这个任务分解为两个主要部分："**前端**"（解析 Verilog 并将其转换为数学模型）和 "**后端**"（使用 SMT 求解器进行验证）。

核心实现思路：有界模型检查（BMC）

CoSA 是一个符号模型检查器。最简单的入门方法是实现**有界模型检查（Bounded Model Checking，BMC）**。CoSA 手册中也提到了 BMC（例如 BMC-FWD）。

BMC 的思想是：

1. 将 Verilog 电路看作一个**符号转换系统（STS）**，它有一个**初始状态** I 和一个**转换关系** T 。
2. 将这个系统 "展开"（unroll）k 个步骤（或时钟周期）。
3. 询问 SMT 求解器："在这个 k 步的执行中，有没有可能违反我的属性？"

实施步骤

1. 前端：Verilog 到数学模型

首先是将 Verilog 源代码转换成 SMT 求解器可以理解的数学公式。

- **挑战**：Verilog 是一种复杂的硬件描述语言。自己编写一个完整的解析器非常困难。
- **实现**：
  1. 使用 **Yosys**（一个开源的 Verilog 综合工具）来解析和综合 Verilog 代码。
  2. 让 Yosys 将综合后的电路导出为一种更简单的、基于 SMT 的格式。**BTOR2** 格式是一个理想的中间格式，因为它非常简洁，并且直接映射到 SMT。

2. 后端：使用 SMT 求解器进行验证

现在有了一个 `.btor2` 文件，它在数学上定义了电路的初始状态 I 和转换关系 T 。使用一个 SMT 库来读取这个模型并执行检查。

- 使用**PySMT** 来管理公式。这是一个与求解器无关的 Python 库。
- **实现（BMC）**：
  1. **定义属性**：从最简单的属性开始：**不变量（Invariant）**。这是一个在任何时候都必须为 True 的公式，例如 `(out < 10_16)`。
  2. **设置求解器**：使用 **PySMT** 作为 SMT 接口。
  3. **展开模型（BMC）**：
     - **第 0 步（初始状态）**：从 `.btor2` 文件中获取初始状态公式 $I(V_0)$ 并将其断言（assert）到 PySMT。
     - **检查属性**：断言不变量是 *错误* 的，即 `NOT(property(V_0))`。
     - **求解**：调用 `solver.check_sat()`。
     - **第 1 到 k 步**：在一个循环中，引入新变量 $V_1, V_2, \ldots, V_k$。
     - **断言转换**：断言转换关系 $T(V_{i-1}, V_i)$。
     - **检查属性**：断言 `NOT(property(V_i))`。
     - **求解**：再次调用 `solver.check_sat()`。
  4. **分析结果**：
     - 如果求解器在任何步骤 j 返回 **SAT（可满足）**：那么就找到了一个反例。求解器会出一个模型（一组赋值），这就是导致属性失败的执行迹线。

    - 如果求解器在所有 k 步都返回 **UNSAT（不可满足）**：这意味着在 k 个周期内没有发现错误。这对应 "UNKNOWN" 结果（有界证明）。

技术栈

- **核心语言：Python**
- **Verilog 解析器：Yosys**
- **中间格式：BTOR2**
- **SMT 接口：PySMT**
- **SMT 求解器：cvc5**

总结工作流程

1. **输入：**用户提供 `design.v` 和一个用 Python 字符串表示的**不变量属性**（例如 `"out < 10_16"`）。
2. **步骤 1：**工具在后台调用 `yosys`，将 `design.v` 转换为 `design.btor2` 。
3. **步骤 2：**使用 **PySMT** 和一个 BTOR2 解析库来读取 `design.btor2`，从而获得初始状态 I 和转换关系 T 。
4. **步骤 3：**解析不变量属性字符串。
5. **步骤 4：**执行 BMC 循环 k 次，每次都向 PySMT 断言转换关系，并询问 `NOT(property)` 是否可能为真。
6. **输出：**报告 "Property holds up to k cycles"（UNSAT）或 "Property VIOLATED at cycle j"（SAT），并打印反例。