International Journal on Software Tools for Technology Transfer (2024) 26:379–401
https://doi.org/10.1007/s10009-024-00751-4

GENERAL

Special Section: ABZ 2020/2021

A journey with ASMETA from requirements to code: application to an automotive system with adaptive features

Paolo Arcaini[1] · Silvia Bonfanti[2] · Angelo Gargantini[2] · Elvinia Riccobene[3] · Patrizia Scandurra[2]

Accepted: 29 April 2024

Abstract

Modern automotive systems with adaptive control features require rigorous analysis to guarantee correct operation. We report our experience in modeling the automotive case study from the ABZ2020 conference using the ASMETA toolset, based on the Abstract State Machine formal method. We adopted a seamless system engineering method: from an incremental formal specification of high-level requirements to increasingly refined ASMETA models, to the C++ code generation from the model. Along this process, different validation and verification activities were performed. We explored modeling styles and idioms to face the modeling complexity and ensure that the ASMETA models can best capture and reflect specific behavioral patterns. Through this realistic automotive case study, we evaluated the applicability and usability of our formal modeling approach.

Keywords Automotive system modeling · Abstract state machines · ASMETA · Self-adaptation · Code generation

1 Introduction

Nowadays, automotive systems employed in modern cars rely on adaptive software control features for providing more safety and comfort. These software-intensive systems require rigorous software engineering processes based on formal methods to guarantee their correct operation, since malfunctions may be harmful to humans and the environment.

In this paper, we report our experience in applying ASMETA [3, 15], a toolset based on the Abstract State Machine [26, 28] formal method (ASMs in brief), to the case study [32] from the ABZ2020 conference, namely an automotive system with adaptive features composed of two components, *Adaptive Exterior Light* (ELS) and *Speed Control System* (SCS). We used ASMETA for specifying, by a refinement-based process, the component models, for simulation and animation of the ASMETA models, for scenario-based validation against the informal requirements, verification of behavioral properties, and code generation to prototype the control light system on an Arduino board. ASMETA employs several constructs in order to allow modularity, rule styles, and idioms to model and best capture specific behavioral patterns. For modeling the adaptive control features, i.e., the adaptive high beam headlights and the adaptive cruise control, we adopted an MAPE-K (Monitor, Analyze, Plan, Execute over a shared Knowledge) [33] feedback control loop as mainstream for engineering self-adaptation by means of a higher software layer (the adaptation layer) that operates on top of the system to adapt it. In particular, we exploited the concept of *self-adaptive ASMs* [9], which allows ASM-based modeling of MAPE-K loops behavior.

The first version of our specification, validation, and verification of the case study appeared in our ABZ conference paper [13]. Additional contributions of this paper with respect to the previous version can be summarized as follows. This work:

✉ S. Bonfanti
silvia.bonfanti@unibg.it

P. Arcaini
arcaini@nii.ac.jp

A. Gargantini
angelo.gargantini@unibg.it

E. Riccobene
elvinia.riccobene@unimi.it

P. Scandurra
patrizia.scandurra@unibg.it

[1] National Institute of Informatics, Tokyo, Japan

[2] University of Bergamo, Bergamo, Italy

[3] University of Milan, Milan, Italy

- captures the version 1.17[1] of the requirements' specification document (in paper [13] we have used version 1.9) and version 1.8 of validation sequences;
- details the requirement traceability method used throughout the formal modeling process;
- shows how to generate, from the ASMETA model, an executable C++ code that has been used for an Arduino board implementation;
- provides a more detailed description of modeling styles and idioms adopted (and available in ASMETA);
- presents a comparison with other case study solutions.

Paper structure Sect. 2 introduces the ASM formal method, the ASMETA tools, the modeling process we adopted and its distinctive features, the development team, and some useful information for nonexperts. Section 3 details our modeling strategy, while Sect. 4 provides models details, including modeling styles and idioms, model snippets, time constraints modeling, and strategies for making models readable/understandable for nonexperts. Section 5 presents the results and insights achieved from the validation and verification activities. Section 6 presents the prototypical implementation in Arduino. Section 7 discusses the ambiguities we have been able to identify in the requirements, and the model changes due to the various versions of the improved requirements; it also underlines possible improvements at the tools level to model system features arisen from the case study. Section 8 compares our solution with those proposed with other formal approaches and reports on other application case studies and existing tools for the ASMs.. Finally, Sect. 9 concludes the paper.

2 Methods and tools introduction

For our case study solution, we used the ASMETA toolset. Since it is based on the ASM formal method, we premise the presentation of ASMETA with a basic introduction of the ASMs.

Abstract State Machines ASMs [26, 28] are a state-based formal method. *States* are mathematical *algebras*; they specify a system configuration by arbitrary complex data, i.e., domains of elements with functions (also boolean) defined on them. State *transitions* are expressed by transition rules describing how the data (function values saved into *locations*) change from one state to the next. Functions of the algebra are distinguished between *dynamic* and *static* depending on whether or not they are updated by rule transitions. Dynamic functions are then classified into *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and its environment); and *derived*, namely those defined in terms of other (dynamic) functions.

```
asm CarSystem001
import StandardLibrary
signature:
// DOMAINS
enum domain PitmanArmUpDown = {DOWNWARD5, ...}
...
// FUNCTIONS
monitored pitmanArmUpDown: PitmanArmUpDown
controlled pitmanArmUpDown_Buff: PitmanArmUpDown
derived turnLeft: PitmanArmUpDown −> Boolean
definitions:
//FUNCTIONS DEFINITION
function turnLeft ($position in PitmanArmUpDown) = ...
...
//RULE DEFINITION
macro rule r_HazardWarningLight = ...
...
// MAIN RULE
main rule r_Main =
     r_HazardWarningLight[]
// INITIAL STATE
default init s0:
function pitmanArmUpDown = NEUTRAL_UD
...
```

Code 1 Example of Asmeta specification

Transition rules have different constructors depending on the update structure they express, e.g., guarded updates (*if-then*, *switch-case*), simultaneous parallel updates (*par*), nondeterminism (*choose*), unrestricted synchronous parallelism (*for-all*), abbreviation on terms of rules (*let*), etc. The *update* rule is the basic unit of rules construction and it has the form $f(t_1,\ldots,t_n) := v$, being f an n-ary function, t_i terms, and v the value of $f(t_1,\ldots,t_n)$ in the next state.

An ASM *run* is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states: starting from an initial state S_0, a *run step* from S_n to S_{n+1} consists in firing, in parallel, all enabled transition rules and leading to simultaneous updates of a number of locations. In case of an inconsistent update (i.e., the same location is updated to two different values) or invariant violations, the model execution stops.

ASMETA Toolset ASMETA[2] is a set of tools around the ASM method for model editing, simulation, validation, verification, and code generation. It has been developed [3] by exploiting the Model-Driven Engineering (MDE) approach for software development starting from the definition of a metamodel for an abstract notation able to capture the *working definition* (see [28, p. 32]) of an ASM. An ASMETA model is fully compliant to such definition and it is composed of (see Code 1):

---

[1] https://abz2021.uni-ulm.de/resources/files/casestudyABZ2020v1.17.pdf.

[2] https://asmeta.github.io/.

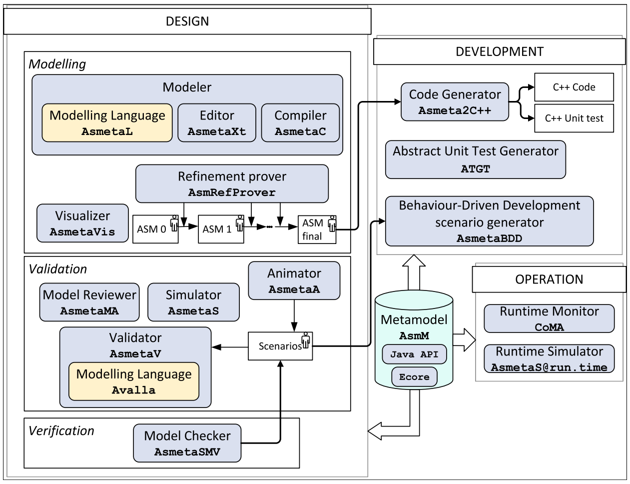


Fig. 1 ASMETA-based development process

- The import section that allows including all or some of the declarations and definitions given in another ASMETA specification.
- The signature section where domains and functions are declared.
- The definitions section where all transition rules and possible invariants, i.e., first-order formulas that must be true in all states, are specified.
- The main rule which is the starting point of the computation at each state; it, in turn, calls all the other transition rules (defined as *macro call rules*[3]). A *run step* of an ASMETA model is the parallel execution of all transition rules, which are directly or indirectly called from the main rule and are enabled to fire.
- The default init section where initial values for the *controlled* functions are defined.

A model without the main rule is called *module*, which consists of declaration and definitions of domains, functions, invariants, and macro call rules, and it can be imported by other models; the keywords `asm` and `module` are used to specify a machine and a module, respectively.

ASMETA tools support the main activities of the software development process from formal requirements' specification to code generation. Figure 1 shows the tools usage in the various stages [15]. At *design* time, ASMETA provides tools for model editing and visualization (the modeling language `AsmetaL`[4] and its editor and compiler, plus the model visualizer `AsmetaVis` for graphical visualization of ASMETA models), model validation (e.g., interactive or random simulation by the simulator `AsmetaS`,[5] animation by the animator `AsmetaA`, scenario construction and validation by the validator `AsmetaV`, and static analysis by the model reviewer `AsmetaMA`), and verification (proof of temporal properties by the model checker `AsmetaSMV`, and proof of correct model refinement by `AsmRefProver`). During software *development*, ASMETA supports automatic code and test case generation from models (the code generator `Asmeta2C++`, the unit test generator `ATGT`, and the acceptance test generator `AsmetaBDD` for complex system scenarios). If the system is available, during its *operation*, ASMETA can be used for runtime monitoring (by the tool `CoMA`) and runtime simulation (by `AsmetaS@run.time`).

Modeling process ASMETA is based on the ASM theory, therefore, as in ASMs, the ASMETA modeling process is *iterative* and based on *model refinement*. Specifically, ASMETA is based on the use of *stuttering refinement* [7] which is a limited form of the general ASM refinement [25].

This refinement-based process allows us to tackle the complexity of the requirements and to bridge, in a seamless manner, specification to code. Requirements modeling starts by developing a high-level ASMETA model (similarly to the ASM *ground model*[28]), which is specified by reasoning on the informal requirements, usually given in natural language. Model signature and rule naming are set by using terms of the application domain and derived from textual requirements; this facilitates the tracing from requirements to model. This high level model (see model ASM_0 in Fig. 1) should be *correct*, i.e., reflect the intended requirements (at a desired level of abstraction), and *consistent*, i.e., no ambiguities of initial requirements should be left. It does not need to be *complete*, i.e., it may not specify some given requirements that are later captured during the refinement process. Indeed, starting from the model ASM_0, through a sequence of *refined* models ASM_1, $\mathsf{ASM}_2, \ldots$, other functional requirements are specified, till reaching a level in which the final model $\mathsf{ASM}_{\mathrm{final}}$ captures all intended requirements at the desired level of abstraction. Each refined model must be proved to be a correct (stuttering) refinement of the previous one and ASMETA supports the designer with a tool (`AsmRefProver`) for automatic checking of the correctness of refinement steps [7].

