Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA

Paolo Arcaini[1], Silvia Bonfanti[2(✉)], Angelo Gargantini[2], Elvinia Riccobene[3], and Patrizia Scandurra[2]

[1] National Institute of Informatics, Tokyo, Japan
arcaini@nii.ac.jp
[2] Department of Economics and Technology Management, Information Technology and Production, Università degli Studi di Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini,patrizia.scandurra}@unibg.it
[3] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. In the context of automotive domain, modern control systems are software-intensive and have adaptive features to provide safety and comfort. These software-based features demand software engineering approaches and formal methods that are able to guarantee correct operation, since malfunctions may cause harm/damage. Adaptive Exterior Light and the Speed Control Systems are examples of software-intensive systems that equip modern cars. We have used the Abstract State Machines to model the behaviour of both control systems. Each model has been developed through model refinement, following the incremental way in which functional requirements are given. We used the ASMETA tool-set to support the simulation of the abstract models, their validation against the informal requirements, and the verification of behavioural properties. In this paper, we discuss our modelling, validation and verification strategies, and the results (in terms of features addressed and not) of our activities. In particular, we provide insights on how we addressed the adaptive features (the adaptive high beam headlights and the adaptive cruise control) by explicitly modelling their software control loops according to the MAPE-K (Monitor-Analyse-Plan-Execute over a shared Knowledge) reference control model for self-adaptive systems.

1 Introduction

Modern control systems, like those in the automotive domain, are software-intensive, have adaptive features, and must be reliable. Formal methods can be applied in order to improve their development and guarantee their correct operational behaviour and safety assurance. In this paper, we report our experience in applying the Abstract State Machine (ASM) formal method to the Adaptive Exterior Light (ELS) and the Speed Control Systems (SCS), which are examples of software-intensive systems that equip modern cars. We used the ASMETA framework, which provides a wide tool support to

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 302–317, 2020.
https://doi.org/10.1007/978-3-030-48077-6_25

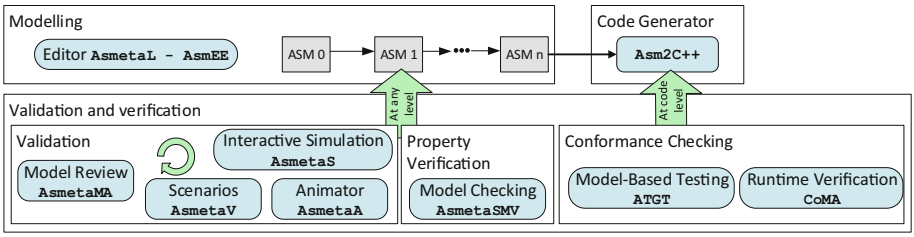

Fig. 1. ASM-based development process

ASMs. ASMETA tries to combine the formality of ASMs with a tools set that supports the editing, simulation and animation of the abstract models, their validation against the informal requirements, and the verification of behavioural properties. Moreover, ASMETA adopts a refinement-based process, and employs several constructs in order to allow modularity.

To address the adaptive features of the case study, i.e., the adaptive high beam headlights and the adaptive cruise control, we exploited the concept of self-adaptive ASMs [5], which allows modelling adaptation logics in terms of MAPE-K (Monitor-Analyse-Plan-Execute over a shared Knowledge) feedback control loops.

The paper is structured as suggested by the *call for paper* of the case study. The following subsection briefly presents the ASM formal method and its supporting tool-set ASMETA. Section 2 explains our modelling strategy. Details about our models and how they capture the requirements are provided in Sect. 3. We have applied several *validation* and *verification* (V&V) activities that are presented in Sect. 4. Section 5 discusses some observations that we draw from this experience, together with some limits of our approach. Section 6 concludes the paper.

1.1 The ASM Method and the ASMETA Tool-Set

Basic Definition. ASMs [10,11] are an extension of Finite State Machines (FSMs) where unstructured control states are replaced by *states* comprising arbitrary complex data (i.e., domains of objects with functions defined on them), and *transitions* are expressed by transition rules describing how the data (state function values saved into *locations*) change from one state to the next. ASM models can be read as "pseudocode over abstract data" which comes with a *well defined semantics*: at each computation step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations.

Modelling Process and Tools. ASMs allow an iterative design process, shown in Fig. 1, based on model refinement. Tools supporting the process are part of the ASMETA (ASM mETAmodeling) framework[1] [4]. Requirements modelling starts by developing a high-level model called *ground model* (ASM 0 in Fig. 1). It is specified by reasoning on the informal requirements (generally given as a text in natural language) and using terms of the application domain, possibly with the involvement of all stakeholders. The ground model should *correctly* reflect the intended requirements and should

[1] http://asmeta.sourceforge.net/.

be *consistent*, i.e., without possible ambiguities of initial requirements. It does not need to be *complete*, i.e., it may not specify some given requirements. The ground model and the other ASM models can be edited in `AsmEE` by using the concrete syntax `AsmetaL` [14]. Starting from the ground model, through a sequence of *refined* models, further functional requirements can be specified until a complete model of the system is obtained. The refinement process allows to tackle the system complexity, and to bridge, in a seamless manner, specification to code. At each refinement level, already at the level of the ground model, different V&V activities can be applied. In Sect. 4, we explain in detail how model validation and property verification are performed in ASMETA tools.

