Implementation and Verification of an Automotive System

Alexandre Abreu, Breno Accioly
Faculty of Engineering of the University of Porto
Porto, Portugal
{up201800168,up201800170}@up.pt

Introduction

This project was made within the framework of the Formal Methods for Critical Systems course at FEUP. To experiment with the validation of a realistic system, an automotive system specification was chosen.

The present report describes the work done in the second part of the project. Initially, the implementation of this part should have been made using a component of the system modelled in the first part, however, because of the high level of abstraction in the developed Alloy model, we've opted to develop it in a different approach, by implementing a priority queue which stored events of our original system. These events were then processed following a design by contracts approach with the Dafny formal specification language. In the following sections, we will discuss in more detail all these topics.

This report includes code in all the sections, but it should be understood without it, the code is there for exemplification or if the reader wants to check or know how something was done.

1 Paradigm

The main objective of this work is to ensure the proper execution of a system component using the Design by Contract paradigm. The mentioned component describes a process queue in which events are processed in their order of priority, for instance, the fault handling events has more priority and should be processed first. Furthermore, each process execution should respect the preconditions and postconditions related to the vehicle lights system as implemented in the Alloy model.

Some examples of predicates translated from the Alloy model are presented below:

```
predicate Valid()
{
    ...
    // ELS-15 | While the ignition is in position KeyInserted: if the light rotary switch
    // is turned to the position On, the low beam headlights are activated
    // with 50% (to save power).
    && (keyStatus == KeyInserted && lightRotary == On ==> lowBeams == 50)
    // ELS-16 | If the ignition is already off and the driver turns the light rotary
    // switch to position Auto, the low beam headlights remain off or are
    // deactivated (depending on the previous state).
    && (!engineOn && lightRotary == Auto ==> lowBeams == 0)
    // ELS-18 | If the light rotary switch is in position Auto and the ignition is On, the
    // low beam headlights are activated as soon as the exterior brightness
    // is lower than a threshold of 200 lx. If the exterior brightness exceeds
    // a threshold of 250 lx, the low beam headlights are deactivated.
    && (engineOn && lightRotary == Auto && exteriorBrightness < 200 ==> lowBeams > 0)
    && (engineOn && lightRotary == Auto && exteriorBrightness > 250 ==> lowBeams == 0)
    // ELS-22 | Whenever the low or high beam headlights are activated, the tail
    // lights are activated, too.
    && (lowBeams > 0 ==> tailLamps > 0)
    // ELS-27 | If reverse gear is activated, both opposite cornering lights are
    // activated.
    && (reverse && voltage > 80 ==> corneringLights > 0)
```

```
        ...
    }
```

2 Test Methods

The first thing done was to create functions that would guide us, like in test-driven development. These need to describe the functionality we want the solver to be able to prove, so, the next three test methods were created:

2.1 Queue

The queue should work like a standard queue, it should provide a way to push values and pop them in order.

```
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 == Brake(2);
    y := x.peek();
    assert y == Voltage(5);
    y := x.pop();
    assert x.size() == 0;
    assert y == Voltage(5);
}
```

2.2 Priority Queue

The priority queue should work like a queue, but elements are ordered by priority first, in case they have the same priority they stay in order of entrance.

```
method TestPriorityQueue()
{
    var q := new PriorityQueue(3);
    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);
}
```

2.3 Vehicle

A Vehicle can be seen as a wrapper over the priority queues, that can add the signals to the queue with priorities calculated inside it, based on the priorities of the types of signals.

```
method TestVehicle()
{
```

```
    var v := new Vehicle();
    assert v.sequences() == [[], [], []];
    assert v.keyStatus == NoKeyInserted;
    assert v.voltage == 10;
    v.addSignal(Reverse(false));
    v.addSignal(Voltage(30));
    v.addSignal(Brake(5));
    assert v.sequences() == [[Voltage(30)], [Brake(5)], [Reverse(false)]];
    var s := v.getFirst();
    assert s == Voltage(30);
    v.processFirst();
    assert v.voltage == 30;
    v.processFirst();
    assert v.queueSize() == 1;
    assert v.sequences()[0] == [];
    assert v.sequences()[1] == [];
    assert v.sequences()[2] == [Reverse(false)];
}
```

3 Method Annotations

The next goal of our project was to make Dafny be able to prove the `asserts` defined in the tests defined in the previous section. To do that, we defined abstract representations for the classes and the pre- and postconditions of the methods, using only the abstract representation of the respective class' state.

The methods were marked as `axioms`, an annotation of Dafny that indicates the `ensures` predicates can be assumed as true inside the method body. In that way, we can see if the test `asserts` are valid without implementing the methods and concrete variables.

All of these were enough for Dafny to be able to prove our assertions in the tests.

