A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System

Sebastian Krings, Philipp Körner,
Jannik Dunkelau, Kristin Rutenkolk

This is a pre-print version archived by P. Körner at
https://pkoerner.github.io/pages-output/publications/.

The final authenticated version is available online at
https://doi.org/10.1007/978-3-030-48077-6_30.

A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System

Sebastian Krings[1], Philipp Körner[2], Jannik Dunkelau[2], and Chris Rutenkolk[2]

[1] Institute for Information Security
Niederrhein University of Applied Sciences
Mönchengladbach, Germany
sebastian.krings@hs-niederrhein.de
[2] Institut für Informatik, Heinrich-Heine-Universität
Universitätsstr. 1, D-40225 Düsseldorf, Germany
{p.koerner,jannik.dunkelau,chris.rutenkolk}@hhu.de

Abstract. In this article, we present an approach to the ABZ 2020 case study, that differs from the ones usually presented at ABZ: Rather than using a (correct-by-construction) approach following a formal method, we use MISRA 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. In consequence, our realization of the ABZ case study can serve as a baseline reference for comparison, allowing to assess the benefit provided by the various formal modeling languages, methods and tools.

1 Introduction

The ABZ 2020 Case Study [18] describes two assistants commonly found in modern cars. The overall system consists of two loosely coupled components, namely an adaptive exterior light system (ELS) and a speed control system (SCS). The ELS controls head- and taillights, setting their brightness depending on the surroundings and user preference. At the same time, the SCS controls the vehicle's speed, again by taking into account the environment as well as parameters given by the driver. Obviously, both are safety critical components, rendering safety and security a development priority.

Used Methods and Tools In this article, we present our implementation of the ABZ 2020 Case Study. Our approach differs from the ones usually followed by the ABZ community: we do not employ a fully formal development method. Instead, we attempted an approach closer to what might happen in industries, 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 performed formal verification attempts directly on

the C code, using the CBMC model checker [11]. Both MISRA C and CBMC will be introduced more thoroughly in Section 2.1 and Section 4.2 respectively. Test-driven development and mocking of test objects will be presented in Section 2.2.

Rationale Often, formal methods practitioners claim to hold a high ground over "traditional" software development or at least that there rarely are disadvantages [14,7]. The argument seems convincing; yet, we are not aware of any (case) study comparing two teams working on the same project, one employing a formal approach and the other working "traditionally". For this case study, we aim at providing a baseline that can be compared to fully formal approaches or other approaches combining formal and informal verification, e.g., as suggested for spacecrafts [21]. We opted to postpone verification as much as possible, as one would expect a group focusing on embedded systems to work. This allows 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 There are several features rendering 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 the actual hardware. Code that interacts with sensors or user input is separated, i.e., it could immediately be linked to actual hardware. Additionally, our implementation makes use of real threads, just as the sub-components of the system would run in parallel. We expect that most specifications using formal methods simply allow some non-determinism concerning the ordering of state transitions. This has some more consequences: Our implementation allows real-time simulation of the system, whereas state transitions using formal methods usually happen instantly and do not amount for any time elapsed during calculations. This also allows usage of our implementation for hardware-in-the-loop tests, which are common for automotive (cf. [13,20]) in order to test the entire system.

Thirdly, MISRA C is a language that stems from the automotive industry. It is a somewhat formal language, in the sense that certain rules are required to be followed. Yet, it is also relatively flexible, since other rules are only advisory.

2 Modeling Strategy & Implementation

In the following section, we will discuss how we approached the initial implementation in C, starting with details on the C dialect we use in Section 2.1. Afterwards, the general structure of our implementation is presented in Section 2.3,

followed by a discussion of the limitations of our approach and implementation in Section 2.4. For the sake of brevity, we will only show small code snippets in this paper. The full model is available at

https://github.com/wysiib/abz2020-case-study-in-c-public.

