Formal Verification in Automotive Industry: Enablers and Obstacles

Mattias Nyberg[1,2(✉)], Dilian Gurov[1], Christian Lidström[2], Andreas Rasmusson[2], and Jonas Westman[1,2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Systems Development Division, Scania AB, Södertälje, Sweden
mattias.nyberg@scania.com

Abstract. We describe and summarize our experiences from six industrial case studies in applying formal verification techniques to embedded, safety-critical code. The studies were conducted at Scania over the period of eight years. Despite certain successes, we have so far failed to introduce formal techniques on a larger scale. Based on our experiences, we identify and discuss some key obstacles to, and enabling factors for the successful incorporation of formal verification techniques into the software development and quality assurance process.

1 Introduction

Formal methods are making their way only slowly into industrial practice for quality assurance of general software (SW). Their adoption in the domain of embedded, safety-critical systems, however, has seen much progress over the last years, as evidenced by the industrial case studies reported in the literature, e.g. see [18]. One reason for this development, from an industrial perspective, is the increased analysis effort advocated by various standards to achieve functional safety of such systems. The automotive functional safety standard ISO 26262, for instance, recommends formal verification for higher levels of criticality. Another reason is related to feasibility: the relatively smaller size of embedded code as compared to arbitrary applications, and the constraints on how such code is structured in order to safeguard against potential unwanted behaviours, make the application of formal analysis and verification techniques a viable complement to the traditional testing approaches. One example of a company developing embedded SW is Scania, a leading manufacturer of commercial vehicles, and specifically heavy trucks and buses. A large part of the embedded C-code developed at Scania is safety-critical, and a considerable effort during code development and deployment is spent on quality assurance. On top of the traditional testing methods, Scania is exploring the possibility for integrating various formal methods, such as deductive verification and model checking, into the code design and quality assurance process. The main motivation for this is the increased safety requirements resulting from innovative trucking solutions such as platooning and autonomous driving.

© Springer Nature Switzerland AG 2018
T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 139–158, 2018.
https://doi.org/10.1007/978-3-030-03427-6_14

In the present paper, we present a summary of our experiences with applying formal verification techniques to a number of industrial case studies at Scania. The first three studies concern the deductive verification of requirements on C-code modules by means of semi-automated annotation of the code (for the given requirements), and the use of an off-the-shelf verification tool, VCC[6], to statically check the annotations. The fourth study investigates the application of two popular model checkers, the Simulink Design Verifier and Uppaal, to verify requirements formalized as Simulink models. The fifth study evaluates the application of learning-based testing, a form of black-box testing executed in a virtualization environment, for checking requirements expressed in temporal logic. The last of these studies concerns the verification of correctness of the requirement breakdown of top-level requirements down to component-level requirements, following an hierarchical architectural description of the system.

Based upon the industrial case studies, but also considering other important aspects affecting the industrial usability of formal verification methods, the main contributions of the paper are to (*i*) summarize and generalize the observations and experiences from the conducted case studies, (*ii*) identify and discuss factors that *enable* wide-scale adoption of formal verification techniques, and tools supporting these, in the software development and quality assurance process within the automotive industry, (*iii*) identify and discuss factors that are *obstacles* to this development, and (*iv*) propose a roadmap of tasks to facilitate a near-time widespread usage of formal verification in industry. With this, our work contributes to the wider, ongoing discussion on how to facilitate the transfer of verification technology from academic research to industrial practice (see, e.g., [2, 18] among many others).

Structure of the Paper. The remainder of the paper is organized as follows. In Sect. 2 we describe the industrial context in which we strive to apply formal methods, while Sect. 3 describes our case studies and identifies enablers and obstacles related to these. In Sect. 4, we discuss other industrial factors, not exposed in the case studies, but still related to the adoption of formal verification. In Sect. 5, we summarize and generalize our observations and experiences, and we discuss how enablers can be utilized and obstacles overcome to reach a successful adoption of formal methods in the automotive industry. Section 6 concludes the paper.

2 Context of Study

This section describes the industrial context in which the case study was conducted. As described in the introduction of the paper, Scania is a leading manufacturer of heavy trucks and buses. The R&D department consists of roughly 4000 engineers and, of these, about 1000 are working on the development of the electrical system in the truck. This includes design of electronic control units (ECUs) but the main effort is the development of embedded SW.

Most parts of the software are safety critical, i.e. a bug has the potential to cause accidents, and in many cases with fatal consequences. In alignment with

the general trend in the automotive area, a substantial part of the current development efforts are related to ADAS (Advanced Driver Assistance Systems) and Autonomous Driving (AD) [16]. This involves a huge amount of safety critical software. To support the development of safety-critical systems, Scania applies the automotive functional-safety standard ISO26262 [10].

In comparison with other automotive companies, Scania relies to a very high degree on in-house development. As a consequence, Scania has developed in-house expertise in SW development. Also, it has been possible to optimize SW with respect to specific Scania needs, for example resulting in higher execution performance, fewer line of codes, and lower complexity.

