Iterative and Scenario-based Requirements Specification in a System of Systems Context

Carsten Wiecher[1][0000−0002−3280−4471], Joel Greenyer[2][0000−0003−0347−0158],
Carsten Wolff[1][0000−0003−3646−5240], Harald Anacker[3], and Roman Dumitrescu[3]

[1] Dortmund University of Applied Sciences and Arts, 44139 Dortmund, Germany
`firstname.lastname@fh-dortmund.de`
[2] FHDW Hannover, 30173 Hannover, Germany
`joel.greenyer@fhdw.de`
[3] Fraunhofer IEM, 33102 Paderborn, Germany
`firstname.lastname@iem.fraunhofer.de`

Abstract. [Context & Motivation] Due to the managerial, operational and evolutionary independence of constituent systems (CSs) in a System of Systems (SoS) context, top-down and linear requirements engineering (RE) approaches are insufficient. RE techniques for SoS must support iterating, changing, synchronizing, and communicating requirements across different abstraction and hierarchy levels as well as scopes of responsibility. [Question/Problem] We address the challenge of SoS requirements specification, where requirements can describe the SoS behavior, but also the behavior of CSs that are developed independently. [Principal Ideas] To support the requirements specification in an SoS environment, we propose a scenario-based and iterative specification technique. This allows requirements engineers to continuously model and jointly execute and test the system behavior for the SoS and the CS in order to detect contradictions in the requirement specifications at an early stage. [Contribution] In this paper, we describe an extension for the scenario-modeling language for Kotlin (SMLK) to continuously and formally model requirements on SoS and CS level. To support the iterative requirements specification and modeling we combine SMLK with agile development techniques. We demonstrate the applicability of our approach with the help of an example from the field of e-mobility.

Keywords: System of Systems Engineering · Requirements Analysis · Requirements Specification · Scenario-based Requirements Modeling

1 Introduction

New methods and tools are needed to meet the challenges in the development of complex socio-technical systems, such as sustainable mobility solutions in metropolitan regions [25]. Systems of connected electrified vehicles can be characterised as a *system of systems* (SoS), where the vehicle can be seen as a *constituent system* (CS) that interacts with changing other CSs to provide an SoS functionality [17].

An interdisciplinary approach for the realization of these systems is *system of systems engineering* (SoSE). The definition of stakeholder needs and required functionalities are key elements of SoSE [20]; the precise specification of requirements is a basis

for the system decomposition and implementation, or the selection of suitable CSs that form an SoS [27]. However, in SoSE, there are different requirements engineering (RE) challenges compared to RE in established systems engineering (SE) processes [26].

According to Maier et al. [23], the operational, managerial, and evolutionary independence are the essential characteristics of an SoS. These characteristics have a significant influence on the applicability of existing RE techniques [24–26]. In contrast to monolithic systems, SoS consist of individual systems that can operate independently and perform a meaningful task, even when not part of an SoS. The development and operation of the CSs is managed independently, in different organizations with different development- and product life cycles. Also, requirements on the CS- and SoS level change frequently and independently, leading to an evolutionary development [26, 23].

Based on these SoS characteristics, Ncube and Lim [24] describe challenges for the SoS RE process: Due to the different systems in an SoS, requirements cover many different disciplines, can be contradictory, unknown or possibly not fully defined. These difficulties overlap with the fundamental problems in RE [7], but, according to Ncube and Lim [24], requirements in an SoS additionally must be considered as requirements for the SoS, which describe the properties of the overall system, or requirements for a CS that describe capabilities of a single system. Since requirements on both levels can change continuously and independently, traditional, linear and top-down requirements specification and decomposition techniques can not be used [24–26].

To address this problem we propose an iterative and scenario-based requirements specification technique. Based on previous work [30, 32, 33] we integrate the Scenario Modeling Language for Kotlin (SMLK) with agile development techniques to support the requirements engineer in the continuous and iterative specification, formalization, and validation of requirements on different levels of abstraction.

This paper makes the following two contributions: First (1), we extend SMLK to enable requirements engineers to intuitively, but formally model the requirements on the SoS-level as well as the interaction between the CSs (CS-level). With these extensions, requirements can be specified and validated independently, which addresses the managerial and operational independence of systems. Nevertheless, both levels of abstraction are connected to allow for the joint execution and testing of the specified behavior on the SoS- and CS-level, in order to detect and resolve contradictions in the requirements on both these levels.

Second (2), we propose a specification method where we combine behavior-driven development (BDD) and test-driven development (TDD) with the scenario-based modeling technique. This enables the iterative specification of system features and usage scenarios to document stakeholder expectations and generate tests steps, which subsequently drive the scenario-based modeling of the system specification.

While numerous approaches exist that suggest using formal scenario models to bridge the gap from informal requirements to the implementation of software-intensive systems [5, 29, 13, 28], the particular contribution of this paper is the extension of scenario-based modeling and programming techniques based on LSC Play-Out [13] and behavioral programming (BP) [15] with BDD and TDD. Enabling this combination of agile development techniques with scenario-based requirements modeling addresses the *coverage and sampling concerns* in scenario-based requirements engineering [28]: by

connecting features with tests (BDD), and tests with the scenario-based requirements model (TDD), we can ensure that every feature is modeled by an appropriate set of scenarios, and that these scenarios are validated by an appropriate set of tests.

