Formal Verification of Automotive Simulink Controller Models: Empirical Technical Challenges, Evaluation and Recommendations

Johanna Nellen[1], Thomas Rambow[2], Md Tawhid Bin Waez[2], Erika Ábrahám[1], and Joost-Pieter Katoen[1]

[1] RWTH Aachen University, Germany
johanna.nellen|abraham|katoen@cs.rwth-aachen.de

[2] Ford Motor Company, Germany|USA
trambow|mwaez@ford.com

Abstract. The automotive industry makes increasing usage of Simulink-based software development. Typically, automotive Simulink designs are analyzed using non-formal test methods, which do not guarantee the absence of errors. In contrast, formal verification techniques aim at providing formal guarantees or counterexamples that the analyzed designs fulfill their requirements for all possible inputs and parameters. Therefore, the automotive safety standard ISO 26262 recommends the usage of formal methods in safety-critical software development.
In this paper, we report on the application of formal verification to check discrete-time properties of a Simulink model for a park assistant R&D prototype feature using the commercial *Simulink Design Verifier* tool. During our evaluation, we experienced a gap between the offered functionalities and typical industrial needs, which hindered the successful application of this tool in the context of model-based development. We discuss these issues and propose solutions related to system development, requirements specification and verification tools, in order to prepare the ground for the effective integration of computer-assisted formal verification in automotive Simulink-based development.

1 Introduction

In modern cars a huge number of embedded software components support the vehicle control. These software components are usually developed in a model-based approach with a graphical modeling language like Simulink/Stateflow that allows automatic code generation for the deployment of the controller. Though the safe operation of a software component is rigorously tested in offline simulations, the absence of errors cannot be guaranteed by testing.

The automotive safety standard ISO 26262 recommends to integrate — besides other approaches — also *formal verification* in the development process of safety-critical software. Formal verification either guarantees that the property

holds for *all* possible input and parameter combinations, or it provides a counterexample (i.e., a system run that violates the property) which can be used to identify the error and to re-design the controller.

In this paper we present an industrial case study from the automotive sector with the aim to *empirically identify and solve technical problems that might arise during the integration of discrete-time formal verification in Simulink-based mass production* of safety-critical systems by engineers who are not formal methods experts. These problems might not be obvious or seem pressing but they become prominent and relevant for large-scale development, a development process with much legacy, or a development without a strong dedicated formal methods team.

We decided to rely on the commercial verification tool *Simulink Design Verifier* (*SLDV*) [1], which is developed by the vendors of Simulink and backed by a dedicated support team. We applied SLDV to analyze a Simulink controller model for a park assistant R&D prototype feature against 41 functional requirements, which were given informally in textual form as a Microsoft Word document. The model is open-loop, as the controlled environment is not included, and contains no continuous-time blocks, such that we could use SLDV's discrete-time verification functionalities. Using formal verification we detected inconsistencies between requirements and their implementations in our model, which demonstrates the importance of formal verification for safety-critical software components.

Besides the verification results, we report on the strengths of SLDV, identify its limitations and collect important general observations.

Though the verification of our model was successful, we encountered different technical challenges. Introducing formal verification into fast pace mass automotive product development by engineers who are not familiar with formal methods is not at all straightforward and needs a high level of automation. We give recommendations to support requirement engineers to build complete, unambiguous and consistent requirements and to help system engineers to develop "verification-friendlier" models. We also give some ideas for new features in verification tools that can support the integration of formal verification into Simulink-based development in the automotive sector.

Related work. Techniques and tools for formal discrete-time verification of Simulink models has been widely studied. Regarding applications, a medical case study using SLDV is presented in [2]. In [3] the authors apply the SMT-based static verification tool *VerSAA* to a Simulink model and also provide a comparison to SLDV. In [4] an SMT-based approach for explicit LTL model checking of Simulink models is presented. A tool chain for the formal verification of Simulink models in the avionics industry using the LTL model checker *DiVinE* and a proprietary verification tool *HiLiTE* is presented in [5, 6].

Some other approaches transform Simulink models into the input modeling language of different verification tools. The authors of [7] transformed Simulink models into the modeling language *Boogie* and compare the performance of the Boogie verification framework with SLDV on an automotive case study. The work [8] translates Simulink to *UCLID* and applies SMT-based bounded model

checking to an automotive case study. The works [9, 10] and the Simulink toolbox *cocoSim* [11] offer translations of Simulink resp. SCADE models to the intermediate language Lustre in order to enable the application of different verification tools. Additionally, [9, 10] report on the experiences with the integration of formal verification in an avionics model-based development process. The *RCRS* project [12, 13] formalizes Simulink models in Isabelle and uses the Isabelle theorem prover for formal analysis. SLDV and UPPAAL were used for the formal verification of an automotive case study in [14].

A project to establish formal verification in the development process in automotive industry is presented in [15]. The work [16] presents a study that explores the extent to which model checking can be performed *usefully* in an industrial environment, where usefully means that model checking is cheaper, faster, more thorough than conventional testing or review or able to find more subtle errors.

