A Requirements Engineering Methodology Combining Models and Controlled Natural Language

Markus Fockel, Jörg Holtmann
Software Engineering, Project Group Mechatronic Systems Design, Fraunhofer Institute for Production Technology IPT
Zukunftsmeile 1, 33102 Paderborn, Germany

Abstract—**The use of models in requirements engineering (RE) for software-intensive embedded systems is considered beneficial. The main advantages of requirements models as documentation format are that they facilitate requirements understanding and foster automatic analysis techniques. However, natural language (NL) is still the dominant documentation format for requirements specifications, particularly in the domain of embedded systems. This is due to the facts that NL-based requirements can be used within legally binding documents and are more appropriate for reviews than models. In order to bridge the gap between both of these documentation formats, this paper proposes a model-driven RE methodology that makes use of requirements models along with a controlled natural language. The methodology combines the advantages of model-based and NL-based documentation by means of a bidirectional multi-step model transformation between both documentation formats. We illustrate the approach by means of an automotive example, explain the particular steps of the model transformation, and present performance results.**

I. Introduction

The use of models in requirements engineering (RE) for software-intensive embedded systems is considered beneficial [26]. The main advantages of requirements models as documentation format are that they facilitate requirements understanding [18] by raising the abstraction level in requirements descriptions [3] and foster automatic analysis techniques. However, natural language (NL) is still the dominant documentation format for requirements specifications, particularly in the domain of embedded systems [26]. The most important reason for this is that all stakeholders—despite possibly different educational backgrounds—are already familiar with the use of NL and therefore need not to learn special nomenclature, as it would be the case with formal modeling notations. NL is hence easy to use [23]. This results in the possibility to use NL requirements within legally binding documents [26] (i.e., to use them as the basis of contractual agreements [18], [26] and as argumentation basis to satisfy domain-specific safety standards [26]). A further advantage of NL requirements is that they are more appropriate for reviews than models [10].

In order to bridge the gap between both documentation formats and to combine their respective advantages (cf. [18]), we propose in this paper a model-driven RE methodology that makes use of requirements models along with a controlled natural language (CNL) (e.g., [29]). A CNL restricts the expressiveness of natural language by only allowing certain formulations, phrases, and a restricted vocabulary. Therefore, we also regard a CNL as a kind of model. Furthermore, this enables an easy automatic processing of the textual requirements. In the methodology, requirements models are applied to elicit, document, and negotiate requirements. This model-driven approach satisfies the demand of requirements engineers to apply models during the actual RE [26], thereby facilitating requirements understanding as well as fostering automatic analysis techniques. Automatic transitions to CNL can be used for a subset of the model artifacts in adequate phases of the RE methodology, enabling the usage within legally binding documents as well as stakeholder reviews in a document-oriented form. Possible changes in the CNL requirements can be propagated back into the requirements model to ensure synchronization and traceability between both documentation formats, as proposed by [18].

The contribution of this paper is twofold. First, we combine two previously conceived, independent RE methods—one model-based and one CNL-based approach—into an integrated, model-driven RE methodology. For this purpose, we extend the model-based approach to comply with the CNL-based approach to be applicable within our combined methodology. This allows arbitrarily many iterations across several abstraction layers of the specified requirements artifacts. Second, we present a multi-step model transformation—previously sketched in [9] as part of a whole development process—for switching between both documentation formats. Thereby, we explain the relations between both documentation formats on the metamodel level. Furthermore, we present performance results of the model transformation.

This paper is structured as follows. Sect. II illustrates the related work regarding the integration of model-based and NL-based RE. Sect. III introduces the previously conceived RE approaches. The methodology combining these two approaches is presented in Sect. IV. Sect. V shows the applicability of the approach by showing an excerpt of an automotive case study [4]. We explain the model transformation in Sect. VI and evaluate its performance in Sect. VII. Sect. VIII summarizes this document and provides an outlook on future work.

II. Related Work

A systematic review of approaches that generate requirements models from natural language has been conducted by

Yue et al. [30]. Examples for such approaches are [11], [1], [15], [6], which process NL requirements and generate UML-like analysis models. The approaches differ in the degree of freedom of the NL (more or less restricted [1], [6] vs. unrestricted [11], [15] NL), the degree of automation (semi-automatic [6] vs. fully automatic [11], [1], [15]), and the kind of generated models (structural domain models [11], [1], [15], [6] vs. use case models [1], [15], [6] vs. scenarios [15] vs. state machines [1]).

