Scenarios in the Loop: Integrated Requirements Analysis and Automotive System Validation

Carsten Wiecher
carsten.wiecher@fh-dortmund.de
Dortmund University of Applied Sciences and Arts
Dortmund, Germany

Sergej Japs
sergej.japs@iem.fraunhofer.de
Fraunhofer IEM
Paderborn, Germany

Lydia Kaiser
lydia.kaiser@iem.fraunhofer.de
Fraunhofer IEM
Paderborn, Germany

Joel Greenyer
joel@jgreen.de
Leibniz Universität Hannover
Hannover, Germany

Roman Dumitrescu
roman.dumitrescu@iem.fraunhofer.de
Fraunhofer IEM
Paderborn, Germany

Carsten Wolff
carsten.wolff@fh-dortmund.de
Dortmund University of Applied Sciences and Arts
Dortmund, Germany

ABSTRACT

The development of safety-relevant systems in the automotive industry requires the definition of high-quality requirements and tests for the coordination and monitoring of development activities in an agile development environment. In this paper we describe a Scenarios in the Loop (SCIL) approach. SCIL combines (1) natural language requirements specification based on Behavior-Driven Development (BDD) with (2) formal and test-driven requirements modeling and analysis, and (3) integrates discipline-specific tools for software and system validation during development. A central element of SCIL is a flexible and executable scenario-based modeling language, the Scenario Modeling Language for Kotlin (SMLK). SMLK allows for an intuitive requirements formalization, and supports engineers to move iteratively, and continuously aided by automated checks, from stakeholder requirements to the validation of the implemented system. We evaluated the approach using a real example from the field of e-mobility.

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; **Agile software development**.

KEYWORDS

Automotive Systems Engineering, Requirements Analysis, System Validation, BizDevOps

ACM Reference Format:
Carsten Wiecher, Sergej Japs, Lydia Kaiser, Joel Greenyer, Roman Dumitrescu, and Carsten Wolff. 2020. Scenarios in the Loop: Integrated Requirements Analysis and Automotive System Validation. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3421264

1 INTRODUCTION

In the automotive industry, innovative functions are increasingly defined by software. These functions are today rarely realized by single components. Instead, functions like smart charging may span multiple subsystems [1]. This poses challenges on electronic control unit (ECU) development [40].

New tools and methods are needed that make complexity controllable through appropriate abstraction and which support agile development in order to react quickly to changes by the customer or other stakeholders [11]. Moreover, the special emphasis within the automotive industry regarding functional safety [20] and security [32] have to be considered. This makes it necessary to coordinate and document requirements for system behavior in a form that is understandable to all stakeholders [27]. Likewise, it must be ensured, through validation, that all development artifacts meet the requirements and expectations of the stakeholders.

To meet these challenges, we propose a *Scenarios in the Loop* (SCIL) approach, which takes a holistic view of the process from textual behavior requirements to the comprehensive testing of development artifacts. SCIL takes up the XIL approach established in the automotive industry [15] [9] [28] and extends it by a tool-supported, scenario-based and intuitive modeling of functional system requirements and tests. By formally describing the requirements, they can be executed and automatically analyzed, and are thus part of the test loop (Requirements in the Loop).

SCIL was developed following the design science research (DSR) approach [8]. A central component of SCIL is the Scenario Modeling Language for Kotlin (SMLK), which allows for an intuitive yet executable modeling of system behavior [41]. SMLK is based on the concepts of Live Sequence Charts (LSC) [5], Behavioral Programming (BP) [14] and the Scenario Modeling Language (SML) [12]. Within the SCIL framework, SMLK is integrated with the Behavior-Driven Development (BDD) tool Cucumber [34], which allows the user to create and edit scenario-based natural language requirements using the gherkin syntax[1] and generate test sequences based

[1] https://cucumber.io/docs/gherkin/

on these scenarios. In addition, the SCIL framework integrates other commercial validation tools widely used in the automotive industry [39] [21] [22], which allow automated execution of tests.

The transition from textual descriptions of system behavior to automated tests of the implemented system is realized via three closely coupled process layers, the *Communication and Documentation Layer*, the *Modeling Layer*, and the *Validation Layer*; these layers realize the *BizDevOps* paradigm [30] in the automotive context, enhancing the collaboration of all stakeholders so that software can be released rapidly, frequently, and more reliably.

We evaluated SCIL using an example from the field of e-mobility. Based on a real requirements specification, we performed the individual SCIL steps from scenario-based documentation and intuitive modeling of the system behavior to automated testing of the implemented function.

The lessons learned are the following. (1) The short iterations in the individual process layers can enable agile cross-disciplinary development. (2) The BDD approach integrates the customer more closely into the development process, since the aligned, scenario-based documentation of expected system behavior provides a clear definition of development goals. (3) The close coupling of the individual process layers increases in efficiency for software and system validation, by re-using a test model for the automated validation of development artifacts. (4) The closed and short feedback loops across the individual process layers enable a well-founded analysis of requirements and development artifacts. This analysis is useful for agile project management. The test model mirrors the development goals and is used in the requirements analysis phase to detect contradictions in the requirements. For software and system validation, the test model monitors the implementation status. The closed feedback loops allow for a continuous comparison and concretization of requirements and implementation.

This paper is structured as follows. We describe background in Sect. 2, the SCIL approach in Sect. 3, and a proof-of-concept application in Sect. 4. We evaluate results in Sect. 5, report related work in Sect. 6, and conclude in Sect. 7.

2 BACKGROUND

2.1 Automotive Systems Engineering

The development of software-intensive electronic control units in the automotive industry is highly distributed, cross-disciplinary and increasingly collaborative in new development networks. The basis for the development of often safety-critical systems are different standards (e.g. ISO26262 [20], ISO/SAE 21434 [32]). These must be taken into account by the stakeholders involved in the development, which leads to extensive process implementations in the companies participating. The maturity level of the process implementations is checked by means of assessments [38]. The development processes include dedicated process steps for requirements analysis and validation of development artifacts on the software and system level. Different role definitions exist for the execution of the individual process steps. To illustrate the SCIL approach, the roles *Customer Interface*, *Requirements Owner*, *System Analyst* and *Validation Engineer* are used here, based on the definitions of Sheard [33] and motivated by Holtmann et al. [16]. These roles are used here to describe the SCIL process and can be used as orientation for the integration of the SCIL approach into existing development processes.

Customer Interface (CI): The CI determines the scheduling, technical and financial framework for product development with the customer and other stakeholders. The coordination and concretization of requirements is an important part of this.

Requirements Owner (RO): The RO translates customer needs into clearly formulated requirements and creates an understanding of system boundaries and interfaces. Based on this, the RO evaluates the impact of requirements changes on the system to be developed.

