From Scenario Modeling to Scenario Programming for Reactive Systems with Dynamic Topology*

Joel Greenyer, Daniel Gritzner, Florian König, Jannik Dahlke, Jianwei Shi, Eric Wete
Leibniz Universität Hannover, Software Engineering Group
Welfengarten 1
D-30167 Hannover
greenyer@inf.uni-hannover.de
daniel.gritzner@inf.uni-hannover.de

ABSTRACT

Software-intensive systems often consist of cooperating reactive components. In mobile and reconfigurable systems, their topology changes at run-time, which influences how the components must cooperate. The Scenario Modeling Language (SML) offers a formal approach for specifying the reactive behavior such systems that aligns with how humans conceive and communicate behavioral requirements. Simulation and formal checks can find specification flaws early. We present a framework for the Scenario-based Programming (SBP) that reflects the concepts of SML in Java and makes the scenario modeling approach available for programming. SBP code can also be generated from SML and extended with platform-specific code, thus streamlining the transition from design to implementation. As an example serves a car-to-x communication system. Demo video and artifact: *http://scenariotools.org/esecfse-2017-tool-demo/*

CCS CONCEPTS

• **Software and its engineering → Specification languages**; **Development frameworks and environments**; **Requirements analysis**; **Formal software verification**; • **Computer systems organization →** *Embedded and cyber-physical systems*;

KEYWORDS

Scenario-based modeling, reactive systems, distributed embedded systems, dynamic topologies, assume/guarantee specifications

ACM Reference format:
Joel Greenyer, Daniel Gritzner, Florian König, Jannik Dahlke, Jianwei Shi, Eric Wete. 2017. From Scenario Modeling to Scenario Programming for Reactive Systems with Dynamic Topology. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122827

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

1 INTRODUCTION

Software-intensive systems in domains like transportation, production or logistics often consist of multiple components that react to external events and cooperate to fulfill the system goals. In mobile and reconfigurable systems, like car-to-x systems or reconfigurable production systems, the system topology changes at run-time, i.e., properties and relationships of the components change. This influences how components must cooperate.

The dynamic topology of the system and the distributed and concurrent nature of the software pose a challenge in the system's development. Formal specification and analysis can help engineers detect design flaws early, and reduce the risk of costly iterations. Scenario modeling methods based on the Scenario Modeling Language (SML) [7] or Live Sequence Charts (LSCs) [5, 9] are particularly well-suited for the early design, because they fit well with how humans conceive and communicate requirements. SML/LSC specification can be executed via the *play-out* algorithm [9, 15], which enables early validation by simulating the scenarios' interplay.

SML is a textual variant of LSCs [5, 9], and extends LSC with the means for specifying environment assumptions in the form of *assumption scenarios*. Assumption scenarios, together with *guarantee scenarios*, which specify the desired system behavior, form an *assume-guarantee specification*. The expressive power of SML is comparable with the GR(1) (Generalized Reactivity of Rank 1) fragment of LTL, which supports expressing many practically relevant properties while efficient controller synthesis and realizability checking algorithms exist for this class of specifications [4, 17].

The modeling and analysis of SML specifications is supported by ScenarioTools[1]. ScenarioTools implements a GR(1) game solving algorithm by Chatterjee et al. [4] for controller synthesis and realizability checking. The algorithm checks whether for any sequence of environment events, the system has a strategy for choosing system events in such a way that when all assumption scenarios are satisfied, so are all guarantee scenarios. The specification is *realizable* if such a strategy exists, and *unrealizable* otherwise.

Once the realizability of a specification is established, an implementation must be constructed that satisfies the specification, and possibly extends the specified behavior with further details, such as platform-specific functionality. For example, the specification of a car-to-x system may mention movement events of cars, like "approaching obstacle". However, how these events are interpreted from a car's GPS or other sensors is platform specific functionality.

[1]http://scenariotools.org

An implementation can be constructed manually, which, however, is error prone. We can also use controller synthesis and generate code from the synthesized strategy. However, controller synthesis cannot handle systems with large or potentially unbounded numbers of components, as for example in a city-wide, open car-to-x system. For such systems it only makes sense to check the specifications in the context of configurations of a few components, for example typical traffic situations in a car-to-x application.

Finally, we can execute the scenarios with the above-mentioned play-out algorithm. Such an approach, however, must allow developers to extend the specified behavior with platform-specific code, and it must support executing the scenarios in a distributed system.

Addressing these requirements, we developed a novel framework for the *Scenario-Based Programming* (SBP) in Java. It reflects the concepts of SML (and LSCs) in Java and allows programmers to program scenarios as special *scenario threads* in Java. The framework is based on the Behavioral Programming framework for Java (BPJ) [11], which we extended to serve as the play-out algorithm.

