Test-Driven Scenario Specification of Automotive Software Components

Carsten Wiecher
IDiAL Institute
University of Applied Sciences and Arts Dortmund,
Leopold Kostal GmbH & Co. KG
Dortmund, Germany
carsten.wiecher@fh-dortmund.de

Joel Greenyer
Software Engineering Group
Leibniz Universität Hannover
Hannover, Germany
greenyer@inf.uni-hannover.de

Jan Korte
Chair for Embedded Systems of the Information Technology (ESIT)
Ruhr-University,
Leopold Kostal GmbH & Co. KG
Bochum/Dortmund, Germany
jan.korte@rub.de

***Abstract*—The rising complexity of automotive software makes it increasingly difficult to develop the software with high quality in short time. Especially the late detection of early errors, such as requirement inconsistencies and ambiguities, often causes costly iterations. We address this problem with a new requirements specification and analysis technique based on executable scenarios and automated testing. The technique is based on the Scenario Modeling Language for Kotlin (SMLK), a Kotlin-based framework that supports the modeling/programming of behavior as loosely coupled scenarios, which is close to how humans conceive and communicate behavioral requirements. Combined with JUnit, we propose the Test-Driven Scenario Specification (TDSS) process, which introduces agile practices into the early phases of development, significantly reducing the risk of requirement inconsistencies and ambiguities, and, thus, reducing development costs. We overview TDSS with the help of an example from the e-mobility domain, report on lessons learned, and outline open challenges.**

***Index Terms*—Automotive Software Engineering, Requirements Analysis, Software Development, Software Test, Test-Driven Development**

I. Introduction

Software is a key innovation factor in the automotive domain. Due to the rising complexity of automotive software it is increasingly challenging to develop the often safety-critical software with high quality and in short time. New methods and tools are needed to support collaborative, distributed, and incremental development, while maintaining a strong focus on validation and reducing uncertainty during development.

Our contribution to this goal is a new requirements specification and analysis technique based on executable scenarios and automated testing, which enables us to introduce agile practices, in particular test-driven development (TDD), already into the initial development phases. We call this technique the *Test-Driven Scenario Specification* (TDSS) process.

TDSS is based on the *Scenario Modeling Language for Kotlin* (SMLK), a Kotlin-based framework for modeling / programming the behavior of a reactive software component using loosely coupled threads of behavior, called *scenarios*. Scenarios can be executed as a *scenario program*. SMLK inherits the concepts of behavioral programming (BP) [10], Live Sequence Charts (LSC) [4] and the Scenario Modeling Language (SML) [7]. Scenarios can extend as well as constrain the behavior of other scenarios. This allows for a flexible modeling of behavior requirements close to how humans conceive and communicate them. SMLK also supports the modeling of test scenarios, which can be executed as JUnit tests, allowing us to model and execute specific scenario-/requirement interplays.

In TDSS, we assume that we are developing a reactive software component and have a source of requirements (e.g., a document or human stakeholder) that specifies what events or data are the inputs and outputs of that components as well as what the desired relationship of these inputs and outputs is over time. Then, TDSS consists in a repeated process where we take a requirement, and first create a test scenario for it, which is expected to fail. We then add one or several scenarios to the specification in order to formalize the requirement and satisfy the test case. If the test fails, this might reveal a simple modeling problem, or an inconsistency in the requirements modeled thus far. If the test passes, the process is repeated, first with further tests for the same requirement, e.g., covering corner cases, and then the process is repeated for the next requirement. Finally, we obtain an executable requirements- and test specification of the component to be developed.

This process greatly reduces the risk of requirement ambiguities or inconsistencies to survive into later development stages. Moreover, the executable test and requirement specifications are useful in later development tasks, such as testing and failure interpretation. TDSS also allows us to consider fault tolerance concerns during the requirement specification and analysis, by carrying out fault injection tests, e.g., testing invalid input values. These tests are required by functional safety standards like ISO 26262 [3], [13].

We assessed the TDSS approach within an ongoing development project at an automotive tier-1 supplier. In this project an onboard-charger (OBC)[1] is developed. The OBC is

This work was supported by Leopold Kostal GmbH & Co. KG.

