RustPLC：基于形式化验证的工业控制编译器

第一性原理分析与实例演练

RustPLC 项目学习笔记

2026 年 2 月

Contents

1 引言：为什么需要 RustPLC

传统 PLC（可编程逻辑控制器）编程使用梯形图、结构化文本（ST）或功能块图（FBD），程序员手动编写控制逻辑，然后通过测试来验证正确性。这种 “先写后测” 的模式存在根本性缺陷：

- 测试只能发现 **bug**，不能证明没有 **bug**——测试覆盖的是有限路径，而工业系统的状态空间是指数级的。
- 安全约束散落在代码各处——互锁逻辑与业务逻辑混杂，难以审计。
- 时序问题难以复现——传感器延迟、气缸行程时间等物理参数在仿真中容易被忽略。

RustPLC 的核心主张是：

> 不要编写控制程序——声明物理事实和安全意图，让编译器在代码运行之前证明它是安全的。

2 第一性原理

RustPLC 的设计建立在以下不可再分的基本原理之上：

2.1 原理一：物理系统是有限状态的

工业控制中的每个设备都有有限的状态集合。气缸只有 “伸出” 和 “缩回” 两个状态，电磁阀只有 “开” 和 “关”，传感器只有 “检测到” 和 “未检测到”。整个系统的状态空间是所有设备状态的笛卡尔积：

$$\mathcal{S} = S_{\text{dev}_1} \times S_{\text{dev}_2} \times \cdots \times S_{\text{dev}_n} \times S_{\text{ctrl}}$$

其中 S_{ctrl} 是控制状态机的状态集。由于每个因子都是有限的，$\mathcal{S}$ 也是有限的——这意味着我们可以用穷举或归纳的方式证明性质。

2.2 原理二：安全性是状态空间上的不变量

“A 缸和 B 缸不能同时伸出” 这样的安全约束，本质上是对状态空间的一个划分：

$$\mathcal{S}_{\text{safe}} = \{s \in \mathcal{S} \mid \neg(\text{cyl\_A.extended} \wedge \text{cyl\_B.extended})\}$$

安全验证的任务就是证明：从初始状态出发，沿任意合法转移路径，系统永远停留在 $\mathcal{S}_{\text{safe}}$ 中。这是一个经典的模型检验（Model Checking）问题。

2.3 原理三：因果律决定信号传播路径

物理信号不能凭空出现。一个传感器要检测到气缸伸出，必须存在完整的因果链：

$$\text{数字输出} \xrightarrow{\text{电气}} \text{电磁阀} \xrightarrow{\text{气动}} \text{气缸} \xrightarrow{\text{物理}} \text{传感器}$$

如果拓扑图中这条路径断开，程序逻辑再正确也无法工作。

2.4 原理四：活性保证系统不会 “卡死”

一个安全但永远不动的系统没有价值。活性（Liveness）要求每个非终态都有出路：要么有超时跳转，要么有明确的 “允许无限等待” 标记（如等待人工按钮）。

2.5 原理五：时序约束来自物理定律

电磁阀有响应时间，气缸有行程时间，电机有加速斜坡。这些不是软件参数，而是物理常数。编译器必须沿拓扑连接链累加这些时间，计算最坏情况关键路径，并与声明的时序约束比较。

3 编译器架构

RustPLC 采用经典的多阶段编译器设计，如图 1 所示。

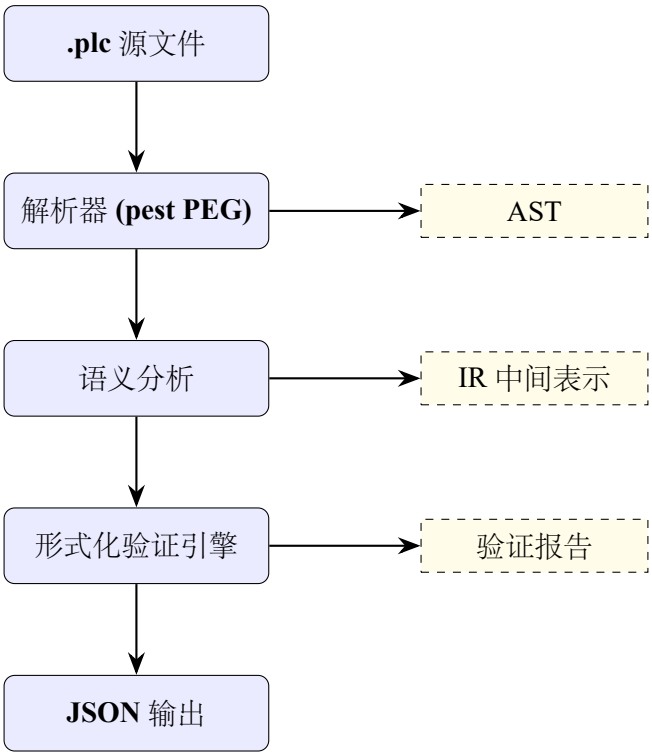

Figure 1: RustPLC 编译流水线

3.1 阶段一：解析（Parser）

使用 `pest` PEG 解析器，将 `.plc` 源文件转换为抽象语法树（AST）。语法文件 `plc.pest` 定义了 150 条规则，覆盖三大段落：

- `[topology]`——设备声明与物理连接
- `[constraints]`——安全/时序/因果约束
- `[tasks]`——控制任务与步骤（状态机）

