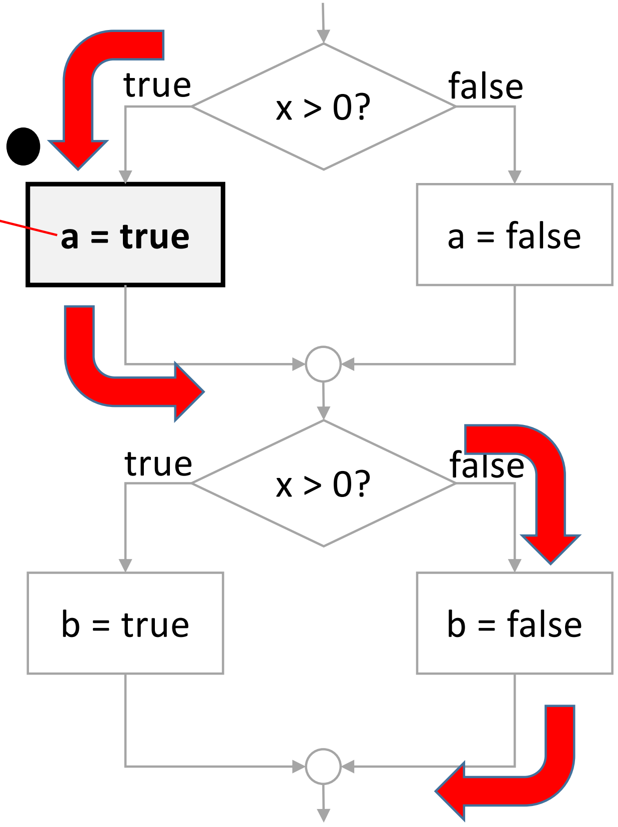
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



Path condition

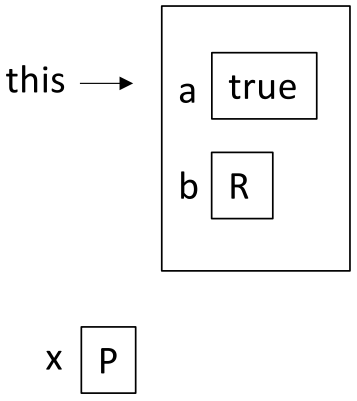
$P > 0$



Symbolic execution: Infeasible path

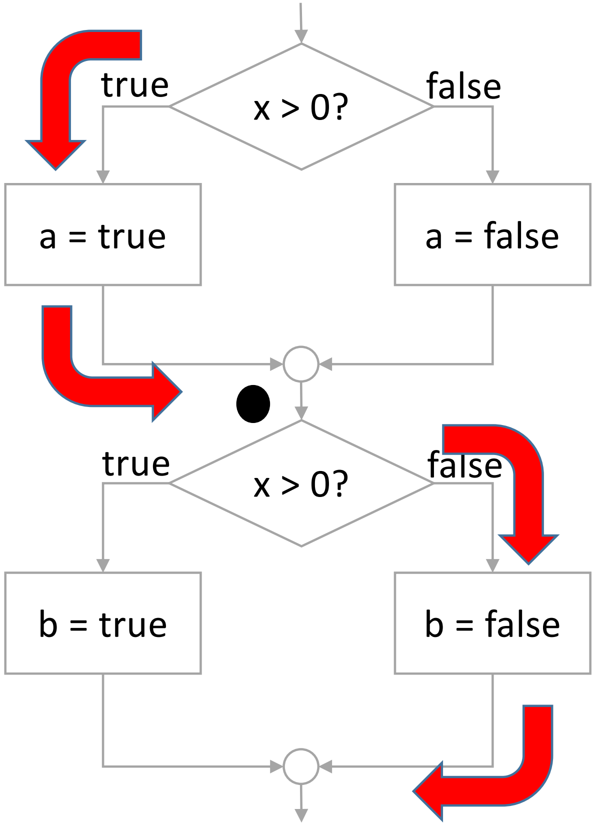
```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```

Symbolic state



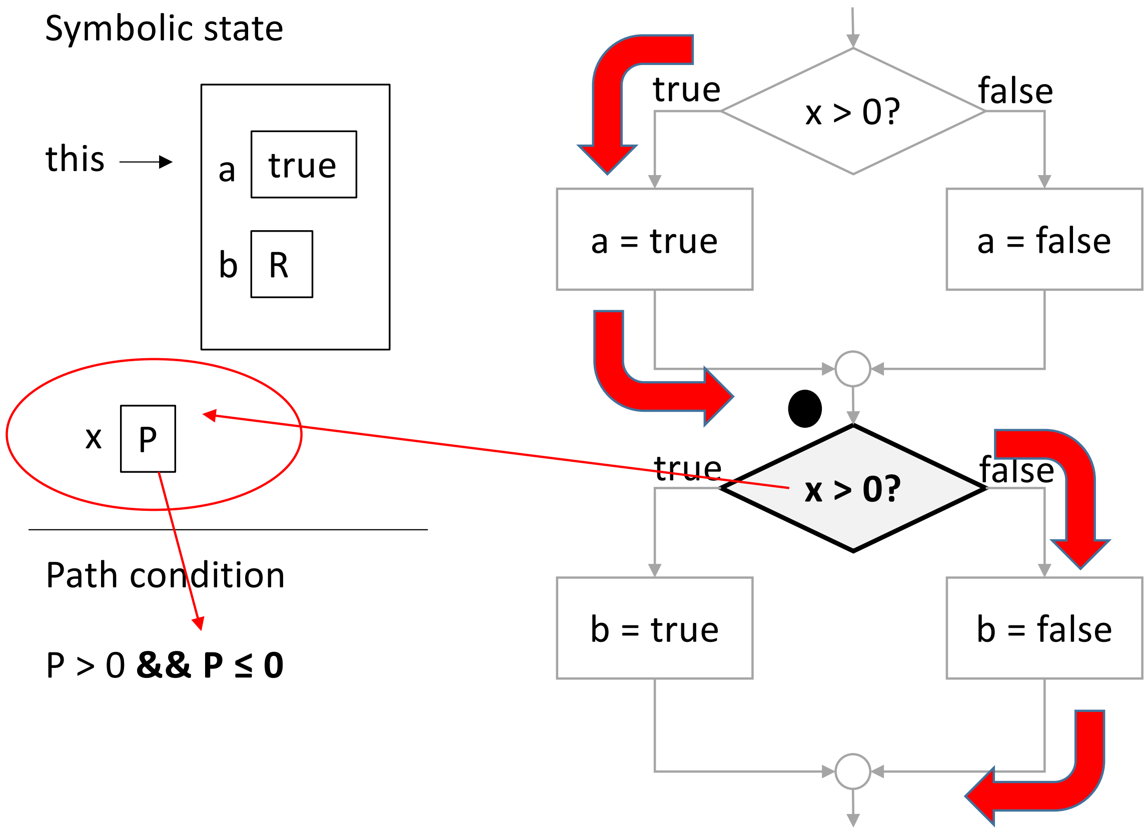
Path condition

P > 0



Symbolic execution: Infeasible path

```
public class IfExample {
   boolean a, b;
   public void m(int x) {
      if (x > 0) {
         a = true;
      } else {
         a = false;
      }
      if (x > 0) {
         b = true;
      } else {
         b = false;
      }
   }
}
```



Path condition

P > 0 **&& P ≤ 0**

Symbolic execution: Infeasible path

Path condition

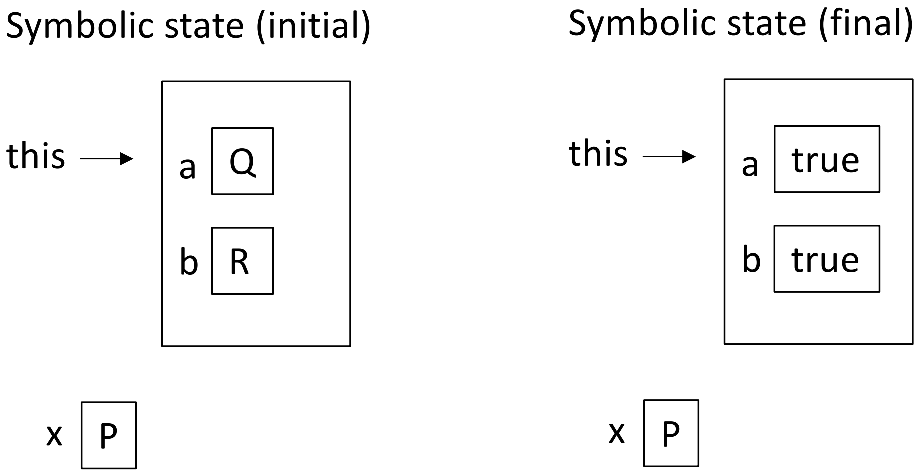
$P > 0 \ \&\& \ P \leq 0$

- The path condition is contradictory
- Therefore no value for P drives the program execution through the red path

Path feasibility and path conditions

- A path is feasible iff its path condition is satisfiable, or equivalently
- A path is infeasible iff its path condition is contradictory
- A solution to a path condition is an assignment to program inputs (i.e., a test case) that drives the program execution through the path

Solving the path condition



Symbolic state (initial)

this → a Q, b R

x P

Symbolic state (final)

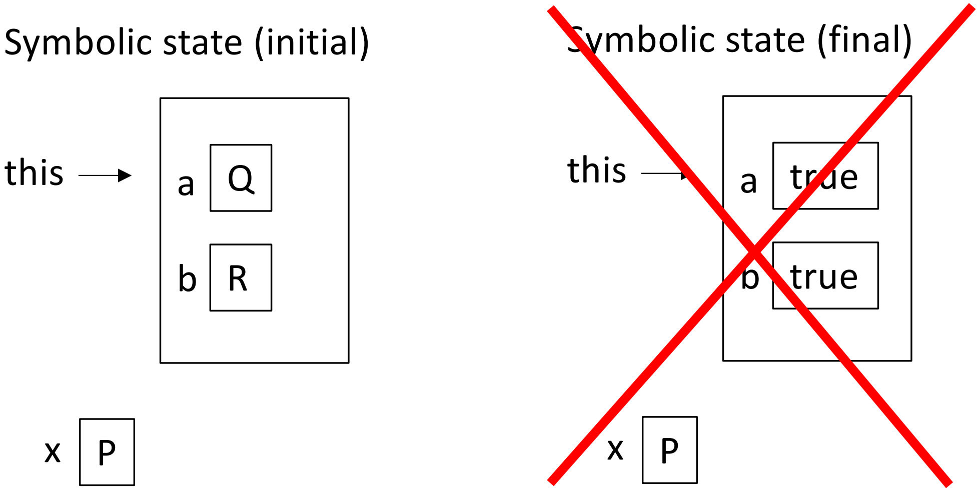
this → a true, b true

x P

Path condition

P > 0

Solving the path condition



Symbolic state (initial)

this →

a Q

b R

x P

~~Symbolic state (final)~~

~~this →~~

~~a true~~

~~b true~~

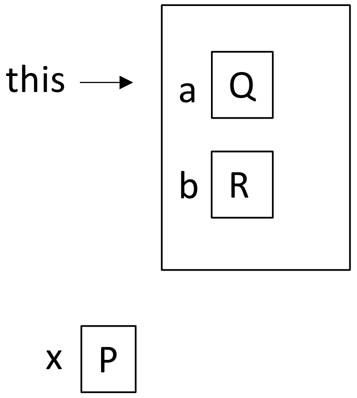
~~x P~~

Path condition

$P > 0$

Solving the path condition

Symbolic state (initial)



this →

a Q

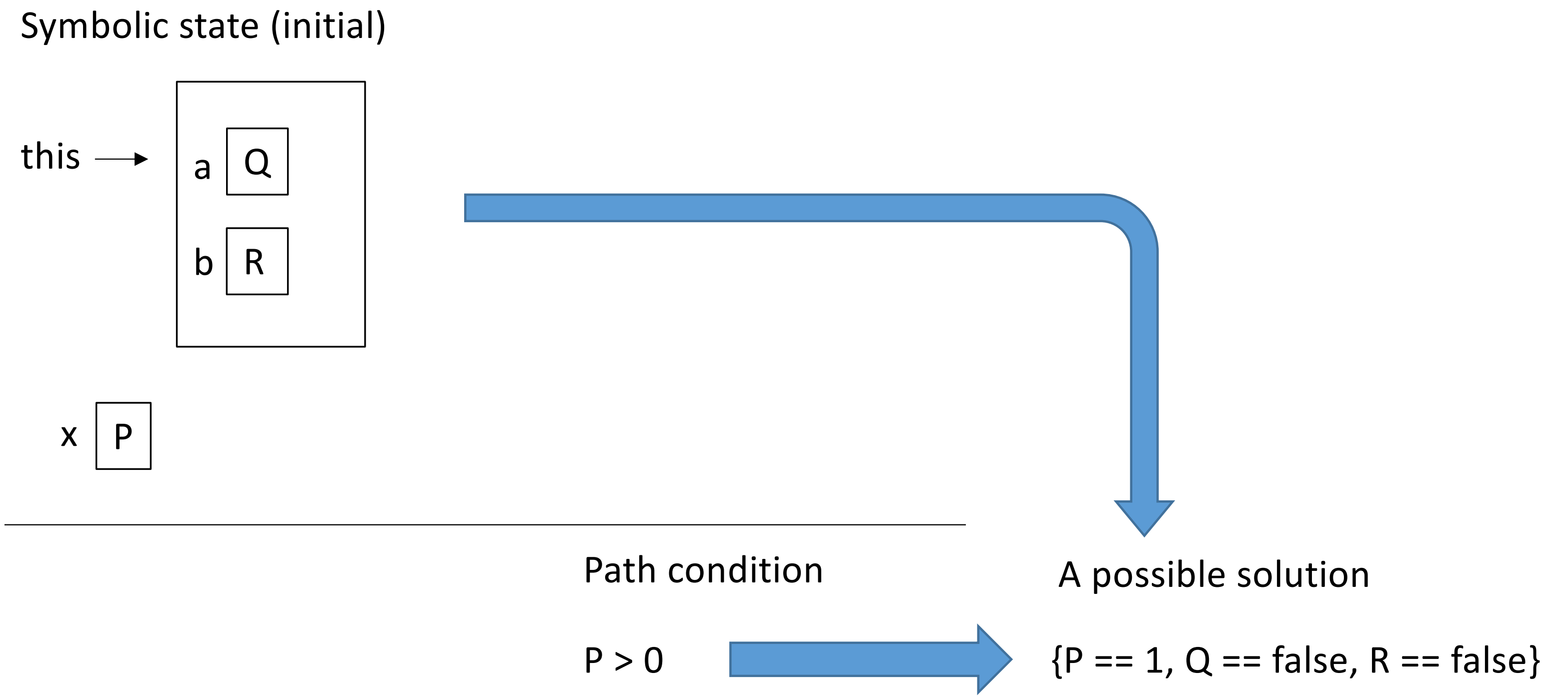
b R

x P

Path condition

P > 0

Solving the path condition



Symbolic state (initial)

Path condition

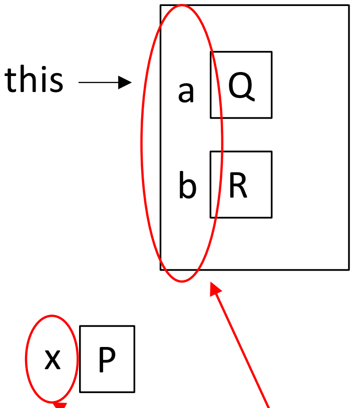
P > 0

A possible solution

{P == 1, Q == false, R == false}

Solving the path condition

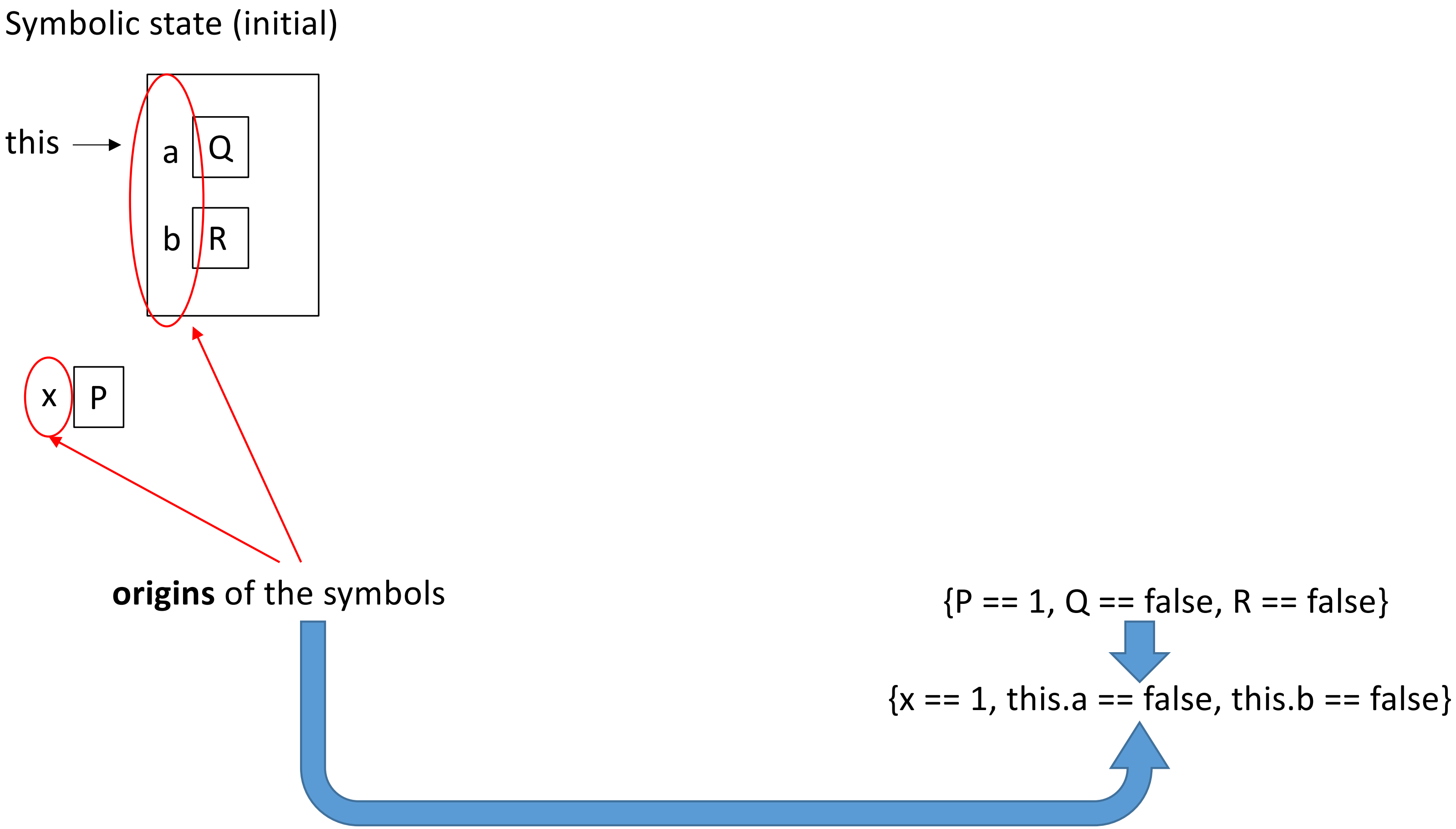
Symbolic state (initial)



origins of the symbols

{P == 1, Q == false, R == false}

Solving the path condition



Symbolic state (initial)

this

a Q

b R

x P

origins of the symbols

{P == 1, Q == false, R == false}

{x == 1, this.a == false, this.b == false}

Solving the path condition

Test case: {x == 1, this.a == false, this.b == false}

All-paths symbolic execution

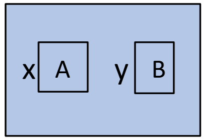
- We can summarize how a program behaves if we perform **all-paths** symbolic execution
- Means executing symbolically all the program paths
- Typically performed in a depth-first fashion:
  - Start executing
  - Arrived at a branch, take an arbitrary direction
  - Abandon the path if its path condition becomes unsat (infeasible path)
  - As a path is fully explored, backtrack to a previous branch and explore the other direction
- The result is a **symbolic execution tree** of states/path conditions

All-paths symbolic execution: Example