[1]https://www.kostal-automobil-elektrik.com/en-gb/produkte/elektronische-steuergeraete/leistungssteuergeraete

an Electronic Control Unit (ECU) of electric vehicles, which is used to convert AC-power from the main to DC-power to charge the vehicle's battery. We applied TDSS to analyze requirements of the function *Derating*, which is responsible for reducing the output power due to thermal reasons.

In our assessment, we went through the software development process in accordance with ASPICE [23], from requirements analysis to model-in-the-loop (MIL) test, and we implemented the software of the derating functionality. As a basis we took real world requirements as they were implemented in the OBC product, and then we added new requirements. It turned out that these requirements contain contradicting statements, which we first discovered during MIL test. In parallel, we ran the development process starting with analyzing the requirements with TDSS, and, in fact, detected the contradicting requirements already in that process. This shows that TDSS has the potential to reduce the development time by starting the implementation phase with a thoroughly analyzed specification.

II. Background

A. Automotive Software Development

Software development in the automotive domain is usually a requirements-driven, model-based approach, where the overall product development is based on the V-model and confirms to the ASPICE and functional safety standards [21]. ASPICE requires a software analysis phase, which usually consists of manual reviews. The requirements that shall be implemented by software are manually transformed into text-based, but formal requirements, e.g., using the MTest Assessable Requirements Syntax (MARS) [20], which is used in the OBC development project at KOSTAL. Based on these formal requirements the software developer builds the software model using Matlab/Simulink. After implementation, a MIL test is used to check the model's behavior. The MIL test requires at least one test sequence and one assessment for each requirement. The test sequence contains input signal vectors to test the behavior regarding the requirement. The assessment is used to verify the model output against the requirements [20]. Because the requirements are written formally, it is possible to derive the assessments automatically from the requirements with state-of-the-art MIL test tools. At KOSTAL, tools like MES Test Manager [19] are used.

B. Derating Example

As an example, we use the function *Derating*, which shall be realized by the OBC. In electric vehicles, the OBC is connected to the car's cooling and high voltage system.

The function *Derating* is responsible for reducing the output power in order to prevent the overheating and destruction of the ECU. The decision to power down is based on the coolant inlet temperature and the power electronics PCB (printed circuit board) temperature. Table I lists 5 basic requirements on the ECU-system level for this function.

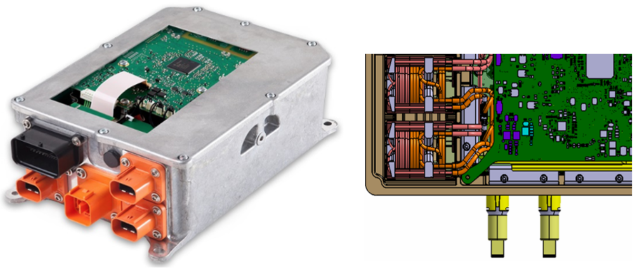


Fig. 1: OBC with high voltage connectors (left) and cooling connectors (right)

TABLE I: Requirements for Derating

ID	Requirement
Req1	No temperature related derating of the available output power shall be commanded, if the coolant inlet temperature sensor reads values between $-40\,°C$ and $65\,°C$.
Req2	If the coolant inlet temperature sensor reads values between $65\,°C$ and $75\,°C$, linear derating with $1/10$ of maximal output power per $1\,°C$ shall be commanded.
Req3	Power-down above $75\,°C$ coolant inlet temperature.
Req4	Power-down if coolant inlet temperature increases more than $5\,°C$ within $5\,s$.
Req5	Power-down if PCB temperature increases more than $20\,°C$ within $3\,s$.

C. Scenario Modeling Language for Kotlin (SMLK)

SMLK is a Kotlin-based framework for modeling the behavior of a software system via loosely coupled threads of behavior, called *scenarios*. A set of scenarios forms a *scenario specification*, which can be executed as a *scenario program*. A scenario program receives a sequence of external events, and can react to each event with one of several events.

SMLK leverages multiple concepts of the Kotlin language, such as coroutines, channels, extensions functions, and higher-order functions, so that scenarios can be modeled concisely.

Inspired by Behavioral Programming [10], each SMLK scenario is a thread of behavior (a Kotlin coroutine) that, by the help of the available programming idioms, can specify a sequence of *sync points* where a special `sync` method is called. At these points, a scenario can specify events that it *requests*, *waits-for*, and/or *forbids*, with the constraint that external events are never requested.

