


SMArDT modeling for automotive software testing

Imke Drave¹ | Steffen Hillemacher¹ | Timo Greifenberg¹ | Stefan Kriebel^{1,2} |
 Evgeny Kusmenko¹ | Matthias Markthaler^{1,2}  | Philipp Orth³ |
 Karin Samira Salman¹ | Johannes Richenhagen³ | Bernhard Rumpel¹ |
 Christoph Schulze¹ | Michael von Wenckstern¹ | Andreas Wortmann¹

¹Software Engineering, RWTH Aachen University, Aachen, Germany

²Development Electric Drive, BMW Group, Munich, Germany

³FEV Europe GmbH, Aachen, Germany

Correspondence

Matthias Markthaler, Department of Development Electric Drive, BMW Group, 80809 Munich, Germany.
 Email: matthias.markthaler@bmw.de

Summary

Efficient testing is a crucial prerequisite to engineer reliable automotive software successfully. However, manually deriving test cases from ambiguous textual requirements is costly and error-prone. Model-based software engineering captures requirements in structured, comprehensible, and formal models, which enables early consistency checking and verification. Moreover, these models serve as an indispensable basis for automated test case derivation. To facilitate automated test case derivation for automotive software engineering, we conducted a survey with testing experts of the BMW Group and conceived a method to extend the BMW Group's specification method for requirements, design, and test methodology by model-based test case derivation. Our method is realized for a variant of systems modeling language activity diagrams tailored toward testing automotive software and a model transformation to derive executable test cases. Hereby, we can address many of the surveyed practitioners' challenges and ultimately facilitate quality assurance for automotive software.

KEYWORDS

automotive software engineering, model-based testing, test case creation

1 | MOTIVATION

The focus of automotive engineering has shifted from mechanical and electrical engineering toward software engineering by the notion of automotive interacting connectivity, autonomous driving, and gesture human machine interaction (cf Figure 1). Current vehicles are already sophisticated systems comprising many interacting software modules to provide system-wide functionalities and features. Considering the current notion of automotive development toward autonomous driving and CAR2X communication,² complexity is increasing and the demand for more flexible, yet robust, interaction between systems in an open environment arises.

Additionally, customers demand a shorter time-to-market and even more functionality. While, in the 1980s, the automotive development cycle took about 5 years, it has already been reduced to three years in the 21st century,³ not to mention the large number of innovations introduced during that period. Considering inflation, the rising customer expectations call for a cost reduction of 4% per year.⁴ Consequently, cost and time reduction while mastering increasing system complexity at the same time became a prime concern in automotive software engineering.

Nowadays, automotive agile development processes are often based on the V-Model,⁵ which structures requirements and specifications of a system in different abstraction layers. Each layer is associated by corresponding quality

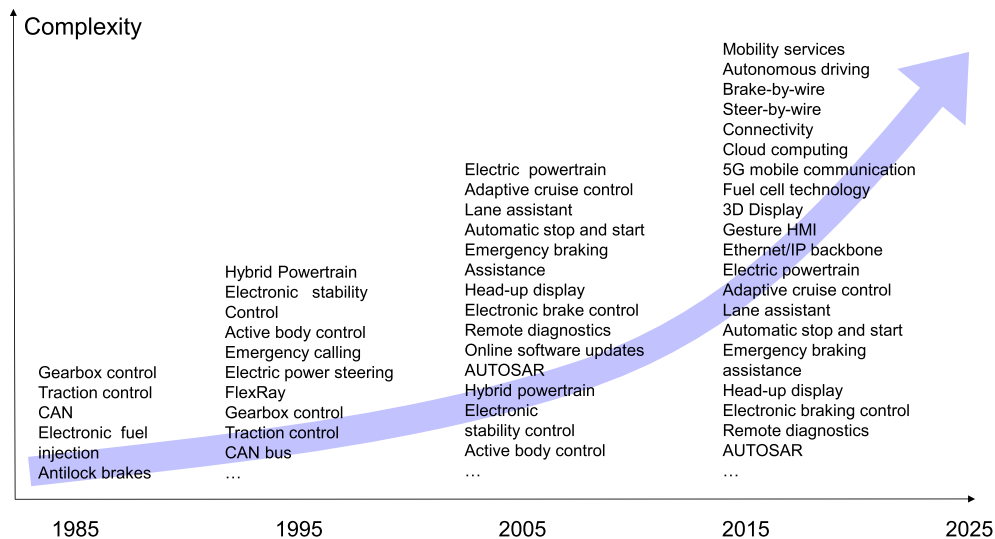


FIGURE 1 Complexity drivers in automotive engineering move from mechanical and electrical engineering to software engineering.¹ 5G, fifth generation; CAN, controller area network [Colour figure can be viewed at wileyonlinelibrary.com]

assurance tasks, eg, a set of tests. These tests are usually created manually, which holds several disadvantages and raises the impression of a *bureaucracy in software development*.

- Links between tests and specifications are vague; thus, consistency between tests, requirements and specifications is difficult to ensure. This prevents test automation, and thus hampers productivity.
- Once a specification is updated or added, it is necessary to update all of the corresponding handwritten tests manually.
- Due to the large expenses and time pressure, a system's functionality is often extended on the lowest abstraction layer only. Engineers therefore do not adopt requirements and specifications on higher abstraction layers in general and thereby introduce possible inconsistencies between requirements, models, and the implementation.

These deficiencies contribute to long and expensive development cycles. New concepts and methods are required to fulfill current customer demands and to enable the development of modern, safe and affordable mobility solutions.

Other engineering domains, such as avionics⁶ and robotics,⁷ master similar challenges by leveraging model-based software engineering^{8–10} together with model-based testing (MBT).^{11–13} The aim is to reduce the conceptual gap between the problem (eg, autonomous driving) and the solution domain (eg, software engineering) by means of abstract, domain-specific, and sustainable models as primary software artifacts. For automotive software engineering, modeling techniques following these paradigms need to be tailored to the specific requirements regarding safety and process standard compliance, while retaining agile efficiency.¹⁴

We developed a methodology to manage requirements, design, and test in the automotive industry to overcome present challenges in automotive software development. The *specification method for requirements, design, and test* (SMArDT)* targets the deficiencies of the established V-Model. The SMArDT relies on model-based software engineering techniques, providing model artifacts on each abstraction layer. Consistency checking between layers and automatic generation and regeneration of test cases reduces bureaucracy imposed by the classical V-Model. We present two central solutions for these challenges within SMArDT, based on experiences conceived by its application at the BMW Group.

The automated consistency assurance concept was developed along with SMArDT and is based on view verification for each abstraction layer. The abstract model-based system artifacts enable us to develop a general and abstract verification concept to obtain meaningful links between requirements, specifications and tests. Hence, with roughly the less effort than with manual test derivation, developers can take advantages of automated test derivation to improve test derivation efficiency, quality, and consistency of results. Especially consistency checking, ie, disallowing the many individual interpretations of test developers that would arise by manual test implementation, is a big advantage of automated test derivation.

*The abbreviation SMArDT is related to the German term “Spezifikations-Methode für Anforderung, Design und Test” (specification method for requirements, design, and test)

Since verification and validation are important for reliable automotive products, testing is a well-known part of the automotive industry. A new concept should recognize existing knowledge of testing strategies and provide assistance to test engineers to tackle inefficiencies in the test creation.

To develop our functional testing strategy, we conducted a systematic survey among test experts at the BMW Group to identify deficiencies of the current testing concepts and opinions on possible efficiency improvements along with the introduction of SMArDT. Based on our findings, we developed a method for systematic test case creation, which enables test experts to adapt test cases of each abstraction layer to specification novelties or changes efficiently. Our methodology also enables a close collaboration between function and test specifiers working on the same artifact, which guards against erroneous function or test specifications at an early stage of the development. The implementation of our method is based on automatic model analysis and model transformation supported by the MontiCore language workbench.¹⁵

In the following, we present the notion of SMArDT in Section 2 and a view verification for consistency assurance between the abstraction layers in Section 3. We present details of functional testing within SMArDT by presenting our survey in Section 4 and the derived extended method for systematic test case creation in Section 5. In Section 6, we present the MontiCore based implementation for automatic test case generation. Our contribution is discussed in Section 7 and related to existing approaches in Section 8. We conclude in Section 9.

2 | PRELIMINARIES

The SMArDT¹⁶ tackles the issues of the V-Model mentioned in Section 1. The method enforces function specification by means of a subset of *systems modeling language* (SysML) diagrams with unambiguous semantics. Thereby, it facilitates higher consistencies between abstraction layers by construction, which is most valuable to the iterative, incremental, and evolutionary agile development projects in automotive engineering. Building upon formal semantics,¹⁷ the diagrams enable further systematic validations such as (1) backward compatibility checking^{18–21} for software maintenance and evolution between different diagram versions of the same layer as well as (2) refinement checking^{22,23} between diagrams of different layers for the detection of interlayer specification inconsistencies. In turn, consistent requirements and models enable a stable generative development process, which facilitates systematic test case creation and guarantees that tests are meaningful and synchronized with the overall system.

Four abstraction layers structure the SMArDT process as illustrated in Figure 2. In particular, they focus on requirements, design, and testing of system engineering artifacts according to the ISO 26262 specifications.²⁴

1. Layer 1 contains a first description of the object under consideration and shows its boundaries from a customer's point of view. This layer elevates requirements and use cases (UCs) for each functionality. The modeling artifacts of this layer are UC diagrams and requirements.
2. Layer 2 specifies the abstract functional specifications without providing implementation specifics by activity diagram (AD), state charts (SCs), sequence diagrams, and internal block diagrams (IBDs).
3. Layer 3 details the abstract description of Layer 2 by adding implementation specifics. The layer comprises refinements of the Layer 2 artifacts and complements them by more detailed internal block diagrams.
4. Layer 4 represents the software and hardware artifacts according to the specification of Layer 3. This layer contains the actual source code.

The SMArDT enables structural verification²⁵ between each layer by formal SysML semantic modeling.¹⁷ A view-based technique and its implementation covering this aspect are described in Section 3. Since SMArDT artifacts are elements of a formalized subset of consistent SysML diagrams,²⁶ a systematic and automated generation of functional test cases for each layer is possible. Automatic generation and transformation for application on all lower layers assure consistency between test cases of each layer. This is illustrated on the right side of Figure 2. For example, Layer 1 describes a functionality of the product on the highest level. Hence, the corresponding test cases cannot be used directly on the lower layers, and therefore must be transformed to multiple low-level test cases.²⁷ This can be achieved by substituting abstract signal names and placeholders with concrete hardware signals and values.

SMArDT Layers 1 and 2 are conceptual in the sense that their diagrams cannot be mapped directly to counterparts in the final implementation. The aspects modeled by these diagrams are usually scattered across several implementation artifacts later on. Signals used in these diagrams are logical, ie, they abstract from signals of the implementation. Consequently, logical signal values encapsulate a range of values present in the implementation. In contrast, the

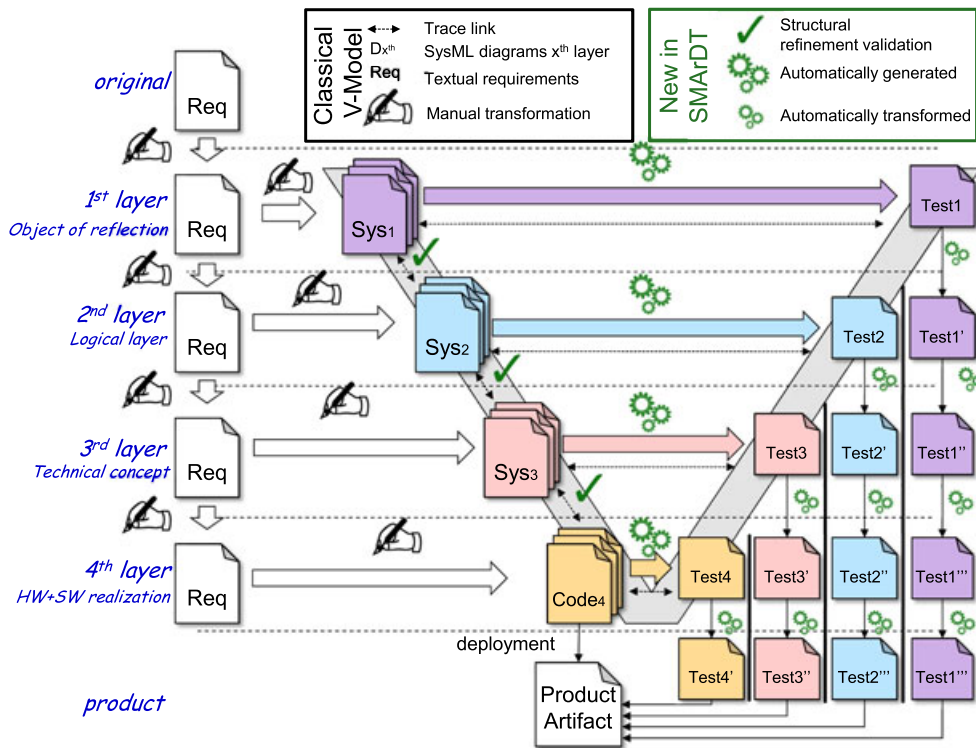


FIGURE 2 Overview of the specification method for requirements, design, and test (SMArDT) methodology. SysML, systems modeling language¹⁶ [Colour figure can be viewed at wileyonlinelibrary.com]

elements of the third and fourth layers have a direct representation within the implementation. The third layer describes platform-independent functionality of a system, whereas the fourth layer contains software and hardware parts specific to a given target platform and handles low-level behavior such as I/O interrupts.

3 | MODEL CONSISTENCY IN SMArDT

The SMArDT development process tackles the deficiencies of the classical V-Model. An important aspect is the consistency between artifacts on different abstraction layers, which allows efficient adaptation of all artifacts to changes or novelties in requirements or specifications on any layer. Consistency between specification artifacts between layers enables automatic transformation of tests to lower levels. Thereby, SMArDT supports evolutionary development in a unique way.