Model to code transformation are supported for C++ code [9], and *conformance checking* is possible to check if the implementation, if externally provided, conforms to its specification. The tool ATGT [13] can be used to automatically generate tests from ASM models and, therefore, to check the conformance offline; CoMA [3], instead, can be used to perform runtime verification, i.e., to check the conformance online.

1.2 Distinctive Features of the Modelling Approach

Machine and Modules. As better explained in Sect. 2, in our modelling activity we strongly make use of *modularization*. To concretely support it, we exploited the concept of *ASM module* as introduced in [11] and provided by the language of the ASMETA tool set. Specifically, an *ASM module* contains the declaration and definitions of domains, functions, invariants, and rules; an ASM *machine* is an ASM module that additionally contains a (unique) *main rule* representing the starting point of the machine execution, and an *initial state*. The keyword `asm` introduces the main ASM while the keyword `module` indicates a module.

Self-adaptive ASMs. To model the adaptive features of the two automotive subsystems, we exploit the concept of MAPE-K (a sequence of four computations Monitor, Analyze, Plan, and Execute over a shared Knowledge) feedback control loop [16] commonly used to structure the adaptation logic of self-adaptive software systems. To this end, we adopt *self-adaptive Abstract State Machines* (self-adaptive ASMs) as defined in [5] to formalize the sequential execution of the four MAPE computations of a MAPE-K loop in terms of ASM transitions rules (see Sect. 3.1, *CarSystem003*).

2 Modelling Strategy

We here describe the general strategy adopted while modelling the ELS and the SCS using ASMs. We explain how our model is structured, how the structure relates to the requirements, the model purpose (in terms of properties addressed and not addressed by our solution), and our formalization approach.

Model Structure. The ELS and SCS sub-systems are loosely coupled (i.e., they work in parallel and share some external signals), so we handle their requirements independently. More specifically, for each subsystem, we developed an ASM specification through a sequence of refined models, following the incremental way in which functional requirements of the software based controllers are described in the requirements

document [15]. The models are numbered from 1 to 9: models from CarSystem001 to CarSystem004 refer to the ELS, while the SCS is modelled from CarSystem005 to CarSystem007. CarSystem008 merges the two systems and CarSystem009 introduces the faults handling and general properties.

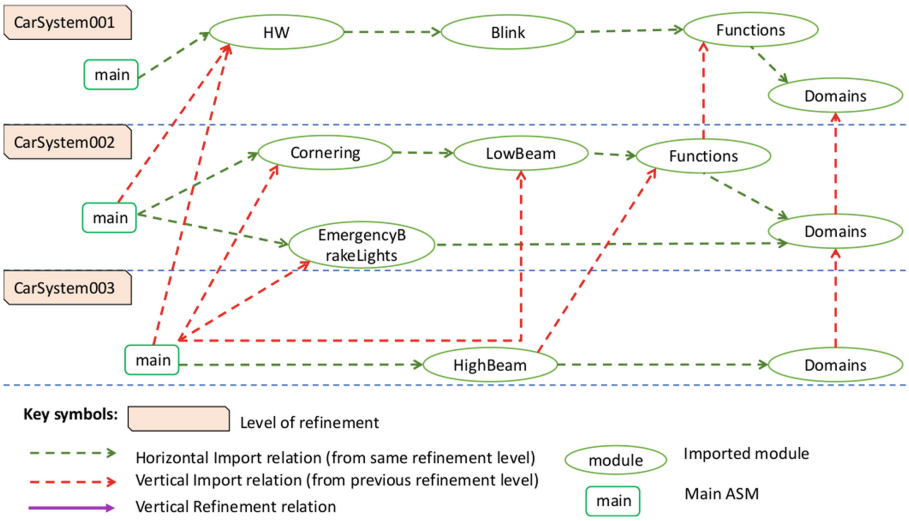

Fig. 2. ASM structure of the first 3 levels

Since, at a given refinement level, only some parts of the model were refined, to keep track of maintained and refined parts and to relate abstract and refined sub-models, we structure an ASM model in *modules*. Therefore, at each level, an ASM results in the horizontal and vertical composition of ASM modules (by exploiting the `import` module feature). For example, Fig. 2 illustrates the ASM models structure in terms of horizontal and vertical imports for the first three refined levels of the *CarSystem* related to the ELS subsystem. Level 1 consists of an ASM *CarSystem001main* that imports module *CarSystem001HW* that imports module *CarSystem001Blink*, till the final module *CarSystem001Domains* is imported[2]. At this level, module relations are all horizontal imports. Similarly, ASM *CarSystem002main* imports (horizontally) module *CarSystem002Cornering* from the same refinement level, and it imports (vertically) module *CarSystem001HW* from the previous refinement level. Note that module *CarSystem002Domains* imports module *CarSystem001Domains*, since the former enlarges (as expected during refinement) the latter. Vertical relations can be also module refinement relations. For example, Fig. 3 focuses on level 4 and illustrates the ASM *CarSystem004main* that imports, among the others, the module *CarSystem004HW* refining module *CarSystem001HW* from level 1, and module *CarSystem004Cornering* refining module *CarSystem002Cornering* from level 2. Other depicted import/refine-

[2] Note that common functions and common domains are declared and defined into specific modules (*CarSystem00XFunctions* and *CarSystem00XDomains*) which are imported by others.

ment relations are self-explanatory. More details on the refinement levels and corresponding models are given in Sect. 3.

