ScenarioTools real-time play-out for test sequence validation in an automotive case study

(article starts on next page)

Electronic Communications of the EASST
Volume 67 (2014)

Proceedings of the
13th International Workshop on Graph Transformation
and Visual Modeling Techniques
(GTVMT 2014)

ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study

Christian Brenner, Joel Greenyer, Jörg Holtmann,
Grischa Liebel, Gerald Stieglbauer, Matthias Tichy

14 pages

Guest Editors: Frank Hermann, Stefan Sauer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study

Christian Brenner[1*], **Joel Greenyer**[2], **Jörg Holtmann**[3], **Grischa Liebel**[4], **Gerald Stieglbauer**[5], **Matthias Tichy**[4]

[1] cbr@uni-paderborn.de
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, Germany.

[2] greenyer@inf.uni-hannover.de
Software Engineering Group,
Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany.

[3] joerg.holtmann@ipt.fraunhofer.de
Software Engineering, Project Group Mechatronic Systems Design, Fraunhofer
Institute for Production Technology IPT, Zukunftsmeile 1, 33102 Paderborn, Germany.

[4] grischa@chalmers.se, matthias.tichy@cse.gu.se
Software Engineering Division, Department of Computer Science and Engineering,
Chalmers | University of Gothenburg, 412 96 Gothenburg, Sweden.

[5] gerald.stieglbauer@avl.com
AVL List GmbH, Hans-List-Platz 1, 8020 Graz, Austria

Abstract:

In many areas, such as automotive, healthcare, or production, we find software-intensive systems with complex real-time requirements. To efficiently ensure the quality of these systems, engineers require automated tools for the validation of the requirements throughout the development. This, however, requires that the requirements are specified in an analyzable way. We propose modeling the specification using Modal Sequence Diagrams (MSDs), which express what a system may, must, or must not do in certain situations. MSDs can be executed via the play-out algorithm to investigate the behavior emerging from the interplay of multiple scenarios; we can also test if traces of the final product satisfy all scenarios. In this paper, we present the first tool supporting the play-out of MSDs with real-time constraints. As a case study, we modeled the requirements on gear shifts in an upcoming standard on vehicle testing and use our tool to validate externally generated gear shift sequences.

Keywords: scenario-based specification, reactive systems, embedded systems, automotive, simulation, validation, testing

[*] supported by the International Graduate School "Dynamic Intelligent Systems"

1 Introduction

In many areas, such as automotive, healthcare, or production, we find software-intensive systems that consist of interacting components and must fulfill complex requirements. These systems must often control complex physical/mechanical processes with critical real-time requirements.

To efficiently ensure the quality of these systems, engineers require automated tools to validate that the final product meets the requirements. Ideally, the engineers should be able to validate already during the design phase whether the design meets the requirements, or—even before attempting to explore possible solutions—to check whether the requirements are inconsistent.

Such automated validation techniques, however, imply that the real-time requirements are rigorously formalized. We advertise a scenario-based approach that allows engineers to specify what a system may, must, or must not do in a certain situation. This approach is in line with how engineers typically specify use cases with positive/negative scenarios. In particular, we propose to use Modal Sequence Diagrams (MSDs) [HM08], a recent variant of Live Sequence Charts (LSCs) [DH01] to model the specification. MSDs/LSCs are sequence diagrams that, by different modalities assigned to messages and conditions, allow us to precisely describe scenarios with liveness (something good must happen) and safety (something bad must not happen) properties.

One key advantage of MSDs/LSCs is that they can be executed with the play-out algorithm, which allows engineers and other stakeholders to understand the behavior emerging from the interplay of the scenarios [HM03]. An MSD/LSC specification can also be used in the testing of an implementation, be it the final product or a previous software-/hardware-in-the-loop setup. Traces of tests runs can be executed together with the scenarios to check if all safety and liveness requirements are met, i.e., the specifications can be operationalized to act as a test oracle.

Timed play-out, along with LSCs extensions to express time constraints, were already proposed by Harel and Marelly [HM02b]. However, this approach supports only discrete time, which is inconvenient when modeling critical real-time systems. There are other approaches, which support validation of traces and model-checking against real-time scenario specifications [LK01, LLNP09]. These approaches, however, do not support interactive play-out.

The contribution of this paper is twofold. First, we present the first tool realization of a real-time play-out tool environment, based on the ScenarioTools tool suite [BGP13]. We represent time constraints in MSDs through resets of and conditions regarding real-valued clock variables, as proposed in [LLNP09]. The tool internally represents the time in the form of zones, which represent possible intervals for clock values. From the perspective of a user performing a step-by-step play-out, this is also convenient: if at certain points during play-out, choices of waiting different amounts of time influence how the execution progresses, e.g., because time conditions will evaluate differently, then we only show these relevant waiting times to the user.

As our second contribution, we show the practical applicability of the approach: First, we show how we could successfully model the requirements on gear shifts in an upcoming standard on vehicle test procedures. Second, we show how we then use ScenarioTools to validate gear shift sequences that can be derived from test cycles generated by a third-party tool that implements this standard.