We illustrate how automated requirement consistency verification is realized within the SMArDT methodology using real world requirements employed for specifications of a publicly available adaptive light system (ALS).^{25,28} The example ALS model controls adaptive high and low beam turn signals as well as cornering and ambient light. Adaptive high and low beam adjust headlamps to the traffic conditions and ensure an optimal illumination experience without dazzling other road users. Cornering lights illuminate the area to the side of a vehicle to look around the bend. Finally, ambient lights welcome the driver with a subtle illumination. The complete collection of the 93 original ALS requirements can be found on our support website.[†]

3.1 | Architecture specification using views

Automotive software systems are often created using component and connector (C&C) languages such as Simulink²⁹ describing functional, logical, or software architectures³⁰ in terms of components executing computations and connectors realizing component interaction and data flows via directed and typed ports. A particular strength of the C&C paradigm

[†]<http://www.se-rwth.de/materials/cncviewscasestudy/>

is the ability to decompose complex components such as the ALS hierarchically. By following the divide and conquer principle, each simple component can be developed by a different team. *Systems modeling language*³¹ and *Modelica*³² are two other popular representatives for C&C modeling languages. As SMArDT relies heavily on model-based design, it is of great importance to be able to handle big model repositories, to compose models developed by different teams and to guarantee their consistency and compatibility throughout the whole development cycle. In this context, the notion of C&C views plays a major role and is therefore indispensable for SMArDT. The C&C views³³ equip C&C developers with a technique enabling them to focus on specific aspects of large C&C models without having to understand the rest of the system unimportant for the viewpoint of interest. For example, a view can be constrained to show a component's decomposition into subcomponents without revealing their respective ports and connections. Hence, the main goal of the C&C views concept is to provide many small, precise, and comprehensible viewpoints of a large C&C model that can be used for specification and validation purposes in SMArDT. Therefore, C&C views introduce four major abstractions: hierarchy, connectivity, data flow, and interfaces of C&C models. The hierarchy of a system's components in C&C views is not required to contain all levels as intermediate components may be skipped; abstract connectors can cross-cut component boundaries and also connect components directly, ie, omitting the interfacing port. Hence, it is allowed to hide interface parts such as port names, types, and array sizes. Abstract effectors describe data flows which abstract from chains of components and connectors.

Intuitively, a C&C model satisfies a C&C view if and only if all elements and relations revealed by the abstract view have a satisfying counterpart in the concrete model. The formal definitions of C&C models, C&C views, and their satisfaction are available in the work of Maoz et al³³ and on the supporting materials on the complementary website.²⁵

Component and connector view verification and witnesses for tracing in SMArDT

Figure 3 shows an example C&C model of an ALS. The modeled functionality controls the flash LEDs for turning as well as the left and right light bulbs for cornering and ambient lights. The ALS consists of the two subcomponents *Flashing* and *HeadLight*. The *HeadLight* component is further decomposed into the subcomponents *CorneringLight*, and *AmbientLight*. The C&C view *ALS1* shown in Figure 4 describes the *CorneringLight* component illuminating the road the driver wants to turn. Thus, the input port *BlinkerLever* of the ALS component has an effect on at least one input port of the *CorneringLight* component, which is modeled by an abstract effector. The computed light values *CLeft* and *CRight* of the *CorneringLight* component are passed directly and without further modification to the output ports of the ALS component modeled by two abstract connectors. The C&C view *ALS2* grasps the relationship between the *Flashing* and the *AmbientLight* component. It specifies that the *FlLeft* (flashing left) output of the

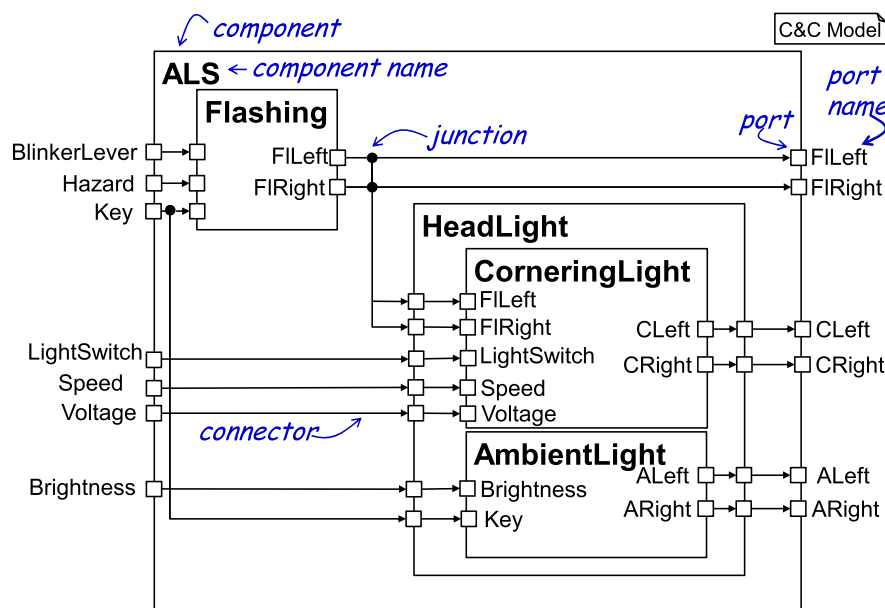


FIGURE 3 Very simple example component and connector (C&C) model for adaptive light system (ALS)²⁸ [Colour figure can be viewed at wileyonlinelibrary.com]

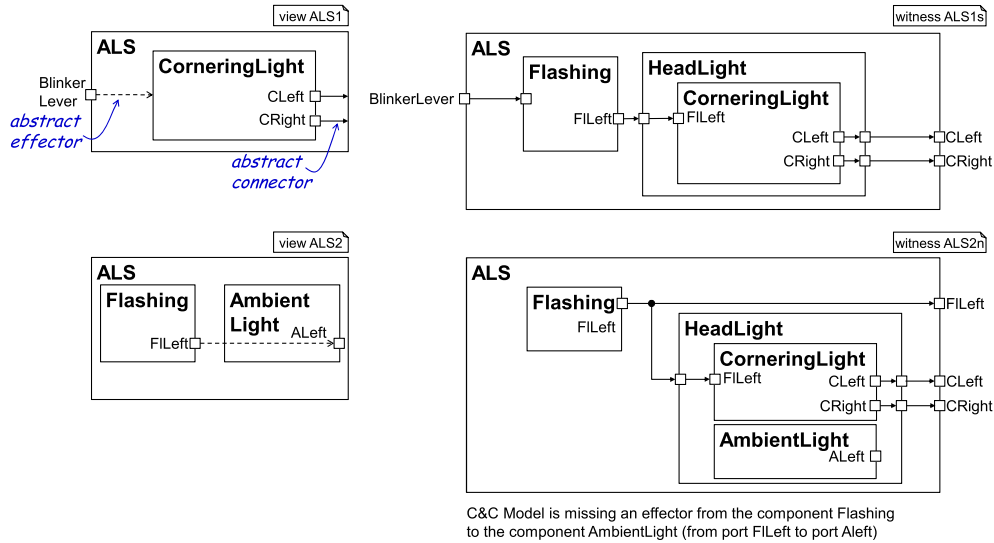


FIGURE 4 Two component and connector (C&C) views *ALS1* (top left) and *ALS2* (bottom left). Witness for satisfaction *ALS1s* (top right) and witness for nonsatisfaction *ALS2n* (bottom right).²⁸ ALS, adaptive light system [Colour figure can be viewed at wileyonlinelibrary.com]

component *Flashing* affects the ambient left light (port *ALeft* of component *AmbientLight*), eg, there is a brighter left ambient light when the left parking mode is activated. The model *ALS* satisfies the view *ALS1*.

The C&C view verification³⁴ receives a C&C model and a C&C view as inputs. The algorithm returns a Boolean, which indicates if the C&C model satisfies the C&C view. Additionally it calculates a local minimal satisfaction, or local nonsatisfaction witnesses, which emphasize improper model elements. For example, *ALS1s*, a witness for satisfaction, is shown in Figure 4. It demonstrates how the C&C model satisfies *ALS1*. The SMArDT approach employs the generated witnesses to automatically generate traceability information between SysML artifacts of Layer 2, often provided as C&C views against Layer 3 model designed as concrete C&C SysML diagrams. The witness itself is a well-formed model. It contains all components of the view as well as their parent components to reveal the entire hierarchy between the two components specified in the view. Therefore, the witness also includes the *HeadLight* component. Furthermore, the *positive satisfaction witness* incorporates all ports corresponding to a view. Consequently, it embraces the *BlinkerLever* port of *ALS* as well as the *CLeft* and *CRight* ports of *CorneringLight*. Additionally, the witness contains all model connectors and all data flow paths. The abstract connector from *CorneringLight* (port *CLeft*) to *ALS* (port *unknown*) introduces the following elements in the witness: (1) port *CLeft* of component *HeadLight*, (2) connector of ports *CLeft* from component *CorneringLight* to component *HeadLight*, and (3) connector of ports *CLeft* from component *HeadLight* to component *ALS*. For the abstract effector from *ALS* (port *BlinkerLever*) to *CorneringLight* the following elements in the chain serve as witness: (1) component *Flashing*, (2) ports *BlinkerLever* and *FLeft* of *Flashing*, (3) connector of ports *BlinkerLever* from *ALS* to *Flashing*, (4) connector of ports *FLeft* from *Flashing* to *HeadLight*, and (5) connector of ports *FLeft* from *HeadLight* to *CorneringLight*.

The model *ALS* does not satisfy the view *ALS2*. Every *negative nonsatisfaction witness* contains a minimal subset of the C&C model and a natural-language text, which explains the reason for nonsatisfaction. These witnesses are divided into five categories, namely, *MissingComponent*, *HierarchyMismatch*, *InterfaceMismatch*, *MissingConnection*, and *MissingEffector*.³⁴ A witness for nonsatisfaction *ALS2n*, namely, a *MissingEffector*, is illustrated in Figure 4. It depicts all outgoing connector/effector chains starting at port *FLeft* of component *Flashing* as well as the abstract effector's target port, *AmbientLight*'s *ALeft*, which is not reachable. It is sufficient to remove the effectors in *ALS2* to make the model satisfy the view even though *Flashing* and *AmbientLight* are direct siblings in the C&C view but not in the C&C model since C&C views allow to abstract away the intermediate component *HeadLight*.

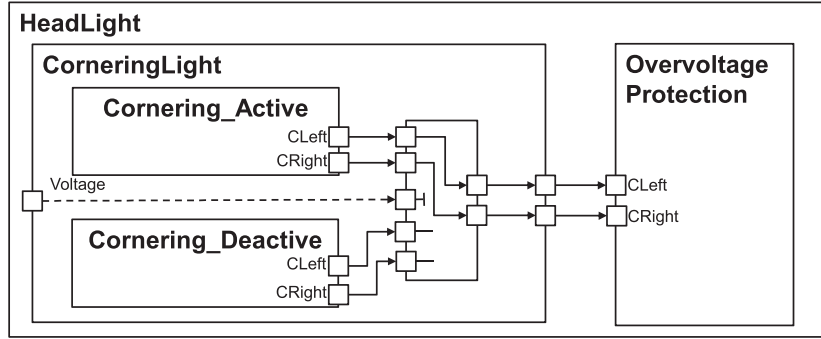
Identifying design inconsistencies with C&C views

The previous paragraphs illustrated how C&C view verification can be used to ensure structural consistency in the SMArDT process and how to generate tracing witnesses between view models of Layer 2 and concrete logical C&C models of Layer 3. In the following, we focus on structural design inconsistencies between different artifact models of the same layer. Figure 5 shows two requirements concerning the *cornering light*. Since the last requirement **AL-139** is a safety

AL-122: With subvoltage the cornering light is not available.

req AL-122

view AL-122



AL-139: With activated darkness switch (only armored vehicles) the cornering light is not activated.

req AL-139

view AL-139

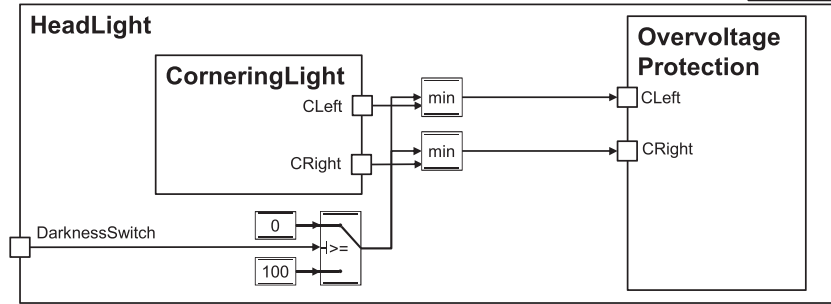


FIGURE 5 Design inconsistency of two component and connector views²⁸

feature for armored vehicles, a dedicated team of safety experts is responsible for it. Hence, two different teams develop the architectural designs (view AL-122 and view AL-139) for the two requirements.

The top design shows the CorneringLight with two modes (subcomponents Cornering_Active, Cornering_Deactive, and MultiSwitch). The mode Cornering_Deactive is activated if the voltage level is low otherwise Cornering_Active is used. In the bottom design model, the min block, in combination with the Switch, deactivate the cornering light by propagating 0% as light value to OvervoltageProtection input ports when the DarknessSwitch port has the value true. The C&C view synthesis³³ is the process of deriving a valid C&C model conforming to all given C&C views. The first part of the synthesis algorithm checks whether views contain design contradictions. If all views as in our example are positive views, ie, only declaring what should be fulfilled but not containing restrictions such as “*component A should not contain port B*,” the contradiction check can be done in polynomial time and scales easily to hundreds of views.²²

The contradiction check for the two views view AL-122 and view AL-139 would result in an error. In the top view, the port CLeft of component CorneringLight is directly connected to the component OvervoltageProtection and no further modification of the passed value takes place. In contrast, in the bottom view, the value from CorneringLight's CLeft is manipulated by the min component before it goes to the OvervoltageProtection's CLeft port. Similar to the C&C view verification process presented, the contradiction algorithm generates an intuitive witness to highlight incompatible parts of two views. The formalized C&C view verification problem, together with the derived contradiction problem, enables early analysis of structural design models in the SMArDT process to detect inconsistencies between different artifacts created by different developers or teams at early development stages. In this way, expensive integration problems occurring at later development stages due to incompatible software modules developed on inconsistent designs can be avoided very efficiently.

