JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 0000; **00**:1–**??**
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Integrated and Iterative Systems Engineering and Software Requirements Engineering for Technical Systems

Jörg Holtmann*, Ruslan Bernijazov,
Matthias Meyer, David Schmelter, and Christian Tschirner

Fraunhofer IEM
Zukunftsmeile 1, 33102 Paderborn, Germany

SUMMARY

The development of software-intensive technical systems involves several engineering disciplines like mechanical, electrical, control, and particularly software engineering. Model-based Systems Engineering (MBSE) coordinates these disciplines throughout the development by means of discipline-spanning processes and a system model. Such a system model provides a common understanding of the system under development and serves as a starting point for the discipline-specific development. An integral part of MBSE is the requirements engineering on the system level. However, these requirements need to be refined for the discipline-specific development to start, e.g., into specific requirements for the embedded software. Since existing MBSE approaches lack support for this refinement step, we conceived in previous work a systematic transition from MBSE to model-based software requirements engineering. We automated the steps of the transition where possible, in order to avoid error-prone and time-consuming manual tasks. In this paper, we extend this approach with support for subsequent process iterations and provide an algorithm for the automated steps. We illustrate the approach and perform a case study with an example of an automotive embedded system. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Systems Engineering; Requirements Engineering; Development Process; Model Transformations; Technical Systems

1. INTRODUCTION

The development process of software-intensive technical systems (e.g., within the automotive industry) involves several engineering disciplines. Each of these disciplines applies its dedicated design methods and languages. This implicates the need for a holistic and interdisciplinary consideration of the overall system under development (SUD) to obtain a common understanding for all roles involved in its development. "Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems" [1] and aims at achieving this common understanding. According to [2], the current transition from document-based Systems Engineering to Model-Based Systems Engineering (MBSE) aspires to overcome the disadvantages of the traditional "throw-it-over-the-wall" development process [3]. The abstraction of the real system to a discipline-spanning system model not only supports a holistic understanding of the SUD for a "First Time Quality Development" but also enables the traceability of all relevant information from customer

*Correspondence to: joerg.holtmann@iem.fraunhofer.de

Contract/grant sponsor: German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL) (managed by the Project Management Agency Karlsruhe, PTKA) and within the ITEA 2 AMALTHEA4public project (managed by the Project Management Agency of the German Aerospace Center, PT-DLR); contract/grant number: 02PQ1040 and 01IS14029I, respectively

requirements to the final system as well as design space exploration in early design phases based on (semi-)automatic analyses.

A high quality requirements engineering is crucial alongside MBSE since the subsequent development relies on it, and defects in the requirements specification are hard and thus costly to fix later (e.g., [4]). These requirements need to be refined, e.g., into specific requirements for the embedded software, to start the discipline-specific development. For this reason, development process standards for technical systems like Automotive SPICE [5] explicitly distinguish between requirements engineering at system and software level.

On the one hand, the embedded software of a technical system comprises *control behavior*, e.g., electronic stability control, that is specified in the discipline of control engineering by means of continuous models (based on differential equations). On the other hand, a growing part of the software is dedicated to the message-based coordination of systems (e.g., car-2-car communication). This *coordination behavior* is specified by means of discrete, state-based, and event-triggered models [6]. We regard the discipline of software engineering as the one that designs this coordination behavior. The requirements engineering for software is regarded as a branch of systems engineering [7]. More specifically, we regard software requirements engineering (SWRE) as a sub-discipline of software engineering, i.e., as requirements engineering for the coordination behavior part of the overall system.

We focus on the requirements engineering core activities *documentation* and *validation* [8]. Similar to the advantages of applying models in MBSE, applying models in requirements engineering supports the requirements validation by fostering automatic analysis techniques. In previous work, we developed a formal SWRE approach based on *Modal Sequence Diagrams* (MSDs) [9]. This scenario-based approach allows to validate the requirements by means of simulation, i.e., the play-out algorithm by Harel and Marelly [10]. Furthermore, the approach enables to formally verify the requirements for consistency and realizability by synthesizing global controllers from the scenarios [11]. These automatic analysis techniques enable the early detection of unintended behavior and inconsistencies between scenarios on requirements level. Our MSD approach considers assumptions on the environment [12] as well as real-time requirements [13] and is applicable to hierarchical component architectures [14], which makes it well suited in the context of technical systems.

Although MBSE endeavors to coordinate the overall development by means of an interdisciplinary system model, the transition from MBSE to SWRE is not trivial. The system model contains much information that is only partly relevant to all involved disciplines. Thus, the software requirements engineer must carefully identify the information in the system model relevant to SWRE and transfer this information into a MSD specification. Despite the system model is amenable for automatisms to extract this information, this is a manual and thus time-consuming and error-prone task.

We apply the specification technique CONSENS (CONceptual design Specification technique for the Engineering of complex Systems) [15, Sect. 4.1] as a concrete MBSE approach. CONSENS is conceived for the interdisciplinary design of complex technical systems, comprises a semi-formal modeling language, and defines a method for the creation of the system model in the concept phase.

Based on CONSENS, two approaches exist for the transition from MBSE to software engineering. On the one hand, Heinzemann et al. [16] present a systematic development process for the software comprising control and coordination behavior. This process utilizes automatisms to derive initial discipline-specific models. However, the approach aims at deriving and refining design models but lacks a transition to SWRE. On the other hand, Anacker et al. [17] present a systematic process for the transition from MBSE to the software design via MSD specifications. However, this approach relies on reusable solution patterns (i.e., there must be prior projects in which these patterns have been successfully developed) and hence does not support greenfield development.

In order to facilitate the transition from MBSE with CONSENS to SWRE with MSDs, we introduced in previous work [18] a systematic process that complements the transition approaches mentioned above. We identified aspects of the system model that are relevant to SWRE and described how SWRE integrates into MBSE. In order to avoid error-prone and time-consuming manual tasks, we automated steps of the transition where possible. In this paper, we extend this approach with improved tool support also for subsequent process iterations and provide an algorithm for the automated steps

DOI: 10.1002/smr

as well as a case study. We illustrate the approach and perform the case study with an example of an automotive embedded system.

In the next section, we present the overview of the transition process. Sect. 3 presents an exemplary initial iteration of this process, whereas Sect. 4 illustrates a subsequent process iteration. Sect. 5 presents the algorithm for the automated steps and points out implementation details. Afterwards, we perform a case study in the following section. Sect. 7 investigates related work. Finally, we summarize this paper and provide an outlook on the future work in Sect. 8.

2. OVERVIEW: INTEGRATION OF MBSE AND SWRE

Systems Engineering has a long tradition. However, due to the various industries the systems engineers work for and their distinct tasks, its job profile is not clearly defined up to now [19]. The lowest common denominator about the job profile is: The systems engineer is the "conductor" of a project, orchestrating all technical tasks [20]. Based on this, Sheard [19] identified twelve possible roles of systems engineers. From these, the following three roles are important for SWRE:

Requirements Owner (RO) The RO starts with the definition of technical requirements on system level (system requirements) based on given customer requirements. An example is the speed of a car: the customer demands a distinct maximum speed leading to a system requirement on the number of valves for the motor.

System Designer (SD) Based on the system requirements, the SD creates the high-level, discipline-spanning system architecture. This usually means the definition of system functions, the selection of adequate top-level components and their allocation to engineering disciplines.

System Analyst (SA) The SA ensures that the SUD meets the system requirements. Typical analyses are "system weight or throughput" for hardware and "memory usage or response times" for software. Usually, these analyses can only be executed at a very late lifecycle stage. The advent of MBSE enables early, abstract simulations on system level. This helps reducing uncertainty and risk in the early stages of the project, thus avoiding time- and cost-intensive iterations in the later stages. Since it is impossible to model all relevant aspects of the system for this role, his task is rather to provide an optimal starting point for the discipline-specific issues.

Besides these roles, the *Customer Interface (CI)* is essential for a successful project. The CI is the "face to the customer" and responsible for elicitating the customer requirements.

Within the software engineering discipline, we focus on the role *Software Requirements Engineer (SW Requirements Engineer)* responsible to document and validate the requirements on the coordination behavior and negotiate them with the SA.

Fig. 1 depicts our idealized overall process for the systematic integration of SWRE with MSDs into MBSE with CONSENS. The process is specified by means of the Business Process Model and Notation (BPMN) [21]. We allocate the particular tasks to the roles mentioned before to clarify the distribution of responsibilities. Whereas the coarse-grained roles Systems Engineer and Software Engineer are specified by means of BPMN *pools*, the fine-grained, specialized roles introduced above are specified by means of BPMN *lanes*.

