Heinrich Heine
Universität
Düsseldorf

A verified low-level implementation and visualization of the adaptive exterior light and speed control system

Sebastian Krings, Philipp Körner, Jannik Dunkelau & Kristin Rutenkolk

Article - Version of Record

Suggested Citation:
Krings, S., Körner, P., Dunkelau, J., & Rutenkolk, K. (2024). A verified low-level implementation and visualization of the adaptive exterior light and speed control system. International Journal on Software Tools for Technology Transfer, 26(3), 403–419. https://doi.org/10.1007/s10009-024-00750-5

Wissen, wo das Wissen ist.

UNIVERSITÄTS- UND
LANDESBIBLIOTHEK
DÜSSELDORF

This version is available at:

URN: https://nbn-resolving.org/urn:nbn:de:hbz:061-20250121-103017-9

Terms of Use:

This work is licensed under the Creative Commons Attribution 4.0 International License.

For more information see: https://creativecommons.org/licenses/by/4.0

International Journal on Software Tools for Technology Transfer (2024) 26:403–419
https://doi.org/10.1007/s10009-024-00750-5

GENERAL

Special Section: ABZ 2020/2021

A verified low-level implementation and visualization of the adaptive exterior light and speed control system

Sebastian Krings[1] · Philipp Körner[2] · Jannik Dunkelau[2] · Kristin Rutenkolk[2]

Accepted: 29 April 2024 / Published online: 27 May 2024
© The Author(s) 2024

Abstract

In this article, we present an approach to the ABZ 2020 case study that differs from those usually presented at ABZ: Rather than using a (correct-by-construction) approach following a formal method, we use C for a low-level implementation instead. We strictly adhere to test-driven development for validation, and only afterwards apply model checking using CBMC for verification. While the approach has several benefits compared to the more rigorous approaches, it also provides less mathematical clarity and overall less thorough verification. In consequence, our realization of the ABZ case study serves as a baseline reference for comparison, allowing to assess the benefit provided by the various formal modeling languages, methods and tools.

Keywords Model checking · Formal methods · Verification · Case study · Test-driven development · MISRA C

1 Introduction

The ABZ 2020 Case Study [28] describes two assistants commonly found in modern cars. The overall system consists of two loosely coupled components, the adaptive exterior light system (ELS) and the speed control system (SCS). The ELS controls head- and taillights, while the SCS controls the vehicle's speed. Both have to take into account the environment and parameters defined by the driver. Obviously, both are safety critical components, rendering safety and security a development priority.

In this article, we present our implementation of ELS and SCS. Our approach differs from the other case study implementations in that we do not employ a fully formal development method. Instead, we attempted an approach closer to what might happen in industry, where formal methods are not common, yet. To do so, we implemented both the ELS and the SCS directly in (MISRA) C, following a test-driven development workflow. Only afterwards, we attempted formal verification directly on the C code, using the CBMC model checker [18]. Both MISRA C and CBMC will be introduced more thoroughly in Sects. 2.1 and 6.2, respectively. Test-driven development and mocking of test objects will be presented in Sect. 2.2.

Rationale Sometimes formal methods' practitioners claim to hold a high ground over "traditional" software development or at least claim that there rarely are disadvantages [11, 24]. The argument seems convincing; yet, only a few case studies have compared two teams working on the same project, one employing a formal approach and the other working "traditionally".

One such study has been performed by Fitzgerald et al. [12, 22] in cooperation with British Aerospace Systems and Equipment Ltd. The authors show that the teams behave and develop differently, i.e., they focus on different areas of the system in development. While the formal approach used in the study shows its merits, the authors also identify drawbacks due to the so far unknown methodology and a requirement for training.

Furthermore, Larsen et al. [35] have discussed how to integrate formal methods in a step-wise fashion. Comparable to our approach, the authors incrementally add formal aspects to an already existing development process. Their incremental approach is also verified by comparing two groups of

✉ P. Körner
p.koerner@hhu.de

S. Krings
sebastian.krings@hhu.de

J. Dunkelau
jannik.dunkelau@hhu.de

K. Rutenkolk
kristin.rutenkolk@hhu.de

[1] Independent researcher, Düsseldorf, Germany

[2] Institut für Informatik, Heinrich-Heine-Universität, Universitätsstr. 1, D-40225 Düsseldorf, Germany

developers, one working with traditional tools for requirement analysis and one working with the workflow supported by formal specification. Again, formal methods proof to be beneficial if integrated into an existing workflow both deliberately and thoroughly.

For this case study, we aim at providing a baseline for comparison with fully formal approaches or other approaches combining formal and informal verification, e.g., as suggested for spacecrafts [43].

We opted to postpone verification as much as possible, to allow a fair evaluation of (dis-)advantages of the individual approaches. Our aim is to examine whether a rigorous approach is beneficial in the context of the case study. If so, we hope to add to the body of evidence that formal methods actually *are* beneficial compared to "traditional" software development.

Distinctive features Several features render our approach unique: Firstly, as the implementation is written in C, it could be directly deployed to an embedded system. Models written in formal specification languages would have to be refined to an implementation level before code can be generated. Furthermore, code generators usually are not proven and might introduce new errors. In cases where code generation is not easily applicable, side-by-side development of code is suggested. However, this approach is error-prone as well.

Secondly, the implementation is close to actual hardware. Code that interacts with sensors or user input is separated, i.e., it could immediately be linked to real sensors. Additionally, our implementation makes use of threads, just as the subcomponents of the system would run in parallel. We expect that most specifications using formal methods simply allow some nondeterminism concerning the ordering of state transitions.

In consequence, our implementation allows real-time simulation of the system, whereas state transitions in formal methods usually happen instantly and do not amount for any time elapsed. This also allows usage of our implementation for hardware-in-the-loop tests, which are common for automotive software [21, 41].

Thirdly, C together with the MISRA rules restricting its usage stems from the automotive industry and is widely used in practice. Thus, our implementation closely mimics real-world development conditions.

Team overview Our team comes from a formal methods background: While all members are very familiar with the B method [1, 2], we did not have particular expertise with C development or verification tooling for C. The basic code structure and the fixture for the test scenarios were developed by SK and PK in a synchronous meeting. Afterwards, SK implemented the ELS, JD was responsible for the SCS, and tests were provided by PK and KR. Formal verification was done by SK and PK. Both the ad-hoc and 3D visualization was provided by KR due to her individual knowledge in this area.

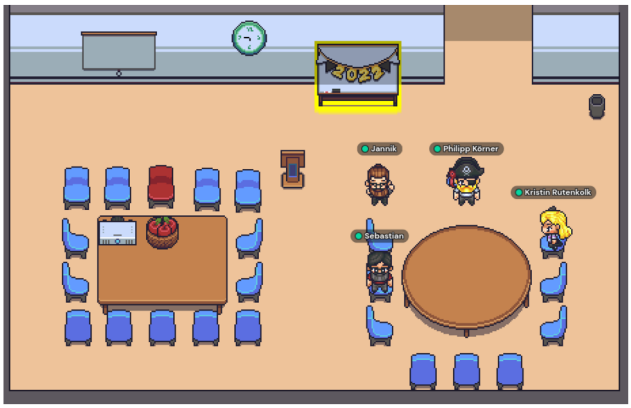

Fig. 1 Meeting in a Virtual Seminar Room. The added avatars and higher immersion are supposed to increase collaboration during the COVID-19 pandemic

Collaboration We believe the ways and techniques for collaboration can influence the overall quality of the results produced by a team of developers, especially when it comes to safety critical software. As we were working from different locations, we used asynchronous messaging via Mattermost[1] for coordination and progress reports. The code was version-controlled using Git and GitHub. Finally, during the COVID-19 pandemic, we additionally employed the software Gather[2] for synchronous meetings discussing progress and next steps. In Gather, one controls an avatar through a virtual world, and video conferences are started automatically based on predefined rooms or proximity. A screenshot of the authors meeting in Gather is shown in Fig. 1.

Additional contributions This article is based on our case study submission [34] and extends it by

- a discussion on how the given requirements are represented and how far we can trace the impact of requirements on the implementation,
- a thorough presentation of our approach to development,
- improved visualization,
- additional information on the development team, as well as the tools and techniques used,
- an evaluation of readability and comprehensibility of our implementation, and
- a comparison to the other case studies.

[1] https://mattermost.com/.

[2] https://www.gather.town/.

