A Scenario-Based MDE Process for Dynamic Topology Collaborative Reactive Systems – Early Virtual Prototyping of Car-to-X System Specifications[1]

Joel Greenyer[2], Larissa Chazette[2], Daniel Gritzner[2], Eric Wete[2]

Abstract: Car-to-X systems are safety-critical dynamic topology reactive systems (DTRSs), consisting of collaborating reactive components with relationships and responsibilities that change at run-time. This induces substantial complexity, and engineers need adequate means to model and validate such systems already during the early design. To address this challenge, we developed a scenario-based development process (SBDP) where DTRS requirements and environment assumptions are modeled as independent scenarios, which can be analyzed formally and compiled into executable code. In this paper we apply SBDP to a Car-to-X driver assistance system. We created a virtual prototyping environment where the generated code is executed in a distributed system with Android devices acting as the cars' dashboards; the driving is simulated by a 3D simulator (OpenDS). This work shows how the scenario-based design approach can be integrated with domain-specific simulators, which can help clarify design issues. Moreover, it shows that scenario-based code could drive the final system.

Keywords: Reactive Systems; Dynamic Topology; Scenario-Based Specification; Collaborative Systems; Model-Driven Engineering; Simulation

1 Introduction

Software-intensive systems in areas like transportation, production, or avionics often consist of multiple reactive components that collaborate with each other and their environment. Moreover, systems like mobile robot systems or cooperating cars (Car-to-X systems) have a *dynamic topology*, which means that relationships between the components can change at run-time, for example due to the physical movement of components. These changing relationships influence the behavior of the components, which must fulfill context-specific responsibilities. We call such systems *Dynamic Topology Reactive Systems (DTRSs)*.

The design of DTRSs can be a complex challenge, not only due to their dynamic topology, but also because of the distributed and concurrent nature of their software, and because they often control complex physical/mechanical processes. Finally, the systems are often safety-critical, and extra rigor is required during design.

[1] Funded by the German Israeli Foundation for Scientific Research and Development (GIF), grant No. 1258.

[2] Leibniz Universität Hannover, Software Engineering Group, Welfengarten 1, 30827 Hannover, Germany greenyer@inf.uni-hannover.de, larissa.chazette@inf.uni-hannover.de, daniel.gritzner@inf.uni-hannover.de

To address this challenge, we developed a formal scenario-based design approach for DTRSs, based on the Scenario Modeling Language (SML) [GGG$^+$17], an extended textual variant of Live Sequence Charts (LSCs) [DH01, HM03], where context-specific requirements and environment assumptions can be specified as a set of separate *guarantee* and *assumption scenarios* that form a scenario-based assume/guarantee specification. This approach has several advantages. First, the scenarios are aligned with how humans conceive and communicate requirements during the early design. Second, they have a formal semantics and can be formally analyzed for inconsistencies [GBC$^+$13]. Third, they can be executed via the play-out algorithm [HM03, GGG$^+$17], which makes it possible to analyze the scenarios via simulation or even to use them as code for the final system.

For the latter purpose, we developed a *scenario-based programming* (SBP) framework that allows developers to program scenarios in Java, or to compile SML specification into SBP code [GGK$^+$17, GGSW17]. The code can also be executed in a distributed system [SGG$^+$17]. This yields an MDE approach, which we call the *Scenario-based Design Process* (SBDP).

In this paper, we show how SBDP is applied to model, formally analyze, and finally generate software for a Car-to-X system that helps drivers to safely pass obstacles that block one lane of a two-lane road. The specific novel aspect presented in this paper is that, instead of testing the software in real driving tests, we created a virtual prototyping system, where the driving is simulated in an interactive 3D driving simulator (OpenDS); the software runs on a distributed system of an obstacle controller component running on a laptop, and the cars' software running on Android devices that also act as the cars' dashboards.

This paper highlights two aspects of our work. First, this simulator acts as a proof-of-concept, showing that the code generated by SBDP could be executed in a distributed Car-to-X system. Second, it demonstrates how scenario-based modeling can be integrated with domain-specific simulators for the benefit that virtual prototyping can help clarify early design issues with stakeholders. In this case, the cars' coordination behavior scenarios can be experienced in a 3D simulation that allows also non-technical stakeholders to assess different driving situations and see how the scenarios interplay in these situations.

