Modeling and programming a leader-follower challenge problem with scenario-based tools

Joel Greenyer[1], Michael Bar-Sinai[2], Gera Weiss[2], Aviran Sadon[2], and Assaf Marron[3]

[1] Leibniz Universität Hannover, Hannover, Germany
[2] Ben-Gurion University of the Negev, Be'er Sheva, Israel
[3] Weizmann Institute of Science, Rehovot, Israel

Abstract. Scenario-based modeling (SBM) is an approach for creating executable models for reactive systems that are comprised of scenarios, each of which describing a separate aspect of the overall desired and undesired system behavior. SBM advantages include model intuitiveness, executability, amenability to formal verification, alignment with the requirements, and incrementality of the development. Here, we showcase two new SBM frameworks (which join the ones for LSC, SML, Java, C++, and others) by demonstrating the respective solutions to the MDETools18 Rover Follower challenge. One is IOSM-K, an LSC/SML-style inter-object scenario modeling framework for the Kotlin language, and the other is BPjs, a framework for behavioral programming for Javascript. We illustrate how the SBM advantages are manifested in developing such composite control logic, and discuss the differences, relative advantages of, and considerations in choosing, each of these SBM frameworks.

1 Introduction

Embedded systems become increasingly complex with inter-system connectivity (e.g., automotive, IoT), and the need to control complex processes in the physical world. This makes it inherently difficult to translate behavioral requirements into working software. Scenario-based modeling (SBM) helps engineers in this task. Specifically, with Live Sequence Charts (LSC) [5, 12] or the Scenario Modeling Language (SML) [8], individual behavioral concerns can be modeled as loosely-coupled *scenarios*. Each scenario describes how the system may, must, or must not react to external events. A scenario specification can be directly executed by running all scenarios in parallel and complying with all their constraints and demands, yielding a coherent system behavior that satisfies the requirements. The execution is useful for analysis and simulation; the scenarios can even serve as the system's final implementation.

Behavioral Programming (BP) [13] transfers the scenario modeling concepts of LSC/SML into a general modeling and programming paradigm, of which LSC and SML are only two manifestations, with additional frameworks developed for Java, C++, Erlang, and other languages. LSC and SML follow an *Inter-Object Scenario Modeling* (IOSM) approach, which assumes that the system consists of communicating objects, and that the events are messages sent from one object to another. This setting especially fits distributed applications. BP generalizes this concept: events can, but are not required to, be bound to behaving objects.

We present two different solutions[4] for the MDETools'18 leader-follower challenge. Based on sensor input of two rovers' coordinates, and the orientation of the follower rover, speeds for the wheels of the follower rover are set such that it maintains a certain distance from the leader.

[4] source code and executable JARs can be found at `scenariotools.org/mdetools2018`.

The first solution is based on the novel Inter-Object Scenario Modeling for Kotlin (IOSM-K) framework, which offers a Kotlin-based programming framework and internal DSL for LSC/SML-style scenarios. IOSM-K is based on a Behavioral Programming for Kotlin (BPK) framework (BPK and IOSM-K were developed by the first-listed author), which exploits Kotlin's coroutines for executing concurrent scenarios. Kotlin is a JVM-based language, but it can also be compiled into native machine code, which is interesting for embedded systems.

The second solution is based on the Behavioral Programming for JavaScript (BPjs) framework that uses the Mozilla Rhino interpreter to execute Javascript code of the b-threads [2]. This yields a rich and intuitive language for modeling and executing the scenarios, while maintaining full access to the underlying state-space, which enables features such as run-time lookahead and verification through back-tracking as explained in Sect. 5.

The advantages of SBM and of both solutions are, first, that they allow engineers to translate behavioral concerns from the requirements directly into operational software with straightforward tracing of bugs to specification inconsistencies. Second, SBM allows for incremental extensions and refinements, supporting a trial-and-error-style development that is often essential in embedded software development. Third, the code-based models can be formally analyzed, e.g., by model checking.

In Sect. 2, we explain the foundations of IOSM and BP; in Sect. 3 and 4 we describe the IOSM-K and BPjs solutions; in Sect. 5 we elaborate on the amenability of SBM to formal verification and synthesis; in Sect. 6 we discuss common and distinguishing features of the solutions, discuss related work, and conclude.

