Scenarios@run.time – Distributed Execution of Specifications on IoT-Connected Robots*

Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Tim Duente, Stefan Dulle, Falk-David Deppe, Nils Glade, Marius Hilbich, Florian Koenig, Jannis Luennemann, Nils Prenner, Kevin Raetz, Thilo Schnelle, Martin Singer, Nicolas Tempelmeier, and Raphael Voges

Leibniz Universität Hannover, Fachgebiet Software Engineering, Welfengarten 1, D-30167 Hannover, Germany,
{greenyer|daniel.gritzner}@inf.uni-hannover.de

Abstract. In many areas we find cyber-physical systems consisting of multiple software-controlled components that communicate to control complex physical processes. As customers demand increasingly rich functionality, the component interactions become more and more complex. We are developing a formal scenario-based method for specifying the inter-component behavior that extends the concepts of Live Sequence Charts. This method is intuitive, yet precise, and automated analysis capabilities help engineers deal with the aforementioned complexity. In particular, the execution via the play-out algorithm supports a simulation of the behavior emerging from the interplay of the scenarios. Deriving a distributed implementation from an inter-component specification, however, is a challenging task. An alternative is the play-out of the specification by the distributed system. In this paper, we present a distributed play-out approach where the components coordinate via MQTT, a protocol used in IoT applications. We demonstrate the approach by a Car-to-X example implemented on Raspberry Pi-based robots.

Keywords: distributed system, play-out, scenario-based specification, executing specifications, Internet of Things, cyber-physical systems

1 Introduction

In many areas, for example, transportation, industry, and healthcare, we find systems consisting of multiple embedded components that cooperate to control complex physical processes and to interact with users. We also call these systems *cyber-physical systems*. Today, these system fulfill increasingly complex and critical functions, which makes their development challenging. One particular source of complexity is the distributed and concurrent nature of the software: single functions of the system are usually realized by the cooperation of several components, and a single component must often fulfill multiple functions at the same time.

* This work is funded by grant no. 1258 of the German-Israeli Foundation for Scientific Research and Development (GIF)

In order to help engineers deal with this complexity, we are developing a formal scenario-based method for specifying the interaction behavior of the components on an inter-component level. This method extends the concepts of Live Sequence Charts (LSCs) [5,11] and Modal Sequence Diagrams (MSDs) [10]. LSCs/MSDs are a visual formalism for specifying how a set of system components may, must, or must not react to external events. We are introducing the Scenario Design Language (SDL), which is a textual language based on LSCs/MSDs.

In particular, on this basis, the play-out algorithm [11] allows engineers to execute and thereby simulate a scenario-based specification. At design-time, this algorithm helps understand the interplay of the scenarios.

Eventually however, the inter-component specification must be transformed into an *intra-component implementation*. Due to the reasons given above, this is a challenging task. We are working on controller synthesis approaches for automating this transition [4] (also others [8,2,9]), but these techniques have limitations. Foremost, these approaches assume a specification for a static set of components. But, for many cyber-physical systems, we must assume that they are *dynamic*, i.e., there are many or even infinitely many configurations of components that may evolve at run-time—imagine a mobile system or communicating cars.

An alternative approach to arrive at a distributed implementation is distributing the play-out algorithm among the components. A naive realization of this approach is to let every component execute a play-out of the complete system with full synchronization of all components after each event, which is of course inefficient. Desirable would be to analyze the dependencies among the scenarios and the components, and minimize the necessary synchronization.

In a student project, we realized the naive approach as a first step towards a more elaborate solution. We implemented a distributed play-out where all components synchronize via the MQTT, a messaging protocol for Internet of Things (IoT) applications. We demonstrate the approach by a Car-to-X example running on Raspberry Pi-based robots (Fig. 1, `https://youtu.be/g0hcGSYC2Wk`).

The idea of distributing play-out is not new [1,13]; the particular novel contribution of this paper is a distributed play-out that (1) supports dynamic systems and dynamic bindings of scenarios to components in the system, and (2) is able to incorporate sensor/actuator events from embedded components in the system. Furthermore (3), we introduce our language SDL and a supporting tool [15].

