Verifying Auto-generated C Code from Simulink
An Experience Report in the Automotive Domain

Philipp Berger[1(✉)], Joost-Pieter Katoen[1], Erika Ábrahám[1], Md Tawhid Bin Waez[2], and Thomas Rambow[3]

[1] RWTH Aachen University, Aachen, Germany
{berger,katoen,abraham}@cs.rwth-aachen.de
[2] Ford Motor Company, Dearborn, USA
mwaez@ford.com
[3] Ford Werke GmbH, Cologne, Germany
trambow@ford.com

Abstract. This paper presents our experience with formal verification of C code that is automatically generated from Simulink open-loop controller models. We apply the state-of-the-art commercial model checker BTC EmbeddedPlatform to two Ford R&D prototype case studies: a next-gen Driveline State Request and a next-gen E-Clutch Control. These case studies contain various features (decision logic, floating-point arithmetic, rate limiters and state-flow systems) implemented in discrete-time logic. The diverse features and the extensive use of floating-point variables make the formal code verification highly challenging. The paper reports our findings, identifies shortcomings and strengths of formal verification when adopted in an automotive setting. We also provide recommendations to tool developers and requirement engineers so as to integrate formal code verification into the automotive mass product development.

1 Introduction

The Need for Formal Verification in Automotive. In the automotive industry an increasing number of features are implemented in software. As a result the complexity and dependence on produced artefacts is on the rise. Additionally, customers demand more flexibility in selecting features leading to an ever increasing number of feature flags and build configurations. The automotive functional safety standard ISO 26262 defines a ASIL (Automotive Safety Integrity Level) classification scheme and recommends appropriate verification techniques for each ASIL level such as testing and *formal verification*. Testing focuses on showing the presence of bugs, whereas the rigorous state exploration provided by formal verification aims to show the absence of bugs. Although testing nowadays is commonplace, the application of formal verification to software artefacts in the industry is in its infancy.

K. Havelund et al. (Eds.): FM 2018, LNCS 10951, pp. 312–328, 2018.
https://doi.org/10.1007/978-3-319-95582-7_18

Verifying Simulink Models. This paper considers the use of formal verification in a model-based system development process in the automotive domain. We concentrate on Simulink, a popular model-based software development tool that is widely used in the automotive industry. Simulink is developed by MathWorks and provides a graphical environment for modeling, simulating and analyzing dynamical systems. Formal verification of Simulink so far has primarily concentrated on the model level verification, which has been done with the first generation of model checkers, i.e., verification tools that focus on verifying models of real artefacts. Experiences using the commercial Simulink Design Verifier (SLDV) [1] as well as using model checkers such as NuSMV, SPIN and Uppaal [2–4] have been reported.

Verifying C Code. In contrast, this paper considers the *formal verification of C code* that is automatically generated from Simulink models. More precisely, we aim at checking whether the requirements imposed on Simulink models are satisfied by the C code that is obtained by push-button technology from these models. Formal verification of auto-generated code is of interest as automotive companies such as Ford Motor Company have been deploying more and more auto-generated code to reduce development time and lower the risk of introducing errors by manual coding. The auto-generated C code may differ from the behavior of the Simulink model due to the lack of formal semantics, or potential bugs in the translation procedure. Program code verification is supported by second-generation model checkers such as CBMC [5], Ultimate Automizer [6], and CPA Checker [7], to mention a few. As our aim is to integrate formal code verification into Ford's mass product development, we focus on a *commercial verification tool for code verification*. We selected the BTC EmbeddedPlatform[1] (BTC, for short) as developed by BTC Embedded Systems AG. This tool includes amongst others CBMC. It has been developed for industrial use and a dedicated support team is available.

Approach. This paper reports on our findings by applying BTC on two R&D prototype case studies: a next-gen *Driveline State Request* and a next-gen *E-Clutch Control*. Their Simulink models consist of a few thousand blocks, and their C code is about 2,000 and 5,000 lines of code, respectively. The formal verification of industrial-scale open-loop[2] controller models is challenging especially due to the diverse feature-set and the extensive use of floating-point variables, see also [8]. We checked these models against 42 and 70 requirements, respectively, which were made available to us in textual form as Microsoft Word documents.

Our Findings. The formal verification—including the formalization of the requirements and running the verification tool—was carried out by researchers having knowledge in model checking. Issues were found in 43% of the 112 requirements. 35 requirements were either ambiguous, incomplete or inconsistent, nine

[1] https://www.btc-es.de/en/products/btc-embeddedplatform/.
[2] Open-loop means that the model does not include the controlled environment.

requirements could not be taken into account due to restrictions of the verifier while four requirements were missing details about the exact algorithm to be implemented. The formal verification revealed 20 code implementations that are inconsistent with the requirements of the prototype features. These errors could all be traced back to the Simulink models. All detected issues were communicated with Ford Motor Company and subsequently resolved. For 29 requirements, only a bounded proof of correctness could be derived. These findings stress the importance of formal verification for automotive software.

