# Consistency Checking Scenario-Based Specifications of Dynamic Systems by Combining Simulation and Synthesis

Jens Frieben[1], Joel Greenyer[2⋆]

Fraunhofer Project Group Mechatronic Systems Design
Software Engineering Department
Zukunftsmeile 1, 33102 Paderborn, Germany
Jens.Frieben@ipt.fraunhofer.de

Politecnico di Milano,
DeepSE Group, Dipartimento di Elettronica e Informazione,
Piazza Leonardo Da Vinci, 32, 20233 Milano, Italy
greenyer@elet.polimi.it

*Preliminary version May 19, 2012.*

**Abstract.** Modern technical systems often consist of multiple components that must fulfill complex functions in diverse and sometimes safety-critical situations. Precisely specifying the behavioral requirements for such systems is a challenge, especially because there may be inconsistent requirements in possibly unforeseen component configurations. We propose a scenario-based specification approach based on Modal Sequence Diagrams and a novel technique for finding inconsistencies in such specification based on a combination of simulation and synthesis techniques. The simulation via the play-out algorithm can be used to analyze the scenario requirements in large and dynamic systems. Play-out, however, may run into avoidable violations, so that the engineer cannot assume the specification's inconsistency nor its consistency. We thus propose to check specification parts for static component configurations via synthesis. Then, if the part specifications are consistent, the resulting controllers can guide the play-out for the complete specification, avoiding more avoidable violations in the next simulation run.

**Keywords:** scenario-based specification, dynamic systems, consistency checking, simulation, synthesis

## 1 Introduction

Modern technical systems in areas like transportation, traffic, or production typically consist of multiple components that must interact to fulfill complex functions in diverse and sometimes safety-critical situations. Moreover, these systems

⋆ This work was elaborated mainly as part of the author's dissertation thesis [8], while he was working at the University of Paderborn, Germany

are often *dynamic*, i.e., the relationships among the components may change or components may leave or enter the system. Precisely and consistently specifying the requirements for such dynamic systems is a major challenge, especially because there may be many, possibly unforeseen configurations of components where components are involved in multiple use cases at once and conflicts among the components' interaction specifications are possible to occur. If such inconsistencies remain undetected, this may lead to costly iterations in the development or to flaws in the final product.

In this paper, we propose first (1) a use case- and scenario-based approach for specifying the interaction behavior of components in a dynamic system. The approach is based on Modal Sequence Diagrams (MSDs) [18,12], a recent variant of Live Sequence Charts (LSCs) [6], that allows the engineer to formally specify what may, must, or must not happen in a system. Second (2), we propose a novel technique for finding inconsistencies in MSD specifications, which is based on the symbiosis of simulation and synthesis techniques.

As an example, we consider the specification of the RailCab system, which is developed at the University of Paderborn. Here, small, autonomous rail vehicles, called *RailCabs*, transport passengers and goods on demand. This system is highly dynamic as relationships among RailCabs and control stations change when for example RailCabs move along track sections, switches, and crossings.

First, to model such systems, we propose a special specification scheme where the requirements described in use cases are formalized by scenario-based *use case specifications*. A use case specification captures the structure and interaction behavior described in a use case formally by using UML collaboration diagrams and MSDs. We extend the MSDs with OCL binding expressions that can be attached to lifelines and allow the engineer to specify precisely which components in a dynamic system shall play which role in a use case. Within ScenarioTools, we have implemented an Eclipse/EMF&UML-based simulation engine for executing such use case specifications via the play-out algorithm [14,18]. This helps the engineer understand the interplay of different MSDs as environment events occur in a particular, maybe structurally evolving, system instance.

The play-out algorithm typically has to make many non-deterministic choices when executing the MSDs. In doing so, it may reach a state where a number of MSD require that something must happen that is forbidden by other MSDs. Such a violating state may indicate that the specification is inconsistent, but it may also be consistent and just the play-out algorithm did not "look ahead" to avoid the violation. Finding this out manually can be a very difficult.

We observe in our example that use cases typically describe the interaction of a fixed set of participants. For this case, we developed a *synthesis* technique that can effectively determine whether such a use case specifications is inconsistent or not. If it is consistent, we can synthesize a *strategy* that demonstrates that there exists a system that can always react to all possible sequences of environment event in a way that satisfies the use case specification.

