BeSoS: A Tool for Behavior-driven and Scenario-based Requirements Modeling for Systems of Systems

Carsten Wiecher[a], Joel Greenyer[b]

[a]*Dortmund University of Applied Sciences and Arts, 44139 Dortmund, Germany*
[b]*FHDW Hannover, 30173 Hannover, Germany*

Abstract

Systems of Systems (SoS), like connected vehicle systems, provide their functionality by the interaction of several constituent systems (CSs). [Problem] Due to the managerial, operational and evolutionary independence of the CSs in an SoS, requirements constantly change over time and linear, top-down requirements engineering methods cannot be applied without significant adaptations. New tools are needed that support the continuous and iterative specification and alignment of requirements across different levels of abstraction. [Principal Ideas] We propose to integrate the behavior-driven development (BDD) approach with an intuitive and executable scenario-based modeling of functional requirements. In this way, stakeholder expectations can be structured via features and documented in natural language as usage scenarios. Based on usage scenarios, the modeling of functional requirements can be driven by tests, allowing for the automated testing and analysis of requirements. This in turn supports the iterative specification of requirements and the alignment of stakeholder needs. [Contribution] In this paper we showcase the tool *BeSoS* that supports the iterative and behavior-driven specification of requirements in an SoS context. We propose a method and describe its tool components using an example. The tool is available here: https://vimeo.com/512739942

Keywords

System of Systems Engineering, Requirements Analysis, Scenario-based Requirements Modeling, Requirements Specification

1. Introduction

With this paper we present the tool *BeSoS*[1], a tool for the behavior-driven and scenario-based requirements modeling and validation. In *BeSoS* we combine the Scenario Modeling Language for Kotlin (SMLK)[2] with the behavior-driven development tool Cucumber[3] to support the requirements engineer in the iterative specification and analysis of functional requirements in a system of systems (SoS) context [1].

In the automotive domain, requirements are usually documented in natural language [2]. Although model-driven methods are widespread in the automotive domain, only a few frameworks explicitly integrate model-driven techniques that support the validation and analysis of requirements [3]. With *BeSoS* we propose an application-oriented approach with the aim of facilitating the use of formal, executable requirements specification and analysis in practice. In collaboration with a Tier1 supplier company, we already showed that the formal and scenario-based modeling can reveal contradictions in automotive requirements specification [4] and can help to bridge the gap between the business and technical domains via continuous validation and short feedback loops [5]. Based on these

In: F.B. Aydemir, C. Gralha, S. Abualhaija, T. Breaux, M. Daneva, N. Ernst, A. Ferrari, X. Franch, S. Ghanavati, E. Groen, R. Guizzardi, J. Guo, A. Herrmann, J. Horkoff, P. Mennig, E. Paja, A. Perini, N. Seyff, A. Susi, A. Vogelsang (eds.): Joint Proceedings of REFSQ-2021 Workshops, OpenRE, Posters and Tools Track, and Doctoral Symposium, Essen, Germany, 12-04-2021
carsten.wiecher@fh-dortmund.de (C. Wiecher); joel.greenyer@fhdw.de (J. Greenyer)
0000-0002-3280-4471 (C. Wiecher); 0000-0003-0347-0158 (J. Greenyer)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

[1] https://bitbucket.org/crstnwchr/besos/
[2] https://bitbucket.org/jgreenyer/smlk/
[3] https://cucumber.io

CEUR-WS.org/Vol-2857/pt1.pdf

findings, we specifically focused on modeling requirements in a system of systems (SoS) context [6] to address the challenges in requirements engineering (RE)[7] for the next generation of automotive systems [8].

Ncube and Lim argue that it is important to identify the SoS type in an early phase of system development, because the SoS type has a significant influence on the applicability of RE methods [7]. We chose to focus on *acknowledged SoS*, which are arguably the most common kind of SoS in transportation and mobility use cases. An acknowledged SoS is an SoS where a central authority can be identified that directs the SoS operation (e.g. a local government). Furthermore, for an acknowledged SoS the requirements, objectives and responsibilities can be recognised on the SoS level, and there can be contractual relationships between the central authority and the individual constituent systems' (CSs) owners. However, the CSs keep their own management, funding, and development processes (cf. [9, 10, 11]).