```
➡  int x, y;
   if (x > y) {
      x = x + y;
      y = x – y;
      x = x – y;
      if (x > y) {
         THROW_EXCEPTION;
      }
   }
```

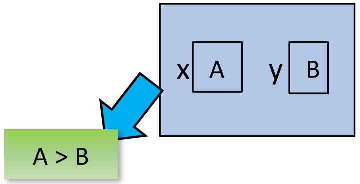
All-paths symbolic execution: Example

```
   int x, y;
➡ if (x > y) {
       x = x + y;
       y = x – y;
       x = x – y;
       if (x > y) {
           THROW_EXCEPTION;
       }
   }
```



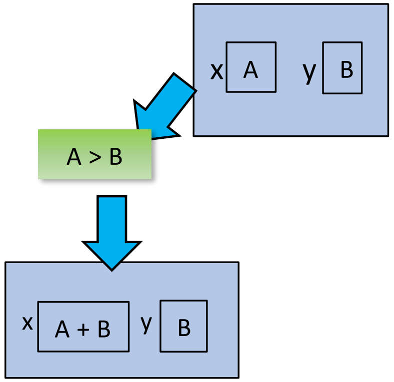
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
    if (x > y) {
        THROW_EXCEPTION;
    }
}
```



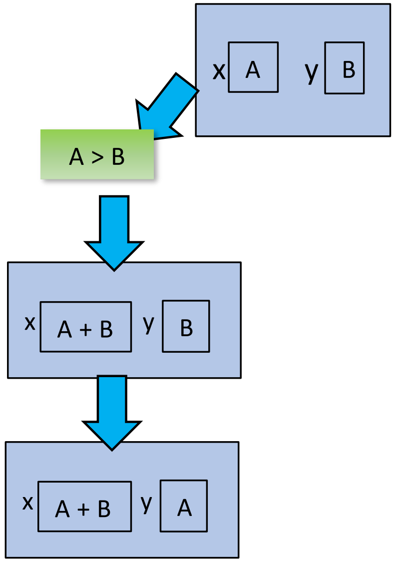
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
   if (x > y) {
      THROW_EXCEPTION;
   }
}
```



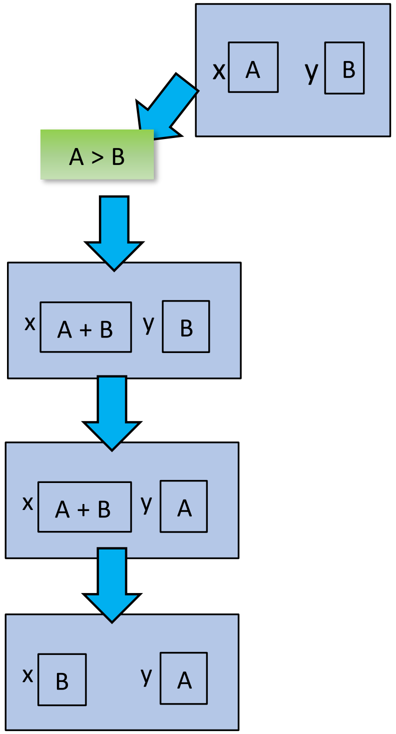
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
   if (x > y) {
      THROW_EXCEPTION;
   }
}
```



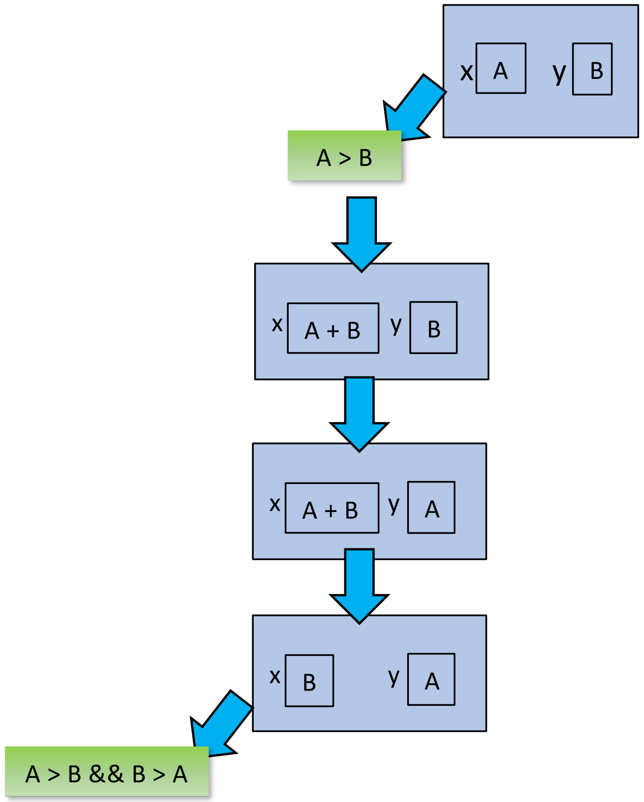
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
➡  if (x > y) {
      THROW_EXCEPTION;
   }
}
```



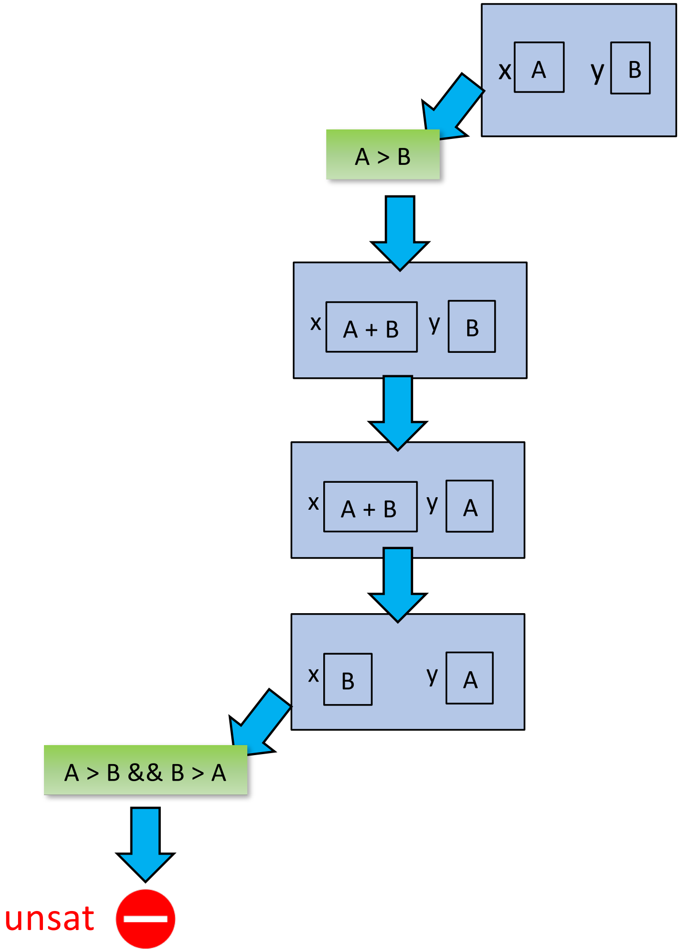
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
    if (x > y) {
        THROW_EXCEPTION;
    }
}
```



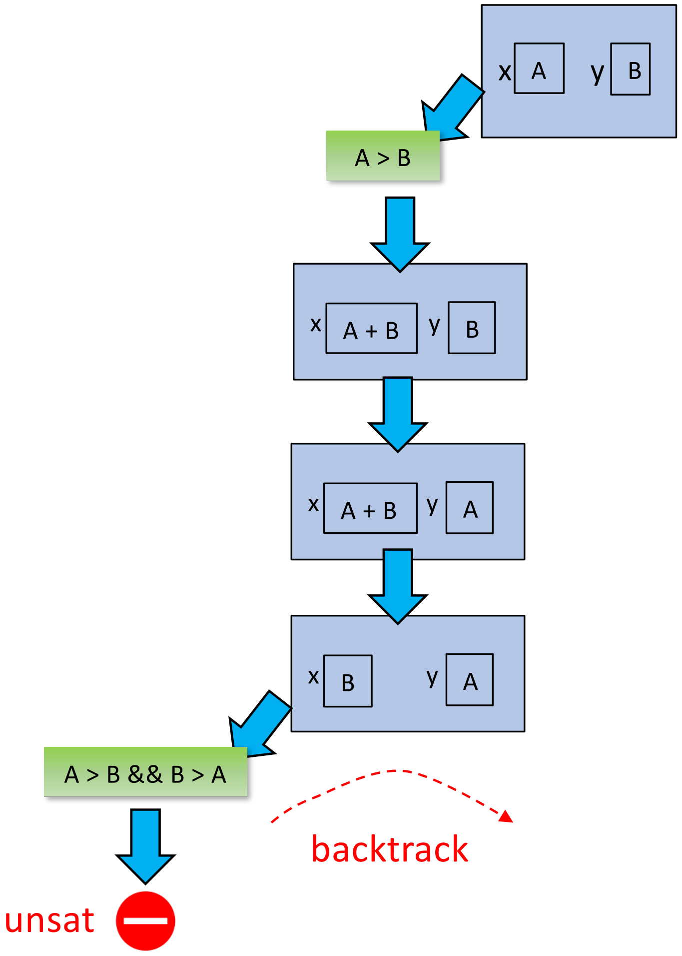
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
   if (x > y) {
➡      THROW_EXCEPTION;
   }
}
```



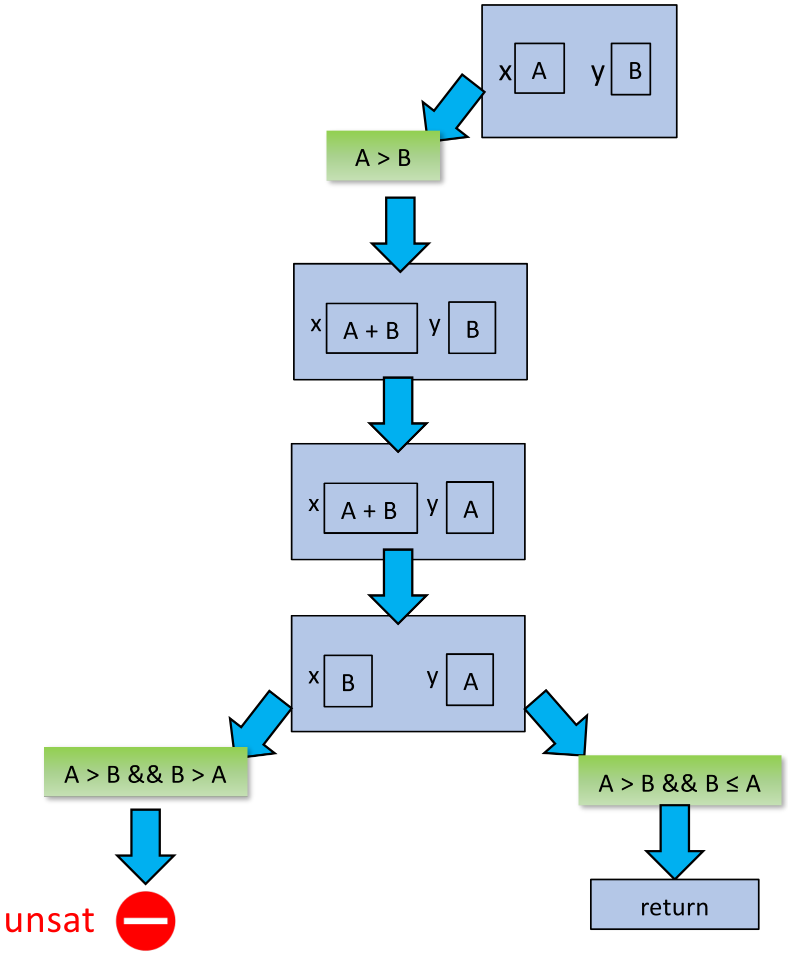
All-paths symbolic execution: Example

```
   int x, y;
   if (x > y) {
      x = x + y;
      y = x – y;
      x = x – y;
➡     if (x > y) {
         THROW_EXCEPTION;
      }
   }
```



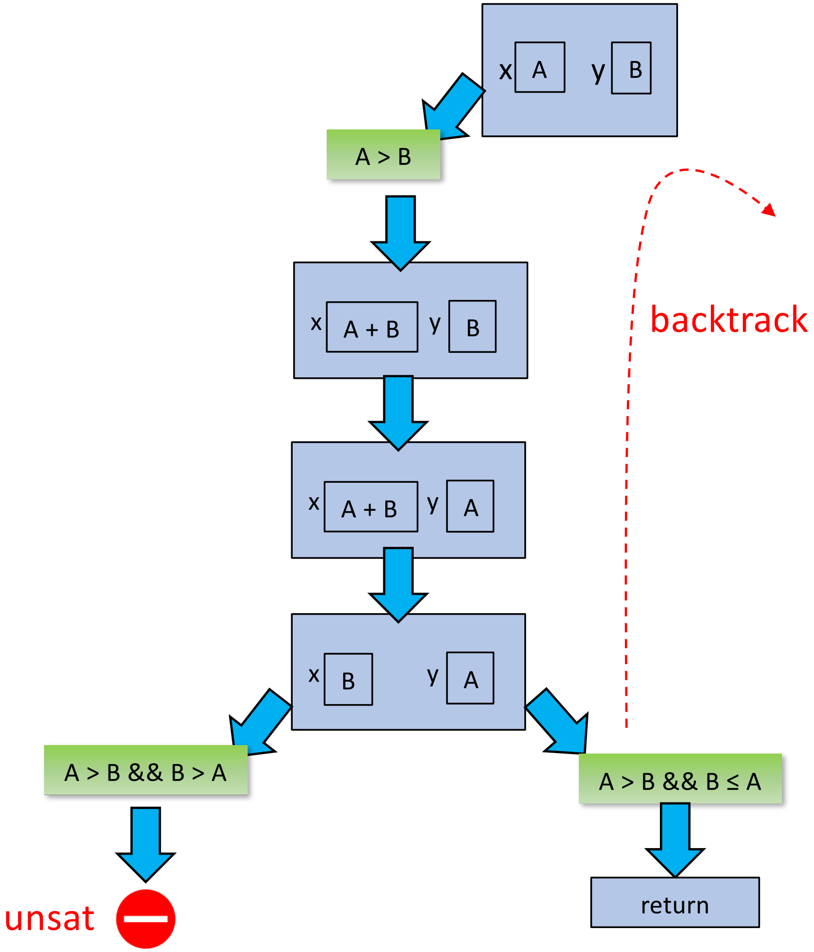
All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
    if (x > y) {
        THROW_EXCEPTION;
    }
}
```