Structure: in Sect. 2 we explain our example informally. SDL specifications are explained in Sect 3. We present our distributed implementation of the play-out in Sect. 4. Finally, we discuss related work in Sect. 5 and conclude in Sect. 6.

2 Example

As an example, we consider the specification of an advanced driver-assistance system that relies on the communication of cars and the road infrastructure (also more generally called *Car-to-X*- or *Vehicle-to-X* communication). Such systems are envisioned to coordinate the traffic more safely and efficiently in the future.

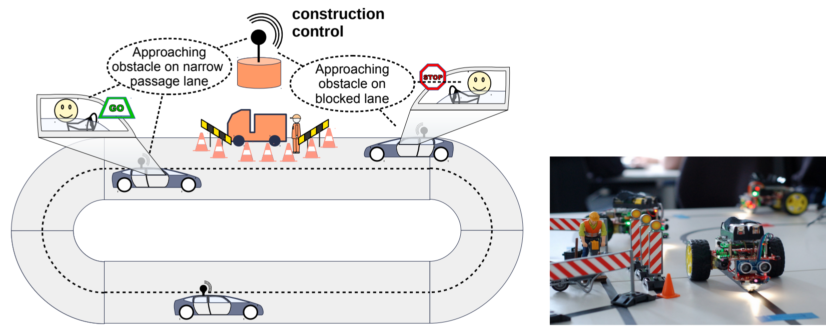

Fig. 1. Car-to-X example overview and photo of the demonstator

One use case that we consider is cars driving on a two-lane street that need to pass road works that block one lane. Fig. 1 shows an example of a simple street system with three cars and one construction site. We conceived the specification for this use case on the basis of descriptions of similar systems on the web[1].

During requirements engineering and early design, the behavior of the system in usually conceived in the form of scenarios. In one scenario, illustrated on the left of Fig. 2, an engineer specifies that (1) whenever a car approaches an obstacle on the blocked lane, (2) either a STOP or GO signal must be shown to the driver (3) before the car reaches the obstacle. In this scenario, the engineer does not specify any condition for when STOP or GO must be shown.

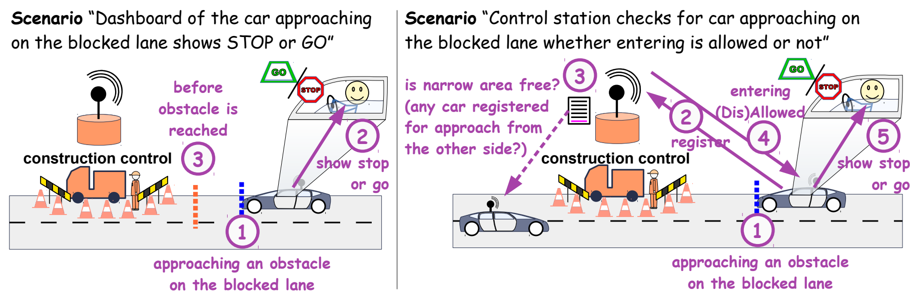

Fig. 2. Sketches of scenarios

In a second scenario, shown on the right of Fig. 2, the engineer specifies that (1) whenever a car approaches an obstacle on the blocked lane, (2) the car must register at the construction control. The construction control (3) must then check whether the narrow passage is free, i.e., whether any car approaching from the other side has already registered. If so, (4) the entering of the narrow passage

[1] see for example the use case by the CAR 2 CAR Communication Consortium "Warning of Roadworks": `https://www.car-2-car.org/index.php?id=149`

by the car approaching on the blocked lane must be disallowed, and it must be allowed otherwise. Last, (5) depending on the decision, the STOP or GO symbol must be shown to the user. Further scenarios would be described similarly.

The two scenarios above demonstrate that scenarios can overlap, i.e., they describe different aspects of the same situation. The final system behavior is defined by satisfying the requirements of all scenarios simultaneously.

