Play-out for Hierarchical Component Architectures

Jörg Holtmann, Matthias Meyer

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

Abstract: One approach to cope with the growing complexity of today's embedded systems software, especially in the automotive domain, is component-based software development. For systems based on hierarchical component architectures like AUTOSAR and developed in conformance with process models such as Automotive SPICE, requirements should be specified for the whole system under development and partitioned subsequently onto the particular components across several hierarchy layers. In previous work, we developed a formal requirements engineering (RE) approach based on a recent Live Sequence Chart (LSC) variant, so-called Modal Sequence Diagrams (MSDs). This scenario-based RE approach allows to validate the requirements by means of simulation (i.e., the play-out algorithm originally conceived for LSCs) and to formally verify the requirements for consistency. However, these scenarios are specified on a plain structural basis that does not reflect the typical structure of component architectures, which are arranged in a hierarchical way and encompass ports, interfaces, and directed connectors. In order to tackle this problem, we introduce in this paper a modeling and simulation approach for MSDs based on hierarchical component architectures. By binding these two aspects together, we foster an integrated and iterative RE and component architecture design.

1 Introduction

Software for embedded systems is becoming more and more complex, especially in the automotive domain. One approach to cope with this complexity is component-based software development, which provides several advantages like fostering reuse and separation of concerns (e.g., AUTOSAR[1]). Development process models for automotive systems (e.g., Automotive SPICE [Aut10] or ISO 26262 [Int11]) propose several requirements engineering (RE) phases intertwined with architecture design phases across multiple abstraction levels. For systems based on hierarchical component architectures and developed in conformance to above mentioned process models, requirements should be specified for the whole system under development and partitioned subsequently onto components across several hierarchy levels. This enables to specify requirements on different abstraction levels and for particular components that can be developed by independent teams.

[1] http://www.autosar.org

In previous work [BGP13, GF12], we developed a formal RE approach based on a recent Live Sequence Chart (LSC) [DH01] variant compliant to the Unified Modeling Language 2 (UML2) [Obj11a], so-called *Modal Sequence Diagrams (MSDs)* [HKM07, HM08]. This scenario-based RE approach allows to validate the requirements by means of simulation (i.e., the play-out algorithm originally conceived for LSCs [HM03] that was later also applied to MSDs [HKM07, HMSB10]). Furthermore, the approach enables to formally verify for consistency of the requirements by synthesizing global controllers from the scenarios. The approach is implemented in the Eclipse-based tool suite SCENARIOTOOLS[2].

However, the scenarios in SCENARIOTOOLS as well as in the original play-out approach [HM03] and its MSD variant [HKM07, HMSB10] are specified based on plain class models. These allow to specify scenarios on one hierarchy level, but are not well suited to support scenarios across several hierarchy levels. Furthermore, each structural entity can communicate with arbitrary other entities. This way of requirements specification does not reflect the typical style of component architectures like AUTOSAR, which are arranged in a hierarchical way and encompass ports, interfaces, and directed connectors.

In order to tackle this problem, we extend SCENARIOTOOLS in this paper by a modeling and simulation approach for MSDs based on hierarchical component architectures. Therefore, we apply a subset of structural models of the Systems Modeling Language (SysML) [Obj10] to specify hierarchical component architectures as a structural basis for the scenario-based requirements. Since SysML and its underlying base language UML2 provide plenty of modeling constructs, we furthermore conceived and formalized modeling rules for the combined structural and scenario models to guide the requirements engineer in specifying models that are applicable in play-out. Finally, we adopted the play-out algorithm to take component architectures into account. By binding scenario-based RE and hierarchical component architectures together, we foster an integrated and iterative RE and component architecture design across multiple hierarchy or abstraction levels.

This paper is structured as follows. In the next section, preliminaries regarding the running example, MSDs, and the play-out are introduced. In Section 3, we investigate related work on the topic of this paper. Section 4 presents our concepts applied on the running example. In the last section, we conclude and give an outlook to future work.

2 Preliminaries

In this section, we will first introduce the running example used within this paper. Based on this example, we will present some basic concepts of MSDs in Section 2.2. Afterward, we shortly explain the conventional play-out of SCENARIOTOOLS.

[2] http://scenariotools.org/

2.1 Running Example

We will consider two cars interacting with each other via Car-2-Car (C2C) communication as *system under development (SUD)*. In the following, we will present a simplified blind overtaking scenario. In this scenario, two cars drive with different speeds on one route, and oncoming traffic can occur on the opposite lane. We simplistically assume that if one car detects an obstacle in its front, then the obstacle is the other car. Furthermore, both cars already have a bidirectional communication connection. When a car detects an obstacle and thus the other car, both cars have to agree on a safe overtaking such that the slower car does not increase its speed while the faster one is overtaking.