At each refinement level, already at that of the model ASM_0, validation and verification (V&V) activities can be performed to assure requirements' satisfaction and property validity by using the already mentioned ASMETA tools available at the design stage.

Distinctive features of ASMETA approach Developing our specification of the case study, we exploited a number of

---

[3] Note that to define a macro call rule in the definition section we use the syntax macro r_rule(*params*), while the macro rule is invoked from another rule as r_rule[*params*].

[4] It is a concrete notation for the abstract one defined by the metamodel reflecting the working definition of an ASM.

[5] The ASMETA model simulator implements the computational paradigm of an ASM run as defined in the previous section.

modeling features that the ASMETA approach inherits from the ASM method and its extension for modeling self-adaptive systems. They can be summarized as follows:

- Models can be specified at any desired *level of abstraction*, so designers can decide the level of details they want to achieve.
- Model *refinement* can be exploited to face with the complexity of requirements' specification, as we show in this case study (see Sect. 4.2).
- Models are *executable* and this feature is fundamental to reproduce requirements' scenarios that are usually given, as it was for the case study, as part of the initial informal requirements; moreover, mainly at operation time, they can be co-executed with system implementations [40].
- Model *modularization* facilitates model scalability and separation of concerns, so tackling the complexity of large requirements' specification. We widely used modules in modeling the case study, as better explained in Sect. 3.
- Adaptive features of the two automotive subsystems of the case study have been modeled by exploiting the concept of *Self-adaptive Abstract State Machine* [9] that uses MAPE-K feedback control loops [33] (sequences of four computations Monitor, Analyze, Plan, and Execute over a shared Knowledge) to structure the adaptation logic of self-adaptive software systems. The four MAPE computations to let the two automotive subsystems self-adapt in certain circumstances have been formalized in terms of transitions rules (see Sect. 4.2, *CarSystem003*).

Relevant information for a nonexpert Since ASMETA models are just ASMs edited in a concrete (and machine-handling) syntax, as ASM models they have a *pseudocode format*, so they can be easily read by practitioners and nonexperts as high-level programs of virtual machines working over abstract data structures. The only insight that needs to be understood is that, at each machine computational step, all possible locations updates are executed in parallel. Moreover, thanks to the ASMETA modeling style that allows reflecting the cause–effect description of the informal requirements in the conditional rule structure, and to name rules and signature elements by terms belonging to the application domain, a nonexpert reader can easily *trace* informal requirements with their formalization in the model. An example of this modeling style and rule/function naming is given in Code 2, which is commented in Sect. 3.

Development team overview The development team consists of the authors, who are experts of the method and the tools, and one master student, less experienced in modeling and analysis, who worked on the Arduino encoding of the specified ASMETA solution of the case study. Each activity was carried out by a subgroup of two people, so two worked on modeling, two on V&V, and two on the implementation. This separation of responsibilities reflects the usual distribution of work among different teams in a development process. They followed an iterative approach, alternating modeling and V&V phases of the development process, and used a private GitHub repository to share artifacts. Once modeling was complete and all necessary details modeled, an implementation to the Arduino platform was developed by the student (supervised by one of the authors) by exploiting the approach of automatic code generation from models (see Sect. 6).

3 Requirements and modeling strategy

The system to model is composed by two subsystems, ELS (Exterior Light System) and SCS (Speed Control System). The ELS is responsible for managing lights of a car: direction indicators, low beam headlights, cornering lights, high beam headlights, and emergency brake, while the SCS tries to maintain or adjust speed of the vehicle according to external influences. The proposed ASMETA model is primarily tailored to the formalization and analysis of functional requirements of the ELS and SCS subsystems in order to provide guarantees of their operational correctness. We here describe how we derived the formal specification from the informal requirements given in [32],[6] the structure of the models and the most important properties addressed, the traceability between requirements and models, how we managed the variability of the requirements, and the features we did not model.

From requirements to specification From the textual description of the system, we derived the transition rules in the same incremental way in which functional requirements are given in the requirements' document [32]. Rules construction reflects in a direct way the cause–effect description of the requirements in the informal text. Meanwhile, domains and functions occurring in the transition rules were defined in the signature section of the model. To facilitate traceability between informal requirements and models, when defining and declaring signature terms, we used names and value ranges as reported in tables at the end of the requirements' document.

An example of requirements' specification is shown in Code 2 that reports part of the requirement ELS-18 and the corresponding specification in the transition rule. *If the light rotary switch* (used to set the lights in the following positions: Off, Auto, On) *is in position Auto* (lightRotarySwitch = AUTO), *the ignition is ON* (engineOn(keyState) is a function

---

[6] The case study description and the requirements are not reported here, please refer to the original documentation [32].

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.

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

Code 2 Translating textual requirements into rules

SCS-2. When pulling the cruise control lever to 1 (Forward), the desired speed is either the current vehicle speed (if there is no previous desired speed) or the previous desired speed (if already set).
SCS-3. If the current vehicle speed is below 20 km/h and there is no previous desired speed, then pulling the cruise control lever to 1 (Forward) does not activate the (adaptive) cruise control.

```
macro rule r_SCSLeverForward =
            if sCSLever = FORWARD_SCS then
            //SCS-3
                if currentSpeed < 200 then
                    if desiredSpeed = 0 then
                        setVehicleSpeed := 0
                    endif
                else
                //SCS-2
                    if desiredSpeed = 0 then
                        par
                            setVehicleSpeed := currentSpeed
                            desiredSpeed := currentSpeed
                        endpar
                    else
                        setVehicleSpeed := desiredSpeed
                    endif
                endif
            endif
```

Code 3 Modeling two requirements dependent on each other

that returns true if the ignition is on), *the brightness* measured by the brightness sensor *is lower than 200 lx* (brightnessSensor < 200), and the low beam headlights are not activated (not lowBeamLightingOn), *the low beam headlights are turned on* by calling the rule r_LowBeamTailLampOnOff.

After a first draft of the specification, model refactoring led to a more concise rule organization due to the fact that, sometimes, requirements are not independent of each other and so they were captured by the same rule. This is the case, for example, of requirements SCS 2 and 3 formalized by the rule r_SCSLeverForward (see Code 3). If current vehicle speed is lower than 20 km/h (SCS 3 – current_Speed < 200) and there is no previous speed to be used when cruise control is active (desiredSpeed = 0), the cruise control is not activated (setVehicleSpeed := 0). If the current vehicle speed is higher than 20 km/h (SCS 2, the condition opposite to that of requirement SCS 3), and there is no previous speed to be used when cruise control is active (desiredSpeed = 0), the cruise control is activated (setVehicleSpeed := currentSpeed) with the desired speed being equal to the current vehicle speed (desiredSpeed := currentSpeed). In case an existing desired speed is already stored, it is used by the cruise control as a target speed to be kept.

Structure of the formal model Since our modeling activity followed the organization of the requirements' document, the overall structure of our models reflects that of the informal documentation; again, this favors the traceability between requirements and models. First, we modeled the two systems separately because they share some signals but never update common parameters (e.g., the SCS updates the speed, and the speed is only read by the ELS). For both subsystem modeling, we went through a sequence of refinement steps by adding details at each level, and then, when the two subsystems needed to cooperate, we merged the two models to obtain the complete system specification. Figure 2 shows the model refinement chain and, for each model in the chain, lists the requirements introduced in each model.[7] The models are numbered from 1 to 9: models from *CarSystem001* to *CarSystem004* refer to the ELS, while the SCS is modeled from *CarSystem005* to *CarSystem007*. *CarSystem008* merges the two systems, and *CarSystem009* introduces the faults handling and general properties.

In modeling the case study, we strongly used ASMETA *modules* to facilitate reuse of signature terms and rules in different parts of the models. Indeed, at each refinement step, only some parts of the model were refined and most of the model remained unchanged. The use of modules helped us keep under control the refinement process by avoiding code duplication and marking those parts of the model that were involved in the refinement step. When a model was refined, all those importing it were refined as well. For example, Fig. 3 illustrates the ASMETA model structure in terms of horizontal and vertical imports for the first three refined levels of the *CarSystem* related to the SCS subsystem (in [13] we presented, instead, the structure of the ELS subsystem).

Level 1 (top of the figure) consists of ASM *CarSystem005main* that imports module *CarSystem005DesiredSpeedCruiseC* that imports module *CarSystem005Functions*, till the final module *CarSystem005Domains* is imported.[8] At this level, module relations are all horizontal imports. Similarly, the model *CarSystem006main* imports (horizontally) module *CarSystem006SpeedLimitTrafficDet* from the same refinement level, which imports the module *CarSystem006DesiredSpeedCruiseC* from the same refinement level obtained by refining the module *CarSystem005DesiredSpeedCruiseC*. *CarSystem006DesiredSpeedCruiseC* imports module *CarSystem006Functions* which (vertically) imports *CarSystem005Functions*, since the former enlarges (as

---

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

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

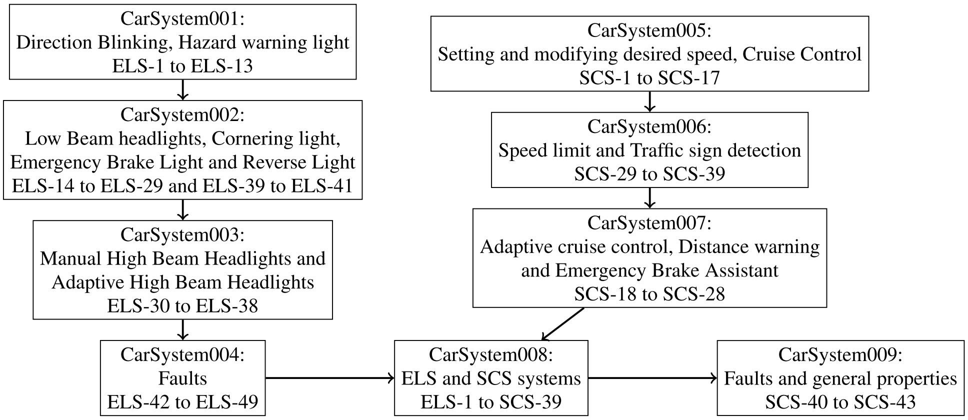


Fig. 2 Chain of refined models and captured requirements

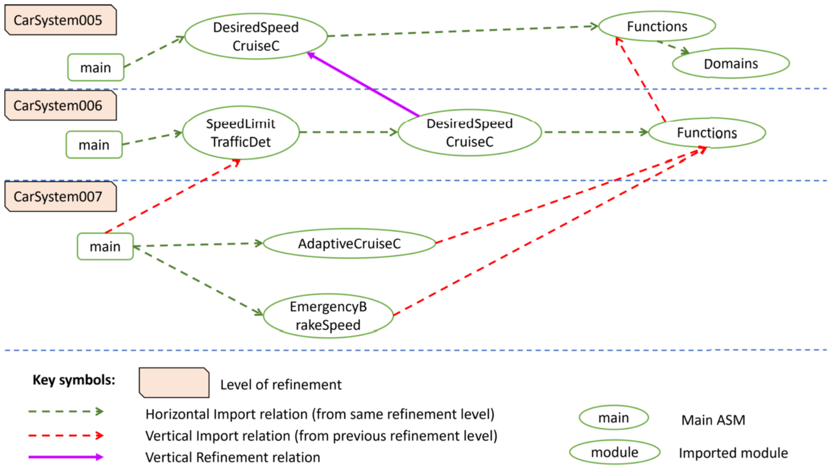


Fig. 3 ASM structure of the first three refinement levels of SCS

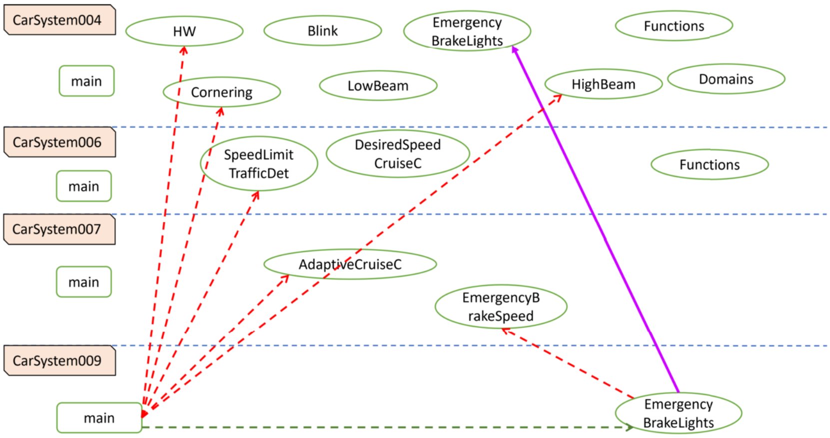


Fig. 4 ASM structure of last refinement level (merge of ELS and SCS)