System Analyst (SA): The SA models and simulates the system to evaluate technical decisions and to understand how the system to be developed will behave in its environment.

Validation Engineer (VE): The Validation Engineer checks whether the developed system meets the requirements and expectations of the stakeholders.

2.2 BizDevOps

In agile software development, the DevOps approach [10] is used to foster cross-functional cooperation amongst the developer teams and the deployment teams. The approach defines an iterative process interlinking the development and refinement of software with the deployment and test in an operative environment in a continuous flow. The Plan – Code – Build – Test flow of the development phase (Dev) leads into the Release – Deploy – Operate – Monitor flow of the operation phase (Ops). The results of the Monitor stage in the Ops flow lead to improvement plans for the Dev phase (new Plan stage). DevOps is used for the cooperation of development and test teams, too. To allow the close involvement of the customer and the consideration of the business objectives (Biz), the approach is extended towards the BizDevOps paradigm [30]. In this case, the Monitor stage of the Ops phase leads into a communication and alignment with the customer and/or the business objectives and forms the Adapt – Align – Define – Approve flow (Biz phase). Feedback from the Monitor stage (of the Ops phase) is combined with the latest customer and market feedback in order to adapt the requirements. Stakeholders align based on a joint view on the intended product. This alignment and the definition of the shared vision of the product can be supported with model-based methods, including the scenarios as a common language for technical and business experts. A review and a joint approval of the updated vision (and model) of the product leads into the planning stage of the next Dev phase. The inclusion of the customers and all relevant stakeholders and the alignment of technical result and business view in each iteration are expected to lead to a much deeper involvement of the customer into the development of new software and a much better alignment with the customer requirements. Requirements refinement and prioritization as much as alignment with business objectives are embedded into the design flow. The usage of scenarios and natural language behavioral descriptions supports the involvement and joint understanding of stakeholders from different domains. The definition of acceptance tests based on such scenarios increase the transparency throughout the whole process. A special emphasis is put on the development of joint metrics in BizDevOps which enable a superior delivery transparency for all stakeholders. SCIL is designed to support BizDevOps in the automotive context.

2.3 Model-based Requirements Engineering

The analysis of requirements in the automotive industry is usually done manually and based on informal requirements in natural language [27] [26]. Although model-based development has a positive impact on development quality and efficiency [17, 29], there is no broad acceptance of model-based methods for early and automated requirements analysis in the automotive industry [26]. Thus, potential benefits such as an early simulation and validation of the system behavior [3, 13] or an automated check of the feasibility of requirements, and a synthesis of test cases based on them [31], is not used. Furthermore, manual work for the transition from one phase to the next is required, which is error prone and slow, compared to model-driven automatic generation of artifacts.

Following the argumentation of Liebel [25] there are two ways to increase acceptance of model-based methods for requirements analysis within the automotive industry. On the one hand, the necessary effort for modeling can be reduced, which increases the subjective acceptance of the methods by the users and thus leads to a sustainable application. On the other hand, the benefits that arise from model-based requirements analysis can be increased, for example by deriving information for downstream process steps such as integration and validation. The SCIL approach adresses both.

2.4 Example of Application

For the evaluation of the SCIL framework a function from the field of e-mobility is used. This function shall realize the locking of the charging plug. If a charging plug is connected to the charging socket of the battery electric vehicle (BEV) and a charging process is to be started, the plug must be locked to prevent the plug from being disconnected while the charging process is active. This function is safety relevant according to ISO 26262.

The structure at ECU level is shown in Figure 1. This function is

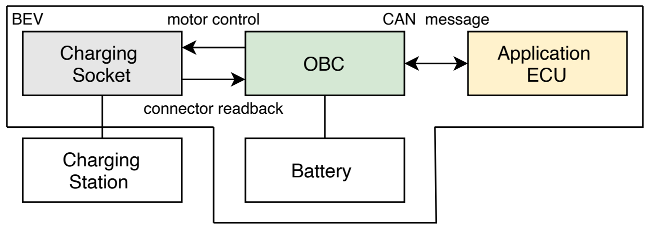


Figure 1: Example of application

implemented by two ECUs; firstly, an Application ECU, which centrally implements high-level functions and provides the interface to other vehicle functions and services, and secondly, an on-board charger (OBC), which is responsible for charging the BEV's battery. Here, the OBC is considered, which directly controls the locking of the charging plug via hardware interfaces (motor control, connector readback). The OBC therefor implements the state based and event driven logic for controlling the locking motor of the charging socket. An extract of the requirements for the function *Plug Lock* to be realized by the OBC is shown in Table 1.

Table 1: Requirements for Plug Lock

ID	Requirement
Req1	If the signal *LockingRequest* reads *"no lock request"* the interlock motor shall not be actuated
Req2	If the signal *LockingRequest* reads *"lock request"* the interlock motor shall be actuated to close the interlock of the plug
Req3	If the signal *LockingRequest* reads *"unlock request"* the interlock motor shall be actuated to open the interlock of the plug
Req4	If the signal *LockingRequest* reads *"no request"* and an actuation is active, this shall be fully finished and not aborted due to value *"no request"*
Req5	When the locking motor reaches an end position, this must be confirmed in order to stop the motor actuation.
Req6	If an actuation of the interlock motor is started and the end position is not confirmed within 2 seconds, the actuation shall be stopped and the interlock motor shall be actuated to open the interlock of the plug

The information that the charging plug should be locked is provided by the Application ECU and is transmitted via the CAN bus to the OBC. In addition to the information whether the charging plug is to be locked or unlocked (Req2 and Req3), there are further requirements in this example which determine that the motor should be stopped when an end position is reached (Req5), which is indicated by the connector readback signal. An error case is also considered if the end position is not reached within a specified timespan (Req6). Moreover, an active process for locking the charging plug should not be interrupted (Req4).

2.5 Scenario Modeling Language for Kotlin (SMLK)

SMLK is used for the formalization and execution of tests and requirements. SMLK is a Kotlin-based framework for scenario-based modeling of behavioral requirements. Kotlin concepts such as higher-order functions, extension functions, channels and coroutines are used to create an internal DSL that enables a concise specification of scenarios. A first application of SMLK is described in [41] and the underlying concepts are described in [14].