The main contribution of this paper is emphasized in Fig. 1 with gray tasks and artifacts. We visualize manual steps by means of BPMN *manual tasks* (hand in the upper left corner of the task). Steps that we could automate are visualized through BPMN *service tasks* (cogwheel in the upper left corner of the task). The step that is tool-supported is specified by means of a BPMN *user task* (person in the upper left corner of the task). Work results are specified as BPMN *data objects* (document icons), and persistent models that are subject to update and retrieval operations are specified as BPMN *data stores* (database icon). Multiple occurrences of the same data store represent a BPMN *data store reference*. We distinguish between two event types that trigger the MBSE and SWRE tasks, respectively. BPMN *none start events* (circle) mark an initial process iteration. BPMN *multiple start events* (circle with pentagon) mark subsequent iterations, e.g., initiated by change requests. We propose to systematically follow every task of our process also in subsequent iterations to avoid

DOI: 10.1002/smr

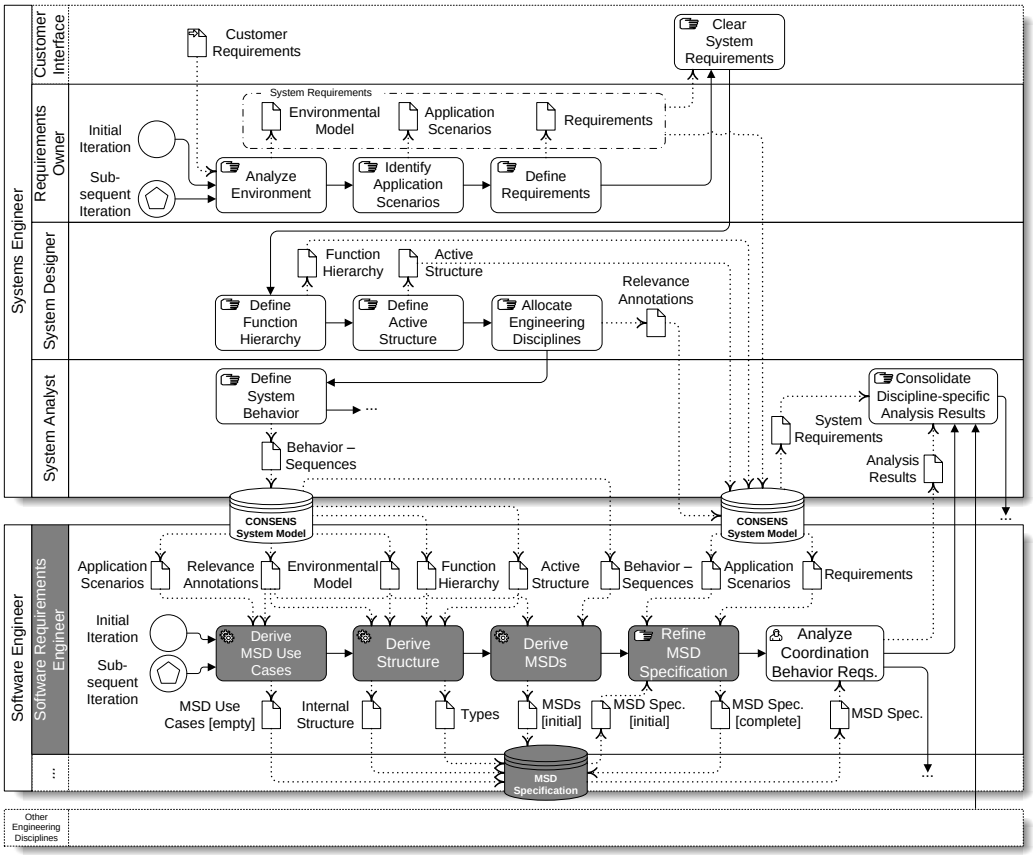

Figure 1. Idealized overall process for the integration of MBSE with CONSENS and SWRE with MSDs

potential design flaws—even if the change to perform allegedly does not influence every task. If it is clear that a subsequent iteration does not influence a particular task, it can be skipped of course.

Note that despite the roles in Fig. 1 might indicate a strict separation of responsibilities, we encourage the involvement of all disciplines in the specification of the system requirements and design since issues resulting from a strict separation of responsibilities can cause major problems [22]. Furthermore, we assume that the Systems Engineer has a "T-shaped" competency profile [23] with deep knowledge in one discipline as well as basic knowledge in the remaining disciplines. This enables the Systems Engineer to understand and respect the needs of all discipline experts.

3. INITIAL PROCESS ITERATION

In the following, we exemplarily perform and explain each of the process steps in Fig. 1 from the last section. For illustration purposes, we use a simplified automotive car access system as a running example for the SUD. This system comprises a door lock electronic control unit (ECU) and a part of the functionality of a Body Control Module (BCM). A BCM centrally controls distributed car body functions like central locking, exterior lights, anti-theft warning system, etc. We focus on the functionality of central locking and speed locking in this section and on a safety-critical crash unlocking function in the next section.

3.1. *MBSE with CONSENS*

The CONSENS modeling language is divided into eight partial models describing different aspects of a technical system: *Environmental Model*, *Application Scenarios*, *Requirements*, *Functions*, *Active Structure*, *Shape*, *System of Objectives*, and *Behavior*. Behavior can be subdivided into *Behavior – States*, *Behavior – Activities*, and *Behavior – Sequences*. We screened the CONSENS language and

DOI: 10.1002/smr

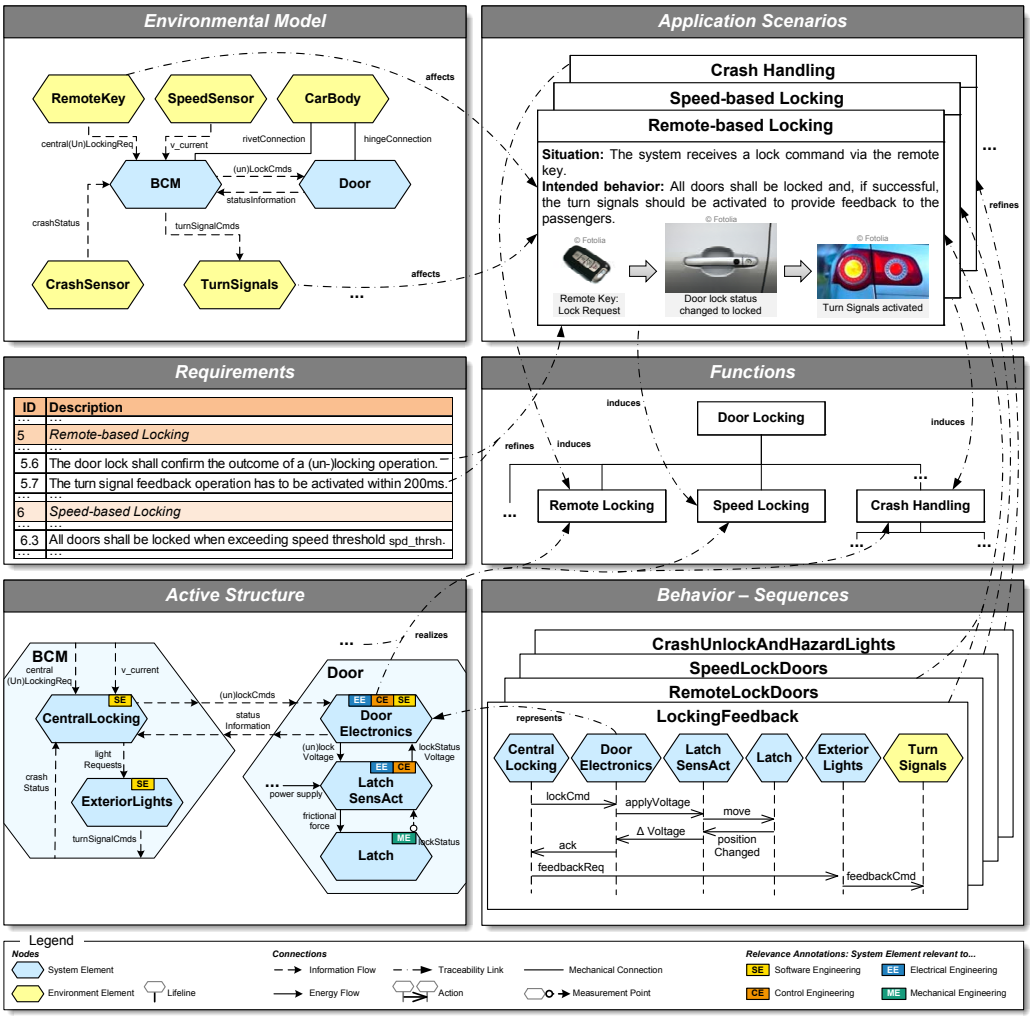

Figure 2. CONSENS system model of the BCM with partial models relevant to SWRE

identified six partial models that contain information relevant to SWRE. The partial model *Shape* describing the geometrical appearance of the SUD and the other behavior models except *Behavior – Sequences* are of no interest for the addressed coordination behavior. Furthermore, in our experience *System of Objectives* is only applied for self-adaptive systems.

All aspects specified with the CONSENS partial models are strongly interconnected. These interconnections are explicitly specified as cross-references between elements of partial models, e.g., bidirectional trace links between requirements and functions. Hereby, traceability can be established and maintained to enable impact analyses.

The process step order for the specification of the particular CONSENS partial models in the systems engineering part of Fig. 1 is based on the method presented in [15, Sect. 3.2]. In the following, we exemplarily perform each of these process steps for the development of our car access system running example. We refer to our previous work [18] for more details about the presented partial models. Fig. 2 depicts excerpts of all SWRE-relevant partial models specified in CONSENS.

3.1.1. *Analyze Environment* The *Environmental Model* (top left of Fig. 2) is a structural partial model and considers the SUDs BCM and Door as black boxes embedded into their environment (e.g., comprised of RemoteKey, SpeedSensor, and others). Thereby, relevant spheres of influence are identified. The CI and RO analyze the environment and jointly identify all relevant elements that influence the SUD. The CI represents the market issues (e.g., external stakeholders like customers), whereas the RO represents all technical issues and is responsible for the correct usage of the method.

DOI: 10.1002/smr

System and environment elements are physical elements like parts, assemblies, or modules but can also refer to non-physical elements like software components. Different kinds of flows (i.e., material flows, energy flows, and information flows like central(Un)LockingReq) and a mechanical connection (e.g., rivetConnection) represent the relationships between the elements. The flows in the system model connect ports of the system or environment elements. These ports are specified by means of port specifications (ports and port specifications are hidden in Fig. 2 to increase the readability). A port specification defines which particular material, energy, or information, respectively, can be transferred over the port. For example, the BCM can receive central (un-)locking requests via the information flow central(Un)LockingReq from the RemoteKey and the current velocity via v_current from SpeedSensor. Mechanical connections directly connect system or environment elements.

3.1.2. *Identify Application Scenarios* Application scenarios (top right in Fig. 2) like Remote-based Locking are initial assumptions of the system's behavior. They describe the most common operation modes of the system and the corresponding behavior in a rough manner. Every application scenario describes a specific technical situation and the required behavior of the system. For instance, Remote-based Locking describes the trigger situation as well as the intended behavior for the overall remote locking functionality including sketches. It specifies that after a successful door locking operation the turn signals have to be activated to provide feedback to the passengers. Crash Handling describes the behavior in case of a crash and is explained in more detail in Sect. 4. The environment elements have affects trace links to application scenarios in which they are involved. For example, the environment element RemoteKey is involved in the application scenario Remote-based Locking whereas SpeedSensor is not involved.