All-paths symbolic execution: Example

```
  int x, y;
➡ if (x > y) {
      x = x + y;
      y = x – y;
      x = x – y;
      if (x > y) {
          THROW_EXCEPTION;
      }
  }
```



x A y B

A > B

x A + B y B

x A + B y A

x B y A

A > B && B > A

unsat

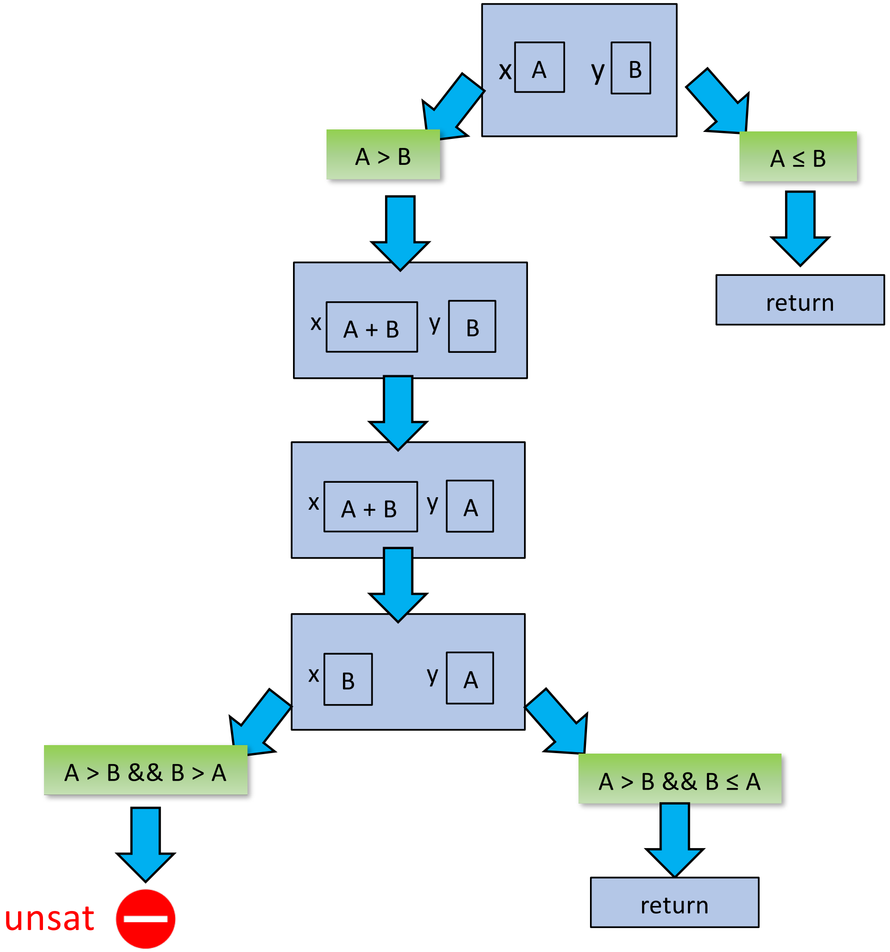
A > B && B ≤ A

return

backtrack

All-paths symbolic execution: Example

```
int x, y;
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
   if (x > y) {
      THROW_EXCEPTION;
   }
}
```



Limits of symbolic execution

- Ability of the constraint solver to handle the path conditions
  - Hard to handle nonlinear constraints
  - Solvers for machine arithmetics (floats) exist but are often slow
- Path explosion problem
  - Symbolic execution does not abstract at loop branches
  - Thus the number of paths can be infinite (or very large)

Problematic constraints: Example

```
double snapLon(double latPoint, double lonPoint, double fix1Lat, double fix1Lon, double fix2Lat, double fix2Lon) {
    PointXY p = calc.toXY(latPoint, lonPoint);
    PointXY l1 = calc.toXY(fix1Lat, fix1Lon);
    PointXY l2 = calc.toXY(fix2Lat, fix2Lon);

    // Find linear equation for line segment (y = mx + b)
    double m = (l2.getY() - l1.getY()) / (l2.getX() - l1.getX());
    double b = l1.getY() - (m * l1.getX());

    // Find linear equation for perpendicular line
    double mP = -1 / m;
    double bP = p.getY() - (mP * p.getX());

    // Find where line segment and perpendicular intersect
    double x = (b - bP) / (mP - m);
    double y = (m * x) + b;
    PointXY intersectXY = new PointXY(x, y);

    // Check if this point does indeed lie on the line segment.
    // If so, it must be the closest point on the line segment to p
    if (((l1.getX() < x && x < l2.getX()) || (l2.getX() < x && x < l1.getX())) &&
       ((l1.getY() < y && y < l2.getY()) || (l2.getY() < y && y < l1.getY()))) {
       return calc.toLL(intersectXY).getLongitude();
    }

    // If this intersection point does not lie on the line segment,
    // the closest point on the line segment to p must be an end point
    // Find the minimum distance to an end point
    double dist1 = calc.distanceXY(p, l1);
    double dist2 = calc.distanceXY(p, l2);
    PointXY minXY = dist1 < dist2 ? l1 : l2;
    return calc.toLL(minXY).getLongitude();
  }
```

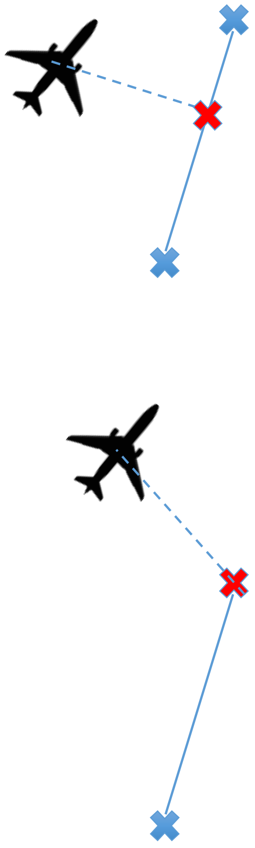
Problematic constraints: Example

```
double snapLon(double latPoint, double lonPoint, double fix1Lat, double fix1Lon, double fix2Lat, double fix2Lon) {
    PointXY p = calc.toXY(latPoint, lonPoint);
    PointXY l1 = calc.toXY(fix1Lat, fix1Lon);
    PointXY l2 = calc.toXY(fix2Lat, fix2Lon);

    // Find linear equation for line segment (y = mx + b)
    double m = (l2.getY() - l1.getY()) / (l2.getX() - l1.getX());
    double b = l1.getY() - (m * l1.get

    // Find linear equation for perpe
    double mP = -1 / m;
    double bP = p.getY() - (mP * p.g

    // Find where line segment and
    double x = (b - bP) / (mP - m);
    double y = (m * x) + b;
    PointXY intersectXY = new Point

    // Check if this point does indee
    // If so, it must be the closest po
    if (((l1.getX() < x && x < l2.getX()) || (l2.getX() < x && x < l1.getX())) &&
       ((l1.getY() < y && y < l2.getY()) || (l2.getY() < y && y < l1.getY()))) {
       return calc.toLL(intersectXY).getLongitude();
    }

    // If this intersection point does not lie on the line segment,
    // the closest point on the line segment to p must be an end point
    // Find the minimum distance to an end point
    double dist1 = calc.distanceXY(p, l1);
    double dist2 = calc.distanceXY(p, l2);
    PointXY minXY = dist1 < dist2 ? l1 : l2;
    return calc.toLL(minXY).getLongitude();
  }
```

```
private double metersPerLonAt(double lat) {
   return METERS_PER_LON_AT_EQUATOR * Math.cos(lat * RADIANS_PER_DEGREE);
}

public PointXY toXY(double lat, double lon) {
      return new PointXY((lon - minLon) * metersPerLonAt(lat), (lat - minLat) * METERS_PER_LAT);
}
```

Problematic constraints: Example

```
double snapLon(double latPoint, double lonPoint, double fix1Lat, double fix1Lon, double fix2Lat, double fix2Lon) {
    PointXY p = calc.toXY(latPoint, lonPoint);
    PointXY l1 = calc.toXY(fix1Lat, fix1Lon);
    PointXY l2 = calc.toXY(fix2Lat, fix2Lon);

    // Find linear equation for line segment (y = mx + b)
    double m = (l2.getY() - l1.getY()) / (l2.getX() - l1.getX());
    double b = l1.getY() - (m * l1.getX());

    // Find linear equation for perpendicular line
    double mP = -1 / m;
    double bP = p.getY() - (mP * p.getX());

    // Find where line segment and perpendicular intersect
    double x = (b - bP) / (mP - m);
    double y = (m * x) + b;
    PointXY intersectXY = new PointXY(x, y);

    // Check if this point does indeed lie on the line segment.
    // If so, it must be the closest point on the line segment to p
    if (((l1.getX() < x && x < l2.getX()) || (l2.getX() < x && x < l1.getX())) &&
      ((l1.getY() < y && y < l2.getY()) || (l2.getY() < y && y < l1.getY()))) {
      return calc.toLL(intersectXY).getLongitude();
    }

    // If this intersection point does not lie on the line segment,
    // the closest point on the line segment to p must be an end point
    // Find the minimum distance to an end point
    double dist1 = calc.distanceXY(p, l1);
    double dist2 = calc.distanceXY(p, l2);
    PointXY minXY = dist1 < dist2 ? l1 : l2;
    return calc.toLL(minXY).getLongitude();
  }
```



Problematic constraints: Example

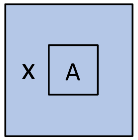
```
double snapLon(double la
    PointXY p = calc.toXY(
    PointXY l1 = calc.toXY(
    PointXY l2 = calc.toXY(

    // Find linear equation
    double m = (l2.getY() -
    double b = l1.getY() - (

    // Find linear equation
    double mP = -1 / m;
    double bP = p.getY() -

    // Find where line seg
    double x = (b - bP) / (
    double y = (m * x) + b;
    PointXY intersectXY =

    // Check if this point d
    // If so, it must be the
    if (((l1.getX() < x && x
      ((l1.getY() < y && y
      return calc.toLL(inte
    }

    // If this intersection p
    // the closest point on
    // Find the minimum
    double dist1 = calc.dis
    double dist2 = calc.dis
    PointXY minXY = dist1
    return calc.toLL(minXY
  }
```

```
timeHorizon > 0 &&
speed > 0 &&
sqrt(((lon - minLon) * 111194.92664455874 * cos(lat * 0.017453292519943295) -
(((lon - minLon) * 111194.92664455874 * cos(lat * 0.017453292519943295) +
cos(heading) * speed * timeHorizon) / (111194.92664455874 * cos((((lat - minLat) *
111194.92664455874 + sin(heading) * speed * timeHorizon) / 111194.92664455874 +
 minLat) * 0.017453292519943295)) + minLon - minLon) * 111194.92664455874 *
cos((((lat - minLat) * 111194.92664455874 + sin(heading) * speed * timeHorizon) /
111194.92664455874 + minLat) * 0.017453292519943295)) * ((lon - minLon) *
111194.92664455874 * cos(lat * 0.017453292519943295) - (((lon - minLon) *
111194.92664455874 * cos(lat * 0.017453292519943295) + cos(heading) * speed *
timeHorizon) / (111194.92664455874 * cos((((lat - minLat) * 111194.92664455874 +
sin(heading) * speed * timeHorizon) / 111194.92664455874 + minLat) *
0.017453292519943295)) + minLon - minLon) * 111194.92664455874 *
cos((((lat - minLat) * 111194.92664455874 + sin(heading) * speed * timeHorizon)
/ 111194.92664455874 + minLat) * 0.017453292519943295)) + ((lat - minLat) *
111194.92664455874 - (((lat - minLat) * 111194.92664455874 + sin(heading) *
speed * timeHorizon) / 111194.92664455874 + minLat - minLat) * 111194.92664455874)
* ((lat - minLat) * 111194.92664455874 - (((lat - minLat) * 111194.92664455874 +
sin(heading) * speed * timeHorizon) / 111194.92664455874 + minLat - minLat) *
111194.92664455874)) - speed * timeHorizon != 0
```

Loops

```
➡ int x;
  while (x >= 0) {
      x--;
  }
  return x;
```

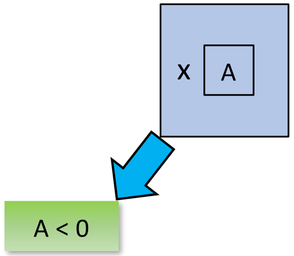
Loops



```
int x;
while (x >= 0) {
    x--;
}
return x;
```

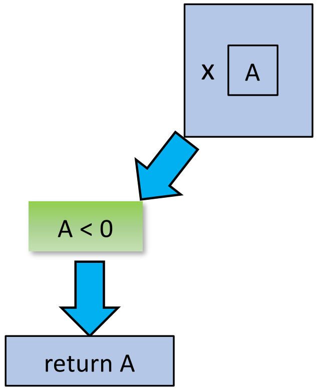
Loops

```
int x;
while (x >= 0) {
   x--;
}
return x;
```



Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



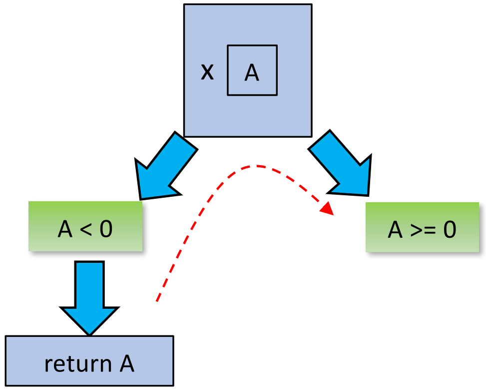
x A

A < 0

return A

Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



x A

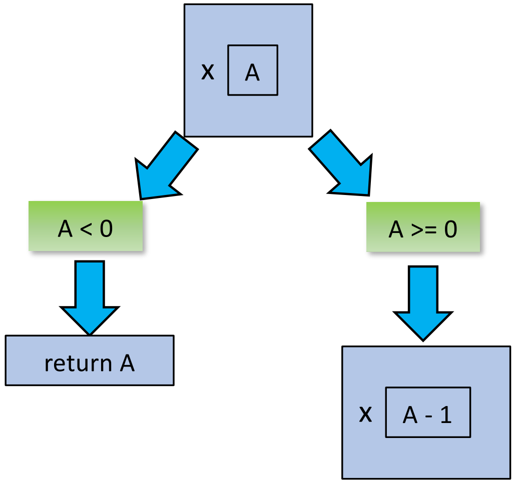
A < 0

A >= 0

return A

Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



x A

A < 0

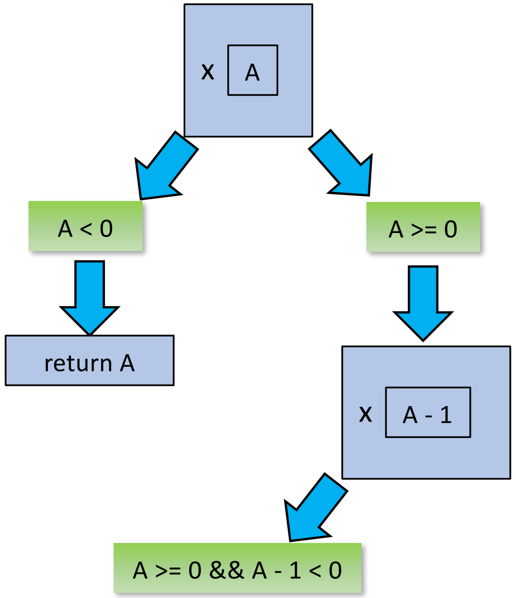
return A

A >= 0

x A - 1

Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



x A

A < 0

return A

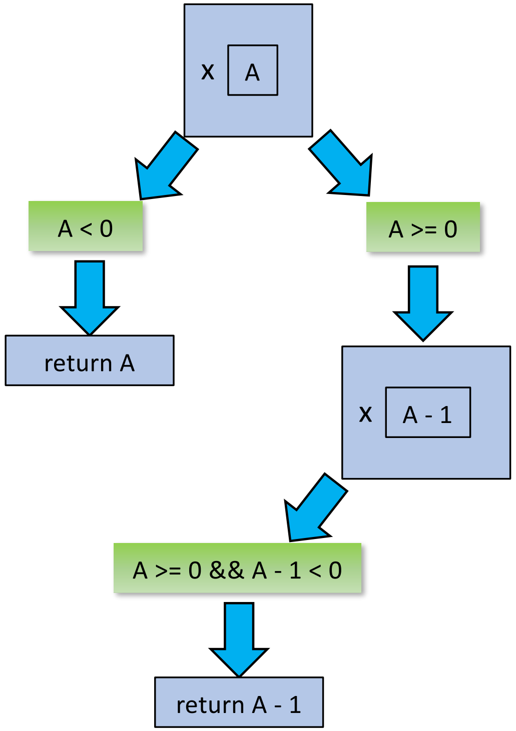
A >= 0

x A - 1

A >= 0 && A - 1 < 0

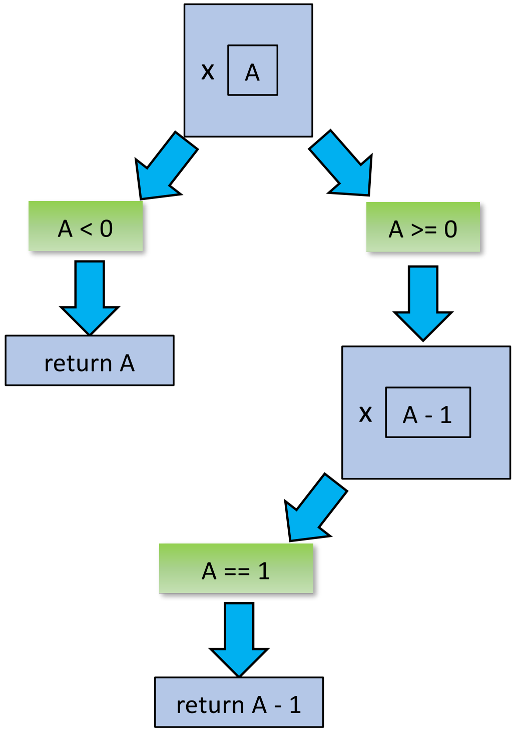
Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