A complementary automotive case study on C code verification using BTC is presented in [17].

Contributions. Although a lot of research has been done on the verification of Simulink models, we are not aware of an *exhaustive analysis of SLDV where different verification approaches and the scalability are evaluated*. Moreover, our aim is to *investigate the gap that still exists to integrate formal verification into Simulink-based development*, even if a verification tool like SLDV is used, that is tightly integrated into Simulink. We present our observations and ideas to *improve the level of automation for the preprocessing of the model, the formalization of a specification, and the feature set of a verification tool*.

Outline. In Sect. 2 we describe our case study, the SLDV verification tool and specify the project goals. The verification process and the results are presented in Sect. 3. In Sect. 4 we list our observations and formalize some recommendations for computer-assisted solutions to integrate formal verification into Simulink-based development. We conclude the paper in Sect. 5.

2 The Case Study

First, we present our case study, the SLDV verification tool and our project goals. Due to confidentiality reasons, we cannot provide access to the concrete model and requirements, but we provide high-level insight into issues that are relevant for understanding the aims and results of this work.

2.1 Controller Model

Simulink. Simulink is an extension of Matlab which allows to build and simulate complex system models. Simulink (SL) models are block diagrams. Stateflow (SF) charts, that are based on finite-state machines and flow diagrams, can be embedded into Simulink models. Blocks and charts can be nested to create a hierarchical structure. For Simulink models which might contain Stateflow charts, in the following we use the abbreviation *SLSF model*.

Besides internal variables, Simulink models define a set of signals, (calibration) parameters and constants whose properties are fixed by attributes like, e.g. the data type, the dimensions, lower and/or upper bounds and initial values. *Signals* can take any value from their data type domain during a simulation while *constants* have a fixed value. *Calibration parameters* allow to define abstract models and specifications, which can be concretized by assigning concrete values to the parameters. E.g. a specification $x = c \cdot y$ parameterized in c can be concretized to $x = 2 \cdot y$ by fixing $c = 2$.

Controller model. Our case study models the R&D prototype feature `Low Speed Control` for a next-gen `Park Assist`. The `Park Assist` allows the vehicle to park automatically and operates at relatively low speeds compared to other driving situations. During assisted parking maneuvers, the vehicle speed has to be controlled at low vehicle speed targets and scenarios like climbing on a curb during parking have to be supported. The selected R&D prototype feature `Low Speed Control` takes the vehicle speed set point from `Park Assist` and controls the combustion engine speed and the brakes during automated parking.

We want to investigate how well Simulink formal verification performs for decision logic, state charts, filters, rate limiters, look-up tables and feedback control. Therefore, we selected our case study such that it contains a mixture of these different kinds of functionalities. The model has 41 open-loop functional requirements, 26 inputs, 5 outputs, 69 calibration parameters and 1095 blocks ($\approx$ 1500 lines of C code). The model contains Boolean, integer and floating-point variables: All input and output signals are scalar with the following data type distribution: 5 Boolean, 12 unsigned 8-bit integer and 14 single-precision floating point. Among the calibration parameters we have 52 scalar parameters (2 Boolean and 50 single-precision floating-point) and 17 parameters are arrays with 7 to 14 single-precision floating-point elements.

Requirements. The 41 textual requirements of the R&D prototype feature `Low Speed Control` describe the functional behavior of the controller and are used to develop the Simulink model. A single requirement typically describes only a part of the model that is implemented in a subsystem.

We classified 39 requirements as safety properties which follow the pattern *"Always* `P`*"*. In 30 of these safety requirements `P` is an invariant without any temporal operators. For the remaining 9 safety properties, `P` describes time-bounded temporal properties. The two remaining requirements are liveness properties which claim that something good eventually happens. They follow the pattern *"The value of* `x` *is eventually equal to* `c`*"* and express unbounded temporal properties.

Since 30 of the 41 requirements contain floating-point variables, the state space is relatively large in most cases. Only 11 requirements are restricted to Boolean and integer data types, which reduces the verification effort.

Most of the requirements can be specified using the following common operators: $+, -, *, /, \texttt{min}, \texttt{max}, \texttt{if}, \texttt{abs}$. Twelve requirements make use of special operators such as saturation, rate limiters, filters, PID controllers, or lookup tables.

2.2 Simulink Design Verifier

For our case study we used the commercial verification tool *Simulink Design Verifier* (*SLDV*) [1, 18]. Due to the tight integration into Simulink, the support team and detailed documentation we expect to minimize problems with embedding formal verification in an automotive development process. SLDV is a toolbox that offers the following analysis methods for SLSF models: Automatic test case generation, static analysis and *discrete-time formal verification*.

The SLDV tool uses software from *Polyspace* [19] and *Prover Technology AB* [20]. The latter offers (un)bounded model checking and test case generation. Unfortunately, the translation of the SLSF model and the specification into the formal input language of the verification engine, the verification engine itself and the generation of counterexamples remains a black box to the user. For `Property Proving` (formal verification) SLDV offers the following options: `FindViolation` checks if a property can be violated within a bounded number of steps, while `Prove` performs an unbounded analysis. `ProveWithViolationDetection` is a combination of `FindViolation` and `Prove` and performs a bounded analysis.

