Generating EAST-ADL Event Chains from Scenario-based Requirements Specifications

Thorsten Koch[1], Jörg Holtmann[1], and Julien DeAntoni[2]

[1] Project Group Mechatronic Systems Design, Fraunhofer IPT
Zukunftsmeile 1, 33102 Paderborn, Germany
{thorsten.koch,joerg.holtmann}@ipt.fraunhofer.de
[2] Univ. Nice Sophia Antipolis, I3S, UMR 7271 CNRS, Sophia Antipolis, France,
julien.deantoni@polytech.unice.fr

Abstract. Real-time software-intensive embedded systems complexity, as in the automotive domain, requires rigorous Requirements Engineering (RE) approaches. Scenario-based RE formalisms like Modal Sequence Diagrams (MSDs) enable an intuitive specification and the simulative validation of functional requirements. However, the dependencies between events occurring in different MSD scenarios are implicit so that it is difficult to find causes of requirements defects, if any. The automotive architecture description language EAST-ADL addresses this problem by relying on event chains, which make dependencies between events explicit. However, EAST-ADL event chains have a low abstraction level, and their relationship to functional requirements has seldom been investigated. Based on the EAST-ADL functional architecture, we propose to use its central notion of *event* to conciliate both approaches. We conceived an automatic transformation from the high abstraction level requirements specified in MSDs to the low abstraction level event chains.

Keywords: requirements engineering, embedded systems, automotive, scenario-based specification, EAST-ADL event chains

1 Introduction

The growing functionality and complexity of today's embedded software-intensive systems that are subject to real-time constraints, like in the automotive domain, require rigorous development processes. This is especially true for the requirements engineering (RE) phase, since the detection and fixing of defects in the system under development (SUD) in subsequent development phases cause costly iterations [15].

On the one hand, scenario-based notations are well suited for the specification of requirements due to their intuitive representation [10]. Scenarios describe sequences of events of tasks that the SUD has to accomplish [10]. In previous work, we conceived a scenario-based RE approach based on a recent Live Sequence Chart (LSC) [3] variant, so-called *Modal Sequence Diagrams (MSDs)* [8]. The scenario-based nature of MSDs enables a visual and intuitive specification of requirements. Furthermore, the underlying formal semantics allows validating the requirements by means of the Play-out algorithm, originally conceived for LSCs [9]. Our MSD Play-out approach implemented in the ScenarioTools[3] tool suite considers assumptions on the environment [2] as well as real-time constraints [1] and is applicable to hierarchical component structures [11], which makes it well suited for automotive systems.

On the other hand, the automotive architecture description language EAST-ADL allows the specification of particular events occurring in an automotive architecture [5]. The specification of so-called *event chains* causally relates these events to each other, which make dependencies between them explicit. Additional real-time constraints restrict the timing of the particular event occurrences. Furthermore, the formalization of EAST-ADL event chains and timing constraints [6, 7] has recently made possible their validation by means of simulation in the TimeSquare[4] tool suite [4].

However, EAST-ADL event chains only describe requirements on event occurrences of an automotive software architecture. Functional requirements are not in their scope, and the relationship to scenario-based requirements has not been investigated, yet. This missing link to functional requirements is problematic, because the explicit dependencies between the events have to be specified in a modeling notation with a very low abstraction level. Thus, the requirements engineer has to manually extract the information in scenario-based requirements and specify it again in an awkward manner by means of EAST-ADL event chains, which is time-consuming and error-prone.

In order to bridge the gap between both formalisms, we conceived an automatic model transformation from MSDs to EAST-ADL event chains using the EAST-ADL functional architecture as a common basis. In this paper, we present a mapping from MSDs to EAST-ADL event chains, which acts as a link between both formalisms throughout a functional architecture. This enables an intuitive specification of scenario-based requirements and reduces effort to obtain a low abstraction level specification by means of EAST-ADL event chains.

