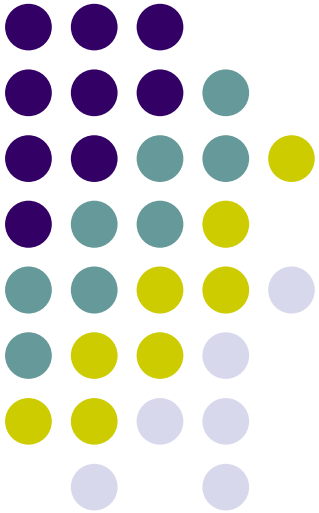
南京大学
Nanjing University

第十二讲 模态逻辑与模型检验



内容提要

- 模态逻辑 **Modal Logic**
  - 从静态真理到动态性质
  - 程序状态形式化描述：转换系统
  - 模态逻辑的算子：□ / ◊
- 线性时序逻辑 **Linear Temporal Logic**
  - **LTL**的算子：**G / F / X / U**
  - 计算机世界的时钟
- 模型检测 **Model Checking**
  - 计算树逻辑**CTL**
  - 现代模型检验技术

Part1- 模态逻辑

静态真理的局限性

- 命题逻辑/一阶语言关注数学真理
  - 给定公式**A**和模型**(M, σ)**，则$A_{M[\sigma]}$唯一确定

- **A：路口是红灯**
  - 12-01分是红灯：**A是真理**
  - 12-02分是绿灯：**¬A是真理**
- **B：x = 5**

```
x = 5
print(f"【状态 0】命题 'x == 5' 是真理吗？: {x == 5}")   # 输出: True

i = 0
while i < 5:
    x = x + 1
    print(f"【状态 2】命题 'x == 5' 是真理吗？: {x == 5}")   # 输出: False
    i = i + 1
```

可能/不可能成立的真理

- C: x = 7; D: x > 100

```
x = 5
i = 0
while i < 5:
    x = x + 1
    print(f"'x == 7'是真理吗? : {x == 7}" +"#\t#"+ f"'x >100 ' 是真理吗? : {x > 100}")
    i = i + 1
```

```
In [23]: %runfile C:/Users/borak/.spyder-py3/temp.py --wdir
'x == 7'是真理吗? : False#   #'x >10 ' 是真理吗? : False
'x == 7'是真理吗? : True#    #'x >10 ' 是真理吗? : False
'x == 7'是真理吗? : False#   #'x >10 ' 是真理吗? : False
'x == 7'是真理吗? : False#   #'x >10 ' 是真理吗? : False
'x == 7'是真理吗? : False#   #'x >10 ' 是真理吗? : False
```

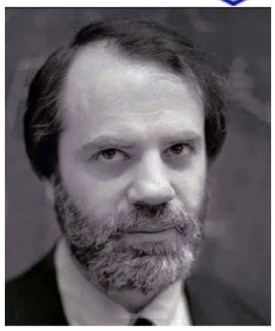
一定会/一定不会成立的真理

- **E: x < 0;　　F: x > 9900**

```python
x = 5
i = 0
while i <10000:
    x = x + 1
    print(f"'x < 0'是真理吗? : {x < 0}" +"#\t#"+ f"'x > 9900 ' 是真理吗? : {x > 9990}")
    i = i + 1
```

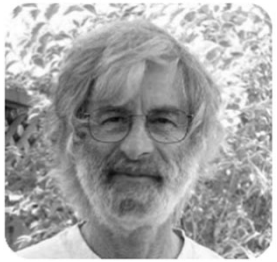
```
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
'x < 0'是真理吗? : False#    #'x > 10000 ' 是真理吗? : False
```