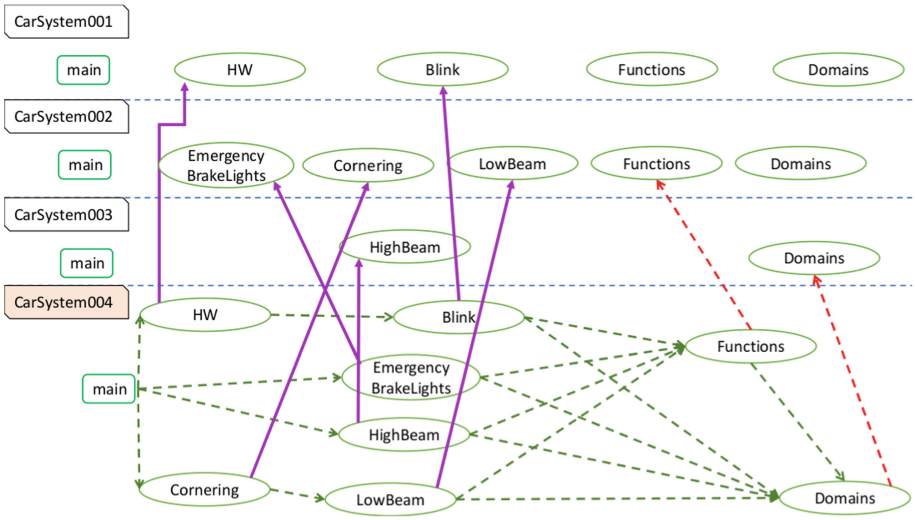

Fig. 3. ASM structure of level 4

To model the overall behaviour resulting from the union of the behaviours of all distributed software we simply exploit the notion of *parallel ASMs* [7]. Therefore, we model software running on ECUs as parallel algorithms working in sequential global time. Whenever necessary (as stated in the requirements document) and to avoid interference, the machine follows only one or a restricted subset of all possible parallel execution paths by using the *mutual exclusive guards pattern* at the level of rules, i.e., the simultaneous activation of certain rules is avoided by modelling them as conditional rules with mutual exclusive guards.

Model Purpose. The proposed ASM model is primarily tailored to the formalization and analysis of functional aspects of the ELS and SCS subsystems in order to provide guarantees of their operational correctness. Modelling and validation activities revealed, however, some statements where the requirements were wrong or unambiguous. We had the possibility to check our doubts with the chairs (working as domain experts) and we got corrected version of the requirements. Further details on these missing/ambiguous aspects are given in Sect. 5.

Except for some features not addressed by our solution (see below), our model(s) capture(s) all requirements described in the document [15]. Each refined model was analyzed using different techniques (see Sect. 4), considering also the description of the operational scenarios that are given as annex part of the requirements. The most important properties addressed by our solution are:

– On the ELS subsystem: *a)* The priority of hazard warning over blinking is guaranteed; *b)* Low beam headlights are turned on and off as required; *c)* Priority of ELS-19

over other requirements has been addressed when the ambient light is activated; *d)* High beam headlights are automatically turned on/off when the light rotary switch is set to Auto; *e)* When subvoltage/overvoltage occurs the system reacts as required.
– On the SCS subsystem: *a)* (Adaptive) cruise control desired speed is set as required: the adaptive cruise control sets automatically the speed to reach the target based on different factors like the current speed and the speed of the vehicle ahead. *b)* Emergency brake intervenes to avoid collisions; *c)* Speed limit and traffic sign detection set the threshold speed when they are activated by the user.

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. In any case, the low beam headlights remain active at least for 3 seconds.

```
macro rule r_LowBeamHeadlights =
  ...
  if (lightRotarySwitch = AUTO and engineOn(keyState) and
                                   brightnessSensor<200) then
    if ((not lowBeamLightingOn)) then r_LowBeamTailLampOnOff[100]
    endif
  endif
  if (lightRotarySwitch = AUTO and engineOn(keyState) and
                                   brightnessSensor>250) then
    if (lowBeamLightingOn and passed3Sec) then
      r_LowBeamTailLampOnOff[0]
    endif
  endif
```

Code 1. Translating text into rules

Not Addressed Features. The main feature not addressed by our solution is the time management. We are not able to deal with continuous time, although a notion of reactive timed ASMs has been proposed [17], there is no tool support for it. To overcome this limitation, we assume that a monitored function notifies the system whether an interval time is passed. For example, requirement ELS-18 states that low beam headlights remain active at least for 3 s. As shown in Code 1, we have introduced a monitored function `passed3Sec` that notifies if 3 s had elapsed since the low beam headlights received the command to be turned off. A further not addressed feature is the frequency of blinking. Since the model does not support continuous time it is not possible to set the direction lamps state (ON or OFF) every second (the duration of a flashing cycle is 1 s). Due to this limitation, we have introduced an enumerative value that indicates the current pulse ratio and we have supposed that the Head-Unit sets the state of direction lamp given the pulse ratio value.