A systematic literature review of approaches that generate natural language requirements from software engineering models has been presented in [18]. Examples for such approaches are [7], [17]. In [7], a process is described that translates functional and behavioral models such as UML activities and Message Sequence Charts into natural language requirements. Similarly, Meziane et al. [17] introduce an approach that derives natural language requirements specifications from UML class diagrams. For this purpose, a rule set is used in conjunction with a linguistic ontology in order to express the diagram components.

Summarizing, in the literature there are either approaches that generate requirements models from NL requirements or approaches that derive NL requirements from requirements or software engineering models. Yet, no approach fosters the benefits of both documentation formats by strategically incorporating the automatic transition in both directions (from requirements models to NL and vice versa). According to [18], an automatic synchronization between both documentation formats is useful to enable an iterative evolution of both documentation formats.

III. Original Requirements Engineering Approaches

In the following, we sketch two RE approaches that constitute the foundation for the combined RE methodology in Sect. IV. Sect. III-A discusses a model-based RE approach. Sect. III-B illustrates a RE approach based on CNL.

A. Model-based Requirements Engineering

The Requirements Viewpoint [5] of the SPES Modeling Framework [2] introduces a model-based representation for common requirements artifacts. The purpose of the SPES Requirements Viewpoint is to document requirements for software-intensive embedded systems in a systematic and seamless model-based manner and hence fosters structured elicitation of requirements. Specifically, the Requirements Viewpoint supports the developers in differentiating stakeholder intentions and problem description. For this purpose, the SPES Requirements Viewpoint features stepwise, artifact-based refinement and allows for traceability between requirements artifacts.

The Requirements Viewpoint features the following four essential artifact types.

1) Context Models: This model type documents interfaces of the system under development (SUD) with entities in its operational context (e.g., external systems or human users). Entities of the operational context are those entities in the SUD's context that directly interact with the SUD and exchange inputs as well as outputs.

2) Goal Models: This model type documents solution-neutral goals and stakeholder intentions regarding the intended functionality of the system as well as desired qualities. Goal models serve as a rationale for solution-oriented requirements.

3) Scenario Models: This model type documents typical interactions between the SUD or its internal subsystems and context entities. Furthermore, scenario models show exemplary fulfillment of the goals specified in goal models.

4) Solution-Oriented Requirements Models: This model type documents functional requirements of the SUD completely and precisely. Solution-oriented requirements can be documented in three perspectives: the static-structural requirements (e.g., the information structure of the system), operational requirements (e.g., requirements pertaining to concrete user functions), and behavioral requirements (e.g., externally observable system states). Using three distinct perspectives, it is possible to document requirements with a concrete reference to a solution concept.

The goal and scenario models consider the SUD only as black box. That is, these models are specified based on the context model and do not consider internal subsystems of the SUD. As subsequent development artifact, the SPES Modeling Framework proposes the so-called Functional Viewpoint [28] consisting of a function hierarchy as white box specification for the SUD. This function hierarchy is manually conceived based on the Requirements Viewpoint.

B. Requirements Engineering with Requirement Patterns

In previous work, we conceived a seamless, model-based design methodology for automotive systems with focus on suppliers [8], [9], [12]. In this methodology we use so-called *Requirement Patterns* as CNL for the specification of functional requirements [13]. This CNL restricts the expressiveness of natural language to enable automatic processing of the requirements while still keeping them understandable for all stakeholders.

Requirement patterns are a means to describe the functionality of the SUD. The patterns allow decomposing the overall system functionality across several abstraction layers. These layers decompose the functionality into so called *systems* (i.e., a grouping of functionality) down to atomic *functions* on the final layer. The input and output signals required and provided by the different systems and functions are also specified. Similar to the method of Structured Analysis (e.g., [24]) and the SPES Functional Viewpoint [28], a function hierarchy that spans a tree with functions as leaves is conceived by using requirement patterns. While decomposing the complete system across subsystems into functions, the inputs and outputs of an element of one abstraction layer are partitioned onto elements of the next deeper abstraction layer in order to cope with the overall complexity of the SUD. In the subsequent development process, based on the function hierarchy, a logical architecture is developed manually and linked to the functions of the function hierarchy to preserve requirements traceability.