2 IOSM and BP Foundations

2.1 Inter-Object Scenario Modeling with the LSC and SML Languages

The concepts of SBM were introduced with the language of Live Sequence Charts (LSC[5]). An LSC specification describes how *objects* in an *object system* shall interact by sending *messages*. Each LSC chart is a form of sequence diagram where *lifelines* represent objects, and arrows represent events of messages being sent between objects. Fig. 1 shows an example of an LSC specification for the challenge problem. LSC lifelines represent system or environment objects, with the latter (the telemetry sensors and the follower rover actuators in Fig. 1) marked with cloud-shaped headers.

By default, LSC charts specify properties to be satisfied by all runs of the system (LSCs can also describe example runs, test cases, etc.). LSC messages can be *hot* (red) or *cold* (blue) and *monitored* (dashed) or *executed* (i.e., 'to be executed') (solid). Fig. 1 shows annotations `(c,m)/(h,m)/(h,e)` for clarity. Cold messages are events that *may* occur, and hot ones *must* occur. Moreover, when a scenario has progressed to a hot message, all events that appear elsewhere in that scenario are *blocked*, enforcing strict ordering of events. When the scenario is at a cold message, it is allowed to have another message occurring out of order, but it leads to an immediate termination of the scenario.

For execution, an execution engine runs the scenarios in parallel, selects one of the messages whose execution is requested by some scenario and is not blocked by any scenario. It then triggers this event and advances all scenarios in which this message is presently enabled

[5] The acronym LSC is used also to refer an individual live sequence chart; plural: LSCs.

(as executed or monitored). When there are no enabled executed messages, the system waits for an environment event. Messages from environment objects can only be monitored and they are triggered externally.

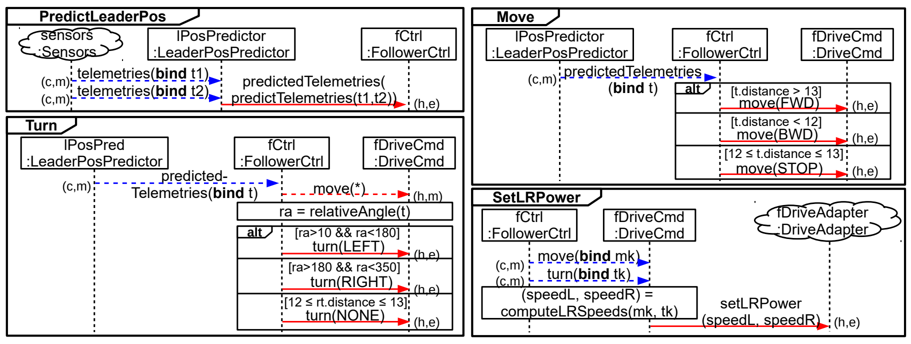

Fig. 1: Rover Follower Live Sequence Charts Specification

The charts in Fig. 1 model the following behavior: Rover telemetries (leader and follower coordinates, their distance, and the follower's orientation) are sent regularly by the sensors. The scenario PredictLeaderPos says that whenever two telemetries are received, the leader position predictor must calculate where the rovers are expected to be a few steps into the future, and send this to the follower control. We omit the details of predictTelemetries. Note that on every telemetries message a new instance of the scenario is activated, so that calculations are based on telemetries 1 and 2, 2 and 3, 3 and 4, etc.

A predicted telemetries message triggers the Move and Turn scenarios. Based on the predicted distance, the Move scenario determines whether the rover should move forward, backward, or stop. The Turn scenario calculates the angle between the follower's heading and the leader's bearing, and the rover must accordingly turn left, right, or not turn. Before requesting the turn commands, the Turn scenario waits for the move commands, so move commands occur before the turn commands. Finally, the chart SetLRPower calculates the speed setting for the left and right wheels of the follower rover based on the move and turn commands.

Listing 1.1 shows the Move scenario in SML textual syntax. Executed hot messages in LSC are modeled as *urgent* and *strict* messages in SML. Strict messages enforce strict event ordering and urgent ones must occur before the next environment message.

2.2 Behavioral Programming

Behavioral programming (BP) generalizes the scenario-based modeling in LSC or SML. While in LSC/SML scenarios are centered around objects that send and receive messages, events in BP can stand on their own. In BP frameworks for programming languages the scenarios, also called *behavioral threads* or *b-threads*, are modeled using standard programming constructs rather than lifelines and messages. This allows for a natural embedding of scenario models in code.

