Considering Architectural Properties in Real-time Play-out

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

Dimitar Shipchanov
Department of Computer Science, University of Paderborn
Warburger Str. 100, 33098 Paderborn, Germany
mitko@mail.upb.de

Abstract: Real-time embedded systems (RTES), as in the automotive domain, provide their functionality by executing software operations on hardware with restricted resources and by communicating via buses. The properties of the underlying architecture, i.e., execution times of software operations and bus latencies, cause delays during the provision of the functionality. At the same time, RTES have to fulfill strict real-time requirements. The fulfillment of such real-time requirements under consideration of delays induced by architectural properties should be taken into account already during requirements engineering (RE) to avoid costly iterations in subsequent development phases. In previous work, we developed a formal 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. Our MSD play-out approach considers assumptions on the environment as well as real-time requirements and is applicable to hierarchical component architectures, which makes it well suited for automotive systems. However, delays induced by architectural properties are not considered. In order to consider this important aspect, we introduce in this paper an approach enabling the annotation of software operation execution times and connector latencies to hierarchical component architectures by means of the MARTE profile. These assumptions about the architectural properties can be verified against the real-time requirements specified in the MSDs by means of simulation. We illustrate the approach by means of an example of an automotive RTES.

1 Introduction

The growing functionality of real-time embedded systems (RTES) in modern vehicles has led to thousands of software operations distributed over 100 electronic control units (ECUs) that communicate via multiple bus systems [VPBK10]. The ECUs executing the software operations have restricted resources leading to best and worst case execution times, and the buses delivering messages for the purpose of communication have latencies. These *architectural properties* cause *delays* during the provision of the RTES functionality. At the same time, RTES have to fulfill strict real-time requirements. The fulfillment of

such real-time requirements under consideration of delays induced by architectural properties should be taken into account as early as possible in the development process, particularly in the requirements engineering (RE) phase. This is due to the fact that the detection and fixing of defects in the system under development (SUD) in subsequent development phases cause costly development iterations (e.g., [Poh10]).

In previous work, 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)* [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]. Play-out enables to execute MSD/LSC specifications, which allows the requirements engineers to understand the behavior emerging from the interplay of the scenarios. Our MSD play-out approach implemented in the SCENARIOTOOLS[1] tool suite considers assumptions on the environment [BGPLM13] as well as real-time requirements [BGH+14] and is applicable to hierarchical component architectures [HM13], which makes it well suited for automotive systems.

However, delays induced by architectural properties as mentioned above are not considered. In order to consider this important aspect, we introduce in this paper an extended real-time play-out approach. Our contribution is twofold. First, we enable the annotation of architectural properties to hierarchical component architectures that serve as structural basis for the MSDs [HM13]. To do so, we annotate execution times to software operations and latencies to communication connectors by means of the UML2 profile Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [Obj11b]. Second, we extend our MSD real-time play-out approach [BGH+14] to investigate delays induced by these architectural properties in the simulation of scenarios. This enables to simulatively verify the assumptions about the architectural properties against the real-time requirements specified in the MSDs.

We illustrate the approach by means of an automotive ECU that controls body and comfort functions of a car, a so-called Body Control Module (BCM). A BCM centrally controls distributed vehicle functions like central locking, turn signals, and brake light and has to interact with several other ECUs to accomplish this task. We focus on one scenario related to the central locking functionality. We assume that the BCM has locked all doors after the car has been started and a certain velocity threshold has initially been reached. In the hazard situation that a crash is detected, the BCM has to unlock all doors within a time limit after a crash was detected, so that all passengers can escape or can be rescued from outside.

This paper is structured as follows. Sect. 2 introduces the fundamentals of MSDs, their structural basis, their real-time requirements, and our real-time play-out approach. Sect. 3 presents the extended modeling and simulation approach. Sect. 4 covers related work. Finally, Sect. 5 summarizes this paper and provides an outlook on future work.

[1] http://scenariotools.org/

2 Foundations

In this section, we introduce component-based MSDs (Sect. 2.1), their basic semantics (Sect. 2.2), and their extensions to specify real-time requirements (Sect. 2.3). In Sect. 2.4, we introduce real-time play-out for the simulative validation of timed MSDs. For illustration purposes, we use the running example of the BCM, which has to unlock all doors in a hazard situation. The particular models are shown in Fig. 1, which we explain step by step in the following subsections. In general, the figure describes the requirement that when a crash is detected by the CrashSensor, the BCM has to send an `unlock` command to the DoorLock and receive a corresponding acknowledgement message within 70 time units.