Two example requirement patterns that are used in the example in Sect. V to describe the function hierarchy are listed below. Requirement patterns consist of static, variable (contained in '<', '>'), alternative ('|'), and optional ('[', ']') parts. A more comprehensive list of the requirement patterns and their applications can be found in [4].

1) The system <*system*> consists of the following subsystem[s]: <*subsystem list*>.
2) The (system <*system*> | function <*function*>) (processes | creates) the following signal[s]: <*signal list*>.

Furthermore, we also provide requirement patterns for so-called solution-oriented requirements (cf. Sect. III-A4). These describe, for example, timing or safety requirements. Since these requirement patterns are not part of this paper due to space limitations, we refer to our technical report [4] for their presentation.

CNL-based requirements can be automatically processed. This allows to assist the user while writing requirements and to perform automatic quality checks [13]. Especially, this eases the transition to model-based development by generating a model representation of the function hierarchy that is kept consistent automatically [9]. We explain this model transformation aspect in detail in Sect. VI.

IV. Combining Model-based and CNL-based Requirements Engineering

For the purpose of combining the advantages of model- and NL-based RE, we propose a model-driven RE methodology that features the bidirectional transition between model-based requirements and CNL. We apply the approaches outlined in Sect. III for the specification of the corresponding artifacts. However, the original model-based approach only considers the SUD as black box (in the SPES Requirements Viewpoint), and the subsequent white box functional decomposition of the SUD (in the SPES Functional Viewpoint) lacks in guidance on how to detail the requirements (particularly the scenarios) w.r.t. the functional decomposition. The goal of our combined approach is to achieve an RE methodology that allows the detailing of requirements artifacts across several abstraction layers, as our CNL-based approach and the SPES Functional Viewpoint already provide by means of a function hierarchy. We hence extended the model-based approach to comply with this functional decomposition of the SUD, which allows arbitrarily many process iterations. The resulting methodology consists of the following process steps illustrated in Fig. 1.

A. Step 1: Specify Context

The model-based documentation of the SUD's context is used to define entities that the SUD interacts with as well as relevant inputs and outputs that are exchanged with them. The inputs and outputs are later on detailed in the scenarios of Step 3. In subsequent iterations each individual part of the SUD functionality is examined separately. The respective other parts of the SUD are considered as additional context entities.

B. Step 2: Specify Goals

Goal models are used to specify prescriptive system capabilities. In addition, the model-based documentation format is used to distinguish between capabilities mandated by legal regulations or desired by stakeholders, to prioritize requirements, and to support development planning. In further iterations, the goal model can be modified to reflect possible changes of stakeholder intentions or complement alternatives.

C. Step 3: Specify Scenarios

Based on the currently considered context, scenario models are used to define exemplary interaction sequences regarding the fulfillment of specific goals. During the specification of the scenarios, abstract inputs and outputs of the current context are detailed. The scenarios are decomposed to each currently considered context for all iterations.