2.2 MSDs

MSD specifications are structured by means of use cases that specify requirements for independent scenarios and consist of several MSDs per use case. In the following, we exemplary focus on one MSD of the use case for the blind overtaking scenario. This is the MSD OvertakeTopLevel depicted in Figure 1. In this MSD, we want to express the requirement that when a car detects an obstacle and thus another car in its front, then it has to agree on a safe overtaking with the other car.

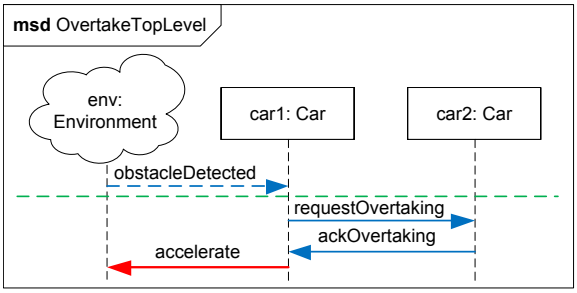

Figure 1: Example MSD

As in all scenario-based modeling languages, an MSD consists of several *lifelines* representing structural entities of the SUD. We distinguish into *environment objects* and *system objects*. The former ones represent the environment that is sensed and manipulated by the SUD, while the latter ones represent several *roles* or components of the SUD. In Figure 1, the environment object env:Environment is represented for illustration purposes with the cloud symbol including the vertical dashed line at its bottom, and the system objects car1:Car and car2:Car—representing two roles of the SUD car—are visualized by rectangles including the vertical dashed lines at the bottom.

The second aspect common to all scenario-based modeling languages are *messages* sent from one lifeline to another, represented by arrows between the lifelines in Figure 1. The actual sending or receiving of a message at runtime or during play-out is called an *event*.

Specific to messages of MSDs are their *execution kind* and *temperature*. In general, an MSD progresses if a message occurs within the system at the point in time specified by the MSD. The execution kind can be *monitored* or *executed*, depicted by dashed and solid arrows in Figure 1, respectively. The intuitive semantics is that monitored messages can but do not need to occur, while executed messages must occur eventually. The temperature specifies the modality of a message and can be *hot* or *cold*, which is depicted by red and blue arrows in Figure 1, respectively. The intuitive semantics of a hot message is that other messages specified by the MSD must not occur at this point in time (i.e., mandatory behavior). For a cold message, a different message specified by the MSD may occur at this point in time, which discards the MSD (i.e., provisional behavior). Occurring messages that are not specified by an MSD do not influence this MSD.

An event is *unifiable* with a message iff its name equals a message name of an MSD and the event sender and receiver correspond to the MSD's message sender and receiver lifelines. An MSD becomes *active*, when an event occurs that is unifiable with the MSD's first message. This first message is always cold and monitored, for example, the message obstacleDetected that is an abstraction from the real-world situation that the sensors of a car detect an obstacle in its front. After this event occurred, the active MSD progresses only when the event requestOvertaking occurs that is unifiable with the identically named message. By means of this message, one car starts the communication with another car to agree on a safe blind overtaking. The current state of an active MSD is also called the *cut*, which can be imagined by a horizontal line between two messages (visualized by the green dashed line in Fig. 1) After the cars have agreed on overtaking, then car1:Car has to start the overtaking procedure by sending the abstracting message accelerate to the environment. Since this is a safety-critical aspect, no other messages are allowed in this state, and accelerate thus is hot.

There are further aspects of MSD specifications that are not in the scope of this paper. First, MSDs also support timing requirements that make them well suited for automotive systems. Second, when events are sent in a wrong order, at a wrong point in time, or not at all, then different kinds of violations depending on the temperature and execution kind of the corresponding message can occur, which can be determined with play-out. At last, we assume in this paper static component configurations and thus a static interpretation of MSD specifications. That is, we consider no dynamic systems encompassing potential additional cars entering or leaving the scenarios or a dynamic reconfiguration of the cars' interior components. We refer to [BGP13, GF12, DH01, HM03, HKM07, HM08, HMSB10] for further information.

2.3 Conventional Play-out

The play-out algorithm serves to validate requirements specified with MSDs in a simulative manner. Play-out operationalizes the MSD semantics outlined in the last subsection. If no MSD is active, play-out waits for environment events to occur such that one or several MSDs become active (e.g., the event obstacleDetected unifiable with the corresponding message from the MSD OvertakeTopLevel in Fig. 1). Afterward, it "plays out" all possible

system events until no MSD is active anymore. Then it again waits for environment events.