3.2 阶段二：语义分析（**Semantic Analysis**）

将 AST 降低为四种中间表示：

IR 结构	数据结构	用途
拓扑图	`petgraph::DiGraph`	设备连接关系
状态机	状态 + 转移 + 守卫	控制流建模
约束集	安全/时序/因果规则	验证目标
时序模型	动作 → 时间区间映射	时序分析

Table 1: 四种中间表示

3.3 阶段三：形式化验证

四个独立的验证引擎并行运行：

1. 安全检查器——有界模型检验（BMC）+ k-归纳，可选 Z3 SMT 求解器
2. 活性检查器——强连通分量（SCC）分析 + 死锁检测
3. 时序检查器——最坏情况关键路径分析
4. 因果检查器——拓扑图 BFS 可达性验证

4 DSL 语言设计

每个 `.plc` 文件由三个段落组成。下面以 “传送带冲压系统” 为例逐段讲解。

4.1 拓扑段 `[topology]`

声明物理设备及其连接关系：

```
[topology]

device Y0: digital_output
device Y1: digital_output
device X0: digital_input
device X1: digital_input
device X2: digital_input
device X3: digital_input

device start_button: digital_input {
    connected_to: X3
    debounce: 20ms
}

device conveyor_motor: motor {
    connected_to: Y0
    rated_speed: 30rpm
    ramp_time: 100ms
```

```
}

device stamp_valve: solenoid_valve {
    connected_to: Y1
    response_time: 15ms
}

device stamp_head: cylinder {
    connected_to: stamp_valve
    stroke_time: 250ms
    retract_time: 200ms
}

device sensor_in_position: sensor {
    type: proximity
    connected_to: X0
    detects: conveyor_motor.position_A
}

device sensor_stamp_down: sensor {
    type: magnetic
    connected_to: X1
    detects: stamp_head.extended
}

device sensor_stamp_up: sensor {
    type: magnetic
    connected_to: X2
    detects: stamp_head.retracted
}
```

`connected_to` 表示上游连接：`stamp_head` 连接到 `stamp_valve`，意味着电磁阀驱动气缸。编译器据此构建有向拓扑图。

4.2 约束段 `[constraints]`

声明安全、时序和因果约束：

```
[constraints]

safety: stamp_head.extended conflicts_with conveyor_motor.on
    reason: "冲压头下压时传送带不能运行"

timing: task.cycle must_complete_within 3000ms
    reason: "单个冲压周期不应超过3秒"

causality: Y1 -> stamp_valve -> stamp_head -> sensor_stamp_down
causality: Y1 -> stamp_valve -> stamp_head -> sensor_stamp_up
```

- **safety**：`conflicts_with` 表示两个状态互斥，`requires` 表示蕴含。
- **timing**：`must_complete_within` 设定上界，`must_start_after` 设定下界。

- **causality**：用 `->` 链声明信号传播路径。

4.3 任务段 `[tasks]`

定义控制逻辑为状态机：

```
[tasks]

task cycle:
    step feed:
        action: set conveyor_motor on
        wait: sensor_in_position == true
        timeout: 1500ms -> goto fault_handler
    step stop_belt:
        action: set conveyor_motor off
    step press_down:
        action: extend stamp_head
        wait: sensor_stamp_down == true
        timeout: 500ms -> goto fault_handler
    step press_up:
        action: retract stamp_head
        wait: sensor_stamp_up == true
        timeout: 500ms -> goto fault_handler
    on_complete: goto ready

task fault_handler:
    step emergency:
        action: retract stamp_head
        action: set conveyor_motor off
    step report:
        action: log "冲压系统故障：动作超时"
    on_complete: goto ready

task ready:
    step wait_start:
        wait: start_button == true
        allow_indefinite_wait: true
    on_complete: goto cycle
```

5 验证引擎详解

5.1 安全检查器：有界模型检验

安全检查器的核心算法是有界模型检验（Bounded Model Checking, BMC），辅以 k-归纳（k-induction）来获得完备证明。

5.1.1 状态空间建模

系统的具体状态是一个元组：

$$\sigma = (q, d_1, d_2, \ldots, d_n)$$

其中 $q \in Q$ 是控制状态机的当前状态，$d_i \in D_i$ 是第 i 个设备的当前状态。

5.1.2 BMC 搜索

从初始状态 σ_0 出发，BFS 展开所有可达状态，直到深度 k：

$$\text{Reach}_k = \{\sigma \mid \exists \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} \sigma,\ \sigma_0 \in \mathcal{I}\}$$