However, even if all use case specifications are consistent, it may be that conflicts among MSDs of different use case specifications occur if use cases *overlap*,

i.e., components are involved in multiple use cases at the same time. To further analyze the specification, we therefore still rely on the simulation via play-out. To improve the play-out, we developed a mechanism that guides the play-out by the strategies that could be successfully synthesized from single use case specifications. This improves the effectiveness of the simulation, giving the engineer more reason to suspect an actual inconsistency if a violation occurs. In the future, this approach could even be extended to successively eliminate all false negatives by synthesizing strategies also for overlapping use case occurrences.

This paper is structured as follows. We explain the foundations of MSDs in Sect. 2 and present an example use case specification in Sect. 3. A strategy that could be synthesized from a use case specification is explained in Sect. 4. In Sect. 5 we then describe our extended play-out algorithm and overview our tool implementation in Sect. 6. We discuss related work in Sect. 7 and conclude in Sect. 8.

## 2 Foundations

MSDs were proposed by Harel and Maoz as a formal interpretation of UML sequence diagrams, based on the concepts of LSCs [12]. In the following, we first explain the basics of MSDs and the play-out algorithm with respect to static systems. In Sect. 2.3 we then explain extensions to MSDs and their interpretation in the context of dynamic systems.

### 2.1 MSD Specifications

An MSD specification consists of a set of MSDs. An MSD can be *existential* or *universal*. Existential diagrams specify sequences of events that must be possible to occur in the system. Universal diagrams specify requirements that must be satisfied by all sequences of events that occur. During specification, the focus typically lies on universal MSDs, since they allow the engineers to express mandatory behavior. We also focus on universal MSDs in this paper.

Each lifeline in an MSD represents an object in an *object system* that consists of *environment objects* and *system objects*. The set of system objects is called the *system*; the set of environment objects is called the *environment*.

The objects can interchange messages. Here we consider only *synchronous* messages where the sending and receiving of the message is a single event. Our approach can, however, be extended also to asynchronous communication. We call the sending and receiving of a message a *message event* or simply *event*.

The messages in a universal MSD can have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*; the execution kind can be either *monitored* or *executed*.

The semantics of these 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 copy* of the MSD or *active MSD* is created. (We consider that an MSD has only one first message.) 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 copy 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 *safety violation* occurs iff in a hot cut a message event occurs that can be unified with a message in the MSD that is not currently enabled. If this happens in a cold cut, it is called a *cold violation*. Safety violations must never happen, while cold violations may occur and result in terminating the active copy of the MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if an active MSD never terminates or progresses to a monitored cut.

As an example, Fig. 1 shows an MSD and an illustration of the considered example. Cold monitored messages are shown as blue, dashed arrows; hot executed messages are shown as red, solid arrows. The dashed horizontal lines in the MSD RequestEnterAtEndOfTrackSection also show the reachable cuts, which are accordingly cold and monitored (c/m) or hot and executed (h/e). Intuitively, this MSD expresses the following requirements. We consider a scenario where a RailCab moves along its current track section. At some point the RailCab rc detects that it reaches the end of the current track section. This is modeled as the message `endOfTS` sent between the environment and the RailCab rc. Now the RailCab rc must send `requestEnter` to the next track section control tsc2, which must reply with `enterAllowed`. These two messages must be sent before the RailCab reaches a point where it is possible for the last time to safely break before entering the switch (modeled by the environment message `lastBrake`).

Messages can also have parameters of certain types. Message events must then carry according parameter values. Here the message `enterAllowed` has a Boolean parameter, representing the choice to allow or deny the RailCab to enter. In this MSD, the required parameter value is not specified, which allows the parameter value to be either true or false. For more details on the interpretation of parameter values, we refer to [8, pp. 33].

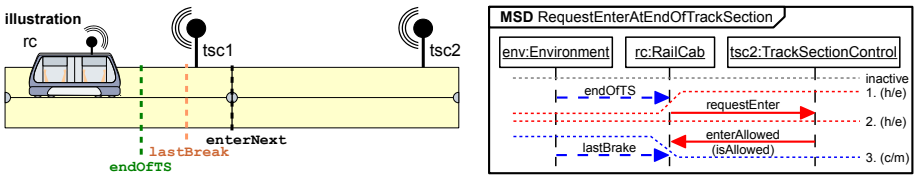


**Fig. 1.** The MSD RequestEnterAtEndOfTrackSection with illustration

We assume that the system is always fast enough to send any finite number of messages before the next environment event occurs. An infinite sequence of message events is called a *run* of the system and its environment. A run *satisfies* an MSD specification consisting of a set of universal MSDs if it does not lead to a safety or liveness violation in any MSD. (Multiple MSDs may be active at the same time.) We say that an MSD specification is *consistent* or *realizable* iff it is possible for the system objects to react to every possible sequence of environment events so that the resulting run satisfies the MSD specification.

