Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools *

Joel Greenyer[1], Daniel Gritzner[1], Guy Katz[2], and Assaf Marron[3]

[1] Leibniz Universität Hannover
greenyer|daniel.gritzner@inf.uni-hannover.de
[2] New York University guy.katz@nyu.edu
[3] Weizman Institute of Science assaf.marron@weizmann.ac.il

Abstract. Software-intensive systems such as communicating cars or collaborating robots consist of multiple interacting components, where physical or virtual relationships between components change at run-time. This dynamic system structure influences the components' behavior, which again affects the system's structure. With the often distributed and concurrent nature of the software, this causes substantial complexity that must be mastered during system design. For this purpose, we propose a specification method that combines scenario-based modeling and graph transformations. The specifications are executable and can be analyzed via simulation. We furthermore developed a formal synthesis procedure that can find inconsistencies or prove the specification's realizability. This method is implemented in ScenarioTools, an Eclipse-based tool suite that combines the Scenario Modeling Language, an extended variant of LSCs, and graph transformations modeled with Henshin. The particular novelty is the synthesis support for systems with dynamic structure.

Keywords: reactive systems, dynamic system structure, scenario-based specification, graph transformation, analysis, specification inconsistency, realizability, controller synthesis, Scenario Modeling Language, Live Sequence Charts

1 Introduction

In domains such as manufacturing, transportation, or logistics, we often find critical systems that consist of multiple software-intensive components that collaborate in order to control physical processes and react to user input. In systems like communicating cars, mobile robot systems or adaptive production systems, the physical or virtual relationships between system components may change at run-time, for example due to the physical movement of components or users, or due to changing roles and responsibilities of the system components. This

* Funded by grant no. 1258 of the German-Israeli Foundation for Scientific Research and Development (GIF). See demo video here: https://youtu.be/p9mo6FJvqEE

dynamic system structure influences the behavior of the software-intensive components, and the software can again influence the system's structure.

Take for example a Car-to-X communication system: the system structure can change due to the movement of the cars or the occurrence of obstacles (change of physical relationships), or due to the assignment of roles, such as leader and followers in a convoy (change of virtual relationships). A car's software must then behave differently depending on the specific traffic situation and the specific role of the car in that context. Furthermore, the car's software can influence how the system structure evolves subsequently, either by advising the driver or by controlling the car directly. On top of this, a car can be involved in different collaborations at the same time, for example convoy management and collision avoidance coordination at an obstacle.

This induces substantial complexity compared to static systems: not only do we need to develop systems with distributed and concurrent software, but the components' behavior is also context sensitive to and in tight interrelation with the evolving system structure.

To master this complexity, we propose a specification method that combines formal scenario-based modeling and graph transformations. This method is implemented in ScenarioTools [7], an Eclipse-based tool suite. It combines the Scenario Modeling Language (SML) and graph transformations modeled with Henshin [6,1]. SML is a textual variant of Live Sequence Charts (LSCs) [4], and extends LSCs with constructs for modeling environment assumptions.

The scenario-based paradigm allows engineers to capture specifications in a way that is very close to how they are naturally conceived and communicated during the early design. The specifications are executable via an extension of the play-out algorithm [4] and so the interplay of the scenarios can be analyzed for inconsistencies by simulation. Since simulation can naturally not prove the absence of flaws, we furthermore developed a formal controller synthesis procedure that can find inconsistencies or prove the specification's realizability.

In this tool demonstration paper, we present the modeling, simulation and controller synthesis capabilities of ScenarioTools based on a Car-to-X example. The modeling approach and a prototype tool were already presented in previous work [8]. We have since reimplemented the tool suite, switching to the textual Scenario Modeling Language (SML). The key novelty, however, is that the synthesis now supports specifications of systems with dynamic structure.

2 Example and Modeling Approach

As an example, we present a Car-to-X system that assists drivers in passing a narrow passage created by road works that block one lane of a two-lane street. Figure 1 shows a sketch where a car approaching the road works on the blocked lane must stop and yield to a car approaching from the opposite direction.

The lanes of the street are subdivided into lane areas. One lane area is blocked by the road works. One scenario from the system's specification (**Scenario 1** illustrated in Fig. 1) demands that whenever a car approaches the obstacle on

the blocked lane, it must show either a STOP or GO signal to the driver, and this signal must be shown before the car finally reaches the obstacle.