We asses the applicability with a proof-of-concept e-mobility application and provide a demonstration tool[4,5] to enable others to use, evolve, and evaluate our approach.

Structure: We describe background in Sect. 2, the scenario-based requirements specification method in Sect. 3, and the proof-of-concept application in Sect. 4. We report related work in Sect. 5 and conclude in Sect. 6.

2   Background

2.1   System of Systems Engineering (SoSE)

For the description of System of Systems (SoS) no generally valid definition yet exists [1, 26]. Hence, a distinction between complex monolithic systems and SoS is often made by the system characteristics. Therefore Maier describes five key characteristics of SoS [23]: (1) *Operational Independence*: Each system that is part of the SoS is independent and can perform a meaningful task, even if it is not integrated into the SoS. (2) *Managerial Independence*: The individual systems are self-administered and individually managed. Consequently they collaborate with the other systems of the SoS, but they operate independently. (3) *Geographic Distribution*: The individual systems of the SoS are distributed over large spatial distances, which means that the exchange of information between the individual systems is of primary importance for collaboration. (4) *Evolutionary Development*: The objectives and functionality of an SoS can change constantly, as they can be added, modified or removed based on experience. Therefore an SoS never appears to be fully completed. (5) *Emergent Behavior*: By the collaboration of the individual systems, a synergism is achieved in which the SoS fulfils a purpose that cannot be achieved by or attributed to any of the individual systems.

These characteristics have a strong influence on the SoS development. To support a structured SoS development Dahmann et al. [4] describe the differences between systems engineering (SE) and SoS engineering (SoSE). Accordingly, SE and SoSE both start with identifying and understanding user capability objectives in order to derive technical requirements for the system to be developed. In SE we subsequently continue with a top-down requirements decomposition and system design, with clear responsibilities in the management and engineering of the system [11]. In SoSE, by contrast, the identified objectives and requirements serve as a basis for the development of new systems *or* the integration of existing systems to build the SoS. Particularly the operational and managerial independence of individual systems is challenging: the existing systems may also fulfill other purposes that may conflict with the SoS objectives and those of its CS. Therefore it is important to understand how the individual systems behave and how this behavior contributes to the overall SoS behavior.

When starting the SoS development it is important to categorize the SoS to be developed at an early stage because this has a significant influence on the RE approach

---

[4]https://bitbucket.org/crstnwchr/besos (includes the proof-of-concept example)
[5]https://bitbucket.org/jgreenyer/smlk/ (required to build the example project)
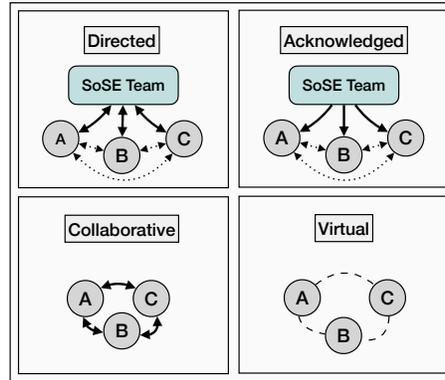
Fig. 1. Different types of SoS [23, 4]

that can be applied [26, 4, 25]. Fig. 1 shows four different SoS types, initially introduced by Maier [23] and extended by Dahmann and Baldwin [4]: A *directed* SoS is designed for specific purposes. The individual systems have the ability to operate independently but are managed by a SoSE Team in a way that they fulfill a specific purpose. In an *acknowledged* SoS the SoSE Team recognises and defines a common purpose and goal, but the CSs retain independent control and goals. The continuous and evolutionary development of the common purpose is based on collaboration between the SoS and the CSs. In a *collaborative* SoS the individual systems are not bound to follow a central management, but voluntarily participate in a collaboration in order to achieve the SoS goal. A *virtual* SoS has neither a leading control nor a common goal. This leads to a high degree of emergent behavior where the exact means and structures that produce the functionality of the system are difficult to recognize and distinguish [26, 27].

This paper focuses on acknowledged SoS and we introduce an example next.

2.2   Example of Application

To illustrate our approach, we introduce an e-mobility system of systems. In [21] Kirpes et al. introduce an architecture model that provides an integrative view on former separated areas of electricity, individual mobility, and information and communication technologies to realize future e-mobility SoS.

Based on the example defined in [21], Fig. 2 shows an SoS user who is interacting with an e-mobility SoS. The main interest of the user is to improve the e-mobility experience and to reduce its costs. These user interests are targeted by a high-level use case that describes how to create an optimized travel plan. At the beginning the user enters travel preferences like start and destination into the smartphone app (APP). The APP then requests further data from other systems that are necessary for the calculation of an optimized route. For example, GPS data of possible routes are requested from a route-planning service (RPS). Usage data of available charging points along the routes are provided by a charging station operation service (CSOS). Recommendations for a battery-saving charging process are provided by a battery-health service (BHS).
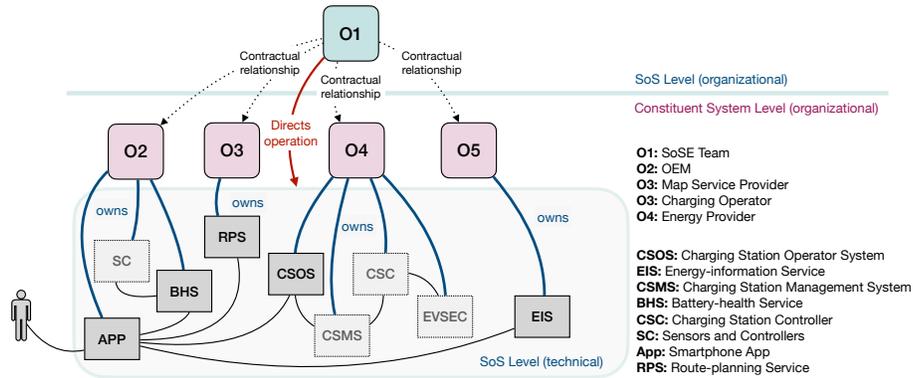
Fig. 2. Smart charging as an acknowledged SoS. Based on [25] and [21].

