ScenarioTools – A Tool Suite for the Scenario-based Modeling and Analysis of Reactive Systems

Joel Greenyer[a], Daniel Gritzner[a], Timo Gutahr[a], Florian König[a], Nils Glade[a], Assaf Marron[b], Guy Katz[c]

[a] *Leibniz Universität Hannover, Germany*
[b] *Weizmann Institute of Science, Israel*
[c] *Stanford University, USA*

Abstract

ScenarioTools is an Eclipse-based tool suite for the scenario-based modeling and analysis of reactive systems. ScenarioTools especially targets the modeling and analysis of systems where the behavior of the components is sensitive to changes in the component structure that can occur at run-time. For example, in a system of communicating cars, the cars' relationships can change due to their movement and influence how cars must interact. The modeling in ScenarioTools is based on the Scenario Modeling Language (SML), an extended variant of Live Sequence Charts (LSCs). For modeling structural changes and conditions, graph transformation rules can be combined with SML. The specifications are executable and can be analyzed by simulation. ScenarioTools further supports a formal synthesis procedure that can find specification inconsistencies or prove the specification's realizability. In this article, we illustrate the features of ScenarioTools by an example and describe its architecture.

Keywords:
reactive systems, dynamic system structure, scenarios, graph transformation, analysis, inconsistency, realizability, controller synthesis

1. Introduction

In this article, we present ScenarioTools, an Eclipse-based tool suite for the scenario-based modeling and analysis of *reactive systems*.

Reactive systems are systems that continuously react to external inputs [1]. We find them especially in the form of *cyber-physical* systems, which are systems of networked embedded components that control physical processes and interact with users or other systems. Such systems fulfill complex tasks in manufacturing, transportation, or logistics. There are a number of characteristics that make the development of such systems challenging.

First, **(1)** the systems are often *safety-critical*, which requires extra rigor, and safety concerns must be considered already during the early design.

Second, **(2)** the system goals can usually not be fulfilled by the software alone, but only in collaboration with its physical/mechanical environment.

Third, **(3)** in systems like communicating cars or adaptive production systems, the relationships between components change at run-time. This may change the roles and responsibilities of the system components, and influences how these components interact. Their interaction, in turn, affects the system's structure. In a system of communicating cars, for example, the system structure changes due to the movement of cars (physical relationships), or due to the assignment of leader- and follower roles in a convoy (virtual relationships). A car's software must then behave differently depending on the traffic situation and its role in it. The software can also influence how the system structure evolves, for example by advising the driver or by controlling the car. We call such systems *structurally dynamic* systems, as opposed to *static* systems, where the component structure does not changed at run-time.

To master these challenges, we propose a formal specification method that combines *scenario-based modeling* and *graph transformation rules*:

Scenario-based modeling allows engineers to capture specifications in a way that is very close to how they would naturally conceive and communicate the requirements, i.e., by describing, in separate stories, how the system may, must, or must not react to certain events. ScenarioTools supports the scenario-based modeling with the *Scenario Modeling Language* (SML), which is a textual variant of Live Sequence Charts (LSCs) [2].

SML specifications are *executable* via an extension of the *play-out algorithm* [3], originally invented for LSCs [4, 2]. This algorithm can be used to simulate the system execution and to analyze the interplay of the scenarios.

SML extends LSCs with the support for modeling environment assumptions in the form of *assumption scenarios*. Together with *guarantee scenarios*, which describe the desired behavior of the software/system, they form an *assume-guarantee* specification. This permits meaningful reasoning on the collaboration of the software and its environment (cf. **(2)** above).

Graph transformation rules (GTRs) are an intuitive formalism for modeling conditions and transformations of models. The model of interest in our case is the structure of system and environment components. GTRs can be integrated with SML to express under which conditions of the component structure certain events are possible to occur, and how they change the component structure when they occur. This provides rich means for modeling aspects of structural dynamism (cf. **(3)** above). For modeling and executing GTRs, ScenarioTools integrates Henshin, an Eclipse-based tool set for the modeling, execution, and analysis of graph transformation systems [5, 6].

Since simulation cannot prove the absence of flaws and extra rigor is required for addressing safety concerns (cf. **(1)** above), we developed a *controller synthesis and realizability checking* algorithm. This algorithm is based on ScenarioTools' capability of building a *play-out graph*, a state-transition graph of all possible play-out executions. A multi-level hashing approach allows for a fast exploration and memory-efficient representation of states. The algorithm considers the play-out graph as an infinite game played by the system against the environment [7]. It checks whether there exists a *strategy* for the system to choose system-controllable transitions such that, no matter what environment-controllable transitions the environment chooses, the resulting path, i.e., execution of the system, will satisfy the SML specification. If no such strategy exists, the algorithm will produce a *counter-strategy* that shows how the environment can force the system to violate the specification. ScenarioTools supports the interactive simulation of such a counter-strategy, which helps in understanding the specification flaw.

In this article, we illustrate the modeling, simulation and synthesis features of ScenarioTools. This article extends our tool demo paper [8][1] with an extended example and a description of the execution engine architecture.

The key novelty w.r.t. previous work [9] is the support for formal controller synthesis for specifications that combine scenarios and GTRs.

Structure: Background and related work appears in Sect. 2, ScenarioTools modeling and analysis features in Sect. 3 and 4, and architecture details in Sect. 5. We give evaluation results in Sect. 6 and conclude in Sect. 7.

[1] tool demo video: `https://youtu.be/p9mo6FJvqEE`)

2. Background and Related Work

There exist a number of approaches for the formal modeling and analysis of use cases and scenarios. See an overview by Liang et al. [10].

Live Sequence Charts (LSCs) [11], allow us to distinguish between scenarios that must be possible to occur in the system (*existential* scenarios) as well as scenarios that must be satisfied by every execution of the system (*universal* scenarios). By using *modalities* for messages, LSCs can express whether events may, must, or must not occur. Also, specifications of universal LSCs are executable via the *play-out* algorithm [4, 2].

Many approaches exist for checking LSC specifications for inconsistencies, or synthesizing implementations from them [12, 13, 14, 15, 16, 7].

For several years, we[2] have developed the ScenarioTools to evaluate our extensions to the scenario-based modeling methodology inspired by LSCs. We have recently switched from a graphical syntax [3, 9] to a textual language, which we call the *Scenario Modeling Language (SML)*. Besides the difference between the graphical vs. textual syntax, SML extends the concepts of LSCs by an explicit concept for modeling environment assumptions through *assumption scenarios* [3]. Another SML feature for modeling environment assumptions is to separate environment events that can occur spontaneously (e.g. "user presses the 'coffee' button on a coffee machine") from *non-spontaneous* environment events that can occur only in reaction to other events (e.g. "user takes coffee cup only after machine releases cup").