Again, the RO is responsible for this task, but is supported by other stakeholders, e.g., from manufacturing and especially validation and verification. He has to consolidate all application scenarios and evaluate their significance.

3.1.3. *Define Requirements* Based on the partial models *Environmental Model* and *Application Scenarios*, the RO defines, specifies, and manages the requirements in the partial model *Requirements* (center left in Fig. 2). All requirements with ID 5.x have refines trace links to the application scenario Remote-based Locking, and all requirements with ID 6.x have links to Speed-based Locking (not depicted in Fig. 2). These trace links enable that all requirements refining an application scenario can be identified from it and vice versa.

3.1.4. *Clear System Requirements* We regard the *Environmental Model*, *Application Scenarios*, and *Requirements* altogether as System Requirements (cf. BPMN group in Fig. 1), which have to be cleared by the CI with the customer before the actual system design starts. They propose a technically oriented solution of the SUD functionality as described by the Customer Requirements (cf. BPMN data input in Fig. 1).

3.1.5. *Define Function Hierarchy* A function is the general and required coherence between input and output parameters, aiming at fulfilling a task. In our example, the function hierarchy (center right in Fig. 2) decomposes the overall functionality Door Locking into the separate functions Remote Locking, Speed Locking, and Crash Handling.

The traceability between application scenarios and functions is established via induces trace links. For example, the function Remote Locking is induced by the application scenario Remote-based Locking, Speed Locking is induced by Speed-based Locking, and the function Crash Handling is induced by the identically named application scenario.

3.1.6. *Define Active Structure* The *Active Structure* (bottom left in Fig. 2) considers the SUDs BCM and Door as white box and modularizes them into subsystems. The BCM comprises the *system elements* CentralLocking and ExteriorLights, which are connected to the environment elements from the *Environmental Model* via delegation connectors. For instance, CentralLocking receives signals via the connector central(Un)LockingReq from RemoteKey as well as via v_current from SpeedSensor. As in the *Environmental Model* (cf. Sect. 3.1.1), the relationships are specified by means of different flow kinds. The Door contains a DoorElectronics subsystem that activates the combined sensor/actuator LatchSensAct by means of the energy flow (un)lockVoltage. LatchSensAct

DOI: 10.1002/smr

can cause frictional force on the actual Latch and senses its lockStatus. We call the combination of *Environmental Model* and *Active Structure* together system architecture.

The system elements have realizes trace links to the functions of the partial model *Functions* if they contribute to the realization of a function. In our example, almost all depicted system elements of the active structure realize the three sub functions (only indicated in Fig. 2). The only exception is ExteriorLights, which has no trace link to Speed Locking.

3.1.7. *Allocate Engineering Disciplines* While conceiving the *Active Structure*, the SD determines during the discussion with the discipline-specific experts (e.g., the Software Engineer in Fig. 1) which system elements are to be concretized in which disciplines. As in [16], we use so-called *relevance annotations* to specify the relevance of a system element to a certain discipline in the active structure. We exploit this information in the transition to SWRE to enable the automatic derivation of a MSD specification (cf. Sect. 3.2). We use the relevance annotation "SE" to specify the relevance of a system element to the discipline of software engineering. We call such system elements *SE-relevant*. Since we regard SWRE as sub-discipline of software engineering (cf. Fig. 1), we also regard all SE-relevant system elements as relevant to SWRE.

For instance, the BCM in the *Active Structure* (cf. Fig. 2) contains only software components, whereas the Door contains system elements that are relevant to different disciplines. The DoorElectronics is an electronic component with deployed software. Experts from the disciplines electrical, control, and software engineering jointly design this system element. This is visualized by the relevance annotations "EE", "CE", and "SE", respectively. The electrical engineer designs the electronic parts, e.g., the energy flows from/to LatchSensAct. The software is jointly designed by a software (coordination behavior) and a control engineer (control behavior).

3.1.8. *Define System Behavior* Mechatronic systems are characterized by different kinds of behavior. This is reflected in CONSENS by a group of different behavior models for which the SA is responsible. In this paper, the *Behavior – Sequences* are important. Similar to other scenario-based notations, they specify partially ordered, discipline-spanning sequences of *actions* executed on system/environment elements, which are represented by *lifelines* similar to UML Sequence Diagrams [24].

The partial model *Behavior – Sequences* (bottom right in Fig. 2) refines the application scenarios and requirements. The partial model of our example contains four scenarios; we focus on the scenario LockingFeedback. This scenario describes the intended behavior of the system elements w.r.t. the activation of the turn signal feedback operation. For example, after a lockCmd has been sent to the DoorElectronics, this system element has to activate the motor of the LatchSensAct by means of the action applyVoltage.

Each Behavior – Sequence has a refines trace link to a corresponding application scenario as depicted in Fig. 2. Furthermore, each lifeline has a represents trace link to a structural element in the *Active Structure* or *Environmental Model*. In Fig. 2, we indicate this correspondence between lifelines and structural elements by name equality and one exemplary trace link.

3.2. *The Transition to SWRE*

The semi-formal CONSENS system model is well suited to facilitate communication between stakeholders like customers, different systems engineering roles, and discipline experts. However, a formal requirements notation is needed to enable automatic techniques for the requirements analysis. As motivated in the introduction, we apply MSDs for this purpose in the discipline of SWRE.

However, it is expensive to develop a complete MSD specification that can be simulated or checked for consistency. This is exaggerated as the system model contains much information only partly relevant to all involved disciplines. In Sect. 3.1, we determined which CONSENS partial models are relevant to SWRE. But as one can see on the system model in Fig. 2, even only parts of the selected partial models are relevant to SWRE. For example, the SW Requirements Engineer is not interested in the electrical details of LatchSensAct or in the operating principles of the mechanical Latch.

Despite being semi-formal, the CONSENS specification provides much information in a model-based and structured way. In order to exploit this information, we present a transition technique to semi-automatically derive MSD specifications from a CONSENS system model in this section. The transition technique is divided into a fully automatic model transformation part that derives an

Prepared using smrauth.cls
DOI: 10.1002/smr

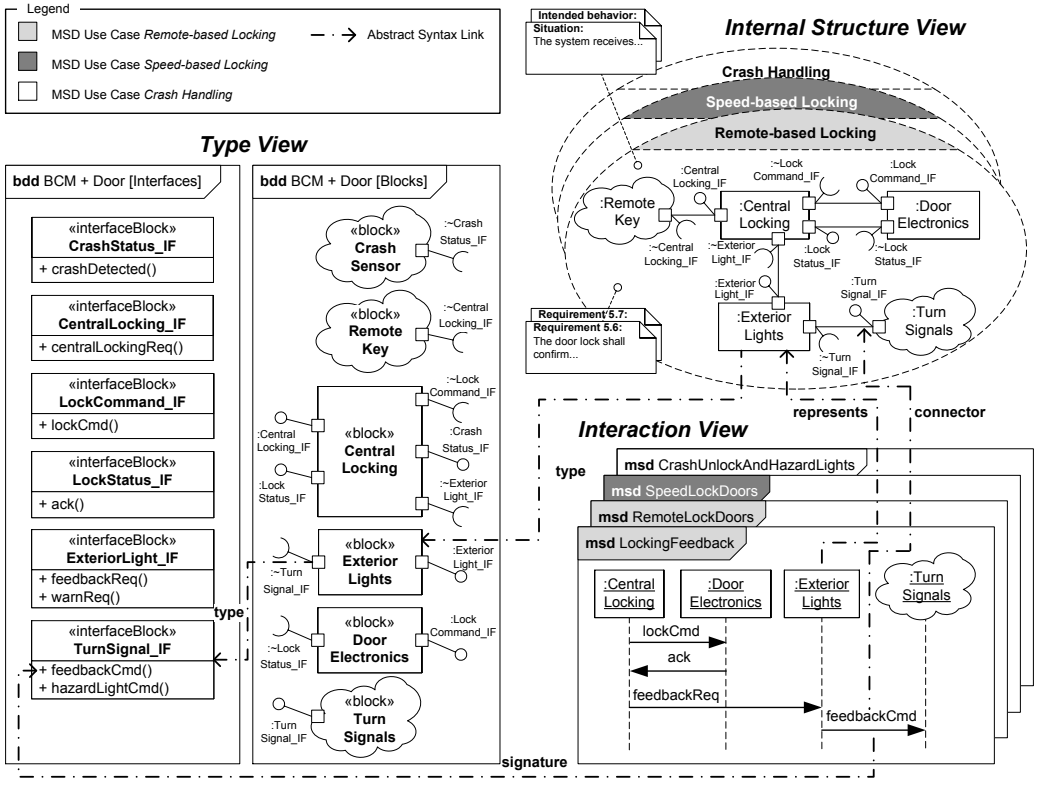

Figure 3. Initially generated MSD specification

initial MSD specification and a manual part, in which the initial MSD specification is systematically refined (cf. Fig. 1). The manual refinement is supported by a set of guidelines that are part of our technique. A key aspect of the automatic part is the identification of information in the system model that is relevant to SWRE. The model transformation exploits the relevance annotations to identify the SE-relevant system elements within the active structure and the trace links between the different partial models. Thereby, information that is not relevant to SWRE is filtered.

The target model of our transition technique is a component-based MSD specification [14]. Such a MSD specification bases on a subset of modeling constructs of both the UML [24] and the Systems Modeling Language (SysML) [25] and extends these constructs by means of the *modal profile*. This profile provides additional MSD-specific modeling constructs for UML Sequence Diagrams, which we present in Sect. 3.2.4. As the *Active Structure* in CONSENS, a component-based MSD specification supports hierarchies. Thus, our transition technique supports these hierarchies, too [26]. However, we also provide a second variant of the technique that flattens these hierarchies. We use this variant in this paper due to space reasons.

Fig. 3 shows the initial MSD specification that is automatically derived from the CONSENS model in Fig. 2. It is structured in three different views, which we introduce in the following.