### 2.2 Play-Out

Harel and Marelly defined an executable semantics for the LSCs, called the *play-out* algorithm [13], that was later also defined for MSDs [18]. The basic principle is that if an environment event occurs and this results in one or more active MSDs with active (enabled executed) system messages, then the algorithm non-deterministically chooses to send a corresponding message if that will not lead to a safety violation in another active MSD. The algorithm will repeat sending system messages until no active MSDs with an active message remain. Then the algorithm will wait for the next environment event, and this process continues.

If the play-out algorithm reaches a state where there are active messages, but they would all lead to safety violations, this is called a *violation*. If the MSD specification is inconsistent, this implies that there exists a sequence of environment events that will lead the play-out algorithm to a violation. Such a situation can, however, also occur if the specification is consistent. That is because the play-out algorithm will often make non-deterministic choices without "looking ahead" if they guarantee it not to run into violations later.

### 2.3 MSDs and Dynamic Systems

When specifying the behavior of dynamic systems, it is often impractical to consider MSDs where each lifeline refers to a concrete object. Instead, *symbolic lifelines* were introduced by Marelly et al. [19,14], which refer to a class. MSDs with symbolic lifelines are also called *symbolic MSD*; MSDs with non-symbolic lifelines, also called *concrete lifelines*, are called *concrete MSDs*. Here, concrete lifelines have an underlined label; the label of symbolic lifelines is not underlined.

In an active copy of an MSD with symbolic lifelines, a symbolic lifeline can be *bound* to an object that is an instance of the class referenced by the lifeline. For a given object system, the semantics of a symbolic MSD is equivalent to a set of concrete MSDs where for each possible combination of bindings of the symbolic lifelines, there exists a concrete MSD with lifelines corresponding to this possible combination of bindings.

Typically, we want to restrict a symbolic MSD to specify the behavior only for combinations of objects that have certain relationships or properties. Then, *binding expressions* can be added to the MSD in order to restrict the possible lifeline bindings. Harel and Marelly define that binding expressions are expressions over object properties or relationships between objects that evaluate to a

Boolean value [14]. A symbolic MSD then only specifies the behavior for the combinations of objects where there exists a set of lifeline bindings where all binding expressions evaluate to true.

Instead of translating symbolic MSDs to sets of concrete MSDs, Harel and Marelly extended the play-out algorithm to handle the dynamic binding of symbolic lifelines, supporting a simple form of binding expressions [14, pp. 209]. In ScenarioTools, we implement similar mechanisms and consider binding expressions of the form `<lifeline-name> := <expr>` where `<lifeline-name>` is the name of a lifeline, also called the *slot lifeline*, and `<expr>` is an OCL expression, also called the *value expression*. The value expression can evaluate to an object that is an instance of the slot lifeline's class.

Lifeline names can be used as variables within value expressions. If a lifeline is bound to an object, so is the corresponding variable. Also other variables can be used in value expression. In the course of progressing an active MSD, there may be for example variables that are assigned parameter values, like the variable isAllowed show in Fig. 1. The details of these mechanisms are not relevant here. Important is that value expressions can only be evaluated when all the variables appearing in the expression are bound.

During play-out, symbolic MSDs and binding expressions are interpreted as follows: As a message event can be unified with a first message in an MSD, an active copy of the MSD is created with the sending and receiving lifelines of the first message bound to the sending and receiving object of the message event. Then the value expressions of the binding expressions are evaluated as soon as that is possible, and the corresponding slot lifelines are bound to the resulting objects. It must not happen that a message is enabled and the sending or receiving lifeline is unbound.

As an example, consider the symbolic variant of the MSD RequestEnterAtEndOfTrackSection shown on the right of Fig. 2, executed in the context of an object system as illustrated on the left. If the message `endOfTS` is sent from the environment object e to the RailCab rc1, an active copy of the MSD is created where the lifeline env is bound to the object e and the lifeline rc is bound to the object rc1. Now the binding expression can be evaluated, which results in binding the lifeline next to the object tsc2.

We also consider that the value expression can evaluate to a set of objects. Then for each object in the set a copy of the active MSD is created with the slot lifeline bound to that object (see also [14, pp. 215]).