The play-out of the MSD OvertakeTopLevel would initially wait for the environment to send an event obstacleDetected to car1. If this happens, the MSD becomes activated and the cut moves to the position as indicated in Figure 1. Afterward, the events corresponding to the following executed messages are triggered by the algorithm. If these events are unifiable with messages in other MSDs, their cuts progress, too. If the last event is sent, the MSD terminates.

3 Related Work

As already mentioned in the introduction, neither the original play-out approach [HM03] nor its MSD variant [HKM07, HMSB10] nor our RE approach implemented in SCENARIOTOOLS [BGP13, GF12] supports modeling and simulation of scenarios for hierarchical component architectures. In this section, we investigate further related work on scenario-based specifications for component-based or hierarchically structured architectures. Furthermore, there are approaches like [LM09] and [SDP10] that reflect hierarchies in combination with scenario-based specifications in a simple way, but do not intend to provide a simulative requirements validation. Since our second main focus is such a requirements validation, we do not consider these approaches in detail.

Combes et al. [CHK08] present a case study that applies LSCs to a component-based architecture for telecommunication systems. These models are validated by means of play-out and formally verified for the non-satisfiability of anti-scenarios using smart play-out. They divide their modeling approach into a structural, a dynamical, and an extra-functional view. The structural view of the system encompasses components, ports, port interfaces containing the messages that can be exchanged between the ports, and connectors specifying communication channels between the component ports. The dynamic view consists of LSCs that are specified based on the structural view. These LSCs describe the dynamic behavior of the particular components and ports. The extra-functional view adds timing constraints to the LSCs. Although the relationship between the LSCs and the component architecture is described very implicitly, it provides a good basis for further formalization. Unfortunately, there are only scenarios that describe interactions between a component and its ports as well as between ports of different components on the same hierarchy level. Thus, there are no means to specify or refine scenarios across component hierarchy levels.

Atir et al. [AHKM08] adapt the original MSD play-out approach [HKM07, HMSB10] to hierarchical object compositions. Therefore, the authors syntactically and semantically extend MSDs by the PartDecomposition concept of UML2 [Obj11a]. Basically, PartDecomposition allows to decompose one lifeline parent-lifeline of a scenario parent-scenario into several lifelines PL = {part-lifeline$_1$, ..., part-lifeline$_n$} of another scenario part-scenario, provided that the structural objects represented by PL are part of the structural object represented by the lifeline parent-lifeline. To adapt the play-out approach to be compatible with the PartDecomposition concept, LSC-trees consisting of several part-scenarios that together reflect the object composition hierarchy including consistency rules are conceived.

A composition algorithm combines this scenario hierarchy to overall flat MSDs that are executable in play-out. Although the approach is able to traverse hierarchies, only class models are supported. Thus, the typical concepts of components like ports, interfaces, and directed connectors are not considered.

Finally, the UML2 [Obj11a] and its derived languages (e.g., SysML [Obj10]) and profiles (e.g., MARTE [Obj11b]) in general provide the means to specify hierarchical components including ports and directed connectors as well as scenarios describing their dynamic behavior. But two main facts exacerbate the modeling of scenario-based requirements based on hierarchical component architectures and particularly their simulative validation. First, UML2 has informally described semantics. Second, the language encompasses a huge amount of modeling constructs resulting in the possibility to express the same aspect by means of different (combinations of) modeling constructs. On the one hand, it is thus difficult for the modeler to find the "right" way of specification since he has plenty of modeling possibilities to choose from if there is no guidance. On the other hand, it is not possible to conceive a simulation algorithm for the execution of scenarios for hierarchical component architectures if there is no formalized operational semantics for scenarios and all possible modeling constructs are allowed. For example, the wide-spread UML2/SysML modeling tools Enterprise Architect[3] and Rational Rhapsody[4] allow a simulation of sequence diagrams, but this simulation only supports a very basic subset of modeling constructs such that is only possible to animate plain messages of one sequence diagram in their partial order.

Summarizing, the investigation of the related work yields that there is no approach that allows to both model and simulate scenario-based specifications for hierarchical component architectures. Some of the formally well-defined play-out approaches reflect either component architectures only on one hierarchy level or hierarchically structured classes, but not hierarchical component architectures. In contrast, UML2 and its derivatives in general allow to model scenarios for hierarchical component architectures but do not provide adequate means for a simulation of the scenarios due to missing formal semantics and a huge amount of modeling constructs.