With this context set, we focus on the modeling and analysis of requirements in an automotive context by mainly addressing the research theme of *multi-level modeling techniques* for SoS requirements, as identified by Ncube and Lim [7]. We propose an iterative and integrated modeling method for the SoS behavior and the CS behavior (see details on the methodology in [6]). The proposed tool is part of an ongoing research in close collaboration between a Tier1 supplier and different research institutions.

2. The BeSoS Tool

The intended user of the tool are requirements engineers that are responsible for the specification of requirements for technical systems in an SoS context. We first show an idealized process including the single steps and involved artifacts. Second, we describe the components of the tool.

2.1. Method

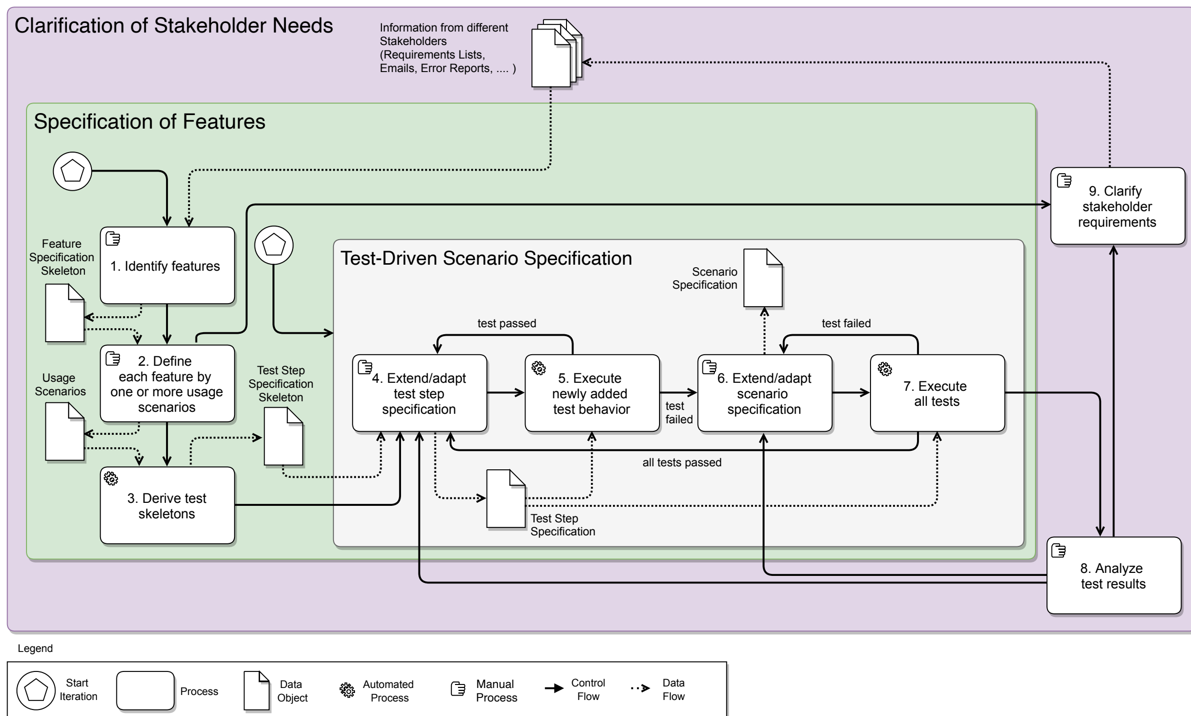

Figure 1: Method for the iterative formalization of requirements.

The aim of the method is to create a formal specification of system requirements on the SoS and CS level that can be validated to iteratively align system requirements with the stakeholder needs.