We also developed an SML-to-SBP compiler, so that SML specifications that were previously checked in ScenarioTools can be translated automatically into SBP. Guarantee scenario threads drive the execution, while assumption scenario threads monitor whether the environment in which the system is deployed satisfies the specified assumptions. SBP programs can be extended with platform-specific functions by adding further SBP modules or other Java code. We also developed a technique for the distributed execution of SBP programs. The distributed execution technique does not yet scale for larger systems, but we are working on improvements [19].

In this tool demo, we present the novel SBP framework by the example of a car-to-x driver assistance system that allows cars to safely pass obstacles that create a narrow passage on a road. We also demonstrate the design method sketched in Fig. 1: (1) Model and (2) analyze SML specification, including, first (a) realizability checking the specification and then (b) verifying that a play-out execution will satisfy the specification; (3) generate SBP code from the SML specification, and (4) extend it with platform-specific functionality.

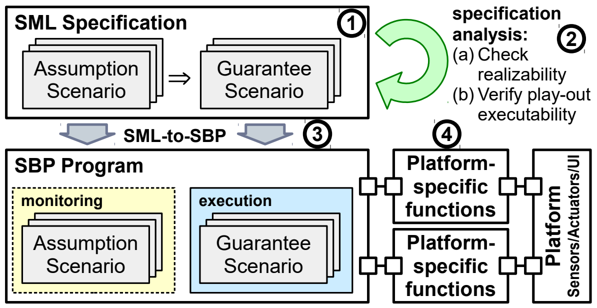

Figure 1: Scenario-based design to implementation method

We introduce the example and background in Sect. 2, present SBP in Sect. 3, discuss related work in Sect. 4, and conclude in Sect. 5.

2 EXAMPLE AND BACKGROUND

We consider a car-to-x (car-to-car and car-to-infrastructure) communication system that assists drivers in passing a narrow passage created by obstacles such as road works. Figure 2 shows a sketch. End points of the obstacle are defined as GPS coordinates. There are three perimeters around each end point. A car entering the outer perimeter *approaches the obstacle*; a car entering the next perimeter *reached the obstacle*; a car entering the innermost perimeter *entered the narrow passage.* Approaching cars must communicate with an *obstacle controller* to coordinate passing the obstacle safely.

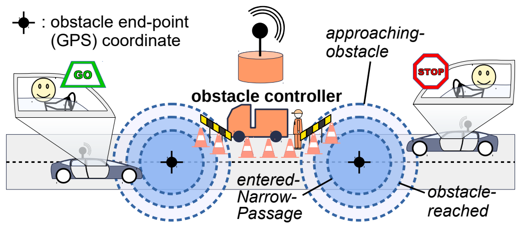

Figure 2: car-to-x narrow passage coordination sketch

Consider two guarantee scenarios for this system. **G1**: *When a car approaches the obstacle, the obstacle controller must allow or disallow the car to enter the narrow passage before the car reaches the obstacle.* **G2**: *When a car approaches the obstacle, it must register at the obstacle controller. Then the obstacle controller must check whether another car is already registered for passing the obstacle. If so, the obstacle controller must add the approaching car to a waiting list and disallow it from entering; otherwise, it must register the car for passage and allow it to enter.*

To specify the system further, more scenarios are added. **G1** and **G2** describe complementary requirements, while both mention allowing or disallowing a car to enter; a non-deterministic choice in **G1** between allowing or disallowing is refined in **G2**.

SML: Listing 1 shows how the two scenarios are modeled in SML. An SML specification models how objects in an *object model* shall interact by sending messages. We consider *synchronous* communication where the sending and receiving of a message is a single *message event.* A message event has one sending and one receiving object, refers to an operation defined for the receiving object, and carries values for parameters if any are defined by its operation. Message event may have *side-effects* on the object model. For example, message events referring to *set*-operations change the corresponding property of the receiving object.

An infinite sequence of message events and object models (that evolve through side-effects from an initial one) is called a *run.*

The object model is partitioned into *controllable* (*system*) objects and *uncontrollable* (*environment*) objects. A message event sent by a system object is a (*controllable*) *system event*; if sent by an environment object, it is an (*uncontrollable*) *environment event.*

An SML specification refers to a *domain* class model that describes a set of possible object models for which the specification models the behavior. For execution and analysis, a concrete (initial, possibly evolving) object model, for example one as shown in Fig. 2, must be provided in an external configuration (omitted for brevity).