Requirements Formalization Approach. To understand and specify the behaviour, we started from the textual description of the systems, and we tried to express the text in terms of transition rules, by supplying the necessary definitions of domains and functions. In naming functions, domains and transition rules, we have used an domain-specific terminology that can be understood by the stakeholders. Furthermore, for the functions defined in the tables at the end of the document of specification we have used the names proposed. An example of requirement formalization is shown in Code 1 that reports the requirement ELS–18 and the corresponding ASM rule. Sometimes the requirements are not independent of each other, for this reason more than one requirement is modelled by one rule.

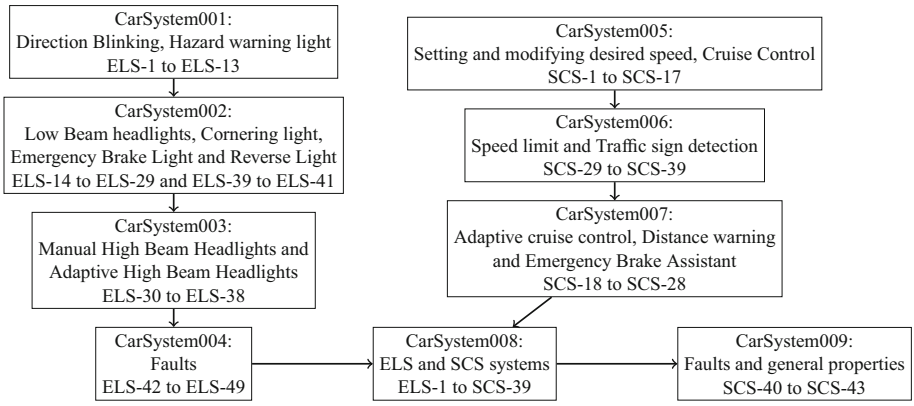

Fig. 4. Chain of refined models and captured requirements

3 Model Details

In this section, we present the result of our modelling of the Adaptive Exterior Light and Speed Control systems[3].

We have proceeded through refinement of the two systems independently and finally we have merged them together to obtain the complete system. Figure 4 shows the model refinement chain and lists the requirements introduced in each model.

Table 1. Models dimension

	Functions				Rules	
	Monitored	Controlled	Derived	Static	n rules declarations	n rules
CarSystem001	4	12	6	1	14	103
CarSystem002	14	26	14	4	26	218
CarSystem003	18	31	24	5	33	244
CarSystem004	19	31	31	5	33	252
CarSystem005	8	4	1		10	78
CarSystem006	12	8	2		14	125
CarSystem007	21	17	6		24	183
CarSystem008	36	46	36	5	56	433
CarSystem009	36	46	37	5	56	433

Table 1 shows the model dimensions in terms of number of functions and rules and the traceability table between requirements and rules/functions (see Table 2). In the sequel, we will show some parts of the models for each subsystem and then we will explain the merging of two systems.

[3] Artifacts are at https://github.com/fmselab/ABZ2020CaseStudyInAsmeta.

Table 2. Traceability table between requirements and rules/functions

	CS001	CS002	CS003	CS004	CS005	CS006	CS007	CS008	CS009
Exterior Light System									
Direction Blinking									
ELS-1 to ELS-7									
Hazard Warning									
ELS-8 to ELS-13									
Low Beam Headlights Cornering Light									
ELS-14 to ELS-29									
Manual Hight Beam Headlights									
ELSL-30 to ELS-31									
Adaptive High Beam Headlights									
ELS-32 to ELS-38									
Emergency Brake Light									
ELS-39 to ELS-40									
Reverse Light									
ELS-41									
Fault Handling									
ELS-42 to ELS-49									
Speed Control System									
Setting and modifying desired speed									
SCS-1 to SCS-12									
Cruise Control									
SCS-13 to SCS-17									
Adaptive Cruise Control									
SCS-18 to SCS-24									
Distance Warning									
SCS-25 to SCS-26									
Emergency Brake Assistant									
SCS-27 to SCS-28									
Speed Limit									
SCS-29 to SCS-35									
Traffic Sign Detection									
SCS-36 to SCS-39									
Fault Handling and General Properties									
SCS-40 to SCS-43									

3.1 Adaptive Exterior Light System

ELS has been modelled into 4 refinement steps which are explained below.

CarSystem001. This model describes the functions of direction blinking and hazard warning. The critical features found in this modelling phase are the following: *a)* the hazard warning has the priority over direction blinking (ELS-3); *b)* the tail lamp is used as an indicator for cars sold in USA and Canada (ELS-6). Code 2 shows that if hazard warning request is activated and any kind of blinking is running, blinking must be stopped and hazard warning is started. In case of direction blinking, the value of pitman arm is stored to restart the request as soon as hazard warning is deactivated; if the pitman arm is moved back to neutral position, the request is cancelled. The tail lamp status is updated to FIX or BLINK. It is BLINK only if blinking is active and car is sold in USA or Canada, otherwise it is FIX. Moreover, the value of light is dimmed by 50% during blinking (see Code 3). To address the requests from the pitman arm, we have defined three functions, *pitmanArmUpDown* for the incoming request, *pitmanArmUpDown_RunnReq* for the running request and *pitmanArmUpDown_Buff* to save the incoming request if it cannot be satisfied in the current state. When the run-

ning request has been processed, the request in the buffer is executed unless a new one arrives.

```
//Hazard Warning
macro rule r_HazardWarningLight =
  if (hazardWarningSwitchOn_Runn) then
    ...
  else // If HW is not running
    ...

  if (hazardWarningSwitchOn) then
    par
      if (pitmanArmUpDown_RunnReq !=
                                NEUTRAL_UD) then
        r_InterruptBlinking[]
      endif
      hazardWarningSwitchOn_Start := true
      //If request from pitman arm UpDown arrives
      // save it into the buffer
      //if UPWARD7 or DOWNWARD 7
      r_SavePitmanArmUpDownReq[]
    endpar
  else
    r_DirectionBlinking[]
  endif
  ...

  endif
```