模态逻辑



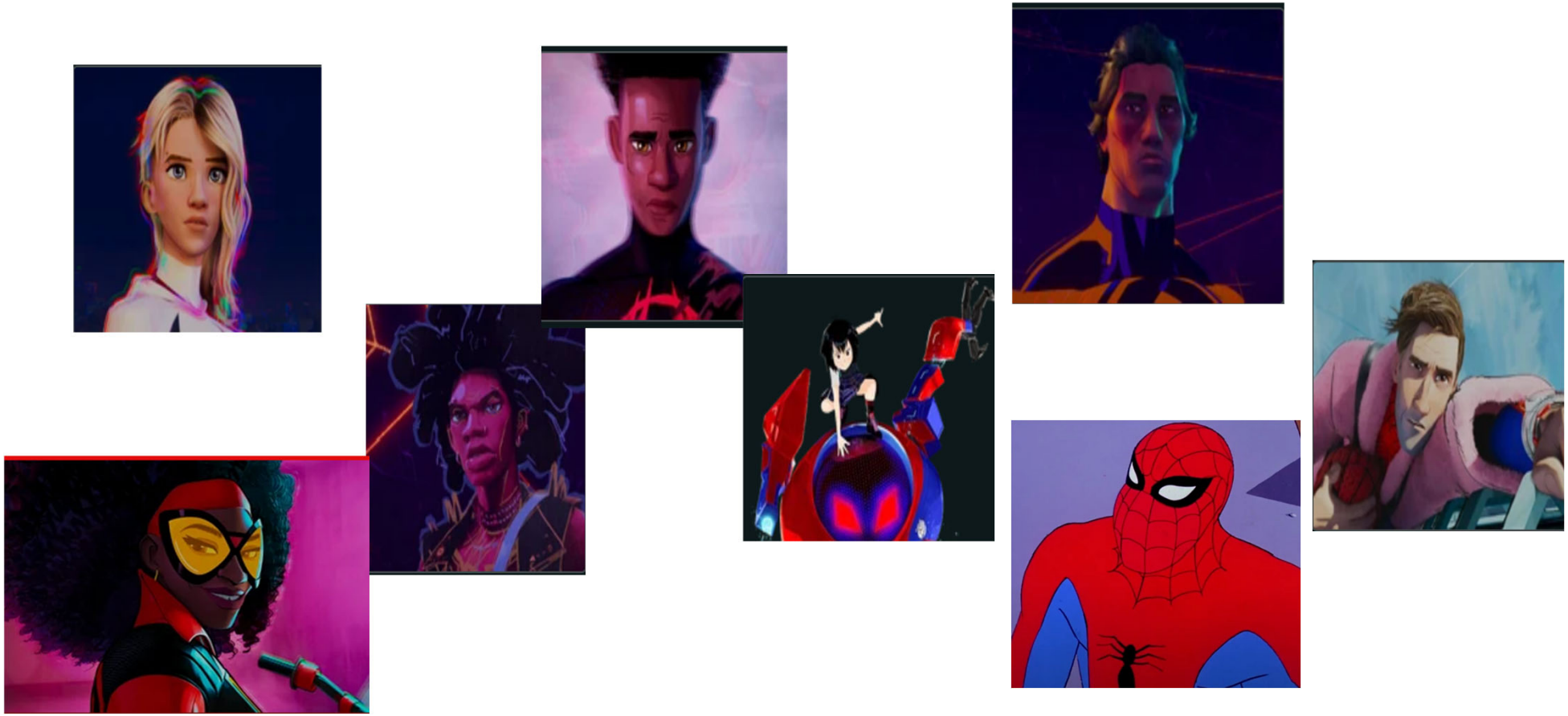
- 哲学思辨：探讨“必然”与“可能”的本质
- 数学基础：为“必然”与“可能”构建数学模型
  - Saul Kripke的“Kripke结构”

- 计算机应用：将“必然”与“可能”与系统运行状态绑定
  - Amir Pnueli：时序逻辑
  - Leslie Lamport：分布式系统逻辑时钟
  - E.Clarke, E.Emerson, J.Sifakis: 模型检验



真理与“世界”

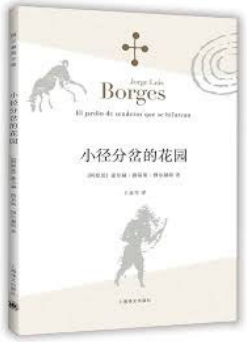
- “真理”不是孤立的，而是严格依赖于你所处的“世界”