Let us assume a scenario program execution where initially no scenario is active (there might be, for initialization purposes), and where the program waits for an external event. This event may trigger scenarios that progress to their first sync point. If then there are requested events, a central event selection algorithm chooses one of these events that is not currently forbidden, executes this event, and notifies all scenarios that requested or waited-for that event. The notified scenarios then proceed to their next sync point, and the process is repeated until no further events are requested. Then the next external event is processed, and so on. If at some point during execution, all requested events are also forbidden, the scenario program terminates and reports the conflict.

In addition to calling `sync` directly, there are convenience methods for requesting and waiting-for events (`request` and `waitFor`), which in turn call `sync`. Events that are forbidden across several sync-points or events that shall interrupt the scenario, can be added to and removed from the sets **`forbiddenEvents`** and **`interruptingEvents`**, respectively.

Events can take different forms, but one specific concept provided by SMLK, is that of an *object event*, which is an event that models the method call or a signal received by an object or component, and which can also have a side effect on the receiving object's properties. Listing 1 shows how object events are declared and created: the function `setDeratingFactor` of the `DeratingComponent` class calls the `event` function, which creates an object event representing that specific method call, with the parameter value passed to the method. A lambda function (within curly braces) can be passed to the `event` function[2] to model the side-effect that the event has upon its execution. Here, the side-effect is setting the attribute **`deratingFactor`** to the value of the passed parameter. The object events `startCycle` and `endCycle` are declared similarly, but have no parameters nor side-effects.

```
class DeratingComponent {
 var coolantTemp = 0 // in degrees celsius
 var deratingFactor = 0.0 // values [0..1]
 fun setDeratingFactor(factor : Double) = event(factor){
      deratingFactor = factor}
 fun startCycle() = event{}
 fun endCycle() = event{}
 ...
}
```

Listing 1: Declaring object events in SMLK

Listing 2 shows how to model a scenario that initializes on each occurrence of the object event `startCycle` received by the object `deratingComponent`. If now the **`coolantTemp`** of `deratingComponent` is in the range of -40 to 65, the scenario reaches a sync point where it requests the execution of `deratingComponent.setDeratingFactor(1.0)`. This means that no derating should occur in that temperature range. This models the requirement **Req1** (cf. Tabl. I).

```
scenario(deratingComponent.startCycle()){
 if (deratingComponent.coolantTemp in -40..65)
  request(deratingComponent.setDeratingFactor(1.0))
}
```

Listing 2: First attempt of formalzing Req1 with SMLK

The `startCycle` and `endCycle` events here are used to specifically model the behavior of software components that are executed in cycles, where they read inputs, execute, and write outputs in regular intervals, as is the case for many embedded and automotive software systems. We assume that `startCycle` is called regularly by an external run-time, then the software component executes and ends its computation before the next cycle starts.

In order to formulate requirement scenarios in that setting, we define the function *`cycleScenario`* as shown in Listing 3, which allows us to concisely create scenarios that model execution cycle requirements. The function takes two arguments: first, the `deratingComponent`, and, second, a scenario lambda `mainScenario`. `mainScenario` is invoked between `startCycle` and `endCycle`, and `startCycle` and `end-Cycle` are forbidden to occur (by adding them to **`forbiddenEvents`**) during the execution of `mainScenario`.

```
fun cycleScenario(deratingComponent : DeratingComponent,
      mainScenario : suspend Scenario.() -> Unit) : suspend
      Scenario.() -> Unit =
scenario(deratingComponent.startCycle()){
 forbiddenEvents.add(deratingComponent.startCycle())
 forbiddenEvents.add(deratingComponent.endCycle())
 mainScenario.invoke(this@scenario)
 request(runnable.endCycle())
}
```

Listing 3: A function for creating scenarios that express requirements of an execution cycle

Now we can formalize the **Req1** as show in Listing 4:

```
cycleScenario(deratingComponent){
 if (deratingComponent.coolantTemp in -40..65)
  request(deratingComponent.setDeratingFactor(1.0))
}
```

Listing 4: Modeling Req1 as a cycleScenario

The scenarios as shown above can be added to a scenario program; we omit details for brevity. For testing scenario programs, SMLK supports modeling test scenarios and executing them as JUnit tests. In test scenarios, external events are requested, and events from the scenario program to be tested can be waited-for. The test passes when both the test and the scenario-program under test terminate without violations. Examples are shown in the next section.

