Tutorial

Program Model Checking

Davino Junior   Luis Melo   Marcelo d'Amorim

Universidade Federal de Pernambuco (UFPE)

SBMF Tutorial, November 21, 2017, Recife-PE, Brazil

Website

http://model-checking.dmtsj.com.br

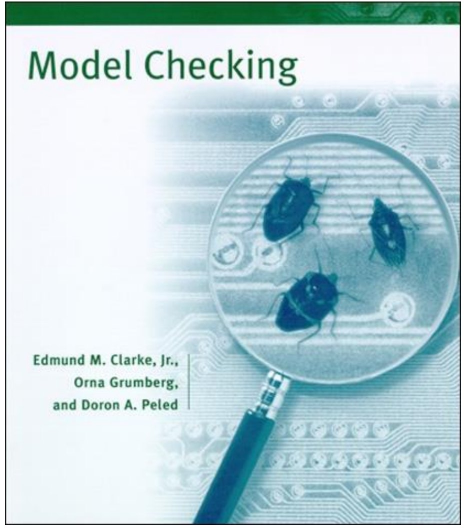
Model Checking (MC)

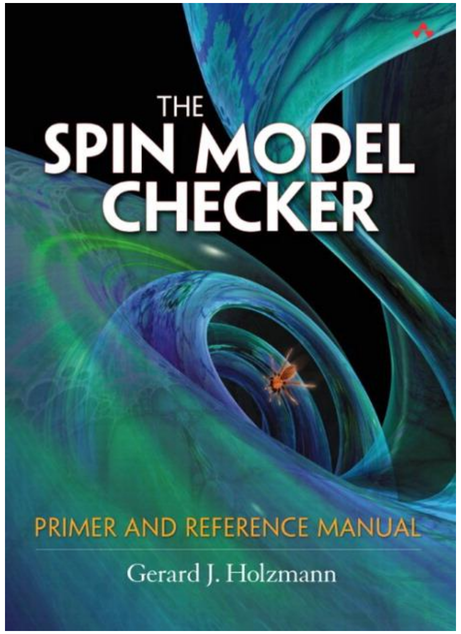
- Method for checking properties in system designs
  - Extremely popular in hardware verification!
- Intuition
  - Model is encoded as Labeled Transition System (LTS)
  - Property is described in some specification language (e.g., LTL, CTL)
  - Verification translated to graph algo

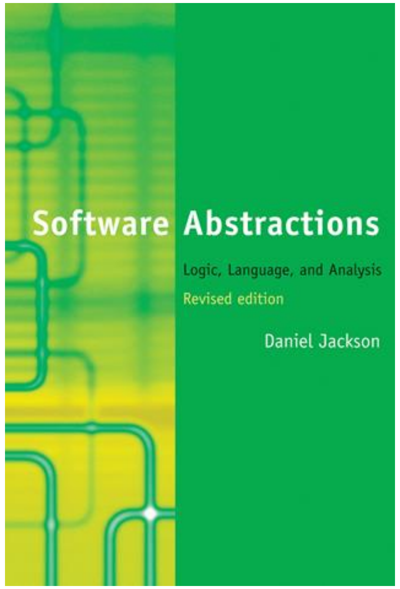
Model Checking (MC)

- Method for checking properties in system designs
  - Extremely popular in hardware verification!
- Intuition
  - Model is encoded as Labeled Transition System (LTS)
  - Property is described in some specification language (e.g., LTL, CTL)
  - Verification translated to graph algo

Vast literature!







Example

- Two symmetric processes with one shared semaphore
- Each process (1, 2) has three states:
  - Non-critical region – N1, N2
  - Trying to acquire semaphore – T1, T2
  - Critical region – C1, C2
- Semaphore can be available or not – S0 or S1
- Initial states: N1 & N2 & S0

Example

- Two symmetric processes with one shared semaphore
- Each process (1, 2) has three states:
  - Non-critical region – N1, N2
  - Trying to acquire semaphore – T1, T2
  - Critical region – C1, C2
- Semaphore can be available or not – S0 or S1
- Initial states: N1 & N2 & S0

It should always be possible to eventually get back to the initial state

Model:

N1 -> T1 T1 & S0 -> C1 & S1 C1 -> N1 & S0	||	N2 -> T2 T2 & S0 -> C2 & S1 C2 -> N2 & S0

Property: AG EF (N1 & N2 & S0)

This property holds for this model. This is verified searching the graph induced by model and property.

Big gap between design and implementation

-> Correct designs do not imply correct implementations
-> Systematic refinement of models is expensive (and rare)

Big gap between design and implementation

This tutorial is about Program Model Checking as opposed to Model Checking of Designs

Design Choices

- Path Exploration: Stateful or Stateless
- State Representation: Explicit or Symbolic

Tradeoff between time and space

- Handling Concurrency
  - Often important source of problems. E.g., data races and deadlocks
- Programming Language
  - Can make a huge difference in complexity. Think of pointers, dynamic binding, reflection, native methods, libraries, etc.

This Tutorial

	Language	State Representation	Concurrency
JPF [1]	Java	Explicit	✓
SPF [2]	Java	Symbolic	✗
CBMC [3]	C ou Java	Symbolic	✓

[1] Java Pathfinder website. https://babelfish.arc.nasa.gov/trac/jpf/
[2] Symbolic Pathfinder. https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[3] CBMC website. https://github.com/diffblue/cbmc/

