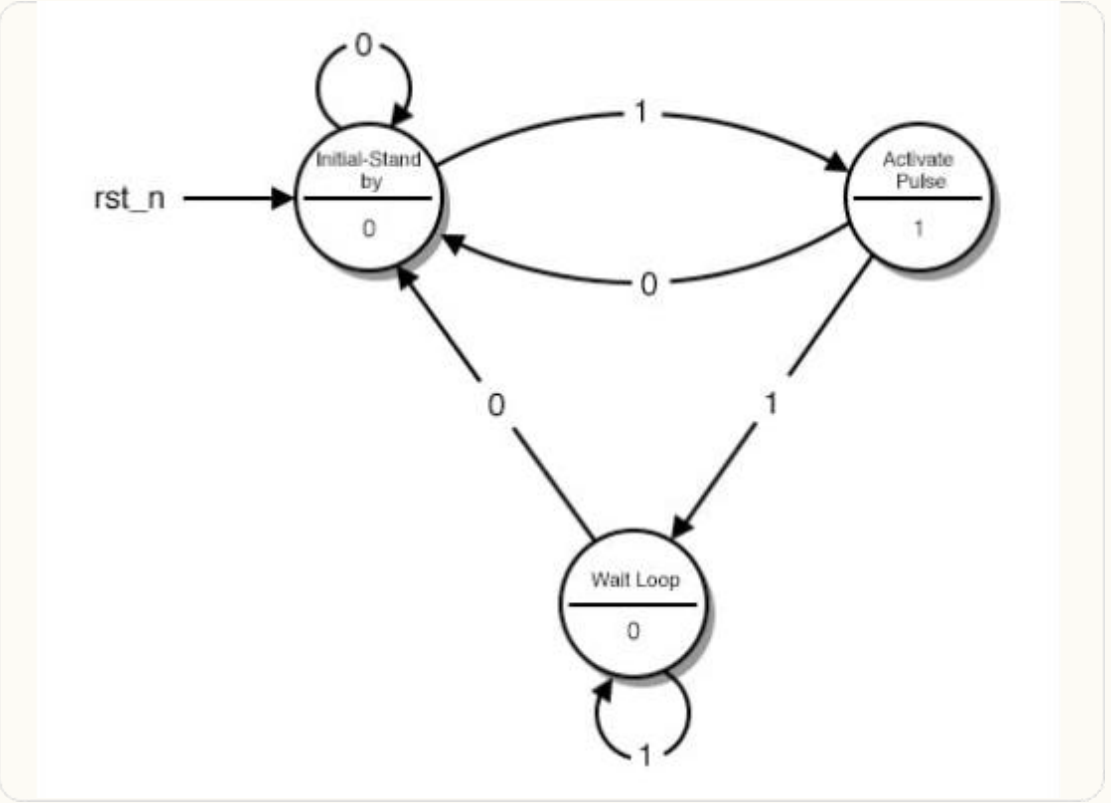
TLA+ 介绍与实践

学号：25031212338
姓名：吕彤

什么是 TLA+？

- 一种用于数字系统（算法、程序、系统）的 **高级建模语言**
- 由图灵奖得主 **Leslie Lamport** 开发。
- 它是一种规范语言 (Specification Language)，而非编程语言
- 核心思想：将系统抽象为 **状态机 (State Machine)**。
- 我们只需定义：
  - 所有合法的初始状态 **(Init)**。
  - 从一个状态转移到下一个状态的合法规则 **(Next)**。



