Supporting the Consistent Specification of Scenarios across Multiple Abstraction Levels

Ernst Sikora, Marian Daun, and Klaus Pohl

Software Systems Engineering
Institute for Computer Science and Business Information Systems
University of Duisburg-Essen, 45117 Essen
{ernst.sikora,marian.daun,klaus.pohl}@sse.uni-due.de

Abstract. [**Context and motivation**] In scenario-based requirements engineering for complex software-intensive systems, scenarios must be specified and kept consistent across several levels of abstraction such as system and component level. [**Question/problem**] Existing scenario-based approaches do not provide a systematic support for the transitions between different abstraction levels such as defining component scenarios based on the system scenarios and the system architecture or checking whether the component scenarios are consistent with the system scenarios. [**Principal ideas/results**] This paper presents a systematic approach for developing scenarios at multiple abstraction levels supported by automated consistency checks of scenarios across these abstraction levels. [**Contribution**] We have implemented the consistency check in a tool prototype and evaluated our approach by applying it to a (simplified) adaptive cruise control (ACC) system.

Keywords: abstraction levels, consistency, interface automata, scenarios.

1 Introduction

Scenario-based requirements engineering (RE) is a well proven approach for the elicitation, documentation, and validation of requirements. In the development of complex software-intensive systems in the embedded systems domain, scenarios have to be defined at different levels of abstraction (see e.g. [1]). We call scenarios that specify the required interactions of a system with its external actors "system level scenarios" and scenarios that additionally define required interactions between the system components "component level scenarios". For brevity, we use the terms "system scenarios" and "component scenarios" in this paper.

Requirements for embedded systems in safety-oriented domains such as avionics, automotive, or the medical domain, must satisfy strict quality criteria. Therefore, when developing system and component scenarios for such systems, a rigorous development approach is needed. Such an approach must support the specification of scenarios at the system and component level and ensure the consistency between system and component scenarios. Existing scenario-based approaches, however, do not provide this kind of support.

R. Wieringa and A. Persson (Eds.): REFSQ 2010, LNCS 6182, pp. 45–59, 2010.

In this paper, we outline our approach for developing scenarios for software-intensive systems at multiple abstraction levels. This approach includes a methodical support for defining scenarios at the system level, defining component scenarios based on the system scenarios and an initial system architecture as well as detecting inconsistencies between system scenarios and component scenarios. Our consistency check reveals, for instance, whether the component scenarios are complete and necessary with respect to the system scenarios. To automate the consistency check, we specify system and component scenarios using a subset of the message sequence charts (MSC) language [2]. The consistency check is based on a transformation of the specified MSCs to interface automata [3] and the computation of differences between the automata or, respectively, the regular languages associated with the automata. We have implemented the consistency check in a prototypical tool and evaluated our approach by applying it to a (simplified) adaptive cruise control (ACC) system.

The paper is structured as follows: In the remainder of this section, we provide a detailed motivation for our approach. Section 2 outlines the foundations of specifying use cases and scenarios using message sequence charts. Section 3 briefly describes our approach for specifying system and component scenarios. Section 4 provides an overview of our technique for detecting inconsistencies between system and component scenarios. Section 5 summarises the results of the evaluation of our approach. Section 6 presents related work. Section 7 concludes the paper and provides a brief outlook on future work.

1.1 Need for Specifying Requirements at Different Abstraction Levels

Abstraction levels are used to separate different concerns in systems engineering such as the concerns of a system engineer and the concerns of component developers. We illustrate the specification of requirements at multiple levels of abstraction by means of an interior light system of a vehicle. At the system level, the requirements for the interior light system are defined from an external viewpoint. At this level, the system is regarded as a black box, i.e. only the externally visible system properties are considered without defining or assuming a specific internal structure of the system. For instance, the following requirement is defined at the system level:

– *R2 (Interior light system)*: The driver shall be able to switch on the ambient light.

The level of detail of requirement R2 is typically sufficient for communicating about the requirements with system users. However, for developing and testing the system, detailed technical requirements are needed. To define these detailed technical requirements, the system is decomposed into a set of architectural components and the requirements for the individual components and the interactions between the components are defined based on the system requirements. For instance, the following requirements are defined based on requirement *R2* (after an initial, coarse-grained system architecture has been defined for the interior light system):

– *R2.1 (Door control unit)*: If the driver operates the ‘Ambient’ button, the door control unit shall send the message LIGHT_AMB_ON to the roof control unit.
– *R2.2 (Roof control unit)*: If the roof control unit receives the message LIGHT_AMB_ON, it shall set the output DIG_IO_AMB to ‘high’.