3 Scenario Design Language Specification

The Scenario Design Language (SDL) is a textual variant of LSCs [5,11] and MSDs [10,3]. It allows us to specify, using scenarios, how a set of objects may, must, or must not react to external events. Scenarios can be *existential* or *universal*. Existential scenarios describe sequences of message events that must be possible to occur; universal scenarios describe properties that must hold for all sequences of message events. Here we focus on universal scenarios only.

Based on our experience with the graphical notation and modeling tools [3,7], we found a textual notation more user friendly and easier to extend.

3.1 Object system and message events

An SDL specification specifies the message-based interaction behavior of objects in an *object model*. The set of objects is partitioned into *controllable* objects, also called the *system*, and *uncontrollable* objects, also called the *environment*.

The objects can interchange *messages*. A message has one sending and one receiving object and refers to an operation of the receiving object's class. A message is a *system message* if it is sent by a system object and it is an *environment message* otherwise. Here we consider only synchronous messages where the sending and receiving together is a single *event*, also called *message event*. This restriction simplifies specifications, especially considering that the passage of time can not yet be modeled using SDL. An infinite sequence of message events is called an *execution* or *run*.

3.2 SDL specification, collaborations, and scenarios

An example SDL specification is shown in Listing 1. An SDL specification has a name (here "CarToX") and refers to a *domain* package (line 3) that contains a class model; the specification then specifies the behavior of instances of that class model. In our example, the class model defines the classes Car, Lane, Construction Control, etc.; we omit details for brevity. The specification furthermore defines that objects of certain classes are controllable or uncontrollable (line 5-7).

Next, an SDL specification defines one or multiple *collaborations*, which, in the style of UML, defines collaborating elements, also called *roles*, and how they collectively accomplish a desired functionality. Roles are typed by domain classes and they represent objects in an object model that can be the sender or receiver of messages. Roles can be *dynamic* or *static*. Dynamic roles can be bound to

different objects in the object model upon the activation of scenarios. Static roles are bound to one object. Here we only consider dynamic roles.

Each collaboration contains a set of *scenarios*. Each scenario refers to a set of roles of its collaboration. A scenario essentially contains a sequence of messages, but can also define conditions, or *alternative*-, *parallel*-, and *loop* fragments. A *message* in a scenario has the form ⟨`sender`⟩`->`⟨`receiver`⟩`.`⟨`operation`⟩, where ⟨`sender`⟩ and ⟨`receiver`⟩ are roles and ⟨`operation`⟩ is an operation of the receiving role's class. A message can have different *modalities*: it can be *strict* or *non-strict* and *requested* and *non-requested*.

We explain the semantics of the scenarios, messages, and the messages modalities by defining the following concepts: (in the lines of [11])

(1) event unification: A message event sent in the object model can be *unified* with a scenario message if the sending and receiving object of the message event are the objects bound to the respective roles of the scenario message.

(2) scenario activation and dynamic role binding: A scenario is *activated* if a message event occurs that can be unified with the first message in that scenario. We also say that an *active copy* of the scenario is created. At the time of activation, dynamic roles are unbound; in this case, a message event can be *unified* with a scenario message if the classes of the sending and receiving objects of the message event are equal to or subclasses of the classes typing the sending and receiving roles of the scenario message. Upon activation of the scenario, the roles of the first message are *bound* to the sending and receiving objects of the activating message event. Then the remaining roles are bound according to *binding expressions* (see e.g. lines 33-36). A role binding is defined for an active scenario, i.e., there can be multiple active copies of the same scenario with different role bindings, for example if different cars approach different obstacles.

(3) progress (enabled messages): In a scenario, messages can be *enabled*. After the activation of a scenario, the message following the first message is *enabled*. When a message event occurs that can be unified with the enabled message, the next message becomes enabled instead. This way, enabled messages indicate the progress of the active scenarios. There can also be multiple enabled messages in an active scenario if for example it contains a parallel fragment.