We illustrate the approach by means of an electronic control unit controlling vehicle body functions, named Body Control Module (BCM). In the considered use case, the BCM has to unlock all vehicle doors after a crash was detected such that all passengers can safely escape or can be rescued from outside.

This paper is structured as follows: The following section introduces the fundamentals of MSDs and EAST-ADL event chains. Sect. 3 presents the transformation approach. Sect. 4 covers related work. Finally, Sect. 5 summarizes this paper and provides an outlook on future work.

2 Foundations

In this section, we introduce relevant foundations for the understanding of this paper: some basic concepts of MSDs (Sect. 2.1) and the EAST-ADL event chains (Sect. 2.2). Both are illustrated on the running example.

[3] `http://scenariotools.org/`

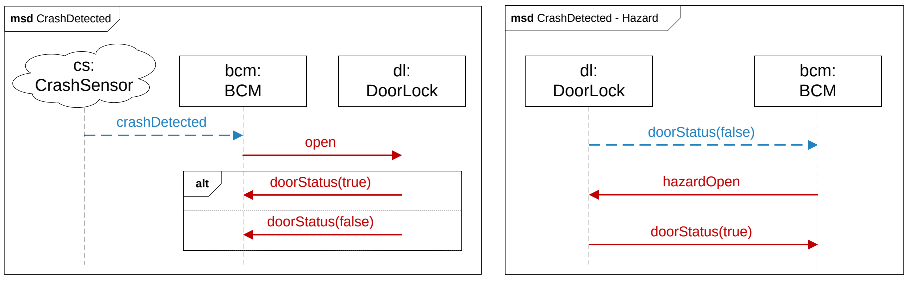
[4] `http://timesquare.inria.fr/`

2.1 Modal Sequence Diagrams

The MSD specification of our running example consists of the two MSDs CrashDetected and CrashDetected-Hazard, depicted in Fig. 1. The first MSD describes the requirements that the doors of the vehicle must be opened (message open) as soon as a crash has been detected (message crashDetected). The MSD CrashDetected-Hazard specifies the requirements that if the open operation fails (message doorStatus(false)), a hazard operation is performed (message hazardOpen) to ensure that the passengers of the vehicle can safely escape or can be rescued from outside.

Basically, an MSD consists of lifelines and messages. Lifelines describe structural entities, which can be distinguished into *environment objects* and *system objects*. Environment objects are depicted as cloud symbols and represent the environment that is sensed and manipulated by the SUD (e.g., lifeline cs:CrashSensor in Fig. 1a). System objects represent components of the SUD (e.g., lifelines bcm:BCM and dl:DoorLock in Fig. 1). Messages, represented by arrows between lifelines, define requirements on the communication between objects. Messages sent from environment objects are called *environment messages*, whereas messages sent from system objects are called *system messages*. They have a *temperature* and an *execution time*. The temperature of a message can be *cold* or *hot* visualized by blue and red arrows in Fig. 1. It is used to distinguish between provisional (cold) and mandatory (hot) behavior. The semantics of a hot message is that other messages specified by the MSD are not allowed to occur at this point in time, while for a cold message, other messages may occur [2]. The execution kind of a message can either be *executed*, depicted by solid arrows, or *monitored* depicted by dashed arrows. An executed message indicates that the message must eventually occur, whereas a monitored message can but need not to occur [2]. The MSD CrashDetected contains an *alternative fragment*, which describes different alternative continuations of the scenario.