1.2 Need for Checking Requirements Consistency across Abstraction Levels

If the component requirements define, for instance, an incomplete refinement of the system requirements, the integrated system will not meet its requirements even if each component satisfies the requirements assigned to it. For example, if the requirement *R2.2* in the previous section was omitted, the door control unit would send the activation signal to the roof control unit, yet the roof control unit would not be able to process this signal and hence it would not switch on the light.

If the system components are developed by separate development teams or even by separate organisations, specification errors such as inconsistencies between system and component requirements may remain hidden until very late stages of the development process, typically until system integration. To avoid rework during system integration caused by such defects (which often leads to project delays), requirements engineers must check early in the development process whether the component requirements are consistent with the system requirements. In addition, for safety-relevant systems, a proof must be provided that each component requirement is necessary (see e.g. [4]). The necessity of a component requirement can be shown by demonstrating that this requirement is needed to satisfy a system requirement.

1.3 Main Objectives of the Scenario-Based RE Approach

Based on the above considerations, the following objectives for a scenario-based approach can be defined:

- *O1: Specification of system scenarios*. The approach should support the specification of scenarios at the system level. For this purpose, it should provide guidelines defining what kind of information should be contained in the system scenarios. The way the system scenarios are specified should ease the transition to component scenarios as well as consistency checking.
- *O2: Specification of component scenarios*. The approach should support the specification of component scenarios based on the system scenarios and a coarse-grained architecture. For this purpose, it should provide guidelines defining what kind of information is added in the component scenarios. Furthermore, the approach should allow structuring the component scenarios differently from the system scenarios, for instance, to improve readability of the scenarios.
- *O3: Consistency checking of system and component scenarios*. The approach should support checking whether the externally visible system behaviour defined at the system level and the one defined at the component level conforms to a defined (possibly project-specific) consistency criterion. To support the removal of detected inconsistencies, the approach should provide a detailed account of all detected differences between the system and component scenarios.

2 Specification of Scenarios Using Message Sequence Charts

A scenario documents a sequence of interactions leading to the satisfaction of a goal of some agent (see e.g. [5]). Multiple scenarios associated with the same goal or set of

goals are typically grouped into use cases (see e.g. [6]). Scenarios can be documented using various formats such as natural language, structured templates, or diagrams.

To facilitate automated verification, we use message sequence charts (see [2]) for specifying and grouping scenarios, both, at the system and the component level. We decided to use message sequence charts since they are commonly known in practice and offer a standardised exchange format. The specification of scenarios using message sequence charts is outlined in this section. The formalisation of message sequence charts employed in our approach is outlined in Section 4.

2.1 Basic and High-Level Message Sequence Charts

The message sequence charts language defines basic message sequence charts (BMSCs) and high-level message sequence charts (HMSCs). The essential elements of a BMSC are instances and messages (see Fig. 1). The essential elements of a HMSC are nodes and flow lines. A node can refer to a BMSC or another HMSC. A flow line defines the sequential concatenation of two nodes. Therein, the sequential order of the nodes may contain iterations and alternatives. Formal definitions of the (abstract) syntax of BMSCs and HMSCs are given, for instance, in [7]. The graphical notation of BMSCs and HMSCs used in this paper is shown in Fig. 1.

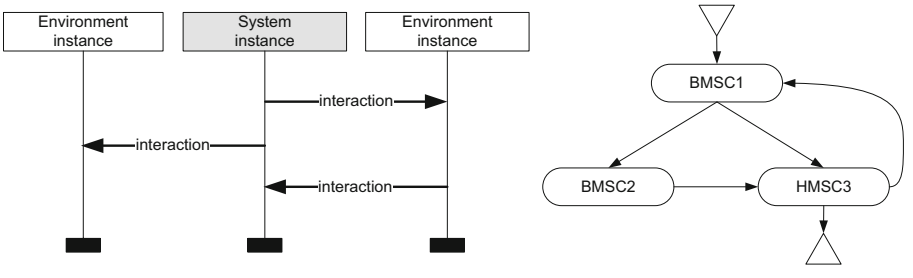

Fig. 1. Graphical notation of BMSCs (left-hand side) and HMSCs (right-hand side)

2.2 Specifying Use Cases and Scenarios Using Message Sequence Charts

Fig. 2 shows the documentation and composition of scenarios using message sequence charts as opposed to the documentation and grouping of scenarios by means of use case templates (see e.g. [6]). In our approach, we use message sequence charts in order to reduce the ambiguity caused by documenting scenarios and their composition using natural language. We use BMSCs for documenting atomic scenarios (or scenario fragments) and HMSCs for scenario composition. Therein, a HMSC can compose several scenario fragments into a larger scenario, several scenarios into a use case, or several use cases into a larger use case (Fig. 2, right-hand side). By using HMSCs, relationships between use cases such as “include” and “extend” as well as the sequential execution of use cases can be defined. A similar approach based on UML activity diagrams and UML sequence diagrams is described in [8]. Note that other information such as use case goals or pre- and post-conditions still need to be documented in the use case template.