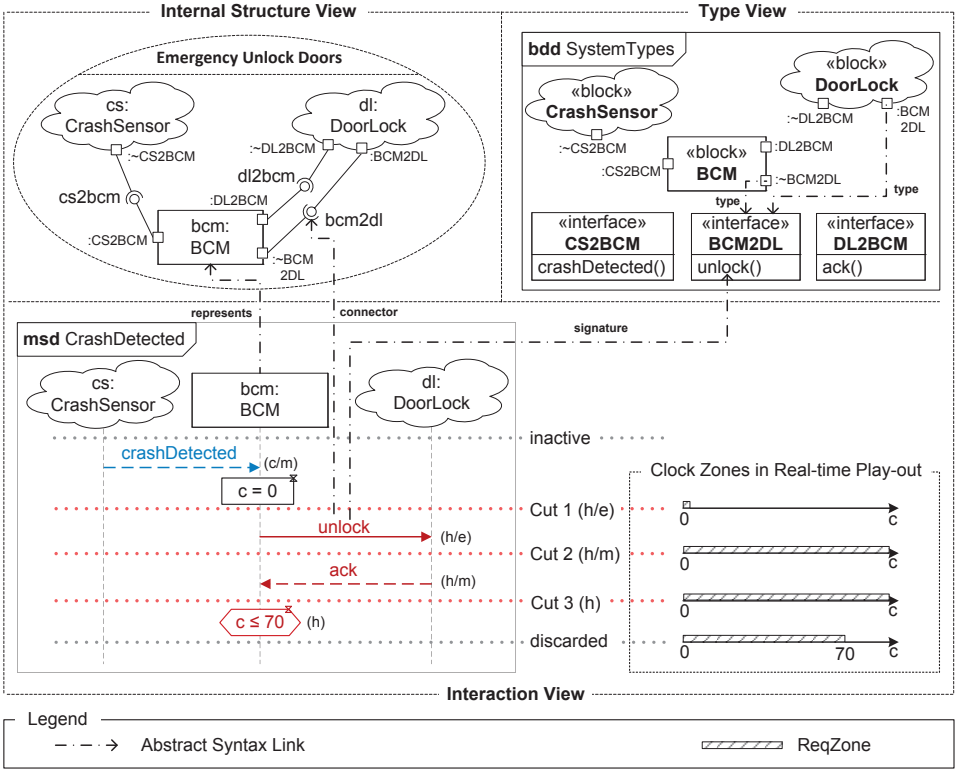

Figure 1: Component-based MSD with clock zones in real-time play-out

2.1 Hierarchical Component Architectures as Structural Basis for MSDs

In [HM13], we introduced hierarchical component architectures as structural basis for MSDs. We use a subset of the structural models of the Systems Modeling Language (SysML) [Obj10] and the UML2 to specify components, ports, and connectors as structural basis of the SUD.

In the Type View, we define component types and interfaces using a SysML Block Definition Diagram System Types, as shown in the upper right corner of Fig. 1. We distinguish between *system (controllable) components* and *environment (uncontrollable) components*. The set of all system components represents (subsystems of) the SUD and the set of all environment components stands for the environment. In Fig. 1, the cloud-shaped elements CrashSensor and DoorLock represent environment component types and BCM represents a system component type. The particular ports of the component types have provided and required interfaces to specify directed message exchange. For example, the port :BCM2DL of DoorLock provides the interface BCM2DL that is required by the port :~BCM2DL of BCM (the ~ specifies that the direction of the interface typing the port is reversed, therefore it is a required interface) [Obj11a]. Thus, BCM and DoorLock have a compatible required and provided interface for sending and receiving the message `unlock`, respectively.

In the Internal Structure View, we use a UML2 Collaboration Diagram Emergency Unlock Doors to specify the actual top-level component-based architecture, which is shown in the upper left corner of Fig. 1. We specify the directed assembly connector cs2bcm from the component cs:CrashSensor to bcm:BCM, and the two assembly connectors dl2bcm and bcm2dl between the components bcm and dl:DoorLock for a bidirectional communication. The latter one connects the aforementioned ports to allow the sending of the message `unlock` from bcm to dl. Note that we do not consider multiple hierarchy levels of component architectures in this paper as in [HM13] due to space restrictions.