如何描述不同的“世界”？

朴素的平行世界

- 莱布尼茨的“可能世界 possible worlds”
  - the best of all possible worlds



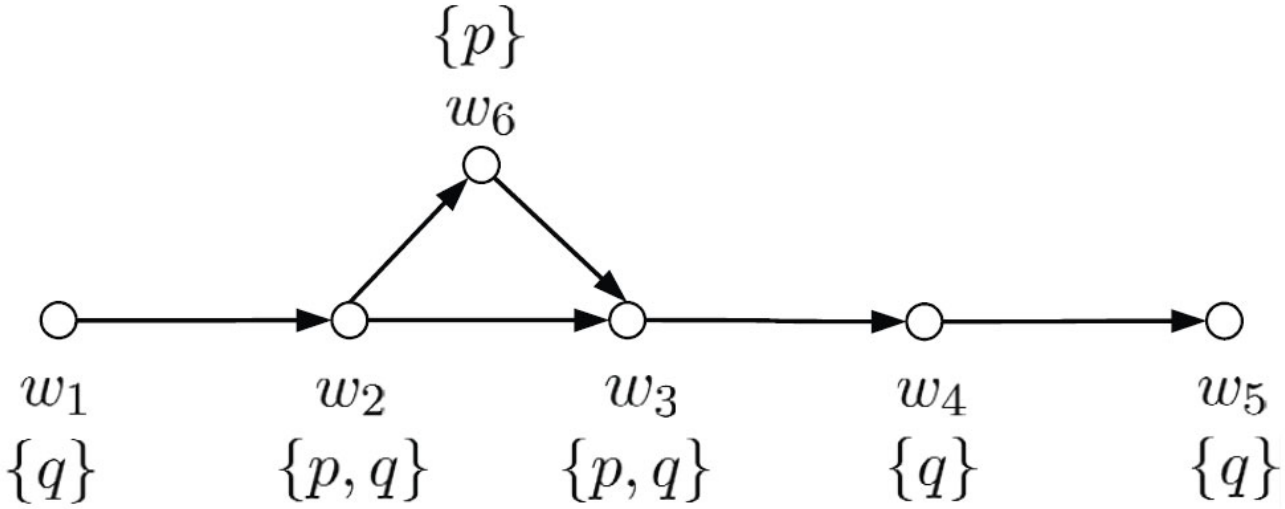
- 博尔赫斯的《小径分岔花园》

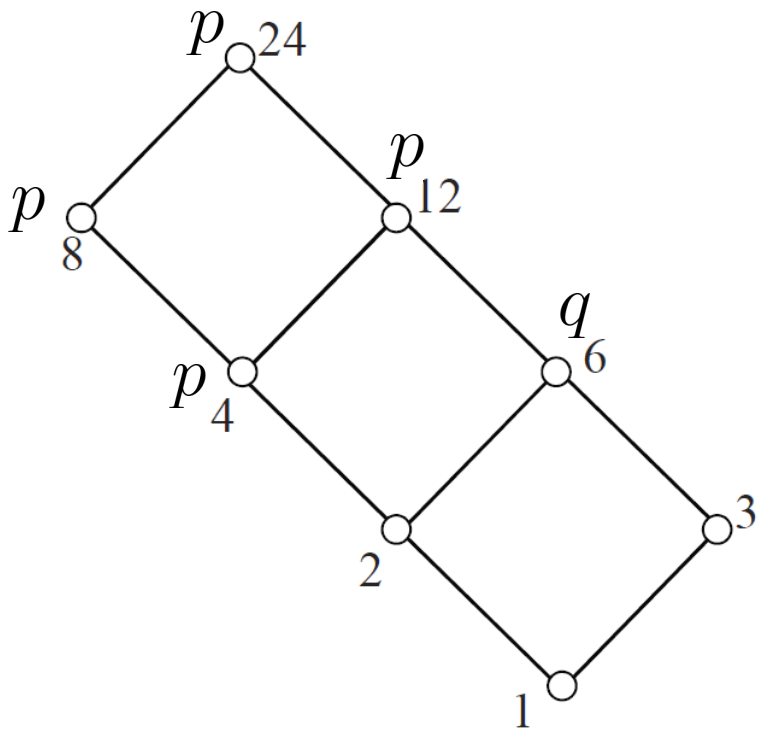
- 艾弗雷特的“多世界诠释”
  - 当对于薛定谔之猫的观测发生时，宇宙分裂成了两个平行世界

Kripke结构：形式化的平行世界

- 三元组模型**M = <W, R, V>**
- **W**：世界的集合
  - 哲学意义：所有平行世界的总和
  - 软件语义：计算机程序运行过程中所有可能的内存状态
- **R**：可达关系
  - 哲学意义：世界之间的“物理法则”或“时间推演”
  - 软件语义：由代码执行或I/O导致的内存状态间的变迁
- **V**：赋值函数
  - 哲学意义：特定世界**w**中命题的赋值情况
  - 软件语义：在特定内存状态下，变量的取值

Kripke结构的例子





标记变迁系统
Labelled Transition System

- 一种描述计算机系统运行的底层抽象 **M (S, Act, ->)**
- **S:** 状态集合
  - 所有可能的内存快照的集合
- **Act:** 动作集合
  - 触发状态改变的时间，如赋值、I/O中断，网络通信等
- **->:** 变迁关系
  - 在特定状态触发特定动作后，跃迁到下一个状态

