A Scenario-based MDE Process for Developing Reactive Systems: A Cleaning Robot Example

Joel Greenyer, Daniel Gritzner, Jianwei Shi, Eric Wete
Software Engineering Group,
Leibniz Universität Hannover, Hannover, Germany
greenyer@inf.uni-hannover.de, daniel.gritzner@inf.uni-hannover.de

***Abstract*—This paper presents the SCENARIOTOOLS solution for developing a cleaning robot system, an instance of the rover problem of the MDE Tools Challenge 2017. We present an MDE process that consists of (1) the modeling of the system behavior as a scenario-based assume-guarantee specification with SML (Scenario Modeling Language), (2) the formal realizability-checking and verification of the specification, (3) the generation of SBP (Scenario-Based Programming) Java code from the SML specification, and, finally, (4) adding platform-specific code to connect specification-level events with platform-level sensor- and actuator-events. The resulting code can be executed on a RaspberryPi-based robot. The approach is suited for developing reactive systems with multiple cooperating components. Its strength is that the scenario-based modeling corresponds closely to how humans conceive and communicate behavioral requirements. SML in particular supports the modeling of environment assumptions and dynamic component structures. The formal checks ensure that the system satisfies its specification.**

I. INTRODUCTION

Software-intensive systems often consist of multiple cooperating reactive components. This is also the case for the challenge problems of the MDE Tools Challenge 2017: the 'Rover' and the 'Intelligent House'. Such systems provide increasingly rich functionality in order to meet growing customer needs. Therefore, developing correct software for these systems is a difficult challenge, especially due to the distributed and concurrent nature of the systems' software.

Model-Driven Engineering tools support engineers by offering problem-specific and platform-independent modeling languages, along with a systematic and automated process of deriving platform-specific, executable code. If the modeling languages have a precise semantics, formal analysis can help analyze the system design and identify problems early and on a problem-specific level.

SCENARIOTOOLS offers such an MDE approach for the development of distributed reactive systems. It supports the *scenario-based* modeling via the Scenario Modeling Language (SML), a textual variant of Live Sequence Charts (LSCs) [1], [2]. SML and LSCs model the behavior of a system as a set of separate *scenarios* that each describe how the system components may, must, or must not react to external events. The resulting specifications are executable via the *play-out* algorithm [2], which enables early validation by simulation, and even executing the scenarios as the actual implementation of the system.

SML in particular supports the modeling of environment assumptions in the form of *assumption scenarios*. They describe what will or will not happen in the environment or how the environment, in turn, will react to system actions. Modeling such assumptions is essential for systems that control processes in the physical world, such as robot control software. Also, SML supports the modeling of systems with *dynamic topologies*, which are systems where the component properties and relationships change at runtime and influence on how the components interact. SML supports dynamic role bindings and intergration with graph transformations for this purpose [3] (the latter will not be the focus here).

SCENARIOTOOLS supports an MDE process called SBDP (*Scenario-Based Development Process*). It consists of the following steps, also see Fig. 1:

1) *Modeling the system structure and behavior:* The structure, i.e., the systen and environment components, their properties and relationships, is modeled using a class diagram; the behavior is specified by an SML assume/guarantee specification that refers to that system structure model.
2) *Realizability-checking and verifying the specification:* SCENARIOTOOLS implements a *formal synthesis* algorithm that checks whether a strategy exists for the system to react to any sequence of environment events in such a way that the specification will be satisfied. If this is not the case, the specification is *unrealizable* and the algorithm produces a *counter-strategy* (similar to a counter-example in model checking) that helps the engineer understand the specification flaw. The algorithm can also *verify* whether implementation code, as created in the next step, will satisfy the specification.
3) *Code generation:* SCENARIOTOOLS provides a Scenario-Based Programming (SBP) framework that reflects the SML concepts in Java; for example. scenarios are programmed by extending special scenario classes. Their execution results in a play-out of the scenarios. This also works in a distributed setting, where components communicate over a network. SCENARIOTOOLS can automatically translate SML into SBP code. Notably, also assumption scenarios are translated into SBP scenarios; they monitor whether the environment satisfies the as-