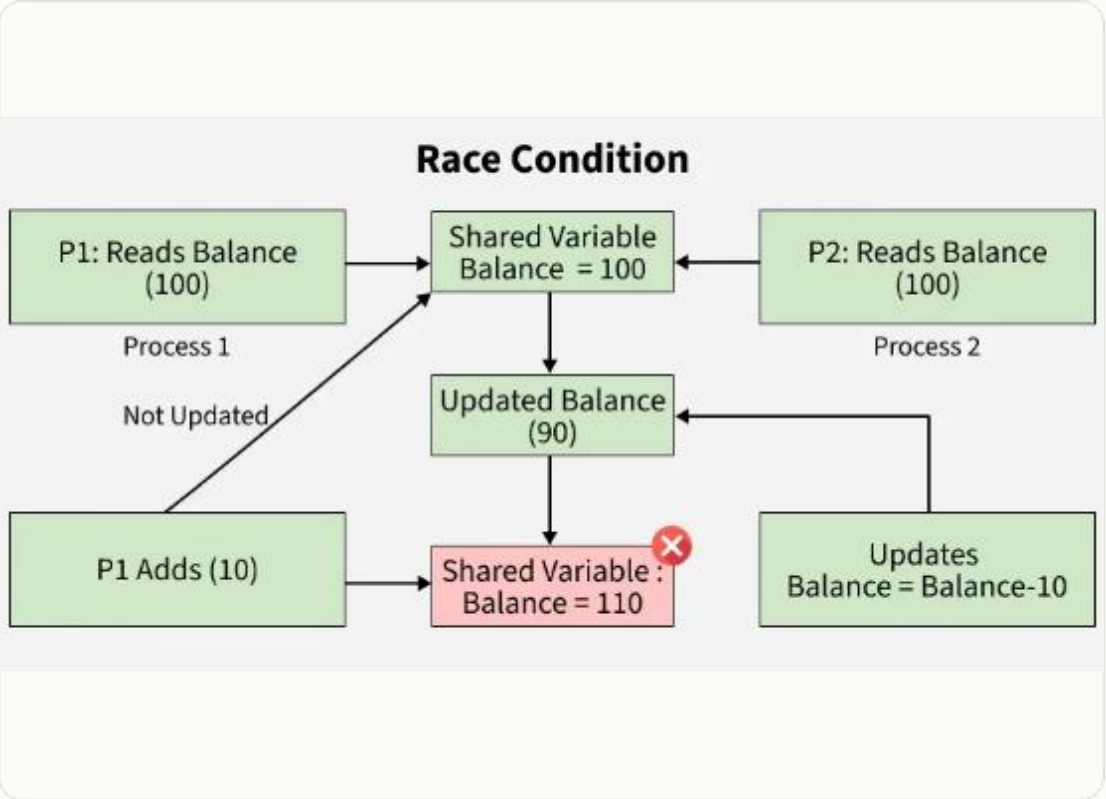
TLA+ 解决了什么问题？

传统的测试（Testing）难以覆盖复杂系统的所有可能执行路径，尤其是在分布式系统中。

痛点：

- **并发问题：** 竞态条件、死锁。
- **不确定性：** 网络延迟、消息丢失、节点失败。
- "我的设计在并发下是否安全？"
- "这个算法会不会导致数据不一致？"

TLA+ 允许我们在编写代码之前，对系统设计进行穷举检查，发现深层次的设计缺陷。



TLA+ 生态系统：TLA+、PlusCal、TLC 的关系

PlusCal (P-Syntax)

一种我们用来编写算法的伪代码语言。语法类似 Pascal，对工程师友好。我们工作的起点。

TLA+ (M-Syntax)

PlusCal 算法的“编译”产物。它是纯粹的、数学化的 TLA+ 规范（即 Init 和 Next 的定义）。

TLC (Model Checker)

TLA+ 规范的“运行”和“验证”工具。TLC 会穷举系统所有可能的执行路径，检查是否违反了我们定义的属性。

流程: 我们编写 PlusCal ➔ [翻译器] ➔ 生成 TLA+ 规范 ➔ [TLC模型检测器] ➔ 输出验证结果

PlusCal ➔ TLC：完整的验证流程

1. **编写 (Write):** 在 TLA+ 模块中编写 PlusCal 算法 (*--algorithm ... *)。

2. **翻译 (Translate):** 运行 PlusCal 翻译器 (例如在 TLA+ Toolbox 中点击 "Translate Algorithm")。

3. **生成 (Generate):** 翻译器自动生成 TLA+ M-Syntax (即 Init, Next, Spec) 规范。

4. **建模 (Model):** 创建一个 TLC 模型，在模型中指定要检查的内容。

5. **配置 (Configure):**

   检查 **Deadlock** (死锁)

