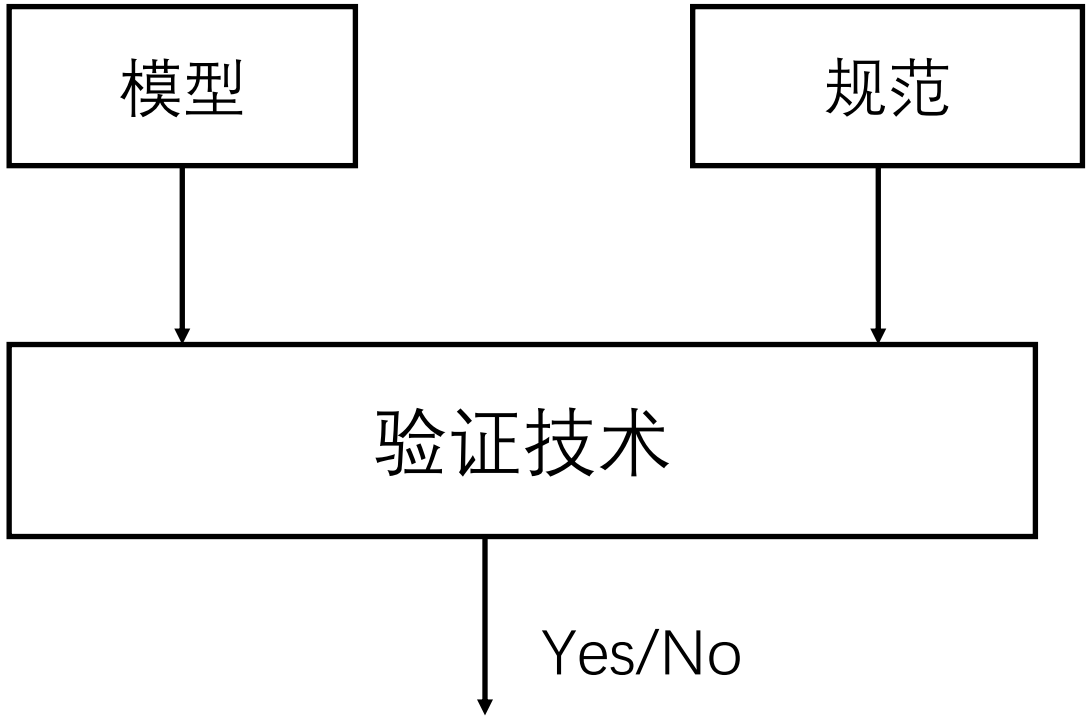
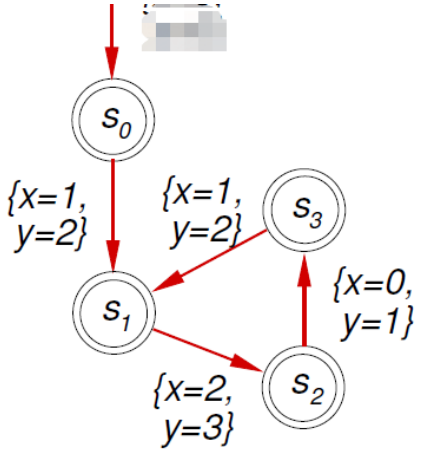
形式化验证技术

李建文

形式化验证

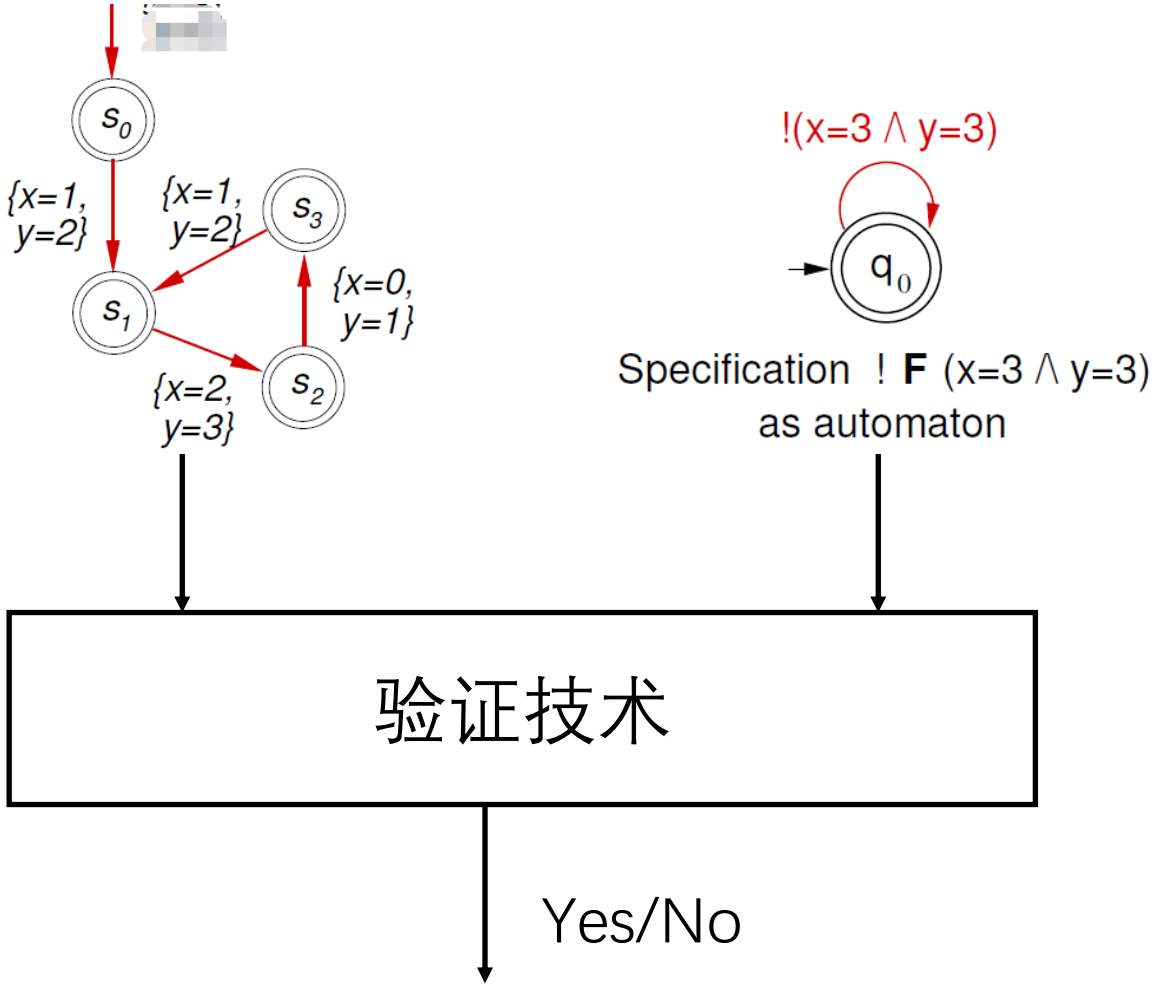


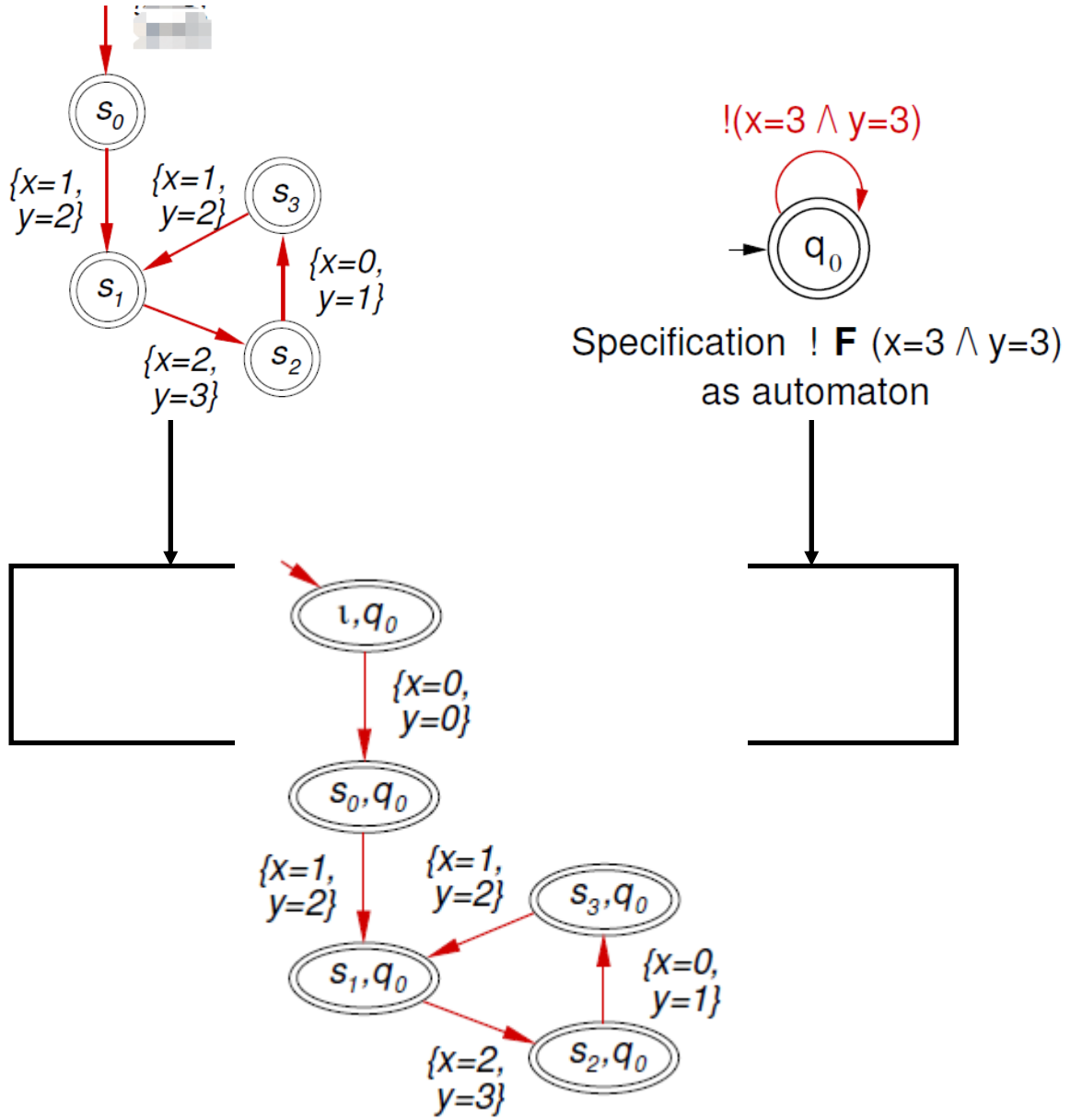


规范

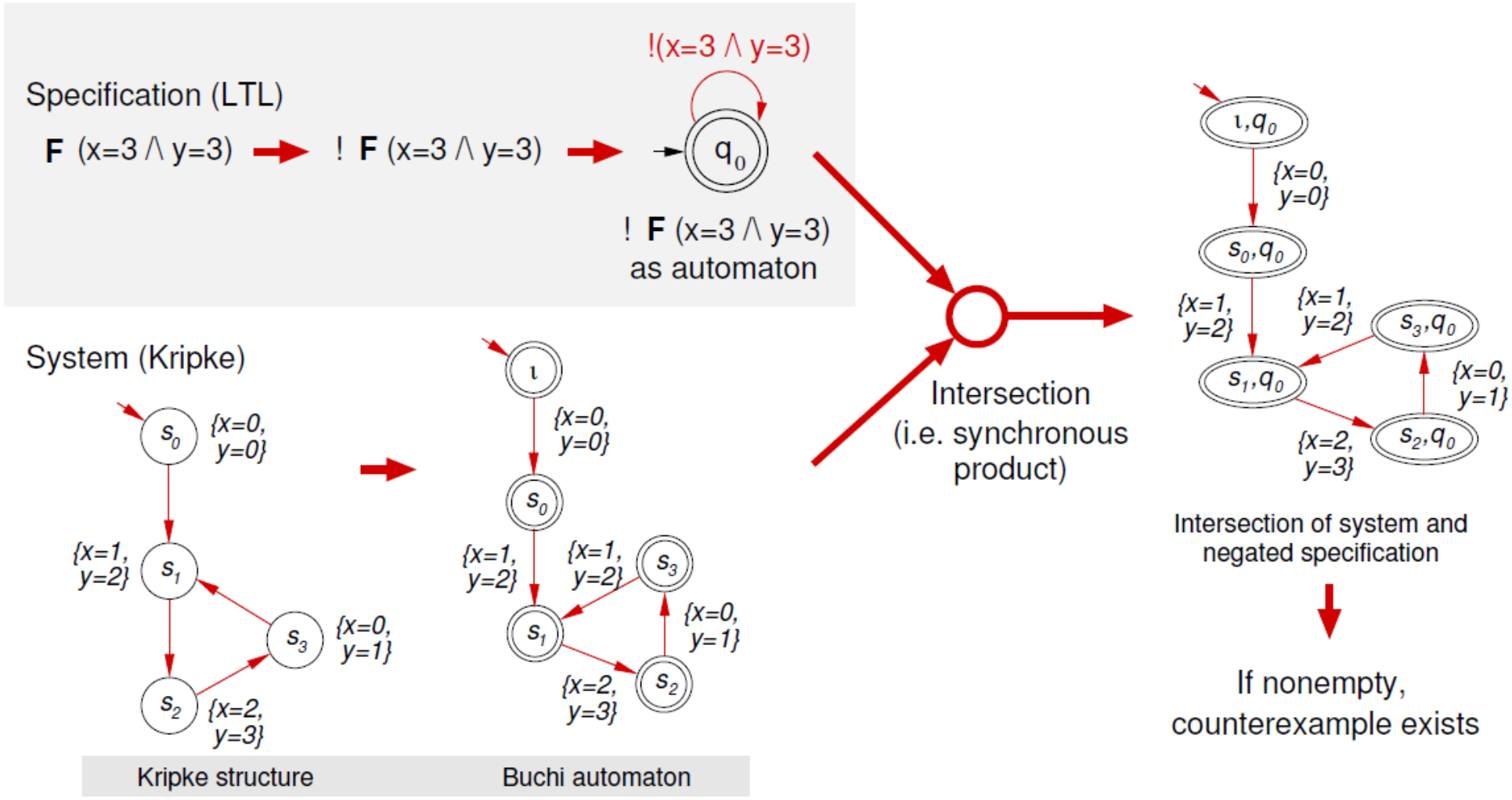
验证技术

Yes/No


s_0
{x=1, y=2}
{x=1, y=2}
s_3
s_1
{x=0, y=1}
{x=2, y=3}
s_2
!(x=3 ∧ y=3)
q_0
Specification ! **F** (x=3 ∧ y=3)
as automaton
验证技术
Yes/No


s_0
{x=1, y=2}
{x=1, y=2}
s_3
s_1
{x=0, y=1}
{x=2, y=3}
s_2
!(x=3 ∧ y=3)
q_0
Specification ! **F** (x=3 ∧ y=3)
as automaton
ι,q_0
{x=0, y=0}
s_0,q_0
{x=1, y=2}
{x=1, y=2}
s_3,q_0
s_1,q_0
{x=0, y=1}
{x=2, y=3}
s_2,q_0

基于自动机的模型检查技术



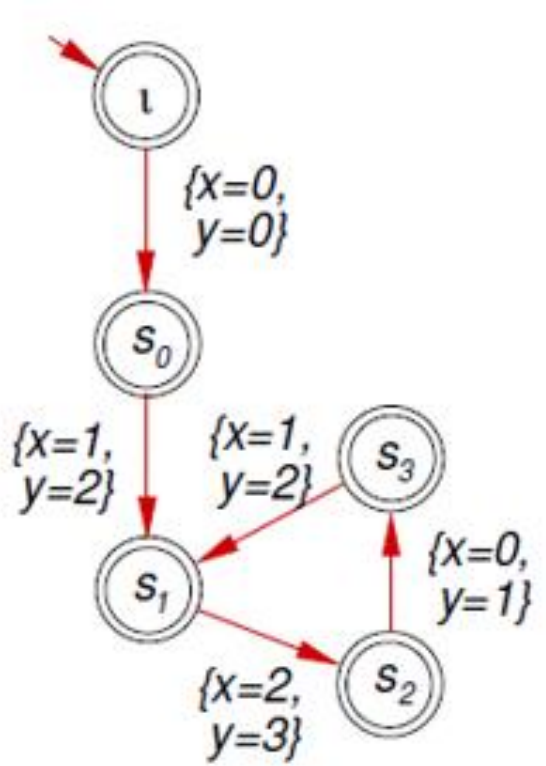
模型

- x := 0; y := 0;
- x +=1; y += 2;
- while (1){
  - if (x <= 1)
    - x += 1, y += 1;
  - else
    - x -= 2, y -= 2;
- }

画出对应的自动机

模型

- x := 0; y := 0;
- x +=1; y += 2;
- while (1){
  - if (x <= 1)
    - x += 1, y += 1;
  - else
    - x -= 2, y -= 2;
- }



性质规范

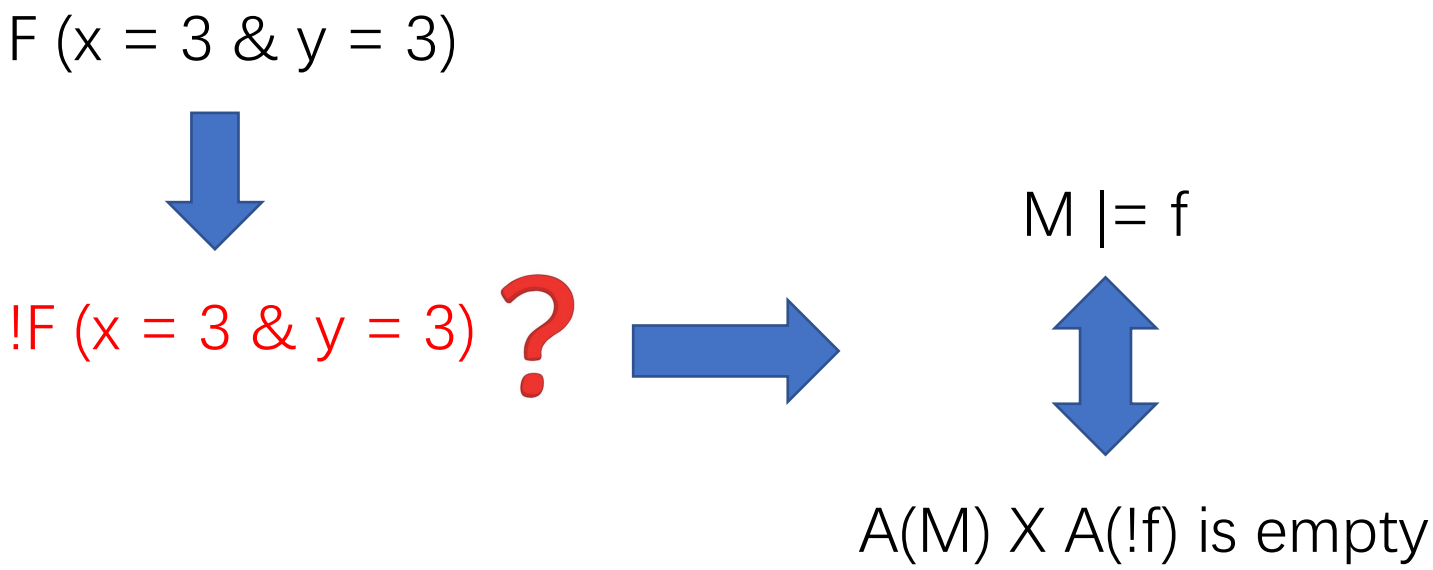
$$F\ (x = 3\ \&\ y = 3)$$

性质规范

F (x = 3 & y = 3)

!F (x = 3 & y = 3) ?

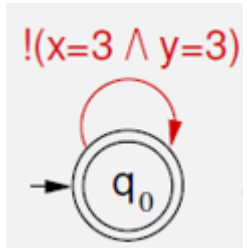
性质规范



性质规范

F (x = 3 & y = 3)

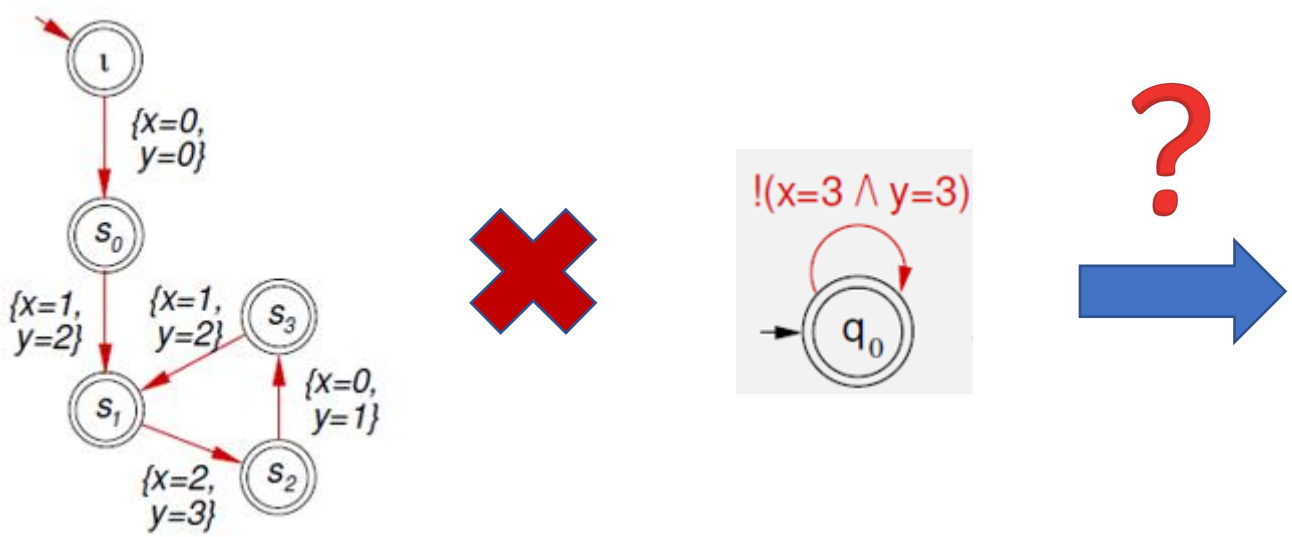
!F (x = 3 & y = 3) ?



