Multiple Analyses, Requirements Once:
Simplifying Testing and Verification in Automotive Model-Based Development

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

[1] RWTH Aachen University, Aachen, Germany
{berger,johanna.nellen,katoen,abraham}@cs.rwth-aachen.de
[2] Ford Motor Company, Dearborn, USA
mwaez@ford.com
[3] Ford Research and Innovation Center Aachen, Aachen, Germany
trambow@ford.com

Abstract. In industrial model-based development (MBD) frameworks, requirements are typically specified informally using textual descriptions. To enable the application of formal methods, these specifications need to be formalized in the input languages of all formal tools that should be applied to analyse the models at different development levels. In this paper we propose a unified approach for the computer-assisted formal specification of requirements and their fully automated translation into the specification languages of different verification tools. We consider a two-stage MBD scenario where first Simulink models are developed from which executable code is generated automatically. We (i) propose a specification language and a prototypical tool for the formal but still textual specification of requirements, (ii) show how these requirements can be translated automatically into the input languages of Simulink Design Verifier for verification of Simulink models and BTC EmbeddedValidator for source code verification, and (iii) show how our unified framework enables besides automated formal verification also the automated generation of test cases.

1 Introduction

In the automotive industry, software units for controllers are often implemented using *model-based development* (*MBD*). The industry standard ISO26262 recommends *formal verification* to ensure that such safety-critical software is implemented in accordance with the functional requirements. The work of our previous two papers [2,20] and this paper not only applies to safety critical automotive software but also to quality management (QM) or non-safety critical automotive software. In fact, we worked only on Ford QM software features in our papers. To optimally exploit recent academic developments as well as the capabilities of state-of-the-art verification tools, Ford Motor Company and RWTH Aachen University initiated an alliance research project to analyze how

K. G. Larsen and T. Willemse (Eds.): FMICS 2019, LNCS 11687, pp. 59–75, 2019.
https://doi.org/10.1007/978-3-030-27008-7_4

formal verification techniques for discrete-time systems can be embedded into Ford's model-based controller development framework, and to experimentally test their feasibility for industrial-scale C code controllers for mass production.

In our previous works [2,20], we considered an MBD process starting with the development of Simulink controller models and using Simulink's code generation functionality to derive C code for software units. For formal verification, we analyzed the feasibility of both *Simulink Design Verifier* (`SLDV`) for Simulink models as well as `BTC` *EmbeddedPlatform* verification tool for the generated C code. Our papers [2,20] present our observations and give recommendations for requirement engineers, model developers and tool vendors how they can contribute to a formal verification process that can be smoothly integrated into MBD.

The most serious pragmatic obstacles that we identified for the integration of formal methods are related to the *requirement specifications*. The requirement specifications were given informally in natural language. All the considered natural language requirements described time-bounded linear temporal logic (LTL) properties, which we manually formalized for both the `SLDV` and the `BTC` verification tools. During the formalization we detected ambiguity, incompleteness or inconsistency for roughly half of the textual requirements.

The manual formalizations needed discussions with requirement engineers to clarify and correct these flaws. However, a high degree of automation is a prerequisite for mass production and the integration of formal methods into the established MBD process at Ford. Automation allows the usage of formal verification within a development team of engineers with little knowledge of formal verification. Ideally, verification is automatically triggered whenever changes have been made to either the requirements, the Simulink model, or the used verification tools. Verification results can then be stored and compared with previous runs, making deviations from previous results easily detectable. All deviations can then be reported to a person with a strong background in formal methods for thorough investigation.

We also encountered problems rooted in the fact that the formalizations for the two different formal tools were done independently due to syntactic differences: in Simulink, requirements are themselves Simulink models that need to be embedded into the models that should satisfy them, whereas in `BTC` the requirements can be specified either using a graphical interface for pattern-based specification or directly in an XML-based file input format.

The independence of multiple requirement formalizations has several disadvantages. First and foremost, basically the same work is done multiple times, using different input languages. In addition, the formalizations have the risk to be slightly different. This may result in potentially incompatible analysis results requiring a deep and time-consuming analysis. *When the formalizations are done independently, they cost additional resources in time and expert knowledge, raising development cost.*

In addition, typically several programming and modeling languages are used within a company such as Ford. The preference of these languages changes over time and each language has its own analyses tools. Different teams within a

company like Ford may use different tools for the same purpose. The fact that almost every formal verification tool has its unique input language is a big obstacle to introduce formal methods into versatile companies like Ford. *A common requirement language for all formal verification tools may help to take advantage of the strengths of different tools*.