4 Scenario-based Requirements for Hierarchical Component Architectures

In order to enable a simulative requirements validation for hierarchical component architectures, we combine ideas of some of the concepts identified in the last section. To specify hierarchical component architectures, we apply SysML blocks specified with Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs). Although SysML has only semi-formal semantics and represents no component model in the sense of [CSVC11], we use this subset of structural models of SysML to specify abstract component architectures due to the following reasons. First, we argue that it is not useful to decide on a specific

[3] http://www.sparxsystems.com/products/ea/

[4] http://www.ibm.com/software/products/us/en/ratirhapfami/

component formalism in the early systems engineering phases, but to use a more abstract notion that still covers all relevant aspects. After the concrete component and implementation language (e.g., AUTOSAR) has been determined, the transition to such component models can be accomplished by model transformations (cf. [GHN10]), conforming with the vision of the Model Driven Architecture [Obj03]. Second, SysML is the conceptual and partly technical foundation of further architecture description or systems engineering languages applied in the development of automotive and mechatronic systems like EAST-ADL [Mod12] or CONSENS [GFDK09], for example. Thus, our approach can be applied onto structural models specified in these languages, too.

Figure 2 shows the overview of our approach and the relation between the artifacts. The figure is arranged as a matrix that sorts the artifacts into different hierarchy or abstraction levels (i.e., Top Level, Subsystem Level, and Subsubsystem Level) and into different view types (i.e., Type View, Internal Structure View, and Interaction View). The hierarchy of the SUD is induced by the hierarchical decomposition of the component architecture. It starts with the Top Level and can contain an arbitrary number of further levels. The view types encompass the Type View containing interface and block definitions specified by means of BDDs, the Internal Structure View containing the internal structure of the particular blocks specified by means of IBDs, and the Interaction View containing the scenarios specified by means of MSDs. In an ideal RE process, these artifacts would be specified from the left to the right at one hierarchy level and from top to bottom across the particular hierarchy levels. Since such an ideal process cannot be guaranteed, of course iterations are allowed.

In the following subsection, we will present the modeling approach. The second subsection presents the implications of the adaption of the play-out algorithm to this new modeling approach, and we summarize in the last subsection. We use the running example from Section 2.1 to illustrate the concepts.

4.1 Modeling Approach

In the following two subsections, we will present more details about the artifacts, the correlations between them, and the modeling rules we conceived by explaining extracts from Figure 2 in the order of the ideal modeling process. Therefore, we will initially specify the environment and two cars as participants of a use case, such that we can specify scenarios like OvertakeTopLevel (cf. Fig. 1) on this structural basis. Afterward, we will decompose the SUD car into several subsystems like sensors, logical computation, and actuators, what induces a further hierarchy level. Thus, the top level requirements have to be partitioned onto these new subsystems and hence the MSDs from the top level have to be refined accordingly. Then, also the subsystems can be decomposed into further components, and so on. Thus, these steps can be repeated until a sufficient granularity level is achieved.

