Post-Hoc Formal Verification of Automotive Software with Informal Requirements: An Experience Report

Gustav Ung
Scania
Södertälje, Sweden
gustav.ung@scania.com

Jesper Amilon
KTH Royal Institute of Technology
Stockholm, Sweden
jamilon@kth.se

Dilian Gurov
KTH Royal Institute of Technology
Stockholm, Sweden
dilian@kth.se

Christian Lidström
KTH Royal Institute of Technology
Stockholm, Sweden
clid@kth.se

Mattias Nyberg
Scania
Södertälje, Sweden
mattias.nyberg@scania.com

Karl Palmskog
KTH Royal Institute of Technology
Stockholm, Sweden
palmskog@kth.se

2024 IEEE 32nd International Requirements Engineering Conference (RE) | 979-8-3503-9511-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/RE59067.2024.00035

***Abstract*—In this paper, we report on our experience with formally specifying and verifying an industrial software module, provided to us by a company from the heavy-vehicle industry. We start with a set of 32 informally stated requirements, also provided by the company. We discuss at length the formalization process of informally stated requirements for the purposes of their subsequent formal verification. Depending on the nature of each requirement, one of three languages was used: ACSL contracts, LTL or MITL. We use the Frama-C deductive verification framework to verify the source code of the module against the formalized requirements, with the outcome that 21 requirements are successfully verified while 6 are not. The remaining 5 requirements could not be verified for the module itself, as they specify behavior outside it. We illustrate what steps we took to convert LTL and MITL formulas into ACSL contracts to enable their verification in Frama-C. Finally, we discuss conclusions we drew from our work, notably that formal-verification-driven development of modules and verified breakdown of system requirements could likely remedy some problems we encountered.**

***Index Terms*—Industrial requirements, requirements formalization, formal verification**

I. Introduction

In the automotive industry, software plays an increasingly important role, serving a wide range of functions such as engine control and braking systems, but also more advanced features such as autonomous and highly automated driving. Automotive software is often of a *safety-critical* nature, where a high level of confidence in its correct behavior is demanded by the stakeholders. This is regulated by standards such as the automotive *functional safety* standard ISO 26262 [25].

Testing is currently the prevalent method for ensuring software correctness in the automotive industry. However, it is both time-consuming and inexhaustive. *Formal verification* is an exhaustive method that can reduce substantially the need for testing. Unlike testing, however, it requires the existence of *formal specifications* against which to verify the code.

There is evidence that formal verification of software is most effectively done in a process where code and formal specifications such as contracts are developed together, incrementally [8], [31], [34], [37], [38]. But until such workflows are adopted by industry, most of the effort to formally verify automotive software is expended *post-hoc*, i.e., after the code base has been developed. And typically, formal specifications are not provided to the team that performs the verification, which then has the freedom to formulate them from scratch.

In this paper, we report on our experience from a scenario in which both the code *and* the requirements were provided by the company (in this case Scania, a heavy vehicle manufacturer), but the requirements were only stated *informally*. This scenario gives rise to certain additional challenges, such as understanding, disambiguating, and finally formalizing the requirements in a way that they can be formally verified.

Our experience is based on a concrete embedded, safety-critical software module written in C, controlling the secondary power steering unit of heavy vehicles, together with a requirements document. The document describes 32 requirements stated in plain English, with some additional pseudo-code, without any attached formal semantics.

We chose *deductive verification* as the technique for formal software verification [1], [27], since we deemed it to be the most scalable and mature technique that operates at the source code level (i.e., does not require the construction of abstract models of the software), and the Frama-C verification framework as a mature, state-of-the-art tool that supports this technique for C programs [27].

The main *novelty* of our work is presented by the specifics of the type of software we consider here, namely control software of an embedded system, implemented as a scheduler

This work has been partially funded by the FFI Programme of the Swedish Governmental Agency for Innovation Systems (VINNOVA) as the AVerT2 project 2021-02519.

2332-6441/24/$31.00 ©2024 IEEE
DOI 10.1109/RE59067.2024.00035

that periodically invokes a list of terminating C functions, each computing actuator outputs from sensor inputs. These specifics of the software allowed us to use deductive verification to verify not only requirements that specify the state transformation resulting from a single invocation of a function, but also requirements of a (real-time) temporal nature related to the periodic invocation of the function.

The main contribution of this paper is the experience report itself, describing and discussing how we use Frama-C to verify the given piece of industrial code. Since the requirements were given in natural language, far from the formal representation required by Frama-C, we proceeded using a three step approach, and the reported experience is therefore also divided into these three steps:

1) We *classify requirements* into categories that allow their *formalization* in a suitable specification language, which we propose for each category.
2) Many of the requirements are not at the module level but at the vehicle level. This requires their *projection* down to the module level. We show how this can be achieved by means of *assume-guarantee style reasoning* [11].
3) Some of the requirements are of a temporal nature, and others even of a real-time nature. We show how such requirements can be *translated* to transformational ones, and be *verified* by means of deductive verification.

Because of their methodological nature, we believe that our experiences can be useful for embedded-software developers from the automotive industry, when confronted with a similar scenario. Our case study is *exhaustive*, in the sense that we considered *all* requirements in the provided document, regardless of how problematic they were to formalize or verify.

The paper is structured as follows. We begin by reviewing the related work in Section II. Section III presents some background on the logics we used to formalize the requirements. Section IV then describes our case study, the STEERING module. In Section V we present our categorization of requirements, while Section VI and Section VII describe our approach to their formalization and verification, respectively. In Section VIII, we discuss our experiences and the obstacles we met, and we conclude with Section IX.

II. RELATED WORK

The case study software module used in the present paper was also used in previous work [20], in which the focus was to investigate the viability of deductive verification in an industrial setting. The paper considered only a small subset of the requirements as the basis for evaluation, corresponding to a subset of the requirements categorized as *transformational* in the present paper. In particular, requirements that needed to take into account other software than the module under verification were excluded. The paper modeled the requirements as a combinational logic circuit, and found verification to be feasible, but asserted, among other things, the need for a formal language and tool support for writing requirements.

In another industrial case study [14], post-hoc verification is applied to safety-critical avionics software. Also there, existing natural language requirements were formalized using ACSL [6] and verified with Frama-C [12]. The paper suggests that a pattern-based approach could possibly mitigate the difficulties arising in such a setting.

Categorization of specifications has been studied since the 1990s, when Dwyer et al. introduced *property specification patterns* [15]. More recently, Grunske et al. have published work in this direction. In one paper [4], they present a *unified pattern catalog* for qualitative, real-time, and probabilistic properties, and provide a tool for helping engineers disambiguate high-level natural language requirements, and mapping them to LTL, CTL, MTL, TCTL, CSL, and PLTL formulas. In another work [36], they present a specification pattern catalog for qualitative and real-time properties expressible in UPPAAL [9], and present an automatic generator of concrete formulas from user-specified pattern-based property descriptions. The catalog is evaluated on three real-time systems, where they found that it can express the properties of interest and automatically generate corresponding observer automata and formulas, such that they could be verified in UPPAAL. Another work [26] presents a user study evaluating the acceptance of formal methods at Bosch. The study found it to be difficult for engineers to understand formal notation, and to identify, understand, and avoid inconsistencies. However, a majority answered that formal methods could make their systems safer.