$$s1 \xrightarrow{act} s2$$

标记变迁系统
Labelled Transition System

- 一种描述计算机系统运行的底层抽象 **M (S, Act, ->)**

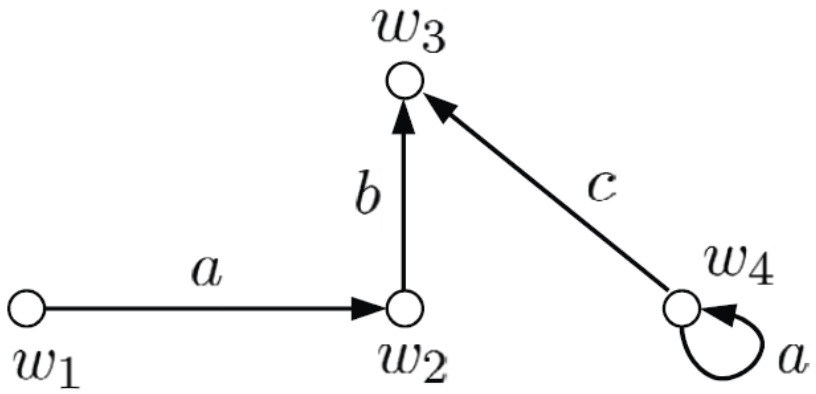


图 15.2: 一个确定转换系统

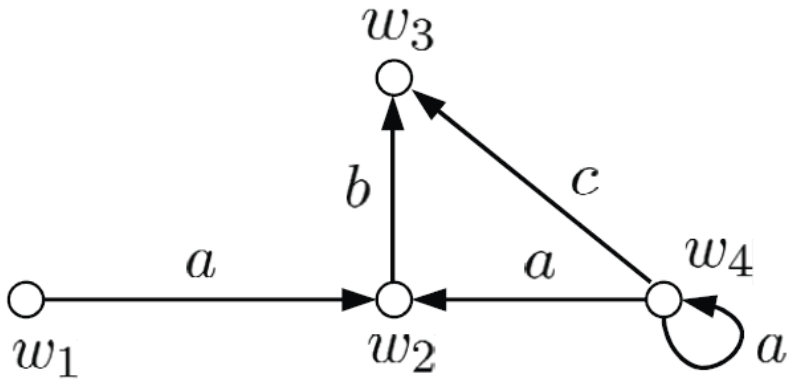
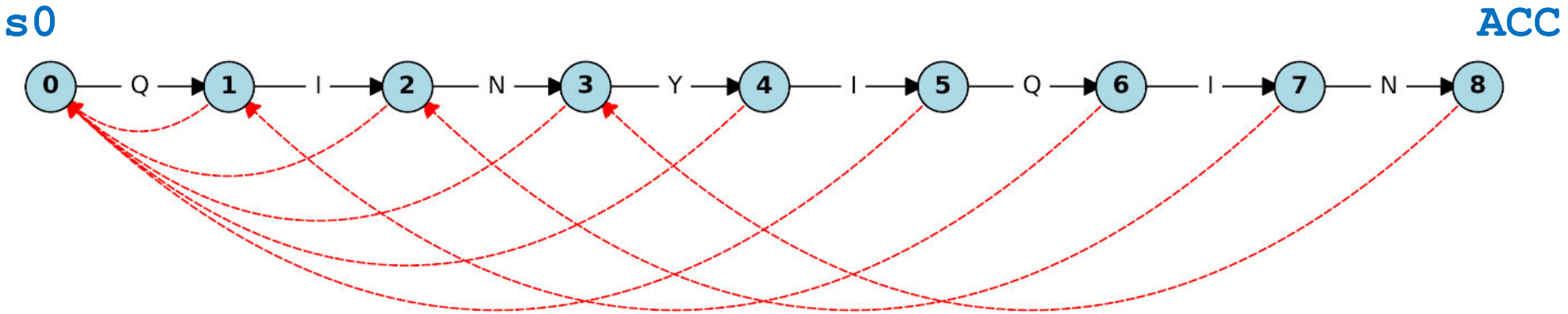


图 15.3: 一个非确定转换系统

自动机 Automata

- 一个添加了两个特殊状态的**LTS**
  - 初始状态**s0**：规定了转换系统的合法起点
  - 接收状态**ACC**：规定了转换系统终止运行的状态



- **NEXT("QINYIQIN")=**

  [0, 0, 0, 0, 0, 1, 2, 3]

必然算子 □

- 在**Kripke结构**模型**M**的世界**w**中，□**P**为真，当且仅当对于所有满足**wRw′**的世界**w′**，**P**在**w′**中都为真
  - 直观描述：在当前世界一步之内的所有平行世界中都满足的性质
  - 程序语义：下一条语句执行之后，系统一定满足的性质

```python
def process_transaction(current_balance, request_amount):
    if request_amount > 0:
        # 分支 A：存款
        new_balance = current_balance + request_amount
    elif request_amount < 0 and current_balance >= abs(request_amount):
        # 分支 B：合法取款
        new_balance = current_balance + request_amount
    else:
        # 分支 C：非法取款（余额不足），拒绝交易
        new_balance = current_balance
    assert new_balance >= 0, "致命错误：余额穿透！"

    return new_balance
```

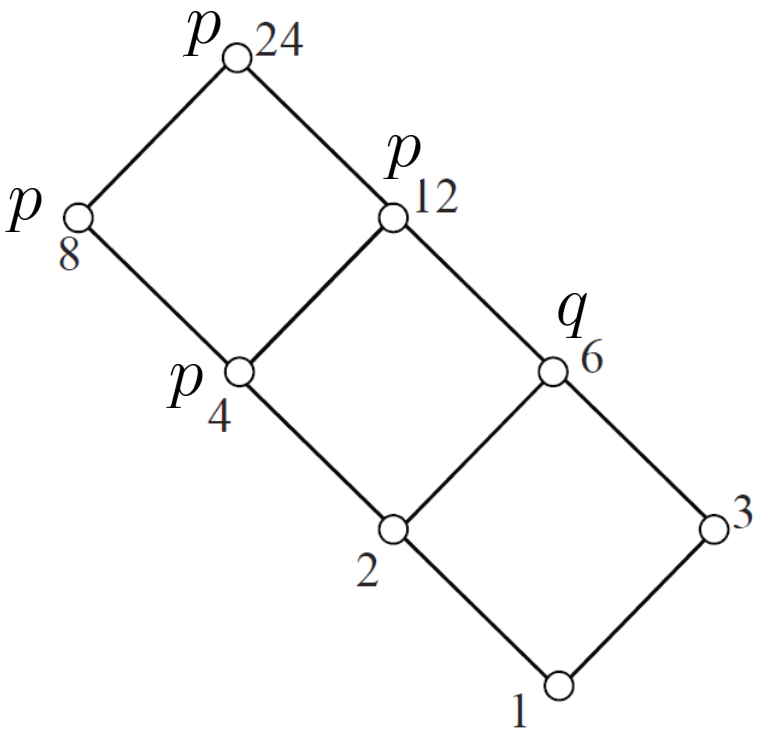
可能算子 ◊

- 在**Kripk结构**模型**M**的世界**w**中，**◊P**为真，当且仅当存在至少一个满足**wRw′**的世界**w′**，**P**在**w′**中都为真
  - 直观描述：在当前世界一步之内的存在一个平行世界满足的性质
  - 程序语义：下一条语句执行之后，系统可能（概率一定不为0）存在的性质

```python
import random

def fetch_data_with_retry():
    max_retries = 3
    current_try = 0
    while current_try < max_retries:
        network_status = random.choice(["TIMEOUT", "CONNECTED"])
        if network_status == "CONNECTED":
            return "Success: Data Fetched"
        else:
            current_try += 1  # 状态转移：重试次数加一
    return "Fatal: DB Unreachable"
```

