Automotive System

Formal Methods for Critical Systems

Alexandre Abreu	up201800168
Breno Accioly	up201800170

Adaptive Light System

- Our project specification defined two systems, for this project, we decided to model the Adaptive Exterior Light System
- It is responsible for the activation and deactivation of the exterior lights
- The system is composed of:
  - **User interface:** light rotary switch, pitman arm, hazard warning and darkness switch
  - **Sensors:** key position, engine status, brightness sensor, brake deflection, battery voltage, steering angle, doors status, oncoming traffic, camera state, speed, reverse gear
  - **Actuators:** direction indicators, low and high beam headlights, cornering lights, brake lights

Structure Model

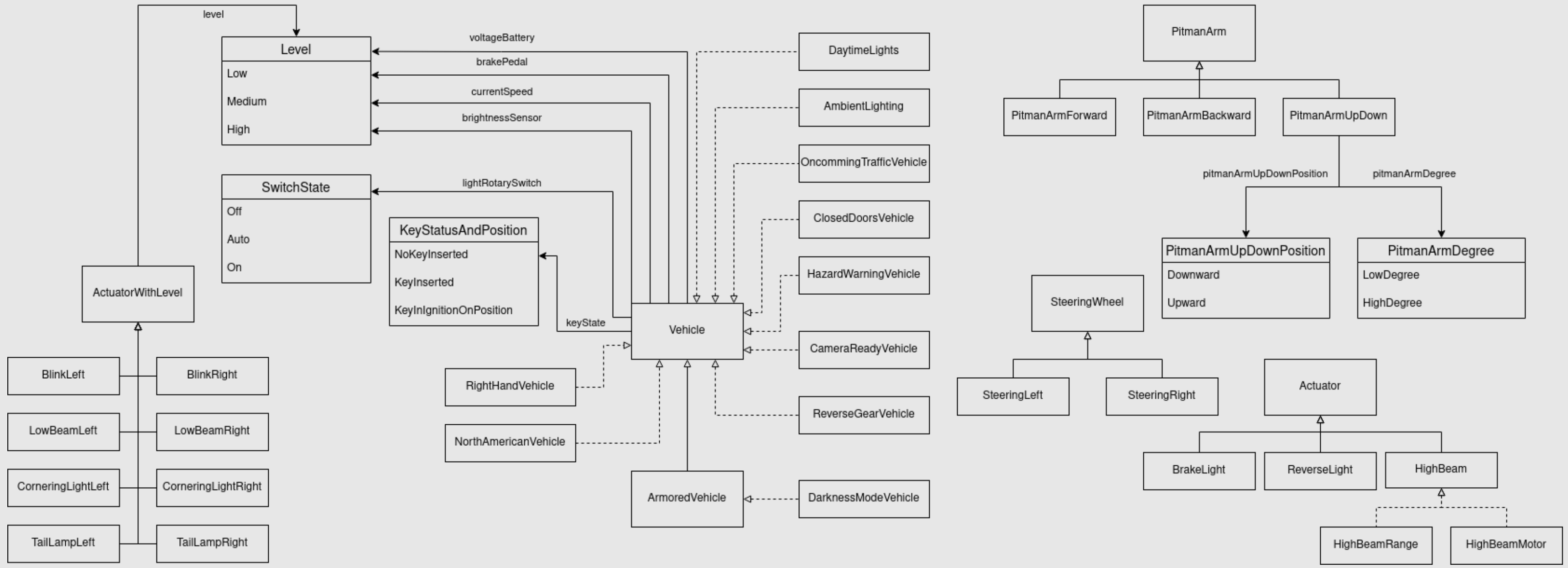
- Our system is modelled around only one vehicle
- Boolean attributes are defined by using subsets in Alloy
- Use of enumerations for some attributes of our system

```
abstract sig SwitchState {}
one sig Off, Auto, On
extends SwitchState {}

abstract sig Level {}
one sig Low, Medium, High
extends Level {}
```

```
one sig Vehicle {
  , var lightRotarySwitch: one SwitchState
  , var keyState: one KeyStatusAndPosition
  , var brightnessSensor: one Level
  , var brakePedal: one Level
  , var voltageBattery: one Level
  , var currentSpeed: one Level
}

lone sig DaytimeLights
       , AmbientLighting
       , RightHandVehicle
       , NorthAmericanVehicle
      in Vehicle {}
```

```
lone var abstract sig PitmanArm {}

lone var sig PitmanArmUpDown extends PitmanArm {
  , var pitmanArmUpDownPosition: one
PitmanArmUpDownPosition
  , var pitmanArmDegree: one PitmanArmDegree
}

lone var sig PitmanArmForward
          , PitmanArmBackward
     extends PitmanArm {}

abstract sig PitmanArmDegree {}
one sig LowDegree, HighDegree
extends PitmanArmDegree {}
```

Class Diagram



Structural Properties

Properties that should hold in the initial state:

1. There are no keys in the car
2. The pitman arm and steering wheel are in neutral position
3. All the vehicle doors are closed
4. The light rotary switch is in the off position
5. The brake pedal is not activated, and the car is not moving
6. There's no subvoltage, overvoltage or oncoming traffic
7. The hazard warning and reverse gear are disabled
8. The camera of the car is ready

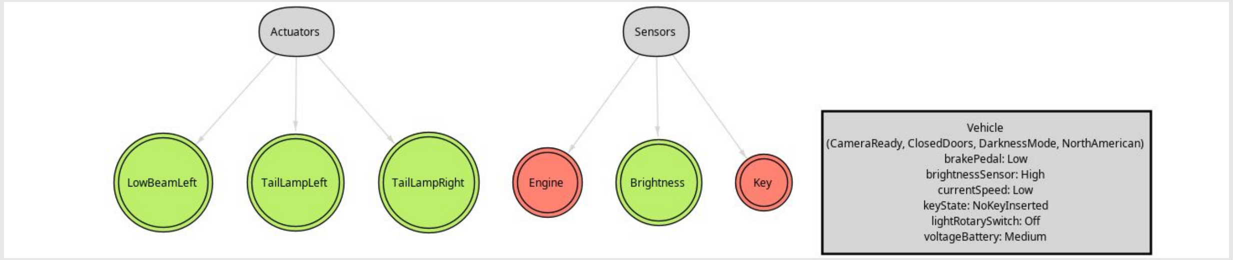
Main identified properties:

1. Only armoured vehicles have a darkness switch in the user interface
2. The pitman arm can not be moved in two different directions at the same time
3. Direction blinking is only available when the ignition is on
4. With subvoltage, adaptative high beam, ambient light, cornering light are not available
5. High beam is activated when adaptative high beam is active and the vehicle is driving quickly enough in a road without oncoming traffic

Validation and Verification

The validation was made by checking whether the properties hold in a specific scenario with the support of the visualizer.

```
run Example2 {
  no ArmoredVehicle
  no NorthAmericanVehicle
  Vehicle . keyState = KeyInIgnitionOnPosition
  Vehicle . lightRotarySwitch = Auto
  PitmanArmUpDown . pitmanArmUpDownPosition = Downward
  PitmanArmUpDown . pitmanArmDegree = HighDegree
}
```



```
Vehicle
(CameraReady, ClosedDoors, DarknessMode, NorthAmerican)
brakePedal: Low
brightnessSensor: High
currentSpeed: Low
keyState: NoKeyInserted
lightRotarySwitch: Off
voltageBattery: Medium
```

When performing the verification steps, some properties did not hold at first, so additional facts were also added to ensure desirable properties.

```
fact {
    {
        activeAdaptiveHighBeam
        Vehicle . currentSpeed != Low
        no OncommingTrafficVehicle
    } => some HighBeam
}
```

Dynamic Elements & Behavioural Properties

Because it would have been impracticable to model all the required temporal constraints in Alloy, we decided to abstract it, by specifying that eventually an action would be completed, but without more detail about how long it will take.

Most of the specification of our system is described as functional requirements. These also represented how the system should behave.

“If the camera recognizes the lights of an advancing vehicle, an activated high beam headlight is reduced to low beam headlight within 0.5 seconds by reducing the area of illumination to 65 meters by an adjustment of the headlight position as well as by reduction of the luminous strength to 30%.”

```
check ELS34 {
  always (
    {
      some OncommingTrafficVehicle
      before some HighBeam
    } => {
      no HighBeam
      some LowBeamLeft
      some LowBeamRight
    }
  )
}
```

Event Modelling

In each step of our system, a set of actions can be performed simultaneously. Each action can also maintain its associated component current state, or change to another by performing only one of the possible actions.

```
fact traces {
    always {
        updateActuators

        maintainKey
        or removeKey
        or insertKey
        or putKeyOnPosition
        or removeKeyFromPosition

        maintainBrakePedal
        or increaseBrakePedal
        or decreaseBrakePedal
        ...
    }
}
```

```
pred mantainKey {
  // Frame Conditions
  keyState' = keyState
}
```

```
pred removeKey {
  // Guards
  Vehicle . keyState = KeyInserted

  // Effects
  Vehicle . keyState' = NoKeyInserted
}
```

The actuators are also updated based on the values of auxiliary predicates. Each actuator has a predicate that indicates if it is necessarily on, and another that indicates if it is necessarily off, both cannot be true at the same time, but there are cases where both can be false, and the model is responsible for their value.

```
pred updateActuators {
  // Frame Conditions
  activeBrakeLight
  => some BrakeLight
  inactiveBrakeLight
  => no BrakeLight

  ...
}
```

Validation and Verification

The validation phase was made by checking whether a state was reachable from the initial state.

```
run Example2 {
  no ArmoredVehicle
  no NorthAmericanVehicle
  eventually Vehicle . keyState = KeyInIgnitionOnPosition
  eventually Vehicle . lightRotarySwitch = Auto
  eventually PitmanArmUpDown . pitmanArmUpDownPosition = Downward
  eventually PitmanArmUpDown . pitmanArmDegree = HighDegree
}
```

Then, the verification phase was performed by checking for counter examples in the defined checks.

```
check ELS12 {
  always (
    {
      before some HazardWarningVehicle
      no HazardWarningVehicle
      some PitmanArmUpDown
      engineOn
    } => eventually some Blink
  )
}
```

```
check ELS39 {
  always (
    Vehicle . brakePedal = Medium =>
    ((some BrakeLight) until (Vehicle . brakePedal != Medium))
    or (always some BrakeLight)
  )
}
```

Additional abstractions were added in this step, because some checks increased the system complexity so much that it could take several minutes to complete a model checking.

Dafny Implementation

Because of the high level of abstraction in the developed Alloy model and also the difficulty to find a component of the system that met the desired criteria, we’ve decided to implement it in a different approach.

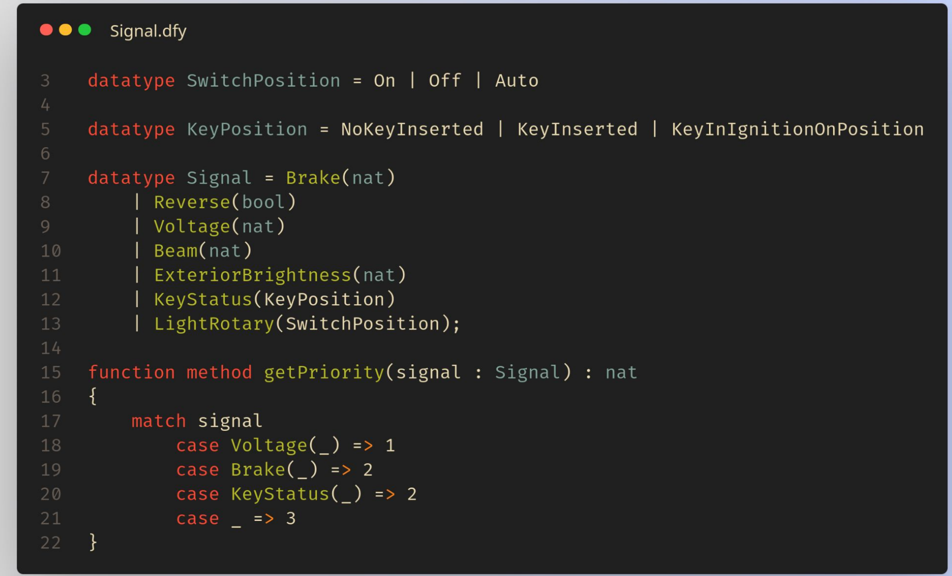
Our implementation describes a process queue in which signals are processed in their order of priority while maintaining the pre and post conditions defined for the vehicle.