Our Recommendations. This paper reports on our findings, identifies shortcomings and strengths of formal verification when adopted in an automotive setting. Our focus points were automation, scalability and usability, that are necessary for an integration into a large-scale automotive development process. Though the verification of both models was successful, we encountered different technical challenges with respect to requirement formalization, tool usage and model structures. Integrating formal verification into mass automotive product development is not easy: engineers are not familiar with formal methods, and are not trained in writing formal requirements. We provide a detailed set of recommendations to ease the requirement formalization, most notably by using specification patterns and the use of pre-defined requirement blocks. We also present some ideas for new features in verification tools that can further support the integration of formal verification into C code targeted development in the automotive sector: The mitigation of spurious counterexamples, enabling batch processing, and support for automated "continuous" verification, i.e., when parts of the Simulink model, the requirement or the tools change.

Main Contributions. To summarize, our main contributions are:

- a detailed report on experiences with using a modern commercial verification tool to formally verify C code automatically generated from Simulink models,
- a detailed set of recommendations for engineers in the automotive domain to enable the use of formal code verification in the design process, and
- a detailed list of recommendations for verification tool builders to integrate their tools into the automotive mass product development.

Organization of the Paper. Section 2 introduces the two R&D prototype case studies. Section 3 briefly introduces the BTC verification tool and its features. Section 4 presents our findings concerning requirement formalization and formal code verification. Section 5 presents our observations and recommendations for enabling and integrating formal verification into the automotive development process. Section 6 concludes the paper.

2 The Case Studies

The aim of this joint project between Ford Motor Company and RWTH Aachen University is the feasibility analysis of discrete-time verification of industrial-scale C code controllers for mass production. The aim includes assessing the

current state of requirement specification within Ford, check the quality of generated C code from Simulink models by a commercial verification tool and identify possible solutions for a highly-automated verification tool-chain that can be integrated into an automotive development process. This section presents the two R&D prototype case studies from Ford. Due to confidentiality reasons, we cannot provide access to model files, source code or requirements.

Table 1. The variables and calibration parameters of the Simulink models.

		Scalar			Array/Matrix		
		bool	int	float	bool	int	float
DSR	I/O Vars	7	16	8	0	1 (1×10)	1 (1×10)
	Calibration Pars	15	3	24	1 (1×12)	1 (1×32)	9 (1×6 . . . 12)
ECC	I/O Vars	3	4	29	0	0	0
	Calibration Pars	7	1	71	0	0	72 (1×2 . . . 11×11)

R&D Prototype Features. Our case study was conducted using the auto-generated code of two R&D prototype Simulink models: a model of the next-gen *Driveline State Request* (DSR) feature and of the next-gen *E-Clutch Control* (ECC) feature. None of these features are safety-critical. Let us describe the role and importance of these two features. Energy saving, exhaust emission reduction, and exhaust noise reduction are among the main objectives in the development of modern vehicles. On flat roads, the engine drag torque typically slows down the vehicle when the driver takes his foot from the accelerator pedal. A vehicle with a combustion engine can be operated in Sailing Mode when the vehicle is rolling without engine drag torque in order to maintain its speed and to save fuel in certain driving situations and where the driver is not actively accelerating or braking the vehicle. In Sailing Mode, the driveline[3] is opened automatically when the driver releases the accelerator pedal. When the driveline is open, the engine can be shut off or run at idle speed without introducing drag torque in order to reduce fuel consumption. Different system- or driver-interactions, such as pressing the brake or accelerator pedal, will result in closing the driveline again. The DSR feature implements the decision logic to open or close the driveline. This feature takes driver interactions and vehicle status information to decide in which situations the driveline should be opened and closed again. The ECC feature calculates the desired clutch torque capacity and corresponding engine control torques or speeds for opening and closing the driveline.

