Scenario-Based Modeling and Programming of Distributed Systems (Extended Abstract)

Joel Greenyer[1][0000−0003−0347−0158]

FHDW Hannover, 30173 Hannover, Germany
joel.greenyer@fhdw.de

Abstract. Software systems become increasingly distributed and interconnected. Single functions are usually realized by the interaction of multiple components, while single components participate in fulfilling multiple functions. During early development, engineers often design new functionalities by first taking an inter-component, interaction-oriented perspective, which must be mapped to a single-component implementation perspective later. This extended abstract reviews past and ongoing work on scenario-based modeling and programming techniques, which support engineers in this process. We also discuss new opportunities for applying these techniques as well as open research challenges.

Keywords: Scenario-Based Design · Reactive Systems · Behavioral Programming

1 Introduction

From micro-services information systems to cyber-physical systems-of-systems, software-intensive systems become increasingly distributed and interconnected. In these systems, single functions are usually realized by the interaction of multiple components, while single components often participate in fulfilling multiple functions. During the design of these systems, engineers often first take a birds-eye, inter-component view on how components shall interact in order to fulfill a certain function. Typically sequence diagrams or interaction overview diagrams are modeled during this phase to support design decision-making and communication. At some stage, these inter-component models must be mapped to specifications and, finally, implementations of the individual components.

Scenario-based modeling techniques aim at supporting engineers in this process. One goal of these techniques is to facilitate the transition from inter-component- to component-specific behavior specifications, and doing so iteratively. The second goal is to support the engineer with behavior analysis techniques already during the inter-component behavior modeling stage, so that engineers can make more informed early design decisions.

In this extended abstract, we give a brief review of the existing work in this context, with a pinch of personal experience and opinion. In particular, we cover own past and current work on the *Scenario Modeling Language* (SML) and the *Scenario Modeling Language for Kotlin* (SMLK), a Kotlin-based internal DSL for scenario-based programming. Concluding, we discuss open research directions.

2 Related Work

Early research on scenario-based modeling was centered around *Message Sequence Charts (MSC)* [26], which also influenced UML Sequence Diagrams. A number of approaches suggested techniques for deriving state-based models from individual MSCs [31, 39] and *high-level MSCs* [30], which are MSCs extended with control-flow constructs. Other approaches considered deriving state-based models in a setting where multiple MSC diagrams could be active at the same time [29, 34], and it was soon recognized that there needed to be a way to describe constraints how scenarios are allowed to overlap. Some approaches used event pre- and post-conditions [40], others required the definition of the component states within the scenarios [8, 9], temporal logic constraints [4, 37, 38]. or proposed modeling positive and negative scenarios [24].

Addressing the problem of specifying overlapping scenarios, Damm and Harel introduced *Live Sequence Charts* (LSC) [5], which recognized that there is a difference between scenarios that should be possible to occur, and scenarios that express rules on the behavior that must always be satisfied. These scenario kinds are called *existential* resp. *universal* scenarios. Moreover, in universal scenarios it should be possible to distinguish between events that must occur and events that may occur or act as triggers for subsequent must-occur events. These are called *hot* (must occur) resp. *cold* (may occur) events. Likewise, hot and cold conditions can model mandatory- or interrupt conditions.

Based on LSC, the Play-In/Play-Out approach was developed [21]. with the vision to interactively "play" with a system to teach it possible, allowed, or forbidden behaviors, in the form of LSC diagrams. These LSC specification can then be executed using the Play-Out algorithm. There were numerous approaches to synthesize state-based models from LSC specifications [2, 20, 32, 33], and checking whether this is at all possible; combinations of must-happen and must-not-happen requirements can render LSC specifications *unrealizable*, meaning that no implementation of the specification exists. Our own work included realizability-checkig and synthesis techniques for timed LSC specifications [10, 18], and LSC-based product line specifications [12, 19]. Also the Petri Nets community suggested analysis techniques for LSCs, for example analyzing LSC specifications by mapping to *Colored Petri Nets* (CPN) [27] or by a procedure for composing Petri Nets from LSC-inspired Petri Net fragments [7].