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 [1] defines a subset of C meant to be used for safety critical systems, in particular in the automotive sector. In fact, both ISO 26262 [19] and the software specification by AUTOSAR [2] reference or 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 meant as advisory only. For instance, there is a required rule stating that any switch statement should have a default label and mandatory rule stating that any path through a non-void function should end in a return statement.

While of course all coding rules could be checked by hand in theory, we used cppcheck[3] to verify compliance of our code to most of the MISRA rules. However, given that not all rules can be statically checked the result is only an indication and some manual review is required as well.

Despite its prevalence in the automotive industry, MISRA C has been criticized regarding both efficiency and ease of use. In particular, the possibilities of false positives [17] and of introducing new errors by (unreflectingly) changing code to adhere to the rules [6] should be carefully considered. Both factors again allow for comparison to the formal development methods present at ABZ. 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 of fixing an issue, an appropriate test case is formulated and execute [4]. Naturally, the test fails, as no code implementing the scenario has been added yet.

Only afterwards, the code is extended and improved to make the test pass. As a result, a high confidence can be achieved. Furthermore, the test suite developed helps during refactoring later on.

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

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

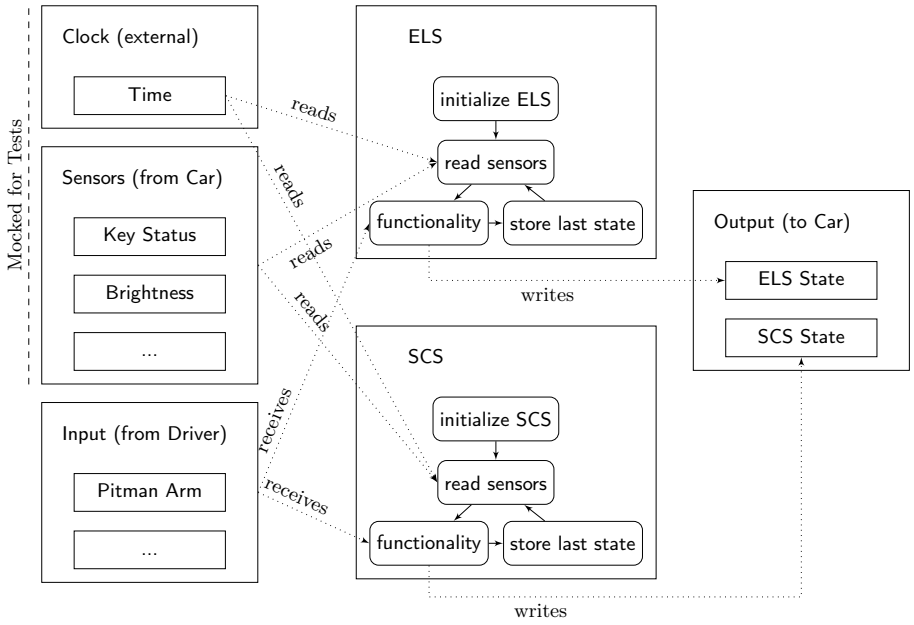

Fig. 1. System Architecture and Internal Communication

and output behavior of program parts [4]. However, rather than implementing the full functionality, a mock is usually much simpler than the code it replaces. For instance, mocks are often supposed to behave deterministically or even to provide constant outputs. For testing purposes, mocks often record the inputs to them and provide them to assertions.

2.3 Code Structure

The overall architecture of our implementation is depicted in Figure 1. We follow a structure that is fairly similar to the one 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 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:

Table 1. Development Time

Task	Time in Hours
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

- A state struct that contains all the data relevant to the subsystem.
- The user interface such that user input can be simulated. 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.

2.4 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 withing 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.

2.5 Formalization Approach

As mentioned earlier, we postponed actual verification work as much as possible. Instead, as our first step, we set up the validation sequences as unit tests first.