Model Characteristics. The aim of this study is to investigate how C code formal verification performs when applied to the auto-generated code of Simulink models having decision logic, state-charts, filters, rate limiters, look-up tables and

[3] A motor vehicle's driveline consists of the parts of the powertrain excluding the engine. It is the portion of a vehicle, after the prime mover, that changes depending on whether a vehicle is front-wheel, rear-wheel, or four-wheel drive.

feedback control. Therefore, we selected two features that contain mixtures of these different kinds of functionalities. The Simulink model for the DSR feature has 42 functional requirements, 28 inputs, six outputs, 53 calibration parameters and 1149 blocks. Calibratable parameters remain constant during software execution but can be adjusted before the execution for tuning or selecting the possible functionalities. Calibration is the adjustment of calibratable parameters of software functions realizing the control functionalities. This model contains several variables as summarized in the first two rows of Table 1. The C code comprises about 2100 lines. The next-gen DSR feature has several calibration parameters, amongst others 42 scalar parameters and nine array parameters of six to 12 single-precision floating-point elements, one Boolean array with 12 elements and one unsigned 8-bit integer array with 32 elements.

The Simulink model for the next-gen ECC feature has 70 functional requirements, 27 inputs, nine outputs, 151 calibration parameters and 2098 blocks. In total, it comprises about 4900 lines of C code. This model contains Boolean, integer and a huge number of floating-point variables, see Table 1. The calibration parameters include 79 scalar parameters and 13 arrays of single-precision floats.

Requirement Characteristics. For the DSR case study, from 42 functional requirements we extracted 54 properties, consisting of 50 invariants and four bounded-response properties. Invariant properties are assertions that are supposed to hold for all reachable states. Bounded-response properties request that a certain assertion holds within a given number of computational steps whenever a given, second assertion holds. For the ECC case study, from 70 functional requirements we extracted 82 invariants and two bounded-response properties.

3 The BTC Tool

We exploited BTC, a commercial tool for formal specification and verification.

3.1 The BTC EmbeddedPlatform

BTC is an integrated development environment featuring requirements-based testing, back-to-back testing and a formal verification suite with an integrated graphical user interface aiding the formal specification. The user interface aims to support industry software engineers without much knowledge in formal methods. An overview of the formal verification portion is illustrated in Fig. 1.

The initial architectural set-up of a new BTC project requires in-depth knowledge of the C code to be verified, its structure and variables.

A BTC Project Setup. Our imported C code defines the architecture of a project, including available functions, variables, entry points and initialization routines. Auto-generated C code for automotive controllers usually consists of an *initialization* and a *step* routine. After selecting a (sub-)set of available functions to

instrument, global and local variables are collected and categorized as "input", "output", "local", "parameter" or "ignore" through manual user input and simple heuristics. The range of input variables and parameters can be restricted by specifying minimal and maximal values. The execution time per step, i.e., the amount of time that passes between two consecutive executions of the main step method, can be configured as a constant when creating the project. The user interface is divided into views, each focusing on different aspects of testing and formal verification. Here, we only consider the latter view.

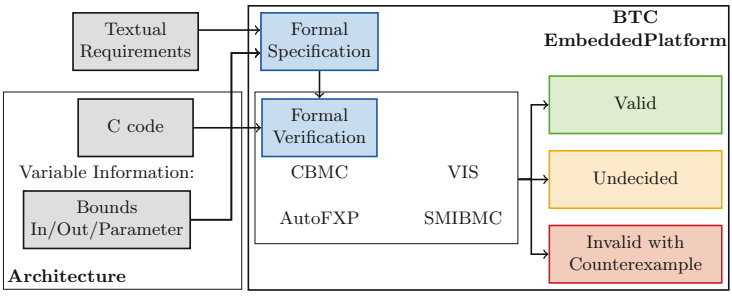

Fig. 1. Overview of the ins and outs of the BTC.

Formal Specification. The requirements of the two case studies are textual, i.e., use natural language (English) for describing functional feature behavior. BTC supports importing textual requirements from various sources, such as Microsoft Excel, allowing engineers to develop the formal (requirement) specification directly on the requirement text.

Pattern-Based Specification. Whereas most model checkers use some form of temporal logic for the property specification, BTC supports safety as well as liveness properties through *pattern-based specification* [9]. These patterns are a kind of template and are based on an intuitive and graphical representation of the formal semantics. The patterns enable the usage of BTC without being an expert in temporal logic. In its most basic form, a BTC pattern-based specification represents an invariant together with start-up delays, or models Trigger/Action conditions with timing information. The BTC user goes through the textual requirement (template), selects and maps parts of the text to macros, place-holders for logical formulas. Take e.g., the expression "When *in reverse*, [...]", containing a high-level description of an external input state condition, namely the gear selection. By creating a macro mapping the term *in reverse* to a formal definition in terms of variables and values like "`selectedGear` $== -1$", the textual requirement is enriched with the information necessary for formalization. Macros can be shared between multiple requirements in the same project/architecture and may contain other macros, but do not support parameters.