Scania has adopted the principle of evolving product lines. This means that there is only one product line, with e.g. an 8-wheel drive mining truck and a city bus representing just two different configurations. The product line is evolving, in the sense that some parts of the construction going into production, typically SW, are changed every week. As a consequence, there are no 'model years'. Furthermore, Scania vehicles are connected, so SW updates can be managed 'over-the-air' and be initiated at any time. All these circumstances sum up to a need for highly competent product-data and configuration-management systems, with the ability to track the exact set of parts and configuration of each Scania vehicle ever produced and over its lifetime. In fact, such highly competent product-data and configuration-management systems do not exist commercially, so Scania has been forced to also develop this in-house. Yet another consequence of the evolving product line, is that a large part of the software is legacy; each single update typically introduces only a small part of completely new SW. Most SW remains the same or is the result of minor incremental improvements.

Regarding processes and organization, Scania has since long adopted the principles of lean and agile. The adoption is general, covering development, production, and after market support, but is particularly articulated in the area of SW development. Focus is not on documentation, for example of requirements, but instead on the people involved and their knowledge and competence. Each developer has the responsibility to understand customer needs, and develop the product according to these needs. Other companies often split the development in 'layers' with different responsibilities such as requirement elicitation, high level design, low level design, implementation (programming), and testing. In contrast, Scania generally adopts a flatter structure where the same engineer may be responsible for all these tasks, but only for their part of the construction. As a result of these principles, Scania is able to obtain an industry-leading product [1] in spite of a lack of heavy documentation.

Lastly, the engineers developing embedded systems at Scania, including programmers, come from a variety of backgrounds. Most common education is mechanical engineering followed by electrical engineering or applied physics. Engineers with computer science background are rare. This means that engineers learn the practice of software engineering and programming, not from state-of-the-art university courses, but from other practicing engineers.

3 Industrial Case Studies

Several studies have been performed at Scania, evaluating the incorporation of different forms of formal verification into parts of the embedded systems development process. The case studies are sorted into 4 groups according to the method of formal verification used. The case study groups are described in Sects. 3.1, 3.2, 3.3 and 3.4, which includes the conclusions drawn for each specific method.

In Sect. 3.5 we summarize the enablers and obstacles identified in the different case studies, and in Sect. 3.6 the results are discussed more generally.

The purpose of the industrial case studies was to evaluate the suitability of different verification techniques for general-purpose use within the automotive industry. The cases were chosen based on criticality to safety, generality, and availability.

In all cases the verification was performed by people with some experience with formal methods but who cannot be considered experts within the respective areas, with the assistance of researchers familiar with the topics. Much of the work was performed within MSc theses, and as such the effort required in terms of time was several months.

3.1 Deductive Verification of C Code

Evaluating and improving upon methods for deductive verification of C code has been an ongoing project at Scania. This section describes three case studies, and the different tools and methods used and developed for this purpose.

Tool Support. The primary tool used in the three studies is VCC [6]. Other tools have also been experimented with, most prominently Frama-C [7] with a plugin called WP. Both of these tools use deductive reasoning to prove properties of C code. The verification is function-modular, meaning that functions are verified independently, relative to the specifications of the other functions. These specifications are given in the form of function *contracts*, which are provided in the source code by means of annotations, and for which each tool has its own annotation language.

Each of the tools has its advantages and disadvantages. For example, while VCC supports more fine-grained reasoning about concurrency and complex data structures, compared to Frama-C it appears less mature and does not support verification of floating-point arithmetic. More details about the tools and our reasoning for choosing them can be found in [12].

In addition to the tools performing the actual verification, we have developed our own prototype tool to automate parts of the annotation process, which we call *Annotation Weaver*. This tool automatically inserts into the code so-called *auxiliary annotations*, which are annotations that are needed independently of the functional requirements, and that can be generated by combining analysis of the source code and data from an interface specification (defining e.g. the types and ranges of input/output variables). Examples include ensuring the validity and separation of pointers.

Industrial Cases. Three different software modules have been used in the context of deductive verification.

The first of these modules is called STEE, which is a C module of the embedded system that controls the dual-circuit steering system in vehicles. The initial work was performed in two MSc projects [8,12], and a report on our continued experiences was published in [9], which we summarize here. This module has an associated specification document, describing 27 requirements on the software. The requirements were stated informally, and some of them were safety-critical. The case study focused on requirements that were strictly *functional*, i.e., requirements that define output values as a function of the input values. In the existing requirements this functional relation was often described by means of intermediary requirement variables (variables that are neither input nor output variables and that might not be represented in the software). A formal requirements model that captured these properties was defined, according to which the module requirements were formalized. We also found that this model allows the set of requirements to be understood and visualized as a combinational logic circuit. A visualization of all case study requirements affecting the output variable SCHS is shown in Fig. 1.

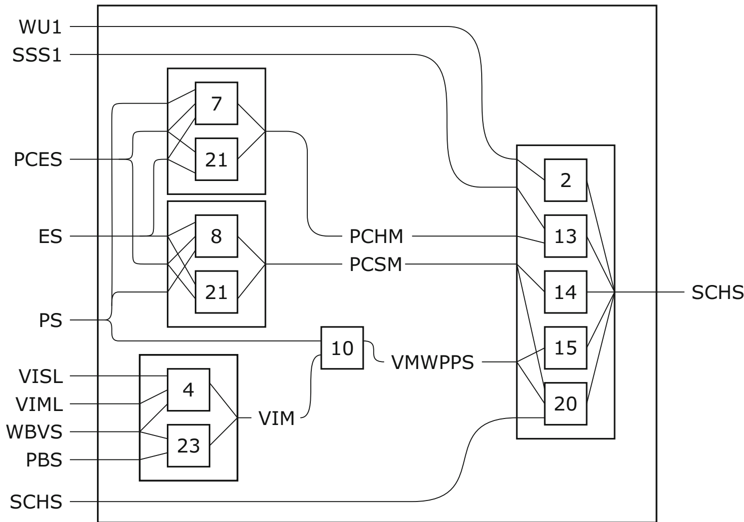

Fig. 1. Combinational logic circuit of case study requirements.