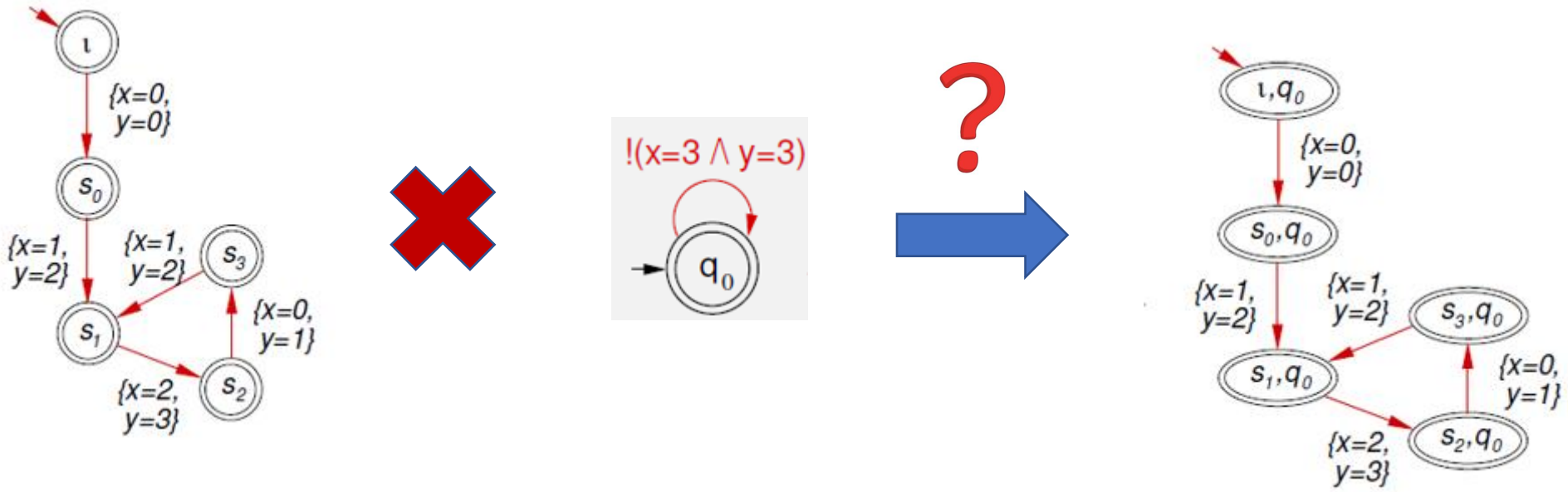
M |= f

A(M) X A(!f) is empty

自动机交集



自动机交集


ι
{x=0, y=0}
s_0
{x=1, y=2}
{x=1, y=2}
s_3
s_1
{x=0, y=1}
{x=2, y=3}
s_2
!(x=3 ∧ y=3)
q_0
?
ι,q_0
{x=0, y=0}
s_0,q_0
{x=1, y=2}
{x=1, y=2}
s_3,q_0
s_1,q_0
{x=0, y=1}
{x=2, y=3}
s_2,q_0

性质规范

GF !(x = 0)

性质规范

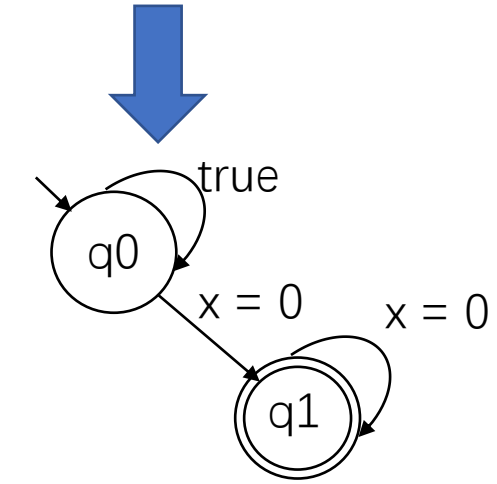
GF !(x = 0)

FG (x = 0)

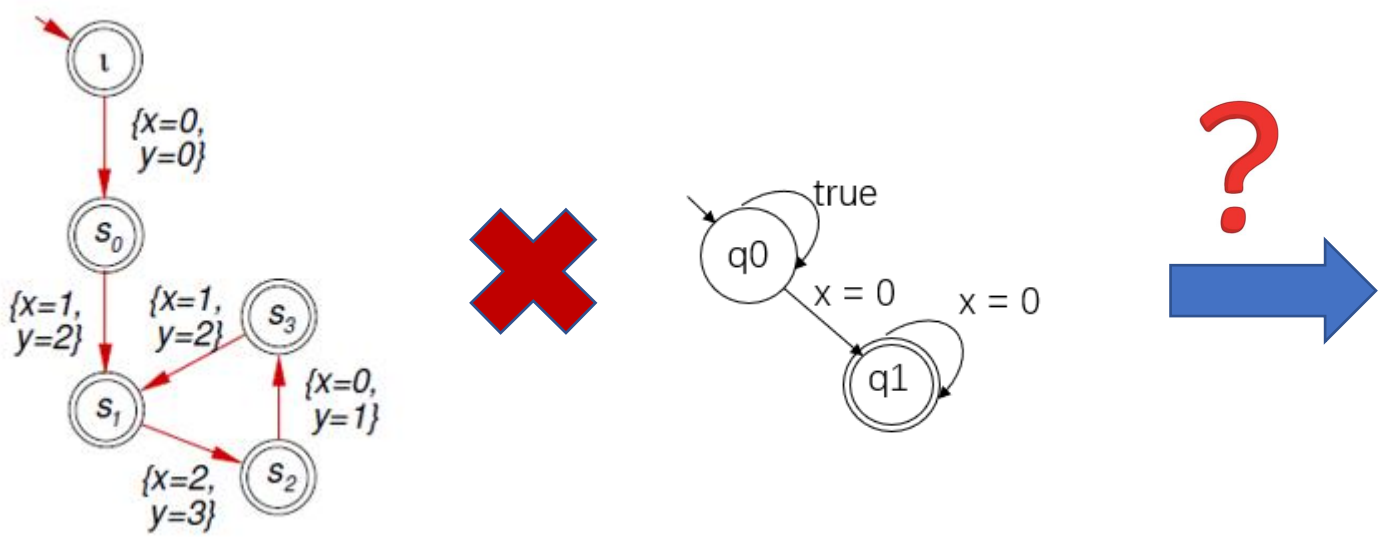
性质规范

GF !(x = 0)

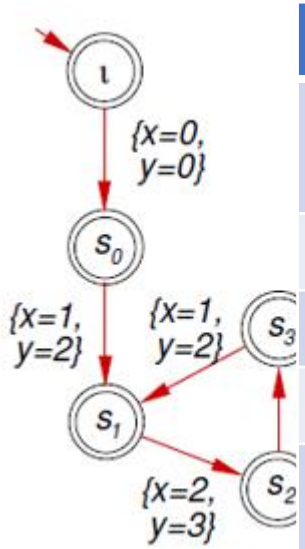
FG (x = 0)



自动机交集

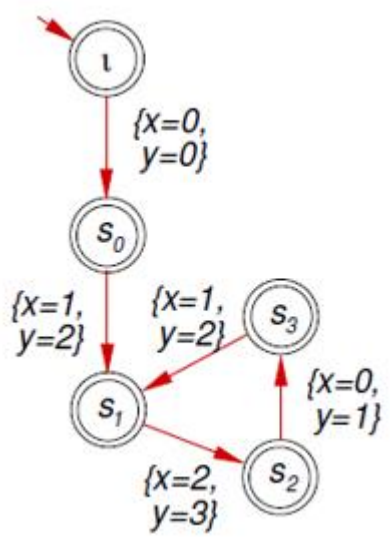


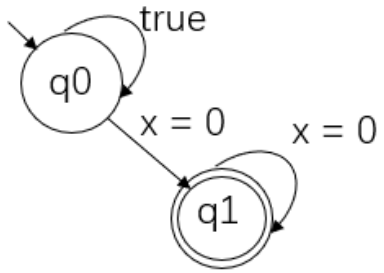
自动机交集

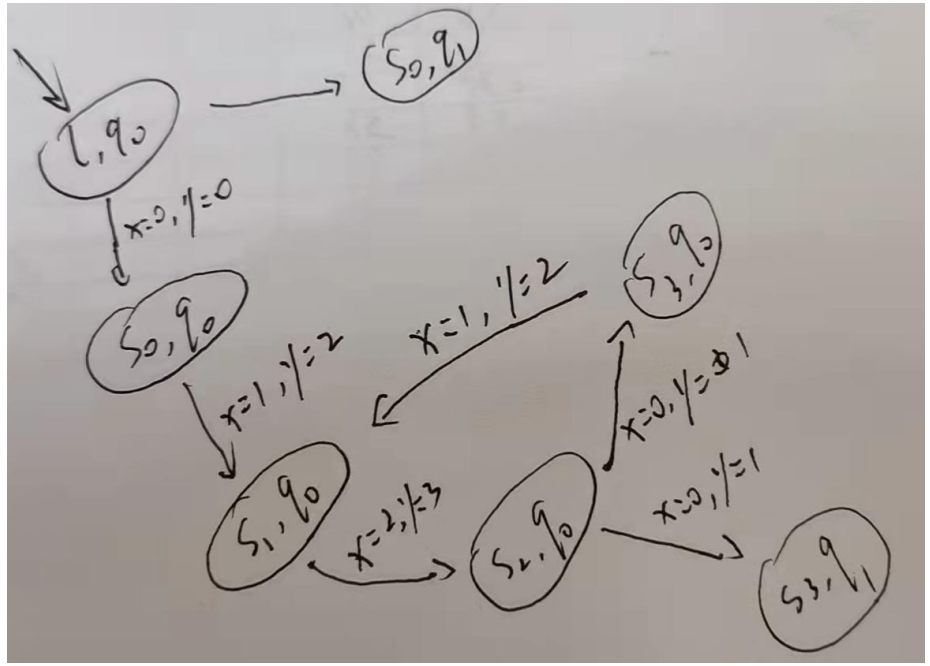


	x=0,y=0	x=1,y=2	x=0,y=1	x=2,y=3
(l,q0)	(s0, q0) (s0, q1)	-	-	-
(s0, q0)	-	(s1,q0)	-	-
(s0, q1)	-	-	-	-
(s1, q0)	-	-	-	(s2, q0)
(s2, q0)	-	-	(s3, q0) (s3, q1)	-
(s3, q0)	-	(s1, q0)	-	-
(s3, q1)	-	-	-	-

自动机交集







软件生命周期

1. 软件定义
2. 软件开发
3. 软件测试
4. 软件验证
5. 软件维护

软件验证

系统验证需要以用户为主体，以需求规格说明书中对软件的定义为依据，由此对软件的各项规格进行逐项地确认，以确保已经完成的软件系统与需求规格的一致性。

规范
(Specification)

软件验证

系统验证需要以用户为主体，以需求规格说明书中对软件的定义为依据，由此对软件的各项规格进行逐项地确认，以确保已经完成的软件系统与需求规格的一致性。

模型(Model)

软件验证

系统验证需要以用户为主体，以需求规格说明书中对软件的定义为依据，由此对软件的各项规格进行逐项地确认，以确保已经完成的软件系统与需求规格的一致性。

模型(Model)是否满足规范(specification)?

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）

如果一个程序P的输入变量有10个，均为整型变量，那么在64位机器上测试用例约有多少组？

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）

如果一个程序P的输入变量有10个，均为整型变量，那么在64位机器上测试用例约有多少组？

$$(2^{64})^{10} \approx (10^3)^{64} = 10^{192} >> 2^{100}$$

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）

如果一个程序P的输入变量有10个，均为整型变量，那么在64位机器上测试用例约有多少组？

$$(2^{64})^{10} \approx (10^3)^{64} = 10^{192} >> 2^{100}$$

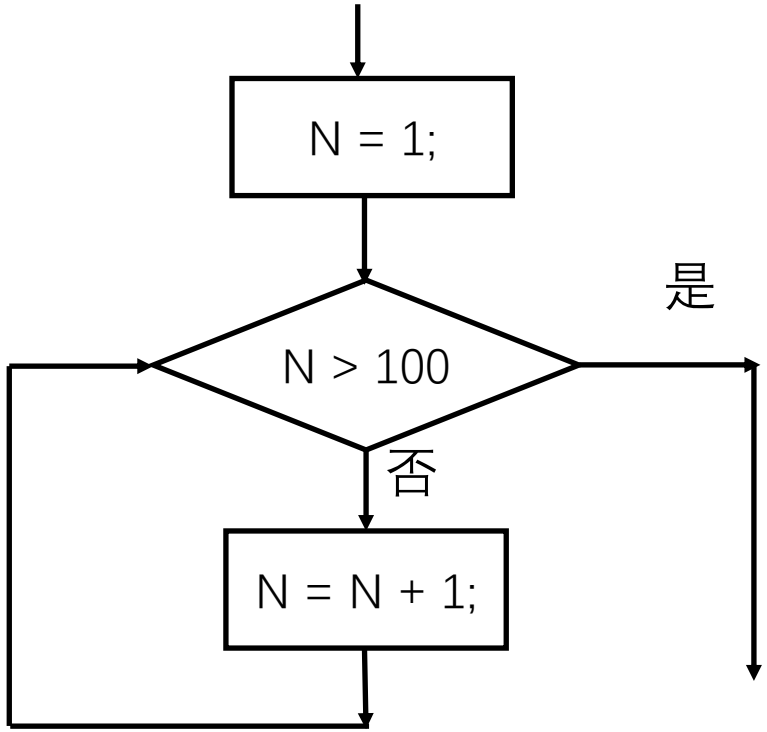
状态空间爆炸!

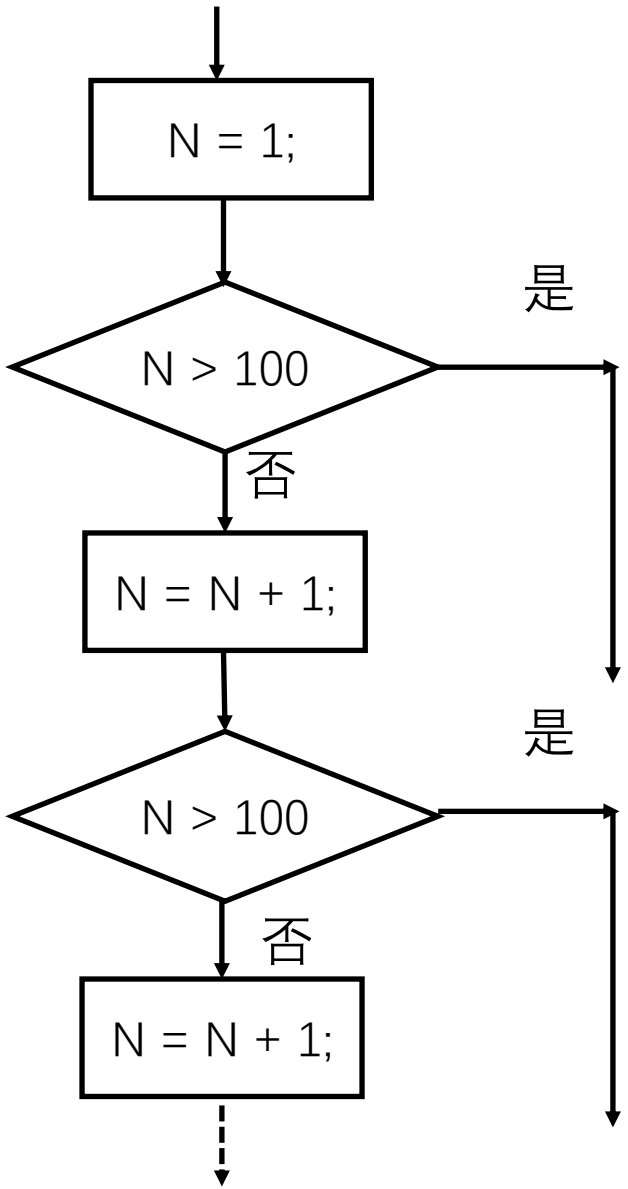
2的一百次方等于：1267650600228229401496703205376

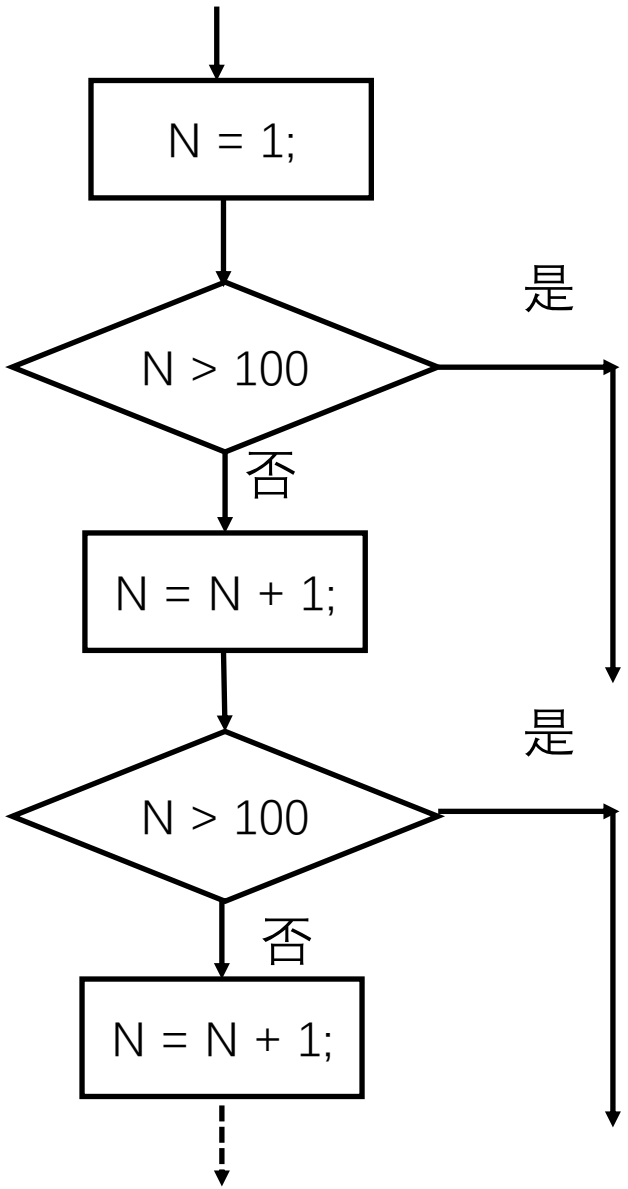
如果存在一张可以充分折叠的纸厚度为0.1毫米，其他厚度忽略不计，对半折一次，则厚度是0.2mm，再对折一次，是0.4mm……由此类推，对折n次，那么纸的厚度是：(2^n)×0.1mm
这个厚度的增长将呈指数增长的趋势，那么折了100次后，厚度达到1268万亿亿千米，若把这个单位换算成“光年”，那么其长度达到“134亿光年”，而宇宙大爆炸至今的全部时间仅仅才137亿年。

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）
- 白盒测试
  - 不可能覆盖所有的条件，分支或是路径（覆盖率）


N = 1;
N > 100
是
否
N = N + 1;


N = 1;
N > 100
是
否
N = N + 1;
N > 100
是
否
N = N + 1;