Then, in a test-driven development manner, we added to the implementation code by only considering the next 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 formal verification and testing environment as done for instance by Chen et al. [10] and others. However, both approached remain distinct rather than being combined into a single verification procedure [22].

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 in C with the Verilog circuit model can be performed by CBMC [12]. Additionally, we would like to consider other tools that work directly on the C code, e.g., Symbiotic [9] or Klee [8].

3 Model Details

In the following, we will detail our implementation idioms we employed to ensure easier handling and verification of the involved state machine, and explore some crucial snippets of our code to show these idioms in practice. Contrary to the proposed outline, we will present key snippets as well in Section 4.

3.1 Idioms

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, e.g.:

```
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 to everyone, e.g., the translation of the steering wheel angle into human-understandable semantics is hard. 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, ...
} steeringAngle;
```

Such a type definition renders it easier to identify, e.g., in what direction the steering wheel is turned and how far, i.e., To check if it is turned far to the left, `st_hard_left_max <= angle && angle <= st_hard_left_min` can be used.

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 and do not consider, e.g., the full range of 32-bit integers in their stead. This will be discussed further in Section 4.3.

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 sub-system 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, *internal* variables that may change frequently, which are not meant to be read by anyone else, 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.

3.2 Timers

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 idea when such time properties shall be tested: then, tests would have to be enriched with additional sleep statements in order to achieve proper timing for the situation under test.

Instead, we introduced an artificial sensor that may be accessed by both sub-systems. The sensor reports the current time in milliseconds, comparable to a common unix timestamp. During test cases, this sensor is mocked and some artificial time is provided. The code does not know anything about time, but just reads a sensor returning an integer value.

The implementation only assumes that one cannot go back in time, no further assumptions regarding the progression of time are made. In consequence, the step functions can simply 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, that 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 the provided by the operating system.

4 Validation & Verification

We tried to validate our implementation throughout the whole development process by using test-driven development, as we will discuss in Section 4.1. 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 Section 4.2.

4.1 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 as well as 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.

A snippet taken from the test case of the requirement ELS-3 is show in Listing 1. The system is initialized to belong to an EU-based car with left-hand drive and without any extras such as ambient light. Initialization and assertions regarding the correctness of the initial state are not shown in the snippet. Afterwards, in lines 2 to 9, 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 that is used to simulate the actual clock as described in Section 3.2. Overall, the test setup phase ensures that our artificial sensors inside the mock report the required values if and when the system reads them.

Line 10 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 and at any time, while it can read sensor data arbitrarily.

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

Listing 1. Test of Requirement ELS-3

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

Line 13 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 `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 20–21 as well as 25–28 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 unit and integration testing as parts of a test-driven development workflow helped us during the initial development. 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 [3] as well. In addition, developing specifications using continuous testing has been suggested for former ABZ case studies in the context of the B method [15,16].

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 (which also contain a Makefile and code for state (graph) visualization), 3786 lines are test code. Comments and blank lines are already excluded.

4.2 Model Checking Using CBMC

As stated above we used CBMC [11] to verify properties of our implementation directly on the MISRA C code. CBMC is a model checker for programs written in C. It uses bounded model checking [5] 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 presence of undefined behavior due to C quirks.

Additionally, it can be used to verify user-given assertions stated as C-style assertions using the macros in `assert.h`. Depending on where they are placed in the code, 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 Figure 1, 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, we can communicate between states and implement a lightweight verification of temporal properties. Of course, this is not as powerful as LTL or CTL, as we have to rely on unrolling.

Listing 2. 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)
```

4.3 Example: 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:

```
assert(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 temporary deactivated during a flashing cycle. When running CBMC, it immediately came up with a counterexample. A snippet can be found in Listing 2.

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 our main step function `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).

CBMC can optionally reduce the output by removing assignments that are 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

Table 2. Example Trace Violating ELS-22.