A second scenario (**Scenario 2** in Fig. 1) extends the behavior described by the first: it requires that when a car approaches the obstacle on the blocked lane, it must register at a control station. This *obstacle control* then must check whether another car has registered for approach from the opposite direction. If so, it must disallow the first car to enter and the STOP signal must be shown to the driver. Otherwise, it must allow the first car to enter and the GO signal must be shown. It can be seen here how a non-deterministic choice between showing STOP or GO in **Scenario 1** is now determined by **Scenario 2**. To specify the system further, more scenarios are added.

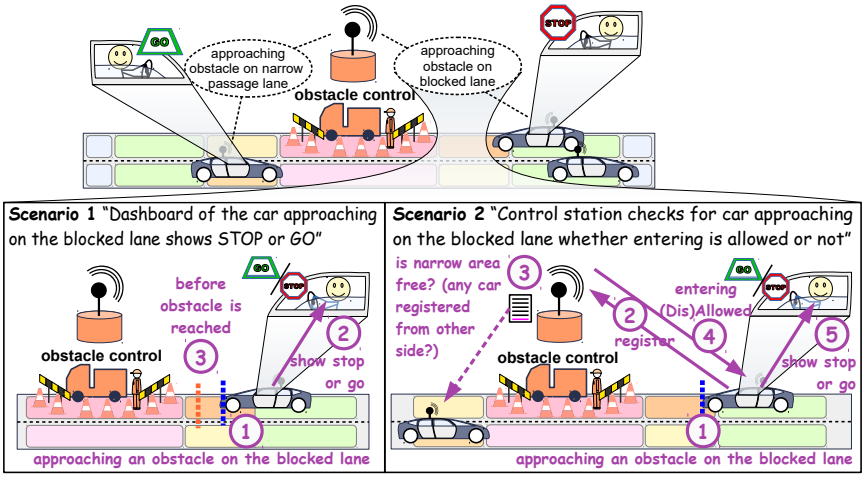

Fig. 1. Car-to-X example overview

Listing 1 shows how the two scenarios illustrated above are modeled with SML in ScenarioTools. The specification CarToX imports an ecore file that contains the class model of the system. Here, it defines classes for cars, lane areas, the obstacle control, etc., including their relationships. For simulation and controller synthesis, an *object model*, which is an instance of this class model, must be defined, with a particular number of cars and obstacles at certain positions.

The specification then defines which classes of objects are *controllable* and which ones are *uncontrollable*. Controllable classes are the components for which we specify the (software) behavior. In our case, this is the car and the control station for an obstacle that blocks one street lane. Uncontrollable classes model environment entities that are the source of environment events that the controllable components react to. In our case, the class Environment is an abstraction of the car's sensors. For example, the environment can send a car an event that it moved to the next lane area or that it approaches a certain obstacle. In the real system a camera- or GPS-based module may send these events.

```
import "car-to-x.ecore"

system specification CarToX {

  domain cartox // reference Ecore package

  define Environment as uncontrollable
  define Car as controllable
  define ObstacleBlockingOneLaneControl as controllable
  define Dashboard as uncontrollable

  collaboration ApproachingObstacleOnBlockedLane{

    dynamic role Environment env
    dynamic role Car car
    dynamic role Dashboard dashboard
    dynamic role ObstacleBlockingOneLaneControl obstacleControl

    // Scenario 1
    specification scenario DashboardOfCarApproachingOnBlockedLaneShowsStopOrGo
    with dynamic bindings [
      bind dashboard to car.dashboard
    ]{
      message env->car.setApproachingObstacle(*)
      alternative{
        message strict requested car->dashboard.showGo()
      } or {
        message strict requested car->dashboard.showStop()
      }
      message env->car.obstacleReached()
    }

    // Scenario 2
    specification scenario ControlStationAllowsCarOnBlockedLaneToEnterOrNot
    with dynamic bindings [
      bind obstacle to car.approachingObstacle
      bind obstacleControl to obstacle.controlledBy
      bind dashboard to car.dashboard
    ]{
      message env->car.setApproachingObstacle(*)
      message strict requested car->obstacleControl.register()
      alternative if [obstacleControl.carsRegisteredOnNarrowPassageLane.isEmpty()]{
        message strict requested obstacleControl->car.enteringAllowed()
        message strict car->dashboard.showGo()
      } or if [!obstacleControl.carsRegisteredOnNarrowPassageLane.isEmpty()]{
        message strict requested obstacleControl->car.enteringDisallowed()
        message strict car->dashboard.showStop()
      }
    }

    assumption scenario ApproachingObstacleOnBlockedLaneAssumption
    with dynamic bindings [
      bind currentArea to car.inArea
      bind nextArea to currentArea.next
      bind obstacle to nextArea.obstacle
    ]{
      message env->car.carMovesToNextArea()
      interrupt if [obstacle == null]
      message strict requested env->car.setApproachingObstacle(obstacle)
    } constraints [
      forbidden message env->car.carMovesToNextArea()
    ]
  } // ... additional collaborations and scenarios
}
```