As in LSC, scenario coordination is done by an execution mechanism: it views each b-thread as a (possibly infinite) sequence of synchronization points marked by each b-thread's

```
guarantee scenario Move{
  var t : Telemetries
  lPosPred -> fCtrl.predictedTelemetries(bind t)
  alternative if [t.distance > 13]{
    urgent strict fCtrl -> fDriveCmd.move(MoveKind.FWD)
  } or if [t.distance < 12] {
    urgent strict fCtrl -> fDriveCmd.move(MoveKind.BWD)
  } or if [t.distance <= 13 && t.distance >= 12] {
    urgent strict fCtrl -> fDriveCmd.move(MoveKind.STOP)
  }
}
```

Listing 1.1: The Scenario Modeling Language version of the Move scenario

invocation of the method `bp.sync` to which the b-thread passes events that it *requests*, events that it *blocks* and events that it *waits for*. The execution mechanism runs all b-threads up to their first `bp.sync`. Then, it chooses an event that some b-thread requested and no b-thread blocked and resumes all the b-threads that requested or waited-for the triggered event. The b-threads re-synchronize and the process repeats. Again, when there is no event to trigger, the system waits for external events.

We adopt the assumption of *Logical Execution Time* [15] in which the execution rate of internal events is much higher than that of external events, so that a burst of internal events between two external ones (*super-step*) takes zero time relative to the external dynamics. For example, the code in Listing 1.3 repeatedly waits for `Telemetry` and reacts with a movement event. As common, e.g., with interrupt-handlers, we assume that the selection of the event is done in a negligible time.

3 Inter-Object Scenario Modeling for Kotlin (IOSM-K) solution

Listing 1.2 shows the IOSM-K code for the scenarios explained in Sect. 2.1 (cf. Fig. 1). Scenarios inherit from the abstract class `Scenario` provided by the IOSM-K framework. Scenarios are activated when a message event occurs that matches an initializing (first waited-for) message event as can be specified by overriding the `initEvents()` function. Once activated, the main logic of the scenario (`mainScenario`) is executed, where message events are requested or waited for via the functions `request` and `waitFor`. These functions take as arguments the sending and receiving objects, the message type, and parameters of the requested resp. waited-for message events. The `request` and `waitFor` functions map to `bsync` calls in the underlying BPK framework. In a static system with a fixed number of objects, the objects can be specified as singleton objects (declaration omitted for brevity).

Messages in LSC/SML that contain wildcard parameter placeholders (‘$\star$’) are modeled in IOSM-K as *symbolic message events*, where `ANY` stands for the wildcard. See, e.g., the move message in the Turn scenario and line 7 of Listing 1.2. Also, where in LSC/SML we have a message that binds a scenario variable (e.g., $t1$, $t2$ in PredictLeaderPos, see Fig. 1), this is a symbolic message in IOSM-K, and the variable values can be assigned by accessing the parameters of the last event that progressed the scenario, via by the variable `lastME` (the ‘`!!`’ is required by Kotlin when accessing a nullable variable without a check for null).