State Variable	Iteration 1	Iteration 2
key_state	NoKeyInserted	KeyInIgnitionOnPosition
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	NoKeyInserted	KeyInIgnitionOnPosition
last_all_door_closed	FALSE	TRUE

writing state variables in a spreadsheet to comprehend the scenario. A (condensed) version can be found in Table 2. Here, the 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.

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. Yet, CBMC comes with a flaw: it is not able to detect integer ranges given by enumerations. This means it frequently finds errors with values for enumerations, that are out of scope. As a consequence, one has to add assumptions about value ranges to the code, which cannot be compiled to actual code. Another assumption that needs to be added is that consecutive timestamps cannot get smaller. Thus, for useful verification, some form of conditional compilation is required.

5 Specification Ambiguities and Flaws

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 for test-driven development. As we only performed validation steps after implementation, the validation steps just uncovered shortcomings of our own implementation and non-compliances w.r.t. 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. 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 does not specify what should happen in case of sub-voltage. The only given information is that the adaptive high beam headlight is not available. Should manual high beam headlight be triggered instead? Should the high beam remain dark? This remains absolutely unclear.

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. This requirement needs some serious polishing.

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, let alone for the possibility to ensure whether more acceleration is required to maintain it or not.

SCS-23 specifies a safety distance of $2.5\text{s} \cdot$ `currentSpeed` for the adaptive cruise control when the current speed is below 20 km/h. It further specifies an absolute distance of 2m if both vehicles are standing. Assuming `currentSpeed` $\in]0, 2.88[$ however, the safety distance according to SCS-23 is below 2m and effectively approaches 0 the closer the vehicle gets to a standstill. But once a standstill is reached, the safety distance is set to 2m and thus is violated instantly. It remains unclear whether these 2m distance is meant as minimum or is intended to delay the reaction to eventual acceleration of the vehicle in front.

SCS-28 references a maximum deceleration value, which was only described for the adaptive cruise control in SCS-20 and SCS-21. We assume that it references the same maximum deceleration of 5 m/s^2. It further specifies the acoustic

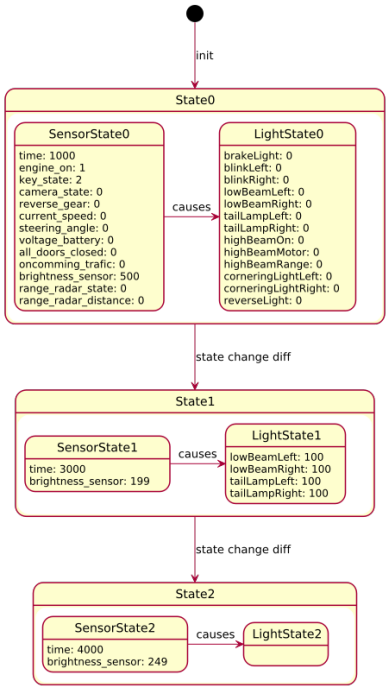

Fig. 2. Ad-hoc Visualization

signal which is to be played if the time to reach a standstill with maximum deceleration (5 m/s^2) is greater than the time until impact. This acoustic signal however may overlap with the signal specification given in SCS-21.

6 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 both on common programming techniques such as test-driven development and formal verification using model checking. As we have not followed a fully formal development method, our implementation can serve as a baseline for comparison with the more formal approaches usually presented at ABZ.

We suspect that more rigorous approaches to software development will show both advantages and disadvantages to our approach. In particular, our approach stays close to the actual system and can easily be deployed to an actual car. Furthermore, our code can be used for simulation and hardware-in-the-loop tests.

However, we certainly missed the expressiveness and mathematical clarity that comes with more rigorous approaches. Compared to a formal method, we could only 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.

In particular, our approach has only very limited support for verifying temporal properties (i.e., just by unrolling properties to a certain degree). Furthermore, we currently do not validate any properties on time constraints aside from simulating an external clock in the test cases.