Event Signals

These signals were modelled using algebraic data types. And they do not have references to other classes and Repr attributes (this will be important later).

Each signal represents an update event in some attribute.

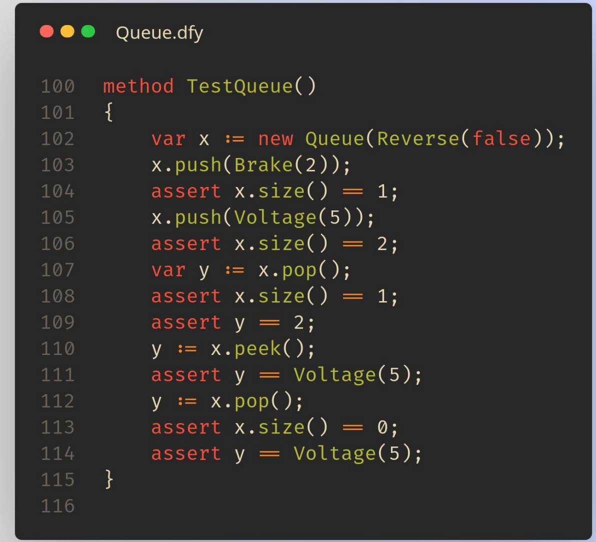
The priority of processing of the signals is dependent on their type.


Signal.dfy

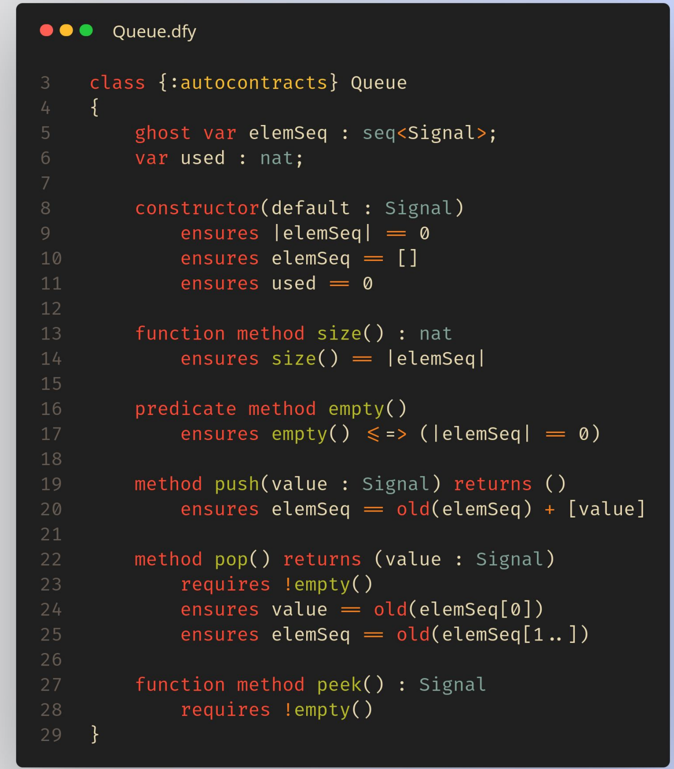
```
datatype SwitchPosition = On | Off | Auto

datatype KeyPosition = NoKeyInserted | KeyInserted | KeyInIgnitionOnPosition

datatype Signal = Brake(nat)
    | Reverse(bool)
    | Voltage(nat)
    | Beam(nat)
    | ExteriorBrightness(nat)
    | KeyStatus(KeyPosition)
    | LightRotary(SwitchPosition);

function method getPriority(signal : Signal) : nat
{
    match signal
        case Voltage(_) => 1
        case Brake(_) => 2
        case KeyStatus(_) => 2
        case _ => 3
}
```

Queue - Contract

Every function was defined by ensures and requires before being implemented, like all the classes that will appear in the next slides.


Queue.dfy

```
method TestQueue()
{
    var x := new Queue(Reverse(false));
    x.push(Brake(2));
    assert x.size() == 1;
    x.push(Voltage(5));
    assert x.size() == 2;
    var y := x.pop();
    assert x.size() == 1;
    assert y == 2;
    y := x.peek();
    assert y == Voltage(5);
    y := x.pop();
    assert x.size() == 0;
    assert y == Voltage(5);
}
```

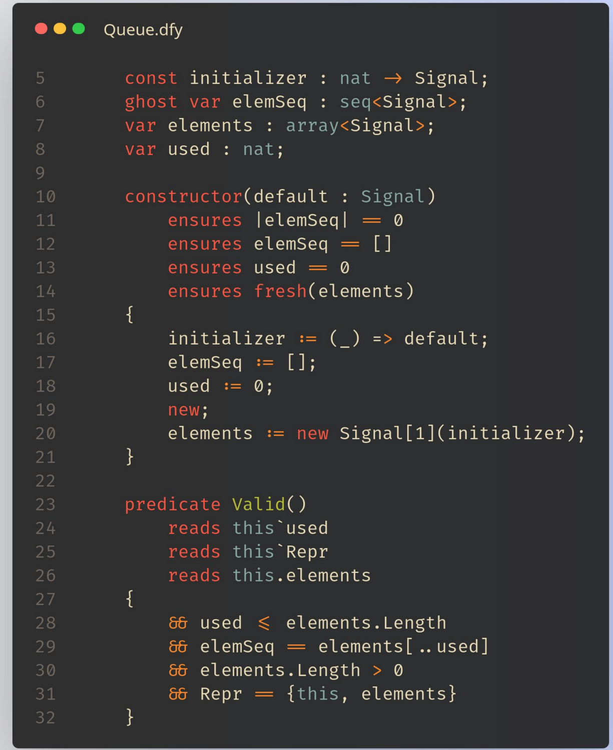

Queue.dfy

```
class {:autocontracts} Queue
{
    ghost var elemSeq : seq<Signal>;
    var used : nat;

    constructor(default : Signal)
        ensures |elemSeq| == 0
        ensures elemSeq == []
        ensures used == 0

    function method size() : nat
        ensures size() == |elemSeq|

    predicate method empty()
        ensures empty() <==> (|elemSeq| == 0)

    method push(value : Signal) returns ()
        ensures elemSeq == old(elemSeq) + [value]

    method pop() returns (value : Signal)
        requires !empty()
        ensures value == old(elemSeq[0])
        ensures elemSeq == old(elemSeq[1..])

    function method peek() : Signal
        requires !empty()
}
```