The Type View provides reusable types (cf. left of Fig. 3). On the one hand, it contains interfaces with operations. These are depicted in the Block Definition Diagram (BDD) named BCM + Door [Interfaces]. On the other hand, this view encompasses structural blocks owning ports that are typed by the interfaces (cf. BDD named BCM + Door [Blocks]).

The Internal Structure View contains *MSD use cases*, which encapsulate the requirements on the coordination behavior of the system or system parts in a specific situation (cf. top right of Fig. 3). We use UML collaborations to specify MSD use cases for software component architectures. These collaborations define the participants of a MSD use case by means of roles typed by the blocks in the Type View. The communication links between these roles are specified by means of ports (defined by the role types) and connectors within the collaboration.

DOI: 10.1002/smr

Each use case contains a set of MSDs, which are specified in the Interaction View (cf. bottom right of Fig. 3). The MSDs formally specify requirements on the coordination behavior of the use case participants. Each lifeline in the MSD represents a role within the collaboration of a specific use case. The messages correspond to the operations of the interfaces in the Type View.

In the following, we explain each particular step of our transition technique as depicted in Fig. 1 based on the running example. The automatic part processes all the SE-relevant information in the CONSENS system model that is formal enough to be subject to a transformation algorithm (cf. the data inputs for the respective tasks in Fig. 1): 1) The *Application Scenarios* are transformed—without interpreting their informal content—to MSD Use Cases (Sect. 3.2.1), 2) the structural models *Active Structure* and *Environmental Model* are transformed to different structural aspects of the MSD specification (Sect. 3.2.2), and 3) the behavioral model *Behavior – Sequences* is transformed to initial MSDs (Sect. 3.2.3). Although we exploit the relevance annotations as in [16], our transformation for this automatic part is more complex due to the facts that source and target models are structurally different and information within different CONSENS partial models has to be collected through trace links. The informal information (i.e., the *Requirements* and the content of the *Application Scenarios*) cannot be processed automatically and hence is subject to the manual refinement step in Sect. 3.2.4. The last two tasks sketch the analysis of a MSD specification (Sect. 3.2.5) and the communication of the yielded analysis results to the SA (Sect. 3.2.6).

3.2.1. *Derive MSD Use Cases* The partial model *Application Scenarios* (cf. Sect. 3.1.2) in the system model provides a system-level structuring of the desired SUD functionality that is similar to the structuring by means of use cases in the MSD specification.

In order to reuse this structuring, we create an empty use case for each SE-relevant application scenario in the system model. We call an application scenario SE-relevant, if at least one SE-relevant system element realizes one of the functions that are induced by this scenario. We use the bidirectional trace links in the system model to determine the SE-relevance of an application scenario and to exactly identify the system elements that are used to realize the functionality specified in this scenario. Furthermore, we use the annotations of these elements: If one is SE-relevant, then also the application scenario is SE-relevant.

For example, the application scenario Remote-based Locking in Fig. 2 is SE-relevant because the system element DoorElectronics is annotated with "SE" and realizes the function RemoteLocking induced by this application scenario. Thus, we add the empty use case Remote-based Locking to the MSD specification, which is depicted by the collaboration Remote-based Locking in Fig. 3. The same principle holds for the indicated MSD use cases Speed-based Locking and Crash Handling.

Furthermore, we transfer informal SE-relevant information in the system model to the MSD specification by annotating it to the corresponding MSD use cases. In context of our example, the informal descriptions Situation and Intended behavior of the application scenario Remote-based Locking are assigned to the identically named MSD use case by means of UML notes. Analogous to these descriptions, the informal requirements 5.6 and 5.7 of the partial model *Requirements* are assigned to this MSD use case by following the appropriate trace links named refines. Again, the same principle holds for the other use cases. By following this approach for all SE-relevant application scenarios, we filter all SE-relevant informal information and transfer it to the MSD specification.

3.2.2. *Derive Structure* The partial models *Environmental Model* (cf. Sect. 3.1.1) and *Active Structure* (cf. Sect. 3.1.6) in CONSENS specify the SUD's environment and system architecture in a discipline-spanning manner, thereby providing a basis for the software architecture of a MSD use case.

In order to derive this software architecture, we apply the same principle as described in Sect. 3.2.1 to determine the SE-relevant system elements realizing an application scenario, i.e., we use the trace links within the system model. We derive a role for the collaboration representing the use case and a corresponding block in the Type View in the MSD specification for each SE-relevant element associated with the corresponding application scenario. For example, the system elements CentralLocking and DoorElectronics realize the application scenario Remote-based Locking and are SE-relevant (cf. Fig. 2). Thus, we derive the roles :CentralLocking and :DoorElectronics within the collaboration Remote-based Locking as well as their typing blocks (cf. Fig. 3).

DOI: 10.1002/smr

Additionally, we consider information flows between SE-relevant elements in the structural partial models in CONSENS as SE-relevant flows. For example, the system element DoorElectronics (cf. Fig. 2) has in- and outgoing information as well as energy flows. As motivated before, the SW Requirements Engineer is mainly interested in the coordination behavior, so we only consider the information flows (un)LockCmds and statusInformation in our transition technique. We use the information flows and their ports to derive ports and connectors for the Internal Structure View, e.g., the ports and connectors of :DoorElectronics in the use case Remote-based Locking (cf. Fig. 3). We also use the port specifications of the ports in the system model to derive ports for the blocks and interfaces in the Type View of the MSD specification, i.e., the ports of the block **DoorElectronics** typed by the interface blocks **LockCommand_IF** and **LockStatus_IF**.

As in CONSENS, a MSD specification distinguishes between environment and system elements. These are visualized by cloud symbols and rectangles, respectively. We consider all CONSENS environment elements that are connected via information flows with SE-relevant system elements as SE-relevant. In our example, this encompasses all environment elements. Furthermore, we use the affects trace links to determine the environment elements involved in an application scenario to derive the environment roles of the corresponding use case. For example, we only derive the environment roles :Remote Key and :TurnSignals for the use case Remote-based Locking and neglect the environment element SpeedSensor in the *Environmental Model*. Besides deriving the environment roles within the collaborations, we generate all corresponding MSD specification elements encompassing interfaces, blocks, ports and connectors analogous to system elements.

3.2.3. *Derive MSDs* Similar to MSDs and scenario-based formalisms in general, the partial model *Behavior – Sequences* specifies partially ordered sequences of actions. Therefore, we reuse the information contained in this partial model to derive initial MSDs for the MSD specification.

We create one MSD in the MSD specification for each Behavior – Sequence in the system model (cf. MSDs in Fig. 3 stemming from Behavior – Sequences in Fig. 2). As it is the case for the structural partial models, the partial model *Behavior – Sequences* contains discipline-spanning information that can be irrelevant to SWRE. For example, the Behavior – Sequence LockingFeedback specifies actions like changing voltages between DoorElectronics and LatchSensAct as well as physical and sensor actions on Latch. Since such actions are not part of the message-based coordination behavior between software components, we only reuse Behavior – Sequence lifelines that represent SE-relevant system/environment elements and the actions between them. For this purpose, we exploit the represents links from a Behavior – Sequence lifeline to system/environment elements (cf. Fig. 2). We consider a Behavior – Sequence lifeline as SE-relevant if it represents a SE-relevant system or environment element (cf. Sect. 3.2.2). Furthermore, we map an initially derived MSD to a MSD use case if the source Behavior – Sequence has a refines trace link to the corresponding application scenario (cf. shades of gray for MSDs and use cases in Fig. 3). The Behavior – Sequence lifelines are mapped to lifelines in the corresponding MSD and the actions are mapped to messages between the corresponding MSD lifelines. SE-relevant lifelines representing environment elements are only mapped to MSD lifelines if the environment element is involved in the application scenario. As a result, we do not derive lifelines and incident messages for LatchSensAct and Latch into the MSD LockingFeedback (cf. Fig. 3), for example.

SE-relevant system elements can cause actions irrelevant to SWRE, e.g., the interactions between DoorElectronics and LatchSensAct. As described in Sect. 3.2.2, we filter SE-relevant connectors based on CONSENS information flows while deriving the Internal Structure View. In the same way, we filter actions that do not correspond to an information flow (like the action applyVoltage via the energy flow (un)lockVoltage in the *Active Structure* and action ΔVoltage via lockStatusVoltage). The SE-relevant actions get an abstract syntax link connector to the corresponding connector in the Internal Structure View (cf. message `feedbackCmd`).

Our experience shows that—in contrast to the partial models *Environmental Model*, *Application Scenarios*, and *Active Structure*—the partial model *Behavior – Sequences* is often specified incompletely or not at all. Thus, this transition step is optional and does not completely replace manual work of the SW Requirements Engineer to specify MSDs.

DOI: 10.1002/smr

3.2.4. *Refine MSD Specification* The initially derived MSDs have no MSD-specific modeling constructs like modal aspects and conditions. This information going beyond the contents of plain UML Sequence Diagrams constitutes the formal semantics of MSDs and enables automatic analysis techniques like simulation. Thus, the initially derived MSD specification has to be refined. CONSENS provides information relevant to this refinement within the partial models *Application Scenarios* and *Requirements*. Since the contained information is informal text, we cannot automate this step. Thus, the SW Requirements Engineer has to perform it manually. Besides the automatically derived informal system model information specific to each MSD use case by means of UML notes, we provide informal guidelines to support the SW Requirements Engineer in this task. These guidelines encompass common refinement patterns for each aspect of a MSD specification, like the most likely source of information (e.g., the part "Situation" of *Application Scenarios*) in CONSENS or certain keywords (e.g., "shall").

We illustrate the refinement exemplarily on the MSD LockingFeedback. Fig. 4 depicts this MSD after the refinement step has been performed. The information that was initially derived (cf. MSD LockingFeedback in Fig. 3) is visualized in gray, and information that is added during the refinement step is emphasized by full colors.