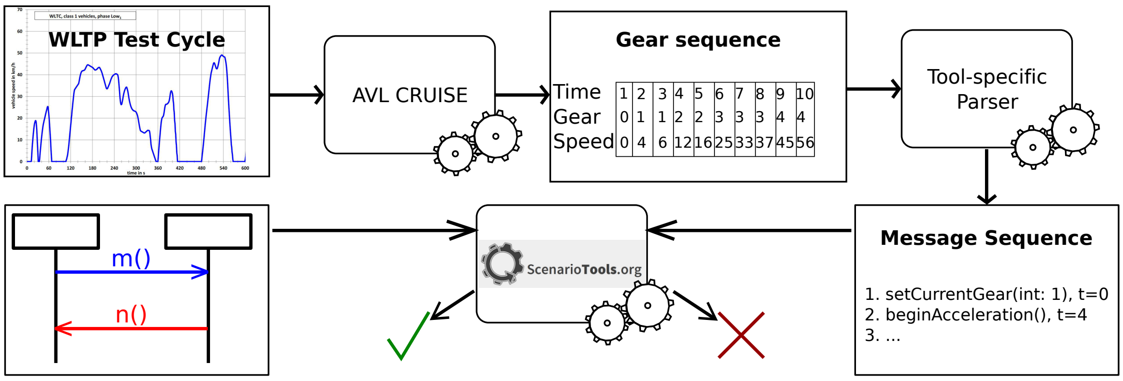
This paper is structured as follows. In Sect. 2, we explain our case study and give background on MSDs in Sect. 3. In Sect. 4 we present our realization of timed play-out. Last, we summarize the implementation of ScenarioTools in Sect. 5 and present related work in Sect. 6.

2 Example: WLTP

In the course of this paper, we use the Worldwide harmonized Light vehicles Test Procedures (WLTP) [Uni14] as a practical example for the application of SCENARIOTOOLS with real-time constraints. The WLTP are currently developed by the United Nations Economic Division and aim at "providing a worldwide harmonized method to determine the levels of gaseous and particulate emissions, CO_2 emissions, fuel consumption, electric energy consumption and electric range from light-duty vehicles in a repeatable and reproducible manner designed to be representative of real world vehicle operation." [Uni14].

Among other things, the WLTP define different test cycles for light vehicles. A test cycle is defined as the vehicle speed at every point in time during the test. Annex II of the WLTP draft describes requirements for the gear selection during these test cycles. For a particular vehicle, there exist tools such as AVL CRUISE [avl] that support optimizing gear-shift sequences over the test cycle time. These, however, have to be validated to conform to the upcoming WLTP.

Our approach is to model the WLTP requirements for gear shifts using MSDs. We then validate the generated gear-shift sequences against the modeled MSD specification. To do this, we parse the generated timed gear shift sequences and create an event sequence where, in addition to gear shifts, we derive events from the speed information that mark the beginning and end of acceleration and deceleration phases in the test cycle. The resulting timed event sequence is then fed into SCENARIOTOOLS and validated against the MSD specification. The overall tool chain for validation of gear shift sequences according to the WLTP is depicted in Fig. 1. In this paper, we focus on the lower part of the figure with respect to the modeled MSDs, ScenarioTools and the validation of message sequences; the integration with AVL Cruise is future work.



Time	1	2	3	4	5	6	7	8	9	10
Gear	0	1	1	2	2	3	3	3	4	4
Speed	0	4	6	12	16	25	33	37	45	56

1. setCurrentGear(int: 1), t=0
2. beginAcceleration(), t=4
3. ...

Figure 1: Possible Tool Chain for Validation of WLTP Gear Shift Sequences

While we have modeled all but one of the gear selection requirements consisting of seven requirements, we only consider two of the requirements for illustration in this paper:
"(b) Gears shall not be skipped during acceleration phases. Gears used during accelerations and decelerations must be used for a period of at least three seconds.
(e) If a gear i is used for a time sequence of 1 to 5 s and the gear before this sequence is the same as the gear after this sequence, e.g. $i-1$*, the gear use for this sequence shall be corrected to* $i-1$*."*

3 Foundations

An MSD specification consists of a set of MSDs. An MSD can be *existential* or *universal* [HM03]. Existential MSDs describe sequences of events that must be possible to occur in a system; universal MSDs describe properties that must hold for executions of the system. We focus on universal MSDs in this paper, since all requirements in our case study can be formalized by universal MSDs.

Each lifeline in an MSD represents an object in an *object system*; an object can be an *environment object* or a *system object*. The set of system objects is called the *system*; the set of environment objects is called the *environment*.

3.1 Basic MSD semantics

The objects can interchange *messages*. A message has one sending and one receiving object and refers to an operation of the receiving object. The *name* of the operation is also that of the message. Here we consider only *synchronous* messages where the sending and the receiving of the message together form a single event, which we call *message event* or simply *event*.

A message in an MSD, also called a *diagram message*, has a name and a sending and a receiving lifeline. A lifeline represents exactly one object in the object system. The messages in an MSD have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*; the execution kind can be either *monitored* or *executed*. In our figures, we label the messages accordingly with (c/m), (c/e), (h/m), and (h/e). Also, hot messages are colored red; cold messages are colored blue. Executed messages have a solid arrow, monitored messages have a dashed arrow.

Intuitively, an MSD progresses as messages occur in the system as described in the MSD. If the progress reaches a message that is monitored, this message may or may not occur. If the message is executed, the message must eventually occur (liveness). If the message is hot, no message must occur (safety) that the scenario specifies to occur only earlier or later. If the message is cold and a message occurs that is specified to occur earlier or later, this "aborts" the progress of the MSD. Messages that are not specified in the MSD are ignored, i.e., they do not influence the progress of the MSD and the MSD does not impose requirements on them.

More specifically, the semantics of the messages is as follows: An event can be *unified* with a message in an MSD iff the event name equals the message name and the sending and the receiving objects are represented by the sending resp. receiving lifelines of the message. When an event occurs in the system that can be unified with the first message in an MSD, an *active MSD* is created. As further events occur that can be unified with the subsequent messages in the diagram, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the locations of the messages that were unified with the message events. If the cut reaches the end of an active MSD, the active MSD is terminated.