Furia et al. propose an automated technique for reducing MTL specifications to specifications over discrete-time models, such as LTL [17]. The work considers only a subset of MTL and is necessarily incomplete. The evaluation on simple examples shows that the problems of incompleteness and computational effort for checking the resulting formulas can, in practice, often be dealt with.

III. PRELIMINARIES

In this section, we describe the basics of deductive verification, and the specification languages ACSL, LTL, and MITL.

A. Deductive Verification and ACSL Contracts

Deductive verification is a common method for verifying computer programs, where one takes as input a program and a formal specification of the program, and verifies by means of logical reasoning, e.g., by utilizing SAT/SMT solvers, that the program satisfies the specification [1], [27]. We illustrate deductive verification here using the ANSI-C Specification Language [6] (ACSL), which is a specification language for C programs. ACSL is the language used by Frama-C [27], a state-of-the-art tool for deductive verification of C programs. For the verification, we use the WP plugin of Frama-C [7], which is based on Hoare logic [21], [30] and Weakest Precondition calculus [13], [22]. For more details on Frama-C and ACSL, we refer to [6], [12], [27].

We shall specifically focus on the ACSL *function contracts*. A first example of an ACSL contract is shown in Fig. 1a, specifying a division function in C. The contract consists of

```
/*@
 requires y != 0;

 ensures
   \result == \old(x/y);
*/
int div(int x, int y);
```

(a) Basic contract

```
/*@
 behavior:
  assumes y != 0;
  ensures
    \result == \old(x/y);
*/
int div(int x, int y);
```

(b) Behavior contract

Fig. 1: ACSL contract examples

```
/*@ ghost int gh_x; */
/*@
  requires \true;
  ensures \result == \old(gh_x) + 1;
*/
int incr() {
  static int x ;
  x++;
  /*@ ghost gh_x = x; */
  return x;
}
```

Fig. 2: ACSL ghost code example

a pre-condition (signified by the **requires** clause) and a post-condition (the **ensures** clause). Intuitively, the pre-condition states what should hold when the function is called, and the post-condition states what should hold upon function return. The pre-condition may also be omitted, in which case the post-condition should hold, regardless of the context from which the function is called (this is equivalent to setting the pre-condition to `\true`). A function *satisfies* a contract if, whenever the function is called and the pre-condition holds, the post-condition holds upon function return. In Fig. 1a, the pre-condition is used to avoid a division-by-zero error, and the post-condition states the desired function output. **\result** represents the return value of the function. Also, **\old** is used in the post-condition to refer to the value of the expression `x/y` *before* the execution of the function.

To verify that a function satisfies its contract, we use WP to verify two properties: that the function `div` is never called from a state where the pre-condition does not hold, and that the post-condition holds after execution of `div` terminates. Contracts can also contain behaviors, as shown in Fig. 1b. Behaviors are similar to contracts, but uses **assumes** instead of **requires**. When verifying a behavior, WP verifies only the second property. Thus, it is not verified that the **assumes** clause holds at each function invocation. This allows for combining several behaviors for the same function, and one may also combine behaviors with **requires** clauses.

Another important aspect of our work is the usage of ACSL *ghost variables* and *ghost code* [16]. Ghost variables are non-program variables that are added to the program for verification purposes, and can only be read from or written to by ghost code. Ghost code can be added anywhere in a program; it may read from any variable, but can only write to ghost variables. One use-case for ghost code is when writing contracts for functions with *local static* variables. In short, local static variables are local variables whose values are stored between function invocations. An example is provided in Fig. 2, where the function `incr` simply increments the local static variable `x`. When specifying this function with a contract, it appears natural to refer to the value of `x`; however, as it is a local variable, it is not in the name scope of the ACSL contract. As a workaround, one can add a *global* ghost variable, `gh_x`, and a ghost code statement at the end of the function, which assigns to the ghost variable the value of the local variable.

B. Linear-time Temporal Logic

Linear-time Temporal Logic (LTL) is a logic that allows specifying properties of systems that evolve in discrete time, i.e., of systems whose executions can be represented as countable sequences of states [33]. The syntax for the subset of LTL used in this paper is the following:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \Box\,\phi \;\mid \phi\,\mathcal{W}\,\phi \mid \ominus\,\phi$$

where ϕ ranges over LTL formulas, and p over a given set of atomic propositions. We use LTL to specify executions of a control software inside an ECU, where states correspond to valuations of program variables at a point in time.

Temporal operators in LTL formulas can refer to *future* states, i.e., forward from the current time, or they can refer to *past* states, backwards in time [18]. We consider two future-time temporal operators, $\Box$ (“always”) and $\mathcal{W}$ (“weak until”), and one past-time temporal operator, $\ominus$ (“previous“). We proceed with an informal description of their semantics; for further details, we refer to [22]. Given a state s in a sequence representing a system execution, we say that $\Box\,\phi$ holds in s if ϕ holds in *every* future state, $\phi_1\,\mathcal{W}\,\phi_2$ holds in s if ϕ_1 holds always, or until ϕ_2 holds, and $\ominus\,\phi$ holds in s if ϕ held in the previous state. For example, the formula $\Box\,(x \geq 0)$ specifies that the value of the variable x should always be non-negative.

C. Metric Interval Temporal Logic

Metric Interval Temporal Logic (MITL) [2] is a logic that allows specifying properties of systems that evolve in continuous time, as long as system executions can be described as countable sequences of successive moments where program states (valuations of variables) are known. The syntax for the subset of MITL we use in this paper is the following:

$$\Psi ::= p \mid \neg\Psi \mid \Psi \vee \Psi \mid \Psi \wedge \Psi \mid \Psi \rightarrow \Psi \mid \Box_I \Psi \mid \boxminus_I \Psi \mid \diamondminus_I \Psi$$

where Ψ ranges over MITL formulas and I over real-time intervals. Intervals are typically *bounded* and then consist of a pair of start and end points $[a, b]$ such that $a \leq b$, where $a, b \in \mathbb{N}$. Intervals may also be *unbounded*, e.g., $[0, \infty)$ contains every non-negative real number.

As in LTL, MITL formulas may refer to future time or to past time. Given an interval I, we say that $\Box_I\,\Psi$ holds now if Ψ continuously holds throughout the entire future time interval I, $\boxminus_I\,\Psi$ holds now if Ψ has held throughout the entire past time interval I, and $\diamondminus_I\,\Psi$ holds now if Ψ has held at some point in the past time interval I. Note that there is no MITL notion of a “previous” operator, due to the denseness of the time domain.