The IOSM-K scenarios very closely represent the LSC/SML scenarios shown above. Only the blocking and interrupting semantics implied by hot/strict resp. cold/non-strict messages must be modeled explicitly. The Turn scenario shows how a set `blockedEvents` can be filled with events that shall be blocked between specific request/waitFor synchronization points. For interrupting events, the set `interruptingEvents` can be used accordingly. This explicit definition may even help engineers and developers to see and understand details that may be difficult to follow in larger LSC/SML scenarios.

```kotlin
class PredictLeaderPos() : Scenario("PredictLeaderPos", ScenarioKind.GUARANTEE) {

  override fun initEvents() = SymbolicMessageEvent(Sensors,PosPredictor, PosPredictor::
       telemetries, ANY)

  override suspend fun mainScenario() {
    val t1 = lastME!!.parameters[0] as Telemetries
    waitFor(Leader,PosPredictor, PosPredictor::telemetries, ANY)
    val t2 = lastME!!.parameters[0] as Telemetries
    request(PosPredictor, FollowerCtrl, FollowerCtrl::predictedTelemetries,
         predictTelemetries(t1,t2))
  }

  private fun predictTelemetries(t1 : Telemetries, t2 : Telemetries) : Telemetries{...}
}

class Move() : Scenario("Move", ScenarioKind.GUARANTEE) {

  override fun initEvents() = SymbolicMessageEvent(PosPredictor,FollowerCtrl,FollowerCtrl::
       predictedTelemetries, ANY)

  override suspend fun mainScenario() {
    val t = lastME!!.parameters[0] as Telemetries
    if (t.distance > 13) request(FollowerCtrl,DriveCmd, DriveCmd::move, MoveKind.FWD)
    else if (t.distance < 12) request(FollowerCtrl,DriveCmd, DriveCmd::move, MoveKind.BWD)
    else request(FollowerCtrl,DriveCmd, DriveCmd::move, MoveKind.STOP)
  }
}

class Turn() : Scenario("Turn", ScenarioKind.GUARANTEE) {

  override fun initEvents() = SymbolicMessageEvent(PosPredictor,FollowerCtrl,FollowerCtrl::
       predictedTelemetries, ANY)

  override suspend fun mainScenario() {
    val t = lastME!!.parameters[0] as Telemetries
    val relativeAngle = relativeAngle(t)
    blockedEvents.add(SymbolicMessageEvent(FollowerCtrl, DriveCmd, DriveCmd::turn, ANY))
    waitFor(FollowerCtrl, DriveCmd, DriveCmd::move, ANY)
    blockedEvents.clear()
    if (relativeAngle > 10 && relativeAngle < 180 ) // turn left
      request(FollowerCtrl,DriveCmd,DriveCmd::turn, TurnKind.LEFT)
    else if (relativeAngle > 180 && relativeAngle < 350 ) // turn right
      request(FollowerCtrl,DriveCmd,DriveCmd::turn, TurnKind.RIGHT)
    else
      request(FollowerCtrl,DriveCmd,DriveCmd::turn, TurnKind.NONE)
  }

  private fun relativeAngle(t: Telemetries){...}
}

class SetLRPower() : Scenario("SetLRPower", ScenarioKind.GUARANTEE) {

  override fun initEvents() = SymbolicMessageEvent(FollowerCtrl,DriveCmd,DriveCmd::move,ANY)

  override suspend fun mainScenario() {
    var mk = lastME!!.parameters[0] as MoveKind
    waitFor(FollowerCtrl, DriveCmd, DriveCmd::turn, ANY)
    var tk = lastME!!.parameters[0] as TurnKind
    val (speedLeft, speedRight) = computeLRSpeeds(mk, tk)
    request(DriveCmd, DriveAdapter, DriveAdapter::setLRPower, speedLeft, speedRight)
  }

  private fun computeLRSpeeds(mk : MoveKind, tk : TurnKind) : Pair<Int, Int>{...}
}
```

Listing 1.2: IOSM-K code for the Rover Follower.

The solution also contains two scenarios (omitted for brevity) that interact with the ‘real’ rover’s sensors and actuators, i.e., the respective network interfaces of its simulator: one regularly retrieves the rovers’ telemetries, the other assigns the wheel speeds.

IOSM-K is similar to a BPJ-based IOSM framework that we presented earlier as part of a Scenario-Based Development Process (SBDP) for *dynamic topology* systems [7, 9]. In SBDP, the modeling starts with SML, for which ScenarioTools [8] supports formal analysis. Then code is generated that can even be executed in a distributed setting. With IOSM-K, we created an alternative compilation target; supporting distributed execution for IOSM-K is ongoing work.

The inter-object modeling concept of IOSM-K is especially suited for systems where collaborating objects can be identified *in the problem domain*. For this challenge problem, it was merely a design choice to introduce the components ‘leader position predictor’, ‘follower control’, ‘drive commands’. In a components-based development approach, we would define such components in order to encapsulate the different parts of the control logic. However, in scenario-based modeling, the behavior is separated in different scenarios—and it is not mandatory to define components, and tie events to them, in order to have a good separation of concerns, as our second solution shows.