In the report, a process was described for the manual annotation and verification of the C module with respect to the formalized requirements. The C module consisted of 10 functions in total, of which one acted as an entry-point function. Thus, the formal requirements could be converted into a contract for this function, which then had to be decomposed through the function hierarchy.

The next case was a C module called VLTQ. This module is responsible for calculation of torque losses. Compared to the module addressed in the previous case study, VLTQ uses more complex programming constructs (such as loops, which are completely absent in STEE, to perform interpolation), and performs floating-point arithmetic.

The specification for this module was given as a flow chart, with a small part of the module specified in more detail in the form of a finite-state machine. The purpose of the case study was two-fold: to continue the exploration of automating the verification process, and to manually verify a single safety-critical aspect of the module. The property to be verified was one stating that the summation of several contributions to the total torque loss should be within a given interval.

The most recent case is the C module UAPC (Unintended Activation of Planetary Clutch), which is a module monitoring a safety-critical output value of another module. The module has a single requirement, which is not strictly functional as it is both temporal and stateful.

Results. Out of the 27 existing STEE requirements, 14 were identified as functional and specific to the case study module. Of these, 10 were successfully verified, and a possible inconsistency was identified between two of the requirements. The remaining 4 requirements were not verified because of lack of time. Verification of the entire annotated module took 165 s, and the function which required the most time took 65 s. The case study also resulted in the identification of guidelines for writing code and requirements that facilitate formal verification, as well as descriptions of several common obstacles and solutions for dealing with them.

We also experimented with inlining functions to avoid having to solve the problem of decomposing function contracts. However, inlining just a few functions of the STEE module resulted in the verification time of VCC increasing to unreasonable levels, in some cases not terminating within hours.

In the VLTQ case we never managed to verify the desired property. One of the reasons for this was the fact that VCC lacks certain fundamental features such as reasoning about floating point arithmetic, something which is commonly found in the C code developed at Scania. Another reason was the lack of specification, both in terms of precise description of the wanted functionality, as well as in description of the interface.

One outcome from the latter case studies was the above-mentioned *Annotation Weaver* prototype tool for automatic generation and insertion of auxiliary annotations. More restrictions on the source code were also identified, in order to ease automation of the verification process.

Conclusions. In our experience report [9], we concluded that deductive verification of embedded C code is a viable option, but requires rigorous formalization and deep understanding of the tools and processes. We also noted that automation of large parts of the process is a requirement for more widespread use, and described possible paths towards this goal.

Our belief in the value of automation of the verification process has since been reinforced. Even with the help of a tool for inserting auxiliary annotations, and with built-up expertise from previous experiences, successfully verifying properties of complex code is still a laborious task.

We also conclude that putting restrictions on the C code is a necessity. This is particularly true for the automated decomposition of contracts, since generating specifications is very hard in the general case. A possible solution to this is the use of smaller scale monitors, such as UAPC, that validate the safety-critical outputs of larger modules. Such constructs may even enable the use of inlining of functions for verification purposes.

The evaluated verification tools also lack certain fundamental features, such as reasoning about floating points in the case of VCC. The use of floating point arithmetic cannot simply be restricted, instead such shortcomings in tools need to be worked around.

Finally, formalizing the requirements involved resolving inconsistencies and ambiguities that are not apparent in informal requirements, but which makes successful verification impossible. Even worse, one module lacked even informal requirements.

3.2 Model Checking of Simulink Models

Another technique for formal verification of requirements that has been explored is model checking, and more details about our experiences can be found in [3].

Tool Support. In this case study the model checkers Simulink Design Verifier (SDV)[1] and UPPAAL [5] were compared qualitatively, with respect to their ability to be used in an actual industrial process for formal verification of requirements. SDV was chosen since it is an embedded model checker in Simulink, which is nowadays a de facto standard for design and evaluation of embedded systems. UPPAAL was chosen since it is a popular and well documented model checker, and has been successfully applied to problems of similar characteristics.

The model checker of SDV is called Prover Plug-In. Given a Simulink model, a requirement specification is expressed as a combination of *Proof objectives* and *Proof assumptions* that define a relation between the inputs and the outputs of the model. Proof objectives and assumptions can be modelled by logical and relational operators, MATLAB functions, or Stateflow graphs. A Proof objective is proven valid if there does not exist any state of the model that violates the Proof objective, given restrictions on the inputs, as specified by Proof assumptions. When a violation is detected, SDV generates a test vector that can demonstrate the violation in simulation.