2.2 Basic MSD Semantics

Based on the component architecture described in the last subsection, the MSD CrashDetected can be specified in the Interaction View (lower left part of Fig. 1). An MSD basically consists of *lifelines* and *messages*. A lifeline represents a participant in the component architecture (e.g., the lifeline/component bcm:BCM). Messages, represented by arrows between lifelines, define requirements on the communication between the components. Messages have a *temperature* and an *execution kind*. The temperature is used to specify provisional (*cold*) or mandatory (*hot*) behavior. For a cold message, a different message specified by the MSD may occur at this point in time.Cold messages are visualized with blue color and are annotated with (c), e.g., `crashDetected`. The semantics of a hot message is that other messages specified by the MSD must not occur at this point in time. Hot messages are shown in red and are annotated with (h), e.g., `unlock` and `ack`.The execution kind determines whether a message is *monitored*, visualized with a dashed arrow and annotated with (m), or *executed*, represented by a solid arrow and annotated with (e). The semantics is that monitored messages can but do not need to occur, while executed messages must occur eventually. In Fig. 1, the messages `crashDetected` and `ack` are monitored and the message `unlock` is executed.

The actual sending or receiving of a message at runtime or during play-out is called an *event*. A message is *unifiable* with an event iff the sending and the receiving lifelines of the message represent the same sending and receiving components of the event, and the event name is the same as the message operation name. Messages that have no preceding

message (i.e., `crashDetected`) are called *minimal messages*. An MSD proceeds when events occur that are unifiable with its messages and becomes *active* if its minimal message is unifiable with the event currently being executed. A *cut* is the current state of the lifeline locations in an active MSD. A location is a point on a lifeline where a message is sent/received. If the cut is immediately before a message, the message is called *enabled*. If any of the enabled messages in the active MSDs is hot, then the cut is hot. Otherwise, it is cold. If at least one enabled message is executed, the cut is executed, otherwise it is monitored. In Fig. 1, Cut 1 is hot and executed, as indicated by the annotation (h/e). Cut 2 is hot and monitored (h/m), and Cut 3 is hot (h). If the cut reaches the end of the diagram, the active copy of the MSD is discarded.

A violation occurs if the currently executed event is unifiable with a message that is specified in an active MSD but is not enabled. There are two outcomes depending on the temperature of the cut. If the cut is cold, a cold violation occurs. This is a legal trace, and active MSD copy is discarded. If the cut is hot, this is a *safety violation*, which must never happen. An executed cut must eventually progress, otherwise it is considered a *liveness violation*.

2.3 Real-time Requirements in MSDs

In [BGH+14], we introduced modeling constructs for the specification of real-time requirements to MSDs. In Fig. 1, these modeling constructs have an additional hour-glass icon. We base on the concept of *clock variables* [AD94], i.e., real-value variables that increase synchronously and linearly with time. A *clock reset* sets the value of a clock to zero, as shown by the c=0 assignment in the rectangle after message `crashDetected` in Fig. 1. *Time conditions* are expressions over a clock variable that evaluate to a Boolean value and are represented by hexagons covering one or more lifelines. They have the form $x \bowtie expr$, where x is a clock variable, $expr$ is an expression evaluating to an integer value, and $\bowtie$ can be any operator from $<, \leq, \geq, >$. Time conditions also have a temperature. Enabled cold time conditions are evaluated immediately. If they evaluate to false, a cold violation occurs. Otherwise, the cut progresses. In contrast, a cut can only progress beyond an enabled hot time condition if it evaluates to true. If an event occurs that would let the cut progress beyond a hot time condition that evaluates to false, a safety violation occurs. If it never evaluates to true, a liveness violation occurs. In the MSD shown in Fig. 1 the hot time condition $c \leq 70$ is depicted, which is also called a *maximal delay*. Thus, the real-time requirement of the MSD specifies that bcm has to send the `unlock` command to dl and receive the corresponding acknowledgment within 70 time units after it received a `crashDetected` message.