Tools related to ScenarioTools are the Play Engine [2] and the PlayGo tool [17]. The Play Engine and PlayGo support the interactive *play-in* of scenarios, which is not supported by ScenarioTools. Also PlayGo supports a controlled natural specification language [18]. PlayGo was furthermore extended by formal synthesis algorithms and the capability to interactively simulate counter-strategies [19]. This approach, however, is limited, as only a subset of LSC constructs is mapped to an external analysis framework; parameterized messages, dynamic polymorphic scenario bindings [20], or dynamic object structures are not supported. This limits the Play Engine and PlayGo w.r.t. the challenges mentioned above.

[2]The current version of ScenarioTools was developed by the first-listed authors from the Leibniz Universität Hannover. Examples were developed by all authors. The ScenarioTools architecture was inspired by a previous version of the tool, which was developed by the first-listed author, C. Brenner, and V. Panzica La Manna [3].

By contrast, the ScenarioTools' synthesis supports SML specifications with parameterized messages, dynamic polymorphic scenario bindings, and, especially, specifications for structurally dynamic systems.

3. Scenario-based Modeling in ScenarioTools

3.1. Example Overview

Our example is an advanced *car-to-x* driver assistance system that shall help cars efficiently pass obstacles that create a narrow passage by blocking one lane of a two-lane road (see top of Fig. 1). Cars approaching on the blocked lane must stop if cars approach the obstacle from the other direction.

One scenario (**Scenario 1** in Fig. 1) says that when a car approaches the obstacle on the blocked lane, it must show either a STOP or GO signal to the driver, and it must do so before the car finally reaches the obstacle.

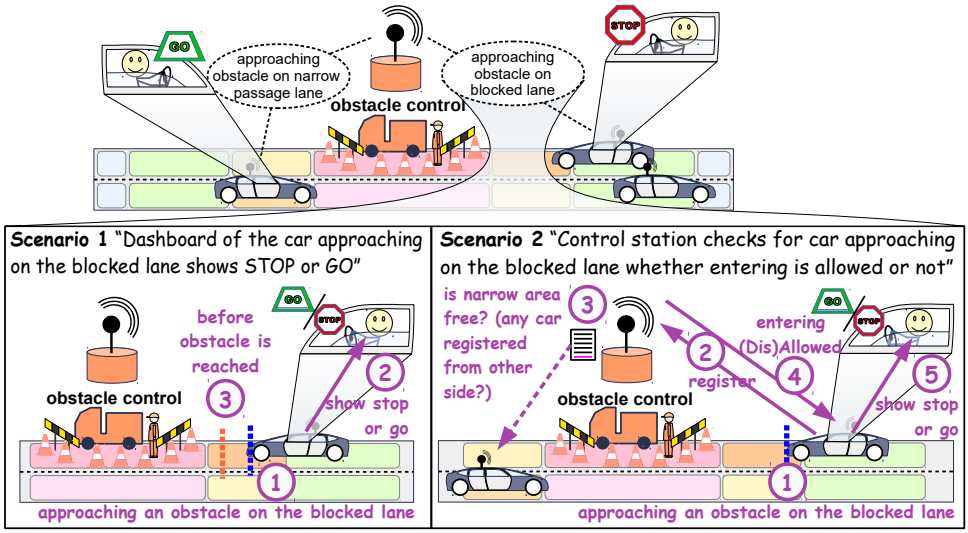

Figure 1: Car-to-X example overview

A second scenario (**Scenario 2** in Fig. 1) refines and extends the behavior described by the first: when a car approaches the obstacle on the blocked lane, it must register at the *obstacle control* station. When another car has previously registered for approach from the opposite direction, the obstacle control must disallow the car approaching on the blocked lane to enter and the car must show the STOP signal to the driver. Otherwise, the obstacle control must allow the car to enter and the car must show the GO signal.

The example shows how a non-deterministic choice between showing STOP or GO in **Scenario 1** is determined by a condition in **Scenario 2**.

To specify the system further, more scenarios are added. The system must satisfy the requirements expressed by all the scenarios.

3.2. Modeling Overview

An SML specification defines how objects in an *object model* may, must, or must not interact. These objects are instances of a *class model*, which an SML specification references. A *run-configuration* maps an SML specification to a particular object model, and it can be read and executed by the ScenarioTools execution engine for simulation and formal analysis. See Fig. 2.

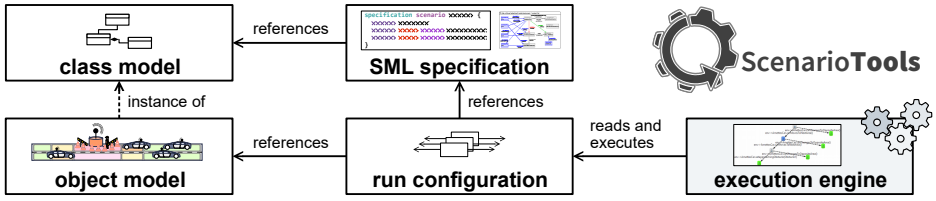

Figure 2: Overview over the involved models for execution

3.3. Class and object model

In ScenarioTools, class models are modeled in Ecore of the Eclipse Modeling Framework (EMF). EMF allows us to create instances of such models. The Car-to-X class model (Fig. 3) models street systems that can contain cars, street sections, and other elements. Street sections have lanes that consist of lane areas, and cars can be in lane areas. Furthermore, there can be obstacles on lane areas, which can be controlled by an obstacle control.

3.4. SML Specification

The first part of the Car-to-X specification is shown in Listing 1. It shows the specification CarToX, which imports the car-to-x.ecore file that contains the cartox package. In the following, we explain the language concepts.

