Task 1: Create algebraic specification of a control system for autonomous cars

```
fmod CORE-TYPES is
  protecting STRING .

  sorts SensorData Object Action State Component PriorityQueue .

  subsort PriorityQueue < Action .

  ops GPS Camera Radar Lidar TrafficSignal SpeedLimit : -> SensorData .
  ops Lane Obstacle Vehicle Pedestrian GreenLight RedLight StopSign Collision :
-> Object .
  ops KeepSpeed Accelerate Decelerate Stop TurnLeft TurnRight AdjustSpeed
Override : -> Action .
  ops EmergencyBrake HandleCollision CombinedAction : -> Action .
  ops Active Manual Emergency : -> State .
  ops engine brake wheels lights : -> Component .
endfm

fmod PROCESSING is
  protecting CORE-TYPES .

  op process : SensorData -> Object .
  op prioritize : Object Object -> Object .

  vars O1 O2 : Object .

  eq process(GPS) = Lane .
  eq process(Camera) = Pedestrian .
  eq process(Radar) = Vehicle .
  eq process(Lidar) = Obstacle .
  eq process(TrafficSignal) = RedLight .
  eq process(SpeedLimit) = Lane .

  eq prioritize(O1, O2) =
    if O1 == Pedestrian or O1 == Obstacle
    then O1
    else O2
    fi .
endfm

fmod DECISION is
  protecting CORE-TYPES .

  op decide : Object State -> String .

  vars O1 : Object .
  vars S : State .

  eq decide(O1, Active) =
    if O1 == Pedestrian
```

```
    then "Car is stopping due to pedestrian detection."
    else if O1 == Obstacle
        then "Emergency brake activated due to obstacle."
        else if O1 == RedLight
              then "Car is stopping at the red light."
              else "Car is maintaining speed."
        fi
    fi
  fi .

  eq decide(O1, Emergency) =
    if O1 == Collision
    then "Car is handling a collision."
    else "Emergency brake activated."
    fi .

  eq decide(O1, Manual) = "Driver override: manual control enabled." .
endfm

fmod EXECUTION is
  protecting CORE-TYPES .

  op execute : PriorityQueue Component -> String .
  op reportState : String -> String .

  vars A : Action .
  vars C : Component .
  var Msg : String .

  eq execute(A, C) =
    if A == Stop and C == brake
    then "Car is stopping using brakes."
    else if A == EmergencyBrake and C == brake
         then "Emergency brake applied."
         else if A == KeepSpeed and C == engine
              then "Maintaining current speed."
              else if A == TurnLeft and C == wheels
                   then "Turning left using wheels."
                   else if A == HandleCollision and C == lights
                        then "Activating hazard lights for collision."
                        else if A == Stop and C == engine
                          then "Car has shut off."
                            else reportState("Invalid action-component
combination.")
                            fi
                        fi
                   fi
              fi
         fi
    fi .

  eq reportState(Msg) = "Error: " + Msg .
endfm
```

```
fmod COMBINE is
  protecting CORE-TYPES .

  op combine : PriorityQueue PriorityQueue -> PriorityQueue .

  vars A B : Action .

  eq combine(A, B) =
    if A == Stop or B == Stop
    then Stop
    else if A == EmergencyBrake or B == EmergencyBrake
         then EmergencyBrake
         else A
         fi
    fi .
endfm

fmod TEST is
  protecting PROCESSING .
  protecting DECISION .
  protecting EXECUTION .
  protecting COMBINE .

  op testObject : -> Object .
  op testState : -> State .

  eq testObject = RedLight .
  eq testState = Active .

  op testProcess : -> Object .
  eq testProcess = process(GPS) .

  op testDecide : -> String .
  eq testDecide = decide(testObject, testState) .

  op testDecideObstacle : -> String .
  eq testDecideObstacle = decide(Obstacle, Active) .

  op testExecuteBrake : -> String .
  eq testExecuteBrake = execute(Stop, brake) .

  op testCombine : -> PriorityQueue .
  eq testCombine = combine(Stop, Accelerate) .
endfm
```

Task 2: Document the solution

Key Decisions

1. Modular Design:

- The system is divided into modules (`CORE-TYPES`, `PROCESSING`, `DECISION`, `EXECUTION`, `COMBINE`, and `TEST`) to ensure maintainability, scalability, and clarity.

2. State-Aware Decisions:

- The `decide` operation tailors responses based on the car's state (`Active`, `Emergency`, `Manual`) for appropriate action in various scenarios.

3. Action Prioritization:

- The `combine` operation ensures safety-critical actions like `Stop` or `EmergencyBrake` override others like `Accelerate`.

4. Traceable Execution:

- The `execute` operation provides descriptive outputs (Example: `"Car is stopping using brakes."`) to enhance traceability and debugging.

5. Error Handling:

- Invalid action-component combinations are reported explicitly using `reportState`.

High-Level Design

1. CORE-TYPES:

- Defines foundational data types (`SensorData`, `Object`, `Action`, `State`, `Component`) used across the system.

2. PROCESSING:

- `process`: Translates sensor data (e.g., `GPS`, `Camera`) into actionable objects like `Lane` or `Pedestrian`.

3. DECISION:

- `decide`: Determines the car's action based on detected objects and the state, returning descriptive strings.

4. EXECUTION:

- `execute`: Links actions to components and outputs feedback.

5. COMBINE:

- `combine`: Merges multiple actions, ensuring safety-critical ones take precedence.