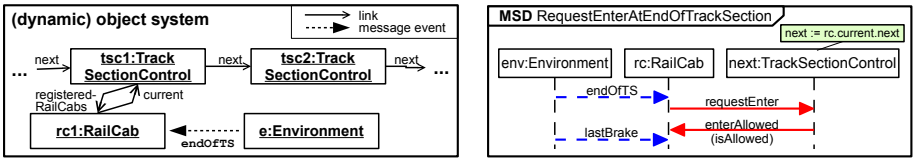


**Fig. 2.** A dynamic object system and the symbolic version of the MSD RequestEnterAtEndOfTrackSection

## 3 Use Case Specifications

We observe that the early, informal requirements of a dynamic system are often structured in use cases that describe (1) a particular configuration of objects and (2), by a number of scenarios, how these objects may, must, or must not react to certain, usually external, events. Instead of specifying the behavior of a dynamic system with a plain set of symbolic MSDs, we thus propose a more systematic approach where an engineer first captures the objects involved in the use case and then specifies the MSDs based on this structure.

### 3.1 Use Case Specification Structure

Figure 3 shows the example of a use case specification for the use case RailCab Obstacle Detected. The use case describes that a RailCab that detects an obstacle must report a hazard and its position to its current track section control, which then must warn the other RailCabs on that track section. The MSDs are in fact an example where the play-out algorithm may choose an execution that inevitably leads to a violation—but we will return to that in Sect. 4. Let us first examine the structure and semantics of such a use case specification.

A use case specification consists of a package with class definitions and a collaboration (dashed ellipse) [1, Sect. 9.3.3]. The collaboration captures the objects participating in a use case and it contains a set of MSDs. The nodes in the collaboration diagram are called *roles*, and each role represents a system or an environment object. Here environment roles are represented by a cloud symbol. The roles are typed by classes, which are modeled in the class diagram. The classes can define attributes, associations, and operations; the latter indicate which messages an instance can receive.

Each lifeline of an MSD represents one role in the collaboration. Connectors between the roles can be used for indicating structurally which roles interact in the use case. In the MSDs it can then be ensured that messages are only modeled between lifelines where their roles are connected.

### 3.2 Use Case Specification Semantics

The advantage of this specification scheme, besides supporting a more structures modeling approach, is that it allows for two different interpretations that are crucial for the symbiosis of simulation and synthesis.

**Symbolic interpretation:** The MSDs are interpreted as symbolic MSDs, as if their lifelines would directly reference the classes that type the roles represented by each lifeline. The object system can be any valid instance of the class model. The collaboration has no particular semantics.

**Static interpretation:** Here we assume an object system where for each role in the collaboration there is a corresponding object of the class typing the role. The MSDs are then interpreted as static MSDs where each lifeline represents the object that corresponds to its role.

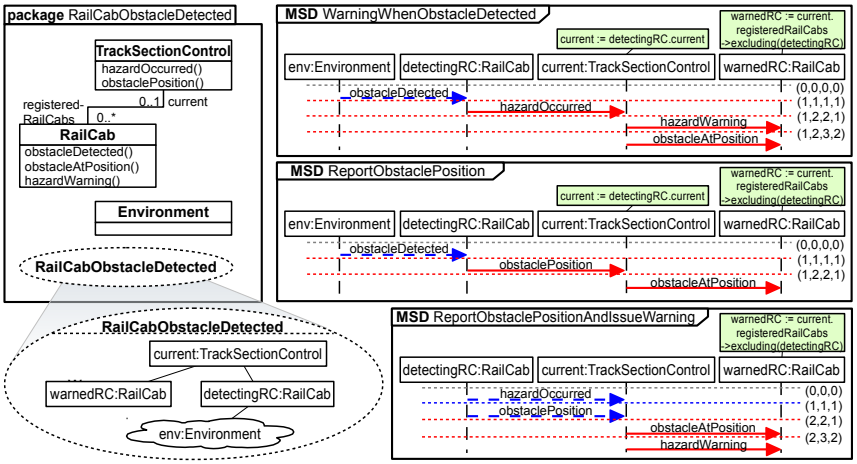


**Fig. 3.** The specification for the use case RailCab Obstacle Detected

The static interpretation makes the formal analysis of the use case specification feasible, which would not be the case with the symbolic interpretation, if we would have to consider many different object systems with different possible bindings of symbolic lifelines. The second-listed author has developed such a analysis technique in his thesis [8], which will be explained Sect. 4.

### 3.3 Combining Use Case Specifications

This modeling scheme also allows different engineers to specify different use case specifications in parallel. These can later be composed as follows.

(1) The class models of the use case specifications can be composed by merging them into one package using UML *package merge* [1, Sect. 7.3.41]. Package merge copies the contents of one or multiple *merged* packages into a *merging* package. Equally named UML elements (classes, attributes, operations, etc.) in the merged packages are mapped to the same element in the merging package.