2 Background on used methodology and tools

Our implementation complies with MISRA C and was developed in a test-driven manner. Afterwards, CBMC was employed for formal verification. Below, we briefly introduce these methods and tools.

Since the actual code was C, we were also able to make full use of common development tools and for example individually choose a preferred IDE or Editor, such as Qt Creator and Visual Studio Code.

2.1 MISRA C

MISRA C is a set of development and style guidelines for C, introduced by MISRA, the Motor Industry Software Reliability Association. The standard [40] defines a subset of C meant to be used for safety critical systems, in particular in the automotive sector. In fact, both ISO 26262 [29] and the software specification by AUTOSAR [23] suggest the usage of MISRA C for automotive applications.

The overall goal of MISRA C is to increase both safety and security by avoiding common pitfalls. Thus, the rules prohibit or discourage the use of unsafe constructs, try to avoid ambiguities, and so on. The MISRA C standard distinguishes between three kinds of rules: those that are mandatory, those that are required but could be ignored if a rationale is given, and rules that are advisory only. For instance, there is a required rule stating that any switch statement should have a default label and a mandatory rule stating that any path through a nonvoid function should end in a return statement.

While most rules could be checked by hand, we used cppcheck[3] to verify compliance of our implementation. Given that some rules are undecidable, the result is only an indication and manual review is required as well.

Despite its prevalence, MISRA C has been criticized regarding both efficiency and ease of use. In particular, the possibilities of false positives [27] and of introducing new errors by (thoughtlessly) changing code to adhere to the rules [9] should be carefully considered. Despite the criticism, MISRA C remains the de facto standard in the automotive industry and is used throughout all production code in this case study.

2.2 Test-driven development and mocking

Test-driven Development is an approach to software development that follows a certain development cycle: before implementing a new feature or fixing an issue, an appropriate test case is formulated and executed [7]. Without code change, the test is expected to fail.

[3] http://cppcheck.sourceforge.net.

Afterwards, the code is extended and improved to make the test pass. As a result, a high test coverage and resulting confidence is achieved. Furthermore, the test suite helps during later refactorings.

To simplify formulating tests and to allow testing program parts in isolation, mocks can be used. A mock is an object or library that simulates the input and output behavior of program parts [7]. However, rather than implementing the full functionality, mocks are usually much simpler than the code they replace. For instance, mocks often behave deterministically or even to provide constant outputs. For testing purposes, mocks can record their inputs and provide them to assertions.

2.3 CBMC

CBMC [18] is a model checker for programs written in C. It uses bounded model checking [8] to verify a default set of properties, mostly related to common programming errors, such as:

- memory safety, including bounds checks and pointer safety,
- occurrence and treatment of exceptions, and
- absence of undefined behavior due to C quirks.

While those are worthwhile to find and correct, they only ensure general correctness but not adherence to the requirements.

To check individual properties, CBMC can be used to verify user-given assertions stated as C-style assertions using the macros in `assert.h`.

3 Case study overview

The proposed case study for the ABZ 2020 [28] concerns itself with two software-realized assistant systems from the automotive domain, namely an adaptive exterior light system (ELS) and a speed control system (SCS). The two software systems are parameterized with configuration options that account for different vehicle specifications either based on country-specific regulations or individual orders. These parameters are:

- an indicator on which side the driver seat is placed to distinguish between left- or right-hand traffic,
- a market code specifying whether the car was build for the USA, Canada, or the EU, and
- the information whether the vehicle is armored,

and are referenced throughout the listed requirements. While these parameters are shared by both subsystems, the case study specification defines further sensors, actuators, user interface, and functional requirements for the ELS and the SCS individually.

3.1 Adaptive exterior light system

The ELS specification concerns itself with the exterior lighting of the vehicle. This includes low and high beam headlights, a cornering light when turning left or right, the turn signal, and the emergency brake light. The driver controls it via a rotary switch, a control lever (referred to as pitman arm), a hazard warning light switch, and a darkness switch (which only exists in armored vehicles).

The ELS has 49 functional requirements specified (ELS-1 to ELS-49) which are grouped into 8 categories:

- Direction blinking (ELS-1 to ELS-7),
- Hazard warning light (ELS-8 to ELS-13),
- Low beam headlights and cornering light (ELS-14 to ELS-29),
- Manual high beam headlights (ELS-30 to ELS-31),
- Adaptive high beam headlights (ELS-32 to ELS-38),
- Emergency brake light (ELS-39 to ELS-30),
- Reverse light (ELS-41), and
- Fault handling (ELS-42 to ELS-49).

Exemplary specifications include duration of flashing light cycles (ELS-10: flashing cycle duration is 1 second), situational light activations (ELS-27: in reverse gear, both cornering lights are active), default brightness factors (ELS-29: 100% for all lights), or behavior based on time-sensitive user input (ELS-2 vs. ELS-4: different flashing behavior when holding the pitman arm for less than 0.5 seconds or longer).

3.2 Speed control system

The specification for the SCS outlines behavior and functionality for a cruise control and an adaptive cruise control, a distance warning, emergency brake assist, a speed limit, traffic sign recognition, and traffic jam following. The driver interacts with the SCS by means of a cruise control lever, the brake pedal, the gas pedal, and the instrument cluster. Meanwhile the vehicle can give feedback and notifications by means of a visual and an acoustic signal.

The SCS comprises 43 requirements (SCS-1 to SCS-41) grouped into 8 categories:

- Setting and modifying desired speed (SCS-1 to SCS-12),
- Cruise control (SCS-13 to SCS-17),
- Adaptive cruise control (SCS-18 to SCS-24),
- Distance warning (SCS-15 and SCS-26),
- Emergency brake assistant (SCS-27 and SCS-28),
- Speed limit (SCS-29 to SCS-35),
- Traffic Sign Detection (SCS-36 to SCS-39), and
- Fault handling and general properties (SCS-40 to SCS-43).

Exemplary specifications include behavior changes of the SCS given user input (SCS-4: pushing lever above 7° point increases desired speed by 10 km/h) which might be time-sensitive (SCS-8: holding lever above 7° point for 2 seconds increases speed every 2 seconds to next ten's place), reactions of the SCS to braking (SCS-16: turn cruise control off when brake is pushed), or reaction to predicted collisions (SCS-28: if time to impact is less than time for standstill, activate brake at 100%).

4 Requirements and modeling strategy

In this section, we give an overview on how we transformed the requirements into code and test, our validation strategy and the limitations of our implementation.

4.1 Process from requirements to code and assertions

We used the requirements given in the case study description without further modification or transformation. For each requirement we covered, we generated:

- Unit tests, which are used for test-driven development. See Sect. 6.1 for details.
- Assertions to be checked via CBMC as presented in Sect. 6. These assertions are meant to verify that properties hold in general rather than just in the test scenarios.

The validation sequences were taken from an Excel file and encoded in integration tests, using the same techniques as the unit tests.

Using CBMC to verify assertions can, of course, result in counterexamples. Those are given as traces, which can be used to create additional validation sequences by replacing erroneous steps by desired ones. Again, these tests can then be used to improve the implementation and ensure the absence of the counterexample.

The combined approach using both testing for validation and model checking for verification has its merits and provides a high degree of certainty. However, it also has its drawbacks. In particular, the double meaning of assertions can lead to confusion: C-style asserts are used both to encode properties for CBMC and to fail tests. Yet, there is no combined methodology to react on failing assertions and errors uncovered by model checking have to be handled differently from failing tests.