Properties that describe a *subsystem* of the model, i.e. properties that are restricted to the input and output signals of a subsystem, can be verified either on the complete model or on a subsystem (bottom-up approach). Subsystem verification is over-approximative because it considers arbitrary input for the subsystem instead of the values which it might receive as input signals in the complete model. As a consequence, properties that could be verified with subsystem verification hold also at the complete model level, but counterexamples on subsystem level might be spurious, i.e. not realizable in the complete model.

If the model includes *calibration parameters*, the verification can be performed either for a fixed model and specification (a concrete calibration parameter valuation is considered) or in one shot for all model and specification instances (all possible calibration parameter values are considered). For historic reasons, we speak of *fixed* or *varying* calibration parameters.

SLDV does not use a formal specification language with a formal semantics. Instead, an SLDV specification is an SLSF model which forms a *verification subsystem* (cf. Fig. 1). This specification language allows the flexibility to use complex operators and it is easy to use for engineers, but there is no formal semantics for Simulink blocks.

Each verification subsystem should contain at least one *proof objective* that outputs `true` as long as its input is constantly `true`. The input of a proof objective is typically the result of an implication or a comparison between the specification and an output of the model.SLDV is shipped with a library that contains — among others — verification subsystems, proof objectives and temporal operators.

The verification result of a requirement is *valid* if the corresponding proof objective returns `true` for all possible input combinations of the model. Otherwise, the verification tool cannot decide if the requirement is valid in the model or not and returns *undecided* or the requirement is *violated* and a counterexample is generated. The counterexample is given on the level of the SLSF model in form

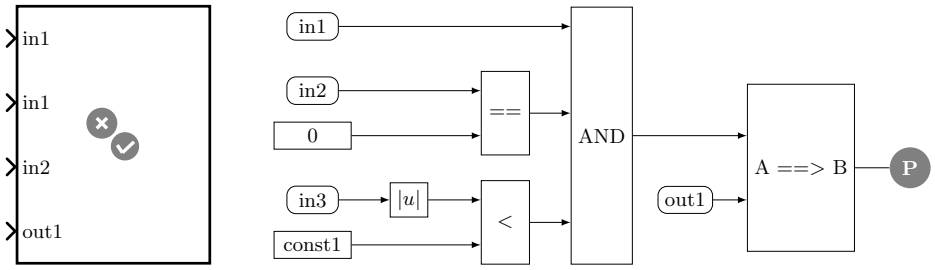

Fig. 1: A verification subsystem (left) and the specification (right) of the requirement: "If all of the following conditions are true: in1 is TRUE; in2 is zero; the absolute value of in3 is below a threshold (const1); then out1 shall be set."

of a *harness model*. This harness model can be used to simulate the execution path that violates the specification. An `HTML` or `PDF` report can be generated that contains the verification results, the analysis options and approximations SLDV applied to the model. An example for an approximation is floating-point arithmetic that is approximated by rational arithmetic.

In this paper, we primarily use the `Property Proving` feature of SLDV with the `Prove` option for an unbounded analysis.

2.3 Goals

Our project goal is to evaluate formal verification of Simulink controller models using SLDV, to identify empirical technical challenges for a tight integration of formal verification into Simulink-based development in the automotive industry and to propose solutions for a higher level of automation and a better support of engineers without expert knowledge in formal methods. We are especially interested in a clear separation of the controller model from the specification, the speed and scalability of the verification tool, the usefulness, generality and reliability of the analysis results and automated batch processing.

We assume that problems with the specification of complex operators like filters, feedback control and lookup tables might occur. Expected challenges for the verification tool are temporal requirements, the high proportion of floating-point variables and the complexity of the model.

3 Feasibility Analysis

In this section, we report on our experiences with formal verification of our SLSF model using SLDV and present the verification results. We performed our experiments using Matlab R2014b on a 2.5GHz Intel Core i5 machine with 8GB RAM running Windows 7 (64-bit).

3.1 Preparation

Requirement issues. We were not surprised to find incomplete, ambiguous and inconsistent formulations in 20 of the 41 textual requirements. Incomplete specifications have been found in 15 requirements, eight ambiguity issues have been detected, and two requirement pairs were inconsistent with each other. Causes for incompleteness were missing declaration of values for discrete signals (e.g., certain gear lever positions), missing information for complex operators like filters or hysteresis functions, or no mentioning of the output signal whose computation is described in the requirement. Reasons for the detected ambiguity in the requirements are imprecise formulations (e.g. to distinguish between the status or the event of status change), formulations that need further explanations and different textual descriptions for the same signal name. An example for inconsistency is two requirements that allow activation and deactivation of a signal to occur simultaneously.

Discussions with the requirement and feature engineers and reviewing the model implementation helped to resolve the issues.

SLDV Specification. Finally, we could manually transform all textual requirements to an SLDV specification in form of new verification subsystems.