3.4.1. Controllable and uncontrollable classes

The SML specification first defines which classes of objects are *controllable* (lines 7-10); all others are *uncontrollable*. Controllable classes represent components for which we specify the (software) behavior. Here, this is

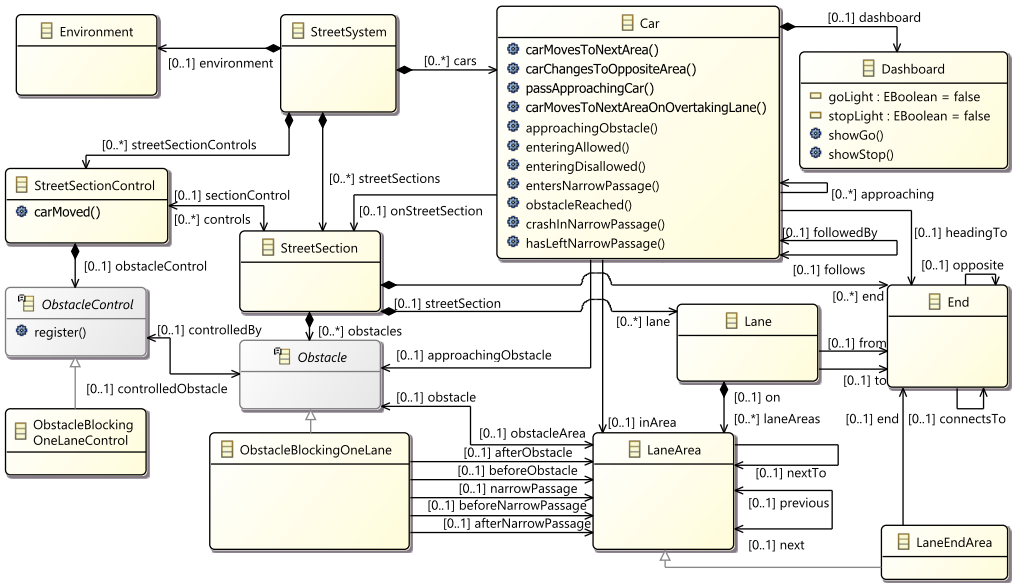

Figure 3: Car-to-X specification class model (Ecore class diagram)

the car and the control station for an obstacle that blocks one street lane. Uncontrollable classes model environment entities or sensors or actuators. Environment objects are the source of *environment events*. In our case, the class Environment is an abstraction of the car's sensors. For example, the environment can issue events about the movement of cars to the next lane area. In the real system, wheel sensors or a GPS may send such an event.

3.4.2. Collaborations, roles, and scenarios

A specification contains one or more *collaborations*. Each collaboration describes how objects interact in a particular situation. A collaboration defines *roles* that represent objects. Furthermore, a collaboration contains *guarantee scenarios* and *assumption scenarios*. Guarantee scenarios describe how the controllable objects may, must, or must not react to environment events. Assumption scenarios describe what can, will, or will not happen in the environment or how the environment will react to the system.

The first two guarantee scenarios in Listing 1 model the scenarios in Fig. 1. CarsMustNotCrash says that no crashes must occur in the narrow passage.

Objects interact by sending *messages*. A message has one sending and one receiving object and refers to an operation defined for the receiving object. We consider *synchronous* messages, where both the sending and receiving of

```
1  import "car-to-x.ecore"
2
3  specification CarToX {
4
5  domain cartox // reference Ecore package
6
7  controllable {
8   Car
9   ObstacleBlockingOneLaneControl
10 }
11
12 non-spontaneous events {
13  Car.setApproachingObstacle
14  ...
15 }
16
17 collaboration ApproachingObstacleOnBlockedLane{
18
19  static role Environment env
20  dynamic role Car car
21  dynamic role Dashboard dashboard
22  dynamic role ObstacleBlockingOneLaneControl obstacleControl
23
24  // Scenario 1
25  guarantee scenario DashboardOfCarApproachingOnBlockedLaneShowsStopOrGo
26  bindings [
27   dashboard = car.dashboard
28  ]{
29   env->car.setApproachingObstacle(*)
30   alternative{
31    strict requested car->dashboard.showGo()
32   } or {
33    strict requested car->dashboard.showStop()
34   }
35   env->car.obstacleReached()
36  }
37
38  // Scenario 2
39  guarantee scenario ControlStationAllowsCarOnBlockedLaneToEnterOrNot
40  bindings [
41   obstacle = car.approachingObstacle
42   obstacleControl = obstacle.controlledBy
43   dashboard = car.dashboard
44  ]{
45   env->car.setApproachingObstacle(*)
46   strict requested car->obstacleControl.register()
47   alternative [obstacleControl.carsRegisteredOnNarrowPassageLane.isEmpty()]{
48    strict requested obstacleControl->car.enteringAllowed()
49    strict car->dashboard.showGo()
50   } or [!obstacleControl.carsRegisteredOnNarrowPassageLane.isEmpty()]{
51    strict requested obstacleControl->car.enteringDisallowed()
52    strict car->dashboard.showStop()
53   }
54  }
55
56  guarantee scenario CarsMustNotCrash {
57   env->car.crashInNarrowPassage()
58   violation [true]
59  }
60  ... // continues in Listing 2
```

Listing 1: Car-to-X SML specification excerpt part 1 (specification scenarios)

a message together form a single *message event*. Message events sent from controllable objects are called *controllable events* or *system events*; message events sent from uncontrollable objects are called *uncontrollable events* or *environment events*. System events can represent the sending of a software message or a signal sent to an actuator. Environment messages represent events in the environment or an external signal sent to the system.

A message event can furthermore have a *side-effect* on the object model, like changing attribute values and reference links of objects. This can represent the changing of reference or variable values in the system's software or the change of the state of the environment. For example, an event carMovesToNextArea, sent from the environment to the car, has the side-effect of changing the car's inArea reference to point to the next lane area. Such side-effects can be modeled by graph transformation rules (see Sect. 3.5). Moreover, message events that refer to operations of the form set⟨Prop⟩(p) will cause the value for attribute/reference ⟨Prop⟩ of the receiving object to change to the value that the message event carries for p.

The side-effect of a message event can also be the creation or destruction of objects in the object model, but this is currently not supported by ScenarioTools. For modeling most systems, this poses no limitation, since object creation and destruction can also be modeled by other means, for example by a pool of objects that can be flagged as "dead" or "alive".

An infinite sequence of object model states and message events is called a *run* of a system. Each scenario either *accepts* or *rejects* a run. A run is *valid* w.r.t. a specification if and only if it is accepted by all guarantee scenarios or it is rejected by at least one assumption scenario, i.e., the system must satisfy the requirements in environments that fulfill the assumptions.

An SML scenario rejects a run if and only if the run causes a *safety violation* or *liveness violation* (see Sect. 3.4.4) in the scenario. Conversely, a scenario accepts a run if it is never activated or progresses without violations.