对每个 `conflicts_with` 约束 ϕ，检查：

$$\forall \sigma \in \text{Reach}_k : \sigma \models \neg\phi$$

5.1.3 k-归纳

如果 BFS 在深度 k 时状态空间已收敛（无新状态），则获得完备证明：

$$\text{Reach}_k = \text{Reach}_{k+1} \implies \text{Reach}_k = \text{Reach}_\infty$$

在我们的传送带冲压示例中，安全检查器在深度 8 获得完备证明，证明了 “冲压头下压时传送带不会运行” 这一安全性质。

5.2 活性检查器：死锁检测

活性检查器执行四项检查：

1. 无超时等待：每个 `wait` 必须有 `timeout` 或 `allow_indefinite_wait`。
2. 不可达声明验证：标记为 `unreachable` 的 `on_complete` 必须所有路径都跳走。
3. 零出度检测：非终态必须有出边。
4. **SCC** 分析：使用 Kosaraju 算法检测可能困住执行的环路。

5.3 时序检查器：关键路径分析

时序检查器沿拓扑连接链累加设备物理参数，计算最坏情况执行时间。

以冲压周期为例：

步骤	关键动作	最坏时间
feed	电机启动 + 等待到位	1500ms（超时上界）
stop_belt	电机停止	100ms
press_down	电磁阀响应 + 气缸行程	500ms（超时上界）
press_up	气缸回程	500ms（超时上界）
总计		**2600ms**

Table 2: 冲压周期最坏情况时序分析

2600ms $<$ 3000ms，时序约束满足。

5.4 因果检查器：信号可达性

因果检查器在拓扑图上执行 BFS，验证声明的因果链中每一跳都有对应的物理连接：

$$\text{Y1} \xrightarrow{\text{electrical}} \text{stamp\_valve} \xrightarrow{\text{pneumatic}} \text{stamp\_head} \xrightarrow{\text{logical}} \text{sensor\_stamp\_down}$$