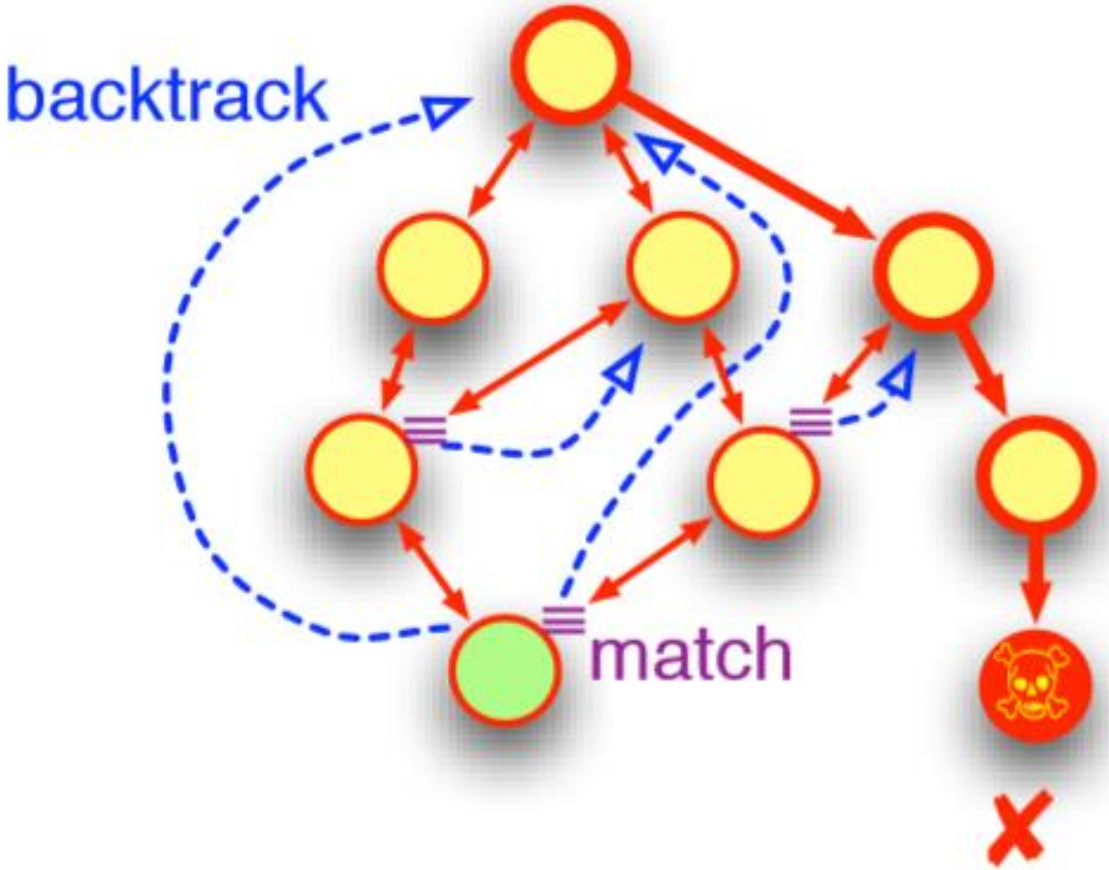
Java Pathfinder (JPF)

What is JPF?

- Virtual Machine for Java with several tweaks;
  - Theoretically executes a Java program in all possible ways
- Checks for concurrency issues (e.g., deadlocks, and race conditions)

Essential capabilities

- Backtracking
  - JPF can restore previous execution states, to see if there are unexplored choices left.
  - JPF allows you to create your own ChoiceGenerator.
  - JPF allows you to explore in a DFS or your own search heuristic.
- State matching
  - JPF checks every new state if it already has been explored, in which case is not explored anymore.



Example Race Condition

```
public class MyRaceCondition {
  private static class Pair {
    String x = "x";
    String y = "y";

    public void update() {
      x = x + y + x;
    }
  }

  private static class RC extends
  Thread {
    Pair p;

    public void run() {
      p.update();
    }
  }
```

```
  public static void main(String[] args) {
    Pair p = new Pair();
    RC rc1 = new RC();
    RC rc2 = new RC();

    rc1.p = p;
    rc2.p = p;

    rc1.start();
    rc2.start();
    rc1.join();
    rc2.join();
    System.out.println("x: " + p.x);
  }
}
```

Example Race Condition

```
============================================= search started:

x: xyxyxyx
x: xyxyxyx

============================================= error 1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer$Pair@15e.x
  Thread-1 at Racer$Pair.update(Racer.java:7)
                  "x = x + y + x;"  WRITE: putfield Racer$Pair.x
  Thread-2 at Racer$Pair.update(Racer.java:7)
                  "x = x + y + x;"  READ:  getfield Racer$Pair.x
```

Example Deadlock

```
public class MyDeadlock {
  private static class Pair {
    String x = "x";
    String y = "y";

    public void update() {
      x = x + y + x;
    }
  }

  private static class RC1 extends
  Thread {
    Pair p;

    public void run() {
         synchronized(p.x) {
         synchronized(p.y) {
               p.update(); }}}
  }

private static class RC2 extends
Thread {
    Pair p;

    public void run() {
         synchronized(p.y) {

         synchronized(p.x) {
               p.update(); }}}
 }

  public static void main(String[]
args) {
    Pair p = new Pair();
    Thread rc1 = new RC1();
    Thread rc2 = new RC2();

    rc1.p = p;
    rc2.p = p;

    rc1.start();
    rc2.start();
    rc1.join();
    rc2.join();
    System.out.println("x: " + p.x);
  }
}
```

Example Deadlock

```
================================================ search started

x: xyxyxyx
x: xyxyxyx

================================================ error 1
gov.nasa.jpf.vm.NotDeadlockedProperty
deadlock encountered:
  thread java.lang.Thread:{id:0,name:main,status:WAITING,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread Racer$RC1:{id:1,name:Thread-1,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  thread Racer$RC2:{id:2,name:Thread-2,status:BLOCKED,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
```

There is no free lunch!

- Scalability Issues – Time and Space
  - Time: Your program will run on top a tweaked JVM on top of the JVM.
  - Space: Path explosion.
- Many parts are not supported (need to model)
  - For example, I/O and GUI.

Demo

- Software Requirements:
  - Docker >= 17.09-CE

- Running:
  - Only output -> docker run –it --rm **lhsm/jpf-examples**
  - Playground -> docker run -it –rm **lhsm/jpf-examples bash**

Symbolic Pathfinder (SPF)

Motivation

- Automatic Input Generation
  - Manually generate inputs for testing **all** paths of a program can be unfeasible

Motivation

- Automatic Input Generation
  - Manually generate inputs for testing **all** paths of a program can be unfeasible

```
public void foo(int x){
     int z;
     if(x >= 10) {
            int y = x – 10;
            z = x / y;
    }
    else {
            ...
    }
}
```

Motivation

Triggers ArithmeticException (DivisionByZero)

- Automatic Input Generation
  - Manually generate inputs for testing **all** paths of a program can be unfeasible

```
public void foo(int x){
     int z;
     if(x >= 10) {
               int y = x – 10;
               z = x / y;
    }
    else {
              ...
    }
}
```

```
@Test
public void test1(){
   ...
   foo (1);
   ...
}
```

```
@Test
public void test3(){
   ...
   foo (10);
   ...
}
```

```
@Test
public void test2(){
   ...
   foo (15);
   ...
}
```

SPF: Symbolic Execution

- Symbolic Execution [1]
  - Execute the program on symbolic inputs
  - Symbolic values represent sets of concrete values
  - Build symbolic tree which encodes many execution paths

- For each possible execution path
  - Build a **path condition (PC)**
  - Check condition satisfability with **solvers**
  - Model checker backtracks if path becomes infeasible
  - Uses SMT solvers to get test inputs

[1] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**, 385–394 (1976).