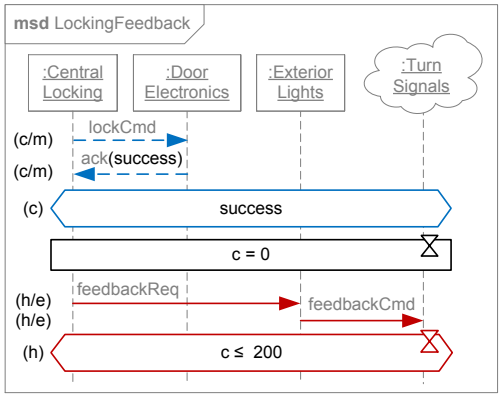

Figure 4. Manually refined MSD LockingFeedback

First, the SW Requirements Engineer has to analyze in which situation the visual feedback shall be given. The guidelines point him to the "Situation" of the corresponding application scenario Remote-based Locking (top right of Fig. 2). The first two messages `lockCmd` and `ack` represent the context of the MSD in which the visual feedback must be given. He sets the *temperature* of these messages to cold (depicted by blue message arrows and indicated by c on the left side of Fig. 4), meaning that other messages specified in the MSD can occur earlier or later. If this is the case, this "aborts" the current progress of the MSD. Furthermore, he sets their *execution kind* to monitored (depicted by dashed message arrows and indicated by m on the left side of Fig. 4), meaning that this message may or may not occur.

Second, the corresponding application scenario specifies that the visual feedback is only given if locking is successful. Thus, the requirements engineer adds a boolean parameter `success` to the message `ack` (and extends the corresponding operation in the interface **LockStatus_IF**, which is not depicted in Fig. 4). Furthermore, he adds a cold *condition* (depicted by the blue hexagon in Fig. 4) with the expression `success`, which evaluates to a Boolean value depending on the value assignment of `success`. If the condition evaluates to `false`, the MSD is discarded. Otherwise, it progresses beyond the condition.

Third, the SW Requirements Engineer analyzes the mandatory behavior of the application scenario. The guidelines instruct him to look for informal information containing keywords like "shall", which is the case for the requirement 5.6 (cf. Fig. 2). Since the intended behavior is to send the messages `feedbackReq` and `feedbackCmd` to activate the turn signal feedback operation, he sets their temperature to hot (depicted by red message arrows and indicated by h on the left side of Fig. 4). This means that other messages specified by the scenario must not occur at this point in time. Furthermore, he sets their execution kind to executed (depicted by solid message arrows and indicated by e on the left side of Fig. 4), which means that the messages must eventually occur.

Finally, requirement 5.7 in the CONSENS requirements specifies that the turn signal feedback operation has to be performed within 200ms. To formalize this requirement, the SW Requirements Engineer for one thing specifies a *clock reset* (depicted by the black rectangle with an hour-glass icon in Fig. 4) for the clock variable c, which sets c to zero. For another thing, he adds a hot *clock condition* (depicted by the red hexagon with an hour-glass icon in Fig. 4) at the end of the MSD. If

DOI: 10.1002/smr

its expression `c ≤ 200` evaluates to `false`, i.e., the sending of the messages `feedbackReq` and `feedbackCmd` lasts longer than 200ms, this is a safety violation.

Of course, the manual refinement of a MSD specification is not restricted to the aspects mentioned above. The SW Requirements Engineer can freely change all aspects of a MSD specification.

3.2.5. *Analyze Coordination Behavior Requirements* The formal semantics of MSDs enable different techniques for the tool-supported analysis of the requirements on the coordination behavior of the SUD. The SW Requirements Engineer is able to identify unintended behavior and scenario inconsistencies on requirements level (e.g., the unintended activation of MSD LockingFeedback by the MSD SpeedLockDoors. We refer to [18] for a detailed example).

3.2.6. *Consolidate Discipline-specific Analysis Results* Beyond the specification of *Behavior – Sequences* and other behavioral partial models, the SA continuously consolidates discipline-specific work products to ensure a reasonable system design. As the SA is not able to master all discipline-specific analysis techniques, it is of high importance to initiate an effective integration of all work products and to balance the needs of the disciplines.

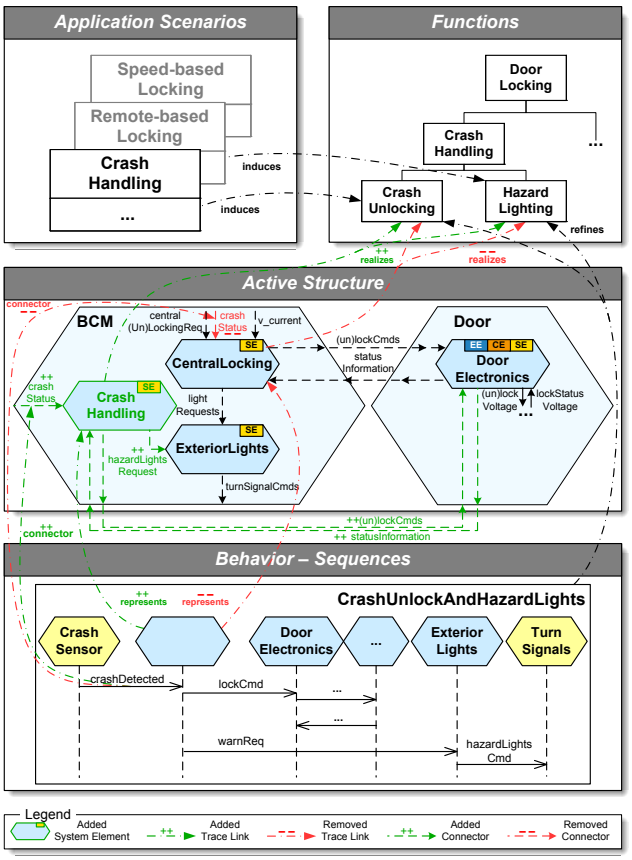

Figure 5. Changes in the CONSENS system model

The SW Requirements Engineer as well as the requirements engineers from the other engineering disciplines hand over their respective analysis results to the SA. He consolidates the different discipline-specific analysis results and decides how to proceed.

4. SUBSEQUENT PROCESS ITERATIONS

In this section, we present a solution on how changes in the system model are automatically handled by our approach. We exemplarily illustrate this change handling by extending our running example (cf. Fig. 5). To this end, we assume a change request that introduces an additional system element in the *Active Structure*. Such a use case implicates a broad range of changes to several of the introduced partial models. Applying these changes to the initial MSD specification presented in Sect. 3 manually would be very tedious and error-prone.

First, we illustrate the system model changes which are conducted by the systems engineer in detail. Afterwards, we present how these changes are applied to the MSD specification automatically in terms of an incremental update mechanism. We apply the graph transformation formalism *Story Diagrams* [27, 28] to visualize manual changes to the system model and automatically propagated changes to the MSD specification.

4.1. *Manual Changes to the CONSENS system model*

The application scenario Crash Handling is safety-critical since it describes functions to unlock the doors (Crash Unlocking) and enable the hazard lights (Hazard Lighting) in case of a crash (cf. functions in Fig. 5). Safety standards like ISO 26262 [29] demand dedicated and expensive safety

DOI: 10.1002/smr

measures in the development of system elements participating in such application scenarios. In the initial system model the system element CentralLocking realized these functions. However, it is also responsible for a multitude of functions that are not safety-critical (cf. application scenario Remote-based Locking). We assume a change request to reduce the effort spent on safety measures in the overall development of the system element CentralLocking. Consequently, as proposed by [29], the safety engineer decides to encapsulate the safety-critical behavior of this system element into the new, dedicated system element CrashHandling.

Several modifications have to be conducted in different partial models to fulfill this change request. In the following, we distinguish between two basic modification operations of Story Diagrams: Adding objects/links to our models (visualized by green outlines and the additional label "++") and deleting objects/links (visualized by red outlines and the additional label "--").

Impact on the partial model *Active Structure* In context of the change request, the Systems Engineer decides to add the new system element CrashHandling to the *Active Structure*. Furthermore, changes to the information flows are required. Three new information flows are added: 1. (un)lockCmds from CrashHandling to DoorElectronics allows the CrashHandling to unlock all doors in case of a crash, 2. statusInformation from DoorElectronics to CrashHandling informs CrashHandling about the current lock status of all doors, and 3. hazardLightsRequest from CrashHandling to ExteriorLights enables the CrashHandling to activate the hazard lights in case of a crash. Conclusively, the Systems Engineer moves crashStatus from CentralLocking to CrashHandling. This is visualized by an add and delete operation, respectively.

Impact on trace links Encapsulating the safety-critical behavior of the system element CentralLocking in the new system element CrashHandling implies several changes to the trace links between a total of three partial models.

First of all, the safety-critical functions CrashUnlocking and Hazard Lighting from the partial model *Functions* are to be realized by the new system element CrashHandling. Consequently, the Systems Engineer moves the existing trace links between CentralLocking and these functions to the new system element CrashHandling. This is indicated by a remove and add operation to the appropriate trace links named realizes.

Secondly, the safety-critical behavior associated to the system element CentralLocking needs to be associated to the system element CrashHandling. Thus, the Systems Engineer applies changes to two types of trace links that have its source in the partial model *Behavior Sequences*. He moves the trace link named represents between the lifeline that formerly represented the system element CentralLocking to the system element CrashHandling. Moreover, the trace links named connector between the messages in the sequence diagram and the flows of the *Active Structure* need to be updated accordingly. We exemplary show this for the trace link of the message crashDetected: The target flow is changed from the (old) incoming flow crashStatus of system element CentralLocking to the (new) incoming flow crashStatus of system element CrashHandling; the other trace links are not shown for readability reasons.

In summary, the rather small change to the system model is spread over a couple of partial models and involves several manual steps already. In the following we will present how these changes are reflected in the MSD specification in a systematic way, minimizing manual steps where possible.

4.2. *Automatic Incremental Updates of the MSD specification*

Everything that is subject to the initial automatic transformation (cf. last paragraph of Sect. 3.2 and Sect. 3.2.1 to 3.2.3) is incrementally updated when changes to the system model occur. Particularly, no manually added information (cf. Sect. 3.2.4) gets lost in this case. The change introduced in the last section influences all three views of a MSD specifciation. In the following, we present the changes for the MSD specification in terms of Fig. 6.