(2) The UML package merge only defines how to merge structural (class) models. It could probably be extended easily to merge also the MSDs—but instead, we just slightly modify the symbolic interpretation of the MSD: We assume that the object system can be any valid instance of the merged class model. Then we interpret the sets of MSDs of all use case specifications like a plain set of (symbolic) MSDs where we interpret an MSD lifeline as if it was typed by the class that its role's class was merged into.

We call the merged package the *integrated package*. With this symbolic interpretation of the MSDs, it forms the *integrated specification* of the system.

If one use case depends on another, for example because one refers to message types (i.e., operations) or object properties already specified in another, this can also be expressed by a package merge where the depending use case specification package merges the package of the use case specification it depends on.

## 4 Synthesis

As already proposed by Bontemps et al. [3], the problem of deciding whether an MSD/LSC specification is consistent can be mapped to a two-player game problem. Intuitively, this means that environment events and system reactions are mapped to "moves" in a game that lead from one game state to another. A game state in this case is essentially a set cut configurations of currently active MSDs. Then it is checked whether there exists a *strategy* for the system against the environment such that a certain winning condition is satisfied. The winning condition here is that never a safety or liveness violation occurs and that there is no infinite sequence of system events, i.e., always eventually a next environment event can occur.

Today there exist a number of tools with efficient algorithms for finding winning strategies in two-player games. Similar to Bontemps et al, we have developed a synthesis approach [8,7] where MSD use case specifications are mapped to the input of Uppaal Tiga [2], a tool based on the Uppaal model-checker that implements an efficient game-solving algorithm [4]. Novel in our approach is that it also supports timed MSDs and MSDs that formulate *environment assumptions*, i.e., properties that the environment must satisfy to "win" against the system. But these novelties are not relevant in the scope of this paper.

In our synthesis approach, if an MSD use case specification is consistent, Uppaal Tiga will synthesize a strategy that shows us how the system can always react to the environment such that the specification is satisfied. Uppaal Tiga can even synthesize a *complete* strategy that shows all the moves to all states in which the system will be able to win. Furthermore, if an MSD use case specification is inconsistent, Uppaal Tiga can synthesize a *counter-strategy* that shows how the environment can always violate the MSD specification.

Listing 1.1 shows a excerpt from a complete strategy synthesized from the use case specification RailCab Obstacle Detected; Uppaal Tiga generates such a textual output to the console or a file. Here only one state in the game and the two winning transitions for this state are shown. As shown here, the sate is essentially a particular configuration of cuts of the active MSDs (see [8, App. C.2] for more information). Here it is the cuts reached in the MSDs after the environment event `obstacleDetected` occurred (compare also with Fig. 3).

**Listing 1.1.** Excerpt from the controller synthesized from the specification of the use case Warn RailCabs On Track

```
Strategy to win:
...
State: (...) ...
WarningWhenObstacleDetected_env=1
WarningWhenObstacleDetected_detectingRC=1
WarningWhenObstacleDetected_current=1
WarningWhenObstacleDetected_warnedRC=1
ReportObstaclePosition_env=1
ReportObstaclePosition_detectingRC=1
ReportObstaclePosition_current=1
ReportObstaclePosition_warnedRC=1
ReportObstaclePositionAndIssueWarning_detectingRC=0
ReportObstaclePositionAndIssueWarning_current=0
```

```
ReportObstaclePositionAndIssueWarning_warnedRC=0
When you are in true, take transition
    systemProcess.systemActive->systemProcess.produceEvent
        { 1, tau, event := detectingRC_current_reportHazard }
When you are in true, take transition
    systemProcess.systemActive->systemProcess.produceEvent
        { 1, tau, event := detectingRC_current_obstaclePosition }
...
```

According to this strategy, the system can satisfy the use case specification against any environment if in this state the RailCab detectingRC sends the message `reportHazard` or `obstaclePosition` to the track section control current. As it is a complete strategy, we know that sending any other system message will not allow the system to "win" against any other environment.

The possible violation is follows: Sending the message `hazardOccurred`, which in this state is enabled in the MSD WarningWhenObstacleDetected, must be followed by sending `obstaclePosition`. This then leads to a situation where there is an active copy of MSD WarningWhenObstacleDetected in cut (1,2,2,1) and an active copy of MSD ReportObstaclePositionAndIssueWarning in cut (2,2,1). In the former, `hazardWarning` must occur, but `obstaclePosition` must not occur. In the latter, `obstaclePosition` must occur, but `hazardWarning` must not occur. Thus a safety or liveness violation is inevitable.