Code 2. Hazard warning has priority over direction blinking

```
  //True if the car is sold in USA or CANADA
function tailLampAsIndicator = (marketCode = USA or
                                marketCode = CANADA)

//Set tail lamp status
macro rule r_setTailLampLeft ($value in
LightPercentage , $status in TailLampStatus) =
  par
    tailLampLeftStatus := $status
    tailLampLeftBlinkValue := $value
  endpar

macro rule r_BlinkLeft ($value in LightPercentage,
                        $pulse in PulseRatio) =
  par
    blinkLeft := $value
    blinkLeftPulseRatio := $pulse
    //ELS−6
    if (tailLampAsIndicator) then
      if ($value = 0) then
        r_setTailLampLeft[0,FIX]
      else
        r_setTailLampLeft[50,BLINK]
      endif
    endif
  endpar
```

Code 3. Tail lamp as indicator during direction blinking and hazard warning

CarSystem002. This introduces the low beam headlights and cornering light, emergency brake light, and reverse light functions. Each of these functions is modelled in an ASM module. Common functions and domains are extended starting from those defined in the *CarSystem001*, while hazard warning and direction blinking are unchanged. Requirements ELS-15, ELS-16, ELS-17, and ELS-19 are interconnected because ELS-19 has the priority over the others if ambient light activated. We have defined a guard called *ambientLightingAvailable* which is true if ambient lighting is activated, the vehicle is not armoured and darkness mode is switched off. Some requirements state that the system performs an action if function X changes its value from state s to state $s + 1$. To detect the value change, we store the value of function X in the previous state (*X_Previous*) and we compare that value with the current value. When the model detects a value change, the system acts as defined by rules. An example is the requirement ELS-19: the low beam headlamps are activated if engine has been stopped. We have captured the key state mutation by checking the value of *keyState_Previous* compared to *keyState*.

CarSystem003. This step introduces the control features for the manual and adaptive high beam headlights (ELS-30 to ELS-38). We first modelled the manual control of the high beam headlights and then, in the same refinement level, the adaptive one. The control variables are the high beam luminous strength (a percentage) and the illumination distance (expressed in meters).

In manual mode (ELS30-ELS31), the user can set a fixed illumination area of 220 m and 100% of luminous strength, or activate the high beam headlights temporary (so-called *flasher*). We had to make the following assumptions due to missing requirements: (i) a maximum illumination area of 360 m and 100% of luminous strength in the flasher mode; (ii) the key is inserted or the engine is on to activate high beam in a fixed way.

```
macro rule r_MAPE_HBH =
  par
    r_Monitor_Analyze_HBH[]
    if adaptiveHighBeamDeactivated then
      highBeamOn := false
    endif
  endpar

macro rule r_Monitor_Analyze_HBH =
  if adaptiveHighBeamActivated then
    par
      if drivesFasterThan(currentSpeed,300)
            and not oncomingTraffic then
        r_IncreasingPlan_HBH[]
      endif
      if oncomingTraffic then
        r_DecreasingPlan_HBH[]
      endif
    endpar
  endif
```

```
macro rule r_Execute_HBH ($setHighBeam in Boolean,
$setHighBeamMotor in HighBeamMotor,
$setHighBeamRange in HighBeamRange) =
r_set_high_beam_headlights[$setHighBeam,
            $setHighBeamMotor,$setHighBeamRange]

macro rule r_IncreasingPlan_HBH =
  let ($d = lightIlluminationDistance(calculateSpeed),
    $l = luminousStrength(calculateSpeed)) in
            r_Execute_HBH[true,$d,$l] endlet

macro rule r_DecreasingPlan_HBH = r_Execute_HBH[true,30,0]

macro rule r_set_high_beam_headlights($v in Boolean,
$d in HighBeamMotor, $l in HighBeamRange) =
  par
    highBeamOn := $v
    highBeamMotor := $d
    highBeamRange := $l
  endpar
```