2.4 Real-time Play-out

The play-out algorithm allows the behavior of the system as specified in MSDs to be simulated, thus enabling the validation of the reaction of the system on environment or user inputs. At the beginning of the simulation, the algorithm waits for an environment

event to occur. When it occurs, MSDs are activated whose initial event is unifiable with the environment event. Next, play-out chooses non-deterministically one of the enabled system events and executes it. This operation is called a step. After the event has been executed, the cut progresses and a new list of enabled messages is created. The process is repeated until there are no active MSDs with executed cut left. The system then waits for the next environment event. In case of a hot violation the algorithms terminates.

In [BGH+14], we introduced a real-time play-out algorithm, which is based on *clock zones* [Dil90, BY04]. A clock zone is the solution set of a clock constraint, where the clock is a real-valued variable [BY04]. The clock zone can be interpreted as a symbolic state, which represents an infinite number of clock values that satisfy the conditions applied over one or more clock variables.

Real-time play-out uses a clock zone to keep track of the time progress, implicitly specified by clock resets and time conditions, during its execution. In this paper, we call this clock zone the *requirements clock zone (ReqZone)*. The ReqZone is associated with each MSD cut, as shown in the lower right corner of Fig. 1. The valuations of clock c are shown in the time lines of the corresponding cut. In Cut 1, the MSD is activated and clock c is reset to zero. Once the `unlock` message is sent, time starts to progress and the clock can have arbitrary values, as indicated by the time line corresponding to Cut 2. After message `ack` in Cut 3, there are no further constraints for clock c, so its value can still be arbitrary. Finally, we assume that `ack` occurs within 70 time units and hence the time condition is fulfilled. Therefore, after evaluating the time condition $c \leq 70$ to true, the upper bound of the clock is modified accordingly. If `ack` would occur later than 70 time units, a liveness violation would occur.

3 Verifying Architectural Properties Against Scenario-based Real-time Requirements

Distributed RTES provide their functionality by executing software operations on distributed hardware with restricted resources and by communicating via buses. For example, in a car, software components are deployed on different ECUs, which need time to process requests and communicate via different kinds of bus systems with differing communication latencies. These architectural properties cause delays during the actual provision of the SUD's functionality and have to be verified w.r.t. real-time requirements.

Therefore, we consider two sets of timing constraints in our extended real-time play-out approach: Real-time requirements specified by means of timed MSDs and assumptions about architectural properties affecting the timing behavior annotated to the component architecture. Based on this information, we compare and analyze the two sets of timing constraints and check whether the proposed solution, i.e., the assumptions about the architectural properties, solves the problem, i.e., the real-time requirements. We postulate that the assumptions about the architectural properties stem from experiences with earlier projects and are quite coarse-grained. Therefore, our approach is not designed to replace state-of-the-art timing analysis tools in subsequent development phases. Instead, it aims at

enabling to identify and resolve possible requirements violations or defects already in the RE phase.

First, in Sect. 3.1, we show how the assumptions about architectural properties are specified in the component architecture. In Sect. 3.2, we present how delays stemming from the architectural properties influence the time progress in real-time play-out. Finally, in Sect. 3.3, we show how to compare this time progress in the simulation with the real-time requirements in MSDs and introduce rules for this comparison. Fig. 2 revisits the models of Fig. 1 and sketches our extended approach.