(4) scenario termination: If the last message in an active scenario is enabled and a message event occurs that can be unified with that last message, then the active scenario *terminates* and is discarded.

(5) violations: If a message event occurs that can be unified with a message in the scenario that is currently not enabled, we call this a *violation* of the scenario. If currently a strict message is enabled, this is a *safety violation*, which is forbidden to occur. If only non-strict messages are enabled, this is an *interrupting violation*, which is allowed, but will lead to a premature termination of the active scenario. If a requested message is enabled forever, because never any message event occurs that progresses or interrupts the scenario, this is a *liveness violation*.

(6) accepting runs: A (universal) scenario *accepts* a run if it does not lead to a safety violation or a liveness violation of the scenario. A specification accepts a run if all of its scenarios accept the run.

The scenarios DashboardOfCarApproachingOnBlockedLaneShowsStopOrGo and ControlStationChecksForCarApproachingOnBlockedLaneEnteringAllowed specify the scenarios introduced informally in Sect. 2.

```
1  system specification CarToX {
2
3    domain cartox //class model
4
5    define Car as controllable
6    define ObstacleControl as controllable
7    define Environment as uncontrollable
8    define Driver as uncontrollable
9    define Construction as uncontrollable
10
11   collaboration ApproachingObstacleOnBlockedLane {
12
13     dynamic role Environment env
14     dynamic role Driver driver
15     dynamic role Car car
16     dynamic role Construction construction
17     dynamic role Lane currentLane
18     dynamic role Lane nextLane
19     dynamic role ObstacleControl obstacleControl
20
21     specification scenario
           DashboardOfCarApproachingOnBlockedLaneShowsStopOrGo
22     with dynamic bindings [
23       bind driver to car.driver
24     ] {
25       message env -> car.approachingObstacle()
26       alternative { message strict requested car -> driver.showGo()
27       } or {        message strict requested car -> driver.showStop() }
28       message env -> car.obstacleReached()
29     }
30
31     specification scenario
           ControlStationChecksForCarApproachingOnBlockedLaneEnteringAllowed
32     with dynamic bindings [
33       bind driver to car.driver
34       bind currentLane to car.currentLane
35       bind construction to currentLane.obstacle
36       bind obstacleControl to construction.obstacleControl
37     ] {
38       message env -> car.approachingObstacle()
39       message strict requested car -> obstacleControl.register()
40       alternative if [ obstacleControl.narrowAreaFree ] {
41         message strict requested obstacleControl -> car.driveAllowed()
42         message strict requested car -> driver.showGo()
43       } or if [ !obstacleControl.narrowAreaFree ] {
44         message strict requested obstacleControl -> car.driveForbidden()
45         message strict requested car -> driver.showStop()
46       }
47
48     assumption scenario ApproachingObstacleEventSequence
49     with dynamic bindings [
50       bind currentLane to car.currentLane
51       bind nextLane to currentLane.next
52     ] {
53       message env -> car.roadSectionEntered()
54       interrupt if [ nextLane.obstacle == null ]
55       message strict requested env -> car.approachingObstacle()
56       message strict requested env -> car.obstacleReached()
57       message strict requested env -> car.overtakingObstacle()
58       message strict requested env -> car.obstacleAreaLeft()
59     }
60
61     specification scenario CarEntersNextLane with dynamic bindings [
```

```
62        bind currentLane to car.currentLane
63        bind nextLane to currentLane.next
64      ] {
65        message env -> car.roadSectionEntered()
66        message strict requested car -> car.setCurrentLane(nextLane)
67      }
68      ... // further scenarios
69    } // end collaboration ApproachingObstacleOnBlockedLane
70    ... // further collaborations
71  }
```

Listing 1. Example SDL specification

3.3 Assumption scenarios

We also support scenarios that allow us to specify what may, will, or will not happen in the environment. These scenarios we call *assumption scenarios* [6,3] as opposed to *specification scenarios* that specify requirements for the software. In our example, we assume that there are different events that occur as a car approaches and then reaches an obstacle, then overtakes the obstacle, and then finally leaves the narrow passage. The assumption scenario ApproachingObstacleEventSequence (line 48) specifies that once `env -> car.roadSectionEntered()` occurs, the events described in the scenario will occur exactly in that order.