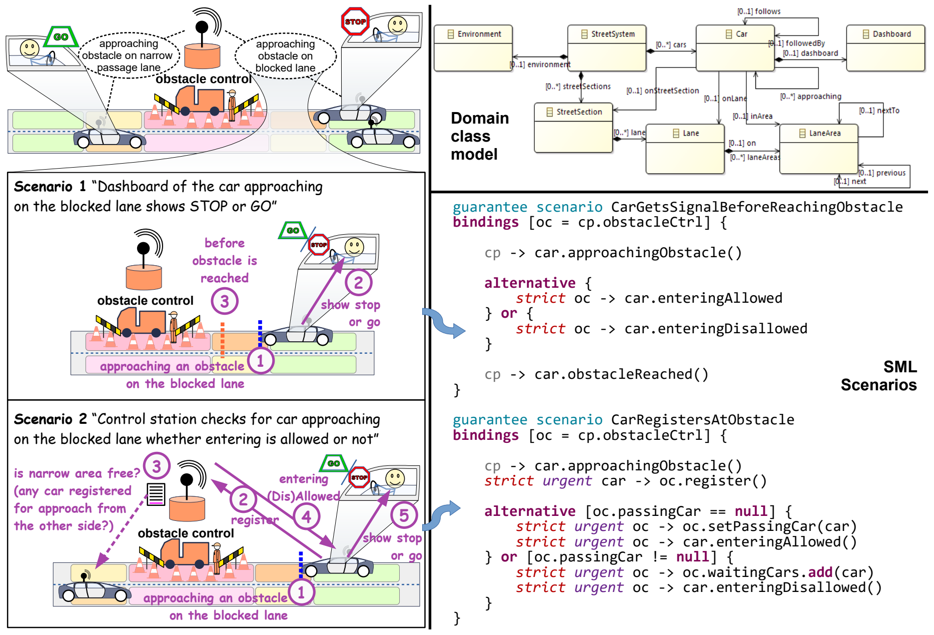
With *Behavioral Programming* (BP) [23], the idea of LSCs was transferred to general-purpose programming languages, for example Java, JavaScript or C++. BP allows programmers to program behavior by loosely coupled *behavioral threads* (b-threads), where each b-thread can add behavior or constrain the behavior of other b-threads, allowing for a flexible composition of behavior.

3 Scenario Modeling Language (SML)

With the goal of extending existing LSC modeling techniques and experimenting with analysis algorithms, we developed the textual Scenario Modeling Language (SML), along with the Eclipse/EMF-based tool suite ScenarioTools. We chose a textual language for improved usability over a graphical notation.

SML extends LSC with the capability to model assume/guarantee specifications, where *guarantee scenarios* specify how a system must react to environment events and *assumption scenarios* specify how the environment behaves. This gives SML an expressive power comparable GR(1) [1], a subset of LTL. A GR(1) specification is an implications of two generalized Büchi properties. For example, we can specify that cars must always be able to pass crosswalks in a traffic system assuming that each crosswalk is always eventually free of pedestrians. ScenarioTools implements a GR(1) game solving algorithm [3] for realizability checking and controller synthesis. Moreover, SML extends previous LSC modeling techniques with features for modeling *dynamic topology systems*, which are systems with evolving structure [14].

Figure 1 shows an example of a Car-to-X system for coordinating cars that approach an obstacle blocking one lane of a two-lane road. The left illustrates two example scenarios with their SML counterpart shown on the right. SML specifications are typed over a domain class model (top right). For a more extensive description of SML and the example, we refer to the original paper [14].


GO
approaching obstacle on narrow passage lane
approaching obstacle on blocked lane
STOP
obstacle control
Environment
StreetSystem
Car
Dashboard
[0..1] follows
[0..*] cars
[0..1] followedBy
[0..1] dashboard
[0..1] environment
[0..*] streetSections
[0..1] onStreetSection
[0..1] onLane
[0..*] approaching
[0..1] inArea
[0..1] nextTo
StreetSection
Lane
LaneArea
[0..*] lane
[0..1] on
[0..*] laneAreas
[0..1] previous
[0..1] next
Domain class model
Scenario 1 "Dashboard of the car approaching on the blocked lane shows STOP or GO"
before obstacle is reached
obstacle control
show stop or go
approaching an obstacle on the blocked lane
Scenario 2 "Control station checks for car approaching on the blocked lane whether entering is allowed or not"
is narrow area free? (any car registered for approach from the other side?)
entering (Dis)Allowed
register
obstacle control
show stop or go
approaching an obstacle on the blocked lane
SML Scenarios

```
guarantee scenario CarGetsSignalBeforeReachingObstacle
bindings [oc = cp.obstacleCtrl] {

    cp -> car.approachingObstacle()

    alternative {
        strict oc -> car.enteringAllowed
    } or {
        strict oc -> car.enteringDisallowed
    }

    cp -> car.obstacleReached()
}

guarantee scenario CarRegistersAtObstacle
bindings [oc = cp.obstacleCtrl] {

    cp -> car.approachingObstacle()
    strict urgent car -> oc.register()

    alternative [oc.passingCar == null] {
        strict urgent oc -> oc.setPassingCar(car)
        strict urgent oc -> car.enteringAllowed()
    } or [oc.passingCar != null] {
        strict urgent oc -> oc.waitingCars.add(car)
        strict urgent oc -> car.enteringDisallowed()
    }
}
```