   添加 **Invariants** (不变式，例如: NoInconsistency, TypeOK)

   (可选) 添加 **Properties** (活性属性)

6. **运行 (Run):** 运行 TLC 模型检测器。TLC 开始穷举所有可能的状态。

7. **分析 (Analyze):** TLC 报告 **"No errors found."** 或 **"Error: ..."** (并提供导致错误的 Error-Trace)。

PlusCal 语法

(*--algorithm ... *): 算法定义块。

variables ...;: 定义系统的全局状态 (所有进程共享)。

process ...: 定义一个并发进程（线程）。

标签 (Labels) (例如: TmVoteReq:): 两个标签之间的代码块被视为一个原子的、不可分割的操作。

await ;: 守卫 (Guard)。进程必须等待 condition 为 TRUE 时，才能执行这个原子步骤。

either { ... } or { ... };: 非确定性 (Nondeterminism)。TLC 会探索所有分支。

goto ...;: 跳转到另一个标签。

PlusCal 编译

PlusCal 如何转换？

PlusCal 翻译器将“伪代码”转换成 Init 和 Next 状态机：

- **variables -> TLA+ VARIABLES:** 成为 TLA+ 的全局 VARIABLES。
- **“魔法”变量 pc:** 自动添加 pc (Program Counter)，保存每个进程的当前标签。 (例如: pc["TM"] = "TmDecision")
- **标签 -> TLA+ Action:** 每一个标签被翻译成一个 TLA+ 动作 (Action)。

示例: 标签到 Action

你的PlusCal代码:

```
L1:
   await x > 0:
   x:=x-1;
   goto L2;
```

翻译器生成的 TLA+ 动作 (概念上):

```
Action_L1(self) ==
      /\ pc[self] = “L1”
      /\ x > 0
      /\ x’= x – 1
      /\ pc’=[pc EXCEPT ![self] = “L2”]
      /\ UNCHANGED <- …>>
Next == \E self \in Procs: (Action_L1(self) \/
Action_L2(self) \/ ...)
```

案例：两阶段提交 (2PC) - 背景与应用

目标: 在分布式系统中实现原子事务 (Atomic Transaction)。

原子性要求: 确保所有参与者必须要么 **全部提交 (Commit)**，要么 **全部中止 (Abort)**。

角色: 1x 协调者 (TM) 和 Nx 参与者 (RM)。

流程: 1. 投票阶段 (Voting) 2. 决定阶段 (Decision)。

实际应用:

分布式数据库 (例如 MySQL Cluster, PostgreSQL)

事务性消息队列 (MSMQ)

金融支付系统 (确保支付和订单状态一致)

2PC 案例：模型变量与属性

variables (系统状态)

- tm_state: 协调者的状态
  ("init", "voting", "committed", "aborted", "failed")
- rm_state: 函数，映射每个 RM 到它的状态
  ("working", "voted_abort", "committed", ...)

process (并发进程)

- Coordinator (TM 进程)
- Participant (多个 RM 进程)

我们要检查的属性 (Invariants)

1. 一致性 (Safety): 绝不能一个 RM 提交，另一个 RM 中止。

```
Inconsistent == \E i, j \in RMs:
        rm_state[i] = "committed" /\ rm_state[j] =
"aborted"

NoInconsistency == ~Inconsistent
```

2. 类型安全 (Sanity Check): 变量永远不会处于“未定义”状态。

```
TypeOK ==tm_state \in {"init", "voting", "committed ",
"aborted", "failed"}
        /\ \A r \in RMs : rm_state[r] \in {"working",
"voted_commit",

"voted_abort", "committed", "aborted"}
```

2PC 案例：版本 1 (协调者可能崩溃)