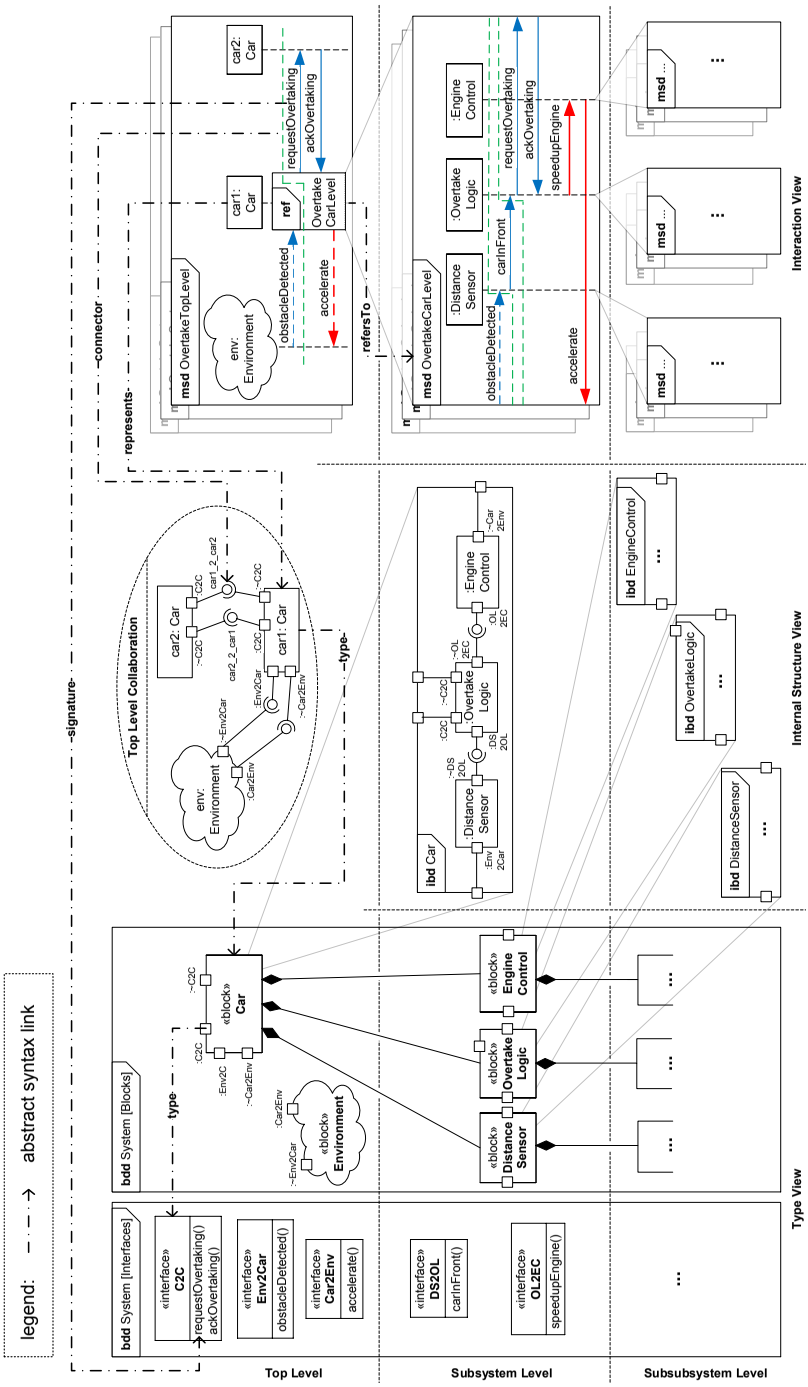

Figure 2: Overview of the approach and relation between artifacts

4.1.1 Modeling Component Architectures and MSDs Across Multiple Hierarchy Levels

Top Level To specify hierarchical component architectures, we have to first define interfaces and blocks in the Type View. Therefore, we start in the Top Level and specify within the BDD System [Blocks] the blocks Environment and Car.

Now we specify, which interfaces between environment and the SUD as well as between several participants of the SUD among each other are relevant for the blind overtaking scenario. The interfaces that we apply in this paper provide means for unidirectional communication. Within the BDD System [Interfaces], we define the interface C2C containing the operations requestOvertaking() and ackOvertaking(). Then we add two ports to Car within the BDD System [Blocks]: One port :C2C that is typed by C2C (cf. the corresponding abstract syntax link type) and hence serves as provided interface, and one port :~C2C that has the same type but is *conjugated* (indicated by the preceding ~). This means that the direction of the interface typing the port is reversed, enabling to apply it as required interface (cf. [Obj11a, Sect. 9.3.12]). For the communication between environment and the car, we specify one interface for each message obstacleDetected and accelerate. Afterward, we add corresponding ports to Environment and Car and type them accordingly.

In the next step, we set up the communicating participants in the Internal Structure View. Like in the original SCENARIOTOOLS approach [BGP13, GF12], a UML2 collaboration together with several MSDs (indicated by the cascading of the MSD diagram in the Interaction View) formally specifies a use case. Thus, we use a collaboration diagram (i.e., the dashed ellipse named Top Level Collaboration) to specify the top level communicating roles env:Environment, car1:Car, and car2:Car. The roles are typed by the corresponding blocks Environment and Car defined in the BDD System [Blocks] within the Type View (indicated by means of the corresponding abstract syntax link type). Furthermore, we define *assembly connectors* between the ports of the particular communication roles: The unidirectional connectors car1_2_car2 and car2_2_car1 connect the corresponding ports of car1 and car2, and env and car1 are connected by two further unidirectional connectors (the connector directions are indicated by means of the "lollipop notation" of UML2).

Based on the structural artifacts, we are now able to specify scenarios in the Interaction View. We still focus on the blind overtaking scenario, that is, the MSD OvertakeTopLevel in the Top Level. Note that if a refining MSD is added on the subordinate hierarchy level (i.e., Subsystem Level), slight changes to an MSD have to be performed at the superordinate hierarchy level. Thus, let us assume that for the following standalone consideration of the Top Level the MSD OvertakeTopLevel looks exactly like the one in Figure 1, but supplemented by the abstract syntax links from Figure 2. The MSD contains the lifelines env:Environment, car1:Car, and car2:Car that represent the corresponding roles from Top Level Collaboration (indicated by means of the corresponding abstract syntax link represents). Having specified the correlations to the roles in Top Level Collaboration, the possible messages that can be sent between the lifelines are determined. Therefore, a connector between the ports of the corresponding roles represented by a lifeline is assigned to a message (e.g., car1_2_car2 is assigned to the message requestOvertaking). Afterward, an operation of an interface corresponding to a connected port is assigned to a message

(e.g., the message requestOvertaking sent from car1 to car2 corresponds to the operation requestOvertaking() in the interface C2C).