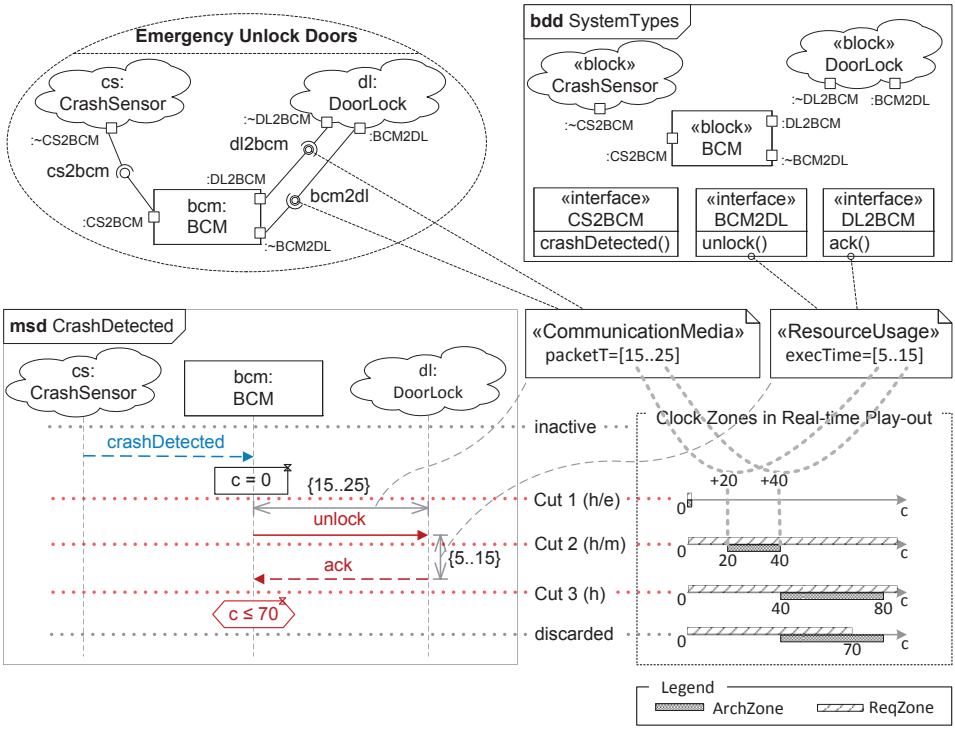

Figure 2: Annotated architectural properties considered in real-time play-out

3.1 Annotating Assumptions about Architectural Properties to Hierarchical Component Architectures

We use a subset of the MARTE profile [Obj11b] to annotate assumptions about architectural properties affecting the SUD's timing behavior to the component architectures that serve as structural basis for MSDs. More specifically, we annotate *communication latencies* to assembly connectors and *software operation execution times* to operations specified in the component interfaces. Both kinds of architectural properties are specified by bounded or half-bounded intervals. This avoids to awkwardly investigate particular con-

crete values in the simulation and enables to consider the whole range between a minimal and maximal value (cf. Sect. 3.3).

In our approach, assembly connectors represent communication media with various physical characteristics. For the analysis of distributed RTES, we are interested in the communication delay between two interacting components caused by the latency of the communication media. For this purpose, we annotate assembly connectors with the communication latency in the architectural model. Fig. 2 shows how the latency is specified using the «CommunicationMedia» stereotype with the attribute packetT from MARTE. It specifies the time needed to transmit a request via this connector. For example, the message `unlock` is sent from bcm:BCM to dl:DoorLock via the connector bcm2dl. The communication latency is between 15 and 25 time units, as specified by the annotation and indicated by the horizontal duration constraint of the message `unlock` in the MSD.

The other cause for delays that we consider are the software operation execution times of RTES. We apply the execTime attribute from the «ResourceUsage» stereotype in MARTE for the annotation of this property. As defined in [Obj11b], it specifies the time a component, whose port implements this interface, will be in use due to usage. Components interact using ports and exchange messages as specified in interfaces. We annotate the software operations defined in an interface and consider their execution time if the interface is implemented as a required one by a component port. For example, the message `unlock` in Fig. 2 is associated with an operation from the BCM2DL interface. The execution time of this operation specifies that it will take between 5 and 15 time units for dl to process the request. This is also indicated by the vertical duration constraint from the receiving of the `unlock` message to the subsequent sending of the `ack` message by dl.

3.2 Considering Architectural Properties in Real-time Play-out

The communication between the components and the execution of software operations of a distributed RTES consume time, expressed using communication latencies and operation execution times. For each message exchanged between two entities there are two possible delay intervals, i.e., the *communication delay interval* and the *operation execution delay interval*. As shown in Fig. 2, there are two duration constraints for the message `unlock` indicating these delays. The communication delay interval is induced by the communication latency and the operation execution delay interval by the operation execution time. The communication delay interval specifies all possible message delays from a message sending event until its receiving event. The operation execution delay interval determines all possible time delays due to processing a message from the receiving event until the sending event of the subsequent message.

During play-out, the delay intervals are considered when an event occurs based on the annotated structural model. To capture and investigate the delay intervals related with the architectural properties we introduce an additional clock zone, called *architectural clock zone (ArchZone)*. Whenever a message with at least one delay interval is sent in real-time play-out, the ArchZone is modified accordingly. We increment the lower-bound constraints for all clock variables in the zone with the lower bounds from the intervals.