To diminish these problems, this paper presents a *common formal requirement specification framework*. We focus on Simulink and C code verification in the automotive domain, but our framework is naturally extensible to further languages and tools. Concretely, the paper makes the following main contributions:

1. We identify *a small fragment of LTL* as a formal specification language that is expressive enough *for the formalization of typical requirements in the context of MBD in the automotive sector*.
2. We describe our *tool* that was designed as a prototype for use inside this research project as a proof of concept. Similar to BTC EmbeddedSpecifier it assists users who are not experts in formal methods to specify unambiguous and complete formal requirements using textual descriptions according to a pattern-based syntax.
3. We propose an approach for the *fully automated translation of the above-specified formal requirements into Simulink models* that can be embedded in `SLDV` verification processes.
4. We describe how to *automatically generate `BTC` models* from those formal requirements for source code verification.
5. We show how to *automatically generate test objectives* from formal requirements that can be used for automated test-case generation.

Our framework is illustrated in Fig. 1. While computer-assisted approaches for formal requirement specification have been proposed (see Sect. 2), we believe that our approach supporting direct analysis using multiple tools at different development levels is novel, especially the automated specification export to Simulink and the generation of test-cases.

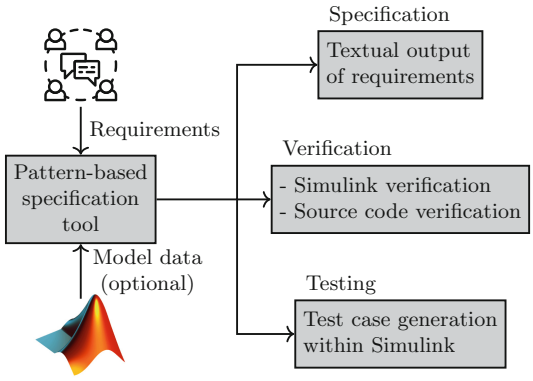


Fig. 1. The structure of our unified specification and analysis framework.

2 Related Work

Patterns for specifying properties in formal verification were introduced by Dwyer et al. [7]. Cheng et al. has extended this work to real-time properties [14] and Grunske introduced a pattern system for probabilistic properties [10].

Autili et al. [1] recently presented a unified pattern catalog that combines known patterns for qualitative, real-time and probabilistic properties. Then the work has been extended by some new patterns. Our works relies on the pattern catalogs from [1,7]. Inspired by our experience with Ford [2,20], we selected a set of patterns that covers more than 90% of our investigated automotive requirements.

Several tools are available for pattern-based specifications. The PSPWizard [17] and the SESAMM Specifier [8] provide for a given pattern library export functionalities to a formal logic or to a textual representation. The SESAMM specifier has been integrated into an industrial toolchain in the automotive domain. The tool PASS (**P**roperty **ASS**istant) [21] guides the user by a set of questions towards a suitable pattern from which a μ-calculus formula and a UML sequence diagram can be generated. The tool PROPEL [22] represents patterns in natural language and by finite-state automata. The COMPASS toolset [6] supports the original patterns by Dwyer, real-time- and probabilistic patterns. While the previous mentioned tools use the pattern catalog from [1,6,7], the work [16] presents different patterns and a tool for the requirement specification and automated translation to a formal language. The tool DDPSL [11] goes a step further by allowing the user to fill the templates in a pattern with assumptions on the variables using a defined set of logical and temporal operators. The ReSA tool [18] allows an automated consistency check of requirements on multiple abstraction levels using SAT checking. The commercial tool BTC EmbeddedPlatform[1] also offers the possibility to formalize textual requirements in a pattern-based language. Former versions of the tool support a pattern catalog but the latest release uses the *universal pattern* [23] that offers a graphical specification for trigger-action based requirements. Our tool focuses on the key patterns but allows for automated generation of test cases, as well as properties for Simulink model and source code verification.

Besides the tools for pattern-based specifications, several experience reports on using specification patterns have been published. In [4], a case study in the field of service-based applications is presented. [24] reports on an approach using pattern-based specifications in the area of work flow modeling. Bosch company investigated the suitability of the pattern catalog from [14] for 289 informal behavioral requirements from their automotive projects. A report on the integration of a pattern-based specification tool in an industrial (automotive) tool chain is given in [8,9]. A restricted set of patterns was used for the formal specifications within the PICASSOS project [5]. A system for modeling and testing flight critical software was presented in [19], but their focus lies on test-case generation and modeling structural aspects of the software system, whereas our focus is on the automated translation of requirements.

3 Pattern-Based Requirement Specification Language

Requirement documents are commonplace in the automotive industry and are usually written in natural language by a large number of stakeholders. These

[1] https://www.btc-es.de/en/products/btc-embeddedplatform/.

Table 1. Pattern distributions for three different controller models.

Pattern	LSC		DSR		ECC	
Invariant	35	85.4%	50	92.6%	80	97.6%
Time-bounded response (exact bound)	5	12.2%	4	7.4%	2	2.4%
Event-bounded response	1	2.4%	0	0.0%	0	0.0%

can include engineers and other people without a strong background in formal methods, which may lead to ambiguous requirements. Specification patterns may assist engineers in writing complete and unambiguous textual requirements. A pattern defines the behavior that is described by a requirement and uses templates for additional information like the specification of events and durations. In contrast to most existing approaches, events are specified by a constrained grammar, and higher-order operators, e.g. hysteresis[2], are supported to enable specifying new operations on events.