3.2 | Formalized ADs

Combining formalized ADs³⁵ with Object Constraint Language (OCL) constraints is another powerful modeling technique, which can be employed for test case creation¹⁶ and is therefore an integral part of the SMArDT modeling tool set.

In the following, we explain how formalized ADs can help finding inconsistencies in requirement specifications along the SMArDT process to generate better test cases and to enhance the overall software quality. The ADs in Figures 6 and 7 model the steering column stalk and the hazard lights behavior as required by *AL-40* and *AL-41*,²⁵ respectively. In Figure 6, the model obtains the state of the steering column stalk through the corresponding input port. In case the steering column stalk is turned to left, the left lights of the car blink with an on/off ratio of 1:1; we refer to this mode as *BLINKING1_1*. This value is written to the output port of the AD. The *else* branch is underspecified, ie, nothing is said about the output for the cases if the steering column stalk is set to right or to straight. Note that this is in accordance with the underlying requirement.

In the AD of Figure 7, the state of the hazard light switch is checked to decide whether the lights need to be turned on or not. However, to determine the concrete blinking mode, the position of the ignition key is required. If the key is inserted, the 1:1 blinking mode is activated. Otherwise, the 1:2 mode is used to reduce energy consumption. The *else* branch is underspecified once again. Note that, according to the requirement, the same physical lights are used for hazard blinking as for direction indication. This leads to a contradiction that becomes obvious in the two ADs, ie, if the key is not inside the lock and the hazard light switch is pressed, according to **AL-41**, the vehicle's indicators lamps should blink with a ratio of 1:2. If, at the same time, the steering column stalk is set to *blinking (left)*, according to **AL-40**, the blinking ratio of the left direction indicators should become 1:1.

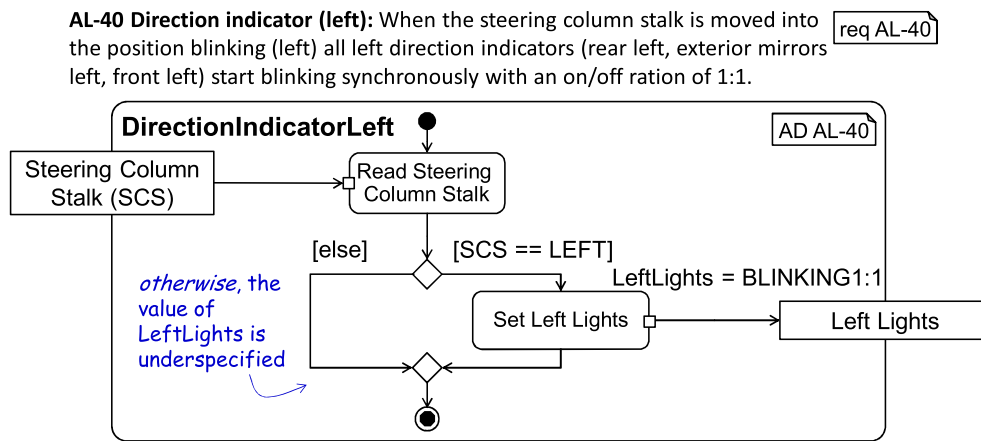


FIGURE 6 Activity diagram for AL-40 describing the steering column stick behavior [Colour figure can be viewed at wileyonlinelibrary.com]

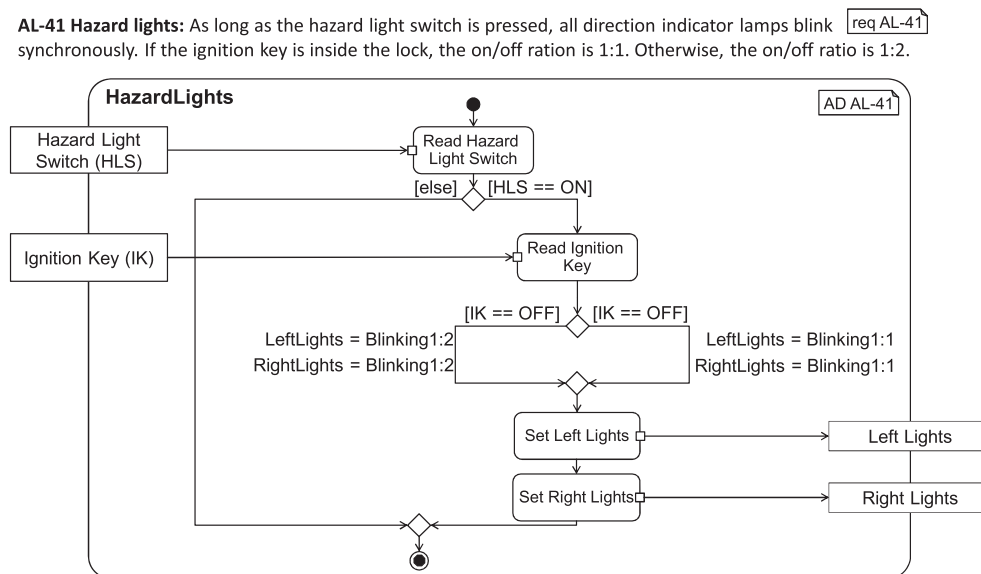


FIGURE 7 Activity diagram for AL-41 describing the emergency lights behavior²⁸

Keeping in mind that a faster blinking drains the battery in a shorter amount of time, this specification error might even become critical for human lives in cases of emergency. The error is discovered automatically with ease by creating a table mapping the three input signal combinations. For Figure 6, we would obtain one single combination, namely, (SCS = LEFT, HLS = DC, IK = DC) → LeftLights = BLINKING1_1, RightLights = DC, where DC stands for “do not care.” Figure 7 produces two combinations one of which is (SCS = DC, HLS = ON, IK = OFF) → (LeftLights = BLINKING1_2, RightLights = BLINKING1_2). Expanding the DC fields to all possible values reveals that the specification requires two contradictory outputs for the same input. Without this consistency check, degenerate test cases might be extracted, eg, accepting any output or requiring two different outputs at the same time.

4 | THE STATE OF TESTING IN AUTOMOTIVE SOFTWARE ENGINEERING

The SMArDT aims at managing requirements, design, and testing by means of model-based software engineering (cf Figure 2) more efficiently than the classic V-Model. The SMArDT methodology strongly relies on consistency verification as well as systematic test creation to decrease the effort of adapting all development artifacts to changing specifications and requirements. As discussed in the previous sections, this is supported by formal models such as C&C views and ADs. As software testing is a well-known field and test engineers already conduct certain strategies, we aimed at utilizing their knowledge to develop a systematic testing concept for SMArDT. Therefore, we conducted a systematic survey on model-based test case creation (MBTCC) at the BMW Group. Aggregated survey results were initially published in 2018.³⁶ This section describes our research questions, the survey design, its findings, and the conclusions in detail.

4.1 | Research questions

Since imposing model-based technologies on engineers top-down has often proven difficult, we investigate possible beneficiaries of MBT at the BMW Group.³⁶ Hence, we survey the target group (test engineers) about MBT and MBTCC. The data allowed to draw conclusions which enabled us to develop an MBTCC method that suits the test engineer's needs. The survey aims at analyzing whether MBT is suitable to improve the quality of testing and to identify application environments. Targets of the quality investigation are overall test coverage and usability of individual test cases. The question whether MBT test cases are suitable to be used in test beds and can be applied to test subsystems or the entire system is also addressed.

Figure 8 illustrates the traditional textual test case creation as applied at the BMW Group and compares it to the concept of MBTCC test case creation. Traditional test case creation is based on natural language requirements, which are used to develop a *test case concept* (Figure 8), ie, the basic test steps. Subsequently, the test engineers implement *executable test cases* derived from the test case concept. Using MBTCC, the intermediate manual translation becomes obsolete due to automatic processing of model-based requirements. The model-based input requirements for MBTCC allow an automatic derivation of the test concept. However, test engineers are still intended to be able to configure certain test case aspects manually.

Usually, test engineers examine the requirements and deploy a *test case concept* (cf Figure 8). This test case concept comprises basic test steps. Based on the concept, the test engineer develops the *automated test case*. Hence, the main research questions of our survey are as follows.

- R1** What is the background of the test engineers? Who can benefit from a (semi)automatic MBTCC?
- R2** Is it possible to enhance the test quality and the test coverage by MBT (from the test engineers' point of view)?
- R3** Which environments are the most promising for MBT in the BMW Group context?

The survey's main purpose is to reveal the potential of MBTCC in contrast to the predominant textual requirements-based test case creation as usually practiced in automotive software engineering. Regarding **R1**, the

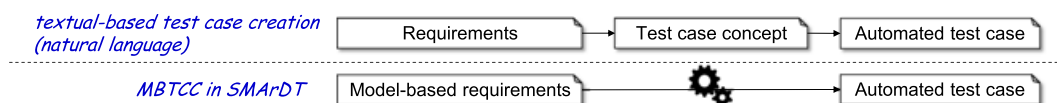


FIGURE 8 Comparison of traditional test case creation method versus model-based test case creation (MBTCC) in specification method for requirements, design, and test (SMArDT) [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Survey questions on background of the participants and main test case artifacts linked to the request questions

ID	Question	Question type	Linked to
Q1	What is your main function?	Multiple Choice	R1
Q2	What did you study?	Multiple Choice	R1
Q3	For how long have you been working in the automotive industry?	Time	R1
Q4	For how long have you been working in the sector of test case creation?	Time	R1
Q5	Which is your focus of testing?	Multiple Choice	R1, R3
Q6	Please indicate the importance of the following resources:	Likert-rating	R2
Q7	Please indicate the input for your test case creation:	Multiple Choice	R1, R2
Q8	Please indicate your test case creation method:	Multiple Choice	R1, R2
Q8a	Please explain your method:	Text	R1, R2
Q9	Please indicate the output of your test case creation:	Multiple Choice	R1, R2
Q9a	Please explain your answer:	Text	R1, R2
Q10	What is the most time-consuming test activity?	Text	R2
Q11	Are the test cases performed automatically?	Multiple Choice	R2
Q11a	In which environment are your test cases performed automatically?	Multiple Choice	R3
Q11b	If not performed automatically - What are the reasons?	Text	R2, R3
Q11c	If not performed automatically - How could test cases be performed automatically?	Text	R2, R3

questionnaire investigates the participants' main function, working background, years in the automotive industry, years of experiences in test case creation, the focus of testing, and the attitude toward MBT. We draw conclusions about the participant's background based on a profile obtained from this information. Moreover, we inquire about artifact sources, starting point for test case creation, degree of automation, test environment, ie, test bed, subsystem or the complete vehicle, purpose of testing, test case creation method, current, and possible estimated test case coverage to elucidate **R2** and **R3**.

4.2 | Study design and conduction

The anonymous survey comprised 27 questions (13 closed questions and 14 open questions) divided into three sections. To clarify the background information, eg, professional education, the first section inquires five background questions (Table 1).

The questions Q1, Q2, Q3, and Q4 investigate the participant's individual background and years of experience (**R1**). Question Q5 investigates the focus of testing to understand the participant point of view (**R1**) and Q5 investigates about the targeted system, ie, software, component, subsystem, or system (vehicle) (**R3**). The second section of the survey investigates the main test case artifacts and proceedings (Table 1).

To compare the different test case creation procedures, the survey investigates the procedure steps and content of the artifacts (Q6, Q7, Q8, Q9, Q11, Q11b, and Q11c). These data enable a detailed comparison of the participant indications and MBTCC (cf Figure 8). Hence, the comparison facilitates an assertion of the benefits from MBTCC (**R1**) and the quality (**R2**). Furthermore, the questions Q11b and Q11b concern test case automation, possible improvements for artifacts (**R2**), and environments (**R3**). In the third section, the survey investigates additional topics that could not be directly associated with the background and the main test case artifacts and procedures (Table 2).

We contacted 196 professionals, test engineers, and test managers of the BMW Group. To decrease the probability that the results are a chance encounter, we calculated with a statistically significant level of 5%. According to this level, we determined a required sample size of 65 out of 196 contacted test engineers for a statistically significant result. Based on the sample size, we are 95% confident that the proportion of all invited test engineers, with an error margin of 10% (approximately 20 participating test engineers) corresponds to the ascertained survey findings. Overall, we received 70 answers, with 69 valid questionnaires. The invalid questionnaire is without any content.

4.3 | Results

The survey provides multiple choice or Likert-rating scales as response possibilities. An external survey department of the BMW Group implemented and hosted the survey, which was conducted between August 23 and September 22, 2017. Exact survey results, however, cannot be published due to confidentiality.

TABLE 2 Survey questions of the third section linked to the request questions

ID	Question	Question type	Linked to
Q12	Please estimate the current test coverage:	Number	R2
Q13	How could you improve the current test coverage?	Text	R2
Q14	Please estimate the possible test coverage:	Number	R2
Q14a	Which resources do you need to improve the test coverage?	Text	R2
Q15	What is your focus on verification and validation?	Likert-rating	R2
Q16	What is a valid test case?	Text	R2
Q17	What is the frequency for delivering your test cases?	Multiple Choice	R1
Q18	What is the frequency for testing of test cases?	Multiple Choice	R1
Q19	Do you think MBTCC creation will help you?	Likert-rating	R1
Q19a	Please explain your choice:	Text	R1
Q19b	What are the benefits/disadvantages of MBT in correlation to test case creation based on natural language based requirements?	Text	R1

Abbreviations: MBT, model-based testing; MBTCC, model-based test case creation.