Operators. The specification language used for defining triggers, actions and invariant conditions contains all basic boolean operators (NOT, logical OR and AND, $\Rightarrow$, $=$, $\neq$), bitwise operators ($\gg$, &, |, $\oplus$) as well as basic math operators ($+$, $-$, $\cdot$, $/$, $<$ and $>$). Additionally, floating-point operators (`fabs`, `feq`, `min`, `max`) and temporal operators for step-based timing information are available.

Verification. After formalizing the requirements, the *Formal Verification* view allows for creating a *proof* for the requirements. For model-checking purposes, BTC uses four back-end verification engines: CBMC [5], SMIBMC, VIS [10] and AUTOFXP. These can be turned on and off—the default, used by us, is to use all four. We treat the verification engine and its back-ends as black box as no further information is available. A time limit, a maximum search depth, the number of loop unrolling iterations and a memory limit can be set. Currently, batch execution of proofs is only possible using a non-standard plugin which is available upon request from BTC. This plugin executes all proofs; selecting a subset is not possible. If a counterexample is found, the analysis terminates and the counterexample is automatically simulated on the C code to account for errors between the internal data representation and the C semantics.

4 Experience Report

This section reports our experiences with the formal verification (using BTC) of the two R&D prototype case studies focusing on verifying the auto-generated C code of a few thousand lines from Simulink models.

4.1 Formalizing the Requirements

For a formal verification, formally specified requirements are indispensable. At the start of this work, the requirements were available in the form of textual phrases formulated in natural language. This section reports the issues that were encountered when formalizing these requirements for subsequent use by the C code verification by BTC. These issues range from incomprehensibility of some of the requirements to incomplete requirements, ambiguities and inconsistencies.

Requirement Formalization. We incrementally formalized the Ford requirements given to us in textual form and turned them into BTC formal requirements. Unclarities or other issues were discussed with the Ford engineers. Small issues involving missing domain knowledge were often solved by looking at the corresponding code implementation. *The process of requirement formalization took most of the total project time of about 900 man hours and involved frequent meetings between the researchers conducting the verification and the Ford engineers.*

While reviewing and formalizing the 112 requirements, we identified 35 issues that can be categorized into ambiguous wording, inconsistencies and underspecification. Without a background in automotive engineering, the learning curve for

formalization, especially regarding domain knowledge, was steep, further indicating issues of underspecification.

Each formalized requirement resulted in one or more *Formal Requirements* for BTC. If a requirement contains multiple cases or can be easily split into distinct logical blocks, we did so. Splitting eases verification by the model checker. *Nine requirements could not be formalized* due to the fact that *BTC does not support the use of array access with variables as index nor the use of lookup tables with pointers in the formal specification. Four requirements have not been considered as they use underspecified complex operators* such as rate limiters and filters.

Table 2. Identified requirement issues.

Case study	Incomplete	Ambiguous	Inconsistent
DSR	12	5	1
ECC	11	3	3

If `varA` is set to [TRUE] and `varB` is above a threshold with hysteresis (`calVarBThres`, `calVarBHyst`), OR (`varA` is set to false and `varC` is above a threshold with hysteresis (`calVarCThres`, `calVarCHyst`) AND (`varC` - `varD`) is above a threshold with hysteresis (`calVarCThres`, `calVarCHyst`)), then `varE` shall be set to true, and to false otherwise.

Fig. 2. Example of an incomplete requirement involving hysteresis.

Incomplete Requirements. We encountered 23 *incomplete requirements* (Table 2). Typical examples of incompleteness are preconditions like "when no preprocessing feature is active" where no information on signals related to the status of preprocessing features is given, and declarations like "variable A shall be input B with hysteresis (lower threshold calibratable C and upper threshold calibratable D)"; see also Fig. 2. Hysteresis is often used to prevent rapid toggling when observing an input signal against some threshold and can be implemented using a set-reset flip-flop.

Other encountered issues are whether the thresholds are strict (e.g., <10) or non-strict (e.g., ≤ 10), and what the initial output state should be. The analyzed requirements include several such abstract high-level descriptions of functionality. In these complex cases, e.g., when using hysteresis, saturation or rate limiting, the requirements are often lacking necessary information for accurately specifying function behavior. Several of these issues could not be resolved without consulting the code. Typical omitted information in the requirements includes the initial configuration, exact (formal) state change conditions including a priority ordering of the signals, and complete documentation of state variables.

For the successful verification of a stateful system, access to all state variables is required. These are usually not visible in the global interface as they are unnecessary for using and embedding the system. Therefore, *we manually adapted the Simulink model by adding necessary variables to the global interfaces*.