If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*. If a hot message is enabled, the cut is also *hot*. Otherwise the cut is *cold*. If an executed message is enabled, the cut is also *executed*. Otherwise the cut is *monitored*. An enabled executed message is called an *active* message. A *violation* occurs if a message event occurs that can be unified with a message in the MSD that is not currently enabled. If the cut is hot, it is a *safety violation*; if the cut is cold, it is called a *cold violation*. Safety violations must never happen, while cold violations may occur and result in terminating the active MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if it does not. Instead, an active MSD is not required to progress in a monitored cut.

We consider reactive systems that can, in principle, run infinitely long. We call an infinite sequence of message events a *run*. A run satisfies an MSD specification if it leads to no safety nor liveness violations in any MSD of the specification. In our case study, we consider only finite traces and define that a finite trace satisfies an MSD specification if there is no safety violation in any MSD. We discuss examples in Sect. 5.

3.2 Parametrized messages, assignments, conditions, and other constructs

An MSD can also contain parametrized assignments, conditions over variables, and constructs for expressing alternative continuations of scenarios or forbidden events. In the following, we will describe these constructs in a by-example fashion.

The two WLTP gear shift requirements depicted in the previous section can be modeled using MSDs. The first sentence of Requirement *(b)*, "Gears shall not be skipped during acceleration phases", is modeled using the two MSDs depicted in Figure 2. Roughly, the MSD NextGear-AfterAccPhaseBegins specifies that after beginning an acceleration phase, the next gear must be one gear lower or higher than the gear selected at the beginning of the acceleration phase, unless the acceleration phase ends before selecting another gear. The MSD NextGearDuringAcc says that if a new gear was selected during an acceleration phase, the next gear must be one gear lower or higher than that gear, again unless the acceleration phase ends before selecting another gear.

We modeled an underlying object system with the objects gs:GearSelector, gc:GearController and env:Environment. The gear selector gs is responsible for choosing an appropriate gear by sending `setActiveGear`-messages to the gear controller gc. (In practice, this would be a component of a test stand or it could even be a human driver.) Allowed gears depend, among other things, on events from the environment env object, which informs the gear selector about the beginning of acceleration and deceleration phases.

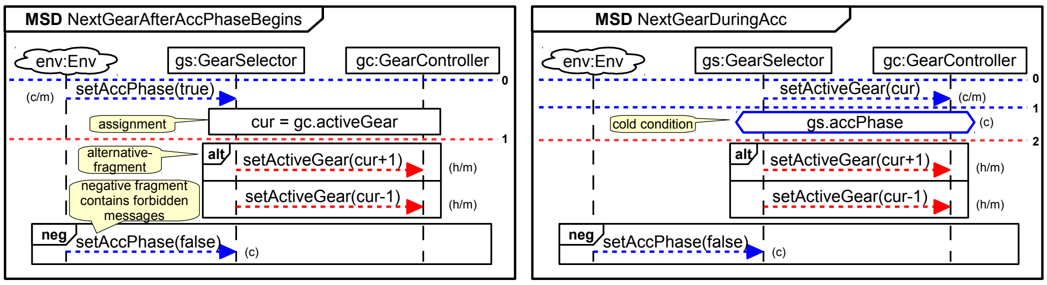


Figure 2: The MSDs NextGearAfterAccPhaseBegins and NextGearDuringAcc (Requirement *(b)*)

The MSD NextGearAfterAccPhaseBegins is activated when the acceleration phase begins `setAccPhase(true)`. As we see, messages can also have *parameters* of certain types (here: Boolean). Message events then always carry corresponding parameter values. In an MSD, either concrete literal values can be specified for message parameters, as for the first message `setActiveGear(true)`. Alternatively, we can specify a variable that, if unbound, does not prescribe any concrete value. If bound, however, we can use the variable in OCL expressions to specify concrete values.

A parametrized message event can be *unified* with a diagram message either if the value carried by the message event matches the value specified for the diagram message or if the diagram message specifies an unbound variable. In the latter case, the unbound variable is bound to the parameter value carried by the message event it was unified with. For example, the MSD NextGearDuringAcc is activated if a message `setActiveGear` occurs with an arbitrary parameter value. The value (here: number of the activated gear) is then bound to the variable cur. As then the variable is bound, it can be used to specify that the next gear must be one gear higher or lower (`setActiveGear(cur-1)` or `setActiveGear(cur+1)`).

Variables can also be bound using *assignments*. Assignments are represented in MSDs by rectangles that cover one or multiple lifelines and contain expressions in the form `<var> = <expr>` where `<var>` is a variable name and `<expr>` is an OCL expression. The MSD NextGearAfterAccPhaseBegins for example contains such an assignment. An assignment is *enabled* if the cut is in front of the assignment on all lifelines it covers. If enabled, the expression is immediately evaluated and the value is assigned to the variable. This variable can again be used to specify that the next gear must be one gear higher or lower. Parametrized messages of the form `set<attr>` assign a value to the attribute `<attr>` of the receiving object, provided the parameter and attribute types match. Here, the gear controller object gc has an attribute activeGear that stores the active gear.

In addition to assignments, MSDs can contain hot or cold *conditions*, which are represented as hexagons that cover one or more lifelines. In our figures, we label the temperature (c) or (h); in addition, cold conditions are colored blue and hot conditions are colored red. Conditions can contain OCL expressions that evaluate to a Boolean value, possibly involving bound variables. Conditions, if enabled, are evaluated immediately. If the expression of a hot or cold condition evaluates to true, the cut progresses beyond the condition. If the expression evaluates to false and the condition is cold, the active MSD terminates (cold violation). If a hot condition evaluates to false, this is a safety violation.