UPPAAL is a model checker based on the theory of timed automata [5] that allows symbolic representation of time. UPPAAL provides a graphical editor, a simulator and a verifier. The verifier verifies properties that are defined as a subset of TCTL (Timed Computation-Tree Logic) and whenever a property is

[1] http://www.mathworks.se/products/sldesignverifier/.

not satisfied, it provides a counter example in the form of a trace that can be explored with the help of the simulator.

Industrial Case. The case study target was the Fuel Level Display (FLD) SW component, whose purpose is to provide an estimate of the total fuel level using a Kalman filter.

A subset of the specification consisting of 7 functional requirements was chosen for verification. Out of these requirements, 5 of them were dependent on time, while the other 2 were not. The emphasis was not placed on proving correctness of the system design, nor on the computational efficiency of the tools, but on the ability of the tools to be used within the organization in an actual process, in order to identify the problems faced by regular engineers performing model checking.

Results. The formalization of the requirements with SDV seemed fairly easy to grasp for the engineers, since it is based on function blocks and this is a well-known concept in systems and control engineering. Moreover, since SDV uses the original Simulink model, the interfaces between the submodules are clearly defined. This becomes very helpful because system requirements are typically defined in terms of these submodules/interfaces.

Table 1 shows the time needed for each verification. It is remarkable that some of the proofs needed a substantial amount of time, despite the simplicity of the system. This raised concerns about the scalability of this technique.

Table 1. SDV verification results.

Req.	Time (s)	Req.	Time (s)
AER201-12	2	AER202-2	3
AER201-13	238	AER202-3	25
AER201-14	238	AER202-2	1
AER201-15	24		

Using Uppaal, the engineers first had to manually construct a model of the FLD component based on timed automata, and subsequently formalize the requirements into TCTL properties. These manual activities required extra supervision, and needed much more time than initially expected. The first problem was associated with the fact that Uppaal has no support for using fixed and floating point numbers in the models. A solution to this is to scale such numbers up to integers, which for complex systems requires certain expertise. Second, due to the degree of abstraction, the engineers had some difficulties with mapping the elements of the timed automata with the real elements of the system.

Due to this, and because of the strict time requirements of the project, not all of the requirements in Table 1 were verified with Uppaal. For the requirements that were verified, the verification time was substantially lower than for SDV, e.g. the verification time of AER202-2 was 0.2s.

Conclusions. The insights gathered from the case study is that SDV offers key features, such as the support for fixed point and floating point numbers and clearly defined interfaces, which appeal to the typical engineer. As an embedded feature of Simulink, which is widely used in industry, it eliminates the problem of having to transform the system to be verified into a formal model. Instead the requirements can be formalized as Simulink models, which the engineers found easy because of their familiarity with Simulink.

Uppaal, on the other hand, offers high performance as compared to SDV, and relies on well-founded theories for handling complexity. Uppaal requires the model to be transformed into timed automata, which is a relatively unknown concept to a typical engineer. As such, the formalization of the system to be verified requires considerable expertise and constitutes a real obstacle for its integration with current industry practices. This process would itself need to be verified or automated.

3.3 Learning-Based Testing

Our next case study evaluated the use of learning-based testing, a form of black-box testing, as a means for quality assurance of embedded safety-critical code.

Tool Support. LBTest [14] (see Fig. 2) is an automated, combined test-stimuli generator and evaluator that implements the methodology of learning-based testing [13]. LBTest is used to generate test-stimuli and evaluate responses from the system-under-test (e.g. an ECU). LBTest receives requirements as input, but has no model of the system-under-test. Instead, LBTest incrementally learns a model of the black-box system by issuing sequences of input and updating its learned model. The learning algorithm generally explores the state-space in a breadth-first manner, and a model checker receives the tentative models learned. Each such tentative model is checked against the requirements and, if the model is non-compatible, the model checker produces a concrete counterexample input/output sequence. This counterexample is sent to the system-under-test for validation. Most often, executing the counterexample will mispredict the output of the system-under-test and this will lead to amendments to the learned model, but when the prediction is correct, a true requirements violation has been detected.

Industrial Case. To evaluate the feasibility of learning-based testing for automotive applications, a benchmark experiment was performed [4,11], comparing LBTest to an existing test-suite on the Dual Circuit Steering (STEE) application, a sub-system of the ECU software.

As LBTest expects requirements formulated in PLTL (Propositional Linear-time Temporal Logic), the set of existing, informal requirements first had to be translated into PLTL. Out of the original 32 informal requirements, 11 could not be formalized in PLTL since some were not describing any actual relation between inputs and outputs and others referred to a specific platform for hardware-in-the-loop (HIL) testing. This ratio seems to be on par with other

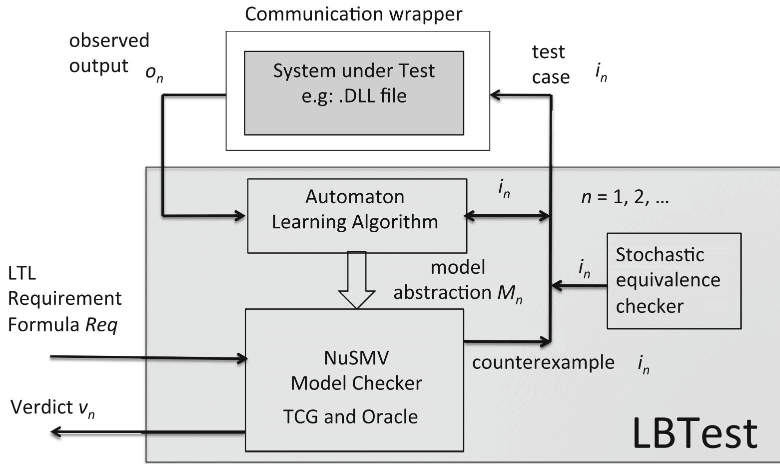

Fig. 2. LBTest architecture

similar studies [4]. The remaining 21 requirements could be formalized in PLTL, although they required extensive reformulation since they made use of variables denoting assumed internal states and internal signals. Also, the informal requirements did not always systematically model all the corner cases necessary for an unambiguous translation into formal requirements.