The integration with a domain-specific simulation tool can also help to validate the environment assumptions in the specification. For example, assumptions on the possible movements of cars could be overly strict, e.g. not consider that cars can do U-turns in certain places where indeed they can. If such assumption scenarios, which are also compiled into code, are violated during simulation, then engineers know which assumptions to re-assess. Formal consistency checks can only unfold their full potential if such assumption validation is done early as well.

Structure: We explain the example in Sect. 2, introduce SML and SBDP in Sect. 3 and 4. We then present the virtual prototyping tool in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

A demo video is available here: `https://youtu.be/Eiljxn3z1T8`

2 Example

As an example, we consider a car-to-x system that assists drivers in passing a narrow passage created by obstacles such as road works. Figure 1 shows a sketch. The dashed lines resemble certain points before the obstacle (approachingObstacle, obstacleReached, and enterNarrowPassage) that the cars will pass and which will trigger certain aspects of the obstacle coordination behavior. In the real system, these points could be markers on the street or derived from GPS or other sensor data (e.g. camera, radar, LiDAR).

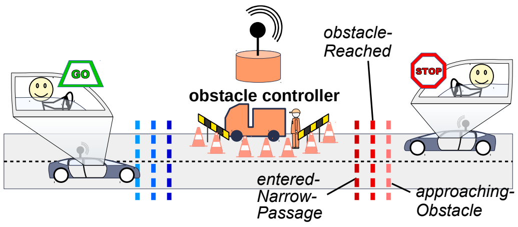

Fig. 1: Sketch of car-to-x narrow passage coordination assistance system

As an example, consider two guarantee scenarios that we formulate for this system:

G1: *When a car approaches the obstacle, the obstacle controller allows or disallows the car to enter the narrow passage before the car enters the narrow passage.* **G2**: *When a car approaches the obstacle, it registers at the obstacle controller. Then the obstacle controller checks whether another car is already registered for passing the obstacle. If so, the obstacle controller adds the approaching car to a waiting list and disallows it from entering; otherwise, it registers the car for passage and allows it to enter.*

G1 and **G2** describe complementary requirements: while both mention allowing or disallowing a car to enter, a non-deterministic choice in **G1** is refined in **G2**.

To specify the system further, more scenarios are added. For example, there are scenarios for cars approaching from the opposite direction or for the behavior of allowing a car to drive as soon as the narrow passage is cleared. There are also scenarios forbidding that cars collide head-on in the narrow passage.

The system is a dynamic topology system, because there can be multiple obstacles, even obstacles appearing or disappearing at run-time, and as cars move in the system they must coordinate around different obstacles, i.e., they must coordinate specifically around the ones that they are approaching. Such behavior can be modeled in SML, but we omit these details for brevity. To present our first proof-of-concept implementation of our 3D virtual prototyping environment, we limit the example to a system with one obstacle.

3 Scenario-Modeling Language (SML)

In SML, scenarios as described above can be modeled formally. Parts of the SML specification for the car-to-x system are shown in Listing 1. An SML specification defines how *objects* in an *object system* shall interact by exchanging messages. A specification first refers to a *domain model* (line 3) that defines the classes of objects that appear in the object systems, e.g. cars or obstacle controllers.

Then, the SML specification defines which classes of objects are *controllable* (line 5). Controllable are the components for which software is to be developed. Objects of classes not listed here are *uncontrollable*. Uncontrollable objects are sensors, actuators, and other external entities like users or external software components.

Furthermore, a specification contains one or more *collaborations* (line 7). A collaboration describes how objects shall interact in order to achieve a certain goal. A collaboration defines *roles* that are typed by classes in the domain model and represent objects in the object system. The behavior is defined by scenarios:

Guarantee scenarios describe what the system components may, must, or must not do in reaction to certain events. *Assumption Scenarios* describe what may, will, or will not happen in the environment of the system, or how the environment, in turn, reacts to the system. Each scenario essentially specifies an order of messages, and can contain control flow constructs like alternatives, parallel fragments, and loops.

