### **Project Assignment: Formal Methods in Software Engineering**

### **Title: Traffic Light Controller**

### **Student Information**

● Full Name: Laiba Yousaf

● Registration No: 2021-GU-821

● Roll No: 19  
Email: laibayousaf997@gmail.com

Ghazi University D.G.Khan

**Overview of the Problem**I am tasked with modeling a traffic light controller system that operates in a predictable and consistent manner. This requires understanding the functional requirements of a traffic light, identifying constraints, and formalizing them into a specification.   
  
**1. Traffic Light States**

A traffic light can be in one of three states:

Red: Indicates that vehicles must stop.

Green: Indicates that vehicles can proceed.

Yellow: Warns vehicles to slow down and prepare to stop.  
  
**Transition Rules**

The traffic light transitions between states in a fixed, cyclic sequence:

Red → Green: From stop to go.

Green → Yellow: From go to prepare to stop.

Yellow → Red: From prepare to stop back to stop.

This cyclic nature ensures smooth traffic flow and prevents conflicts. No deviations from this order are allowed.

**Constraints**

The following constraints must be satisfied:

* **Mutual Exclusivity:**

Only one light can be active at any given time.

* **Safety:**

There should never be a situation where two or more lights are active simultaneously. This ensures no conflicting signals are given to drivers.

* **Predictable Behavior:**

The system must always follow the defined sequence: Red → Green → Yellow → Red.

No invalid states should occur (e.g., jumping directly from Red → Yellow or activating multiple lights).

**Formal Representation**I aim to use a formal specification language like Alloy to describe the system. The representation includes:

1. **States:**
   * Represent the traffic light's states (Red, Green, Yellow) as distinct entities.
2. **Transitions:**
   * Define the valid transitions between these states.
3. **Constraints:**
   * Enforce rules like mutual exclusivity and sequence order.

#### **Example in Simple Terms:**

| **Current State** | **Next State** | **Allowed?** |
| --- | --- | --- |
| **Red** | **Green** | **✅ Yes** |
| **Green** | **Yellow** | **✅ Yes** |
| **Yellow** | **Red** | **✅ Yes** |
| **Red** | **Yellow** | **❌ No** |
| **Green** | **Red** | **❌ No** |

### **System Inputs and Outputs**

1. **Inputs**:
   * A timer or sensor triggers transitions after a fixed duration (e.g., 30 seconds for Red, 60 seconds for Green).
   * No manual intervention is required.
2. **Outputs**:

Only one light is illuminated at a time.  
  
Ensures clarity and prevents ambiguity for drivers.

**Formal Specification Using Alloy   
  
Setting Up Alloy**

**Download Alloy Analyzer:**

* + Download and install Alloy Analyzer for your operating system.

**Open Alloy Analyzer:**

* + Launch the application and create a new file for your model.

### **Defining the States**

In Alloy, signatures (sig) are used to define the main entities in the system. For the traffic light, define three states:

| sig Light {  red: lone Bool, // Light can either be Red (True) or not (False)  green: lone Bool, // Light can either be Green (True) or not (False)  yellow: lone Bool // Light can either be Yellow (True) or not (False)  } |
| --- |

**Adding Constraints**

To ensure the system behaves correctly, we’ll add constraints using facts.

Constraint 1: Only One Light is Active at a Time

| fact onlyOneActive {  all l: Light | lone l.red + l.green + l.yellow  } |
| --- |

**Constraint 2: Valid Transitions**

| pred validTransitions(current: Light, next: Light) { (current.red and next.green) or // Red → Green (current.green and next.yellow) or // Green → Yellow (current.yellow and next.red) // Yellow → Red } |
| --- |

**Constraint 3: No Invalid States**

| fact noInvalidStates {  all l: Light | not (l.red and l.green)  and not (l.red and l.yellow)  and not (l.green and l.yellow)  } |
| --- |