III. Test-Driven Scenario Specification

TDSS combines formal scenario-based specification and analysis with TDD [6]. TDD was originally designed for software unit testing, where tests are written and executed before writing the program code. In small iterations tests and code are extended and adapted following the red/green/refactor approach [2]. The first step is writing a new test to validate a new functionality, which is expected to fail initially (red). In the second step the implementation must be extended or adapted until the tests pass (green). The third step (refactor) is cleaning up the code, without changing the functionality. We propose the TDSS process as shown in Fig. 2.

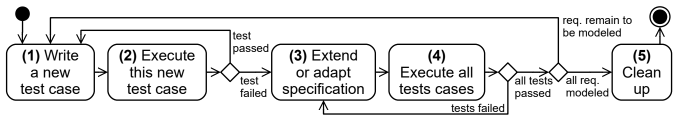

(1) Write a new test case
(2) Execute this new test case
test passed
test failed
(3) Extend or adapt specification
(4) Execute all tests cases
all tests passed
tests failed
req. remain to be modeled
all req. modeled
(5) Clean up

Fig. 2: Test-Driven Scenario Specification (TDSS) Process

1) **Write a new test case:** Take a stakeholder requirement and write a test for that requirement.
2) **Execute this new test case:** We expect the test to fail as the requirement is not yet specified as an executable scenario. If the test passes, it is a special case where the requirement is already covered by existing scenarios.

[2]In Kotlin, a lambda passed as the last function parameter can be written outside of the parentheses, see https://kotlinlang.org/docs/reference/lambdas.html#passing-a-lambda-to-the-last-parameter

3) **Extend or adapt specification:** Formally model the requirement. Possibly, this requires adapting existing scenarios in order to resolve inconsistencies.
4) **Execute all test cases:** Run all tests in order to validate the interplay of all scenarios and, hence, the interplay of all the requirements, specified thus far.
5) **Clean up scenarios and document requirements:** Clean the specification code and update, if it exists, the corresponding text-based requirement description. The result is a formal requirement and test specification as the basis for subsequent development activities.

As an example of the TDSS process, let us consider the test-first specification of **Req2**. As a test case, we can choose one particular and simple run that covers this requirement, for example where the coolant inlet temperature is 70. This should result in a derating factor of 0.5 to be set within the next execution cycle. Listing 5 shows how the test looks like.

```
1 @Test
2 fun ‘At 70 degree the DeratingFactor is 0-5‘()
3  = runTest(deratingScenarioProgram){
4  forbiddenEvents.add(deratingComponent receives DeratingComponent::
       setDeratingFactor)
5  request(deratingComponent.setCoolantTemperature(70))
6  request(deratingComponent.startCycle())
7  waitFor(deratingComponent.setDeratingFactor(0.5))
8  waitFor(deratingComponent.endCycle())
9 }
```

Listing 5: JUnit test implementation

In line 4, we forbid any **setDeratingFactor** event from occurring, except where and with what parameter value it is explicitly waited for later on. In line 5, we set the coolant temperature value. In line 6 we start the cycle and then (line 7) we expect the derating factor to be set to 0.5. Finally, we wait-for the derating component to end its current execution cycle.

Because it is tedious to always model each step in the process of (1) setting input values, (2) starting the cycle, (3) expecting output values to be set, and (4) waiting for the end of the cycle, we extract this code in a function `deratingIOSequence`, which does this for us and just receives two input values (coolant inlet temperature and pcb temperature) and one output value (derating factor). Listing 6 shows how the test can be written now in a shorter form:

```
@Test
fun ‘At 70 degree the DeratingFactor is 0-5‘() = runTest(
      deratingScenarioProgram){
 deratingIOSequence(Triple(70, 70, 0.5))
}
```

Listing 6: Simplified JUnit test implementation

After running this test (Step 2) we see that the test failed, of course, because the requirement is not modeled yet. After adding the cycle scenario shown in Listing 7 (Step 3) all tests written so far are executed (Step 4) and pass.

```
// Derating factor derates linearly from 1.0 to 0.0 when coolant
      temperature is between 65 and 75
cycleScenario(deratingComponent){
if (deratingComponent.coolantTemp in 65..75)
request(deratingComponent.setDeratingFactor((75.0-deratingComponent
      .coolantTemp)/10))
}
```

Listing 7: Scenario for derating within a specific temperature range

These steps can be repeated with further tests for the same requirement, for example testing corner cases at 65 and 66 degrees coolant inlet temperature.

