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 of 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/esefse-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

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 or reconfigurable production systems, the system topology changes at run-time, i.e., their 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 ScenarioTools1. 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.

---

1Funded by the German Israeli Foundation for Scientific Research and Development (GIF), grant No. 1258.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE’17, September 4–8, 2017, Paderborn, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09 ... $15.00
https://doi.org/10.1145/3106237.3122827

1http://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 LCSs) 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 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 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
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 constrains 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.

SCENARIO-TOOLS 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, SML code can be generated.

3 SCENARIO-BASED PROGRAMMING

SBP [13] is based on the BPJ [11] framework. 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. SML 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.

---

2http://www.wisdom.weizmann.ac.il/~bprogram/bpj/
public class CarRegistersAtObstacleScenario
extends CarsPassObstacleCollaboration {
    @Override
    protected void initialise() {
        addInitMessage(new Message(cp, car, "approachingObstacle"));
    }

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

    @Override
    protected void registerRoleBindings() {
        bindRoleToObject(cp, oc, oc, "enteringDisallowed");
    }

    @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()));
            } else if (getLastMessage().equals(new Message(cp, car, "waitingCars", car.getBinding()))) {
                requestedMessages.add(new Message(STRICT, oc, oc, "enteringDisallowed"));
        }
        doStep(requestedMessages, waitedForMessages);
        // Determine which alternative was chosen
        if (getlastMessage().equals(new Message(cp, car, "approachingObstacle")) {
            request(STRICT, oc, car, "waitingCars", car.getBinding());
        } else if (getlastMessage().equals(new Message(cp, car, "enteringDisallowed")) {
            request(STRICT, car, oc, "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 CarPassObstacleCollaboration is an intermediate class generated from the collaboration; it provides the definitions of the roles (cp, car, oc).

The method initialise() is called when the SBP program starts. The initialising 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.

4 RELATED WORK

In previous work [6, 8], we presented a prototype for the distributed execution of SML specifications by executing them by the Scenario-Tools 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.

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