Both MSDs in Fig. 2 contain an *alternative fragment*, which specifies two alternative continuations of the scenario. In this case, the alternative is a non-deterministic choice, allowing either shifting to a gear one lower or higher than the gear selected before. A cut in front of the alternative fragment means that both first messages in the nested fragments are enabled. If a corresponding event occurs, the cut progresses only inside the respective nested fragment.

Furthermore, an MSD can contain *forbidden messages*. In SCENARIOTOOLS, forbidden message are specified in a negate fragment at the end of an MSD. The negate fragment is not part of the MSD that is reachable by the cut, i.e., an active MSD terminates if the cut reaches a negate fragment. It is a cold or safety violation if a message event occurs that cannot be unified with an enabled message, but can be unified with a cold resp. hot forbidden message. If a negate fragment contains multiple messages, they are all considered individual forbidden messages.

3.3 Real-time constraints

Time constraints can be modeled in MSDs by referring to *clock variables*. Clock variables are a concept adopted from timed automata [AD94]; they are real-value variables that increase synchronously and linearly with time. In MSDs, clocks can be reset to zero in assignments and we can also formulate conditions over clock variables.

As shown in Fig. 3, clock resets and conditions over clock variables, called *time conditions*, have an additional hour-glass icon. Intuitively, the MSD MinimumGearUsePeriod models the second sentence in Requirement *(b)*, “Gears used during accelerations and decelerations must be used for a period of at least three seconds.”. The MSD becomes active whenever a gear shift message occurs. If the gear shift occurs during an acceleration or deceleration phase, a clock is reset. A hot time condition says that every gear shift during the following three seconds is not allowed. Similarly to the previous MSDs, ending the acceleration or deceleration phase will also cause a cold violation.

Timed conditions can contain expressions of the form $x \bowtie expr$ where x is a clock variable, *expr* is an expression evaluating to an integer value, and $\bowtie$ is an operator $<, \leq, >, \geq$. Cold time conditions are treated like cold untimed conditions. For hot timed conditions, we distinguish *minimal delays* ($\bowtie \in \{>, \geq\}$) and *maximal delays* ($\bowtie \in \{<, \leq\}$). If a minimal delay is enabled, but evaluates to false, the cut progresses as soon as it becomes true. Meanwhile the cut is hot, i.e., no message that is not currently enabled in the active MSD is allowed to occur. If a maximal delay is enabled and evaluates to false, this is a liveness violation of the MSD (because the MSD cannot progress).

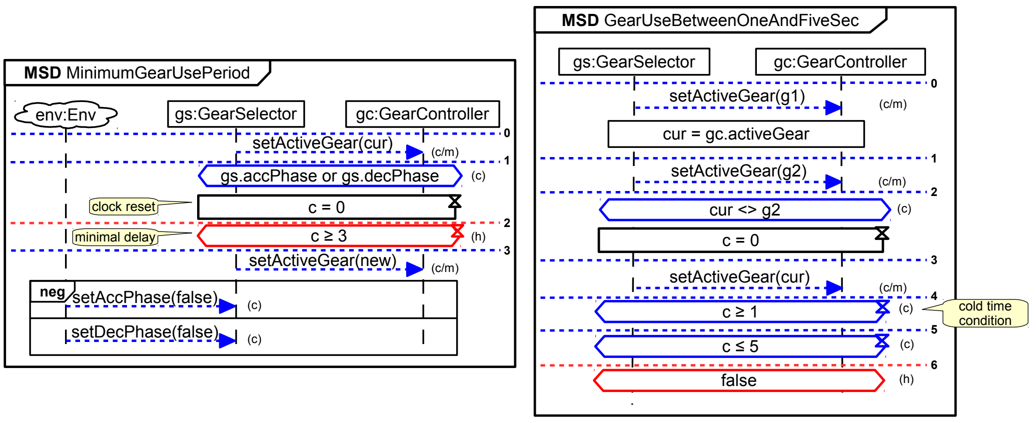


Figure 3: The MSDs MinimumGearUsePeriod and GearUseBetweenOneAndFiveSec (Req. *(e)*)

The MSD GearUseBetweenOneAndFiveSec models our interpretation of Requirement *(e)*. Note that Requirement *(e)* describes a requirement on valid gear sequences, but also suggests how to fix an invalid gear sequence. In MSD GearUseBetweenOneAndFiveSec, we only model the requirement. The MSD becomes active whenever a gear shift occurs. If the gear is shifted to a different gear and then back, a safety violation occurs if the gear in between was active one to five seconds. (In that case, the cut will progress to the hot `false` condition.)

3.4 The Play-Out Algorithm

Harel and Marelly defined an executable semantics for the LSCs, called the *play-out* algorithm [HM02a], that was later also defined for MSDs [MH06]. The basic principle is that if an environment event occurs and this results in one or more active MSDs with active system messages, then the algorithm non-deterministically (or by user interaction) chooses to send a corresponding message if that will not lead to a safety violation. The algorithm will repeat sending system messages until no active MSDs with active system messages remain. Then the algorithm will wait for the next environment event, and this process continues. (It is assumed that the system is always fast enough to send any finite number of messages before the next environment event occurs.) If the play-out algorithm reaches a state where there are active system messages, but they all lead to safety violations, the algorithm terminates unsuccessfully. If such a state is encountered, this shows engineers that the specification is either unrealizable or under-specified.

The play-out algorithm is implemented in the PLAY ENGINE [HM03] and the PLAYGO tool [Pla]. For more information on the SCENARIOTOOLS implementation of play-out, we refer to last year's GT-VMT publication [BGP13].