The scenario-based nature of MSDs enables a high-level specification of requirements with separation of concerns. However, in big specifications the implicit event dependencies between several scenarios (e.g., message doorStatus(false)



(a) The MSD CrashDetected (b) The MSD CrashDetected-Hazard

Fig. 1. MSDs for the crash detection use case

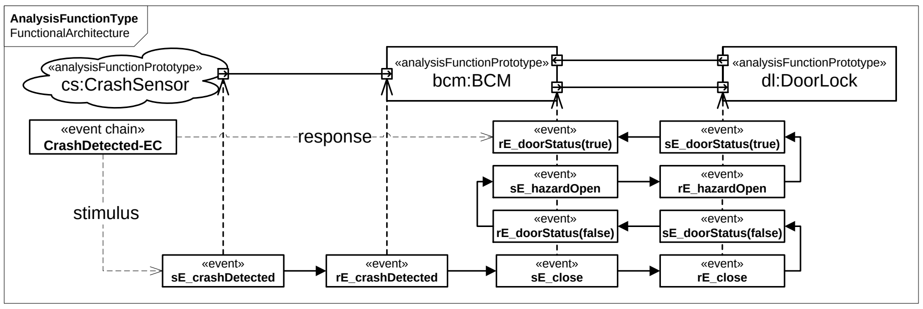

AnalysisFunctionType
FunctionalArchitecture
«analysisFunctionPrototype» cs:CrashSensor
«analysisFunctionPrototype» bcm:BCM
«analysisFunctionPrototype» dl:DoorLock
«event chain» CrashDetected-EC
response
stimulus
«event» rE_doorStatus(true)
«event» sE_doorStatus(true)
«event» sE_hazardOpen
«event» rE_hazardOpen
«event» rE_doorStatus(false)
«event» sE_doorStatus(false)
«event» sE_crashDetected
«event» rE_crashDetected
«event» sE_close
«event» rE_close

Fig. 2. The BCM example functional architecture in EAST-ADL including event chain for alternative (2) of MSDs in Fig. 1

in both MSDs) can complicate the investigation of requirements defects like an undesired activation of an MSD.

2.2 EAST-ADL Event Chains

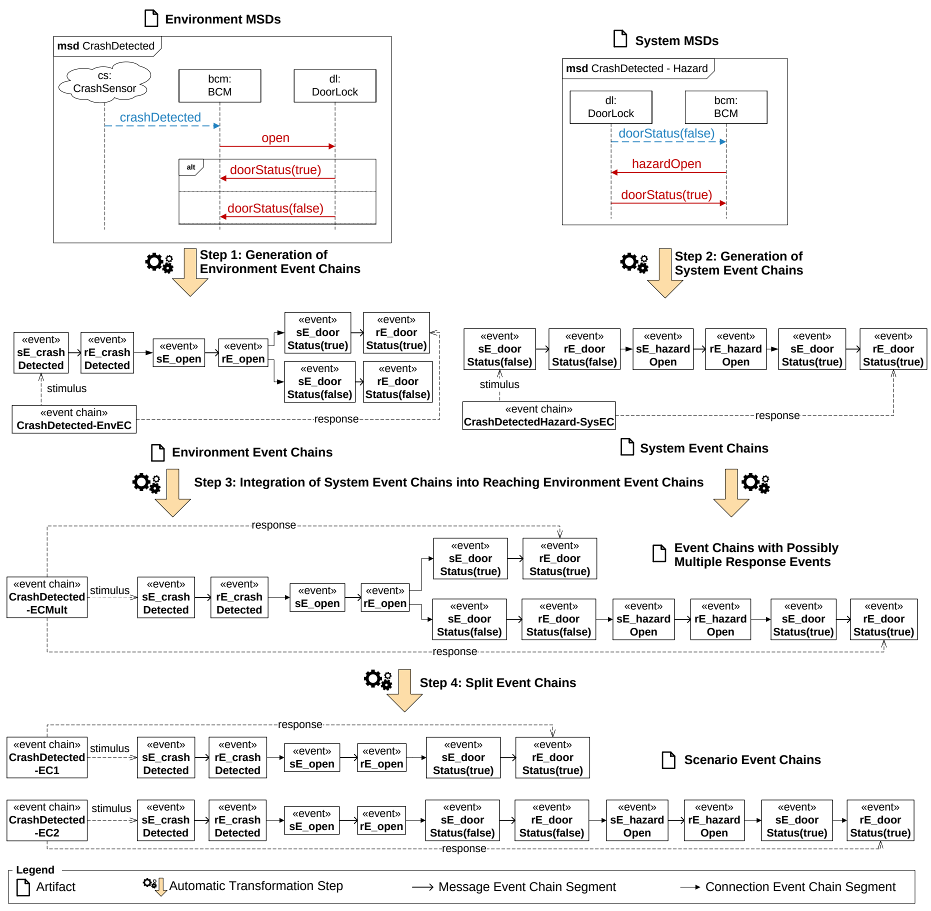
The *Electronics Architecture and Software Technology - Architecture Description Language* (EAST-ADL) is an architecture description language for automotive embedded systems [5]. The EAST-ADL provides a unified notion for all important engineering information including the functional and non-functional properties of the system.