Listing 1. Part of the Car-to-X SML specification

Next, a specification contains *collaborations*. A collaboration defines *roles* that represent collaborating objects in the system. Furthermore, a collaboration defines *scenarios* that describe requirements on how the controllable objects must or must not react to environment events (*specification scenarios*) or they describe what can or cannot happen in the environment (*assumption scenarios*). The two specification scenarios shown in Listing 1 model the scenarios in Fig. 1.

Each scenario describes a valid sequence of message events, where each message event is the sending of a message from one object to another. ScenarioTools supports alternative, parallel, and loop constructs within the scenarios. Furthermore, the messages in the scenarios can have the modalities strict and requested. In a nutshell, when a strict message is expected by a scenario, no message event must occur in the system that corresponds to a message in that scenario that is not currently expected. Such violations are called *safety violations*.

The modality requested indicates that the message must eventually occur. If a scenario never progresses at a requested message, this is a *liveness violation*. Hence, these modalities allow us to specify safety and liveness properties.

The scenarios also define how the roles used by its messages shall be *bound* to objects in the object model. The binding of the sending and receiving roles of the first message are given through the occurrence of the event that initiates the scenario. The binding of the other roles in the scenario is defined through *binding expressions* that refer to properties of objects bound to other roles. This way we can define a behavior that is sensitive to the current system structure.

Listing 1 also shows an assumption scenario that describes that after a car moved onto a new area, and the area after this now current area has a obstacle, the car will eventually receive the event that it is approaching that obstacle.

Reference or attribute values of objects can change as a side-effect of messages prefixed with set. For example, when setApproachingObstacle(obstacle) is received by a car, the car's value for the reference ApproachingObstacle is set to obstacle. This way we can specify in the second scenario where the car shall register.

Furthermore, messages can be associated with graph transformation (GT) rules. Figure 2 shows a rule from the Car-to-X example. A GT rule for a message must have at least two parameters that get bound to the sending and receiving object of the message event. A GT rule serves two purposes. First, it restricts that messages can only occur when their corresponding GT rule is applicable in the current object model (for details, see [1]). Second, it can describe a transformation that specifies the side-effect of that message.

The example GT rule in Fig. 2 expresses that on the occurrence of the event carMovesToNextArea, the receiving car's inArea link will change to express that the car moves to the next area relative to its current area. Moreover, the rule constraints that the event cannot occur, for example, when the next lane area is occupied an obstacle. Also, the car cannot advance to the next lane area if it is following a car that still resides on the same lane area. This way, GT rules associated with environment events also formulate assumptions on when these environment events can or cannot occur.

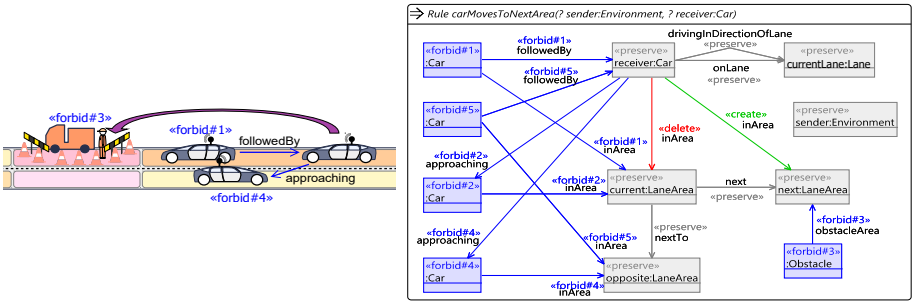

Fig. 2. A GT rule that describes when and how a car moves to the next lane area.