Results. To estimate the strength of LBTest to an existing test-suite (piTest), 10 errors were manually injected into the system-under-test and the recalls of the errors for each test-tool were compared. LBTest failed to detect 2 of 10 injected errors whereas piTest failed to detect 3 of 10 injected errors [4]. The types of errors that were detected or missed by the two tools may give an indication of their respective strengths and weaknesses. The errors not detected by LBTest were both changes to boundary values, whereas the errors not detected by piTest where changes to output values. No conclusive analysis was made for the missed errors. One integration of the testing procedure took about 7 h to terminate for an empirically estimated final model coverage at 97%. There are many potential ways to improve the performance of LBTest, but at the time of the experiment, the long turnaround time was an impediment to more extensive mutation testing.

Conclusions. Having performed the case study, we believe that empirical black-box methods, where tests are automatically generated from formal requirements, add testing-value since they should be less likely to miss errors due to feature interactions. Also, black-box testing puts minimal constraints on how the actual software is produced (i.e., what languages, tools or processes are used to produce the software), and does not even require access to the source code. However, as indicated by the 3 out of 10 missed injected errors, empirical methods can never conclusively prove the absence of bugs, and also, no amount of testing will automatically generate high quality software-design.

3.4 Verification of Requirements and Variability

At Scania, an in-house tool for specification, structuring, and verification of requirements has been developed for research purposes [17]. This section describes a case study where this in-house tool was used to perform a breakdown of a top-level safety requirement (a safety goal) all the way down to requirements on software modules.

Tool Support. In the in-house tool, requirements can be specified and linked together across different abstraction levels, forming a hierarchical breakdown of top-level requirements down to low-level requirements on software and hardware. The tool also integrates with other parts of the Scania toolchain by fetching data that various other tools have made available in a central database. This database contains information about internal software variables, CAN signals, truck configuration parameters (such as names, ranges, and types) and other data that has proven valuable for formal reasoning about specifications.

The tool supports formal specification of requirements in a simple logic language, as well as assigning validity of requirements in terms of production dates and truck configurations. These features enable simple sanity checks such as syntactic validation of requirements, and whether two linked requirements are ever valid in the same configuration or at the same production date, as well as checks of more complex properties such as completeness and consistency. Utilizing the data from other tools, we can semantically check whether values used in requirements and configuration specifications are in the actual range of the referenced signals or parameters. Combining these formal aspects, it is possible to verify, for example, whether a high-level requirement is fulfilled (i.e. semantically entailed) by its linked lower level requirements at a certain production date for all configurations in which the high-level requirement is valid. All checks and verification are performed on-the-fly, using the SMT solver Z3 [15].

Industrial Case. The case study target was the dual-circuit steering system, already described in Sect. 3.1. In the case study, a safety goal for the dual-circuit system was formalized and broken down in several levels until requirements could be allocated to the STEE module. Verification was performed at each level to check that every requirement was entailed by its linked requirements at the lower level.

Results and Conclusions. An experience gathered from the case study was that the task of formalizing and structuring requirements to be able to verify entailment is truly difficult. Especially when working with the high-level requirements, despite appearing easy to formulate in concept, many ambiguities and unclarities arose when trying to formalize them. One reason for this difficulty with formalizing the requirements was the low quality of the existing informal requirements, which in turn suggests that writing requirements in general is hard. A feature of the tool that gave support during the formalization was the on-the-fly check/verification of requirement conditions. The fact that the feedback was

provided almost immediately was critical since it was then in most cases clear what action caused the violation of a condition. It was even found that getting such immediate feedback was so important that it could be beneficial to even increase performance at the cost of added false-negatives.

3.5 Summary of Enablers and Obstacles

A comprehensive summary of the enablers and obstacles identified in the different industrial case studies is shown in Table 2, listed in the order they are discussed below.

In all of the case studies problems with performance and scalability were encountered. This can to some extent be mitigated by taking verification into account both when formalizing the requirements (so that they can be efficiently verified) as well as during development of the application, whether it is written in C or modelled in Simulink (e.g. by putting restrictions on the allowed constructs). In any case, a deep understanding of the relevant tools is necessary.

Another common thread is that formalization and (in the case of deductive verification) annotation of requirements is a very time-consuming and complex task. As such, this process needs to be automated as far as possible, and for tasks that cannot be automated, as much support (e.g. feedback) as possible needs to be provided by tools. Because of the complexity, formalization is also prone to being erroneous, and one solution to this are tools supporting the validation of requirements.