更多的例子



- $\mathfrak{M}, 4 \Vdash \Box p$;
- $\mathfrak{M}, 6 \Vdash \Box p$;
- $\mathfrak{M}, 2 \nVdash \Box p$; 以及
- $\mathfrak{M}, 2 \Vdash \Diamond(q \wedge \Box p) \wedge \Diamond(\neg q \wedge \Box p)$。

Part2- 线性时序逻辑

一个用户登录的例子

- 算法视角：
  - 给定输出(用户名+密码)，输出结果(是否成功登录)
  - **正常状态下直接停机**
- 应用：
  - 监听端口(用户名+密码)，处理请求 (给与登录连接)
  - **正常状况下永不结束**

```python
def verify_login(username, password):
    """验证一次即结束（停机）"""
    db = {"alice": "12345"}
    if username in db and db[username] == password:
        return True
    return False
# 验证: assert verify_login("alice", "12345") == True
```

```python
def login_server():
    """永远运行的认证服务（不停机）"""
    while True:
        req = wait_for_request()   # 1. 接收请求
        if check_db(req):          # 2. 验证密码
            do_heavy_logging()     # 3. 耗时的日志写入
            send_success(req)      # 4. 发送响应
```

Kripke模态逻辑的局限性

- 对于用户登录模块的功能性需求：
  - 永远不会宕机
  - 用户正确的登录请求一定会被响应
- **必然算子□的局限性：**无法进行遍历
  - **□(不宕机)**只能保证系统的下一步执行不会宕机
  - 验证永远不宕机**需要写出无穷嵌套的□□□...**
- **可能算子◊的局限性**：无法保证在到达指定状态
  - **◊(响应请求)**仅仅意味存在一条连通到响应状态的路径
  - 如果路径上存在一个死循环，那么**请求无法在有限事件被被响应**

Kripke模态逻辑的局限性

- 对于用户登录模块的功能性需求：
  - 永远不会宕机
  - 用户正确的登录请求一定会被响应

```
def login_server():
    while True: # 无限循环的生命周期
        req = wait_for_request()
        # 针对 □ 的局限：
        # 我们需要“全局”的不宕机，但 □ 只能看透接下来的一两行代码
        if check_db(req):
            while logging_queue_is_full():
                wait() # 针对 ◇ 的局限：系统可能在这里陷入无限等待
            # 即使图上画着这条路，由于上面可能死循环，
            # 实际运行轨迹可能永远达不到下面这行！
            send_success(req)
```

线性时序逻辑LTL

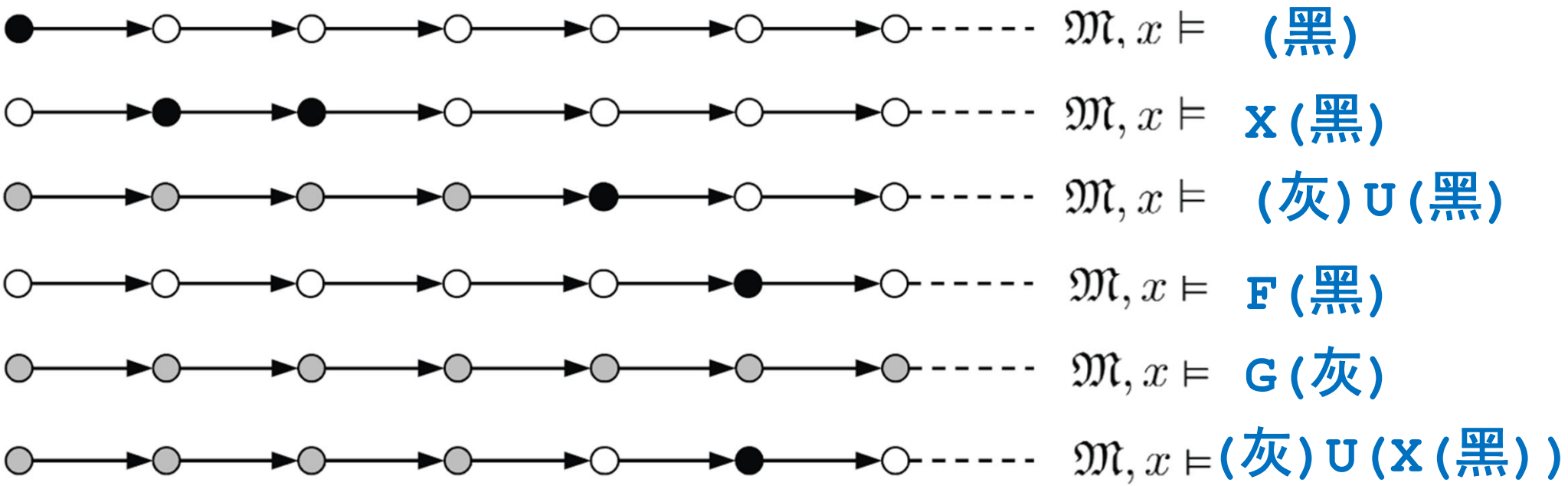
- **Amir Pnueli:** 我们无需关心所有可能世界，而只需专注于系统执行的线性轨迹**(program trace)**