Summarizing the topmost hierarchy level of the component architecture, until now we are able to specify MSDs based on components similar to the approach of Combes et al. [CHK08], which specifies component-based LSCs on one hierarchy level. One difference to the modeling approaches of [CHK08] and [Obj11b, Sect. A.3.2] is that the lifelines in our approach do not represent ports but components. This is due to the fact that ports typically just relay messages. Thus, respecting them in interactions would provide no additional value to the understanding of an interaction [FMS08] but would result in unnecessarily complex scenarios due to additional lifelines for all ports.

Subsystem Levels Having fully specified the Top Level, we are now able to decompose the component architecture and refine the MSDs accordingly in order to traverse to the Subsystem Level. Therefore, we first define within the BDD System [Blocks] the new blocks DistanceSensor, OvertakeLogic, and EngineControl as well as composite associations from Car to them in order to specify their hierarchical relationship.

Afterward, we define within the BDD System [Interfaces] additional interfaces for the communication between the new components, that is, the interfaces DS2OL and OL2EC. Then we add ports typed by these interfaces to the blocks. For the delegated communication of the new components with their superordinate component Car, the interfaces Env2Car, C2C, and Car2Env from the Top Level can simply be reused.

Now we can define the Internal Structure View of Car. Therefore, we create an IBD Car and add the *parts* :DistanceSensor, :OvertakeLogic, and :EngineControl typed by the new blocks and interconnect them by means of connectors. Furthermore, the internal parts are connected with the superordinate Car component by means of *delegation connectors* (i.e., the plain lines between the ports of the diagram frame and the ports of the internal parts).

Finally, we can move to the Interaction View, where we have to refine the MSDs from the Top Level to reflect the decomposition of the component architecture. Therefore, we have to perform slight changes to the top level MSD OvertakeTopLevel as mentioned before: We use the so-called InteractionUse concept of UML2, which basically is a reference from one lifeline to another MSD. That is, we define a new MSD OvertakeCarLevel and add an InteractionUse to the lifeline car1:Car of the MSD OvertakeTopLevel from the Top Level that references OvertakeCarLevel (indicated by the label of the InteractionUse and the abstract syntax link refersTo). Afterward, we add the lifelines representing the parts :DistanceSensor, :OvertakeLogic, and :EngineControl in the Internal Structure View to OvertakeCarLevel. The interrelation between the messages of OvertakeTopLevel and of OvertakeCarLevel is established by so-called *gates*, which connect the messages toward and from the InteractionUse on the lifeline car1:Car in OvertakeTopLevel with the equally named messages in OvertakeCarLevel (i.e., obstacleDetected, requestOvertaking, ackOvertaking, and accelerate). In OvertakeCarLevel, these messages are then connected with the inner lifelines. Besides taking care of this "outer" behavior prescribed by the superordinate MSD, messages covering the internal behavior have to be added. Thus, we add a message carInFront from :DistanceSensor to :OvertakeLogic to specify that the sen-

sor subsystem informs the logic subsystem about the obstacle detection, and we add a message speedupEngine from :OvertakeLogic to :EngineControl to specify that the logic subsystem requests the engine to speedup.

Summarizing the subsystem level, we have now taken into account the decomposition of the component architecture as well as the refinement of the MSDs w.r.t. the new hierarchy level. In contrast to the approach of Atir et al. [AHKM08], we apply InteractionUses instead of PartDecompositions—which are in fact just an extension of InteractionUses—since only one PartDecomposition is allowed per lifeline, while multiple InteractionUses can be added to one lifeline. Thus, this enables to specify a one-to-many relation between one lifeline and multiple refining MSDs if desired. We currently do not verify comprehensively for consistency or the correctness of a refinement relation between superordinate and subordinate MSDs as in [AHKM08], what is future work.

4.1.2 Modeling Rules

The models containing the hierarchical component architectures and the scenario-based requirements outlined in the last section can get quite complex. Furthermore, there are several ways to specify an aspect by means of UML2/SysML. To execute the models by means of play-out, valid models are needed. This model validity is influenced by their completeness and correctness. In order to guide the requirements engineer in specifying models that are valid for play-out, we have conceived modeling rules. To enable a direct feedback to the modeler in SCENARIOTOOLS, we formalized the modeling rules by means of the Object Constraint Language (OCL) [Obj12]. In the following, we will present a small excerpt of these modeling rules in an exemplary manner.