In SMLK, several *scenarios* are combined within a *scenario specification* and can be executed as a *scenario program*. As an example, the requirement **Req2** specified as scenario has the form shown in Listing 1.

```
scenario(OBC.LockingRequest("lock request")){ //trigger event
    request(OBC.actuateLockingMotor()) //requested event
}
```

Listing 1: SMLK scenario specification

A scenario can have a *trigger event*. Here it is `OBC.LockingRequest ("lock request")`. When this event occurs, the scenario is *activated*, and the *body* of the scenario, enclosed in curly braces `{...}`, is executed. The body can contain any (Kotlin) code as well as the special statements `waitFor` and `request`, which are also called *sync-points*. The above scenario, after activation, immediately requests the event `OBC.actuateLockingMotor()`.

The events shown above have the form `<object>.<method(<parameters>)>` and are called *object events*. Object events are used for modeling a method call or a signal received by an object (or component instance) of an underlying *object model*. Object events can have side-effects on the object model. Scenarios can read the object model anywhere in the scenario body, but can only make changes through object events.

When a scenario program executes a set of active scenarios, each scenario progresses its execution until all active scenarios reach a sync-point, where each scenario requests an event or waits for an event. The scenario program then chooses a requested event, and notifies all active scenarios that request or wait for that event to resume their execution. Also, new scenarios may be activated when the selected event matches a scenario's trigger event. Scenarios are terminated when they terminate their execution body. This event-selection and -execution cycle is repeated until no event is requested or a special termination event occurs; then the scenario program terminates.

At sync-points, scenarios can also *block* events. Blocked events will not be chosen until the blocking scenarios progress and the event is no longer blocked.

This style of programming and the principle of execution is called *Behavioral Programming* (BP) [14]. With BP, complex system behavior can be described intuitively, and often the behavior of a set of scenarios can be extended or constrained iteratively just by adding scenarios, only sometimes requiring changes in the prior set of scenarios.

SMLK also allows scenario programs to be *open*, which means that they can receive events from external programs, possibly again scenario programs. This way, we can use the same language to also model/program tests and environment models for a given scenario program, even allowing for a test-first scenario-based modeling/programming approach, which we described previously [41].

3 SCENARIOS IN THE LOOP (SCIL)

Based on the context and principles outlined above, this section describes the SCIL approach. The SCIL approach is built on three process layers (*Communication and Documentation Layer*, *Modeling Layer*, *Validation Layer*), which, via suitable tool support, enable an iterative and scenario-based requirements analysis [41] and automated system validation. The individual process steps on the different layers, their relationships and the required or arising artifacts are outlined in Fig. 2. The roles introduced in chapter 2.1 are used to describe the process execution.

3.1 Communication and Documentation (C&D) Layer

The starting point are *stakeholder requirements*, which can exist in any form. A *requirement* in this context is a solution-neutral problem description. The requirement describes *what* is to be developed and is a documented condition or capability that a system must provide or possess (according to [23] and [19]). Requirements are not only influenced by the client (e.g. OEM), but also by persons and organizations not directly involved in the development (e.g. legislators, end customers, training personnel), which are summarized here by the term stakeholder [23].

In the automotive context, customer requirements are usually documented in text form in IBM Doors [18]. These requirements are referenced with relevant standards and technical specifications of other stakeholders, which can be in other formats. Based on these requirements, the RO analyzes the available information regarding system boundaries and derives the necessary interfaces of the system to be realized (1). Based on the system interfaces, *features* to be implemented are defined (2). For each feature a feature file is created, which is used to document the *usage scenarios* (3). A usage scenario describes the expected behavior of the system to realize the feature from the perspective of a user or client of the system by describing conditions and events of a particular execution of the system.

The steps (1-3) are repeated as soon as new information is available, existing requirements are changed, or new requirements are added. Following these steps, features and usage scenarios are added iteratively.

The result of the C&D layer has the form shown in Listing 2. This example shows **Req2** formulated as scenario in the gherkin syntax. For structuring, gherkin provides prepositions and adverbs that precede the textual description (*Given, When, Then, And, But*).

```
1 Feature: plug lock
2
3     @charging
4     Scenario: lock request (Req2)
5         When the signal "LockingRequest" reads "lock request"
6         Then the interlock motor shall be actuated to close
               the interlock of the plug
```

Listing 2: Documenting requirements with gherkin - example based on Req2

In addition, there are further keywords that allow for a grouping of different functionalities (*Feature, Rule*) using *tags*. For example, as shown in line 3, the scenario *lock request* can be assigned to the functionality *charging*, because the parent function battery charging is only executed correctly if the charging plug is locked, i.e. the scenario *lock request* has been executed.

This first part of the SCIL process describes the transition of informal requirements to a structured documentation in feature files including usage scenarios. This is driven by the RO and supported by the SA; the SA carries out the modeling, simulation and analysis of the system behavior on the basis of the resulting usage scenarios. Especially the specification of the individual usage scenarios in (3) is done jointly by RO and SA. This RO/SA collaboration, on the one hand, corrects inaccuracies in the stakeholder requirements and, on the other hand, ensures that the behavioral requirements are correctly described as scenarios, thus providing a common understanding of the system to be developed as a starting point for further modeling and analysis. Inaccuracies and questions that arise when defining the features and concrete usage scenarios are discussed in an early development phase between RO and stakeholders via the CI (10).