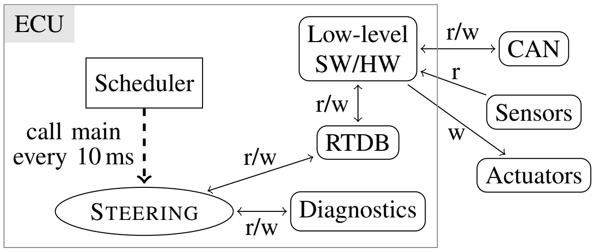

Fig. 3: System architecture as relating to Steering

IV. The Steering Module

The Steering module is a piece of software aimed at controlling the secondary power steering unit of a heavy vehicle, which should take over control of the power steering in case the primary power steering unit malfunctions. The Steering module is thus safety-critical, since loss of secondary power steering could mean complete loss of power steering capabilities.

The source code of Steering is proprietary and cannot be shared. Instead, we describe all relevant aspects. Steering is written in C [23], and is developed according to internal coding guidelines largely identical to those of MISRA-C [29]. It is executed on one of several Electronic Control Units (ECUs) in the embedded system. The source code file consists of roughly 1400 lines of code, not counting lines imported through the header (e.g., type definitions, macros), and contains 10 functions. One of these acts as a top-level, or *main* function, to be repeatedly called by the scheduling software, once every 10 ms. The other 9 functions are internal, and serve to manage the complexity of the module by dividing its functionality. The control flow is simple, containing no recursion or loops. The module (in isolation) is strictly sequential, although as part of the larger system it is not.

Other than the scheduler, Steering communicates directly with two other infrastructure modules in the ECU: a diagnostics module, and a real-time database (RTDB) module which facilitates communication between the different application level modules, and through CAN [24] also other ECUs. This architecture can be seen in Fig. 3, where the letters “r” and “w” denote reads and writes, respectively, and SW/HW means software/hardware. The implementation is generally structured as follows: when the main function is called, all inputs from the system are read by calling functions in RTDB, then several computations are made using these inputs and static variables within the module, before finally writing all outputs to RTDB. Reads and writes to the diagnostics module occur throughout computing results, and may be part of, or the result of, those computations. For brevity, and when it is clear from context, we simply use Steering to refer to the main function of Steering.

Steering is accompanied by a document specifying 32 safety requirements on the module. The requirements are given in a combination of natural language and pseudo-code, examples of which are provided throughout Section V. Furthermore, the document specifies the *interface* (*inputs* and *outputs*) of Steering, e.g., which CAN signals or sensors that the module can read from or write to. These are all communicated through RTDB. It also defines so-called *requirement variables*. While the interface inputs and outputs also exist as *program variables* in the source code, the requirement variables do not. It is, however, possible to map each requirement variable either directly to a specific program variable, or to some relation between several program variables. In some cases, requirement variables are used to store values between executions of Steering, thus acting as both input and output.

The source code and requirements presently analyzed are taken from a development repository.

V. Categorization of Requirements

In this section, we categorize the 32 requirements on Steering into three categories: *transformational*, *discrete-time temporal*, and *real-time temporal* requirements. More concretely, 18 requirements are classified as transformational, 3 as discrete-time temporal, and 6 as real-time temporal. The remaining 5 requirements describe runtime configuration and error handling outside Steering, which is out of scope of this paper. Each category is described in further detail below, and is illustrated with examples. We present each requirement exactly as it was written in the requirements document. The requirements are reviewed holistically, i.e. both the natural language description and the pseudo-code are considered to be part of the requirement, where the latter might refine the requirement with specific parameter values. Some time related variables are specified outside of the requirements, and will be stated separately.

A. Transformational Requirements

We consider requirements transformational when they specify how individual, terminating executions of the main function of Steering change the state of the system.

Req. 1 is an example of a transformational requirement. The requirement states that, if the module has concluded that the secondary steering should be activated, and the parking brake is not activated, then the electric motor, which powers the secondary steering, should be activated. Observe that the requirement is stated first as natural language, and then restated in pseudo-code.

Requirement 1

When the secondary steering circuit handles steering the electric motor providing flow to the circuit shall be activated whenever steering might be needed. In this function, steering is defined as needed when the parking brake is not set.

```
While Secondary Circuit Handles Steering == True
  If ParkingBrakeSwitch == ParkingBrakeNotSet
    ElectricMotor = On
```

This requirement is transformational in the sense that it defines a relationship between one interface input (ParkingBrakeSwitch), one requirement input variable (SecondaryCircuitHandlesSteering), and one interface output (ElectricMotor) of the Steering main function.

Another transformational requirement is Req. 2. The requirement is, on its own, quite straightforward, but its formalization (in Section VI) manifests some noteworthy aspects. The requirement states that, if the dual steering function is not enabled (not equal to zero), then the two CAN-signals shall be emitted as not available, and the electrical motor shall be turned off.

Requirement 2

```
If UP_DUAL_STEERING_E == 0
  FailureInSteeringCircuit1 = NotAvailable
  FailureInSteeringCircuit2 = NotAvailable
  ElectricMotor = Off
```

B. Discrete-time Temporal Requirements

Discrete-time temporal requirements are considered those that specify some behavior over time, but do not mention any specific real-time duration or threshold, and thus can be interpreted over discrete time steps. A common example are requirements on how to treat signals that go into an error state. A concrete example for such a signal is Req. 3.

Requirement 3

If VehicleSpeed has the status Error or NotAvailable and the vehicle variable ”Vehicle is moving” was false before the status went to Error or NotAvailable the variable ”Vehicle is moving” shall be false.

Else If VehicleSpeed has the status Error or NotAvailable and the vehicle variable ”Vehicle is moving” was ”True” before the status went to Error or NotAvailable the variable ”Vehicle Is Moving” shall be ”True” until ParkingBrakeSwitch has the value ”ParkingBrakeSet”. If the parking ”ParkingBrakeSwitch” also goes to Error or NotAvailable the variable ”Vehicle is moving” shall be ”True” until the COO falls asleep.

The requirement describes the expected behavior in the case when the VehicleSpeed signal starts reporting error (or not available). If the vehicle was not moving before this occurred, it should continue to be considered as not moving. If the vehicle was moving, it should continue to be considered as moving until the parking brake is applied, or, if also the parking brake signal goes to an error state, until the main ECU (COO) shuts down.

We deem this to be a temporal requirement, since it apparently specifies the behavior of STEERING over sequences of time steps; in particular, the usage of the word “until” suggests such an interpretation. And since there is no explicit reference to time in the requirement, we consider this as a *discrete* temporal requirement, and not a *real-time* one.

C. Real-time Temporal Requirements

Real-time temporal requirements are considered temporal requirements that are explicit about some time duration or threshold. A general example is to define a threshold time for how long a vehicle can (continuously) be in a given state before some action must be taken.

A specific example is Req. 4. The requirement specifies a threshold (in elapsed real time) for when a flow sensor should be treated as malfunctioning. In particular, if the sensor keeps reporting that there is flow for a longer time than the given threshold, even though the state of the rest of the vehicle system indicates that the sensor should not be reporting flow, then the sensor should be treated as malfunctioning.

As will be seen in Section VI, formalizing this requirement is not entirely straightforward due to the presence of two separate real-time thresholds: one threshold (5 s) for when the electric motor shall be considered turned off, and one threshold (1 s, i.e., time to detect a sensor malfunction) for when the sensor shall be considered malfunctioning.