For example, in Fig. 2, when the message `unlock` is sent, we sum the lower bounds of the communication and operation execution intervals, i.e., $15 + 5 = 20$. Then, we increment the lower bound of the clock variable c in the ArchZone using that sum, as shown in the time line corresponding to Cut 2. Similarly, we increment the upper-bound constraints for all clocks using the upper bounds from the intervals, i.e., we add the upper bounds $25 + 15 = 40$ to the clock variable c in the ArchZone in Cut 2. The clock valuations for Cut 3 are computed analogously by summing up the communication delay interval of dl2bcm and the operation execution delay of `ack()`.

3.3 Comparing Architectural Time Progress with Real-time Requirements

The ReqZone and ArchZone are modified independently at the occurrence of an event and a time condition, respectively. After each such event in the simulation, we compare the two clock zones to verify whether the real-time requirements are violated by the architectural time progress. A clock zone can be interpreted as an infinite set of clock values that satisfies certain time constraints. Based on this set-oriented interpretation, we identify three possible outcomes for the comparison of the two zones.

1. The ArchZone is a subset of ReqZone: In this case, no matter what the clock valuations in ArchZone are, they will always be within the desired real-time requirements specified by the ReqZone. Therefore, we say that the zones are *consistent* and no violations can occur. This situation holds for the cuts 1 to 3 in Fig. 2.

2. The two zones are disjoint: Clearly, they will never be consistent, i.e., all clock values defined by ArchZone are not defined by ReqZone, and there will always be a violation of the real-time requirements when we consider the delays induced by the architectural properties. In this case, the semantics of the time condition temperature as presented in Sect. 2.3 hold.

3. The two zones intersect: In this case, both zones have clock values in common, but there are also values in the ArchZone that are not defined by the ReqZone. The intersection indicates that there could be an inconsistency between the zones that might lead to a violation of the real-time requirements. This is the case for the cut terminated in Fig. 2, where the ArchZone has the clock valuation $40 \leq c \leq 80$ and the ReqZone has the valuation $0 \leq c \leq 70$. In this case, it is up to the play-out user to decide whether to follow a pessimistic procedure (i.e., resolve the inconsistency since there could be a possible violation of a real-time requirement) or to judge whether the situation can occur in reality (i.e., possibly not to resolve the inconsistency).

Using this approach, we are able to detect inconsistencies between the assumptions about the architectural properties affecting the SUD's timing behavior and real-time requirements. In the context of the software development process, if the zones are inconsistent, this means that the real-time requirements in the specification are not satisfied by the architectural properties. There are two possible approaches to resolve the discrepancy. One is

to redesign the architecture of the system, so that the real-time requirements are satisfied. The other approach is to renegotiate the real-time requirements with the customer.

4 Related Work

Harel and Marelly [HM02] introduced the first version of the play-out algorithm for timed LSCs. Unfortunately, this timed play-out bases on a discrete time model, which is inconvenient for the specification of real-time systems since time has to be discretized a priori and time progress is influenced by discrete ticks (i.e., 71 particular ticks have to be performed to violate the time condition in our example). In contrast, we use a continuous time model, which aggregates infinitely many states only differing in concrete clock values to symbolic states (i.e., $c \leq 70$ and $c > 70$). Goknil et al. [GDPFM13] present an approach for the simulation of real-time requirements that are typical for automotive systems. Both approaches only consider requirements but neglect assumptions about architectural properties that affect the timing behavior. Therefore, they can only identify contradictory or under-specified real-time requirements but cannot verify whether the assumed architecture would fulfill the requirements.

Larsen et al. [LLNP09] present an approach for the formal verification of behavioral models in terms of timed automata w.r.t. real-time requirements specified in LSC scenarios. The approach bases on the UPPAAL model checker that also features simulation. The authors do not consider the architecture of the SUD. Maoz and Harel [MH11] visualize traces of reactive systems and thereby enable the user of their approach to inspect these traces in terms of high abstraction level requirements specified by means of MSDs. They do not focus on embedded systems and hence cover real-time requirements only in a very restricted manner. Lettrari and Klose [LK01] propose an approach to monitor and test real-time systems against real-time requirements specified by means of Message Sequence Charts. As our previous work on verifying traces of gear shift sequences against real-time MSD specifications [BGH+14], all of these approaches require behavioral design models or the fully implemented system as input for the verification against the requirements. Therefore, these approaches are only applicable in later development phases but not in RE.