And information on current electricity prices in the region is provided by the energy-information service (EIS).

These different systems, which are required to provide information to calculate an optimal route based on user preferences, are developed and managed by four different system owners (OEM, Map Service Provider, Charging Operator, and Energy Provider). We also see one SoSE Team which defines the overall SoS functionality, directs the operations, and has a contractual relationship with the owners of the CSs. According to [23] and [4] this example has the characteristics of an acknowledged SoS: we have recognised requirements, objectives and responsibilities on the SoS level and a contractual relationship between the SoSE Team and the individual constituent systems owner. However, the constituent systems keep their own management, funding and development approaches (cf. [25]).

2.3 Scenario Modeling Language for Kotlin (SMLK)

SMLK is a Kotlin-based implementation of the Behavioral Programming (BP) paradigm [15]. In BP, a program consists of a number of *behavioral threads*, which we also call *scenarios*. Scenarios are loosely coupled via shared events and can model individual behavioral aspects or functional requirements of a system. Scenarios can *request* events that shall happen, be *triggered by* or *wait for* events requested by other scenarios, or (temporarily) *forbid/block* events. During execution, the scenarios are interwoven to yield a coherent system behavior that satisfies the requirements of all scenarios.

Listing 1.1 shows two SMLK scenarios that can be represented graphically as shown in Fig. 3. Both scenarios are triggered by the event of a user entering the travel preferences in the app. This event is modeled as an interaction event of the object `user` sending the object `app` a message `addTravelPreferences`. In the first scenario, the parameters `fromLoc` and `toLoc` are variables bound to the parameter values carried by the triggering event when the scenario is triggered and initialized. The SMLK code in the listing shows this binding of the parameter values explicitly (lines 2 and

3). The second scenario is triggered by the same event, but does not use the parameter values; the sequence diagram expresses this by using asterisks.

After the trigger event, the first scenario requests that the app sends the Route Planning Service (rps) a message to calculate the route between fromLoc and toLoc, and then requests that the rps shall respond with a route. Then the app shall optimize the route and show it to the user.

The second scenario describes the interaction of the app and the Charging Station Operating System (csos). After the triggering event, the scenario requests that the app sends the csos a request to send GPS position data of available charging stations. The scenario then requests that the csos shall respond with such a list. This interaction must happen before the app optimizes the route, i.e., the event app.optimizeRoute() is blocked until the second scenario terminates; only then can be first scenario proceed.

In these example scenarios, the route details and charging location list contents are not relevant, so mock instances are created by helper functions. When at a later point the behavior is refined, these parameter values may be replaced by other values, e.g., a detailed and correct route may be calculated elsewhere. The scenario method requestParamValuesMightVary allows us to request events with supplied default parameter values, but it will accept also events sent between the same objects, and with the same signature, but with different parameter values.
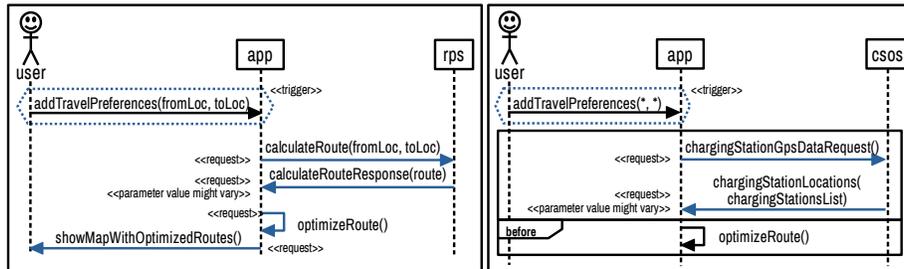

Fig. 3. Graphical representation of the SMLK scenario in Listing 1.1

```
1   scenario(user sends (app receives App::addTravelPreferences)){
2       val fromLoc = it.parameters[0] as String
3       val toLoc = it.parameters[1] as String
4       request(app sends rps.calculateRoute(fromLoc, toLoc))
5       val route = createMockRoute()
6       requestParamValuesMightVary(rps sends app.calculateRouteResponse(route)
7       request(app.optimizeRoute())
8       request(app sends user.showMapWithOptimizedRoute())
9   },
10  scenario(user sends (app receives App::addTravelPreferences)){
11      scenario {
12          request(app sends csos.chargingStationGpsDataRequest())
13          val chargingStationsList = createMockChargingStationsList()
14          requestParamValuesMightVary(csos sends app.considerChargingStationLocations(
                chargingStationsList))
15      }.before(app.optimizeRoute())
16  }
```

Listing 1.1. Example scenario from the e-mobility system specification

3   Scenario-based Requirements Specification in a System of Systems Context

To develop an SoS, usually existing systems are integrated by new systems to comprise a new SoS. While the new systems may be under a direct managerial and operational control, existing systems may be under the managerial and operational control of another organization. Over time, systems that are under external control may change, which leads to the necessity to continuously (1) analyze how the changes in one system impact the SoS functionality, and (2) how other systems may have to be adapted to ensure that the SoS functionality can still be provided. This requires the SoSE team to continuously analyze, specify, and align requirements across different hierarchy levels.

