Modelling and Validation 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.

It is composed by two subsystems, but only one of them was implemented: Adaptive Exterior Light. In summary, this subsystem includes sensors and interface components, that serve as inputs to the system, and actuators, that interact with the lamps of the car and serve as output. The activation and deactivation of the actuators depend on complex functions based on the context of the system, and the aim of the project is to model them in Alloy.

1 System Structure Modelling

Our model is composed of different units related to the Adaptative Exterior Light System of a vehicle, namely the user interface, the sensors, and the actuators.

As a design choice, we modelled our system to have only one car in the universe, known as *Vehicle*. This can simplify the model as some components of the vehicle don't need to have relationships defined, after all, if they exist, then they can only be part of the only vehicle in the system.

Signatures

Some components of our system, as the *current speed* and the *headlights*, can be attributed the value of an enumerator *Level*, that takes the states *Low*, *Mid* and *High*, and the non-existence of the component can be interpreted as a neutral value or disabled stated, depending on our needs for that attribute.

The components that are binary in nature, are modelled as subsets of *Vehicle*, so, the attribute is false if the set is empty, and true otherwise. These include the following attributes: *HazardWarning*, *ClosedDoors* and *OncommingTraffic*.

Other components, like *keyState* and *lightRotarySwitch* are represented with enums that include all their possible states.

Actuators are lone signatures, where the emptiness of the respective set indicates that the actuator is not activated. Some of them also have levels, having those defined as attributes. *HighBeam* has two subsets that indicate if it *highRange* and *highMotor* are true.

Techniques for Boolean Fields

As previously mentioned, most boolean attributes were represented by using subsets in Alloy, that allowed them to be easily tested by the `some` keyword, which is faster than boolean objects for the analyser. For the armoured attribute of vehicles, a subtype was used.

Visualization

As there is no easy way to visualize the structure, a layer above the system was designed just for visualization purposes. It includes the creation of dummy elements that can be used to represent the absence or presence of a real sensor or actuator. The theme hides the true elements whilst showing the dummy elements renamed to represent the first ones. It also creates grouping objects that only exist to have elements of the same nature together, e.g., the dummy actuators are related to a *ActuatorSystem* object, so they appear close to each other.