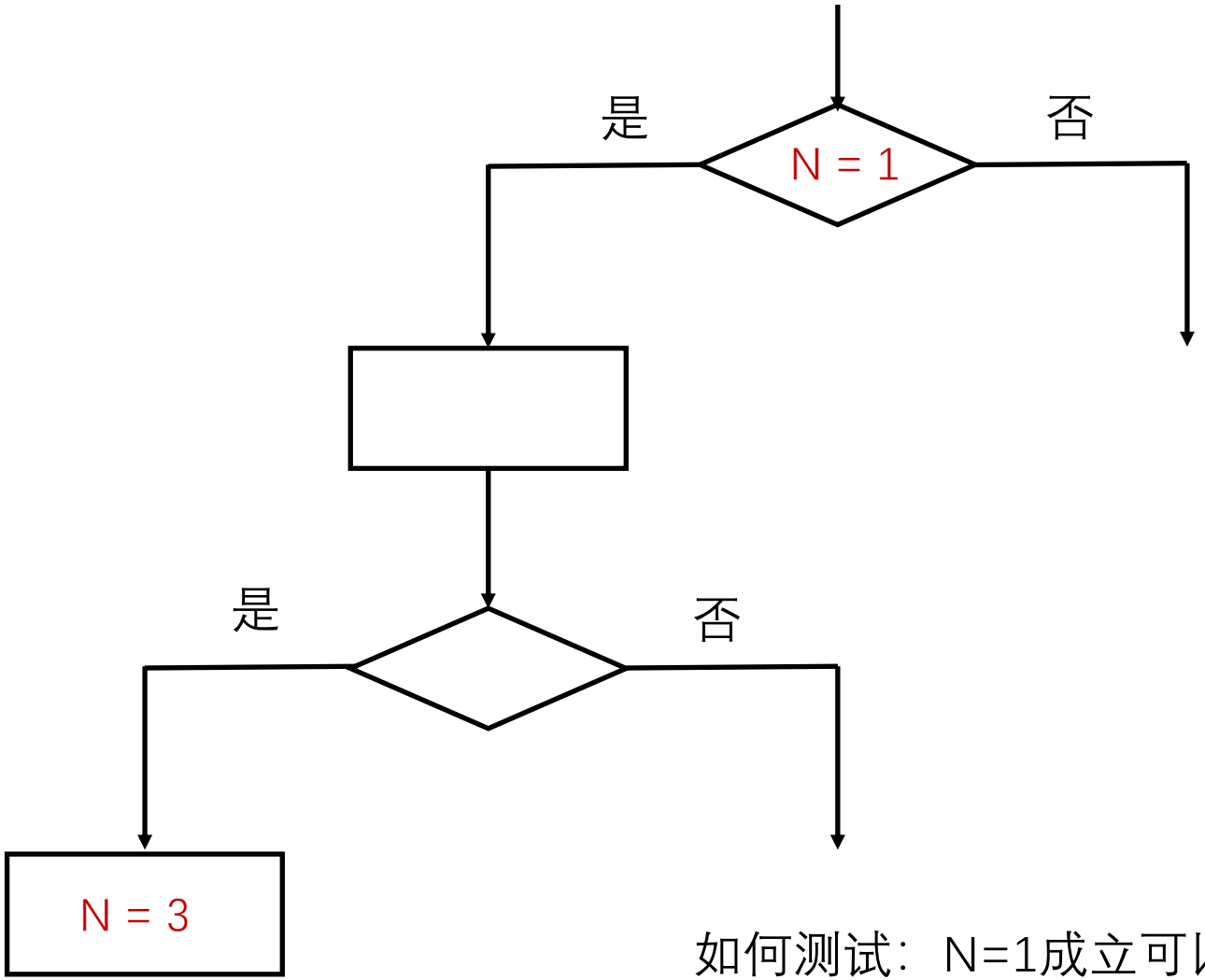


$\approx 2^{100}$

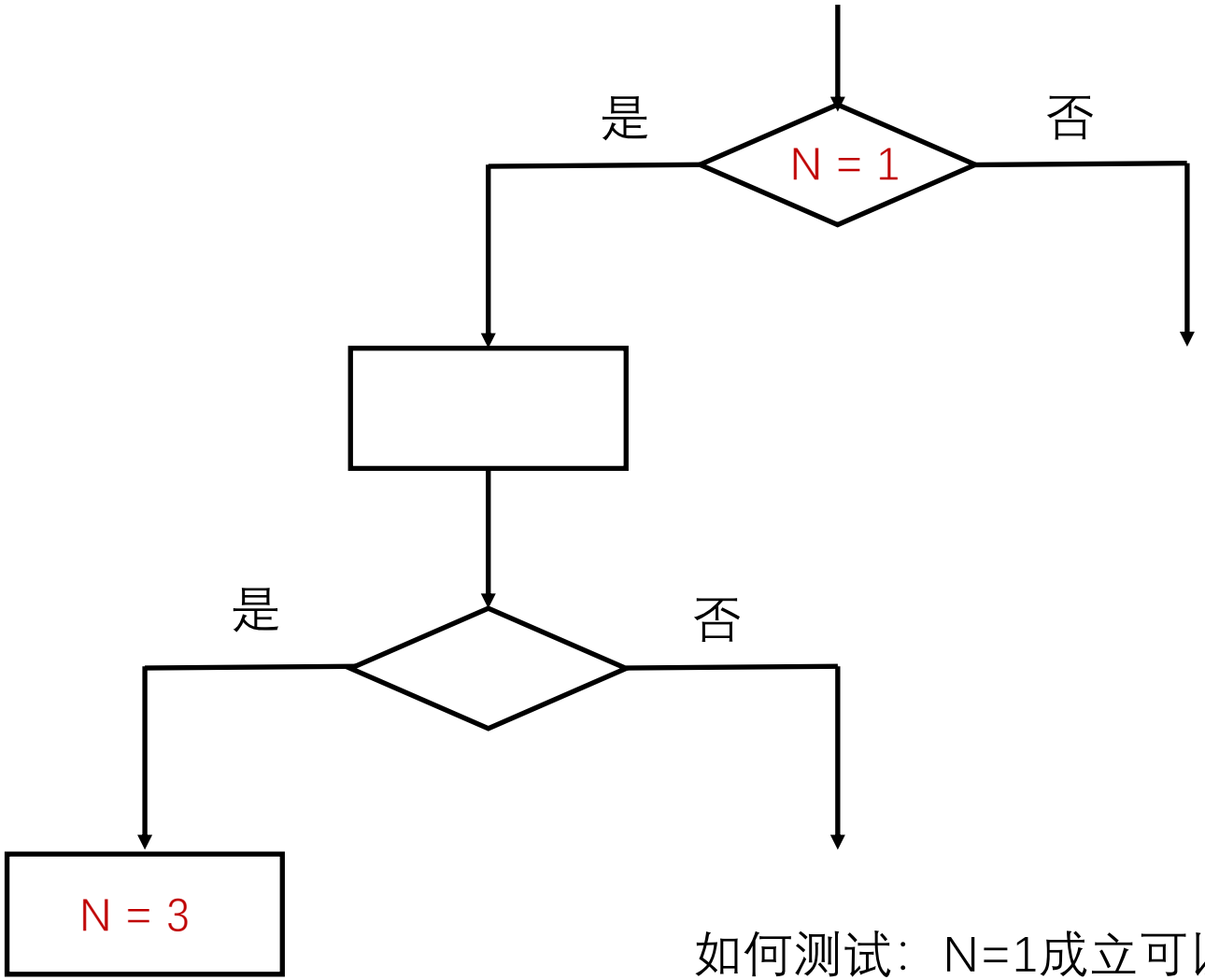
测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）
- 白盒测试
  - 不可能覆盖所有的条件，分支或是路径（覆盖率）

- 测试只能针对特定的规范（assertion, crash），很难针对一般性的规范



如何测试：N=1成立可以推出将来某一时刻N=3?



如何测试：N=1成立可以推出将来某一时刻N=3?

Global(N=1 => Future (N=3))

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）
- 白盒测试
  - 不可能覆盖所有的条件，分支或是路径（覆盖率）

- 测试只能针对特定的规范（assertion, crash），很难针对一般性的规范
- 自动化测试依赖不具有一般性

测试的局限性

- 黑盒测试
  - 不可能穷举所有的输入（测试用例）
- 白盒测试
  - 不可能覆盖所有的条件，分支或是路径（覆盖率）

- 测试只能针对特定的规范（assertion, crash），很难针对一般性的规范
- 自动化测试依赖不具有一般性
  - Java, C/C++

解决方案：用数学证明的方法来解决程序的正确性问题。

定理证明(Theorem Proving)

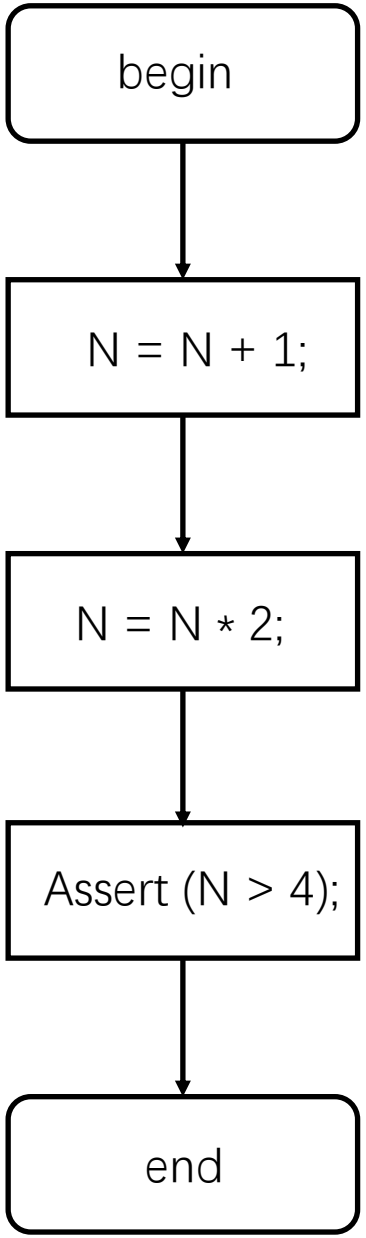
定理证明 (Theorem Proving)