As explained in the last section, the newly added system element CrashHandling undertakes the crash handling tasks of CentralLocking in a second development process iteration. To reflect this change in the system model, the system element CrashHandling is assigned via the partial model

Prepared using smrauth.cls
DOI: 10.1002/smr

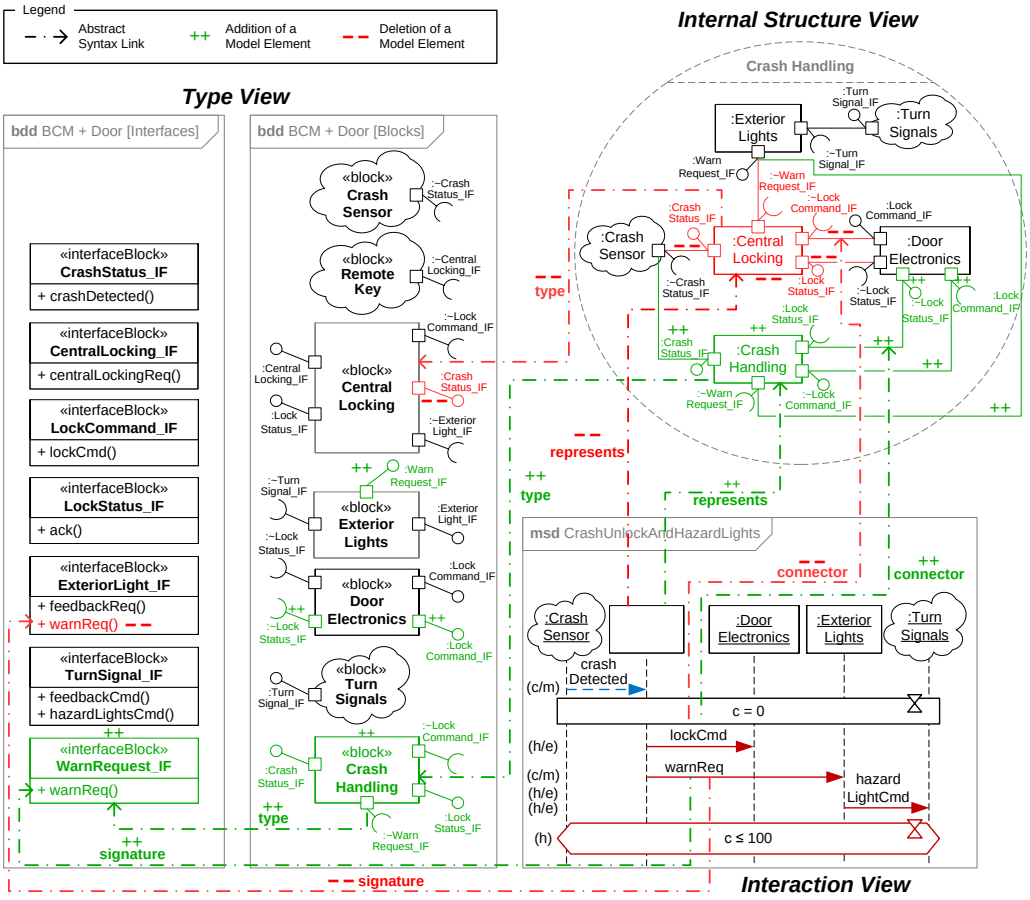

Figure 6. Automatically updated MSD specification

Functions to the application scenario Crash Handling, whereas CentralLocking loses this assignment (cf. Fig. 5). Therefore, the incremental update adds the role :CrashHandling and removes the role :CentralLocking including their incident connectors to/from the MSD use case Crash Handling in the Internal Structure View (cf. top right of Fig. 6). The existing connectors to :CentralLocking are removed from the ports of the interacting roles :CrashSensor and :ExteriorLights, and new connectors are added between these ports and the corresponding ports of :CrashHandling. In contrast, two additional connectors are added to the system element DoorElectronics in the partial model *Active Structure* (cf. Fig. 5). That is, the existing ports of :DoorElectronics interact with :CentralLocking in the other use cases in terms of the MSD specification. Thus, two new ports are added to :DoorElectronics and connected to :CrashHandling, and the connectors of the existing ports of :DoorElectronics are removed.

Some abstract syntax links to the Type View as well as its contents (cf. left of Fig. 6) have to be adapted to gain a type-compliant Internal Structure View. For example, the incremental update has to add a block **CrashHandling** including ports and an abstract syntax link from the corresponding role to it. The blocks **CrashHandling** and **ExteriorLights** get ports typed by the new interface **WarnRequest_IF** such that they are able to communicate with each other (cf. the corresponding information flow hazardLightsRequest added to the *Active Structure* in Fig. 5 and the corresponding connector added to the Internal Structure View in Fig. 6). In the course of adding the information flow hazardLightsRequest from the system element CrashHandling to ExteriorLights, the Systems Engineer decides to remove the possibility of CentralLocking to send a warnReq signal via this flow (not depicted in Fig. 5) to ExteriorLights. Thus, the corresponding operation `warnReq()` is removed from the interface **ExteriorLight_IF** in the Type View of the MSD specification. The port :CrashStatus_IF of **CentralLocking** is removed, whereas **CrashHandling** gets an identically

DOI: 10.1002/smr

named port (cf. movement of the ingoing information flow crashStatus from CentralLocking to CrashHandling in Fig. 5). The last change to the Type View adds two additional ports :~LockStatus_IF and :LockCommand_IF to **DoorElectronics** allowing the additional ports of the corresponding role in the Internal Structure View.

After the structural changes have been performed, the Interaction View (cf. bottom right of Fig. 6) can be updated. First, the end of the abstract syntax link named represents of the lifeline formerly representing the role :CentralLocking is moved to the newly added role :CrashHandling. Second, the ends of all abstract syntax links named connector of this lifeline's ingoing and outgoing messages are moved to the changed incident connectors of :CrashHandling. We exemplarily visualize this for the message `lockCmd`; the other abstract syntax links are not shown for readibility reasons. Last, the signatures of messages have to be changed if the corresponding interfaces have changed. This is the case for the message `warnReq`. Thus, the end of its abstract syntax link signature is moved to the corresponding operation of the new interface **WarnRequest_IF**.

In summary, our example shows that a rather small change in the system model can imply a multitude of changes in the MSD specification in terms of concrete and abstract syntax. Thus, the automatic incremental update of our transition technique saves a lot of manual effort on keeping both models consistent. Currently, the SW Requirements Engineer has to avoid changes in the MSD specification without contacting the Systems Engineer since our transition technique is not bidirectional. However, this approach is compliant to the MBSE idea envisioning the system model to orchestrate the particular engineering disciplines.

5. IMPLEMENTATION

We documented the transition technique and described as well as implemented the particular mapping rules from a CONSENS system model to a MSD specification in [26]. We implemented the mappings with the Eclipse implementation [30] of QVT Operational (QVT-O) [31] model transformations. The source model is a system model specified in a CONSENS language variant based on the SysML profile SYSML4CONSENS [32] in the UML/SysML modeling tool Papyrus [33]. The target model is a MSD specification for hierarchical component architectures [14] based on the UML/SysML modal profile provided by the SCENARIOTOOLS MSD [34] tool suite. Since both the source and target model make use of the UML/SysML profiling extension mechanism, the approach can also be implemented in other UML/SysML tools. The current version 3.5.0 of the QVT-O Eclipse implementation is also capable of additively updating the target model w.r.t. the given mappings without deleting elements that were added in the course of the manual refinement. Thus, we apply this feature for our incremental updates.

Algorithm 1 presents an abstract pseudocode version of the overall implementation, which is exemplarily executed in the last two sections. The algorithm enables the replicability to other MBSE and SWRE approaches as pointed out in the related work section. There are two execution scenarios for this algorithm: The initial generation of a MSD specification and the incremental update of an existing one. The system model is passed as value for the input parameter $systemModel$ in both application scenarios. In the first execution scenario, the two other inout parameters $msdSpecification$ and $qvtoTraceFile$ are initially set to `null`. After the initial generation, they contain the generated MSD specification and a QVT-O trace file automatically generated by the QVT-O implementation. A trace file contains all trace links between mapped elements created by QVT-O during the transformation. In the second execution scenario, the existing MSD specification and the existing trace file are passed to the algorithm as values for the corresponding parameters. After the incremental update, these parameters contain the updated MSD specification and the updated trace file. The operation `map` is provided by the QVT-O implementation and calls the QVT-O mappings implemented by us. `map` generates an object of a given type in the target model out of a given object in the source model. Additionally, QVT-O creates a trace link between the source object and the generated target object in the trace file. However, `map` is only executed if no trace link from the source object to any target type object exists, such that no superfluous target objects are created if the transformation is executed multiple times. The operation `resolve` is provided by the QVT-O implementation and uses the trace links to determine a target type object from a given source object.