4 The Behavioral Programming for Javascript (BPjs) solution

The second solution is based on the BPjs tool described in [2]. The model consists of a set of b-threads, each corresponding to a specific case-study requirement. Specifically, we focus on the two requirements that (1) *the rover needs to follow the leader*, and (2) *the rover should always stay at a safe distance from the leader*. The model listing 1.3) illustrates how blocking can be used to integrate competing requirements that are isolated in separate b-threads. Our actual development steps also showed how, as stated above, BP can facilitate incremental development of control software where trial-and-error is often a preferred way for defining the desired behavior. We designed the b-threads such that each b-thread represents a separate requirement.

```
importPackage(Packages.il.ac.bgu.cs.bp.leaderfollower.events);

var AnyTelemetry = bp.EventSet("Telemetries", function (e) {
  return e instanceof Telemetry;
});

var esFBwardEvents = bp.EventSet("FBwardEvents", function (e) {...});

bp.registerBThread("Go", function () {
  while (true) {
    bp.sync({waitFor: AnyTelemetry});
    bp.sync({request: StaticEvents.GO_TO_TARGET});
}});

bp.registerBThread("SpinToTarget", function () {
  while (true) {
    var et = bp.sync({waitFor: AnyTelemetry});
    var degToTarget = compDegToTarget(et.LeadX, et.LeadY, et.RovX, et.RovY, et.Compass);
    while (Math.abs(degToTarget) > 4) { // must correct rover orientation
      if (degToTarget > 0) {
        bp.sync({request: StaticEvents.TURN_RIGHT, block: esFBwardEvents});
      } else {
        bp.sync({request: StaticEvents.TURN_LEFT, block: esFBwardEvents});
      }
      et = bp.sync({waitFor: AnyTelemetry, block: esFBwardEvents});
      degToTarget = compDegToTarget(et.LeadX, et.LeadY, et.RovX, et.RovY, et.Compass);
}}});

var tooClose = 12.5;
var tooFar = 15;
```

```
31
32 bp.registerBThread("NotTooClose", function () {
33   while (true) {
34     var lastTelemetry = bp.sync({waitFor: AnyTelemetry});
35     while (lastTelemetry.Dist < tooFar) {
36       if (lastTelemetry.Dist >= tooClose-(tooFar-tooClose)) {
37         var slowDownPower=Math.round((lastTelemetry.Dist-tooClose)/(tooFar-tooClose)*100);
38         bp.sync({waitFor: [StaticEvents.TURN_RIGHT,StaticEvents.TURN_LEFT],request:
                GoSlowGradient(slowDownPower), block: StaticEvents.GO_TO_TARGET});
39       } else {
40         bp.sync({waitFor: [StaticEvents.TURN_RIGHT,StaticEvents.TURN_LEFT],request:
                GoSlowGradient(-100), block: StaticEvents.GO_TO_TARGET});
41       }
42       lastTelemetry = bp.sync({waitFor: AnyTelemetry, block: StaticEvents.GO_TO_TARGET});
43 }}});
44
45 function compDegToTarget(xL, yL, xR, yR, CompassDeg) {...}
```

Listing 1.3: Follower Rover Control program in BPjs (abbreviated for brevity)

The first line imports a Java package that abstracts from the interaction between the scenario-based model and the (simulated) rover. Specifically, this Java application sends `Telemetry` events that carry the rovers' telemetry data. Additionally, it translates actuation events into commands sent to the rover wheels.

Line 3 defines an event set that is used to wait for all events of type `Telemetry`. Since telemetry events differ in their data, a b-thread cannot list all of them individually. Instead it can specify to wait for a set of events that satisfy a given predicate. This technique is also used also in lines 7 for waiting for all forward/backwards movement events.

Lines 9 to 13 are a b-thread modeling the requirement that the rover should drive forward to follow the leader. It is a two-step scenario that waits for a telemetry event, and reacts with requesting the `GO_TO_TARGET` event.

Lines 15 to 27 are a b-thread for the requirement of the follower to aim at the leader before driving towards it. The scenario uses a helper function `compDegToTarget` that computes the angle in degrees between the follower and the leader. If this angle is outside of a defined range, the b-thread requests the event of turning right or left while blocking all forward/backwards movement events. The abstraction layer changes wheel speeds to actuate the turn, and the orientation is checked again. When the angle is within the allowed range, i.e., the follower is aimed at the leader, forward movement is no longer blocked. Note that the b-thread is modeled to be self-contained as much as possible. By contrast, say, in a classical a rule-based approach (which SBM is reminiscent of) coding these two movements as separate rules, may appear similar to the BP solution. But for implementing event blocking and compositional behavior, one would have to maintain shared variables, helper-events and additional application logic.