Requirement 4

If the electric motor (providing flow to the secondary steering unit) is not running there should not be a flow in the secondary power steering circuit. If the sensor is indicating flow a DTC should be set and a yellow warning indicated in the instrument cluster (the latter is handled in AER417_12).

```
When ElectricMotor == Off for 5 seconds
 If FlowSwitch == Flow for Time To Detect A
   Secondary Steering Sensor Malfunction
     SensorFailureSteering2 = True
 Else if FlowSwitch == NoFlow for Time To Detect A
   Secondary Steering Sensor Malfunction
     SensorFailureSteering2 = False
```

The next requirement needs some notion of an instantaneous transition, while expressing that a signal continuously holds for a specific time interval. In particular, the requirement states that when the electric motor is turned on, then it may not turn off for a certain time. As will be seen in Section VIII, formalizing this requirement is also not straightforward.

Requirement 5

```
When ElectricMotor goes from Off to On
  Hold ElectricMotor state until
  minimum connected time
```

VI. FORMALIZATION

This section presents our approach to formalizing the requirements in each of the categories defined in Section V. For each category a specification language was chosen, namely ACSL contracts for transformational requirements, LTL for discrete-time temporal requirements, and MITL for real-time temporal requirements. In summary, the formalization process consisted of the following steps (but since the process was iterative, not necessarily once, and in this order):

1) Agreeing on the correct interpretation.
2) Projecting the requirement onto the STEERING module.
3) Choosing the appropriate classification category.
4) Formalizing the requirement in the specification language picked for the chosen category.

The first step is necessitated by the ambiguity of natural language. In some cases, ambiguities were resolved by studying the source code to be verified. This strategy can lead to interpretations biased towards the source code, which is not a major problem in this study since it is assumed that the source code captures the engineers’ intention. The second step is required as some of the requirements specified behavior outside of the control of the STEERING module. As this work aimed at verifying only the STEERING module, we *projected* such requirements onto STEERING, and formalized only those parts that were the responsibility of STEERING. The remaining parts were taken as *assumptions* on the operating environment of STEERING. Indeed, most of the requirements were stated at the system-level (typically an ECU combined with actuators, sensors, and other hardware components), and thus required projection onto STEERING. The third and fourth steps should be self-explanatory. During the formalization process, one may discover that the requirements are internally inconsistent, meaning that, e.g., two requirements are contradictory. This would necessitate that the requirements are reworked, since internally inconsistent requirements are not realizable in a system. When formalizing the requirements of STEERING, we did not encounter any internal inconsistency.

A. Formalization of Transformational Requirements

We formalize transformational requirements by means of ACSL contracts, since it is the annotation language of the verification tool used in our work (see Section VII), and thus our formalized contracts can be used directly in verification without further processing. The formalization of Req. 1 is relatively straightforward.

Formalization of Requirement 1

```
ensures SecondaryCircuitHandlesSteering == \true
  && \old(parkingBrakeSwitch) == ParkingBrakeNotSet
  ==>  ElectricMotor == On;
```

Next is the formalization of Req. 2. The formalization of this requirement is of interest as it requires a high degree of projection onto STEERING. One may think that the translation to ACSL is straightforward. However, due to the principle of separated concerns, there exists a separate module for variant control, as well as modules for controlling CAN and actuating the electric motor. Therefore, the formalization of the aspects relevant for STEERING simply asserts that the module does not initialize or modify the state of the concerned signals.

Formalization of Requirement 2

```
ensures !\old(FailureSteering1Created)
     && !\old(UP_DUAL_STEERING)
  ==> !FailureSteering1Created;
ensures !\old(FailureSteering2Created)
     && !\old(UP_DUAL_STEERING)
  ==> !FailureSteering2Created;
ensures !\old(ElectricMotor1Created)
     && !\old(UP_DUAL_STEERING)
  ==> !ElectricMotor1Created;
```

B. Formalization of Discrete-time Temporal Requirements

To formalize the discrete-time temporal requirements, we chose LTL. The underlying assumption for all formalizations is that executions can be treated as discrete time-step traces.

To illustrate this, recall Req. 3, which states what should happen when VehicleSpeed goes from a non-error value to error. Note that the requirement consists of two cases specifying what should happen if, before the switch, VehicleIsMoving was true or false, respectively. For better presentation of the formula, we first define a *switch* pattern as syntactic sugar for expressing states in which a formula becomes true:

$$\mathbf{Sw}(\phi) := \phi \land (\ominus\neg\phi)$$

Our formalization of Req. 3 is then the following formula.

Formalization of Requirement 3

$$\begin{aligned}
&\Box\ (((\mathbf{Sw}(\mathsf{VehicleSpeedErrorOrNA}) \land \ominus\neg\mathsf{VehicleIsMoving}) \rightarrow\\
&\qquad(\neg\mathsf{VehicleIsMoving}\,\mathcal{W}\,\neg\mathsf{VehicleSpeedErrorOrNA})) \land\\
&\quad((\mathbf{Sw}(\mathsf{VehicleSpeedErrorOrNA}) \land \ominus\mathsf{VehicleIsMoving}) \rightarrow\\
&\qquad(\mathsf{VehicleIsMoving}\,\mathcal{W}\\
&\qquad\quad((\mathsf{ParkingBrakeSet} \land \neg\mathsf{ParkingBrakeErrorOrNA}) \lor\\
&\qquad\quad(\mathsf{ParkingBrakeErrorOrNA} \land \mathsf{COOIsShutDown}) \lor\\
&\qquad\quad\neg\mathsf{VehicleSpeedErrorOrNA}))))
\end{aligned}$$

Note that the variables VehicleSpeedErrorOrNA and ParkingBrakeErrorOrNA capture when the respective corresponding signal either goes into an error state, or becomes not available. Since the verification is agnostic to these cases being treated individually, we merge them, for brevity, already in the formalization.

Observe also that the formalization uses twice the weak until operator $\mathcal{W}$. In the second part, this corresponds straightforwardly to the usage of the word "until" in Req. 3. For the first part, there is no explicit mention of the "until"-part, but we have judged it to be an implicit assumption that the requirement only applies as long as the signal for the vehicle speed is error or not available.

C. Formalization of Real-time Temporal Requirements

The real-time requirements can be formalized using a temporal logic based on a semantics suitable for real-time signals. While many such logics exist, MITL, as described in Section III, was chosen for this study because it is relatively simple but still expressive enough for the considered requirements. Using MITL, we formalize Req. 4 as follows:

Formalization of Requirement 4

$$\begin{aligned}
&\Box_{[0,\infty)}\ (\boxminus_{[1,6]}\,\mathsf{ElectricMotorOff} \rightarrow\\
&\quad((\boxminus_{[0,1]}\,\mathsf{FlowSensorFlow} \rightarrow \mathsf{SteeSensorFail}) \land\\
&\quad(\boxminus_{[0,1]}\,\neg\mathsf{FlowSensorFlow} \rightarrow \neg\mathsf{SteeSensorFail})))
\end{aligned}$$

For this particular formalization, the time unit is 1 s. However, note that we can choose different time units for different formalizations. The logic formula asserts that, if the electric motor has been continuously off for 5 s, then, if the flow switch reports either flow or no flow continuously for one second, the sensor should be marked with failure or no failure, respectively. Note that the "timer" for the electric motor should reach 5 s before the corresponding flow-timers start.