SCENARIOTOOLS supports the play-out and also the step-by-step simulation of message events and an automated validation of given event sequences against an MSD specification. We use the latter feature in our case study as explained in Sect. 2. The functionality is based on a common *runtime* package. In the next section, we provide details of how this runtime is extended to support real-time MSDs.

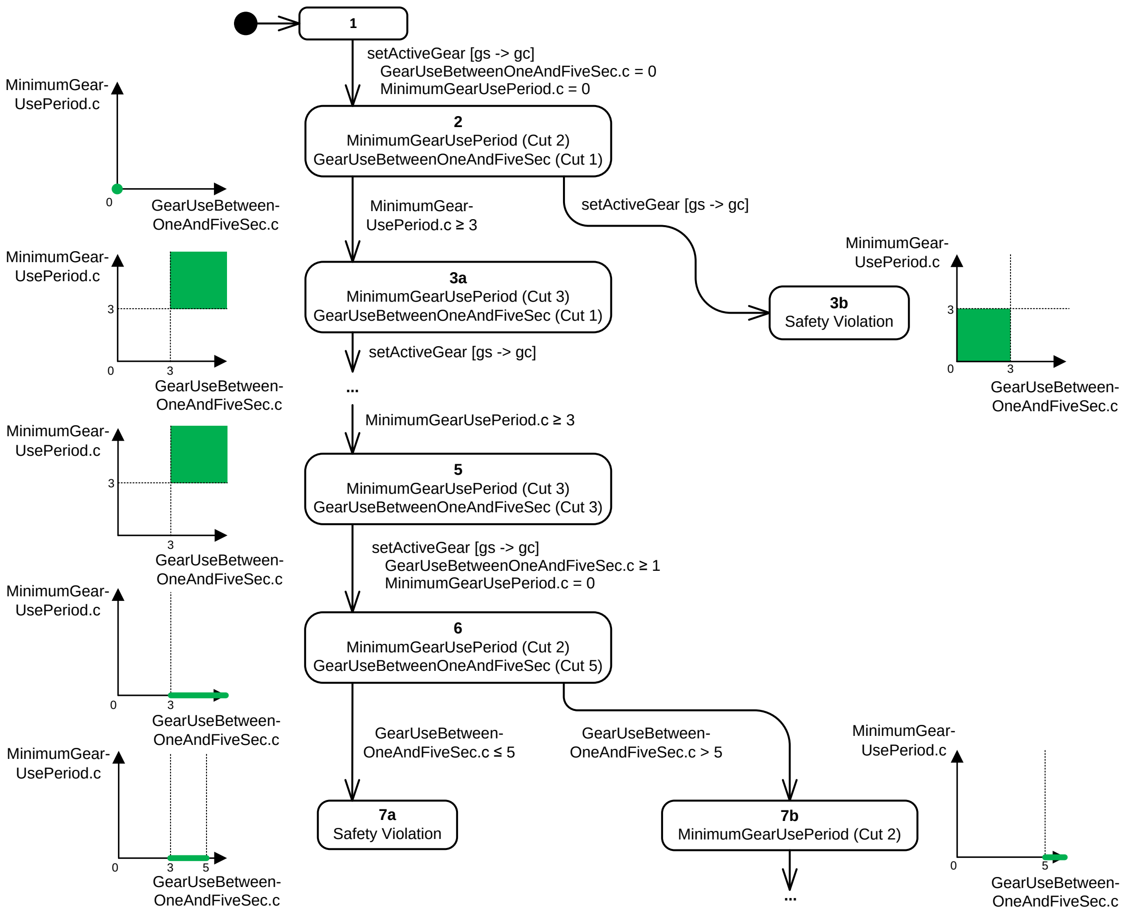
4 Timed Simulation

Real-time systems can behave differently depending on the passing of time. In the context of MSDs, this means that different sets of messages can be allowed, forbidden, or mandatory. However, the systems we consider do not change their behavior arbitrarily, but rather define the same behavior for some interval of time. This allows us to represent time symbolically instead of considering concrete clock values individually. For this, we use the concept of clock zones [Dil90] as known from timed automata [AD94].

The task of the SCENARIOTOOLS runtime is, on the one hand, to determine the currently possible events that can change the state of the MSD specification. On the other hand, the runtime computes the successor state for the event selected by the user (or by an algorithm) to be executed. In case of the untimed SCENARIOTOOLS runtime, the events are message events. In the timed SCENARIOTOOLS runtime, there can also be events that correspond to time conditions.

The *state* during the execution of an untimed MSD specification is defined by the current cuts in all active MSDs and the values of all variables. To support real-time behavior, the notion of a state must be extended. In a real-time setting, a state is also defined by its clock zone, which models the possible values of clocks within the state. There can also be several states that only differ in their clock zone. A state, therefore, can actually be the symbolic representation of infinitely many states with concrete clock values (since we assume a continuous time domain). This is analogous to the definition of symbolic states in the zone graph of a timed automaton [Dil90, BY04].

Figure 4 depicts an excerpt of the timed runtime state space induced by the MSDs MinimumGearUsePeriod and GearUseBetweenOneAndFiveSec (cf. Fig. 3) in a notation similar to zone graphs. Each state either lists the active MSDs including their current cuts or declares the occurrence of a safety violation. The corresponding clock zone for each state is visualized by means of time axes of the clocks of MinimumGearUsePeriod and GearUseBetweenOneAndFiveSec, respectively. Initially (in state 1), no active MSDs and no clocks exist. When the message setActiveGear is sent, both MSDs are activated. On activation, the clocks of both MSDs are initialized with 0 (cf. clock zone of state 2). We will explain further steps of the example along with the following explanation of the timed runtime.