Goals. The pattern-based specification language should produce human readable specifications. A formal semantics avoids ambiguities and allows the automated generation of tool-specific requirement specifications. Our aim is to keep the pattern language simple such that no expert knowledge is needed and the learning curve for requirement engineers is low. We believe that a limited number of simple patterns reduces incorrect choices of patterns or scopes when writing requirements while still covering a high percentage of requirements.

Why Yet Another Specification Language? Tools like `BTC EmbeddedPlatform` come with their own, existing, pattern-based specification language and there are existing tools for pattern-based specification. Nonetheless we decided that creating our own language and tool was the better choice. A key difference from many established pattern-based specification languages is that we also require the events to be specified using a constrained grammar, enforcing the events to be formalizable properties. This, in turn, allows us to immediately export the entire property to a supported format without the need for any further user interaction.

Adding new features or constructs like higher-order operators (e.g. hysteresis) is easy to achieve, requiring only very modular changes to the grammar and the back-end exporter classes. We want to be able to create our own pool of higher-order operators for event specification that can be used to ease the burden of formalization for the engineers. Our own language allowed us to do rapid prototyping while coming up with new ideas, without the burden of getting all stakeholders of an established language on board beforehand.

Syntax. We used [1,7,14] as a starting point to design our *pattern-based requirement specification language* $\mathcal{L}$, whose grammar is shown in Fig. 2; for more details see also Appendix A.

[2] Hysteresis is a functionality often used to prevent rapid toggling when observing an input signal against some threshold by introducing an upper and a lower delta.

```
specification: scope pattern;
scope:         initially | globally;
pattern:       invariant | response;
initially:     'At system start,';
globally:      'At each time step,';
invariant:     '[' event '] holds.';
response:      'if [' event '] has been valid for [' duration '],'
               'then in response, after a delay of [' duration '],'
               '[' event '] is valid for [' duration '].';
event:         identifier | event AND event | ...| term ≤ term | ...
term:          identifier | term + term |...
duration:      uint unit;
uint:          [1..9] [0..9]*;
unit:          'simulation steps' | 'milliseconds' | 'seconds'
               | 'minutes' | 'hours';
```

Fig. 2. Syntax of our pattern-based requirement specification language.

Requirement *specifications* consist of a scope followed by a pattern. We start with a limited set of scopes and patterns that can be extended later to cover further specification types. However, these limited sets were sufficient to formalize more than 90% of the requirements in all three case studies (Low Speed Control for Parking (LSC), Driveline State Request (DSR) and E-Clutch Control (ECC)) we considered in [2,20] (see Table 1). Other internal case studies from Ford show similar results.

Currently two *scopes* are supported: the `initially` scope is used to express that a property should hold at system start, i.e. at time step 0 of a simulation before any operations have been performed, while the `globally` scope expresses that a property should hold at each time step of an execution, but starting after the first execution. In [1,7,14] there are further scopes like `before R`, `after Q`, `between Q and R` and `after Q until R` that can be considered for future inclusion.

We support two *patterns* for defining which property is required to hold for a given scope. The `invariant` pattern allows to state that a certain event holds (at each time step within the specified scope), and covers both the `absence` and the `universality` patterns from [7] if the negation of events is supported. The `response` pattern specifies causalities between two events: the continuous validity of a trigger event for a given trigger duration implies that after a fixed separative duration the response event holds continuously for a given response duration.

The *events* in the above patterns are built from identifiers (`signals`, `constants` and (calibration) `parameters`) using a set of `functions` and `operators`. We support those operators and functions that were used in our case studies, including the Boolean operators AND, OR, NOT and IMPLIES, the relational operators $<$, $\leq$, $>$, $\geq$ and $=$, the arithmetic operators $+$, $-$, $\cdot$ and $/$, absolute value, minimum, maximum, functions for bit extraction (bit x of variable y) and

$$
\begin{array}{lll}
\text{scopes:} & [\![\texttt{initially pattern}]\!] & = [\![\texttt{pattern}]\!] \\
& [\![\texttt{globally pattern}]\!] & = \bigcirc\, \Box\, [\![\texttt{pattern}]\!] \\
\text{patterns:} & [\![[e]\ \textit{holds.}]\!] & = [\![e]\!] \\
& [\![\textit{if}\ [e_P]\ \textit{has been valid for}\ [d_P],\ \textit{then in response, after a delay of}\ [d], & \\
& \ [e_Q]\ \textit{is valid for}\ [d_Q].]\!] & \\
& & = \left(\Box^{[\leq [\![d_P]\!]}\ [\![e_P]\!]\right) \rightarrow \left(\Diamond^{\,[=[\![d_P]\!]+[\![d]\!]}\ \Box^{[\leq [\![d_Q]\!]}\ [\![e_Q]\!]\right) \\
\text{events:} & [\![\texttt{identifier}]\!] & = \texttt{identifier} \quad \ldots \\
& [\![e_1\ \texttt{AND}\ e_2]\!] & = [\![e_1]\!] \wedge [\![e_2]\!] \quad \ldots \\
& [\![t_1 \leq t_2]\!] & = [\![t_1]\!] \leq [\![t_2]\!] \quad \ldots \\
\text{durations:} & [\![n\ \textit{seconds}]\!] & = \frac{1000 \cdot n}{D_{\text{Step}}} \quad \ldots
\end{array}
$$