Ambiguous Requirements. We identified *eight cases of ambiguity in the prototype features, mostly related to cases of missing parentheses*. We identified cases where in chains of AND- and OR-conditions either applying the mathematical operator precedence did not make sense when considering the condition content or where there were an uneven number of opening and closing parentheses present.

Inconsistencies. Another group of issues focuses on inconsistencies between requirements. We found *four issues in the prototype features where a commonality shared between several requirements is invalidated by another one*. Take, e.g., n requirements of the form "When in state X, do..." among which a single requirement specifying similar behavior, is missing the "When in state X" scope. It is unclear whether this omission is intentional or not. While this inconsistency is not a problem of the formalization per se, these discrepancies are often closely related to incompleteness issues in requirements, e.g., when "boilerplate" information is omitted because preconditions like being in a specific state are made implicit by a chapter heading or a requirement name. While implicit preconditions may be acceptable for textual requirements, it hampers formal verification.

Table 3. Verification results on the case studies using BTC.

Case study	Calibration type	Valid	Unknown	Invalid
DSR	Fixed	24 (Bounded: 7)	8	10
	Varying	23 (Bounded: 8)	12	7
ECC	Fixed	44 (Bounded: 22)	16	10
	Varying	36 (Bounded: 24)	20	14

4.2 Formal Verification of Auto-generated Code

We now report our findings when applying BTC to the C code of the two R&D prototype features. All verification experiments were carried out on a Intel Core i7-6700HQ machine with 16 GB RAM, running Windows 7 (64-bit), BTC v2.0.3[4] and Matlab R2015b. The maximal verification time was set to 7200 s.

Verification Results. Each formalized requirement was formally verified on the C code of the entire respective feature with fixed calibration parameters. To ease the verification, we fixed the time bounds in the Ford requirements to five simulation steps (50 ms). Typical time bounds in the model vary between 50 ms and 5 s.

[4] The most recent version of BTC as of submission is v2.1.0.

For three requirements involving direct lookup tables (i.e., no interpolation)—not natively supported by BTC—we embedded their data directly into the formal specification. Table 3 summarizes our verification results. We consider the fixed calibration parameters unless stated otherwise; we will discuss the varying calibration parameters later on. In our experience, model checking is performed by BTC in three phases:

1. CBMC runs for a number of iterations, in many cases returning counterexamples (if any) in a matter of seconds.
2. If CBMC is not able to refute the property, a combination of AUTOFXP and SMIBMC is used.
3. If the combination is not successful, i.e., results in an unbounded proof, the tool switches back to CBMC, providing a bounded result.

Most of our results seem to be mostly a result of CBMC as we did not observe termination during the AutoFXP/SMIBMC phase. Runtime data like memory consumption or CPU time is only shown while the tools are actively running and can unfortunately not be obtained once model checking has terminated.

Implementation Flaws. In total, BTC found 20 *cases of invalid code implementations* against the formal requirement specification with fixed calibration parameters of the R&D prototype features. These include:

- Four instances of *incorrect relational operators* (e.g., $<$ instead of $\leq$).
- Four instances of *incorrect variables* used in comparisons. As inputs are processed, for many there are secondary variables available containing derived versions of the input, for example with rate-limiting applied. When comparing inputs, it is important to select the right variant.
- 12 cases where *variables were named differently* than in the specification.
- One instance where the implementation contained a fix for a logical error in the specification which was not passed back and reflected in the requirements.
- One instance where an *output signal was unexpectedly delayed* by one time step even though the delay was not apparent from the requirement specification.
- Two instances of *incorrect use of negations*.
- Various instances of the *incorrect use of chained `if`-statements*. When a requirement contains several consecutive `if`-statements followed by a final `else`-statement, the activation condition for the `else` section is comprised of the negation of the disjunction over all preceding `if`-conditions. When changing the `else if` blocks, correcting all dependents is easily forgotten.
- One instance where *an initialization step was not implemented*. This omission leads to minor initial differences.
- Two instances where *part of a comparison was omitted*.
- An *offset* mentioned in a requirement that was *not used* in the code.
- One case where an *activation/deactivation was unintentionally implemented* using a state toggle such that no signal has precedence over the other.

All detected issues are also present in the respective Simulink models and thus not the result of incorrect C code generation.