In the EAST-ADL [5, Part VI], an *event* is the abstract representation of a specific system behavior that can be observed at runtime. An *event chain* describes the causal order for a set of functionally dependent events. Each event chain has exactly one stimulus and response event, which describe the start and end point of the chain. Furthermore, an event chain can be hierarchically decomposed into an arbitrary number of sub-chains, so-called *event chain segments* that also have exactly one stimulus and response event.

Fig. 2 depicts the EAST-ADL functional architecture of the running example. Furthermore, we add events and the event chain CrashDetected-EC to illustrate the same interaction as specified in alternative (2) of the MSD CrashDetected activating the second MSD. Obviously, the specification of all particular events has a lower abstraction level than the specification of message exchange within MSDs, but the event chain makes the dependencies between both scenarios explicit.

3 Transformation Approach

In this section, we present our transformation approach for the generation of EAST-ADL event chains from MSD specifications using the EAST-ADL functional architecture as common basis. The transformation approach has been implemented in the ScenarioTools tool-suite and covers the MSD messages, alternative fragments and real-time constraints. However, due to space limitations,



Environment MSDs
msd CrashDetected
cs: CrashSensor
bcm: BCM
dl: DoorLock
crashDetected
open
alt
doorStatus(true)
doorStatus(false)
System MSDs
msd CrashDetected - Hazard
dl: DoorLock
bcm: BCM
doorStatus(false)
hazardOpen
doorStatus(true)
Step 1: Generation of Environment Event Chains
Step 2: Generation of System Event Chains
«event» sE_crash Detected
«event» rE_crash Detected
«event» sE_open
«event» rE_open
«event» sE_door Status(true)
«event» rE_door Status(true)
«event» sE_door Status(false)
«event» rE_door Status(false)
stimulus
«event chain» CrashDetected-EnvEC
response
«event» sE_hazard Open
«event» rE_hazard Open
«event chain» CrashDetectedHazard-SysEC
Environment Event Chains
System Event Chains
Step 3: Integration of System Event Chains into Reaching Environment Event Chains
«event chain» CrashDetected -ECMult
Event Chains with Possibly Multiple Response Events
Step 4: Split Event Chains
«event chain» CrashDetected -EC1
«event chain» CrashDetected -EC2
Scenario Event Chains
Legend
Artifact
Automatic Transformation Step
Message Event Chain Segment
Connection Event Chain Segment

Fig. 3. Overview of the Transformation Approach

we do not detail the transformation of real-time constraints in this paper (please refer to [13] for more details about real-time constraint transformations).

Our transformation approach is implemented by means of QVT Operational [14] model transformations (partially supported by Java black-box libraries) and encompasses four steps, which are depicted in Fig. 3. In our approach, an EAST-ADL event chain is the description of the SUD's reaction to an environment message specified in an MSD. We call this type of event chain *scenario event chain*, which is the final result of our transformation (i.e., CrashDetected-EC1 and CrashDetected-EC2). The stimulus of a scenario event chain is always the sending event of an environment message. In the following, we describe each step by means of the running example.