Fig. 3. Semantics of our pattern-based requirement specification language.

time delays (value of x n steps ago). The complete ANTLR grammar for events is presented in Appendix A. We plan in future work to incorporate more advanced operators like *state change* ("the value of [param] transitions from [const1] to [const2]"), different variants of *hysteresis* functions, *saturation*, *rate limiter* and *ramping up* functions and *lookup tables*. Note that though custom operators and functions allow users a more efficient specification, special operators (e.g. lookup tables) might not be realizable in all specification languages for which export is provided.

Semantics. We define the semantics of requirement specifications using linear temporal logic with quantitative temporal operators to express time durations, that is MTL [15]. The main semantical components are shown in Fig. 3 using only future temporal modalities (straightforward and therefore not listed in Fig. 3 are the semantics for events and durations, see Appendix A for a complete definition). We use D_{Step} to denote the step-size, here in milliseconds. We support durations that are multiples of D_{Step}. An equivalent semantics using past temporal modalities is given in Appendix A. The difference in terms of a time-shift between the formulations using past (resp. future) operators is illustrated in Fig. 4. The two equivalent semantics support the export of a pattern-based specification into different specification languages. For example the specification language of the analysis tool `SLDV` only supports past temporal operators.

4 Pattern-Based Requirement Specification Tool

We have implemented a pattern-based specification tool as a prototype for use inside this research project to support requirement engineers in writing unambiguous and complete textual requirements. Our focus was to create a modular tool that is easy to learn and extendible, if in the future a larger set of scopes, patterns or operators needs to be supported.

A user can either import signals, calibratables and constants from a file, or create, change and delete them manually. Calibratable parameters remain constant during software execution but can be adjusted before the execution for

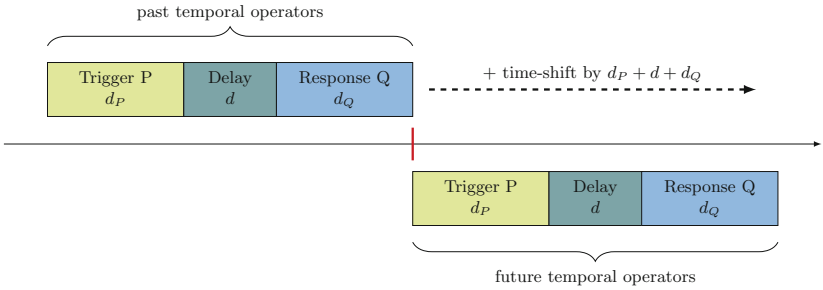


Fig. 4. Evaluation of a response pattern with past or future operators. The present is represented by the red tick on the timeline. (Color figure online)

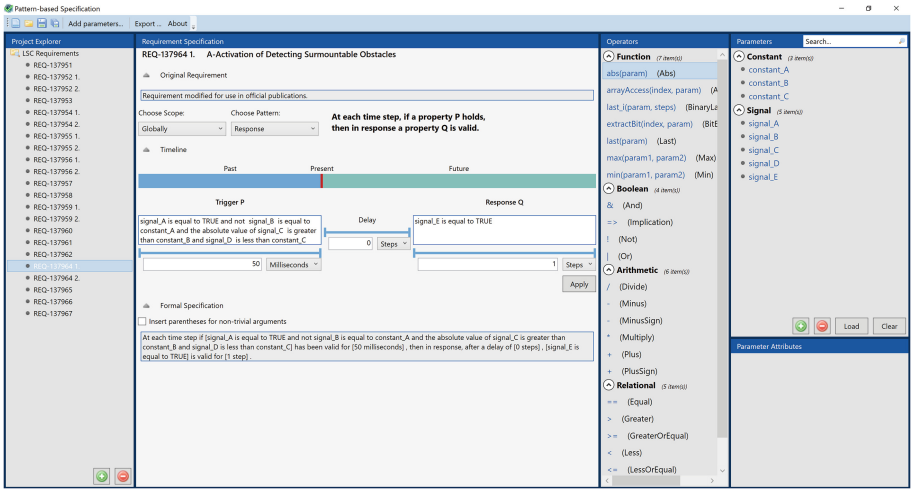


Fig. 5. User interface of our pattern-based specification tool.

tuning or selecting the possible functionalities. Captured data includes a name, description, minimum and maximum values, dimensions, a value, the data type and the variable type (signal, calibratable or constant). With the information of available variables readily available, the tool checks specified events for whether all referenced variables actually exist.