In several case studies the formalization was even found to be impossible due to ambiguities and contradictions in the requirements, which suggests that formalization is a valuable task in itself since it exposes such deficiencies.

3.6 Discussion

In order to formally verify a system/SW against its requirements, both the system/SW and its requirements first need to be formalized, i.e. described mathematically. Formalizing complex system or SW properties is indeed difficult, and an observation from working with the presented case studies is that this difficulty seems to grow with the level of abstraction. That is, it is part of every-day development to engineer formal products, e.g. code. Reasoning about what the code does is hard, but feasible. Formalizing what the code does, e.g. in terms of pre- and post-conditions, is very hard. Formalizing the overall property that the code is intended to do is even harder. What is most difficult is formalizing and structuring requirements to be able to verify relations between requirements.

This difficulty is reflected in the quality of engineering artifacts at these abstraction levels. This was an issue in all of the case studies: the high-level engineering artifacts that we were supposed to verify against were highly ambiguous and the only artifact that could truly be trusted was the code (*"code is king"*) or the model that the code was generated from. Thus, we had to look into the code/model to resolve the ambiguities and 'understand' what the high-level artifacts were expressing. Naturally, we could claim that we had good judgment

Table 2. Identified *enablers* (E) and *obstacles* (O) in each case study group using methods: (1) deductive verification, (2) model checking, (3) learning-based testing, and (4) requirements verification.

ID	Conclusion	E/O	CS Group			
			1	2	3	4
1	Deep understanding of tool and method required	O	✓	✓		✓
2	Restrictions on code	E	✓			
3	Tool lacks capabilities (e.g. floats)	O	✓	✓		
4	Automation	E	✓	✓	✓	✓
5	Inefficiency/scalability issues	O	✓	✓	✓	
6	Monitor small parts of code	E	✓			
7	Low-quality informal requirements	O	✓	✓	✓	✓
8	Lack of requirements	O	✓			
9	Formalization of requirements is hard	O	✓	✓		✓
10	Formal modeling of the code is hard	O		✓		
11	Time-consuming manual work	O	✓	✓		
12	Familiar requirement formalization	E		✓		
13	Feedback to users	E	✓			✓

when doing this; however, the question still stands if we simply verified that the implementation does what it expresses rather than what it is intended to do.

One difference that might contribute to this discrepancy in quality between high-level artifacts and code is that when working with the latter, one receives almost immediate feedback from the tools operating on it. In contrast, when working with structuring high-level requirements, there is little to no feedback support provided by tools. Working towards providing such support in the requirements and variability case study, we indeed found that it was critical that a user gets immediate feedback when violating a requirement condition, so that it becomes evident which action caused the violation. We even found that getting such immediate feedback was so important that it could be beneficial to increase performance at the cost of increased false-negatives.

Another observation is that the difficulty in formalizing grows if the formalization is to be done by someone who is not an expert in the chosen formalism. This was observed in the case study comparing UPPAAL and SDV where a big obstacle for the integration of UPPAAL with current industry practices is the fact that the user is forced to formalize a system and its requirements as timed automata and CTL formulas, which are relatively unknown formalisms to a typical engineer. UPPAAL also does not provide support for fixed and floating point numbers, which means that it requires scaling up to integer numbers, which requires some expertise. In contrast, using SDV requires only knowledge in modelling using Simulink blocks, a formalism used daily by many engineers.

However, as shown in the case study, SDV does not scale very well, and thus, cannot be used to verify an arbitrary model; knowing which models can and which cannot be verified (due to, e.g., scaling issues) requires experience and inside-knowledge of the underlying theory and applied algorithms.

Thus, both SDV and UPPAAL suffer from critical drawbacks with respect to being applied in practice. Most notably, their usage is rather restricted to people with expertise and in-depth understanding of the tools and their underlying theory. This observation is common for all of the tools used in the case studies. E.g. when using VCC understanding the underlying theory of the tool is essential to writing annotations that may be successfully verified and, perhaps more importantly, to avoid writing annotations specifying an incorrect behaviour of the code that is then successfully verified because of software bugs.

In addition to this similarity, VCC suffers from the same problem with scalability, which was made evident when trying to inline the helper functions of the STEE module in order to avoid the manual decomposition of top-level function contracts.

Using exhaustive, formal methods in parallel with developing the software is more likely to ensure that good software development practices will be used (e.g. designs with minimal amounts of undefined internal state).

4 Other Factors

In Sect. 3, we draw conclusions of enablers and obstacles based upon solely the case studies. In the present section, we try to add the perspective of other factors, namely management and SW architecture.

Management Commitment. In general, engineers and management are typically positive to, and understand, the benefits from using formal verification such as higher confidence, automation, etc. However, there is often a lack of understanding of what is actually needed to make these methods work in practice. This includes not understanding the level of effort and competence needed to formalize the systems and requirements to be analyzed.

Formal verification is not yet state-of-practice in the automotive industry. This means that to start practicing formal verification, new costly resources and competences need to be added. In a cost-aware development department, any new cost needs to be strongly motivated. To motivate a recruitment in the area of formal verification, upper management needs to be convinced about the benefits. However, as observed in the case studies, it is not at all clear that formal verification is technically ready for general automotive industrial practice. Furthermore, management decisions are often sensitive to general trends; i.e. if it is known that other companies are practicing formal verification or are planning to do so, it is easier to motivate recruitment. However, currently, formal verification is not part of a general automotive trend.