- **LTL**中算子描述了程序轨迹上的线性性质，而非可能状态间的依赖关系
  - **G算子 (Globally):** 所有时刻的状态都满足的性质
  - **F算子 (Finally):** 在某个时刻的状态满足的性质
  - **X算子 (Next):** 在下一个时刻的状态满足的性质
  - **U算子 (Until):** 在事件Q之前必须满足的性质

[1] Pnueli, A. (1977). The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, 46-57.

线性时序逻辑LTL

- **Amir Pnueli:** 我们无需关心所有可能世界，而只需专注于系统执行的线性轨迹(program trace)



G算子和F算子

- **G算子：**通常用于描述系统的安全底线**（Safety）**
  - 无论系统运行多久，绝不可能出现未校验密码的登录
  - **G （登陆成功 -> 密码已校验）**
- **F算子：**通常用于描述系统的功能性目标**（Liveness）**
  - 只要接收到了用户的合法请求，无论中间经历多长时间的运行，系统迟早会给出响应
  - **G （收到请求 -> F(发送响应)）**

```
def login_server():
    while True:
        req = wait_for_request()
        # [G 的约束点]：任何时候执行到 send_success，前置条件必须已满足。
        if check_db(req):
            do_heavy_logging()
            # [F 的约束点]：只要进了这个 if 分支，这条时间线未来必定会穿过下面这一行。
            send_success(req)
```

X算子和U算子

- **X算子：**通常用于描述语句/指令级别的依赖关系
  - 密码校验通过后，紧接着的第一件事必须是记录安全日志，中间不能被插入任何其他敏感操作
  - **密码已校验 -> X （执行日志记录）**
- **U算子：**通常用于描述整个轨迹上的依赖关系
  - 在用户取得成功登录令牌**（事件Q）**之前，系统不得允许数据访问**（事件P）**
  - **（拒绝访问） U （登录令牌下发）**

```
if check_db(req):    # 条件达成
            # [X 的约束点]：校验成功后的“下一步”必须是日志
            do_heavy_logging()
            send_success(req) # 令牌下发（Q发生）
        # [U 的约束点]：在 send_success 执行之前，
        # 任何尝试访问 user_data() 的操作都会被 Until 规则阻挡。
```

更复杂的用户登录

- 现代软件系统往往使用**分布式架构**
  - 客户端与服务器端通过网络连接
  - 服务器段内部的各个模块基于微服务架构部署在云端
- 分布式架构下的银行用户登录
  - **模块A：**部署在客户端，用于登录校验
  - **模块B：**部署在银行服务器，负责用户数据管理
  - **安全性需求：**任何情况下，转账操作必须发生在合法登录之后
    - `G (执行转账 -> logged == True)`

更复杂的用户登录

- 单机架构下的银行用户登录
  - 12:00:00 状态更新：已登录
  - 12:00:10 执行操作：用户转账
  - G(执行转账 -> logged == True)

```
def user_login_on_A():
    if verify_password():
        set_state(logged=True)
        time_A = get_perfect_physical_time()
        log(f"[{time_A}] 状态更新：已登录")
        send_to_Bank("Token_OK")

def transfer_money_on_B():
    wait_for("Token_OK")
    time_B = get_perfect_physical_time()
    log(f"[{time_B}] 执行操作：余额转账")
```

更复杂的用户登录

- 分布式架构下的银行用户登录
  - **12:05:00** 状态更新：已登录
  - **12:00:10** 执行操作：用户转账
  - **G(执行转账 -> logged == True)**

```
def user_login_on_A():
    set_state(is_logged_in=True)
    # A 机器"自以为"现在是 12:05
    time_A = "12:05:00"
    log(f"[{time_A}] 状态更新：已登录")
    send_to_Bank("Token_OK")

def transfer_money_on_B():
    wait_for("Token_OK")
    # B 机器"自以为"现在才 12:00
    time_B = "12:00:05"
    log(f"[{time_B}] 执行操作：余额转账")
```

分布式系统的逻辑时钟

- **Leslie Lamport:** 既然绝对时间不可靠，就只能依靠“逻辑时间”
  - 验证**LTL**不需要知道每个节点运行轨迹上的所有事件发生的绝对时间
  - 验证**LTL**只需要知道影响系统功能的关键事件间的“因果关系”
- 逻辑时钟 **Happens-before ->**
  - 同一节点：**（先执行的） A -> （后执行的） B**
  - 不同节点：**（发送消息的） C -> （接收消息的） D**
  - 传递闭包：**A->B 且 B->C，则 A->C**

[2] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 558-565.