For dynamic topology systems, the scenarios can specify topological conditions under which they apply and how roles bind to objects depending on the structural context. An SML specification can also specify how the system topology evolves on the occurrence of events, such as “the car moves”. Special messages can modify properties of receiving objects (*set-*, or *add-/remove-* messages for single- or multi-valued properties). Topology changes can also be modeled via graph transformation rules or programed transformation rules [GGG$^+$17]; we omit details for brevity.

Listing 1 shows how the two scenarios **G1** and **G2** presented on Sect. 2 are modeled using the SML language. The listing also shows a simple assumption scenario, which specifies that when the obstacle controller disallows a car to enter the narrow passage, the car must not enter until the obstacle controller allows it.

The *CoordinateProcessor* represents a sensor component for detecting approachingObstacle, obstacleReached, and enterNarrowPassage positions on the road. It also holds a pointer to the obstacle controller of the obstacle that the car is currently approaching; this pointer is updated specifically to the topological context, i.e. when the car passes one obstacle and approaches another, this link changes as well.

The scenario semantics, more specifically, is as follows:

Object system, message events, and run: We consider *synchronous* communication where

```
specification CarToXSpecification {

 domain cartox

 controllable { Car ObstacleController }

 collaboration CarsPassObstacle {
  dynamic role CoordinateProcessor cp
  dynamic role ObstacleController oc
  dynamic role Car car

  guarantee scenario CarGetsSignalBeforeReachingObstacle
  bindings [oc = cp.obstacleController] {
   cp -> car.approachingObstacle()
   alternative {
    strict oc -> car.enteringAllowed()
   } or {
    strict oc -> car.enteringDisallowed()
   }
   cp -> car.enterNarrowPassage()
  }

  guarantee scenario CarRegistersAtObstacle
  bindings [oc = cp.obstacleController] {
   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()
   }
  }

  assumption scenario DriverObeysSignal
  bindings [cp = car.cp] {
   oc -> car.enteringDisallowed()
   oc -> car.enteringAllowed()
  } constraints [ forbidden cp -> car.enterNarrowPassage() ]
  ...
 }
 ...
}
```

List. 1: Part of car-to-x SML specification

the sending and receiving of a message is a single *message event* (the concepts can be extended to asynchronous messages as well). A message event has one sending and one receiving object, refers to an operation defined for the receiving object, and carries values for parameters defined by its operation. A message event is *(un)controllable* if the sending object is (un)controllable. A message event may have *side-effects* as already mentioned above. An infinite sequence of message events and object systems (that evolve from an initial one) is called a *run*.

Active scenarios, role binding: A scenario *accepts* or *rejects* a run, and is interpreted as follows w.r.t. a run: As a message event occurs that corresponds to the first scenario message, an *active copy* of that scenario, also called *active scenario*, is created, and the sending and receiving roles of the scenario message are *bound* to the sending and receiving objects of the message event. Then *binding expressions* are evaluated to calculate bindings for other roles. The active scenario progresses on the occurrence of further events that match enabled messages under consideration of the assigned role bindings. An active scenario terminates when its final message is enabled and a matching event occurs. A scenario accepts a run if and only if there is never any *violation* in the process, as will be described in the next paragraph. There can be multiple active scenarios at the same time, even of the same scenario.

Message modalities (strict and urgent), violations, constraints: As long as a *strict* message is enabled, no message events must occur that corresponds to a message in the same scenario that is not currently enabled. If such a message does occur, this is called a *safety violation*. If a system message (sending role is typed by controllable class) is enabled that is *urgent*, this means that a corresponding message must occur before the next environment event occurs. If this does not happen, this is called a *liveness violation*. SML supports other modalities, also for modeling unbounded liveness properties, but we omit them for brevity. A scenario can also have a *constraints* section with *forbidden* messages. They represent events that must not occur while the scenario is active, otherwise leading to a safety violation.