Undecided. A total of 24 requirements of the R&D prototype features are undecided, i.e., we were unable to determine whether the implementation and specification match. More precisely:

- Nine requirements contain *properties unsupported* (lookup tables containing pointers, array access with variables as index) by BTC.
- For three requirements we encountered *spurious counterexamples* which were automatically detected as spurious by BTC. Whenever a counterexample for a property is detected, there is a chance that this counterexample exists only due to imprecision introduced by abstraction, for example when approximating floating-point numbers. Thus, each counterexample is simulated on the C code to ensure it is a valid counterexample under the C semantics. Currently, spurious counterexamples prevent further analysis with this combination of formal specification and C code in BTC.
- For two requirements, the tool reported that the verification *unexpectedly terminated*. We are working with BTC to fix these issues.
- For six requirements, structural properties of the implementation prevent formal verification as *necessary outputs are overridden before they can be captured*. Note that in these cases while formalization was seemingly successful, we discovered during the analysis of counterexamples that because of how outputs are stored or combined formalization does not work.
- Four requirements were not formalized, mainly due to the mixture of underspecification and the use of complex operators (low-pass filter, first-order lead filter, second-order notch filter and rate-limiting) made the *formalization process too time-consuming*.

Fixed and Varying Calibration Parameters. We also analyzed the features with *varying calibrations*. Where model checking with fixed calibrations only checks conformance to the specification in one specific calibration setting, with varying calibrations conformance is checked for all possible calibration valuations allowed by the configured calibration bounds. With 53 calibration parameters in the DSR feature and 151 in the ECC feature, we expected the number of undecided results to go up significantly, but out of 42 requirements (DSR) and 70 requirements (ECC) *only the result of five (DSR) and nine (ECC) requirements changed*. We are unable to fully explain the overall lack of impact. We did notice a significant change of depth reached in bounded model checking, with differences being as high as depth 151 in fixed calibration versus a depth of 8 in varying calibration within the same time bound. With DSR, for one requirement the result changed from unbounded satisfied to bounded satisfied, two requirements changed to undecided because of a detected spurious counterexample and two more resulted in time-outs during preprocessing (also now undecided). With ECC, for five requirements the result changed from valid to invalid, three requirements changed from valid to undecided, one requirement changed from invalid to undecided and two requirements changed from unbounded satisfied to bounded satisfied. We found *no cases where a requirement was inconsistent with its implementation while analyzing with fixed calibrations but satisfied with varying calibrations*. During this analysis we uncovered

several issues in the implementation that were not visible during analysis with fixed calibrations.

Bounded and Unbounded Results. BTC was not able to derive an unbounded proof of correctness for a total of 29 requirements. Whereas a bounded proof for depth n only guarantees a safety property to hold for the first n steps, an unbounded proof (if successful) proves such a property for any depth. In the bounded case [11], correctness is up to depth n and no guarantees are given for depths beyond n. An unbounded proof of correctness usually includes deriving loop invariants for all included loops in the C code, a hard problem that is undecidable in general. Heuristics for generating these loop invariants often only work on small and simple (e.g., linear) loops with few variables and even fewer floating-point operations [12].

Subsystem Verification. With subsystem verification the goal is to reduce model checking complexity by reducing C code size. When a formalized property is handled entirely within some distinct part of the C code, all surrounding and unrelated code can be removed, replacing the original interface with one only containing inputs, parameters and outputs relevant to the selected property. We picked a set of ten requirements from the DSR case study specifying the behavior of a stateflow chart to compare formal verification on controller level with that on subsystem level. We extracted the subsystem from the Simulink model by hand and generated code from the reduced model. While BTC supports subsystem verification using a hierarchical architecture representation, the auto-generated C code in our use case was not in suitable form, making manual preparation necessary. Using the subsystem significantly shrunk the overall complexity of the analyzed model ($\approx$ 350 lines of code) and reduced the number of variables and calibration parameters. The reduced system has six inputs (four Boolean, one (1×16) array of Boolean, one unsigned 8-bit integer), two outputs and one parameter (one (1×32) array of unsigned 8-bit integer). However, the input variable domains might grow since values are no longer restricted by other upstream parts of the model. In the full feature, the inputs of this subsystem are derived from external inputs and may be between tight bounds that are not apparent nor documented explicitly because of implemented behavior. Therefore, subsystem verification over-approximates correctness with respect to the correctness within the entire system. A specific input valuation leading to a property violation might be caught and avoided beforehand in the upstream components of the entire system, but this input restriction is not apparent anymore in the subsystem. While BTC was not able to produce unbounded correctness proofs for the ten requirements when verifying the code of the complete feature, using subsystem verification it was able to do so.

Invalid Verification Results. While investigating issues with `NaN` (not a number) values on some floating-point variables, we discovered conflicting results for properties with comparisons involving floating-point variables that can become

`NaN` where an incorrect simplification step potentially leads to invalid results. The issue has been fixed in a new version of BTC.

5 Reflections and Recommendations

This section reflects on our findings and presents our recommendations towards requirement engineers, verification tool developers as well as tool users.