如果任何一跳断开（例如 `connected_to` 配置错误），编译器会报告断裂位置并给出修复建议。

6 完整示例：传送带冲压系统

6.1 场景描述

一条自动化产线上，工件由传送带送入冲压工位。传感器检测到工件到位后，传送带停止，冲压头下压完成冲压，然后回位，传送带继续运行。

关键安全约束：冲压头下压时传送带绝对不能运行，否则工件移位会导致冲压偏移，损坏模具甚至造成安全事故。

6.2 系统拓扑

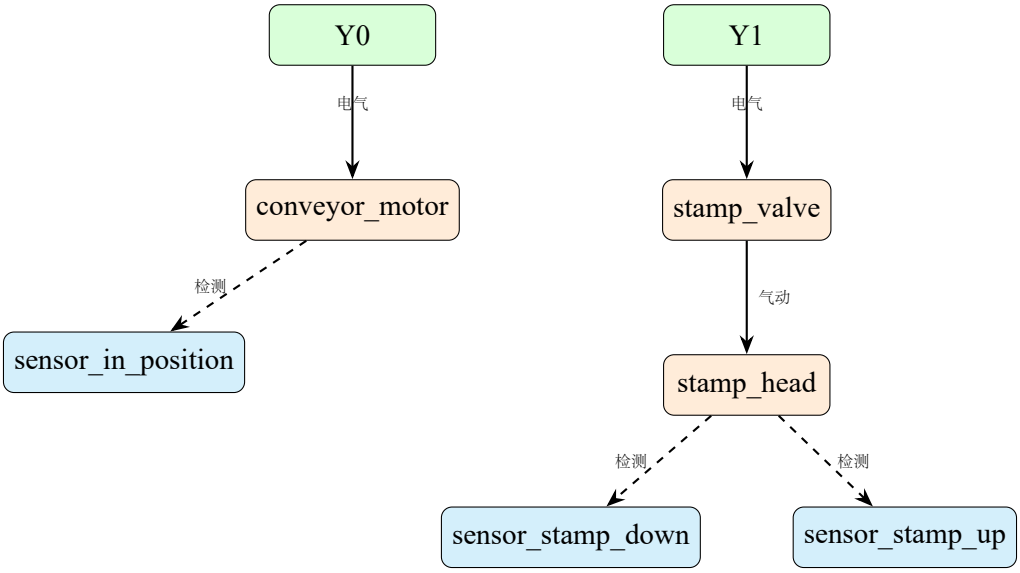

Figure 2: 传送带冲压系统拓扑图

6.3 状态机

6.4 运行验证

在项目根目录执行：

```
$ cargo run -- examples/conveyor_stamp.plc
```

输出：

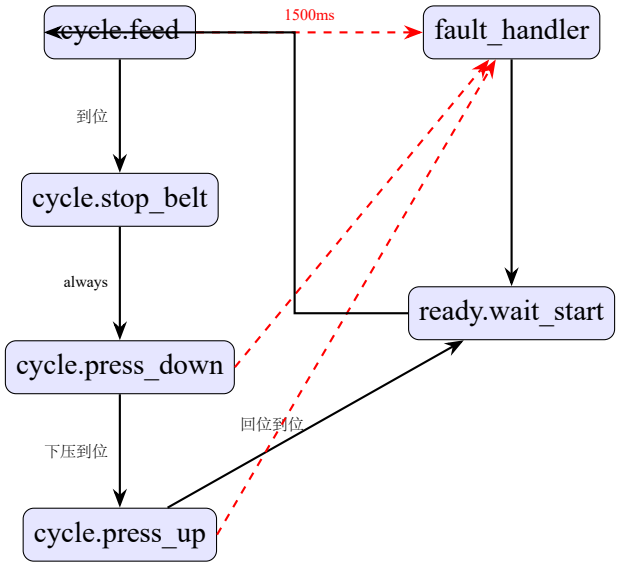

Figure 3: 传送带冲压系统状态机（红色虚线为超时跳转）

```
验证通过:
  - Safety: 完备证明（深度 8）
  - Liveness: 通过
  - Timing: 通过
  - Causality: 通过
```

四项验证全部通过，含义如下：

- **Safety** 完备证明：在深度 8 的状态空间搜索中，状态已收敛。数学上证明了在任意执行路径上，冲压头下压时传送带都不会运行。
- **Liveness** 通过：所有等待都有超时保护或明确的无限等待许可，不存在死锁。
- **Timing** 通过：最坏情况关键路径 2600ms < 约束上界 3000ms。
- **Causality** 通过：从 Y1 到 sensor_stamp_down 和 sensor_stamp_up 的信号传播路径在拓扑图中完整连通。

7 技术栈与依赖

依赖	版本	用途
`pest`	2.7	PEG 解析器生成器
`petgraph`	0.6	图数据结构与算法（SCC, BFS）
`serde`	1.0	JSON 序列化
`thiserror`	1.0	错误类型派生
`z3`（可选）	0.12	SMT 求解器增强安全证明

Table 3: RustPLC 依赖

Rust 2024 Edition，总代码量约 7,700 行，52 个测试用例。

8 与传统 PLC 编程的对比

特性	传统 **PLC**	**RustPLC**
安全验证	运行时测试	编译时形式化证明
互锁逻辑	手动编码	声明式约束
时序分析	示波器实测	编译器自动计算
因果追踪	人工审查	拓扑图自动验证
死锁检测	长时间运行观察	SCC 静态分析
错误定位	调试器	精确行号 + 修复建议

Table 4: 传统 PLC 编程 vs RustPLC

9 深度追踪：一条安全约束如何变成数学证明

前面我们从宏观上介绍了编译器架构和验证引擎。但“自动化验证”到底是怎么实现的？本节以传送带冲压系统中的一条安全约束为例，逐函数、逐数据结构地追踪它从 DSL 文本变成数学证明的完整路径。

我们追踪的目标是这一行：

```
safety: stamp_head.extended conflicts_with conveyor_motor.on
```

整个过程经历 5 个阶段，涉及 12 个关键函数和 8 个数据结构。

9.1 阶段一：文本 → **AST**（解析）

9.1.1 **PEG** 语法匹配

`pest` 解析器首先用 `plc.pest` 中的语法规则匹配这行文本。相关规则链为：

```
constraint_declaration = { safety_constraint | timing_constraint | ... }

safety_constraint = {
    "safety" ~ ":" ~ state_reference ~ safety_relation ~ state_reference
    ~ reason_clause?
}

state_reference = @{ identifier ~ "." ~ identifier }
safety_relation = { "conflicts_with" | "requires" }
```

匹配结果是一棵 pest 语法树，其中 `state_reference` 匹配到 `stamp_head.extended` 和 `conveyor_motor.on`，`safety_relation` 匹配到 `conflicts_with`。

9.1.2 AST 构建

`parse_constraints_section()` 遍历语法树，遇到 `safety_constraint` 规则时调用 `parse_safety_constraint()`（`src/parser/mod.rs:206`）：

```rust
fn parse_safety_constraint(pair: Pair<Rule>)
    -> Result<SafetyConstraint, PlcError>
{
    let line = line_of(&pair);
    let mut left = None;
    let mut relation = None;
    let mut right = None;
    let mut reason = None;

    for part in pair.into_inner() {
        match part.as_rule() {
            Rule::state_reference if left.is_none()
                => left = Some(parse_state_reference(part)?),
            Rule::safety_relation
                => relation = Some(parse_safety_relation(part)?),
            Rule::state_reference
                => right = Some(parse_state_reference(part)?),
            Rule::reason_clause
                => reason = Some(parse_reason_clause(part)?),
            _ => {}
        }
    }

    Ok(SafetyConstraint { line, left, relation, right, reason })
}
```

产出的 AST 节点为：

```
SafetyConstraint {
    line: 47,
    left:  StateReference { device: "stamp_head",     state: "extended" },
    relation: ConflictsWith,
    right: StateReference { device: "conveyor_motor", state: "on" },
    reason: Some(" 冲压头下压时传送带不能运行...")
}
```

