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
```

Solution Documentation

Test cases:

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.

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

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)`.

1. `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
```

```
red testDecide .
```

```
red testDecideObstacle .
```

```
red testExecuteBrake .
```

```
red testCombine .
```

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`

Test Case	Operation	Inputs	Expected Output
testDecide	decide	RedLight, Active	"Car is stopping at the red light."
testDecideObstacle	decide	Obstacle, Active	"Emergency brake activated due to obstacle."
testExecuteBrake	execute	Stop, brake	"Car is stopping using brakes."
testCombine	combine	Stop, Accelerate	Stop