Fig. 1. Car-to-X SML specification example

Based on ScenarioTools, we experimented with different analysis techniques, for example symbolic execution [17], or automatic repair techniques [36].

4 Some Lessons Learned

The work on ScenarioTools and various examples [11, 13, 15, 16] has taught us several lessons:

1. LSC-/SML-style specification techniques indeed offer an intuitive way of modeling reactive behavior. They are more flexible and requirements-oriented than state-based models, but also more easily comprehensible than temporal logics due to offering a more step-by-step modeling style.
2. Constructs that forbid behavior should be used *sparingly* and *carefully*, as otherwise contradictions will be introduced easily. Strict messages (or hot messages in LSC), which are messages that, when enabled, forbid events appearing elsewhere in the scenario, are *harmful* in bigger scenarios, because one quickly looses the understanding in what constraints they actually imply. It is better to specify forbidden messages explicitly.
3. Simulation and formal realizability checking techniques are helpful in finding specification flaws. And specification flaws, just like bugs in other programs, are guaranteed to be introduced sooner or later.
4. For complex problems, it is difficult to understand the output generated by realizability checks. Even with interactive counter-play-out techniques that help the user by simulation to understand how the environment can force a violation, it may be difficult to understand the cause of the problem.
5. It is *difficult to trust* formal checks alone. Even if realizability checking returns a positive result, it is often difficult to understand which behaviors are or are not possible under the specification. The pragmatic approach is to specify *tests* that exercise intended use cases beginning-to-end. We suggest a *test-driven* approach, where scenarios are added just to a degree that satisfies the next test case. This way, it is possible to arrive at a specification with a trust that all use cases are covered. When changes break previous tests, it is immediately clear what change affected which steps of which previous use cases. Test-driven specification can be combined with realizability checking. The latter may uncover problematic cases that are not covered by the tests, for example untested combinations of inputs that lead to unforeseen combinations of active scenarios.

5 Scenario Modeling Language for Kotlin (SMLK)

Even though significant work went into building SML and tool support in ScenarioTools, we felt that eventually these tool developments would not help to make scenario-based modeling techniques better accepted in practice. SML and ScenarioTools could not fulfill the following requirements:

1. **Provide richer means for *programming* within the scenarios**: For example, doing things like filtering a list or querying complex conditions, can be done with ScenarioTools, but only indirectly by calling external methods. SML is extensible, but extending it to a fully-fledged programming language would be a major effort.
2. **Better integration with other libraries or frameworks**: Integrating the ScenarioTools execution engine with external frameworks, such as a driving simulator or Android apps [15], is possible, but requires a cumbersome development of integration layers.
3. **Lightweight and fast execution**: The ScenarioTools execution engine is heavy-weight as it depends on EMF. Moreover, because the engine is based on the interpretation of SML/EMF models, it has a relatively high memory footprint and sometimes initial startup times of a few seconds.