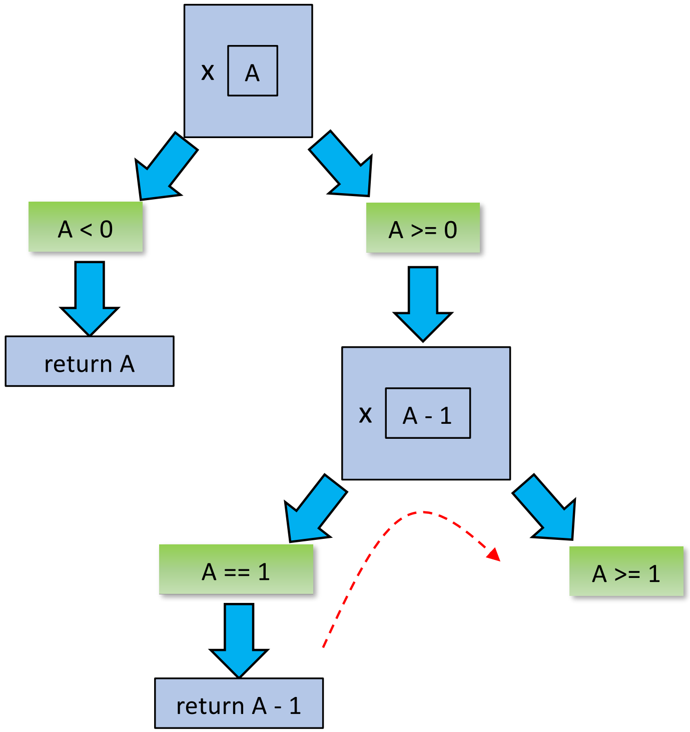
Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



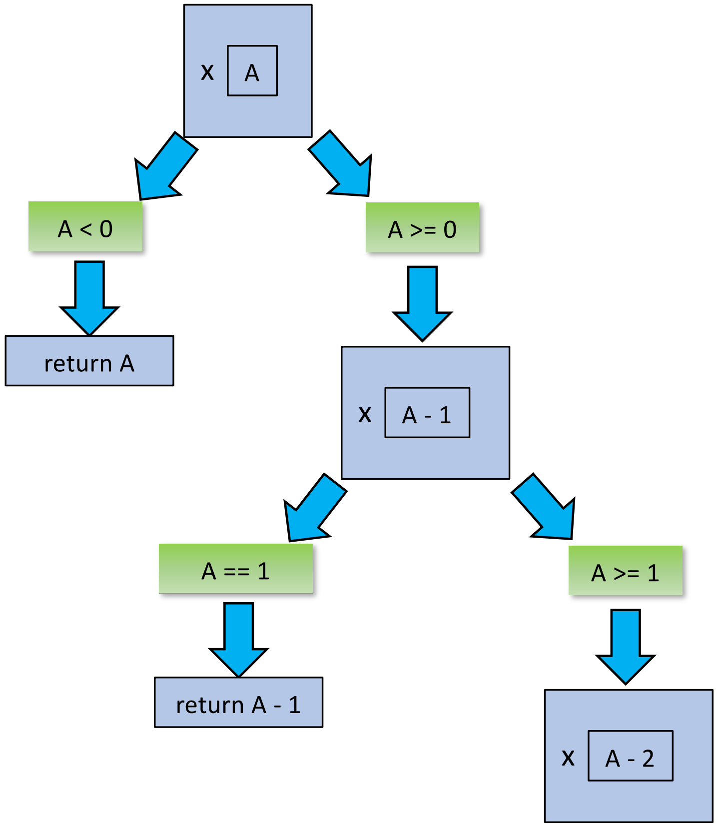
Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



x A

A < 0

return A

A >= 0

x A - 1

A == 1

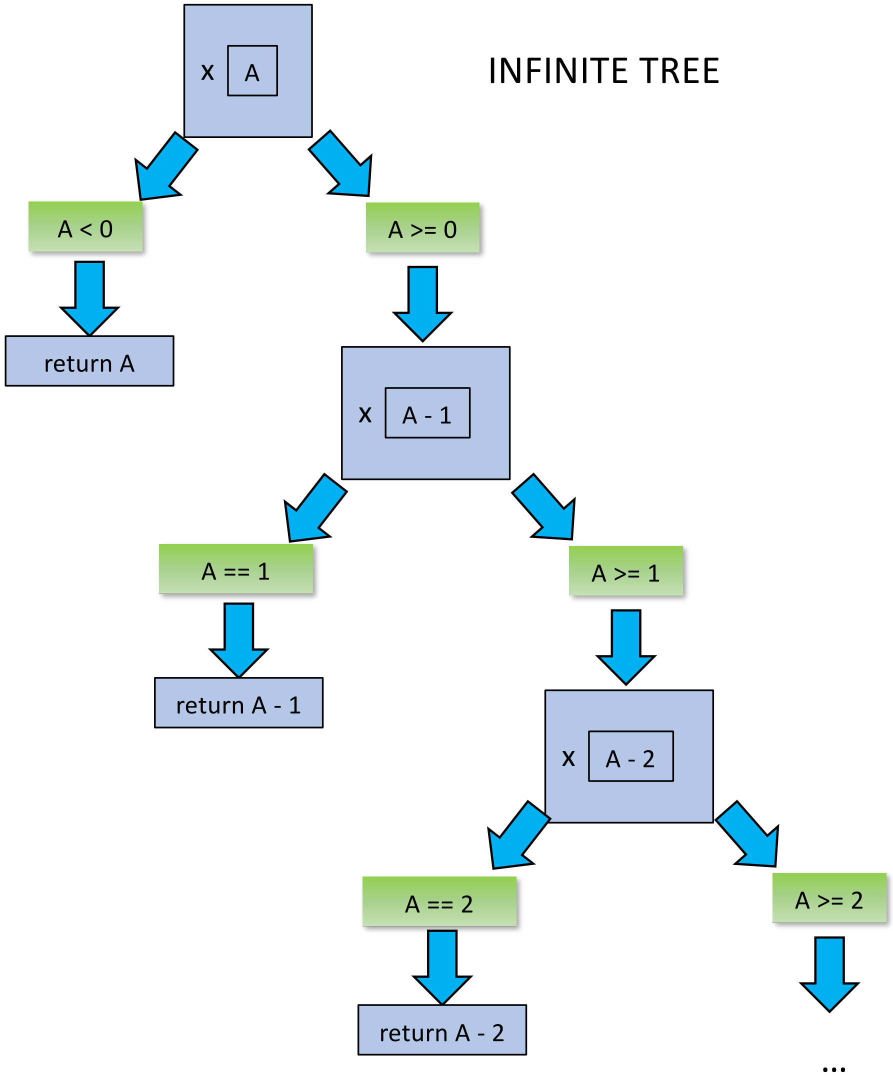
return A - 1

A >= 1

x A - 2

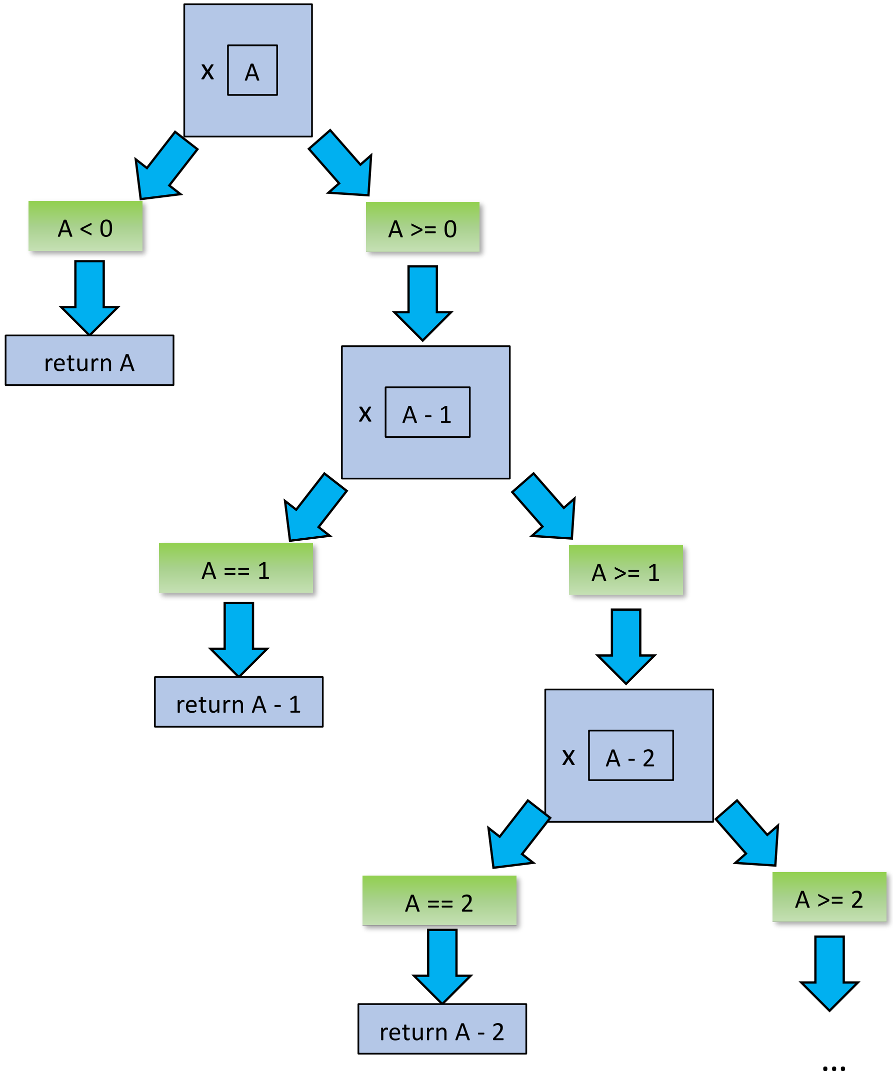
Loops

```
int x;
while (x >= 0) {
    x--;
}
return x;
```



Loops

- This tree has infinite finite-length paths (from root to a return)
- And one infinite-length path (the rightmost in the tree)
- But the program always terminates
- Thus the infinite-length path is infeasible



Applications of symbolic execution

Applications of symbolic execution

- Automatic generation of tests
- Assertion-based program verification
- Program analysis
    - Determining functional equivalence
    - Worst case execution time estimation for real-time software
    - …

Assertion-based verification

- Symbolic execution can be used to perform assertion-based verification
- Idea: inject assertion in the code, and find if at least one feasible path exist to the violation of the assertion
- In its most classical incarnation is based on the concept of Hoare triples, `{precondition} Program {postcondition}`, meaning that:
  - **If** `precondition` is true in the initial state…
  - …**then**, after the execution of `Program`, `postcondition` is true in the final state
- (This is partial correctness, because it does not require that `Program` always terminates when `precondition` is true in the initial state)

Assertion-based verification: Template

```
{precondition}
Program
{postcondition}
```

⇒

```
assume(precondition);
Program;
assert(postcondition);
```

```
void assume(boolean b) {
  if (!b) DISCARD_PATH_AND_BACKTRACK;
}

void assert(boolean b) {
  if (!b) THROW_EXCEPTION;
}
```

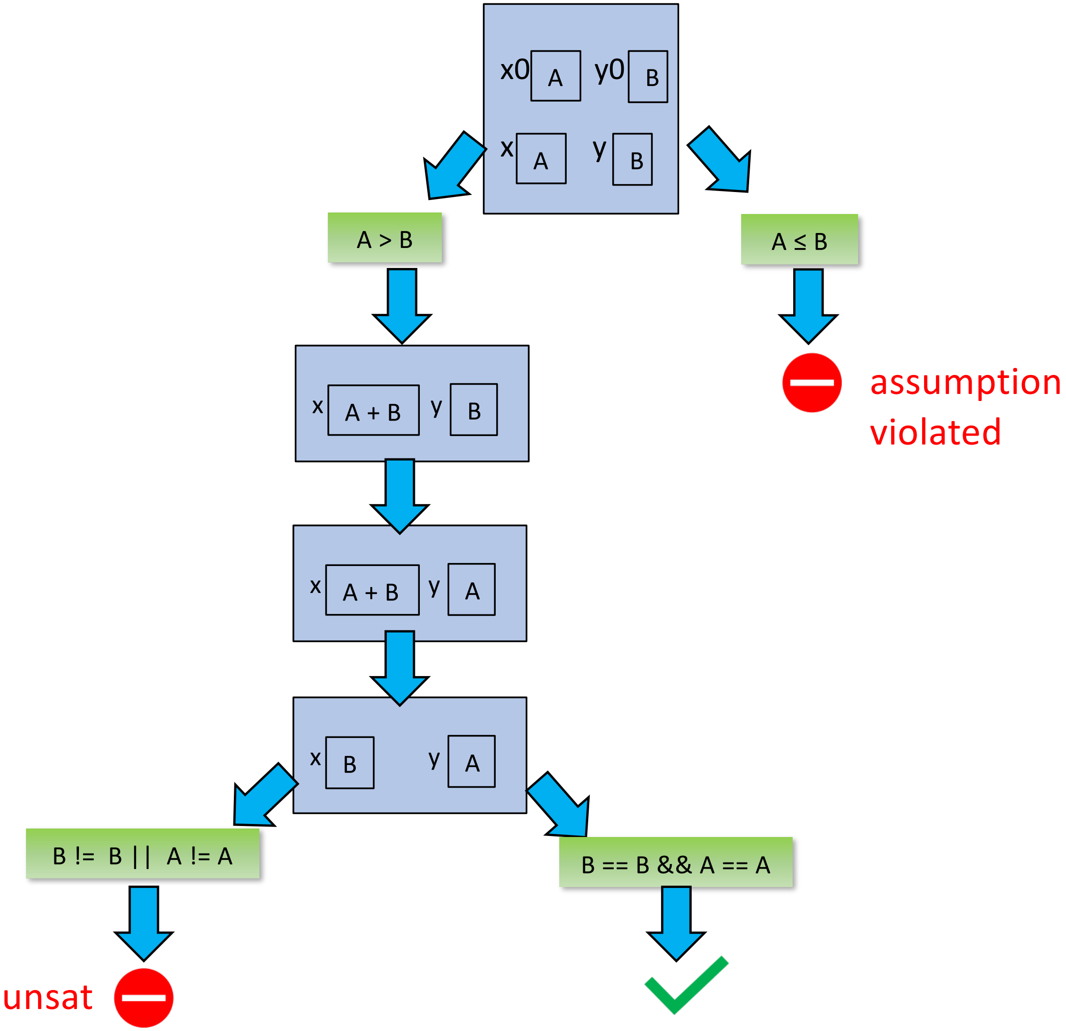
Example

```
int x, y;

{x == x0 && y == y0 && x > y}
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
}
{x == y0 && y == x0}
```

```
int x, y;

int x0 = x, y0 = y;
assume(x > y);
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
}
assert(x == y0 && y == x0);
```

Example

```
int x, y;

int x0 = x, y0 = y;
assume(x > y);
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
}
assert(x == y0 && y == x0);
```



x0 A y0 B

x A y B

A > B

A ≤ B

assumption violated

x A + B y B

x A + B y A

x B y A

B != B || A != A

B == B && A == A

unsat

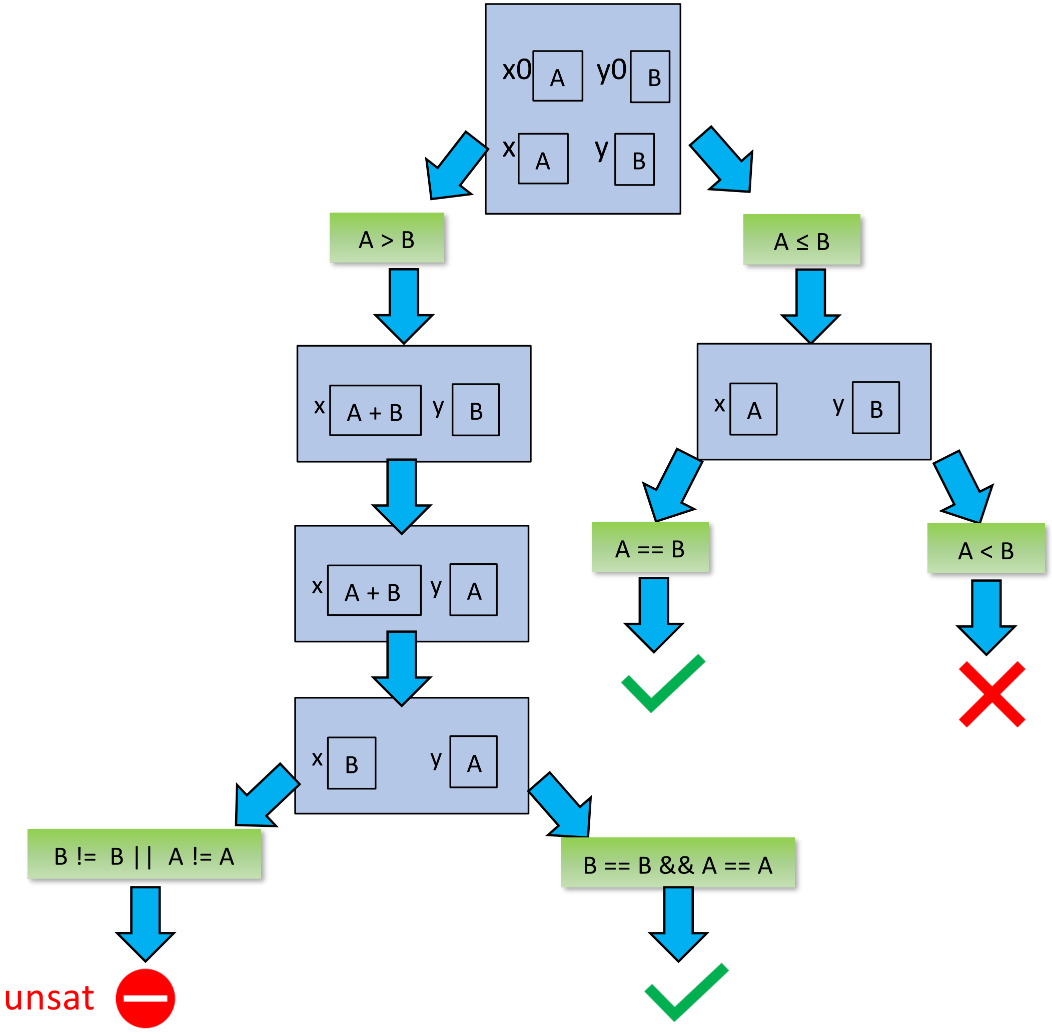
Another example

```
int x, y;

{x == x0 && y == y0}
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
}
{x == y0 && y == x0}
```

```
int x, y;

int x0 = x, y0 = y;
//assume(true);
if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
}
assert(x == y0 && y == x0);
```