In our development, we started by running the above two b-threads to see how they perform. After a short debug process, we realized that these were not enough: indeed the rover followed the leader, but sometimes it got too close to it, violating a requirement that has not yet been implemented. We could, of course, adjust the existing b-threads such that the `GO_TO_TARGET` event is not requested when the distance to the leader is below the threshold, which is how such issues are usually dealt with in standard modeling and programming languages. However, as this is a separate behavioral concern, using SBM, we decided to implement this requirement in a separate scenario. Thus, lines 32 to 43 are an 'anti-scenario' b-thread that uses the BP event blocking idiom to forbid the `GO_TO_TARGET` event (which means going at maximum speed) if the distance is too small. Instead we use feedback-like control logic that requests the event

`GoSlowGradient(power)` which adjusts wheels speeds proportionally to the distance. `power` can be positive or negative, so the follower may drive in reverse if needed.

Note the `waitFor: [TURN_RIGHT, TURN_LEFT]` part in the third b-thread. This is a general design pattern we term "break-upon" [1], where a pending request of some b-thread becomes obsolete when certain events are triggered. In such a case, it is the responsibility of the requesting b-thread to drop its request to avoid undesired behavior. See Sect. 5 for a discussion of how such bugs can be detected automatically.

While small and simple, this model already demonstrates how blocking can be used to integrate positive and negative specifications when new requirements emerge during the development process. As demonstrated, this is especially useful for command and control software where typical trial-and-error creates new requirements that are better implemented in separate b-threads for maintenance and readability purposes.

5 Formal Analysis

A key advantage of SBM is the amenability of the model to formal analysis that is focused on conflicts and omissions among the requirements and among the key design decisions from very early stages in prototype development.

In the BPjs solution, formal explicit-state model checking can be done via exhaustive execution with back tracking. To this end, BPjs can store the state of each b-thread at each synchronization point. Using this built-in model checker and simple rover-simulation b-threads, we detected, for example, that if the follower's speed cannot exceed 120% of the leader's speed, it is possible that the distance between the vehicles will grow above the threshold. We also detected the lack of the "break upon" part discussed in Sect. 4 by asserting that all actuation events should be consistent with the last telemetry data. The result was an automatically generated trace of the system where a `GoSlowGradient` appears at a wrong time, a problem whose solution is described in Sect. 4 above. See the online repository for more details.

For the IOSM-K language, no such built-in model checker exists. However, IOSM-K can be generated from SML, for which realizability-checking and controller synthesis algorithms are offered by ScenarioTools. These algorithms check whether a strategy exists to correctly execute the scenarios for all admissible environment behaviors [8, 14]).

Other verification techniques for SBM include standard model checking of the composite set of state machines as abstracted from the scenarios' code (see, e.g. [11]), and using SMT solvers like Z3 or CVC4 to prove correctness of system behaviors based on assumptions and guarantees of the individual scenarios (see, e.g., [10]).

6 Discussion and Conclusion

Advantages and commonalities of both solutions: The above demonstrations of SBM in Kotlin and in Javascript first show the common SBM advantages: intuitiveness of scenarios (despite presence of few technical programming constructs), alignment of scenarios with requirements and design documents, weaving-in of new scenarios with little or no change to existing ones, formal verification capabilities, and, most importantly, direct executability of the model, without transformation or synthesis.

Differences in the solutions: There are also differences in the solutions, which, in fact, were developed by separate teams. The conceptual key differences include:

Inter-object vs. standalone events: The IOSM-K solution is based on inter-object message events and the BPjs one uses events not tied to objects. This raises the question of when to choose which style of modeling. Our answer is that inter-object events are suited where interacting objects are identifiable in the problem domain, e.g., when modeling a system of collaborating cars, robots, etc. However, decomposing the system into sub-components, as in a classical component-oriented approach, is not required to have a good separation of concerns in the scenarios. Indeed, with both frameworks, IOSM-K and BPjs, it would be possible to mix both event styles and use either where it fits best.

Scenario-composition styles: The second difference in the two solutions is the style in which the scenarios compose. The first solution (IOSM-K) reflects a rather sequential composition of the control logic: first compute the predicted rover coordinates, then decide on move and turn options, then set the wheel speeds. The scenarios trigger each other in a sequential fashion, with the exception that the `Turn` scenario synchronizes with the `Move` scenario for ensuring a particular order in the move and turn events. In the BPjs solution, the `SpinToTarget` and `NotTooClose` scenarios may block the `GO_TO_TARGET` event requested in the `Go` scenario. Here, the `Go` scenario is a default behavior that can be blocked if there are overriding concerns. We do not prefer one style of scenario composition over the other—which one to choose depends on how the behavioral concerns are conceived in the requirements.