Example

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

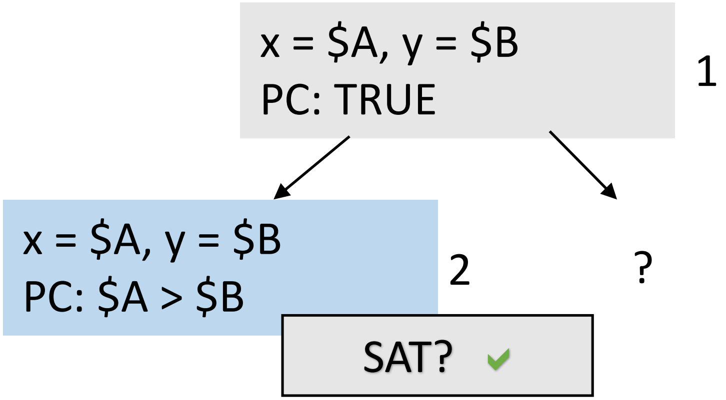
Program Counter

x = $A, y = $B
PC: TRUE

1

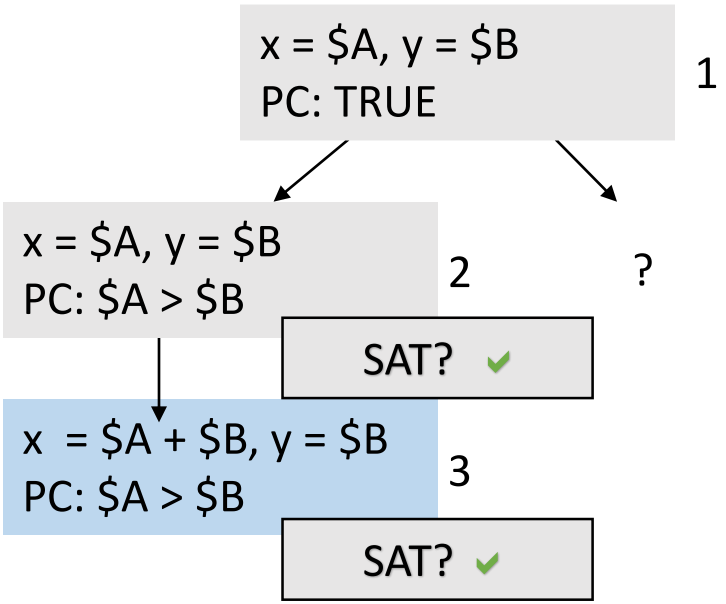
Example

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



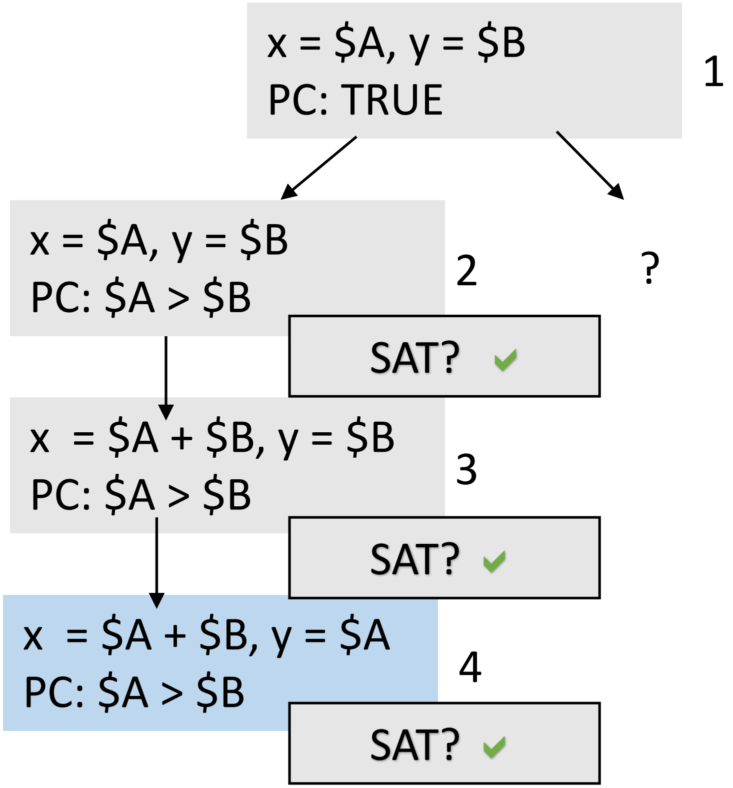
Example

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



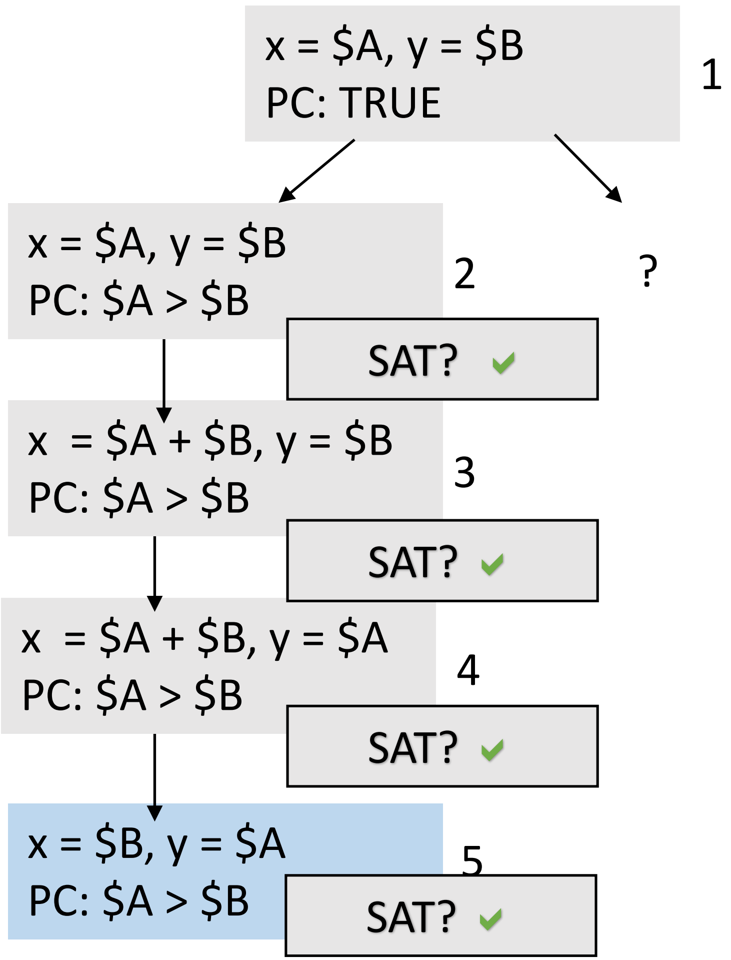
Example

```
int x, y
if(x > y){
   x = x + y;
   y = x – y;
   x = x – y;
   if(x > y){
         assert false;
   }
   print(x, y);
```