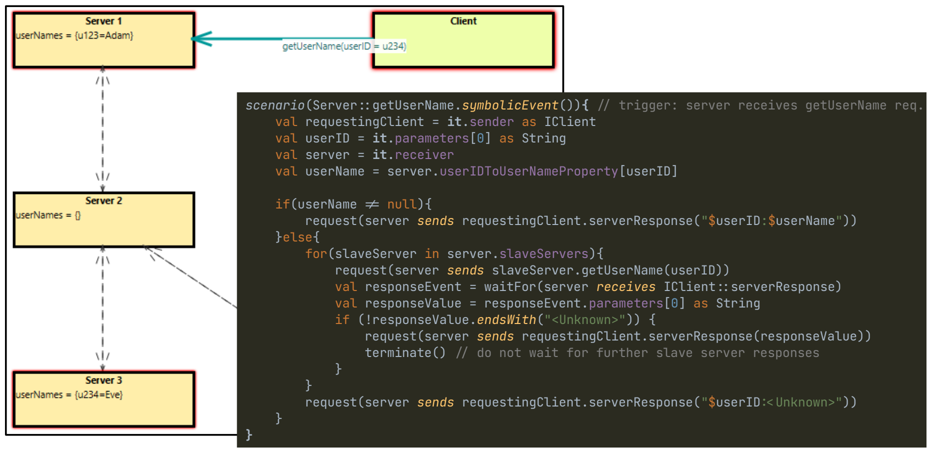
In addressing these requirements, we created an SML-like internal DSL in Java [28], based on the behavioral programming framework for Java, BPJ [22]. The approach, however, was too verbose. A few years later, Kotlin struck our attention, mainly for two reasons: First, Kotlin supports co-routines, which are lightweight threads that are less resource-intensive than Java threads. Therefore, they seemed a natural candidate for executing many concurrent scenarios. Second, Kotlin provides many features for building concise internal DSLs [1].

Exploiting these features, we created the Scenario Modeling Language for Kotlin (SMLK)[2]. In SMLK, multiple scenarios can be executed as a *scenario program*. Following the behavioral programming approach [23] a scenario program is executed as follows: The active scenarios execute their code independently until they reach *synchronization points* where each scenario can requests, wait-for or forbid events from happening. A central event selection algorithm selects an event that is requested by at least one scenario and not forbidden by any other scenario. All scenarios requesting or waiting for that event are then notified and can again independently execute their behavior until reaching the next synchronization point. This process is repeated until all scenarios terminate.

Figure 2 shows an example of an SMLK scenario for a simple hierarchical name lookup system: a client can send a request to a server to obtain the name for a user ID. If the requested server does not have any name stored locally for the given user ID, it will request servers one hierarchy level below. If one of the servers has the name, it will be returned upwards in the hierarchy to the requesting client. Otherwise the client receives an "unknown" response. The shown scenario may be triggered for multiple servers in a server hierarchy.

The client-server interaction illustration to the left of Fig. 2 is a screenshot from a JavaFX-based interaction animator application that animates the message interaction between objects[3].

[1] see for example`https://www.baeldung.com/kotlin/dsl`
[2] see `https://bitbucket.org/jgreenyer/smlk/`
[3] the animator project Git repository: `https://bitbucket.org/jgreenyer/smlk-animator/`; the hierarchical name lookup system example Git repository: `https://bitbucket.org/jgreenyer/smlk-animator-project-template/`



```
scenario(Server::getUserName.symbolicEvent()){ // trigger: server receives getUserName req.
    val requestingClient = it.sender as IClient
    val userID = it.parameters[0] as String
    val server = it.receiver
    val userName = server.userIDToUserNameProperty[userID]

    if(userName ≠ null){
        request(server sends requestingClient.serverResponse("$userID:$userName"))
    }else{
        for(slaveServer in server.slaveServers){
            request(server sends slaveServer.getUserName(userID))
            val responseEvent = waitFor(server receives IClient::serverResponse)
            val responseValue = responseEvent.parameters[0] as String
            if (!responseValue.endsWith("<Unknown>")) {
                request(server sends requestingClient.serverResponse(responseValue))
                terminate() // do not wait for further slave server responses
            }
        }
        request(server sends requestingClient.serverResponse("$userID:<Unknown>"))
    }
}
```

Fig. 2. An example SMLK scenario of a simple hierarchical name lookup system with interaction animator UI screenshot

SMLK integrates with JUnit and supports a test-driven scenario-specification methodology [42,44] and combining inter-component scenarios with component-specific scenarios [41,43].

SMLK can also be used to specify traffic scenarios, similar to OpenSCENARIO[4] or Traffic Sequence Charts [6], which are aimed at modeling tests for autonomous vehicles. Figure 3 shows a composed SMLK scenario that specifies a simple traffic scenario of one vehicle overtaking and then cutting a second vehicle [5]. Here, SMLK is integrated with SUMO; the box on the right of Fig. 3 shows snapshots of the resulting visual simulation output produced by SUMO.