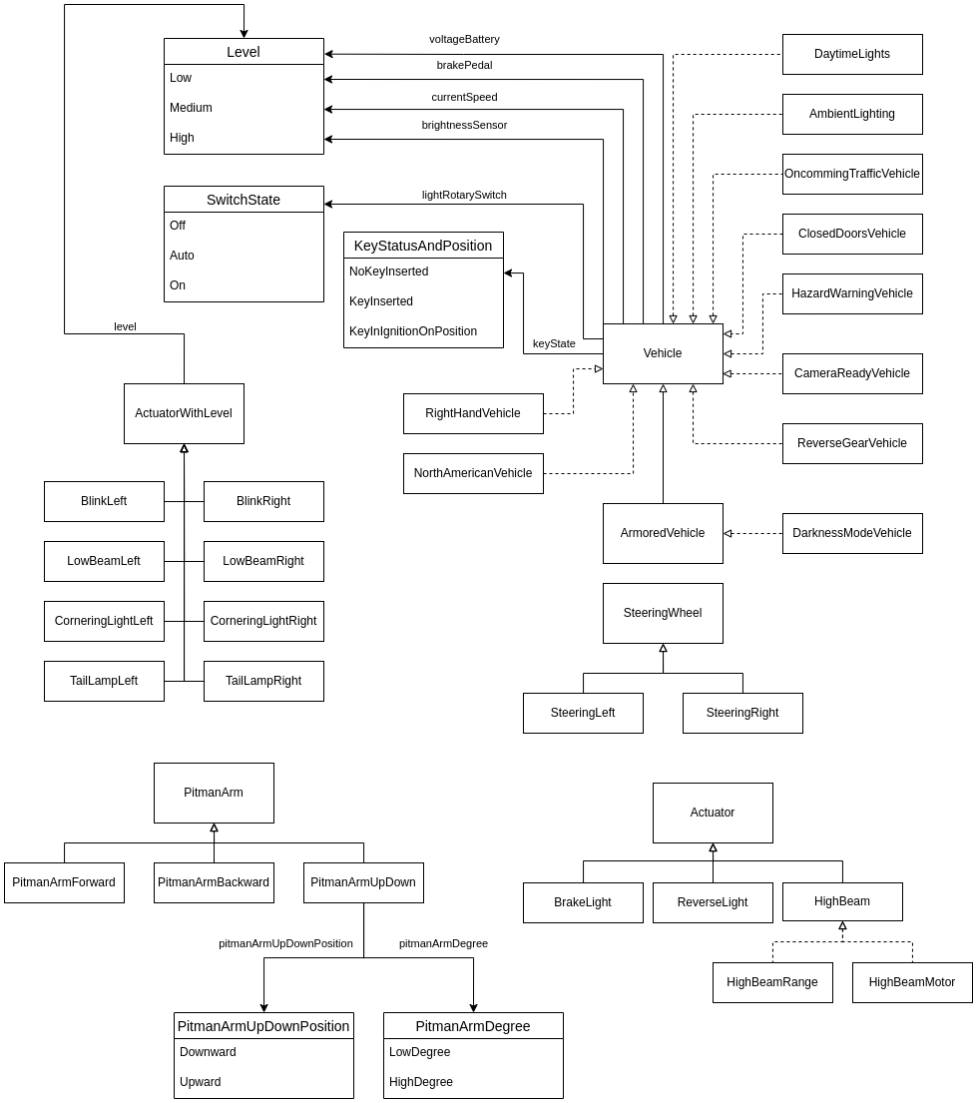

Figure 1: Diagram of main signatures of the model. Dashed arrows represent subsets.

2 Structural Properties Specification

To model the structural properties of our system, we analysed all the requirements, searching for the ones that did not depended on dynamic elements. Their specification is as follows:

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 time;
3. Direction blinking is only available when the ignition is on;
4. With subvoltage, adaptive high beam, ambient light, cornering light are not available;
5. High beam is activated when adaptive high beam is active and the vehicle is driving quickly enough in a road without oncoming traffic;
6. and others.

Initial State

We also got some restrictions on the initial state of the system, based on our assumptions or just simplifications of it. So, in the initial state, the following items are valid:

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

3 Structural Model Validation

In order to validate our structural model, we created scenarios that our car could be in, and checked whether an instance for each one of those scenarios could be found.

For instance, in the scenario with an armoured North American car where the darkness mode is available, it is possible to obtain the following diagram:

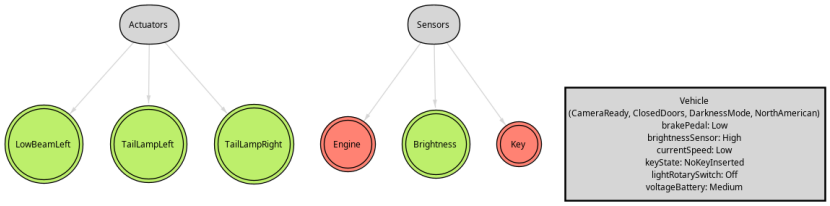

Figure 2: Result of the execution of a scenario.

It is important to notice that only sensors and actuators that are not in the default value are represented in the visualization to get a more concise diagram. The "neutral" states are defined per element, and, even though most of them are intuitive, some could have different states as the "neutral" ones.

4 Structural Properties Verification

Some properties are satisfied just by the modelling itself, like property #1. By making an armoured vehicle an extension, we can define an attribute that only this instance can possess:

```
sig ArmoredVehicle extends Vehicle {
    , var darknessMode: lone Yes
}
```

For other properties, such as property #5, additional facts were needed to implement it:

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

All the properties were verified using the visualizer and running examples and scenarios to check if the model was consistent.

5 Dynamic Elements Introduction

As it is difficult to model time intervals in Alloy, we opted to abstract the temporal properties of our system and specify that a trace for an action is eventually completed. Thus, we were not able to ensure properties as ELS-10, which states that the duration of a flashing cycle is 1 second.