Our scenario-based requirements specification approach supports an iterative and integrated behavior modeling and analysis on the SoS and CS level.

Based on the definitions in [14] we introduce the term *inter-system scenarios* to model the behavior on the SoS level and *intra-system scenarios* to model the CS behavior. Also we show how both views can be integrated to allow for the joint execution and testing of the integrated SoS and CS behavior.

3.1   Inter-System Scenarios

The goal of modeling inter-system scenarios is to conceive an validate how SoS use cases can be realized by the interaction of users, existing systems, and new systems to be developed. The inter-system scenario modeling process starts by defining the use cases, the structural SoS architecture, and then detailing and validating the use cases using scenarios and repeated simulation.

When modeling this behavior, certain assumptions are made about the behavior of the existing systems, possibly based on available documentation or communication with experts from the respective organizations.

Two exemplary inter-system scenarios are already introduced in Listing 1.1, where we first modeled the interaction between the app, rps and the SoS user, and in the second scenario, between the user, app and csos. In this example, we see that we are able to model the interaction between selected systems, where new requirements can be considered by iteratively adding new scenarios to the *SoS scenario specification*. By adding these inter-system scenarios the introduced modeling concepts allow to focus on a high level system interaction; Although we are able to partly ignore specification details (e.g. exact route information in Listing 1.1 line 14), we are able to execute and validate the interaction between the CS. This supports on the SoSE team to get a better understanding of the overall system behavior.

3.2   Intra-System Scenarios

Once a satisfactory concept of the inter-system behavior is established, the inter-system specification must be supplemented and refined in two ways: First (1), it is necessary to specify the behavior of the existing systems in more detail in order to validate whether the inter-system interaction behavior is indeed aligned with the behavior of the existing

systems. Second (2), the behavior of the new systems to be developed must be detailed, possibly detailing their component structure and internal interactions, in order to provide a thorough basis for their development.

Our approach supports modeling the behavior on this more detailed hierarchy level with scenarios as well, and even to integrate their execution in order to simulate and validate behavioral requirements consistency across the different hierarchy levels.

To better distinguish between these two hierarchy levels, we distinguish the *inter-system* level and *intra-system* level as outlined in Fig. 4. The SoS scenario specification is located in the inter-system view, and individual CS scenario specifications are located in the intra-system view. When defining the internal behavior of a selected CS, we switch the perspective from the SoSE Team to a systems owner who is responsible for the development of a system. This can be e.g. the map service provider who is responsible for the development of the rps (see Fig. 2).
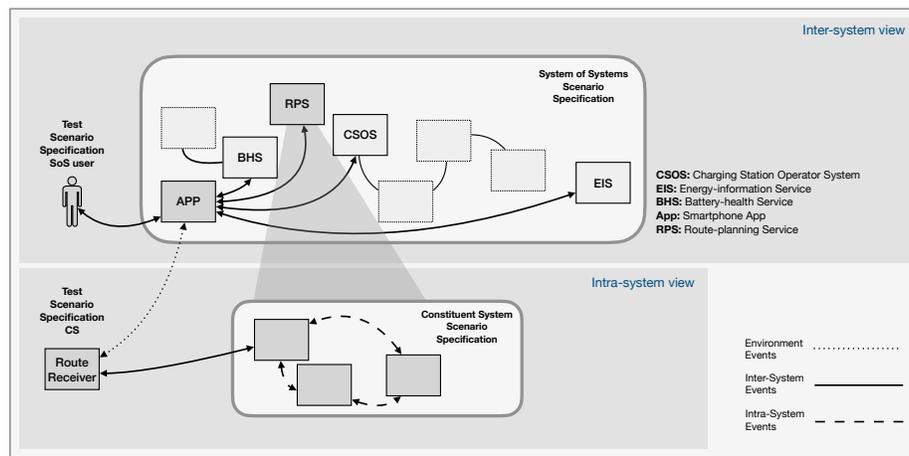

Fig. 4. Inter- and intra-system view to continuously concretise requirements on CS level, while also considering the overall SoS behavior.