Satisfying and SML specification, realizability: A run *satisfies* an SML specification if (a) it leads to no violations of any guarantee scenario or (b) there is a violation in at least one assumption scenario. Rationale: The guarantees need only be satisfied in environments that satisfy the assumptions. We assume a setting where the controllable objects are fast enough to send any finite number of messages before the next environment event occurs. If there exists a *strategy* for the controllable objects to send controllable messages in reaction to any sequence of uncontrollable events such that the resulting run satisfies the specification, then the specification is *realizable*; Otherwise it is *unrealizable*, which means that the environment can force the system to violate guarantees while satisfying the assumptions.

Play-out: The scenarios can also be executed via the *play-out algorithm* [HM03]. In a nutshell, the play-out algorithm waits for uncontrollable events until one activates one or several guarantee scenarios with enabled urgent controllable messages. The algorithm then selects one of these messages and executes the corresponding message event. This process is repeated until there are no further active guarantee scenarios with enabled urgent controllable messages. Then the play-out algorithm again waits for the next uncontrollable event, and the process is repeated.

4 Scenario-Based Design Process (SBDP)

The Scenario-Based Design Process (SBDP) is illustrated in Fig. 2. It is supported by ScenarioTools[3], an Eclipse-&EMF-based tool suite. After modeling the SML specification ①, this specification can be analyzed via the play-out algorithm or a formal realizability checking algorithm ②. The latter reduces the realizability checking problem to

[3] `http://scenariotools.org`

the problem of solving a *GR(1) game* [CDHL16]. For example. if we forgot the assumption DriverObeysSignal, then this algorithm can detect that the system cannot guarantee to avoid head-on collisions in the narrow passage.

Next, the SML-to-SBP compiler generates Scenario-Based Programming (SBP) code, where each scenario is a special thread; they interact to realize a play-out execution of the scenarios [GGK+17, KÏ7]. In the next step ④, code is added to bridge problem-specific events in the specification to platform-specific events. For example, the cars' position events may be derived from GPS sensor data.

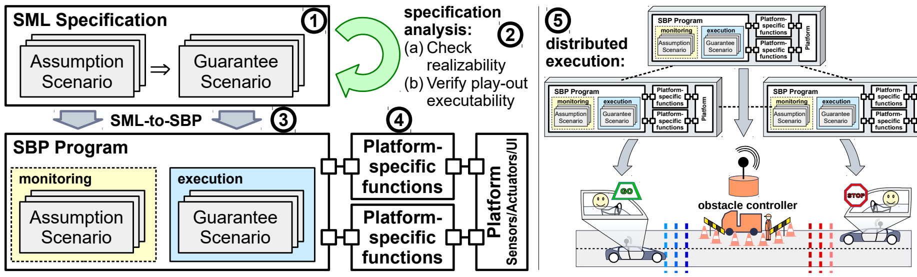

Fig. 2: Scenario-Based Development Process in a Distributed System

The SBP code can be deployed on a single node, or in a distributed setting ⑤. The latter works via a naive replicate-and-project approach that copies the complete code to each node in the system, while each node has a specific setting defining which object(s) in the object system a particular node represents. When running, all nodes synchronize on every event in the system via the network, which guarantees that all nodes' execution states are kept consistent. This, however, also creates a communication overhead, which we are currently seeking to reduce [SGG+17].

5 Virtual Prototyping Tool

We created a virtual prototyping tool by integrating the SBP code generated for our car-to-x example with OpenDS [MMMM13], a Java-based open source driving simulator tool. Here we overview the simulator's architecture.

The top of Fig. 3 shows how the simulator is operated: Two test drivers drive their cars in a multi-user interactive 3D driving simulation. Next to the screens that show the 3D simulation, mobile Android devices, acting as elements in the cars' dashboard, show the drivers whether they are allowed to drive or not.

The underlying architecture is shown on the bottom of Fig. 3: The multi-user driving simulation is realized by two connected OpenDS instances, running on two PCs (laptops). Three SBP components are deployed on three different hardware nodes: One component, the obstacle controller, runs on a PC (laptop), and the two car controller components run on mobile Android devices. In a real car-to-x system, the obstacle controller would run on a node of the road infrastructure. In a decentralized car-to-x system, each segment of a road or each city block possibly could have such a control station. Or, as illustrated in Fig. 1, if the obstacle appears in the form of a road work site, workers could set up such a communication node to run the obstacle controller on. In a real car-to-x system, the car components would run in the cars and show the signals on the dashboard or in a head-up display.