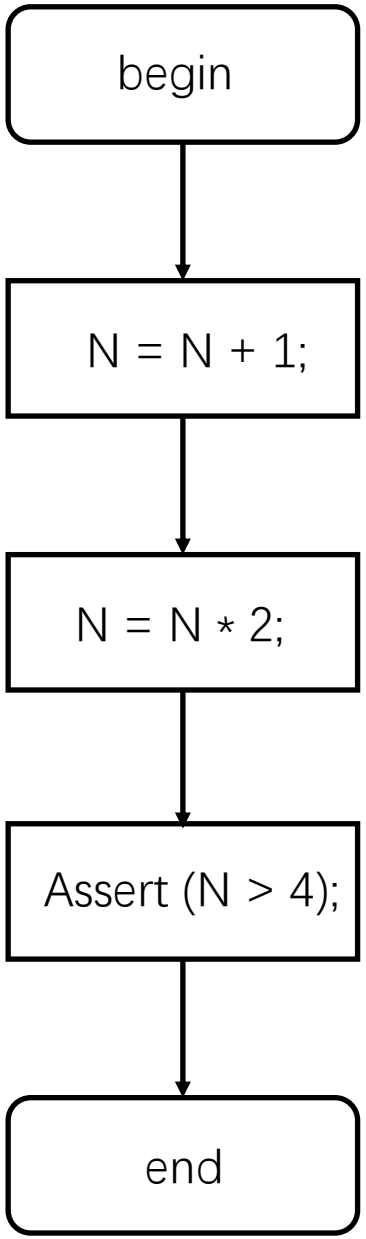
概念早于计算机出现的时间，来源于数学。

用数学推理的方法来证明计算机系统的正确性。

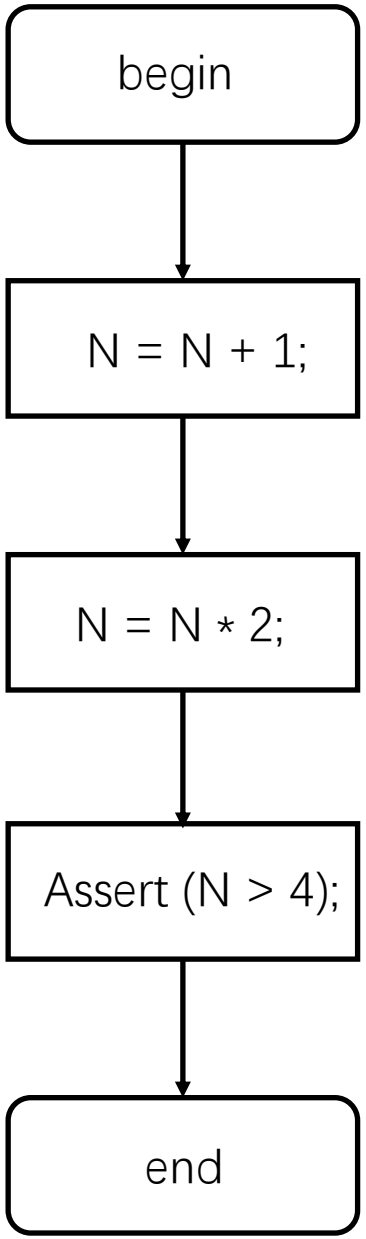
优点：可以避免状态空间爆炸问题。

缺点：无法完全实现自动化，需要大量的人工介入。


begin
N = N + 1;
N = N * 2;
Assert (N > 4);
end

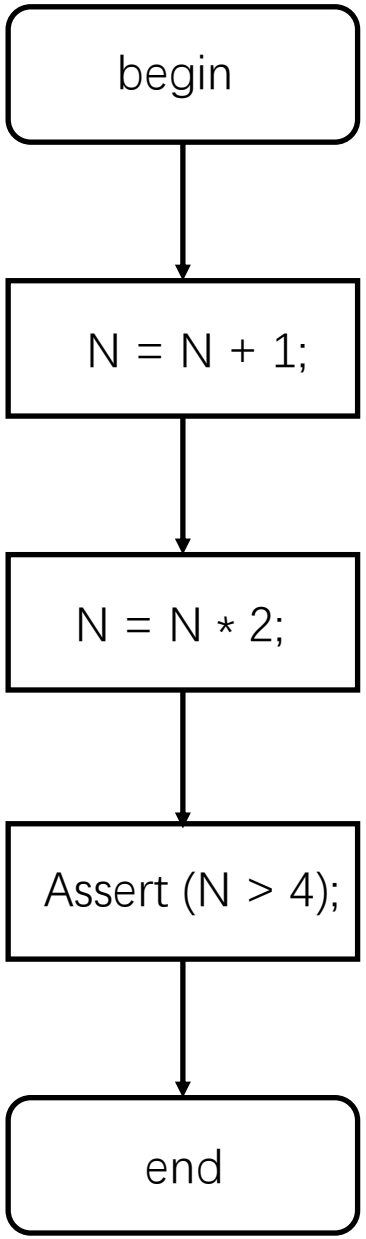


Pre: N > 1



Pre: N > 1

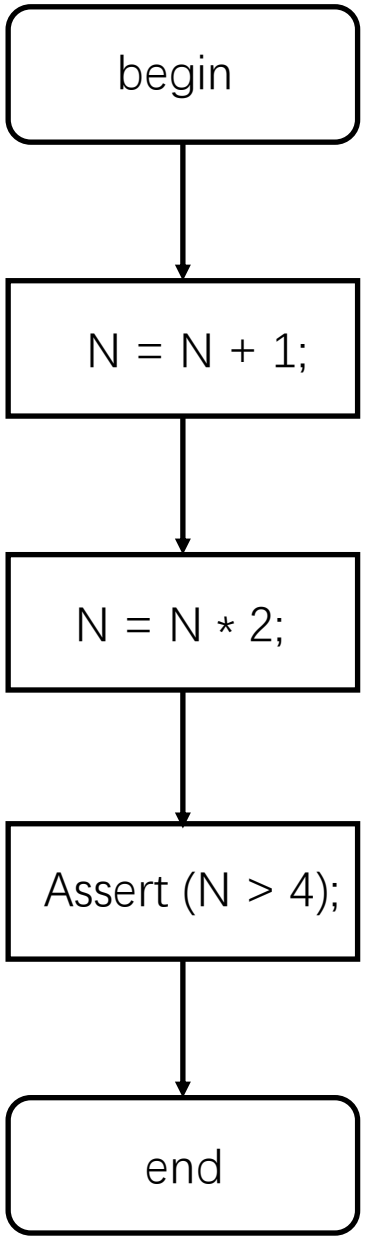
Post/Pre: N > 2



Pre: N > 1

Post/Pre: N > 2

Post/Pre: N > 4

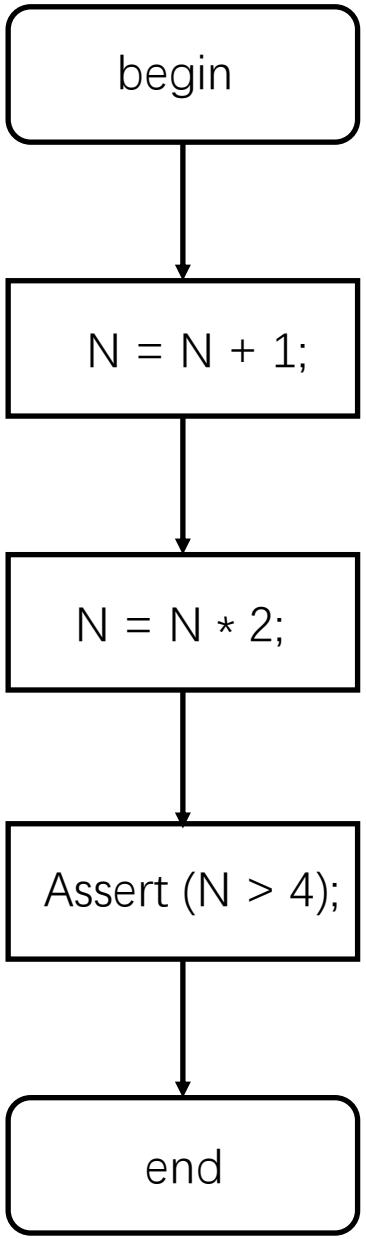


Pre: N > 1

Post/Pre: N > 2

Post/Pre: N > 4

Post: N > 4



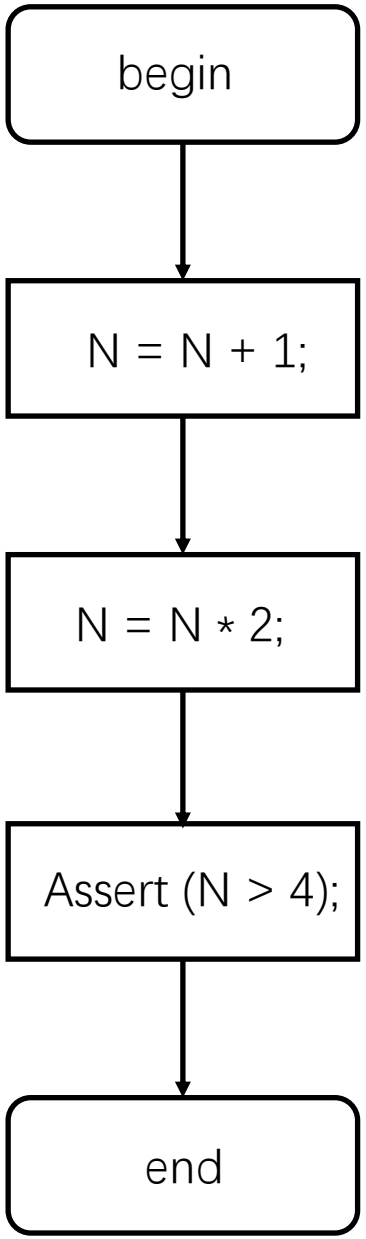
Pre: N > 1

Post/Pre: N > 2

Post/Pre: N > 4

Post: N > 4

定理证明

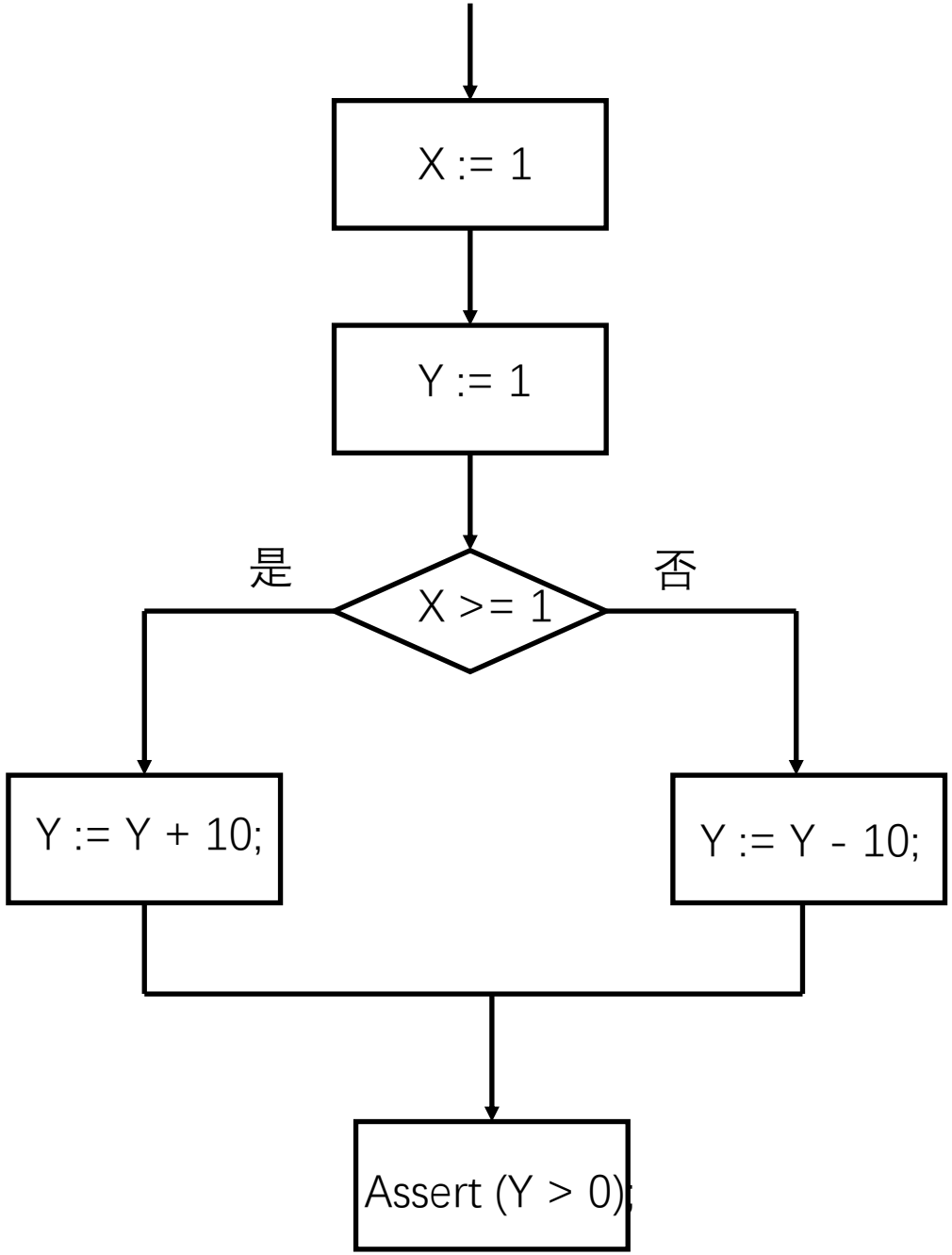


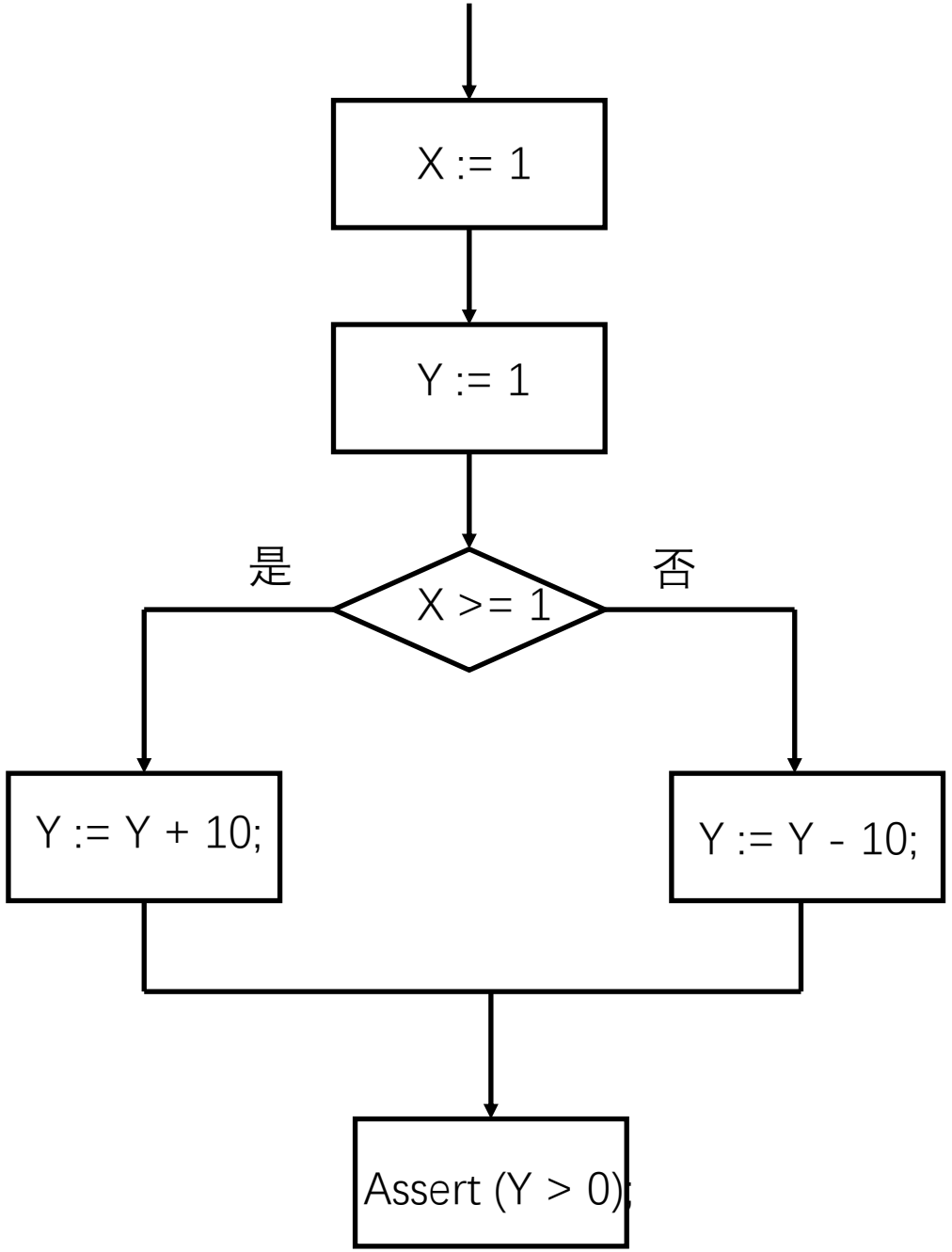
Pre: N > 1

Post/Pre: N > 2

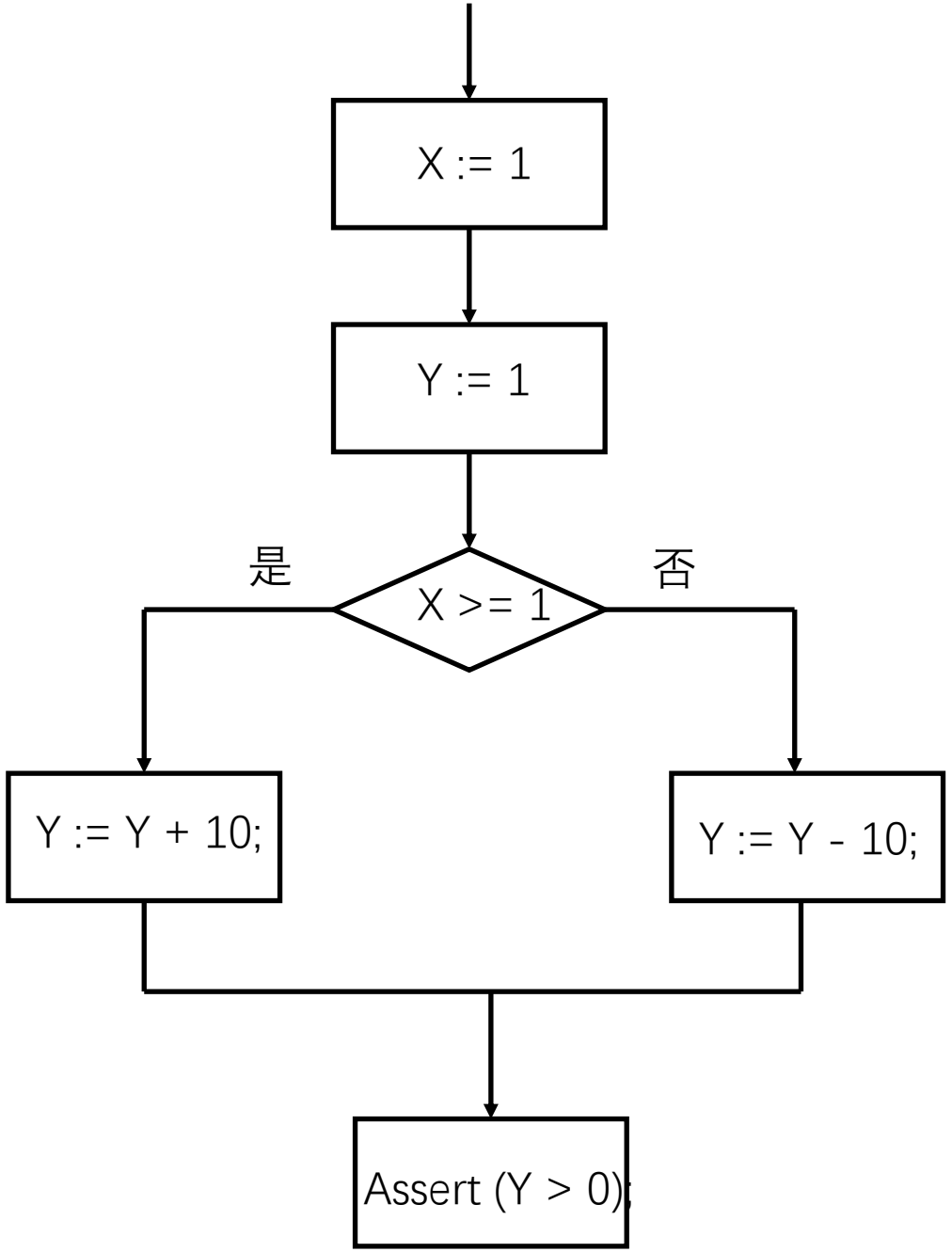
Post/Pre: N > 4

Post: N > 4

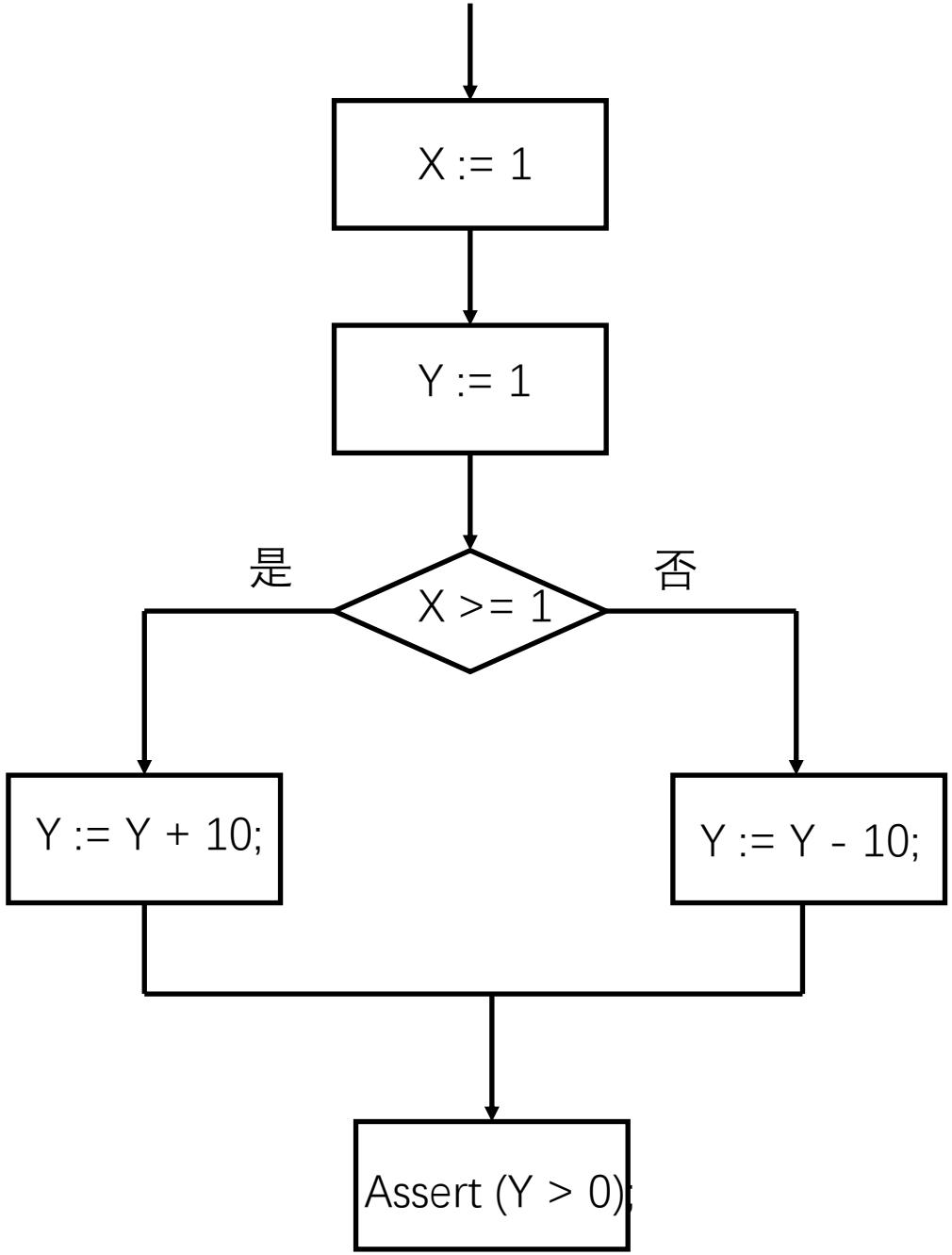

X := 1
Y := 1
是
X >= 1
否
Y := Y + 10;
Y := Y - 10;
Assert (Y > 0);