TABLE 3 Survey answers of the participants to background with computed mean value (\bar{x}), standard deviation (s), and sample size (n)

ID	Question / Answer options	Result	ID	Question / Answer options	Result
Q1	What is your main function?	($n = 69$)	Q2	What did you study?	($n = 68$)
	Development of requirements:	8.96%		Computer science:	8.70%
	Create test cases:	26.87%		Electrical engineering:	39.13%
	Automate test cases:	5.97%		Industrial engineering:	4.35%
	Execution of test cases:	13.43%		Mechanical engineering:	23.19%
	Analise testing results:	17.91%		Mechatronics:	15.94%
	Test management:	26.87%		Other:	8.70%
Q3	For how long have you been working in the automotive industry?	($n = 69$)	Q4	For how long have you been working in the sector of test case creation?	($n = 66$)
	\bar{x} and s in years:	$\bar{x} = 10.16$ $s = 8.08$		\bar{x} and s in years:	$\bar{x} = 4.33$ $s = 4.60$
Q5	Which is your focus of testing?	($n = 69$)			
	Software:	15.94%			
	Component:	24.64%			
	Subsystem:	17.39%			
	System (vehicle):	42.03%			

The participants' background revealed that more than 90% are indeed mainly working on testing (cf Q1 in Table 3). Generally, the professional background of the participants is electrical engineering (Q2), followed by mechanical engineering (automotive engineering), mechatronics, computer science, industrial engineering, and related studies, ie, physics, mathematics, chemistry, or economics. The range of experience in years is from 0.5 up to 41 years in Q3 and 0.5 to 20 years in Q4. Hence, each mean value (\bar{x}) of Q3 and Q4 resembles to its belonging standard deviation (s).

The most important resources for test case creation (Q6), according to its score of 78%, are natural language requirements, followed by personal experience, existing test cases, error descriptions, and unified modeling language (UML) or SysML models, respectively (Figure 9). Regarding the input for the test case creation (Q7), three out of four test engineers start test case creation with requirements. About 14% start with test concepts and 9% with drafts of test cases (Table 4). To sharpen the different steps between requirements and the final automated test case, the questionnaire distinguishes the *test case concept* and the *test case draft*. The test case concept is an informal or formal test case not depending on any tool, whereas the test case draft is a more specific form of a test case implemented in a certain test case environment tool. The data of question Q9 reveal that half of the participants create a concept of a test case; about 12% create a draft of a test case and about 36% an automated test case. To determine the most common scenario, we directly linked the participants' indications to the *starting point of test case creation* with the *output of test case creation*. Hence, each participant's starting point was linked to each output, eg, requirements concept of a test case, requirements draft of a test case, requirements-automated test case, and rough test case description concept of a test case. Subsequently, we compared the combinations. One third of the processes start with requirements and end in executable test cases (requirements-automated test case). More than half of the interviewees create their test cases based on a defined method (Table 4). About 40% of participants adjust their test case creation depending on the type of test case, while automatically

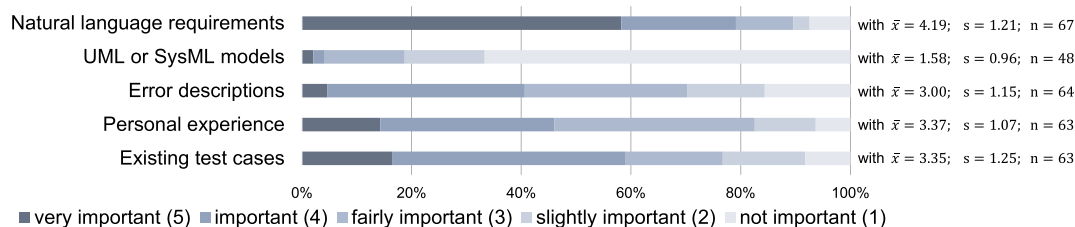


FIGURE 9 Results of Q6, “Please indicate the importance of the following resources,” with computed mean value (\bar{x}), standard deviation (s), and sample size (n). SysML, systems modeling language; UML, unified modeling language [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 4 Survey answers to artifacts of the participants with computed mean value (\bar{x}), standard deviation (s), and sample size (n)

ID	Question / Answer options	Result	ID	Question / Answer options	Result
Q7	Starting point of test case creation	($n = 69$)	Q8	Test case creation method	($n = 69$)
	Requirements:	76.56%		Defined method:	56.25%
	Rough test case description:	14.06%		Depends on the test case:	40.63%
	Basic test case structure:	9.38%		Automatically generated:	3.13%
Q9	Output of test case creation	($n = 69$)	Q11	Test cases automatically performed	($n = 67$)
	Concept of a test case:	51.67%		Yes:	34.34%
	Draft of a test case:	11.67%		No:	10.44%
	Automated test case:	36.67%		Partly ($\bar{x} = 68\%$; $s = 26\%$):	55.22%
Q11a	Environment of automated test cases	($n = 68$)			
	HiL (Hardware in the Loop):	81.67%			
	SiL (Software in the Loop):	5.00%			
	MiL (Model in the Loop):	1.67%			
	Vehicle in the Loop:	11.66%			

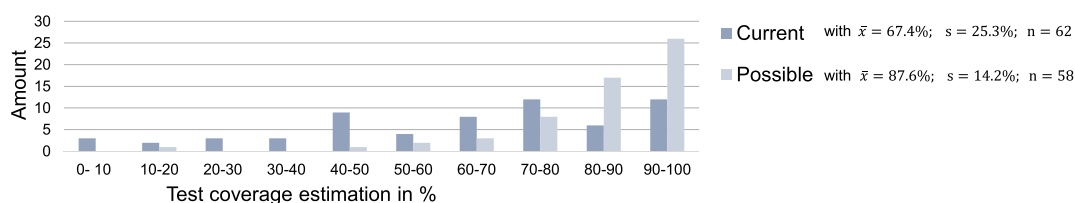


FIGURE 10 Results of Q12 and Q14: Estimated current and possible test coverage³⁶ with computed mean value (\bar{x}), standard deviation (s), and sample size (n) [Colour figure can be viewed at wileyonlinelibrary.com]

generated test cases are less used for test case creation. In Q10, about two third of the participants indicated that the most time-consuming activity is related to requirement clarification.

Regarding the basic testing environment, most test engineers (42%) focus on creating test cases for the entire system (vehicle) (Table 3). The remaining engineers concentrate on creating component tests (24.6%), subsystem tests (17.4%), or software tests (16%). Consequently, automatically performed test cases targeting model in the loop (MiL) represent 1.7% of the total, while software in the loop (SiL), 5% (cf Table 4). Test cases, mainly designed for hardware in the loop, comprise 81.6%. In contrast to the basic environment for testing (Q5), the system environment association *vehicle in the loop* is indicated as less important (11.7%).

To investigate potential test coverage improvements, the participants were questioned regarding the current and the possible coverage (Figure 10). The data of questions Q12 and Q14 reveal that a current estimation mean value (\bar{x}) of 64.7% and a standard deviation (s) of 25.3% are about 20% lower than the mean of the possible estimated coverage of 81.6% and a standard deviation of 14.2%. Thus, according to the interviewees there is a possible improvement for test coverage of about 20%.

Furthermore, the survey investigates quality focus (cf Figure 11). *Functionality* rates as the most important factor for test case quality (R2). Subsequently, *robustness* is indicated as the second most important with a similar mean value to *reliability* and *functional safety*. *Efficiency*, followed by *reusability*, *integrability*, *effectiveness*, and *maintainability* are

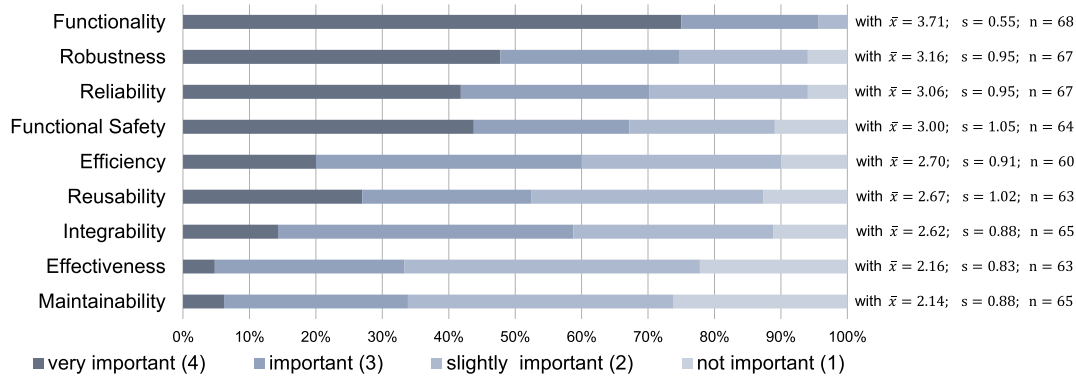


FIGURE 11 Results of Q15, “What is your focus on verification and validation?” with computed mean value (\bar{x}), standard deviation (s), and sample size (n) [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 5 Survey answers to frequency of test case creation and test case execution with sample size (n)

ID	Question / Answer options	Result	ID	Question / Answer options	Result
Q17	Frequency for delivering test cases ($n = 58$)		Q18	Frequency for testing of test cases ($n = 61$)	
	Milestones(intervals longer than one month):	46.30%		Milestones(intervals longer than one month):	77.19%
	Monthly:	5.56%		Monthly:	3.51%
	Weekly (Sprints from one up to 3 weeks included):	35.19%		Weekly (Sprints from one up to 3 weeks included):	12.28%
	Daily:	3.70%		Daily:	7.02%
	One time:	9.25%		One time:	0.00%

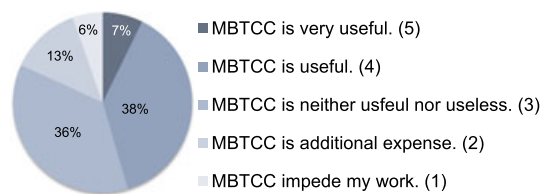


FIGURE 12 Expected benefits (Q19) from model-based test case creation (MBTCC)³⁶ with computed mean value (\bar{x}), standard deviation (s), and sample size (n) [Colour figure can be viewed at wileyonlinelibrary.com]

indicated as less important. The indications about the validity of test cases (Q16) revealed that generally test cases have to fulfill certain specification. Especially the automated test cases are valid only, if they are accurately executed in the corresponding test environment (test bed).

To investigate the frequency of the creation and testing process, the survey asks for cycles. The questionnaire distinguishes between milestone based, monthly, weekly, daily or onetime delivering, and testing, respectively (Table 5). The data reveal that delivering and testing is mainly based on certain milestones and weekly sprints.

Figure 12 shows that 45% of the participants consider MBTCC as useful or very useful, 36% of them indicated a neutral opinion toward MBTCC, and 19% consider it as a disruption. With the purpose of statistical data analysis an ordinary numerical scale was labeled with a Likert-scale, from one (*MBTCC impedes my work*) to five (*MBTCC is very useful*), respectively. Furthermore, the answers of Q19a and Q19b facilitate understanding the indications in Q19. Furthermore, the results of Q19a and Q19b facilitates understanding the indications in Q19. Besides a majority of positive associated explanations, we analyzed the negative associations. A common concern is that the requirements are too convoluted to be modeled. One participant admits that she/he does not know how to model the current requirements. Other participants assume expenses in the maintenance of the model.

Based on a mean value (\bar{x}) of 3.29, a standard derivation (s) of 0.98, a sample size (n) of 55, and a majority of positive explanations, we conclude on a generally positive expectation from MBTCC.

To investigate the aspect of a potential correlation between the personal background and the expected benefits of MBT, we linked question Q19 with the profile (**R1**). Hence, we assumed that their opinion about the benefit of MBTCC is not

TABLE 6 Correlation between background (Q1 to Q4) with expectation on model based testing (Q19)³⁶ with computed correlation coefficient (r), p -value (p), and sample size (n)

ID	Background question	r	p	n
Q1	What is your main function?	-0.05	0.371	53
Q2	What did you study?	0.03	0.421	55
Q3	Experience in automotive industry	0.19	0.088	55
Q4	Experience in test case creation	0.10	0.240	55

influenced by their background. The correlation coefficients (r) of the questions Q1 and Q2 are lower than the correlation coefficients of the questions Q3 and Q4 (cf Tables 6 and 3). Moreover, the p -values are higher for question Q1 and question Q2. The correlation coefficients lower than $|\pm 0.20|$ reinforces our hypothesis about only a weak connection between the participant's opinion (cf Figure 12) and the background (cf Table 6). Yet, the p -values indicate statistically insignificant results.

4.4 | Survey conclusion

A considerable share of the participants (**R1**) acknowledges the benefits of MBTCC, regardless of their backgrounds. The survey results allow to conclude that they have a positive attitude toward MBTCC (cf Figure 12). Based on the correlation coefficients, we assume that the positive perception is independent of the experience level, professional background, or focus of testing (cf Table 6). Despite the fact that a widely-adopted model-based approach is not in place, yet, more than one third of the participants expect to benefit from MBTCC, hence SMArDT (cf Figure 9).

The evaluation reveals that a considerable share of test engineers use personal experience, old test cases, and error descriptions as basis for test case creation (cf Figure 9). In case of new or changed requirements demand small adoptions of new test cases, test engineers usually adapt existing test cases to create new ones. SMArDT with MBTCC enhances this test case creation aspect. Instead of only adapting existing test cases, the new test cases are linked to a model. Hence, test case modifications are easy to understand.