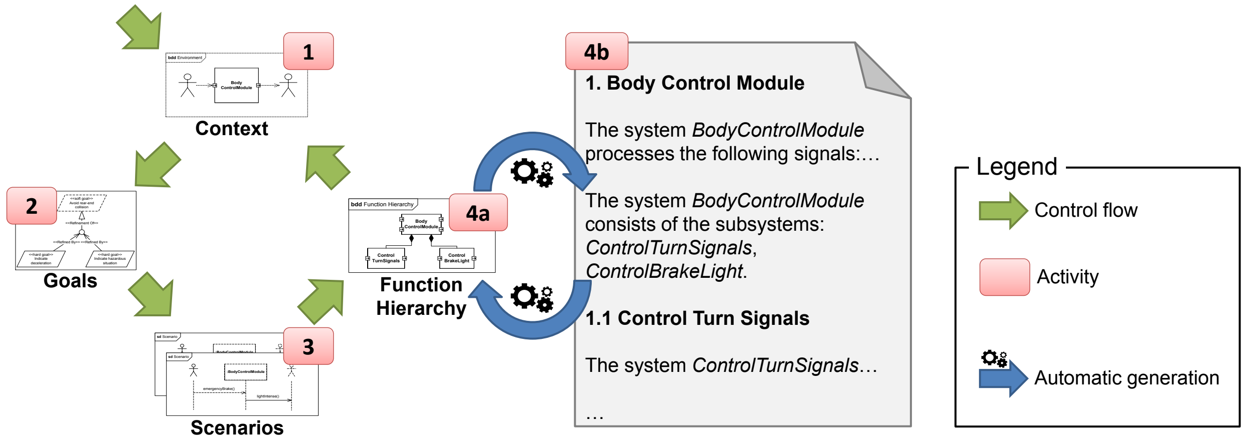
D. Step 4: Specify Function Hierarchy

Based on the context and the scenarios, functional requirements are specified that detail the abstract capabilities specified in the goals. To do so, we decompose the overall functionality of the SUD by means of a function hierarchy as sketched in Sect. III-B. These functional requirements can hence be understood as a white box specification of the functions the SUD must possess and can therefore be used to support negotiation with stakeholders (e.g., customers, system architects and developers). When a new level of subsystems is added to the function hierarchy, another iteration of the process is started for each new subsystem.

We consider the function hierarchy as an artifact that multiple stakeholders are interested in. That means, this artifact is used for reviews and negotiations at intermediate RE stages and is part of the final system requirements specification. As motivated in the introduction, a document-oriented form is needed for such reviews and legally binding documents. Therefore, as depicted in Fig. 1, we enable to transform the model-based function hierarchy (4a) into NL shaped by requirement patterns (4b) and vice versa. This is facilitated by means of the bidirectional transformation approach described in Sect. VI. The switch to the CNL representation allows a document-oriented review of the requirements by stakeholders that are not familiar with the model-based notation. Furthermore, changes to the CNL requirements can automatically be transferred into the model-based notation, fostering synchronization and traceability between both documentation formats.

V. Example Application

In this section, we illustrate our methodology by means of an example system from the automotive domain: the Body Control Module (BCM). The BCM is an electronic control unit (ECU) that centrally controls distributed vehicle functions including the turn signals, brake light, and central locking. We have conducted a comprehensive case study which shows the detailed application of the approach presented in Sect. IV [4]. In this paper, we show an excerpt to illustrate how the steps outlined in Sect. IV are conducted. We place particular



1. Body Control Module

The system *BodyControlModule* processes the following signals:…

The system *BodyControlModule* consists of the subsystems: *ControlTurnSignals*, *ControlBrakeLight*.

1.1 Control Turn Signals

The system *ControlTurnSignals*…

…

Fig. 1. Conceptual overview of the combined approach

emphasis on the transition between model-based functional requirements and their natural language representation. In the following subsections, we specifically focus on the function “emergency braking”: if the driver pushes the brake pedal strongly because of an emergency, the brake lights shall light up more intensely than normal and, in addition, the hazard lights shall flash.

A. Step 1: Specify Context

In the first step, the context of the BCM is analyzed and documented in a context model. This allows distinguishing between the functionality of the BCM and the functionality of external systems. As our methodology deals with requirements, the interactions abstract from the final realization of inputs and outputs into electrical signals or software function calls. The context of the BCM is depicted in Fig. 2. A SysML internal block diagram [22] is used to describe realization-independent functions and abstract information flows. External context functions represent functionality provided by external systems (e.g., ECUs) and are depicted as actors to emphasize that these context functions interact with the system, which in this model is considered a black box. The interactions between context functions and the BCM are modeled using input and output flow ports. In this example, the BCM interacts with the car’s dashboard to give visual feedback to a driver, for example, by toggling individual dashboard lights to indicate activated hazard lights or turn signals. Thus, the BCM interacts with the function of controlling the dashboard DashboardControlling. By means of the function BrakePedalLevelSensing, the BCM is informed about the push of the brake pedal by the driver. The BCM activates the left and right turn signal lamps using the functions LeftIndication and RightIndication. The brake lights are activated by means of the function BrakeLightSwitching. The information that is exchanged between the SUD and the context functions is only defined abstractly (cf. the italic port names in Fig. 2) in this first step of the process. It will be detailed in following steps and iterations of the process.