3.4 Dynamic object model and message side-effects

The objects in an object model can carry values for attributes and links to other objects. These properties can be used to specify dynamic role bindings or condition expressions. They can also change as a side-effect of message events. By convention, message events that refer to operations of the form `set⟨Property⟩(value)` set a property value of the receiving object.

The scenario CarEntersNextLane in Listing 1 for example describes that, when a car enters a road section, it must also update its pointer to the current lane.

3.5 The play-out algorithm

The play-out algorithm [11] is an executable semantics for LSC/MSD, and also SDL specifications, which we extended to also consider assumption scenarios [3]. Play-out takes as input an SDL specification and a concrete object model that is an instance of the domain class model specified in the SDL specification. In a nutshell it works as follows: when an environment event occurs that activates or progresses one or multiple specification scenarios into a state where requested system messages are enabled, then a corresponding system message event is selected and executed, provided that it does not lead to a safety violation in any specification scenario. If subsequently further requested system messages are enabled in specification scenarios, repeatedly a next system message is chosen for execution. If no requested system messages are enabled in specification scenarios,

the algorithm waits for the next environment event to occur. Then this process is repeated.

During the play-out of our example specification shown in Listing 1, after an occurrence of `env -> car.approachingObstacle()`, we arrive in a state where the messages in lines 26+27 and 39 are enabled. `car -> driver.showGo()` and `car -> driver.showStop()` are requested, but they are currently blocked, because they would lead to a violation of the scenario ControlStationChecksForCarApproachingOnBlockedLaneEntering-Allowed, which requires `car -> obstacleControl.register()` and `obstacleControl -> car.driveAllowed()`/`obstacleControl -> car.driveForbidden()` to occur before STOP or GO is shown to the user.

4 Distributed Execution

We support the modeling and play-out of SDL specification within our Eclipse-based tool suite ScenarioTools [15]. As domain models, an SDL specification can refer to ECore class models of the Eclipse Modeling Framework (EMF). EMF allows us to create instances of that class model, for example to create an object model of the car system shown in Fig. 1. The play-out then interprets an SDL specification based on such an object model.

To realize the execution on a distributed robot system, we run the play-out engine of ScenarioTools on the robots or other components, for example the control station. In our case, the components are controlled by Raspberry Pis, for which the Java SE Virtual Machine exists. In our yet naive approach, each component runs a play-out of the complete system, and all components must synchronize, via MQTT, on each message event.

The complete behavior of the robots is modeled via SDL. The only exceptions are platform-specific mappings of sensor/actuator events, e.g., a value change of the RGB sensor may be mapped to a message event in the SDL specification. These mappings are implemented in Java. Our example does not consider the dynamic creation or destruction of objects at runtime. But the structural relationships between objects are dynamic and change during the execution.

We explain how our system works step by step, see the numbers in Fig. 3. Let us start with a robot that picks up an environment event via a sensor. In our example, the robots are equipped with RGB sensors to detect color tapes on the track (see photo in Fig. 1) that represent reaching certain points, e.g. "approaching obstacle". This event generates a message which is then published via the local MQTT client (1+2). The MQTT broker then broadcasts this message (3) to all clients. Upon receiving messages, each client will check if it corresponds to one of its actuator events, which would then be executed (4). This of course requires a platform-specific mapping of message events to sensor-/actuator events.

Each message received is forwarded to the *executor* (5), which calls the ScenarioTools play-out engine to execute it (6). ScenarioTools then updates a list of enabled requested system messages (7). If this list contains events sendable by the client, the executor selects one of them and publishes it (8).