9.2 阶段二：AST → IR（语义分析）

`build_constraint_set_from_ast()`（`src/semantic/mod.rs:104`）接收整个 AST，对每条 safety 约束执行两步操作：

9.2.1 步骤 2a：验证引用合法性

调用 `validate_state_reference()` 检查：

- `stamp_head` 是否在 `[topology]` 中声明过？——是，类型为 `cylinder`。

- `extended` 是否是 `cylinder` 的合法状态?——是，气缸的默认状态域为 `{extended, retracted}`。
- `conveyor_motor` 是否存在? ——是，类型为 `motor`。
- `on` 是否是 `motor` 的合法状态? ——是，电机的默认状态域为 `{on, off}`。

如果任何一项检查失败，编译器会报告精确行号和修复建议，编译终止。

9.2.2 步骤 2b：降低为 IR

验证通过后，AST 节点被转换为 IR 结构：

```
SafetyRule {
    left:     StateExpr { device: "stamp_head",     state: "extended" },
    relation: ConflictsWith,
    right:    StateExpr { device: "conveyor_motor", state: "on" },
    reason:   Some(" 冲压头下压时传送带不能运行...")
}
```

这个 `SafetyRule` 被存入 `ConstraintSet.safety` 向量，等待验证引擎消费。

9.3 阶段三：IR → SafetyModel（模型构建）

安全检查器的入口是 `verify_safety()`（`src/verification/safety.rs:124`），它首先调用 `SafetyModel::from_inputs()` 构建验证模型。这是最关键的一步。

9.3.1 步骤 3a：设备域枚举

`collect_device_domains()`（`safety.rs:358`）为每个设备建立有限状态域：

```
DeviceDomain { name: "stamp_head",     states: ["extended","retracted"],
               default_state: 1 }  // 初始为 retracted
DeviceDomain { name: "conveyor_motor", states: ["on","off"],
               default_state: 1 }  // 初始为 off
DeviceDomain { name: "stamp_valve",    states: ["on","off"],
               default_state: 1 }
...（共 13 个设备）
```

同时建立两级索引：

- `device_index`: `"stamp_head"` → `9`, `"conveyor_motor"` → `7`, `...`
- `device_state_index[9]`: `"extended"` → `0`, `"retracted"` → `1`

9.3.2 步骤 3b：转移效果提取

`transition_effects()`（`safety.rs:430`）将状态机中每条转移的 `action` 映射为设备状态变更。例如：

```
转移: cycle.feed -> cycle.stop_belt
  动作: set conveyor_motor on
  效果: { device_id=7 -> state_id=0 }  // conveyor_motor := on

转移: cycle.stop_belt -> cycle.press_down
  动作: set conveyor_motor off
  效果: { device_id=7 -> state_id=1 }  // conveyor_motor := off

转移: cycle.press_down -> cycle.press_up
  动作: extend stamp_head
  效果: { device_id=9 -> state_id=0 }  // stamp_head := extended
```

关键观察：`extend stamp_head` 发生在 `press_down` → `press_up`，而此时 `conveyor_motor` 已经在上一步被设为 `off`。这正是安全约束能通过的原因——但编译器不靠“看”，它靠穷举证明。

9.3.3 步骤 3c：搜索深度规划

`scc_minimum_depth()`（`safety.rs:510`）用 Kosaraju 算法找到状态机中最大的强连通分量（SCC），取 $|\text{SCC}|+1$ 作为搜索深度下界。最终深度取 max(状态数, SCC 深度)。

在我们的例子中：状态机有 7 个状态（cycle 的 4 步 + fault_handler 的 2 步 + ready），存在 `ready` → `cycle` → `ready` 的环路，SCC 大小为 7，所以搜索深度为 $\max(7, 7+1) = 8$。

9.4 阶段四：BFS 穷举搜索

9.4.1 步骤 4a：约束绑定

`bind_rule()`（`safety.rs:584`）将 IR 层的字符串约束绑定为数值索引：

```
RuleBinding {
    left_device:  9,   // stamp_head
    left_state:   0,   // extended
    right_device: 7,   // conveyor_motor
    right_state:  0,   // on
}
```

这样后续的冲突检测只需比较两个整数，无需字符串操作。

9.4.2 步骤 4b：BFS 展开

`analyze_rule()`（`safety.rs:609`）是核心搜索函数。它维护一个 BFS 队列和一个已访问状态集：

```
fn analyze_rule(model, rule, max_depth) -> SearchOutcome {
    let initial = initial_concrete_state(model);
    // initial = (control=0, devices=[1,1,1,...,1,1,...])
```

```
    //            所有设备处于默认状态(off/retracted)

    let mut queue = VecDeque::from([initial]);
    let mut visited = HashMap::new();

    while let Some(node) = queue.pop_front() {
        // 冲突检测：O(1) 整数比较
        if conflicts(&node.state, rule) {
            return counterexample(path);
        }

        if node.depth == max_depth { continue; }

        for edge in outgoing_edges(node) {
            let next = apply_edge(edge, &node.state);
            if !visited.contains(&next) {
                queue.push_back(next);
            }
        }
    }

    SearchOutcome { counterexample: None,
                    fully_explored: true }
}
```