3.4.3. Scenario activation and progress

On the occurrence of a message event that corresponds to the first message in a scenario, an *active copy* of that scenario, also called *active scenario*, is created. On the occurrence of further message events that correspond to the subsequent messages in the scenario, the scenario progresses. SML supports alternative, parallel, and loop constructs within the scenarios to control the flow of progress. The state of progress is defined by the set of messages that the scenario waits for to occur next. We call these messages *enabled*.

There can be multiple enabled messages if the scenario contains alternative or parallel fragments. When the active scenario progresses until its end, the active scenario terminates and is discarded. There can be multiple active scenarios at the same time, even multiple active copies of the same scenario.

3.4.4. Message modalities, violations, and interrupts

The messages in the scenarios can have the modalities strict and requested. The strict modality allows us to express at which state of progress of the scenario the order of events as described by the scenario must not be violated. More specifically, when a strict message is enabled, then no message event must occur that corresponds to a message in the same active scenario that is not currently enabled. Otherwise, it is a *safety violation*. If, instead, there are no strict messages enabled, message events that are expected only elsewhere in the active scenario are allowed, but lead to the termination of the active scenario. Such a termination is also called an *interrupt*.

The modality requested indicates points where the scenario must progress. If a requested message is enabled forever, this is called a *liveness violation*.

Safety violations can also be caused by violation expressions. If in an active scenario the progress reaches a violation expression and the condition evaluates to true, this is also a safety violation. If the condition evaluates to false, the active scenario progresses before the next message event occurs. Similarly, interrupt expressions can lead to the interrupt of the active scenario.

3.4.5. Roles and dynamic binding

Each message in a scenarios has a sending and a receiving role. Roles can be *static* or *dynamic*. Each static role has a fixed binding to one object in the object model, which is configured in the run-configuration (see Listing 3). A dynamic role can have a different binding for each active scenario. The binding of the sending and receiving roles of the first message are given through the occurrence of the message event that activates the scenario. The bindings of the other roles in the scenario are defined through *binding expressions* that refer to properties of objects bound to other roles. This enables us to specify behavior that is sensitive to the current state of the object model. All roles are bound at the time of scenario activation.

3.4.6. Assumption scenarios and non-spontaneous events

Listing 2 shows two assumption scenarios. The first one describes that after a car moved onto a new area, and the area after this now *current*

```
1     ... // continued from Listing 1
2    assumption scenario ApproachingObstacleOnBlockedLaneAssumption
3    bindings [
4     currentArea = car.inArea
5     nextArea = currentArea.next
6     obstacle = nextArea.obstacle
7    ]{
8     env->car.carMovesToNextArea()
9     interrupt [obstacle == null]
10    strict requested env->car.setApproachingObstacle(obstacle)
11   } constraints [
12    forbidden env->car.carMovesToNextArea()
13   ]
14
15   assumption scenario DriverObeysStopSignal {
16    car->dashboard.showStop()
17    car->dashboard.showGo()
18   } constraints [
19    forbidden env -> car.carMovesToNextArea()
20    forbidden env -> car.carMovesToNextAreaOnOvertakingLane()
21   ]
22  } // ... additional collaborations and scenarios
23 }
```

Listing 2: Car-to-X SML specification excerpt part 2 (assumption scenarios)

area has an obstacle, then the car will eventually receive the event that it is approaching that obstacle (env->car.setApproachingObstace(obstacle)). The scenario also has a *forbidden message* in its constraints section, which models message events that must not occur while the scenario is active. If forbidden message events do occur, this is a safety violation. Here, it says that a second carMovesToNextArea will not occur before the setApproachingObstace event.

Reference or attribute values of objects are changed as a side-effect of messages prefixed with set. In this example, when setApproachingObstacle(obstacle) is received by a car, the car's value for the reference approachingObstacle is set to the object carried for the obstacle parameter. By setting this attribute, we can specify at which obstacle control the car shall register.

Another form of environment assumptions are *non-spontaneous* events. These are environment events that only occur in reaction to other events. In the SML specification, we can list operations of classes in a corresponding section (Listing 1, lines 12-15). Message events that refer to operations listed there are called *non-spontaneous* and it is assumed that they will occur only when a corresponding message is enabled in an active assumption scenario.

By adding the operation Car.setApproachingObstacle to the non-spontaneous events section, and in combination with the assumption scenario ApproachingObstacleOnBlockedLaneAssumption, we can express that a setApproachingObstace event can indeed only occur when the car has moved to a

lane area that is followed by another lane area with an obstacle on it.

The last assumption scenario models the assumption that a car will not advance to the next lane area after the STOP signal was shown to the driver and before the GO signal is shown again, i.e., drivers obey the STOP signal.

3.5. SML and Graph Transformation Rules

Message events can be associated with graph transformation rules (GTRs). A GTR describes a transformation of the object model that is the side-effect of an occurrence of the message event, and it also describes a condition under which the occurrence of the message event is possible. Depending on whether the GTR corresponds to a system or environment event, the condition is either a guarantee property or an assumption property. I.e., if a system or environment event occurs that is forbidden by a corresponding GTR, then this is a safety violation in the guarantees or assumptions.

Figure 4 shows a rule from the Car-to-X example. A GTR corresponds to a message event if its name is equal to the name of the message event's operation. Furthermore, the GTR must have two *in*-parameters corresponding to the message event's sending and receiving objects. If the operation is parameterized, the GTR must have further corresponding *in*-parameters.

Background: GTR parameters are a feature of HENSHIN. Each parameter corresponds to a node in the GTR. Before applying the GTR, bindings for the *in*-parameters must be provided. The GTR is applicable if a match of the GTR's left-hand-side (lhs) graph can be found in the host model where the nodes corresponding to *in*-parameters are bound to the specified objects. A GTR application consists in modifying the objects matched by the GTR's lhs such that it matches the GTR's right-hand side graph. For details see [5, 6].

A message event with a corresponding GTR can only occur if the GTR is applicable in the current object model for the *in*-parameters that are provided by the message event as described above. Upon the occurrence of the message event, the GTR is applied. If there are multiple matches of the GTR in the current object model, one is selected non-deterministically.

The example GT rule in Fig. 4 expresses that on the occurrence of the event carMovesToNextArea, the receiving car's inArea link will change to express that the car moves to the next area relative to its current area. Moreover, the rule constrains that the event cannot occur, for example, when the next lane area is occupied an obstacle. Also, the car cannot advance to the next lane area if it is following a car that still resides on the same lane area.