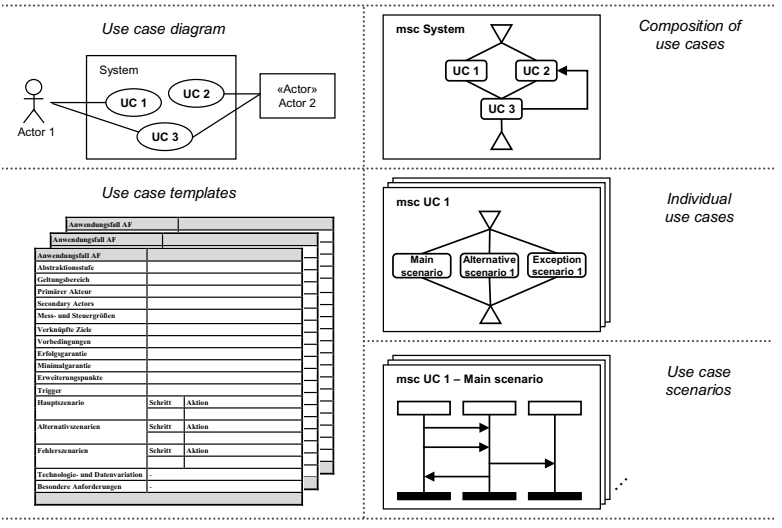

Fig. 2. Templates (left) vs. messages sequence charts (right) for documenting scenarios

3 Specification of Scenarios at Two Abstraction Levels

Our overall approach consists of three main activities (see Fig. 3). We outline these three activities and their major inputs and outputs in the following subsections.

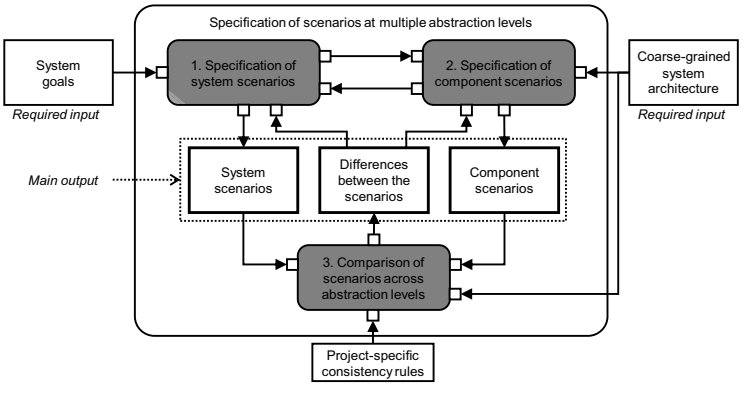

Fig. 3. Overall approach for specifying scenarios at multiple levels of abstraction

3.1 Specification of System Scenarios and Use Cases

The specification of system scenarios includes specifying individual system scenarios using BMSCs and interrelating the individual scenarios using HMSCs. Systems scenarios are identified and defined based on the goals of the external system actors. A system scenario should document the major interactions between the system and its

environment required to satisfy the associated goal. For each system goal, the relevant system scenarios satisfying this goal should be documented.

When specifying the system scenarios, the requirements engineers need to ensure that a black box view of the system is maintained. In other words, the specified system scenarios should only define the external interactions of the system and no internal interactions since defining the internal interactions is the concern of lower abstraction levels. Furthermore, system scenarios should be defined at a logical level, i.e. independently of a specific implementation technology such as a specific interface design or a specific system architecture. Fig. 4 shows a simple system scenario for the interior light system example introduced in Section 1.1.

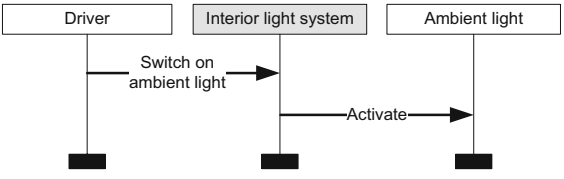

Fig. 4. Simple example of a system scenario

In early phases of use case development, the focus should be placed on the main or normal course of interactions that is performed to satisfy the use case goal(s). In later stages, alternative and exception scenarios should be added and related to the main scenario. However, in order to maintain the black box view, alternative and exception scenarios should be defined only if they are required independently of the internal structure of the system and the technology used for realising the system.

3.2 Specification of Component Scenarios and Use Cases

The definition of component scenarios comprises the definition of BMSCs and HMSCs at the component level. Typically, one starts defining component scenarios based on the defined system scenarios taking the (coarse-grained) system architecture into account. Furthermore, additional scenarios can be defined at the component level which are not based on the system scenarios such as scenarios for error diagnosis.