5 Conclusion and Outlook

This paper presents an extension of our requirements modeling and simulation approach based on the scenario-based formalism of MSDs and implemented in the tool suite SCENARIOTOOLS. The extension enables to consider assumptions about architectural properties of the SUD affecting timing behavior and to verify whether they fulfill real-time requirements in real-time play-out. To achieve this objective, we extended on the one hand the modeling approach with the possibility to annotate architectural properties (i.e., operation execution times and connector latencies) to the hierarchical component architectures serving as structural basis for MSDs. We use a subset of the MARTE profile [Obj11b] for

these annotations. On the other hand, we extended real-time play-out to consider message delays (i.e., operation execution and communication delays) induced by the architectural properties and to verify them w.r.t. real-time requirements specified in MSDs. For the verification, we compare the time progress influenced by the message delays with the allowed time intervals specified by the real-time requirements.

Our extended approach enables to take assumptions about architectural properties affecting the timing behavior into account already in the RE phase. The simulation helps to identify possible violations of real-time requirements due to message delays induced by architectural properties. Therefore, possible requirements violations or defects can be identified early in the development process so that a different architecture can be chosen or the real-time requirements can be renegotiated with the customers. Thus, the approach helps to avoid costly iterations in subsequent development phases.

In the future, we plan to extend our approach into two different directions. On the one hand, our RE approach also covers the formal verification for the consistency of MSD specifications [GF12]. We plan to combine this formal verification approach with the extension presented in this paper in order to prove that the SUD, considering its architectural properties, does not violate real-time requirements instead of manually simulating all possible execution paths. On the other hand, we want to annotate assumptions about probabilistic architectural properties (e.g., quality of service aspects of bus systems) to the hierarchical component architecture and consider them in the requirements simulation.

Acknowledgments. 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 thank Thorsten Koch for helpful comments for the camera ready version of this paper.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[BGH+14] C. Brenner, J. Greenyer, J. Holtmann, G. Liebel, G. Stieglbauer, and M. Tichy. ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study. In *Proc.* 13^{th} *Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*, 2014.

[BGPLM13] C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions. In *Proc.* 12^{th} *Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, 2013.

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

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

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

[GDPFM13] A. Goknil, J. DeAntoni, M.-A. Peraldi-Frati, and F. Mallet. Tool Support for the Analysis of TADL2 Timing Constraints Using TimeSquare. In *Proc.* 18^{th} *Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 145–154, 2013.

[GF12] J. Greenyer and J. Frieben. Consistency Checking Scenario-Based Specifications of Dynamic Systems by Combining Simulation and Synthesis. In *Proc.* 4^{th} *Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 2:1–2:9. ACM, 2012.

[HM02] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc.* 10^{th} *IEEE Int. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems 2002 (MASCOTS 2002)*, pages 193–202. IEEE Computer Society, 2002.

[HM03] D. Harel and R. Marelly. *Come, let's play: Scenario-based programming using LSCs and the play-engine*. Springer, 2003.

[HM08] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *SoSyM*, 7:237–252, 2008.

[HM13] J. Holtmann and M. Meyer. Play-out for Hierarchical Component Architectures. In *Proc.* 11^{th} *Workshop Automotive Software Engineering*, volume P-220 of *LNI*, pages 2458–2472. Bonner Köllen Verlag, 2013.

[LK01] M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In *≪UML≫ 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 317–328. Springer, 2001.

[LLNP09] K. G. Larsen, S. Li, B. Nielsen, and S. Pusinskas. Verifying Real-Time Systems against Scenario-Based Requirements. In *FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 676–691. Springer, 2009.

[MH11] S. Maoz and D. Harel. On tracing reactive systems. *SoSyM*, 10(4):447–468, 2011.

[Obj10] Object Management Group. OMG Systems Modeling Language (OMG SysML): Version 1.2, OMG Document Number: formal/2010-06-01, 2010.

[Obj11a] Object Management Group. OMG Unified Modeling Language (OMG UML) Superstructure: V2.4, OMG Document Number: ptc/2010-11-14, 2011.

[Obj11b] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems: Version 1.1, OMG Document Number: formal/2011-06-02, 2011.

[Poh10] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010.

[VPBK10] K. Venkatesh P., M. Broy, and I. Krüger. Scanning Advances in Aerospace & Automobile Software Technology. *Proceedings of the IEEE*, 98(4):510–514, 2010.