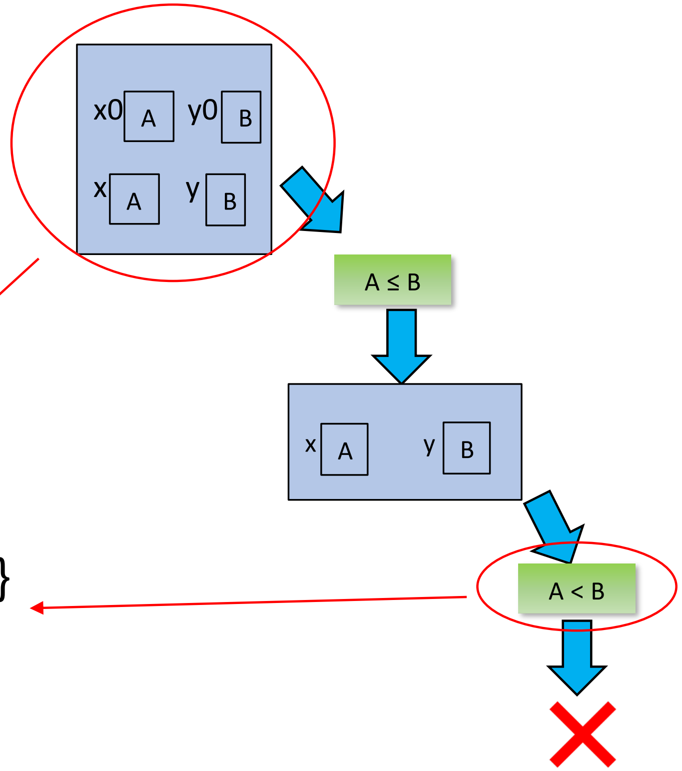
Another example

```
int x, y;

int x0 = x, y0 = y;
//assume(true);
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
}
assert(x == y0 && y == x0);
```



Another example

```
int x, y;

int x0 = x, y0 = y;
//assume(true);
if (x > y) {
   x = x + y;
   y = x – y;
   x = x – y;
}
assert(x == y0 && y == x0);
```



Test case: {x == 0, y == 1}

Counterexample

Assertion-based verification of program with loops

- When the program has loops, things are more complex:
  - The number of feasible paths may become infinite
  - If the program may diverge, there are path with infinite length
- A small improvement: visit the symbolic execution tree breadth-first, instead of depth-first does not “get stuck” in loops in trivial cases
  - Will terminate (and return a counterexample) if the program is incorrect
  - Will terminate if the program has a finite number of finite-length paths
  - Otherwise, will not terminate (it is a semi-algorithm)
- Approaches:
  - Perform bounded symbolic execution
  - Summarize loops with loop invariants

Bounded symbolic execution

- Bounded symbolic execution explores a finite portion of the symbolic execution tree
- Several possible bounds:
  - On the ranges of the symbolic values
  - On the number of iterations of loops
  - On the length of the paths
  - On the total number of explored paths
  - …
- Consequence:
  - Verification is **sound**: if it finds a counterexample, the program is incorrect
  - But it is not **complete**: if it does not find a counterexample, we cannot infer that the program is correct

Loop invariants

- Loop invariants allow to perform inductive verification of programs with loops
- They can be exploited to transform the verification of a program with loops in the verification of a collection of loop-free programs
- Automatically inferring loop invariants is an undecidable problem, and they must be provided by means of manual annotations

Breaking loops with help of invariants

```
{precondition}
Program_1;
while (loop_cond)
//loop_invariant
  Loop_body;
Program_2;
{postcondition}
```

```
assume(precondition);
Program_1;
assert(loop_invariant);
```

```
assume(loop_invariant && loop_cond);
Loop_body;
assert(loop_invariant);
```

```
assume(loop_invariant && !loop_cond);
Program_2;
assert(postcondition);
```

Example

```
int x;

{x >= 0}
while (x >= 0) {
  //loop invariant: x >= -1
  x--;
}
{x == -1}
```

```
int x;
assume(x >= 0);
; //do nothing
assert(x >= -1);
```

```
int x;
assume(x >= -1 && x >= 0);
x--;
assert(x >= -1);
```

```
int x;
assume(x >= -1 && !(x >= 0));
; //do nothing
assert(x == -1);
```

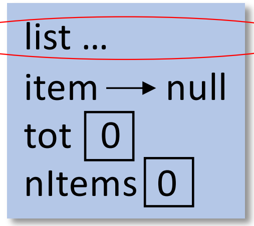
Symbolic execution of heap manipulating programs

Objects as inputs

- In our example inputs had primitive (int, boolean) types
- What if an input has reference type?
- (Use case: programs that manipulate data structures)
- We describe the **generalized symbolic execution** approach (Khurshid, Pasareanu, Visser, TACAS 2003)

Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}
```



list …

item → null

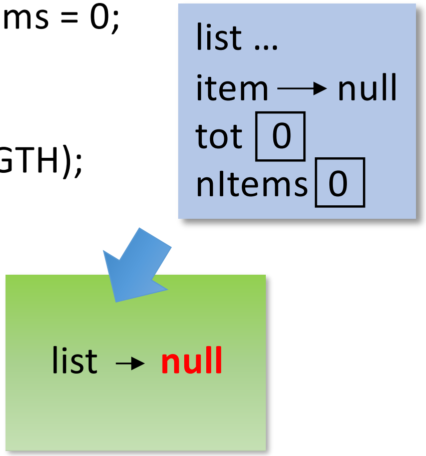
tot 0

nItems 0

The input is a (doubly-linked, circular) list

Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}
```

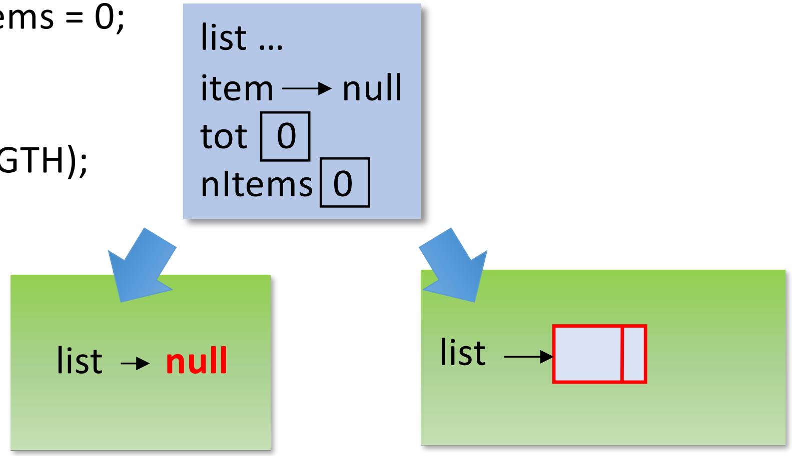


list …
item → null
tot 0
nItems 0

list → **null**

Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



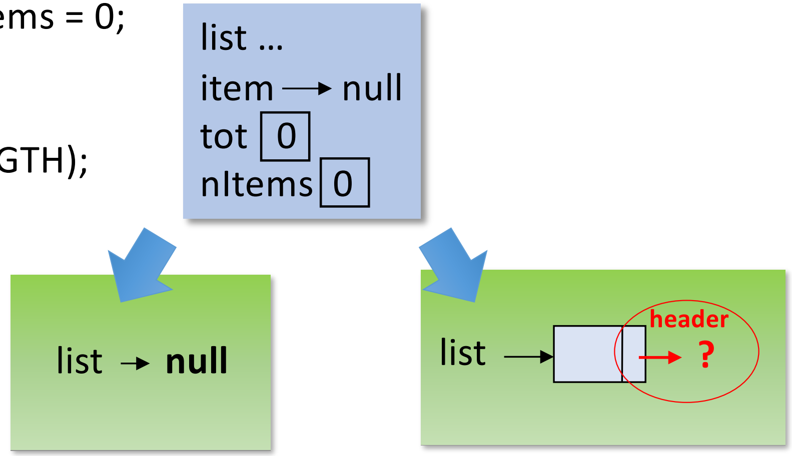
list …
item → null
tot 0
nItems 0

list → **null**

list →

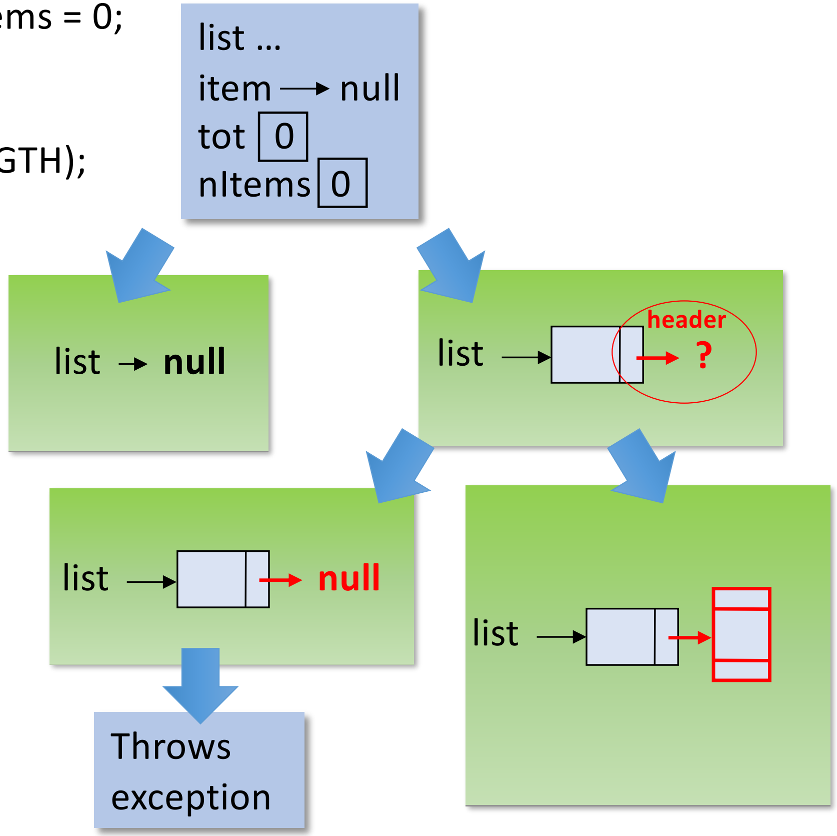
Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



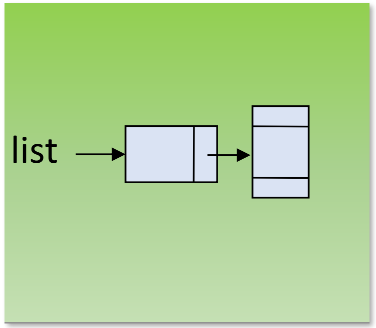
Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
➡  for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```

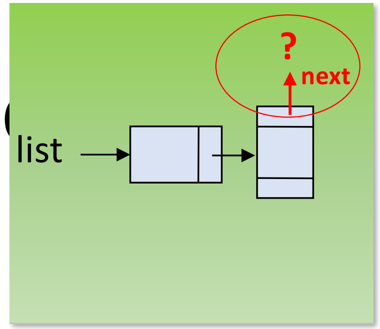


Generalized symbolic execution

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



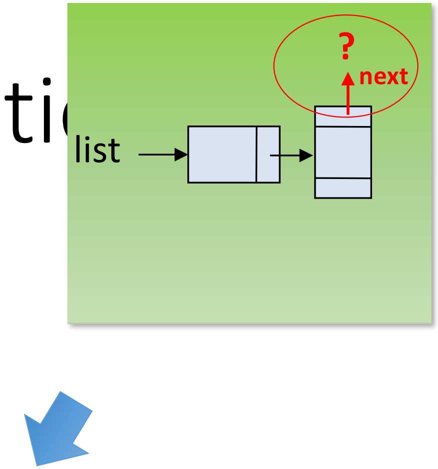
Generalized symbolic executi

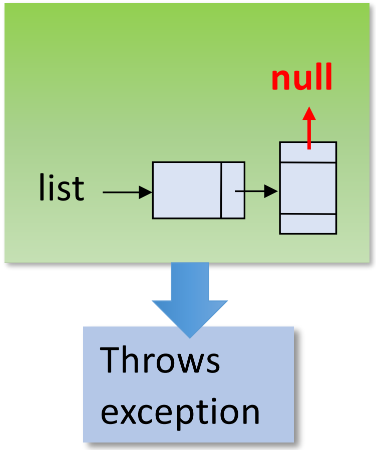


```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
➡ for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```

Generalized symbolic executi

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```

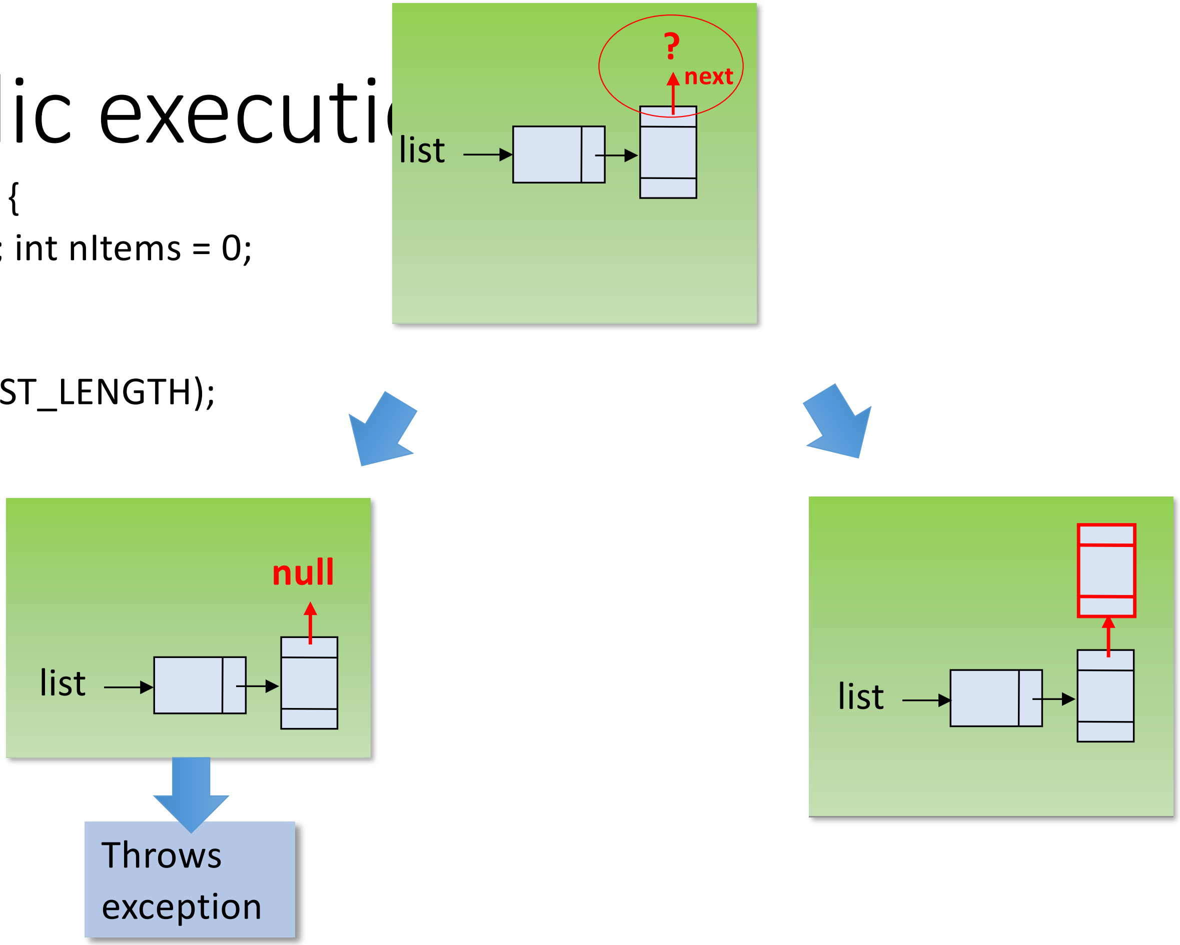




Throws exception

Generalized symbolic executi

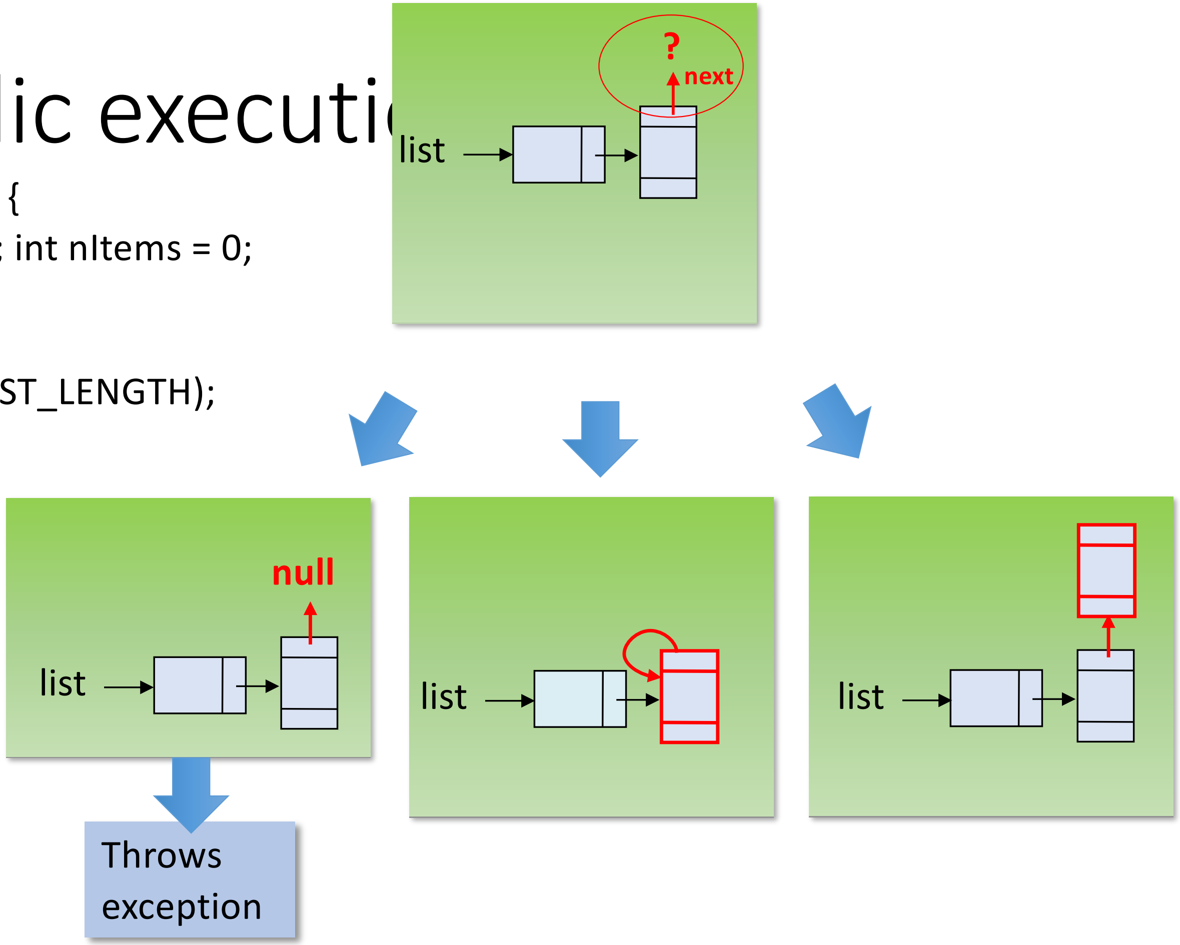
```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