4.2 Code structure

The overall architecture of our implementation is depicted in Fig. 2. We follow a structure that is fairly similar to that the specification provides. Since two subsystems are specified, the code is separated into two folders, one for the cruise control and the other for the light system. This is to help

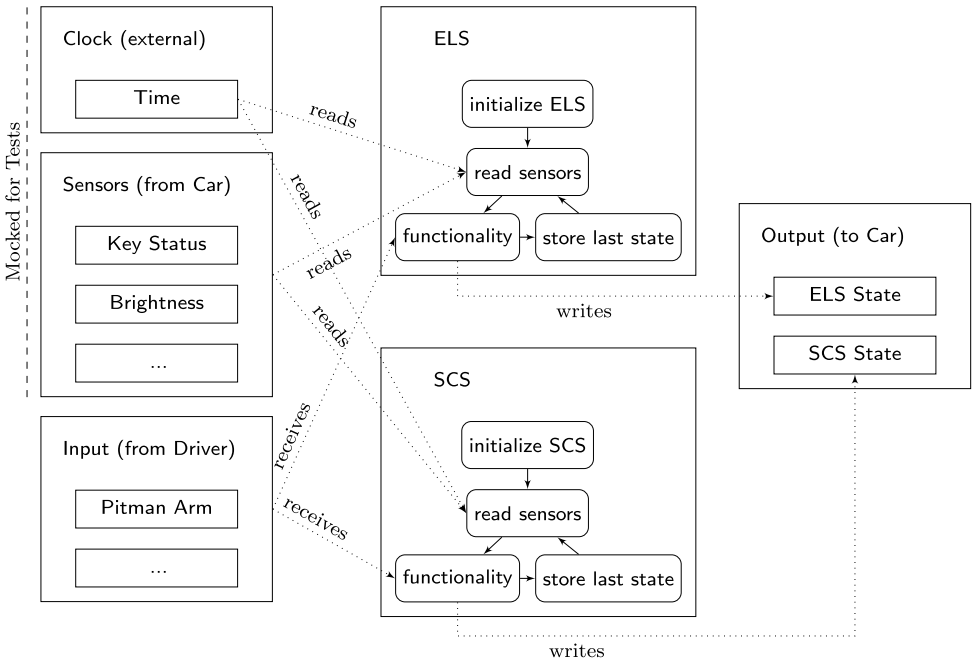

Fig. 2 System Architecture and Internal Communication. ELS and SCS are strictly decoupled; clock, sensors, and driver input are external to the controlling system itself

ensure that the systems are independent of each other. Shared type definitions, e.g., the pedal deflection, the sensor state enumeration, and shared sensors, are stored separately. An artificial time sensor was introduced for testing, but can easily be replaced by an actual clock.

Each of the subsystems is split into three header files and implementations. The first header file declares the accessible and shared sensors for the subsystem, and contains relevant type definitions. Another header file defines the user interface, e.g., how the pitman arm may be moved or what input the pedals for gas and brakes may yield. The last header file contains definitions for the actuators, i.e., what the system is allowed to do. Only the latter two header files are actually implemented, eventually resulting in three C files:

- A state struct that contains all the data relevant to the subsystem.
- The user interface to simulate inputs. This changes some internal variables that keeps track of the state of the UI; in a deployed system, this can be replaced by additional sensors. The attributes correspond to the signals that the subsystem has to communicate.
- The realization of the state machine with several guarded state transitions. This is the actual implementation of the specified safety properties.

For the test cases, sensors are mocked. In order to get an actual executable, real sensors have to be linked during compilation. The time spend for development, validation and verification is given in Table 1.

For the sake of brevity, we will only show small code snippets in this paper. The full implementation is available at https://github.com/wysiib/abz2020-case-study-in-c-public.

Table 1 Development Time

Task	Time (h)
basic implementation and code structure	2
ELS implementation, tests and scenarios	30
SCS implementation, tests and scenarios	22
model checking	3
refactoring and code cleanup	2
state visualization	6
domain-specific visualization	25

4.3 Traceability of requirements

As the other case studies, we do not employ a fully formal approach to traceability. The only form of traceability we provide is by using naming patterns. Our unit tests in general contain the requirement they are concerned with in the name of the test routine. Larger integration tests also reference the validation sequence they represent, which in turn contains the requirements justifying individual steps.

As a consequence, we can only trace which test cases are validating certain requirements and to what extent requirements are covered. An example is given in listing 5, in which we called the test `els3_a_left`. This indicates it is a test for first part (a) of the requirements ELS-3, which focuses on the left-hand side direction blinking. Comments aside, we have no link between a requirement and the individual part of the code realizing it. One could argue that there is a such a link due to the nature of test-driven development and the use of

a version control system (VCS): the VCS would allow us to spot the code written immediately after the test case, i.e., the code that made the test case pass for the first time. Ideally, the code would be in the same commit as the test case or in the one immediately after.

However, this link is weak, as both test cases and code might be changed later on in subsequent commits. Furthermore, a subsequent code change might be related to one or more already existing code changes. Refactorings add an additional layer of complexity that the simple immediate test-to-source link could not follow.

4.4 Variability of requirements

The specification document gives requirements for cars with different features. Examples include the driver position, which can be on the left- or right-hand side of the car; a market code (USA, Canada, or EU), which influences how direction blinking works (European cars have a dedicated tail-lamp which American cars do not); and armored vehicles, which have an additional darkness switch in order to suppress certain light features.

In our current implementation, we handled the requirements' variability by introducing artificial sensors. These sensors produce a constant value for each of the above feature and are assigned to local variables during the system's initialization. While this approach was easy to implement on top of our initial system, it also pollutes both the code and program state to some extent.

In the automotive industry, a commonly used approach for handling variability is software product line development [19, 32]. If the variability had been larger, we could have split ELS and ECS into a common base product used for all market segments and used a software product line approach to develop individual manifestations, e.g., for Canada vs. USA.

Given that the variability in the requirements was not that large, we opted for the simpler implementation in order to concentrate on validation and verification.

4.5 Properties addressed & limitations

Due to time constraints, we opted not to implement every single requirement but tried to cover as much as possible. Aside from the emergency brake light, all requirements have been taken into account for the ELS. For the SCS, we implemented about two-thirds of the requirements, up to (including) SCS-28. While it would be nice to have a more complete implementation, we do not think that it would impact our gathered conclusions.

A feature of the requirements that is not addressed satisfyingly are timers. We are convinced that any modern CPU to be used in cars is fast enough to execute an iteration of the state machine within a reasonable time frame. Thus, any real system realized following our approach should be able to guarantee execution within the smallest time resolution that is relevant to the subsystems and their respective requirements.

Yet, it is hard to give any real-time guarantees. The only evidence that can be given is to run the system often enough and measure whether execution is kept in the specified tolerances. However, this is still better than what we expect of more formal approaches, which usually do not account for wall time at all.

5 Model details

In the following, we will detail implementation idioms we employed to simplify handling and verification of the involved state machine, and explore some snippets of our code to showing these idioms in practice.

5.1 Formalization approach

As stated, we postponed verification as much as possible. Instead, as our first step, we set up the validation sequences as test cases. Then, following test-driven development, we added to the implementation code by only considering the first failing assertion in a scenario. Once the test passed, we moved on to the next. In a second step, we added test cases that are directly related to one or sometimes several requirements.

Finally, we set up CBMC and tried to verify the properties described by the requirements. As stated, we use the same code for testing and formal verification, avoiding any translation between verification and testing environments as done for instance by Chen et al. [16] and others. However, both approaches remain distinct rather than being combined into a single verification procedure [44].

5.2 Modeling idioms

Besides sticking to the MISRA C guidelines and test-driven development we also adopted two further idioms during modeling: only use enumeration types, and do not expose mutability. We will motivate and elaborate on these in the following subsections.

5.2.1 Use enumeration types

We opted to define all types as enumeration types. This is to be expected for some data types, which are true enumerations, such as:

```
typedef enum {Ready, Dirty, NotReady}
        sensorState;
```

Yet, we also defined integer types as enumerations:

```
typedef enum {
    percentage_low = 0,
    percentage_high = 100
} percentage;
```

The reasons for this are twofold: first, we can easily identify thresholds and the value range for each type. While percentages are straightforward, other values such as the steering wheel angle are not easily represented in a human-understandable format. An excerpt of the corresponding type definition is as follows (analogously for turning the steering wheel to the right):

```
typedef enum {
    st_calibrating = 0,
    st_hard_left_max = 1,   /* 1.0 deg */
    st_hard_left_min = 410,
    st_soft_left_max = 411, /* 0.1 deg */
    st_soft_left_min = 510,
    st_neutral_maxl = 511, st_neutral = 512,
    ... /* analogous for the right side */
} steeringAngle;
```

Such a type definition renders it easier to identify, e.g., in what direction the steering wheel is turned and how far. For instance,

```
st_hard_left_max <= angle &&
angle <= st_hard_left_min
```

can be used to check if the wheel has been turned far to the left.

C behavior is undefined if a value that is out of range of the corresponding enumeration is passed. Thus, our second intention was that model checking tools could easily deduce the actual value range rather than having to consider integers exhaustively. This will be discussed further in Sect. 6.2.