Queue - Implementation

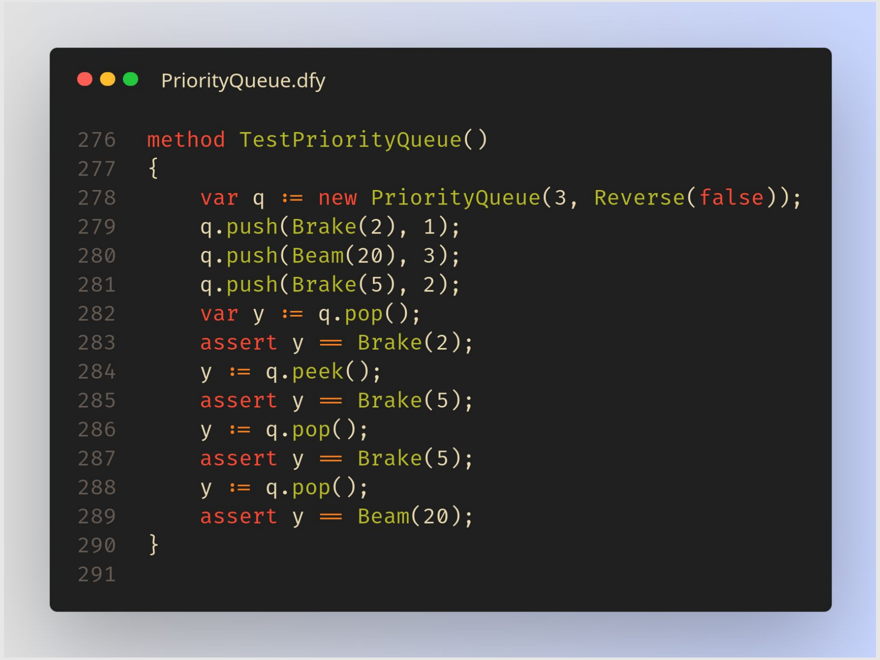
We created a Valid() method, and an array attribute for the concrete implementation.

And a new method, grow(), that is only required for limitations in the array element.



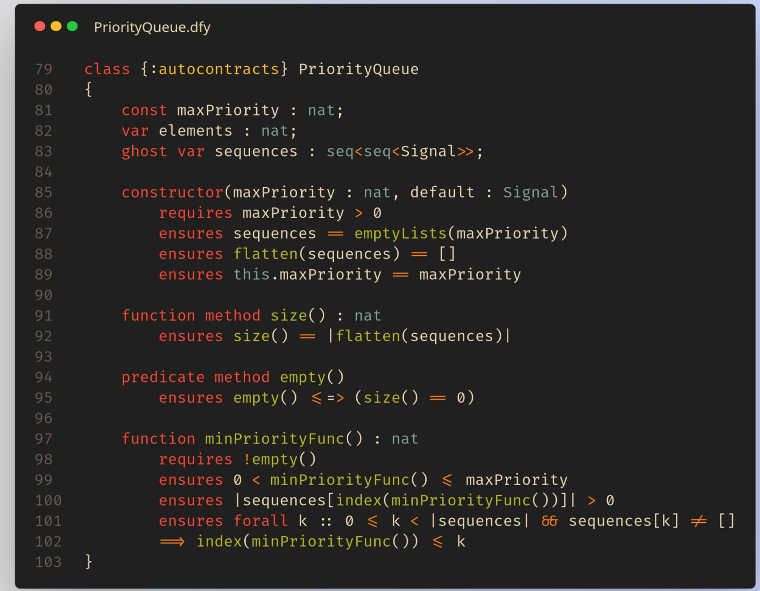
```
    const initializer : nat -> Signal;
    ghost var elemSeq : seq<Signal>;
    var elements : array<Signal>;
    var used : nat;

    constructor(default : Signal)
        ensures |elemSeq| == 0
        ensures elemSeq == []
        ensures used == 0
        ensures fresh(elements)
    {
        initializer := (_) => default;
        elemSeq := [];
        used := 0;
        new;
        elements := new Signal[1](initializer);
    }

    predicate Valid()
        reads this`used
        reads this`Repr
        reads this.elements
    {
        && used <= elements.Length
        && elemSeq == elements[..used]
        && elements.Length > 0
        && Repr == {this, elements}
    }
```

Priority Queue - Contract

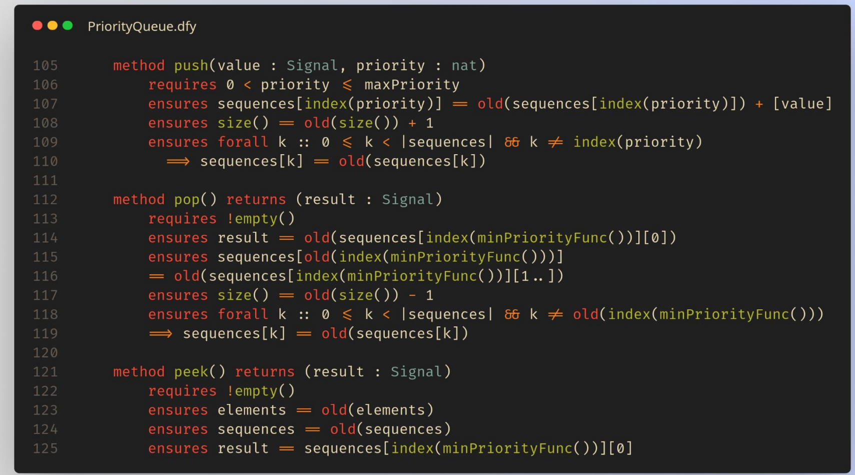

PriorityQueue.dfy

```
method TestPriorityQueue()
{
    var q := new PriorityQueue(3, Reverse(false));
    q.push(Brake(2), 1);
    q.push(Beam(20), 3);
    q.push(Brake(5), 2);
    var y := q.pop();
    assert y == Brake(2);
    y := q.peek();
    assert y == Brake(5);
    y := q.pop();
    assert y == Brake(5);
    y := q.pop();
    assert y == Beam(20);
}