Related Work. The main property of SBM distinguishing it from standard programming, executable modeling in, say, fUML [18] and Alf [17], rule-based or publish-subscribe systems [6], or highly intuitive visual languages like Statecharts or UML-RT [20], is its compositional semantics. Specifically, in SBM, independently specified threads of behavior that run in parallel can be *automatically* composed by the infrastructure with support for, e.g., forbidden behavior, strict event ordering, or satisfying multiple concurrent requests with a single event triggering. This also enables alignment of the software structure with the original requirements. The same advantage holds also in comparing to systems geared towards parallel synchronized executions and optional voting like SystemC [19], Esterel [3], or multi-agent systems. SBM is distinguished from the separation of concerns in Aspect Oriented Programming (AOP) [16] and similar techniques in that there is no distinction of base and modifications or exceptions thereto, and that the anchors of specifications are usually meaningful system events rather than particular pre-existing method names. Scenarios also manage states conveniently; state management in AOP join points requires non-trivial programming. The BIP (behavior, interaction, priority) [4] language has similar goals and terminologies, however, it focuses on developing a system that is correct-by-construction while SBM concentrates on programming in a natural way, and turns to techniques like model-checking to discover design issues. We refer to previous publications introducing SBM, IOSM, and BP concepts like [5, 12, 13], for a more detailed analysis of related work.

Conclusion: We demonstrated the scenario-based modeling approach by two different solutions to the follower rover challenge problem. This highlights the general principles of SBM and their power and applicability regardless of the underlying technology. Moreover, it shows that SBM principles enable language designers and tool builders to create powerful modeling environments that meet specific engineering needs, thus contributing to the community's ongoing efforts to accelerate and simplify the development of high-quality systems.

References

1. A. Ashrov, M. Gordon, A. Marron, A. Sturm, and G. Weiss. Structured behavioral programming idioms. In *EMMSAD 2017*.
2. M. Bar-Sinai, G. Weiss, and R. Shmuel. BPjs – a framework for modeling reactive systems using a scripting language and BP, 2018. `https://arxiv.org/abs/1806.00842`.
3. G. Berry. The foundations of Esterel. In *Proof, Language, and Interaction*, 2000.
4. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. *CONCUR*, 2008.
5. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80, 2001.
6. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.
7. J. Greenyer, L. Chazette, D. Gritzner, and E. Wete. A scenario-based MDE process for dynamic topology collaborative reactive systems – early virtual prototyping of car-to-x system specifications. In *Proc. Workshops zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES)*, volume 2060. CEUR, 2018.
8. J. Greenyer, D. Gritzner, T. Gutjahr, F. Konig, N. Glade, A. Marron, and G. Katz. ScenarioTools - a tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Comp. Prog.*, 149(Supplement C):15 – 27, 2017. Spec. Issue on MODELS'16.
9. J. Greenyer, D. Gritzner, J. Shi, and E. Wete. A scenario-based MDE process for developing reactive systems: A cleaning robot example. In *MODELS'17 Satellite Events*. CEUR, 2017.
10. D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On composing and proving correctness of reactive behavior. *EMSOFT*, 2013.
11. D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of safety and liveness violations in reactive programs. *Tran. on Comp. Collective Intelligence (TCCI) XVI*, 2014.
12. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
13. D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Comm. of the ACM*, 55(7).
14. D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Jour. Computer System Sciences*, 78:3:970–980, 2012.
15. T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *Control Systems Magazine, IEEE*, 2003.
16. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Euro. Conf. on Object-Oriented Prog. (ECOOP)*, 1997.
17. OMG. Action Language for Foundational UML (Alf), V. 1.1, July 2017. OMG document `formal/17-07-04`.
18. OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.3, July 2017. OMG document `formal/17-07-02`.
19. OSCI. Open SystemC Initiative. IEEE 1666 Lang. Ref. Manual. http://www.systemc.org.
20. E. Posse and J. Dingel. An executable formal semantics for uml-rt. *Software & Systems Modeling*, 15(1):179–217, Feb 2016.