That aside, all state properties given in the specification could in theory be verified using our approach even though we have not fully implemented all of them. Furthermore, given that we can place assertions everywhere in our C source code, we could reason about intermediate states as well.

Method and Tool Review We are surprised how easy it was to implement the case study in C, especially as 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.

CBMC was a great tool that found counterexamples, e.g., to the requirement ELS-22. Yet, we have to make the following observations: first, the output was barely readable, i.e., 52 state transitions represent two high level states after the initialization. As a result, we wrote our own state graph visualization tool based on plantuml[5] (cf. Figure 2). Second, for the initial error, a simple assertion would already have tripped the test case.

The majority of our time, we spent implementing test cases for the individual requirements. Being aware of typical formal method workflows, we think that this must be done in every case study. Otherwise, without using animation to verify that the behavior is correct, one cannot have sufficient confidence in the model. This, combined with the tooling that is available for C code, makes us excited to see other case studies, and challenge them to name benefits of their individual approaches, as we now know the extent of access to (semi)-formal development the embedded software community has.

Nonetheless, we think these tools allow for interesting research for 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. It remains open how hard the translation process is and whether the power of these tools is sufficient.

References

1. *MISRA C:2012 – Guidelines for the use of the C language in critical systems.* MISRA, 2013.
2. *General Specification of Basic Software Modules.* AUTOSAR, Munich, 2019.
3. H. Baumeister. Combining Formal Specifications with Test Driven Development. In *Proceedings XP/Agile Universe*, volume 3134 of *LNCS*. Springer, 2004.
4. K. Beck. *Test-driven Development: By Example.* Kent Beck signature book. Addison-Wesley, 2003.

[5] https://plantuml.com/

5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
6. C. Boogerd and L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *Proceedings ICSM*, pages 277–286. IEEE, 2008.
7. J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE software*, 12(4):34–41, 1995.
8. C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings OSDI*, volume 8, pages 209–224. USENIX Association, 2008.
9. M. Chalupa, M. Vitovská, and J. Strejček. Symbiotic 5: Boosted Instrumentation. In *Proceedings TACAS*, volume 10806 of *LNCS*, pages 442–446. Springer, 2018.
10. M. Chen, A. P. Ravn, S. Wang, M. Yang, and N. Zhan. A Two-Way Path Between Formal and Informal Design of Embedded Systems. In *Proceedings UTP*, volume 10134 of *LNCS*, pages 65–92. Springer, 2017.
11. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
12. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings DAC*, pages 368–371. IEEE, 2003.
13. H. K. Fathy, Z. S. Filipi, J. Hagena, and J. L. Stein. Review of hardware-in-the-loop simulation and its prospects in the automotive area. In *Modeling and simulation for military applications*, volume 6228. SPIE, 2006.
14. A. Hall. Seven myths of formal methods. *IEEE software*, 7(5):11–19, 1990.
15. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the ABZ Landing Gear System using ProB. In *ABZ 2014: The Landing Gear Case Study*, volume 433 of *CCIS*, pages 1–17. Springer, 2015.
16. D. Hansen, M. Leuschel, D. Schneider, S. Krings, P. Körner, T. Naulin, N. Nayeri, and F. Skowron. Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In *Proceedings ABZ 2018*, volume 10817 of *LNCS*, pages 292–306. Springer, 2018.
17. L. Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49(5):475–482, 2007.
18. F. Houdek and A. Raschke. Adaptive Exterior Light and Speed Control System.
19. ISO. Road vehicles – Functional safety, 2011.
20. M. Short and M. J. Pont. Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation. *Journal of Systems and Software*, 81(7):1163–1183, 2008.
21. M. Yang and N. Zhan. *Combining Formal and Informal Methods in the Design of Spacecrafts*, volume 9506 of *LNCS*, pages 290–323. Springer, 2016.
22. J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In *Proceedings CAV*, volume 1254 of *LNCS*, pages 376–387. Springer, 1997.