For each requirement a separate verification subsystem was created and added to the model to have the flexibility to add/remove certain blocks and to copy them for verification on model- or subsystem-level. Some requirements have been easy to handle while for roughly half of the requirements discussions with the requirement and control engineers and/or information from the controller model were needed to clarify all issues. Finally, all 41 requirements could be specified for SLDV, although twelve requirements make use of *complex operators* like feedback control, rate limiters, filters and lookup tables, which might be difficult to express in a common specification language like formal logics.

Block replacement. A compatibility check of SLDV on our model revealed a set of *custom blocks* that are not supported by SLDV. These blocks include additional functionality for code generation but can be replaced by blocks from the standard library with equivalent functionality. We used the block replacement feature of SLDV to automate the replacement. This feature allows to continue the iterative development with the original model but to generate a model with replaced blocks for formal verification.

3.2 Verification

We analyzed each of the 41 requirements, specified as 41 independent verification subsystems, separately on model- and subsystem-level using either fixed or varying calibration parameters. To do so, we temporarily removed the other 40 verification subsystems that contain the specification of the other requirements.To analyze the impact of bounded temporal operators in the specification on the running time of the verification, we use fixed time bounds (five simulation steps) as an upper limit. This reduces the number of calibration parameters in our

Table 1: Verification results and accumulated running times for model and subsystem verification using varying and fixed calibration parameters.

	Model Verification		Subsystem Verification	
	Varying	Fixed	Varying	Fixed
Valid	24	26	25	26
Unknown	13	10	5	4
Invalid	4	5	11	11
Running time $> 300s$	7	7	2	2
Running time $> 7200s$ (TO)	1	1	1	1
Acc. running times [s]	22062	21465	7672	7783

model. We also fix the lookup-table data for the verification. Thus, for our analysis we consider only 47 out of 69 varying calibration parameters. An overview of the verification results is given in Table 1. For a majority of requirements we got conclusive results. However, the analysis revealed inconclusive results for up to 31% (resp. 12%) of the requirements on model- (resp. subsystem-)level. Reasons for inconclusive results are nonlinear behavior in the model and timeouts (running times $>$ 7200s) for temporal requirements.

Invalid verification results. Simulink Design Verifier detected eleven instances of invalid implementation against the specifications. Most of them were implementation flaws like missing or wrong operators. In other cases, implementation details were omitted in the textual requirements, parameters have been incorrectly calibrated, or the initialization causes requirement violations.

Analysis time. We did not notice a significant difference in the analysis time for valid, unknown and invalid verification results on subsystem-level: 39 verification results were delivered within four seconds. An exception are two *temporal properties* for which we observed running times above 300 seconds. On model level, 22 verification results are available within four seconds, 12 results have running times between 11 and 133 seconds and the running times of the remaining 7 requirements are above 300 seconds. Further investigations revealed that all running times above 300 seconds can be explained by temporal behavior either in the analyzed requirement or in a subsystem that delivers an input for the analyzed subsystem.

Model- vs. subsystem-level. We compare the results for verification on model- and subsystem-level: The number of *inconclusive verification results* can be reduced from 10 for fixed (respectively, 13 for varying) calibration parameters on model-level to 4 for fixed (resp. 5 for varying) calibration parameters on subsystem-level. Also the *analysis time* can be reduced by almost four hours if the verification is applied on subsystem-level. An explanation for both observations can be that on model-level nonlinear and/or temporal behavior in other parts of the model is propagated.

On subsystem-level we revealed two requirement violations for requirements that were proven valid on model-level. A missing minimum operator and time delays in the implementation that are not reflected in the textual requirement are the reasons for the invalid results on subsystem-level. These conflicting results were either caused by considering arbitrary inputs on subsystem-level, leading to spurious counterexamples in which some subsystem inputs are not realizable at the model-level. An alternative explanation can be a bug that produces false *valid* verification results in the verification on model-level.

Varying vs. fixed calibration parameters. Using subsystem verification, the running time could be decreased if we defined fixed (instead of varying) values for 47 calibration parameters, while for model verification the running time increased, both differences being within 3%.

Based on the experience with our model, we believe *SLDV to scale well with increasing state space size.* Although on subsystem-level only a few of the calibration parameters influence the verification, on model-level the effect on the state space is much stronger. Thus, we have the impression that SLDV can handle a large number of input signals and parameters with large data type domains quite well. To our surprise, *the number of inconclusive verification results does not change much with increasing state space*: For verification on model-level three and on subsystem-level one additional requirements could be decided using fixed calibration parameters.

Simultaneous execution on model-level. We found a serious tool issue in SLDV (R2014b – R2017a): When running the formal verification on model-level simultaneously for all 41 requirements, we detected *conflicting analysis results*, i.e. a valid and an invalid result for the same requirement. We observed a conflict for two different verification runs using simultaneous execution of all requirements. Another conflict occurred between a verification run using simultaneous execution (valid) and a verification run analyzing only the respective requirement either on model- or subsystem-level (invalid). We could confirm the requirement violations, i.e. the valid results are caused by a bug. Another finding using R2016b were verification runs using simultaneous execution of all requirements where *all requirements were reported as violated and no counterexamples were generated*. However, MathWorks assured that these problems are resolved in release R2017b. Note that these bugs in SLDV can also be an explanation for the conflicting results of a single requirement on model- and subsystem-level.