The intra-system scenarios are added to an individual CS scenario specification, with the goal to model requirements which are needed to build the CS and its subsystems. One example intra-system scenario is shown in Listing 1.2.

```
1   scenario(routeRequester sends(rps receives Rps::calculateRoute)){
2       val fromLocString = it.parameters[0] as String
3       val toLocString = it.parameters[1] as String
4       request(rpsController sends gpsService.getLocations(fromLocString, toLocString))
5       val fromLoc = getLocation(fromLocString)
6       val toLoc = getLocation(toLocString)
7       request(gpsService sends rpsController.locations(fromLoc, toLoc))
8       request(rpsController sends routePlaner.calculateRoute(fromLoc, toLoc))
9       val route = calculateRoute(fromLoc, toLoc)
10      request(routePlaner sends rpsController.calculatedRoute(route))
11      request(rpsController sends routeRequester.calculateRouteResponse(route))
12  }
```

Listing 1.2. CS scenario specification of the RPS

The scenario specifies how the internal components of the rps (rpsController, gspService, and routePlanner) interact when receiving a request to calculate a route. Eventually (line 11), the calculated route will be returned to the requesting object.

When looking at the scenario in more detail, we see that the scenario is triggered when a `routeRequester` sends the `rps` the message `calculateRoute`. This event is requested on the inter-system level, see the first scenario in Listing 1.1 (line 4).

One difference is, however, that in the intra-system scenario, we abstract from the app as being the source of the `calculateRoute` request (and the recipient of the route as a reposonse, see line 11). Instead, we assume that there is an abstract external route-requesting entity that requests a route to be calculated by the rps. We do this to separate the intra-system specification of a system from the particular SoS context defined on the inter-system level, as the system may also be used in other contexts.

The inter-system and intra-system level scenario execution can nevertheless be integrated, because the type of `routeRequester` is an interface that is also implemented by `app` (without showing the code in more detail for brevity). Hence it is possible that the event of the app requesting to calculate a route triggers the scenario shown here, and indeed the app would then receive the calculated route as a response.

The event parameters on the intra-system level may vary or be more detailed than the values assumed on the inter-system level where, for example, we used simple mock values (see Listing 1.1, lines 5 and 13). It is possible for intra-system scenarios to provide more detailed parameter values where the inter-system level scenarios request events by using the `requestParamValuesMightVary` command. (see Listing 1.1 line 14).

3.3   Specification Method

To support the requirements engineer in modeling system requirements with SMLK, we propose an iterative method based on agile techniques. Fig. 5 shows an overview of the single steps. We start with the specification of the inter-system behavior by applying the BDD approach. Here we first define the expected system behavior from the SoS user perspective. Therefor we create a *SoS feature specification* where each feature is defined by one or more *usage scenarios* written in the gherkin syntax[6]. Listing 1.3 shows a first feature specification that describes a user interaction with the app. On this hierarchy level, the SoS feature specification allows the SoSE team to define what is expected from the SoS and to document this expectations in a comprehensible form.

```
1   Feature: Retrieve travel preferences and display optimized route
2
3       Scenario: Add travel preferences to the app
4           When the SoS user adds travel preferences to the app
5           Then the app displays a set of optimized routes
```

Listing 1.3. Initial feature specification including a usage scenario to describe the user interaction with the SoS.

Based on this SoS feature specification we generate test skeletons as shown in Listing 1.4. These test skeletons are then used to drive the modeling of the inter-system
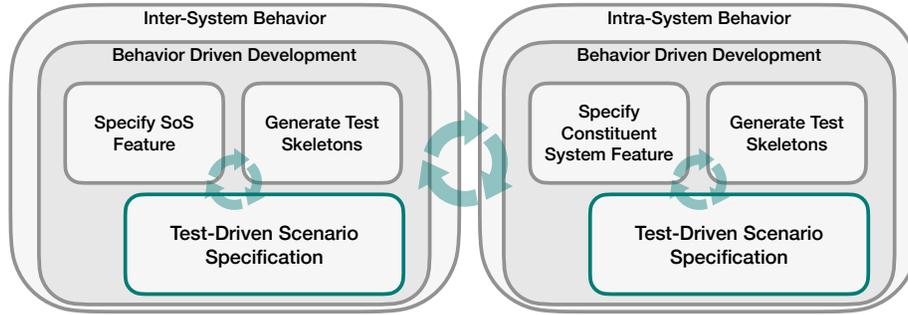
---

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

Fig. 5. Continuous and iterative scenario specification

behavior. To support a structured and iterative modeling of system requirements, we embed the Test-Driven Scenario Specification (TDSS) [32] into the BDD approach. In this way, we combine the comprehensible specification of expected system behavior with the formal and scenario-based modeling of system requirements.

```
1   When("^the EV user adds travel preferences to the App$") {
2       //implement here
3   }
4   Then("^the App displays a set of optimized routes$") {
5       //implement here
6   }
```

Listing 1.4. Generated test steps.