Y = 1

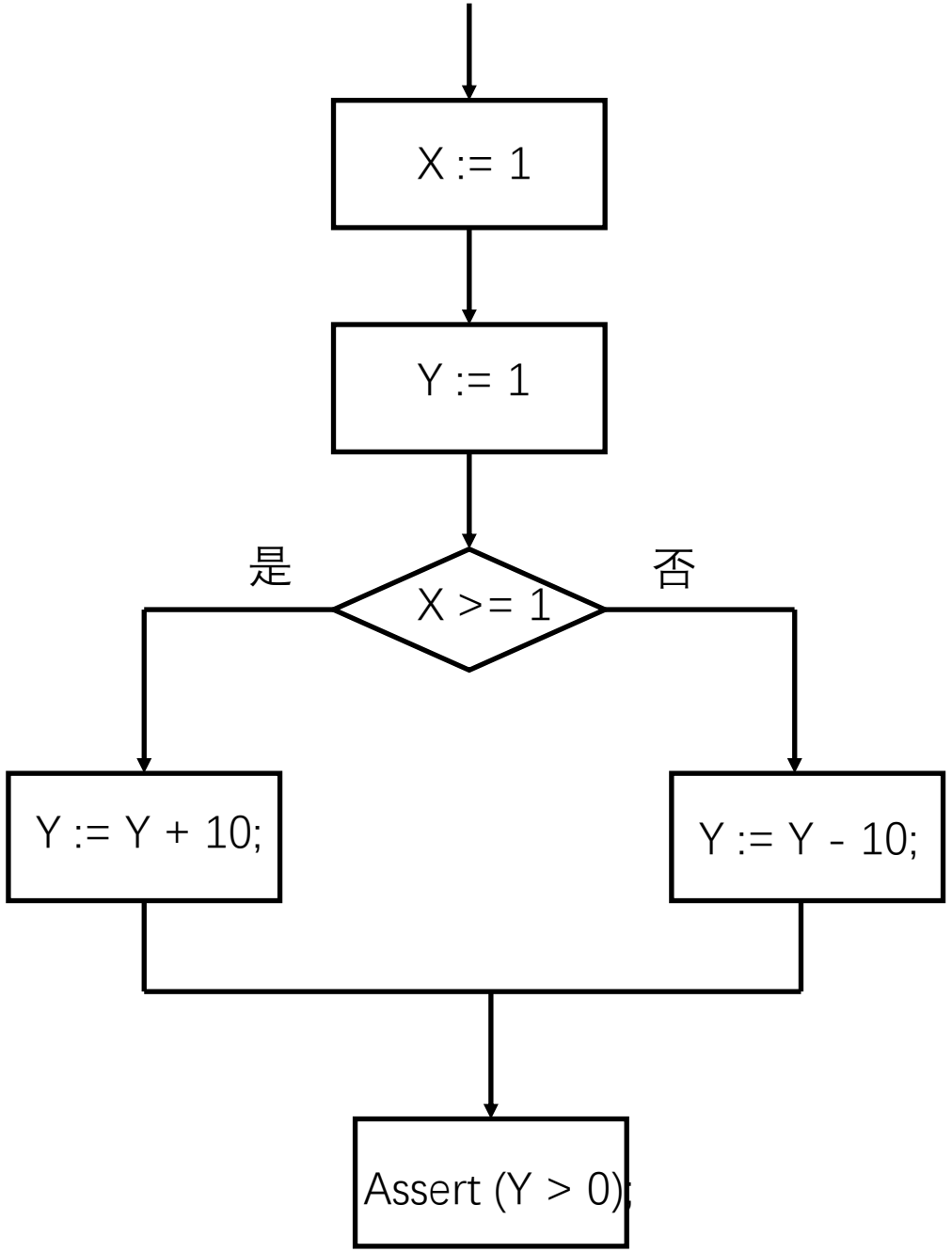


Pre: X = 1; Y = 1



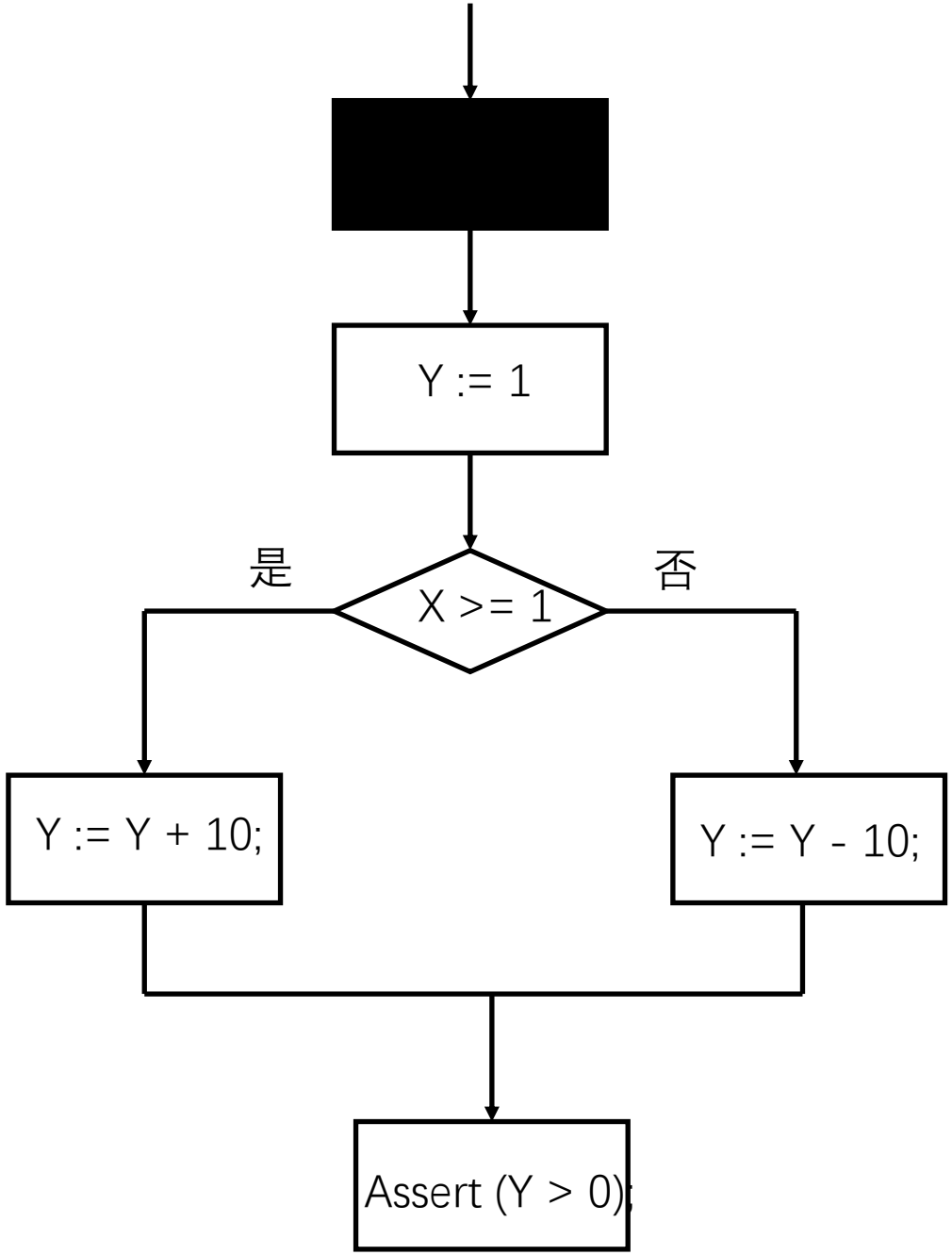
Pre: X = 1; Y = 1

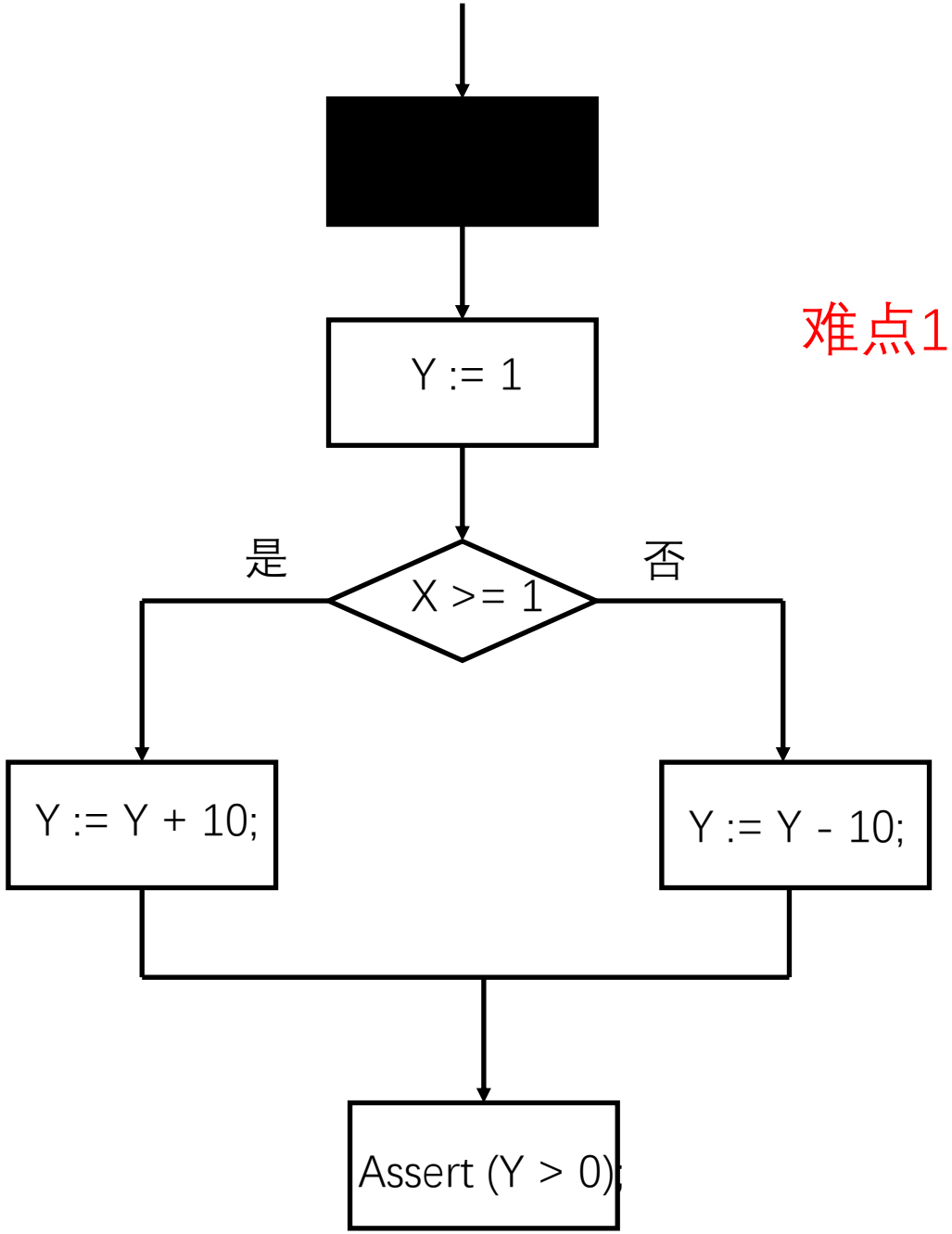
Pre/Post: X = 1, Y = 11



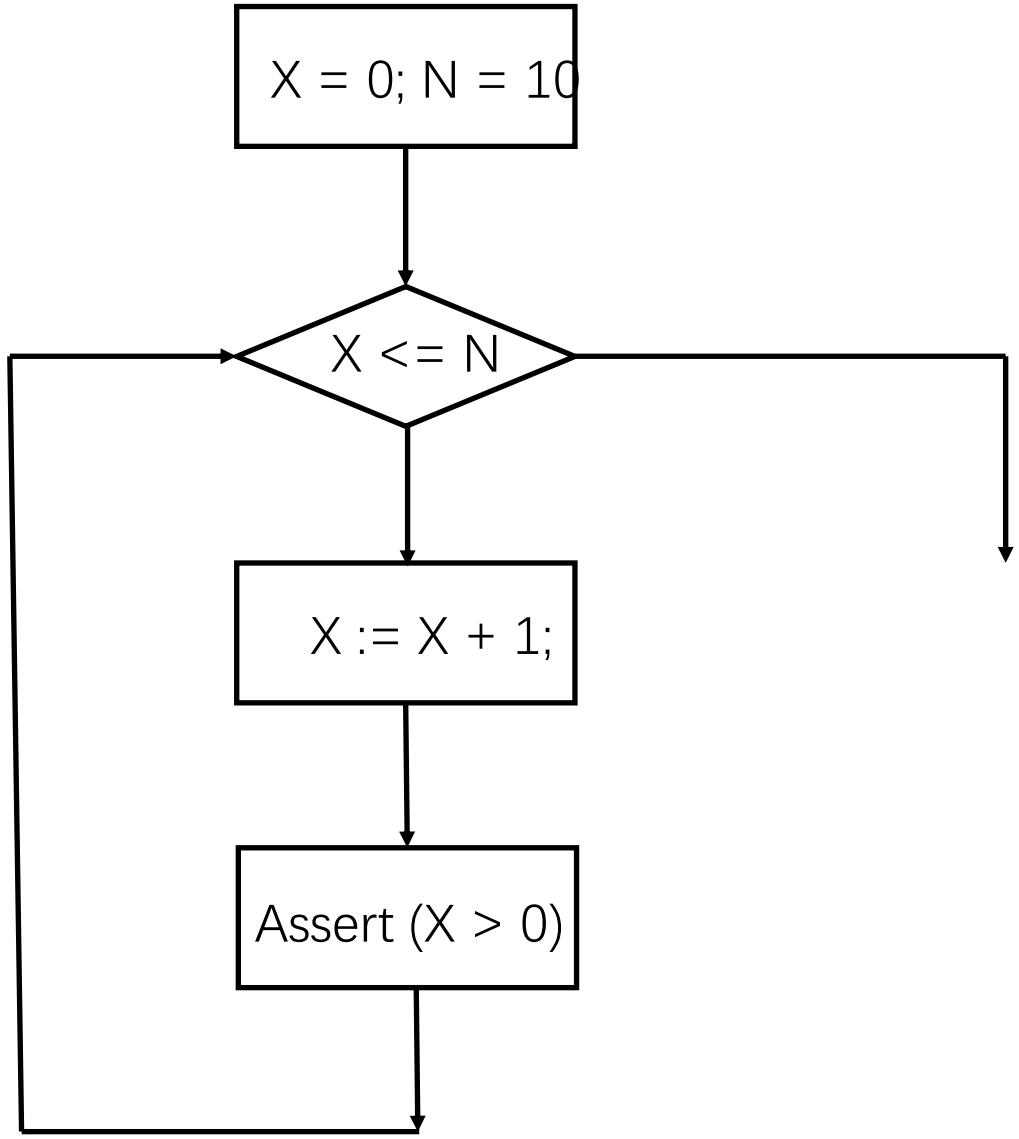
Pre: X = 1; Y = 1

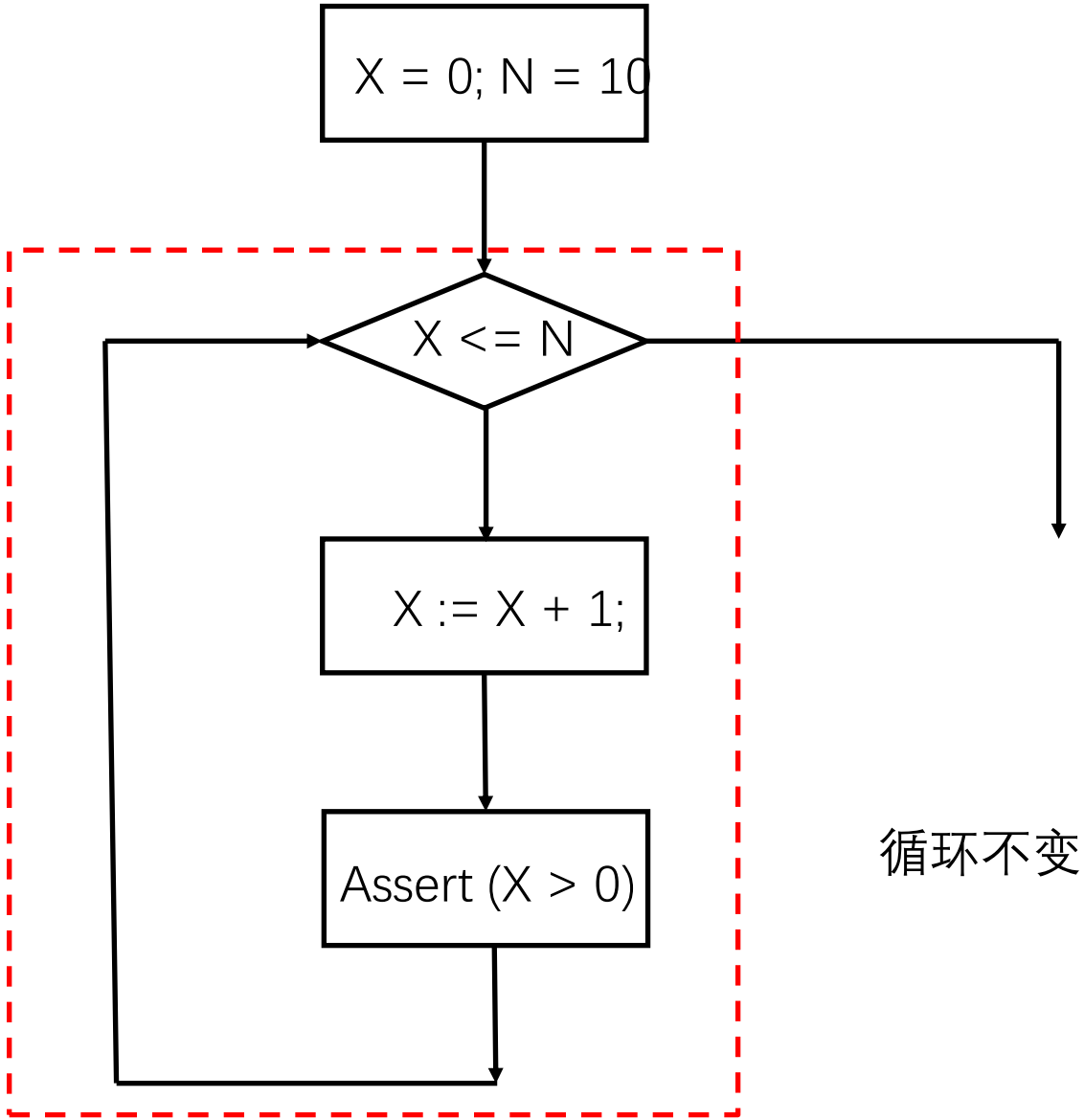
Pre/Post: X = 1, Y = 11


Y := 1
是
X >= 1
否
Y := Y + 10;
Y := Y - 10;
Assert (Y > 0);

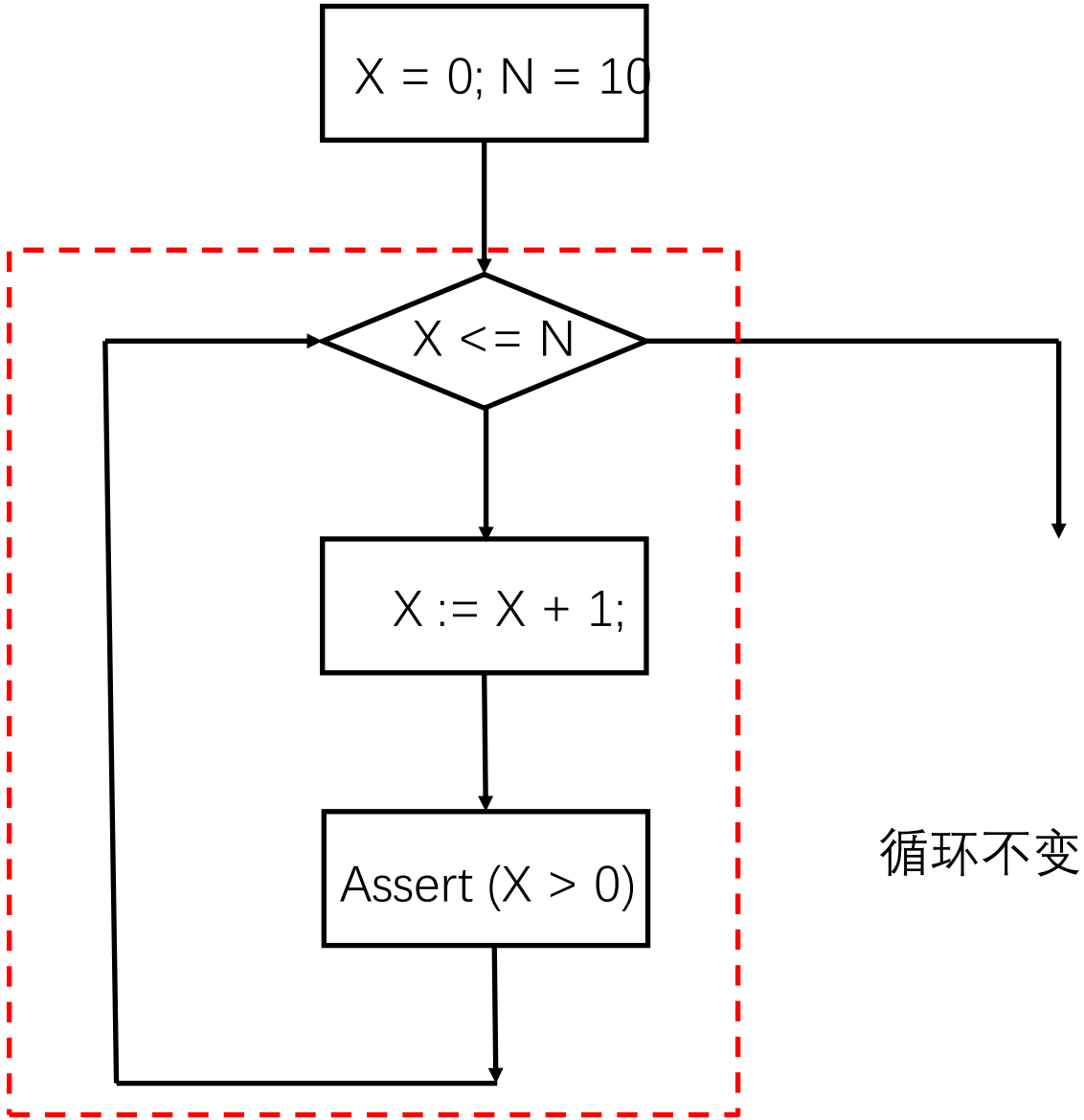


难点1：如何得到恰当的前置条件?


X = 0; N = 10
X <= N
X := X + 1;
Assert (X > 0)

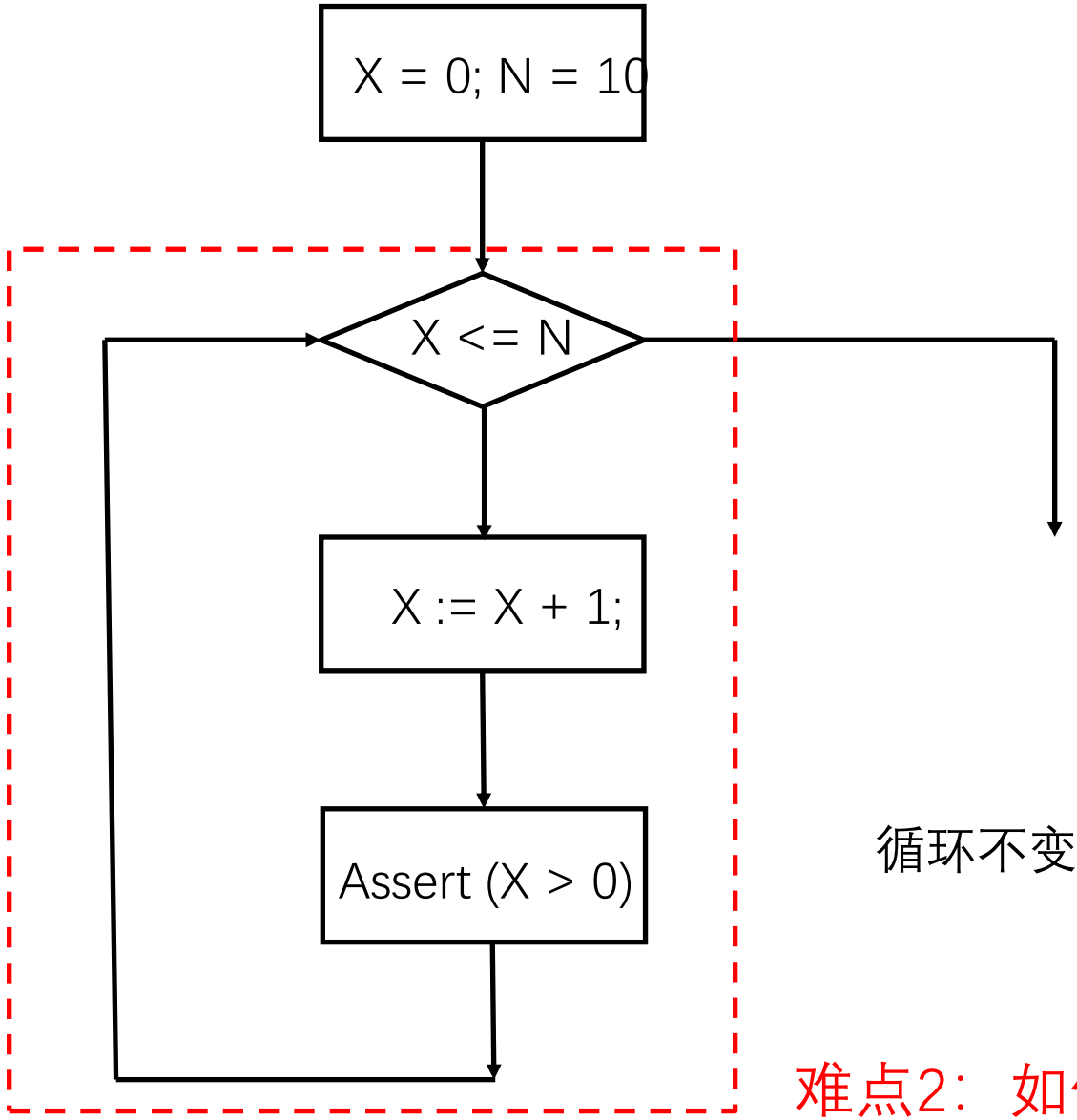


循环不变式：X>0



循环不变式：循环内部永远成立的约束。

循环不变式：X>0



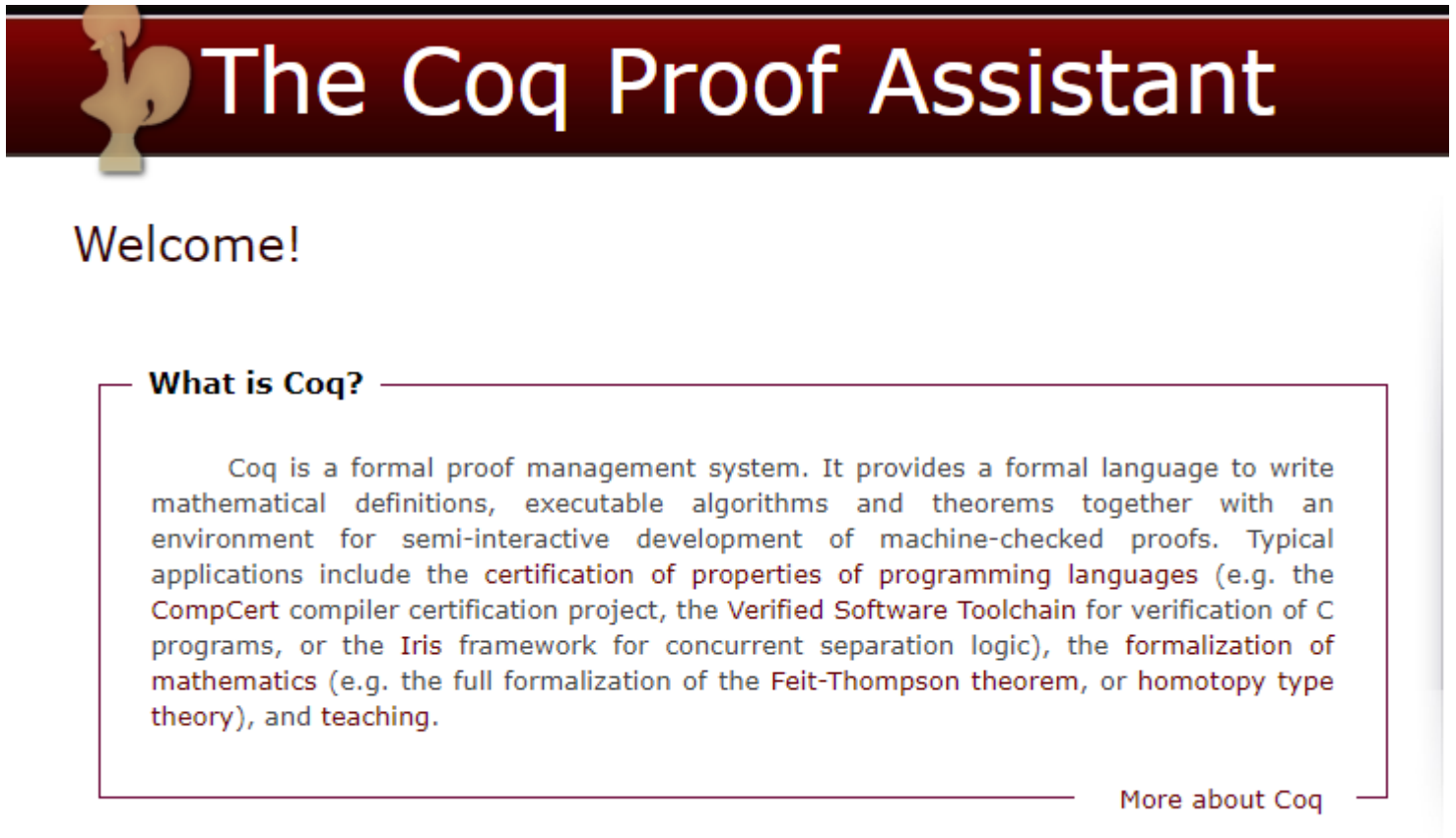
循环不变式：循环内部永远成立的约束。

循环不变式：X>0

难点2：如何生成恰当的循环不变式?

Coq定理证明器

Thierry Coquand



First released in 1989.
Received 2013 ACM Software System Award.

Isabelle定理证明器

Isabelle

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

TECHNISCHE UNIVERSITÄT MÜNCHEN

First introduced in 1986.

Lawrence Paulson

What is Isabelle?

Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.

Now available: Isabelle2020 (April 2020)

Isabelle定理证明器

Isabelle

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

TECHNISCHE UNIVERSITÄT MÜNCHEN

First introduced in 1986.

Lawrence Paulson

What is Isabelle?

Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.

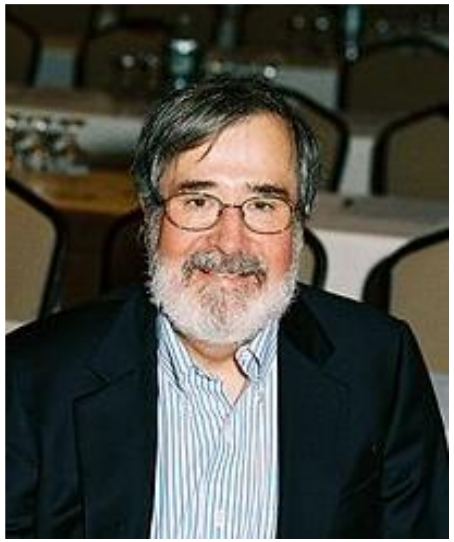
Now available: Isabelle2020 (April 2020)

In 2009, verified the first formal proof of a general-purpose operating system kernel seL4. The proof comprises over 200,000 lines of proof script to verify 7,500 lines of C. The proof uncovered 144 bugs in an early version of the C code of the seL4 kernel, and about 150 issues in each of design and specification.

模型检查(Model Checking)

模型检查的诞生（1987）

The birth of model checking. Edmund Clarke, 2008.



Edmund Clarke

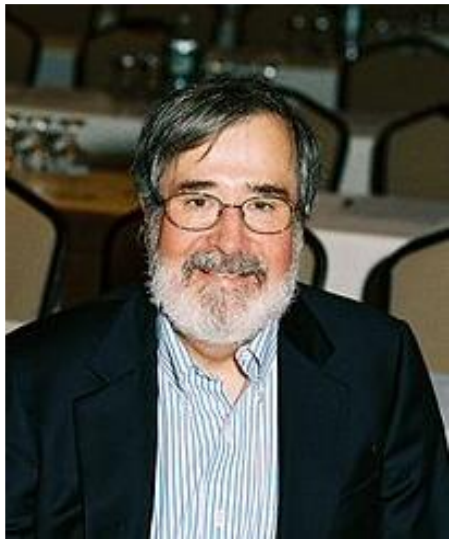
E. Allen Emerson

Joseph Sifakis

2007年图灵奖获得者

模型检查的诞生（1987）

The birth of model checking. Edmund Clarke, 2008.