6. TEST:

- Provides test cases to validate the system's functionality, including:

- `process`: Ensures correct object translation.
- `decide`: Verifies decision-making for specific states.
- `execute`: Tests proper action execution.

Task 3: Prepare some test cases (scenarios, inputs)

1. `testObject` and `testState`

Represents the initial inputs to the system.
Description

- `testObject` is set to `RedLight`, simulating a traffic signal detection.
- `testState` is set to `Active`, representing the car's current state.

```
Maude> red testObject .
reduce in TEST : testObject .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Object: RedLight

Maude> red testState .
reduce in TEST : testState .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result State: Active
```

- `testObject` provides the object being processed (`RedLight`).
- `testState` provides the car's state (`Active`).

2. `testProcess`

Validates the `process` operation for sensor data.
- Processes `GPS` data and translates it into an object.

```
Maude> red testProcess .
reduce in TEST : testProcess .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Object: Lane
```

- `GPS` data is interpreted as detecting a `Lane`.

3. `testDecide`

Validates the `decide` operation for decision-making based on objects and state.

- Determines the car's action based on `testObject` (`RedLight`) and `testState` (`Active`).
- The car's action based on `object` (`Pedestrian`) and `state` (`Active`).

```
Maude> red testDecide .
reduce in TEST : testDecide .
```

```
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "Car is stopping at the red light."

Maude> red decide(Pedestrian, Active) .
reduce in TEST : decide(Pedestrian, Active) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "Car is stopping due to pedestrian detection."
```

- The car stops upon detecting the `RedLight` in the `Active` state.
- The car stops when detecting the `Pedestrian` in the `Active` state.

4. `testDecideObstacle`
Tests the car's decision when encountering an obstacle

- Evaluates the decision-making logic when the `object` is `Obstacle` and the `state` is `Active`.

```
Maude> red testDecideObstacle .
reduce in TEST : testDecideObstacle .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "Emergency brake activated due to obstacle."
```

- The car applied Emergency brake to avoid `Obstacle`.

5.`testExecuteBrake`
Verifies the `execute` operation for stopping the car.

- Executes the Stop action using the brake component.

```
Maude> red testExecuteBrake .
reduce in TEST : testExecuteBrake .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "Car is stopping using brakes."
```

- The car stops using its brake.

6. `testCombine`
Validates the combination of two actions using `combine`.

- Combine the action `Stop` and `Accelerate`.

```
Maude> red testCombine .
reduce in TEST : testCombine .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
result Action: Stop
```

- The `Stop` action takes priority over `Accelerate`.

7. Test Case Summary

Test Case	Operation	Inputs	Expected Output
`testObject`	N/A	N/A	`RedLight`
`testState`	N/A	N/A	`Active`
`testProcess`	`process`	`GPS`	`Lane`
`testDecide`	`decide`	`RedLight`, `Active`	`"Car is stopping at the red light."`
`decide`	`decide`	`Pedesritan`, `Active`	`"Car is stopping due to pedestrian detection."`
`testDecideObstacle`	`decide`	`Obstacle`, `Active`	`"Emergency brake activated due to obstacle."`
`testExecuteBrake`	`execute`	`Stop`, `brake`	`"Car is stopping using brakes."`
`testCombine`	`combine`	`Stop`, `Accelerate`	`Stop`

Task 4: Evaluation of the Autonomous Car System

1. Level of Abstraction

- **Details Ignored:** Internal sensor operations, environmental factors (e.g., weather, road conditions), and physical mechanics (e.g., brake force).
- **Strength:** Focuses on decision-making and execution without unnecessary complexity.
- **Limitation:** Cannot simulate real-world imperfections or failures.

2. Level of Approximation

- **Over-Specification:** Ensures safety with strict prioritization of critical actions (e.g., Stop, EmergencyBrake).
- **Under-Specification:** Assumes perfect sensor data, lacks handling of conflicting or missing inputs.
- **Balance:** Skews towards safety by being more restrictive.

3. Ambiguity vs. Precision

- **Precision:** Operations (process, decide, execute) and priorities are explicitly defined, ensuring clarity.
- **Ambiguity:** Undefined behaviors for sensor failures or conflicting inputs.
- **Assessment:** High precision within scope but ambiguous for unhandled edge cases.

4. Completeness

- **Coverage:** Covers core scenarios (e.g., stopping for pedestrians, responding to obstacles).
- **Missing:** Handling sensor failures, environmental factors, and equally critical actions.
- **Assessment:** Functionally complete for its scope but not exhaustive for real-world complexities.

Task 5: Opinion about the modeling language, tool, and the whole approach (methodology) based on your personal experience

1. Modeling Language (Maude)

- **Strengths:** Highly expressive, modular, and supports executable specifications for testing.
- **Limitations:** Steep learning curve and cryptic error messages make debugging challenging.

2. Tool (Maude Interpreter)

- **Strengths:** Efficient for prototyping and rule-based rewriting.
- **Limitations:** Lacks IDE support (e.g., syntax highlighting, visualization) and can be hard to debug large models.

3. Methodology

- **Strengths:** Ensures precision, modularity, and rigorous validation for safety-critical systems.
- **Limitations:** Models can become complex, and adoption is limited by the expertise required.

4. Practical Use

- **Feasibility:** Practical for safety-critical systems with proper IDE support; otherwise, limited to niche or academic projects.
- **Scenarios:** Validating complex designs, ensuring requirement consistency, and precise system documentation.