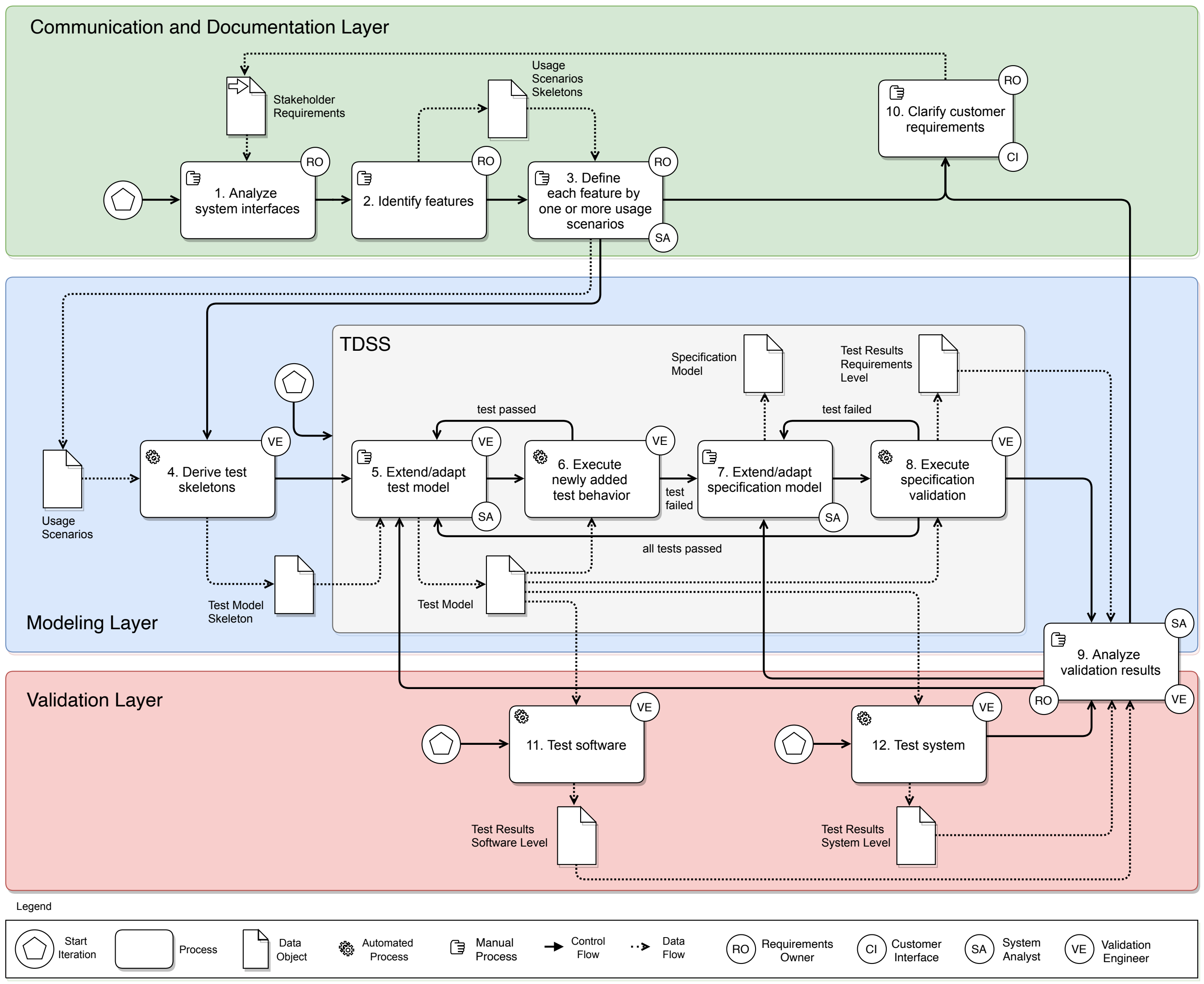

Communication and Documentation Layer
Stakeholder Requirements
Usage Scenarios Skeletons
1. Analyze system interfaces
2. Identify features
3. Define each feature by one or more usage scenarios
10. Clarify customer requirements
RO
SA
CI
Modeling Layer
TDSS
Specification Model
Test Results Requirements Level
test passed
test failed
Usage Scenarios
4. Derive test skeletons
5. Extend/adapt test model
6. Execute newly added test behavior
7. Extend/adapt specification model
8. Execute specification validation
test failed
all tests passed
Test Model Skeleton
Test Model
VE
9. Analyze validation results
Validation Layer
11. Test software
12. Test system
Test Results Software Level
Test Results System Level
Legend
Start Iteration
Process
Data Object
Automated Process
Manual Process
Control Flow
Data Flow
Requirements Owner
Customer Interface
System Analyst
Validation Engineer

Figure 2: Idealized SCIL Process

3.2 Modeling Layer

The clarified usage scenarios of the C&D layer are input to the *modeling layer*. Within the modeling layer, the usage scenarios, supplied in the gherkin syntax (as shown in Listing 2), are used to systematically model SMLK *specification scenarios* (as shown in Listing 1). In contrast to usage scenarios, the SMLK specification scenarios each formalize requirements on all executions of the system and, composed, yield an (over time more and more complete) executable and testable model of the desired system.

```
When("^the signal \"([^\"]*)\" reads \"([^\"]*)\"$") {
      arg0: String, arg1: String ->
      // implement here}
Then("^the interlock motor shall be actuated to close the
     interlock of the plug$") {
      // implement here}
```

Listing 3: Generated test skeleton - based on Req2

As the first step in the modeling layer (4), the VE uses cucumber to automatically derive test skeletons from usage scenarios. As an example, a generated test skeleton for **Req2** has the form shown in Listing 3.

Now we enter the test-driven scenario specification (TDSS) subprocess (as described in previous work [41]). In step (5), the validation engineer completes the generated test skeletons to a test that sends SMLK object events as input to an SMLK program and formulates assertions on the reactions of that SMLK program.

```
When("^the signal \"([^\"]*)\" reads \"([^\"]*)\"$") {
    arg0: String, arg1: String ->
    if ((arg0 == "LockingRequest")&&(arg1 == "lock request"))
        send(obc.LockingRequest(arg1))}
Then("^the interlock motor shall be actuated to close the
     interlock of the plug$") {
    receive(obc.actuateLockingMotor("close"))}
```

Listing 4: Modeling test behavior using SMLK events

For example the test skeleton for **Req2** is enriched as described in Listing 4.

In the *When* section we send the event `obc.LockingRequest("lock request")`. In the *Then* section we expect the response of the scenario program to be `obc.actuateLockingMotor("close")`.

The `receive` function takes as argument an event or a set of events that the scenario program is expected to emit next. The test fails if the scenario program emits another event than one that is expected by `receive`.

In addition to `receive`, tests can also use the function `eventually`, which takes as argument an event or a set of events that the scenario program is expected to emit *at some point in the future*. As an optional argument, `eventually` can be passed an event or a set of events that is forbidden while an expected event did not occur. I.e., on `eventually`, the test repeatedly accepts any event from the scenario program until either an expected event occurs (and then the test progresses) or a forbidden event occurs and the test fails.

The modeling of the test behavior is carried out jointly by VE and SA, to get a common understanding of the expected system behavior.

In (6) the VE triggers the execution of the modeled test behavior. Since the scenario specification model was not extended until now, the test will likely fail. Consequently the SA specifies the expected system behavior and extends the specification model (7).

For example, to get a passed test result for the test shown in Listing 4, the SA has to add the SMLK scenario already shown in listing 1 to the scenario specification. With the last TDSS step the VE triggers all available usage scenarios and the underlying test sequences (8) in order to detect whether the change to the specification model had side-effect on other features.

This TDSS procedure is highly iterative; after several iterations the tests and specification model describes the expected system behavior sufficiently well and can be used as reference for the validation of the implemented function on software and system level.

Therefore, in the *validation layer* the test model created in the modeling layer is used by the VE to trigger automated software tests (11) and system tests (12).

If these tests fail on the validation layer, or when inconsistencies are detected in the modeling layer, the issues are analyzed in (9) jointly by SA, RO and VE. Inaccuracies, false assumption or misunderstandings on requirements and implementation level are highlighted in (9) and must be clarified also by including different stakeholders via the CI (10).