自动化！

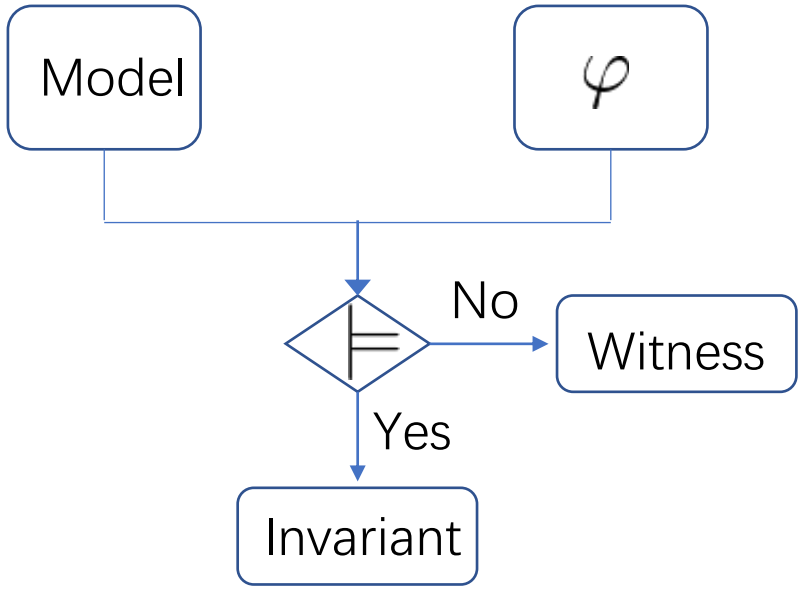
Edmund Clarke

E. Allen Emerson

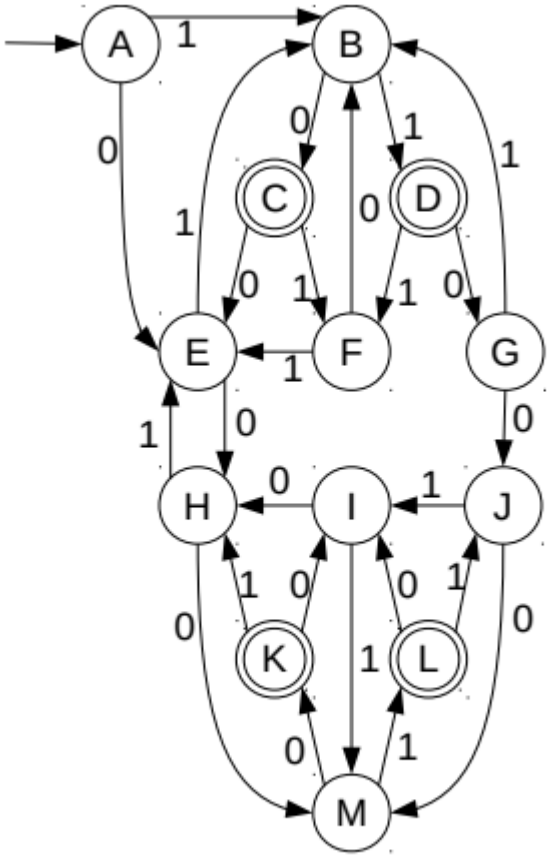
Joseph Sifakis

2007年图灵奖获得者

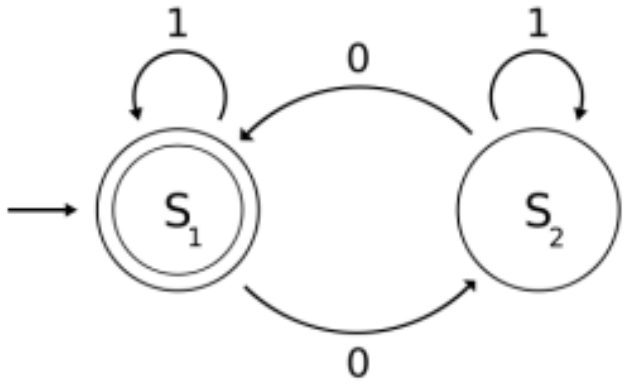
Model Checking



模型检查的基本思想

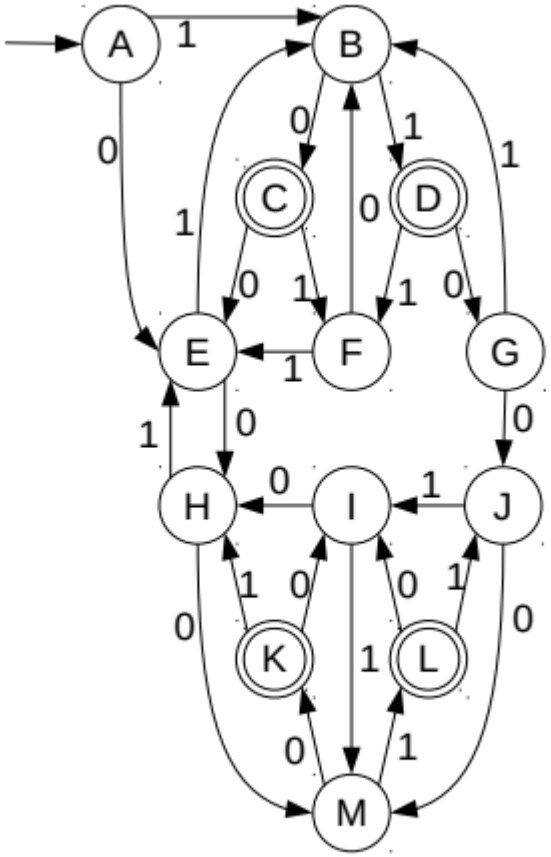


模型

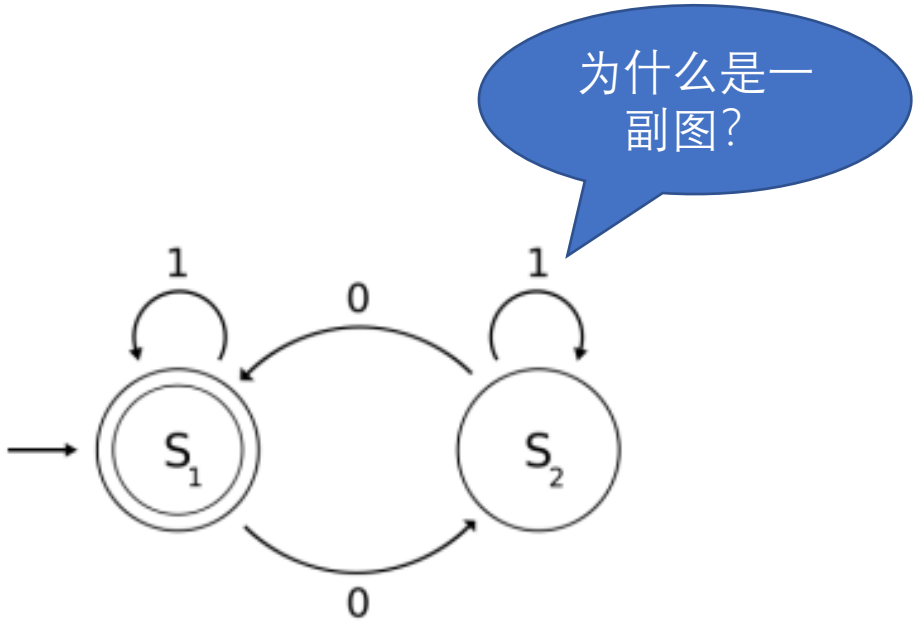


规范

模型检查的基本思想

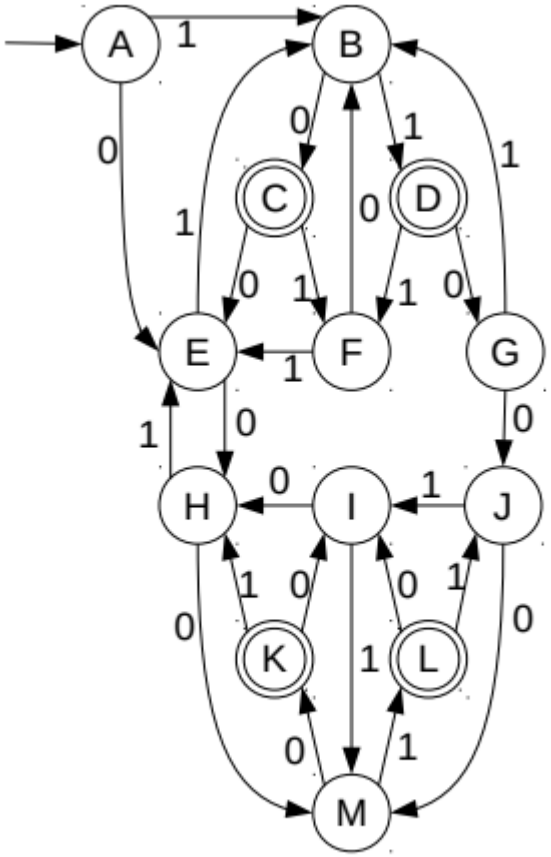


模型

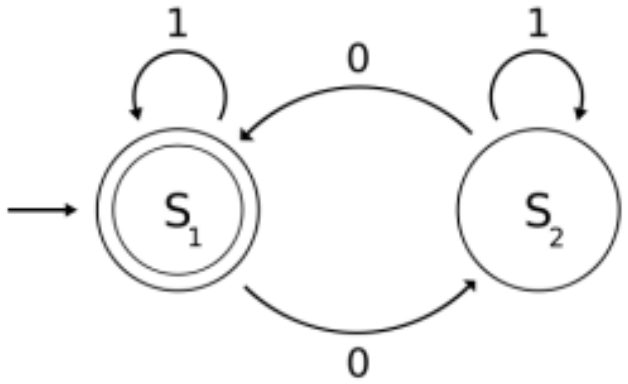


规范

模型检查的基本思想

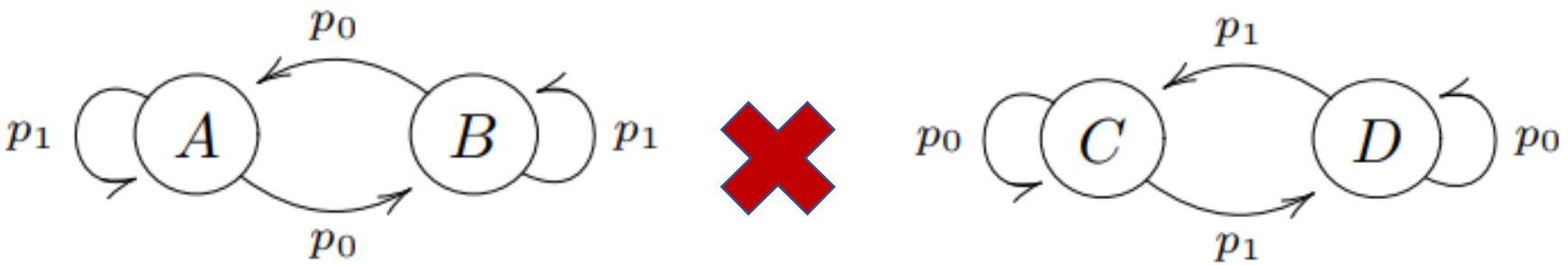


模型

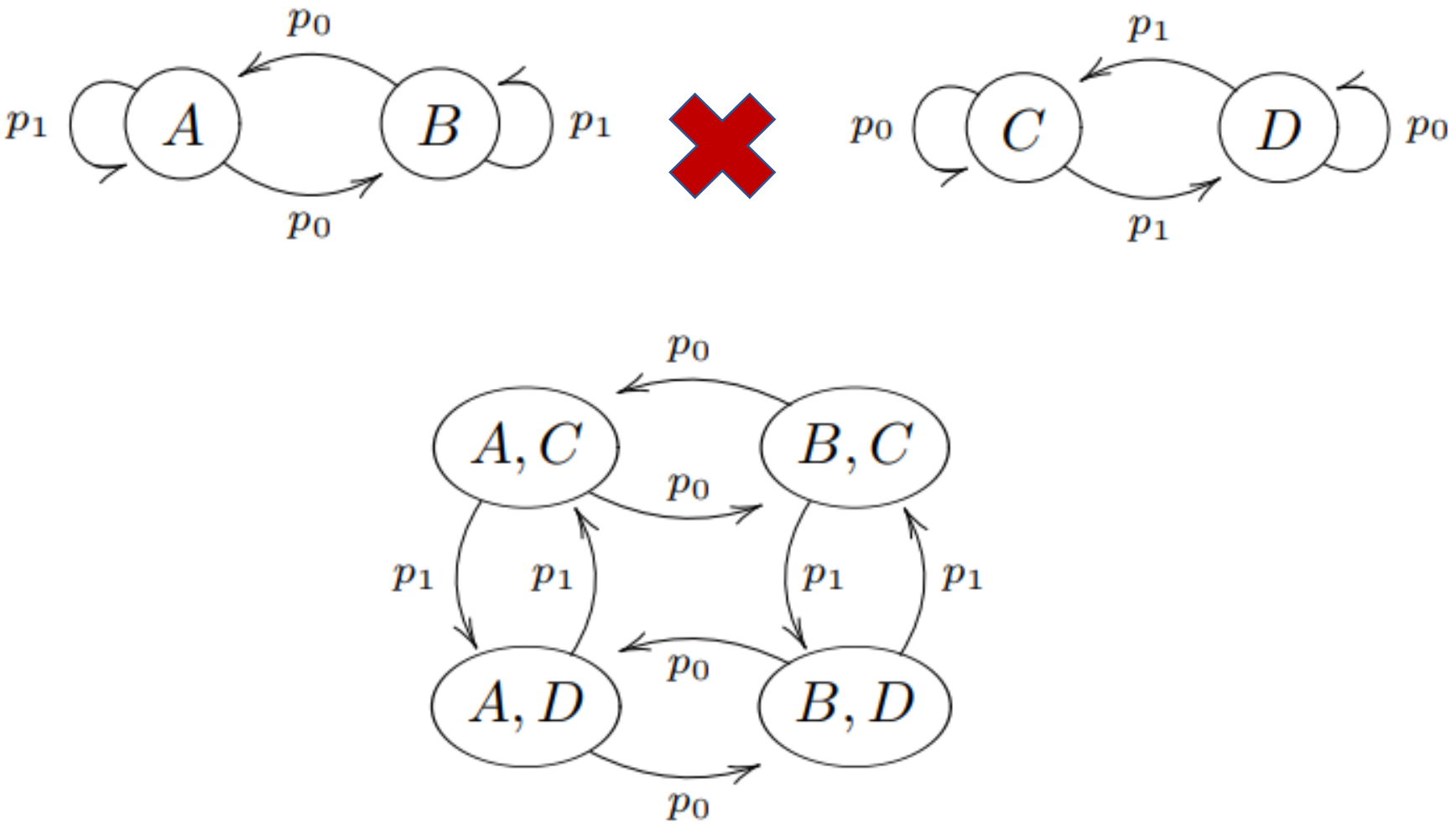


规范

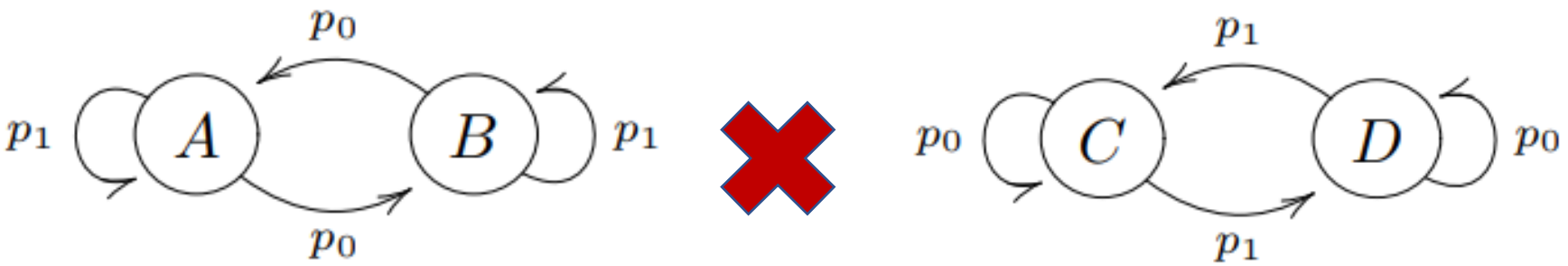
图的交集

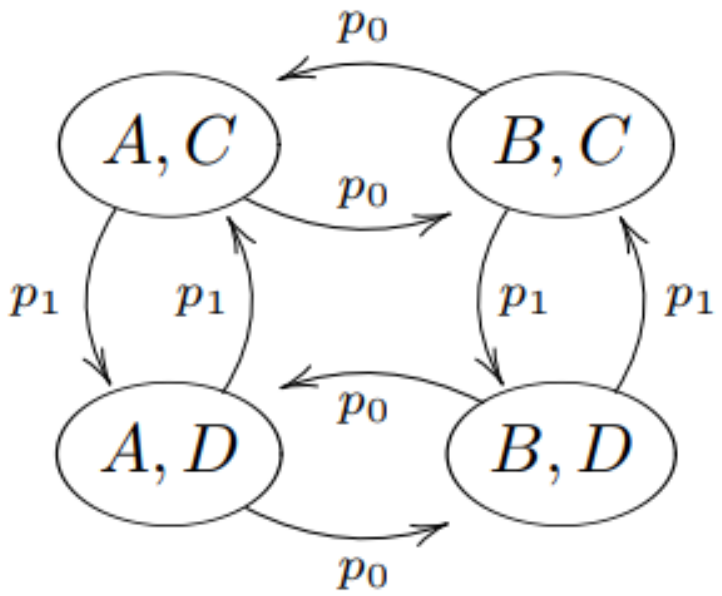


图的交集



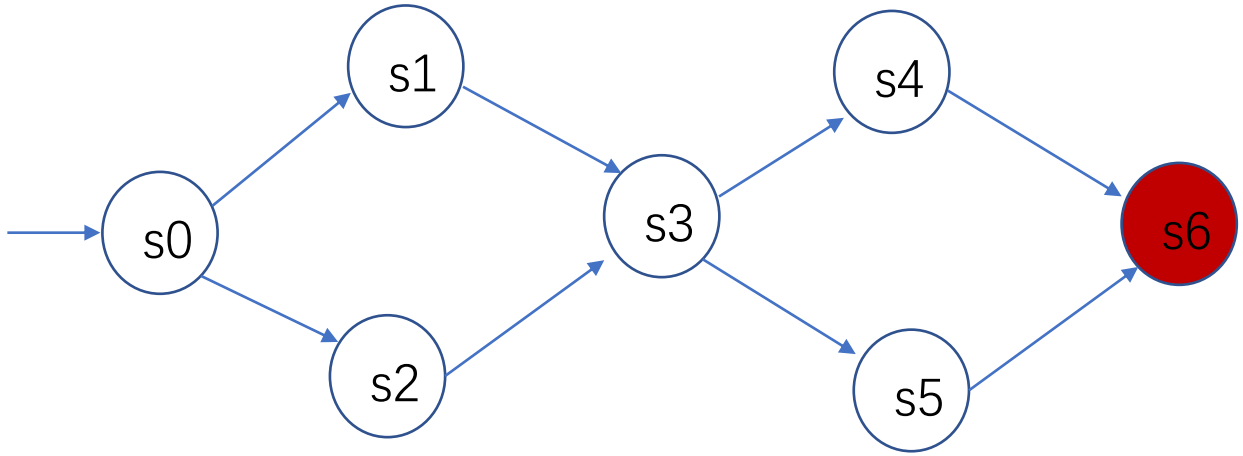
图的交集



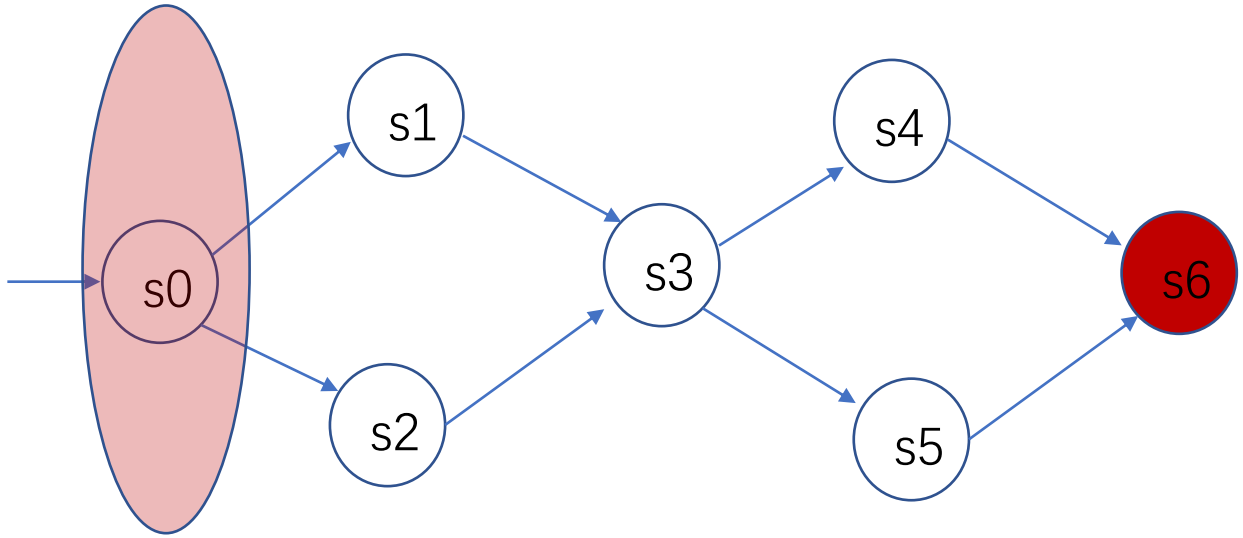


通过图搜索来解决验证问题!