The current version of the tool provides export functionality for a selected requirement or for all of them. Export formats are textual (.txt), `SLDV` (.m), `BTC` (.spec) and C (.c) specifications. The last one is compatible with the SV-COMP standard [3] and can be used for formal verification with e.g. the *Ultimate Automizer* [12].

The requirement specification panel in Fig. 5 is the main panel of our tool. A scope and a pattern must be selected for the requirement. A textual translation of the scope and pattern is given as well as a visualization that shows the time steps where the chosen pattern is evaluated, see Fig. 5. Events are built

using operators, functions, signals, constants and calibration parameters. For each event, a duration and a time unit can be specified. Additionally, for patterns with more than one event, a time delay between events can be specified, again together with a time unit.

If the pattern-based specification is incomplete or if it contains specification errors, the lower part of the specification panel provides the list of errors and warnings. When all issues are resolved, a textual formal specification is generated from the specification. The modular set-up of the tool allows to add further exporters, e.g. to generate specification in a logic like MTL in a straightforward manner.

Our prototypical implementation supports the functions `abs(param)`, `min(param1, param2)`, `max(param1, param2)`, `last(param)`, `last(param, steps)` and `extractBit(index, param)`. For an explanation see Appendix A. Parenthesis expressions can be built using `(param)` and the basic boolean operators `not`, `and`, `or` and `implication` are provided.

5 Requirement Specification Export to Verification Tools

5.1 Export to SLDV

A formal pattern specification is exported to Simulink in the form of a Matlab script. This script generates a specification block inside a model on the currently selected hierarchy level. For verification on model-level, the topmost level of a model should be selected, whereas for verification on subsystem-level the topmost level of the subsystem should be selected. To implement the semantics of $\mathcal{L}$ in Simulink, we use a custom-build, modular and interchangeable block library and existing Simulink logic blocks.

The following requirement is used as a running example to illustrate the various steps:

Example 1. At each time step if [(((`signal_A` is equal to `TRUE`) and ((not `signal_B`) is equal to `constant_A`)) and ((the absolute value of `signal_C`) is greater than `constant_B`)) and (`signal_D` is less than `constant_C`)] has been valid for [50 ms], then in response, after a delay of [0 steps], [`signal_E` is equal to `TRUE`] is valid for [1 step].

To support the requirement specification for SLDV, we implemented a Simulink library with building blocks for all elements of our requirement specification language. The library provides sub-libraries for the specification of scopes, patterns and events.

Verification Subsystem. Figure 6 shows the topmost generated block, a verification subsystem. Its input are all input and output signals of the Simulink model that are used by the generated requirement specification. The content of verification subsystems is considered during formal verification but ignored during code generation and is not part of the generated code. The top-level verification subsystem contains a separate verification subsystem for each requirement.

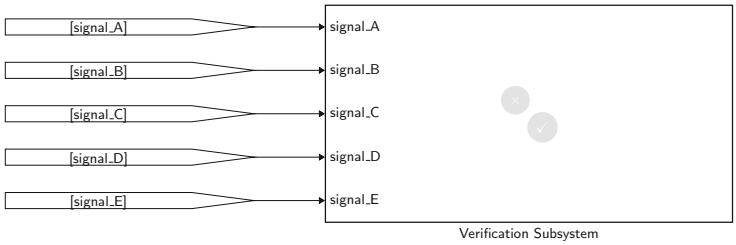


Fig. 6. A sample verification subsystem block.

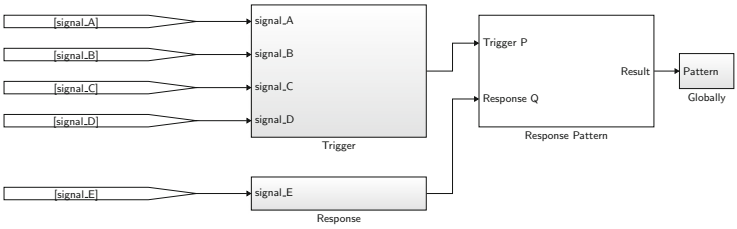


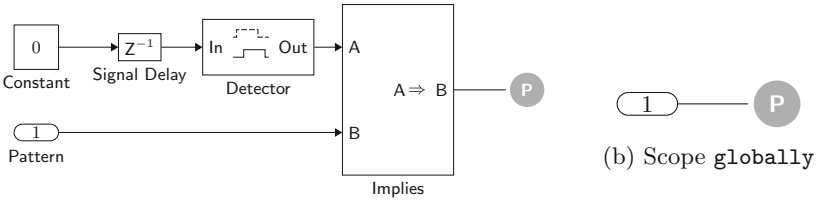
Fig. 7. The `responseTbEb` pattern of the verification subsystem in Fig. 6.