Note that this is only one of several reasonable interpretations of the requirement. For example, it is reasonable to choose an interpretation where the flow switch should hold its state for 1 s whenever the motor has been turned off for 4 s, leading to a slightly different formalization.

Next follows the formalization of Req. 5. The formalization is harder to grasp than Form. 4, and the problem is that MITL lacks a natural way of expressing the notion of a next state.

Formalization of Requirement 5

$$\Box_{[0,\infty)} (\mathsf{ElectricMotor} \wedge (\diamondminus_{[0,1]} \neg\mathsf{ElectricMotor}) \\ \rightarrow \Box_{[0,mct-1]} \mathsf{ElectricMotor})$$

In this formalization, ElectricMotor denotes that the electric motor is on, while ¬ElectricMotor denotes that it is off. Furthermore, *mct* denotes the minimum time that the electric motor should be on. Additionally, it is assumed that *mct* is an integer greater than 1. Hence, it is suitable to choose a time unit in terms of milliseconds for this formalization.

The intuition behind the formalization is the following. First, notice that $\Box_{[0,mct-1]}$ ElectricMotor entails ElectricMotor. Second, since ElectricMotor and ¬ElectricMotor exclude each other, $\diamondminus_{[0,1]}$ ¬ElectricMotor will stay true for exactly 1 time unit after ElectricMotor becomes true. Therefore, as soon as ElectricMotor goes from false to true, it will stay true for 1 time unit, and then for $mct - 1$ more time units, i.e., for a total of mct time units.

VII. Verification

For verification of formalized requirements, we relied on the WP plugin of Frama-C [7], with results presented in Section VII-D. For the transformational requirements, verification was straightforward, since their formalizations were already expressed using ACSL. For the two temporal requirement categories, we develop ACSL contracts that, if satisfied by individual executions of Steering, entail that the corresponding temporal specification formula holds for executions of the whole system—under certain assumptions on the scheduler and other parts of the environment. The results of this development process, which was carried out in an ad-hoc fashion, are presented below for each of the formalized temporal requirements in Section VI.

One obstacle to verification was that program variables are in the scope of an ACSL contract only if they exist in the source code, as either global variables or function parameters. This was a problem since, as mentioned in Section IV, some of the requirement variables did not have a counterpart program variable, or its counterpart was a (static) local variable, and thus not in scope for the ACSL contract. Our solution to this was to add ghost variables for such requirement variables, as well as ghost code to ensure that the ghost variable represents the intended value when verifying the contract.

A. Verification of Transformational Requirements

To verify the transformational requirements, we annotated the main function of Steering function with corresponding ACSL contracts. Due to Frama-C modularity, we also had to annotate helper functions (called by the main function) with contracts. It is noteworthy that SecondaryCircuitHandlesSteering is a local variable in the source code, wherefore we modeled it with a ghost variable, and added ghost code to the source code, mirroring all assignments to the local variable with equivalent assigns to gh_SecondaryCircuitHandlesSteering.

B. Verification of Discrete-time Temporal Requirements

To verify the LTL formalization of Req. 3 with Frama-C, it must first be translated into an ACSL contract. We explain our translation process and argue briefly why the resulting contract captures the transformational aspects of the LTL formula. We then present the assumptions that are needed for the contract to entail the LTL specification at the system level.

To obtain an ACSL contract from the LTL formula, we first simplify the latter into an *invariant formula* of the form $\Box\,\phi$, with the additional property that ϕ does not contain any occurrences of temporal operators, except for $\ominus$. Concretely, we replaced the occurrences of $\mathcal{M}$ and $\mathcal{W}$, using the $\ominus$ operator. This resulted in the following formula, in which the only temporal operators used are (the outer) $\Box$ and $\ominus$:

$$\begin{aligned}
\Box\ ((&\ominus(\mathsf{VehicleSpeedErrorOrNA} \wedge \neg\mathsf{VehicleIsMoving}) \\
&\quad \rightarrow \neg\mathsf{VehicleIsMoving}) \wedge \\
&((\ominus(\mathsf{VehicleSpeedErrorOrNA} \wedge \mathsf{VehicleIsMoving} \wedge \\
&\quad \neg\mathsf{ParkingBrakeSet} \wedge \neg\mathsf{ParkingBrakeErrorOrNA}) \\
&\quad \rightarrow \mathsf{VehicleIsMoving}) \wedge \\
&(\ominus(\mathsf{VehicleSpeedErrorOrNA} \wedge \mathsf{VehicleIsMoving} \wedge \\
&\quad \mathsf{ParkingBrakeErrorOrNA} \wedge \neg\mathsf{COOIsShutDown}) \\
&\quad \rightarrow \mathsf{VehicleIsMoving})))
\end{aligned}$$

In general, we do not obtain a logically equivalent LTL formula using this translation. However, for the purpose of sound verification, it suffices to establish that the translated formula entails the original one. We do not prove entailment formally here[1], but give the intuition behind the argument. The first part of the translated formula states that if the vehicle was previously considered to be moving, then it should continue to be considered as such (given that the vehicle speed has error status). The outer $\Box$ operator captures that this should hold in every state, entailing the first part of the original formula. For the second part, the argument is similar, but we split the disjunction in the right-hand side of the $\mathcal{W}$ expression into two distinct cases, and add, for each case, the negation of the right-hand side to the left-hand side of the implication.

Next, from the translated LTL formula, we obtain an ACSL contract as follows. First, we remove the $\Box$ operator and change all occurrences of $\ominus$ to `\old`. Then, we let the resulting formula be the post-condition, with the pre-condition being empty. Changing $\ominus$ into \old can be justified rather easily, as both relate a previous state with the current one. Removing the outer $\Box$ operator is justified by further assumptions on the operating environment (see assumption 1 in the list of assumptions presented below).

However, the resulting contract was not possible to verify for the Steering implementation. In particular, the second part of the post-condition could not be verified, since the implementation seemingly makes additional assumptions on

[1] Our pen-and-paper proof uses the fixed-point characterization of LTL and the contrapositive law of propositional logic.

the relationship between the variables. To enable verification, we therefore encoded those additional assumptions using ACSL *behaviors*. The final ACSL contract is the following:

```
ensures part1:
  \old(VehicleSpeedErrorOrNA && !gh_VehicleIsMoving)
    ==> gh_VehicleIsMoving == \false;
behavior part2a:
 assumes COOISShutDown ==> parkingBrakeErrorOrNA &&
         parkingBrakeErrorOrNA ==> !parkingBrakeSet;
 ensures:
   \old(VehicleSpeedErrorOrNA && gh_VehicleIsMoving
      && !parkingBrakeSet && !parkingBrakeErrorOrNA)
     ==> gh_VehicleIsMoving == \true;
behavior part2b:
 assumes parkingBrakeErrorOrNA ==> !parkingBrakeSet;
 ensures:
   \old(VehicleSpeedErrorOrNA && gh_VehicleIsMoving
      && parkingBrakeErrorOrNA && COOISShutDown)
     ==> gh_VehicleIsMoving == \true;
```

Observe that since VehicleIsMoving is a local variable in Steering, we converted it into a ghost variable called `gh_VehicleIsMoving` and added corresponding ghost code, similarly to when we verified transformational requirements.