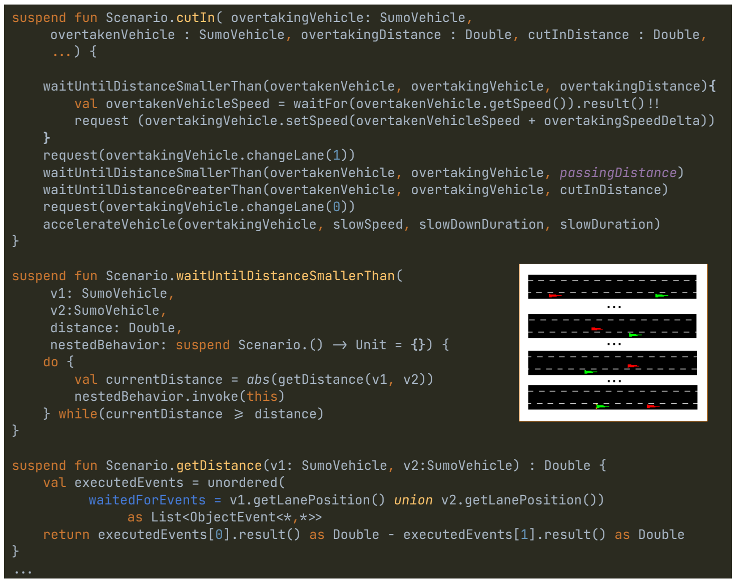
This example shall demonstrate the flexibility that SMLK offers in composing such scenarios: The main scenario `cutIn` starts by waiting until the overtaking vehicle approaches the second vehicle. This is performed by a sub-scenario `waitUntilDistanceSmallerThan` that, as a parameter, can receive a scenario that is executed in each step while waiting for the distance condition to render true. Here, this is used to ensure that the overtaking vehicle is faster than the vehicle to be overtaken. The distance is calculated by a sub-scenario `getDistance` that waits for the vehicles' position events, and then returns the difference.

Of course, multiple such scenarios can be executed in parallel to direct and coordinate the movement of many vehicles.

6 Conclusion and Open Research Challenges

We hope that we could provide an interesting, albeit self-biased, overview on the past work on and current state of scenario-based modeling techniques. Further

[4] `https://www.asam.net/project-detail/asam-openscenario-v20-1/`
[5] inspired by `https://releases.asam.net/OpenSCENARIO/1.0.0/ASAM_OpenSCENARIO_BS-1-2_User-Guide_V1-0-0.html#_cut_in`



```
suspend fun Scenario.cutIn( overtakingVehicle: SumoVehicle,
     overtakenVehicle : SumoVehicle, overtakingDistance : Double, cutInDistance : Double,
     ...) {

    waitUntilDistanceSmallerThan(overtakenVehicle, overtakingVehicle, overtakingDistance){
        val overtakenVehicleSpeed = waitFor(overtakenVehicle.getSpeed()).result()!!
        request (overtakingVehicle.setSpeed(overtakenVehicleSpeed + overtakingSpeedDelta))
    }
    request(overtakingVehicle.changeLane(1))
    waitUntilDistanceSmallerThan(overtakenVehicle, overtakingVehicle, passingDistance)
    waitUntilDistanceGreaterThan(overtakenVehicle, overtakingVehicle, cutInDistance)
    request(overtakingVehicle.changeLane(0))
    accelerateVehicle(overtakingVehicle, slowSpeed, slowDownDuration, slowDuration)
}

suspend fun Scenario.waitUntilDistanceSmallerThan(
     v1: SumoVehicle,
     v2:SumoVehicle,
     distance: Double,
     nestedBehavior: suspend Scenario.() → Unit = {}) {
    do {
        val currentDistance = abs(getDistance(v1, v2))
        nestedBehavior.invoke(this)
    } while(currentDistance ⩾ distance)
}

suspend fun Scenario.getDistance(v1: SumoVehicle, v2:SumoVehicle) : Double {
    val executedEvents = unordered(
          waitedForEvents = v1.getLanePosition() union v2.getLanePosition())
              as List<ObjectEvent<*,*>>
    return executedEvents[0].result() as Double - executedEvents[1].result() as Double
}
...
```