3.3 Validation Layer

For the test of development artifacts, different tools and frameworks are used in the automotive context. To realize an efficient test automation in (11) and (12), we used the tool ContinoProva [22] for our application of SCIL. ContionoProva is a commercial tool that controls other tools (including Vector CANoe [39], iSystems winIDEA [21], National Instruments Labview [36], TheMathworks Matlab/Simulink [37]) via extensible services. In this way, test sequences can be specified consistently and across tools and executed centrally. ContinoProva supports a location-independent test execution via a client/server architecture. A test specification is created and executed via the client. The individual test steps are sent to the ContinoProva server, which then executes them using the connected tools. New tools can be added via the corresponding services, allowing the validation layer to be adapted to the needs of the individual development projects. The basic structure for the technical implementation of the validation layer is outlined in Figure 3.

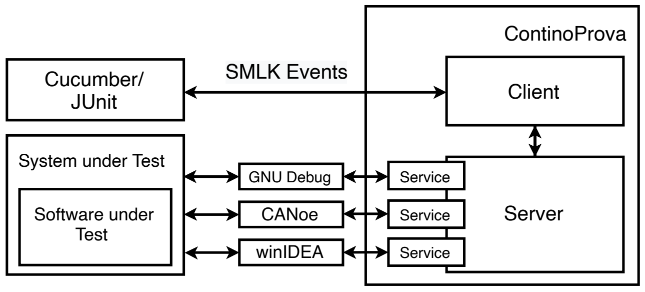


Figure 3: Exchange of events between test model and implemented system

For the application of SCIL, the test specification in the ContinoProva client was replaced by the test model. This means that the test model that was initially used for test-driven requirements analysis can now be used to test development artifacts. For this purpose the ContinoProva client was extended by a WebService. Via this WebService, events from the test model are sent to the ContinoProva client, which then sends concrete test steps to the ContinoProva server, which in turn addresses the different tools via the extensible services, thus enabling the execution of the tests.

The software test is possible via a software simulation environment. The AUTOSAR [2] compliant program code of the ECU function (here Plug Lock) is first compiled using a GCC compiler and can thus be executed within the simulation environment on the development platform on the basis of a GNU debugger. In the further progress of development, the individual ECU functions are integrated into the software system and compiled using a cross-compiler. The software can then be tested on the target ECU using different debug environments. SCIL uses the tool winIDEA [21]. If the software is executed on ECU, a simulation of the vehicle environment is also required to test the integrated software. The signals necessary for the correct operation of the software are provided via the CAN bus or other bus systems. SCIL uses CANoe [39], which is widely used in the automotive industry.

3.4 Summary of the SCIL Results

The iterative approach, from informal stakeholder requirements to the executable specification of scenarios and automated tests of development artifacts supports the application of the BizDevOps paradigm in the automotive industry, whereby the SCIL approach is distinguished from the state of the art in the automotive industry by four points:

- The customer is more closely integrated into product development through a BDD-supported scenario specification. The uniform and tool-supported documentation of behavior in feature files via usage scenarios promotes a common understanding of the functionalities to be implemented and thus creates transparency, which in general supports the Adapt – Align – Define – Approve flow of the Biz phase.

- In conjunction with the test and specification model, the usage scenarios are the basis for an automated requirements analysis and provide important analysis results even before the implementation phase. These analysis results can be discussed with the customer at an early stage, thus supporting an early improvement of initially vague requirements and building the foundation of the Plan – Code – Build – Test flow of the Dev phase.
- On this basis, a clear and joint target picture is created for the client and product developer, which can be checked by the test model iteratively created in the analysis phase. This test model can be used for automated software and ECU testing and thus enables monitoring of the implementation status. In this way, the objective manifested in the test model drives the development and the test case creation does not take place after (waterfall model) or parallel (V-model) to the implementation phase. This provides the foundation for the Release – Deploy – Operate – Monitor flow of the Ops phase.
- In addition, SCIL supports a strongly iterative approach in shortest possible loops within the three process layers. Thus, new system behavior is documented incrementally over further scenarios (C&D layer). The documented behavior is incrementally modeled and automatically analyzed (modeling layer) and the implementation status is automatically validated in short iterations (validation layer). Also the different process layers are strongly connected. The defined usage scenarios bridge the gap between C&D layer and modeling layer (BizDev). The resulting test model connects modeling layer and validation layer (DevOps) and the joint analyzation of validation results by RO, SA and VE leads to valuable feedback from modeling and validation layer to the original stakeholder requirements (BizDevOps).

4 PROOF-OF-CONCEPT APPLICATION OF SCIL

Based on our SCIL implementation and the previously introduced SCIL process, we describe in this section the exemplary application of SCIL in an industrial context at a tier 1 supplier company. This application of the SCIL approach is based on the example introduced in chapter 2.4. This example was initially elaborated together with experts involved in the implementation of the function. The individual SCIL steps were then run through by a member of our team who has no in-depth knowledge of the plug lock function and was not involved in the implementation of the function.

The implementation was largely completed at this point, so it was possible to include also the validation layer in the proof-of-concept study. Thus it was possible to run through the complete chain of test of the development artifact up to the stakeholder requirements and assess how the SCIL approach supports on the individual layers.

The goal of this proof-of-concept study was to assess the feasibility of the SCIL approach. The results were used as a basis to discuss the approach with further experts who were not involved in SCIL development, but in the development of the *Plug Lock* function.

Initially we ran the steps in the C&D layer. Based on the requirements in Table 1 and the described context of the function, three interfaces were identified (1) (see step numbers in Fig. 2). First, the *CAN message*, which is used by the OBC to receive signals from the Application ECU for unlocking or locking the charging plug. Second, the *motor control* output signal, which is used by the OBC to directly control the locking actuator of the charging socket. Third, the value of a *read-back contact* of the connector locking is read in by the OBC, which indicates the locking state.

Based on this structural model, a feature file was created (2) for the function *Plug Lock* and the requirements described in Table 1 were specified as scenarios within this file (3). Result of the activities in the C&D layer are the usage scenarios shown in Listing 5.

```
Feature: Plug Lock
    Scenario: no lock request (Req1)
        When the signal "LockingRequest" reads "no lock
             request"
        Then the interlock motor shall not be actuated
  ...

    Scenario: finish actuation (Req4)
        Given the interlock motor is actuated to close the
             interlock of the plug
        When the signal "LockingRequest" reads "no request"
        Then the interlock motor shall not interrupt the
             active plug lock actuation

    Scenario: confirm end position (Req5)
        Given the interlock motor is actuated to close the
             interlock of the plug
        When the locking motor reaches an end position
        Then this must be confirmed in order to stop the
             motor actuation
  ...
```

Listing 5: Feature Plug Lock in Cucumber

Within these first three steps, it already turns out that the uniform behavior specification can create a common understanding of the expected system behavior. The scenarios defined in (3) characterize a certain behavior under specific conditions in a particular environment or situation, which is supported by specifying the systems interfaces in (1) and defining separate features in (2).