Besides the correctness of the transformations described above, the verification also relies on several important *assumptions* on both Steering and its operating environment. In particular, we make the following three assumptions:

1) The scheduler operates in an infinite loop, and calls Steering in each iteration of the loop.
2) The value of VehicleIsMoving is not altered by other modules between calls to Steering.
3) A single execution of Steering can be considered *atomic* (e.g., interface inputs do not change values between start of execution and reading them).

The first two assumptions are dependent on the environment in which Steering operates, i.e., on the scheduler and other modules called by the scheduler, as well as the behavior of, e.g., CAN signals. The third assumption depends on the relation between the requirement and the implementation. These three assumptions are all necessary for bridging the semantic gap between ACSL contracts and LTL formulas[2].

After obtaining the contract and encoding assumptions, verification proceeded as for the transformational requirements.

C. Verification of Real-time Temporal Requirements

We exemplify how MITL formulas were translated to ACSL contracts using Form. 4. As with the LTL formula, we first present the translation, and then state the assumptions needed to relate the ACSL contract with the MITL formula.

The contract obtained from Form. 4 represents the transformational aspects of the requirement. We do not show the entire ACSL contract for Req. 4, but briefly explain its structure. For each occurrence of $\diamondminus$, there must exist a corresponding timer variable in the source code. If, for example, the motor is turned on, then its timer is set to 0, otherwise the timer is incremented by 10 (ms). Whenever the timer has reached its target, then the inner formula must be implied. This pattern is nested within the inner formula for the flow sensor timers. The following partial contract illustrates the structure:

```
ensures \old(ELECTRIC_MOTOR_ACT) ==>
    ti_electricMotorOff == 0;
ensures !\old(ELECTRIC_MOTOR_ACT)
    ==> (ti_electricMotorOff
    == \old (ti_electricMotorOff) + 10
    || ti_electricMotorOff == STEE_TI_ELMOTOR_OFF
    || ti_electricMotorOff == 0
);
    // ...
```

We verify the ACSL contract under the following assumptions on Steering and its operating environment:

1) The scheduler operates in an infinite loop, iterating with a frequency of 10 ms, and calling Steering once in each iteration.
2) If the electric motor is set to off during execution of Steering, then it remains so until the next execution.
3) If the actual flow is low in two subsequent executions of Steering, it is continuously low in-between those executions.
4) As in 3), but for the flow being high.

One may note here that the conceptual gap between MITL and ACSL can be seen as wider than the gap between LTL and ACSL, as the former requires moving from a continuous-time domain to a discrete one. The first assumption is largely the same as in the LTL case, but with the added assumption that Steering is called every 10 ms. This addition is important because of the latter assumptions. The next three assumptions are needed on the environment to ensure that signals will not oscillate between consecutive executions of Steering. The timing aspect in the first assumption is therefore needed to ensure that the last two assumptions are satisfiable.

D. Verification Results

The 27 requirements applicable to Steering could be formalized with the methods used in Section VI. All verification was carried out using Frama-C and its WP plugin. The transformational requirements were straightforward to verify, since they were formalized using ACSL contracts.

During the verification stage, 21 out of 27 extracted ACSL contracts were successfully verified. The 6 requirements that could not be verified were all real-time temporal. However, these requirements could be verified by slightly modifying the code.

The verification time was 475 s of CPU time with an Intel Core i7-9859H. There were around 1.5k lines of annotations, including contracts for infrastructure software. Both Alt-ergo and Z3 were used as backend solvers for WP.

Each requirement was treated individually, but similarly to the translation process described in Req. 3 and Req. 4 in general. As also illustrated above, the verification process typically relied on some further assumptions on the environment that Steering operates in. These assumptions typically include assumptions on the diagnosis system, RTDB, and signal status validation code. Some requirements could not be verified

[2] A formal connection could be made, as is done, e.g., in [3], using the TLA framework [28]

directly despite having a formalization. The reason is that we have additional code that is specified in another requirements document. Since the other requirements document was out of scope of this study, we had to add additional assumptions on some ACSL contracts. After the assumptions were added, the contracts could be verified.

VIII. Discussion

Below we discuss the main obstacles we met in this work.

A. Ambiguity of Requirements

A major obstacle has been the ambiguity of the original requirements. Considerable time was spent trying to understand and interpret the intention behind each requirement. Furthermore, many requirements were given in two versions; one version in natural language, and one version in pseudo code. In most cases, it was not obvious whether those two versions were equivalent, or even consistent with each other.

In order to resolve the ambiguity, and in accordance with the post-hoc approach, we studied the source code, and assumed that source code and requirements match. In this way, most of the ambiguity could be resolved. There were, however, two situations that presented an obstacle for the disambiguation; the first, when requirements did not match the code, as discussed below in Section VIII-D, and the second, when, due to the complexity of the code, it was not possible to understand the implemented behavior.

In the context of this work, but also more generally, as reported in [31], we suggest that there are three underlying sources of ambiguity in requirements. Firstly, the engineers writing requirements lack proper training and knowledge in how to write unambiguous requirements. Secondly, due to the ambiguous nature of natural language, writing unambiguous requirements using natural language is intrinsically impossible. Thirdly, the industrial context does not provide sufficient incentive to write unambiguous requirements; rather, there is only an incentive to write sufficiently good requirements for their intended use, which is mainly test case generation, documentation, and compliance with process standards. In these use cases, ambiguous requirements might be acceptable, since they are combined with other informal explanations and clarifications given, e.g., in e-mails or orally.

If the goal is pre-hoc formal verification (as opposed to post-hoc), ambiguous requirements would not be acceptable. Then, the engineering process would need to be adapted to also produce unambiguous requirements.

B. Incompleteness of Requirements

An important aspect of formal verification is *completeness* of the set of requirements. In the present context, completeness means that the set of requirements on the software module (here, Steering) semantically entails a given set of (higher-level) requirements on the system (here, the vehicle). In our case, such higher-level requirements were available; however, for a formal analysis of completeness all requirements need a formal representation, including the higher-level ones.

In order to analyze completeness formally, we have identified three challenges. The first is requirement ambiguity, as discussed in Section VIII-A. Second, to formalize requirements on different levels, and for different parts of the system architecture, different specification languages are required (e.g., ACSL contracts and temporal logics). A formal treatment would require a unified framework for the logics used, which, to the best of our knowledge, does not exist. The third challenge is that *entailment* is typically defined so that falsehood entails any formula, meaning that a set of inconsistent requirements would be complete w.r.t. to any other requirement. Thus, the concept of *completeness of requirements* needs a careful formal definition, to match the industrial interpretation. For a possible solution to this, see [32].