Example

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



Example

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

x = $A, y = $B
PC: TRUE
1

x = $A, y = $B
PC: $A > $B
2
SAT? ✔

?

x = $A + $B, y = $B
PC: $A > $B
3
SAT? ✔

x = $A + $B, y = $A
PC: $A > $B
4
SAT? ✔

x = $B, y = $A
PC: $A > $B
5
SAT? ✔

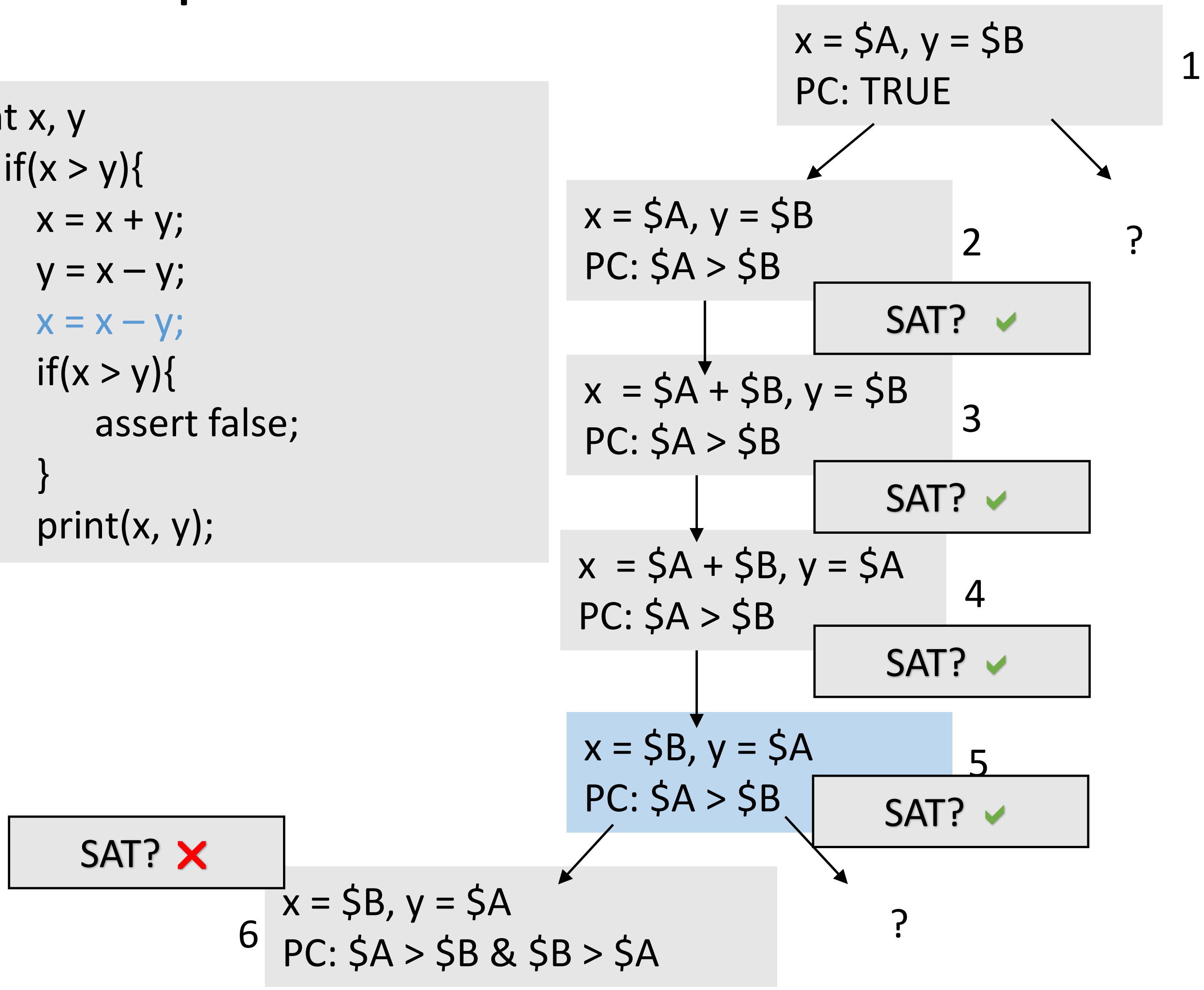
SAT? ✘

6
x = $B, y = $A
PC: $A > $B & $B > $A

?