## 5 Combining Play-Out and Synthesis

The naive play-out of an integrated specification containing the use case RailCab Obstacle Detected could easily run into the above-mentioned, avoidable violation. In many cases, this could be avoided by using the strategies that could be successfully synthesized from single use case specifications. In the following, we explain an extension of the play-out algorithms that is guided by these strategies.

The principle of this extension is shown in Fig. 4. At the bottom, it sketches a RailCab object system where the environment just sent the message `obstacleDetected` to the RailCab rc2. On the top left, two active MSDs are shown, which are activated as a result. The lifeline bindings are indicated by the small labels on the lifelines. The remaining parts of the figure are explained in the following.

The main challenge in employing the strategies synthesized from use case specifications is to determine in a state during play-out which state in which strategy (or which states in which strategies) to inquire about which message event can be safely executed next. Intuitively, we have to determine where a use case "occurs". We define a *use case occurrence* as a set of active MSDs where the MSDs belong to the same use case occurrence and the lifelines representing the same role are bound to the same object.

After finding the active MSDs that make up a use case occurrence (1), we create an *active copy* of the corresponding strategy (also called *active strategy*). Then we find a state in the this strategy that corresponds to the cuts of the active MSDs (2). This state is also called the *current state* of the active strategy for the use case occurrence.

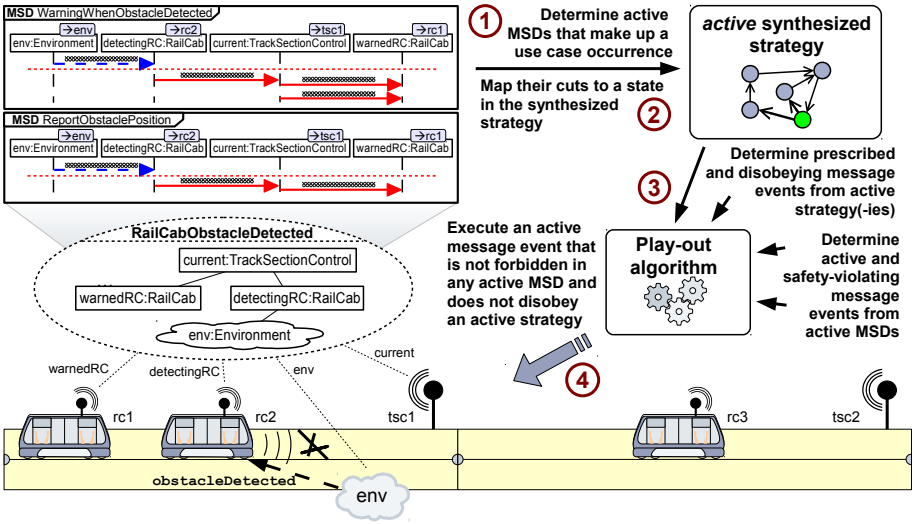


**Fig. 4.** Guiding the play-out by strategies synthesized from use case specifications

Once such corresponding states are found for all occurrences of use case for which a strategy is provided, we determine the *prescribed* and *disobeying* message events (3). A message event is *prescribed* if it corresponds to an event that labels a transition leaving the current state of the active strategy; a message event is *disobeying* if it corresponds to a message event that labels a transition that is not leaving the current state. (For brevity, we skip a more formal definition when message events correspond in this case.)

As in regular play-out, we also have to determine which message events are active and safety-violating in the active MSDs. We can now execute an active event that is not safety-violating any other active MSD, and not disobeying any active strategies (4). This process is repeated until the there are no more active events. Then the system waits for the next environment event.

In this extended play-out it is not guaranteed that never an active strategy must be disobeyed or never a safety violation occurs. This may still happen when use cases overlap, i.e., objects participate in multiple use case occurrences at once. Then again safety violations and events disobeying active strategies may not necessarily mean that the specification is inconsistent—it could still be that the system in the past could have chosen another sequence of steps to avoid this. The second-listed author also describes an extension of this approach for employing strategies synthesized from *composed use case specifications*, but these concepts are not yet implemented.

After an active strategy was disobeyed, the play-out can still continue, but then it may no longer be possible to find current state for the disobeyed active strategy. Note that if the strategies used in this process are not complete, it becomes more likely that active strategies must be disobeyed.

Note also that we can only identify use case occurrences if all lifelines of an active MSD are bound. For this process to work properly, there should thus not be an active MSD with unbound lifelines. It remains to be investigate if maybe the play-out can follow multiple active strategies in parallel for different "candiadate" use case occurrences as long as lifelines are unbound.