9.4.3 步骤 4c：冲突检测

`conflicts()`（`safety.rs:703`）极其简洁：

```
fn conflicts(state: &ConcreteState, rule: RuleBinding) -> bool {
    state.device_states[rule.left_device] == rule.left_state
    && state.device_states[rule.right_device] == rule.right_state
}
```

即：当前状态中 `stamp_head` 是否为 `extended`（索引 0），同时 `conveyor_motor` 是否为 `on`（索引 0）？

9.4.4 搜索过程可视化

以下是 BFS 展开的前几步（简化表示，仅显示两个关键设备的状态）：

注意深度 3 的关键行：`stamp_head` 变为 `extended` 时，`conveyor_motor` 已经是 `off`——因为 `stop_belt` 步骤在 `press_down` 之前执行了 `set conveyor_motor off`。

BFS 在深度 8 穷尽了所有可达状态，没有发现任何一个状态同时满足 `stamp_head=extended` 且 `conveyor_motor=on`。

9.5 阶段五：证明判定与报告

9.5.1 收敛判定

回到 `analyze_rule()` 的返回值：

深度	控制状态	**stamp_head**	**conveyor_motor**	冲突**?**
0	cycle.feed	retracted	off	No
1	cycle.stop_belt	retracted	**on**	No
1	fault_handler.emergency	retracted	off	No
2	cycle.press_down	retracted	**off**	No
3	cycle.press_up	**extended**	**off**	No
3	fault_handler.emergency	retracted	off	No
4	ready.wait_start	retracted	off	No
⋮	⋮	⋮	⋮	⋮

Table 5: BFS 搜索过程（关键设备状态追踪）

```
SearchOutcome {
    counterexample: None,      // 未找到反例
    fully_explored: true,      // 深度 8 内无新状态
}
```

`fully_explored = true` 意味着 $\text{Reach}_8 = \text{Reach}_9$，即状态空间已收敛。这等价于 k-归纳的归纳步成立：

$$\text{Reach}_8 = \text{Reach}_\infty \implies \forall \sigma \in \text{Reach}_\infty : \neg(\text{stamp\_head.extended} \wedge \text{conveyor\_motor.on})$$

9.5.2 报告生成

`verify_safety_with_config()`（`safety.rs:132`）汇总所有约束的检查结果：

```
SafetyReport {
    level: Complete,           // 完备证明
    explored_depth: 8,
    warnings: [],              // 无警告
}
```

最终由 `verify_all()`（`verification/mod.rs:54`）将四个引擎的结果汇总为 `VerificationSummary`，输出到 stderr：

```
验证通过：
  - Safety: 完备证明（深度 8）
```

9.6 完整数据流总结

9.7 反例：如果约束被违反会怎样？

为了对比，我们看看如果把 `stop_belt` 步骤删掉（传送带不停就冲压），会发生什么。编译器会在 BFS 中发现一条违反路径：

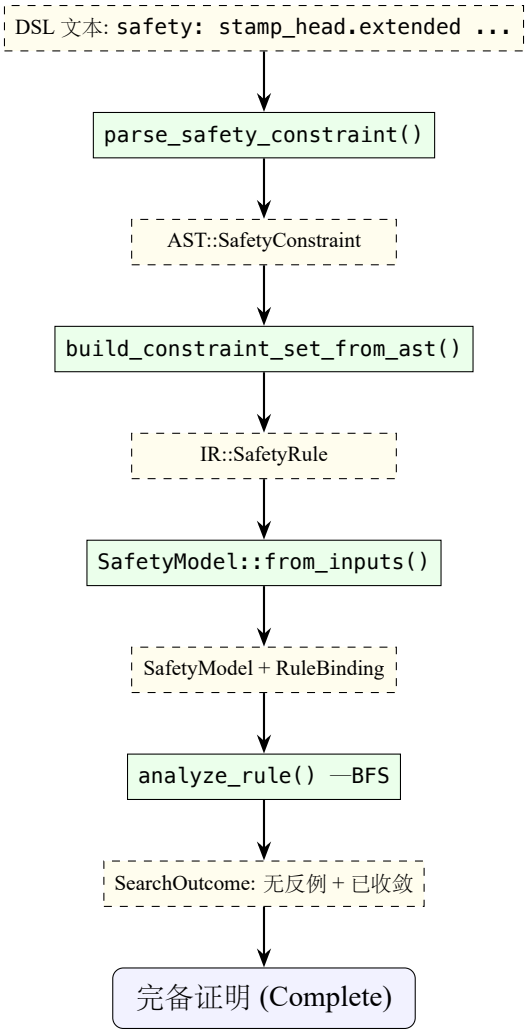

Figure 4: 安全约束从文本到证明的完整数据流

```
ERROR [safety] 状态互斥违反
  位置: <input>:47:1
  约束: stamp_head.extended conflicts_with conveyor_motor.on
  违反路径:
    1. 初始状态 cycle.feed
    2. cycle.feed --[condition; 动作: set conveyor_motor on]--> cycle.press_down
    3. cycle.press_down --[condition; 动作: extend stamp_head]--> cycle.press_up
    4. 在 cycle.press_up 检测到冲突: stamp_head.extended 与 conveyor_motor.on 同时为
  建议: 请在触发 conveyor_motor.on 之前确保 stamp_head.extended 已复位,
        或调整并行/跳转逻辑避免两者同时成立
```