Throws exception

Generalized symbolic executi

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



Throws exception

Generalized symbolic execution: pros

- Sound and complete: Exhaustively analyzes all possible alias combinations
- Easy to implement: Materialize concrete references and fresh initial objects as an input field or variable is accessed

Generalized symbolic execution: cons

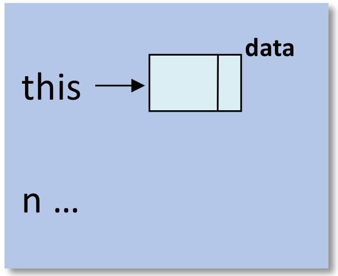
- Blows the number of paths
- One cause is that generalized symbolic execution introduces branches in the symbolic execution tree earlier than necessary
- Moreover, most paths assume inputs that violate their own representation invariants

Too early branching

- Materialization distinguishes a lot of cases that may not lead to different behaviors later in the execution
- Effect:
  - Branches in the symbolic execution tree are introduced that do not correspond to branches in code
  - Thus, different feasible paths in the symbolic execution tree may correspond to the same program path…
  - …yielding useless repeated executions of the same behavior
- Variants exist that recover in part the situation:
  - “Lazier” generalized symbolic execution (Deng, Lee, Robby, ASE 2006)
  - “Lazier#” generalized symbolic execution (Deng, Robby, Hatcliff, SEFM2007)

Lazier and lazier# techniques: Example

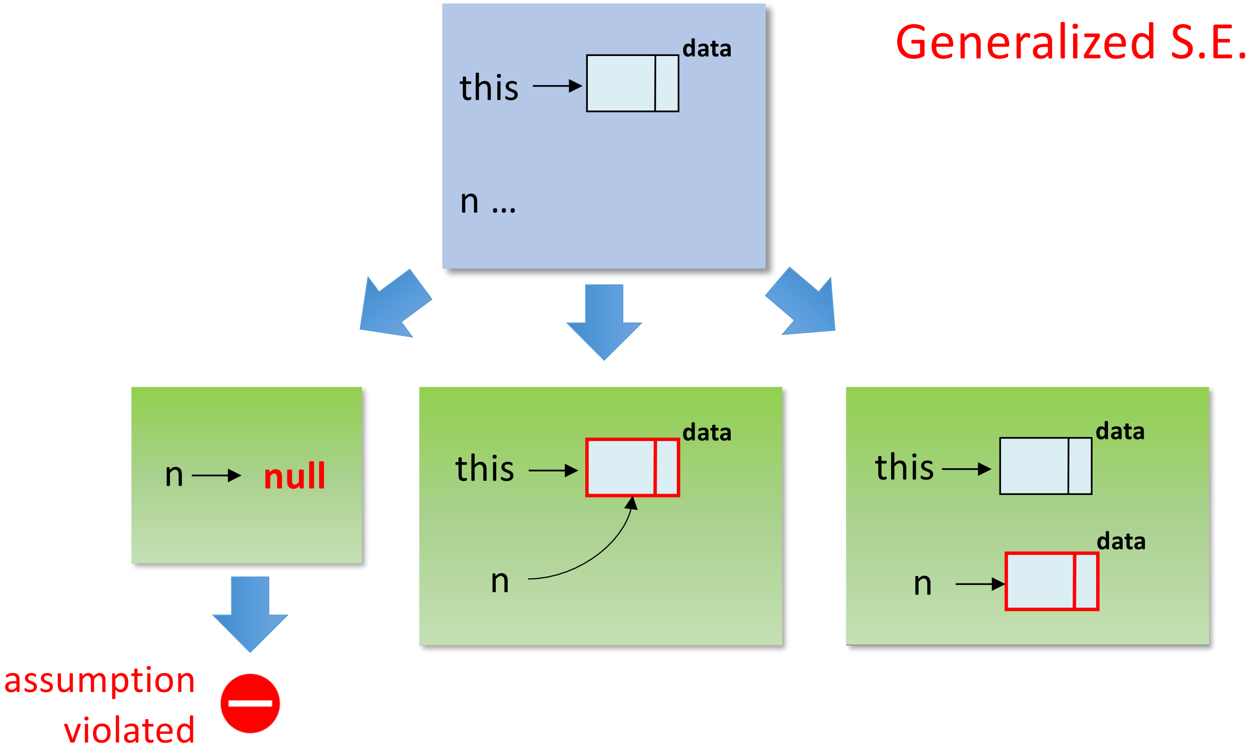
```
public class Node {
 Object data;
 Node next;

 void swap(Node n) {
  assume(n != null);
  Object tmp = this.data;
  this.data = n.data;
  n.data = tmp;
}
```



Generalized S.E.

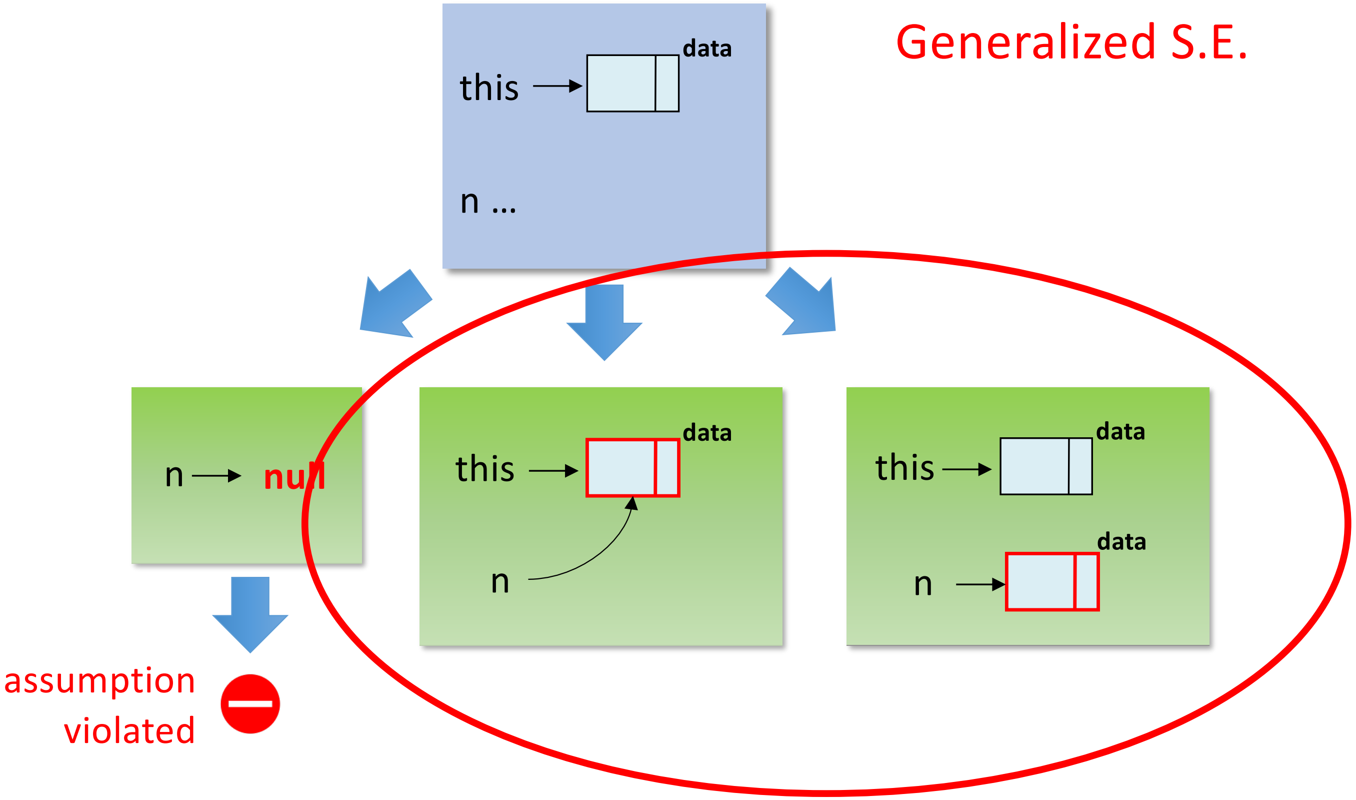
Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
➡   Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```