Coordinator (TM) 关键代码

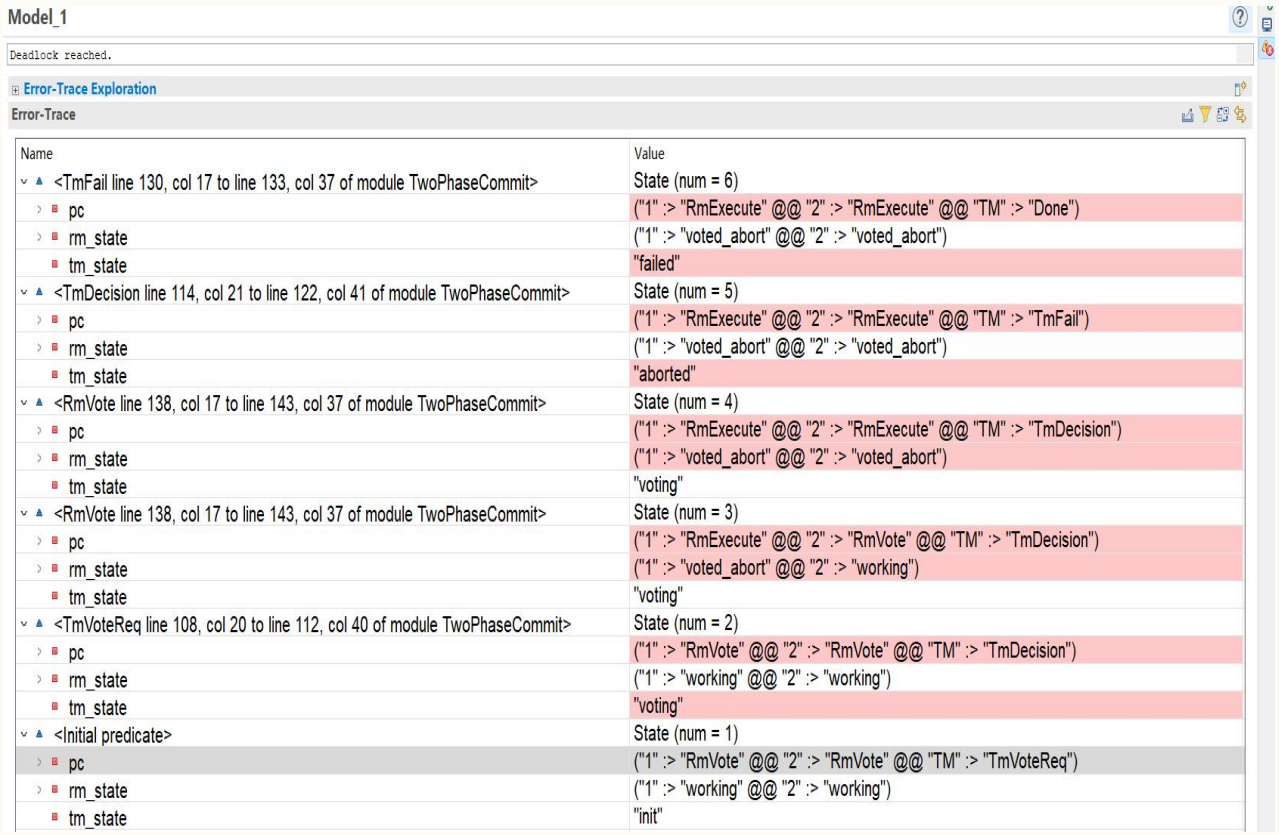
```
process (Coordinator \in {"TM"}) {
    TmVoteReq:
        await tm_state = "init";
        tm_state := "voting";
        goto TmDecision;

    TmDecision:
        await \A r \in RMs : rm_state[r] \in {"voted_commit", "voted_abort"};

        either
            {
                await \A r \in RMs : rm_state[r] = "voted_commit";
                tm_state := "committed";
            }
        or
            {
                await \E r \in RMs : rm_state[r] = "voted_abort";
                tm_state := "aborted";
            }
        ;

        either
            { goto TmEnd; }
        or
            { goto TmFail; }

        ;

    TmEnd:
        await \A r \in RMs : rm_state[r] \in {"committed", "aborted", "working"};
        tm_state := "init";
        goto TmVoteReq;

    TmFail:
        tm_state := "failed";
        \*skip;
}
```