```

Priority Queue - Contract


PriorityQueue.dfy

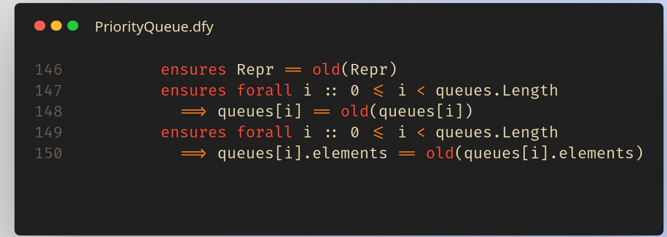
```
class {:autocontracts} PriorityQueue
{
    const maxPriority : nat;
    var elements : nat;
    ghost var sequences : seq<seq<Signal>>;

    constructor(maxPriority : nat, default : Signal)
        requires maxPriority > 0
        ensures sequences == emptyLists(maxPriority)
        ensures flatten(sequences) == []
        ensures this.maxPriority == maxPriority

    function method size() : nat
        ensures size() == |flatten(sequences)|

    predicate method empty()
        ensures empty() <==> (size() == 0)

    function minPriorityFunc() : nat
        requires !empty()
        ensures 0 < minPriorityFunc() <= maxPriority
        ensures |sequences[index(minPriorityFunc())]| > 0
        ensures forall k :: 0 <= k < |sequences| && sequences[k] != []
        ==> index(minPriorityFunc()) <= k
}
```


PriorityQueue.dfy

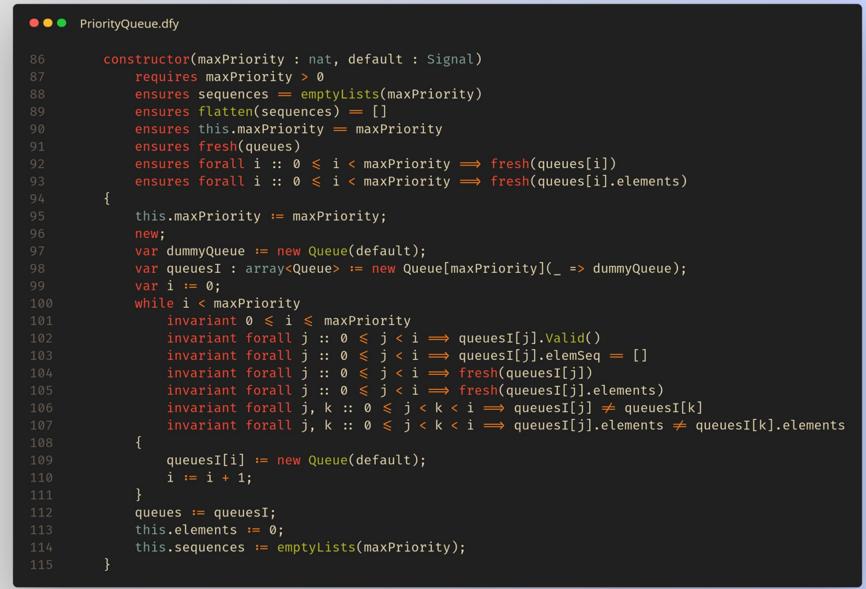
```
    method push(value : Signal, priority : nat)
        requires 0 < priority <= maxPriority
        ensures sequences[index(priority)] == old(sequences[index(priority)]) + [value]
        ensures size() == old(size()) + 1
        ensures forall k :: 0 <= k < |sequences| && k != index(priority)
          ==> sequences[k] == old(sequences[k])

    method pop() returns (result : Signal)
        requires !empty()
        ensures result == old(sequences[index(minPriorityFunc())][0])
        ensures sequences[old(index(minPriorityFunc()))]
        == old(sequences[index(minPriorityFunc())][1..])
        ensures size() == old(size()) - 1
        ensures forall k :: 0 <= k < |sequences| && k != old(index(minPriorityFunc()))
        ==> sequences[k] == old(sequences[k])

    method peek() returns (result : Signal)
        requires !empty()
        ensures elements == old(elements)
        ensures sequences == old(sequences)
        ensures result == sequences[index(minPriorityFunc())][0]
```

Priority Queue - Implementation

Dealing with references in Dafny is very complex, because the solver needs to know if the references are not duplicated and do not point to memory that cannot be updated. To solve that, we had to include ensures clauses that refer to the concrete implementation of the classes.


PriorityQueue.dfy

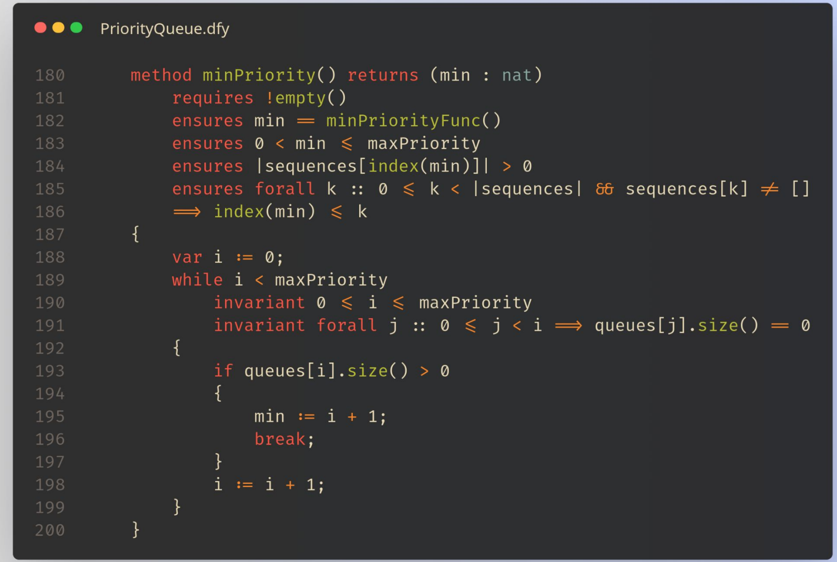
```
ensures Repr == old(Repr)
ensures forall i :: 0 <= i < queues.Length
  ==> queues[i] == old(queues[i])
ensures forall i :: 0 <= i < queues.Length
  ==> queues[i].elements == old(queues[i].elements)
```