expected during refinement) the latter. *CarSystem007* refinement level imports (vertically) modules from *CarSystem006* (*CarSystem006SpeedLimitTrafficDet* and *CarSystem006Functions*) and defines two new modules (*CarSystem007AdaptiveCruiseC* and *CarSystem007EmergencyBrakeSpeed*) to model new behaviors. Figure 4 focuses on the last refinement level and illustrates the model *CarSystem009main* that imports modules from previous refinements, both ELS (*CarSystem004*) and SCS (*CarSystem006* and *CarSystem007*) subsystems. Moreover, in this last refinement level, the module *CarSystem009EmergencyBrakeLights* is refined from *CarSystem004EmergencyBrakeLights*. Others depicted import/refinement relations are self-explanatory. More details on the refinement levels and corresponding models are given in Sect. 4. Table 1 shows the model dimensions in terms of number of functions and rules.

Table 1 Model dimensions

	Functions				Rules	
	monitored	controlled	derived	static	# rule declarations	# 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

Remark 1
Note that, according to the model chain shown in Fig. 2, the model *CarSystem008* merges the two models of the subsystems SCS for the speed control and ELS for the light control. This model is a multiagent ASMETA model with two types of agents, one – the ESL – behaving as modeled by the rules in the model *CarSystem004*, and the other – the SCS – behaving as modeled by the rules in the model *CarSystem007*. The main rule in the model *CarSystem008* executes in parallel all rules of the two agents. Since in this case study we do not have more than one agent of the same type, to speed up the model checking verification, we simplify the usual structure of an ASMETA multiagent model that would have defined two subsets of Agent, one for agents of type ELS and the other for agents of type SCS, associating to each the relative agent's program (i.e., the main rule of the model *CarSystem004* for the agents of type ELS, and the main rule of the model *CarSystem007* for the agents of type SCS), and defining an execution schedule for the agents (in parallel for the agents ELS and SCS). The simplification does not affect the validation and verification results.

Traceability between formalization and requirements
As already explained, traceability between given requirements and models is facilitated by the style of the ASMETA modeling: rule structure straightforwardly reflects the cause–effect description of the informal requirements, and naming rule and signature elements use terms which belong to the application domain and are self-explanatory for the readers familiar with the case study. However, in order to precisely document traceability between requirements and formalization, we introduced tables as shown in Tables 3 and 4, reported in Appendix A and available online. In these tables, for each requirement, we specify in which refinement step it is introduced (bright green) and which rule formalizes it. For example, the requirement ELS-1 is implemented in the rules r_RunDirectionBlinking and r_CompleteFlashingCycle in the module *CarSystem001Blink*. The availability of such tables helped us a lot, especially when we found errors during validation and verification activities because the problem identification was more immediate knowing where a specific requirement was exactly specified.

Variability of requirements The proposed automotive system belongs to a family of systems that can be configured on the basis of law restrictions or customers' preferences. For example, in the USA and Canada, the tail lights realize the direction indicator lamps (ELS-23). This means that different configurations lead to different behaviors of the system. Therefore, when modeling the case study, we had to face with this system configuration problem, capturing the fact that on the base of the market and type of car, the system behaves differently. To model the system configuration variability, we have used *flags* as static functions which cannot be changed during the execution. For example, the function marketCode on domain MarketCode = {USA, CANADA, EU} is used to denote the country where a given requirement formalization should hold; it is used within the definition of other functions (see, e.g., the definition of the function tailLampAsIndicator in Code 7) to discriminate among countries. The use of these flag functions, reduces, however, the readability and maintainability of the models because, to simulate different behaviors, we need to change them manually in the model.

Most important properties addressed by the models
The ASMETA specification of the case study allowed us to guarantee the correct operation of the ELS and SCS subsystems by proving a number of properties summarized as follows (verification results are reported in Sect. 5.2):

- 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 automatically sets the speed to reach the target based on 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.

Although we addressed all the previous properties, some features are still not captured, as explained below.

Features not addressed In our models, we did not explicitly model the *time* because, although ASMETA supports it in modeling and validation phases, as discussed in Sect. 7.2, no verification tool deals with time. For this reason, we modeled the time using the pattern explained in Sect. 4.1.

Another not addressed feature is the frequency of blinking. Since the model does not support time, it is not possible to set the direction lamps state (ON or OFF) every second (the duration of a flashing cycle). Due to this limitation, we introduced an enumerative indicating the current pulse ratio, and we assumed that the Head-Unit sets the state of direction lamp given the pulse ratio value.

Moreover, since ASMETA allows modeling discrete-event systems and not continuous systems, we had to transform some continuous functions into discrete ones suitable for modeling and analysis purposes, as better explained in Sect. 8.

4 Model details

The previous section provides insights into how we approached the formalization of the requirements and presents the overall structure of the chain of refined models. We here describe styles and idioms identifiable in our modeling and present key snippets of our models. A final discussion on how we improved model readability, by exploiting characteristics of our approach, follows.

4.1 Modeling styles and idioms

We here describe some relevant modeling styles and idioms used in developing our solution for the case study. These are modeling patterns that can be used to solve recurrent problems regarding how to specify computational or control behaviors.

Parallel ASM In order to have a simpler and more manageable system model for analysis purposes, we abstract from the control software components distribution on the hardware nodes. So we simply exploited the notion of *synchronous parallel ASM* [26, 28] to model the overall behavior resulting from the union of the behaviors of all distributed control software running on ECUs (Electronic Control Units) [17]. Therefore, we modeled software running on ECUs as synchronous 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 modeling them as conditional rules with mutually exclusive guards. Moreover, in our model we assume that the two main subsystems (ELS and SCS), coherently with the description of the case study, run on the same physical control unit, so we assume that the two subsystems run synchronously.

Since we did not model the system as a distributed system being composed of a collection of machines (agents), we did not adopt the class of *concurrent ASMs* [27], which exploit the intuitive understanding of concurrency dealing with computations of multiple ASM agents running asynchronously, each with its own clock, and interacting via reading/writing values of designated locations. Such concurrency theory is vital for modeling distributed systems but not yet completely supported in ASMETA, which allows specifying independent multiagent ASMs executing in parallel with explicit scheduling provided at model level by the main rule.

ECA rule In the requirements' document, most actions are executed when an event occurs and a condition is satisfied. To model this behavior, we have extensively used conditional rules and often ECA (Event Condition Action) rules of the form:

```
if external_event then
   if internal_condition then  actions  endif
endif
```

to differentiate *external events* (predicated over monitored functions) from *internal conditions* (predicates over controlled and derived functions), where *actions* correspond to state updates. An example of an ECA rule is shown in Code 3, where sCSLever = FORWARD_SCS is an external event and currentSpeed < 200 is an internal condition.

MAPE-K loop To model the system adaptive features, we adopted an MAPE-K feedback control loop, which is typically used in self-adaptive systems as self-adaptation mechanism. Conceptually, an MAPE-K loop is a sequence of four computations: Monitor–Analyze–Plan–Execute (MAPE) over a knowledge base (K). In self-adaptive ASMs [9], it

```
signature:
    enum domain KeyPosition = {NOKEYINSERTED | KEYINSERTED
    | KEYINIGNITIONONPOSITION} // Key state
        monitored keyState: KeyPosition // Position of the key
        controlled keyState_Previous: KeyPosition

definitions:

macro rule r_ReadMonitorFunctions =
        keyState_Previous := keyState
```

Code 4 Value change detection

is realized by means of a schema of four rules, one per MAPE computation, while the knowledge is modeled by means of functions, since in ASMs system memory is represented in terms of functions. See, for example, the ELS model *CarSystem003* in Sect. 4.2.

Value change detection When modeling, we were faced with the problem of formalizing *function value change events* occurring when a value change of a function X from state $s-1$ to state s triggers an action from state s to state $s+1$. To specify this kind of change event, we have introduced two functions: a monitored function X, yielding the value of the function at the current state, and a controlled function X_Previous which stores the value of the function at the previous state. When a value change (i.e., X ! = X_Previous) is detected, the system executes the specified actions.

An example are the functions to monitor the change of *key* state. In the model, we introduced the monitored function keyState and the controlled keyState_Previous (see Code 4 for their declaration). At each step, we assign to keyState_Previous the current value of keyState. When keyState ! = keyState_Previous, the model detects that the key state has changed.

Time pattern Often, informal requirements refer to something that has to occur on an expired time interval or at a precise moment in time. Some of the requirements of the case study have statements in this form. We abstract from the use of timers, and we suppose that it is up to the environment to notify when an interval of time is elapsed or a time moment is reached, and we use a boolean monitored function to model such events. For example, requirement ELS-18 states that low beam headlights remain active at least for 3 seconds. As shown in Code 5, we introduced the monitored function passed3Sec that notifies if 3 seconds had elapsed since the low beam headlights received the command to be turned off.

Recently we have extended the time feature in ASMETA with the introduction of the TimeLibrary (see paragraph “Time modeling” in Sect. 7.2 for further details).

```
macro rule r_LowBeamHeadlights =
    ...
    if lightRotarySwitch = AUTO and engineOn(keyState) and
                                        brightnessSensor>250 then
        if lowBeamLightingOn and passed3Sec then
            r_LowBeamTailLampOnOff[0]
        endif
    endif
```

Code 5 Using boolean monitored functions as timers

```
//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
                r_SavePitmanArmUpDownReq[]
            endpar
        else
            r_DirectionBlinking[]
        endif
        ...
    endif
```

Code 6 Hazard warning has priority over direction blinking

```
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 := $pulse
        //ELS-6
        if tailLampAsIndicator then
            r_setTailLampLeft[50,BLINK]
        else
            r_setTailLampLeft[$value,FIX]
        endif
    endpar
```

Code 7 Tail lamp for direction blinking and hazard warning

4.2 Models snippets

As explained in Sect. 3, we modeled the ELS and SCS systems separately, then we merged them together. We present here some snippets of the models.