1
setActiveGear [gs -> gc]
GearUseBetweenOneAndFiveSec.c = 0
MinimumGearUsePeriod.c = 0
2
MinimumGearUsePeriod (Cut 2)
GearUseBetweenOneAndFiveSec (Cut 1)
MinimumGear-UsePeriod.c ≥ 3
setActiveGear [gs -> gc]
3a
MinimumGearUsePeriod (Cut 3)
GearUseBetweenOneAndFiveSec (Cut 1)
3b
Safety Violation
setActiveGear [gs -> gc]
...
MinimumGearUsePeriod.c ≥ 3
5
MinimumGearUsePeriod (Cut 3)
GearUseBetweenOneAndFiveSec (Cut 3)
setActiveGear [gs -> gc]
GearUseBetweenOneAndFiveSec.c ≥ 1
MinimumGearUsePeriod.c = 0
6
MinimumGearUsePeriod (Cut 2)
GearUseBetweenOneAndFiveSec (Cut 5)
GearUseBetween-OneAndFiveSec.c ≤ 5
GearUseBetween-OneAndFiveSec.c > 5
7a
Safety Violation
7b
MinimumGearUsePeriod (Cut 2)
...
MinimumGear-UsePeriod.c
GearUseBetween-OneAndFiveSec.c
0
3
5

Figure 4: Zone Graph Excerpt for State Space of the MSDs in Fig. 3

Transitions in a timed setting can be labeled with message events, just like in the untimed setting. However, the progressing of enabled messages in active MSDs can be followed by the immediate execution of subsequently enabled clock resets, the evaluation of subsequently enabled time conditions, or a combination thereof. These resets and conditions thus influence the shape of the clock zone of the successor state. Moreover, if, following a message event, conditions are enabled that in the current clock zone can evaluate to either true or false, then there can be also multiple *different* successor states, with different zones, following the *same* message event. This corresponds to the notion of sending a message before or after a certain condition holds.

All these cases occur in our example and we explain them in the following:

In state 2: The MSD MinimumGearUsePeriod (cf. Fig. 3) is in cut 2 before the hot time condition $c \geq 3$. The clocks in this state have both been newly created and are initially zero. Therefore, the condition is not yet fulfilled. Waiting until $c \geq 3$ will make the MSD progress to cut 3. This corresponds to the transition from state 2 to state 3a (see Fig. 4).

For minimal delays (hot conditions with lower bounds), like in this case, we compute the zone of the successor state (here: state 3a) based on the zone of the current state (here: state 2) as follows. First, we model the passing of time by removing the upper bounds of all clocks. Then, we intersect the resulting zone with the condition (here: $c \geq 3$). As both clocks have been created at the same time and progress uniformly, their value is equal. Therefore, although $c \geq 3$ only refers to MinimumGearUsePeriod.c, both clocks are ≥ 3 now.

Alternatively, the message event `setActiveGear` can occur in state 2 before $c \geq 3$ becomes true. As cut 2 is hot, this causes a safety violation (state 3b). We compute the zone of this state as for state 3a, except that we intersect the current zone with the *negated* condition, i.e. $c < 3$.

In state 5: In both MSDs in Fig. 3, the message setActiveGear is enabled, corresponding to the transition to state 6. When the corresponding message event occurs, the MSD MinimumGearUsePeriod is terminated and created again (in cut 1), because `setActiveGear` is both the final and the initial message of the MSD. The MSD GearUseBetweenOneAndFiveSec progresses to cut 4. As time may pass arbitrarily before message events, the upper bounds of all clocks are removed for computing the clock zone of the successor state (here they are already unbounded).

After handling the message event, any subsequent clock resets and untimed conditions are immediately processed. Now we have to distinguish two kinds of time conditions. There can be conditions that will be either true or false for all clock values in the current clock zone, and there can be time conditions that evaluate to true for some clock values of the clock zone and to false for others. We call the former *decided time conditions* and the latter *undecided time conditions*. In state 5, the cold condition $c \geq 1$ in GearUseBetweenOneAndFiveSec is a decided time condition, while the cold condition $c \leq 5$ is an undecided time condition.

The condition $c \geq 1$ is decided, because in the current clock zone we have GearUseBetweenOneAndFiveSec.c ≥ 3. The condition $c \leq 5$ is undecided, because it is fulfilled for $3 \leq$ GearUseBetweenOneAndFiveSec.c ≤ 5, but unfulfilled for GearUseBetweenOneAndFiveSec.c > 5.

The runtime processes decided time conditions immediately, but does not process undecided time conditions. Therefore, from state 5 to state 6, the cut of MSD GearUseBetweenOneAndFiveSec advances to cut 5, but not yet any further. If there are undecided time conditions, different decisions can be made—either by the user or by another component. The decision to be made is whether the condition should evaluate to true or false, which implies retrospectively when the message `setActiveGear` was sent.