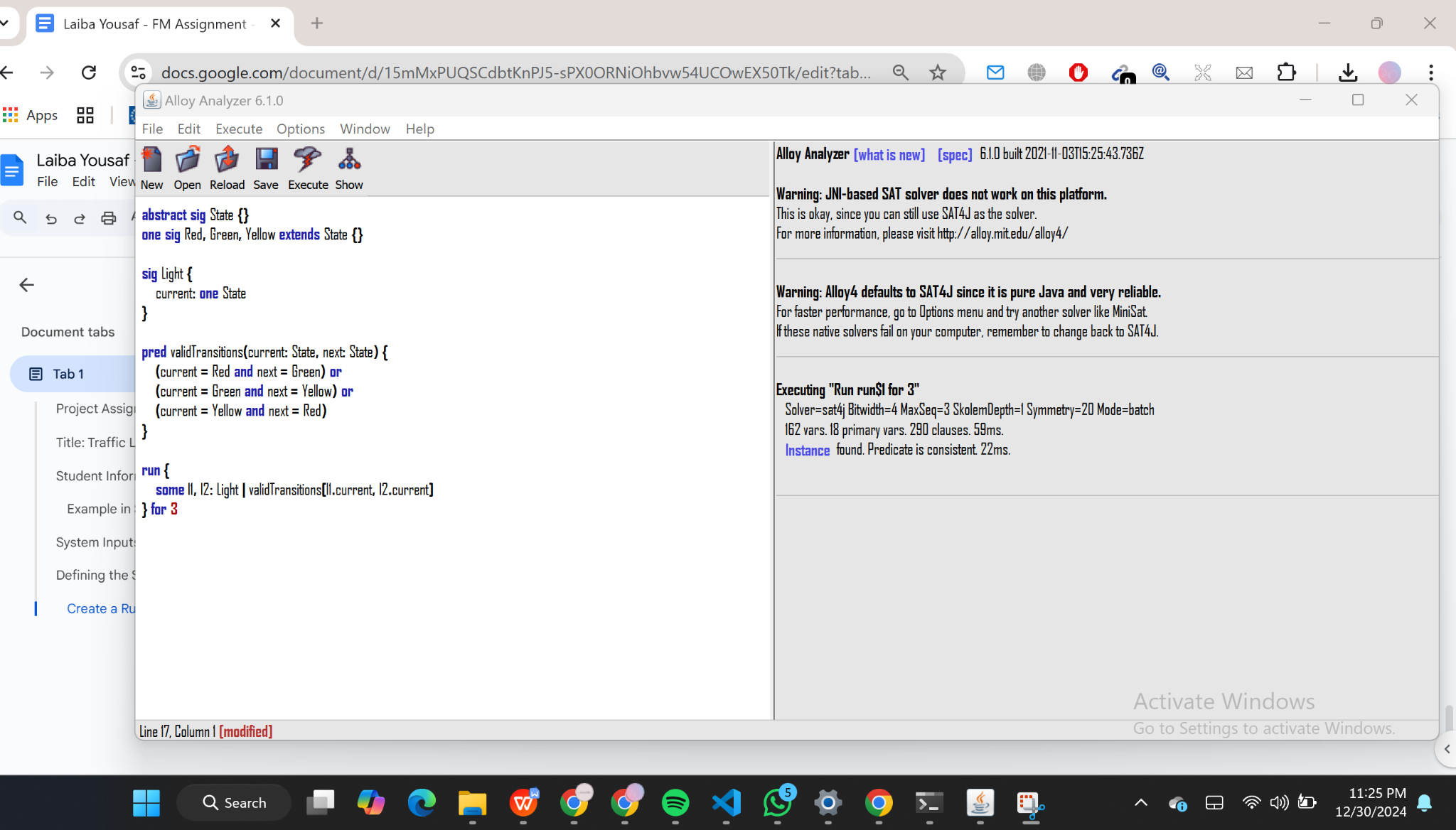
**Simulating the Model**

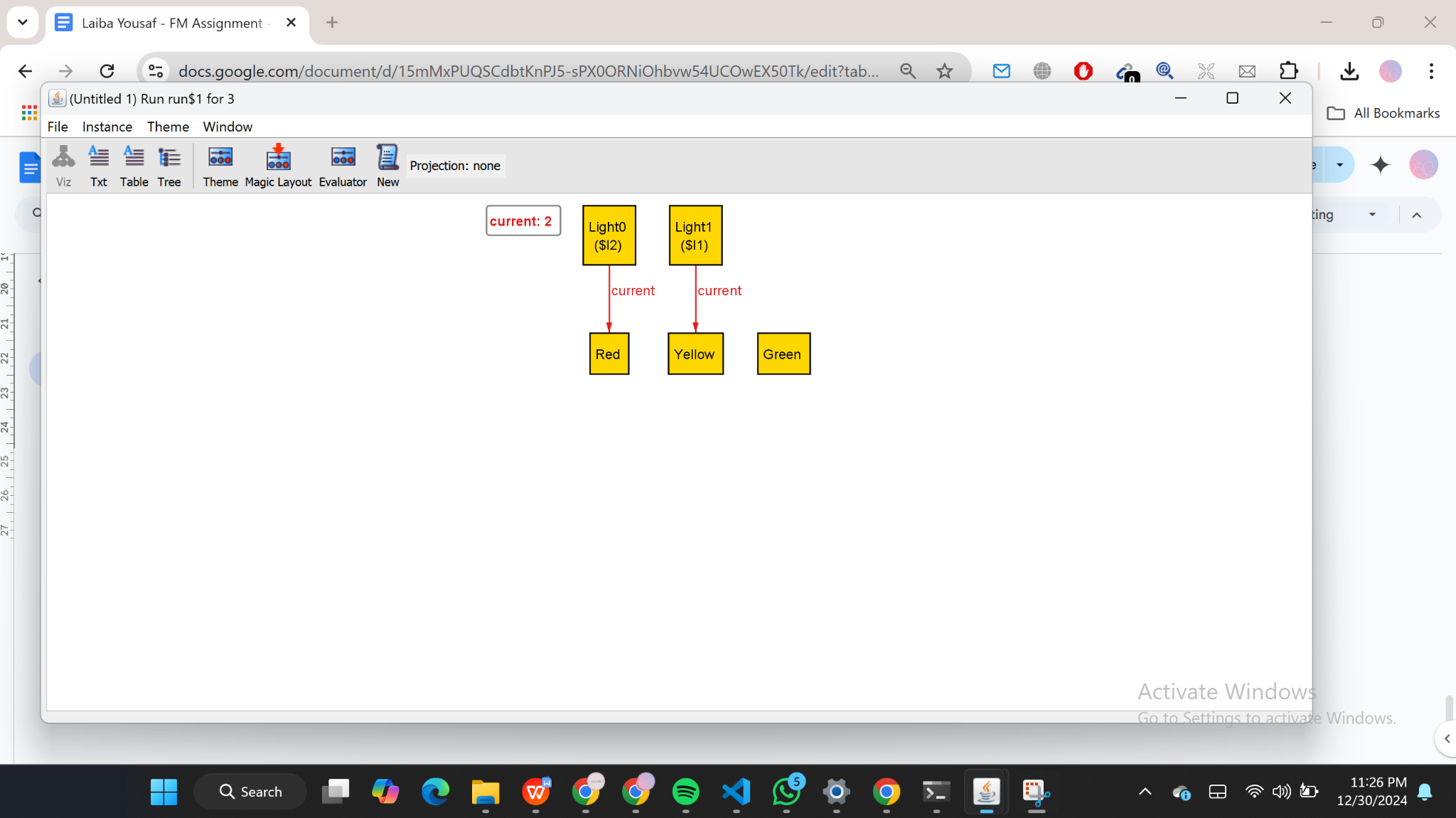
#### Create a Run Command

To test and visualize the model, write a run statement:

| run {  some l1, l2: Light | validTransitions[l1, l2]  } for 3 |
| --- |

.

**Interpreting the Output:**

After running the model, Alloy will show a graph:  
  


**Model Implementation**

#### **Step 1: Setting up the Traffic Light System in Python**

1. Represent States as Constants or Enums:
   * You can represent the traffic light states using strings or enums. In this example, we'll use strings for simplicity, but you could use Python’s Enum class for better structure.
2. Implement Transitions:
   * Create a class TrafficLight that tracks the current state and has a method transition to change the state based on the current one.
3. Enforce Constraints:
   * Ensure the system always follows the sequence: Red → Green → Yellow → Red.
   * Implement the logic so that no two lights are active simultaneously.

**Writing the Python Code**

| | **# Python implementation of the Traffic Light system**  class TrafficLight:  def \_\_init\_\_(self):  self.state = "Red" # Initial state is Red  def transition(self):  """Handle state transitions based on the current state"""  if self.state == "Red":  self.state = "Green"  elif self.state == "Green":  self.state = "Yellow"  elif self.state == "Yellow":  self.state = "Red"  def current\_state(self):  """Return the current state of the traffic light"""  return self.state  # Test the traffic light transitions  if \_\_name\_\_ == "\_\_main\_\_":  traffic\_light = TrafficLight()    # Simulate traffic light transitions  for \_ in range(6): # Cycle through the transitions multiple times  print(f"Current state: {traffic\_light.current\_state()}")  traffic\_light.transition() | | --- | |
| --- | --- |

**Verifying State Transitions with KLEE  
  
Use KLEE for Symbolic Execution**

Now that KLEE is installed, you can use it to analyze the C code for the traffic light system.