Participant (RM) 关键代码

```
process (Participant \in RMs) {
    RmVote:
        await tm_state = "voting" /\ rm_state[self] = "working";

        either
            { rm_state[self] := "voted_commit"; }
        or
            { rm_state[self] := "voted_abort"; }
        ;

        goto RmExecute;

    RmExecute:
        await tm_state \in {"committed", "aborted", "failed"};

        if (tm_state = "committed") {
            rm_state[self] := "committed";
            goto RmEnd;
        } else if (tm_state = "aborted") {
            rm_state[self] := "aborted";
            goto RmEnd;
        } else {
            \*skip;
            \*goto RmExecute;
            await FALSE;
        };
        \* goto RmEnd;

    RmEnd:
        \* await tm_state = "init" \/ tm_state = "failed";
        rm_state[self] := "working";
        goto RmVote;
}
```

2PC 错误分析：TLC 发现死锁!

TLC 检查 **Deadlock (死锁)** 时，报告 **Error: Deadlock reached.**



1 **State 1 (故事开始):** tm_state: "init", rm_state: [rm ->"working"]

↓ **Action:** 协调器 (TM) 行动。

2 **State 2 (协调器已请求投票):** tm_state: "voting"

↓ **Action:** (x2) 两个 RM (参与者) 相继行动。

3 **State 3 & 4 (所有参与者均已投票):** rm_state: [rm -> "voted_abort"] (两个RM都投了反对票)

↓ **Action:** 协调器 (TM) 做出决定。

5 **State 5 (协调器决定“中止”并即将崩溃):** tm_state: "aborted", **pc: [TM -> "TmFail"]** (关键！ TLC 探索了 "goto TmFail" 路径)

↓ **Action:** 协调器 (TM) 执行“崩溃”。

6 **State 6 (死锁状态):** tm_state: "failed", **pc: [TM -> "Done"]** (协调器进程已死), **pc: [RMs -> "RmExecute"]** (两个参与者卡住)

2PC 错误分析：死锁解读

State 5: 关键转折点

含义： 这是最关键的一步！协调器在 TmDecision 标签下有两个选择（either { goto TmEnd; } or { goto TmFail; }）。

模型检查器 (TLC) 为了测试所有可能性，在这里选择了 **"TmFail"（失败）路径**。

State 6: 死锁状态

含义： 这就是最终的死锁！

- 协调器 (TM) 进程结束 (pc["TM"] = "Done")。
- 参与者 "1" 和 "2" 仍然在 RmExecute 标签处等待。
- 它们读到 tm_state = "failed"，进入 else { **await FALSE;** } 分支。
- await FALSE 是一个永远无法满足的条件，所以两个参与者被 **永久阻塞**。

结论: 证明了如果协调者在特定窗口崩溃，整个系统将永久阻塞。

2PC 案例：版本 2 (修复版)

1. **Coordinator (TM):** 移除了 TmFail 路径。只验证“理想”情况。 TmDecision: ... (做出 commit/abort 决定) ... **goto** TmEnd; * 只有一个正常路径

```
process (Coordinator \in {"TM"}) {
    TmVoteReq:
        await tm_state = "init";
        tm_state := "voting";
        goto TmDecision;

    TmDecision:
        await \A r \in RMs : rm_state[r] \in {"voted_commit", "voted_abort"};

        either
            {
                await \A r \in RMs : rm_state[r] = "voted_commit";
                tm_state := "committed";
            }
        or
            {
                await \E r \in RMs : rm_state[r] = "voted_abort";
                tm_state := "aborted";
            }
        ;

        goto TmEnd;

    TmEnd:
        await \A r \in RMs : rm_state[r] \in {"committed", "aborted", "working"};
        tm_state := "init";
        goto TmVoteReq;
}
```

2. **Participant (RM):** 简化了 RmExecute。 RmExecute: **await** tm_state \in {"committed", "aborted"}; **if** (tm_state = "committed") { rm_state[self] := "committed"; } **else** { rm_state[self] := "aborted"; }; **goto** RmEnd;