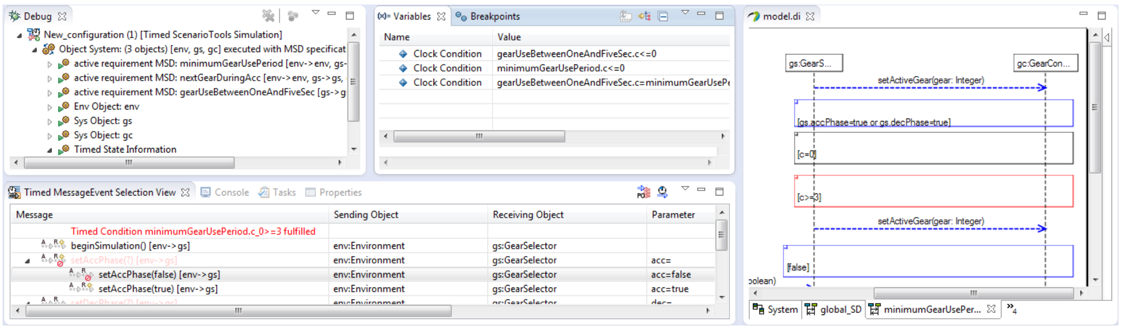
In state 6: The runtime creates two outgoing transitions representing the choice mentioned above. The clock zone of the successor state is created based on this decision and the current clock zone. For the case of a fulfilled condition, the clock zone is intersected with the condition, here resulting in $3 \leq$ GearUseBetweenOneAndFiveSec.c ≤ 5 in state 7a. For the opposite case, the clock zone is intersected with the *negation* of the condition, here resulting in GearUseBetweenOneAndFiveSec.c > 5 in state 7b. In both cases, the cut is progressed accordingly. In state 7a, the condition is now always fulfilled and the MSD GearUseBetweenOneAndFiveSec advances to cut 6, yielding a safety violation. In state 7b, the condition is always unfulfilled which results in a cold violation, terminating the MSD GearUseBetweenOneAndFiveSec.

The runtime handles clock resets by setting the value of the affected clock to zero in the zone of the successor state. This is done, for example, when processing the transition from state 5 to state 6 in MinimumGearUsePeriod, where MinimumGearUsePeriod.c is reset.

5 Implementation

SCENARIOTOOLS consists of several Eclipse plug-ins that are based on a common runtime logic [BGP13]. We implemented our timed extension as presented in Sect. 4 by creating a specialized timed version of this runtime plug-in. Furthermore, we created a timed simulation plug-in and added an option to automatically play back a number of message events specified in a csv file in which every message event is supplied with a time stamp. Using this play-back, we can validate the gear sequences against the MSDs formalizing the WLTP requirements as explained in Sect. 2.

For efficiently representing the clock zones in the timed runtime and for performing basic operations on them, we are using a Java implementation of the DBM library described in [BY04]. This implementation is an extension of the framework described in [EH11].



```
New_configuration (1) [Timed ScenarioTools Simulation]
  Object System: (3 objects) [env, gs, gc] executed with MSD specificat
    active requirement MSD: minimumGearUsePeriod [env->env, gs-
    active requirement MSD: nextGearDuringAcc [env->env, gs->gs,
    active requirement MSD: gearUseBetweenOneAndFiveSec [gs->g
    Env Object: env
    Sys Object: gs
    Sys Object: gc
    Timed State Information
```

Name	Value
Clock Condition	gearUseBetweenOneAndFiveSec.c<=0
Clock Condition	minimumGearUsePeriod.c<=0
Clock Condition	gearUseBetweenOneAndFiveSec.c=minimumGearUsePe

Message	Sending Object	Receiving Object	Parameter
Timed Condition minimumGearUsePeriod.c_0>=3 fulfilled			
beginSimulation() [env->gs]	env:Environment	gs:GearSelector	
setAccPhase(?) [env->gs]	env:Environment	gs:GearSelector	acc=
setAccPhase(false) [env->gs]	env:Environment	gs:GearSelector	acc=false
setAccPhase(true) [env->gs]	env:Environment	gs:GearSelector	acc=true
setDecPhase(?) [env->gs]	env:Environment	gs:GearSelector	dec=

Figure 5: A Screenshot of the Timed Simulation in SCENARIOTOOLS

Fig. 5 shows a screenshot of the timed simulation's user interface in the SCENARIOTOOLS perspective in Eclipse. It shows the currently active MSDs and the objects in the system (Debug view), the equations that define the current clock zone (Variables view), and the possible next events to evaluate, including time events (Timed MessageEvent Selection View). On the right hand side, a view of the UML editor Papyrus shows one of the MSDs involved in the simulation. We extended Papyrus for visualizing message temperatures and execution kinds.

The following example message sequence can be successfully simulated and is thus a valid gear sequence according to the WLTP:

1. *gs, gc.setActiveGear(int: 1),t=1*
2. *env, gs.setAccPhase(boolean: true),t=2*
3. *gs, gc.setActiveGear(int: 2),t=5*
4. *gs, gc.setActiveGear(int: 3),t=9*
5. *env, gs.setAccPhase(boolean: false), t=10*

Requirement *(b)* is not violated as no gear is skipped during the acceleration phase and as gear two is used for a period of four seconds. As no gear sequence in the example matches the sequence in Requirement *(e)*, this requirement is fulfilled as well.

Changing the gear sequence to the following would lead to a safety violation, as the sequence does not conform to Requirement *(b)*:

1. *env: gs.setAccPhase(boolean: true),t=2*
2. *gs: gc.setActiveGear(int: 2),t=5*
3. *gs: gc.setActiveGear(int: 3),t=7*

In detail, message 2 leads to an activation of MSD MinimumGearUsePeriod, progressing to cut 1. The cold condition in this MSD becomes enabled and is directly evaluated to true, followed by a clock reset at $t = 5$. The cut progresses to cut 2, leading to the minimal delay $c \leq 3$ becoming enabled and the cut being hot. Message 3 arrives two seconds ($t = 7$) after the previous message, while the cut is still hot. This situation corresponds to the transition between the states 2 and 3b in the zone graph of Fig. 4 and leads to a safety violation.

Note that in order to interpret the trace, end users will not have to understand the MSD representation of the WLTP requirements. We imagine mapping the violating event back to AVL CRUISE tool, where a violation in the gear shift sequence is highlighted, in combination with an informal description of the violated requirement.