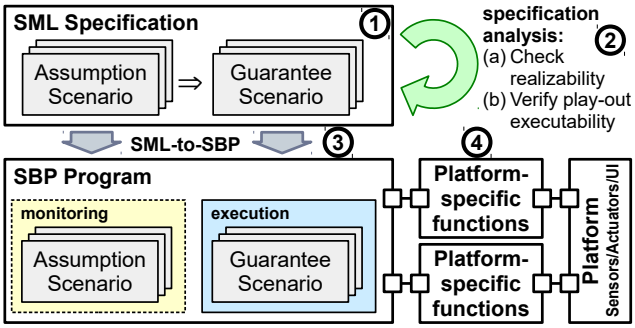

Fig. 1. Scenario-based Design Process (SBDP) illustration

sumptions (run-time assumption validation).

4) *Extension with platform-specific code:* Code is added to bridge problem-specific events in the specification to platform-specific sensor and actuator events. *Example:* In the specification it may be natural to describe an event *robot enters room*. On the platform, this involves inferring information from multiple sensors. Such a logic may itself be a complex reactive subsystem; then it can also be implemented in SBP or by applying the process here.

In this paper, we describe SBDP by the example of a cleaning robot system (our variant of the *Rover* challenge problem) (Sect. III). We show parts of the SML specification, give a brief explanation of play-out and the formal checks (Sect. IV), and explain the generated SBP code (Sect. V). We also describe some platform-specific functions required to execute on a RaspberryPi-based robot (Pi2Go) (Sect. VI). Last, we overview related work (Sect. VII) and conclude (Sect. VIII).

The SBDP approach appears also in [4], but with a different example. The contribution of this paper is demonstrating SBDP for the development of mobile robot system for comparison in the MDE Tools Challenge 2017 workshop.

II. Challenge Problem: A Vacuum Cleaner Robot System

As an example, we consider a vacuum cleaning robot system where robots move between rooms in a given layout. Figure 2 shows a simple layout with three rooms and one robot. Some rooms are equipped with dirt sensors that notify a central robot manager when a room gets dirty. The robot manager then orders a robot to start moving and cleaning. When moving, the robot's battery discharges and it has to recharge at a charging station before it runs out of charge.

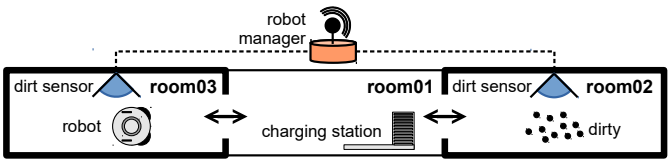

Fig. 2. Vacuum cleaner robot system sketch, simple 3-room topology

III. Scenario-Based Specification

An SML specification defines how objects in an *object model* must interact by sending messages. We consider *synchronous* communication where the sending and receiving of a message is a single *message event*. A message has one sending and one receiving object and it refers to an operation or property defined for the receiving object. When referring to a property, a message event has a side-effect on that property value for the receiving object, such as adding/removing elements from a list (e.g. rManager→rManager.dirtyRooms.add(dirtyableRoom)) or setting a property value (robotCtrl→robotCtrl.setMoving(true)). A *run* of a system is an infinite sequence of message events and object model configurations that evolve from an initial object model due to message events side-effects.

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

An SML specification defines the valid runs of a system by a set of *assumption-* and *guarantee scenarios*. Guarantee scenarios describe how system objects may, must, or must not react to environment events; assumption scenarios describe what may, will, or will not happen in the environment, or how the environment may, will, or will not react to system events. A run *satisfies* an SML specification if it satisfies all guarantee scenarios or violates at least one assumption scenario.

Listing 1 shows parts of the SML specification of the cleaning robot system. An SML specification refers to a *domain* class model (line 5) that describes a set of possible object models for which the specification models the behavior. In this example, the class model defines rooms, robots, their software controller, and the robot manager. For execution and analysis, an initial object model, for example one as shown in Fig. 2, must be provided in an external configuration.

In line 7, the specification defines controllable objects by listing their classes in a corresponding section. All other objects are environment objects.