In this example, the class model defines cars, the obstacle controller, and a coordinate processor. Cars and obstacle controllers are system objects. The coordinate processor is environment; it is an abstraction of a car's position sensors that, with knowledge of obstacle

```
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.obstacleReached()
  }

  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() ]
  ...
 }
 ...
}
```

Listing 1: Part of car-to-x SML specification

end points, can generate the events approachingObstacle, obstacleReached, and enterNarrowPassage. Such abstractions are typical during the early design.

Scenarios are organized in *collaborations*, which define *roles* that represent objects in an object model. The scenarios refer to roles in *messages*, which are used to specify valid orders of message events.

A scenario 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 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 copy progresses on the occurrence of further events that match enabled messages.

When a *strict* message is enabled, it means that message events are *forbidden* that corresponds to a message in the same scenario that is not currently enabled. If an *urgent* message is enabled, this means that a corresponding message must occur before the next environment event. SML supports other modalities, also for modeling unbounded liveness properties, but we omit them for brevity.

A scenario can have a *constraints* section with *forbidden* messages. They represent events that must not occur while the scenario is active. This way, the assumption scenario in List. 1 says that when the obstacle controller disallowed a car to enter the narrow passage, the car will not enter before the obstacle controller allows it.

A run *satisfies* an SML specification if it leads to no violations of any guarantee scenario or there is a violation in at least one assumption scenario. One general underlying assumption is that the system is fast enough to send any finite number of system events in response to an environment event.

The *play-out* algorithm [3, 9, 10] executes an SML specification by executing message events corresponding to enabled urgent system messages in active guarantee scenarios, as long as they are not forbidden by any other active guarantee scenario. If there are no such events, play-out waits for the next environment event.

ScenarioTools can build a graph of all possible play-out executions. It is the basis for realizability checking and verifying whether runs resulting from play-out will be valid, see step (2) in Fig. 1.

Realizability checking is based on a GR(1) game solving algorithm by Chatterjee et al. [4]. It may fail due to contradictions in guarantees or missing environment assumptions. For instance, our example specification cannot guarantee that no crashes will occur if the assumption shown in List. 1 is missing.

If the specification is realizable, we must then verify whether any play-out execution that naively selects system events as outlined above will satisfy the specification. This is done by checking whether for all play-out states the strategy generated during realizability checking identifies all choices of system steps as winning. If this is not the case, it means that the specification is under-specified and guarantee scenarios can be added to restrict the choices for play-out. If the verification is successful, SBP code can be generated.

3 SCENARIO-BASED PROGRAMMING

SBP [13] is based on the BPJ [11] framework[2]. A BPJ program consists of a collection of *BThreads* that collaborate by calling a *bSync()* method. All BThreads yield at bSync() calls, where they either *request*, *wait for*, or *block* events. Then an event that is requested by a BThread and not blocked by any other BThread is chosen, and BThreads that requested or waited for that event are notified of the event and resume execution until the next bSync() call. Then the process is repeated. SBP extends BPJ as follows:

Message events and object model: Events in BPJ can be any kind of objects. In SBP, events are message event objects that have a sending and receiving object, a method name, and a list of parameter values. The object model is stored as a structure of Java objects.

Scenarios threads: Scenario threads are special BThreads that call bSync() in such a way that it reflects the scenario semantics sketched above. Urgent system messages are mapped to requested events, and non-urgent messages are mapped to waited-for events. When a strict message is enabled, all events in the scenario that are not requested or waited-for are blocked.

Transformation threads: Side-effects on the object model, for example of set-messages, are implemented as special BThreads that wait for the events and perform the object system manipulation.

Event queue thread: This BThreads receives external events and requesting them for execution. *Platform-specific code* for generating external events for the SBP program for example from sensors can be added to place such events in the event queue.

Spectator threads: These BThreads wait for events that shall have a side-effect on the platform, i.e., signaling an actuator or showing information on the UI. Platform-specific code can be added as spectator threads to translate SBP events to lower-level functions.