Code 4. MAPE loop may start and stop the adaptive high beam headlight

In adaptive mode (ELS32-ELS38), the illumination of the road is automated depending on the incoming vehicles (as detected by a built-in camera) and optimized to illuminate the appropriate area according to the vehicle speed, the last being the current speed of the vehicle or the target speed provided by the advanced cruise control in case it is activated. The illumination distance and the luminous strength are adjusted according to characteristic curves provided in the requirements document. In order to calculate such values, we had to reverse engineered the formulas as suggested in the additional information provided with the specification document. We modelled this adaptive behaviour in terms of a MAPE-K feedback control loop that starts with the rule *r_MAPE_HBH* (see Code 4). A control loop in self-adaptive systems is a sequence of four computations: Monitor-Analyse-Plan-Execute (MAPE) over a knowledge base. In self-adaptive ASMs [5], it is modelled by means of four rules, one per MAPE computation, while the knowledge is modelled by means of functions, since in ASMs system memory is represented in terms of functions. In our case, the MAPE loop consists of the following rules invoked in a waterfall manner within one single ASM-step machine (see Code 4): *r_Monitor_Analyze_HBH*, where monitor and analyze computations are modelled as a unique activity; *r_IncreasingPlan_HBH* and *r_DecreasingPlan_HBH* to plan the adaptation if necessary: light illumination distance and luminous strength are increased or decreased according to the vehicle speed; *r_Execute_HBH* to set the values as planned: *highBeamOn* to activate/deactivate the high beam, *highBeamRange* and *highBeamMotor* for the high beam luminous strength and illumination distance.

```
function subVoltage = (currentVoltage < 85)

macro rule r_CorneringLights =
//ELS−46 cornering lights on if no subvoltage
  if (not subVoltage) then
  //Turn on cornering light
    ...
  else
  //ELS−46 cornering lights off if subvoltage
  if (corneringLightRight != 0 or
    corneringLightLeft != 0) then
    r_CorneringLightsOff[]
  endif
endif
```

Code 5. Subvoltage handling

```
function overVoltage =  (currentVoltage > 145)

function = (100−(currentVoltage−145)∗20)

function setOverVoltageValueLight($value in Integer) =
if (overVoltage and $value > overVoltageMaxValueLight) then
  overVoltageMaxValueLight else $value endif

macro rule r_setTailLampLeft ($value in
LightPercentage , $status in TailLampStatus) =
par
  tailLampLeftStatus := $status
  tailLampLeftBlinkValue := setOverVoltageValueLight($value)
endpar
```

Code 6. Overvoltage handling

CarSystem004. This modelling phase introduces fault handling, in particular how the software reacts to overvoltage or subvoltage. When subvoltage is present, some functionalities like cornering light and parking light are not available. This has been addressed by adding a guard that, in case of subvoltage (the voltage value is less than 8.5 V), disables them (see Code 5). In case the voltage is more than 14.5 V, the system is in overvoltage. The maximum value of lights is computed by the *setOvervoltage-ValueLight* function: it returns the minimum between current light value and the value calculated by *overVoltageMaxValueLight* function. In this step of refinement, we have refined the modules whose behaviour is affected by the voltage value.

3.2 Speed Control System

The final model of Speed Control System has been addressed through three steps of refinement explained in the following.

CarSystem005. In the first model of SCS, we have implemented the functionalities of cruise control and setting and modifying desired speed (see *CarSystem005Desired-SpeedCruiseC* module). Desired speed and target speed are modified based on SCS lever position when (adaptive) cruise control is activated.

CarSystem006. This step of refinement introduces the speed limit and traffic sign detection functionalities. If traffic sign detection is on, the target speed is modified by the recognized traffic sign value. The speed limit modifies the desired speed which must not be exceeded by the current speed.

CarSystem007. This step of refinement introduces the adaptive cruise control and distance warning from the vehicle ahead (from SCS-18 to SCS-26), and the brake assistant (from SCS-27 to SCS-28) to initiate braking in critical situations. Similarly to the adaptive high beam headlights (see refinement CarSystem003), we modelled the adaptive behaviour of the cruise control in terms of a MAPE-K feedback control loop that monitors the distance from the vehicle ahead, plans and executes acceleration/deceleration automatically, including braking until a full standstill and starting from a standstill.

3.3 Merging ELS and SCS Models

Once we have developed the ELS and SCS separately, we have merged them to obtain a model that includes both systems.

CarSystem008. This step of refinement has been obtained easier than what we expected due to the modularity followed in the previous steps of refinement. Once the main module (*CarSystem008main*) has been defined, we have simply imported the modules previously developed. All rules are executed in parallel and no inconsistent update has been found because the systems are independent of each other, they have only common inputs. A schema that shows how we have imported modules is available on-line.

CarSystem009. We have introduced the requirements from SCS-40 to SCS-43. The dangerous situations in SCS-40, SCS-41 and SCS-42 are already managed by the model; in this step of refinement we have integrated SCS-43 by refining the guards of CarSystem004EmergencyBrakeLights module. The brake lights are activated either by the brake pedal pressed by the user or the system activates the emergency brake.

4 Validation and Verification

Validation and verification are supported by a set of ASMETA tools. In this section, we report results and tools used for each activity, and explain the changes to the models that resulted from the validation and the verification.

Validation. Model validation helps to ensure that the specification really reflects the intended requirements, and to detect faults and inconsistencies as early as possible with limited effort. While writing models, we have started the validation activity by using the animator `AsmetaA` [8] which uses tables to convey information about states and their evolution. We have performed *interactive animation* that consists in providing inputs (i.e., values of monitored functions) to the machine and observing the computed state. The animator, at each step, performs *consistent updates checking* to check that all the updates are consistent (in an ASM, two updates are inconsistent if they update the same location to two different values at the same time), and invariant checking.