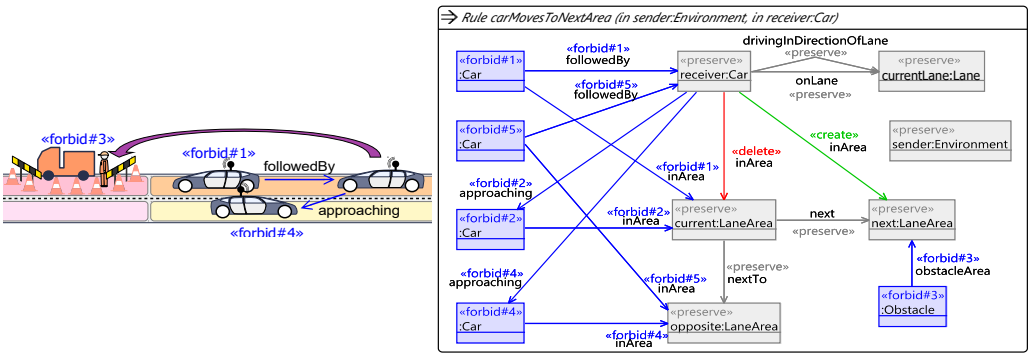

Figure 4: A GT rule that describes when and how a car moves to the next lane area.

There can also be GTRs without side-effect. The GTR in Fig. 5 models the condition under which message events env->car.crashInNarrowPassage() can occur, namely when a cars approaches another car in a narrow passage.

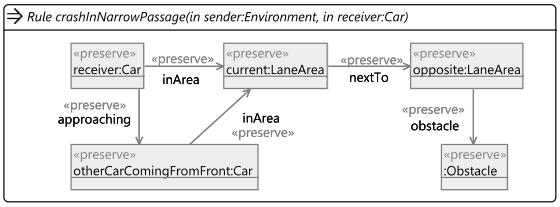

Figure 5: The condition for env->car.crashInNarrowPassage() events modeled as a GTR.

3.6. Run configuration

The run-configuration for the Car-to-X example is shown in Listing 3. It refers to the CarToX SML specification and an object model in the file StreetSectionWithObstacleAndTwoCars.xmi. The run-configuration also specifies the binding of all static roles to objects in the object model. Here the role env in collaboration CarsRegisterWithObstacleControl is mapped to the object env.

4. Simulation and Realizability Checking

4.1. Simulation

ScenarioTools supports the execution and simulation of the SML+GTR specifications, based on an extended play-out algorithm [2, 3].

```
import "../car-to-x.sml"
configure specification CarToX
use instancemodel "StreetSectionWithObstacleAndTwoCars.xmi"

rolebindings for collaboration CarsRegisterWithObstacleControl {
   object StreetSectionWithObstacleAndTwoCars.env plays role env
}
```

Listing 3: Run configuration

4.1.1. The ScenarioTools *extension of the play-out algorithm*

The algorithm repeatedly selects and executes events based on the currently active scenarios (see Alg. 1). Initially, there are no active scenarios. The execution of an event entails the initialization, progress, and termination of scenarios, and performing the event's side-effect on the object model.

System step: If there are system message events that correspond to enabled requested messages in active guarantee scenarios, it is the system's turn (l. 2). All of these system events are candidates for execution, unless they are blocked by active guarantee scenarios or disallowed by their corresponding GTR (l. 3). If the resulting candidate event set is not empty, one of them is chosen for execution (l. 5) and the process is repeated. Otherwise, this implies a safety violation of the guarantees (l. 7). The algorithm then continues with environment steps to see whether the environment forced a guarantee violation only at the expense of a later assumption violation.

Environment step: If it is not the system's turn, the system waits for the next environment event. The candidate events are such environment events that are spontaneous or corresponds to an enabled message in an active assumption scenario (l. 9). Furthermore, the event must not lead to a violation in any active assumption scenario and, if a GTR is associated with that event, the GTR must allow the event to occur (l. 10). If no such event exists, it means that violation of the assumptions and the algorithm terminates. Otherwise, an environment event is selected and executed (l. 12).

Assumption/guarantee violations: During event execution, an environment or system event may cause a violation of a guarantee resp. assumption scenario. The algorithm terminates on assumption violations, but continues on guarantee violations in order to check (as above) whether the environment forced a guarantee violation at the expense of a later assumption violation.

Event selection: The event selection (*selectFrom()*) is non-deterministic. It can be a user choice in the interactive simulation (cf. Sect. 4.1.2); for controller synthesis / realizability checking, all choices are explored (cf. Sect. 4.2).

Algorithm 1 Play-out event selection and execution

1: Σ_{Sys} = System message events that correspond to enabled requested messages in guarantee scenarios
2: **if** $\Sigma_{Sys} \neq \emptyset$ **then** // system step:
3: $\Sigma_{ex} = \Sigma_{Sys} \setminus$ Message events that are blocked by an active guarantee scenario or disallowed by a corresponding GTR.
4: **if** $\Sigma_{ex} \neq \emptyset$ **then**
5: $\sigma_{ex} =$ *selectFrom*(Σ_{ex}); *perfromStep*(σ_{ex}); **goto** 1
6: **else** // Safety-violation occurred in guarnatees
7: **goto** 9 // Cont. env. steps, check for subseq. assumption violation.
8: **else**// environment step:
9: Σ_{Env} = Environment message events that are spontaneous or correspond to enabled messages in assumption scenarios.
10: $\Sigma_{ex} = \Sigma_{Env} \setminus$ Message events that are blocked by an active assumption scenario or disallowed by a corresponding GTR.
11: **if** $\Sigma_{ex} \neq \emptyset$ **then**
12: $\sigma_{ex} =$ *selectFrom*(Σ_{ex}); *perfromStep*(σ_{ex}); **goto** 1
13: **else**
14: *terminate*("*Assumption violation occurred*")

4.1.2. The ScenarioTools *simulation UI*

The ScenarioTools simulation integrates into the Eclipse debug environment (see Fig. 6). The user can select message events in a *Message Event Selection* view. The progress of active scenarios is highlighted in the SML editor. A graphical *Simulation Graph* visualizes the explored states and supports jumping back and forth in the execution. The active scenarios and the object model are displayed in the *Debug* view. The *Variables* view shows active scenario role bindings, scenario variables values, and object properties.

4.2. Realizability Checking via Formal Controller Synthesis

The controller synthesis feature of ScenarioTools allows us to compute a strategy for the system to react to any sequence of environment events in such a way that the specification is satisfied. A specification is *realizable* if such a strategy exists and *unrealizable* otherwise. The synthesis builds a state graph of all play-out executions, including the changing object models, and runs a game-solving algorithm on it [7]. If the specification is unrealizable, the synthesis produces a a *counter-strategy* of how the environment can force a violation of the specification.