Figure 1 shows the process supported by BeSoS; it covers different areas within the RE process: *elicitation*, *modeling*, and *requirement analysis*, which we grouped into *clarification of stakeholder needs*, *specification of features*, and *test-driven scenario specification* (TDSS). Each of these areas includes artifacts with different degrees of formality. The starting point is a collection of informal information artifacts. This can be requirements lists, emails, validation results, or other sources of information that reflect stakeholder needs in different formats and levels of abstraction.

To drive the scenario-based requirements modeling, we first identify *features* and create separate *feature files* (1). Subsequently, we derive *usage scenarios* for each feature (2), which describe the stakeholder needs in a structured and comprehensive form, using behavior-driven development (BDD) techniques. In (3) we use this structured specification of features and usage scenarios to automatically derive *test skeletons*. Test skeletons provide a Given-When-Then structure for a testing framework, and serve as the starting point for the TDSS sub-process, cf. [4]. In TDSS, the requirements are modeled, driven by the tests, using an executable scenario modeling language. Driven by tests, the requirements models are extended and completed in an iterative way, continuously validating them, and checking and resolving inconsistencies, until all requirements are modeled, i.e., all generated tests are passed.

Since we generate the tests from usage scenarios that relate to features that in turn are derived from stakeholder needs, we create a closed loop with analyzing the test results (8) and clarifying stakeholder requirements (9).

All steps are consistently supported by *BeSoS* by integrating the technologies provided by Cucumber, JUnit, and SMLK: The specification of features (1) and usage scenarios (2) are supported by the Cucumber tool that also allows the generation of test steps (3). The test-driven requirements modeling (4,6) is is done by using SMLK. Within the TDSS sub-process, the actual test execution (5,7) is directed by JUnit, which in turn integrates with Cucumber and links test results with features, this way supporting steps 8 and 9.

2.2. Architecture

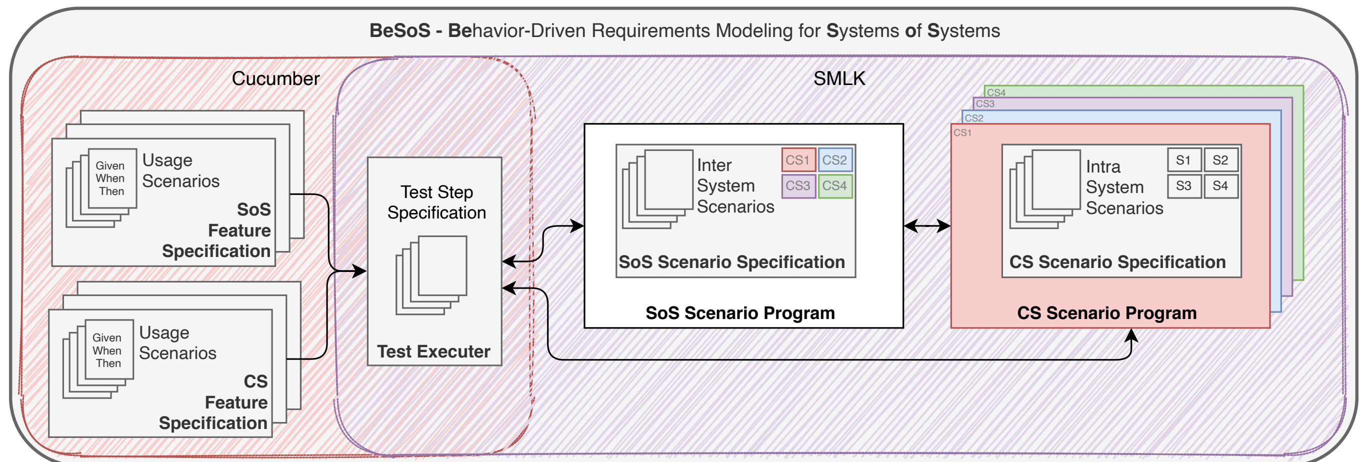

Figure 2: Architecture of the BeSoS tool that combines SMLK and Cucumber.