5.2.2 Do not expose mutability

It is easy to write broken code when using mutable structs, especially if they are used in order to communicate between threads. Instead, we pass *values* to and from interface functions. This means, that values are copies of the data which are not referenced from anywhere else in the program and the receiver may do however they please with it. An example is that the state from the light subsystem can be queried (for test cases). The returned value will never change unless the test case chooses to do so; no action in the ELS influences it. This also allows reading multiple output variables consistently.

On the other hand, frequently changing *internal* variables, are declared as local (using the `static` keyword). They are always stored in the same "place" and may not be exposed; in particular, there are no getter functions for these variables.

Listing 1 Sensor Reads and CBMC Assumptions

```
keyState ks = get_key_status();
__CPROVER_assume(ks == NoKeyInserted || ks ==
     KeyInserted || ks ==
    KeyInIgnitionOnPosition);
bool engine_on = get_engine_status();
__CPROVER_assume(engine_on == true ||
    engine_on == false);
voltage voltage_battery = get_voltage_battery
    ();
__CPROVER_assume(voltage_battery >=
    voltage_min && voltage_battery <=
    voltage_max);
...
__CPROVER_assume(implies(ks ==
    KeyInIgnitionOnPosition, engine_on ==
    true));
__CPROVER_assume(implies(engine_on == true,
    ks == KeyInIgnitionOnPosition));
```

5.3 Coding examples

Below, we present some key snippets of our implementation. We focus on the concept of the ELS systems, as the SCS is structured the same way.

The core of our ELS is the `light_do_step` function, spanning over almost 300 lines of C code, that is called in a loop. Some auxiliary functions exist to properly set the high beam light, blinkers, etc., where it was appropriate to heed the DRY principle. The `light_do_step` function can be divided in three major parts, described below.

Sensor reads and type information for CBMC First, all relevant sensors are read and stored locally. For verification with CMBC (as discussed in Sect. 6.2), it is necessary to provide type information for integer ranges and enums. Listing 1 shows this for three examples: first, all possible states of the key are enumerated. Second, as C represents booleans as integers, boolean values must be specified to be exactly true or false. Third, integer ranges such as the possible values for the battery voltage have to be provided as an axiom. Additionally, we add assumptions based on the specification, e.g., that the engine state is linked to the key position.

As noted, we implemented time as a sensor as well. Listing 2 shows that we also had to add assumptions that the timestamp only increases.

Implementation of requirements Requirements are encoded by a collection of if-statements. Interestingly, no else-branch exists in the function — most likely because the specification does not contain the words "else" or "otherwise".

Listing 2 Time as a Sensor

```
size_t tt = get_time();
__CPROVER_assume(tt >= when_light_on);
__CPROVER_assume(tt >= blink_timer);
__CPROVER_assume(tt >= ambi_light_timer);
__CPROVER_assume(tt >= pitman_arm_move_time);
```

Listing 3 Implementation of two Requirements

```
// ELS-16 (has priority over ELS-17)
if(!engine_on && (last_lrs != lrs_auto)
    && (get_light_rotary_switch() == lrs_auto
    )) {
    set_all_lights(0);
}
...
// ELS-41: reverse gear
if(reverse_gear) {
    set_reverse_light(100);
}
if(!reverse_gear) {
    set_reverse_light(0);
}
```

Listing 4 Verification of two Requirements

```
// ELS-22: low beam => tail lights
assert(implies(blinking_direction != hazard,
    implies(get_light_state().lowBeamLeft >
    0, get_light_state().tailLampLeft > 0 ||
    get_light_state().tailLampRight > 0)));
...
// ELS-41: reverse gear turns on reverse
    lights
assert(implies(reverse_gear, get_light_state
    ().reverseLight > 0));
assert(implies(!reverse_gear, get_light_state
    ().reverseLight == 0));
```

In the snippet in listing 3, we show how two smaller requirements are realized.

Assertions The last part of the `light_do_step` function contains code for invariant verification. Listing 4 contains assertions that can be checked using CBMC to verify two requirements.

5.4 Modeling of time constraints

When writing code that takes time into account, one is easily tempted to access the current time provided by the operating system. This is a bad when time-based properties are to be tested, as tests would have to be enriched with sleep statements to achieve proper timing for the situation under test.

Instead, we introduced an artificial sensor reporting the current time in milliseconds, comparable to a unix timestamp. For testing, this sensor is mocked, and some artificial time is provided. The code does not know anything about time, it just reads a sensor returning an integer value.

The only assumption made is that one cannot go back in time. In consequence, the step functions can be called in a continuous loop, independent of the computing speed and time needed for a single iteration. On fast hardware, there might even be several executions within the same timestamp (e.g., if the resolution is milliseconds) or timestamps might pass without an execution following (e.g., when using nanoseconds). Mocking the sensor also has the advantage that test scenarios, which would take several minutes of wall time, can be executed in milliseconds instead.

If the entire piece of software was to be shipped, it would be trivial to swap out the sensor: One only has to link an implementation that provides the real time, which may be provided by the operating system.

5.5 Readability and comprehensibility

MISRA C is already designed to improve readability [5], and given that C is a widely used language we assume our implementation to still be accessible for nonexpert developers unfamiliar with formal methods. Of course, the usual coding guidelines for increasing readability apply to this case study as well, e.g., naming conventions, limits on line length or nesting depths, consistent indentation, and the like.

In the following, we will revisit our implementation and discuss the readability of the code and hold it against this assumption.

5.5.1 Readability metrics over ELS and SCS

An intuitive metric for readability seems to be the number of lines of code (LOC). ELS and SCS include 605 and 642 LOC, respectively, not counting 328 blank and 200 comment lines. However, more precise metrics for readability have been suggested.

Buse and Weimer [13] show that the average number of identifiers per line, the average line length, or average nesting depth are negatively correlated with readability. Simultaneously, the average number of comment lines, and the average number of semantically breaking blank lines are positively

correlated with readability. The negative impact of nesting on readability is further pointed out by Johnson et al. [31]. Taking this into account, our code still seems to be readable. We observe a minimum amount of nesting, with only if-constructs introducing mostly only one extra level of indentation, nesting for at most two levels. The average line lengths for the ELS and SCS are 34.37 and 36.27 characters, respectively, with maxima of 170 and 125 characters. These numbers suggest that the majority of lines is short and comprehensible, with some outliers rendering individual parts of the code less readable.

5.5.2 Subjective readability of ELS and SCS

Besides readability metrics, a more subjective way of estimating the code readability is to simply try reading it again. The main interest hereby lies within our step functions, which are continuously looped over. For the ELS and SCS modules, these functions consist of 276 and 127 LOC, respectively. Both start by accessing all the sensors, partly without processing their return values. The rest of the implementation follows a clear pattern: if-statements checking for a condition to act upon. While the code lacks some comments which explain why certain things are done, the references to the respective requirement from the case study accompany the code fragments as annotations. Overall, as the code is not written in a high-level specification language which more closely captures natural language, the overall readability seems to be limited by the general readability of (MISRA) C code. While this can be seen as a drawback, one could also argue that no further understanding of higher mathematics or set theory is required as, for instance, in certain formal languages. Thus, we believe the code remains equally readable to experts and nonexperts alike.

5.5.3 Readability of the unit tests

Following, we want to take a closer look at our tests' readability. Each unit tests follows a pattern of arrange, act, assert, as shown in listing 5. As this is a well-known technique, we assume the tests are comprehensible by nonexperts.

However, we can observe two points negatively impacting readability. Firstly, some tests involve multiple assertions. Especially in terms of time-sensitive behaviors, we observe patterns where the test advances the timer, asserts a specific property (e.g., light on or off) then repeats the process to assert the property change after a certain time.

Secondly, as this was such a common pattern specifically in the ELS, we introduced macros which reduced boilerplate code, but might have reduced readability. The macro `progress_time_partial` in listing 5, line 31, for instance, advances time for a given time frame and asserts that a property retains a given value along the way. While incredibly valuable for writing the tests, we acknowledge that the macro's name is not descriptive enough as it does not convey its role as an assert statement. Hence, the readability of the test itself decreases.