For this example, we only considered Requirements *(b)* and *(e)*. The actual WLTP Annex II contains further requirements which leads to further Modal Sequence Diagrams and more messages. We only present a limited set here in order to make the example easier to understand. Out of the complete requirements set, we are able to model all but one, in which we need to be able to count how often a certain scenario occurs. We are currently evaluating how to do this and plan to use the simulation in a joint project between academia and industry.

6 Related Work

Timed play-out, along with LSCs extensions to express time constraints, were already proposed by Harel and Marelly [HM02b]. However, their approach supports only discrete time and the user is required to execute clock ticks to represent the passage of time. SCENARIOTOOLS instead shows users exactly the decisions on the progress of time which are relevant for different continuations of the execution.

The idea of validating traces against an LSC specification were already explored in the Tracer project [MKH07]. The Tracer tool visualizes concurrent activations and progress of scenarios and allowed and forbidden violations, but no time constraints were considered.

Lettrari and Klose propose an approach for monitoring and testing real-time systems against high-level Message Sequence Charts (hMSCs) [LK01]. hMSCs are less convenient for scenario-based specification than LSCs/MSDs because the concurrent progress of scenarios must be modeled explicitly, using a notation similar to activity diagrams. Also, hMSCs have no support for modeling cold (allowed) violations of scenarios; our example heavily relies on that concept.

MSDs/LSCs with real-time requirements were already proposed by Larsen et al. [LLNP09]; they describe a model-checking approach for verify real-time system designs against real-time LSC specifications. The approach is based on a mapping from real-time LSCs to timed automata and the analysis is performed by the UPPAAL model checker. The UPPAAL model checker also supports the simulation of the generated timed automata, but this simulation does not resemble an interactive play-out of of the LSC specifications. Moreover, the approaches only supports core LSC language features; parametrized messages, for example, are not supported.

7 Conclusion and Outlook

Embedded systems must often fulfill complex and critical real-time requirements that, to be able to apply automated validation and verification techniques, must be rigorously formalized. We propose to use a scenario-based approach, in particular using real-time MSDs. These MSDs can be used throughout the system and software development process, as we outlined in this paper: from analyzing real-time specifications by early play-out simulations to the validation of test traces, where MSDs act as a test oracle.

In this paper, we have presented our extension of SCENARIOTOOLS to support real-time MSDs and we have demonstrated the applicability of real-time MSDs and the tool in a practical case study.

We pursue different directions for future work. First, we plan a controlled experiment to compare scenario-based specification of requirements with MSDs and state-based specification of requirements with timed automata with respect to understandability as well as quality of the developed models. Second, based on the SCENARIOTOOLS runtime, we now want to investigate how to efficiently synthesize controllers from real-time MSD specifications.

Acknowledgements: We thank Nadja Marko, Alfred Wallner, and Christian Webel for discussions on the WLTP standard. This research is partially funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster “Intelligent Technical Systems OstWestfalenLippe” (it’s OWL) and is managed by the Project Management Agency Karlsruhe (PTKA). The research leading to these results has received partial funding from the European Union’s Seventh Framework Program (FP7/2007-2013) for CRYSTAL – Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830 and from Vinnova under DIARIENR 2012-04304.

References

[AD94] R. Alur, D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science* 126(2):183–235, 1994.

[avl] AVL CRUISE. last accessed Jan. 2014.
https://www.avl.com/cruise1

[BGP13] 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)*. Volume 58. EASST, 2013.

[BY04] J. Bengtsson, W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*. LNCS 3098, pp. 87–124. Springer, 2004.

[DH01] W. Damm, D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*. Volume 19, pp. 45–80. Kluwer Academic, 2001.

[Dil90] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In Sifakis (ed.), *Automatic Verification Methods for Finite State Systems*. LNCS 407, pp. 197–212. Springer, 1990.

[EH11] T. Eckardt, C. Heinzemann. Providing Timing Computations for FUJABA. In *Proc. 8th Int. Fujaba Days*. 2011.

[HM02a] D. Harel, R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. *Software and System Modeling (SoSyM)* 2:2003, 2002.

[HM02b] D. Harel, R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*. Pp. 193–202. 2002.

[HM03] D. Harel, R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[HM08] D. Harel, S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)* 7(2):237–252, May 2008.

[LK01] M. Lettrari, J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In Gogolla and Kobryn (eds.), *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. LNCS 2185, pp. 317–328. Springer, 2001.

[LLNP09] K. Larsen, S. Li, B. Nielsen, S. Pusinskas. Verifying Real-Time Systems against Scenario-Based Requirements. In Cavalcanti and Dams (eds.), *FM 2009: Formal Methods*. LNCS 5850, pp. 676–691. Springer, 2009.

[MH06] S. Maoz, D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proc. 14th Int. Symp. on Foundations of Software Engineering*. SIGSOFT '06/FSE-14, pp. 219–230. ACM, New York, NY, USA, 2006.

[MKH07] S. Maoz, A. Kleinbort, D. Harel. Towards Trace Visualization and Exploration for Reactive Systems. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*. Pp. 153–156. Sept 2007.

[Pla] PlayGo Tool. last accessed Jan. 2014.
http://www.weizmann.ac.il/mediawiki/playgo/

[Uni14] United Nations Economic Comission for Europe. Worldwide Harmonised Light Vehicle Test Procedures. Draft 26.08.2013, 2014. last accessed Jan. 2014.
https://www2.unece.org/wiki/download/attachments/5801176/26.08.2013%20Draft.pdf?api=v2