We also detected nondeterminism in the results (the order and number of counterexamples changes). This results from the fact that SLDV implements a portfolio solution where different counterexample search strategies are applied.

3.3 Scalability

For a better understanding of the scalability of SLDV with respect to temporal requirements on our model we analyzed the following property from the set of requirements for an increasing time duration d and fixed calibration parameter values on both model- and subsystem-level: "*If a Boolean signal* `in1` *is* `true` *for*

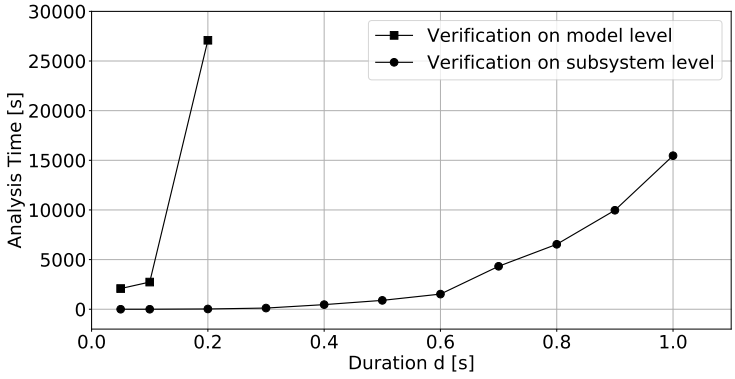

Fig. 2: The analysis time needed to prove a temporal property with increasing time duration d on model- and subsystem-level with fixed calibration parameters.

longer than a time duration d then the Boolean output signal `out1` *shall be set; it shall be cleared otherwise.*" The Boolean signal `in1` is computed using four floating-point signals `in2`, `in3`, `in4` and `in5` with single precision: "*The Boolean signal* `in1` *is* `true` *if any of the following conditions is* `true`*: The signal* `in2` *is above* 4000.0 *with offset* −100.0*, (the signal* `in3` *is above* 2000.0 *with offset* −30.0 *and the absolute value of the signal* `in5` *is below* 0.2*,), or the signal* `in3` *is not increasing and the signal* `in4` *is below the signal* `in3` *with offset* −30.0".

We considered time durations d between 0.05 and 1 second and a fixed simulation step size of 0.01 second. Note that the temporal operators of SLDV only support simulation steps as the unit for time bounds, we converted the time durations. The results are presented in Fig. 2.

The running time grows exponentially in the time duration d. The results show that on subsystem-level up to 80 simulation steps can be handled with a time out of 7200 seconds. More than four hours are needed to complete the analysis for 100 simulation steps. The results for model verification are much worse due to the complexity of the model. Already for 20 simulation steps more than seven hours were needed for the analysis.

For a more general result on the scalability of SLDV, a more rigorous analysis on a larger benchmark set would be needed.

4 Lessons Learned

In this section, we report our observations and recommendations for formal verification of SLSF models using SLDV.

4.1 Specification

Interferences. We encountered several examples with different requirements for the same output signal. These interferences may lead to inconsistencies.

Proper specification of priorities can be applied to solve this kind of inconsistency. Requirement engineers may also follow the defensive approach to use exactly one requirement for each disjunctive part of the model.

Specification language. The Simulink Design Verifier uses Simulink as specification language. On the one hand, Simulink offers a rich set of operators that are often not available in other specification languages. To speed up the specification work, *a custom library with efficient implementations of commonly used operators might be helpful.* For example, a hysteresis function and customized lookup tables can be provided. This approach also ensures that the same block parameters (e.g. interpolation and extrapolation methods) are used for each occurrence of an operator. On the other hand, we are not aware of a formal semantics for Simulink, and using the same language for modeling and specification can easily lead to false-positive verification results. *A formal specification language may assist the requirement engineers to avoid incompleteness and ambiguity* [21]. This language should be easy to understand for engineers without expert knowledge in formal methods. We propose a pattern-based textual language as in [22, 23]. This formal specification language would also help to separate the formalization of a property from its Simulink implementation.

Compositional reasoning. All 41 requirements have been formulated on subsystem level. Thus, verification on subsystem-level and bottom-up compositional reasoning can be applied to verify the complete model. The main benefits of this approach are shorter analysis time and stronger verification results.

4.2 Model

Floating-point numbers. A big challenge for the formal verification are the *floating-point numbers*, which are approximated by SLDV. Although it is not possible to eliminate all floating-point signals, we encourage to search for integer implementations with equivalent functionality. The R&D prototype `Low Speed Control` contains timers with floating-point arithmetic, which often leads to undecided results or to spurious counterexamples, the latter because floating-point approximation leads to property violation but the provided counterexample cannot be confirmed by simulation, where the approximation is not applied.