Let us assume that we repeated the process also for **Req1** and **Req3**, and we now consider **Req4**. In contrast to the requirements modeled before, this requirement specifies how to deal with temperature changes over a period of time.

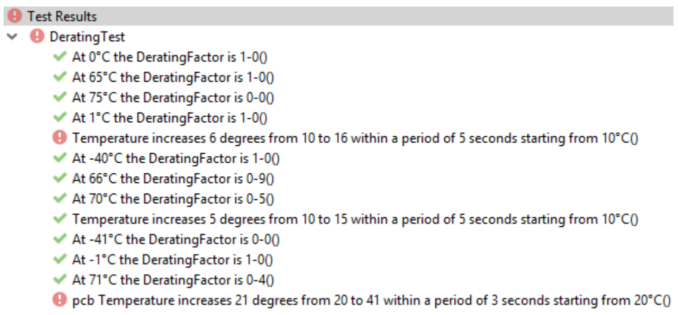
We write and execute tests as before (Steps 1 and 2), and see them fail. Then (Step 3) we model **Req4** as shown in Listing 8, as a scenario that starts when the coolant temperature is set. We then read temperature parameter value from this initial event (line 5) and initialize a variable `currentTemp` that we update with the coolant temperature values (line 21) for the next 50 cycles, which corresponds to a duration of 5 seconds. If `currentTemp` deviates from the initial temperature more than 5 degrees, then we request to execute an exceptional temperature increase event (line 16), as well as to set the derating factor to `0.0` (line 17). Events are added to **`forbiddenEvents`** (lines 9, 10, 15) to indicate that they must only occur unless specifically requested or waited-for, i.e., the scenario is strict about where execution cycles start and end, and in the case of an exceptional temperature increase, does not allow setting the derating factor to values other than `0.0`.

```
1  // Shutdown if coolant temp increases more than 5 degrees within a
       period of 5 seconds
2  scenario(deratingComponent receives DeratingComponent::
       setCoolantTemperature){
3
4   // read new coolant temperature value
5   val initialTemp = it.parameters[0] as Int
6   var currentTemp = initialTemp
7
8   // allow cycles’ start/end only where waited for below.
9   forbiddenEvents.add(deratingComponent.startCycle())
10  forbiddenEvents.add(deratingComponent.endCycle())
11
12  for(i in 1..50){ // for 50 cycles = 5 seconds...
13   waitFor(deratingComponent.startCycle())
14   if (currentTemp - initialTemp > 5){
15    forbiddenEvents.add(deratingComponent receives DeratingComponent
          ::setDeratingFactor)
16    request(deratingComponent.exceptionalTemperatureIncrease())
17    request(deratingComponent.setDeratingFactor(0.0))
18    break // end the for loop and, hence, the scenario
19   }
20   waitFor(deratingComponent.endCycle())
21   currentTemp = waitFor(deratingComponent receives DeratingComponent
          ::setCoolantTemperature).parameters[0] as Int
22  }
23 }
```

Listing 8: Scenario to shutdown on sudden temperature increases

Note that new activations of this scenario are created with every cycle, when the coolant temperature is set. So, up to 50 active instances of this scenario may execute at a time. This is not how we would implement this behavior in the final software, but it is an intuitive formalization of the requirement.