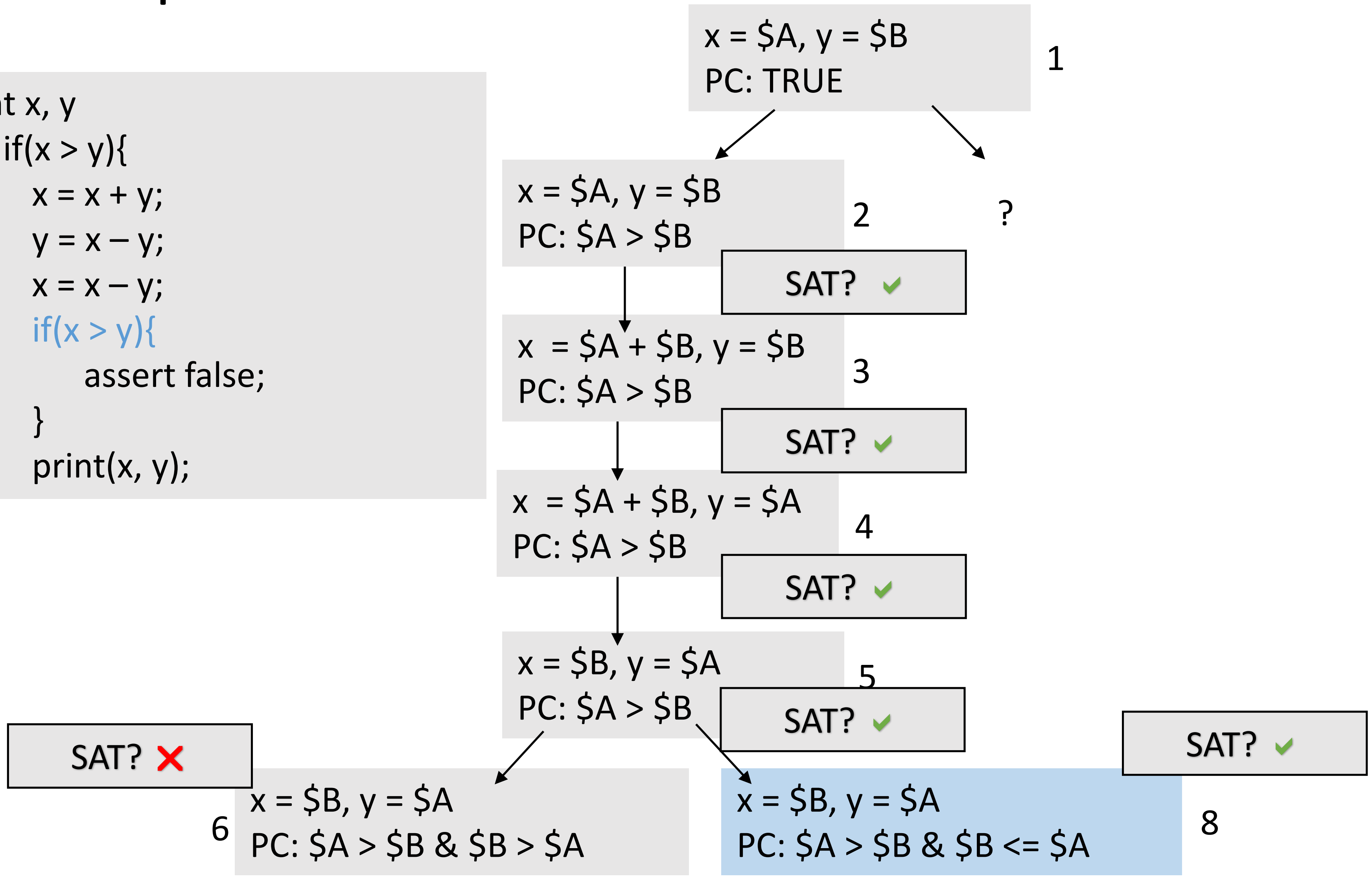
Example

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



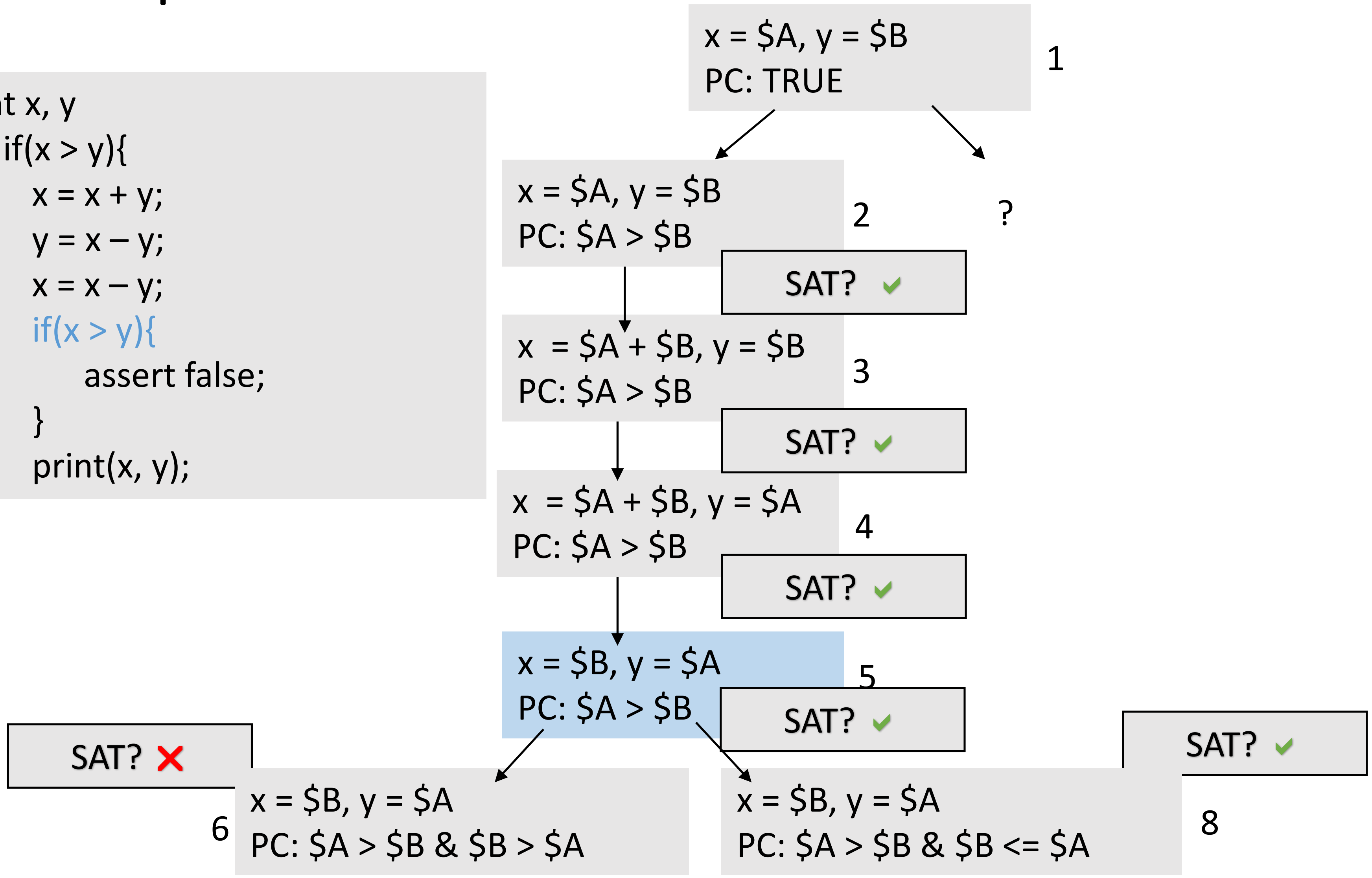
Example

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



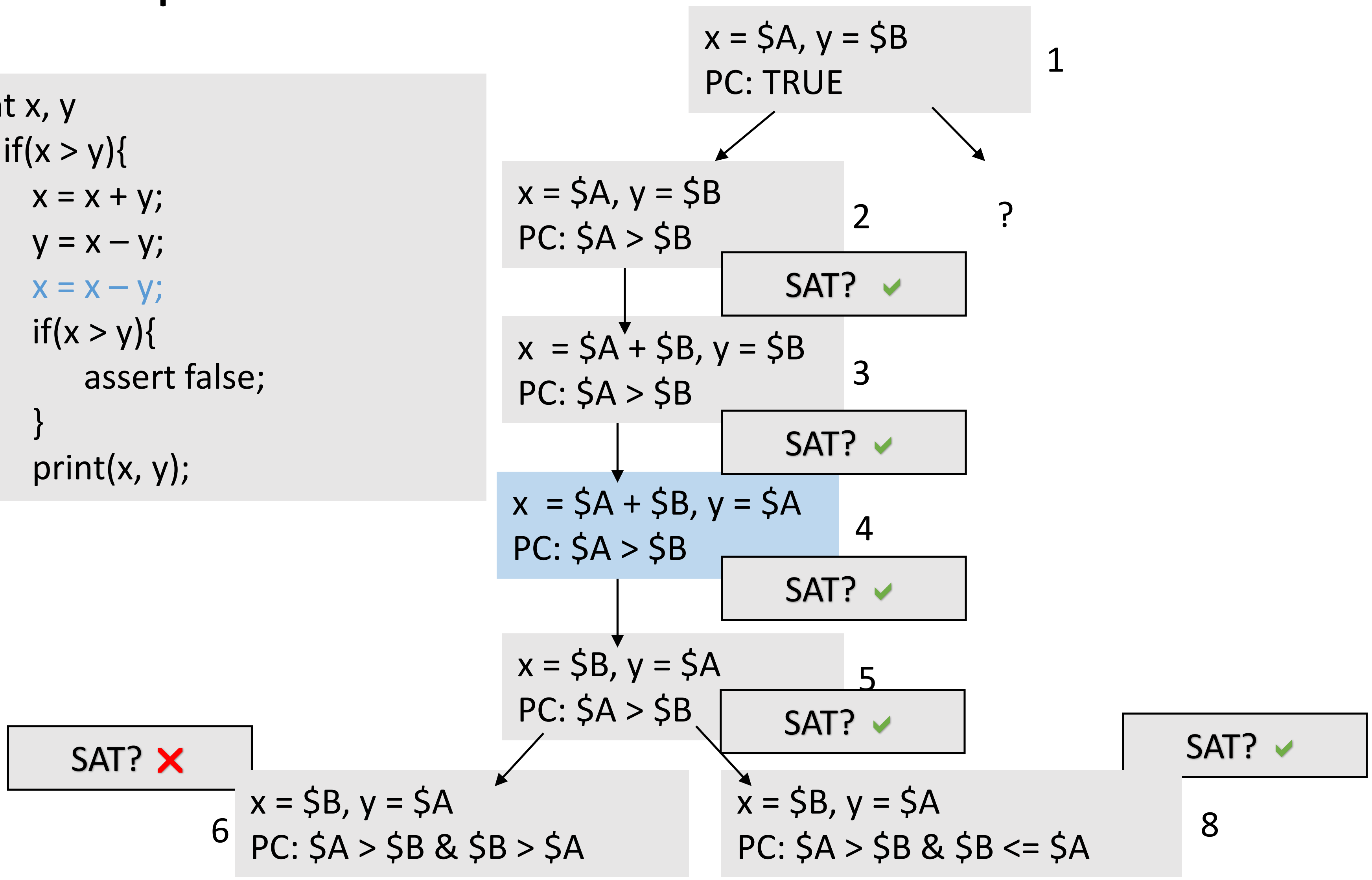