Having safety critical software should increase the need for formal verification. However, ISO26262 *recommends* but does not *require* the usage of formal

verification. So there has to be reason to go beyond what is required in ISO26262. One reason could be that the cost or time required of verifying a system with formal methods is lower than for traditional methods. This should be more articulated for parts of the vehicle that are more safety critical, i.e. having the highest safety integrity level. Since ADAS and AD involves more safety critical components, this trend might very well reach a turning point when the amount of verification of highly critical components is so large that the limitations of current testing methods become a blocker for introducing this new technology into the vehicles.

Legacy Systems. Formal verification comes with restrictions on the code and requirements having certain structures and being written following certain principles. Having systems consisting of large amounts of legacy code with legacy requirements often implies that these restrictions are not respected. Thus formal verification can not be used or the SW and requirements need to be rewritten, making formal verification harder to justify from a cost perspective.

Architecture and SW Complexity. The fact that embedded systems are developed in-house, may have both positive and negative effects on architecture and code complexity. A software tailor made for only one use case, is likely to have the consequence of a less general architecture. This stands in contrast to SW developed by suppliers, that are mandated to follow industry standards, such as AutoSAR, and provide a general architecture possible to adapt to all its customers. The comparably smaller focus on SW architecture makes systematic usage of formal verification difficult, especially using compositional verification.

On the other hand, a more streamlined and less general SW is likely to also have a positive effect on SW complexity. An example is the execution model used at Scania. Since it is known exactly when and which code is needed to be executed, relying on a general-purpose operating system is unnecessary, and a very simple real time scheduling principle can be used, for example relying to a high degree on fixed time scheduling, avoiding many issues associated with concurrency.

Product line principles have likely a negative effect on SW complexity. A similar negative effect may be seen in the requirements. For example, to support many product variants, requirements need in general to be parameterized or instantiated into many variants. This combined with the need for keeping requirements decomposition consistent is a substantial challenge.

In addition to architecture, a general adoption of formal methods and tools requires a high degree of information consistency across the organization. For example if an effort is made to prove the correctness of a system producing a signal X, but then the system consuming the signal has a different interpretation of the same signal X, a correctness proof of the second system will not help to guarantee correctness of the total combined system. To avoid this situation, general cross-system usage of formal verification needs to be backed up by a formal data-model, and in turn an information model.

4.1 Summary of Enablers and Obstacles

In accordance with Sect. 3, we summarize in Table 3 the enablers and obstacles identified above.

Table 3. Identified *enablers* (E) and *obstacles* (O) related to other factors.

ID	Conclusion	E/O
14	Methods are efficient for at least some cases	E
15	Resistance to process and methodology changes	O
16	Evidence of method matureness and efficiency is missing	O
17	Method usage requires new recruitement	O
18	Not required by the process standards	O
19	Many upcoming safety critical applications	E
20	Unnecessarily complex software	O
21	Bad system and SW architecture	O
22	Legacy SW	O
23	Generic parameterized software	O
24	Standardized architectural frameworks	E
25	Complex variability	O

5 Exploiting Enablers and Overcoming Obstacles

Tables 2 and 3 summarize 25 enablers and obstacles found in the case studies and the analysis of business-related factors inherent to the automotive industry. As a next step, we analyze the obstacles to identify remedies, and analyze the enablers to identify means through which they can be exploited. This is presented below, by grouping together similar remedies and exploitations. Based on this, we then suggest a roadmap of concrete tasks to be performed.

Improvement of Formal Verification Tools (3, 4, 5, 10, 11, 13). Tools need to be improved to add capabilities, e.g., handling of floats. Tools also need to be largely automated, especially concerning annotation of the code to be verified. Tools need to become more efficient for larger problems, i.e., scalability needs to be addressed. Tools need to have immediate and informative feedback to the user. If the target language of the verification tool is different from the implementation language of the code, then automated transformation from code to the verification target language is needed. Alternatively, tools need to be developed such that they can handle the implementation language of the code.

Improvement of Methodology on SW and Architecture (1, 2, 17). Restrictions on code for verification tools to be efficient need to be identified. Also, patterns of code, where formal verification is likely to work well, need to be defined. In general, the methodology has to be simple, and education material for engineers has to be developed. The aim should be a methodology that does not require deep expert knowledge. As part of the methodology improvement, general requirements on the code, SW and system architecture need to be identified.

Improvement of SW and Systems (6, 20, 21, 22, 23, 25). Development of new SW and systems need to follow the general requirements on the code, SW and system architecture identified above in the remedy on SW and architecture methodology. This includes structuring the code into highly safety-critical components, to be formally verified, and non- or less safety-critical components, that do not need to be verified formally. To avoid complex verification tasks, complex SW has to be avoided. In addition, legacy SW and systems need to be refactored using the same principles.

Improvement of Tools and Methodology for Requirements Engineering (7, 8, 9, 12, 13, 24). Engineers writing requirements, informal or formal, need knowledge of how to write good requirements. It is the authors' observation that existing books and industrial courses in requirements engineering are not at all sufficient to make engineers produce requirements of high enough quality to be formalized. The problem is that requirements engineering researchers, who are typically authors of books, and experienced industrial practitioners, who are typically the teachers in courses, have no or very little experience in writing formal requirements used in formal verification. So what is needed is a collaboration between requirements engineering researchers, industrial practitioners, and researchers in formal verification, to produce new books and courses.