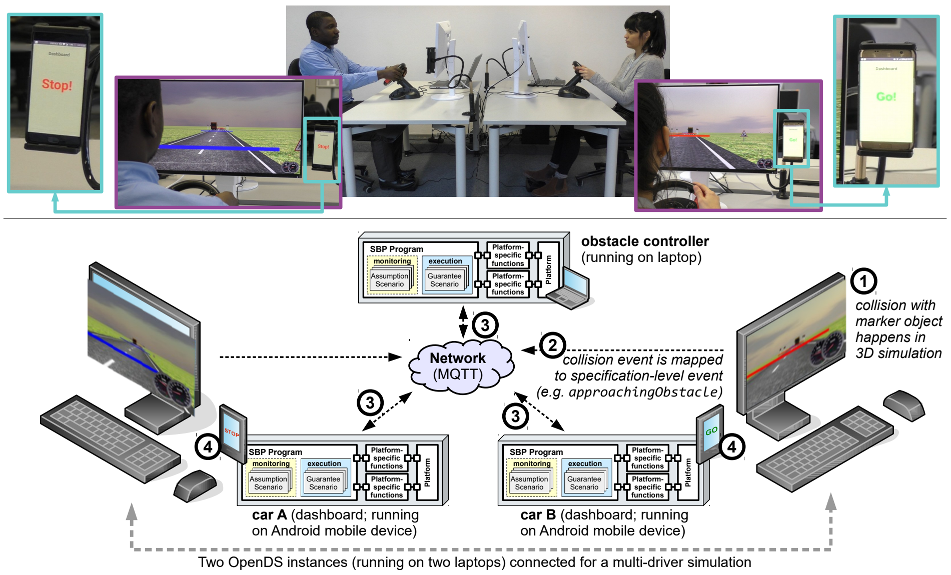
Stop!
Go!
SBP Program
monitoring
execution
Assumption Scenario
Guarantee Scenario
Platform-specific functions
Platform
obstacle controller
(running on laptop)
1
collision with marker object happens in 3D simulation
2
collision event is mapped to specification-level event (e.g. approachingObstacle)
3
Network (MQTT)
4
STOP
GO
car A (dashboard; running on Android mobile device)
car B (dashboard; running on Android mobile device)
Two OpenDS instances (running on two laptops) connected for a multi-driver simulation

Fig. 3: Test drivers operating the simulator and the underlying architecture

In a simulation, the obstacle coordination behavior is invoked as follows: As the drivers drive their cars in the 3D simulation and approach an obstacle, their cars collide with marker objects placed in the scene within certain distances of the obstacle (red and blue bars in the top part of Fig. 3). They represent certain points of interest around the obstacle (cf. Fig. 1). When a collision occurs ①, this event is translated into a corresponding specification-level event, e.g., approachingObstacle, which is then sent over a network ②. In our case, the network is an MQTT network, but also other network protocols can be used. All events are broadcast to all SBP components, which then collaboratively react to the events ③, which finally leads to STOP/GO signals being shown on the Android devices ④.

In order to achieve this OpenDS-SBP integration, it is mainly required to map simulation events, like cars colliding with position markers, into specification-level events ②. OpenDS offers ways to integrate such trigger code.

6 Related Work

There is previous work on executing LSCs with 3D simulations [HSKS08], but our work uniquely combines distributed play-out for a dynamic topology systems with a 3D simulator.

There are a number of approaches for modeling reactive system that also address dynamic topologies: MechatronicUML is a component- and statechart-based modeling methodology where reconfigurations can be modeled with graph transformations [BDG+14]. Kuhn et al. present a role-based modeling framework (FRaMED) for context-sensitive systems and systems where component relationships and roles may change [KBRA16]. In [TKG17], the authors propose a framework for systems in dynamic cyber-physical spaces based on bigraph transformations. With respect to these approaches, ours is different in that it supports a more flexible behavior modeling approach based on scenarios.