There are several modeling rules that aim at the completeness and correctness of the component architectures. On the one hand, this encompasses some simple rules checking that all types and interfaces are assigned to the Internal Structure View (e.g., the abstract syntax links type from the collaboration role car1:Car to the block Car and from the block port :C2C to the interface C2C) and all ports are connected. On the other hand, this encompasses more complex rules checking for the compatibility of all connected ports in terms of their provided and required interfaces (cf. [Obj11a, Sect. 8.3.4]) and checking that all required interfaces are "satisfied" by features of possibly several provided interfaces. For example, the port :~C2C of the collaboration role car1 is compatible with the opposite port :C2C of car2, and all its required operations are provided by the opposite port.

Furthermore, we check for the completeness of MSDs and their correct correlation to the component architectures. Besides some simple rules checking for the plain assignments of roles and parts to lifelines as well as connectors and interface operations to messages, there are again more complex rules. One of these rules demands that a connector assigned to a message must connect the two roles or parts that represent the sending and receiving lifeline, respectively. For example, the connector car1_2_car2 assigned to the message requestOvertaking that is sent from the lifeline car1 to car2 connects the corresponding roles. Furthermore, the sending direction of a message w.r.t. the direction of the assigned connector—determined by the interfaces of the connected ports—is checked. For instance, the interface of the end port of the connector car1_2_car2 correctly provides the corre-

sponding operation requestOvertaking(), such that the message can indeed be sent from car1 to car2. Finally, also some aspects regarding hierarchy level spanning messages of superordinate and subordinate MSDs (e.g., correct connection via gates and temperature) are checked.

4.2 Component-based Play-out

In order to take hierarchical component architectures as described in the last section into account, we adapted the play-out algorithm of SCENARIOTOOLS that is based on class models. From a technical point of view, the implementation of our simulation approach is similar to the one described in [BGP13, GF12]. From a conceptual point of view, two main aspects regarding the component architectures had to be considered.

The first aspect is that play-out has to operate with component-based architecture concepts such as ports and connectors as described in the last section. The main difference between the conventional object-based play-out is the way messages between entities are handled. In object-based play-out, an event directly relates to the sending and the receiving object. There is no restriction on how events between two objects are delivered. In contrast, in component-based play-out events between components can be exchanged only via predefined and directed routes consisting of assembly and delegation connectors.

The second aspect refers to the relation between a superordinate and its subordinate MSDs and can be comprehended by considering the play-out of OvertakeTopLevel and OvertakeCarLevel in combination:

1. The activation of a subordinate MSD takes place as soon as an event occurs that triggers an InteractionUse and thus the referred MSD. For example, when obstacleDetected occurs, then OvertakeCarLevel is activated.

2. A cut spanning hierarchy levels cannot be correlated to exactly one MSD anymore, but is determined by considering both a superordinate and a subordinate MSD together. For example, if the cut is between obstacleDetected and requestOvertaking in OvertakeTopLevel, then it is additionally either between obstacleDetected and carInFront or between carInFront and requestOvertaking in OvertakeCarLevel (cf. the visualized cuts in Fig. 2).

4.3 Summary

We evaluated the modeling approach, the modeling rules, and the adapted play-out algorithm by means of a proof of concept. Therefore, we specified several comprehensive models that were the basis of the running example we presented in excerpts in this paper. Based on these models, we investigated the conceptual differences to conventional play-out as described in the last section. Furthermore, we discovered that the modeling rules provide much value in order to specify valid models that are executable in play-out.

However, we also identified that the usability to refine an MSD is not high, what is mainly related to usability issues of SCENARIOTOOLS' underlying modeling tool suite and to an inconvenient UML2 metamodel w.r.t. InteractionUses. But, we believe that some of these refinement steps can be automated in order to improve the usability.

5 Conclusion and Future Work

In this paper, we presented a concept for the specification and simulative validation of scenario-based requirements based on hierarchical component architectures. We use SysML BDDs and IBDs for the specification of the component architectures and MSDs for the formal specification of scenario-based requirements. Furthermore, we guide the requirements engineer in specifying component architectures and scenarios that are valid for the requirements simulation by providing modeling rules formalized by means of OCL. We adapted the MSD play-out approach of SCENARIOTOOLS to work w.r.t. the SysML BDDs and IBDs. The evaluation of the approach by means of a proof of concept yielded that it is applicable and that the modeling rules provide valuable means for guiding the requirements engineer in specifying combined structural and scenario models that are valid for the adapted play-out algorithm. Summarizing, this new approach enables an integrated RE and component architecture design across multiple hierarchy or abstraction levels, as demanded by the automotive-specific process models [Aut10, Int11].