PriorityQueue.dfy

```
    constructor(maxPriority : nat, default : Signal)
        requires maxPriority > 0
        ensures sequences == emptyLists(maxPriority)
        ensures flatten(sequences) == []
        ensures this.maxPriority == maxPriority
        ensures fresh(queues)
        ensures forall i :: 0 <= i < maxPriority ==> fresh(queues[i])
        ensures forall i :: 0 <= i < maxPriority ==> fresh(queues[i].elements)
    {
        this.maxPriority := maxPriority;
        new;
        var dummyQueue := new Queue(default);
        var queuesI : array<Queue> := new Queue[maxPriority](_ => dummyQueue);
        var i := 0;
        while i < maxPriority
            invariant 0 <= i <= maxPriority
            invariant forall j :: 0 <= j < i ==> queuesI[j].Valid()
            invariant forall j :: 0 <= j < i ==> queuesI[j].elemSeq == []
            invariant forall j :: 0 <= j < i ==> fresh(queuesI[j])
            invariant forall j :: 0 <= j < i ==> fresh(queuesI[j].elements)
            invariant forall j, k :: 0 <= j < k < i ==> queuesI[j] != queuesI[k]
            invariant forall j, k :: 0 <= j < k < i ==> queuesI[j].elements != queuesI[k].elements
        {
            queuesI[i] := new Queue(default);
            i := i + 1;
        }
        queues := queuesI;
        this.elements := 0;
        this.sequences := emptyLists(maxPriority);
    }
```

Priority Queue - Performance

To calculate the minimum priority we need loops, so, instead of using the minPriorityFunc() inside the methods, we created a method minPriority() that has performance of iterative code and always returns the same result.



PriorityQueue.dfy

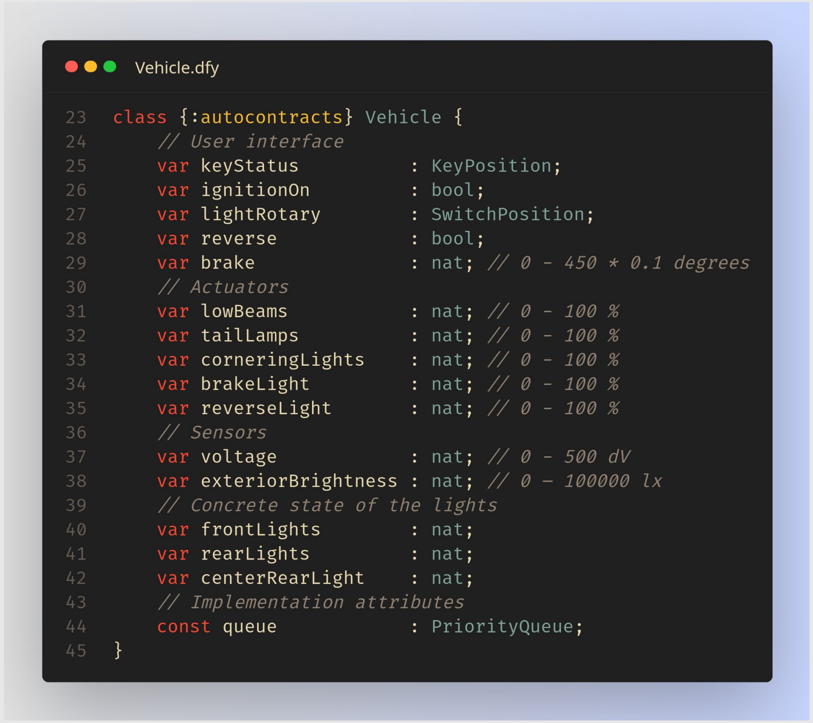
```
    method minPriority() returns (min : nat)
        requires !empty()
        ensures min == minPriorityFunc()
        ensures 0 < min <= maxPriority
        ensures |sequences[index(min)]| > 0
        ensures forall k :: 0 <= k < |sequences| && sequences[k] != []
        ==> index(min) <= k
    {
        var i := 0;
        while i < maxPriority
            invariant 0 <= i <= maxPriority
            invariant forall j :: 0 <= j < i ==> queues[j].size() == 0
        {
            if queues[i].size() > 0
            {
                min := i + 1;
                break;
            }
            i := i + 1;
        }
    }
```

Vehicle - Variables

Some attributes are changed via signals, and others are dependent on the values of those attributes.

User interface and sensors can be changed with the use of Signals (as presented before).

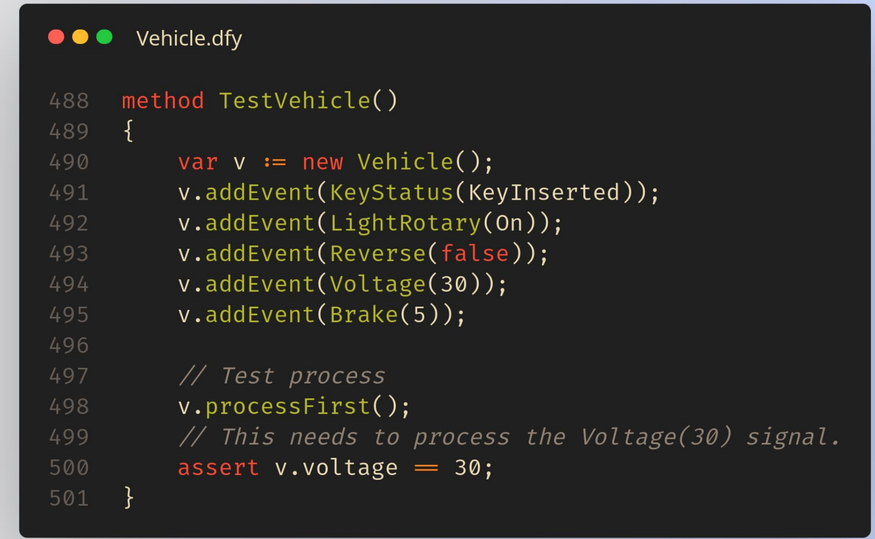
Actuators and lights status are calculated using the first attributes.