5.5.4 Code pollution due to CMBC annotations

As C is not designed for formal verification, we found that some annotations for CMBC started to pollute the code. While adding asserts into the code to introduce invariants is straightforward and immerses into the C code quite well, we needed to add further axioms to the code so CMBC was able to properly work with our enum types. This resulted in cluttering of the sensor reads as can be seen in listing 1. Here, we ended up with one big block of sensor read and value axiom pairs which impacts readability. However, in most formal languages, this would be a nonissue as they were designed with invariants and axioms in mind and include them as first class citizens appropriately. Furthermore, we added these axioms at the very end of the development process whereas they are much more involved in early development stages in fully formal methods.

6 Validation & verification

We tried to validate our implementation throughout the whole development process by using test-driven development, as we will discuss in Sect. 6.1. In addition, we used the CBMC model checker to fully verify different properties of our implementation directly on the C code as we will describe in Sect. 6.2.

6.1 Test-driven development using cmockery

We used test-driven development based on the provided scenarios. For this, we rely on Google's cmockery library,[4] which provides a unit testing framework and allows mocking functions. Since we did not want to execute all tests in real-time, we mocked functions that extract sensor data and the current time in our test cases. We used two different kinds of test cases for a first quick validation:

- The provided scenarios were automatized and used as integration tests.
- In addition, we implemented unit tests for all requirements given in the specification document. Of course, each unit test only covers a minimal scenario that shows how the requirement is supposed to be understood and automatizes the verification of that single scenario.

[4] https://github.com/google/cmockery.

Listing 5 Test of Requirement ELS-3

```
1  void els3_a_left(void **state) {
2    init_system(leftHand, false, EU, false, false);
3    sensors_and_time sensor_states = {0};
4
5    assert_light_state(((light_state) {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0}));
6
7    // ignition: key inserted + ignition on
8    sensor = update_sensors(sensor, sensorTime, 1000);
9    sensor = update_sensors(sensor, sensorBrightnessSensor, 500);
10   sensor = update_sensors(sensor, sensorKeyState, KeyInIgnitionOnPosition);
11   sensor = update_sensors(sensor, sensorEngineOn, 1);
12
13   mock_and_execute(sensor_states);
14
15   sensor = update_sensors(sensor, sensorTime, 2000);
16   pitman_vertical(pa_Downward5);
17   mock_and_execute(sensor_states);
18
19   assert_partial_state(blinkLeft, 100, blinkRight, 0);
20   pitman_vertical(pa_ud_Neutral);
21   sensor = update_sensors(sensor, sensorTime, 2000);
22   mock_and_execute(sensor);
23
24   pitman_vertical(pa_Upward7);
25
26   progress_time_partial(2000, 2499, blinkLeft, 100, blinkRight, 0);
27   progress_time_partial(2500, 2999, blinkLeft, 0, blinkRight, 0);
28
29   int i;
30   for (i = 3; i < 6; i++) {
31      progress_time_partial(i*1000,       i*1000 + 499,
32                            blinkLeft, 0, blinkRight, 100);
33      progress_time_partial(i*1000 + 500, i*1000 + 999,
34                            blinkLeft, 0, blinkRight, 0);
35   }
36 }
```

A snippet taken from the test case of the requirement ELS-3 is shown in listing 5. The system is initialized to belong to an EU-based car with left-hand drive and without any extras such as ambient light, followed by the initialization and assertions regarding the correctness of the initial state. Afterwards, in lines 8 to 15, we update the sensors to the values they should hold at the start of the test scenario and the code setting up the mocked functions is called. In particular, we set the time sensor used to simulate the actual clock as described in Sect. 5.4. Overall, the test setup phase ensures that our artificial sensors report the required values.

Line 16 shows the difference between sensors and driver interaction: While sensors have to be mocked in order to simulate an actual system, user input is given directly. This corresponds to what will happen in an actual car: the system has to react to user input immediately, while it can read sensor data arbitrarily.

Line 19 asserts that the left blinker is on 100% and the right one is on 0% once the step function was executed after the user input was given. We use the function `assert_partial_state`, since we only make an assertion regarding the two variables `blinkLeft` and `blinkRight`, rather than making an assertion over all state variables.

Finally, Lines 26–27, as well as 31–34, assert that for each millisecond in the time interval, the provided values remain the same, i.e., that the step function does not change output values during that time frame.

As can be seen, we have implemented different C macros to simplify test case development:

- `assert(_partial)_state` which checks if the internal states of ELS and SCS correspond to given assertions. The assertions can specify the state both partially, as done in the listing, and fully.
- `progress_time(_partial)` combines assertions on the state with a progression of time as reported by the time sensor.

Validation results As expected, using test-driven development provided the usual benefits:

- having to formulate test cases helped us gain an understanding of the requirements and how they are supposed to work,
- refactoring was made easier and more secure, and
- the implementation was closer to the actual specification from the start.

The fact that we are working with an actual implementation made test-driven development come naturally. However, different ways of combining formal methods with test-driven development have been discussed [6] as well. In addition, developing specifications using continuous testing has been suggested for former ABZ case studies in the context of the B method [25, 26].

Influences on code Using the macros above, our initial design of splitting sensors, user input, and actuators did not have to be adapted further to be testable. Yet, it created a vast amount of code entirely dedicated to testing. Of 5223 source code lines (including a Makefile and code used for visualization but not counting comments and blank lines), 3786 lines are test code. In general, we needed quite a lot of code for the tests as we had to create the appropriate infrastructure to be able to handle the progression of time in the test cases. Thus, future additional test will probably not increase the number of lines of code needed for testing as much.

6.2 Model checking using CBMC

As stated above, we used CBMC to verify properties of our implementation directly on the C code. Depending on where we place C-style assertions, they correspond to different kinds of properties commonly used in state-based formal methods:

- If placed at the end of the loop implemented by the ELS and the SCS state machines depicted in Fig. 2, assertions correspond to safety invariants that have to hold in every state reachable by one of the subsystems.
- If placed anywhere inside the loop, assertions can be used as invariants on intermediate states.

- If placed outside the loop, we can check if properties hold after a certain number of iterations (controlled by CBMC's unrolling preferences).
- By using additional variables for unrolling state traces, we can implement a lightweight verification of temporal properties. Of course, this is not as powerful as LTL or CTL.

Examplary verification of ELS-22 Requirement ELS-22 is a great example for an invariant. It states "Whenever the low or high beam headlights are activated, the tail lights are activated, too". For this, we can add an assertion such as:

```
implies(get_light_state().lowBeamLeft > 0,
        get_light_state().tailLampLeft > 0 ||
        get_light_state().tailLampRight > 0)
```

The disjunction in the second part of the implication is important for American cars: as tail lamps are used for indicators, it is accepted behavior if one tail lamp is temporarily deactivated during a flashing cycle. When running CBMC, it immediately came up with a counterexample. A part of the output trace can be found in listing 6.

The counterexample shows how the two system variables `ks`, i.e., the key state, and `engine_on`, i.e., the engine's ignition state, change while `light_do_step` is executed.

The main issue with such a counterexample is that each variable assignment, function call, and return from a function introduces a new state. While this representation mimics the internal workings of the C code, it does not correspond to the mental model: comparable to common state-based formal methods, we regarded a state change to include multiple variables at once.

Hence, as we were only interested in comparing state variables per full iteration of `light_do_step`, the output was barely readable to us (the counterexample consists of more than 200 lines and 50 low-level states).

CBMC can optionally reduce the output by removing assignments unrelated to the property. This did not work well for us, as the assignment of signals for the low beam headlights was removed as well. We ended up manually writing state variables in a spreadsheet to comprehend the scenario and ultimately created our own visualization which we will present in Sect. 7.2. A (condensed) version of the trace be found in Table 2 (using "NoKey" for "NoKeyInserted" and "KeyIn" for "KeyInIgnitionOnPosition"). Here, the (high-level) state changes between two full iterations of our step function are shown, rather than changes of individual variables during the execution. This representation aligned better to our mental model of the implementation and was thus more helpful for debugging.

The error in our code was that, based on ELS-17, only the low beam headlights were activated due to activated daytime running light. This was not uncovered by the test scenarios, since daytime light was only tested by night, where, coincidentally, other triggers activated the tail lamps. As this trace did not contain assignments of the tail lamp variables, we had to look up their initial (unchanged) values and add them to the table manually.