The verification subsystem subsumes the implementation of the actual requirements, i.e. encoding the expected functional behavior, by separating it into parts: Transformations on inputs, and implementing timed behavioral aspects. A requirement specification consists of three parts: a set of events, a pattern and a scope; each is represented by distinct blocks in the library. Figure 7 shows an example requirement specification that consists of a `globally` scope, a `response` pattern and two events.

Scopes. A scope block defines the time steps during which a pattern needs to be evaluated. The pattern result is a Boolean input parameter. At each simulation step, either the pattern result or `true` (if the pattern result needs not to be evaluated at the current time step) is the input of a proof objective. During formal verification, `SLDV` analyzes this proof objective. A requirement is violated if the input of a proof objective can be `false` at any simulation step.

The `initially` scope evaluates the pattern result only at system start, while the `globally` scope evaluates the pattern result at each time step. Figure 8(a) and (b) show the implementations of scopes `globally` and `initially`, respectively. The delay block is initialized with the value 1, while all subsequent output values will be 0. The time shift (see Sect. 3) is realized by the Detector block.

Patterns. A pattern receives the Boolean signals from the events as inputs along with the time duration and delays between events specified as mask parameters of the pattern. A pattern block ensures the correct order of events and handles timing aspects like event durations and delays between events. Simulink blocks



(a) Scope `initially` (b) Scope `globally`

Fig. 8. Proof objectives for the scopes `initially` and `globally`.

for time durations and delays are provided by our Simulink specification library. The output of a pattern block is again a Boolean signal. In Fig. 7, the blocks Trigger and Response contain the part of signal transformation, whereas the block labeled Response Pattern represents the details of the duration- and delay checks, as shown in Fig. 9. Inside this subsystem block, the event order (trigger before response) is established together with the specified time delay between the two events.

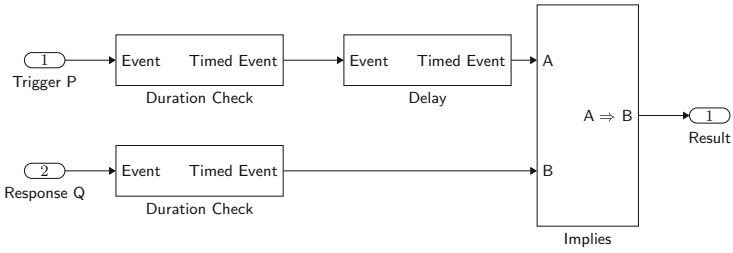


Fig. 9. The implementation of the Trigger/Response pattern.

In our example, the trigger has to be true for 50 ms. This duration is checked by the Duration Check block which returns a Boolean true iff its input evaluated to true for a given number of time steps. A delay block is then used to account for the response duration and a possible fixed delay between trigger and response.

Events. Each event is specified in its subsystem. The event subsystems are connected with the input signals of the verification subsystem using From blocks. An event is built using the blocks provided by our Simulink specification library. These building blocks must be connected in accordance with the rules of our event grammar. The output of an event specification is a Boolean signal. Figure 10 shows the necessary signal transformations for the trigger of the example requirement.

Connection to the Simulink Model. After the automated insertion of the verification subsystem at a user-chosen level in the model, the inputs of the

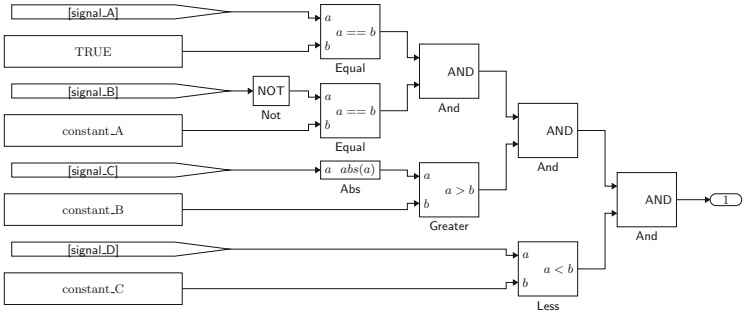


Fig. 10. The logic of the trigger condition of the example requirement.

verification system need to be connected to the corresponding signals in the model. Because of possible data dependency issues, we use global data store blocks for accessing the signals. For selecting the source signal, we traverse the model in a hierarchical approach and try to find the first match of a named signal matching the one being looked for. A data store write is then inserted into the model at the matched location, allowing us to generate the corresponding global data store read block next to our verification subsystem.

5.2 Export to `BTC EmbeddedPlatform`

We support the export of formalized requirements to `BTC`'s input format, so-called SPEC files. They contain an XML-based structured representation of the requirements and their patterns. Small transformations are applied during export to match `BTC`'s pattern semantics. We consider the time step 0 to be the first time step in the `initially` scope. This means that we start to evaluate the pattern directly after initialization, i.e. before the first computation step. In contrast, `BTC` starts the evaluation after the first computation step. It is not possible to check initial variable valuations in BTC, therefore, an error is presented when exporting a requirement with scope initially to BTC. The generated SPEC files can then be imported into `BTC EmbeddedPlatform` and used for verification.