基于逻辑时钟的用户登录

- **Lamport逻辑时钟**：每个节点维护自己的本地时钟 C_i
  - 节点进行本地操作时，将C_i的值递增1
  - 节点发送消息时，同时发送当前的C_i的值
  - 节点接收到消息时：
    - 消息附带的C_j的值小于C_i，不做操作
    - 消息附带的C_j的值大于C_i，将C_i置为C_j+1

```
clock_A = 0
def login_A():
    global clock_A; clock_A += 1 # 发生事件，逻辑时间走动
    set_state(logged=True)
    log(f"[逻辑时间 {clock_A}] 状态更新：已登录")
    send_to_Bank(msg="Token_OK", timestamp=clock_A)
```

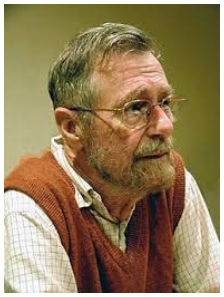
clock_A = 1
状态更新：已登录

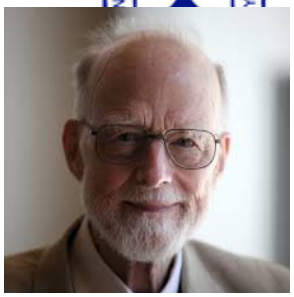
```
clock_B = 0
def receive_and_transfer_B(msg, sender_timestamp):
    global clock_B
    # 【算法灵魂】：向因果律低头！
    # 收到 A 的时间戳，强行将自己的时间拨快到比对方更大
    clock_B = max(clock_B, sender_timestamp) + 1
    clock_B += 1 # 准备转账，逻辑时间再次走动
    log(f"[逻辑时间 {clock_B}] 执行操作：余额转账")
```

clock_B = 3
执行操作：用户转账

Part3- 模型检验

如何保证程序的正确性





- 传统计算机科学家的做法：
  - 在程序中插入**前置/后置条件**
  - 人工推演**循环不变式**以验证**前置/后置条件**是否成立

```
# 目标：计算 A * B（假设 A, B >= 0）
def fast_multiply(A, B):
    x, y, z = A, B, 0
    # Pre-condition: x == A ∧ y == B ∧ z == 0
    while x > 0:
        [redacted]
        if x % 2 != 0:
            z = z + y
        y = y * 2
        x = x // 2
    # Post-condition: x == 0 ∧ z == A * B
    return z
```

Pre-condition:
程序/代码块起始时的某种性质

loop-invariant:
循环过程中保持的某种性质

Post-condition:
程序/代码块结束时的某种性质

如何保证程序的正确性

- 通过“计算”进行逻辑验证
  - 将目标程序的所有状态空间描述为一张图（LTS）
  - 将需要验证的属性描述为逻辑公式（LTL/CTL）
  - 遍历整个图，判断每条trace/节点是否均满足目标逻辑公式

```python
# 目标：计算 A * B (假设 A, B >= 0)
def fast_multiply(A, B):
    x, y, z = A, B, 0
    # Pre-condition: x == A ∧ y == B ∧ z == 0
    while x > 0:
        if x % 2 != 0:
            z = z + y
        y = y * 2
        x = x // 2
    # Post-condition: x == 0 ∧ z == A * B
    return z
```

s0:x=3, y=4, z=0

s1:x=1, y=8, z=4

s2:x=0, y=6, z=12

霍尔逻辑 vs. 模型检验

- 霍尔逻辑：
  - 证明计算机程序代码中的“数学定理”
  - 优点：直接覆盖所有输入空间
  - 缺点：难度巨大，复杂度超出人类极限

- 模型检验：
  - 基于实例和图搜索，求程序运行过程的“算术解”
  - 优点：工程化，可行性高
  - 缺点：状态空间爆炸

计算树逻辑CTL

- **LTL**的局限性：只能描述一条程序执行轨迹上的性质
- **CTL**：将程序轨迹展开为其所有可能的执行路径的聚合
  - 若程序从某个节点开始其状态存在不同的可能性**(外部输入/分支/系统终端)**，则计算树会分叉

```
def thread_A():
    global x, y
    # 【分叉点 1：外部输入】
    user_cmd = wait_for_network_packet()
    # 【分叉点 2：条件分支】
    if user_cmd == "ADD":
        x = x + 1
    else:
        x = x - 1
    # 【分叉点 3：系统调度/并发中断（最隐蔽的非确定性）】
    result = x + y
    print(result)
def thread_B():
    global y
    y = y + 100 # 这个操作随时可能像幽灵一样切入 thread_A 的执行流
```

计算树逻辑CTL的量词

- **CTL**在**LTL**的线性路径算子基础上增加了两个量词，用于在计算树中选择关注的程序轨迹
  - **CTL**的公式中，量词必须和算子同时组合出现
- **A算子**(**All**/必然/所有路径)
  - 从当前节点出发的所有程序轨迹，通常关注**safety**

```
import threading
temp = 20; heater_on = False

def thread_1_cooler():      # 模拟空调制冷
    global temp, heater_on
    temp = temp - 5; heater_on = False

def thread_2_heater():    # 模拟打开加热器
    global heater_on, temp
    heater_on = True; temp = temp +5

t1 = threading.Thread(target=thread_1_cooler)
t2 = threading.Thread(target=thread_2_heater)
t1.start()
t2.start()
```

AG(temp < 20 -> heater_on == False)