DOI: 10.1002/smr

Algorithm 1 Derive MSD Specification

```
1: procedure DeriveMSDSpecification(in systemModel, inout msdSpecification, inout qvtoTraceFile)
2:   for all appScenario ∈ systemModel.applicationScenarios do
3:     SE-relevantSystemElements := sysElem ∈ systemModel.activeStructure | (∃ funct ∈
   systemModel.functions | (appScenario induces funct ∧ sysElem realizes funct ∧ sysElem is SE-relevant))
4:     if SE-relevantSystemElements <> ∅ then                    ▷ Application Scenario is SE-relevant
5:       msdUseCase := map appScenario to Collaboration          ▷ Derive MSD Use Cases (cf. Sect. 3.2.1)
6:       msdSpecification += msdUseCase
7:       for all systemElement ∈ SE-relevantSystemElements do    ▷ Derive Structure (cf. Sect. 3.2.2)
8:         block := map systemElement to SystemBlock
9:         for all port ∈ systemElement.ports | (port is connected through an InformationFlow) do
10:          block += map port to Port                           ▷ Mapping of interfaces not shown
11:        msdRole := systemElement map to Property
12:        msdRole.type := block
13:        msdSpecification += block
14:        msdUseCase += msdRole
15:      for all envElem ∈ systemModel.environment | (envElem affects appScenario ∧
   ∃ informationFlow | (informationFlow connects envElem to e ∈ SE-relevantSystemElements)) do
16:        environmentBlock := map envElement to EnvironmentBlock
17:        ...                                                   ▷ Rest analogous to system elements
18:      for all informationFlow ∈ (systemModel.activeStructure ∪ systemModel.environment) do
19:        if informationFlow.connectedElements were mapped to roles in msdUseCase then
20:          msdRolesToConnect := resolve informationFlow.connectedElements to roles in msdUseCase
21:          msdUseCase += Connector between msdRolesToConnect
22:      for all behaviorSequence ∈ systemModel.behaviorSequences do   ▷ Derive MSDs (cf. Sect. 3.2.3)
23:        if behaviorSequence refines appScenario then
24:          msd := map behaviorSequence to MSD
25:          for all behaviorSequenceLifeline ∈ behaviorSequence.lifelines do
26:            ...                                               ▷ Map Lifelines of Behavior Sequences to Lifelines of MSDs
27:          for all behSeqMsg ∈ behaviorSequence.messages do
28:            ...                                               ▷ Map Messages of Behavior Sequences Messages of MSDs
             msdUseCase += msd
29:  if qvtoTraceFile <> null then   ▷ Extension for deletion in updates (Java black-box library, highly abstracted)
30:    for all traceRecord ∈ qvtoTraceFile do
31:      if traceRecord.getSourceModelElement() was deleted from systemModel then
32:        delete traceRecord.getTargetModelElement() from msdSpecification
```

In contrast to the additive incremental updates described above, the current version of the QVT-O implementation cannot delete target model elements after the corresponding source model elements were deleted. To overcome this problem, we use a Java black-box library that analyzes the trace file and explicitly propagates deletions of elements in the system model to deletions of the corresponding elements in the MSD specification similar to the proposal in [35] (cf. bottom part of Algorithm 1).

6. CASE STUDY

We conduct a case study based on the guidelines by Kitchenham et al. [36] for the evaluation of our approach. In our case study, we investigate the usefulness of our approach within our domain. We perform the case study based on the running example in this paper and do not aim at generalizing the case study conclusions to all possible CONSENS system models and MSD specifications.

6.1. *Case Study Context*

The objective of our case study is evaluating whether our transition technique is useful for the SW Requirements Engineer, i.e., whether it reduces his manual effort and guides him with reasonable MSD specifications as a basis for the manual refinements. For this purpose, we use an extended version of the presented running example that comprises both the initial as well as the iteratively changed CONSENS system model and MSD specification. Since our tooling bases on SYSML4CONSENS (cf. Sect. 5), we use a SYSML4CONSENS model in the case study.

6.2. *Setting the Hypotheses*

We define two evaluation hypotheses for our case study. Our first evaluation hypothesis H1 is that our transition technique reduces the engineering effort for conceiving and modeling a MSD

DOI: 10.1002/smr

specification based on a SYSML4CONSENS system model. Our second evaluation hypothesis H2 is that our transition technique guides the SW Requirements Engineer by generating and updating a correct MSD specification w.r.t. to a SYSML4CONSENS system model. We consider a MSD specification *correct* iff all SE-relevant and no SE-irrelevant elements of the SYSML4CONSENS model are present in the automatically generated/updated MSD specification (cf. Algorithm 1).

For evaluating H1, we determine the amounts of the overall (H1.1) and the SE-relevant (H1.2) model elements in the initial SYSML4CONSENS system model (cf. Sect. 3.1). Furthermore, we determine the amounts of model elements that are initially generated (H1.3) (cf. Sect. 3.2.1 to Sect. 3.2.3) and manually refined (H1.4) (cf. Sect. 3.2.4) in the MSD specification. We determine the model element amounts (H1.1), (H1.3), and (H1.4) automatically based on the respective models. We determine (H1.2) manually since the filtering of SE-relevant model elements is hidden in the transformation. Thus, there is no corresponding intermediate SE-relevant SYSML4CONSENS system model. On the counting of model elements, we orientate towards the manual modeling within the corresponding tools (cf. Sect. 5) since the granularity level w.r.t. the question "what is a distinct model element?" is very arguable. We do not count the generation of informal SYSML4CONSENS information into comments in the MSD specification for (H1.3) and (H1.4), because they do not contribute to the MSD core semantics but represent additional information. We follow a similar approach for the changed SYSML4CONSENS system model (cf. Sect. 4.1) and the updated MSD specification (cf. Sect. 4.2) but focus on the changed model elements. For this purpose, we define the variables (H1.1-update) for the amount of system model elements that were manually changed and (H1.2-update) for the amount of MSD specification model elements that were automatically updated due to the system model changes. We determine these amounts manually.

For evaluating H2, we give the initial SYSML4CONSENS system model (cf. Sect. 3.1) and the initially generated MSD specification (cf. Sect. 3.2.1 to Sect. 3.2.3) to a MSD expert. The expert has approx. one year of experience in modeling and simulating MSD specifications but has little experience with CONSENS. The expert qualitatively evaluates whether all SE-relevant (H2.1) and whether no SE-irrelevant (H2.2) elements of the SYSML4CONSENS model are present in the initial MSD specification. In the same way, the expert investigates the changed SYSML4CONSENS model (cf. Sect. 4.1) and updated MSD specification (cf. Sect. 4.2) and evaluates (H2.1) and (H2.2) again.

6.3. *Preparing the Input Models*

The SYSML4CONSENS system model has to be complete and correct w.r.t. to the mapping rules of the transformation to be subject to a successful model transformation. For example, all trace links as specified in Fig. 2 have to be specified, all ports with incident information flows have to be typed with the corresponding port specifications, and the relevance annotations have to be specified. For this purpose, we specify a SYSML4CONSENS system model that adheres to the modeling constraints informally specified in [26, Sect. 6.1] in preparation of the case study. Furthermore, we prepare a variant of this system model as presented in Sect. 4 for the evaluation of the incremental updates.

6.4. *Validating the Hypotheses*

The determination of the model element amounts for H1 yields the results as listed in Table I. In terms of relationships between the amounts for the initial generation, ~40% (H1.2/H1.1) SE-relevant elements are filtered in the SYSML4CONSENS system model and ~87% (H1.3/H1.4) of the final MSD specification is generated. The amounts for the incremental update yield that manually changing 31 system model elements (H1.1-update) results in automatically updating 22 model elements in the MSD specification (H1.2-update). The lesser amount of the updated MSD specification model elements in comparison to the amount of the system model elements arises from the flattening of the architecture hierarchies in the transformation to the MSD specification.

The evaluation of H2 yields that (H2.1) all SE-relevant elements and (H2.2) no superfluous SE-irrelevant elements are present in the initially generated MSD specification. This also encompasses the informal system model information (i.e., information from the partial models *Application Scenarios* and *Requirements*), which is specifically aggregated and annotated by the automatic transformation for each respective MSD use case. Particularly, the expert discovers that these annotations are very

DOI: 10.1002/smr

ID	Model Element Amount Variable	Value
(H1.1)	Overall system model elements	606
(H1.2)	Automatically filtered SE-relevant system model elements	240
(H1.3)	Automatically generated MSD specification model elements	263
(H1.4)	Overall MSD specification model elements after manual refinement	304
(H1.1-update)	Changed system model elements	31
(H1.2-update)	Automatically updated MSD specification elements	22

Table I. Results for model element amount variables for hypothesis H1

helpful, because they provide much more context information than the mere MSD specification. If these annotations would not be generated, he would have to awkwardly identify this information in the system model in a manual way. In terms of the incremental update, the expert performs the transformation on the changed system model. The evaluation again yields that all SE-relevant elements (H2.1) and no superfluous SE-irrelevant elements (H2.2) are present in the updated MSD specification. Furthermore, the expert discovers that none of his manual refinements from the initial process iteration are lost.

6.5. *Analyzing the Results*

The results for H1 show three aspects. First, a large part of the system model is automatically filtered such that the SW Requirements Engineer does not have to care about the SE-irrelevant model elements. Second, the biggest part of the MSD specification is automatically generated and only a small amount has to be specified manually. Third, the results for the incremental update indicate that the amount of automatically updated MSD specification model elements is approximately equal to the manually changed system model elements. All these activities would have been performed completely manually without our transition technique. Thus, we consider H1 as fulfilled. The results for H2 show that the generated and updated MSD specification is reasonable in the sense that it only encompasses SE-relevant modeling elements. Thus, we also consider H2 as fulfilled.

In summary, the fulfilled hypotheses yield that our transition technique reduces the manual effort of the SW Requirements Engineer and guides him with reasonable MSD specifications. This gives rise to the assumption that the approach is indeed useful for him.

The most important threats to validity in our case study are as follows: (1) We only considered one example and thus cannot generalize the fulfillment of the statements. Nevertheless, the example is typical for a software-intensive technical system and hence we do not expect large deviations for other systems. (2) The domain expert has little experience with CONSENS and thus could have judged incorrectly whether only SE-relevant elements are present in the MSD specification. Anyhow, the domain expert could judge the correct execution by investigating the SYSML4CONSENS model with relevance annotations and reenacting Algorithm 1. (3) The manual counting of model elements in the context of (H1) might be incorrect. However, we do not expect a large discrepancy between actual and counted elements such that the derived conclusion regarding (H1) is not affected.

7. RELATED WORK

Similar to our work, Böhm et al. [37] present a model-based methodology for the transition from systems to software engineering that is compliant to established process models. They also address that these process models demand a software requirements analysis phase prior to the software design: A system element to be concretized in software engineering is input to the context model of the SPES requirements viewpoint [38]. This context model is comparable to our *Environmental Model* on system level. However, they only consider structural but no behavioral models, which are crucial for the specification of functional requirements. Furthermore, they do not provide automatisms to support the transition.