The TDSS approach includes the steps outlined in Fig. 6. In the first step we extend the generated test skeletons (1). Here, we e.g. model that the user adds travel preferences to the app (Listing 1.5 line 2) and eventually receives a map with optimized routes (line 5). After we added these functions we execute the SoS feature specification (2)
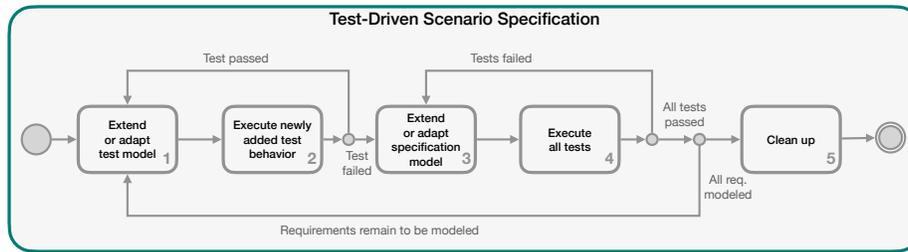

Fig. 6. Test-Driven Scenario Specification (TDSS) [32]

whereupon the single test steps and finally the events within the test steps are executed. At this point in time we did not model the inter-system behavior and consequently the test fails, because the app will not send the optimized route to the SoS user as expected in line 5.

```
1  When("^the EV user adds travel preferences to the App$") {
2      trigger(user sends app.addTravelPreferences("Dortmund", "Paderborn"))
3  }
4  Then("^the App displays a set of optimized routes$") {
5      eventually(app sends user.mapWithOptimizedRoutes())
6  }
```

Listing 1.5. Generated test steps.

Therefore we extend our SoS scenario specification with the inter-system scenarios (3) which we already introduced in Listing 1.1. We then run the test again to ensure that the modelled system requirements meet the expectations (4). If we have modeled additional tests in previous iterations, we now run them as well to ensure that there are no unexpected interactions between the individual tests and system requirements. If there are more requirements that need to be modeled, we perform further iterations. When all requirements on the SoS level known at this time have been modeled and tested, the SoS feature specification can be cleaned up. Afterwards the detailed specification of selected systems under development follows.

This iterative approach supports the modeling of the interaction of all CSs within the SoS. In this way we are able to iteratively document the expectations from an SoS user perspective and model and test the interaction between the CSs. Thereby new systems and behavior can be added as needed to realize the expected behavior. When we have gone through several iterations, the SoSE team gets a better understanding of which systems are needed and what information these systems have to exchange with each other. Subsequently we can switch to the intra-system level and focus on the requirements specification for a selected CS within the SoS. Based on our example outlined in Fig. 2 we now switch from the SoSE team perspective to e.g. the perspective of the map service provider, who is responsible for the development of the rps. As shown in Fig. 5 we execute the same specification method, but we create an independent *CS feature specification*, generate independent test steps and create an CS scenario specification. This allows the independent specification and modeling of the requirements for the CS, which addresses the managerial, operational and evolutionary independence of systems in an SoS. In this way, system requirements can be specified without seeing the system in an SoS context. But, at the same time, both views can be integrated (as described in Sec. 3.2), which allows the joint execution of the SoS behavior and the internal behavior of single already specified systems. In this way it's possible to detect contradictions between requirements on both levels. For example, if requirements have been specified at CS level that appear to have nothing to do with the SoS behavior but still influence the expected SoS behavior, the joint execution of the scenario specifications can be used to detect and resolve these dependencies.

4   Proof of Concept

To assess the applicability of our approach we integrated SMLK with the BDD tool Cucumber and executed the previously described specification method based on the example introduced in Section 2.2.

On SoS level we started with the feature specification as already shown in Listing 1.3. Subsequently we generated the test skeletons and added the SMLK events as

shown in Listing 1.5. Following the TDSS approach we executed the SoS feature spec-
ification (Step 1 in Fig. 6) and got a failed test result as shown in Fig. 7. Subsequently

Fig. 7. First TDSS run on SoS level

we extended the SoS scenario specification as shown in Listing 1.1 to specify the SoS
behavior. After we added these scenarios we executed the test again and finally received
the expected event, resulting in a positive test result as shown in Fig. 8.

Fig. 8. Execute tests after adapting the SoS scenario specification

After we successfully defined a first interaction on inter-system level, we switched
to the intra-system level and added a CS scenario specification to model the internal
behavior of the rps as shown in Listing 1.2.

Now we executed the same SoS feature again resulting in a negative result, because
the rps internal behavior was not yet specified and hence the CS scenario program didn't
send the `calculateRouteResponse(route)` message to the app.

To fix this we executed the TDSS process within the intra-system view, based on
the CS feature specification shown in Listing 1.6.

```
1    Feature: Calculate route - RPS
2
3        @RpsSystem
4        Scenario: Calculate route based on user travel preferences
5        When the app sends travel preferences to the rps
6        Then the rps responds route information including gps data
```

Listing 1.6. Feature on CS level

Finally we got passed test results again, but now we also considered the rps inter-
nal behavior specification. And, by using *tags* within the different feature specifications
(e.g. `@RpsSystem`) and by applying the concepts described in Sect. 3.2, we were not
only able to validate the integrated SoS and CS behavior, but we also could indepen-
dently test the requirements of single CS.

To allow others to use, validate and evolve our approach, we describe the architec-
ture and functional principles of the developed tool in [31] as a companion to this paper.
Here, we also describe the method we outline in Fig. 5 in more detail. And we provide

information about the necessary resources[7,8,9,10] to build and execute the example we use in this paper.

5   Related Work

In this paper we use SMLK, which was extended to support an iterative and continuous modeling of system behavior in an SoS context. This modeling language is based on Live Sequence Charts (LSCs) [6]. A recent LSC variant are Modal Sequence Diagrams (MSDs) [12]. By modeling behavioral requirements with the help of MSDs, different works argue that this formal requirements modeling can increase the requirements quality (e.g. [19], [10]), but these approaches are based on traditional SE and do not consider the SoS characteristics and their impact on the requirements specification.