4.2.1 Adaptive exterior light system

We modeled the ELS subsystem in four refinement steps as follows.

CarSystem001 In the first step, we defined the model for direction blinking (in *Blink* module) and hazard warning (in *HW* module) functional requirements. Codes 6 and 7 show the most critical features addressed, which are respectively the priority of hazard warning over direction blinking (ELS-13) and the use of tail lamp as indicator for a car sold in the USA and Canada (ELS-6). In Code 6, in case hazard warning is not running (line 5) and the hazard warning request arrives (line 7), blinking is stopped (line 10) and hazard warning starts (line 12). Possible active blinking request (line 13) or future blinking requests are stored and restarted as soon as hazard warning is deactivated (if pitman arm is still in the blinking position). We defined three functions to manage the pitman arm requests: 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 running request has been processed, the request in the buffer is executed unless a new one arrives. Functions and domains are defined respectively in *Functions* and *Domains* modules. The tail lamp status can be in two states, FIX or BLINK. It blinks if blinking is required and the car is sold in US or Canada (line 2).

CarSystem002 This step models the low beam headlights and cornering light, emergency brake light, and reverse light functions (from ELS-14 to ELS-29 and from ELS-39 to ELS-41). We defined three different modules: *LowBeam* for the low beam headlights behavior, *Cornering* to model cornering lights and reverse light functions, and *EmergencyBrakeLights* for the emergency brake behavior. Common functions and domains are extended starting from those defined in the *CarSystem001*, while hazard warning and direction blinking are unchanged. A peculiarity of this level is that requirement ELS-19 has the priority over ELS-15, ELS-16, and ELS-17 if ambient light is activated. We modeled this situation by defining a guard called ambientLightingAvailable which is true if ambient lighting is activated, the vehicle is not armored and darkness mode is switched off. When modeling low beam headlights, we applied one of the idioms presented in Sect. 4.1 to detect a value change. An example is the requirement ELS-15: when the ignition is in position KeyInserted, if the light rotary switch is turned to the position On, the low beam headlights are activated at 50%. We captured the light rotary switch mutation by checking the value of light_Rotary_Switch_Previous compared to light_Rotary_Switch as shown in Code 8.

CarSystem003 This refinement step introduces the control features for the manual and adaptive high beam headlights (ELS-30 to ELS-38). Due to some incompleteness of the requirements in manual mode, we had to make the following assumptions: (*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. Instead, considering the adaptive behavior, we have modeled it in terms of an MAPE-K feedback control loop that starts with the rule r_MAPE_HBH (see Code 9). The MAPE loop consists of the following rules invoked in a waterfall manner within one single machine step (see Code 9): rule r_Monitor_Analyze_HBH monitors and analyzes computations; rules r_IncreasingPlan_HBH and r_DecreasingPlan_HBH plan the adaptation if necessary: light illumination distance (computed by the function lightIlluminationDistance(calculateSpeed)) and luminous strength (computed by the function luminousStrenght(calculateSpeed)) are increased or decreased according to the vehicle speed; rule r_Execute_HBH sets the values as planned: highBeamOn to activate/deactivate the high beam, highBeamRange and highBeamMotor for the high beam luminous strength and illumination distance.

```
macro rule r_LowBeamHeadlights = ...
    if lightRotarySwitchPrevious != ON and lightRotarySwitch = ON then
        if keyState = KEYINSERTED then
            r_LowBeamTailLampOnOff[50]
    endif endif ...
```

Code 8 Value change detection in *Carsystem002LowBeam*

CarSystem004 In this model, we introduce fault management in case of overvoltage or subvoltage. In case of subvoltage (voltage lower than 8.5 V), ELS functionalities like cornering light and parking light are disabled (see Code 10).

If the system is in overvoltage (i.e., the voltage is more than 14.5 V), the function setOvervoltageValueLight computes the maximum value of lights: it returns the minimum between current light value and the value calculated by overVoltageMaxValueLight function. In this refinement step, we have refined the modules whose behavior is affected by the voltage value (see Code 11[9]): *Blink*, *Cornering*, *EmergencyBrakeLights*, *HighBeam*, *HW*, and *LowBeam*.

4.2.2 Speed control system

The Speed Control System was modeled in three steps of refinement, as explained in the following.

CarSystem005 This model implements the functions to set and modify the desired speed when cruise control is active

---

[9] Note that the ASMETA syntax for expressing a conditional term and a conditional rule might appear the same since they use the same keywords; however, a conditional term (as that used to define the function setOverVoltageValueLight) is evaluated in the current state to provide the value of the conditional term, while a conditional rule (as the rule r_CorneringLights in Code 10), if invoked, induces function updates in the next state.

```
asm CarSystem003
...
signature:
static percentageHBM: Integer –> HighBeamMotor
derived lightIlluminationDistance: CurrentSpeed –> HighBeamMotor
derived luminousStrength: CurrentSpeed –> HighBeamRange
 ...
definition:
function percentageHBM($x in Integer) =
            if $x <= 65 then 0
            else if $x <= 100 then 1
            else if $x <= 120 then 2 ...
            endif endif endif

function lightIlluminationDistance($y in CurrentSpeed) =
                let ($x = $y/10) in
                        if ($x <= 171.0 ) then
                percentageHBM(rtoi($x * $x * 2.0 * 0.00025 + 95.0))
                        else 11 //300 m
                        endif
                endlet

function luminousStrength($x in CurrentSpeed) =
        if $x <= 1200 then rtoi((7*$x/10 + 60.0)/9.0)
        else 100 endif

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 9 MAPE loop to start and stop the adaptive high beam headlight

```
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 10 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 11 Overvoltage handling

```
macro rule r_UpwardDownward5 ($speed in CurrentSpeed) =
            par
                if sCSLever = UPWARD5_SCS then
                    if $speed <= (2000–10) then
                        $speed := $speed + 10
                endif endif
                if sCSLever = DOWNWARD5_SCS then
                    if $speed–10>10 then
                        $speed := $speed – 10
                    else
                        $speed := 10
                    endif
                endif
            endpar
```

Code 12 Set desired speed when cruise control is active

(from SCS-1 to SCS-17). The behavior is implemented in the module *CarSystem005DesiredSpeedCruiseC* and a snippet is shown in Code 12. When the driver moves the cruise control lever to position two (sCSLever = UPWARD5_SCS) or three (sCSLever = DOWNWARD5_SCS) the speed is increased or decreased accordingly by 1 km/h in the range 1–200 km/h.[10]

CarSystem006 This refinement step introduces the traffic sign detection functionalities. Code 13 shows the modeling of requirements SCS-36, SCS-37, SCS-38, and SCS-39. When the adaptive cruise control is active (cruiseControlMode = CCM2), the traffic sign detection is activated by the user (trafficSignDetectionOn) and the user is not braking (brakePedal!=0); if a traffic sign is detected (detectedTrafficSign = SIGNDETECTED) the vehicleSpeed is set to the detected

---

[10] Speed range is 0–5000 in the models as written in the requirements to allow speed resolution of 0.1 km/h without the use of real numbers.

```
macro rule r_TrafficSignDetection =
    if cruiseControlMode = CCM2 and trafficSignDetectionOn and
    not manualSpeed and brakePedal!=0 then
        if gasPedal = 0 then
            par
                if detectedTrafficSign = UNLIMITED then
                    if desiredSpeed < 1200 then
                        setVehicleSpeed := 1200
                    else
                        setVehicleSpeed := desiredSpeed
                    endif
                endif
                if detectedTrafficSign = SIGNDETECTED then
                    setVehicleSpeed := speedLimitDetected
                endif
                if detectedTrafficSign = NOSIGNDETECTED then
                    skip
                endif
            endpar
        endif
    endif
```

Code 13 Traffic sign detection

value (speedLimitDetected), if the detected value is UNLIMITED the speed is set accordingly to ELS-39.

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. We have modeled the adaptive behavior using the MAPE-K feedback control loop, as done in module *CarSystem003HighBeam*. The cruise control monitors the distance from the vehicle ahead, and plans and executes acceleration/deceleration automatically, including braking until a full standstill and starting from a standstill.

4.2.3 Merging ELS and SCS models

After having developed the ELS and SCS separately, we merged them to obtain a model that includes both systems. Thanks to the use of modules in the previous steps of refinement, this phase turned out to be easy.

CarSystem008 In this refinement step, we defined the main module (*CarSystem008main*) and then we simply imported the modules previously developed. When merging the two subsystems, we did not found inconsistent updates because the systems are independent of each other, and they have only common inputs.

CarSystem009 In the last refinement step, we 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 due to the modeling strategy adopted. The requirement SCS-43 was integrated by refining the module *CarSystem004EmergencyBrakeLights*: the brake lights are activated either by the brake pedal pressed by the user or by the emergency brake automatically activated by the system.

4.3 Readability and understandability of models

To make the models more readable for nonexperts, first, we applied *modularization*, which is a known good practice in code development, as it allows producing code with higher cohesion and lower coupling [42]. As a by-product of modularization, the code is more readable, because each software module focuses on a specific task, so readers of the code can focus on understanding each functionality of the code at a time, instead of trying to understand all the functionalities at once. We believe that the advantages of modularization also apply to models, and so we used it in our modeling.

Then, we have defined domains and functions using a domain specific terminology that can be understood by the stakeholders as used in the requirements' document. For example, the status of engine function is called engineOn which is true if the engine is on, false otherwise. Moreover, the rule's name exemplifies the content of the rule. For example, the rule that turns on or off the lights for direction blinking right is called r_BlinkRight, and the rules that implement the MAPE-K control loop include the phase of the computation computed by the rule (e.g., r_IncreasingPlan_HBH models the planning phase). In addition, we included requirements' numbers as comments in the models to identify where each requirement is modeled (and supported the user with the traceability table as explained in Sect. 1).

5 Validation and verification

The ASMETA provides different tools to perform validation and verification. We here describe the applied tools and report interesting findings. Moreover, we explain how we changed the models according to the analysis results.

5.1 Strategies and tools for validation

Model validation is applied to gain confidence that the specification reflects the intended requirements; since it is usually applied from the early stages of model development, it allows detecting faults and inconsistencies with limited effort. The ASMETA toolset supports validation by means of different tools; we describe their application to the case study in the following.