With the increasing complexity of the ELS and SCS system models, we have intensely used the scenario-based validation `AsmetaV` [12] that allows to build and execute *scenarios* of expected system behaviours. In scenario-based validation, the designer provides a set of scenarios specifying the expected behaviour of the models (using the textual notation `Avalla` [12]). These scenarios are used for validation by instrumenting the simulator `AsmetaS` [14]. During simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict. `Avalla` provides constructs to express execution scenarios in an algorithmic way, as interaction sequences consisting of actions committed by the user to set the environment (i.e., the values of monitored/shared functions), to check the machine state, to ask for the `execution` of certain transition rules, and to enforce the machine itself to make one `step` as reaction of the user actions. Code 7 shows an example of scenario: it specifies the behaviour of the second validation sequence for the exterior light provided as part of the documentation[4]. More scenarios are available on our online repository, including all those provided with the case study document.

[4] See the *ValidationSequences_v1.8.xlsx* document in the web site of the case study.

```
//Light switch to AUTO -> lights remain off
set lightRotarySwitch := AUTO;
set keyState := NOKEYINSERTED;
set hazardWarningSwitchOn := false;
set darknessModeSwitchOn := false;
set brightnessSensor := 100;
set pitmanArmUpDown := NEUTRAL_UD;
set reverseGear := false;
set brakePedal := 0;
step
//Ignition to ON position -> no effect on light
check tailLampLeftBlinkValue = 0;
check tailLampLeftFixValue = 0;
check tailLampRightBlinkValue = 0;
check tailLampRightFixValue = 0;
//Cornering lights off
check corneringLightRight = 0;
check corneringLightLeft = 0;
//Low beam headlight
check lowBeamLeft = 0;
check lowBeamRight = 0;
check reverseLight = 0;
check brakeLampLeft = 0;
check brakeLampRight = 0;
check brakeLampCenter = 0;
```

```
//Turn the light rotary switch ON
set lightRotarySwitch := ON;
step
//Ignition to KeyInserted -> lights reimain off
set keyState := KEYINSERTED;
step
//Engine start -> light on with 100%
set keyState := KEYINIGNITIONONPOSITION;
step
check tailLampLeftBlinkValue = 0;
check tailLampLeftFixValue = 100;
check tailLampRightBlinkValue = 0;
check tailLampRightFixValue = 100;
//Cornering lights off
check corneringLightRight = 0;
check corneringLightLeft = 0;
//Low beam headlight
check lowBeamLeft = 100;
check lowBeamRight = 100;
check reverseLight = 0;
check brakeLampLeft = 0;
check brakeLampRight = 0;
check brakeLampCenter = 0;

.
```

Code 7. Scenario for normal light, no daytime light, ambient light, night

Although interactive and scenario-based simulations are very useful to get a fast understanding of the developed models and quickly detect possible modelling errors, they do not allow to perform an exhaustive check. Therefore, we performed *model review* using the `AsmetaMA` tool [2], as a complementary validation technique: it is a form of static analysis to determine if a model has sufficient *quality* attributes (as minimality, completeness, consistency). This automatic activity can find problems that could pass undetected during interactive simulation and scenario validation, which cannot be exhaustive and perform only some system executions. For example, CarSystem003 has a rule that decides when to switch on the parking lights (`parkingLightON := true`). By introducing requirement ELS-46, however, we added a rule to switch the lights off in case of subvoltage (`parkingLightON := false`). In the first version of the CarSystem004, the two rules sometimes conflicted, so leading to an inconsistent update. Model review allowed us to spot this problem, that we solved by introducing a guard to avoid the conflict.

Verification. Formal verification of ASMs is possible by means of the tool `AsmetaSMV` [1], and both *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas are supported. To perform model checking with NuSMV (the model checker `AsmetaSMV` is built on) that requires a finite state space, we reduce infinite domains in the original models to finite domains.

For the case study, some properties can be naturally derived from the requirements. The typical form of a requirement is "when/if ... then ..." describing that when something happens (a given external input received, a given state condition, etc.), some actions must be taken. Such kind of properties are naturally translated in temporal properties as $\square(\phi \rightarrow \bigcirc(\psi))$ or $\square(\phi \rightarrow \Diamond(\psi))$. However, these kinds of properties (in particular the first one) are also reflected in the structure of ASM rules derived from the requirements, that take the form of **if** ... **then** ... **else** ... **endif** rules. In this case, tempo-

ral properties assume the form of *redundant specifications* that can be used to enforce the model and make it robust against possible wrong future modifications of the model. An example of such kind of property that we developed related to requirement ELS-18 is the CTL property: **ag**((lightRotarySwitch = AUTO and engineOn and brightnessSensor $<$ 200)implies **ax**(lowBeamLightingOn)).

In addition to these straightforward properties, we specified more general properties that are not directly related to single requirements. For example, we specified the following three properties:

- both direction indicators blink iff the car is in hazard warning:
  ag((blinkLeft != 0 and blinkLeftPulseRatio != NOPULSE and blinkRight != 0 and blinkRightPulseRatio != NOPULSE)= hazardWarningSwitchOn_Runn)
- if tail lamps are blinking, the car is not European:
  ag((tailLampLeftStatus=BLINK or tailLampRightStatus=BLINK)implies marketCode!=EU)
- the market code of a car cannot be changed:
  forall $c **in** MarketCode **with** (marketCode = $c implies **ag**(marketCode = $c))

5 Discussion

We here provide a more detailed discussion about our experience in modelling and analysing the case that took totally less than one month (around one week for understanding the requirements, and the remaining time for the model development and V&V). We report flaws we discovered in the requirements documents, features that we would have expected in the documents, and also missing functionalities of our framework that would have been helpful (besides the temporal aspects discussed in Sect. 2).