Autonomous transport robots used in production environments like factories and storage halls are another example of DTRSs. Distributed decision making algorithms such as shown in [SSJ16] are developed to support the development of collaborative teams of robots on these environments. Fault tolerance, flexibility and security are some of the properties that can be verified through model checking.

7 Conclusion

We showed how a scenario-based design process for DTRSs can be integrated with domain-specific simulators, for the example for car-to-x systems. The resulting virtual prototyping system can be created early, which can facilitate the validation of environment assumptions and the clarification of design issues with stakeholders.

For future work we plan to improve the distributed execution infrastructure, but also investigate how such systems can be used to systematically and automatically test systems with different initial topological configurations. We are also curious how to harness user feedback in order to refine, extend, or change the specification.

References

[BDG+14] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Sebastian Thiele, Wilhelm Schäfer, Matthias Meyer, Uwe Pohlmann, Claudia Priesterjahn, and Matthias Tichy. The MechatronicUML design method – process and language for platform-independent modeling. Technical report, Heinz Nixdorf Institute, 2014.

[CDHL16] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Veronika Loitzenbauer. Conditionally Optimal Algorithms for Generalized Büchi Games. In P. Faliszewski, A. Muscholl, and R. Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz IntProceedings in Informatics (LIPIcs)*, pages 25:1–25:15, Dagstuhl, Germany, 2016.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, volume 19, pages 45–80, 2001.

[GBC$^+$13] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications. In *Proc. 9th Joint Meeting on the Foundations of Software Engineering, ESEC/FSE 2013*, 2013.

[GGG$^+$17] Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Florian König, Nils Glade, Assaf Marron, and Guy Katz. ScenarioTools – A tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming*, 149(Supplement C):15 – 27, 2017. Special Issue on MODELS'16.

[GGK$^+$17] Joel Greenyer, Daniel Gritzner, Florian König, Jannik Dahlke, Jianwei Shi, and Eric Wete. From Scenario Modeling to Scenario Programming for Reactive Systems with Dynamic Topology. In *Proc. 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 974–978, New York, NY, USA, 2017. ACM.

[GGSW17] Joel Greenyer, Daniel Gritzner, Jianwei Shi, and Eric Wete. A Scenario-based MDE Process for Developing Reactive Systems: A Cleaning Robot Example. In L. Burgueño et al., editors, *Proc. MODELS 2017 Satellite Events*, volume 2019 of *CEUR Workshop Proceedings*, pages 71–80. CEUR, 2017.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[HSKS08] David Harel, Itai Segall, Hillel Kugler, and Yaki Setty. Crafting Game-models Using Reactive System Design. In *Proc. 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 121–128, New York, NY, USA, 2008. ACM.

[KÏ7] Florian Wolfgang Hagen König. Szenariobasierte Programmierung und verteilte Ausführung in Java. Master's thesis, Leibniz Universität Hannover, Software Engineering Group, 2017.

[KBRA16] Thomas Kuhn, Kay Bierzynski, Sebastian Richly, and Uwe Assmann. FRaMED: Full-fledge Role Modeling Editor (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA, 2016. ACM.

[MMMM13] Rafael Math, Angela Mahr, Mohammad M Moniri, and Christian Müller. OpenDS: A new open-source driving simulator for research. *GMM-Fachbericht-AmE 2013*, 2013.

[SGG$^+$17] Shlomi Steinberg, Joel Greenyer, Daniel Gritzner, David Harel, Guy Katz, and Assaf Marron. Distributing Scenario-based Models: A Replicate-and-Project Approach. In *Proc. 5th Int. Conf. on Model-Driven Engineering and Software Development - Vol. 1: MODELSWARD*, pages 182–195. INSTICC, ScitePress, 2017.

[SSJ16] Bernd-Holger Schlingloff, Henry Stubert, and Wojciech Jamroga. Collaborative embedded systems-a case study. In *Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC), 2016 3rd Int. Workshop*, pages 17–22. IEEE, 2016.

[TKG17] Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi. Modeling and Verification of Evolving Cyber-physical Spaces. In *Proc. 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 38–48, New York, NY, USA, 2017. ACM.