The writing of the models was supported by the animator `AsmetaA`, which allows performing a lightweight validation of the model under development. Indeed, `AsmetaA` provides information of the model states and their evolution by means of an intuitive tabular notation, as shown in Fig. 5. By using `AsmetaA`, we conducted *interactive animation*: we provided inputs (i.e., values of monitored functions) to the machine, and observed the computed state in order to check whether

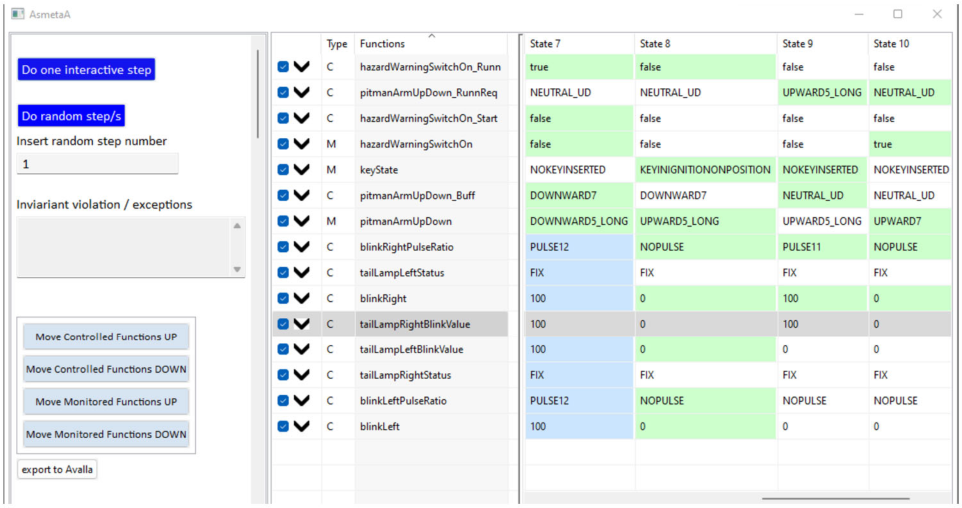


Fig. 5 `AsmetaA` running the ELS subsystem

	Type	Functions	State 7	State 8	State 9	State 10
☑	C	hazardWarningSwitchOn_Runn	true	false	false	false
☑	C	pitmanArmUpDown_RunnReq	NEUTRAL_UD	NEUTRAL_UD	UPWARD5_LONG	NEUTRAL_UD
☑	C	hazardWarningSwitchOn_Start	false	false	false	false
☑	M	hazardWarningSwitchOn	false	false	false	true
☑	M	keyState	NOKEYINSERTED	KEYINIGNITIONONPOSITION	NOKEYINSERTED	NOKEYINSERTED
☑	C	pitmanArmUpDown_Buff	DOWNWARD7	DOWNWARD7	NEUTRAL_UD	NEUTRAL_UD
☑	M	pitmanArmUpDown	DOWNWARD5_LONG	UPWARD5_LONG	UPWARD5_LONG	UPWARD7
☑	C	blinkRightPulseRatio	PULSE12	NOPULSE	PULSE11	NOPULSE
☑	C	tailLampLeftStatus	FIX	FIX	FIX	FIX
☑	C	blinkRight	100	0	100	0
☑	C	tailLampRightBlinkValue	100	0	100	0
☑	C	tailLampLeftBlinkValue	100	0	0	0
☑	C	tailLampRightStatus	FIX	FIX	FIX	FIX
☑	C	blinkLeftPulseRatio	PULSE12	NOPULSE	NOPULSE	NOPULSE
☑	C	blinkLeft	100	0	0	0

it is as expected. Moreover, `AsmetaA`, at each step, also performs *consistent updates checking*, i.e., it checks that all the updates are consistent (see Sect. 2); as an additional feature, if invariants are specified in the model, `AsmetaA` checks them at each step.

Although `AsmetaA` is useful to get an early validation of the model, its interactive nature does not allow consistently applying it to large models having long simulation sequences. Therefore, with the increasing complexity of the ELS and SCS system models, we applied `AsmetaV`, a tool of the ASMETA toolset for *scenario-based validation*, an automated validation approach. Specifically, `AsmetaV` allows designing *scenarios* of possible model evolutions and to specify the expected system behavior; these scenarios are then automatically run by `AsmetaV`.

In order to write a scenario, a designer can use the textual notation `Avalla`, which has constructs to express execution scenarios as interaction sequences consisting of actions committed by the user to `set` the environment (i.e., the values of monitored/shared functions), to `check` whether the machine state satisfies some *assertions*, 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. `AsmetaV` takes as input a scenario and automatically interacts with the simulator `AsmetaS` to execute it; during simulation, `AsmetaV` captures any violation of the assertions and, if none occurs, it returns a *PASS* verdict.

`AsmetaV` allowed us to specify all the validation sequences provided with the case study.[11] We report all the specified scenarios in our online repository. In [13], we reported the second scenario of the validation sequences for the exterior light provided with the case study, and here we report snippets of the scenario specifying the behavior of the fourth validation sequence for the exterior light (see Code 14) and the corresponding output of `AsmetaV` execution where all the assertions have been successfully checked (see Code 15).

```
begin checkStaticFunctions
        check marketCode = EU;
        check armoredVehicle = false;
        check daytimeLight = true;
        check ambientLighting = true;
end
begin checkInitState
        check pitmanArmUpDown = NEUTRAL_UD;
        check pitmanArmUpDown_RunnReq = NEUTRAL_UD;
        ...
end

set lightRotarySwitch := OFF;
set keyState := KEYINIGNITIONONPOSITION;
...
step
check tailLampLeftBlinkValue = 0;
check tailLampLeftFixValue = 100;
...
set lightRotarySwitch := AUTO;
set passed3Sec := false;
...
...
set brightnessSensor := 300;
step
check tailLampLeftBlinkValue = 0;
check tailLampLeftFixValue = 100;
...
set passed30Sec := true;
step
check tailLampLeftBlinkValue = 0;
check tailLampLeftFixValue = 0;
...
```

Code 14 Scenario for normal light, daytime light, ambient light, night

`AsmetaV` allows checking the *coverage* of the models, i.e., which rules are executed during a scenario execution. We computed the cumulative coverage of the scenarios derived from the validation sequences. We notice that not all the rules are covered by the scenarios, meaning that not all the requirements are considered in the validation sequences; for example, the activation of cornering lights (ELS-24), requirements ELS-42 to ELS-49 regarding fault handling of

---

[11] See https://abz2020.uni-ulm.de/resources/files/ValidationSequences_v1.8.xlsx.

```
check succeeded: marketCode = EU
check succeeded: armoredVehicle = false
check succeeded: daytimeLight = true
check succeeded: ambientLighting = true

check succeeded: pitmanArmUpDown = NEUTRAL_UD
check succeeded: pitmanArmUpDown_RunnReq = NEUTRAL_UD
...
check succeeded: tailLampLeftBlinkValue = 0
check succeeded: tailLampLeftFixValue = 100
...
...
check succeeded: tailLampLeftBlinkValue = 0
check succeeded: tailLampLeftFixValue = 100
...
check succeeded: tailLampLeftBlinkValue = 0
check succeeded: tailLampLeftFixValue = 0
...
```

Code 15 AsmetaV output for the scenario for normal light, daytime light, ambient light, night

ELS subsystem, and SCS-40 to SCS-43 regarding fault handling and general properties of SCS system are not covered by any scenario. Having such coverage information is very important, as it can guide in the specification of new scenarios for covering the uncovered rules.

Interactive and scenario-based simulations, as provided by `AsmetaA` and `AsmetaV`, allow checking a specific subset of behaviors of the model, but they do not allow performing an exhaustive check. To this aim, we performed *model review* using the `AsmetaMA` tool [2], an exhaustive validation technique that can check specific types of properties. It is a form of static analysis that determines whether a model has specific *quality* attributes, such as minimality, completeness, and consistency; for example, the tool checks whether the model execution can lead to inconsistent updates (an inconsistency problem); it checks whether some function is neither read nor updated (a minimality problem); please refer to [2] for the complete description of all the properties checked. Although the technique is not able to detect violations of the requirements, it can find different problems related to implementation errors.

Insights obtained from validation Among all the validation activities, scenario-based validation with the `AsmetaV` tool turned out to be the most useful one. Indeed, it allowed us to specify all the scenarios provided with the case study and repeatedly run them over the models under development. In this way, the development is similar to Test Driven Development: (i) an initial model is developed; (ii) scenarios are run over the model; (iii) if any assertion in the scenarios is violated, the model is fixed by the developer accordingly. The development continues from point (ii) until all scenarios pass.

Several problems were detected in this way. For example, we understood that desiredSpeed and targetSpeed are two different concepts that can have different values, and that they are updated with two different policies; by only reading the requirements, we assumed that the user could modify them only together.

In some cases, we found inconsistencies between the scenarios and the requirements. For some of such inconsistencies, we realized that the scenario description was more trustworthy, and it was the one to consider in the development. For example, the requirements' document uses the terms *desired*, *target*, and *set* speed, in an improper way, sometimes interchangeably; instead, the description of the scenarios clearly distinguishes them and shows their different roles in the system.

For other inconsistencies, instead, we were not sure which document to trust. For example, regarding the *traffic sign detection*, the requirements' document states that only the target speed is modified when the sign is recognized; however, in the sixth scenario of *Validation Sequences Speed*, the car modifies both the desired and target speed when it detects the sign. In this case, we still decided to rely on the scenario description, as we believe that stakeholders can properly specify concrete behaviors of a system (as in a scenario), but they may be ambiguous when specifying general requirements (as in the requirements' document).

5.2 Verification approach

In the ASMETA toolset, formal verification is supported by the tool `AsmetaSMV` [1]. It allows for the verification of both *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). LTL and CTL properties can be specified using the ASMETA signature directly in the ASMETA model. `AsmetaSMV` translates the ASMETA model into a model for the model checker NuSMV (which is used as a back-end tool), runs the verification with NuSMV, and parses and pretty-prints the verification results. In this way, the model checker is transparent to the final user, who must only be able to specify properties in LTL and CTL.

We derived different properties from the requirements. In the requirements' document, a typical requirement looks like this: "when/if ... then ... ". It tells that when something happens (a given external input received, a given state condition, etc.), some actions must be taken. These requirements have been captured as one of the following two LTL temporal properties (in the `AsmetaL` notation):[12]

$$\mathbf{g}(\phi \text{ implies } \mathbf{x}(\psi))\mathbf{g}(\phi \text{ implies } \mathbf{f}(\psi)) \tag{1}$$

depending on whether the requirement specifies that ψ must happen "immediately" when ϕ happens (first property), or ψ must "eventually" happen (second property). For example,

---

[12] Here **g** stands for the *always* operator, **x** for the *next* operator, and **f** for the *eventually* operator.

```
Execution of NuSMV code ...
–– specification g((blinkLeft != 0 and blinkLeftPulseRatio != NOPULSE and
blinkRight != 0 and blinkRightPulseRatio != NOPULSE) =
hazardWarningSwitchOn Runn) is true
–– specification G((tailLampLeftStatus=BLINK or tailLampRightStatus=BLINK)
implies marketCode!=EU) is true
–– specification AG true is true
```

Code 16 Traffic sign detection

we specified the following property for capturing requirement ELS-18:

g((lightRotarySwitch = AUTO and engineOn and brightnessSensor < 200)
implies **x**(lowBeamLightingOn))

It is easy to see that the types of property reported in Eq. (1) (especially the first one) are reflected in the structure of ASMETA rules that have been derived from the requirements that are specified as conditional rules (possibly ECA rules). So temporal properties act as *redundant specifications* that enforce the model and make it robust against possible wrong future modifications (as in regression testing).

In addition to these redundant specifications, we also specified other properties that do not capture a specific requirement, but more general properties that should be guaranteed by the system. For example, we specified the following three properties:

- the direction indicators blink simultaneously only when the car is in hazard warning:

  g((blinkLeft != 0 and blinkLeftPulseRatio != NOPULSE and
  blinkRight != 0 and blinkRightPulseRatio != NOPULSE) =
  hazardWarningSwitchOn_Runn)

- if tail lamps are blinking, the car is not European:

  g((tailLampLeftStatus=BLINK or tailLampRightStatus=BLINK)
  implies marketCode!=EU)

- the market code of a car must always remain the same:

  forall $c **in** MarketCode **with**
  (marketCode = $c implies **g**(marketCode = $c))

All of the properties listed here (and also others) have been successfully verified by the verification tool AsmetaSMV, which output is shown in Code 16.