[2]http://www.wisdom.weizmann.ac.il/~bprogram/bpj/

```java
public class CarRegistersAtObstacleScenario
  extends CarsPassObstacleCollaboration /*extends Scenario*/{

 @Override
 protected void initialisation() {
  addInitMessage(new Message(cp,car, "approachingObstacle"));
 }

 @Override
 protected void registerAlphabet() {
  setBlocked(cp, car, "approachingObstacle");
  setBlocked(car, oc, "register");
  setBlocked(oc, oc, "setPassingCar", car.getBinding());
  setBlocked(oc, car, "enteringAllowed");
  setBlocked(oc, oc, "waitingCars", car.getBinding());
  setBlocked(oc, car, "enteringDisallowed");
 }

 @Override
 protected void registerRoleBindings() {
  bindRoleToObject(oc, cp.getBinding().getObstacleCtrl());
 }

 @Override
 protected void body() throws Violation {
  request(STRICT, car, oc, "register");
  // Begin Alternative
  List<Message> requestedMessages = new ArrayList<Message>();
  List<Message> waitedForMessages = new ArrayList<Message>();
  if ((oc.getBinding().getPassingCar() == null)) {
   requestedMessages.add(new Message(
      STRICT, oc, oc, "SETPassingCar", car.getBinding()));
  }
  if ((oc.getBinding().getPassingCar() != null)) {
   requestedMessages.add(new Message(
      STRICT, oc, oc, "ADDWaitingCars", car.getBinding()));
  }
  doStep(requestedMessages, waitedForMessages);
  // Determine which alternative was chosen
  if (getLastMessage().equals(new Message(
     oc, oc, "SETPassingCar", car.getBinding()))) {
   request(STRICT, oc, car, "enteringAllowed");
  } else if (getLastMessage().equals(new Message(
     oc, oc, "ADDWaitingCars", car.getBinding()))) {
   request(STRICT, oc, car, "enteringDisallowed");
  }
  // End Alternative
 }
}
```

Listing 2: SBP code for CarRegistersAtObstacle scenario

Initializer thread: This is a special spectator thread that waits for initializing scenario events. On the occurrence of such an event, it creates a copy of the respective scenario thread and starts it.

Scenario threads can be generated automatically from SML scenarios. Listing 2 shows the scenario thread class generated from the second SML scenario in List. 1. All shown methods override methods from the Scenario class, the class `CarPassObstacle-Collaboration` is an intermediate class generated from the collaboration; it provides the definitions of the roles (`cp`, `car`, `oc`).

The method `initialization()` is called when the SBP program starts. The initializing message is used by the initializer thread to know on which event a copy of this scenario should be started.

When the scenario is started, the other methods are executed in the shown order. `registerAlphabet()` defines all messages that occur in the scenario. They are registered as blocked messages since they are blocked when a strict message is enabled. When a non-strict message is enabled, these messages are waited for.

The method `registerRoleBindings()` computes bindings for roles that are not bound based on the initializing event.

The `body()` method represents the scenario after the first message. In the first line we see a call to the `request()` method that corresponds to the urgent register message in the SML version. The remaining code represents the alternative.

Distributed execution: As the systems targeted by SML/SBP are distributed, it it desirable to deploy SBP programs in a distributed architecture, such as a system of cars. We implemented a proof-of-concept technique that relies on the replication of the SBP program for all components and the synchronization of all components upon each event in the system [8]. This helps to keep the states of the SBP programs consistent, but creates a communication overhead. We are currently working on reducing this overhead [19].

As proof-of-concept, we could successfully run the SBP program for the car-to-x application on Android phones. With platform-specific extensions, they act as a car's dashboard and position sensor, so we can exercise the example driving in real cars. The communication can take place over different kinds of networks. We currently use MQTT, which is widely used in IoT applications.

4 RELATED WORK

In previous work [6, 8], we presented a prototype for the distributed execution of SML specifications by executing them by the ScenarioTools runtime, which interprets the SML specification. This, however, requires many dependencies and a plug-in container, which made it difficult to deploy of some platforms, such as Android. The approach presented here is more light-weight and better extensible.

Harel and Maoz describe a mapping from LSCs to AspectJ (S2A) for play-out [14]. It is similar to SBP, but SBP scenario threads resemble the SML scenarios more closely than the AspectJ code. Thus, scenario programming in SBP is more natural. Also, there is no extensions of S2A to execute in a distributed architecture.

SBP is sceario-based and focuses on the inter-component communication. There are other approaches for developing reactive systems with dynamic topologies and adaptive behavior that instead focus on modeling the individual component behaviors. For example, Bagheri et al. [2] use actors for developing adaptive systems and systems with spatial dynamics, such as air traffic control systems. DEECO [1] is a framework for adaptive component systems. Also, MechatronicUML [18] and UML-RT or Papyrus-RT [12] can be used to model adaptive component systems.

Piechnick et al. present Smart Application Grids, a technique for exchanging context information in distributed and adaptive systems [16]. This work is related to our problem of maintaining consistent views on the object model in a distributed setting once we abandon the synchronization of all components on each event.