Listing 6 Partial CBMC Output

```
State 59 file light/light-impl.c line 242
    function light_do_step thread 0
-------------------------------------------
 ks=/*enum*/NoKeyInserted
    (00000000000000000000000000000000)

State 63 file light/light-impl.c line 242
    function light_do_step thread 0
-------------------------------------------
 ks=/*enum*/KeyInIgnitionOnPosition
    (00000000000000000000000000000010)

State 65 file light/light-impl.c line 244
    function light_do_step thread 0
-------------------------------------------
 engine_on=FALSE (00000000)

State 69 file light/light-impl.c line 244
    function light_do_step thread 0
-------------------------------------------
 engine_on=TRUE (00000001)
```

Table 2 Example Trace Violating ELS-22

State Variable	Iteration 1	Iteration 2
key_state	NoKey	KeyIn
engine_on	FALSE	TRUE
all_doors_closed	FALSE	TRUE
brightness	0	37539
speed	0	936
daytime_light_was_on	FALSE	TRUE
low_beam_left	0	100
low_beam_right	0	100
last_engine	FALSE	TRUE
last_key_state	NoKey	KeyIn
last_all_door_closed	FALSE	TRUE
tail_lamp_left	0	0
tail_lamp_right	0	0

Verification results However, the assertion still failed to verify. Upon further analysis of the property, we discovered

a conflict between ELS-22 and hazard blinking in Canadian and US cars. In those cases, hazard blinking deactivates both tails lights for the dark cycle, thus violating the property. We extended our assertion by checking our variable for blinking direction beforehand:

```
assert(implies(blinking_direction != hazard,
                /* old assertion */));
```

Afterwards, we were able to successfully verify the property using CBMC.

Influences on code At first glance, using CBMC only required to add assertions to the code. As assertions are often introduced as part of understanding certain scenarios, this does not change the modeling strategy itself. Some additional assertions were required to let CBMC detect integer ranges that are defined by an `enum` (as otherwise, counterexamples would find, e.g., percentage values larger than 100, cf. Sect. 5.2), and that consecutive timestamps cannot get smaller.

A huge issue we encountered very late is that, by default, the tool only is able to work with *internal* state changes, i.e., state that the implementation adds. *External* state changes, i.e., state that is modified by user interaction or the environment, were not part of our first verification attempts. This can be imagined as verification of a car whose engine cannot be started.

CBMC can incorporate such external state changes if the corresponding function has `nondet_` as a prefix of its name (which, naturally, collided with our naming conventions). Two modeling issues come with that: first, constant values (such as the market code or installed features) might change on every call now. Thus, we had to refactor so that the values are retrieved once and the getter functions were replaced by using these local variables. Second, we now cannot express certain *orderings* of external states: an example is the light rotary switch that has to move from the *off* position via the *auto* position before it can be set to *on*. We found no way to encode this behavior.

7 Other observations

Our implementation work allowed us to identify several flaws in the specification, as well as shortcomings of our implementation strategy. In the following, we document such issues and give suggestions and solutions.

7.1 Specification ambiguities, flaws, and suggested improvements

During development, we identified several shortcomings or ambiguities within the specification. These issues were found during analysis of the requirements and during implementing test cases. As we only performed validation steps after implementation, the validation steps just uncovered shortcomings of our own implementation and noncompliances with respect to the specification. Due to page limitations, we will only present some of them:

ELS-37 is somewhat broken or at least highlights an incompleteness in the specification. For now, there is no way to discern whether an adaptive cruise control is part of the vehicle; from the specification, we had to assume that it is installed in every system. Then, according to SCS-1, there does not even have to be a desired speed at all times: right after engine startup no previous desired speed is available. We think that, in order to make sense at all, it rather should be "is active" than "is part of the vehicle". Also, this is the only part of the specification that refers to an *advanced* cruise control.

ELS-42 to ELS-46 only partially specify what should happen in case of subvoltage. Instead, information should be given for all light components how the software should react or whether they remain unaffected. Only the requirement ELS-43 mentions a single unaffected component. In particular, turn signals, the regular low beam headlights and the emergency break light remain unspecified. Additionally, ELS-43 references ELS-31 for a situation where the light rotary switch is in position *Auto* and the pitman arm is *pushed*; ELS-31, however, describes behavior if that switch is in the *On* position and the pitman arm is *pulled*.

ELS-19 contains a contradiction: first, it states that ambient lighting *prolongs* already active low beam headlights. Later, it says that the headlamps "remain active or *are activated"*. We think that some actions are reasonable to activate the headlight even if it was not on before (e.g., opening the doors). Others definitely should not activate the headlight (e.g., if the brightness falls below the specified threshold, as passing cars and the setting sun might trigger the brightness sensor). Also, it does not have any constraints regarding the light rotary switch: if the switch is in the "off" position, we think the ambient light should not activate at all.

While `currentSpeed` is specified as a sensor in the ELS, it is not clear how the SCS accesses this value. No sensor is provided according to the specification, and only the brake pressure is mentioned as actuator but not the gas pedal. Thus, the SCS as specified appears to only be responsible for determining the desired speed but not for actually deploying it to the current speed? To our understanding, the measured current speed should be a sensor to the SCS to allow it to work properly.

SCS-23 specifies a safety distance for the adaptive cruise control of $2.5\,\mathrm{s} \cdot$ `currentSpeed` when the current speed is below 20 km/h, and a safety distance of 2 m if both vehicles are standing. Assuming `currentSpeed` < 2.88, however, the

safety distance according to SCS-23 is below 2 m and effectively approaches 0 the closer the vehicle gets to a standstill, e.g., $2.5\,\text{s} \cdot 2.8\,\text{km/h} = 2.5\,\text{s} \cdot 0.77\,\text{m/s} = 1.925\,\text{m} < 2\,\text{m}$. But once a standstill is reached, the safety distance is reset to 2 m and thus violated instantly. It remains unclear whether this 2 m distance is meant as minimum or intended to delay the reaction to eventual acceleration of the vehicle in front.

SCS-28 specifies an acoustic signal which is to be played if the time to reach a standstill with maximum deceleration is greater than the time until impact. SCS-21, however, already specified another acoustic signal to be played when maximum deceleration of 3 m/s^2 is insufficient to prevent impact. As it is not specified what SCS-28 considers as maximum deceleration value these signals might overlap if 3 m/s^2 can be assumed again. While the specification does state the maximum brake-implied deceleration to be 6 m/s^2, it is unclear whether this is the deceleration to be considered for SCS-28.

7.2 Improvements to our employed methodology

We are surprised how easy it was to implement the case study in C, given that none of the authors is a professional C developer. While we were unsure during implementation, given our test harness and the results of CBMC, we now have more confidence in the correctness of our implementation. However, CBMC's output was hard to interpret as we discussed above. Performance not being a primary concern, our result turned out simple enough to not warrant further optimization.

To improve, we created different visualizations. One such visualization is a state visualization based on PlantUML[5] (cf. Fig. 5). A second visualization was a domain-specific visualization in C++ with OpenGL, using the existing sources directly as part of the compilation.

However, development was incomplete and thus omitted for the initial article. Revisiting the visualizations for this extended article, we found that both are not fully satisfactory: The PlantUML-based visualization is very technical, directly referring to implementation details. While it is still beneficial for understanding test failures, it relies on knowledge about the internal workings of the implementation and is thus not presentable to external stakeholders.

In contrast, the C++ visualization is domain-specific, i.e., it shows a car and its actuators, and is thus understandable without knowing implementation details (cf. Fig. 3). However, as the visualization was directly linked to the C sources, it proved rather inflexible and prone to breakage when implementation details changed. While it was still useful as a mere demonstration tool, its value for development was diminished.

Fig. 3 OpenGL based domain-specific visualization, showing the car and its current speed

To improve, we decided to further separate implementation and visualization. Our goal was to keep interacting with the implementation simple, but also allow replacing it with a new version straightforwardly.

To achieve this, we added a small sensor implementation, to be controlled from the outside and linked as a shared library. On top, we used F# to develop a visualization using the RayLib library. This approach worked well, reducing the communication of the two components to a simple and somewhat stable C interface.

While the older OpenGL-based Visualization looked pleasing, it almost completely omitted numerical feedback. This decision struck us as too extreme in retrospect, so we opted for a more minimal look, but including both a domain-specific visualization and values of state variables. For example, the steering angle directly corresponds to the displayed image of a steering wheel, as well as to the traced out path the vehicle is currently headed (cf. Fig. 4). We found the visuals very helpful in gaining a quick understanding of the implementation's behavior. The geometric intuition provided by the 3D view was a welcome addition.