Consider a timer using floating-point arithmetic and a counter using integer arithmetic, that are both initialized with 0. In each simulation step, the timer is updated according to $\texttt{timer} := \texttt{timer} + \texttt{timestep}$ while the counter is increased by one: $\texttt{counter} := \texttt{counter} + 1$. The timer is reset if the upper bound `ub` is reached ($\texttt{timer} \geq \texttt{ub}$). The counter restarts if an upper limit $\texttt{c} := \lfloor \frac{\texttt{ub}}{\texttt{timeStep}} \rfloor$ is reached ($\texttt{counter} \geq \texttt{c}$). Because controller models use fixed-step solvers, the bound `c` can always be computed. Using such transformations, *we recommend to replace floating-point timers by equivalent integer counters* in Simulink models. We also found another example in the model where the number of operations on floating-point variables could be reduced by an alternative implementation.

Value domains. To keep the state space as small as possible and the analysis time low, a system engineer should specify lower and upper bounds for all input,

output and calibration parameters reducing the admissible valuations as much as possible. E.g., the data type domain of a floating-point signal representing the vehicle speed can be restricted to values between 0 and 320 km/h.

Stateflow models. Special attention is needed for Stateflow implementations. We observed time delays, that are not reflected in the textual requirements. For example, moving from one state to another takes a simulation step, and outgoing transitions of the new state are not evaluated immediately. The engineer should be aware of the time delays he/she introduces in the model and the textual requirements should be updated. To facilitate the specification for a Stateflow chart, *a variable storing the active state of the chart should be introduced.*

Calibration parameters. A special class of calibration parameters are time bounds. Our case study includes such a calibration parameter whose value should be set to the simulation step size before each simulation run. To verify such models for all possible step sizes, we propose to automatically replace all occurrences of such calibration parameters by `Weighted Sample Time` blocks or to add an assertion stating the equality between the calibration parameter and the output of a `Weighted Sample Time` block, before formal verification is started.

4.3 Verification Tool

Usability. Simulink Design Verifier is intuitive to use and easy to integrate in Simulink. Although no expert knowledge is necessary to apply formal verification on a Simulink model, finding explanations for certain verification results and the development of workarounds are hardly possible without special expertise. The black box implementation of SLDV hinders even experts to exploit the strengths of the underlying verification techniques and to avoid their weaknesses.

Inconclusive results. We encountered a lot of inconclusive verification results. Most of them occurred due to nonlinear behavior in the model. Often the issues could be resolved by verification on subsystem level, which indicates that the nonlinear behavior on model-level is propagated from other parts of the model. However, it is hard to identify the block causing the nonlinear behavior. It would be very helpful to have tool support. We also suspect that sometimes issues with floating-point arithmetic are reported as inconclusive due to nonlinear behavior in the model.

Floating-point variables. SLDV approximates floating-point arithmetic by rational arithmetic. This approximation often results in inconclusive verification results or spurious counterexamples, which cannot be confirmed by simulation where the approximation is not applied. Even more dangerous are results reporting correctness, because they are not reliable: the generated `C`-code of a successfully verified Simulink model might violate its requirements. We would appreciate to get more information how the approximation is done, how it affects the verification outcome, whether the analysis tool uses exact or inexact computations and an explanation for non-expert users that counterexamples might not be reproducible via simulation.

Specification implementation. To analyze whether the model is initialized correctly, we compared two different implementations of a correct-initialization requirement for an unsigned integer variable: the first one uses an `Extender` block (a temporal operator provided by SLDV) and the second one a `Delay` and an `Implication` block. To our surprise, the running times were quite different: while the verification result was available in less than a second for the first approach, the analysis time was 23 seconds for the second one. This example demonstrates once more that for an *efficient* implementation sometimes knowledge of the verification techniques or even implementation details are necessary.

Model- vs. subsystem-level verification. The strengths of model-level verification are the clear separation between specification and implementation and the possibility to analyze all enabled proof objectives simultaneously. Drawbacks are longer analysis times and more inconclusive results. Verification on subsystem level is faster and independent of other parts of the model, yielding conclusive results in more cases. Verification results on subsystem-level can be reused if a subsystem is embedded into another model, while results on model-level cannot be transferred. However, the user needs to know the model well to be able to identify a subsystem that assures the validity of a requirement, independently of its context. Moreover, for subsystem-verification the chosen subsystem must be modified to be an *atomic unit*. Though this modification does not change the behavior of the subsystem, it affects the processing order of blocks in the model. Therefore, these modifications must be undone after verification. An automated solution would be helpful that temporarily treats the subsystem as atomic during verification while the original model remains unaffected.