## 6 Realization and Evaluation

The concepts introduced here have been implemented in an Eclipse-based tool suite called ScenarioTools[1]. Figure 5 gives an overview of ScenarioTools and the supported modeling and analysis process.

In the first step, a UML-based MSD specification of the system is modeled. For modeling, ScenarioTools extends the Topcased UML-Editor. The figure shows a number of packages that represent use case specifications (1). As mentioned before, use case specifications can be modeled in separate packages, dependencies can be expressed by package merge relationships, and finally all use case specifications are merged into an integrated package.

The figure here also shows a *base package*. We suppose that sometimes, prior to specifying the use cases, the requirements engineers already want to formally capture a structural (class) model of the system. This is also called *domain modeling*, and fosters a common understanding of the domain. This can be done in this separate package from where classes, associations, and attributes can be reused in the use case specifications. To do this, the use case specifications define merge relationships to the base package.

In the second step, after formally specifying the use cases, we want to create an instance system, or possible many instance systems, to carry simulations of the specified behavior. To be able to do this, we create an EMF/ECore[2] class model that corresponds to the merged class model of the UML-based MSD specification. This transformation is described by Triple Graph Grammar (TGG) that can be executed using the TGG Interpreter[3]. The transformation not only creates the ECore class model, but also a *correspondence model* that stores a detailed mapping between classes, properties, associations and operations in the UML and ECore class models.

The Eclipse/EMF framework allows us to automatically generate simple editors from the ECore class model. With the help of these editors, instance models can easily be created (3). In the RailCab case, this could be a particular RailCab track system with a particular number of RailCabs currently on certain track sections.

Based on an instance model, we can now simulate the behavior defined in the UML-based MSD specification (4). To know which MSD lifelines can be bound to which objects and which messages can be sent between the objects, we exploit the information in the above-mentioned correspondence model.

[1] `http://www.cs.uni-paderborn.de/index.php?id=scenariotools`
[2] `http://www.eclipse.org/emf/`
[3] `http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter`

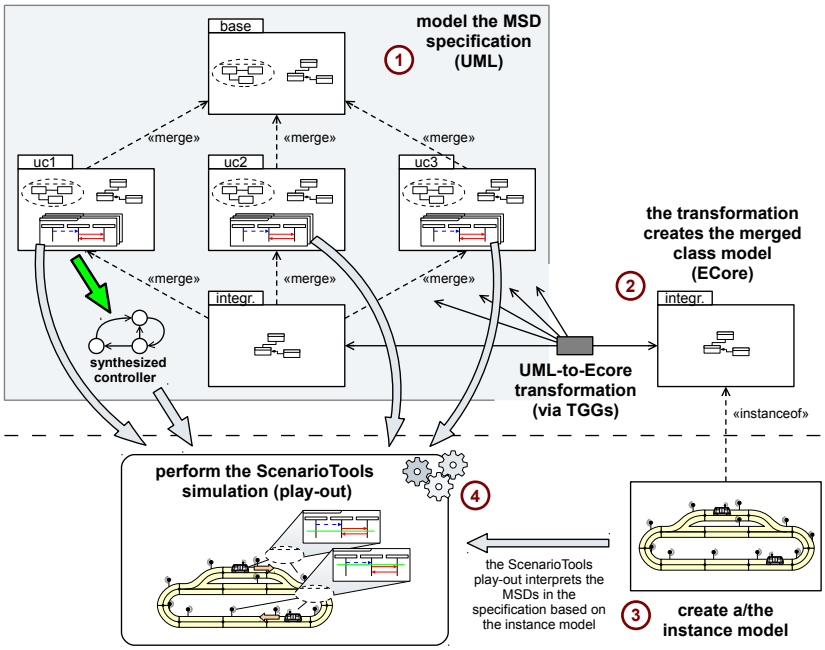


**Fig. 5.** Overview of the ScenarioTools simulation

The figure also illustrates that the play-out can be guided by strategies that could be successfully synthesized from use case specification.

ScenarioTools supports different simulation modes: a user-guided step-by-step selection of system and environment events and a random execution.

## 7 Related Work

In the past, many approaches for the scenario-based specification of system requirements have been proposed. Many, however, did not regard that scenarios can be overlapping [15] or they only regarded existential scenarios, i.e., descriptions of what must be possible to occur [17]. Others considered combining existential scenarios with pre-and post-conditions on messages [21], automata [20], or safety properties in temporal logic [5] for expressing also mandatory requirements. With these additions also came the problem of checking the consistency of the specification, which is addressed in these papers. To the best of our knowledge, all these approaches did not consider the specification of dynamic systems.