render_path()（safety.rs:708）沿 BFS 的 parent 指针回溯，重建从初始状态到违反状态的完整路径，让工程师能精确定位问题。

10 验证实验：6 个例子逐一验证四大引擎

光讲原理不够，下面我们用 6 个可运行的 .plc 例子，逐一触发四个验证引擎的通过和失败，亲眼看编译器怎么抓 bug。

所有例子均在 examples/ 目录下，可直接运行：

```
$ cargo run -- examples/ex1_safety_pass.plc
```

10.1 例 1：安全性验证——顺序执行，通过

场景：双气缸交替冲压，声明 cyl_A.extended conflicts_with cyl_B.extended。控制逻辑是顺序执行：先伸 A、缩 A，再伸 B、缩 B，不会同时伸出。

```
[constraints]
safety: cyl_A.extended conflicts_with cyl_B.extended
    reason: "两个气缸同时伸出会发生机械碰撞"

[tasks]
task main:
    step extend_A:
        action: extend cyl_A
        wait: sensor_A == true
        timeout: 500ms -> goto fault
    step retract_A:
        action: retract cyl_A
    step extend_B:
        action: extend cyl_B
        wait: sensor_B == true
        timeout: 500ms -> goto fault
    step retract_B:
        action: retract cyl_B
    on_complete: goto main
```

运行结果：

```
验证通过:
  - Safety: 完备证明（深度 7）
  - Liveness: 通过
  - Timing: 通过
  - Causality: 通过
```

编译器在深度 7 穷尽了所有可达状态，数学上证明了两个气缸不可能同时伸出。

10.2 例 2：安全性验证——并行执行，失败

同样的设备和约束，但把控制逻辑改成 `parallel` 同时伸出两个气缸：

```
[tasks]
task main:
    step both:
        parallel:
            branch_A:
                action: extend cyl_A
            branch_B:
                action: extend cyl_B
```

运行结果：

```
ERROR [safety] 验证失败
  位置: <input>:23:1
  原因: 约束 cyl_A.extended conflicts_with cyl_B.extended
        在可达路径上可同时成立
  分析: 违反路径:
        初始状态 main.both
        -> main.both__parallel_1_fork
        -> main.both__parallel_1_branch_1
        -> main.both__parallel_1_join
        在 join 检测到冲突: cyl_A.extended 与 cyl_B.extended 同时为真
  建议: 请在触发 cyl_B.extended 之前确保 cyl_A.extended 已复位
```

编译器通过 BFS 找到了一条从初始状态到冲突状态的具体路径：`parallel` 的两个分支在 `join` 点汇合时，两个气缸都处于伸出状态。

对比：例 1 和例 2 的设备、约束完全相同，唯一区别是控制逻辑。顺序执行通过，并行执行被拦截——这就是形式化验证的价值。

10.3 例 3：活性验证——缺少超时，失败

场景：气缸伸出后等待传感器信号，但没有 **timeout**，也没有 `allow_indefinite_wait`。

```
[tasks]
task main:
    step go:
        action: extend cyl
        wait: sensor == true
        # 没有 timeout! 传感器坏了就永远卡死
```

```
    step back:
        action: retract cyl
    on_complete: goto main
```

运行结果：

```
ERROR [liveness] 验证失败
  位置: <input>:21:1
  原因: task main.go 的 wait 条件 `sensor == true` 缺少 timeout 分支，
        且未设置 allow_indefinite_wait
  分析: 若传感器信号长期不满足（线路故障/执行器卡滞/设备离线），
        控制逻辑会永久停留在该等待点
  建议: 请为该 step 添加 `timeout: < 时长> -> goto < 恢复 task>`

ERROR [liveness] 验证失败
  位置: <input>:21:1
  原因: 检测到强连通分量 [main.back, main.go] 不包含 timeout
        或 allow_indefinite_wait 出边
  分析: 一旦进入该循环，若条件长期不满足，流程会在环内反复执行
        且没有超时/人工等待豁免出口
```

活性检查器报了两个问题：

1. 单点问题：`wait` 没有 `timeout`，传感器坏了就卡死。
2. 结构问题：`main.go` → `main.back` → `main.go` 形成 SCC 环路，环内没有任何超时出口。