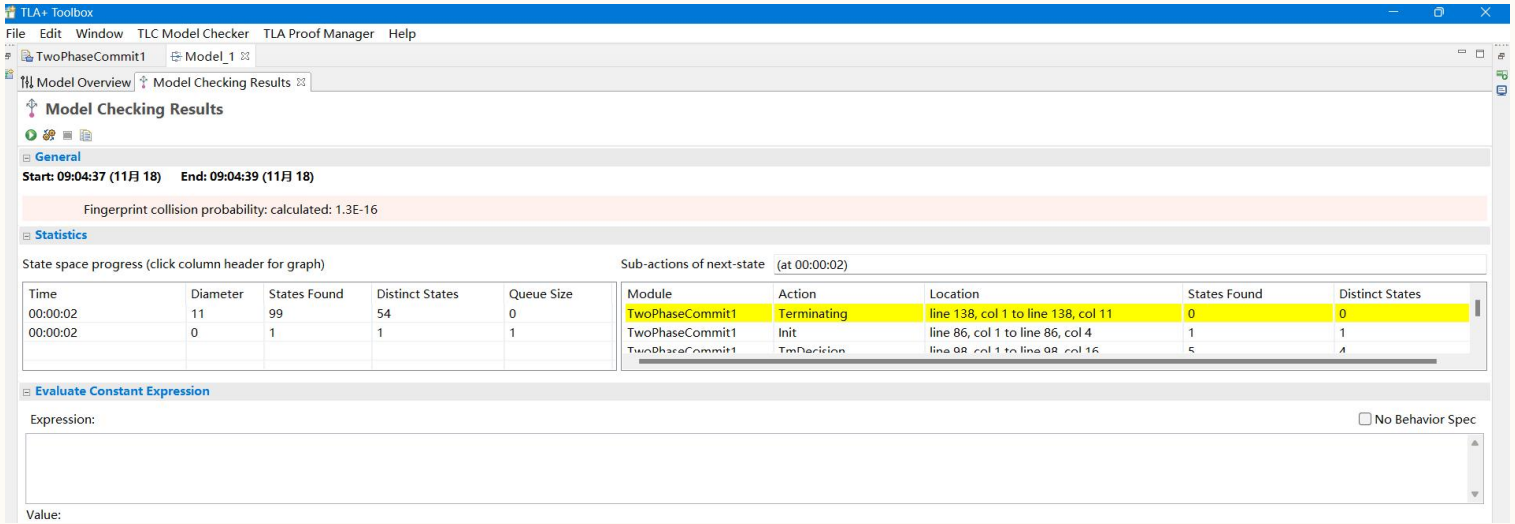
```
process (Participant \in RMs) {
    RmVote:
        await tm_state = "voting" /\ rm_state[self] = "working";

        either
            { rm_state[self] := "voted_commit"; }
        or
            { rm_state[self] := "voted_abort"; }
        ;

        goto RmExecute;

    RmExecute:
        await tm_state \in {"committed", "aborted"};

        if (tm_state = "committed") {
            rm_state[self] := "committed";
        } else {
            rm_state[self] := "aborted";
        };
        goto RmEnd;

    RmEnd:
        rm_state[self] := "working";
        goto RmVote;
}
```

2PC 案例：版本 2 结果分析 (成功!)

TLC 检查项:

Deadlock (死锁)

Invariants: NoInconsistency, TypeOK



TLC 结果: No errors found.

TLC 报告解读:

TLC 探索了 99 个状态 (States Found)，发现了 54 个不同的系统状态 (Distinct States)。

在所有路径中：没有发现死锁，且 NoInconsistency 和 TypeOK 始终保持为真。

结论: 我们证明了，在这个“理想”模型中：

Safety (安全性): 系统绝对不会出现不一致情况。

Liveness (活性): 系统不会发生死İo。

总结：我们得到了什么？

在设计阶段发现Bug: TLA+ 在我们写下任何代码之前，就发现了2PC协议中因协调者崩溃而导致的“阻塞”缺陷。

管理复杂性: TLA+ 和 TLC 充当了“超级测试员”，替我们探索了所有我们人脑无法穷举的并发交错情况。

精确的思考: PlusCal 迫使我们清晰地定义：

系统的状态是什么？

原子操作的边界在哪里？

每一步操作的前提条件 (await) 是什么？

TLA+ 不仅仅是一个工具，它是一种设计思维的革命。

感谢聆听

Q & A