As mentioned before, the implementation of the function was largely completed at the time of evaluation and the requirements specification already contained a very detailed description of the function. A comparison of the already existing requirements with the requirements converted into usage scenarios indicated that the procedure for this function scales and that the structured documentation in usage scenarios supports the communication with the customer (10). One problem was that with a high number of requirements, the dependencies among the scenarios could no longer be overlooked manually. For this reason, we started to dive into the tool supported modeling layer, to start with the test driven behavior modeling and thereby have a better understanding of the interplay of the requirements.

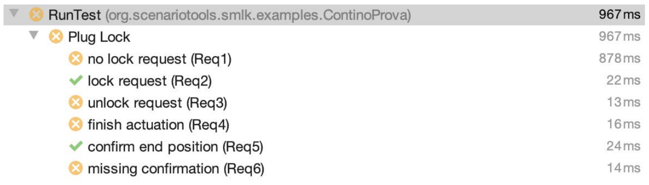
Based on the generated test model skeleton (4), Listing 6 shows the result of the test case modeling (5) for **Req5** after the first TDSS iteration. In the *Given* section the preconditions were specified first. Here the external event `obc.LockingRequest("lock request")` is sent to the scenario program (line 2). This event models the CAN message that is sent by the Application ECU to the OBC.

```
1  Given("^the interlock motor is actuated to close the
       interlock of the plug$") {
2      send(obc.LockingRequest("lock request"))
3      receive(obc.actuateLockingMotor("close"))}
4  When("^the locking motor reaches an end position$") {
5      send(obc.endPositionReached())}
6  Then("^this must be confirmed in order to stop the motor
       actuation$") {
7      receive(obc.lockingMotorStopped())}
```

Listing 6: Test-case for Req5

In line 3 we expect that the scenario program will send an event which indicates that interlock will be closed, whereby the precondition is fulfilled. If this event occurs, we enter the *When* section and send the event `obc.endPositionReached()` (line 6). This event models that the read-back connector indicates that an end position is reached. Next, in line 9 within the *Then* section, we expect that the scenario program stops the motor actuation.

For the first TDSS test run, all tests of the function *Plug Lock*, including the above test-case, showed a negative result. This was expected, because the system behavior had not yet been modeled. Consequently, we modeled the missing system behavior (6) as already described in Sect. 3.2 in order to react according to the behavior required in the test case, which led to the test results shown in Fig. 4. The modeling of the other test cases and the expected system behavior was done in the same way until all test cases showed positive results.



```
▼ ⊗ RunTest (org.scenariotools.smlk.examples.ContinoProva)   967ms
  ▼ ⊗ Plug Lock                                               967ms
      ⊗ no lock request (Req1)                                878ms
      ✓ lock request (Req2)                                    22ms
      ⊗ unlock request (Req3)                                  13ms
      ⊗ finish actuation (Req4)                                16ms
      ✓ confirm end position (Req5)                            24ms
      ⊗ missing confirmation (Req6)                            14ms
```

Figure 4: Execute test behavior (step 8)

However, in parallel to the refinement of the test model and the usage scenarios, the test model could already be used to check the implementation status on the software (11) and system level (12) within the validation layer.

For this purpose we used the tools described in chapter 3.3. The test behavior previously described in Listing 6 was now used to test the AUOTSAR-compliant implementation of the *Plug Lock* function.

Within the validation layer the external events were no longer sent to the specification model, but via the WebService to ContinoProva. The SMLK events of the test model were then sent to the software simulation environment via ContinoProva's tool-specific services. The resulting behavior of the implemented software is shown in Fig. 5.

The figure shows the value changes over time of four variables, and describes the behavior of the software in response to external events of the test model. The first variable `LockingControl` is set to 1 by the test model and signals that there is a request for connector locking. The software responds to this event by setting the `MotorControl` variable to 1, which corresponds to an active locking control. This response corresponds to the behavior described in the `Given` section of the usage scenario (see Listing 6). The locking actuator is now being actively controlled. The next event from the test

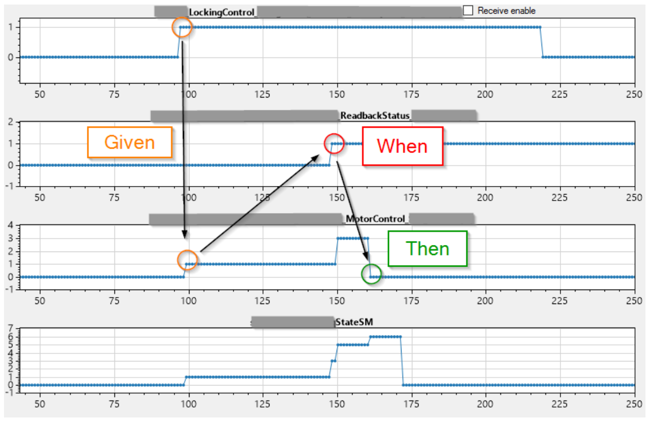


Figure 5: Plot of software simulation (Req5)

model now signals the end position of the connector locking via the variable `ReadbackStatus` (`When`), whereupon the software stops the motor control (`Then`). The variables `LockingControl`, `ReadbackStatus` and `MotorControl` describe the behavior of the software function based on its external interfaces. Where the input signals are set by the test model and the output signal is evaluated by the test model. The fourth variable `StateSM` shows the individual states of the implemented software as reacting to the changing input signals. As this variable represents an implementation specific aspect of the software that is not subject of the requirements description, values of this variable are not tested here.

We could now analyze these test results and compare them with the behavior recorded previously when testing the specification model (step 8). For the test of **Req5** we obtained the same results in the validation layer as before in the modeling layer, which corresponds to the test results shown in Fig. 4.

After this first successful iteration of the overall process, further TDSS iterations in the modeling layer and the validation of the implementation could be continued. After several iterations, all requirements described in Table 1 were sufficiently described via test and specification models and the implementation of the function was shown to be complete via software validation.

Since the implementation of the function already existed, a test of the function integrated on the ECU was also possible. Due to the tools used by the SCIL framework and the resulting decoupling of the test model and tool-specific implementation details, only the mapping of the SMLK events had to be adapted for testing the function integrated on the ECU (see Sect. 3.3). In the ECU, the variable `LockingControl` of the software function is replaced by a CAN signal that is sent to the ECU via ContinoProva, the corresponding service and finally via the CANoe tool.