3.2.1 Definition of Component Scenarios Based on System Scenarios

A component scenario can define a possible realisation of a system scenario. For this purpose, the component scenario must define the interactions among the system components required to realise the interactions with the environment defined in the system scenario. The refinement of an individual system scenario is accomplished by the following steps (see e.g. [9]):

- *Step 1*: Identification of the components that are responsible for realising the system scenario. The instance representing the system in the system scenario is replaced by the identified set of components in the component scenario.
- *Step 2*: Each scenario step (i.e. each system-actor interaction) defined in the system scenario is assigned to a component. Therein, either the names of the scenario steps defined in the system scenario remain unchanged, or a unique mapping between

the message names is established. This is a necessary condition to facilitate consistency checking across abstraction layers.
– *Step 3*: The component scenario is completed by adding the required, system-internal interactions (or signal flows) between the system components.

A simple example of a component scenario is depicted in Fig. 5.

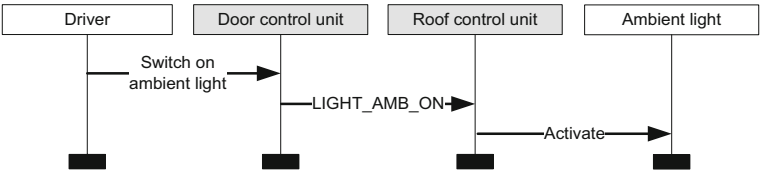

Fig. 5. Simple example of a component scenario

Note that one system scenario may be detailed by several component scenarios which define different possibilities for realising the system scenario through different system-internal interactions.

3.2.2 Definition of Additional Component Scenarios

Typically, additional component scenarios must be defined to deal with specific conditions that are considered at the component level such as temporary or permanent component failures or error diagnosis functionality (see Fig. 6 for an example). These scenarios may or may not include interactions with external actors.

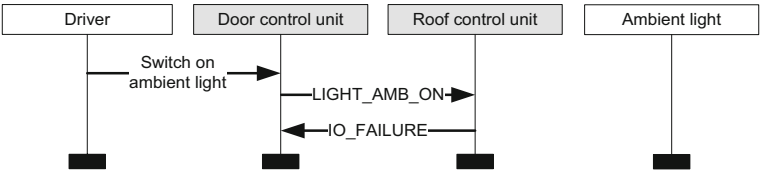

Fig. 6. Example of an additional, level-specific scenario

3.2.3 Component-Level Use Cases

Similar to the grouping of system scenarios into system-level use cases, component scenarios are also grouped into component-level use cases. A necessary condition for the completeness of the component-level use cases is that each system-level use case is detailed by at least one component-level use case. However, our approach does not require that the HMSC structure of the component-level use case is identical with the HMSC structure of the corresponding system-level use case (see Fig. 7). Rather, we advise that the component-level use cases are structured in a way that is convenient for the component level. For instance, it may be convenient to decompose a component-level use case into several more fine-grained use cases and relate the composed use case to the system-level use case. Furthermore, the comprehensibility and changeability of component-level use cases can often be improved by extracting and merging redundant scenario fragments.

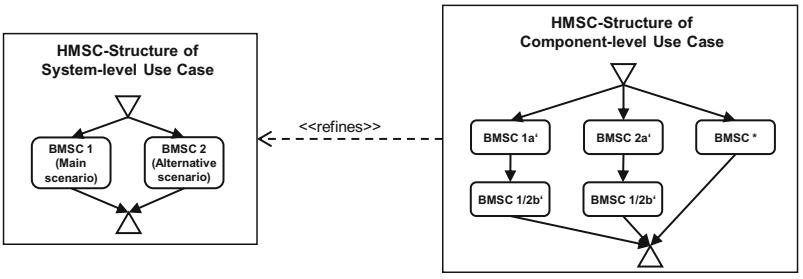

Fig. 7. HMSC structures of a system-level and a component-level use case

3.3 Comparison of System and Component Scenarios

The goal of comparing system and component scenarios is to identify differences in the specified sequences of interactions at the system level and the component level. The possible causes of such differences include:

- Omission of a specific system-actor interaction at some abstraction level
- Definition of an additional system-actor interaction at some abstraction level
- Changed order of the interactions at the system or the component level
- Definition of an additional scenario at some abstraction level that causes a difference in the specified, external behaviour

The comparison is performed for each pair of scenarios related by a “refines” link (see Fig. 7). The comparison can be applied to individual scenarios (i.e. one system scenario is compared to one component scenario), to use cases (i.e. a system-level use case is compared to a component-level use case), or to sets of interrelated use cases (see Section 2.2). The decision whether a detected difference is considered as an inconsistency or not is influenced by project-specific consistency rules. These rules can, for instance, allow or forbid the definition of additional system-actor interactions at the component level (see Section 4.4).