1. Write your C code (like the one from earlier) and save it as traffic\_light.c.

Compile the C code to LLVM bitcode: In the terminal, navigate to the folder containing your traffic\_light.c file and run:  
bash  
Copy code  
clang -emit-llvm -c traffic\_light.c -o traffic\_light.bc

Run KLEE on the LLVM bitcode: Now that you have the bitcode file (traffic\_light.bc), you can use KLEE to symbolically execute the code:  
bash  
Copy code  
klee traffic\_light.bc

1. View the output: KLEE will explore all possible execution paths. You can inspect the output in the terminal to see the results of symbolic execution, such as any paths where the constraints may be violated.

#### Step 4: Debugging with KLEE

After running KLEE, you may want to explore the results in more detail:

* KLEE generates logs and explores paths. Look for output in the terminal or the klee-out directory where KLEE stores its execution results.

**Testing the Python Implementation  
  
Testing:**

The goal of this phase is to validate the transitions in the traffic light controller system using Python’s unittest module. This ensures that all transitions follow the specified rules and constraints (e.g., Red → Green → Yellow → Red).

**Steps for Testing:**

Set Up the Testing Environment:

Make sure the TrafficLight class implementation is in a Python file (e.g., traffic\_light.py).

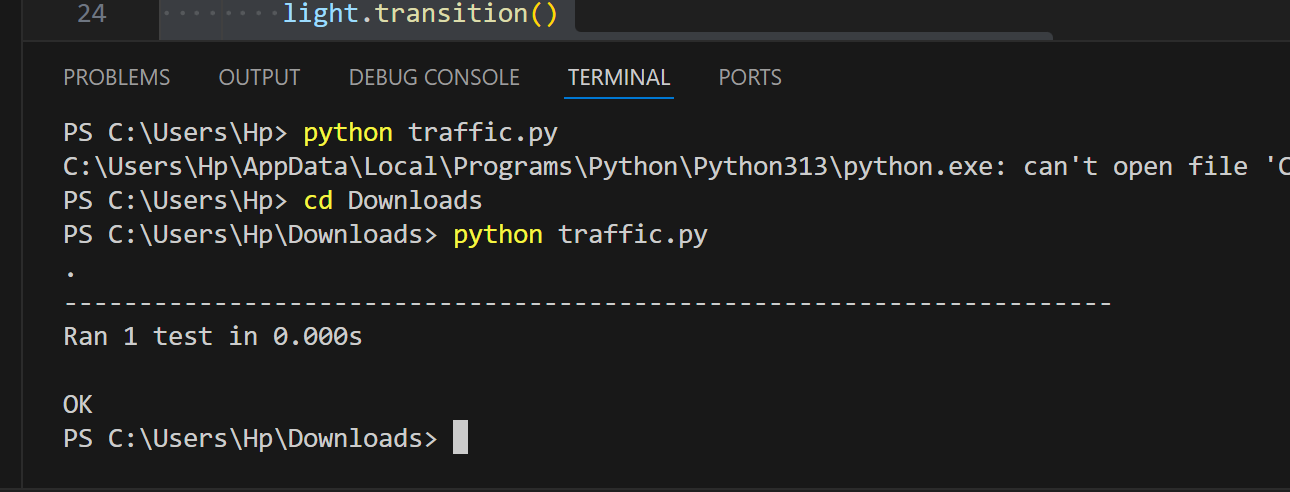
Import the unittest module for creating and running test cases.

**Create Test Cases:**

Define a test suite to verify the behavior of the traffic light transitions.

Each test will check the initial state and the sequence of transitions to ensure correctness.

| import unittest  class TrafficLight:  def \_\_init\_\_(self):  self.state = "Red"  def transition(self):  if self.state == "Red":  self.state = "Green"  elif self.state == "Green":  self.state = "Yellow"  elif self.state == "Yellow":  self.state = "Red"  def current\_state(self):  return self.state  class TestTrafficLight(unittest.TestCase):  def test\_transitions(self):  light = TrafficLight()  self.assertEqual(light.current\_state(), "Red")  light.transition()  self.assertEqual(light.current\_state(), "Green")  light.transition()  self.assertEqual(light.current\_state(), "Yellow")  light.transition()  self.assertEqual(light.current\_state(), "Red")  if \_\_name\_\_ == "\_\_main\_\_":  unittest.main() |
| --- |

**Screenshot of Output  
  
  
  
This indicates that all test cases passed successfully, confirming that the implementation meets the required constraints.**