Vehicle.dfy

```
class {:autocontracts} Vehicle {
    // User interface
    var keyStatus           : KeyPosition;
    var ignitionOn          : bool;
    var lightRotary         : SwitchPosition;
    var reverse             : bool;
    var brake               : nat; // 0 - 450 * 0.1 degrees
    // Actuators
    var lowBeams            : nat; // 0 - 100 %
    var tailLamps           : nat; // 0 - 100 %
    var corneringLights     : nat; // 0 - 100 %
    var brakeLight          : nat; // 0 - 100 %
    var reverseLight        : nat; // 0 - 100 %
    // Sensors
    var voltage             : nat; // 0 - 500 dV
    var exteriorBrightness : nat; // 0 – 100000 lx
    // Concrete state of the lights
    var frontLights         : nat;
    var rearLights          : nat;
    var centerRearLight     : nat;
    // Implementation attributes
    const queue             : PriorityQueue;
}
```

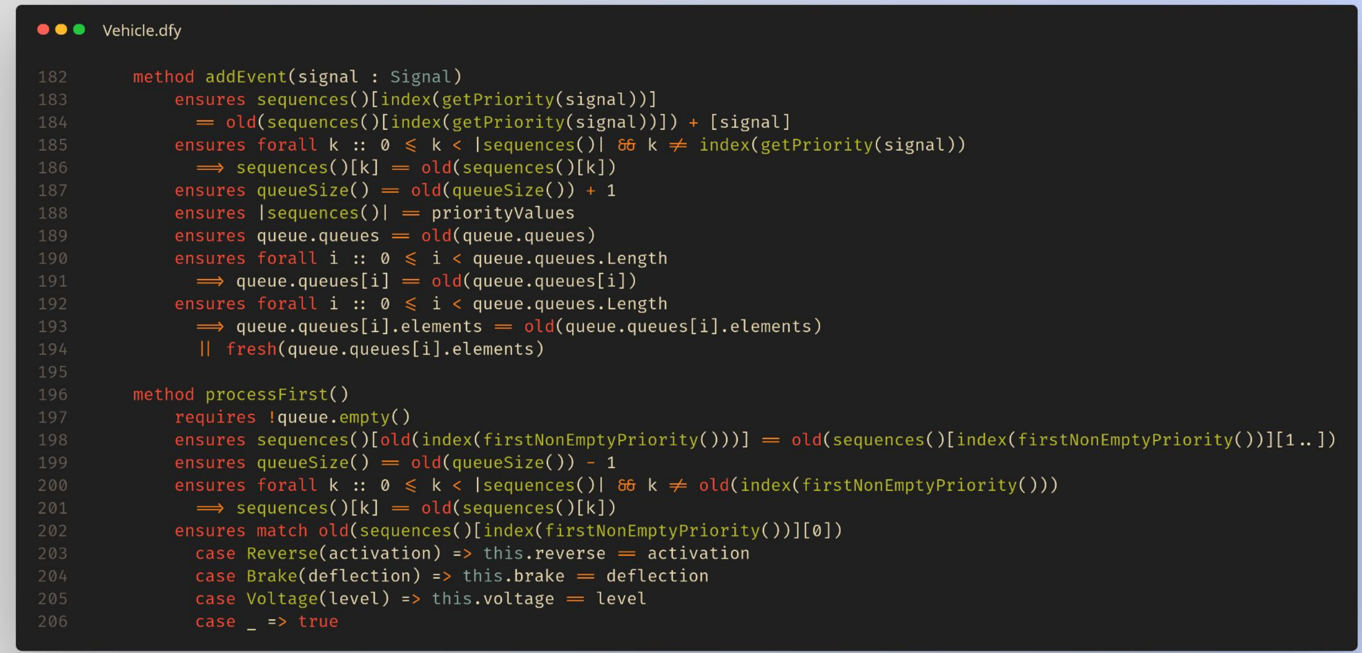
Vehicle - Contract



Vehicle.dfy

```
method TestVehicle()
{
    var v := new Vehicle();
    v.addEvent(KeyStatus(KeyInserted));
    v.addEvent(LightRotary(On));
    v.addEvent(Reverse(false));
    v.addEvent(Voltage(30));
    v.addEvent(Brake(5));

    // Test process
    v.processFirst();
    // This needs to process the Voltage(30) signal.
    assert v.voltage == 30;
}
```