5.1 Requirements

Requirement Completeness. A big hurdle during this work was domain knowledge implicitly required to understand the textual requirements of Ford. This domain knowledge ranges from simple things such as unknown abbreviations to structural details like how the data flow influences delays on certain variables (breaking circular dependencies requires using one-step delays). Certain information is ubiquitous in the Ford development process and not re-iterating every detail makes requirements short and concise. For the same reason, some requirements do not state their full preconditions but instead rely on their positioning inside the requirement catalogue, e.g., in a chapter containing all requirements related to being in a certain state, the subsidiary requirements do not state explicitly that being in that state is a precondition. Four of the requirement violations were due to unexpected delays introduced to break circular dependencies in the model. In practice, certain bounded and small deviations like one- or two-step delays are acceptable while basic behavior is correctly implemented, it highlights *the conflict between the ease-of-use of textual requirements and the degree of precision which is required for formal verification*.

Requirement Interdependencies and Priorities. We found several instances where two requirements could apply simultaneously to the same output variable. In these cases, a priority chain needs to be in place for *defining precedence*. It is also advisable to reorder requirements such that *conditions are grouped per output variable* so that analyzing a single requirement should be sufficient to determine all pre- and postconditions for any given output variable.

Environmental Assumptions. While performing verification with varying calibrations, we encountered situations where the upper bound of a hysteresis block was calibrated below its lower bound. *Calibration parameters should therefore be clearly documented including side conditions and interdependencies*, such that appropriate assertions can be added for verification.

Specification Patterns. For effectively applying formal verification, we recommend engineers switching from textual requirement specifications to an approach that guarantees unambiguous requirements like a *pattern-based approach* [13]. These use specification patterns—templates phrased in natural language

with holes that need to be completed by the verification engineers with conditions on code variables. Tools tailored to the knowledge of the engineers should be provided in order to help and enforce writing clear, unambiguous and complete specifications in a format that is agnostic of verification tools and can be automatically transformed for any chosen tool. Specification patterns have been used in automotive [14], aerospace [15] and service-oriented computing [16].

Tool Support and Automation. We believe that while enforcing completeness, verification tools need to provide the engineers with the *ability of building a library of more complex pre-defined formal requirement specification blocks*. Specification blocks enable more consistent requirements presented at a high level of abstraction while still supporting a formal semantics. *We found several instances of complex behavior requiring large and equally complex formalizations that are ideal for attracting small mistakes during formalization*.

5.2 Code Verification

Code Complexity and Subsystem Verification. We found that BTC works *extremely well on invariant properties with no or very few floating-point variables involved*. Counterexamples are typically found within seconds and even unbounded proofs are mostly found in less than a minute. Floating-point numbers are ubiquitous in the automotive domain and pose a challenge to most, if not all, verification tools, see also a discussion on the results in the latest software verification competition [8]. While BTC scales well with the number of parameters, we found that *subsystem verification is a necessity for tackling more complex properties*. In a proof-of-concept of subsystem verification, we achieved unbounded (rather than bounded) verification results for all requirements.

Requirement Robustness. While experimenting with scalability we discovered that even small changes to the requirements can have great impact on the verification times. Take, e.g., a requirement describing that event A implies that event B occurs exactly one time step later. This requirement can be modeled using a Trigger/Action (response) pattern or using an Invariant pattern. While these specifications are semantically equivalent, the corresponding verification times are not. In some instances, the Invariant approach took seconds to verify whereas the Trigger/Response variant took several hours. We recommend tool vendors to adapt *internal optimization routines* such that, by default, the simplest requirement representation is used.

Counterexample Verification. During verification, we encountered several instances where BTC concluded after internal simulations that a discovered counterexample was in fact spurious, i.e., the counterexample was introduced due to used approximations. Verification tools should *automatically mitigate spurious counterexamples* whenever possible.

Verification Times. In our experiments, we noticed that generally an unbounded result is obtained within the first minute or no such result is obtained at all. Outliers are obtained for *temporal* properties that usually include a timer that has to expire. We encoded these properties using a Trigger/Action pattern where the delay between the two is given by a calibration parameter. As BTC does not offer *access to verification timing data*, a more detailed analysis of verification times is not possible. In academia, CPU time and memory consumption are common practice [8] and *the* means to compare verification algorithms and tools. This comparison is useful for industrial applications too as it enables comparing different verification engines (even within a single tool) and provides a means to study scalability.

Tool Automation. Repetitive verification tasks are prone to human errors. Hence we envision a fully automated *continuous integration pipeline for mass product development where model checking is performed whenever the specification, the code or the tools change*. Similar approaches—referred to as continuous verification—have been advocated for adaptive software [17]. Therefore, verification tools should support batch processing of verification tasks, provide a better automation interface and a structured output of all relevant result data. A new version of BTC released after conducting our study comes with an automation API and Jenkins integration, enabling continuous verification.