BeSoS consists of the components shown in Fig. 2. Features can be specified on the SoS- and CS-level and are stored in separate files, including one or several usage scenarios written in the Gherkin syntax[4] as shown in Listing 1.

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

```
1  Feature: send advertisement with available charging stations near vicinity
2    Scenario: electric vehicle changes position
3      When the electric vehicle changes its position
4      Then the advertisement SoS offers available charging stations in the closer vicinity
```

Listing 1: Feature specification using the Gherkin syntax.

When conceiving a new SoS, we propose to start with the definition of *features* that describe how users and external systems interact with the SoS. Formally modeling these interactions is done, on the one hand, with *usage scenarios* and detailed *test step specifications* that describe the expected reactions of the SoS to external events. On the other hand, we suggest to use *inter-system scenarios* [6], motivated by [12], for modeling the SoS end-to-end system interactions. The goal is to conceptualize which CSs are necessary to provide the SoS functionality, and how these systems must interact. The inter-system scenarios form the *SoS scenario specification*. Together with the tests, this specification is executable as an *SoS scenario program*.

Based on the SoS scenario specification, we can detail the expected behavior of the to-be-developed CSs in separate *CS scenario specifications* [6]. The goal here is to provide a thorough basis for implementing new CSs. With specific CS-level *feature specification* and detailed *test step specifications*, these CS specifications can be executed as independent *CS scenario programs*. But the CS specifications can also be executed in conjunction with the SoS scenario program. This way, both views can be aligned in order to identify possible contradictions between the expected SoS behavior and specified behavior of individual CSs.

2.3. Example

To illustrate the usage of the tool, we consider an example SoS with systems that interact in order to provide a driver of an electric vehicle with current price information of nearby charging stations. We assume that an electric vehicle regularly sends its current location to an advertising service, which then collects pricing information about nearby charging stations. The advertising service collects this information and sends it to the user via a their smartphone app.

We consider three viewpoints in this example (VP1-VP3, see Fig. 3). VP1: SoS-level features and detailing tests; VP2: inter-system scenarios; VP3: CS specification scenarios. For VP1 we consider the SoS as a black-box and create the SoS feature specification as exemplary shown in Listing 1. This feature documents that an offer with available charging stations should be made to the SoS user when the vehicle sends an updated location.

Following the *BeSoS* method, we generate test steps based on the defined usage scenario (Listing 1) as shown in Listing 2. (We keep it simple for brevity; a test could also specify that specific prices of specific nearby stations are correctly displayed.) Subsequently we enter the TDSS sub-process; with starting the TDSS iterations we change to the viewpoint VP2 and specify of inter-system scenarios.

```
1  When("the \"Electric Vehicle\changes its position") {
2      trigger(gpsSensor sends electricVehicle.positionChanged(VehiclePosition(23232.323, 323.2323))) // manually added SMLK code
3  Then("the \"Advertisement SoS\offers available charging stationin the closer vicinity") {
4      eventually(smartphoneApp sendsosUser.showAdvertisement())} // manually added SMLK code
```

Listing 2: Generated test steps.

To pass the test in Listing 2, we model SoS requirements as a scenario, see Listing 3 (also see bottom of Fig. 3). This scenario is triggered when the GPS sensor sends the position changed message to the electric vehicle. In the body of the scenario we model the interactions between the CSs: the vehicle sends its current position to the advertisement service (line 3) that in turn inquiries information about charging stations in the vehicle's vicinity (line 4). This information is then provided (line 7) and the advertisement service sends an offer to the smartphone app (line 10),