In contrast to the old visualization, we included an interactive component, enabling the user to experiment and explore the behavior. By default, an automatically animated car will perform a lemniscate around an attractor point. This was surprisingly effective in finding behavior that does not conform with expectations: For instance, when using the pedals some reaction is to be expected. Yet, even though the pedal position indeed changed as seen in the visualization,

[5] https://plantuml.com/.

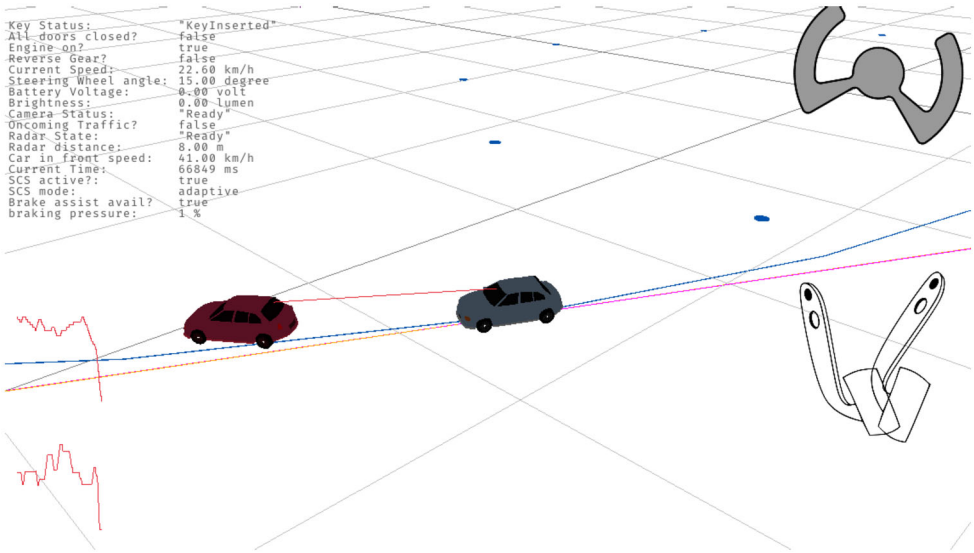

Fig. 4 Newly Developed RayLib Visualization, showing the car and visually representing the status of its actuators

nothing happened with respect to the car's movement. In consequence, we noticed that neither does the gas pedal cause acceleration, nor does the brake pedal decelerate the vehicle. As far as the SCS is concerned, the brake pedal merely disables cruise control as mandated by SCS-16.

Using C for implementation proved very flexible, as there exists a plethora of ways to interact with other languages. Thus, it would have been easy to use other ways of animation. For instance, we were able to execute our implementation in a browser by compiling it to wasm via the Emscripten Compiler Frontend (emcc) and then interact with it using JavaScript.

7.3 Note about deriving a software implementation

As we have started from a low-level implementation in C, the software implementation was always readily available. Hence, in our case, the "model" can be directly compiled and executed. However, testing the executable would still be interesting if we look beyond our simple tests, i.e., with an actual implementation at hand, hardware-in-the-loop tests would be desirable.

8 Comparison

In the 2020 ABZ Proceedings, five other contributions were published, all of them providing verified formal models rather than implementations:

- Arcaini et al. [4], who utilized abstract state machines (ASMs) [10] and the ASMETA framework [3],
- Cunha et al. [20], who modeled their solution in Electrum [37], an extension to Alloy [30],

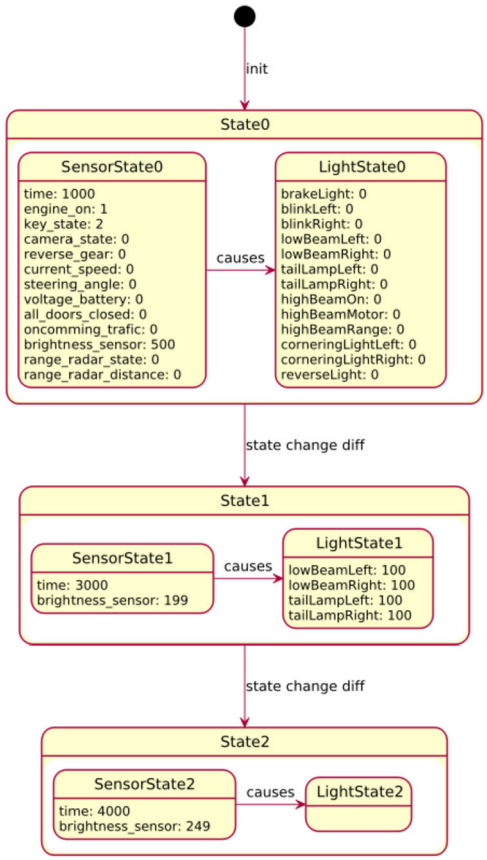

Fig. 5 Visualization using PlantUML

- Leuschel et al. [36], who developed their solution in classical B [1] and later translated to Event-B for proof [2],
- Mammar et al. [38, 39], used Event-B for two distinct models of the ELS [39] and the SCS [38].

The main difference between our approach and the other case studies is that we tried to verify an existing low-level implementation after it has been developed. In comparison, the formal approaches undertaken by the other case studies tend towards following a correct-by-construction approach. That is, they later derive an implementation from the formal model by code generation, e.g., as possible for B [42].

The case study implemented using ASMs by Arcaini et al. [4] uses a formalism of a much higher abstraction level than our concrete implementation. However, code generation is available for ASMs as outlined by the authors. As a result, a C++ (rather than plain C) implementation could be derived from the formal models and might look somewhat like our direct implementation.

Furthermore, ASMETA allows doing conformance testing, i.e., deriving unit tests from the formal model rather than writing them by hand as we did.

The actual implementation aside, Arcaini et al. designed their models in roughly the same fashion as we designed our implementation. ELS and SCS are coupled very loosely and developed as independently as possible. At the same time, they share some signals, comparable to the actuators we defined above. During development, features were added gradually while keeping a (proven) refinement chain intact. While we added features in roughly the same fashion and order, we had no access to a formal proof of refinement. Thus, we had to rely on our test cases entirely.

The approach followed by Arcaini et al. does not verify timing issues, as there is no continuous time in ASM. This is a weakness when compared to our low-level implementation which could be executed in realtime.

The case study performed by Cunha et al. [20] follows an approach to verification and validation that is similar to ours. Initially, the authors use test case given as animation scenarios (i.e., small test cases) and reference scenarios (i.e., the execution sequences given in the specification). This is comparable to the test-driven development we employed as discussed in Sect. 6.1.

Formal verification of the requirements is performed after (some of) the development steps. This is again is comparable to our approach of using CBMC on an already existing implementation (cf. Sect. 6.2).

The case study by Leuschel et al. [36] models time in the same way we did. There is a dedicated model implementing timers based on elapsed milliseconds. In contrast to our simple clock module shown in Fig. 2, Leuschel et al. use a more involved timer, supporting deadlines and even triggers.

The case study also uses a domain specific visualization tailored specifically for the case study. Here, using a well-established formal method shows its merits. For B and Event-B, a multitude of animation frameworks and tools is available and can be used without much overhead. In the case study, a visualization tool called "VisB" is used, which allows modifying SVG graphics based on state variables. While this approach is comparable to our visualization, no custom implementation aside from some glue code connecting the image to the state values was needed.

Visualization and timers aside, the B and Event-B models are much more formal than our implementation and rely heavily on proof (by Rodin) and model checking (by ProB) rather than testing.

The two articles by Mamar et al. concentrate on the ELS [39] and the SCS [38] individually. A particular focus is on the model's differentiation between the two systems and the environment. The model for the environment closely resembles the sensors and inputs modules shown in Fig. 2.

Again, the case study uses ProB for model checking. Given that the authors intentionally tried to avoid rather costly LTL model checking and proof, their properties resemble what we check using CBMC. In particular, simple properties on states sequences are rendered model checkable by storing the prestate in individual variables available for comparison with the current state.

Further related work An alternative to both our immediate low-level implementation and to the code generation approaches that would usually follow with the other case studies is to embed the formal models in runtime code directly, i.e., without compiling them to some other language.

For B, this has been outlined in a demonstration of the ETCS hybrid level 3, where a classical B model is able to control real trains [26]. The approach uses the ProB Java API [33] to connect the formal model to the outside world and allows interacting with it.