Bug Reporting. Because source code, models and requirement documents usually are confidential, reporting bugs and spurious counterexamples is difficult and time consuming, usually involving an engineer shrinking and anonymizing the code by hand. Automated tools for this purpose such as CReduce [18] require an adequate automation interface. We recommend to include the *automated generation of minimal anonymized examples of bug triggers* in verification tools.

Counterexample Representation. Currently counterexamples can be either viewed as charts showing variable values in each time step or by exporting a stimulation vector containing the generated sequence of input values as a Microsoft Visual Studio C code project. It would be very helpful to *see what part of a requirement is inconsistent with its implementation*, especially when dealing with large conjunctions. Going through a huge number of variables, comparing them to other variables or constants and evaluating the boolean expressions one by one on a sheet of paper is very time consuming and error prone. A graphical tree representation of the formal specification could be beneficial.

6 Conclusion

In this paper, we reported our experiences and presented our recommendations concerning applying formal verification with BTC on two case studies provided

by Ford Motor Company. We performed formal verification of 7000 lines of code generated from two Simulink models implementing 112 textual requirements. We identified 35 requirements of the R&D prototype features which are either ambiguous, incomplete or inconsistent; nine cannot be verified due to restrictions of the verifier; while four could not be formalized. Formal verification revealed 20 code implementations that were inconsistent with the requirements.

We spent more than 70% of the project time on requirement analysis and formalization. The overhead should be much larger if performed by automotive engineers whom do not have experience in formal methods. In the automotive industry, both the software-requirements and their implementations change rapidly within strict deadlines. Natural languages, hence, are preferred over formal notations to write requirements in the practice although natural languages often lead to ambiguity, incompleteness and inconsistency. Moreover, we experienced that not all open-loop requirements can be formalized and are supported by the formal verification tools. We also observed spurious counterexamples and unexpected terminations. Recently formal verification tools matured a lot but have yet to provide unbounded and decisive results for most industrial cases.

Our case studies show the benefits that formal verification can add into the automotive development process. We, therefore, believe the use of formal verification will increase slowly but surely in this domain. For this progress, we recommend to develop a technique—such as automated conversion of textual requirements into formal requirements—that will allow engineers to use formal specifications for a fast pace industry without introducing much overhead. Tool vendors need to increase the percentages of unbounded and conclusive results. Additionally, to enable an easier integration of verification tools into the (automotive) industrial design process, we also recommend improving the usability of these tools such as automated mitigation of spurious counterexamples and better diagnostic feedback for the refuted properties.

Acknowledgments. We thank BTC Embedded Systems AG for their continuing support and helpful advice. We are grateful to Johanna Nellen, William Milam and Cem Mengi for fruitful discussions on formal verification and Simulink.

References

1. Nellen, J., Rambow, T., Waez, M.T.B., Ábrahám, E., Katoen, J.P.: Formal verification of automotive Simulink controller models: empirical technical challenges, evaluation and recommendations. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 382–398. Springer, Cham (2018)
2. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_33
3. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool chain to support automated formal verification of avionics simulink designs. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 78–92. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32469-7_6

4. Filipovikj, P., Mahmud, N., Marinescu, R., Seceleanu, C., Ljungkrantz, O., Lönn, H.: Simulink to UPPAAL statistical model checker: analyzing automotive industrial systems. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 748–756. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_46
5. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
6. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
7. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
8. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
9. Bienmüller, T., Teige, T., Eggers, A., Stasch, M.: Modeling requirements for quantitative consistency analysis and automatic test case generation
10. Brayton, R.K., et al.: VIS: a system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_95
11. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
12. Bagnara, R., Mesnard, F., Pescetti, A., Zaffanella, E.: The automatic synthesis of linear ranking functions: the complete unabridged version. CoRR abs/1004.0944 (2010)
13. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Trans. Softw. Eng. **41**(7), 620–638 (2015)
14. Filipovikj, P., Nyberg, M., Rodríguez-Navas, G.: Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In: RE, pp. 444–450. IEEE Computer Society (2014)
15. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2011)
16. Bianculli, D., Ghezzi, C., Pautasso, C., Senti, P.: Specification patterns from research to industry: a case study in service-based applications. In: Software Engineering. LNI, vol. 227, pp. 51–52. GI (2014)
17. Calinescu, R., Ghezzi, C., Kwiatkowska, M.Z., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM **55**(9), 69–77 (2012)
18. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI, pp. 335–346. ACM (2012)