In case an inconsistency is identified, human judgement is required to determine necessary corrections. For instance, an additional sequence of interactions at the component level may be caused either by a missing system scenario or by an unwanted, additional component scenario. Hence, the stakeholders must decide whether the system scenarios, the component scenarios, or even both must be corrected to resolve a detected inconsistency.

4 Computation of Differences between Scenarios across Two Abstraction Levels

In this section, we outline our algorithm for computing the differences between scenarios defined at two different abstraction levels. Fig. 8 shows an overview of the major steps of the algorithm. The input of the algorithm comprises two message sequence chart (MSC) specifications, i.e. a system-level and a component-level specification, and, in addition, the coarse-grained system architecture. The MSC

specifications represent a pair of scenarios (or use cases) related to each other by a "refines" link. Each MSC specification may comprise several HMSCs and BMSCs. The algorithm is applied to each such pair of MSC specifications individually. The architecture that is provided as input defines the decomposition of the system into a set of components. It hence interrelates the instances defined in the two MSC specifications.

In Step 3.1 (see Fig. 8), the scenarios are normalised in order to ensure that identical message labels have identical meanings. Step 3.2 transforms the normalised MSCs into interface automata. It results in a system-level automaton P_H (Step 3.2a) and a component-level automaton P_L (Step 3.2b). Step 3.3 computes the differences between P_H and P_L. It results in two automata $P_{H\text{-}L}$ and $P_{L\text{-}H}$.

The individual steps are explained in more detail in Subsections 4.1 to 4.3. In addition, we outline the analysis of the computed difference automata in Section 4.4.

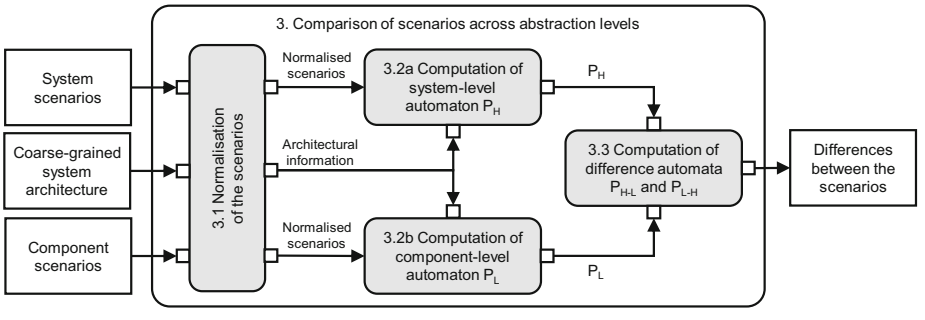

Fig. 8. Main steps of the comparison of system and component scenarios

4.1 Normalisation of the Message Sequence Charts

To compute the differences between the system and component scenarios, the message sequence charts documenting the scenarios must match some input criteria. We assume that each message sent or received by an instance has a label that uniquely identifies its message type. Furthermore, we assume that messages of a specific type sent to the environment or received from the environment are labelled identically at the system level and the component level.

The information which components decompose the system is taken from the architecture model. For the system scenarios, we assume that the instance representing the system is named consistently with the representation of the system in the architecture model. For the component scenarios, we assume that the instances representing system components are named consistently with the representations of the components in the architecture model.

To ensure that the above assumptions hold, a normalisation step is performed in which the instance names and message labels are checked and adapted, if necessary.

4.2 Transformation of the Scenarios into Interface Automata

To facilitate computing the differences between system and component scenarios, we employ a transformation of MSCs into automata. Interface automata [3] offer several

advantages that make them particularly suitable for our approach. For instance, the set of actions of an interface automaton is partitioned into a set of input actions, a set of output actions, and a set of internal actions. This corresponds to the event types defined for MSCs (receive, send, internal). Furthermore, interface automata do not enforce that each input action is enabled in every state. This is also true for the MSCs in our approach since we assume that, after reaching a specific location of an instance line, only the specified events are allowed to occur.

We briefly outline the transformation of the scenarios into interface automata:

1. *Construction of instance automata*: In this step, an interface automaton is computed for the system (system level) as well as for each component (component level). For this purpose, first each BMSC is transformed into a set of partial automata based on the algorithm described e.g. in [10]. Subsequently, the partial automata are concatenated by inserting τ-transitions (i.e. invisible transitions) as defined by the HMSC edges. Environment instances are disregarded in this step.
2. *Elimination of τ-transitions and indeterminism*: In this step, non-deterministic transitions and the τ-transitions inserted during concatenation are eliminated in the interface automata that were constructed in the first step. For performing this step, standard algorithms for automata such as those described in [11] can be used. The results of this step are the system-level automaton P_H and a set of component-level automata.
3. *Composition of the instance automata*: In this step, the automata derived from the component scenarios are composed to a single component-level automaton P_L by means of the composition operator defined for interface automata (see [3]).