5.3 Export to Textual Requirements

Formally specified requirements can easily be exported to textual form. As many engineers and stakeholders without a solid background in formal methods are involved in the design, testing and implementation of the defined software components, it is vital to present the agreed-upon requirements in a textual representation, which is easy to understand, distribute and review. Our export feature for textual requirements additionally supports automatically introducing parenthesis around all non-trivial arguments used in the specification to prohibit misinterpretations or misunderstandings of the written specification—a problem we encountered several times in [2,20].

5.4 Export to SV-Comp-style C Code

To enable the use of state-of-the-art academic C code model checkers, we explicitly encode our pattern semantics in C code. This enables to embed all assumptions and behavior directly in the code, instead of going around it with LTL specifications or similar, as supported by some tools. We built a boiler-plate framework for initializing parameters and calibratables (enabling verifying with varying calibrations) and updating input variables after every step. We decided to use the established `__VERIFIER_error();` functionality for encoding violations of the behavior allowed by the patterns as supported by many code verifiers such as more than 20 tools participating in the SV-comp.

6 Requirement-Based Test Vector Generation

The automated generation of an `SLDV` specification can be reused for automated requirement-based test vector generation. The Automotive Functional Safety standard ISO26262 [13] recommends to identify missing test vectors and unintended behavior of the implemented model by: "For each requirement, a set of test vectors should be generated. Afterwards, the structural coverage of the requirement-based test vectors shall be measured according to a suitable coverage metrics. The industry norm recommends different coverage metrics depending on the ASIL-level of the model. In case the coverage metrics reveals uncovered parts of the model, a further analysis is needed: either test vectors are missing or unintended functionality in the model has been detected".

If requirements are verified using formal verification and the implemented requirement is shown to be valid, additional, manual creation of test vectors should not be necessary. Manual creation of test vectors is a tedious work and should be limited to those requirements that are not tested using formal verification. We propose to reuse the automated generation of `SLDV` requirement specification for generating test vectors for these same requirements. For this purpose, we annotate the generated specification with so-called *test objectives* (see Fig. 11) automatically. The test objectives specify the signal valuations that must be considered during test-vector generation.

The set of requirement-based test vectors depends on the chosen coverage metric. For *condition coverage*, a set of test vectors is required such that each condition takes every possible value, while for *decision coverage* a set of test vectors must generate every possible outcome for each decision. Decision coverage is closely related to *branch coverage*, where conditional and unconditional branches are considered. According to ISO26262, branch coverage is suitable for requirement coverage at software unit-level for ASIL A to C. However, for ASIL D, *modified condition/decision coverage* (MC/DC) is highly recommended. Additionally, it is required that all conditions contributing to a decision must independently affect the outcome of the decision.

To achieve condition coverage, test objectives must be added to all Boolean input signals. For decision coverage, test objectives are needed for all Boolean output signals. If test objectives are added to all Boolean output signals and

to all Boolean input signals of blocks with more than one input parameter, *condition/decision coverage* is achieved, which guarantees both condition and decision coverage. For MC/DC coverage, test objectives are hard to generate and currently out of scope of our project. One way to at least partly cover MC/DC would be to generate test objectives for all Boolean combinations of possible input signals. For an OR block, we currently generate vectors for both outcomes, but "true" could be generated by inputs 01, 10 or 11—by adding additional logic we can enforce all combinations to be generated.

Alternatively, we propose to use the built-in function of `SLDV` to compute a set of test vectors for MC/DC coverage. Unfortunately, this functionality is currently only available on model-level. To get requirement-based test vectors for the model, MC/DC must be checked at requirement (i.e. subsystem) level while test vectors must be generated for the complete model.

To automate the requirement-based test vector generation, we added test objectives for *condition/decision coverage* to all blocks in our Simulink formal specification library. The relational operators compute Boolean output signals that also must be annotated with test objectives. Additional test objectives are necessary for all temporal operators to assure the correct length of generated test cases. Since we handle Boolean signals only, all test objectives can take the values `true` and `false`. Figure 11 presents the implementation of the annotated Boolean *Or* operator from our specification library.

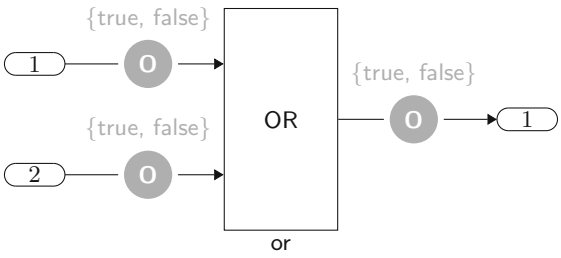


Fig. 11. A logic OR block with test objectives attached.

Annotating the specification library allows the flexibility of adding/removing test objectives without adapting the source code of the specification tool. This enables the user to maintain a set of specification libraries for different coverage metrics or to create a library without any test objective annotations.