9 Conclusions

To summarize, we have implemented a low-level version of the ABZ 2020 case study in MISRA C, a language commonly used in the automotive industry. We relied on test-driven development for validation as well as on formal verification using model checking.

Compared to case studies using more rigorous approaches, our approach shows both advantages and disadvantages. In particular, our implementation stays close to the actual system, can easily be deployed to an actual car, and could be used for simulation and hardware-in-the-loop tests. Furthermore, due to the popularity of C in the automotive industry, it is more approachable by developer untrained in formal methods. However, while CBMC is an excellent tool, we seemed to have hit every pitfall and in many instances actually did not verify anything. Overall, for an untrained developer, it is very easy to show "the wrong thing" and obtain a "proven correct" signal while the code may still be faulty.

In our attempts to address the (eventually) uncovered issues, we certainly missed the expressiveness and mathematical clarity of more rigorous approaches. In particular, the semantics of a guard substitutions would have been very useful, as different actions (e.g., controlling the low beam headlights and the cornering light) can be regarded in isolation instead of considering 300 lines of a single step function. Further, having invariants and other properties as first class citizens rather than inserting them artificially via macros and external functions is clearer and more convenient. Compared to a formal method, we were only able to do very lightweight verification of temporal properties and would certainly have favored to be able to model check LTL or CTL properties. Thus, while we were able to verify our implementation to a certain degree, we suspect that a more thorough approach would be able to provide stronger guarantees.

Furthermore, we currently do not validate any properties on time constraints aside from simulating an external clock in the test cases. As part of possible future work, we intend to use CBMC to try to provide real-time guarantees and to verify the correct behavior in presence of scheduling and limited by the actual specifications of an embedded device.

Both could be verified by providing a Verilog model of the hardware, sensors and connections. Afterwards, co-verification of the implementation with the Verilog circuit model can be performed by CBMC [17]. Additionally, we would like to consider other tools that work directly on the C code, e.g., Symbiotic [15] or Klee [14].

Further future research could be done in the combination of formal and informal approaches, e.g., when thinking about code generators: proven invariants on a high-level model could be compiled to C assertions. Then, they could be verified on the low-level code as well, effectively demonstrating the correctness of the code generators.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (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**(2), 155–166 (2011)
4. 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.) Proceedings ABZ, pp. 302–317. Springer, Cham (2020)
5. Bagnara, R., Bagnara, A., Hill, P.M.: The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software. In: Podelski, A. (ed.) Proceedings Static Analysis, pp. 5–23. Springer, Cham (2018)
6. Baumeister, H.: Combining Formal Specifications with Test Driven Development. Proceedings XP/Agile Universe, LNCS, vol. 3134. Springer, Berlin (2004)
7. Beck, K.: Test-Driven Development: By Example. Kent Beck Signature Book, Addison-Wesley (2003)
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings TACAS, LNCS, vol. 1579, pp. 193–207. Springer, Berlin (1999)
9. Boogerd, C., Moonen, L.: Assessing the value of coding standards: an empirical study. In: Proceedings ICSM, pp. 277–286. IEEE, New York (2008)
10. Börger, E., Gargantini, A., et al.: Proceedings ASM, vol. 2589. Springer, Berlin (2003)
11. Bowen, J.P., Hinchey, M.G.: Seven more myths of formal methods. IEEE Softw. **12**(4), 34–41 (1995)
12. Brookes, T.M., Fitzgerald, J.S., Larsen, P.G.: Formal and informal specifications of a secure system component: final results in a comparative study. In: Gaudel, M., Woodcock, J. (eds.) FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18–22, 1996, Proceedings, vol. 1051, pp. 214–227. Springer, Berlin (1996)
13. Buse, R.P., Weimer, W.R.: Learning a metric for code readability. IEEE Trans. Softw. Eng. **36**(4), 546–558 (2010)
14. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings OSDI, vol. 8, pp. 209–224. USENIX Association (2008)
15. Chalupa, M., Vitovská, M., Strejček, J.: Symbiotic 5: boosted instrumentation. In: Proceedings TACAS, LNCS, vol. 10806, pp. 442–446. Springer, Berlin (2018)
16. Chen, M., Ravn, A.P., Wang, S., Yang, M., Zhan, N.: A two-way path between formal and informal design of embedded systems. In: Proceedings UTP, LNCS, vol. 10134, pp. 65–92. Springer, Berlin (2017)
17. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: Proceedings DAC, pp. 368–371. IEEE, New York (2003)
18. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings TACAS, LNCS, vol. 2988, pp. 168–176. Springer, Berlin (2004)
19. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Boston (2002)
20. Cunha, A., Macedo, N., Liu, C.: Validating multiple variants of an automotive light system with electrum. In: Raschke, A., Méry, D., Houdek, F. (eds.) Proceedings ABZ, pp. 318–334. Springer, Cham (2020)
21. Fathy, H.K., Filipi, Z.S., Hagena, J., Stein, J.L.: Review of hardware-in-the-loop simulation and its prospects in the automotive area. In: Modeling and Simulation for Military Applications, vol. 6228. SPIE, Bellingham (2006)
22. Fitzgerald, J.S., Brookes, T.M., Green, M.A., Larsen, P.G.: Formal and informal specifications of a secure system component: first results in a comparative study. In: Naftalin, M., Denvir, B.T., Bertran,

M. (eds.) FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24–18, 1994. Proceedings, Lecture Notes in Computer Science, vol. 873, pp. 35–44. Springer, Berlin (1994)
23. General Specification of Basic Software Modules. AUTOSAR, Munich (2019)
24. Hall, A.: Seven myths of formal methods. IEEE Softw. **7**(5), 11–19 (1990)
25. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. In: ABZ 2014: The Landing Gear Case Study, CCIS, vol. 433, pp. 1–17. Springer, Berlin (2015)
26. Hansen, D., Leuschel, M., Schneider, D., Krings, S., Körner, P., Naulin, T., Nayeri, N., Skowron, F.: Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with real trains. In: Proceedings ABZ, LNCS, vol. 10817, pp. 292–306. Springer, Berlin (2018)
27. Hatton, L.: Language subsetting in an industrial context: a comparison of MISRA C 1998 and MISRA C 2004. Inf. Softw. Technol. **49**(5), 475–482 (1998)
28. Houdek, F., Raschke, A.: Adaptive Exterior Light and Speed Control System
29. ISO: Road Vehicles – Functional Safety (2011)
30. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)
31. Johnson, J., Lubo, S., Yedla, N., Aponte, J., Sharif, B.: An empirical study assessing source code readability in comprehension. In: Proceedings IEEE ICSME, pp. 513–523 (2019)
32. Käköla, T., Duenas, J.C.: Software Product Lines. Springer, Berlin (2006)
33. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. Form. Methods Syst. Des., 1–28 (2020)
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.) Proceedings ABZ, pp. 382–397. Springer, Cham (2020)
35. Larsen, P.G., Fitzgerald, J.S., Brookes, T.M.: Applying formal specification in industry. IEEE Softw. **13**(3), 48–56 (1996)
36. 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.) Proceedings ABZ, pp. 335–350. Springer, Cham (2020)
37. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Proceedings ACM SIGSOFT, FSE 2016, pp. 373–383. Association for Computing Machinery, New York (2016)
38. Mammar, A., Frappier, M.: Modeling of a speed control system using event-B. In: Raschke, A., Méry, D., Houdek, F. (eds.) Proceedings ABZ, pp. 367–381. Springer, Cham (2020)
39. 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.) Proceedings ABZ, pp. 351–366. Springer, Cham (2020)
40. MISRA C:2012 – Guidelines for the use of the C language in critical systems. MISRA (2013)
41. Short, M., Pont, M.J.: Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation. J. Syst. Softw. **81**(7), 1163–1183 (2008)
42. Vu, F., Hansen, D., Körner, P., Leuschel, M.: A multi-target code generator for high-level B. In: Proceedings IFM, pp. 456–473. Springer, Berlin (2019)
43. Yang, M., Zhan, N.: Combining Formal and Informal Methods in the Design of Spacecrafts. LNCS, vol. 9506, pp. 290–323. Springer, Berlin (2016)
44. Yuan, J., Shen, J., Abraham, J., Aziz, A.: On combining formal and informal verification. In: Proceedings CAV, LNCS, vol. 1254, pp. 376–387. Springer, Berlin (1997)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.