Model-based test case creation in general enables users to generate test cases directly from requirement models. Hence, new or adjusted requirements should be derived instantly instead of adjusting existing test cases. Moreover, MBTCC should provide automatic checks to test if the test cases are relevant for testing model changes. Error cases are integrated in the early functional behavior requirements. Thus, these cases can be included in test cases with MBTCC. The clear and understandable specification of SMArDT tackle the time-consuming activity of requirements clarification. In addition, SMArDT involves test engineers in an early development phase and facilitates the automated test case creation. Hence, MBTCC should enable generating test cases independent and/or aligned to milestones, sprints, or regular deliveries. The SMArDT thereby deploys personal experience, and thus implicit expert knowledge becomes explicit in the diagrams. In consequence, the process is less vulnerable to volatile expertise caused by personal changes, besides reinforcing the test case creation and supporting hypothesis (**R2**).

The second functional and the third technical layers of SMArDT address functionality, functional safety, robustness, and reliability (cf Section 2). According to our survey, participants already acknowledge the focus on verification and validation. The concluded potential for test case coverage reveals that MBTCC should be capable of enhancing the test case quality and further consolidates our hypothesis **R2**.

According to the participating test case engineers, system tests, subsystem tests, and component tests are the most promising environments for MBTCC at the BMW Group (**R3**). The low approbation of SiL and MiL shown by the survey is probably related to the group of participants. The target group of the survey is composed of test engineers and not software developers, who usually perform SiL and MiL tests. In relation to the system's basic environment, automatically executable tests are primarily performed on hardware in the loop environments (cf Section 4.3), whereas the focus of testing is often the complete vehicle or its subsystems. The findings also reveal that MiL and SiL play a less important role. These findings of MiL and SiL as well as the indicated less considered sources of UML and SysML models (cf Figure 9) could be because of the target group of test engineers. Test engineers usually use requirement specifications in natural language while software developers work with models.

5 | A SMArDT APPROACH TO MBTCC FOR AUTOMOTIVE SOFTWARE TESTING

Our survey findings reveal that model-based software engineering in the automotive industry does not fully exploit the potential of the model-artifacts created by requirements engineers, yet (cf Section 4). The aim of MBTCC is to utilize model artifacts for purposes beyond documentation. The SMArDT requires a systematic testing concept for test case creation on every abstraction layer to decrease test case adaptation and creation efforts, which have proven problematic in classical V-Model development processes. An MBTCC approach seems beneficial and applicable to SMArDT's formalized SysML models. However, existing approaches to automatic test case generation³⁷⁻⁴² do not suffice to fulfill project-specific requirements on the modeling language. This is due to the fact that (1) within SMArDT's second layer, data types are not yet specified but no extra testing diagram is created for test case creation. Thus, input for test case creation are production models of the second layer without data types. (2) The BMW Group's production models contain modeling constructs such as cross-synchronization.⁴³ Existing AD semantics do not allow such constructs.

Therefore, we developed a SMArDT concept for automatic test case generation. As the models are consistent throughout all SMArDT layers (cf Section 3), MBTCC provides means to transform test cases of one layer to corresponding test cases on all lower levels. Thereby, tests are also consistent on each layer and provide additional support for consistency checking. By means of MBTCC, test case adaptation effort could be reduced significantly, as our survey revealed (cf Section 4). Based on the results, we concluded that a semiautomatic test case derivation by MBTCC would support test engineers. This includes adapting existing test cases to requirement changes and deriving test cases for complex system tests. Hence, our MBTCC approach for SMArDT constitutes an extension to the current testing concept. Due to the model-based functional system requirements, we aim to automate manual steps in the test case creation process. This section presents our MBTCC approach for the second SMArDT layer. It utilizes SysML ADs to generate test cases that verify functional properties automatically. Test engineers may utilize these test cases for direct testing on any test bed, or as starting point to formulate test cases that are more complex.

5.1 | Model-based test case creation concept

In Section 3, it has been shown how the formalized SMArDT model artifacts can be used to assure consistency between models on each layer. Our MBTCC approach uses these models to derive functional test cases in a systematic and (semi)automated way.

Figure 13 illustrates how MBTCC is integrated within the second layer of SMArDT. System requirements are elevated on the first SMArDT layer and modeled by UC and composite SysML diagrams. The testing strategy on this layer, which utilizes these artifacts, has not yet been conceptualized. The requirement diagrams are passed to the second SMArDT layer for abstract functionality modeling by means of SysML ADs. Representing first class requirement function models, they are input for the (semi)automated test case derivation concept presented here. The resulting test cases are used in a verification step to map signals to components of the system under test and can be executed on a test bed.

Model-based test case creation is applied in the *Test Case Derivation* step of Figure 13. The formal SysML input models contain references to logical information flow entities and thereby specify different system functionalities. Within the second layer of SMArDT they are represented by logical signals, which can be mapped to hardware and software signals

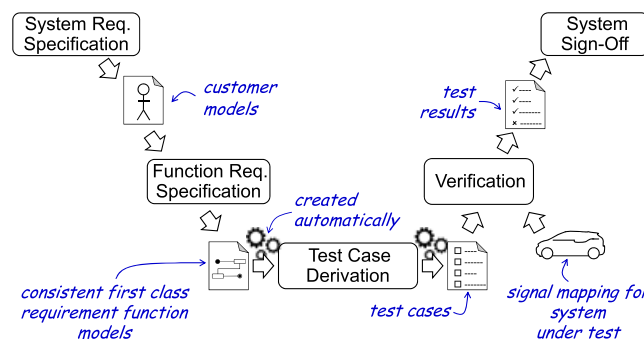


FIGURE 13 Including the test case derivation in the overall systems engineering process³⁶ [Colour figure can be viewed at [wileyonlinelibrary.com](https://onlinelibrary.wiley.com)]

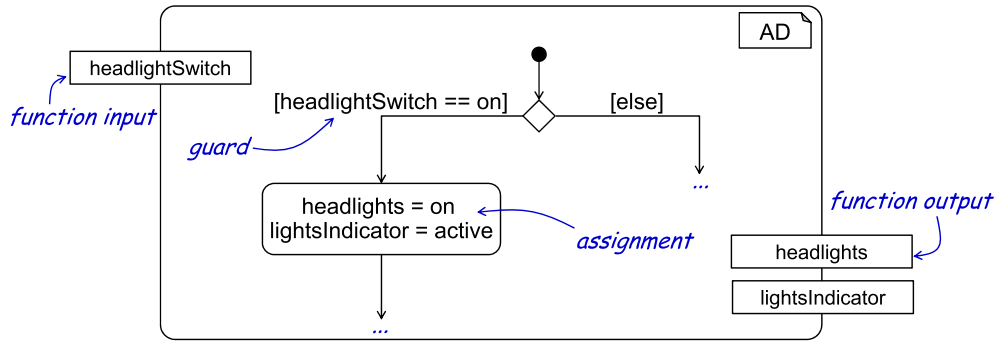


FIGURE 14 Excerpt of an activity diagram (AD) describing requirements on the activation of headlights controls³⁶ [Colour figure can be viewed at wileyonlinelibrary.com]

on the third and fourth layers. Our MBTCC-based generation of test cases yields a set of test cases according to the path coverage criterion C2c,⁴⁴ ie, loops are considered up to a finite number of iterations. Path coverage implies the presence of at least one test case for each path of the AD's graph representation. The paths considered here correspond to functional execution paths. An execution path corresponds to a certain input and temporary signal value assignment such that all its guard conditions hold, ie, a signal interpretation. For each path, its validating signal interpretation can be calculated by evaluating all guard conditions and signal assignments that correspond to a Boolean satisfiability problem (SAT). If a signal interpretation exists among all signals of the paths and their possible value set, the path is considered valid.

Figure 14 presents an excerpt of a simplified version of the running example AD introduced in Section 3. Although supported by our method, we omit object flows at this point. The interface, ie, input and output ports, contains information about the input and output signals of the function modeled by the AD. For a given path, all possible signal interpretations according to the inputs are determined based on its guard conditions and assignments. Boolean guard conditions restrict the range of viable input signal values for the path's execution; signal assignments define its result output signal value set. Consider the leftmost path in Figure 14, which is only entered, if signal `headlightsSwitch` is set to `on`. Given the set of possible values for each signal in the diagram's scope, the path is valid if `on` is contained in `headlightsSwitch`'s possible value set. Otherwise, the path is marked invalid, ie, no signal interpretation exists, such that the path is executed; thus, no test case is needed.

For loop handling, we set the C2c-limit of loop iterations to two. Initial applications of our method showed that a limit of two iterations suffices for the test case creation of current models. A higher number of iterations deluges the amount of test cases without contributing important content to the test cases. Even so, our tooling provides a way to configure the limit if a higher number of loop iterations needs to be tested.

5.2 | Test case structure

Our method structures test cases into three blocks, ie, *preconditions*, *actions*, and *postconditions*. The preconditions define the initial situation prior to a test case's execution. If all preconditions are fulfilled, the action block is executed. The execution of a test case corresponds to the execution of its respective path in the source AD. Thus, the action block represents the core of the derived test case. Finally, the post conditions define the expected situation after the execution of a test case.

Each block consists of atomic test steps. An atomic test step provides information about (1) the type of action, ie, setting (`Set`) or checking an information (`Check`); (2) the name of the logical information; and (3) the value to which an information is set or which is expected when reading the information.

Considering our running example in Figure 14, the AD models the functionality of the light switch. A test case to check this particular functionality of the system should validate whether the actual headlights turn on and the cockpit indicator shows the correct headlight state. Table 7 shows the action block of an exemplary test case corresponding to the leftmost execution path. Its action block contains three atomic test steps. Each step either sets or checks the value of a logical signal. More specifically, checking input signal values in a guard condition results in a `Set` step, while value assignments result in `Check` steps. The signal names of the test case refer to the switch for the headlights (`headlightsSwitch`) and the indicator light (`lightsIndicator`) in the cockpit as well as the actual headlights (`headlights`).

Table 7 shows that the sequence order of atomic test steps matches the order of passed token elements in the corresponding AD (cf Table 7 and Figure 14). Since test cases correspond to valid paths through an AD, the occurrence of

TABLE 7 Exemplary action block of a derived test case³⁶

Action	Parameter	Expectation
Set:headlightsSwitch	on	
Check:headlights		on
Check:lightsIndicator		active

information flow entities, ie, signals, on the respective path determines the ordering within the action block. Timing information, necessary for execution on a test bed, may be added later on. As it depends on functionality's actual source code implementation, however, this information should not be part of second layer SMArDT models as this layer describes abstract functionality without implementation specifics. For example, timing information could be necessary between Set:headlightsSwitch and Check:headlights.

5.3 | Test case execution

Before running the test case generator, additional information for the generation can be provided. This information is not contained in the model and facilitates generating specific test cases. (1) The test case generator enables the user to add customized precondition and postcondition (cf Section 5.2). Often test cases need a specific system state, ie, a set of hardware and software signals. These signals are set in the precondition. After the action block, the postcondition sets the signals in the original state. These conditions have to be added, due to not being part of the specific modeled function. (2) Including an environment model enables the user to assign specific testing values to variables, eg, temperature to minimum, 30 degree Celsius, or maximum. Moreover, default values, derived from the model, can be overwritten.

On the second layer of SMArDT, abstract ADs only reference logical information names rather than specific software and hardware signals (eg, bus signals). To execute the derived cases, however, it is important to map the logical signals to their specific software and hardware counterparts beforehand as shown in Figure 13. The mapping defines how to execute each atomic test step of a derived test case in the system under test. If necessary, timing information is added. Thus, a single atomic step in a test case might result in several test steps on the actual test bed.

To guarantee that a derived test case can be executed automatically, each atomic test step must be automatically executable on its own. Hence, such a mapping between the logical information flows of a derived test case and the system under test is crucial for our method. It ensures that each derived test case can be executed on a test bed without the need of manually mapped logical information flows to specific signals. Thus, the mapping highly depends on the system under test as well as the test bed. Currently, this mapping is manually created as part of our method, however, developing an automated solution is work in progress.

6 | A SMArDT TEST CASE GENERATOR

The MBTCC method presented requires a stable and adaptive implementation of a test case generator. Our intention is to target test case creation within SMArDT. Our method does not prescribe a modeling environment, thus, the implementation aims to be independent of the modeling tool. To serve this aim, we used the MontiCore language workbench to develop a domain specific language tool, *ADs for SMArDT* (AD4S), into which ADs modeled in any modeling environment can be translated easily.

6.1 | The MontiCore language workbench

MontiCore is an extensible workbench for engineering compositional, textual modeling languages. Developers define languages as context-free grammars that specify concrete and abstract syntax. MontiCore generates the model-processing infrastructure (eg, parsers) to facilitate checking and transforming models of the language into code automatically. Various MontiCore languages have been engineered and applied to different domains,[‡] including automotive, cloud computing, smart homes, robotics, and software engineering itself. For the latter, we have developed the UML/programmable (UML/P) family of modeling languages,⁴⁵ which is a subset of UML⁴⁶ that is refined to enable pervasive model-driven

[‡]See <http://monticore.de/languages/>

engineering without the underspecification discrepancies of UML. This subset includes ADs, class diagrams, SCs, and sequence diagrams. All of these languages are well integrated with each other and come with comprehensive tooling (model checkers, model transformations, code generators, etc). As SysML ADs are closely related to UML/P ADs, we created AD4S as a derivation of the UML/P version.

MontiCore also provides an infrastructure to include the well-formedness rules of input models via context conditions, which is of special value for SMArDT and MBTCC. As shown, model inconsistency checks between layers require formalized models. For the automatic derivation of functional test cases, the input models need to represent a certain logic; otherwise, they are not machine readable preventing any kind of automatic processing. In the following sections, we will present the MontiCore-based AD4S, which provides the working environment for the test case generator including an adaptive data structure and well-formedness checks.

6.2 | Activity diagrams for SMArDT

In the context of SMArDT, our aim was to develop a tool for automatic test case generation. The implementation should be independent of the modeling environment and aid in identifying erroneous models at an early development stage within SMArDT. Hence, we developed AD4S using a MontiCore context-free grammar, shown in Figure 15, as a derivation of the UML/P AD.