Barbieri et al. [39] present a systems engineering approach based on the SysML extension SysML4*Mechatronics*. They describe a coarse-grained design process and introduce stereotypes for system elements to be concretized in their respective engineering disciplines. These stereotypes are similar to our relevance annotations. However, the approach aims on analyzing discipline-spanning

DOI: 10.1002/smr

change influences and not on the transition to discipline-specific models. Furthermore, the discipline-specific information in the system model is not exploited to facilitate automated tool support.

The SysML-based MBSE methods Object-Oriented Systems Engineering Method (OOSEM) [40] and SYSMOD [41] serve the same purpose as CONSENS. OOSEM only distinguishes between software and hardware, and SYSMOD promotes a more software-focused approach since the hardware elements are often considered as environment elements. In contrast, CONSENS promotes a cross-disciplinary approach facilitating the communication between all engineering disciplines at eye level. Furthermore, OOSEM and SYSMOD only propose to refine the system requirements within a typical software engineering process, but the transition to this process is out of their scope. Nevertheless, the basic principle of our transition from MBSE to SWRE could also be transferred to OOSEM and SYSMOD by applying Algorithm 1 to the underlying SysML extensions and adapting the manual steps to the methods.

Several approaches exist to systematize and automate the transition from MBSE to other discipline-specific design and analysis models, e.g., in the area of control engineering. The OMG developed a bidirectional transformation from SysML to Modelica [42]. Similarly, Cao et al. present an integration between SysML models with MATLAB/Simulink based on a bidirectional model transformation [43]. Such approaches focus on the derivation of discipline-specific design and analysis models. In contrast to our work, the derivation or refinement of discipline-specific requirements from system requirements is not addressed.

8. CONCLUSIONS AND OUTLOOK

In previous work [18], we introduced a systematic process to integrate SWRE into MBSE based on CONSENS and MSDs. We assigned the particular process steps to different roles involved in the integrated development process. We screened the CONSENS language and identified six partial models in CONSENS that contain information relevant to SWRE. The presented technique enables to filter the SWRE-relevant information from these interdisciplinary partial models and automates process steps by means of model transformations where possible. This automation enabled the generation of initial MSD specifications in one process iteration. In this paper, we extended this approach by an automated incremental update of the MSD specification according to changes in the system model in subsequent process iterations. Furthermore, we extended the transformation by annotating informal system model information to the MSD specification, provided an algorithm for the automated steps, and conducted a case study to investigate the usefulness of the technique.

The transition process guides the software requirements engineer to perform the particular steps in a systematic way. Thereby, the task assignment to roles clarifies the distribution of responsibilities. Due to our screening, the SW Requirements Engineer is aware of the partial models that he has to analyze. Our process automation increases his efficiency by reducing manual work, which is time-consuming and error-prone. Furthermore, the automatically derived parts of a MSD specification assure that he does not overlook SWRE-relevant information. The incremental updates facilitate the application in iterative settings. The algorithm enables the replicability to other MBSE and SWRE approaches, and the case study indicates the usefulness of the approach.

Future work is diverse. First, we want to combine our approach with the existing approaches for the transition from MBSE with CONSENS to the software design for the coordination and control behavior. That is, we want to combine it, on the one hand, with the transition to software design models [16] including the inter-component coordination behavior [17]. On the other hand, we want to combine it with the transition to the software design using solution patterns [17] to enable the reuse of the involved models. Second, we also want to consider requirements for the control behavior and other disciplines to enable an interdisciplinary requirements engineering. Finally, we want to extend our approach with bidirectional model synchronization techniques. This enables propagating changes in the MSD specification influencing the system model back to CONSENS automatically.

DOI: 10.1002/smr

REFERENCES

1. Walden DD, Roedler GJ, Forsberg KJ, Hamelin RD, Shortell TM (eds.). *Systems Engineering Handbook: A Guide for System Lifecycle Processes and Activities*. 4^{th} edn., Wiley, 2015.
2. INCOSE. *A World in Motion: Systems Engineering Vision 2025* 2014.
3. Schäfer W, Wehrheim H. The challenges of building advanced mechatronic systems. *FOSE*, ACM, 2007; 72–84.
4. Boehm BW. Seven basic principles of software engineering. *J. Syst. Software* 1983; **3**(1):3–24.
5. VDA QMC WG 13 / Automotive SIG. Automotive SPICE process reference / assessment model: Version 3.0 2015.
6. Pohlmann U, Holtmann J, Meyer M, Gerking C. Generating Modelica models from software specifications for the simulation of cyber-physical systems. *SEAA*, IEEE, 2014; 191–198
7. Nuseibeh B, Easterbrook S. Requirements engineering: A roadmap. *FOSE*, ACM, 2000; 35–46.
8. Pohl K, Rupp C. *Requirements Engineering Fundamentals*. Rocky Nook, 2011.
9. Harel D, Maoz S. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling* 2008; 7:237–252.
10. Harel D, Marelly R. *Come, let's play: Scenario-based programming using LSCs and the play-engine*. Springer, 2003.
11. Greenyer J, Brenner C, Cordy M, Heymans P, Gressi E. Incrementally synthesizing controllers from scenario-based product line specifications. *ESEC/FSE*, ACM, 2013; 433–443.
12. Brenner C, Greenyer J, Panzica La Manna V. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. *GT-VMT*, 2013.
13. Brenner C, Greenyer J, Holtmann J, Liebel G, Stieglbauer G, Tichy M. ScenarioTools real-time play-out for test sequence validation in an automotive case study. *GT-VMT*, 2014.
14. Holtmann J, Meyer M. Play-out for hierarchical component architectures. *Proc. 11^{th} Workshop Automotive Software Engineering, GI-Edition - Lecture Notes in Informatics (LNI)*, vol. P-220, Bonner Köllen, 2013; 2458–2472.
15. Gausemeier J, Rammig FJ, Schäfer W (eds.). *Design Methodology for Intelligent Technical Systems: Develop Intelligent Technical Systems of the Future*. Lecture Notes in Mechanical Engineering, Springer, 2014
16. Heinzemann C, Sudmann O, Schäfer W, Tichy M. A discipline-spanning development process for self-adaptive mechatronic systems. *ICSSP*, ACM, 2013; 36–45.
17. Anacker H, Gausemeier J, Dumitrescu R, Dziwok S, Schäfer W. Solution patterns of software engineering for the system design of advanced mechatronic systems. *MECATRONICS REM*, IEEE, 2012; 101–108.
18. Holtmann J, Bernijazov R, Meyer M, Schmelter D, Tschirner C. Integrated systems engineering and software requirements engineering for technical systems. *ICSSP*, ACM, 2015; 57–66.
19. Sheard SA. Twelve systems engineering roles. *INCOSE Intl. Symposium* 1996; **6**(1):478–485, doi:10.1002/j. 2334-5837.1996.tb02042.x.
20. Ryschkewitsch M, Schaible D, Larson W. The art and science of systems engineering. *Systems Research Forum* 2009; **3**(2):81–100.
21. Object Management Group. Business Process Model and Notation (BPMN): Version 2.0.2 2014.
22. Boehm BW. Unifying software engineering and systems engineering. *Computer* 2000; **33**(3):114–116.
23. Guest D. The hunt is on for the renaissance man of computing. *The Independent (London)* 1991; **17**.
24. Object Management Group. OMG Unified Modeling Language (OMG UML): Version 2.5 2015.
25. Object Management Group. OMG Systems Modeling Language (OMG SysML): Version 1.4 2015.
26. Bernijazov R. Systems and software requirements engineering for cyber-physical systems. Bachelor's thesis, Paderborn University 2015.
27. Fischer T, Niere J, Torunski L, Zündorf A. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. *TAGT, LNCS*, vol. 1764, Springer, 2000; 157–167.
28. Tichy M, Henkler S, Holtmann J, Oberthür S. Component story diagrams: A transformation language for component structures in mechatronic systems. *OMER4*, 2008; 27–38.
29. International Organization for Standardization. ISO 26262: Road vehicles – functional safety 2011.
30. Eclipse QVT Operational: http://projects.eclipse.org/projects/modeling.mmt.qvt-oml. Last accessed Feb. 2016.
31. Object Management Group. Meta Object Facility 2.0 Query/View/Transformation specification: Version 1.2 2015.
32. Kaiser L, Dumitrescu R, Holtmann J, Meyer M. Automatic verification of modeling rules in systems engineering for mechatronic systems. *ASME IDETC/CIE*, ASME, 2013.
33. Papyrus modeling environment: http://www.eclipse.org/papyrus. Last accessed Feb. 2016.
34. SCENARIOTOOLS MSD: http://scenariotools.org/projects2/msd. Last accessed Feb. 2016.
35. Romeikat R, Roser S, Müllender P, Bauer B. Translation of QVT relations into QVT operational mappings. *ICMT, LNCS*, vol. 5063, Springer, 2008; 137–151.
36. Kitchenham B, Pickard L, Pfleeger SL. Case studies for method and tool evaluation. *IEEE Software* 1995; **12**(4):52–62, doi:10.1109/52.391832.
37. Böhm W, Henkler S, Houdek F, Vogelsang A, Weyer T. Bridging the gap between systems and software engineering by using the SPES modeling framework as a general systems engineering philosophy. *Systems Engineering Research*, 2014; 187–194.
38. Daun M, Tenbergen B, Weyer T. Requirements viewpoint. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. chap. 4, Springer, 2012; 51–68.
39. Barbieri G, Kernschmidt K, Fantuzzi C, Vogel-Heuser B. A SysML based design pattern for the high-level development of mechatronic systems to enhance re-usability. *IFAC World Congress*, IFAC, 2014; 3431–3437.
40. Friedenthal S, Moore A, Steiner R. *A Practical Guide to SysML*. 3^{rd} edn., Morgan Kaufmann, 2014.
41. Weilkiens T. *SYSMOD – The Systems Modeling Toolbox: Pragmatic MBSE with SysML*. Leanpub, 2015.
42. Object Management Group. OMG SysML-Modelica Transformation: Version 1.0 2012.
43. Cao Y, Liu Y, Paredis CJ. System-level model integration of design and simulation for mechatronic systems based on SysML. *Mechatronics* 2011; **21**(6):1063 – 1075.

DOI: 10.1002/smr