With this scenario included, we run all tests again (Step 4) and see failed tests (see. Fig. 3). The reason for this failure is a contradiction in requirements **Req1** and **Req4**: In **Req1**, we do not expect the output power to derate when we measure a temperature in the -40 to 65 degree range, but in **Req4**, we expect the output power to be disabled when we measure a 5 degree increase in coolant inlet temperature within 5 seconds. Hence disabling of the output power could also happen in the temperature range given in **Req1**. This leads to a violation while running the scenarios, since different values for the derating factor are expected in the tests.



```
Test Results
  DeratingTest
    At 0°C the DeratingFactor is 1-0()
    At 65°C the DeratingFactor is 1-0()
    At 75°C the DeratingFactor is 0-0()
    At 1°C the DeratingFactor is 1-0()
    Temperature increases 6 degrees from 10 to 16 within a period of 5 seconds starting from 10°C()
    At -40°C the DeratingFactor is 1-0()
    At 66°C the DeratingFactor is 0-9()
    At 70°C the DeratingFactor is 0-5()
    Temperature increases 5 degrees from 10 to 15 within a period of 5 seconds starting from 10°C()
    At -41°C the DeratingFactor is 0-0()
    At -1°C the DeratingFactor is 1-0()
    At 71°C the DeratingFactor is 0-4()
    pcb Temperature increases 21 degrees from 20 to 41 within a period of 3 seconds starting from 20°C()
```

Fig. 3: Test results with inconsistent requirements

The stakeholders must now resolve this inconsistency. Let us assume that **Req4** is meant to be a special case of, and shall have priority over, **Req1**. That is, the output power shall be reduced if an exceptional temperature increase occurs. To resolve this inconsistency in the scenarios, we add `exceptionalTemperatureIncrease` as interrupting event in the scenarios of requirements **Req1** and **Req2**, see Listing 9.

```
// Derating factor derages linearly from 1.0 to 0.0 between
      [65..75]
cycleScenario(deratingComponent){
interruptingEvents.add(deratingComponent.
      exceptionalTemperatureIncrease())
if (deratingComponent.coolantTemp in 65..75)
request(deratingComponent.setDeratingFactor((75.0-deratingComponent
      .coolantTemp)/10))
}
```

Listing 9: Scenario from Listing 7 with added interrupt on exeptional temperature increase

After executing all tests again (Step 4), we see that all tests pass. After treating also Req5, we clean up and update the textual requirements. We also translate the requirements in the MARS syntax, so they can be used as test assessments later on (cf. Sect. II-A). Listing 10 shows this exemplary for **Req2**.

```
WHILE signal CoolantTemperature is greater than 65 and signal
      CoolantTemperature is less than 75 and from when the system
      starts until when diff(signal CoolantTemperature,parameter
      CoolantInitTemperature) becomes a value greater than 0.01 or
      when diff(signal PcbTemperature,parameter PcbInitTemperature)
       becomes a value greater than 0.0667 or when signal
      CoolantTemperature becomes greater than 75 THE signal
      DeratingFactor SHALL be equal to parameter DeratingOffset +
      parameter DeratingSlope * signal CoolantTemperature
}
```

Listing 10: Extended requirement Req2 for MIL test

IV. Evaluation

In parallel to the TDSS process, one member of out team went through the classical automotive software development process outlined in Sect. II-A with the task to incorporate the new requirements **Req4** and **Req5**. He was not made aware of the requirement inconsistency. After analyzing the requirements, he implemented and tested the new functionality.

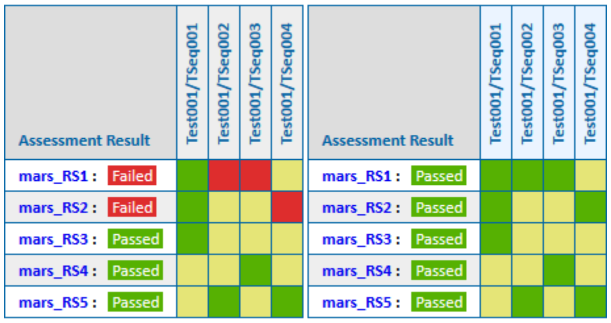
This team member formalized **Req2** in MARS as documented in Listing 11 and specified and executed the following four MIL tests of the implemented (Simulink) model:

- **Tseq001:** Tests the requirements **Req1** to **Req3** by slowly increasing the coolant temperature from 50 $°C$ to 100 $°C$.
- **Tseq002:** Tests the requirements **Req5** by fast increasing the pcb temperature at a coolant temperature of 50 $°C$.
- **Tseq003:** Tests the requirements **Req4** by fast increasing the coolant temperature at a pcb temperature of 50 $°C$.
- **Tseq004:** Tests the requirements **Req5** by fast increasing the pcb temperature at a coolant temperature of 70 $°C$.

```
WHILE signal CoolantTemperature is greater than 65 and signal
      CoolantTemperature is less than 75  THE signal DeratingFactor
       SHALL be equal to parameter DeratingOffset + parameter
      DeratingSlope * signal CoolantTemperature
}
```

Listing 11: Unextended requirement Req2 for MIL test

The MIL test results are shown on the left of Fig. 4. The requirements **Req1** and **Req2** are failing in the test sequences 2 to 4, which are testing the power down ability of the component. This happens because of the contradictory requirements, as described above. Only after resolving this conflict and repeating the requirement analysis, implementation, and testing steps, was the conflict resolved (see Fig. 4, right side).



Assessment Result		Test001/TSeq001	Test001/TSeq002	Test001/TSeq003	Test001/TSeq004
mars_RS1 :	Failed	🟩	🟥	🟥	🟨
mars_RS2 :	Failed	🟩	🟨	🟨	🟥
mars_RS3 :	Passed	🟩	🟨	🟨	🟨
mars_RS4 :	Passed	🟨	🟨	🟩	🟨
mars_RS5 :	Passed	🟨	🟩	🟨	🟩

Assessment Result		Test001/TSeq001	Test001/TSeq002	Test001/TSeq003	Test001/TSeq004
mars_RS1 :	Passed	🟩	🟩	🟩	🟨
mars_RS2 :	Passed	🟩	🟨	🟨	🟩
mars_RS3 :	Passed	🟩	🟨	🟨	🟨
mars_RS4 :	Passed	🟨	🟨	🟩	🟨
mars_RS5 :	Passed	🟨	🟩	🟨	🟩

Fig. 4: MIL test results with contradictory requirements on the left and fixed requirements on the right.

The result of this small experiment is that, by using TDSS, we were successfully able to detect contradicting requirements already before the software model is implemented and tested. In this experiment, the time and effort of applying TDSS vs. fixing the inconsistency later in the standard development process was similar. We plan further case studies to investigate whether TDSS can reduce the development time and effort.

V. Related Work

Attempts to combine TDD with requirements specification are not new. Most approaches, however, propose using tests as means to specify natural language requirements more precisely [11]. Behavior-Driven Development (BDD) is a similar agile practice that proposes to use concrete behavior examples in order to focus the development. Lettrari and Klose [16] describe an approach for using UML sequence diagrams to model monitors and tests of real-time requirements.

Jones [14] describes a test-first approach for a specification in the form of a decision table. In decision tables, however, it is difficult to formalize requirements over multiple steps.

In the context of scenario-based specifications akin LSC, there exists the idea to use existential scenarios as test descriptions [9], [17], [22]. Our work follows this idea and presents a particular technology that supports and showcases its application in the automotive domain.

This work extends previous work of evaluating scenario modeling and automated consistency checks in the automotive domain [8]. In contrast to that work, this paper focuses on test-driven requirements validation, which we see as an often more practical and pragmatic approach than doing computationally expensive consistency proofs and interpreting their results.

VI. Conclusion and Outlook

In this paper, we presented TDSS, a new approach for the test-driven, scenario-based requirement specification and analysis, combining agile practices with formal specification and analysis. This paper is also the first to showcase SMLK, which we used to implement and evaluate TDSS.

We performed a small experiment that shows the applicability and benefits of TDSS, as we could successfully detect a requirement inconsistency using TDSS, which was initially overlooked when following a classical development process.

Further results of our assessment are that scenario-based specification, in the form of SMLK, is indeed suited to formalize functional requirements in an intuitive way. Simple requirements can be modeled concisely. More complicated requirements require more detailed thinking and modeling, but can still be modeled close to how the requirements are formulated. This brings two psychological and factual benefits: The ability to immediately test the modeled requirements brings high confidence and a feeling of control to the requirements specification and analysis phase. When the final implementation is tested, it brings a high confidence of its correctness as there is an automated three-way validation of the developed behavior: tests vs. requirements vs. implementation.

We could also show that the modeled requirements can easily be reused as assessments inside component tests. This highly reduces the effort in later component tests.