4.3 Computation of the Differences between the Scenarios

The comparison of the automata shall reveal differences between the traces of the automata with regard to the externally observable system behaviour (similar to weak trace equivalence; see [12]). For this purpose, the traces consisting only of input and output actions of the interface automata P_H and P_L need to be compared with each other. The set of traces of the automaton P_H is called the language of P_H and denoted as L_H. The set of traces (of input and output actions) of P_L is called the language of P_L and denoted as L_L. To compare the two languages, two differences must be computed:

- $L_{H\text{-}L} = L_H \setminus L_L = L_H \cap \neg L_L$ and
- $L_{L\text{-}H} = L_L \setminus L_H = L_L \cap \neg L_H$

Hence, for computing the desired differences, the intersection and the complement operator for automata must be applied [11]. Since the complement operator requires a deterministic finite automaton as input, P_H and P_L must be transformed into deterministic automata. Furthermore, the internal actions defined in P_L must be substituted by τ-transitions. Due to space limitations, we omit the details here.

4.4 Analysis of the Computed Differences

The requirements engineers can interpret the resulting automata in the following way:

- $L_{H\text{-}L} = \emptyset$ and $L_{L\text{-}H} = \emptyset$: In this case the externally observable traces of both automata are identical, i.e. the scenarios at the system level and the component are consistent to each other.
- $L_{L\text{-}H} \neq \emptyset$: In this case, the component scenarios contain traces that are not defined at the system layer. The requirements engineers have to analyse these in order to determine which of them are valid or desired and which ones are not. Traces contained in $L_{L\text{-}H}$ that are considered valid may indicate missing system scenarios. However, the project-specific consistency rules may also allow such traces under certain conditions.
- $L_{H\text{-}L} \neq \emptyset$: In this case, the system scenarios contain traces that are not defined at the component level. The requirements engineers have to analyse these traces in order to determine which of these traces are valid. Traces contained in $L_{H\text{-}L}$ that are considered valid indicate missing component scenarios.

For supporting the interpretation and analysis of the computed differences, we generate graphical representations of the difference automata using the environment described in [13]. The analysis results are used to drive the further development and consolidation of the system and component scenarios.

5 Evaluation of the Approach

We have performed a preliminary evaluation of our approach by applying it to a (simplified) adaptive cruise control (ACC) system based on [14]. In a first step, use cases were identified for the ACC system, both, at the system level and the component level. All in all, eleven use cases were identified. Fig. 9 shows an excerpt of the use case diagram.