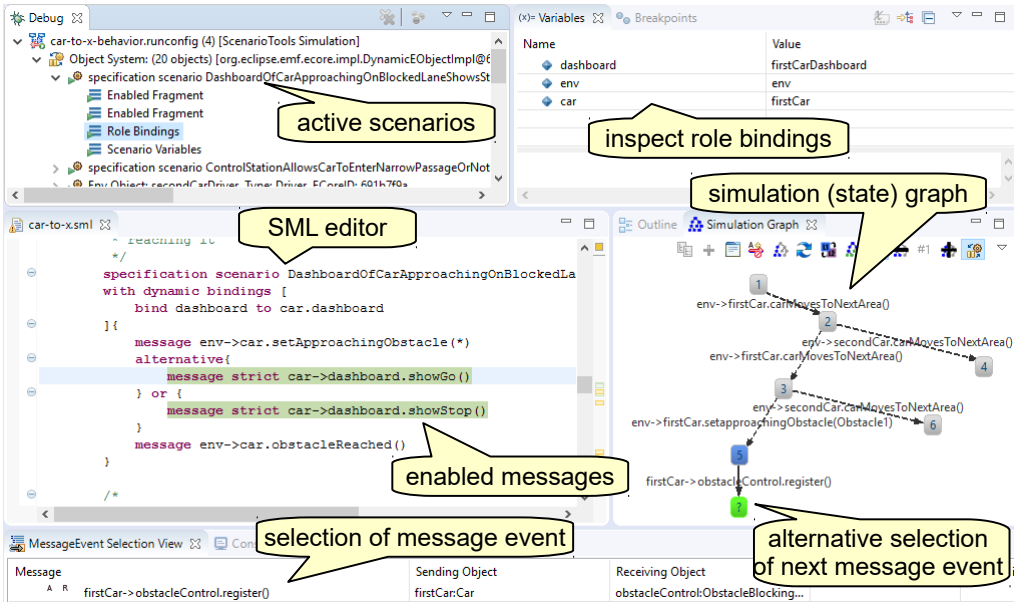

Figure 6: The ScenarioTools simulation perspective.

ScenarioTools supports the simulation of strategies and (counter-) strategies; the latter is helpful for understanding specification flaws.

Let us assume that in our example, the engineer forgot the assumption scenario DriverObeysStopSignal. The synthesis can find that now the software cannot avoid crashes of cars, which are forbidden by the scenario CarsMustNotCrash. Figure 7 shows how the user would explore the counter-strategy.

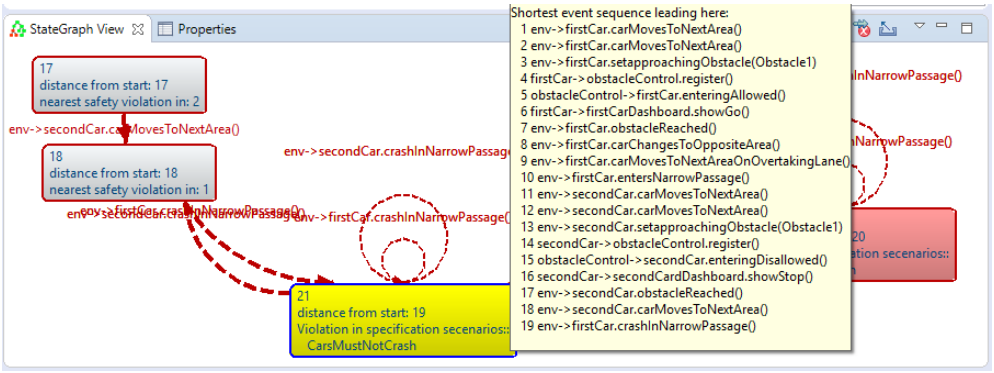

Figure 7: The ScenarioTools State Graph for exploring a counter-strategy.

5. Software Architecture

The execution and formal analysis capabilities are based on its execution engine that interprets an SML specification with an object model (cf. Fig. 2).

5.1. Runtime Model

The execution engine consists of a *runtime* model that reflects run-time concepts such as an *active scenario* or an active scenario's *role bindings*. The classes encapsulate the execution logic for the respective run-time concepts, for example the active scenario class implements a performStep function for progressing an active scenario for a given message event.

Figure 8 shows a simplified class diagram of the runtime model. We omit reference names where they are unnecessary. The model references elements from the run configuration model, the SML language model, and Ecore.

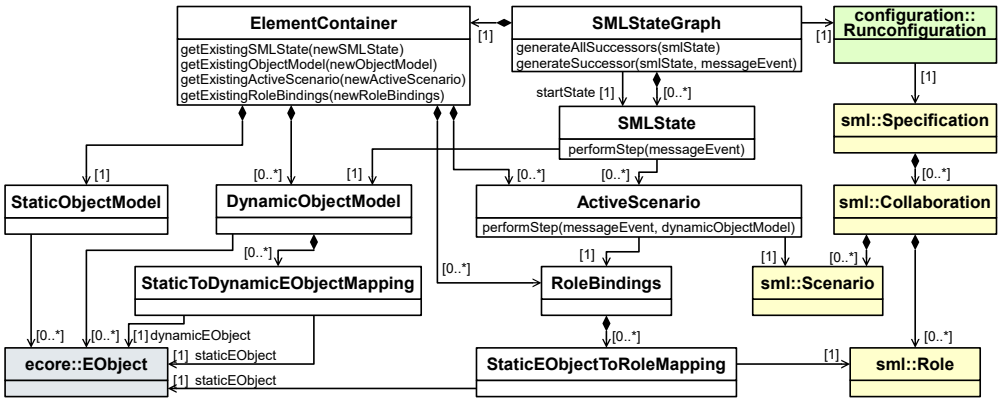

Figure 8: The main concepts of the ScenarioTools runtime model (simplified)

The root element of a runtime instance is a state graph that represents all executions of a run configuration. A state graph consists of states and transitions (omitted in the diagram); it provides the operation generateAllSuccessors, which, for a given state, explores all states that result from the execution of all possible message events in that state (cf. Alg. 1). The state graph also has a operation generateSuccessor, which explores only one successor state. The latter operation is used by the simulation environment.

The generation of a successor state works by copying a given state and applying the state's performStep operation. When building a state graph, however, identical states must not occur twice, which means that the generateSuccessor operation must first attempt to find an identical already explored

state, if it exists, and return it. Otherwise, it returns the newly explored state. More details on this state lookup procedure appear in Sect. 5.2.