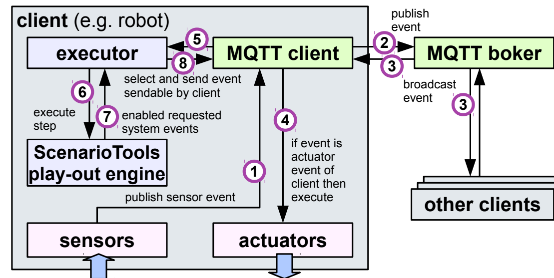

Fig. 3. Example of how messages generated by sensors propagate through the system.

In this approach, it may happen that an executor chooses a message A which leads to a safety violation due delays in the network communication. A message B already chosen and sent by another executor may alter the state of the play-out such that the message A is actually blocked leading to the aforementioned violation. This issue could be prevented by a defining a globally deterministic strategy for choosing events, or having an engineer explicitly specify the necessary synchronization.

Strengths of this approach are that scenarios are intuitive to write and assumption scenarios can help identify issues arising from engineers assuming that the environment to behaves differently.

5 Related Work

There exist approaches for synthesizing distributed finite-state controllers from LSC/MSD specifications [12,9,2,4], but they all assume a fixed and static object model. Ideas for distributing play-out exist [1,13], but they also assume a static object model, and the play-out is distributed for performance, rather than aiming to fit the architecture of a distributed embedded system.

Sousa et al. [16] describe a framework for executing DSL models, but do not address distributed execution. Sampaio et al. [14] show how to control a power micro-grid by executing an MGridML model that specifies event-condition-action rules. Distributed components are controlled by a central controller.

6 Conclusion

We developed a technique for the distributed play-out of SDL specifications, a textual variant of LSCs/MSDs. The approach can be used to implement specifications of complex systems with distributed and concurrent behavior. The novelty is that it supports dynamic object models and dynamic role bindings.

We evaluated the approach by a Car-to-X example that can be executed by Raspberry Pi-based robots. Our naive approach has limitations in performance

and scalability, which we plan to address in future work. Furthermore, we aim to consider runtime-adaptation to specification changes and recovery from failures.

References

1. Barak, D., Harel, D., Marelly, R.: Interplay: Horizontal scale-up and transition to design in scenario-based programming. Software Engineering, IEEE Transactions on 32(7), 467–485 (July 2006)
2. Bontemps, Y., Heymans, P.: From Live Sequence Charts to State Machines and Back: A Guided Tour. IEEE Transactions on Software Engineering 31(12), 999–1014 (2005)
3. Brenner, C., Greenyer, J., Panzica La Manna, V.: The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In: Proc.12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013). vol. 58. EASST (2013)
4. Brenner, C., Greenyer, J., Schäfer, W.: On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering (FASE 2015), Lecture Notes in Computer Science, vol. 9033, pp. 51–65. Springer (2015)
5. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer Academic Publishers (2001)
6. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn, Paderborn (October 2011)
7. Greenyer, J., Haase, M., Marhenke, J., Bellmer, R.: Evaluating a formal scenario-based method for the requirements analysis in automotive software engineering. In: Proc. 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE 2013 (2015)
8. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Foundations of Computer Science 13:1, 5–51 (2002)
9. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling. vol. 3393, pp. 309–324. Springer (2005)
10. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (2008)
11. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
12. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design. pp. 378–398. FMCAD '02, Springer (2002)
13. Merom, R.: Playing together: Distributed collaborative Play-Out of Live Sequence Charts. Master's thesis, Weizmann Institute of Science, Israel (2006)
14. Sampaio Jr., A., Costa, F., Clarke, P.: A model-driven approach to develop and manage cyber-physical systems. In: Proc. Models@run.time 2013 (2013)
15. ScenarioTools website. `http://scenariotools.org`
16. Sousa, G.C.M., Costa, F.M., Clarke, P.J., Allen, A.A.: Model-driven development of dsml execution engines. In: Proc. 7th Workshop on Models@Run.Time. pp. 10–15. MRT '12, ACM, New York, NY, USA (2012)