5.3 Model changes upon validation and verification

As described in Sect. 5.1, scenario-based validation with the validator `AsmetaV` was used to guide the development, and so constantly lead to adjustments of the developed models. However, also the other V&V approaches allowed us to discover errors for which we had to change our models.

For example, in the model *CarSystem003*, the rule r_Low_Beam_Head_lights decides when to switch the parking lights on (parkingLightON := true) based on some conditions. When modeling requirement ELS-46 in *CarSystem004* to switching off of the parking lights in case of subvoltage (parkingLightON := false), we generated a conflict (i.e., updating parkingLightON simultaneously to two different values) in particular cases (called *inconsistent update* in the ASM theory [26]). Such inconsistency was found by the model reviewer `AsmetaMA` (see Sect. 5.1) that gave, as a counterexample, the simulation sequence leading to such inconsistency. We then modified *CarSystem004* by introducing a guard that avoids the conflict. Note that all the scenarios were executed correctly on the first version of *CarSystem004*, showing that a single validation technique is usually not sufficient, and different complementary techniques should be used.

5.4 How verification capabilities influenced modeling

Verification solvers, such as model checkers or SMT solvers, usually do not support all the constructs/features provided by expressive modeling languages such as ASMs; for example, the model checker NuSMV, that is used as back-end tool of `AsmetaSMV` (see Sect. 5.2), only supports finite domains. Knowing these limits, one may upfront limit themselves only to the supported constructs/features, in order to be able to directly use such solvers. We believe that this approach, although sound, does not allow exploiting all the benefits that come from modeling (that are not only related to verification, but also to documentation, system understanding, requirements tracing, etc.).

Therefore, when writing the models, we did not consider the limitations coming from verification solvers, and we implemented the models in the most natural way, so to also guarantee qualitative properties such as the *readability* of the model [12]. Only when we had to perform model checking, we modified the models to remove unsupported features. The modification is performed in two phases. In the first phase, we apply the *flattener* [11] of the ASMETA toolset that automatically performs some transformations; specifically, it transforms all the transition rules (of any type) in the parallel execution of a set of suitable conditional rules; moreover, for each function location, all its possible evaluations are represented explicitly in the flattened model, by nesting them in proper conditional rules (refer to [11] for more details); note that these transformations have been proved to be sound, and so can always be applied automatically. In the second step, instead, we must manually apply other transformations, guaranteeing that they are sound. For example, we must transform the infinite domains in finite ones.

6 ELS system implementation

The ASMETA toolset supports the generation of executable C++ code by means of the tool `Asmeta2C++` [23]. For this project, we have extended the translator to support all the characteristics of the CarSystem specification. As the target platform, we have identified Arduino, a simple microcontroller which provides all the necessary features and is widely available.

Modularity in ASMETA is translated in C++ into multiple inheritance. In this way, a class will inherit all the functions defined in the imported modules. For instance, *CarSystem003HighBeam* which imports both *CarSystem003Domains* and *CarSystem002Functions* is translated into the following code:

```
class CarSystem003HighBeam:
public virtual CarSystem003Domains,
public virtual CarSystem002Functions {
...
};
```

Inputs and outputs are translated to electronic components in the implementation: the lights are simulated by LED lights, while monitored functions are implemented by switches and potentiometers (like the steeringAngle). The mapping from functions to hardware components is given in a json file and the necessary code is automatically generated.

At the time of developing the case study, *Time* was not explicitly treated in ASMETA, so we needed to implement a link between the time of the microcontroller (given by the function `millis()`) and all the monitored and controlled quantities that depend on time. The translator transforms each monitored boolean function passedXSec to a boolean variable, and introduces in Arduino a new long variable last_XSec and the code that:

resets the timer by setting the value of last_XSec to the current time;

checks the actual passed time and, if necessary, sets passedXSec to true.

For example, the passed_3sec is translated to

```
bool passed_3sec;
long last_3sec;
...
last_3sec = millis();
...
if (millis() - last_3sec > 3000)
    passed_3sec = true;
...
```

A similar approach is taken for lights blinking, which has been modeled by different values of the function blinkXPulseRatio defined on an enumerative set {NOPULSE, PULSE11, PULSE12}. The actual blinking is implemented by hand directly in the code that checks the value of blinkXPulseRatio and the current time and controls the lights by letting them blinking if necessary.

We were able to build a working Arduino prototype. A video of the implementation[13] is available. Although it is very simple and its main aim is the demonstration of a working system, its code could be used in a real implementation.

7 Other observations

We here provide a more detailed discussion on model changes due to requirements improvement between the version available at the conference time and the last version of the requirements documentation. We also underline possible tools improvement to deal with needs arisen from our experience in modeling and analyzing the case study. Finally, we add some observations on derivation of implementation from models.

7.1 Requirements ambiguities and model improvements

Modeling and validation activities revealed some statements where the requirements were wrong or ambiguous (these have been discussed in [13]). We had the possibility to check our doubts with the chairs (acting as domain experts) and we got corrected versions of the requirements.

The solution in [13] captures the version 1.9 of the requirements' specification document. We later updated our models to the last version 1.17 of the documentation, and the model updating was relative simple owing to the use of modules. For example, in version 1.11 the requirement ELS-40 specifies that "if the brake pedal is deflected more than 40.0°, all brake lamps flash" (in the previous versions it was required that only the third brake lamp flashed). To satisfy the new requirement, we have updated the rule r_EmergencyBrakeLights in module *CarSystem002EmergencyBrakeLights* as shown in Code 17. Further improvements of the requirements have been addressed similarly.

Most of the requirements updates have clarified the doubts, but in some cases they have created ambiguity, like, for example, the clarification of requirement ELS-48 in version 1.10 which created a conflict with requirement ELS-47. In this case, we kept the old version, which appeared more correct to us.

---

[13] https://foselab.unibg.it/asmeta/videos/carsystem_arduino.mp4.

```
//ELS-40 old version
macro rule r_EmergencyBrakeLights =
            par
             ...
               if brakePedal > 200 and brakeLampCenterStatus = FIX then
                        brakeLampCenterStatus := BLINK
               endif
            ...
            endpar

//ELS-40 V 1.11
macro rule r_EmergencyBrakeLights =
            par
                  ...
                  if brakePedal > 200 then
                        par
                              brakeLampCenterStatus := BLINK
                              tailLampLeftStatus := BLINK
                              tailLampRightStatus := BLINK
                        endpar
                  endif
               ...
            endpar
```

Code 17 Update of requirement ELS-40

7.2 Method and tools improvements

Requirements coverage As explained in Sect. 5.1, scenarios derived from the validation sequences turned out to be extremely useful, and we used them to guide the modeling. In order to check whether all the requirements are covered by the scenarios, we had to extend the validator `AsmetaV` with the ability to compute coverage at the level of the single rule; indeed, the original tool was able to report only coverage at the level of macro rule, but this is not sufficient to check if requirements are covered, as multiple requirements can be implemented in a given macro rule.

Scenario derivation and animation We have extensively used two model validation techniques, namely animation and scenario-based validation, which are supported by two nonintegrated tools. To save animation sessions in terms of scenarios, and also to animate existing scenarios, we have extended the animator [22] which now permits exporting animated sequences into `Avalla` scripts. These can be re-executed to validate the model when changing it (in a kind of regression testing using "record and replay").

Models for formal verification As described in Sect. 5.3, doing formal verification requires the user to manually adapt the models to remove unsupported features, such as infinite domains. As future work, we plan to make this approach semiautomatic, by suggesting to the user possible sound transformations. A different approach would to use, as back-end solvers, infinite state model checkers such as nuXmv[14] that accepts infinite domains. However, that would come at the cost of the class of properties that we can verify.

Time modeling The case study presented in this paper relies on time constraints, as well as many real systems. In this specification (see "Time pattern" in Sect. 4.1), we dealt with the time in terms of suitable monitored functions capturing the time elapsed, as is usual in many ASM specifications [26, 28]. Afterwards, we have worked on the introduction of time in ASMETA models. We have implemented the library TimeLibrary that introduces *time* as a special monitor function to manage the time. Moreover, the user can define his timers and the library provides functions and rules to operate on timers, like to check if a desired amount of time is passed, to reset and start a timer, and to set the timer duration and time unit. More details about the use of TimeLibrary can be found in [21].

7.3 Derivation or verification of software implementation

Sect. 6 presents how we are able to translate the ASMETA specification into executable C++ code for Arduino by using the `Asmeta2C++` tool [23]. With minimal adjustments for the links with HW sensors and actuators, this code could be deployed on the real microcontroller with the assurance that its behavior conforms to the specification and all the safety properties are preserved. This is a typical Model Driven Engineering (MDE) approach that aims at exploiting models also during the implementation phase, through a set of suitable transformations like ours.

8 Other case study solutions and related work

In this section, we compare the presented solution of the case study with others proposed in the literature, and we list other case studies where ASMETA has been applied. Moreover, we recall other currently supported tools of the ASM method.

Comparison with other case study solutions We here describe other existing formal specification and analysis solutions for the same case study. Table 2 summarizes the main approaches, including our solution, and compares them according to the following characteristics:

- Adopted modeling formalisms and analysis tools;
- Target control features/subsystems;
- Model scope or purpose (V&V, implementation/prototyping, etc.);
- Distinctive modeling and analysis strategy.

The approach by Cunha et al. [29] adopts variants in Electrum, a lightweight formal specification language that extends Alloy with mutable relations and temporal logic to

---

[14] https://nuxmv.fbk.eu/.

Table 2 A comparison with other case study solutions

Approach	Formalisms & techniques/tools	Target control features/subsystems	Model purpose	Distinctive strategy
Cunha et al. [29]	• Electrum • variants, mutable relations and linear temporal logic	ELS subsystem with multiple variants	• V&V of functional requirements of system model variants • scenario animation	feature-oriented design
Leuschel et al. [35]	• Classical B for modeling • Event-B in Rodin for proof • VISB for graphical model visualization	subsets of ELS and SCS subsystems	• verification of functional requirements • visual model animation	compositional and modular system modeling via machine inclusion and operation calls
Mammar et al. [38]	• Event-B • stepwise refinement with RODIN provers	ELS subsystem	• modeling and verification of functional requirements • validation by scenarios • model refinement	Proof reuse
Mammar et al. [37]	• Event-B • stepwise refinement with RODIN provers	SCS subsystem	• modeling and verification of functional requirements • validation by scenarios • model refinement	Proof reuse
Krings et al. [34]	• low-level implementation with MISRA C • CBMC model checker	ELS and SCS state machines	Verified low-level implementation in C	test-driven development and mocking of test objects
(this work)	• ASMs • ASMETA tools for specification, analysis and prototyping	• ELS subsystem • SCS subsystem	• V&V of functional requirements • scenario-based animation • (correct-by-construction) prototyping in the Arduino board	Modular and idioms-based design

model the ELS subsystem. They explored different strategies to address variability, using pure Electrum and also an annotative language extension. Their model abstracts real-time aspects and the integer nature of the signals, and no particular duration is imposed to states.