LSCs introduced a rigorous semantics for expressing universal and existential requirements [6] and only with symbolic lifelines [19], the behavior of dynamic systems could be specified in a formal scenario-based way. Many approaches for consistency checking LSC specifications and synthesizing controllers from them were proposed [9,11,3,16], but they only consider static systems.

Another approach for improving the play-out of LSC specifications is smart play-out [10]. Here model-checking is employed for finding a sequence of steps for the system in reaction to an environment event that avoids avoidable violations. The problem here is that this approach can only "look ahead" until the next environment event occurs, thus not all avoidable violations can be anticipated. Also, smart play-out only works in a static setting.

Maoz et al. presented an alternative implementation of the play-out algorithm using AspectJ [18]. This implementation is extensible to plug-in different play-out "strategies", which for example allows for integrating smart play-out. This implementation of the play-out algorithm is also used in the PlayGo tool[4]. The website also mentions that synthesized strategies and counter-strategies can be executed using this tool, but no details have been published thus far. Kugler et al. also mention to execute synthesized controllers [16], but these are not combined with play-out.

## 8 Conclusion and Outlook

We presented a novel extension of the play-out algorithm which combines play-out of MSDs in a dynamic object system with strategies synthesized from specification parts. This helps avoid more avoidable violations and improves the play-out because engineers now have more reason to suspect an actual inconsistency when violations occur.

We are currently developing a new version of the ScenarioTools play-out and synthesis and plan to extend this approach to also support environment assumptions and parametrized messages. We also plan to further investigate the symbiosis of synthesis and play-out for composed use cases.

## References


1. UML 2.4.1 superstructure specification (August 2011), OMG document `formal/2011-08-06`
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) Proc. 19th Int. Conf. on Computer Aided Verification (CAV'07). LNCS, vol. 4590, pp. 121–125. Springer, Berlin, Germany (July 2007)
3. Bontemps, Y., Schobbens, P.Y.: Synthesis of open reactive systems from scenario-based specifications. In: Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD 2003), 18-20 June 2003, Guimaraes, Portugal. pp. 41–50 (2003)
4. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) Proc. 16th Int. Conf. on Concurrency Theory (CONCUR'05). LNCS, vol. 3653, pp. 66–80. Springer, San Francisco, CA, USA (August 2005)


[4] `http://www.weizmann.ac.il/mediawiki/playgo/index.php`

5. Damas, C., Lambeau, B., van Lamsweerde, A.: Scenarios, goals, and state machines: a win-win partnership for model synthesis. In: Proc. 14th Int. Symp. on Foundations of Software Engineering (ACM SIGSOFT '06/FSE-14). pp. 197–207. ACM (2006)
6. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. In: Formal Methods in System Design. vol. 19, pp. 45–80. Kluwer Academic Publishers (2001)
7. Greenyer, J.: Synthesizing modal sequence diagram specifications with Uppaal-Tiga. Tech. Rep. tr-ri-10-310, University of Paderborn (February 2010)
8. Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (October 2011)
9. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. In: Foundations of Computer Science. vol. 13:1, pp. 5–51 (2002)
10. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD 2002, Portland, OR, USA, November 6-8, 2002. pp. 378–398 (2002)
11. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In: Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 309–324. Springer (2005)
12. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling (SoSyM) 7(2), 237–252 (May 2008)
13. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: The play-in/play-out approach. Software and System Modeling (SoSyM) 2, 2003 (2002)
14. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (August 2003)
15. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: Proc. IFIP WG10.3/WG10.5 Int. Workshop on Distributed and Parallel Embedded Systems (DIPES '98). pp. 61–71. Kluwer Academic Publishers, Norwell, MA, USA (1999)
16. Kugler, H., Plock, C., Pnueli, A.: Controller synthesis from LSC requirements. In: Chechik, M., Wirsing, M. (eds.) Proc. 12th Int. Conf. of Fundamental Approaches to Software Engineering, FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer (2009)
17. Maier, T., Zündorf, A.: The Fujaba statechart synthesis approach. In: Proc. 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003 (2003)
18. Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Proc. Int. Symp. on Foundations of Software Engineering (FSE'05). pp. 219–230 (2006)
19. Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Proc. 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02). ACM SIGPLAN Notices, vol. 37, pp. 83–100 (November 2002)
20. Sikora, E., Daun, M., Pohl, K.: Supporting the consistent specification of scenarios across multiple abstraction levels. In: Wieringa, R., Persson, A. (eds.) Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol. 6182, pp. 45–59. Springer (2010)
21. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proc. 22nd Int. Conf. on Software Engineering, ICSE '00. pp. 314–323 (2000)