修复方法：加一行 `timeout: 500ms -> goto fault`。

10.4 例 4：时序验证——物理上不可能，失败

场景：要求 100ms 内完成，但光气缸行程就要 300ms。

```
[topology]
device valve: solenoid_valve { connected_to: Y0, response_time:
    20ms }
device cyl: cylinder { connected_to: valve, stroke_time: 300ms,
                       retract_time: 300ms }

[constraints]
timing: task.main must_complete_within 100ms
    reason: "要求 100ms 完成"

[tasks]
task main:
    step go:
        action: extend cyl
        wait: sensor == true
        timeout: 500ms -> goto err
    on_complete: goto main
```

运行结果：

```
ERROR [timing] 验证失败
  位置: <input>:17:1
  原因: task.main must_complete_within 100ms;
        最坏情况下无法在 100ms 内完成，当前关键路径为 500ms
  分析: task main 的最坏关键路径时间 = 500ms（顺序 step 累加）
  建议: 请放宽 must_complete_within 阈值，或缩短动作响应/行程时间
```

时序检查器沿拓扑链累加物理参数：电磁阀响应 20ms + 气缸行程 300ms = 320ms，加上 timeout 上界 500ms，远超 100ms 约束。这不是软件 bug，是物理定律不允许。

10.5 例 5：因果验证——接线错误，失败

场景：valve 实际连接的是 cyl_A，但因果链声明写成了 cyl_B。

```
[topology]
device valve: solenoid_valve { connected_to: Y0 }
device cyl_A: cylinder { connected_to: valve }
device cyl_B: cylinder { connected_to: valve }
device sensor: sensor { connected_to: X0, detects:
    cyl_A.extended }

[constraints]
# 错误：valve 连的是 cyl_A，但这里写了 cyl_B
causality: Y0 -> valve -> cyl_B -> sensor
```

运行结果：

```
ERROR [causality] 验证失败
  位置: <input>:18:1
  原因: 检测到因果链断裂: cyl_B -> sensor
  分析: 期望链路: Y0 -> valve -> cyl_B -> sensor
        实际链路: Y0 -> valve -> cyl_B -> ???
  建议: 请在 [topology] 中检查 sensor 的 connected_to / detects 配置，
        确保链路 cyl_B -> sensor 可达
```

因果检查器在拓扑图上做 BFS，发现 cyl_B 到 sensor 之间没有连接。sensor 的 detects 写的是 cyl_A.extended，不是 cyl_B。这种接线错误在传统 PLC 编程中只能靠现场调试发现，RustPLC 在编译时就拦截了。

10.6 例 6：四项全部通过——完整工业场景

场景：单气缸往复，安全 + 时序 + 因果 + 活性全覆盖。

```
[constraints]
safety: cyl.extended conflicts_with cyl.retracted
    reason: "气缸不可能同时处于伸出和缩回状态"
timing: task.work must_complete_within 2000ms
    reason: "单次往复不超过 2 秒"
causality: Y0 -> valve -> cyl -> sensor_ext
causality: Y0 -> valve -> cyl -> sensor_ret
```

```
[tasks]
task work:
    step push:
        action: extend cyl
        wait: sensor_ext == true
        timeout: 600ms -> goto fault
    step pull:
        action: retract cyl
        wait: sensor_ret == true
        timeout: 600ms -> goto fault
    on_complete: goto idle

task fault:
    step safe:
        action: retract cyl
    step alarm:
        action: log "气缸动作超时"
    on_complete: goto idle

task idle:
    step wait:
        wait: start_btn == true
        allow_indefinite_wait: true
    on_complete: goto work
```

运行结果：

```
验证通过：
  - Safety: 完备证明（深度 6）
  - Liveness: 通过
  - Timing: 通过
  - Causality: 通过
```

10.7 实验总结

例	场景	Safety	Liveness	Timing	Causality
1	双缸顺序执行	✓	✓	✓	✓
2	双缸并行执行	FAIL	FAIL	–	–
3	等待无超时	–	FAIL	–	–
4	时序过紧	–	–	FAIL	–
5	因果链断裂	–	–	–	FAIL
6	单缸完整场景	✓	✓	✓	✓

Table 6: 6 个例子的验证结果汇总

每个验证引擎都被至少一个正例和一个反例覆盖。正例证明编译器不会误报，反例证明编译器能精确定位问题并给出修复建议。

11 总结

RustPLC 将工业控制编程从 “编写程序然后测试” 提升到 “声明意图然后证明”。它的五个第一性原理——有限状态、安全不变量、因果律、活性、物理时序——构成了一个完整的形式化验证框架。

通过 6 个可运行的例子，我们验证了：

- 安全检查器能区分顺序执行（安全）和并行执行（冲突），给出完备证明或违反路径；
- 活性检查器能发现缺少超时的等待点和 SCC 死循环；
- 时序检查器能根据物理参数计算最坏关键路径，拦截不可能满足的时序约束；
- 因果检查器能在拓扑图上发现接线错误，精确报告断裂位置。

这正是 RustPLC 的价值所在：让安全成为编译器的责任，而非程序员的负担。