只要气温低于20度，不允许开启加热器

计算树逻辑CTL的量词

- **CTL**在**LTL**的线性路径算子基础上增加了两个量词，用于在计算树中选择关注的程序轨迹
  - **CTL**的公式中，量词必须和算子同时组合出现
- **E算子**(**Exists**/存在/至少一条路径)
  - 从当前节点出发的所有程序轨迹，通常关注**liveness**

```
import threading
temp = 20; heater_on = False

def thread_1_cooler():      # 模拟空调制冷
    global temp, heater_on
    temp = temp - 5; heater_on = False

def thread_2_heater():    # 模拟打开加热器
    global heater_on, temp
    heater_on = True; temp = temp +5

t1 = threading.Thread(target=thread_1_cooler)
t2 = threading.Thread(target=thread_2_heater)
t1.start()
t2.start()
```

EF (heater_on == True)

存在一条轨迹，使得加热器最终出于开启状态

CTL的算子组合

- **AG(P)：** 绝对安全
  - 从当前出发，所有的未来路径上，永远满足**P**。
- **AF(P)：** 必然的最终结果
  - 从当前出发，所有的未来路径，最终都必定会走到满足**P**的状态。
- **EF(P) ：** 可达的希望
  - 从当前出发，存在至少一条路径，最终能走到满足**P**的状态。
- **EG(P)：** 可能的缺陷
  - 从当前出发，存在至少一条路径，永远满足**P**。

CTL vs. LTL

- 描述“世界”的视角差异
  - LTL的视角：站在“线上”看未来
  - CTL的视角：站在“树上”看未来

- CTL能表达，但是LTL不能表达
  - 系统总是有可能被重置回 25 度的安全初始状态
  - `AG(EF(temp==25)`
- LTL能表达，但是CTL不能表达
  - 如果系统的温度最终超过 30 度，那么制冷器就必定会开启
  - `GF(temp > 30) -> GF(cooler_on)`

标记算法 Labeling Algorithm

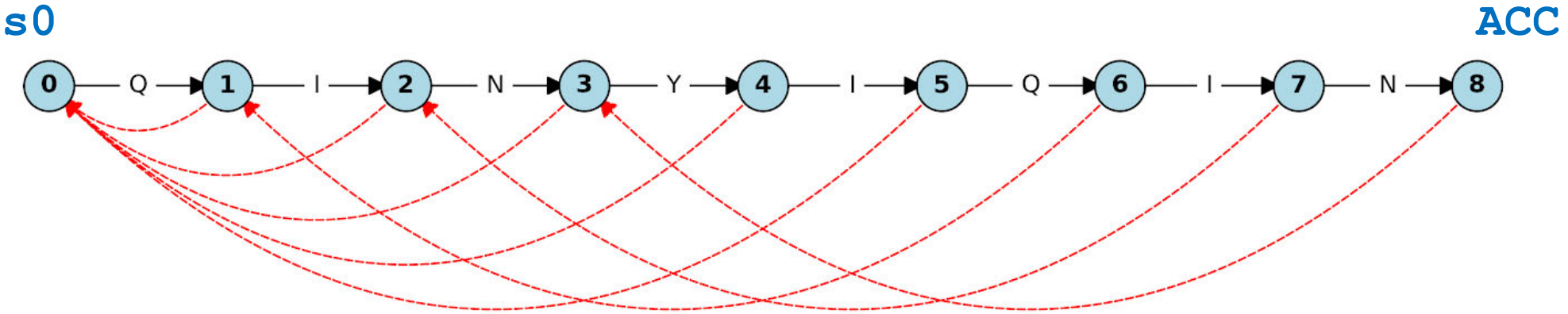
- 核心思想：
  - 将CTL公式的检测问题转化为图的遍历问题
  - 通过两层遍历降低算法的复杂度

- 外层遍历：转换CTL公式
  - 将CTL公式转化为抽象语法树AST
  - 对AST进行后序遍历
- 内层遍历：遍历目标系统的状态空间
  - 根据AST的遍历结果在系统状态图上进行标记
  - 根据系统状态图的遍历结果传播标记信息

又是KMP！

NEXT(“QINYIQIN”)=

[0, 0, 0, 0, 0, 1, 2, 3]



- KMP vs. Labeling Algorithm
  - next数组：外层遍历，将CTL公式转化为AST
  - 扫描目标数组：内层遍历，在状态图上进行标注

状态空间爆炸

- 为什么KMP算法能够work？
  - 目标数组：1维空间
  - next数组：1维空间
- 目标系统的状态空间：爆炸！
  - 数据变量组合
    - Boolean变量 ： 2 个状态。
    - Integer变量：2^32个状态
  - 并发/分布式行为组合
    - 每个线程的状态
    - 线程间交互行为的状态

现代模型检验技术

- 从状态空间本身入手：符号模型检验
  - 利用高阶数据结构（例如BDD）来抽象地表示状态
- 更高效地探索状态空间：搜索和剪枝
  - 利用状态间的dependency，大幅削减计算树和目标程序状态
- 转换目标：有界模型检验
  - 将确保全局状态的性质转化为寻找n-步范围内的反例
- 工程学改进：抽象与精化
  - 用抽象状态提升检验效率，对反例进行精化以降低误报率