Transformation Steps 1 and 2: For the first two steps, we divide the set of MSDs into *environment MSDs* and *system MSDs*. We qualify an MSD as an environment MSD, if its first message is an environment message; or as a system MSD, if its first message is a system message. These MSDs, representing a sequential order of messages, are respectively transformed into *environment event chains* and *system event chains*. For each MSD message, the transformation algorithm creates a *message event chain segment* by setting the sending event of the message as stimulus and the receiving event to the response.

Based on the running example, the transformation algorithm starts with the processing of the environment MSD CrashDetected. Therefore, it creates a new environment event chain CrashDetected-EnvEC and a message event chain segment for the first message crashDetected consisting of sE_crashDetected and rE_crashDetected.

The next element that occurs in the MSD is the open message. The transformation algorithm creates again a message event chain segment and in addition a *connection event chain segment*. A connection event chain segment preserves the order of two subsequent messages, e.g., crashDetected and open. Therefore, the stimulus of the connection event chain segment is set to the receiving event rE_crashDetected, and the response to the sending event of sE_open.

The next element that occurs in the MSD is the alternative fragment. MSD messages within the alternative fragment are transformed in the same way as other MSD messages. However, to preserve the order between the last message before the alternative fragment and the first message in each alternative, the transformation algorithm creates a set of connection event chain segments from rE_open to sE_doorStatus(true) and sE_doorStatus(false).

The two alternatives contain only one message, and thus, these messages are the last messages in the MSD. For a last message, the transformation algorithm has to consider two cases. First, if the message is not the first message in another MSD (e.g., doorStatus(true)), the currently considered alternative is terminated and the response event of the last message is added to the set of response events. Second, if the message is the first message in another MSD (e.g., doorStatus(false)), the transformation algorithm only marks the MSD as *reachable*. We call a system MSD *reachable*, if and only if its first message occurs in another processed MSD.

In the second step, the transformation algorithm processes the reachable system MSD CrashDetected-Hazard in the same manner, which results in the system event chain CrashDetectedHazard-SysEC.

Transformation Steps 3 and 4: In the third step, the transformation algorithm merges the system event chains with the event chains of the MSDs from which they are reachable. To accomplish this step, the event chain CrashDetected-EnvEC is first copied to a new event chain CrashDetected-ECMult. Afterwards, the event chain segments and the response events of CrashDetectedHazard-SysEC are attached to the event chain path that corresponds to the MSD message that has reached the MSD CrashDetected-Hazard (path containing sE_doorStatus(false) and rE_doorStatus(false)).

In Sect. 2.2, we stated that an event chain is only allowed to have one stimulus and one response event. However, in our running example, the event chain CrashDetected-ECMult contradicts this definition. Hence, in the fourth step, the transformation algorithm splits all event chains with multiple response events and creates a set of event chains; one for each response event (e.g., both occurrences of rE_doorStatus(true) in the event chain CrashDetected-ECMult). To accomplish this step, the transformation algorithm performs a backward search for each response event. After the completion of this transformation step, we obtain the two well-formed scenario event chains CrashDetected-EC1 and CrashDetected-EC2 that have exactly one stimulus and one response event.

After the application of the transformation approach, we can apply both simulation approaches in a complementary manner. On the one hand, requirements engineers can simulate the particular scenarios in ScenarioTools and investigate the behavior emerging from the interplay of multiple scenarios. On the other hand, they can simulate the resulting event chains within TimeSquare and visualize explicit event dependencies between different scenarios, enabling to detect requirements defects caused by undesired activations of MSDs.