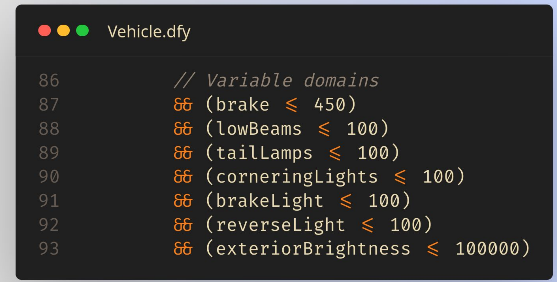
Vehicle - Contract



Vehicle.dfy

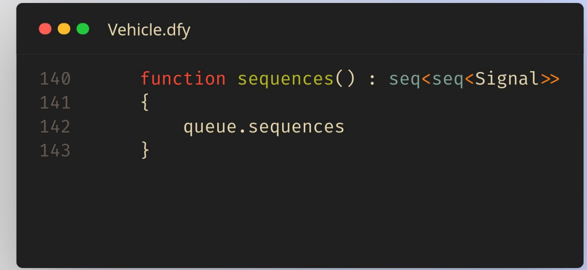
```
    method addEvent(signal : Signal)
        ensures sequences()[index(getPriority(signal))]
          == old(sequences()[index(getPriority(signal))]) + [signal]
        ensures forall k :: 0 <= k < |sequences()| && k != index(getPriority(signal))
          ==> sequences()[k] == old(sequences()[k])
        ensures queueSize() == old(queueSize()) + 1
        ensures |sequences()| == priorityValues
        ensures queue.queues == old(queue.queues)
        ensures forall i :: 0 <= i < queue.queues.Length
          ==> queue.queues[i] == old(queue.queues[i])
        ensures forall i :: 0 <= i < queue.queues.Length
          ==> queue.queues[i].elements == old(queue.queues[i].elements)
          || fresh(queue.queues[i].elements)

    method processFirst()
        requires !queue.empty()
        ensures sequences()[old(index(firstNonEmptyPriority()))] == old(sequences()[index(firstNonEmptyPriority())][1..])
        ensures queueSize() == old(queueSize()) - 1
        ensures forall k :: 0 <= k < |sequences()| && k != old(index(firstNonEmptyPriority()))
          ==> sequences()[k] == old(sequences()[k])
        ensures match old(sequences()[index(firstNonEmptyPriority())][0])
          case Reverse(activation) => this.reverse == activation
          case Brake(deflection) => this.brake == deflection
          case Voltage(level) => this.voltage == level
          case _ => true
```

Vehicle - Valid



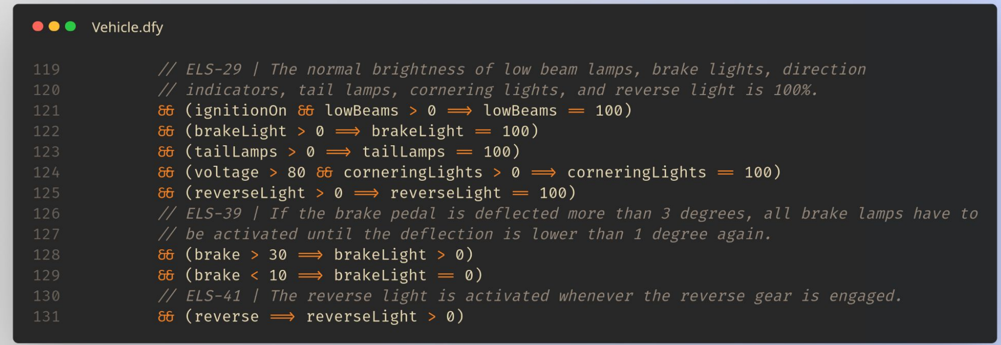
Vehicle.dfy

```
        // Variable domains
        && (brake <= 450)
        && (lowBeams <= 100)
        && (tailLamps <= 100)
        && (corneringLights <= 100)
        && (brakeLight <= 100)
        && (reverseLight <= 100)
        && (exteriorBrightness <= 100000)
```



Vehicle.dfy

```
    function sequences() : seq<seq<Signal>>
    {
        queue.sequences
    }
```



Vehicle.dfy

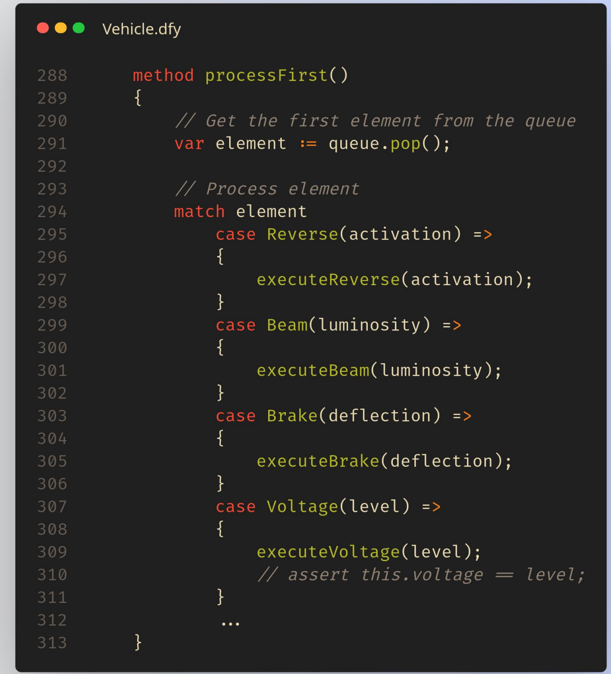
```
        // ELS-29 | The normal brightness of low beam lamps, brake lights, direction
        // indicators, tail lamps, cornering lights, and reverse light is 100%.
        && (ignitionOn && lowBeams > 0 ==> lowBeams == 100)
        && (brakeLight > 0 ==> brakeLight == 100)
        && (tailLamps > 0 ==> tailLamps == 100)
        && (voltage > 80 && corneringLights > 0 ==> corneringLights == 100)
        && (reverseLight > 0 ==> reverseLight == 100)
        // ELS-39 | If the brake pedal is deflected more than 3 degrees, all brake lamps have to
        // be activated until the deflection is lower than 1 degree again.
        && (brake > 30 ==> brakeLight > 0)
        && (brake < 10 ==> brakeLight == 0)
        // ELS-41 | The reverse light is activated whenever the reverse gear is engaged.
        && (reverse ==> reverseLight > 0)
```

Vehicle - Implementation

All notes presented in the priority queue section still apply. But now we have one more layer to access the reference attributes.

The processFirst() method was implemented by having one method for each type of event.

Most of the array and element processing happens in the PriorityQueue class.



Vehicle.dfy

```
    method processFirst()
    {
        // Get the first element from the queue
        var element := queue.pop();

        // Process element
        match element
            case Reverse(activation) =>
            {
                executeReverse(activation);
            }
            case Beam(luminosity) =>
            {
                executeBeam(luminosity);
            }
            case Brake(deflection) =>
            {
                executeBrake(deflection);
            }
            case Voltage(level) =>
            {
                executeVoltage(level);
                // assert this.voltage == level;
            }
            ...
    }
```

Questions?