Leuschel et al. [35] modeled and verified parts of the automotive case study using the classical B in the early phases of software modeling, while for proving the system is safe and functionally correct they have used Event-B for proof as supported by the platform Rodin. They adopted the classical B's machine inclusion mechanism along with operation calls for applying a compositional and modular modeling strategy. A recent graphical model visualization engine, called VISB, is also used to have a lightweight visualization with off-the-shelf images enriched with formal model expressions updated according to the current state of the formal model. They have modeled time as a discrete integer variable representing elapsed time in milliseconds.

Mammar et al. [38] provided an Event-B model of the ELS subsystem via stepwise refinement. The model has been validated using PROB by scenarios, and all proof obligations have been discharged using the RODIN provers. To speed up the proof phase, they have included in the guards of the model some properties tagged as theorems in order to prove them only once and reuse them in all the proofs. A discrete integer variable is used to model the time progression together with event progress. Some of the same authors also modeled the SCS subsystem using a similar stepwise refinement approach with Event-B and the Rodin provers [37].

Rather than adopting a (correct-by-construction) specification approach with a formal method, Krings et al. [34] used MISRA C, a language used in the automotive industry, for providing directly a low-level implementation for the case study. They used a test-driven development for validation by setting up the validation sequences as unit tests first, and only at the end applied model checking using CBMC for verifying different properties directly on the C code. Their verification approach has a very limited support for temporal properties, and they do not verify time properties aside from simulating an external clock in the test cases.

The aforementioned approaches are tailored to model, validate, verify, and coding components/subsystems of the case study using different state-based formalisms, methods, and strategies. They, somehow, complement each other and could be seamlessly integrated along a formal development process. Unfortunately, the abstracted time model adopted by most of the solutions limits reasoning about real-time requirements. Moreover, such state-based formal methods are able to model discrete-event systems and not continuous systems. So they provide limited support for modeling real-world dynamic systems where continuously changing state variables of a system are typically modeled by differential equations. Related to ASMs, the concept of *Continuous ASM* was introduced in [16] for the controller synthesis problem, but this theory is not supported yet by ASMETA. For the purpose of the case study, however, some form of discretization was adopted, such as the formulas provided in the requirement document for the characteristic curves of the high beam headlight illumination distance and strength depending on the vehicle speed as shown in Code 9. Both functions are computed using the value returned by the discretized function percentageHBM, modeled as ASMETA static function.

Other related work ASMETA is an active open-source academic project. Over the years, it has been improved with techniques and tools to face the challenges of modern systems. Details on latest advancements can be found in [15].

Other case studies to which ASMETA has been applied, include systems in the context of medical devices (PillBox [18], hemodialysis device [10], amblyopia diagnosis [4], PHD Protocol [19], Mechanical Ventilator Milano [20]), software control systems (Landing Gear System [5], Hybrid European Rail Traffic Management System [31]), cloud- [8] and service-based systems [39, 40], self-adaptive systems [9, 14, 24]. Application domains under current investigations are those of IoT security, autonomous and evolutionary systems, cyber-physical systems, and medical software certification.

Besides ASMETA tools, there are others that have been developed over the years around the ASMs. Among those, CoreASM [30] and CASM [36] are currently actively supported, while others, as the AsmGofer [41], are dead projects. CoreASM was designed and developed at the same time when the ASMETA project started and shares with ASMETA the same overall goals of being a toolset for ASM editing, simulation, and verification. Although the design choices of the two toolsets are different (ASMETA has been developed by exploiting the Model-driven Software Engineering (MDE) approach stating from developing a metamodel for the abstract syntax of an ASM model, while CoreASM was developed as extension plugins around a core kernel of the language), the similarities between ASMETA and CoreASM models are so close that a tentative was done in the past to define a unique syntax for both environments [6]. CASM arrived later, due to its authors' intention to provide the ASMs with faster and better performing tool support, which could be more adequate to handle industry-size applications. The CASM language was inspired by the CoreASM language, but it is a static strong inferred typed language. The application of CASM to academic or industrial case studies is still very limited and it is not possible to evaluate the reals advantages it offers with respect to ASMETA. Our choice to use ASMETA for the specification of the case study is mainly due to the fact that it is an in-house toolset developed by the authors of the paper, and because of the variety of tools provided for different phases (design, development, and operation) and that allowed us the possibility to develop a prototype in Arduino.

9 Conclusions

We presented our experience in applying a formal engineering method based on ASMs and the toolset ASMETA for the specification, validation & verification, and prototyping in Arduino of an automotive system with adaptive control features for exterior lights and speed of modern cars. We discussed our modeling approach and underlying strategies, and analysis techniques to face system complexity and peculiarities. We also provided strengths and weaknesses of our approach throughout the paper, and compared it with other related case study solutions according to specific criteria.

This case study provided us a further opportunity to test our approach in terms of robustness, scalability with the increasing complexity of the models and validation scenarios, and usability of ASMETA tools for complex systems. Further details about our efforts in addressing usability in the ASMETA toolset, a critical review of what has been more successful and what less based on our previous developed case studies (but still valid for that proposed here), and other directions that could further increase the adoption of the proposed method can be found in [12].

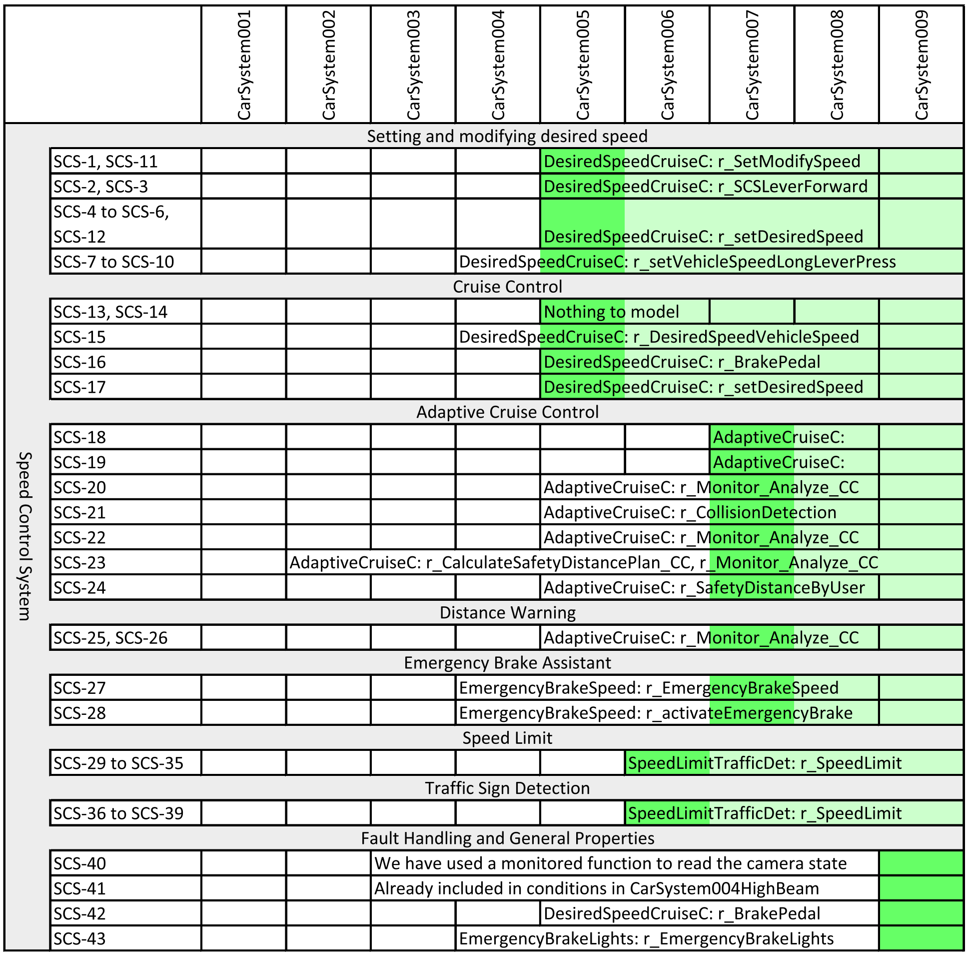
Appendix A: Traceability tables

In this appendix, we report the traceability tables used during models implementation, to map requirements with models and rules.

Table 3 Traceability table between requirements and rules/functions of ELS subsystem

		CarSystem001	CarSystem002	CarSystem003	CarSystem004	CarSystem005	CarSystem006	CarSystem007	CarSystem008	CarSystem009
Exterior Light System	Direction Blinking									
	ELS-1, ELS-5	Blink: r_RunDirectionBlinking, r_CompleteFlashingCycle							🟩	🟩
	ELS-2, ELS-4	Blink: r_CompleteFlashingCycle			🟩				🟩	🟩
	ELS-3, ELS-7	Blink: r_DirectionBlinking			🟩				🟩	🟩
	ELS-6	Blink:r_BlinkLeft, r_BlinkRight			🟩				🟩	🟩
	Hazard Warning									
	ELS-8	HW: r_AdaptHazardWarningPulseRatio, r_StartHazardWarning							🟩	🟩
	ELS-9	HW: r_AdaptHazardWarningPulseRatio							🟩	🟩
	ELS-10	Domain: PulseRatio = {NOPULSE | PULSE11 | PULSE12}							🟩	🟩
	ELS-11	HW: r_HazardWarningLight			🟩				🟩	🟩
	ELS-12, ELS-13	HW: r_SavePitmanArmUpDownReq							🟩	🟩
	Low Beam Headlights Cornering Light									
	ELS-14 to ELS-19		LowBeam: r_LowBeamHeadlights						🟩	🟩
	~~ELS-20~~	| Deleted requirement |								
	ELS-21		LowBeam: ambientLightingAvailable						🟩	🟩
	ELS-22		LowBeam: r_LowBeamTailLampOnOff						🟩	🟩
	ELS-23		LowBeam: tailLampLeft, tailLampRight						🟩	🟩
	ELS-24		Cornering: r_CorneringLightsOn. r_CorneringLights						🟩	🟩
	ELS-25 to ELS-27		Cornering: r_CorneringLights						🟩	🟩
	ELS-28		LowBeam: r_parkingLight, r_LowBeamHeadlights						🟩	🟩
	ELS-29		LowBeam: r_LowBeamHeadlights						🟩	🟩
	Manual Hight Beam Headlights									
	ELS-30, ELS-31			HighBeam: r_Manual_high_beam_headlights					🟩	🟩
	Adaptive High Beam Headlights									
	ELS-32			HighBeam: adaptiveHighBeamActivated					🟩	🟩
	ELS-33		HighBeam: r_Execute_HBH, r_IncreasingPlan_HBH, r_Monitor_Analyze_HBH							🟩
	ELS-34			HighBeam: r_DecreasingPlan_HBH, r_Monitor_Analyze_HBH						🟩
	ELS-35			HighBeam: r_Monitor_Analyze_HBH					🟩	🟩
	ELS-36			HighBeam: lightIlluminationDistance					🟩	🟩
	ELS-37			HighBeam: calculateSpeed					🟩	🟩
	ELS-38			HighBeam: adaptiveHighBeamDeactivated					🟩	🟩
	Emergency Brake Light									
	ELS-39, ELS-40		EmergencyBrakeLights: r_EmergencyBrakeLights						🟩	🟩
	Reverse Light									
	ELS-41		Cornering: r_ReverseLight						🟩	🟩
	Fault Handling									
	ELS-42				HighBeam: adaptiveHighBeamActivated				🟩	🟩
	ELS-43				HighBeam: adaptiveHighBeamDeactivated				🟩	🟩
	ELS-44				LowBeam: ambientLightingAvailable				🟩	🟩
	ELS-45				CorneringLights: r_CorneringLights				🟩	🟩
	ELS-46				LowBeam: r_parkingLight				🟩	🟩
	ELS-47		Functions: setOverVoltageValueLight, setOverVoltageValueHighBeam							🟩
	ELS-48				Nothing to model				🟩	🟩
	ELS-49				HighBeam: adaptiveHighBeamDeactivated				🟩	🟩