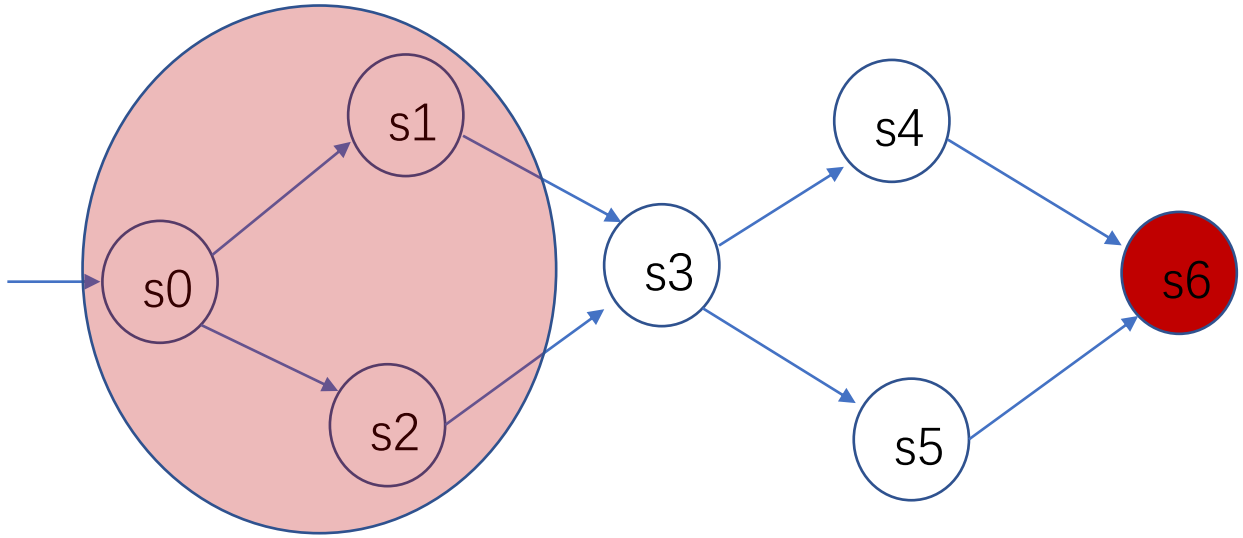
图上的可达性分析



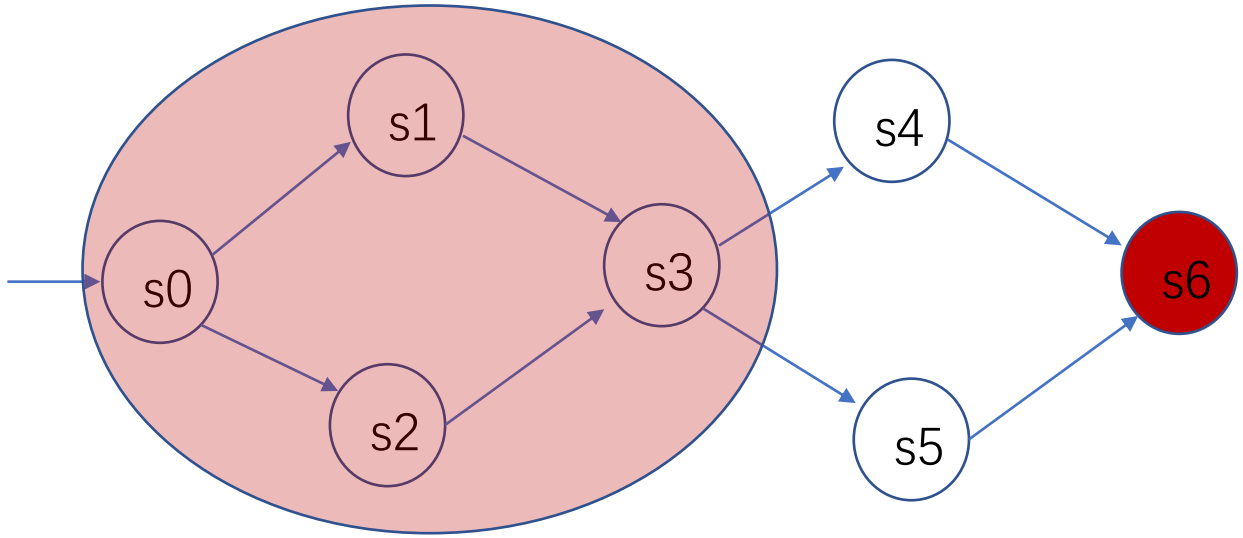
图上的可达性分析



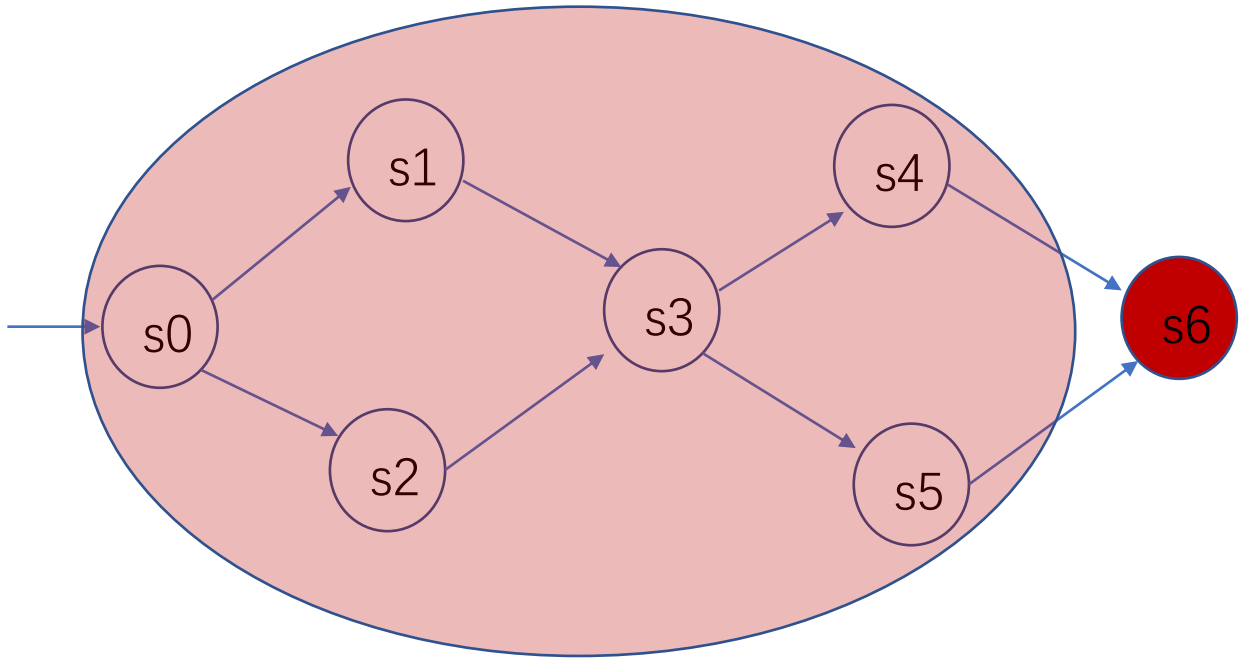
图上的可达性分析



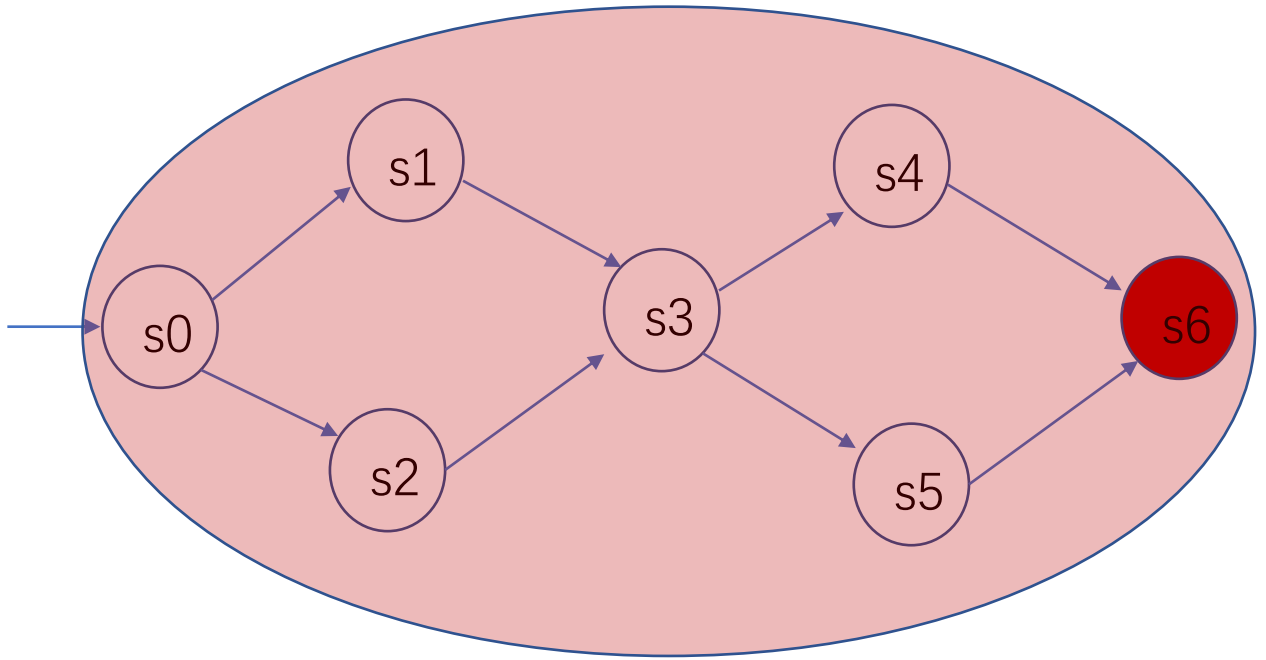
图上的可达性分析



图上的可达性分析

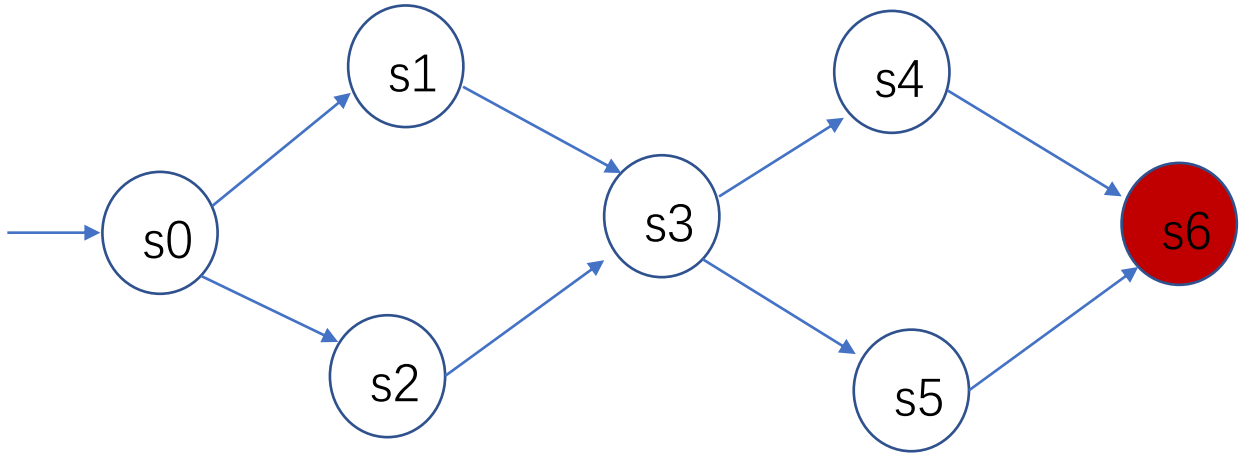


图上的可达性分析

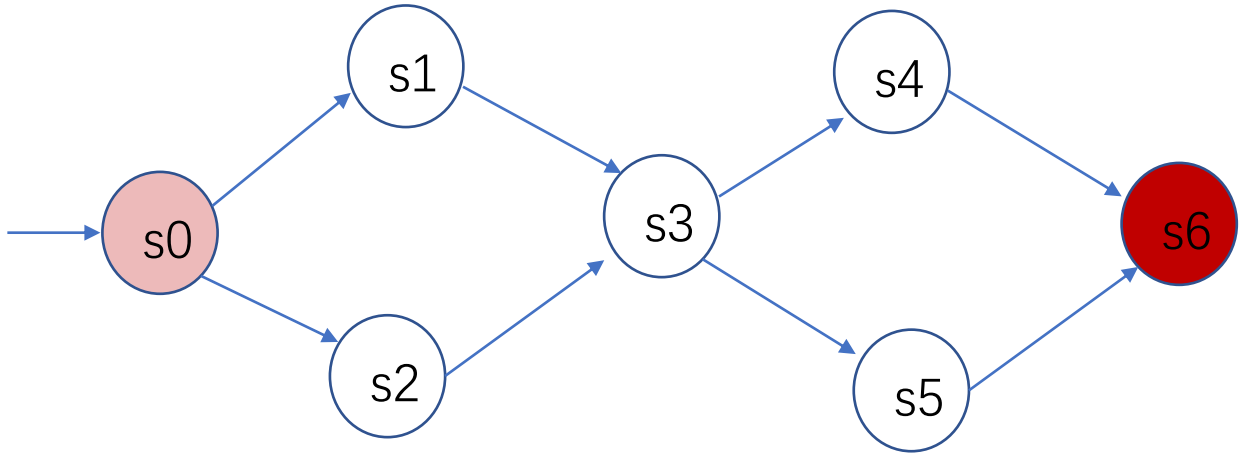


广度优先搜索

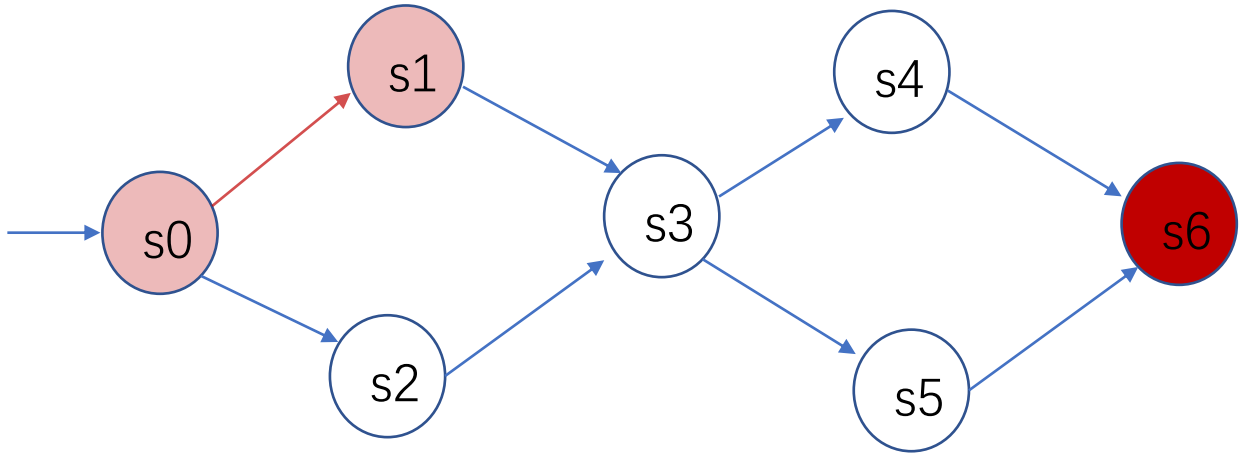
图上的可达性分析



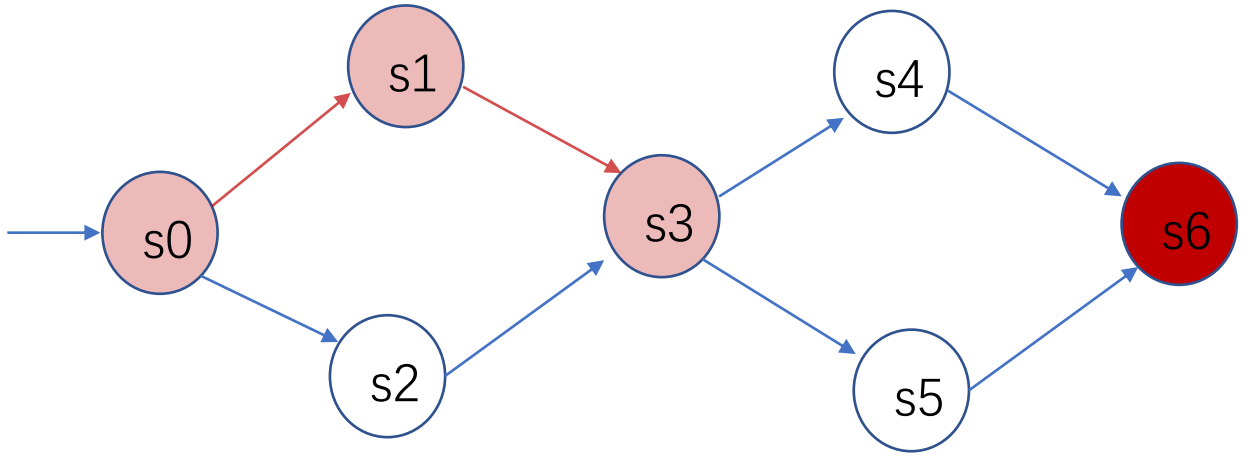
图上的可达性分析



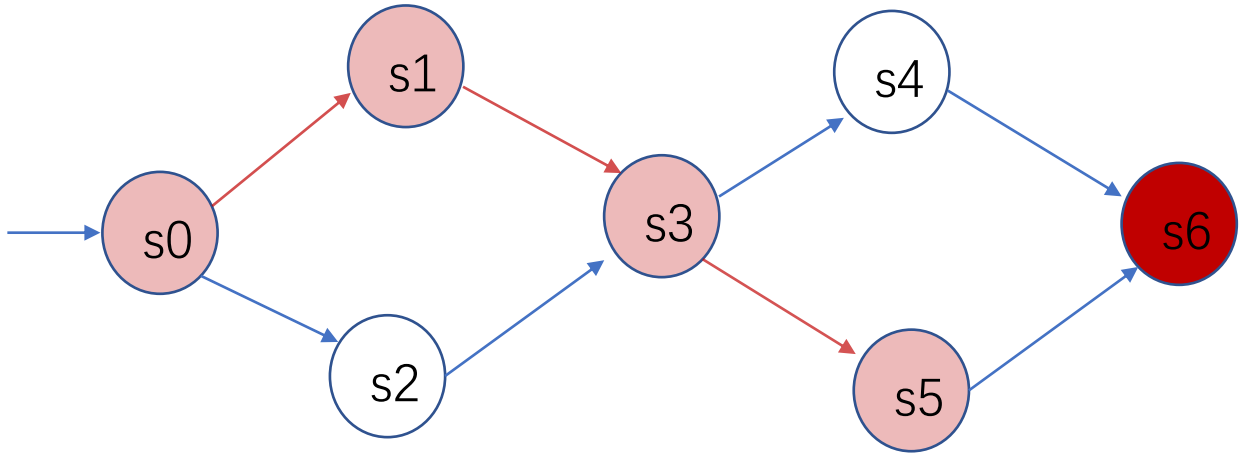
图上的可达性分析



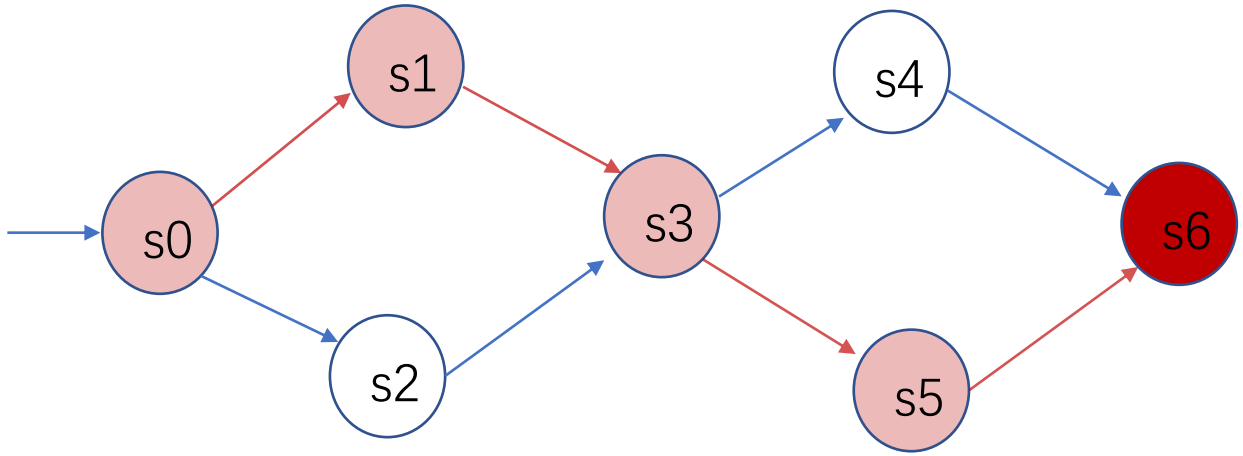
图上的可达性分析



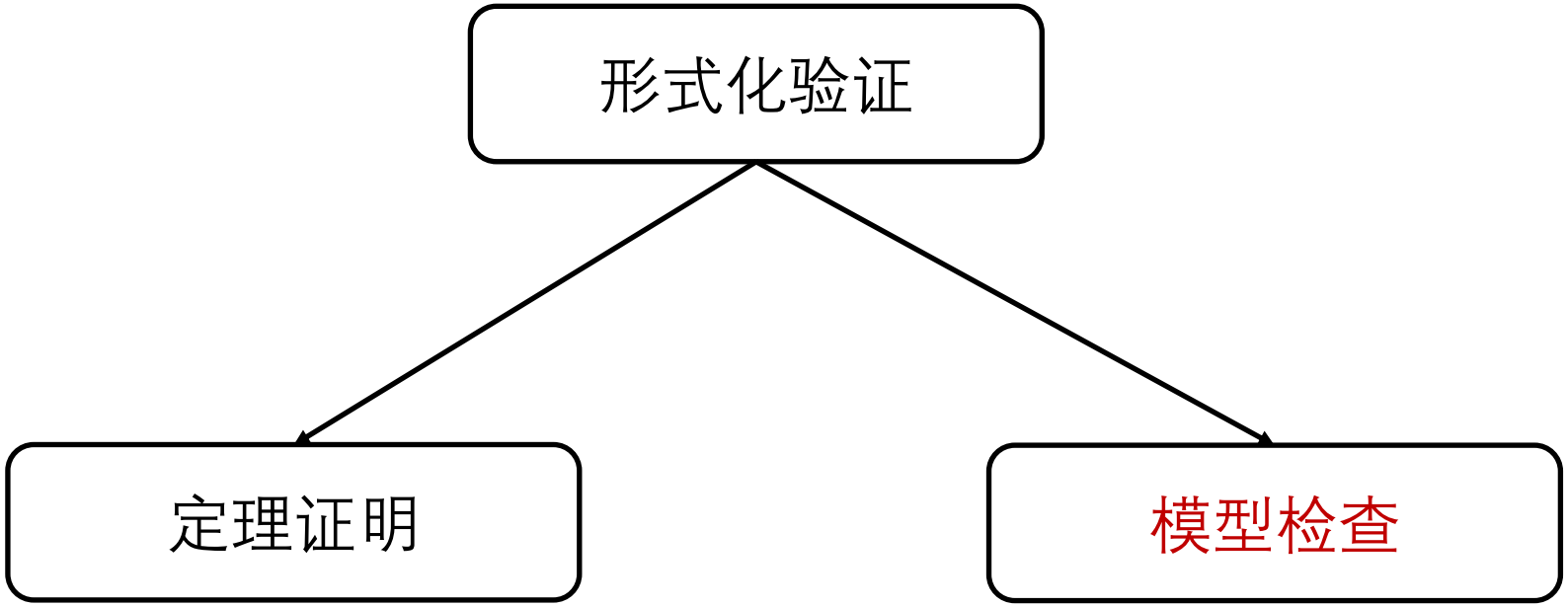
图上的可达性分析

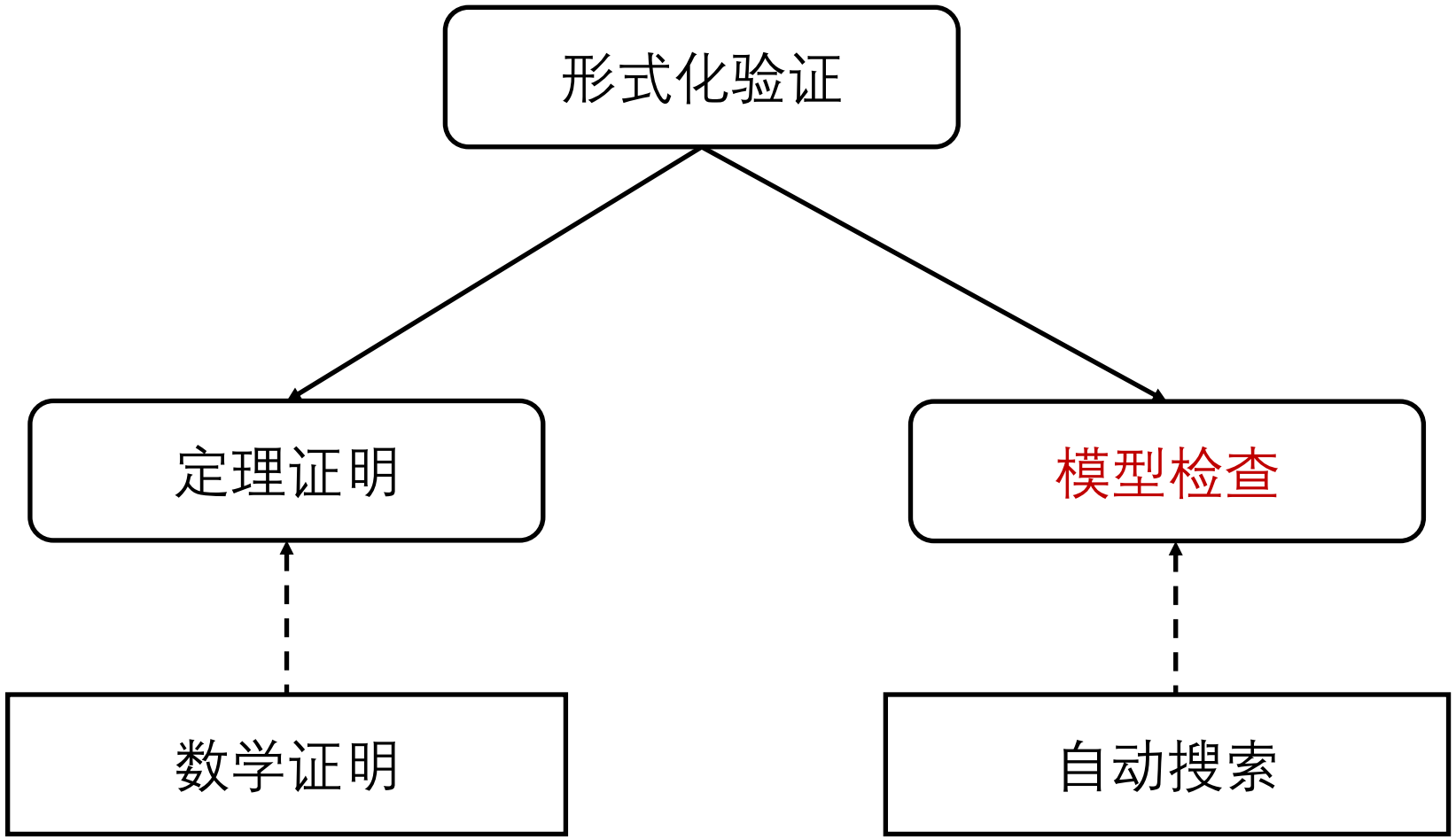


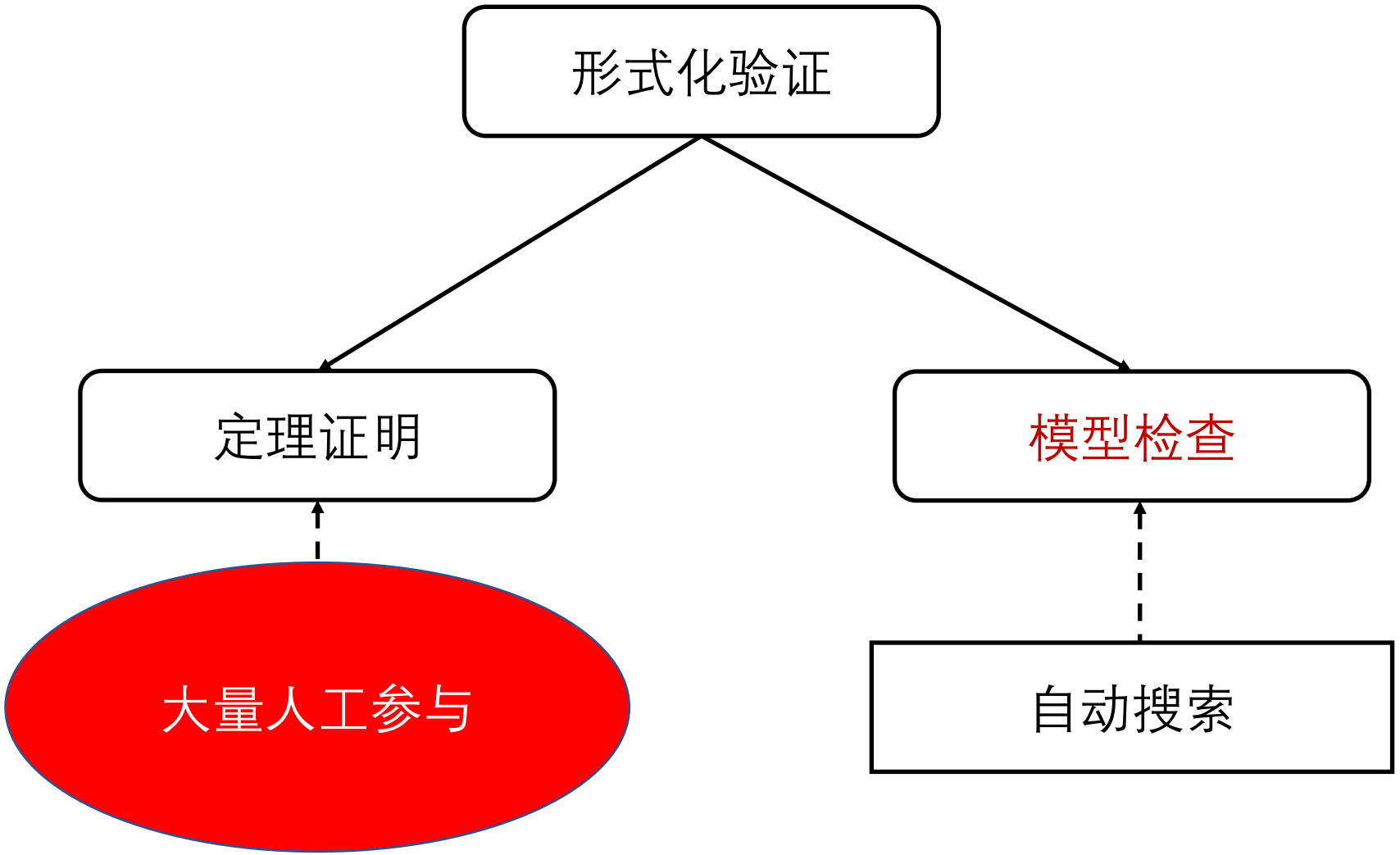
图上的可达性分析

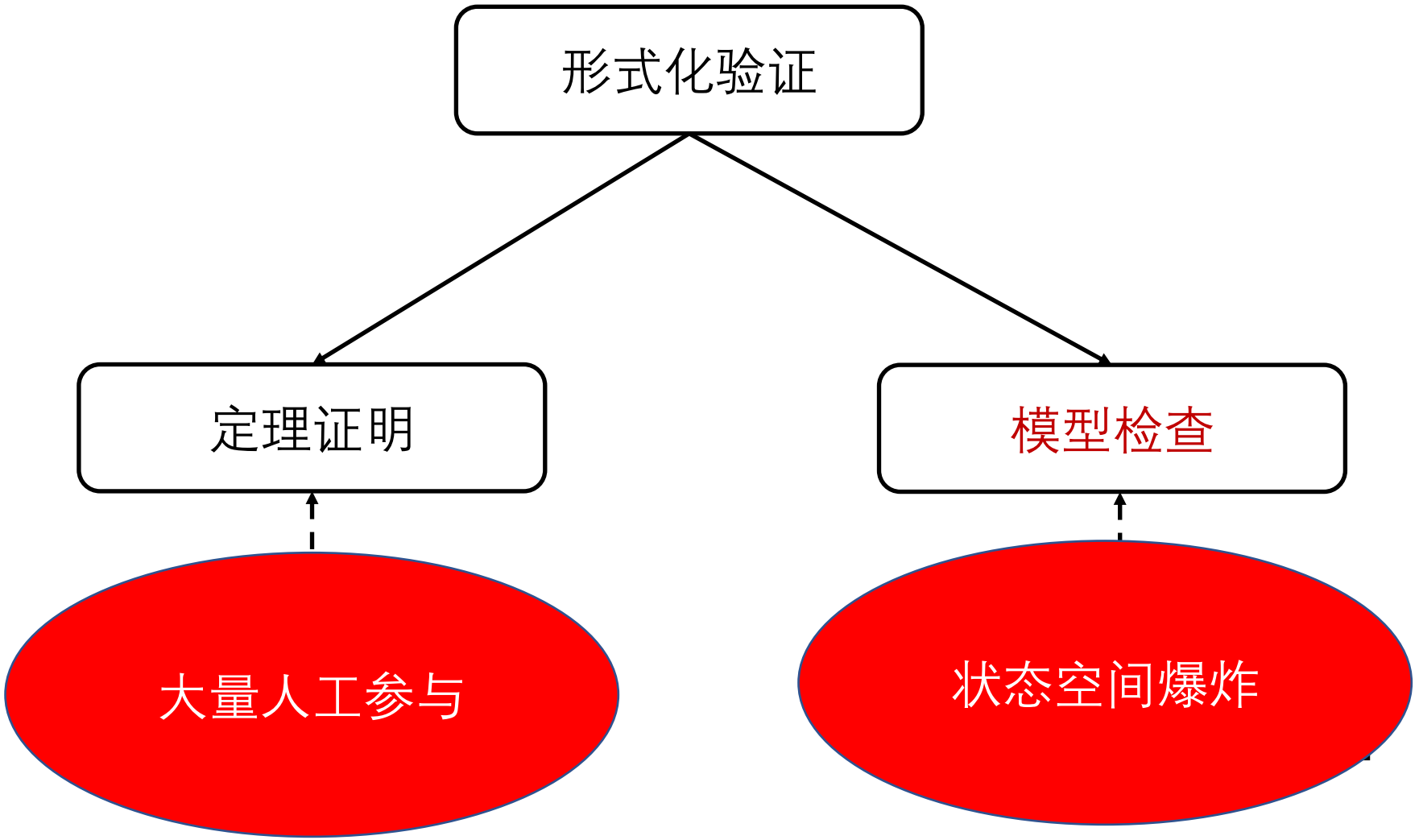