C. Wrong Scope of Requirements

The term *scope of a requirement* here refers to the technical component, signals, and variables the requirement refers to. Related to this, three kinds of issues were observed. The first is that some of the requirements do not refer to the Steering module. In fact, and as reported in Section VII-D, out of the 32 requirements, 3 requirements are about configuration handling, and 2 about error handling, which are both implemented completely outside the Steering module.

The second issue is that, although the requirement expresses an intended function of Steering, some requirements refer to signals or variables outside the Steering module. As explained in Section VII, this was handled by projecting down the requirement onto Steering. That is, we created a breakdown to a set of requirements, where one is the requirement on Steering. To avoid the second issue, one option is to ensure that the scope matches exactly the interface of the software module.

Another option is to formulate the requirements using *assume-guarantee style contracts* [11], where both the assumptions and guarantees are allowed to refer to signals and variables outside of the interface. For example, a requirement stating "the output shall equal the unsigned vehicle speed" for a module that only takes a vehicle speed estimate, $\mathsf{vehSpeedEst}$, as an interface input, can be formulated as a contract with the assumption $\mathsf{vehSpeedEst} = \mathsf{vehSpeed}$, where $\mathsf{vehSpeed}$ is outside of the interface. With the guarantee $\mathsf{output} = |\mathsf{vehSpeed}|$, the module can satisfy the contract by using the assumption and computing $|\mathsf{vehSpeedEst}|$. An ACSL contract can then be defined, in which $\mathsf{vehSpeed}$ is represented by a ghost variable.

The third issue is that requirements refer to local variables of the Steering module. This was handled by introducing ghost variables in the ACSL contracts, while ensuring that all requirements referencing the same variable are formalized with an ACSL contract using the same ghost variable.

D. Mismatch Between Requirements and Code

Out of the 27 requirements applicable to Steering, 6 could not be verified even after matching the formalization of the requirements against the code. Note that, while Frama-C could not conclude that the code satisfies these 6 contracts, it could also not conclude that the code *does not* satisfy them. Rather, the verification result was inconclusive (timeout). This is the typical behavior of Frama-C, and deductive verification tools in general, when the code does not satisfy a contract. However, following the verification results, we performed a code analysis, from which we could judge the code to be incorrect w.r.t. the 6 requirements. To further convince ourselves, we re-implemented some parts of the code, so that the new code did satisfy the contracts.

Based upon discussions with engineers and results from testing, we judge the source of the mismatch to lie in the 6 requirements, i.e., we assume that the code is correct w.r.t. to the engineers' intention. We suggest two explanations as to why the 6 requirements are incorrect. Firstly, as mentioned above, engineers are in general not sufficiently trained to formulate correct requirements w.r.t. the intended behavior. Secondly, the requirements were not used directly for any formal part of the development. The only connection to a formal activity, that could potentially indirectly detect the incorrectness, has been their use as input when writing test cases, which have then been applied to STEERING. Interestingly, we found no reports of failed tests, so the test cases must have matched the code even though the 6 requirements did not.

E. Difficulty of Formalizing the Original Requirements

The work of formalizing the original informal requirements, all the way to ACSL contracts, presented a number of challenges. As explained above, the ambiguity and scope issues in particular caused problems. However, even after these were resolved, there were additional challenges.

To formalize the original requirements, we had to identify suitable requirement specification languages. We aimed for formal notations that are (1) sufficiently expressive to exactly match the interpretations of the requirement, (2) reasonably easy to use, (3) supported by the tool(s) used, i.e. Frama-C, and (4) are as few as possible to allow various analyses involving multiple requirements, such as consistency checks. During the work, we investigated the following languages: LTL, MTL, MITL, STL, TCTL, MFOTL, ACSL, timed automata, and hybrid automata. No single language met all four criteria, and so, balancing the four, we opted for LTL, MITL, and ACSL.

It should be noted that in total, the formalization task required a significant amount of work. Most of the work was performed in meetings with a mix of engineers, PhD students, and faculty. Each meeting was attended by 2-5 persons, and in total, we used around 40 hours of meeting time. During the meetings, only a few of the requirements within each category (see Section V) were typically discussed, since the remaining ones were conceptually similar.

One issue discussed in the meetings was how to represent temporal properties in the logics used. Another issue was, as described in Section VII, the need for ghost variables to represent requirement variables. Moreover, in some cases one requirement variable could refer to either the old value, or a new value calculated during execution of STEERING, making it difficult to relate requirements in a coherent way.

It can be noted that we did not systematically use *requirements patterns* [5]. The reasons are that, either the given requirement was easy to formalize, so there was no need to get help from a pattern, or alternatively, the requirement was so difficult that patterns did not help. That is, for the difficult requirements, there was typically no matching pattern. Furthermore, the language ACSL is, to the best of our knowledge, not supported by any pattern catalogue.

F. Requirements Breakdown Not Verified

As explained in Section VIII-C, many requirements have a wider scope than the STEERING module, and have to be projected onto STEERING. This step includes creating a breakdown from the formalized requirement down to a set of requirements in which one is an ACSL contract for the STEERING module. In general, it is important that such a breakdown is correct, and thus needs to be formally verified. This was not done because of three reasons. Firstly, the focus of the work was verification of the STEERING module, meaning that in the breakdown, the only requirement formalized in addition to the original requirement, was the ACSL contract for the STEERING module. Therefore, the requirements for the neighboring components were only implicit. Secondly and thirdly, in accordance with the discussion in Section VIII-B about completeness, for a formal analysis of the breakdown, we need a unified logical framework and working formal definition of breakdown.

G. Excluded Verification of Library Functions

As stated in Section I, the goal of the work was to verify the code contained in one module (i.e., source-code file). This code has a number of calls to functions defined in other files, referred to as library functions. Verification of these library functions was consequently not considered, i.e., we did not aim to verify the complete translation unit. To make the verification task possible, we specified ACSL function contracts for each library function. This was a significant challenge due to a lack of respecting modularity in the software design.

A consequence of adding contracts for the library functions is that the verification of STEERING is conditional on the correctness and implementation of those contracts. Thus, what we actually verify is that STEERING implements a contract in a wider sense, i.e., a contract with the assumption being the ACSL contracts for the library functions, and the guarantee being the main ACSL contract for STEERING. In accordance with established definitions of contracts [10], [32], [35], the semantics is that for any set of software modules, implementing the ACSL library function contracts (the assumption), these software modules integrated with the STEERING module shall implement the ACSL contract of STEERING (the guarantee).

H. Verification With Frama-C was Time Consuming

The formalization process resulted in a set of requirements on the main function of STEERING. Since our aim was to verify the entire STEERING source code, there was also a need to break down the requirements on the helper functions, i.e., the non-library functions called by the STEERING main function. Additional manual work was needed to annotate the library functions, and add ghost annotations. All combined, we estimate this effort to have taken several hundreds of person-hours by research engineers and PhD students.