Table 4 Traceability table between requirements and rules/functions of SCS subsystem



		CarSystem001	CarSystem002	CarSystem003	CarSystem004	CarSystem005	CarSystem006	CarSystem007	CarSystem008	CarSystem009
Speed Control System	Setting and modifying desired speed									
	SCS-1, SCS-11					DesiredSpeedCruiseC: r_SetModifySpeed				
	SCS-2, SCS-3					DesiredSpeedCruiseC: r_SCSLeverForward				
	SCS-4 to SCS-6, SCS-12					DesiredSpeedCruiseC: r_setDesiredSpeed				
	SCS-7 to SCS-10				DesiredSpeedCruiseC: r_setVehicleSpeedLongLeverPress					
	Cruise Control									
	SCS-13, SCS-14					Nothing to model				
	SCS-15				DesiredSpeedCruiseC: r_DesiredSpeedVehicleSpeed					
	SCS-16					DesiredSpeedCruiseC: r_BrakePedal				
	SCS-17					DesiredSpeedCruiseC: r_setDesiredSpeed				
	Adaptive Cruise Control									
	SCS-18							AdaptiveCruiseC:		
	SCS-19							AdaptiveCruiseC:		
	SCS-20					AdaptiveCruiseC: r_Monitor_Analyze_CC				
	SCS-21					AdaptiveCruiseC: r_CollisionDetection				
	SCS-22					AdaptiveCruiseC: r_Monitor_Analyze_CC				
	SCS-23		AdaptiveCruiseC: r_CalculateSafetyDistancePlan_CC, r_Monitor_Analyze_CC							
	SCS-24					AdaptiveCruiseC: r_SafetyDistanceByUser				
	Distance Warning									
	SCS-25, SCS-26					AdaptiveCruiseC: r_Monitor_Analyze_CC				
	Emergency Brake Assistant									
	SCS-27				EmergencyBrakeSpeed: r_EmergencyBrakeSpeed					
	SCS-28				EmergencyBrakeSpeed: r_activateEmergencyBrake					
	Speed Limit									
	SCS-29 to SCS-35						SpeedLimitTrafficDet: r_SpeedLimit			
	Traffic Sign Detection									
	SCS-36 to SCS-39						SpeedLimitTrafficDet: r_SpeedLimit			
	Fault Handling and General Properties									
	SCS-40			We have used a monitored function to read the camera state						🟩
	SCS-41			Already included in conditions in CarSystem004HighBeam						🟩
	SCS-42					DesiredSpeedCruiseC: r_BrakePedal				🟩
	SCS-43				EmergencyBrakeLights: r_EmergencyBrakeLights					🟩

Acknowledgements We thank the student Giorgio Brugali who performed the translation from the ASMETA models to Arduino code and realized the Arduino implementation.

Funding Open access funding provided by Università degli studi di Bergamo within the CRUI-CARE Agreement. P. Arcaini is supported by MIRAI Engineerable AI Project (No. JPMJMI20B8), JST; and ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST, Funding Reference number: 10.13039/501100009024 ERATO.

References

1. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Proc. Of the Second International Conference on Abstract State Machines, Alloy, B and Z, ABZ'10, pp. 61–74. Springer, Berlin (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: Muñoz, C. (ed.) Proc. Of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp. 4–13, NASA, Langley Research Center, Hampton (2010)
3. 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). https://doi.org/10.1002/spe.1019
4. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Formal validation and verification of a medical software critical component. In: ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015, pp. 80–89. IEEE, Los Alamitos (2015). https://doi.org/10.1109/MEMCOD.2015.7340473
5. Arcaini, P., Gargantini, A., Riccobene, E.: Rigorous development process of a safety-critical system: from ASM models to Java code. Int. J. Softw. Tools Technol. Transf. **19**(2), 247–269 (2015). https://doi.org/10.1007/s10009-015-0394-x
6. Arcaini, P., Bonfanti, S., Dausend, M., Gargantini, A., Mashkoor, A., Raschke, A., Riccobene, E., Scandurra, P., Stegmaier, M.: Unified syntax for abstract state machines. In: Butler, M., Schewe,

K.D., Mashkoor, A., Biro, M. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23–27, 2016. Proceedings, vol. 9675, pp. 231–236. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_14

7. Arcaini, P., Gargantini, A., Riccobene, E.: SMT-based automatic proof of ASM model refinement. In: De Nicola, R., Kühn, E. (eds.) Software Engineering and Formal Methods, pp. 253–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_17
8. Arcaini, P., Holom, R.M., Riccobene, E.: ASM-based formal design of an adaptivity component for a cloud system. Form. Asp. Comput. **28**(4), 567–595 (2016). https://doi.org/10.1007/s00165-016-0371-5
9. 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). https://doi.org/10.1145/3019598
10. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Integrating formal methods into medical software development: the ASM approach. Sci. Comput. Program. **158**, 148–167 (2018). https://doi.org/10.1016/j.scico.2017.07.003
11. Arcaini, P., Melioli, R., Riccobene, E.: AsmetaF: a flattener for the ASMETA framework. In: Masci, P., Monahan, R., Prevosto, V. (eds.) Proc. 4th Workshop on Formal Integrated Development Environment, Oxford, England, 14 July 2018, Electronic Proceedings in Theoretical Computer Science, vol. 284, pp. 26–36. Open Publishing Association (2018). https://doi.org/10.4204/EPTCS.284.3
12. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Addressing usability in a formal development environment. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Formal Methods. FM 2019 International Workshops, pp. 61–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_6
13. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an automotive software-intensive system with adaptive features using ASMETA. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 302–317. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_25
14. Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P.: MSL: a pattern language for engineering self-adaptive systems. J. Syst. Softw. **164**, 110558 (2020). https://doi.org/10.1016/j.jss.2020.110558
15. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA approach to safety assurance of software systems. In: Raschke, A., Riccobene, E., Schewe, K.D. (eds.) Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday, pp. 215–238. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
16. Banach, R., Zhu, H., Su, W., Wu, X.: Asm, controller synthesis, and complete refinement. Sci. Comput. Program. **94**, 109–129 (2014). https://doi.org/10.1016/j.scico.2014.04.013
17. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Log. **4**, 578–651 (2003). https://doi.org/10.1145/937555.937561
18. Bombarda, A., Bonfanti, S., Gargantini, A.: Developing medical devices from abstract state machines to embedded systems: a smart pill box case study. In: Mazzara, M., Bruel, J.M., Meyer, B., Petrenko, A. (eds.) Software Technology: Methods and Tools, pp. 89–103. Springer, Cham (2019)
19. Bombarda, A., Bonfanti, S., Gargantini, A., Radavelli, M., Duan, F., Lei, Y.: Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using abstract state machines. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) Testing Software and Systems, pp. 67–85. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31280-0_5
20. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Developing a prototype of a mechanical ventilator controller from requirements to code with ASMETA. In: Gleirscher, M., van de Pol, J., Woodcock, J. (eds.) Proc. First Workshop on Applicable Formal Methods, AppFM@FM 2021, Virtual, 23rd November 2021, EPTCS, vol. 349, pp. 13–29 (2021). https://doi.org/10.4204/EPTCS.349.2
21. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E.: Extending ASMETA with time features. In: Raschke, A., Méry, D. (eds.) Rigorous State-Based Methods, pp. 105–111. Springer, Cham (2021)
22. Bonfanti, S., Gargantini, A., Mashkoor, A.: AsmetaA: animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp. 369–373. Springer, Cham (2018)
23. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from abstract state machines specifications. J. Softw. Evol. Process **32**(2), e2205 (2020). https://doi.org/10.1002/smr.2205
24. Bonfanti, S., Riccobene, E., Scandurra, P.: A runtime safety enforcement approach by monitoring and adaptation. In: Biffl, S., Navarro, E., Löwe, W., Sirjani, M., Mirandola, R., Weyns, D. (eds.) Software Architecture, pp. 20–36. Springer, Cham (2021)
25. Börger, E.: The ASM refinement method. Form. Asp. Comput. **15**, 237–257 (2003)
26. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Berlin (2018). https://doi.org/10.1007/978-3-662-56641-1
27. Börger, E., Schewe, K.: Concurrent abstract state machines. Acta Inform. **53**(5), 469–492 (2016). https://doi.org/10.1007/s00236-015-0249-7
28. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Berlin (2003)
29. Cunha, A., Macedo, N., Liu, C.: Validating multiple variants of an automotive light system with electrum. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 318–334. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_26
30. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: an extensible ASM execution engine. Fundam. Inform. **77**(1–2), 71–103 (2007)
31. Gaspari, P., Riccobene, E., Gargantini, A.: A formal design of the hybrid European rail traffic management system. In: Proc. Of the 13th European Conference on Software Architecture, ECSA'19, vol. 2, pp. 156–162. Assoc. Comput. Mach., New York (2019). https://doi.org/10.1145/3344948.3344993
32. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 281–301. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_24
33. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
34. Krings, S., Körner, P., Dunkelau, J., Rutenkolk, C.: A verified low-level implementation of the adaptive exterior light and speed control system. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 382–397. Springer, Cham (2020)
35. Leuschel, M., Mutz, M., Werth, M.: Modelling and validating an automotive system in classical B and Event-B. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 335–350. Springer, Cham (2020)
36. Lezuo, R., Paulweber, P., Krall, A.: CASM – optimized compilation of abstract state machines. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2014, pp. 13–22. ACM, New York (2014)

37. Mammar, A., Frappier, M.: Modeling of a speed control system using Event-B. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 367–381. Springer, Cham (2020)
38. Mammar, A., Frappier, M., Laleau, R.: An Event-B model of an automotive adaptive exterior light system. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods, pp. 351–366. Springer, Cham (2020)
39. Mirandola, R., Potena, P., Riccobene, E., Scandurra, P.: A reliability model for service component architectures. J. Syst. Softw. **89**, 109–127 (2014). https://doi.org/10.1016/j.jss.2013.11.002
40. Riccobene, E., Scandurra, P.: Towards ASM-based formal specification of self-adaptive systems. In: Ait Ameur, Y., Schewe, K.D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp. 204–209. Springer, Berlin (2014)
41. Schmid, J.: Compiling abstract state machines to C++. In: Moreno-Dìaz, R., Quesanda-Arencibia, A. (eds.) Formal Methods and Tools for Computer Science, Eurocast (2001). Universidad de Las Palmas de Gran Canaria. Extended Abstract
42. Sommerville, I.: Software Engineering, 9th edn. Addison-Wesley, Reading (2010)