Fig. 3. An example of composing SMLK scenario for specifying traffic scenarios that can be executed in the SUMO traffic simulator

research is yet required to evaluate and improve these techniques. We conclude with a list of three interesting research directions:

1. **Evaluation using larger-scale case studies:** The existing literature of scenario-based modeling/programming techniques only reports on small- and medium-size case studies. It is time to exercise the development of more complex distributed cyber-physical systems or micro-service systems in order to reflect on the strengths and weaknesses of scenario-based techniques.
2. **Improve realizability checking techniques and comprehensibility of the results:** We highlighted the benefit and problems of formal realizability checking techniques. There are different ways in which these techniques could be improved. For example, could synthesized strategies or counter-strategies be presented to the user in more concise forms instead of producing huge state graph models? Could program repair techniques be applied in order to suggest scenario specification repairs if realizability checking fails?
3. **Specify and simulate traffic scenarios for testing autonomous systems**: This seems a fitting area for applying scenario-based specification techniques as they support a step-by-step operational description of what shall happen in some places while also allowing for a more abstract, non-deterministic specification of behavior elsewhere. Tailoring tests for autonomous vehicles that effectively and efficiently test behaviors of interest and support assessing residual risks [25,35] is a great challenge. Could controller synthesis and learning techniques support test engineers in this process?

Acknowledgements The work on SML, ScenarioTools and the modeling and analysis of dynamic topology systems was funded by grant no. 1258 of the German-Israeli Foundation for Scientific Research and Development (GIF). The work was the result of a fruitful collaboration notably with David Harel, Assaf Marron, Shlomi Steinberg, Daniel Gritzner, Nils Glade, Timo Gutjahr, Florian König, Eric Wete, Jianwei Shi, and Larissa Chazette. The work on test-driven scenario specification employing SMLK is the result of a collaboration with Carsten Wiecher. Integrating SMLK with SUMO for traffic system specifications was supported by Maximilian Hölscher.

References

1. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (May 2012). https://doi.org/10.1016/j.jcss.2011.08.007
2. Bontemps, Y., Heymans, P., Schobbens, P.Y.: From live sequence charts to state machines and back: a guided tour. IEEE Transactions on Software Engineering **31**(12), 999–1014 (2005). https://doi.org/10.1109/TSE.2005.137
3. Chatterjee, K., Dvorák, W., Henzinger, M., Loitzenbauer, V.: Conditionally Optimal Algorithms for Generalized Büchi Games. In: Faliszewski, P., Muscholl, A., Niedermeier, R. (eds.) 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016). Leibniz IntProceedings in Informatics (LIPIcs), vol. 58, pp. 25:1–25:15. Dagstuhl, Germany (2016). https://doi.org/http://dx.doi.org/10.4230/LIPIcs.MFCS.2016.25
4. 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
5. Damm, W., Harel, D.: Lscs: Breathing life into message sequence charts. Formal Methods in System Design **19**(1), 45–80 (Jul 2001). https://doi.org/10.1023/A:1011227529550
6. Damm, W., Kemper, S., Möhlmann, E., Peikenkamp, T., Rakow, A.: Using traffic sequence charts for the development of havs. In: European Congress on Embedded Real Time Software and Systems 2018. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018) (2018), `https://hal.archives-ouvertes.fr/hal-01714060/file/ERTS_2018_paper_17.pdf`
7. Fahland, D.: Oclets — scenario-based modeling with petri nets. In: Franceschinis, G., Wolf, K. (eds.) Applications and Theory of Petri Nets, Lecture Notes in Computer Science, vol. 5606, pp. 223–242. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02424-5_14
8. Giese, H., Henkler, S., Hirsch, M., Klein, F.: Nobody's perfect: Interactive synthesis from parametrized real-time scenarios. In: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. p. 67–74. SCESM '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1138953.1138967
9. Giese, H., Tissen, S.: The SceBaSy Plugin for the Scenario-Based Synthesis of Real-Time Coordination Patterns for Mechatronic UML. In: Proceedings of the 3rd International Fujaba Days 2005, Paderborn, Germany (2005)

10. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn, Paderborn (October 2011)
11. Greenyer, J., Bar-Sinai, M., Weiss, G., Sadon, A., Marron, A.: Modeling and programming a leader-follower challenge problem with scenario-based tools. In: Hebig, R., Berger, T. (eds.) Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings, vol. 2245, pp. 376–385. CEUR (2018)
12. Greenyer, J., Brenner, C., Cordy, M., Heymans, P., Gressi, E.: Incrementally synthesizing controllers from scenario-based product line specifications. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. p. 433–443. ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491445
13. Greenyer, J., Chazette, L., Gritzner, D., Wete, E.: A scenario-based MDE process for dynamic topology collaborative reactive systems – early virtual prototyping of Car-to-X system specifications. In: Schaefer, I., Cleophas, L., Felderer, M., Becker, S., Böhm, W., Fahland, D., Fill, H., Heinrich, R., Kirchner, K., Köhler-Bußmeier, M., Konersmann, M., Mayr, H.C., Moldt, D., Oberweis, A., Reher, F., Riebisch, M., Sauer, S., Schlingloff, H., Thalheim, B., Vogelsang, A., Weißbach, R., Weyer, T. (eds.) Proceedings Workshops zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES), Joint Proceedings of the Workshops co-located with Modellierung 2018, Braunschweig, Germany, February 21, 2018. CEUR Workshop Proceedings, vol. 2060, pp. 111–120. CEUR-WS.org (2018)
14. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools. In: de Lara, J., Clarke, P.J., Sabetzadeh, M. (eds.) Proceedings of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016). vol. 1725, pp. 16–32. CEUR (2016)
15. Greenyer, J., Gritzner, D., Katz, G., Marron, A., Glade, N., Gutjahr, T., König, F.: Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. Procedia Technology (Proceedings of the 3nd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering, SysInt 2016) (2016), 3rd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering (SysInt 2016)
16. Greenyer, J., Gritzner, D., Shi, J., Wete, E.: A scenario-based MDE process for developing reactive systems: A cleaning robot example. In: Burgueño, L., Corley, J., Bencomo, N., Clarke, P.J., Collet, P., Famelis, M., Ghosh, S., Gogolla, M., Greenyer, J., Guerra, E., Kokaly, S., Pierantonio, A., Rubin, J., Ruscio, D.D. (eds.) Proceedings of MODELS 2017 Satellite Events, co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017). CEUR Workshop Proceedings, vol. 2019, pp. 71–80. CEUR (2017)
17. Greenyer, J., Gutjahr, T.: Symbolic execution for realizability-checking of scenario-based specifications. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 312–322 (2017). https://doi.org/10.1109/MODELS.2017.35
18. Greenyer, J., Kindler, E.: Compositional synthesis of controllers from scenario-based assume-guarantee specifications. In: Moreira, A., Schätz, B., Gray, J., Val-

lecillo, A., Clarke, P. (eds.) Model-Driven Engineering Languages and Systems: Proceedings of the ACM/IEEE 16th International Conference (MODELS 2013), Lecture Notes in Computer Science, vol. 8107, pp. 774–789. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_47

19. Greenyer, J., Molzam Sharifloo, A., Cordy, M., Heymans, P.: Features meet scenarios: modeling and consistency-checking scenario-based product line specifications. Requirements Engineering **18**(2), 175–198 (2013). https://doi.org/10.1007/s00766-013-0169-4, `http://dx.doi.org/10.1007/s00766-013-0169-4`
20. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Foundations of Computer Science **13:1**, 5–51 (2002)
21. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
22. Harel, D., Marron, A., Weiss, G.: Programming coordinated behavior in java. In: D'Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming. pp. 250–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_12
23. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Comm. ACM **55**(7), 90–100 (2012). https://doi.org/10.1145/2209249.2209270
24. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software & Systems Modeling **4**(4), 355 (Oct 2005). https://doi.org/10.1007/s10270-005-0087-0
25. Hollander, Y.: Estimating the residual risk of adas / av (Feb 2021), `https://blog.foretellix.com/2021/02/01/estimating-the-residual-risk-of-adas-av/`
26. ITU: ITU-TS Recommendation Z.120: Message Sequence Charts (MSC) (1996)
27. Khadka, B., Mikolajczak, B.: Transformation from live sequence charts to colored petri nets. In: Proceedings of the 2007 Summer Computer Simulation Conference. p. 673–680. SCSC '07, Society for Computer Simulation International, San Diego, CA, USA (2007)
28. König, F.W.H.: Szenariobasierte Programmierung und verteilte Ausführung in Java. Master's thesis, Leibniz Universität Hannover, Software Engineering Group (2017), `http://jgreen.de/wp-content/documents/msc-theses/2017/Koenig2017.pdf`
29. Koskimies, K., Systa, T., Tuomi, J., Mannisto, T.: Automated support for modeling OO software. IEEE Software **15**(1), 87–94 (1998). https://doi.org/10.1109/52.646888
30. Krüger, I.: Distributed System Design with Message Sequence Charts. Ph.D. thesis, Technische Universität München, Institut für Informatik (2000)
31. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In: DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems. pp. 61–71. Kluwer Academic Publishers, Norwell, MA, USA (1999)
32. Kugler, H., Plock, C., Pnueli, A.: Controller synthesis from LSC requirements. In: Chechik, M., Wirsing, M. (eds.) Proceedings of the 12th International Conference of Fundamental Approaches to Software Engineering, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Lecture Notes in Computer Science, vol. 5503, pp. 79–93. Springer (2009)
33. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS

2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. pp. 77–91 (2009)
34. Maier, T., Zündorf, A.: The Fujaba Statechart Synthesis Approach. In: Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003 (2003)
35. Neurohr, C., Westhofen, L., Butz, M., Bollmann, M.H., Eberle, U., Galbas, R.: Criticality analysis for the verification and validation of automated vehicles. IEEE Access **9**, 18016–18041 (2021). https://doi.org/10.1109/ACCESS.2021.3053159
36. Schmelter, D., Greenyer, J., Holtmann, J.: Toward learning realizable scenario-based, formal requirements specifications. In: 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), 4th International Workshop on Artificial Intelligence for Requirements Engineering (AIRE). pp. 372–378 (Sept 2017). https://doi.org/10.1109/REW.2017.14
37. Uchitel, S., Brunet, G., Chechik, M.: Behaviour Model Synthesis from Properties and Scenarios. In: Proceedings of the 29th international Conference on Software Engineering, ICSE '07. pp. 34–43. IEEE Computer Society, Washington, DC, USA (2007). https://doi.org/http://dx.doi.org/10.1109/ICSE.2007.21
38. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. IEEE Transactions on Software Engineering **35**(3), 384–406 (2009). https://doi.org/10.1109/TSE.2008.107
39. Uchitel, S., Kramer, J.: A Workbench for Synthesising Behaviour Models from Scenarios. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001. pp. 188–197 (2001). https://doi.org/10.1109/ICSE.2001.919093
40. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd international conference on Software engineering, ICSE '00. pp. 314–323 (2000). https://doi.org/10.1109/ICSE.2000.870422
41. Wiecher, C., Greenyer, J.: Besos: A tool for behavior-driven and scenario-basedrequirements modeling for systems of systems. In: Aydemir, F., Gralha, C., Abualhaija, S., Breaux, T., Daneva, M., Ernst, N., Ferrari, A., Franch, X., Ghanavati, S., Groen, E., Guizzardi, R., Guo, J., Herrmann, A., Horkoff, J., Mennig, P., Paja, E., Perini, A., Seyff, N., Susi, A., Vogelsang, A. (eds.) Joint Proceedings of REFSQ-2021 Workshops, OpenRE,Posters and Tools Track, and Doctoral Symposium, Essen, Germany, 12-04-2021. CEUR Workshop Proceedings, vol. 2857. CEUR (2021)
42. Wiecher, C., Greenyer, J., Korte, J.: Test-driven scenario specification of automotive software components. In: The 2nd International Workshop on Modeling in Automotive Software Engineering, Proceedings of MODELS 2019 Workshops (to appear), co-located with ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019), Munich, Germany, September 2019. CEUR Workshop Proceedings, CEUR (2019)
43. Wiecher, C., Greenyer, J., Wolff, C., Anacker, H., Dumitrescu, R.: Iterative and scenario-based requirements specification in a system of systems context. In: Dalpiaz, F., Spoletini, P. (eds.) Requirements Engineering: Foundation for Software Quality (REFSQ 2021). pp. 165–181. Springer International Publishing, Cham (2021)
44. Wiecher, C., Japs, S., Kaiser, L., Greenyer, J., Dumitrescu, R., Wolff, C.: Scenarios in the loop: Integrated requirements analysis and automotive system validation. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. MODELS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3421264