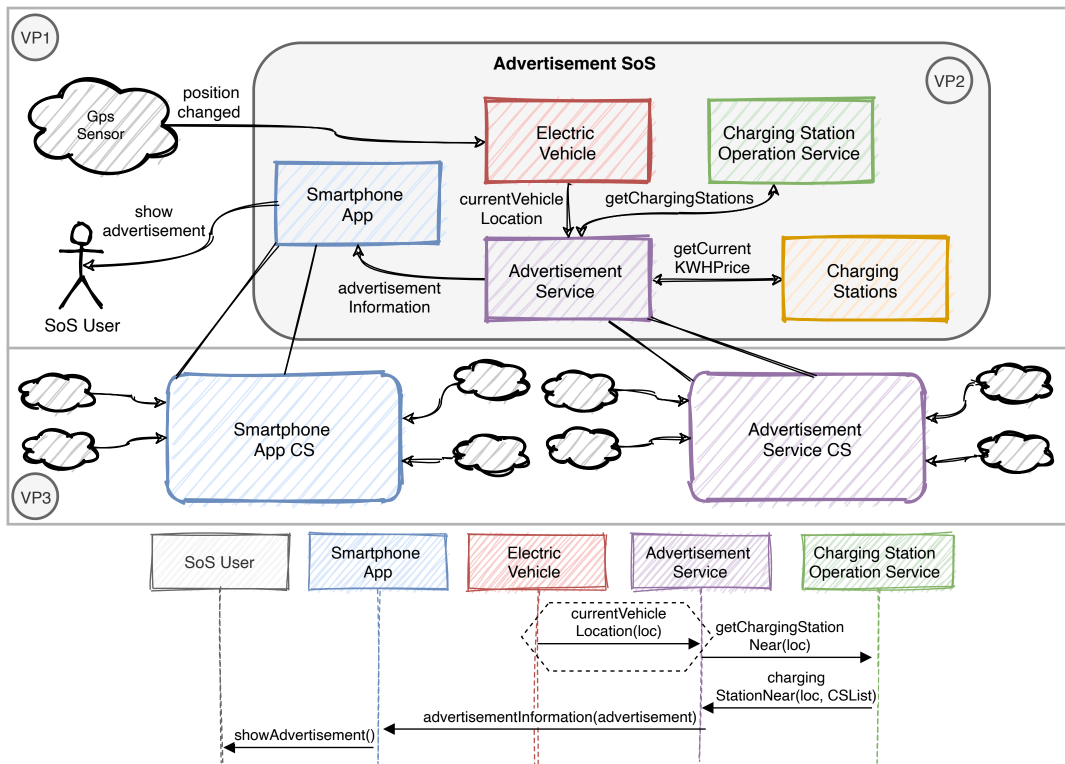

Figure 3: SoS example: e-mobility advertisement SoS

which displays it to the SoS user (line 11). Let us suppose that on this level, we have not yet detailed the way in which charging stations become known to the operations service. To nevertheless have a meaningful and executable specification, we can provide prototypical parameter values within the scenario (lines 5 and 6). Likewise, let us suppose it is to be clarified how the advertisement service actually compiles the pricing information. Should the service query each charging station every time? Should the charging stations receive regular pricing updates? That may be an architectural decision to be taken later. At this stage, we provide a prototypical advertisement object here as well (line 9).

```
1  scenario(gpsSensor sends electricVehicle receives ElectricVehicle::positionChanged){
2      val loc = it.parameters[0] as Location
3      request(electricVehicle sends advertisementService.currentVehicleLocation(loc))
4      request(advertisementService sends chargingStationOperationService.getChargingStationsNear(loc))
5      val dummyCS1 = ChargingStation("EcoBigCharge")
6      val dummyCS2 = ChargingStation("StarCharge")
7      requestParamValuesMightVary(chargingStationOperationService sends advertisementService.chargingStationsNear(loc, listOf(dummyCS1, dummyCS2)))
8      // To be refined: how does the advertisementService actually collect the price updates to forward to the user?
9      val dummyAdvertisement = Advertisement(mapOf(dummyCS1 to 295, dummyCS2 to 289)) // use dummy values for now
10     requestParamValuesMightVary(advertisementService sends smartphoneApp.advertismentInformation(dummyAdvertisement))
11     request(smartphoneApp sends sosUser.showAdvertisement())
12 }
```

Listing 3: SoS Scenario Specification

In this fashion, we can incrementally add further scenarios and CSs until the previously defined test cases are satisfied.