Lazier and lazier# techniques: Example

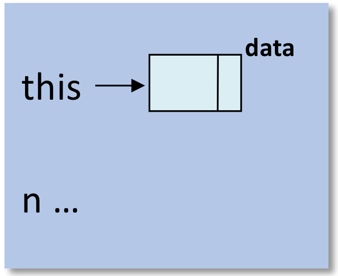
```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```



Why should we distinguish these two subcases now?

Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
➡   assume(n != null);
    Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```

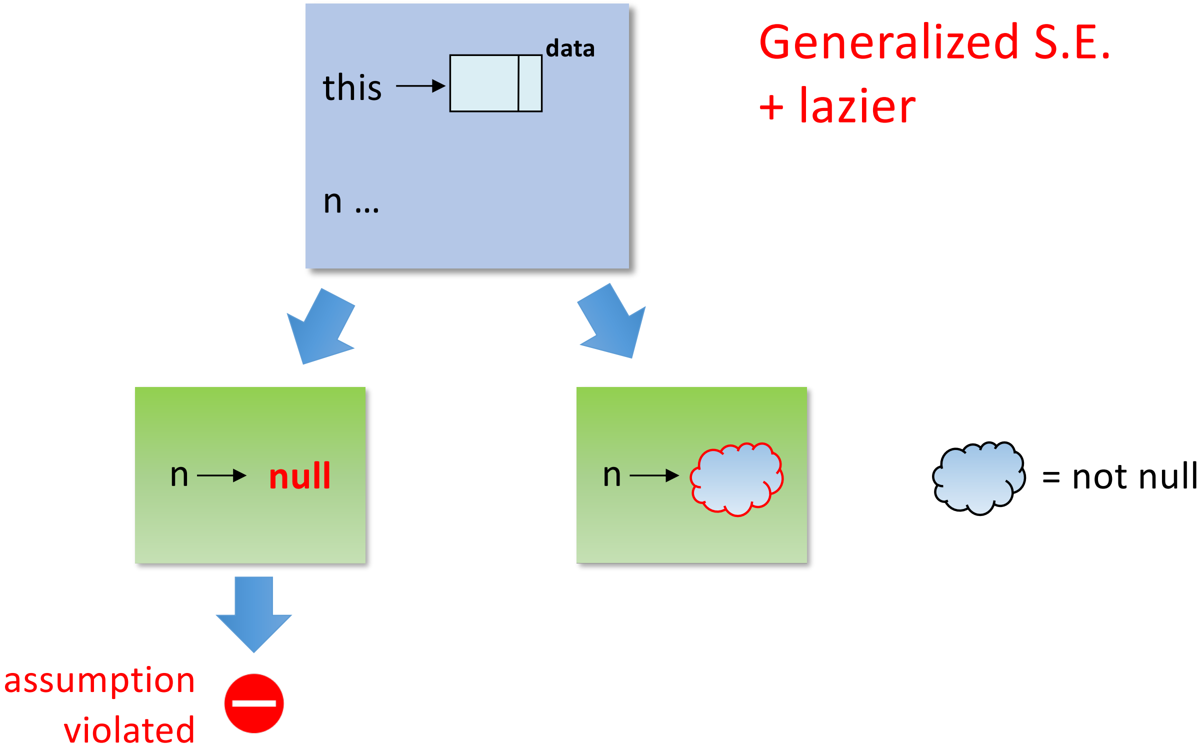


Generalized S.E.
+ lazier

Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```

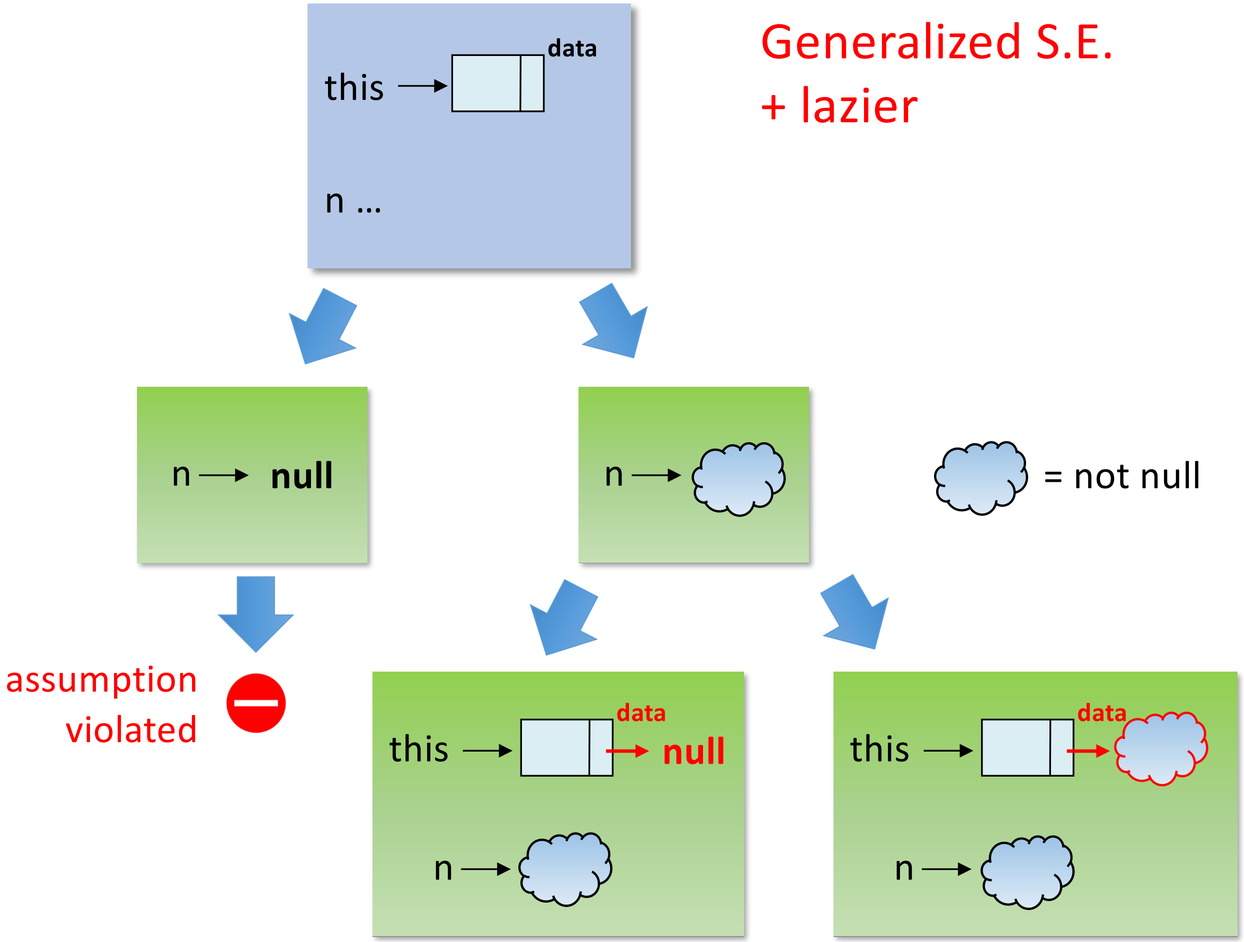
Generalized S.E.
+ lazier



Lazier and lazier# techniques: Example

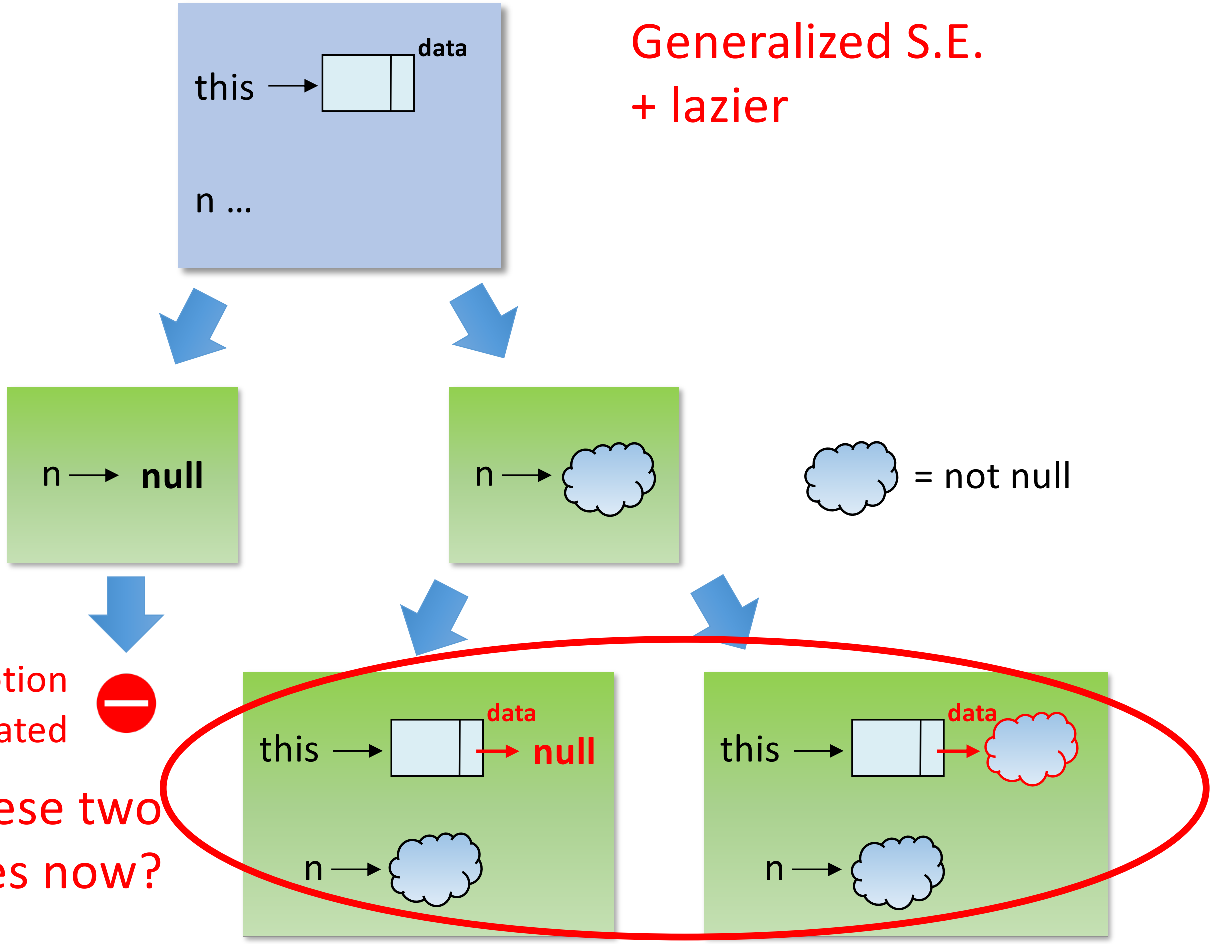
```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
➡   this.data = n.data;
    n.data = tmp;
}
```

Generalized S.E.
+ lazier



Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```



Generalized S.E.
+ lazier

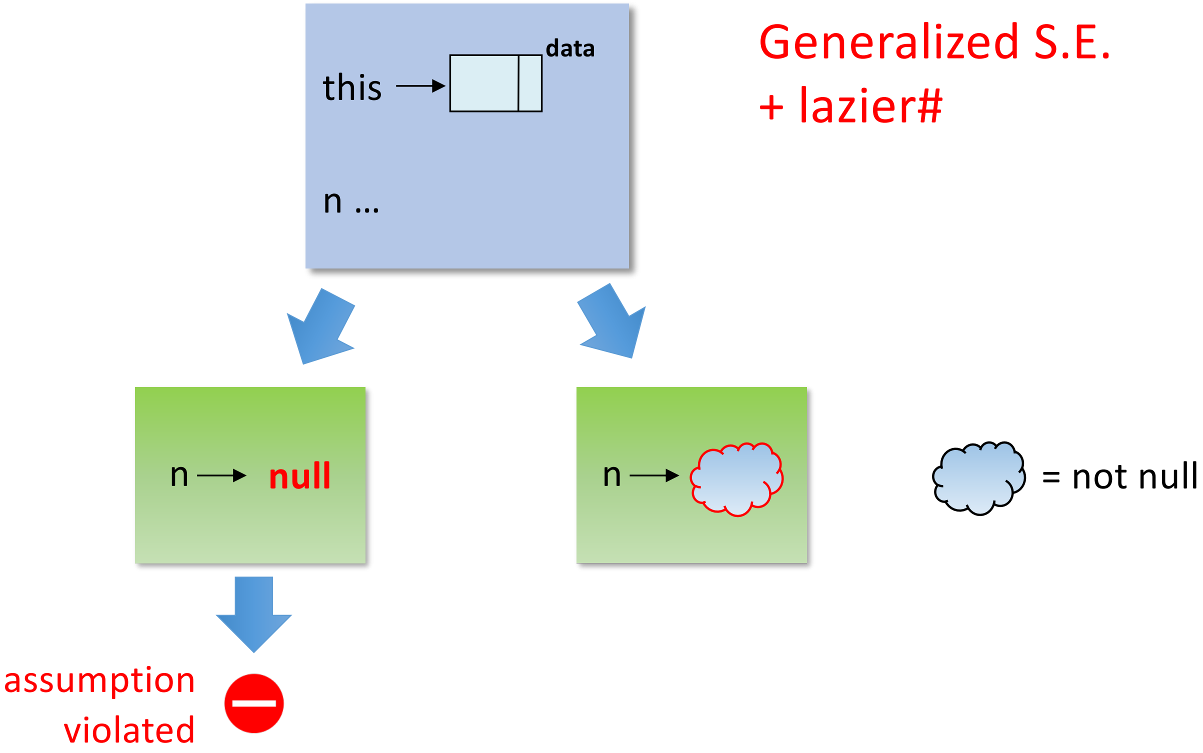
assumption
violated

Why should we distinguish these two
subcases now?

Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
    this.data = n.data;
    n.data = tmp;
}
```

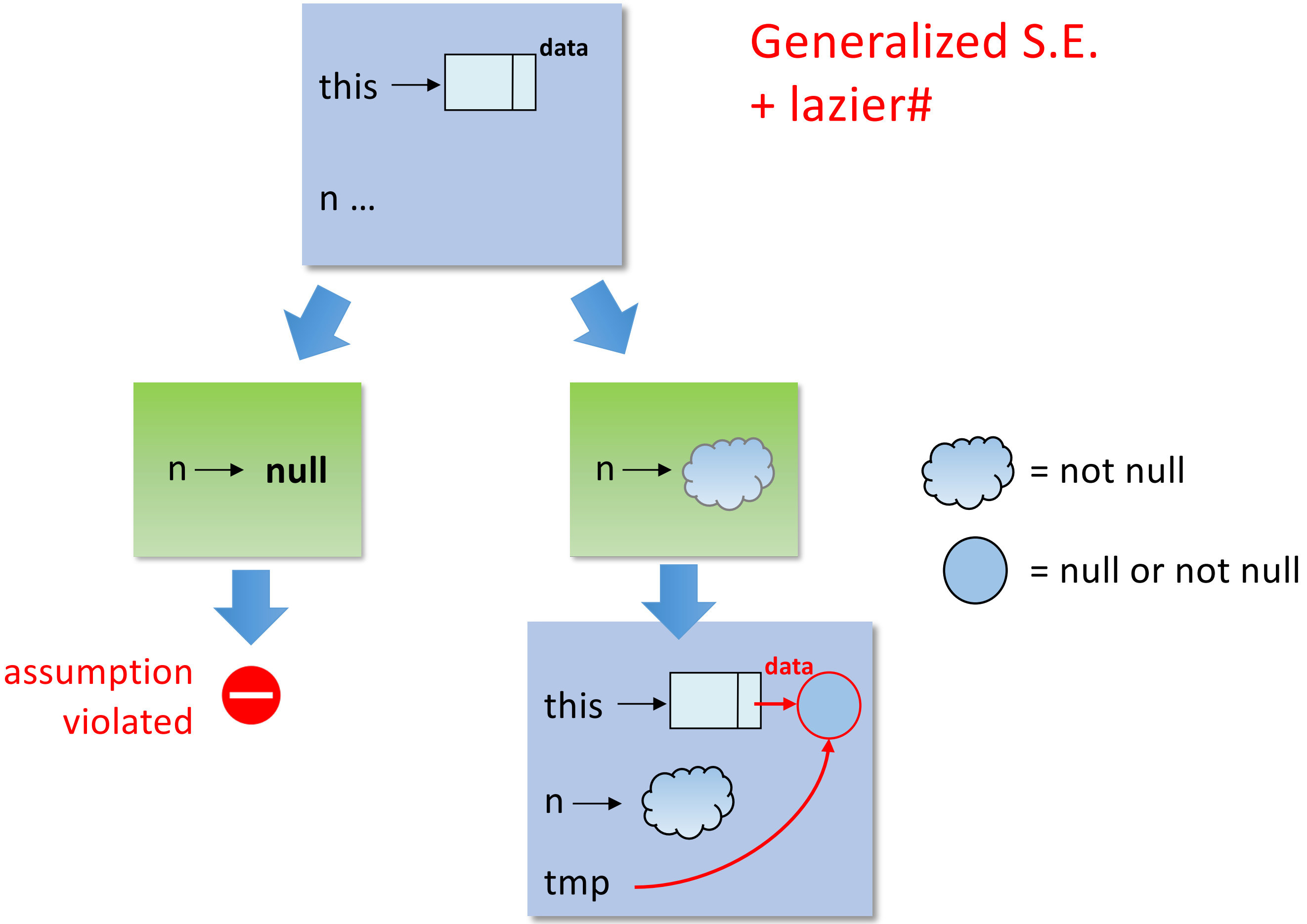
Generalized S.E.
+ lazier#



Lazier and lazier# techniques: Example

```
public class Node {
  Object data;
  Node next;

  void swap(Node n) {
    assume(n != null);
    Object tmp = this.data;
➡   this.data = n.data;
    n.data = tmp;
}
```

Generalized S.E.
+ lazier#

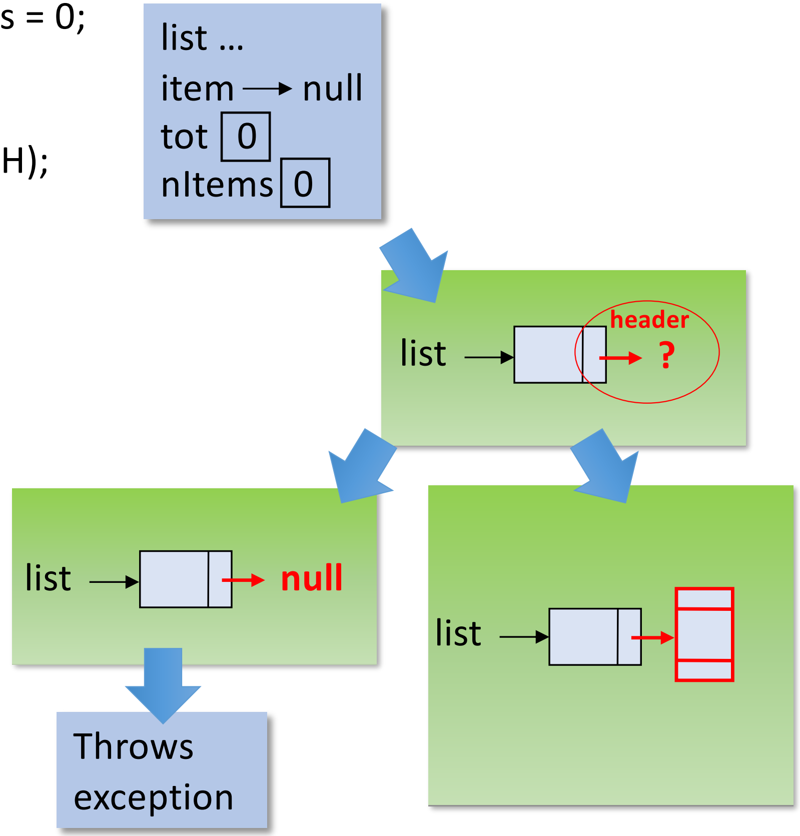


Representation invariants violations

- A **representation invariant** describes the correct shape of the objects of a given class
- For instance, for the LinkedList example, the representation invariant is that, for all LinkedList l:
  - l.header != null
  - l.header.next.next…next ends in l.header (same for l.header.prev.prev…prev)
  - For all nodes n reachable from l, n.next.prev == n.prev.next == n
  - The size is the number of nodes minus one
- Since generalized symbolic execution analyzes all alias combinations, it may consider shapes that violate representation invariants
- These are ill-formed input, and usually we are not interested in analyzing how the program behaves with ill-formed inputs

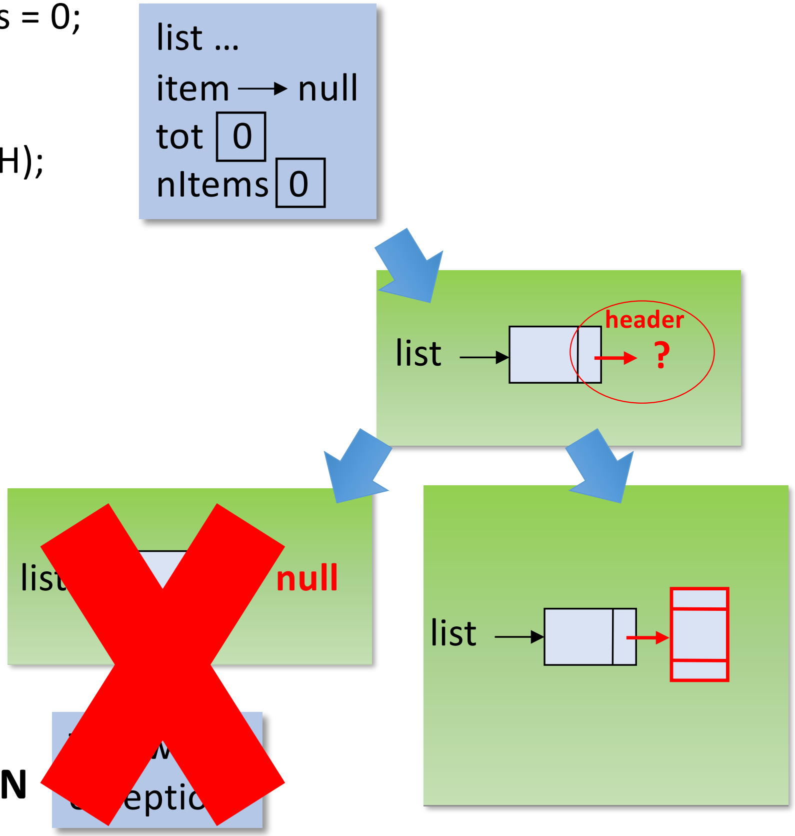
Representation invariant violations

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



Representation invariant violations

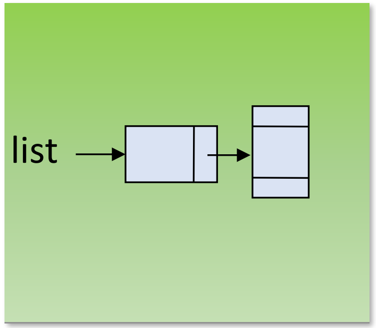
```
  int sum(LinkedList<Integer> list) {
     Integer item = null; int tot = 0; int nItems = 0;
➡   for (item : list) {
        tot += item.intValue();
        assert(++nItems <= MAX_LIST_LENGTH);
     }
     return tot;
  }

  public class LinkedList<Z> {
     int size = 0;
     Entry header = new Entry();
     class Entry {
        Entry next, prev;
        Z value;
     }
     …
  }
```