The *non-spontaneous events* section (line 9) lists events that we assume cannot happen spontaneously, but only when an assumption scenario specifies that they can happen. In this example, rooms can become dirty spontaneously, but a robot's room only changes when the robot was ordered to move.

The remaining specification consists of scenarios that can be organized in *collaborations*. A collaboration defines *roles*, which represent interacting objects. In this case, all scenarios are grouped in one collaboration.

A scenario specifies what are valid sequences of message events and object model conditions. Message events are modeled as *scenario messages*. A scenario message refers to a sending and a receiving role, an operation or property defined by the receiving role's class, and possibly message parameter expressions.

When a message event occurs in the system that matches the first message in a scenario, an *active copy* of the scenario

is created and the sending and receiving roles are bound to the sending and receiving objects of the message event. Further roles appearing in the scenario must be bound based on *binding expressions* (e.g. line 36) or they are bound to objects referenced as parameters by the message event (e.g. line 26). The active copy of the scenario progresses as further message events occur that match the subsequent messages in the context of the previously determined role bindings.

The messages can have different modalities such as *strict*, *urgent*, or *eventually*. When the scenario progresses to these messages—we say that they become *enabled*—the modalities influence what may, must, and must not happen next. When a scenario proceeds to a *strict* message, it means that no message event must occur that corresponds to a message in the same scenario that is not currently enabled. When an *urgent* system message is enabled, it means that a corresponding message event must occur before the next environment event occurs. When an *eventually* message is enabled, a corresponding message event must occur eventually. Violations of the eventually modality are called *liveness violations*, violations of the strict and urgent modalities are *safety violations*.

A scenario can also specify *wait*, *interrupt*, and *violation* conditions. When an active scenario progresses to an interrupt condition that evaluates to true, the active scenario terminates; otherwise it progresses immediately. When the active scenario reaches a wait condition that evaluates to false, the active scenario does not progress until the condition renders true; when it evaluates to true, it immediately progresses beyond it. Wait conditions can have the eventually modality, meaning that the condition must become true eventually (otherwise, this is again a liveness violation). If an active scenario progresses to a violation condition that evaluates to true, this is a safety violation.