The pitman arm and switches functionalities, as well as dynamic elements that not depended on directly from user interaction, were added to the system and all the variable elements were marked as such.

6 Behavioural Properties Specification

Most of the functional requirements for the system represented desirable properties for the dynamic behaviour of the system. We modelled them in Alloy, maintaining the time abstraction. So, this part of the project was very straightforward as it was defined by the implementation of the final properties defined in the specification.

They were split into six modules inside the *properties* folder, where each one of them represents the properties of the different functions of the adaptive light system. Some of them were not done because of conflicts with the abstraction or were hard to interpret or ambiguous.

For example, ELS-34 represents the way some requirement properties were removed, but others were preserved. It states:

> "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%."

And it was implemented with the following check, where the time requirements were removed and the evolution of the luminosity needed for the high beams to turn into low beams were abstracted (i.e., the removal of the high beams and addition of low beams implicitly represent the 30% reduction of the luminous strength):

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

In other property, defined in ELS-23:

> "In USA or Canada, tail lights realize the direction indicator lamps. In case of direction blinking or hazard blinking, blinking has preference against normal tail lights."

It is an example of a conflict with our system model. This happens because the Blink actuators already have the role of the tail lamp, making the differentiation between cars from North America or from other countries not relevant.

7 Event Modelling

In every update of our system, the actuators are "recalculated" based on the system context. The trace indicates which actions are mutually exclusive and which actions do not interfere with each other, for example, in the following exert, there are 5 possible changes of the key state and 3 for the brake state, one for each can occur, and the "nop" operation is declared implicitly by the trace composed by the combination of all the operations that maintain their respective state.

```
fact traces {
    always {
        updateActuators

        maintainKey
        or removeKey
        or insertKey
        or putKeyOnPosition
        or removeKeyFromPosition

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

A lot of sensors and user interface components can freely change their states to any other one, so they are not included in *traces*.

The recalculation of the actuators is based on the *updateActuators* fact. It uses auxiliary predicates that indicate if the actuator needs to be active or if it needs to be inactive, the "need" concept is very important, because sometimes both can be false, and the system is free to comply to our past promises (e.g., when tip blinking, eventually the respective blink actuator will be activated, this can occur in a state that does not need the actuator to be active nor inactive). The predicate is exemplified below:

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

  activeReverseLight
  => some ReverseLight
  inactiveReverseLight
  => no ReverseLight

  ...
}
```

One special mention is that, we needed to define lots of cycles that were defined by a property p that needed to be true until q was true, but the `until` keyword in allow would only be valid if q was true in some state of the future, so, we used special case adaptations of the following predicates to include the cases where q is never true:

```
pred cycle {
    (
        always
        changing
        and not q
    ) or (
        changing until q
    )
}

pred changing {
    eventually p
    eventually no p
}
```

8 Behavioural Model Validation

In order to validate whether the system behaviour is properly working, after inserting any dynamic element, we checked if the model could eventually evolve to a state that the dynamic element was active and valid using specific scenarios. And all the static properties needed to be valid in all states of the found traces.

Some minor changes were made, as our system did not break many rules in this step, at least when considering only the evolution of the model and maintenance of the structural properties (the behavioural properties will be discussed in the next section and resulted in much more big changes in the written code).

9 Behavioural Properties Verification

The properties' verification was made by making use of the checks mentioned in the section 6 and verifying whether no counterexamples were found. During our tests, a lot of properties didn't hold at first, but after understanding why they didn't work we got all the checks implemented working, the reasons for a counterexample to be found were:

- We had to define the priority of some requirements in relation to others when they would conflict otherwise;
- Our implementation needed to be reformulated in the right way to follow the requirements.

With time, as the system became increasingly more complex, the checking times also increased, that's why, in the final iteration, we abstracted the three cycles of the tip-blinking function, that would otherwise add so much complexity to the system (in the form of a sequence of six nested `eventually` predicates) that the model checker could even take minutes to verify all the property checks.