A state references a set of active scenarios, which each reference a scenario from the specification. An active scenario references role bindings and enabled messages (not shown in the diagram) that define its state of progress. The role bindings map which objects the roles in the scenario are bound to.

A state furthermore references an object model. We maintain different variants of it: there is one *static* object model, which is a copy of the initial object model. In addition, there are *dynamic* object models, which are the object models that evolve from the initial object model when applying the message event side-effects to it. A dynamic object model also contains a mapping between its objects and the objects in the static object model.

This mapping reduces redundancy in the runtime model: two states can reference different dynamic object models, but can still reference the same active scenario, since its role bindings refer to the static object model.

Figure 9 shows a simplified instance of the runtime model for three states of the Car-to-X example. In state s1, has an active copy of the assumption scenario ApproachingObstacleOnBlockedLaneAssumption, due to a previous occurrence of env->car1.moveToNextArea(). In this active scenario, the non-spontaneous event env->car1.setApproachingObstacle(o1) is enabled. State s2 is the successor of state s1 after the occurrence of this event.

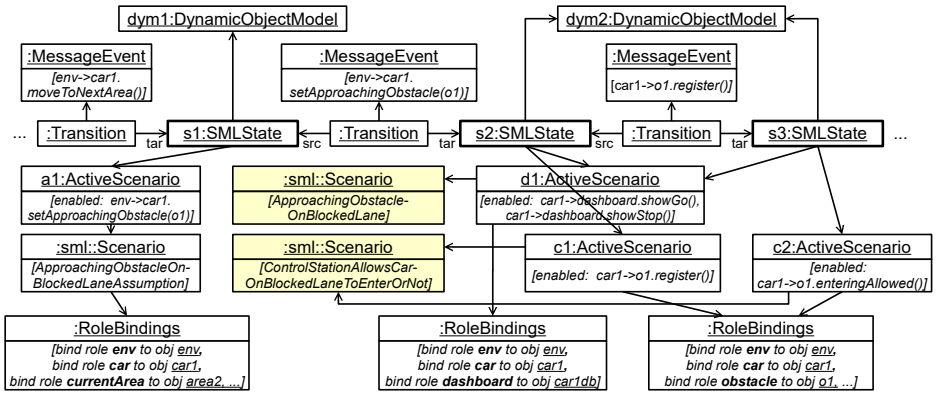

Figure 9: Runtime model instance for the execution of the car-to-x example (simplified)

In state s2, the assumption scenario ApproachingObstacleOnBlocked-LaneAssumption is no longer active, but instead the guarantee scenarios ApproachingObstacleOnBlockedLane and ControlStationAllowsCarOnBlocked-

LaneToEnterOrNot are activated. Also state s2 references another object model, because of the side-effect of the previous message event.

State s3 is the successor of s2 after the occurrence of car1->o1.register(). This event progresses the active copy of ControlStationAllowsCarOnBlocked-LaneToEnterOrNot (c2), but not the active copy of ApproachingObstacleOn-BlockedLane (d1), which is shared with state s2. The two active copies c1 and c2 share the same role bindings. Because car1->o1.register() has no side-effect, states s2, s3 reference the same object model.

5.2. State Lookup Procedure

This state lookup involves different lookup methods of the state graph's element container (Fig. 8), which uses hash tables for an efficient lookup.

Whenever the successor of a state is explored for a given message event, first, a copy of the state, its dynamic object model, its active scenarios, and their role bindings is created. Then we invoke the state's performStep operation for the given message event, which applies any side-effects on the dynamic object model, invokes the performStep operation of its active scenarios and creates new active scenarios or terminates others.

Next, we invoke the element container's lookup methods. First, we check whether a dynamic object model that is identical to the copied and changed one already exists. If so, we reset the state's reference to its dynamic object model to the existing one. The redundant copy will be garbage-collected. If an identical dynamic object model does not yet exist, the newly created dynamic object model is added to the element container's object model hash table. Second, the same procedure is applied to all of the active scenarios' role bindings, third, to the active scenarios, and, last, to the state itself.

6. Empirical Results

We show performance results for performing synthesis on variants of the Car-to-X example to give an impression of the capabilities of the tool.

The specification has 16 scenarios, which describe the cars' behavior when entering the narrow passage, and the behavior for avoiding a collision.

We considered 8 variants of the situation shown in Fig. 1, with one to four cars coming from left or right, see left column of Tab. 1. For example, *1-2* means one car coming from the left and two cars coming from the right.

We also tested with two variants of the specification, where the assumption scenario DriverObeysStopSignal is included or not (column 2 of Tab. 1).

If it is not included, the crashes cannot be avoided where cars approach each other, and thus the specification is unrealizable (column 3 of Tab. 1)[3].)

Cars	Obey STOP?	Realizable?	Explored States	Explored Transitions	Synthesis Time (sec.)
0-1	yes	yes	26	37	0.1
0-2	yes	yes	306	702	1.7
0-3	yes	yes	2224	6836	14.2
0-4	yes	yes	12280	46392	11.2
1-1	yes	yes	784	1748	4.0
1-2	yes	yes	10100	29152	61.0
1-3	yes	yes	80230	282084	626.3
2-2	yes	yes	-	-	-
0-1	no	yes	26	37	0.1
0-2	no	yes	331	782	2.4
0-3	no	yes	2939	9238	19.3
0-4	no	yes	20441	77880	166.1
1-1	no	no	324	795	1.5
1-2	no	no	3879	12973	25.1
1-3	no	no	34703	141310	331.4
2-2	no	no	424	877	3.0

Table 1: Experimental evaluation of realizability checking different car-to-x examples. The last three columns show the number of explored states and transitions as well as the checking times. Measurements were taken on a laptop with 8GB RAM, an Intel Core i7-2720QM processor at 2,2 Ghz, running Windows 10 and Java 1.8.0_112 HotSpot 64 bit. The Eclipse version is 4.6.2, with EMF version 2.13.0 and Henshin version 1.5.0.

We could not check the realizable case 2-2, because the memory limit of 8 GB was exceeded. The tool performs well for a couple of ten thousand states. This may seem few—but, note that each state is a complex data structure, including the object model, and state exploration is a complex operation, with event selection that may involve multiple GTR matching attempts.

The ScenarioTools website and repository contain examples of other reactive systems: a safety-critical high-voltage coupling system for electrical cars [21], a vacuum cleaner robot, an elevator, or a production robot.