Example

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



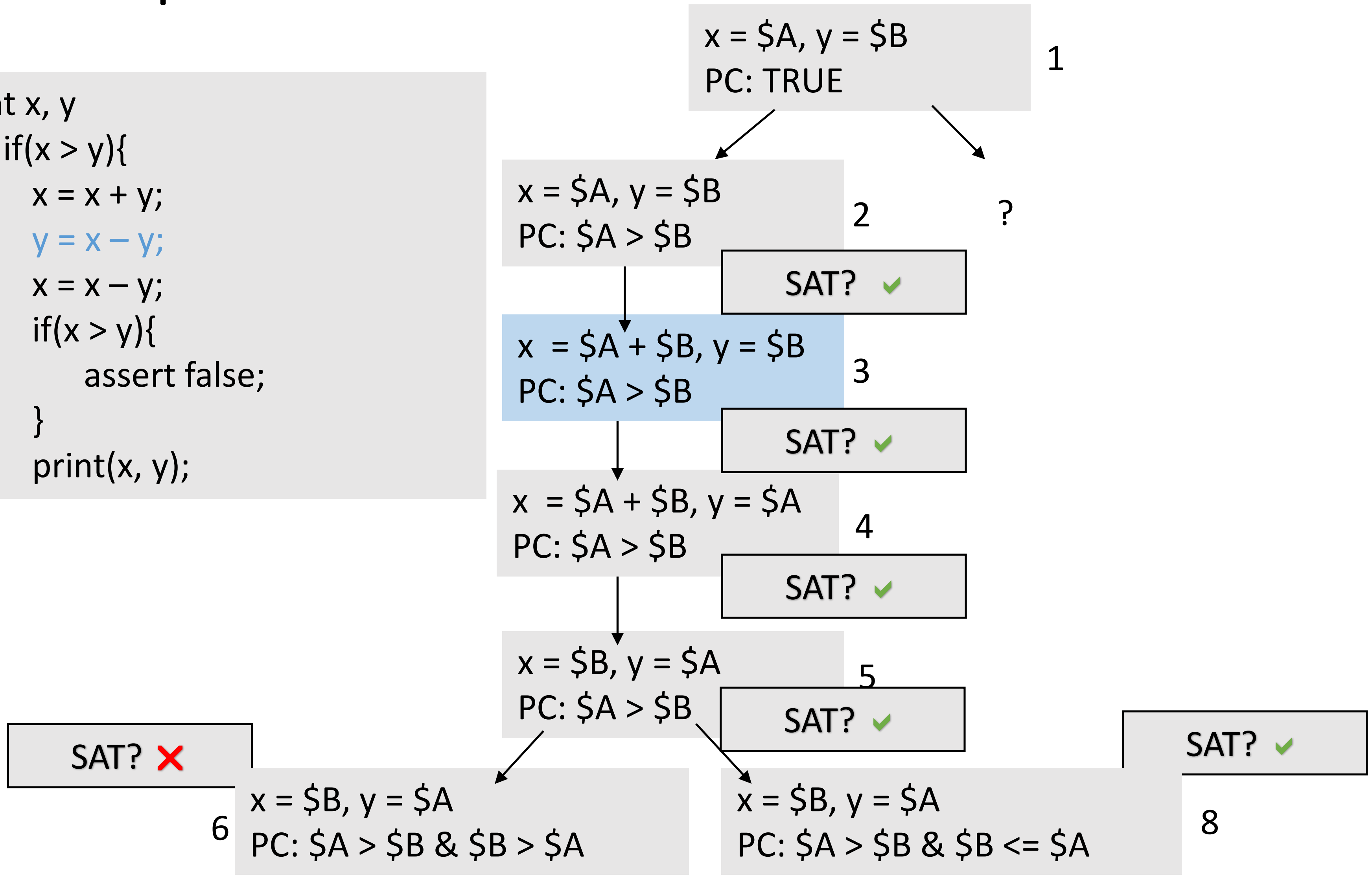
Example

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



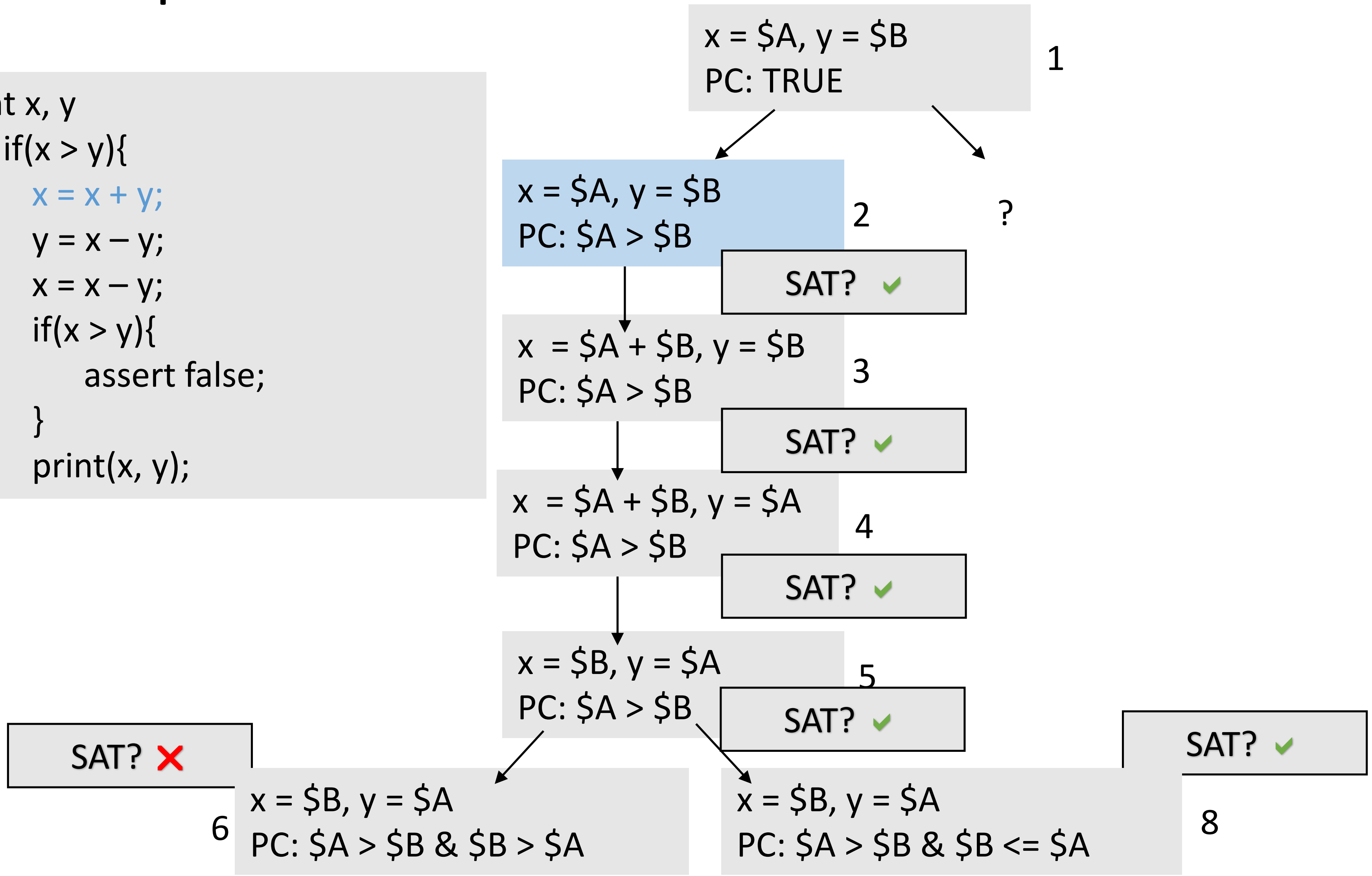
Example

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



Example

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



Example

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

x = \$A, y = \$B
PC: TRUE
1

x = \$A, y = \$B
PC: \$A > \$B
2
SAT? ✔

?