5 EVALUATION & DISCUSSION

After executing a proof-of-concept iteration of the SCIL process, the approach was discussed with experts on different levels (senior manager, project manager, software engineer) at a tier 1 supplier company. In order to highlight differences to the established approach, difficulties in implementing the function were inquired first. These key points emerged:

- **Vague requirements**: At the start of the software implementation, requirements were available in an under-specified form. In the course of the implementation it turned out that a number of constraints, error cases, etc. were not taken into account and were only further elaborated during the implementation phase.
- **Stakeholder integration**: For the further concretization of the requirements to be implemented, to a large extent further information from the client and other stakeholders was necessary.
- **Manual requirements analysis**: Scenarios were also used here to further concretise the behavior. There is currently a large number of scenarios and there are still contradictions between requirements that were not noticed during the documentation and manual analysis of the requirements and only became clear in the implementation phase.

After going through the SCIL approach with the experts directly involved in the realization of the *Plug Lock* function in different roles, the general feedback was positive: the SCIL framework with the described process layers and with appropriate tool support addresses the existing challenges very well.

Nevertheless, it was unclear whether the SCIL approach can support the development of complex functions in an agile project environment and lead to efficiency improvements. The question arose whether the additional effort for formalizing requirements is reasonable, or whether the established approach with a comparatively late documentation and coordination of the desired system behavior is ultimately just as efficient in this context. For this purpose, further evaluations must be carried out on the basis of this work.

It was also questionable whether the way of formalizing requirements using SMLK, which is supplemented within the SCIL framework with the cucumber tooling, will meet with sustainable acceptance in the automotive sector. In principle, acceptance can be expected if the aforementioned increase in efficiency or quality can be proven. The general acceptance must be examined in further evaluations.

However, at the current point in time we already see that with the technologies and methods combined in SCIL (TDSS, SMLK, BDD, Cucumber), based on [41] [14] [12] [5], a formal, intuitive and iterative development work is promoted. This iterative development is close to the project work in industrial practice. In addition, the test model and the tools used within the validation layer allow automated tests to be performed. There is significant potential for increasing efficiency here compared to downstream test case specification and execution.

6 RELATED WORK

To our best knowledge, there is only one commercial tool (Argosim Stimulus [6]) that takes up the idea of an early automated requirements analysis (Requirements in the Loop). Stimulus enables the simulation of textual requirements in combination with state machines and a component architecture. In contrast to Stimulus, SCIL supports an agile approach based on BDD, which addresses iterative system development in industrial practice and integrates the client more closely into product development. In addition to requirements analysis, Stimulus enables Model in the Loop (MIL) and Software in the Loop (SIL) testing, whereas SCIL also addresses validation at ECU level via the validation layer.

Drave et al. describe a SCIL-like approach [7]. Motivated by the inadequate representation of an agile development process for complex systems by the V-Model, Drave et al. propose a specification method for requirements, design, and test methodology (SMArDT) based on SysML. Here too, different process layers are introduced. Customer requirements are captured via use cases, the behavior is described via activity and sequence diagrams and formalized via object constraint language to analyze requirements and generate test cases. In contrast to SMArDT, SCIL aims to reduce the effort for formalizing requirements by using SMLK along with TDSS and the BDD approach in conjunction with close feedback loops between the process layers. This is to take into account the results of Liebel et al. [26] [27], which show that established model-based methods for requirements analysis (as e.g. object constraint language) in the automotive context do not find broad acceptance by practitioners. Based on this, SCIL follows a more intuitive modeling approach and integrates the customer more closely into the product development.

The approach to describe complex behavior via scenarios is not new. Other work shows that it is beneficial to use scenarios to describe complex system behavior [35] and to validate this behavior based on scenarios [4] [24]. We take up these findings and present the SCIL Framework, a tool with which scenarios can be formalized in an application-oriented manner, thus supporting collaborative development in an cross-disciplinarly automotive context.

7 CONCLUSION AND OUTLOOK

In this paper we describe the SCIL framework. The chosen process layers paired with the used agile methods and appropriate tool support promote cross-disciplinary development work. At the core of this is the continuous consultation and analysis of behavior requirements to create high quality requirements and prevent costly development iterations. At the same time, the test model enables an automated validation of development artifacts, whereby validation is placed at the center of the development. The validation drives the modeling and implementation phase and is not understood as a downstream process step.

With the implementation of the SCIL framework, modeling techniques are used [14] [12] [5] [10] [30], which are not widely adopted in the automotive context so far. In future work, the SCIL framework will be used to further investigate its profits in industrial practice. Especially the questions resulting from the first SCIL application (Sect. 4) and the expert feedback (Sect. 5) shall be investigated by applying SCIL in further development projects. Based on this and the early results described in this paper, we plan to conduct a comprehensive evaluation of the SCIL approach in future work.

REFERENCES

[1] Albert Albers, Armin Kurrle, and Georg Moeser. 2014. Modellbasiertes Anforderungsmanagement von Systems-of-Systems am Beispiel des vernetzten Fahrzeugs. *Tag des Systems Engineering* (2014), 371–382. https://doi.org/10.3139/9783446443761.037

[2] AUTOSAR. 2019. AUTOSAR - Enabling Innovation. https://www.autosar.org/

[3] Christian Brenner, Joel Greenyer, Jörg Holtmann, Grischa Liebel, Gerald Stieglbauer, and Matthias Tichy. 2014. {ScenarioTools} Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study. In *Electronic Communications of the EASST*, Vol. 67. EASST.

[4] Hans Buwalda. 2004. Soap Opera Testing. In *Testing & Analysis*. 30–37.

[5] Werner Damm and David Harel. 2001. {LSCs}: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, Vol. 19. 45–80.

[6] DassaultSystems. 2020. STIMULUS - Requirement simulation – CATIA – Dassault Systèmes®. https://www.3ds.com/products-services/catia/products/stimulus/

[7] Imke Drave, Steffen Hillemacher, Timo Greifenberg, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Software - Practice and Experience* 49, 2 (2019), 301–328. https://doi.org/10.1002/spe.2650

[8] Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes. 2015. *Design science research: A method for science and technology advancement.* 1–161 pages. https://doi.org/10.1007/978-3-319-07374-3

[9] Tobias Düser. 2010. *X-in-the-Loop - an integrated validation framework for vehicle development using powertrain functions*. Ph.D. Dissertation. https://doi.org/10.5445/IR/1000020671

[10] A Dyck, R Penners, and H Lichter. 2015. Towards Definitions for Release Engineering and DevOps. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. 3.