深度优先搜索


形式化验证
定理证明
模型检查


形式化验证
定理证明
模型检查
数学证明
自动搜索


形式化验证
定理证明
模型检查
大量人工参与
自动搜索


形式化验证
定理证明
模型检查
大量人工参与
状态空间爆炸

验证的困境

	测试	验证	
		定理证明	模型检查
模型	易	难	难
性质	易	难	难
算法	易	无自动化	自动化
完备性	否	是	是

验证的困境

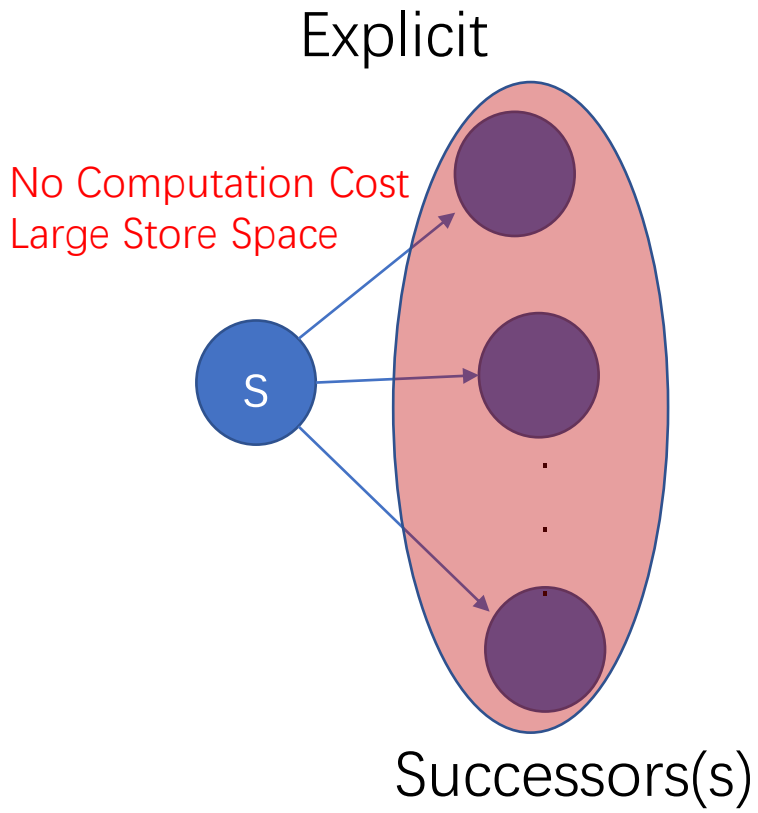
	测试	硬件验证	
		定理证明	模型检查
模型	易	难	易
性质	易	难	易
算法	易	无自动化	自动化（难）
完备性	否	是	是

来自工业界的挑战

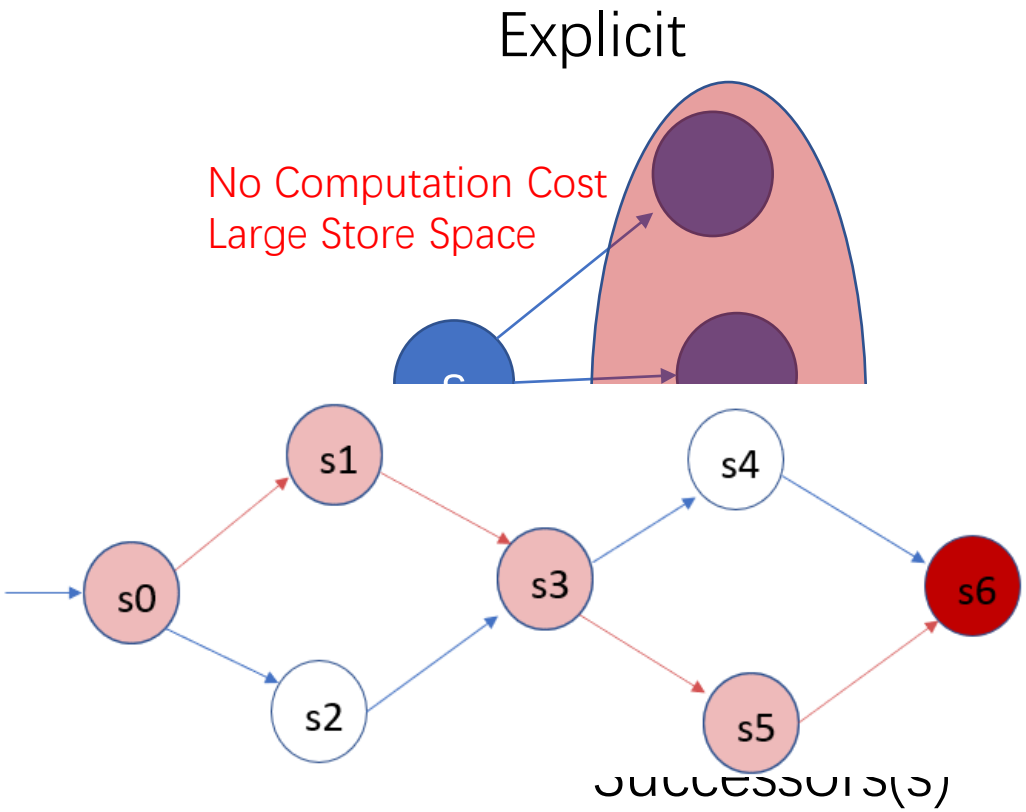
传统搜索算法实效!

$\geq 2^{1000000}$ states

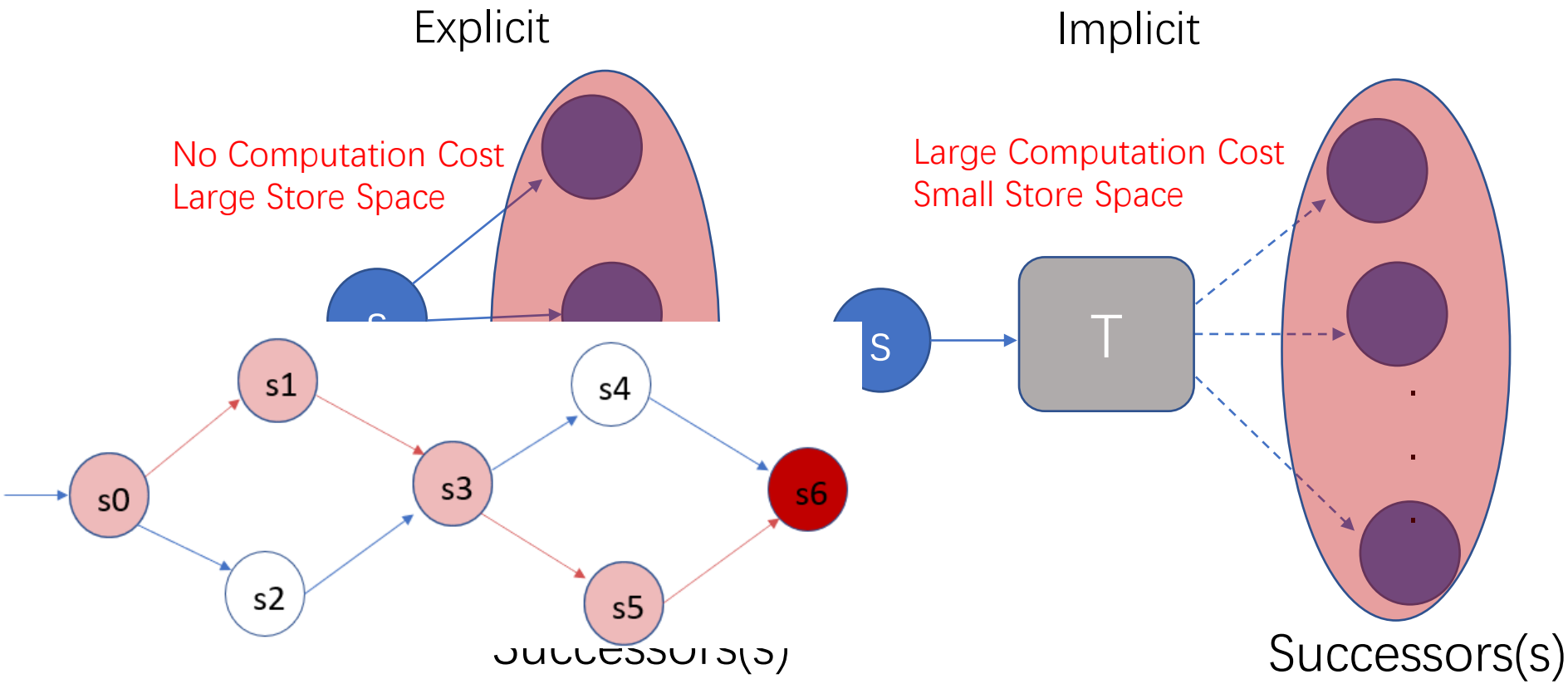
显示(Explicit)模型 vs. 隐式(Implicit)模型



显示(Explicit)模型 vs. 隐式(Implicit)模型



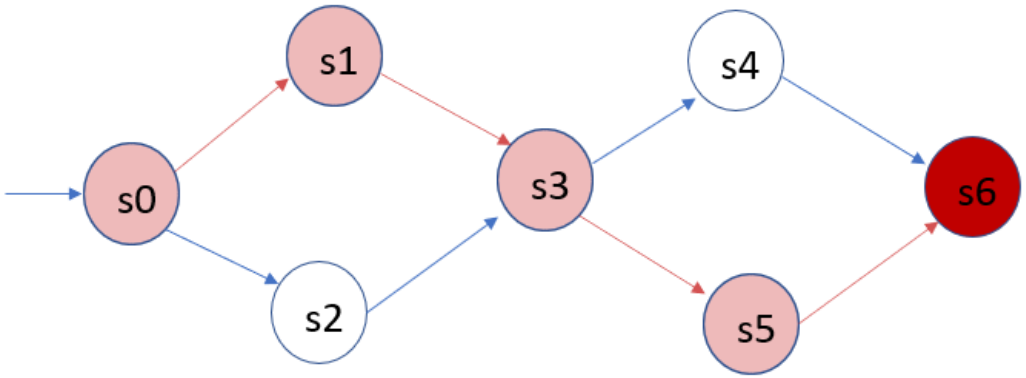
显示(Explicit)模型 vs. 隐式(Implicit)模型



显示(Explicit)模型 vs. 隐式(Implicit)模型

Explicit

No Computation Cost
Large Store Space



successors(s)

Implicit

Large Computation Cost
Small Store Space