Example scenarios explained: The RobotManagerAddsDirtyRoomToListAndSelectsRobot scenario specifies that when a room is reported as being dirty, the robot manager must add the room to a list of dirty rooms and select any of its associated robots, resp. their controllers, for cleaning it. The scenario RobotManagerOrdersRobotToCleanRoom specifies that after selecting a robot controller, the robot manager must signal that robot controller to start moving the robot. The EventuallyCleanDirtyRoom scenario specifies that once a robot manager registers a room in a list of dirty rooms, this room must eventually be removed from that list. The scenario MoveRobotWhenOrderedTo specifies that after a robot is ordered to move, the robot must move to any of the adjacent rooms. The *any* in the parameter expression means that one of the adjacent rooms of the robot's current room can be selected non-deterministically. How the robot should select a route to the next dirty room is not specified in this scenario—it is often desirable during the early design to under-specify in such a way, when a concrete solution has not yet been decided upon. The behavior can be refined by adding further scenarios that, for example, select a next rooms from a list of rooms that was computed as the shortest path to a dirty room. (Details are omitted for brevity.)

```
1  import "vacuumrobot.ecore"
2
3  specification VacuumrobotSpecification {
4
5  domain vacuumrobotmodel // class model
6
7  controllable { VacuumRobotManager VacuumRobotController }
8
9  non-spontaneous events { VacuumRobotController.setRoom
       VacuumRobotController.setCharge}
10
11 collaboration VacuumrobotCollaboration {
12
13  static role VacuumRobotManager rManager
14  dynamic role DirtyableRoom dirtyableRoom
15  dynamic role Room room
16  dynamic role VacuumRobot robot
17  dynamic multi role VacuumRobotController robotCtrl
18
19  guarantee scenario
        RobotManagerAddsDirtyRoomToListAndSelectsRobot{
20   dirtyableRoom->rManager.dirtDetected
21   strict urgent rManager->rManager.dirtyRooms.add(dirtyableRoom)
22   strict urgent rManager->rManager.orderRobotToCleanRoom(
         rManager.vacuumRobotControllers.any())
23  }
24
25  guarantee scenario EventuallyCleanDirtyRoom{
26   rManager->rManager.dirtyRooms.add(bind dirtyableRoom)
27   wait eventually [!rManager.dirtyRooms.contains(dirtyableRoom)]
28  }
29
30  guarantee scenario RobotManagerOrdersRobotToCleanRoom{
31   rManager->rManager.orderRobotToCleanRoom(bind robotCtrl)
32   strict urgent rManager->robotCtrl.startMoving()
33  }
34
35  guarantee scenario MoveRobotWhenOrderedTo
36  bindings [ robot = robotCtrl.robot ]{
37   rManager->robotCtrl.startMoving()
38   interrupt [robotCtrl.moving]
39   var Room currentRoom = robotCtrl.room
40   strict urgent robotCtrl->robot.moveToAdjacentRoom(currentRoom.
         adjacentRooms.any())
41   strict urgent robotCtrl->robotCtrl.setMoving(true)
42  }
43
44  guarantee scenario RobotMustEventuallyRecharge{
45   robot->robotCtrl.setCharge(*)
46   wait eventually [robotCtrl.charge == robot.maxCharge]
47  }
48
49  guarantee scenario RobotMustNotRunOutOfCharge{
50   robot->robotCtrl.setCharge(*)
51   violation [robotCtrl.charge <= 0]
52  }
53  // further guarantee scenarios ...
54
55  assumption scenario RobotArrivesInRoom {
56   robotCtrl->robot.moveToAdjacentRoom(bind room)
57   var Room currentRoom = robotCtrl.room
58   interrupt [!currentRoom.adjacentRooms.contains(room)]
59   eventually robot->robotCtrl.arrivedInRoom(room)
60  }
61
62  assumption scenario RobotDischargesOrRecharges {
63   robot->robotCtrl.arrivedInRoom(bind room)
64   alternative [room.hasChargingStation]{
65    strict committed robot->robotCtrl.setCharge(robot.maxCharge)
66   } or [!room.hasChargingStation]{
67    strict committed robot->robotCtrl.setCharge(
              robotCtrl.charge-1)
68   }
69  }
70  // further assumption scenarios ...
71 }
72
```

Listing 1. Part of the vacuum robot system SML specification

The assumption scenario RobotArrivesInRoom specifies that when a robot is ordered to move to an adjacent room, and only if the room is really adjacent to its current room, it will eventually arrive in that room. The scenario RobotDischargesOrRecharges specifies the assumption that the robot's battery will discharge when moving to a next room, and it will be recharged if it moves to a room with a charging station.

IV. Play-out, Realizability Checking, Verification

The *play-out* algorithm defines an execution semantics for SML specifications. In brief, it works as follows: Initially, the system waits for environment events to occur. When an environment event occurs that activates or progresses guarantee scenarios such a way that urgent or eventually system messages are enabled, the system non-deterministically chooses a corresponding system message for execution, as long as it is not *blocked* by another active guarantee scenario, i.e., would lead to a safety-violation. The system can send further messages until there are no more enabled system messages marked as urgent or eventually. Then the system waits for the next environment event and the process is repeated. We assume that the system is fast enough to execute any finite number of message events before the next environment event occurs. ScenarioTools supports an interactive play-out of the SML specification, which can be used for validation.

Moreover, ScenarioTools is capable of building a graph of all possible play-out executions. In that graph, edges are labeled with environment and system message events; the structure can be viewed as an infinite two-player game, played by the system against the environment. The system wins the game if, no matter what environment events the environment chooses, the system can chose system events in such a way that the resulting path corresponds to a run that satisfies the specification. If this is not the case, the specification is *unrealizable* and must be fixed. The game is a GR(1) game that ScenarioTools solves by an implementation of the GR(1) game algorithm by Chatterjee et al. [5]. If the specification is realizable, the algorithm produces a strategy of how the system can satisfy the specification; if the specification is unrealizable, the algorithm produces a counter-strategy of how the environment can always force a violation. The counter-strategy helps understand the specification flaw.