The next step is to specify the behavior of the individual CSs, and we switch to the viewpoint VP3. Here we abstract from a concrete SoS context and support the application of the proposed method independently for each

CS, by using abstract interfaces. As an example, Listing 4 shows a CS scenario for the advertisement service. This scenario is triggered when the vehicle sends its current location to the advertisement service (line 1). Then the scenario specifies to retrieve a list of charging stations near that location from the charging station operation service (line 2). This is similar to the SoS-level scenario—some redundancies can be expected. Next, however, the scenario specifies how the pricing information shall be obtained (lines 8-12). Indeed the taken approach is to ask each charging station nearby. The information is collected in a map that is defined within the scenario (line 6), and this map is now used to create an advertisement object (line 14), which is then sent to the smartphone app (line 15).

In this CS-level scenario we use the keyword `request` where the specified CS sends messages and `waitFor` where the CS receives messages. Moreover, as we no longer resort to prototypical parameter values, we no longer use `requestParamValuesMightVary`. In lines 7, 13 we see a construct **`scenario`** `... before <event>`, which specifies that the nested scenario must occur before the occurrence of `<event>`. In this case, collecting the price information must happen before sending the advertisement information to the smartphone app. Also thinking about what events are forbidden in certain event sequences is important, especially when jointly executing scenarios on the SoS and CS-level. Here this forces the two scenarios to synchronize on the event, while the prototypical value suggested in the SoS-level scenario will be overwritten with the value provided in the CS-level scenario.

```
1  scenario(electricVehicle sends AdvertisementService::currentVehicleLocation.symbolicEvent()){
2  val loc = it.parameters[0] as Location
3  request(advertisementService sends chargingStationOperationService.getChargingStationsNear(loc))
4  val availableChargingStationsEvent = waitFor(chargingStationOperationService sends AdvertisementService::chargingStationsNear.symbolicEvent())
5  val availableChargingStations = availableChargingStationsEvent.parameters[1] as List<ChargingStation>
6  val chargingStationsCurrentPrice = mutableMapOf<ChargingStation, Int>()
7  scenario {
8      for(chargingStation in availableChargingStations){
9          request(advertisementService sends chargingStation.getCurrentKWHPrice())
10         val replyEvent = waitFor(chargingStation sends AdvertisementService::updateCurrentKWHPrice.symbolicEvent())
11         chargingStationsCurrentPrice.put(chargingStation, replyEvent.parameters[1] as Int)
12     }
13 } before (advertisementService sends smartphoneApp receives SmartphoneApp::advertismentInformation)
14 val advertisement = Advertisement(chargingStationsCurrentPrice)
15 request(advertisementService sends smartphoneApp.advertismentInformation(advertisement))
16 }
```

Listing 4: CS Scenario Specification

3. Closing

In this paper we present the tool *BeSoS* that integrates the BDD paradigm with an intuitive and scenario-based requirements modeling. The contribution is twofold: 1) With the comprehensive specification of features that drive the intuitive and iterative modeling of functional requirements, we enable practitioners to use executable, formal requirements specification and analysis techniques. Especially in the context of SoS with its independent and evolving CSs, we believe that this tool and the proposed iterative method can be helpful. 2) Using formal scenario models to bridge the gap from informal requirements to the design and implementation of systems is not new. Different approaches argue that this is beneficial [13, 14, 15, 16]. The contribution of this work is that *BeSoS* supports scenario-based modeling and programming techniques in a SoS context, based on LSC Play-Out [15] and behavioral programming [17]. Thereby, the integration of the BDD and TDSS approach [4] addresses the *coverage and sampling concerns* in scenario-based requirements engineering [16]: following the method shown in Fig. 1, we can ensure that every feature is modeled by an appropriate set of scenarios (BDD), and that these scenarios are validated by an appropriate set of tests (TDSS) (see also [6]).