x = \$A + \$B, y = \$B
PC: \$A > \$B
3
SAT? ✔

x = \$A + \$B, y = \$A
PC: \$A > \$B
4
SAT? ✔

x = \$B, y = \$A
PC: \$A > \$B
5
SAT? ✔

SAT? ✘
6
x = \$B, y = \$A
PC: \$A > \$B & \$B > \$A

SAT? ✔
x = \$B, y = \$A
PC: \$A > \$B & \$B <= \$A
8

Example

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

x = $A, y = $B
PC: TRUE
1

x = $A, y = $B
PC: $A > $B
2
SAT? ✔

x = $A, y = $B
PC: $A <= $B
8
SAT? ✔

x = $A + $B, y = $B
PC: $A > $B
3
SAT? ✔

x = $A + $B, y = $A
PC: $A > $B
4
SAT? ✔

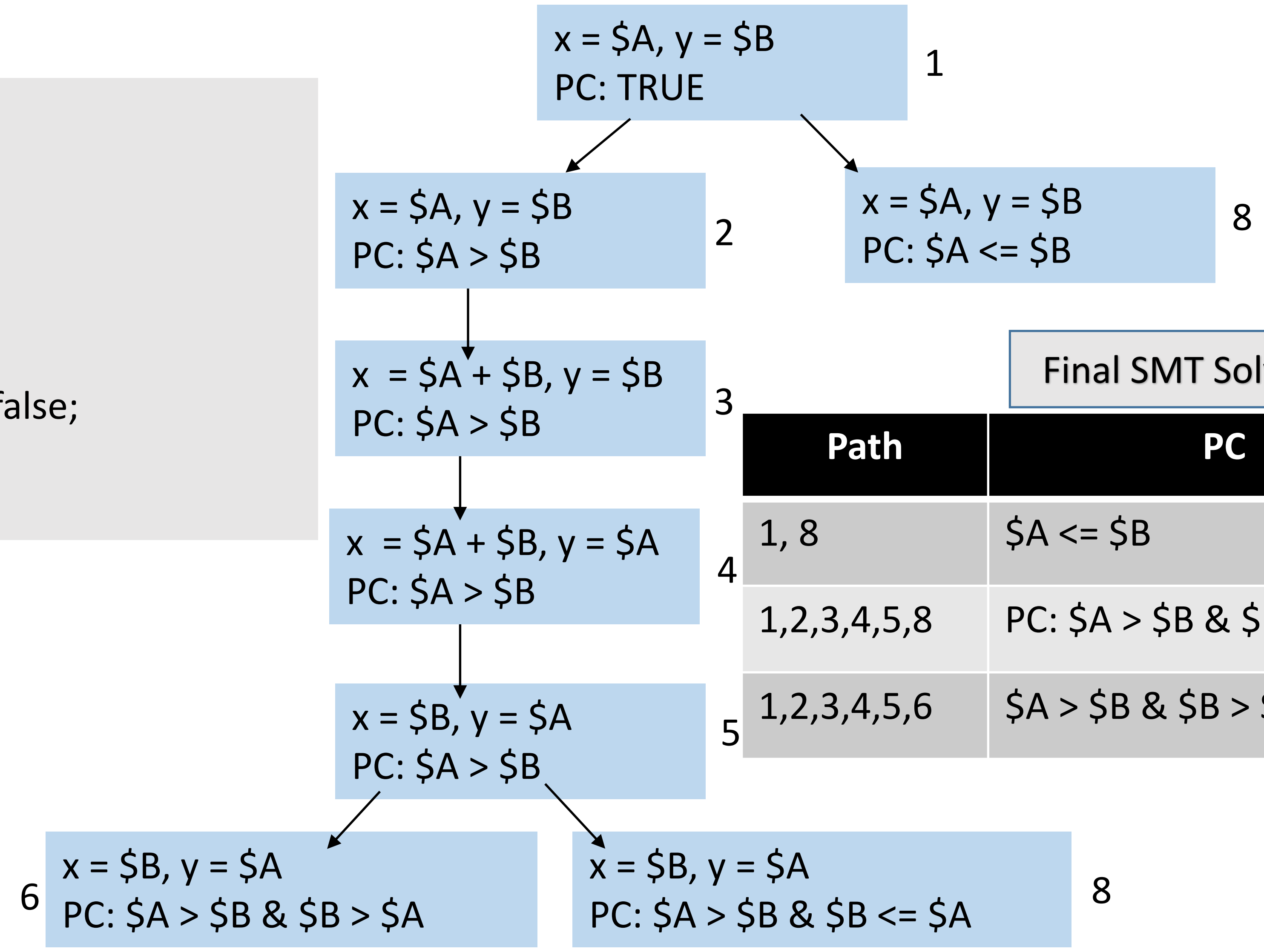
x = $B, y = $A
PC: $A > $B
5
SAT? ✔

SAT? ✘
6
x = $B, y = $A
PC: $A > $B & $B > $A

SAT? ✔
x = $B, y = $A
PC: $A > $B & $B <= $A
8

Example

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



Final SMT Solver Results

Path	PC	Program Input
1, 8	$A <= $B	$A = 1, $B = 1
1,2,3,4,5,8	PC: $A > $B & $B <= $A	$A = 2, $B = 1
1,2,3,4,5,6	$A > $B & $B > $A	-

Symbolic Pathfinder (SPF)

- Performs symbolic execution of Java bytecodes
- Built as a JPF module
  - Search engine used to explore symbolic execution tree
- Multiple decision procedures/contraint solvers
  - Used to check path conditions
- Test suites with high coverage
  - Generates JUnit tests

SPF: Implementation

- JPF infrastructure
  - Replaces/extend standard concrete with symbolic execution
- Attributes associated with program state
  - Fields, stack operands, variables stored as **symbolic information**
- Allows mixed concrete and symbolic execution
  - Can change from concrete to symbolic execution on-the-fly
- Listeners
  - Act as monitors for symbolic analysis

SPF

Pros (+)

- Automated
- Extesibility
- Availability (Open source)

Cons (-)

- JPF limitations

Demo

- Software Requirements:
  - Docker >= 17.09-CE

- Running:
  - Only Output -> docker run –it --rm **davinomjr/spf-examples**
  - Playground -> docker run -it –rm **davinomjr/spf-examples bash**

CBMC

Bounded Model Checking

- Translates program into formula (in some decidable theory) and uses a (SAT, SMT) solver to find counter-examples
- Compared to Symbolic Execution (SE):
  - Both BMC and SE: use symbolic inputs and use solvers to find bugs
  - BMC explores several paths simultaneously; SE explores paths one-at-a-time
    - Tradeoff between time and space

Bounded Model Checking of C [1]

```
x = x + 1
if (x != 1) {
 x = 2;
 if (z) x++;
}
assert (x <= 3);
```

goto program

```
x = x + 1
if (x != 1) {
 x = 2;
 if (z) x++;
}
assert (x <= 3);
```

SSA

```
x1 = x0 + 1
if (x1 != 1) {
 x2 = 2;
 if (z0) x3 = x2 + 1;
 x4 = phi(x3, x2)
}
x5 = phi(x4, x1)
assert (x5 <= 3);
```

[1] Clarke et al, DAC'03, Behavioral consistency of C and verilog programs using bounded model checking

Bounded Model Checking of C [1]

Logical Formula

C :=
x_1 = x_0 + 1 &
x_2 = 2 &
x_3 = x_2 + 1 &
x_4 = ite(z_0,x_3,x_2) &
x_5 = ite((x_1!=1),x_4,x_1)

P := x_5 <= 3;

C & !P

Bit-Blasting

SAT solver

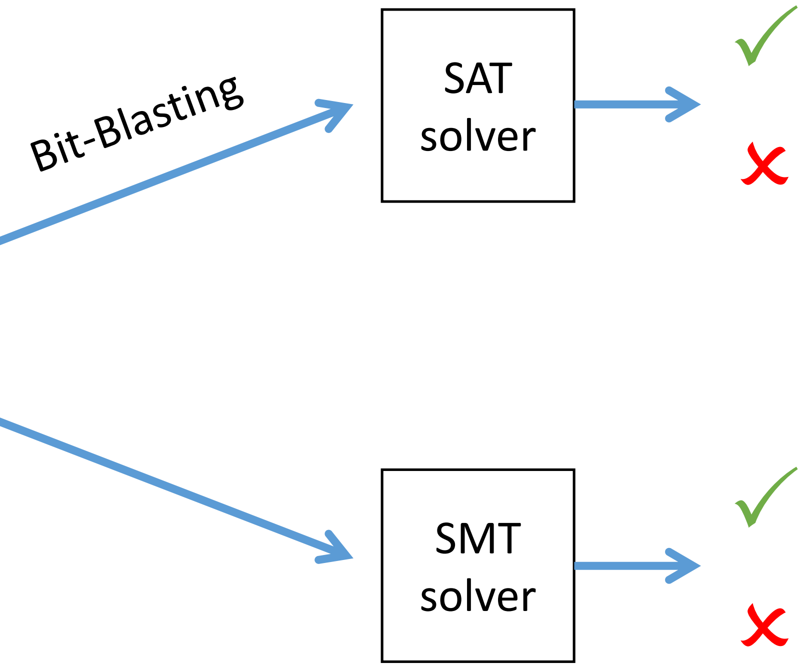
SMT solver

✓ counterexample

✗ properties hold up to some bound

[1] Clarke et al, DAC'03, Behavioral consistency of C and verilog programs using bounded model checking

Bounded Model Checking of C [1]



- Challenges
  - Scalability: Formula can grow big!
  - Precision: Pointers, reflection, etc.

[1] Clarke et al, DAC'03, Behavioral consistency of C and verilog programs using bounded model checking

Tutorial Website:

http://model-checking.dmtsj.com.br