If the specification is realizable, but the strategy identifies that not all system moves in the play-out graph are winning, it means that during play-out, the system may make choices that do not lead to a valid run. For example, when the cleaning robot is in room 01, and room 02 gets dirty, but the robot only moves between room 01 and room 03, the dirty room may never be cleaned. In such a case, the specification can be refined, for example as explained before, in order to constrain the system choices, so that each play-out choice is winning.

When this property is satisfied, we can generate SBP code from the SML specification, which will then perform a play-out of the specification that is guaranteed to satisfy the specification.

```
1  public class RobotManagerOrdersRobotToCleanRoomScenario extends
       VacuumrobotCollaboration {
2
3   @Override
4   protected void registerAlphabet() {
5    setBlocked(rManager, rManager, "orderRobotToCleanRoom",
6         RANDOM);
7    setBlocked(rManager, robotCtrl, "startMoving");
8   }
9
10  @Override
11  protected void initialisation() {
12   addInitializingMessage(new Message(rManager, rManager,
13        "orderRobotToCleanRoom", RANDOM));
14  }
15
16  @Override
17  protected void body() throws Violation {
18   robotCtrl.setBinding((VacuumRobotController) getLastMessage().
       getParameters().get(0));
19   request(STRICT, rManager, robotCtrl, "startMoving");
20  }
21 }
```

Listing 2. SBP code for the RobotManagerOrdersRobotToCleanRoom scenario

V. Scenario-Based Programming (SBP)

ScenarioTools provides the SBP framework in which scenarios specifications can be programmed in a way that is very similar to SML. SBP builds on the Behavioral Programming for Java framework, BPJ [6]. A BPJ program consists of a collection of collaborating *BThreads*. Scenarios can be programmed in SBP by extending a special scenario BThread class, and SBP adds certain BThreads so that the scenario BThreads are executed according to the play-out algorithm.

SBP is suitable for programming scenarios manually, but it is also a convenient target for code generation from SML. Listing 2 shows the SBP scenario generated from the RobotManagerOrdersRobotToCleanRoom scenario. The class overrides different methods from a scenario superclass: the registerAlphabet method registers all messages appearing in the scenario. The initialisation method registers the messages upon which an instance of the scenario BThread will be started. The body method implements the scenario behavior after the occurrence of the first message. Here, a role binding must be performed based on a parameter that was carried by the initializing message event. A call of the request method makes the BThread yield and hand over different message events to the play-out event selection mechanism. For more details, see [4], [7].

VI. Platform-Specific Extension

The SBP code from the example can be executed in a distributed setting, where the robot controller runs on a RaspberryPi-based robot (Pi2Go) that mimics a cleaning robot, and the robot manager runs on a different computer, both communicating via MQTT. The Pi2Go moves on a grid that represents the room layou, see Fig. 3. The dirtDetected events can be injected via a GUI. moveToAdjacentRoom events are translated into commands that make the robot turn and follow a line to the next 'room'. When the robot arrives at the next 'room', another platform-specific component translates the sensor inputs into arrivedInRoom and setCharge events.

See the demo video here: https://youtu.be/VsSbueeIVYk.

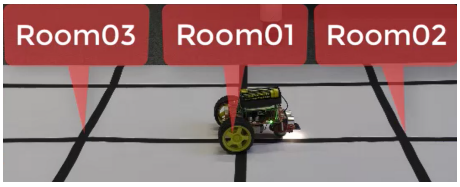

Fig. 3. The RaspberryPi robot in action

VII. Related Work

Harel and Maoz describe a mapping from LSCs to AspectJ (S2A) for play-out [8]. It is similar to SBP, but SBP scenario threads resemble the SML scenarios more closely than the AspectJ code. Also, no extension exists for the distributed execution of S2A code.

Other MDE tools for designing reactive systems exist, but most use component- and state-based models, such as MechatronicUML [9], UML-RT resp. Papyrus-RT [10], Matlab Simulink Stateflow, or the P language [11].