In future work we plan to address further test case specification challenges [15], e.g., reducing the test specification effort by synthesizing tests automatically based on combinatorial test design [18]. We also plan to conduct further case studies, in order to more systematically assess the time and effort reduction that TDSS might bring. We will also investigate how TDSS can be combined with integrated and iterative systems engineering methodologies [12] and agile system engineering methods and practices [1], [5].

References

[1] A. Albers, J. Heimicke, M. Spadinger, N. Reiß, J. Breitschuh, T. Richter, N. Bursac, and F. Marthaler. Eine systematik zur situationsadäquaten mechatroniksystementwicklung durch asd - agile systems design. Technical report, Karlsruher Institut für Technologie (KIT), 2019.

[2] K. Beck. *Test-driven development: by example*. Addison-Wesley, 2003.

[3] D. Cotroneo and R. Natella. Fault injection for software certification. *IEEE Security Privacy*, 11(4):38–45, July 2013.

[4] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80, 2001.

[5] D. Feldmüller. 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)*, pages 30–35. IEEE, 2018.

[6] S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, and C. Poole. Test driven development (tdd). In *Marchesi M., Succi G. (eds) Extreme Programming and Agile Processes in Software Engineering. XP 2003. Lecture Notes in Computer Science, vol 2675. Springer, Berlin, Heidelberg.* Springer, 2003.

[7] J. Greenyer, D. Gritzner, T. Gutjahr, F. König, N. Glade, A. Marron, and G. Katz. Scenariotools – a tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming*, 149(Supplement C):15 – 27, 2017. Special Issue on MODELS'16.

[8] J. Greenyer, M. Haase, J. Marhenke, and R. Bellmer. Evaluating a formal scenario-based method for the requirements analysis in automotive software engineering. In *Proc. 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2013, 2015.

[9] D. Harel, H. Kugler, and G. Weiss. Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. In S. Leue and T. J. Systä, editors, *Scenarios: Models, Transformations and Tools*, pages 26–42, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Comm. ACM*, 55(7):90–100, 2012.

[11] D. Hoffman and P. Strooper. Prose + test cases = specifications. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, pages 239–, Washington, DC, USA, 2000. IEEE Computer Society.

[12] J. Holtmann, R. Bernijazov, M. Meyer, D. Schmelter, and C. Tschirner. Integrated and iterative systems engineering and software requirements engineering for technical systems. In *Journal of Software Evolution and Process, Special Issue on International Conference on Software and Systems Process 2015*, 2015.

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

[14] E. L. Jones. Test-driven specification: Paradigm and automation. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 796–797, New York, NY, USA, 2006. ACM.

[15] K. Juhnke, M. Tichy, and F. Houdek. Poster: Challenges with automotive test case specifications. In *2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings*, pages 131–132. IEEE, 2018.

[16] M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time uml models. In M. Gogolla and C. Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 317–328, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[17] L. Li, H. Gao, and T. Shan. An executable model and testing for web software based on live sequence charts. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, June 2016.

[18] V. P. L. Manna, I. Segall, and J. Greenyer. Synthesizing tests for combinatorial coverage of modal scenario specifications. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 126–135, Sept 2015.

[19] Model Engineering Solutions. MES Test Manager (MTest). `https://model-engineers.com/de/quality-tools/mtest/`, 2019. last access: 6.2019.

[20] Model Engineering Solutions. MES Test Manager (MTest) - overview assessment framework. `https://model-engineers.com/de/quality-tools/mtest/?file=files/upload/quality-tools/mtest/mes_test_manager_flyer.pdf`, 2019. last access: 6.2019.

[21] H.-L. Ross. *Functional Safety for Road Vehicles*. Springer International Publishing, 2016.

[22] G. Sibay, S. Uchitel, and V. Braberman. Existential Live Sequence Charts Revisited. In *Proceedings of the ACM/IEEE 30th Int. Conference on Software Engineering ICSE '08*, pages 41–50, 2008.

[23] VDA QMC Working Group 13 / Automotive SIG. Automotive spice - process reference model - process assessment model version 3.1. http://www.automotivespice.com/fileadmin/software-download/AutomotiveSPICE_PAM_31.pdf, 2017.