However, some issues were identified that restrict the usability w.r.t. the modeling of MSD refinement. Besides taking care of these open issues, future work encompasses several aspects. First of all, we want to investigate a refinement relation of scenarios across the hierarchy levels induced by the component architecture. The consistency rules for LSC-trees and part scenarios identified in [AHKM08] will serve as starting point for this. Second, the approach should be extended by the possibility to apply play-out on selected levels of the component hierarchy. Therefore, the guided simulation of [GF12] could be adapted. Third, we want to extend the MSD language to cover more aspects specific to automotive systems. This comprises the transition to AUTOSAR and thus its different communication patterns (i.e., client-server and sender-receiver), more detailed timing information (cf. [Obj11b]), and corresponding analysis techniques. Finally, we want to further evaluate the extended approach within the context of automotive systems development.

Acknowledgments

The authors would like to thank the student project group SafeBots III and particularly its RE subgroup for conceiving and implementing the concepts presented in this paper, namely Christopher Brune, Simon Schwichtenberg, and Dimitar Shipchanov. Furthermore, the authors would like to thank Marcel Sander for productive discussions on some theoretical aspects of this work.

This research and development project is funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL) and managed by the Project Management Agency Karlsruhe (PTKA). The authors are responsible for the contents of this publication.

References

[AHKM08] Yoram Atir, David Harel, Asaf Kleinbort, and Shahar Maoz. Object Composition in Scenario-Based Programming. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316. Springer, Berlin/Heidelberg, 2008.

[Aut10] Automotive Special Interest Group (SIG). Automotive SPICE: Process Reference Model – Release v4.5, 2010.

[BGP13] Christian Brenner, Joel Greenyer, and Panzica La Manna, Valerio. The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions. In *12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, 2013.

[CHK08] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using live sequence charts and the Play-Engine tool. *Software and Systems Modeling*, 7(2), 2008.

[CSVC11] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[FMS08] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML – The Systems Modeling Language*. Elsevier/Morgan Kaufmann, Burlington, Massachusetts, USA, 2008.

[GF12] Joel Greenyer and Jens Frieben. Consistency Checking Scenario-Based Specifications of Dynamic Systems by Combining Simulation and Synthesis. In *Proceedings of the Fourth Workshop on Behaviour Modelling – Foundations and Applications*, BM-FA '12, New York, NY, USA, 2012. ACM.

[GFDK09] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20(4):201–223, 2009.

[GHN10] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science (LNCS)*, pages 555–579. Springer, Berlin/Heidelberg, 2010.

[HKM07] David Harel, Asaf Kleinbort, and Shahar Maoz. S2A: A Compiler for Multi-modal UML Sequence Diagrams. In MatthewB Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science (LNCS)*, pages 121–124. Springer, Berlin/Heidelberg, 2007.

[HM03] David Harel and Rami Marelly. *Come, let's play: Scenario-based programming using LSCs and the play-engine*. Springer, Berlin/Heidelberg, 2003.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7:237–252, 2008.

[HMSB10] David Harel, Shahar Maoz, Smadar Szekely, and Daniel Barkan. PlayGo: Towards a Comprehensive Tool for Scenario Based Programming. In *25th IEEE/ACM International Conference on Automated Software Engineering 2010 (ASE '10)*, pages 359–360, 2010.

[Int11] International Organization for Standardization. ISO 26262: Road Vehicles – Functional Safety, 2011.

[LM09] David Lo and Shahar Maoz. Mining Hierarchical Scenario-Based Specifications. In *24th IEEE/ACM International Conference on Automated Software Engineering 2009 (ASE '09)*, pages 359–370, 2009.

[Mod12] Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles (MAENAD) project. EAST-ADL Domain Model Specification – Version M2.1.10.20120629, 2012.

[Obj03] Object Management Group. MDA Guide – Version 1.0.1, 2003.

[Obj10] Object Management Group. OMG Systems Modeling Language (OMG SysML) – Version 1.2, 2010.

[Obj11a] Object Management Group. OMG Unified Modeling Language (OMG UML) Superstructure – V2.4, 2011.

[Obj11b] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems – Version 1.1, 2011.

[Obj12] Object Management Group. OMG Object Constraint Language (OCL) – Version 2.3.1, 2012.

[SDP10] Ernst Sikora, Marian Daun, and Klaus Pohl. Supporting the Consistent Specification of Scenarios across Multiple Abstraction Levels. In Roel Wieringa and Anne Persson, editors, *REFSQ*, volume 6182 of *Lecture Notes in Computer Science (LNCS)*, pages 45–59. Springer, Berlin/Heidelberg, 2010.