Unfortunately, raising the stringency of requirements will often make it harder to write the requirements. This in turn will reduce the willingness to at all write requirements. A big part of the solution can be requirements-authoring tools with excellent support for the user. A source of inspiration could be SDV with their formal requirement language that is very intuitive to the engineers. Another part of the solution can be a tight connection to the SW architecture (with formal references to component interfaces), as we demonstrated in [17].

Another complementary direction is to rely on requirements patterns, and make them highly accessible to the user. This is more likely to succeed if an architectural framework is assumed, and even more so if standardized architectural frameworks, such as AutoSAR, are considered.

Technology Transfer (14, 15, 16, 18, 19). The key to technology transfer are successful pilot projects using the new methodologies and tools. These success stories need also to be shared across companies. When successful projects can be demonstrated, confidence in formal verification as an efficient method

will grow. This then lays the ground for more strongly recommended or even demanded formal verification in revised process standards, such as future editions of ISO26262.

5.1 Suggested Roadmap

As can be noted, many of the remedies are in control of, and can be solved within, the research community. However, several of the remedies can only be implemented with help from industry. Also, to maximize the likelihood of a near-time widespread usage of formal verification in industry, the order of the remedies should be carefully chosen. Below we provide a roadmap, with a suggested order of remedies, and also a suggested allocation of tasks to the research community and industry, respectively.

Task 1. Improvements of the formal verification tools. This is a task that the research community needs to take.

Task 2. Along with Task 1, the general requirements on the code, SW and system architecture need to be identified. This task needs to be taken by the research community, but in collaboration with industry.

Task 3. Improvement of tools and methodology for requirements engineering. Researchers from the areas of requirements engineering and formal verification need to collaborate. Also, a close collaboration with industry is needed.

Task 4. Technology transfer. When tools and methodology for requirements authoring and tools for formal verification are ready, they need to be applied in real industrial pilot projects developing new systems. Industry will need to lead this task, but researchers should be involved in coaching.

Task 5. Improvement of legacy SW and systems to meet the requirements identified in Task 2 on general requirements on the code, SW and system architecture. After having completed Task 4, industry should be ready to take on this task by itself.

6 Conclusion

Despite providing a straightforward way towards applying formal verification, the suggested roadmap requires considerable work and dedication over an extended period of time. Without realizing this, it is easy to get caught in a typical Catch 22 of formal methods: on the one hand, to truly make such methods work requires significant effort and resources (for education, development, organization, etc.), while on the other hand, industry wants to see proof that they really work before putting such an amount of effort and resources in realizing them.

It is therefore important to emphasize that a move towards a more formalized way of working, despite requiring a lot of effort, could actually be beneficial

regardless of whether formal methods end up being used or not, since it forces an organization to establish a more rigorous and structured development. At that point, then, actual usage of formal methods can be considered a bonus from this effort.

References

1. Scania tops prestigious European truck test for the second year running. http://news.cision.com/scania/r/scania-tops-prestigious-european-truck-test-for-the-second-year-running,c2460100. Accessed 22 Apr 2018
2. Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_3
3. Ali, S., Sulyman, M.: Applying model checking for verifying the functional requirements of a Scania's vehicle control system. Master's thesis, Mälardalen University (2012)
4. Bäckström, S.: Learning-based testing of automotive ECUs. Master's thesis, KTH Royal Institute of Technology, School of Computer Science and Communication (2016)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
6. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
7. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
8. Eriksson, J.: Formal requirement models for automotive embedded systems. Master's thesis, KTH Royal Institute of Technology (2016)
9. Gurov, D., Lidström, C., Nyberg, M., Westman, J.: Deductive functional verification of safety-critical embedded C-Code: an experience report. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) FMICS/AVoCS -2017. LNCS, vol. 10471, pp. 3–18. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_1
10. ISO26262: Road vehicles - functional safety. Standard ISO26262, International Organization for Standardization (2011)
11. Khosrowjerdi, H., Meinke, K., Rasmusson, A.: Learning-based testing for safety critical automotive applications. In: Bozzano, M., Papadopoulos, Y. (eds.) IMBSA 2017. LNCS, vol. 10437, pp. 197–211. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64119-5_13
12. Lidström, C.: Verification of functional requirements of embedded automotive C code. Master's thesis, KTH Royal Institute of Technology (2016)
13. Meinke, K.: Automated black-box testing of functional correctness using function approximation. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, 11–14 July 2004, pp. 143–153, Boston, Massachusetts, USA (2004)

14. Meinke, K., Sindhu, M.: LBtest: A learning-based testing tool for reactive systems. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, 2013, pp. 447–454 (2013)
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Watzenig, D., Horn, M.: Automated Driving: Safer and More Efficient Future Driving. Springer, New-York (2016). https://doi.org/10.1007/978-3-319-31895-0
17. Westman, J., Nyberg, M.: Providing tool support for specifying safety-critical systems by enforcing syntactic contract conditions. Requirements Engineering (2018)
18. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009)