4 Related Work

Chen et al. [17] propose a modeling approach for specifying timing requirements on the base of functional requirements. They have extended the Problem Frame formalism with the recent formalization [6, 7] of EAST-ADL event chains and timing constraints. The event chains and timing constraints have to be specified awkwardly in the underlying formalization, which is in contrast with our more intuitive representation of scenario-based requirements. Klein and Giese [12] present Timed Story Scenario Diagrams (TSSDs), a visual notation for scenario specifications that takes structural system properties into account. In TSSDs, it is possible to specify time constraints that allow setting lower and upper bounds for delays. There is no mention of analysis support for TSSDs. Priesterjahn et al. [16] present an automatic approach that generates a timed failure propagation model from a system model for fault tolerance analysis based on timed automata. The transformation is similar to our approach, but they focus on reliability, while we focus on timed requirements.

5 Conclusion and Outlook

In this paper, we presented a transformation approach from high abstraction level scenario-based requirements to low abstraction level event chains while using an EAST-ADL functional architecture as common basis. We apply MSDs as concrete formalism for scenario-based requirements and EAST-ADL as modeling notation for event chains. Our approach combines intuitive but formal scenario-based requirements specifications on a high abstraction level with the possibility to visually inspect explicit event chains induced by the scenarios.

The future work encompasses several aspects. On the one hand, we want to evaluate our approach and the opportunities w.r.t. real-time requirements in combining the two simulative validation approaches in a complementary manner. On the other hand, we want to reuse the EAST-ADL event chains in the subsequent software development process within Autosar.

Acknowledgments. This research is partially funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL) and is managed by the Project Management Agency Karlsruhe (PTKA). This work is also partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

1. 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 *Graph Transformation and Visual Modeling Techniques*, 2014.
2. C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *Graph Transformation and Visual Modeling Techniques*, 2013.
3. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.
4. J. DeAntoni and F. Mallet. TimeSquare: Treat your models with logical time. In *TOOLS (50)*, volume 7304 of *LNCS*, pages 34–41. Springer, 2012.
5. EAST-ADL Association. EAST-ADL Domain Model Specification: Version V2.1.12, 2013.
6. A. Goknil, J. DeAntoni, M.-A. Peraldi-Frati, and F. Mallet. Tool support for the analysis of TADL2 timing constraints using TimeSquare. In *ICECCS*, pages 145–154. IEEE, 2013.
7. A. Goknil, J. Suryadevara, M.-A. Peraldi-Frati, and F. Mallet. Analysis support for TADL2 timing constraints on EAST-ADL models. In *ECSA*, volume 7957 of *LNCS*, pages 89–105. Springer, 2013.
8. D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7:237–252, 2008.
9. D. Harel and R. Marelly. *Come, let's play: Scenario-based programming using LSCs and the play-engine.* Springer, 2003.
10. J. Hassine, J. Rilling, and R. Dssouli. An evaluation of timed scenario notations. *Journal of Systems and Software*, 83(2):326–350, 2010.
11. J. Holtmann and M. Meyer. Play-out for hierarchical component architectures. In 11th *Workshop Automotive Software Engineering (ASE 2013)*, volume P-220 of *LNI*, pages 2458–2472, 2013.
12. F. Klein and H. Giese. Joint structural and temporal property specification using timed story scenario diagrams. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 185–199. Springer, 2007.
13. T. Koch. Combining scenario-based and architecture-based timing requirements. Master's thesis, University of Paderborn, Paderborn, 2013.
14. Object Management Group. Meta object facility (MOF) 2.0 query/view/transformation specification: Version 1.1, OMG document number: formal/2011-01-01, 2011.
15. K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques.* Springer, 2010.
16. C. Priesterjahn, C. Heinzemann, and W. Schäfer. From timed automata to timed failure propagation graphs. In 4th *IEEE Workshop on Self-Organizing Real-time Systems (SORT 2013)*. IEEE, 2013.
17. Xiaohong Chen, Jing Liu, Frédéric Mallet, and Zhi Jin. Modeling timing requirements in problem frames using CCSL. In *APSEC*, pages 381–388, 2011.