References

[1] J. Lane, D. Epstein, What is a System of Systems and Why Should I Care, 2013.

[2] G. Liebel, M. Tichy, E. Knauss, O. Ljungkrantz, G. Stieglbauer, Organisation and communication problems in automotive requirements engineering, Requirements Engineering 23 (2018) 145–167. doi:10.1007/s00766-016-0261-7.
[3] G. Liebel, M. Tichy, E. Knauss, Use, potential, and showstoppers of models in automotive requirements engineering, Software and Systems Modeling 18 (2019) 2587–2607. URL: https://doi.org/10.1007/s10270-018-0683-4. doi:10.1007/s10270-018-0683-4.
[4] C. Wiecher, J. Greenyer, J. Korte, 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), Munich, Germany, 2019, pp. 12–17. doi:10.1109/MODELS-C.2019.00009.
[5] C. Wiecher, S. Japs, L. Kaiser, J. Greenyer, R. Dumitrescu, C. Wolff, 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. doi:https://doi.org/10.1145/3417990.3421264.
[6] C. Wiecher, J. Greenyer, C. Wolff, H. Anacker, R. Dumitrescu, Iterative and Scenario-based Requirements Specification in a System of Systems Context, in: F. Dalpiaz and P. Spoletini (Eds.): REFSQ 2021, LNCS 12685, 2021, pp. 1–17. doi:https://doi.org/10.1007/978-3-030-73128-1_12.
[7] C. Ncube, S. L. Lim, On systems of systems engineering: A requirements engineering perspective and research agenda, Proceedings - 2018 IEEE 26th International Requirements Engineering Conference, RE 2018 (2018) 112–123. doi:10.1109/RE.2018.00021.
[8] O. M. Hoehne, G. Rushton, A System of Systems Approach to Automotive Challenges, in: SAE Technical Paper, SAE International, 2018. URL: https://doi.org/10.4271/2018-01-0752. doi:10.4271/2018-01-0752.
[9] M. W. Maier, Architecting Principles for Systems-of-Systems, INCOSE International Symposium 6 (1996) 565–573. doi:10.1002/j.2334-5837.1996.tb02054.x.
[10] J. S. Dahmann, K. J. Baldwin, Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering, in: 2008 2nd Annual IEEE Systems Conference, 2008, pp. 1–7.
[11] C. Nielsen, P. Larsen, J. Fitzgerald, J. Woodcock, J. Peleska, Systems of Systems Engineering, ACM Computing Surveys 48 (2015) 1–41. doi:10.1145/2794381.
[12] D. Harel, R. Marelly, A. Marron, S. Szekely, Integrating Inter-Object Scenarios with Intra-object Statecharts for Developing Reactive Systems, IEEE Design and Test (2020) 1–19. doi:10.1109/MDAT.2020.3006805. arXiv:1911.10691.
[13] C. Damas, B. Lambeau, A. van Lamsweerde, 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, SIGSOFT '06/FSE-14, Association for Computing Machinery, New York, NY, USA, 2006, p. 197–207. URL: https://doi.org/10.1145/1181775.1181800. doi:10.1145/1181775.1181800.
[14] J. Whittle, J. Schumann, Generating statechart designs from scenarios, in: Proceedings of the 22nd International Conference on Software Engineering, ICSE '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 314–323. URL: https://doi.org/10.1145/337180.337217. doi:10.1145/337180.337217.
[15] D. Harel, R. Marelly, Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach, SoSyM 2 (2003) 82–107.
[16] A. Sutcliffe, Scenario-based requirements engineering, in: Proceedings of the IEEE International Conference on Requirements Engineering, 2003, pp. 320–329. doi:10.1109/ICRE.2003.1232776.
[17] D. Harel, A. Marron, G. Weiss, Behavioral programming, Comm. ACM 55 (2012) 90–100. doi:10.1145/2209249.2209270.