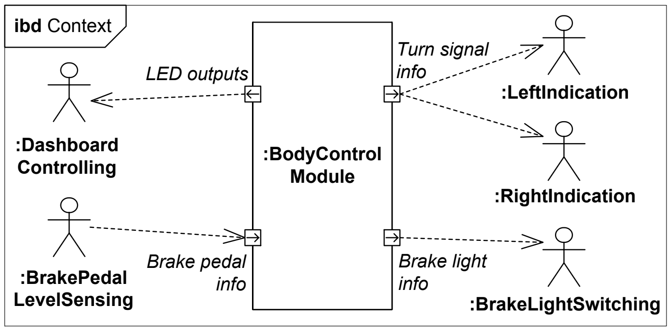


Fig. 2. Context of the Body Control Module

B. Step 2: Specify Goals

In the next step, the goals for the BCM are derived based on the context analysis from Step 1. The goals prescribe intended capabilities of the system that must integrate into the context as defined in the previous step. This means that goals must document the needed functionality to transform the defined inputs into the defined outputs of the context model. For example, when an external system is providing information about strong brake pedal actuation, the BCM must ensure that the hazard lights flash and, correspondingly, must indicate this to the driver by interacting with the dashboard control function. In Fig. 3 an example of a KAOS goal diagram [27], which specifies the goals of the BCM, is shown. Goal diagrams are used to refine top level goals (e.g., business objectives) into more detailed system capabilities. For example, one goal for the BCM is to avoid rear-end collisions. This goal is refined into the subgoals “Indicate deceleration” and “Indicate hazardous situation”. The intention of these goals is to document the need for the BCM to provide the capability of notifying the following traffic when the vehicle brakes abruptly.

Goals are distinguished into *soft goals* and *hard goals*. Soft goals, are goals whose fulfillment depends on some degree of interpretation (e.g., goals pertaining to the quality aspect “usability”). The fulfillment of hard goals can be verified unambiguously. The vague goal “avoid rear-end collision” is a

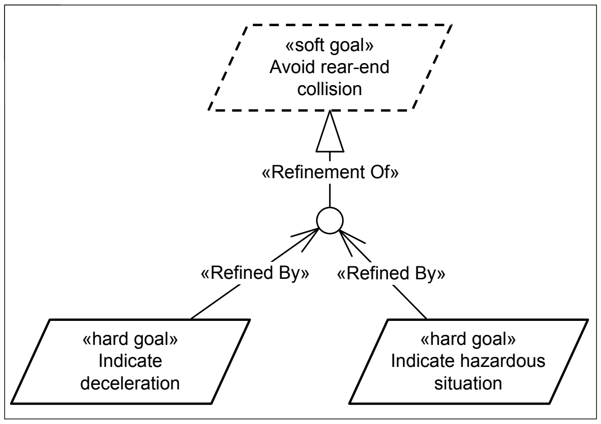


Fig. 3. Goals of the Body Control Module

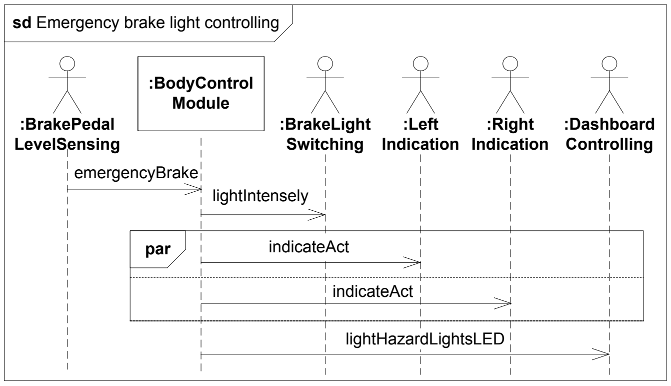


Fig. 4. Scenario “Emergency brake light controlling”

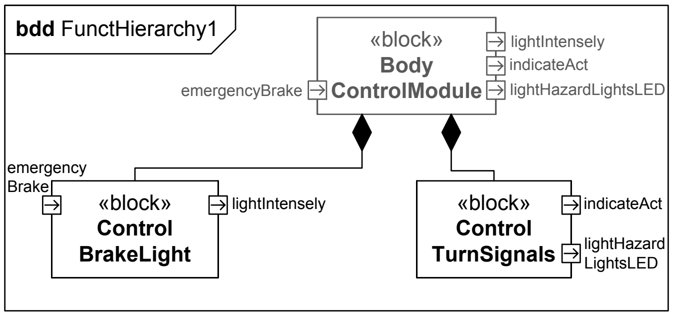


Fig. 5. Initial function hierarchy with inputs and outputs from the scenarios (faded out parts) and negotiated next abstraction layer (strong parts)

soft goal, as its fulfillment depends on complex elements like the current traffic situation and the driver’s behavior. On the contrary, the two refining hard goals can be verified: either the brake lights are active or not and similarly either the hazard lights flash or not.