3 Simulation and Synthesis

ScenarioTools supports the execution and interactive simulation of the combined scenario- and GT rule specifications, based on an extended play-out algorithm [4,3]. The ScenarioTools simulation component is integrated into the Eclipse debug environment. After each step, the current state of progress of the different scenarios is highlighted in the SML editor. A graphical *state view* visualizes the explored states and supports jumping back and forth in the execution.

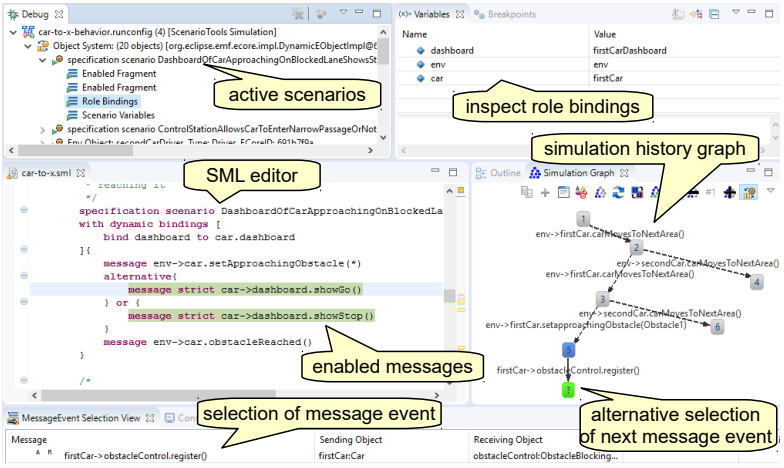

Fig. 3. The ScenarioTools simulation perspective.

The controller synthesis feature of ScenarioTools allows us to check whether the specification is realizable or not, i.e., whether an environment that satisfies the assumption scenarios can force the system into a safety or liveness violation of the specification scenarios. This works by creating an explicit state graph of all play-out executions, including the changing object structures, and running game-solving algorithms on this graph. ScenarioTools also supports visualizing strategies or counter-strategies that show how the system can or

cannot guarantee to satisfy the specification. Using synthesis for our Car-to-X example, we can, for instance, find out that the software cannot avoid crashes of cars unless we assume that drivers obey the dashboard signals.

4 Related Work

The two notions at the core of our technique—scenario-based modeling and systems with dynamic structure—have each been studied extensively. There exist many scenario-based modeling approaches based on MSCs, UML SDs, and LSCs. There also exist approaches combining scenarios with other behavior models, such as state machines or temporal logics. Likewise, there are many approaches for modeling systems with dynamic structures, especially graph transformations.

The modeling and analysis of LSCs is supported also by the PlayGo [5] tool. The ScenarioTools approach presented here, however, is unique in its combination of formal, executable scenario specifications with graph transformations to model the message-based interaction of components in a system, the evolution of the system structure, and the interrelation between these two aspects.

Another related approach is MechatronicUML [2], which combines state-based modeling and graph transformations for systems with dynamic structures. Compared to the state-based modeling of MechatronicUML, the scenario-based modeling of ScenarioTools targets an earlier design and specification phase.

Acknowledgment: We thank Timo Gutjahr, Florian König, and Nils Glade for their work on ScenarioTools.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. 13th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 121–135 (2010)
2. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schäfer, W., Meyer, M., Pohlmann, U., Priesterjahn, C., Tichy, M.: The MechatronicUML design method – process and language for platform-independent modeling (2014)
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. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
5. Harel, D., Maoz, S., Szekely, S., Barkan, D.: Playgo: Towards a comprehensive tool for scenario based programming. In: Proc Int. Conf. on Automated Software Engineering. pp. 359–360. ASE '10, ACM, New York, NY, USA (2010)
6. Henshin website. `https://www.eclipse.org/henshin/`
7. ScenarioTools website. `http://scenariotools.org`
8. Winetzhammer, S., Greenyer, J., Tichy, M.: Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems. In: System Analysis and Modeling: Models and Reusability, LNCS, vol. 8769, pp. 126–141. Springer (2014)