7 Conclusion and Future Work

In this paper we presented a prototypical pattern-based specification tool together with automated translations to `SLDV` and `BTC EmbeddedPlatform` together with an adaption of the `SLDV` input for automated test-case generation. This corresponds to the vision of enabling engineers to specify requirements with formal

semantics *once* and then applying the requirements in *multiple analyses*. The tool was designed as a prototype for use inside this research project as a proof of concept.

Although a big step towards a highly automated automotive verification process has been made within this project and investigations by Ford have been producing encouraging results, this is only a proof-of-concept and many open problems still need to be resolved.

As future work we plan the extension of our pattern set with a few further relevant elements like time- and event-bounded response patterns. We plan to tackle the automated translation of textual legacy requirements into formal notation. Scripts are needed to further automate verification at different development levels with suitable configuration parameters, and to trigger the verification process if changes are applied to the model or the requirements. Another module should monitor the verification results and automatically report conspicuous behavior if the comparison with previous results reveals deviations. In case of invalid verification results, counterexamples should be analyzed.

We plan to use the export of formalized requirements to SV-COMP like C-code patterns in order to benchmark academic C-code model checkers on industrial examples against commercial tools.

A Appendix

A version of this paper containing the full appendix can be found at http://arxiv.org/abs/1906.07083.

References

1. 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). https://doi.org/10.1109/TSE.2015.2398877
2. Berger, P., Katoen, J.-P., Ábrahám, E., Waez, M.T.B., Rambow, T.: Verifying auto-generated C code from simulink. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 312–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_18
3. 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
4. Bianculli, D., Ghezzi, C., Pautasso, C., Senti, P.: Specification patterns from research to industry: a case study in service-based applications. In: Proceedings of ICSE, pp. 968–976. IEEE (2012). https://doi.org/10.1109/ICSE.2012.6227125
5. Botham, J., et al.: PICASSOS - Practical applications of automated formal methods to safety related automotive systems. In: SAE Technical Paper. SAE International (2017). https://doi.org/10.4271/2017-01-0063
6. Bozzano, M., Cimatti, A., Katoen, J., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2011). https://doi.org/10.1093/comjnl/bxq024

7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of ICSE, pp. 411–420. ACM (1999). https://doi.org/10.1145/302405.302672
8. Filipovikj, P., Jagerfield, T., Nyberg, M., Rodriguez-Navas, G., Seceleanu, C.: Integrating pattern-based formal requirements specification in an industrial tool-chain. In: Proceedings of COMPSAC, pp. 167–173. IEEE (2016). https://doi.org/10.1109/COMPSAC.2016.140
9. Filipovikj, P., Nyberg, M., Rodriguez-Navas, G.: Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In: Proceedings of RE, pp. 444–450. IEEE (2014). https://doi.org/10.1109/RE.2014.6912296
10. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proceedings of ICSE, pp. 31–40. ACM (2008). https://doi.org/10.1145/1368088.1368094
11. Guglielmo, L.D., Fummi, F., Orlandi, N., Pravadelli, G.: DDPSL: an easy way of defining properties. In: Proceedings of ICCD, pp. 468–473. IEEE (2010). https://doi.org/10.1109/ICCD.2010.5647654
12. 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
13. ISO Central Secretary: Road vehicles - Functional safety. Standard ISO 26262–1:2011. International Organization for Standardization, Geneva, CH (2011). https://www.iso.org/standard/62711.html
14. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of ICSE, pp. 372–381. ACM (2005). https://doi.org/10.1145/1062455.1062526
15. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
16. Liu, S., Wang, X., Miao, W.: Supporting requirements analysis using pattern-based formal specification construction. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 100–115. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_7
17. Lumpe, M., Meedeniya, I., Grunske, L.: PSPWizard: machine-assisted definition of temporal logical properties with specification patterns. In: Proceedings of SIGSOFT/FSE, pp. 468–471. ACM (2011). https://doi.org/10.1145/2025113.2025193
18. Mahmud, N., Seceleanu, C., Ljungkrantz, O.: Resa tool: structured requirements specification and sat-based consistency-checking. In: FedCSIS, pp. 1737–1746 (2016)
19. Moitra, A., et al.: Towards development of complete and conflict-free requirements. In: RE, pp. 286–296. IEEE Computer Society (2018)
20. 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). https://doi.org/10.1007/978-3-319-95582-7_23
21. Remenska, D., Willemse, T.A.C., Templon, J., Verstoep, K., Bal, H.: Property specification made easy: harnessing the power of model checking in UML designs. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 17–32. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43613-4_2
22. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: PROPEL: an approach supporting property elucidation. In: Proceedings of ICSE, pp. 11–21. IEEE (2002). https://doi.org/10.1109/ICSE.2002.1007952

23. Teige, T., Bienmüller, T., Holberg, H.J.: Universal pattern - Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In: Proceedings of MBMV (2016)
24. Wong, P.Y.H., Gibbons, J.: Property specifications for workflow modelling. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 56–71. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00255-7_5