Naturally, the language contains a nonterminal for each basic UML/P AD-element. As Node is an interface, hierarchical structuring of Activity elements is possible. Each Activity has a list of Nodes, which may be simple Actions or Activities themselves (cf lines 2 to 6, Figure 15). An Action represents the most basic kind of node in an AD. Activities and Actions may be referenced by a requirement specified in SMArDT Layer 1. To cover all SMArDT Layer 1 requirements, test cases have to be linked to the requirements contained in the underlying diagram. Hence, Activity and Action objects may be linked to an arbitrary number of Requirements. These elements consist of a simple string name, a value that holds the requirement's text, and a security attribute. The latter holds the information whether the requirement, and thus, the activity modeled by the Node is safety critical. This information is needed to identify test cases for functional safety. A Node also holds a list of ControlNodes, which is an interface for the basic control nodes of UML/P ADs and an implementation of Node. Figure 15 shows their implementations in lines 14 to 20.

The UML/P AD connection between nodes, ie, edges, are also part of the AD4S (lines 23 to 30). The id simply serves for identification purposes while source and target reference the connected nodes. A guard on a transition is optional, but is an important construct in SMArDT ADs. Here, AD guards serve two purposes, ie, (1) if source is a DecisionNode, it should always contain a condition in form of a Boolean expression. Without it, path validation as explained in Section 5 is not possible. The condition may be else, dictating the path in case all other conditional guards

```

01 grammar ActivityDiagram {
02     interface Node;
03     Activity implements Node = "activity" Name Requirements? "{"
04         Parameter*
05         (Node | ControlNode | Edge)*
06     "}" ";" ;
07     Parameter = "in:" | "out:" Name ("," Name) ";" ;
08
09     Action implements Node = "action" Name Requirements? ;
10
11     Requirements = "[" req:Requirement ("," req:Requirement)* "]" ";" ;
12     Requirement = "(" Name "," value:String "," security:["security"] ")" ";" ;
13
14     interface ControlNode extends Node;
15     InitialNode implements ControlNode = "initial" ";" ;
16     FinalNode implements ControlNode = "final" ";" ;
17     FlowFinalNode implements ControlNode = "flowfinal" ";" ;
18     ForkNode implements ControlNode = "fork" Name ";" ;
19     JoinNode implements ControlNode = "join" Name ";" ;
20     DecisionNode implements ControlNode = "decision" Name ";" ;
21     MergeNode implements ControlNode = "merge" Name ";" ;
22
23     Edge = "transition" id:String? source:Name ("->" guard:Guard? target:Name)+ ";" ;
24     Guard = "[" Variable+ | ("else" ("/" Variable+)) | (BooleanExpr ("/" Variable+)) "]" ;
25     Variable = signal:Name ("=" value:Name)? ";" ;
26 }

```

Annotations in the figure:

- interface definition of activity nodes (points to line 02)
- support for nested ADs (points to line 05)
- basic action node (points to line 09)
- Infrastructure for requirement handling (points to lines 11-12)
- definitions of basic control nodes (points to lines 14-21)
- guards for Boolean expressions, variable definitions or assignments (points to lines 24-25)

FIGURE 15 MontiCore (MC) context-free grammar of the activity diagram for specification method for requirements, design, and test (AD4S) [Colour figure can be viewed at wileyonlinelibrary.com]

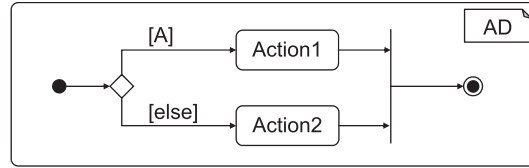


FIGURE 16 Deadlock: exactly one token is placed at the initial node. None of the contained activity diagram (AD) elements, ie, *DecisionNode* and *action*, create other token. The join node waits until two tokens from the two incoming edges arrive, but the join node will only ever be reached by one token and can never continue

of the branch falsify. (2) They can be used to model signal assignments or definitions, as modeled by the last option of *Guard*. A combination of both is possible.

The calculation of all paths within an AD requires well-formedness in a way similar to UML/P ADs, which is based on Petri-Net semantics as suggested by UML ADs.⁴⁶ MontiCore provides the valuable concept of context conditions, which we utilized to impose well-formedness constraints on SMarDT ADs.

6.3 | Well-formedness constraints for ADs for SMarDT

The SMarDT method relies strongly on automated processing of *SysML* models. The method described in Section 5 derives functional test cases for each execution path modeled by an AD. To ensure correct path calculation, we need to impose strict well-formedness constraints on the input diagrams. The control flow graph cannot impose all of these constraints; instead, MontiCore provides a context condition infrastructure including interfaces and visitors for model checking. Therefore, we equipped AD4S with the following context conditions:

(1) Diagrams may not contain deadlocks that occur in cyclic waiting situation. Figure 16 illustrates an example. The join node will continue if two incoming flows have arrived.⁴⁷ (2) Infinite loops cannot prevent path calculation. They occur whenever a node can be reached from itself and no exit condition will ever come true. (3) Branches must always be complete, ie, its guard conditions must cover the diagram's entire value range. This is necessary for path validation, because branching is not decidable otherwise. (4) Input variable values must not be changed on any path of an AD. (5) Names and IDs of nodes and edges must be unique. (6) Edges originating from a *DecisionNode* need to always specify a Boolean guard condition.

6.4 | A tool for automatic test case generation

Activity diagram for SMarDT is designed to provide a mean for MBTCC in the automotive industry independent of the modeling environment. Our tool is equipped with a parser, which translates ADs in extensible markup language (XML) to AD4S.

The output of the modeling tool is therefore translated to XML before parsed into AD4S. The AD4S-representation (Figure 19) can now be used to derive test cases which are stored in a format executable by functional test execution tools. The corresponding tool chain comprises the following components: (1) The AD-converter transforms the ADs exported from the *SysML* modeling tool to AD4S; (2) the test case creator transforms the ADs into an intermediate test case representation. In this step, the configuration (cf Section 5) to customize the generated test cases is processed; and (3) a test case exporter transforms the resulting test cases to the desired output format. Currently, the tool supports spreadsheet files (XLSX) or formats executable on automotive test beds. Figure 17 shows an illustration of the overall toolchain and the underlying process.

Figure 18 shows an excerpt of the XML-transformation of the running example introduced in Section 4. The XML-elements represent the basic AD-elements *Diagram*, *Node*, *Pin*, and *Edge*, which are subtyped by the *type* and *nodeType*-attributes. Edges reference source and target nodes via the *Node* or *Pin* attribute id. Guards are represented by children of *Edge*.

The XML is given to the *ADConverter* and parsed into the AD4S representation of the original model. Figure 19 shows the AD4S representation of the running example. As AD4S is a MontiCore DSL tool, generated parsers can be used out of the box. Once the model is parsed, the generated context condition checker assures syntactic and semantic well-formedness of the input. Erroneous models can be identified at this stage of the test case generation process and

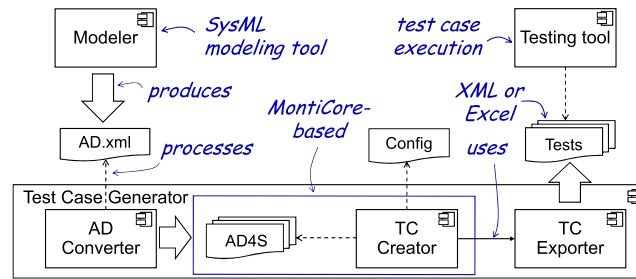


FIGURE 17 Model-based test case creation tool chain. AD, activity diagram; AD4S, activity diagram for specification method for requirements, design, and test; SysML, systems modeling language; TC, test case; XML, extensible markup language [Colour figure can be viewed at wileyonlinelibrary.com]

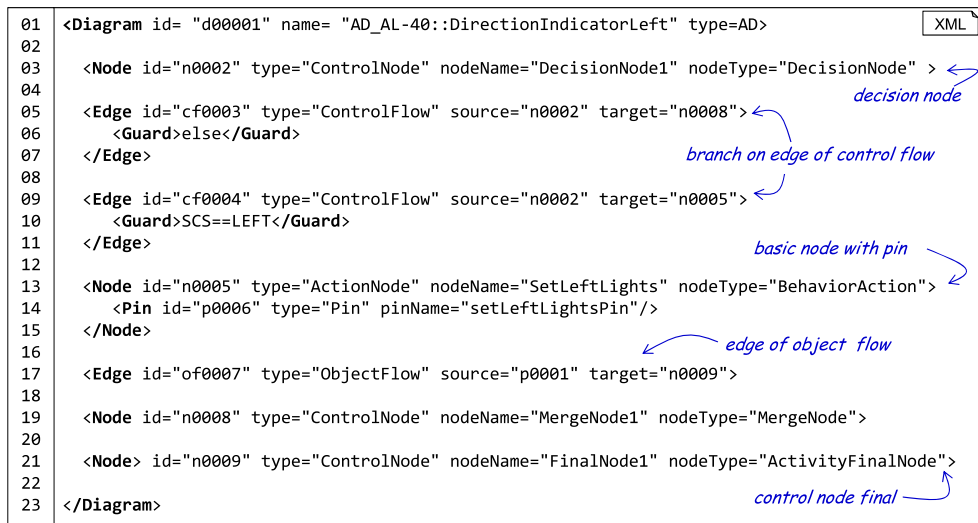


FIGURE 18 Extensible markup language (XML) transformation result of the running example Section 4 [Colour figure can be viewed at wileyonlinelibrary.com]

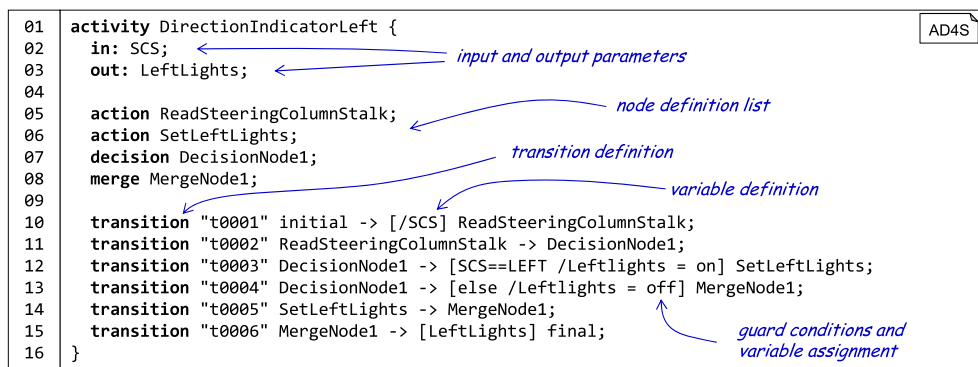


FIGURE 19 Textual representation of a formalized activity diagram in activity diagram for specification method for requirements, design, and test (AD4S) [Colour figure can be viewed at wileyonlinelibrary.com]

no further generation is performed. This feature of our MBTCC tool significantly benefits the notion of SMArDT as it prevents erroneous models from being passed down to the implementation Layers 3 and 4.

A configuration model to include additional information needed by test beds in the test case automatically can configure the TCcreator. This component creates the test cases for each path accordingly before handing them to the TCexporter. The testing tool responsible for executing the test cases takes the exported files as input. Generally, the

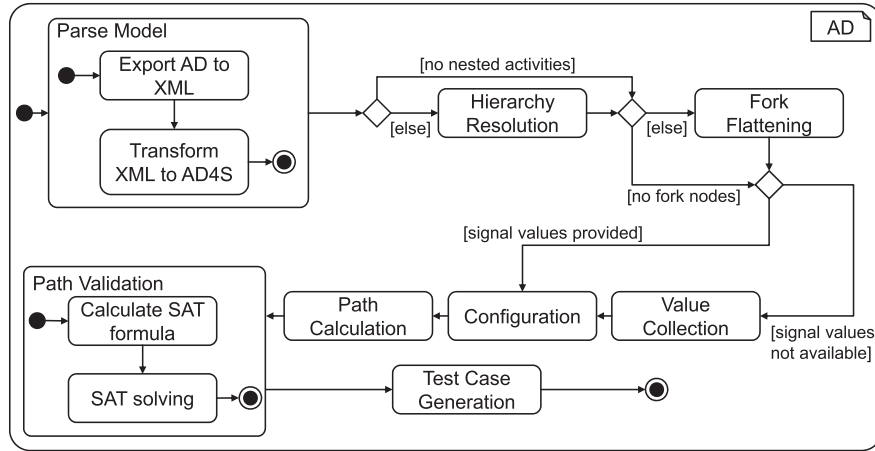


FIGURE 20 Activity diagram (AD) showing the intermediate steps of the automatic test case generation implemented by the test case generator. AD4S, activity diagram for specification method for requirements, design, and test; XML, extensible markup language

modular infrastructure allows choosing the input and output formats according to the desired testing tool input format by exchanging the corresponding parts of the MBTCC toolchain.

Figure 20 shows an AD to illustrate all intermediate steps of the test case generation process. First, the model editor export is transformed into XML, which can be translated to XML. As the editor's export usually contains information irrelevant for test case generation, the export is filtered and only necessary elements are stored in the XML. In a second step, a translator translates the XML-AD to AD4S. Further processing steps can now utilize the abstract syntax tree (AST) of this model to generate test cases.

To be able to calculate the execution paths represented in the AD, model-to-model transformations⁴⁸ of the AST resolve nested activities and concurrency, if present. As for hierarchies, the nested activities are mounted in the uppermost AST by connecting the edge targeting the nested AST with the nested initial node and creating an edge connecting the nested final and the subsequent node at the superordinate level. Resolving concurrency requires an interpretation of its meaning. In this case, as the test cases are functional, only the outcome, ie, the values of the output signals, is of interest. We therefore, transform concurrent execution paths into one sequential sequence of nodes and edges based on the assumption that parallel activities are independent and are thus executable in arbitrary order without imposing changes to the final output. The transformation result is an AD4S model free of nested activities or forked paths.