Harel et al. describe an extension to behavioral programming that allows the integration of behavioral programs that operate on different hierarchy levels and time scales [16]. Indeed, we also use this approach to integrate different SMLK scenario programs that execute the behavior on the inter- and intra-system level.

Simulation-based analysis and design is commonplace in cyber-physical systems of systems, e.g. using actor-oriented frameworks or co-simulation [9, 22]. We aim to provide similar means for the thorough specification and analysis of *requirements* of systems of systems. To the best of our knowledge, this is a new approach.

Other works address model-based RE in the SoS context. Holt et al. describes an ontology for model-based SoS requirements engineering [18]. Albers et al. show how SoS requirements can be specified based on use-cases and sequence diagrams within SysML [2]. However, an early, iterative and formal specification of requirements, with the goal to execute and test these requirements specifications is not considered in these approaches.

6   Summary and Outlook

In this paper, we propose a technology to continuously model behavior requirements in an SoS context. Our approach supports requirements engineers in the iterative specification, modeling and testing of requirements. With the use of SMLK, the system behavior can be modeled textually through scenarios. This scenario-based modeling is close to how engineers communicate system behavior and hence enables a feasible formalization of requirements. To further support and structure the formalization process, we integrated SMLK with agile techniques and appropriate tooling. This fosters the iterative formalization, and by testing the formalized requirements specifications, we get early feedback about the expected system behavior and possible contradictions in requirements. Due to the proposed coupling of inter- and intra- system scenarios, we are also able to execute and test the system behavior on different hierarchy levels.

---

[7]https://bitbucket.org/crstnwchr/besos/

[8]https://bitbucket.org/jgreenyer/smlk/

[9]https://cucumber.io

[10]https://www.jetbrains.com/idea/

And by integrating the BDD tool cucumber, we are able to specify the expected system behavior with the help of features and usage scenarios written in natural language, which supports the communication of expected system behavior in a multi-disciplinary development team.

For future work, we plan to integrate our previous work [32] and the modeling concepts shown in this paper with an automated test case creation proposed in [8] to further reduce the modeling effort. Also, as already started in previous work [33], we plan to integrate the results of this paper in an automotive development process and validate the applicability within an ongoing research project. As shown in [32], we are able to find contradictions in automotive requirements specifications, but the open questions are if the approach is scalable and whether the effort for the requirements modeling is justified.

Another possible direction for future work is focusing on stakeholder needs in a SoS context. In this paper we already integrated the BDD approach to validate requirements and align stakeholder expectations. This could be done more systematically by integrating goal modeling approaches [3].

References

1. Albers, A., Mandel, C., Yan, S., Behrendt, M.: System of systems approach for the description and characterization of validation environments. In: Proceedings of International Design Conference, DESIGN. vol. 6, pp. 2799–2810 (2018). https://doi.org/10.21278/idc.2018.0460
2. Albers, A., Kurrle, A., Moeser, G.: Modellbasiertes Anforderungsmanagement von Systems-of-Systems am Beispiel des vernetzten Fahrzeugs. In: Tag des Systems Engineering (TdSE) : Bremen, 12. - 14. November 2014. Hrsg.: M. Maurer. pp. 373–382. Hanser, München (2015)
3. Aydemir, F.B., Dalpiaz, F., Brinkkemper, S., Giorgini, P., Mylopoulos, J.: The next release problem revisited: A new avenue for goal models (2018). https://doi.org/10.1109/RE.2018.00-56
4. Dahmann, J.S., Baldwin, K.J.: Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering. In: 2008 2nd Annual IEEE Systems Conference. pp. 1–7 (2008)
5. Damas, C., Lambeau, B., van Lamsweerde, A.: Scenarios, goals, and state machines: A win-win partnership for model synthesis. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 197–207. SIGSOFT '06/FSE-14, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1181775.1181800, https://doi.org/10.1145/1181775.1181800
6. Damm, W., Harel, D.: {LSCs}: Breathing Life into Message Sequence Charts. In: Formal Methods in System Design. vol. 19, pp. 45–80 (2001)
7. Fernández, D.M., Wagner, S.: Naming the pain in requirements engineering. Empirical Software Engineering p. 183 (2013). https://doi.org/10.1145/2460999.2461027
8. Fischbach, J., Vogelsang, A., Spies, D., Wehrle, A., Junker, M., Freudenstein, D.: SPECMATE: Automated Creation of Test Cases from Acceptance Criteria. In: Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation, ICST 2020. pp. 321–331 (2020). https://doi.org/10.1109/ICST46399.2020.00040
9. Fitzgerald, J., Pierce, K., Larsen, P.G.: Co-modelling and co-simulation in the engineering of systems of cyber-physical systems. In: 2014 9th International Conference on System of Systems Engineering (SOSE). pp. 67–72 (2014). https://doi.org/10.1109/SYSOSE.2014.6892465

10. Fockel, M., Holtmann, J., Koch, T., Schmelter, D.: Formal , Model- and Scenario-based Requirement Patterns. In: 6th International Conference on Model-Driven Engineering and Software Development (2016). https://doi.org/10.5220/0006554103110318