[11] Dorothee Feldmüller. 2018. Usage of agile practices in Mechatronics System Desing - Potentials, Challenges and Actual Surveys. In *2018 19th International Conference on Research and Education in Mechatronics (REM)*. IEEE, 30–35.

[12] Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Florian König, Nils Glade, Assaf Marron, and Guy Katz. 2017. ScenarioTools – A tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming* 149, Supplement C (2017), 15–27. https://doi.org/10.1016/j.scico.2017.07.004

[13] Joel Greenyer, Maximilian Haase, Jörg Marhenke, and Rene Bellmer. 2015. Evaluating a formal scenario-based method for the requirements analysis in automotive software engineering. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* (2015), 1002–1005. https://doi.org/10.1145/2786805.2804432

[14] David Harel, Assaf Marron, and Gera Weiss. 2012. Behavioral programming. *Comm. ACM* 55, 7 (2012), 90–100. https://doi.org/10.1145/2209249.2209270

[15] Jürgen Häring, Joachim Löchner, and Thomas Schöpfner. 2018. Die Zukunft im Griff, Virtualisierte Tests und XiL für automatisiertes Fahren Fahrerassistenzsysteme. *Automobil Elektronik* (2018).

[16] Jörg Holtmann, Ruslan Bernijazov, Matthias Meyer, David Schmelter, and Christian Tschirner. 2016. Integrated and iterative systems engineering and software requirements engineering for technical systems. *Journal of Software: Evolution and Process* 28, 9 (2016), 722–743. https://doi.org/10.1002/smr.1780

[17] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (2014), 144–161. https://doi.org/10.1016/j.scico.2013.03.017

[18] IBM. 2020. IBM Engineering Requirements Management DOORS Family. https://www.ibm.com/ca-en/marketplace/requirements-management

[19] IEEE Standards Board. 1990. *IEEE Standard Glossary of Software Engineering Terminology.* Technical Report. 84 pages.

[20] International Organization for Standardization (ISO). [n.d.]. Road vehicles – Functional safety – Part 6: Product development at the software level, ISO 26262-6:2018.

[21] ISystem. 2020. winIDEA - IDE, Debug and Trace Tool - iSYSTEM - Enabling Safer Embedded Systems. https://www.isystem.com/products/winidea-ide-debug-and-trace-tool.html

[22] ITPowerSolutions. 2020. ContinoProva | ITPS DE. https://itpower.de/de/produkt/continoprova/

[23] Andrej Janzen, Axel Hoffmann, and Holger Hoffmann. 2013. Working Paper Series Chair for Information Systems Anforderungsmuster im Requirements Engineering. (2013).

[24] Cem Kaner. 2003. An Introduction to Scenario Testing. , 10 pages.

[25] G Liebel. 2016. *Model-Based Requirements Engineering in the Automotive Industry: Challenges and Opportunities*. Technical Report. Chalmers University of Technology and Goeteborg University, Goeteborg.

[26] Grischa Liebel, Matthias Tichy, and Eric Knauss. 2019. Use, potential, and showstoppers of models in automotive requirements engineering. *Software and Systems Modeling* 18, 4 (2019), 2587–2607. https://doi.org/10.1007/s10270-018-0683-4

[27] Grischa Liebel, Matthias Tichy, Eric Knauss, Oscar Ljungkrantz, and Gerald Stieglbauer. 2018. Organisation and communication problems in automotive requirements engineering. *Requirements Engineering* 23, 1 (2018), 145–167. https://doi.org/10.1007/s00766-016-0261-7

[28] Udo Lindemann, Albert Albers, Matthias Behrendt, Simon Klingler, and Kevin Matros. 2016. Verifikation und Validierung im Produktentstehungsprozess. *Handbuch Produktentwicklung* (2016), 541–569. https://doi.org/10.3139/9783446445819.019

[29] Parastoo Mohagheghi and Vegard Dehlen. 2008. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Model Driven Architecture – Foundations and Applications*, Ina Schieferdecker and Alan Hartman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 432–443.

[30] Eveline Oehrlich. 2020. What is BizDevOps? | The Enterprisers Project. https://enterprisersproject.com/article/2019/9/devops-what-is-bizdevops

[31] Valerio Panzica La Manna, Itai Segall, and Joel Greenyer. 2015. Synthesizing tests for combinatorial coverage of modal scenario specifications. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015 - Proceedings*. 126–135. https://doi.org/10.1109/MODELS.2015.7338243

[32] Christoph Schmittner, Gerhard Griessnig, and Zhendong Ma. 2018. Status of the Development of ISO/SAE 21434. In *Systems, Software and Services Process Improvement*, Xabier Larrucea, Izaskun Santamaria, Rory V O'Connor, and Richard Messnarz (Eds.). Springer International Publishing, Cham2018, 504–513.

[33] Sarah A. Sheard. 1996. Twelve Systems Engineering Roles. *INCOSE International Symposium* 6, 1 (1996), 478–485. https://doi.org/10.1002/j.2334-5837.1996.tb02042.x

[34] SmartBearSoftware. 2020. BDD Testing & Collaboration Tools for Teams | Cucumber. https://cucumber.io/

[35] A. Sutcliffe. 2003. Scenario-based requirements engineering. *Proceedings of the IEEE International Conference on Requirements Engineering* 2003-Janua, October 2003 (2003), 320–329. https://doi.org/10.1109/ICRE.2003.1232776

[36] TexasInstruments. 2020. What is LabVIEW? - NI. https://www.ni.com/en-ca/shop/labview.html

[37] TheMathworks. 2020. Simulink - Simulation and Model-Based Design - MATLAB & Simulink. https://www.mathworks.com/products/simulink.html

[38] VDA QMC Working Group 13 / Automotive SIG. 2015. Automotive SPICE Process Assessment: Reference Model. (2015), 132. http://www.automotivespice.com/fileadmin/software-download/Automotive{_}SPICE{_}PAM{_}30.pdf

[39] VectorInformatik. 2020. CANoe – ECU & Network Testing | Vector. https://www.vector.com/int/en/products/products-a-z/software/canoe/

[40] Andreas Vogelsang. 2020. Feature dependencies in automotive software systems: Extent, awareness, and refactoring. *Journal of Systems and Software* 160, 2019 (2020), 1–37. https://doi.org/10.1016/j.jss.2019.110458

[41] Carsten Wiecher, Joel Greenyer, and Jan Korte. 2019. Test-Driven Scenario Specification of Automotive Software Components. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (2019), 12–17. https://doi.org/10.1109/MODELS-C.2019.00009