In software testing, it is necessary to compare actual signal values to expected values. The value expectation may be present in form of a mapping between information flow entities and the actual platform-specific signals as described in Section 5.3. However, this mapping is not always provided. The value collection step therefore collects all possible values occurring in the logical information flows of the input AD. The process checks every occurrence of an information flow in the AD and gathers all of its values, giving a set of possible values for each logical information flow.

Additional configurations are applied in the next step. Our method provides means to customize the generation to suit the requirements of test beds. Hence, prior to the actual path calculation, configurations can lead to additional transformations of the AST.

Once the AST is transformed according to all configurations, the process calculates all execution paths of the transformed AST according to the C2c path coverage criterion (cf Section 5). The AST now corresponds to a directed graph; thus, we apply depth-first traversal get a set of all paths, regardless their validity, which is determined in the next step.

Naturally, the guard conditions along a path will only hold for certain signal interpretation sets. A path is valid if at least one signal interpretation exists such that all its guard conditions hold. All possible signal interpretations for the input AD have been found in the value collection step, if not provided as a mapping. Determining whether a path is valid or not according to this definition depends on the satisfiability of the conjunctions of all guard conditions along the path and is solved using the Z3 Theorem Prover.⁴⁹ For the left path of the exemplary AD shown in Figure 14, the calculated formula is

$$\text{headlightSwitch} = \text{on} \wedge \text{headlights} = \text{on} \wedge \text{lightsIndicator} = \text{active}.$$

If a path is valid, Z3 creates a witness for a possible solution of the conjunction. This witness is used to determine the value assignments for input information flows fulfilling the combined expression of a given path. If no witness exists, the path is invalid and therefore discarded.

After validating each path, the set of all valid paths and respective signal interpretations are used as input for the test case generation. For each path of the set, the atomic test actions are exported into the format supported by the tool responsible for the test execution as configured by the user.

7 | DISCUSSION, EVALUATION, AND LESSONS LEARNED

This section discusses the survey results, test cases evaluation, and highlights lessons learned from deploying SMaRT MBTCC at the BMW Group regarding typical threats and empirical research.

7.1 | Threats to study validity

Distributing the survey by email and forwarding it lead to threats to the internal validity. Despite the careful selection of our participants by their specialization in the context of test case creation, we cannot eliminate the threat of wrong interpretations and duplicated submissions. We intend to reach a greater number of external test engineers involved in the process of test case creation by forwarding the email. Due to not all the external companies gave permission to collect their test engineer's opinions, we can neither ensure that all the companies were included in our survey, nor that all the test engineers were questioned. For instance, the automation of test cases is usually assigned to external service providers. If all external test engineers were included, the distribution of Table 4 could be different.

To preserve independence of development and testing, test engineers are often from external companies. As we could not reach a higher rate of participants, there is a possibility that the results could have been different. So we added fictitious circumstances "all resources and time you need" for possible coverage (cf Figure 10). By doing so, the questions could be perceived as a performance review and lead to adjusted estimations.

Besides threats to internal validity, threats to construct validity arise from employing an online survey. The type of questions, their style, and options that were presented as answers might have influenced the participants' response. For instance, to get a solid statement, the question about the focus on quality assurance has no alternative for a neutral statement, except "I do not know" (cf Figure 11).

Furthermore, threats to statistical validity arise from the mentioned statistically insignificant results, ie, indicated by the p -values of Table 6. We are 95% confident that the margin of error for our sampling procedure and its results is no more than ± 20 participants with a total sample of 70 participants in the mentioned survey (see Section 4.3). Based on the margin error and the possibility that not all the participants answered every questions, as well as the results being statistical insignificant, we assume trends. Besides the survey data, we compared the generated test cases to traditionally manual created test cases in the following section.

7.2 | Evaluation

Based on a promising e-drive function, a team of test engineers made an evaluation on the test cases generated with the test case generator. This function suited the demands of the test case generator. Together with the professional testers, we conclude that the MBTCC method has potential to save up to 50% of efforts, measured in hours, per test case (cf Figure 21). For evaluation team of professional test, engineers chose 12 generated test cases of one e-drive function and manually remodeled them, while their expended efforts in time were tracked.

Regarding to the results showed in Figure 21 we could affirm that the main saving potential is in developing the test case concept. With this concept, the test generator directly creates test cases based on the functional model, instead of interpreting the requirements and work out manually test cases. In addition, the model has to be evaluated for the test case generator and the test engineer has to set up the test generator.

The team scanned 260 manual test cases for the aforementioned e-drive function. A share of 8% of the test cases are based on volatile experience of the testers and cannot be formalized with MBTCC. For the remaining 92% of test cases, there is a potential for savings to 46% (cf Figure 22). Assuming that the test engineers pick one promising e-drive function of more than 30 e-drive functions, we finally conclude that on average the MBTCC method can save up to 46% for promising e-drive functions with specific decisions in the functions.

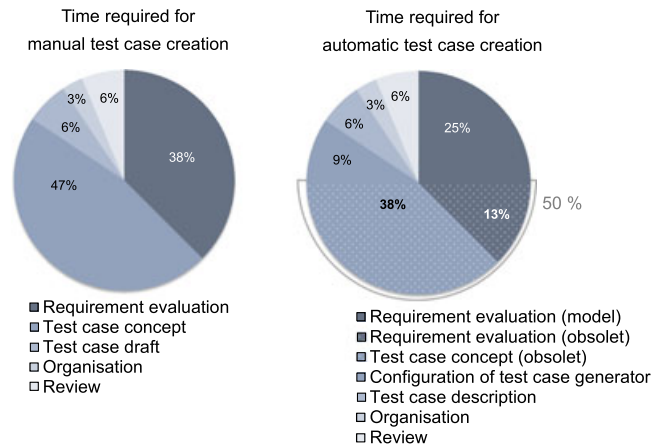


FIGURE 21 Potential for automated test case creation in time (hours) required for related activities. The function suited the demands of the test case generator [Colour figure can be viewed at wileyonlinelibrary.com]

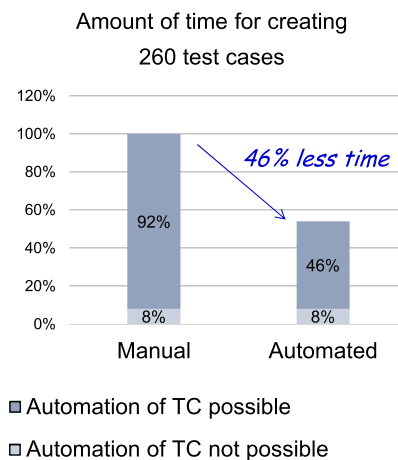


FIGURE 22 Best case scenario: comparison of time required for 260 manual and 260 automated created test cases (TCs) [Colour figure can be viewed at wileyonlinelibrary.com]

7.3 | Lessons learned from applying MBTCC at BMW Group

The successful introduction of new methodologies requires identifying and convincing all involved stakeholders. All persons involved have to be convinced to benefit from the initial effort, even if the current advantage is not perceptible at first sight. Hence, necessary organizational or procedural changes have to be supported or at least understood. The changes of MBTCC are considered necessary and helpful by all partners in general. Nevertheless, we underestimated the effort required to convince all involved stakeholders. Miscommunication lead to a misunderstanding of the term *automated test case creation*. Some testers interpreted the term as a full work replacement by a more or less simple algorithm. The persons involved misinterpreted the new approach. The requirement specifier perceived an additional workload. The created test cases perceived that their experienced based and intellectual input could not be replaced. As a result, both sides sealed themselves off instead of achieving a cooperation between specifier and tester. The initial misinterpretation of both sides was replaced by the mindset to include the test engineers' expertise into the model. From this moment, the team members of MBTCC collaborated, discussed constructively, and supported the new method.

Despite the initial challenges, all involved parties agreed that the required additional effort learning MBTCC and adapting iteratively existing models significantly reduces with further tasks and ultimately pays off for all parties. Besides a discussion of the method, the effort was to overcome the lack of common *SysML* understanding.⁵⁰ The test case generator needs well-formed models that force the parties to settle a basic understanding in model elements and syntax. The well-formedness rules created a common understanding in reading the model. Hence, domain expert and people with less expertise could understand the models. To assist the modelers the method uses automatic model

checks for well-formedness and generator compatibility to identify and correct models in an early stage of development. Consequently, the misunderstandings decreased and there is less misinterpretation of models.

With the growth of the models, logical correctness and well-formedness of the models become more important. Our tool chain discards models that are not well-formed. Hence, modeling errors are already detected at an early stage of SMArDT. This strict adherence was initially perceived as a handicap. Often, parts of a model were logical correct and well-formed, while other parts were still under construction. This led to the idea of extending the configuration of our method and tool chain to enable the creation of test cases only for parts of a model. This challenge is still under investigation.

Moreover, large projects with an immense number of requirements and corresponding models become more and more difficult to cope with. It is therefore of great importance to automate the process of consistency checking between requirements and various models representing different parts of the overall system as well as the correctness of model refinements. Hence, a series of quality assuring techniques for structural and behavioral verification based on formal modeling languages need to be integrated seamlessly in the SMArDT process and should be impossible to forego. A continuous integration of models in SMArDT improves the quality of the software design dramatically and delivers a solid basis for a generative development process. Particularly, we observed that it reduces the number of semantical errors in the generated test cases due to ambiguous specifications and facilitates the integration of software modules developed by different teams in late development stages.

Another challenge is the request for test case generation of concatenated ADs. The test engineers want to derive test cases for a list of separate functions. Activity diagrams should be “executed” in succession and test cases should be created accordingly. This can already be done for ADs, which have distinct inputs and outputs. For ADs with intersecting sets of information flow entities test case creation becomes more complex. The highly increasing number of paths, which need to be covered, pose the actual challenge. We quickly realized that the C2c coverage criterion is not proper for this UC. Hence, we are currently investigating new coverage criteria to fulfill the request of the test engineers.

The feedback for our method and its tool chain was very positive. The interaction between experts of different domains (testing, modeling, engineering, and informatics) contributes to the success of SMArDT and leads to a high-quality product.

8 | RELATED WORK

Research on improving and tailoring the V-Model mostly focuses on its integration into different business structures.⁵ The SMArDT methodology on the other hand focuses on the formal and technical aspects of the specification diagrams of the different layers in the V-Model to ensure traceable, verifiable, consistent, and particularly, testable artifacts throughout the whole development process.

Our idea with its presented survey on MBT in automotive industry relates to other MBTCC approaches. One online survey⁵¹ investigates quality assurance in the automotive, telecommunication and banking companies. This survey considers the different test steps used for testing, who creates and runs the test cases, how they create test cases, the methods for testing, the current situation of test case automation, and the performance of testing. The results indicate that test cases are still mainly created manually. The focus of testing is associated with functionality tests (60% to 80%)⁵¹ and it corresponds to our results (70% to 80%). Furthermore, the online survey⁵¹ investigated the effectiveness of the test cases. Nearly 60%⁵¹ of the participants estimate that their test cases reveal most of the defects, while 20% estimate that several defects can be identified. This data fits to the current test/requirement coverage presented in our survey. In contrast to our approach, the online survey⁵¹ focused on quality assurance in different sectors; consequently, it is more generic. Our survey investigates especially test case creation in the automotive domain, in which test engineers are chosen as participants and the structure of the test cases is known. Another automotive survey⁵² concentrates on the tool usage during requirements engineering and testing instead of the potential benefits of MBTCC. Considering this focus, these findings⁵² did not address our research questions.

Besides the surveys that were already presented, there is a case study on MBT in the automotive industry conducted in the BMW Group.⁵³ The case study aims to compare the quality between the model-based tests and the traditional “hand-crafted” tests. In contrast to the system structure diagrams of a single automotive network controller, our approach targets full e-drive functions modeled with ADs and SCs. The results of the case study are promising. The automated MBT approach detected more defects than the traditional hand-crafted tests. Especially the defects caused by inconsistent requirements that could be also identified. Despite of this, neither the traditional nor the automated tests revealed all errors. Hence, we conclude that it reinforces our findings of the additional value of MBTCC.

Apart from the particular industry approaches, MBTCC is subject to active research.¹² The topic of (partially or fully) automated derivation of executable test cases from various input modeling languages has been brought forward by diverse approaches. Most approaches concentrate on UML or SysML and can be distinguished based on their support of the specific input models and on their support of concurrency in the input models. Various approaches to MBTCC employ SCs or sequence diagrams as input models.³⁷⁻³⁹ Furthermore, other concepts employing ADs as input models for MBTCC do not support concurrency at all.⁴⁰⁻⁴² Besides the target on the language usage, other approaches presented different tools to support and to facilitate MBT, which include, for instance, the tool MBTsuite.⁵⁴ It employs ADs for test case creation and it is capable of exporting these ADs into a variety of different outputs. Compared with our approach, the tool does not support concurrency in its input models. Another tool, UMLTest,⁵⁵ which derives test cases from SCs, also does not support concurrency of input models. On the other hand, UMLTGF,⁵⁶ which is similar to our idea, processes ADs and supports ADs featuring concurrency. In contrast to our concept, UMLTGF proposes not to involve prematurely the tester in details of the implementation. Furthermore, the tool is unable to produce the output format required by the BMW Group.

Being one of the most popular model based engineering tool suites, Simulink²⁹ provides means for requirement management enabling one to link requirements to models and tests. Furthermore, it enables checks ensuring the compatibility with standards such as ISO 26262 and design guidelines. Tests can be generated automatically using the SignalBuilder. Therefore, Simulink provides means to measure the test coverage in models and generated code. The StateFlow tool performs consistency and completeness checks on state machines. It lacks automated consistency checks, however, able to discover behavioral or structural inconsistencies as we introduced them in SMArDT using C&C views and ADs.