3.1 Queue

```
class Queue
{
    ghost var elemSeq : seq<Signal>;

    constructor {:axiom} (default : Signal)
        ensures |elemSeq| == 0
        ensures elemSeq == []

    function method {:axiom} size() : nat
        ensures size() == |elemSeq|

    predicate method {:axiom} empty()
        ensures empty() <==> (|elemSeq| == 0)

    function method {:axiom} peek() : Signal
        requires !empty()
        ensures peek() == elemSeq[0]

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

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

3.2 Priority Queue

```
class PriorityQueue
{
    ghost var sequences : seq<seq<Signal>>;

    constructor {:axiom} (maxPriority : nat)
        requires maxPriority > 0
        ensures sequences == emptyLists(maxPriority)
        ensures flatten(sequences) == []

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

    predicate method {:axiom} empty()
        ensures empty() <==> (size() == 0)

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

    method {:axiom} push(value : Signal, priority : nat)
        requires 0 < priority <= |sequences|
        ensures |sequences| == old(|sequences|)
        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 {:axiom} pop() returns (result : Signal)
        requires !empty()
        ensures |sequences| == old(|sequences|)
        ensures result == old(sequences[index(minPriorityFunc())][0])
        ensures sequences[old(index(minPriorityFunc()))] == old(sequences[index(minPriorityFunc())][
        ensures size() == old(size()) - 1
        ensures forall k :: 0 <= k < |sequences| && k != old(index(minPriorityFunc()))
        ==> sequences[k] == old(sequences[k])

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

3.3 Vehicle

One thing to note is that the abstract representation of a vehicle signal queue is the same as its underlying queue plus the vehicle state attributes. This is clear in the contract of `sequences()`.

```
class Vehicle {
    // [..] Attributes

    constructor {:axiom} ()
        ensures sequences() == emptyLists(priorityValues)
        ensures voltage == 100
```

```
    ensures exteriorBrightness == 50000
    ensures keyStatus == NoKeyInserted
    ensures lightRotary == Off
    ensures reverse == false
    ensures brake == 0
    ensures engineOn == false
    ensures lowBeams == 0
    ensures tailLamps == 0
    ensures corneringLights == 0
    ensures brakeLight == 0
    ensures reverseLight == 0

function {:axiom} sequences() : seq<seq<Signal>>
    ensures sequences() == queue.sequences

function method {:axiom} queueSize() : nat
    ensures queueSize() == |flatten(sequences())|

predicate method {:axiom} emptyQueue()
    ensures emptyQueue() <==> (queueSize() == 0)

function {:axiom} firstNonEmptyPriority() : nat
    requires !emptyQueue()
    ensures 0 < firstNonEmptyPriority() <= |sequences()|
    ensures |sequences()[index(firstNonEmptyPriority())]| > 0
    ensures forall k :: 0 <= k < |sequences()| && sequences()[k] != []
    ==> index(firstNonEmptyPriority()) <= k

function {:axiom} front() : Signal
    requires !emptyQueue()
    ensures front() == sequences()[index(firstNonEmptyPriority())][0]

method {:axiom} getFirst() returns (result : Signal)
    requires !emptyQueue()
    ensures !emptyQueue()
    ensures result == front()
    ensures sequences() == old(sequences())

method {:axiom} addSignal(signal : Signal)
    requires 1 <= getPriority(signal) <= |sequences()|
    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

method {:axiom} processFirst()
    requires !emptyQueue()
    ensures sequences()[old(index(firstNonEmptyPriority()))] == old(sequences()[index(firstNonEm
    ensures queueSize() == old(queueSize()) - 1
    ensures forall k :: 0 <= k < |sequences()| && k != old(index(firstNonEmptyPriority()))
      ==> sequences()[k] == old(sequences()[k])
    ensures match old(front())
      case Voltage(level) => voltage == level
        case _ => voltage == old(voltage)
    ensures match old(front())
      case ExteriorBrightness(level) => exteriorBrightness == level
```

```
            case _ => exteriorBrightness == old(exteriorBrightness)
        // [...] Other cases
}
```

4 Implementation Attributes

4.1 Queue

The queue works with an array, so, we added three attributes:

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

The initializer is needed because we need it to create arrays, and used as a variable that tells us how many elements we have, when the array is full we need to allocate a bigger array.

4.2 Priority Queue

The priority queue has an array of queues, so, we added the following attributes:

```
const maxPriority : nat;
var queues : array<Queue>;
var elements : nat;
```

`maxPriority` is the number of elements in `queues`, and elements is the sum of the number of elements in all elements of `queues`.

5 Class Invariants

The following invariants were created before having to prove correctness of references, as talked in section 6.1. So, more constraints related to this problem were added afterwards.

5.1 Queue

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

5.2 Priority Queue

```
predicate Valid()
{
    && |sequences| == maxPriority
    && queues.Length == maxPriority
    && elements == |flatten(sequences)|
}
```

5.3 Vehicle

The invariant of a vehicle is composed by the requirements of the system, discussed in section 1, and the domain of the state attributes, like:

```
predicate Valid()
    ...
    && (brake <= 450)
    && (lowBeams <= 100)
    && (tailLamps <= 100)
    && (corneringLights <= 100)
    && (brakeLight <= 100)
    && (reverseLight <= 100)
    && (exteriorBrightness <= 100000)
    ...
}
```

6 Method Definitions

After that, we implemented the methods so that they followed the annotations made before.

6.1 References and Objects in Dafny

Because of the complexity when using references in Dafny, we needed to include additional postconditions to ensure that object references are managed correctly.

One example of this is the priority queue `pop()` contract, which was updated to include:

```
method pop() returns (result : Signal)
    ...
    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)
    ...
```

These postconditions are needed to require that:

- References are not duplicated in containers, which are the arrays, so that altering one object does not alter another one;
- References does not change, so that we do not update references to things we cannot modify;
- When a reference changes it is an entirely new value, freshly allocated, for the same reason as the previous item.