5 CONCLUSION

We presented a method for moving from scenario-based specifications to scenario-based implementations. The core novelty is SBP, a framework for the scenario-based programming in Java. It reflects the concepts of SML and LSCs and allows programmers to implement scenarios in a way that closely resembles their modeling in SML or LSCs. SBP programs can be generated automatically from previously checked SML specifications, they can be extended with platform-specific code, and we showed how they can be deployed on distributed architectures. This streamlines the transition from scenario-based specification to implementation. Future work consists in improving the distributed execution techniques.

REFERENCES

[1] Rima Al Ali, Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. 2014. DEECo: An Ecosystem for Cyber-physical Systems. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 610–611. https://doi.org/10.1145/2591062.2591140

[2] Maryam Bagheri, Ilge Akkaya, Ehsan Khamespanah, Narges Khakpour, Marjan Sirjani, Ali Movaghar, and Edward A. Lee. 2017. *Coordinated Actors for Reliable Self-adaptive Systems*. Springer International Publishing, Cham, 241–259. https://doi.org/10.1007/978-3-319-57666-4_15

[3] Christian Brenner, Joel Greenyer, and Valerio Panzica La Manna. 2013. 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). *Electronic Communications of the EASST* 58.

[4] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Veronika Loitzenbauer. 2016. Conditionally Optimal Algorithms for Generalized Büchi Games. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier (Eds.), Vol. 58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:15. https://doi.org/10.4230/LIPIcs.MFCS.2016.25

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

[6] 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. 2015. Scenarios@run.time – Distributed Execution of Specifications on IoT-Connected Robots. In *Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015 (to appear)*. http://jgreen.de/wp-content/documents/2015/scenarios-at-runtime.pdf

[7] Joel Greenyer, Daniel Gritzner, Guy Katz, and Assaf Marron. 2016. Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *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)*, Juan de Lara, Peter J. Clarke, and Mehrdad Sabetzadeh (Eds.), Vol. 1725. CEUR, 16–32.

[8] Joel Greenyer, Daniel Gritzner, Guy Katz, Assaf Marron, Nils Glade, Timo Gutjahr, and Florian König. 2016. 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)* 26 (2016), 552 – 559. https://doi.org/10.1016/j.protcy.2016.08.069 3rd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering (SysInt 2016).

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

[10] David Harel and Rami Marelly. 2003. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. *SoSyM* 2 (2003), 82–107.

[11] David Harel, Assaf Marron, and Gera Weiss. 2012. Behavioral programming. *Comm. ACM* 55, 7 (2012), 90–100. https://doi.org/10.1145/2209249.2209270

[12] Nafiseh Kahani, Nicolas Hili, James R. Cordy, and Juergen Dingel. 2017. Evaluation of UML-RT and Papyrus-RT for Modelling Self-adaptive Systems. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering (MISE '17)*. IEEE Press, Piscataway, NJ, USA, 12–18. https://doi.org/10.1109/MiSE.2017..4

[13] Florian Wolfgang Hagen König. 2017. *Szenariobasierte Programmierung und verteilte Ausführung in Java*. Master's Thesis. Leibniz Universität Hannover, Software Engineering Group. http://jgreen.de/wp-content/documents/msc-theses/2017/Koenig2017.pdf

[14] Shahar Maoz, David Harel, and Asaf Kleinbort. 2011. A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 18 (Sept. 2011), 41 pages. https://doi.org/10.1145/2000799.2000804

[15] Rami Marelly, David Harel, and Hillel Kugler. 2002. Multiple Instances and Symbolic Variables in Executable Sequence Charts. *SIGPLAN Not.* 37, 11 (2002), 83–100. https://doi.org/10.1145/583854.582429

[16] Christian Piechnick, Maria Piechnick, Sebastian Götz, Georg Püschel, and Uwe Aßmann. 2015. Managing Distributed Context Models Requires Adaptivity too. In *Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015*. http://ceur-ws.org/Vol-1474/MRT15_paper_6.pdf

[17] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of Reactive(1) Designs. In *Verification, Model Checking, and Abstract Interpretation*, E.Allen Emerson and KedarS. Namjoshi (Eds.). Lecture Notes in Computer Science, Vol. 3855. Springer Berlin Heidelberg, 364–380. https://doi.org/10.1007/11609773_24

[18] D. Schubert, C. Heinzemann, and C. Gerking. 2016. Towards Safe Execution of Reconfigurations in Cyber-Physical Systems. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. 33–38. https://doi.org/10.1109/CBSE.2016.10

[19] Shlomi Steinberg, Joel Greenyer, Daniel Gritzner, David Harel, Guy Katz, and Assaf Marron. 2017. Distributing Scenario-based Models: A Replicate-and-Project Approach. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*. INSTICC, ScitePress, 182–195. https://doi.org/10.5220/0006271301820195