[3] The Car-to-X project is located at `https://bitbucket.org/jgreenyer/scenariotools-sml-examples/src/master/org.scenariotools.sml.henshin.example.models2016v2.car-to-x/`, the different instance models tested are located in the subfolder `/instance-models/OneStreetWithObstacle`. The synthesis is performed based on the On-The-Fly Büchi game solving algorithm (on a runconfig file, right-click→ScenarioTools Synthesis→On-The-Fly Büchi.

7. Conclusions

We introduced ScenarioTools, an Eclipse-based tool suite for formal scenario-based modeling and analysis of reactive systems. Especially, ScenarioTools targets the modeling of cyber-physical systems that consist of multiple interacting components and have a dynamic system structure. For modeling these systems, ScenarioTools offers the means for modeling not only the desired system behavior, but also for modeling environment assumptions and modeling changes in the system structure. The resulting specifications are executable, and can be analyzed by simulation and formal controller synthesis. We demonstrated the main features of the modeling language and the modeling and analysis tools by the help of a running example.

Acknowledgements

This work is funded by grant no. 1258 of the German-Israeli Foundation for Scientific Research and Development (GIF).

References

[1] D. Harel, A. Pnueli, On the development of reactive systems, in: K. R. Apt (Ed.), Logics and Models of Concurrent Systems, Springer Berlin Heidelberg, 1985, pp. 477–498.

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

[3] C. Brenner, J. Greenyer, V. Panzica La Manna, The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions, in: Proc. 12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013), Vol. 58, EASST, 2013.

[4] D. Harel, R. Marelly, Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach, SoSyM 2 (2003) 82–107.

[5] Henshin website, `https://www.eclipse.org/henshin/`.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place EMF model transformations, in: Proc. 13th Int. Conf. on Model Driven Engineering Languages and Systems, 2010, pp. 121–135.

[7] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, E. Gressi, Incrementally synthesizing controllers from scenario-based product line specifications, in: Proc. 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2013, 2013.

[8] J. Greenyer, D. Gritzner, G. Katz, A. Marron, Scenario-based modeling and synthesis for reactive systems with dynamic system structure in scenariotools, in: J. de Lara, P. J. Clarke, M. Sabetzadeh (Eds.), 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), Vol. 1725, CEUR, 2016, pp. 16–32.

[9] S. Winetzhammer, J. Greenyer, M. Tichy, Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems, in: System Analysis and Modeling: Models and Reusability, Vol. 8769 of LNCS, Springer, 2014, pp. 126–141.

[10] H. Liang, J. Dingel, Z. Diskin, A comparative survey of scenario-based to state-based model synthesis approaches, in: Proc. Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, SCESM '06, ACM, New York, NY, USA, 2006, pp. 5–12.

[11] W. Damm, D. Harel, LSCs: Breathing life into message sequence charts, in: Formal Methods in System Design, Vol. 19, 2001, pp. 45–80.

[12] D. Harel, H. Kugler, Synthesizing state-based object systems from LSC specifications, Foundations of Computer Science 13:1 (2002) 5–51.

[13] Y. Bontemps, P. Heymans, From Live Sequence Charts to State Machines and Back: A Guided Tour, IEEE Transactions on Software Engineering 31 (12) (2005) 999–1014.

[14] S. Uchitel, G. Brunet, M. Chechik, Synthesis of Partial Behavior Models from Properties and Scenarios, IEEE Transactions on Software Engineering 35 (3) (2009) 384–406. doi:10.1109/TSE.2008.107.

[15] K. G. Larsen, S. Li, B. Nielsen, S. Pusinskas, Scenario-based analysis and synthesis of real-time systems using Uppaal, in: Proc 13th Conf. on Design, Automation, and Test in Europe (DATE'10), 2010.

[16] D. Harel, I. Segall, Synthesis from scenario-based specifications, Journal of Computer and System Sciences 78 (3) (2012) 970 – 980. doi:10.1016/j.jcss.2011.08.008.

[17] D. Harel, S. Maoz, S. Szekely, D. Barkan, Playgo: Towards a comprehensive tool for scenario based programming, in: Proc Int. Conf. on Automated Software Engineering, ASE '10, ACM, New York, NY, USA, 2010, pp. 359–360.

[18] M. Gordon, D. Harel, Generating executable scenarios from natural language, in: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing: 10th Int. Conf., CICLing 2009, Mexico City, Mexico, March 1-7, 2009., Springer Berlin Heidelberg, 2009, pp. 456–467. doi:10.1007/978-3-642-00382-0_37.

[19] S. Maoz, Y. Sa'ar, Counter play-out: Executing unrealizable scenario-based specifications, in: 2013 35th Int. Conf. on Software Engineering (ICSE), IEEE, 2013, pp. 242–251. doi:10.1109/ICSE.2013.6606570.

[20] S. Maoz, Polymorphic scenario-based specification models: semantics and applications, Software and System Modeling 11 (3) (2012) 327–345.

[21] J. Greenyer, M. Haase, J. Marhenke, R. Bellmer, Evaluating a formal scenario-based method for the requirements analysis in automotive software engineering, in: Proc. 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, 2015.

Required Metadata

Current executable software version

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	1.0
S2	Permanent link to executables of this version	`http://scenariotools.org/downloads/update-site/`
S3	Legal Software License	Eclipse Public License - v1.0
S4	Computing platform/Operating System	Microsoft Windows, Mac OS X, Linux
S5	Installation requirements & dependencies	not applicable
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	`http://scenariotools.org/tutorials/`
S7	Support email for questions	`greenyer@inf.uni-hannover.de`, `daniel.gritzner@inf.uni-hannover.de`

Table 2: Software metadata (optional)

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.0
C2	Permanent link to code/repository used of this code version	`https://bitbucket.org/jgreenyer/scenariotools-sml/`, examples in `https://bitbucket.org/jgreenyer/scenariotools-sml-examples/`
C3	Legal Code License	Eclipse Public License - v1.0
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Java, Eclipse, EMF, Xtext, GEF, Henshin
C6	Compilation requirements, operating environments & dependencies	Java 8, Eclipse Modeling Tools 4.7. (Oxygen), EMF, Xtext 2.12, GEF, Henshin 1.5 (Henshin requires GMF 3.2.1 and M2Eclipse http://www.eclipse.org/m2e/, see developer setup documentation linked below)
C7	If available Link to developer documentation/manual	`http://scenariotools.org/downloads/download/`
C8	Support email for questions	`greenyer@inf.uni-hannover.de`, `dgritzner@inf.uni-hannover.de`

Table 3: Code metadata (mandatory)