IX. CONCLUSION

As an experience report, this paper has presented observations, insights, and lessons learned from an endeavor of using Frama-C to formally verify a piece of real industrial code, i.e. the STEE module. To summarize the work, it is relevant to ask: after performing the formal verification, can we conclude that the STEERING module was correctly implemented? In spite of the significant effort, our answer is unfortunately "no". Based upon the discussion in Section VIII, the reasons for this are the following:

1) Of the 27 ACSL contracts that were produced, 6 could not be verified by Frama-C.
2) It is not known whether the set of requirements is complete or not.
3) Due to their ambiguity, it is not known whether the requirements were correctly formalized or not.
4) Due to the too wide scope of the requirements, it is not known whether the ACSL contracts are a correct projection of the requirement onto the STEERING module.
5) The functionality of STEERING depends on library functions that were not formally verified.

However, it should immediately be added that, even though we could not formally conclude that the code is correct, we could also not conclude that the code is incorrect, i.e., it is fully possible that the code meets all intended expectations.

To address the problems listed above, we suggest the following general remedies:

- Exploit *formal-verification-driven development*, i.e., as in test-driven development, create the formal requirements *before* or together with the development of the software, to allow an *iterative* formal verification of the software w.r.t. the formal requirements. This would solve (or at least alleviate) problems 1, 3, and 4.
- Formally verify the breakdown of system-level requirements. This would solve problem 2. However, this would require also the system-level requirements to be formalized. Furthermore, as discussed in Section VIII, how to formally verify breakdowns is also an area where more research is needed, e.g., to support multiple requirements languages in the analysis.
- Problem 5, i.e., to formally verify the library functions, could be solved using Frama-C in much the same way as we verified the code of the STEERING module. Alternatively, an informal approach, such as testing, can be applied. However, to support formal reasoning using a combination of results from testing and formal verification, there is a need for a formal framework that underpins such a combination. Such frameworks have been investigated previously, e.g. see [19], but, to the best of our knowledge, not in the context of ASCL contracts.

In addition to the three suggestions above, which all aim to enable formal verification, the problem posed by the significant effort, time, and competence required for the formalization of the original requirements needs to be addressed. Possible solutions are the use of assistance from generative AI, and the design of alternative formal specification languages that are easier to use and understand, even for practitioners.

In summary, in order to reach a mature technology for formal verification of industrial software, there are still challenges remaining. It should be noted, however, that, based on the experiences reported in the present paper, the core of the verification method itself, namely the use of Frama-C to verify industrial code against ACSL contracts, can be deemed to be sufficiently useful. We believe that the remaining challenges can be solved, thus making formal verification technically viable in industry. What would then be left to answer is whether this technology is sufficiently cost-efficient as compared to informal techniques, which are the state-of-practice today. The answer to this question will surely depend on the level of criticality of the software at hand.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[2] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, jan 1996.

[3] Jesper Amilon, Christian Lidström, and Dilian Gurov. Deductive verification based abstraction for software model checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, pages 7–28, Cham, 2022. Springer International Publishing.

[4] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.

[5] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.

[6] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST. http://frama-c.com/download/acsl.pdf.

[7] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP Plug-in Manual*. CEA LIST. https://frama-c.com/download/frama-c-wp-manual.pdf.

[8] Bernhard Beckert, Thorsten Bormer, and Daniel Grahl. Deductive verification of legacy code. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, pages 749–765, Cham, 2016. Springer International Publishing.

[9] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, page 232–243, Berlin, Heidelberg, 1996. Springer-Verlag.

[10] Albert Benveniste, Benoit Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Formal Methods for Components and Objects*, pages 200–225. Springer, Berlin, Heidelberg, 2008.

[11] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. *Contracts for System Design*, volume 12. Now Publishers, 2018.

[12] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*. CEA LIST, Inria. http://frama-c.com/download/frama-c-user-manual.pdf.

[13] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[14] Frank Dordowsky. An experimental study using acsl and frama-c to formulate and verify low-level requirements from a do-178c compliant avionics project. *Electronic Proceedings in Theoretical Computer Science*, 187:28–41, August 2015.

[15] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In *Computer Aided Verification*, pages 1–16, Cham, 2014. Springer International Publishing.

[17] Carlo A. Furia, Matteo Pradella, and Matteo Rossi. Automated verification of dense-time MTL specifications via discrete-time approximation. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2008.

[18] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, page 163–173, New York, NY, USA, 1980. Association for Computing Machinery.

[19] Holger Giese, Stefan Henkler, and Martin Hirsch. Combining formal verification and testing for correct legacy component integration in mechatronic UML. In Rogério de Lemos, Felicita Di Giandomenico, Cristina Gacek, Henry Muccini, and Marco Vieira, editors, *Architecting Dependable Systems V*, volume 5135 of *Lecture Notes in Computer Science*, pages 248–272. Springer, Berlin, Heidelberg, 2008.

[20] Dilian Gurov, Christian Lidström, Mattias Nyberg, and Jonas Westman. Deductive functional verification of safety-critical embedded C-Code: An experience report. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 3–18, Cham, 2017. Springer International Publishing.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[22] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd edition, 2004.

[23] International Organization for Standardization. Programming languages—C, 1999. ISO/IEC 9899:1999.

[24] International Organization for Standardization. Road vehicles controller area network (CAN), 2015. ISO 11898-1:2015.

[25] International Organization for Standardization. Road vehicles—Functional safety, 2018. ISO 26262:2018.

[26] Arut Prakash Kaleeswaran, Arne Nordmann, Thomas Vogel, and Lars Grunske. A user study for evaluation of formal verification results and their explanation at Bosch. *Empirical Software Engineering*, 28:125–, 2023.

[27] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 27(3):573–609, may 2015.

[28] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[29] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004. https://misra.org.uk/misra-c.

[30] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag, Berlin, Heidelberg, 2007.

[31] Mattias Nyberg, Dilian Gurov, Christian Lidström, Andreas Rasmusson, and Jonas Westman. Formal verification in automotive industry: Enablers and obstacles. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, page 139–158, Berlin, Heidelberg, 2018. Springer-Verlag.

[32] Mattias Nyberg, Jonas Westman, and Dilian Gurov. Formally proving compositionality in industrial systems with informal specifications. In *International Symposium on Leveraging Applications of Formal Methods*, pages 348–365. Springer, 2020.

[33] Amir Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[34] Tobias Runge, Thomas Thüm, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. Comparing correctness-by-construction with post-hoc verification—a qualitative user study. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops*, pages 388–405, Cham, 2020. Springer International Publishing.

[35] Alberto L Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 18(3):217–238, 2012.

[36] Thomas Vogel, Marc Carwehl, Genaína Nunes Rodrigues, and Lars Grunske. A property specification pattern catalog for real-time system verification with UPPAAL. *Information and Software Technology*, 154:107100, 2023.

[37] Bruce W. Watson, Derrick G. Kourie, Ina Schaefer, and Loek Cleophas. Correctness-by-construction and post-hoc verification: A marriage of convenience? In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, pages 730–748, Cham, 2016. Springer International Publishing.

[38] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), oct 2009.