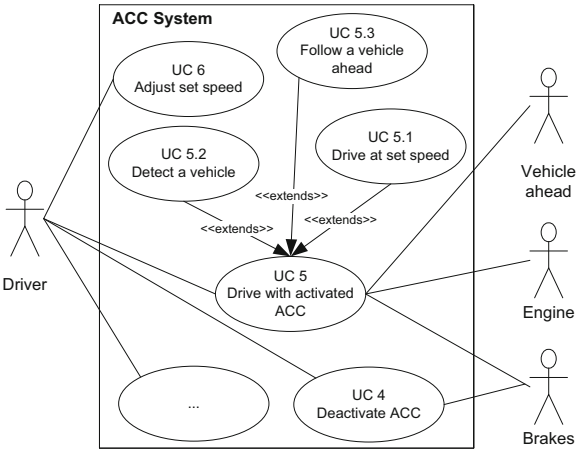

Fig. 9. Excerpt of the use case diagram defined for the ACC system

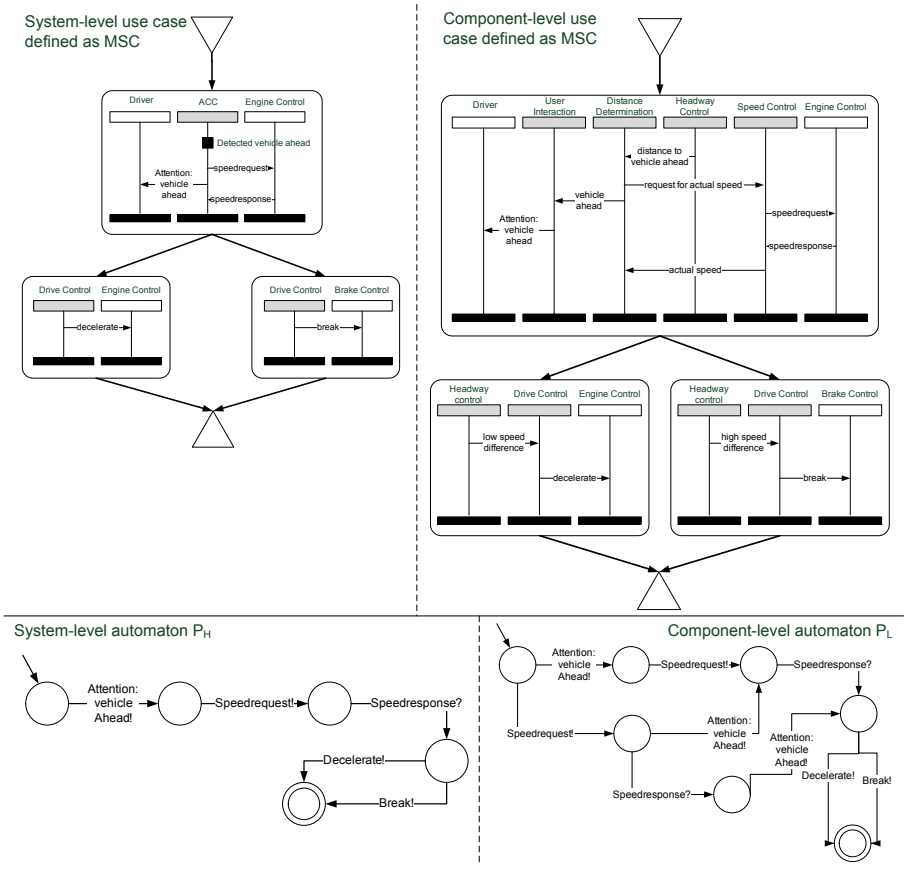

Fig. 10. System- and component-level specifications of an exemplary use case

For the identified use cases, scenarios were specified by means of message sequence charts. The specification activity resulted in eleven HMSCs and nineteen BMSCs at the system level and eleven HMSCs and twenty-five BMSCs at the component level. The consistency checking was performed using a prototypical implementation of the algorithm described in Section 4. Based on the computed differences, the system scenarios and component scenarios were consolidated to remove inconsistencies. Fig. 10 depicts an exemplary system-level use case defined for the ACC system, the corresponding component-level use case as well as the automata P_H and P_L computed for these use cases. The difference $L_{H\text{-}L}$ for the use case depicted in Fig. 10 is empty which means that the component-level use case realises all scenarios defined by the system-level use case. The automaton representing the difference $L_{L\text{-}H}$ is shown in Fig. 11. The sequences of actions which lead from the start state to the final state of this automaton are included in the component-level use case but are not defined by the system-level use case and thus may indicate inconsistencies. The evaluation demonstrated the importance of an automated, tool-supported consistency

check between system and component scenarios. The simplified ACC system used in the evaluation was already too complex to detect all inconsistencies between system and component scenarios manually. The prototypical tool revealed a large number of inconsistencies in the initial use cases and thus contributed significantly to improving the consistency of the use cases across the two abstraction levels. However, since the evaluation has been performed in an academic environment, further investigations concerning the applicability and usefulness of the approach in an industrial environment are needed.

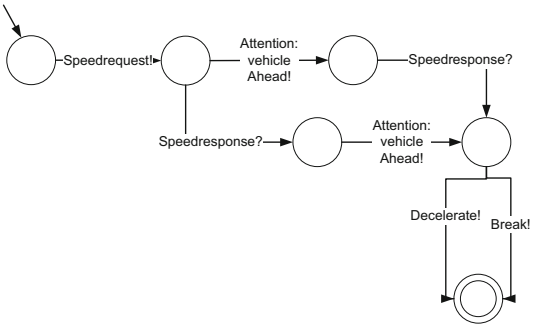

Fig. 11. Difference automaton representing $L_{L\text{-}H}$

Scalability or performance problems did not occur during the consistency check of the ACC scenarios. For instance, the computation of the difference automaton depicted in Fig. 11 took approximately 100 milliseconds. Still, for very complex use cases (such as a composition of all use cases defined for a system into a single use case), a more efficient implementation might be needed.

6 Related Work

Although scenario-based approaches exist that support, in principle, the development of scenarios at different abstraction levels such as FRED [15] and Play-in/Play-out [16], these approaches do not offer the required rigor for safety-oriented development. The checking of the consistency of the scenarios across abstraction levels is not supported by these approaches. Play-in/Play-out merely supports checking whether a set of "universal" scenarios realise a defined "existential" scenario which is not sufficient for proving cross-level consistency.

Approaches that support the formal verification of scenarios suffer from other deficiencies limiting their applicability within our approach. Existing techniques that, in principle, support the verification of MSCs across different abstraction levels provide insufficient support for HMSCs or do not support HMSCs at all. In [9], severe restrictions are imposed by requiring identical HMSC structures at the system level and the component level. The goal of temporal-logic model checking is typically to reveal a single counter example. In contrast, our approach computes extensive differences

between the scenarios defined at two abstraction levels. Furthermore, to apply temporal-logic model checking, use cases must be encoded using temporal logic which limits the applicability of such an approach in practice.