Recommended tool-chain. A first verification run should be performed on model-level using varying calibration with all proof objectives enabled and with a short running time e.g., 100 seconds. If verification on model-level yields inconclusive results, the analysis of the corresponding requirement should be repeated on subsystem-level with a larger running time for temporal requirements. Since the analysis time of SLDV for temporal requirements can increase exponentially in the number of simulation steps, we recommend to start with a small number of time steps (e.g. 5), which can slowly be increased to more realistic values as long as the analysis time remains acceptable. It might also be possible to include a different verification tool that scales better for temporal properties in the tool chain. Note that if any parameter value or data type domain is changed in the model, e.g. if lookup table data is replaced, former verification results cannot be trusted anymore for all analyses on model-level and for those task on subsystem-level, where the changed parameter is used in the subsystem.

Counterexamples. If verification on model-level verification reports a counterexample we recommend to simulate it to strengthen its reliability. Subsystem verification assumes arbitrary subsystem-inputs and may therefore produce spurious counterexamples, but these subsystem-counterexamples cannot be easily simulated at model level because the inputs for the subsystem's environment are not fixed. If the user can argue that the inputs in the subsystem-counterexample are not possible in the full model, we recommend to limit the domains for the

subsystem's inputs by adding assumptions and re-check the limited subsystem. Otherwise, if a subsystem-counterexample seems plausible, we propose to add assumptions to restrict the subsystem's behavior to the counterexample and apply model-level verification to this restricted model to search for an extension at the model level. Restricting the subsystem's behavior to the counterexample is doable, however, it is a non-trivial and tedious task. Furthermore, we observed one case where subsystem verification returned a counterexample but at model-level the requirement could be proven, indicating the spuriousness of the subsystem counterexample. However, we found strong indications that instead a software bug produced a false *valid* verification result on model-level.

Batch verification. For large-scale applications, formal verification with a higher level of automation is needed that offers one-click solutions for the sequential verification of a set of requirements both on model- and on subsystem-level. Currently, the simultaneous verification of a set of requirements is possible on model level, but we have the impression that this mode does not process the requirements sequentially one after the other, but it rather uses an incremental verification technique (possibly incremental SMT-solving) that considers all requirements simultaneously. Although we are aware that using the provided API, it is possible to develop a custom solution for batch processing on model- and subsystem-level, a built-in solution would be appreciated.

Temporal requirements. One weakness of SLDV is the verification of temporal requirements. In some cases, checking temporal requirements over just 5 simulation steps took more than two hours. For the temporal operators provided by SLDV, numerical values are needed to specify upper bounds for the time steps. The support of constants or calibration parameters would offer much more flexibility, e.g. if different bounds need to be checked. We also noticed that the upper bounds are restricted to values of ≈ 128 simulation steps (depending on the temporal operator), while in practice larger time bounds might be needed.

Calibration parameters. We like the automated solution to verify models for all calibration parameter values from some user-defined intervals, which is of high relevance in the automotive sector. However, it is not clear what such an interval domain means for, e.g., lookup tables data. We encountered a bug in R2014b where calibration parameters of data type *Boolean* were not supported. An available bug-fix for R2015b could be adapted to R2014b. Furthermore, calibration parameter values can be restricted to intervals but we have found no ways to restrict such hyperrectangle-domains further by putting restrictions on the relation of different parameters, e.g. assuming that the value of one parameter is not larger than the value of another one. As some blocks work correctly only for certain calibration parameter combinations, it was sometimes necessary to add assumptions to such blocks when verifying over calibration parameter domains. For example, a saturation block expects an upper-bound value to be larger or equal to a lower-bound value; if the definition of these bounds involve some calibration parameters then we need to add such assertions to the model.

Reliability. During our experiments we detected (and reported) some bugs in SLDV. We are aware that software for formal verification is quite complex and

that bugs in the code are likely. Thus, our recommendation is to use more than one analysis tool for a better reliability on the verification results (though, for Simulink unfortunately the number of available tools is quite small). It would also be helpful to have open-source tools, which offer the possibility for temporary patches by the user, such that he/she can proceed and does not need to wait for the next release. Furthermore, we strongly recommend to use the latest tool release if possible to avoid resolved bugs.

Verification report. SLDV generates a report containing information like verification results, analysis times and applied approximations. Counterexamples can be simulated which is very helpful for the detection of property violations. Unfortunately, the time that is needed for translation and compilation of the model to create the input for the verification engine is not listed in the report.

5 Conclusions and Discussion

In this paper, we shared our experiences with the application of formal verification to an automotive controller model using the commercial verification tool SLDV. Despite the mixture of different functionalities in the model and all requirement issues, we achieved verification results for all 41 functional requirements of the R&D prototype feature `Low Speed Control`.

SLDV is easy to use, well integrated into Simulink and provides features for a high degree of automation like block replacement and support for calibration parameters. Still, the level of automation could be further improved, e.g. by one-click solutions for batch processing. All in all, we experienced SLDV to be scalable for open-loop controller models, especially using subsystem-level verification, but temporal properties bring SLDV to its limits. Closed-loop models that contain plant models with continuous time and varying-step solvers are currently not supported. A serious concern is the missing formal semantics for Simulink and the black box implementation of the verification tool. We further discovered a serious tool issue that leads to contradicting verification results. We recommend to use SLDV release R2017b, where the bug is supposedly resolved.