Scenarios. The scenarios provided with the case study requirements turned out to be very useful as, in some cases, allowed us to clarify some misunderstanding we had when we developed the models starting from the requirements document. For example, observing the scenarios, we realized that `desiredSpeed` and `targetSpeed` are two separate entities that can assume different values with two updating policies, while, only reading the requirements document, we thought that the user could modify both at the same time. However, in other cases, the description of the scenarios and the description of the requirements were inconsistent. For some of such inconsistencies, it was clear that the document to trust was the scenario description; for example, the requirements document uses in an improper way the terms *desired*, *target*, and *set* speed, sometimes using them interchangeably; the scenarios document, instead, clearly distinguishes them and allows to observe their different roles. Other inconsistencies, instead, were less easy to disambiguate. This was the case of *traffic sign detection* for which the requirements document states that only the target speed is modified when the sign is recognized; however, scenario 6 of *Validation Sequences Speed* shows a case in which the car, when detects the sign, modifies both desired and target speed.

Requirements Coverage. Since the scenarios turned out to be so useful, we would have liked to have a more exhaustive validation sequences in the informal documentation in order to build a set of scenarios *covering* all the requirements; using the coverage feature of our validator `AsmetaV`, we realized that this is not the case. For example,

requirements ELS-42 to ELS-47 are not covered by any validation sequence and therefore by any of our scenario. Moreover, when doing this coverage checking, we realized a limit of our coverage tool that can only provide coverage information at the level of macro rule (similarly to call coverage). This coarse grained level of coverage may be not informative enough in situations in which requirements are mapped at the level of, e.g., branches of conditional rules. As future work, we plan to extend our coverage evaluator to provide information as decision and condition coverage.

Scenario Derivation and Animation. As said previously, we have extensively used scenario-based validation during the modelling activity. We realized that a better integration with model animation [8] would permit to save animation sessions in terms of scenarios, and also to animate existing scenarios. For this work, we have developed the former technique that allows us to export into Avalla scripts the animations we perform, and re-execute them later when changing the model (in a kind of regression testing using "record and replay").

Parametric ASMs. The systems described in the case study actually represent a family of systems that can be configured on the base of the market and type of car. Different configurations lead to different behaviours of the systems. In order to model all these features, we had to introduce *flags* to be set in the initial state: this reduces the readability and maintainability of the models. It would be useful to have a *parametric version* of the ASM model, similarly to what done for software programs using Software Product Lines. A recent approach has been proposed in this context for ASM [6], and we plan to consider it in future usages.

Implementation. In modelling the case study, we did not consider any implementation. However, the ASMETA framework provides support in this sense. First of all, a translator to C++ [9] is available; it could be applied to generate a first prototype of the implementation, which could be then further extended by developers. Instead, if a system implementation is available, *conformance checking* approaches can be applied, in terms of *model-based testing* and *runtime verification* both supported by ASMETA.

6 Conclusions

We have presented the specification, validation, and property verification of an automotive system by using ASMs. We have discussed our experience in modelling an adaptive exterior light and the speed control systems that equip modern cars, also addressing the adaptive features of the two systems in terms of MAPE-K feedback control loops. We have also shown how to write and run scenarios and verify properties addressed by our models. We have found some misunderstanding in the document of requirements, because the described behaviour was different from the behaviour expected in validation sequences. We have found some limitations in our tools, e.g., the coverage evaluator provides only coverage in terms of macro rules, that we plan to overcome with future improvements. On the other hand, this case study has provided us the opportunity to test our framework in terms of robustness and user experience in modelling complex systems. We have noticed that the framework is particularly suitable to handle the increasing complexity of the models: the support for modularization (at the level

of modelling and scenario construction) allows producing refined model and refined scenarios with limited effort.

References

1. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 61–74. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6
2. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of abstract state machines by meta property verification. In: Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), pp. 4–13. NASA (2010)
3. Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: conformance monitoring of Java programs by abstract state machines. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 223–238. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_17
4. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Softw. Pract. Exp. **41**, 155–166 (2011)
5. Arcaini, P., Riccobene, E., Scandurra, P.: Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. **11**(4), 25:1–25:35 (2017)
6. Benduhn, F., Thüm, T., Schaefer, I., Saake, G.: Modularization of refinement steps for agile formal methods. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 19–35. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_2
7. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Log. **4**, 578–651 (2003)
8. Bonfanti, S., Gargantini, A., Mashkoor, A.: AsmetaA: animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 369–373. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_25
9. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from abstract state machines specifications. J. Softw. Evol. Proc. **32**(2), e2205 (2020)
10. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
11. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
12. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 71–84. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_7
13. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from ASM specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_15
14. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. UCS **14**(12), 1949–1983 (2008)
15. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system (2019)
16. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
17. Slissenko, A., Vasilyev, P.: Simulation of timed abstract state machines with predicate logic model-checking. J. Univers. Comput. Sci. **14**(12), 1984–2006 (2008)