Furthermore, our approach can be regarded as a further step towards a methodical support for the transition between requirements and design as it facilitates the formally consistent specification of black-box scenarios and design-level scenarios. The approach thus complements less formal approaches such as [18] which aim at supporting the communication between requirements engineers and architects.

7 Conclusion

The approach presented in this paper closes a gap in the existing, scenario-based requirements engineering methods. It supports the development of scenarios at different abstraction levels and therein facilitates cross-level consistency checking. Cross-level consistency is important, for instance, for constructing safety proofs and to avoid requirements defects which lead to significant rework during system integration.

The approach employs the message sequence charts (MSC) language as a formal, visual specification language for scenarios and use cases. Individual scenarios are specified as basic message sequence charts (BMSCs). High-level message sequence charts (HMSCs) interrelate several BMSCs and allow for iterations and alternatives. The consistency check offered by our approach aims at detecting differences in the traces of externally observable events specified at the system level and those specified at the component level. The approach thus reveals, for instance, whether the traces of a component-level MSC are complete and necessary with respect to a system-level MSC. The approach is not based on simple, syntactic correspondences but rather employs a transformation of MSCs into interface automata. This makes the approach robust against changes at the syntactic level such as restructuring an MSC.

We have demonstrated the feasibility of our approach by applying it to the specification and consistency checking of requirements for an adaptive cruise control system. Our approach has proven useful for supporting the specification of the scenarios at the system and component level. We hence consider objectives O1 and O2 defined in Section 1.3 to be met. The consistency of the scenarios was checked using a prototypical tool. Thereby, a large amount of inconsistencies could be resolved which were difficult or even impossible to detect manually. We hence consider objective O3 (see Section 1.3) to be met. The approach can be applied in settings where consistency across different abstraction levels must be enforced and the use of formal specification and verification methods is accepted. A detailed evaluation of the applicability of our approach in industrial settings is ongoing work.

Acknowledgements. This paper was partly funded by the German Federal Ministry of Education and Research (BMBF) through the project “Software Platform Embedded Systems (SPES 2020)”, grant no. 01 IS 08045. We thank Nelufar Ulfat-Bunyadi for the rigorous proof-reading of the paper.

References

[1] Gorschek, T., Wohlin, C.: Requirements Abstraction Model. Requirements Engineering Journal (REJ) 11, 79–101 (2006)

[2] International Telecommunication Union. Recommendation Z.120 - Message Sequence Charts, MSC (2004)

[3] De Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proc. of the ACM SIGSOFT Symp. on the Foundations of Software Engineering, pp. 109–120 (2001)

[4] RTCA: DO-178B – Software Considerations in Airborne Systems and Equipment Certification (1992)

[5] Potts, C.: Using Schematic Scenarios to Understand User Needs. In: Proc. of the ACM Symposium on Designing Interactive Systems – Processes, Practices, Methods and Techniques (DIS 1995), pp. 247–266. ACM, New York (1995)

[6] Pohl, K.: Requirements Engineering – Foundations, Principles, Techniques. Springer, Heidelberg (to appear 2010)

[7] Peled, D.: Specification and Verification using Message Sequence Charts. Electr. Notes Theor. Comp. Sci. 65(7), 51–64 (2002)

[8] Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In: Proc. of the Intl. Conference on Software Engineering, pp. 314–323 (2000)

[9] Khendek, F., Bourduas, S., Vincent, D.: Stepwise Design with Message Sequence Charts. In: Proc. of the IFIP TC6/WG6.1, 21st Intl. Conference on Formal Techniques for Networked and Distributed Systems, pp. 19–34. Kluwer, Dordrecht (2001)

[10] Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: Proc. of the IFIP WG10.3/WG10.5, Intl. Workshop on Distributed and Parallel Embedded Systems, pp. 61–71. Kluwer, Dordrecht (1999)

[11] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Reading (2006)

[12] Milner, R.: Communication and Mobile Systems – The Pi Calculus. Cambridge University Press, Cambridge (1999)

[13] Gansner, E., North, S.: An Open Graph Visualization System and its Applications to Software Eengineering. Software - Practice and Experience (1999)

[14] Robert Bosch GmbH: ACC Adaptive Cruise Control. The Bosch Yellow Jackets (2003), http://www.christiani-tvet.com/

[15] Regnell, B., Davidson, A.: From Requirements to Design with Use Cases. In: Proc. 3rd Intl. Workshop on Requirements Engineering – Foundation for Software Quality, Barcelona (1997)

[16] Harel, D., Marelly, R.: Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)

[17] Ohlhoff, C.: Consistent Refinement of Sequence Diagrams in the UML 2.0. Christian Albrechts Universität, Kiel (2006)

[18] Fricker, S., Gorscheck, T., Byman, C., Schmidle, A.: Handshaking with Impementation Proposals: Negotiating Requirements Understanding. IEEE Software 27(2), 72–80 (2010)