We strongly suggest to not only apply verification on the Simulink model where counterexamples can be analyzed relatively easy and comfortable in the high-level, hierarchical, graphical simulation environment. Additionally, verification on generated C code [17] can be beneficial since data types like floating-points and data type domains can be handled exactly.

Although SLDV is quite comfortable to use, there is still a gap which needs to be closed for a smooth integration in the industrial Simulink-based development. For future work, we want to put further effort into closing this gap and developing computer-assisted methods for complete, unambiguous and consistent requirement writing and for verification into Simulink-based development.

Acknowledgments. The authors want to thank Petter Nilsson, Philipp Berger, William Milam and Cem Mengi for numerous fruitful discussions on Simulink verification. We also express our appreciation to the MathWorks support team for their fast response and helpful advice.

References

1. MathWorks: Simulink Design Verifier, `https://de.mathworks.com/products/sldesignverifier.html`.
2. Gholami, M.R., Boucheneb, H.: Applying formal methods into safety-critical health applications. In: Proceedings of IMBSA'14. Volume 8822 of LNCS., Springer International Publishing (2014) 195–208
3. Boström, P., Heikkilä, M., Huova, M., Waldén, M., Linjama, M.: Verification and validation of a pressure control unit for hydraulic systems. In: Proceedings of SERENE'14. Volume 8785 of LNCS., Springer International Publishing (2014) 101–115
4. Barnat, J., Bauch, P., Havel, V.: Temporal verification of Simulink diagrams. In: Proceedings of HASE'14, IEEE (2014) 81–88
5. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool chain to support automated formal verification of avionics Simulink designs. In: Proceedings of FMICS'12. Volume 7437 of LNCS., Springer Berlin Heidelberg (2012) 78–92
6. Barnat, J., Brim, L., Beran, J.: Executing model checking counterexamples in Simulink. In: Proceedings of TASE'12, IEEE (2012) 245–248
7. Reicherdt, R., Glesner, S.: Formal verification of discrete-time MATLAB/Simulink models using Boogie. In: Proceedings of SEFM'14. Volume 8702 of LNCS., Springer International Publishing (2014) 190–204
8. Herber, P., Reicherdt, R., Bittner, P.: Bit-precise formal verification of discrete-time MATLAB/Simulink models using SMT solving. In: Proceedings of EMSOFT'13, IEEE (2013) 1–10
9. Cofer, D.: Model checking: Cleared for take off. In: Proceedings of SPIN'10. Volume 6349 of LNCS., Springer Berlin Heidelberg (2010) 76–87
10. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Proceedings of FMICS'07. Volume 4916 of LNCS., Springer Berlin Heidelberg (2008) 68–84
11. Bourbouh, H., Garoche, P.L., Garion, C., Gurfinkel, A., Kahsai, T., Thirioux, X.: Automated analysis of Stateflow models. In: Proceedings of LPAR'17. Volume 46 of EPiC Series in Computing., EasyChair (2017) 144–161
12. Dragomir, I., Preoteasa, V., Tripakis, S.: Compositional semantics and analysis of hierarchical block diagrams. In: Proceedings of SPIN'16. Volume 9641 of LNCS., Springer International Publishing (2016) 38–56
13. Preoteasa, V., Dragomir, I., Tripakis, S.: Type inference of Simulink hierarchical block diagrams in Isabelle. In: Proceedings of FORTE'17. Volume 10321 of LNCS., Springer International Publishing (2017) 194–209
14. Ali, S., Sulyman, M.: Applying model checking for verifying the functional requirements of a Scanias vehicle control system. Master's thesis, Mälardalen University (2012)
15. Botham, J., Dhadyalla, G., Powell, A., Miller, P., Haas, O., McGeoch, D., Rao, A.C., O'Halloran, C., Kiec, J., Farooq, A., Poushpas, S., Tudor, N.: PICASSOS – Practical applications of automated formal methods to safety related automotive systems. In: SAE Technical Paper, SAE International (2017)
16. Bennion, M., Habli, I.: A candid industrial evaluation of formal software verification using model checking. In: Proceedings of ICSE Companion'14, ACM (2014) 175–184
17. Berger, P., Katoen, J.P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifiying auto-generated C code from Simulink — an experience report in the automotive

domain —. In: Proceedings of FM'18. LNCS, Springer International Publishing (2018) To appear.
18. MathWorks: Simulink Design Verifier – User's guide, `https://de.mathworks.com/help/pdf_doc/sldv/sldv_ug.pdf`.
19. MathWorks: Polyspace, `http://www.mathworks.com/products/polyspace/`.
20. Prover Technology AB: Prover Plug-In, `http://www.prover.com`.
21. Bozzano, M., Bruintjes, H., Cimatti, A., Katoen, J.P., Noll, T., Tonetta, S.: The compass 3.0 toolset (short paper). In: Proceedings of IMBSA'17. (2017)
22. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of ICSE'99, ACM (1999) 411–420
23. 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 Transactions on Software Engineering **41**(7) (2015) 620–638