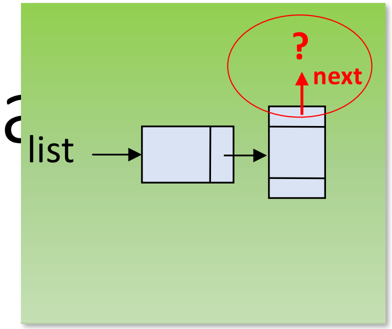
REPRESENTATION INVARIANT VIOLATION

Representation invariant violations

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
➡  for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



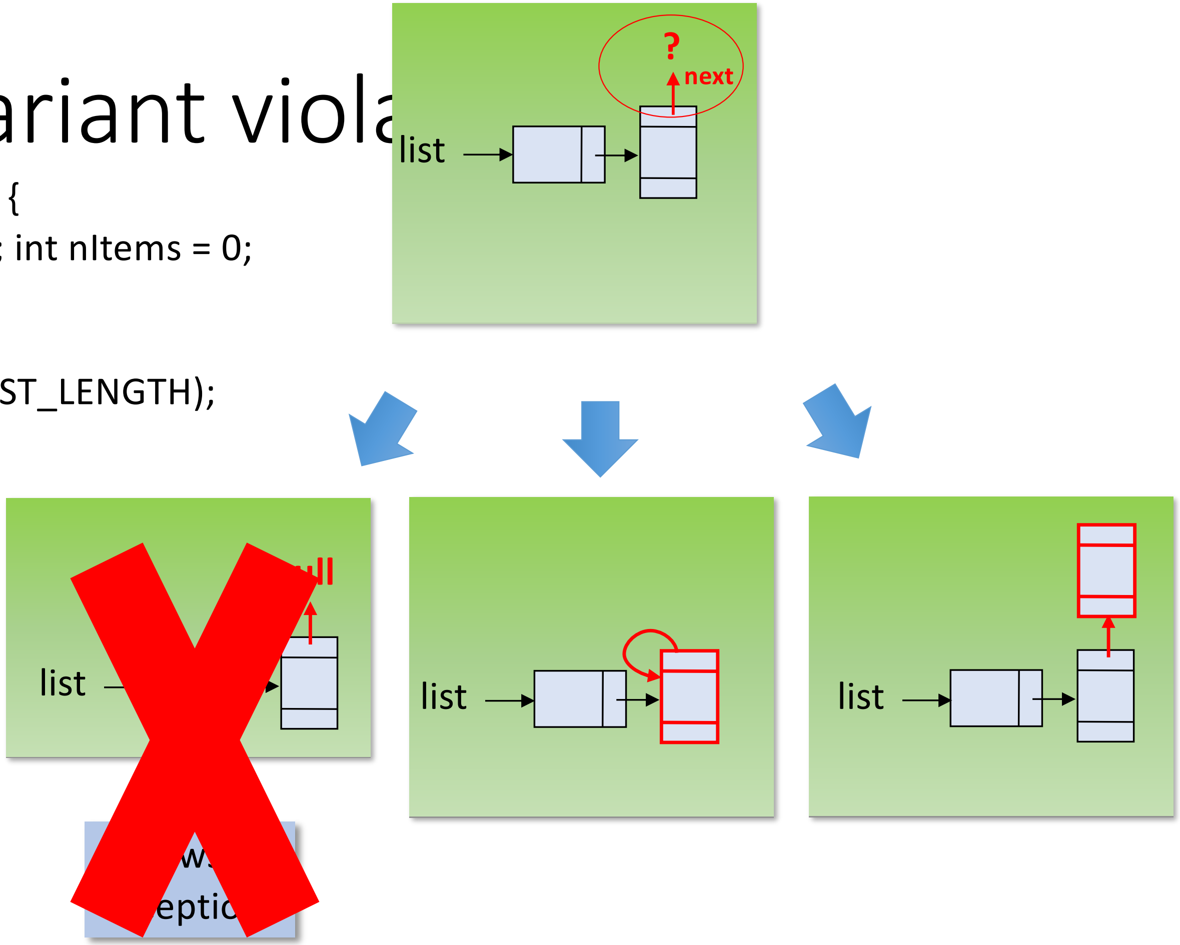
Representation invariant viola



```
int sum(LinkedList<Integer> list) {
    Integer item = null; int tot = 0; int nItems = 0;
    for (item : list) {
        tot += item.intValue();
        assert(++nItems <= MAX_LIST_LENGTH);
    }
    return tot;
}

public class LinkedList<Z> {
    int size = 0;
    Entry header = new Entry();
    class Entry {
        Entry next, prev;
        Z value;
    }
    …
}
```

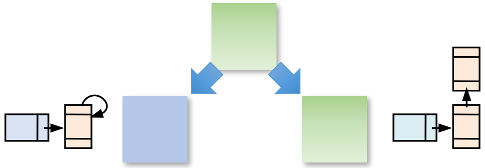
Representation invariant viola

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}

public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
   …
}
```



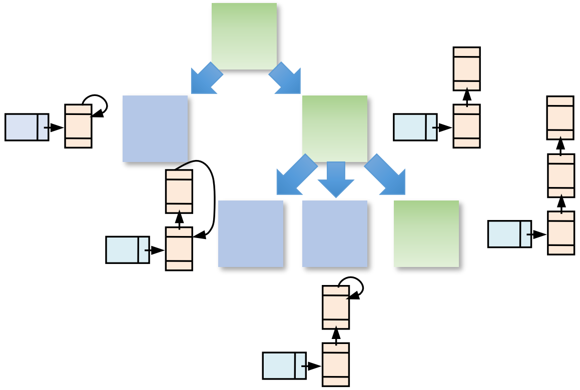
REPRESENTATION INVARIANT VIOLATION

…and here are the false alarms



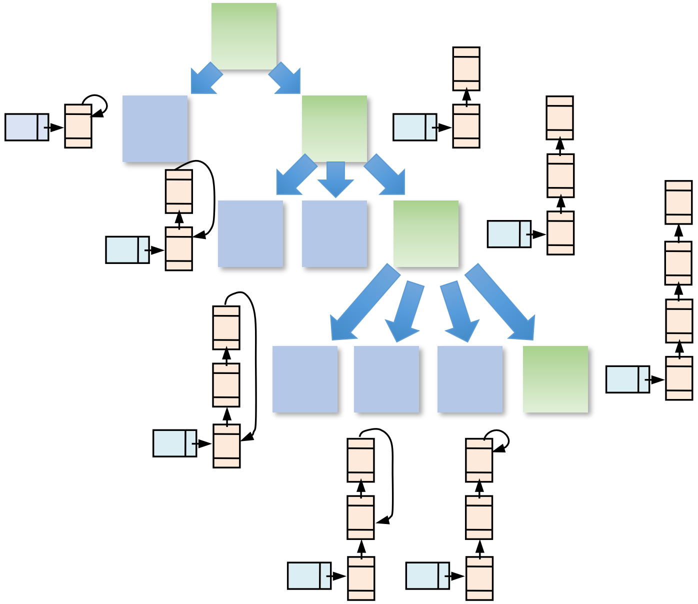
#pathsTot = 1

…and here are the false alarms



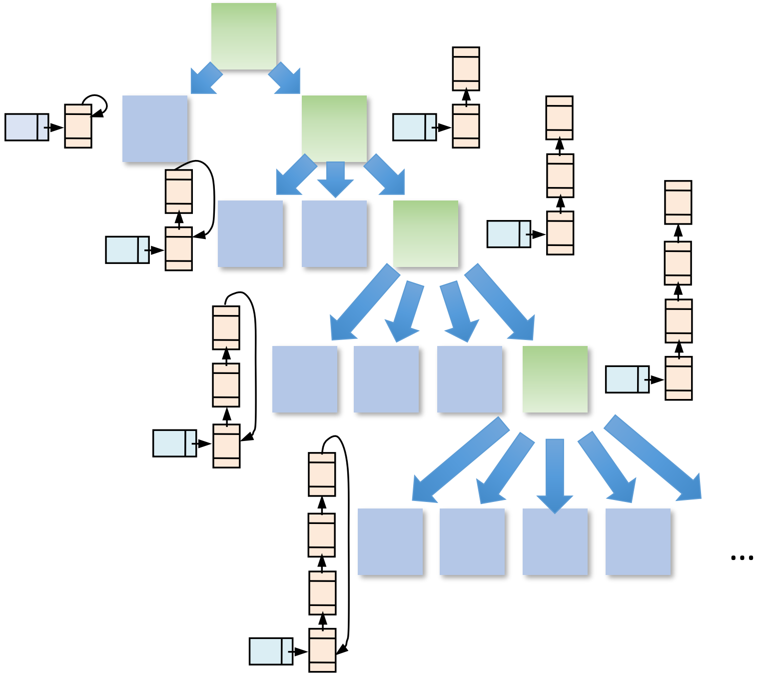
#pathsTot = 1 + 2

…and here are the false alarms



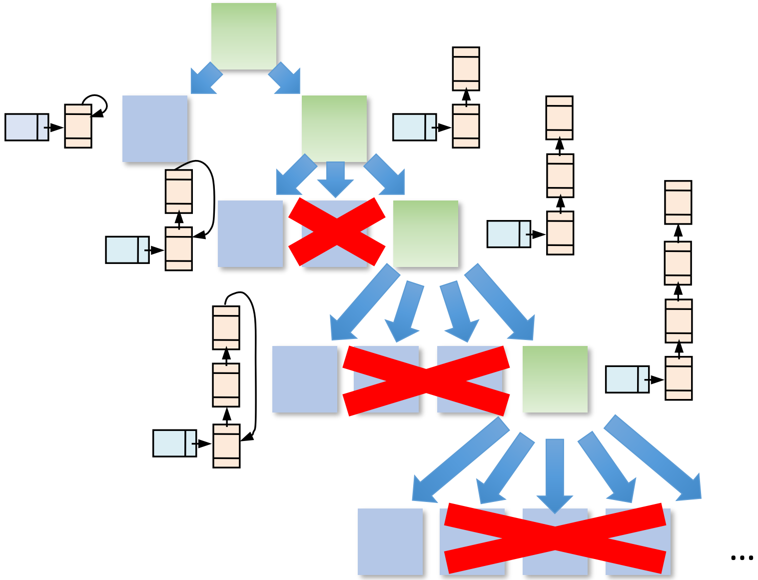
#pathsTot = 1 + 2 + 3

…and here are the false alarms



#pathsTot = $1 + 2 + 3 + 4 \ldots = O(\text{MAX\_LIST\_SIZE}^2)$

…and here are the false alarms



#pathsTot = 1 + 2 + 3 + 4 … = O(MAX_LIST_SIZE^2)

#pathsOk = 1 + 1 + 1 + 1 … = **O(MAX_LIST_SIZE)**

Possible approaches

- Inject representation invariants as background knowledge in the decision procedure / solver
- The assume repOk technique
- The conservative repOk technique (Visser, Pasareanu, Khurshid, ISSTA 2004)

Exploiting the solver

- It can be done if the representation invariants on the inputs can be expressed as formulas that the decision procedure / solver can reason upon
- Unfortunately, many kind of common invariants are hard to express and reason upon
  - There is no mainstream logic of pointers that is sufficiently expressive and has a good solver
  - All the current SMT solver support very weak, insufficient logics
  - Some constraints that cannot be expressed: cyclicity, reachability, counting
  - More powerful logics are still experimental (see e.g. Rakamaric, Bingham, Hu, VMCAI 2007)

The assume repOk technique

- An utmost trivial technique (with many disadvantages)
- A **repOk** method is a method in the programming language of the program under analysis that checks a representation invariant:
  - Takes as input an object of a given class
  - Returns true iff the object satisfies the representation invariant of the class
  - Returns false otherwise
  - (note that it must always terminate, even if the input object is ill-formed!!!)
- The assume repOk technique boils down to introduce an assume(repOk(input)) statement at the beginning of the program under analysis
- The rationale is that only the computations for which the input satisfies the representation invariant will survive this statement

Assume repOk, example

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}
```

```
public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }
}
```

Assume repOk, example

```
int sum(LinkedList<Integer> list) {
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}
```

```
public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }

   public boolean repOk() {
      int i = 0;
      for (Entry e = header; e != header; e = e.next) {
         if (e == null) return false;
         if (e.next == null) return false;
         if (e.next.prev != e) return false;
         ++i;
      }
      return (size == i - 1);
   }
}
```

Assume repOk, example

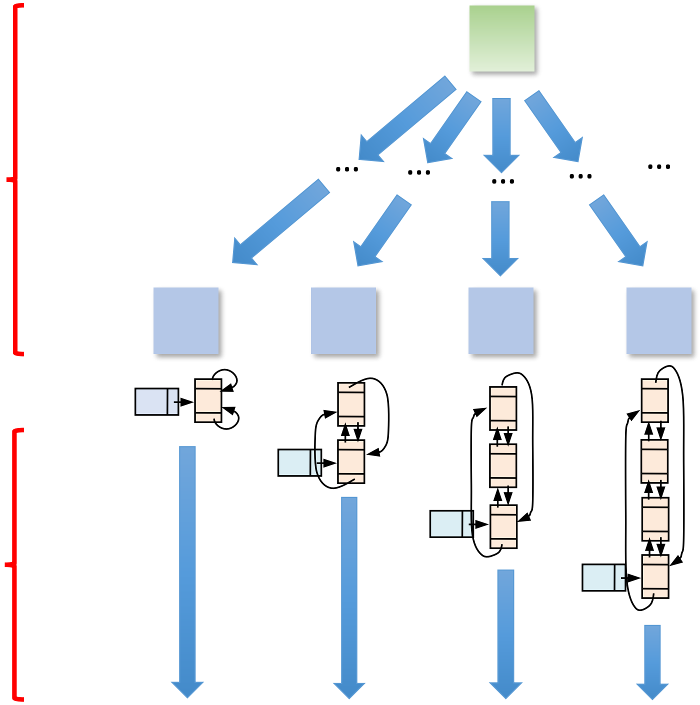
```
int sum(LinkedList<Integer> list) {
   assume(list.repOk());
   Integer item = null; int tot = 0; int nItems = 0;
   for (item : list) {
      tot += item.intValue();
      assert(++nItems <= MAX_LIST_LENGTH);
   }
   return tot;
}
```

```
public class LinkedList<Z> {
   int size = 0;
   Entry header = new Entry();
   class Entry {
      Entry next, prev;
      Z value;
   }

   public boolean repOk() {
      int i = 0;
      for (Entry e = header; e != header; e = e.next) {
         if (e == null) return false;
         if (e.next == null) return false;
         if (e.next.prev != e) return false;
         ++i;
      }
      return (size == i - 1);
   }
}
```

Assume repOk, pros and cons

- Pros:
  - All the expressive power of a programming language
  - Does not need any special support from decision procedure
  - Does not need any special support from symbolic executor
- Cons:
  - Writing a correct repOk method can be tricky: Are you really sure that the LinkedList.repOk() method in the previous slide always terminates?
  - Interplays badly with generalized symbolic execution: The execution of the repOk method completely materializes the input, and symbolic execution of target code degenerates in concrete execution

Assume repOk, interplay with GSE



Assume repOk statement

Rest of the program

The conservative repOk technique

- If an initial assume repOk statement fully materializes the input…
- …it is because the repOk method accesses all the fields in the input to perform its check
- Idea: the repOk method is not allowed to access a field of the input if it was not already accessed before
  - Result: no materializations during the conservative repOk execution, only the code under analysis may materialize a reference
  - As the repOk method arrives to an unaccessed field, it “gives up” checking the data structure and conservatively returns true
  - When the target program will access a field for the first time (and a reference is materialized), the conservative repOk method is rerun to check the input

Conservative repOk, pros and cons

- Pros:
  - All the expressive power of a programming language
  - Does not require support from the decision procedure
  - Better than assume repOk (yields “more symbolic” results)
- Cons:
  - Requires some support from the symbolic executor (automatically rerun the the conservative repOk upon every reference materialization)
  - Might be slow because of the many reruns of conservative repOk methods
  - If writing a correct repOk can be hard, writing a correct conservative repOk can be even harder…
  - …and there is no (automatic) way to make a repOk into a conservative one

Conservative repOk, example

```
public class LinkedList<Z> {
    …
    public boolean repOk() {
        int i = 0;
        for (Entry e = header; e != header; e = e.next) {
            if (e == null) return false;
            if (e.next == null) return false;
            if (e.next.prev != e) return false;
            ++i;
        }
        return (size == i - 1);
    }
}
```

Conservative repOk, example

```
public class LinkedList<Z> {
   …
   public boolean repOk() {
      int i = 0;
      for (Entry e = header; e != header; e = e.next) {
         if (e == null) return false;
         if (e.next == null) return false;
         if (e.next.prev != e) return false;
         ++i;
      }
      return (size == i - 1);
   }

   public boolean conservativeRepOk() {
      if (!isMaterialized(this, "header")) return true;
      if (header == null) return false;
      Entry tmp = header;
      int i = 0;
      HashSet<Entry> workset = new HashSet<>();
      do {
         if (tmp != header && !workset.contains(tmp)) {
            workset.add(tmp);
         } else if (tmp != header && workset.contains(tmp)) {
            return false;
         }
         if (isMaterialized(tmp, "next")) {
            if (isMaterialized(tmp.next, "previous") &&
                 tmp.next.previous != tmp) {
               return false;
            }
            tmp = tmp.next;
         } else {
            return i < size;
         }
         ++i;
      } while (tmp != header);
      return (size == i – 1);
   }
}
```