The presented transformation from originally natural-language requirements, to model-based test cases, to AD4S, and to executable function tests relates to approaches translating natural language requirements to test cases directly. For instance, the *aToucan4Test* framework⁵⁷ transforms an input restricted to a subset of structured English. This approach does not support the precision and complexity achieved through manual function test modeling by domain experts.

Moreover, various domains already leverage automated test case creation,^{58,59} such as telecommunication, finance, transport, and medical software engineering. The study by Weißleder et al.⁵⁹ confirmed MBTCC in these industries increases testers' efficiency. This especially holds for requirements adjustments that detect defects in earlier stages of development and increases functional test coverage. These findings reinforce our observations. Despite these findings, automated test case creation in automotive software engineering with its established development and testing processes is rarely documented. Usually, research focus on specific tooling or propose just general and abstract overviews.^{60,61}

9 | CONCLUSION

Increasing connectivity, coupled with shortened and frequently modified processes, challenge the automotive industry to achieve shorter time-to-market without quality loss. To maintain high-quality standards, testing is of crucial importance in automotive software engineering. To tackle these challenges, agile processes according to the V-Model dictate the automotive development cycle. However, the classical way does not suffice because system adaptation is only reasonable on the lowest abstraction layers. To maintain a high-quality standard and provide more functionality in shorter time, artifacts on all abstraction layers must be consistent with the development stage at any time. The SMArDT aims to provide mechanisms for efficient adaptation by relying on model-based software engineering techniques. Consistency assurance between abstraction layers and automatic test case generation holds the potential to decrease adaptation efforts. As a showcase, we discussed how C&C views and ADs are employed to tame large models with a big number of requirements, thereby improving the quality of the generated test cases. Furthermore, we investigated how MBTCC is perceived to improve testing among experts at the BMW Group. We received 69 valid questionnaires out of initially 196 invitations sent. The study revealed that the participants expect to benefit from automated test case creation regardless of their backgrounds. Furthermore, the participants expect the test quality to increase and perceive MBTCC as most useful for system, subsystem, and component tests. The survey findings combined with experiences found in the automotive industry served as a basis for an approach to extend an existing specification method for requirements and testing.

The method is reduced to an adjustable tool chain that is capable of transforming functional requirements modeled as C&C views, ADs, and SCs from various input formats into executable test cases for various output formats. A team of test engineers evaluated 10 test cases and concluded that the SMArDT MBTCC could potentially save efforts up to 50%. In summary, the initial efforts required for convincing all parties to implement our proposed method has resulted in positive feedback. The method, motivated by our survey results and evaluation, has a huge potential. Moreover, the

method improves the cooperation between specifier and tester. Hence, SMArDT is less vulnerable to personnel changes, supported by exchanging knowledge and combining function and test models to avoid redundancies.

Future work comprises the analysis and combination of further modeling languages and formal methods for the model based development in the SMArDT process. Further potential of SMArDT needs to be revealed within large and long-lasting case studies in automotive projects but also in other disciplines calling for a digitalization of their systems engineering processes. Thereby, generative aspects beyond test case extraction need to be investigated in detail as well.

ACKNOWLEDGEMENTS

We would like to thank all participants of the survey. Moreover, we would like to thank the involved test engineers, who supported and evaluated our MBTCC method.

ORCID

Matthias Markthaler  <http://orcid.org/0000-0003-3672-8426>

REFERENCES

1. Ebert C, Favaro J. Automotive software. *IEEE Softw.* 2017;34(3):33-39.
2. Rauch A, Klanner F, Rasshofer R, Dietmayer K. Car2x-based perception in a high-level fusion architecture for cooperative perception systems. Paper presented at: IEEE Intelligent Vehicles Symposium; 2012; Alcalá de Henares, Spain.
3. Prasad B. Analysis of pricing strategies for new product introduction. *Pricing Strategy Pract.* 1997;5(4):132-141.
4. Mohr D, Müller N, Krieg A, et al. The road to 2020 and beyond: what's driving the global automotive industry? *McKinsey Co Automot Assembly Latest Think.* 2013;28(3):2014.
5. *V-Modell XT: Part 1: Fundamentals of the V-Modell.* Technical Report. Berlin, Germany: Federal Government of Germany; 2006.
6. Feiler PH, Gluch DP. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language.* Boston, MA: Addison-Wesley; 2012.
7. Schlegel C, Haßler T, Lotz A, Steck A. Robotic software systems: from code-driven to model-driven designs. Paper presented at: International Conference on Advanced Robotics 2009; Munich, Germany.
8. Selic B. The pragmatics of model-driven development. *IEEE Softw.* 2003;20(5):19-25.
9. France R, Rumpe B. Model-driven development of complex software: a research roadmap. Paper presented at: Future of Software Engineering (FOSE). 2007; Minneapolis, MN.
10. Völter M, Stahl T, Bettin J, Haase A, Helsen S. *Model-Driven Software Development: Technology, Engineering, Management.* Hoboken, NJ: John Wiley & Sons; 2013. *Wiley Software Patterns Series.*
11. Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A. *Model-Based Testing of Reactive Systems.* Berlin, Germany: Springer Science+Business Media; 2005.
12. Dias Neto AC, Subramanyan R, Vieira M, Travassos GH. A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction With the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE); 2007; Atlanta, GA.
13. Pretschner A. Model-based testing. In: Proceedings of the 27th International Conference on Software Engineering; 2005; Saint Louis, MO.
14. Broy M. Challenges in automotive software engineering. In: Proceedings of the 28th International Conference on Software Engineering; 2006; Shanghai, China.
15. Hölldobler K, Rumpe B. *MontiCore 5 Language Workbench Edition 2017.* Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag; 2017.
16. Hillemacher S, Kriebel S, Kusmenko E, et al. Model-based development of self-adaptive autonomous vehicles using the SMARDT Methodology. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development; 2018; Funchal, Portugal.
17. Hamilton MH, Hackler WR. A formal universal systems semantics for SysML. Paper presented at: 17th International Symposium of the International Council on Systems Engineering; 2007; San Diego, CA.
18. Rumpe B, Schulze C, Von Wenckstern M, Ringert JO, Manhart P. Behavioral compatibility of simulink models for product line maintenance and evolution. In: Proceedings of the 19th International Conference on Software Product Line (SPLC); 2015; Nashville, TN.
19. Richenhagen J, Rumpe B, Schloßer A, Schulze C, Thissen K, Von Wenckstern M. Test-driven semantical similarity analysis for software product line extraction. In: Proceedings of the 20th International Systems and Software Product Line Conference (SPLC); 2016; Beijing, China.
20. Bertram V, Roth A, Rumpe B, Von Wenckstern M. Extendable toolchain for automatic compatibility checks. In: Proceedings of the International Workshop on OCL and Textual Modeling (OCL); 2016; Saint Malo, France.
21. Kusmenko E, Shumeiko I, Rumpe B, Von Wenckstern M. Fast simulation preorder algorithm. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development; 2018; Funchal, Portugal.

22. Maoz S, Ringert JO, Rumpe B. Verifying Component and Connector Models Against Crosscutting Structural Views (Extended Abstract). In: *Software Engineering & Management 2015, GI-Edition Lecture Notes in Informatics*. Vol. 239. Bonn, Germany: Bonner Köllen Verlag; 2015:110-111.
23. Rumpe B, Wortmann A. Abstraction and refinement in hierarchically decomposable and underspecified CPS-architectures. In: Lohstroh M, Derler P, Sirjani M, eds. *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Cham, Switzerland: Springer International Publishing; 2018:383-406.
24. ISO. 26262: Road vehicles – functional safety. Geneva, Switzerland. 2011.
25. Bertram V, Maoz S, Ringert JO, Rumpe B, Von Wenckstern M. Case study on structural views for component and connector models. Paper presented at: International Conference on Model-Driven Engineering and Software Development; 2017; Porto, Portugal.
26. Rumpe B. Model-based testing of object-oriented systems. In: *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2003.
27. Pretschner A, Slotosch O, Aiglstorfer E, Kriebel S. Model-based testing for real. *Int J Softw Tools Technol Transf*. 2004;5(2-3):140-157.
28. Kriebel S, Kusmenko E, Rumpe B, Von Wenckstern M. Finding inconsistencies in design models and requirements by applying the SMARTD process. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES); 2018; Germany.
29. MathWorks. Simulink User's Guide. R2018a. MATLAB & SIMULINK; 2018.
30. Taylor RN, Medvidovic N, Dashofy EM. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: Wiley; 2009.
31. OMG Systems Modeling Language (OMG SysML) version 1.4. Needham, MA: Object Management Group; 2015.
32. Modelica Association et al. The Modelica Language Specification. Linköping, Sweden; 2005.
33. Maoz S, Ringert JO, Rumpe B. Synthesis of component and connector models from crosscutting structural views. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13); 2013; Singapore.[§]
34. Maoz S, Ringert JO, Rumpe B. Verifying component and connector models against crosscutting structural views. In: Proceedings of the 36th International Conference on Software Engineering; 2014; Hyderabad, India.
35. Maoz S, Ringert JO, Rumpe B. *An Operational Semantics for Activity Diagrams Using SMV*. Technical Report AIB-2011-07. Aachen, Germany: RWTH Aachen University; 2011.
36. Markthaler M, Kriebel S, Salman KS, et al. Improving model-based testing in automotive software engineering. Paper presented at: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP); 2018; Gothenburg, Sweden.
37. Kansomkeat S, Rivepipoon W. Automated-generating test case using UML state chart diagrams. In: Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology; 2003; Johannesburg, South Africa.
38. Shirole M, Suthar A, Kumar R. Generation of improved test cases from UML state diagram using genetic algorithm. In: Proceedings of the 4th India Software Engineering Conference; 2011; Thiruvananthapuram, India.
39. Vu T-D, Hung PN, Nguyen V-H. A method for automated test data generation from sequence diagrams and object constraint language. In: Proceedings of the Sixth International Symposium on Information and Communication Technology; 2015; Hue, Vietnam.
40. Hettab A, Kerkouche E, Chaoui A. A graph transformation approach for automatic test cases generation from UML activity diagrams. In: Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering; 2015; Yokohama, Japan.
41. Lasalle J, Peureux F, Guillet J. Automatic test concretization to supply end-to-end MBT for automotive mechatronic systems. In: Proceedings of the First International Workshop on End-To-End Test Script Engineering; 2011; Toronto, Canada.
42. Khandai M, Acharya AA, Mohapatra DP. Test case generation for concurrent system using UML combinational diagram. *Int J Comput Sci Inf Technol*. 2011;2(3):1172-1181.
43. Bolton C, Davies J. Activity graphs and processes. In: Grieskamp W, Santen T, Stoddart B, eds. *Integrated Formal Methods: Second International Conference, IFM 2000 Dagstuhl Castle, Germany, November 1-3, 2000 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2000:77-96.
44. Tian J. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Hoboken, NJ: John Wiley & Sons; 2005.
45. Rumpe B. *Modeling With UML: Language, Concepts, Methods*. Cham, Switzerland: Springer International Publishing; 2016.
46. OMG Unified Modeling Language (OMG UML) infrastructure version 2.3 (10-05-03). Needham, MA: Object Management Group; 2010.
47. Sugunasil P. Detecting deadlock in activity diagram using process automata. Paper presented at: International Computer Science and Engineering Conference; 2016; Chiang Mai, Thailand.
48. Rumpe B. *Agile Modeling With UML: Code Generation, Testing, Refactoring*. Cham Switzerland: Springer International Publishing; 2017.
49. De Moura L, Bjørner N. Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008:337-340.
50. Harel D, Rumpe B. Meaningful modeling: what's the semantics of "semantics"? *Computer*. 2004;37(10):64-72.
51. Spillner A, Vossberg K, Winter M, Haberl P. Wie wird in der Praxis getestet? Wie wird in der Praxis getestet? Online-Umfrage in Deutschland, Schweiz und Österreich: Online-themenspecial Testing. *Objektspektrum*. 2011:2011.

[§][Correction added on 19 November 2018, after first online publication: the conference proceedings title in reference 33 has been corrected]

52. Altinger H, Wotawa F, Schurius M. Testing methods used in the automotive industry: results from a survey. In: Proceedings of the 2014 Workshop on Joining Academia and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA); 2014; San Jose, CA.
53. Pretschner A, Prenninger W, Wagner S, et al. One evaluation of model-based testing and its automation. In: Proceedings of the 27th International Conference on Software Engineering (ICSE); 2005; St. Louis, MO.
54. MBTSuite - The Testing Framework. 2017. <http://www.mbtsuite.com/>. Accessed June 11, 2017.
55. Offutt J, Abdurazik A. Generating tests from UML specifications. In: <<UML>>'99 — *The Unified Modeling Language: Beyond the Standard Second International Conference Fort Collins, CO, USA, October 28-30, 1999 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1999:416-429.
56. Yuan J, Wang L, Li X, Zheng G. UMLTGF: a tool for generating test cases from UML activity diagrams based on grey-box method. *J Comput Res Dev*. 2006;1:008.
57. Yue T, Ali S, Zhang M. RTCM: a natural language based, automated, and practical test case generation framework. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis; 2015; Baltimore, MD.
58. Keum C, Kang S, Ko I-Y, Baik J, Choi Y-I. Generating test cases for web services using extended finite state machine. In: *Testing of Communicating Systems: 18th IFIP TC 6/WG 6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2006.
59. Weißleder S, Güldali B, Mlynarski M, et al. Modellbasiertes testen: hype oder realität? *Objektspektrum*. 2011;2011(06):59-65.
60. Utting M, Legeard B. *Practical Model-Based Testing: A Tools Approach*. Burlington, MA: Morgan Kaufmann; 2010.
61. Zander J, Schieferdecker I, Mosterman PJ. *Model-Based Testing for Embedded Systems*. Boca Raton, FL: CRC Press; 2011.

How to cite this article: Drave I, Hillemacher S, Greifenberg T, et al. SMArDT modeling for automotive software testing. *Softw Pract Exper*. 2019;49:301–328. <https://doi.org/10.1002/spe.2650>