11. Gausemeier, J., Moehringer, S.: VDI 2206- A New Guideline for the Design of Mechatronic Systems. In: IFAC Proceedings Volumes. pp. 785–790. Elsevier (2002). https://doi.org/10.1016/s1474-6670(17)34035-1, http://dx.doi.org/10.1016/S1474-6670(17)34035-1

12. Harel, D., Maoz, S.: Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. In: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. pp. 13–20. SCESM '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1138953.1138958, http://doi.acm.org/10.1145/1138953.1138958

13. Harel, D., Marelly, R.: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. SoSyM **2**, 82–107 (2003)

14. Harel, D., Marelly, R., Marron, A., Szekely, S.: Integrating Inter-Object Scenarios with Intra-object Statecharts for Developing Reactive Systems. IEEE Design and Test pp. 1–19 (2020). https://doi.org/10.1109/MDAT.2020.3006805

15. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Comm. ACM **55**(7), 90–100 (2012). https://doi.org/10.1145/2209249.2209270

16. Harel, D., Marron, A., Wiener, G., Weiss, G.: Behavioral Programming, Decentralized Control, and Multiple Time Scales. In: Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11. pp. 171–182. SPLASH '11 Workshops, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2095050.2095079, https://doi.org/10.1145/2095050.2095079

17. Hoehne, O.M., Rushton, G.: A System of Systems Approach to Automotive Challenges. In: SAE Technical Paper. SAE International (2018). https://doi.org/10.4271/2018-01-0752, https://doi.org/10.4271/2018-01-0752

18. Holt, J., Perry, S., Brownsword, M., Cancila, D., Hallerstede, S., Hansen, F.O.: Model-based requirements engineering for system of systems. In: Proceedings - 2012 7th International Conference on System of Systems Engineering, SoSE 2012. pp. 561–566 (2012). https://doi.org/10.1109/SYSoSE.2012.6384145

19. Holtmann, J., Bernijazov, R., Meyer, M., Schmelter, D., Tschirner, C.: Integrated and iterative systems engineering and software requirements engineering for technical systems. Journal of Software: Evolution and Process **28**(9), 722–743 (2016). https://doi.org/10.1002/smr.1780, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1780

20. INCOSE: INCOSE systems engineering handbook: a guide for system life cycle processes and activities. John Wiley & Sons (2015)

21. Kirpes, B., Danner, P., Basmadjian, R., de Meer, H., Becker, C.: E-Mobility Systems Architecture: a model-based framework for managing complexity and interoperability. Energy Informatics **2**(1) (2019). https://doi.org/10.1186/s42162-019-0072-4

22. Lee, K., Hong, J.H., Kim, T.: System of Systems Approach to Formal Modeling of CPS for Simulation-Based Analysis. ETRI Journal **37**, 175–185 (2015). https://doi.org/10.4218/etrij.15.0114.0863

23. Maier, M.W.: Architecting Principles for Systems-of-Systems. INCOSE International Symposium **6**(1), 565–573 (1996). https://doi.org/10.1002/j.2334-5837.1996.tb02054.x

24. Ncube, C.: On the engineering of systems of systems: Key challenges for the requirements engineering community. In: 2011 Workshop on Requirements Engineering for Systems, Services and Systems-of-Systems, RESS 2011 - Workshop Co-located with the 19th IEEE International Requirements Engineering Conference. pp. 70–73. IEEE (2011). https://doi.org/10.1109/RESS.2011.6043923

25. Ncube, C., Lim, S.L.: On systems of systems engineering: A requirements engineering per-spective and research agenda. Proceedings - 2018 IEEE 26th International Requirements Engineering Conference, RE 2018 pp. 112–123 (2018). https://doi.org/10.1109/RE.2018.00021
26. Nielsen, C., Larsen, P., Fitzgerald, J., Woodcock, J., Peleska, J.: Systems of Systems Engineering. ACM Computing Surveys **48**, 1–41 (2015). https://doi.org/10.1145/2794381
27. Odusd, A., Sse, T.: Systems Engineering Guide for Systems of Systems. Tech. Rep. August, Office of the Under Secretary of Defense (2008). https://doi.org/10.1109/EMR.2008.4778760
28. Sutcliffe, A.: Scenario-based requirements engineering. In: Proceedings of the IEEE International Conference on Requirements Engineering. pp. 320–329 (2003). https://doi.org/10.1109/ICRE.2003.1232776
29. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd International Conference on Software Engineering. p. 314–323. ICSE '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/337180.337217, https://doi.org/10.1145/337180.337217
30. Wiecher, C.: A Feature-oriented Approach: From Usage Scenarios to Automated System of Systems Validation in the Automotive Domain. In: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion). Virtual Event, Canada (2020). https://doi.org/https://doi.org/10.1145/3417990.3419485
31. Wiecher, C., Greenyer, J.: Besos: A tool for behavior-driven and scenario-based requirements modeling for systems of systems, preprint (2021)
32. Wiecher, C., Greenyer, J., Korte, J.: Test-Driven Scenario Specification of Automotive Software Components. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 12–17. Munich, Germany (2019). https://doi.org/10.1109/MODELS-C.2019.00009
33. Wiecher, C., Japs, S., Kaiser, L., Greenyer, J., Dumitrescu, R., Wolff, C.: 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) (2020). https://doi.org/https://doi.org/10.1145/3417990.3421264