VIII. Conclusion

a) Strengths:: The main strength of the approach is the requirements-aligned nature of the modeling: engineers can formally model the behavior similar to use case scenarios. Scenarios can be added to add behavior as well as restrict previous behavior. Moreover, the scenarios specify the interaction of components rather than describing the behavior of each of the components. Further strengths are:

- The formal realizability checking and verification that it supports.
- The assumption validation at run-time (assumption scenario monitoring).
- The distributed execution capabilities.
- Support for systems with dynamic topology

b) Weaknesses:: The flexible modeling with overlapping behavior aspects spread across many scenarios has the problem that detecting and understanding specification flaws can be difficult. However, the same problem arises also when using informal use-cases and scenarios—then the problem is only worse, because no automated checks can help find them early. Further weaknesses are:

- Currently, only the generation of SBP Java code is supported. The concepts, however, can also be mapped to C++ code.
- The SBP code relies on the play-out algorithm where active scenarios are run as separate threads that coordinate for event-selection. This creates memory and time overhead, which may be critical for applications where hardware resources are sparse. We propose alternative approaches in this case [12], [13].
- Currently, ScenarioTools supports no real-time constraint in SML, but a previous tool version shows how real-time constraints can be integrated [12].

Acknowledgments:: This work is funded by the German Israeli Foundation for Scientific Research and Development (GIF), grant No. 1258.

References

[1] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," in *Formal Methods in System Design*, vol. 19, 2001, pp. 45–80.

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

[3] J. Greenyer, D. Gritzner, G. Katz, and A. Marron, "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)*, J. de Lara, P. J. Clarke, and M. Sabetzadeh, Eds., vol. 1725. CEUR, 2016, pp. 16–32.

[4] J. Greenyer, D. Gritzner, F. König, J. Dahlke, J. Shi, and E. Wete, "From scenario modeling to scenario programming for reactive systems with dynamic topology," in *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 4-8, 2017 (to appear)*. ACM, 2017.

[5] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer, "Conditionally Optimal Algorithms for Generalized Büchi Games," in *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Faliszewski, A. Muscholl, and R. Niedermeier, Eds., vol. 58. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 25:1–25:15.

[6] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Comm. ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[7] F. W. H. König, "Szenariobasierte Programmierung und verteilte Ausführung in Java," Master's Thesis, Leibniz Universität Hannover, Software Engineering Group, 2017. [Online]. Available: http://jgreen.de/wp-content/documents/msc-theses/2017/Koenig2017.pdf

[8] S. Maoz, D. Harel, and A. Kleinbort, "A compiler for multimodal scenarios: Transforming lscs into aspectj," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 18:1–18:41, Sep. 2011.

[9] D. Schubert, C. Heinzemann, and C. Gerking, "Towards safe execution of reconfigurations in cyber-physical systems," in *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, April 2016, pp. 33–38.

[10] N. Kahani, N. Hili, J. R. Cordy, and J. Dingel, "Evaluation of uml-rt and papyrus-rt for modelling self-adaptive systems," in *Proceedings of the 9th International Workshop on Modelling in Software Engineering*, ser. MISE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 12–18.

[11] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "Drona: A framework for safe distributed mobile robotics," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ser. ICCPS '17. New York, NY, USA: ACM, 2017, pp. 239–248.

[12] C. Brenner, J. Greenyer, and W. Schäfer, "On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications," in *Fundamental Approaches to Software Engineering (FASE 2015)*, ser. Lecture Notes in Computer Science, A. Egyed and I. Schaefer, Eds. Springer, 2015, vol. 9033, pp. 51–65.

[13] D. Gritzner and J. Greenyer, "Controller synthesis and PCL code generation from scenario-based GR(1) robot specifications," in *Proceedings of the The 4th International Workshop on Model-driven Robot Software Engineering, STAF 2017, Marburg, Germany (to appear)*, 2017.

Appendix

See a demo video here: https://youtu.be/VsSbueeIVYk.

ScenarioTools can be installed following the setup instructions: http://scenariotools.org/downloads/download/.

The example project is located in https://bitbucket.org/jgreenyer/scenariotools-sml-examples/src//org.scenariotools.examples.sbp.vacuumrobotv2/?at=master (Import to Eclipse runtime workspace mentioned in the above setup instructions.)