C. Step 3: Specify Scenarios

In the third step, scenarios are defined that exemplify adequate goal fulfillment. In addition, scenarios must be consistent to the context models such that each context function identified in Step 1 is represented in at least one scenario, as otherwise the interaction between the system and that particular context function remains unspecified. Alternatively, definition of scenarios may lead to the identification of additional goals and additional context functions. Fig. 4 shows a scenario fulfilling the goal “Indicate hazardous situation” from Fig. 3 as a SysML sequence diagram. When the BCM is informed by the context function BrakePedalLevelSensing that an emergency braking occurred (i. e., when strong brake pedal actuation is recognized), it signals the context function BrakeLightSwitching to light the brake lights at the rear of the car intensely. Furthermore, the BCM causes the left and right turn signals as well as the corresponding LEDs in the dashboard to flash, thereby realizing the goal. This is done by sending the messages indicateAct and lightHazardLightsLED, respectively.

D. Step 4: Specify Function Hierarchy

In the fourth step, a function hierarchy for the BCM is specified based on the capabilities defined in the goal models. As described in Sect. III-B, the function hierarchy decomposes the SUD’s overall functionality into systems that group parts of the overall functionality. During decomposition, the input and output ports are partitioned onto subsystems, possibly necessary ports for the internal communication between the SUD’s subsystems are defined, and all capabilities documented in the goals (see Fig. 3) are distributed among the subsystems.

The faded out part of Fig. 5 shows the initial function hierarchy of the BCM as a SysML block definition diagram. The function hierarchy aggregates all previously considered context models, in this case, the complete system BodyControlModule from the initial context analysis (cf. Fig. 2). The inputs and outputs in the function hierarchy, that is, the input and output ports of BodyControlModule, are derived from the scenarios (cf. Fig. 4). Since the function hierarchy is a different view on the SUD subsystems of the context models, the abstract input and output ports from the context analysis are replaced by the detailed inputs and outputs of the scenarios.

1) Transition from Requirements Models to Controlled Natural Language: The model-based function hierarchy as depicted in the previous paragraph can be used to generate a set of requirements in CNL by means of requirement patterns. This way, the functional requirements that have been systematically defined throughout RE can be subject to review and change during contract negotiations. Using requirement patterns to restrict the number of permissible formulations, changes that are made during reviews and negotiations can be transferred back again to the model-based representation of the function hierarchy. For example, the faded out part of the function hierarchy shown in Fig. 5 can be represented using the Requirement Patterns from Sect. III-B as shown in Table I.

On basis of these CNL requirements the further development of the BCM is negotiated among the stakeholders. In this example, the result of such a negotiation is the next level of decomposition of the function hierarchy: two new subsystems are defined and the inputs and outputs of the SUD are distributed among them. As the basis for the negotiation is

TABLE I
CNL REQUIREMENTS GENERATED FROM THE INITIAL FUNCTION HIERARCHY

ID	Requirement text
R01	The system *BodyControlModule* processes the following signal: *emergencyBrake*.
R02	The system *BodyControlModule* creates the following signals: *lightIntensely*, *indicateAct*, *lightHazardLightsLED*.

TABLE II
MANUALLY ADDED CNL REQUIREMENTS

ID	Requirement text
R03	The system *BodyControlModule* consists of the following subsystems: *ControlBrakeLight*, *ControlTurnSignals*.
R04	The system *ControlBrakeLight* processes the following signal: *emergencyBrake*.
R05	The system *ControlBrakeLight* creates the following signal: *lightIntensely*.
R06	The system *ControlTurnSignals* creates the following signals: *indicateAct*, *lightHazardLightsLED*.

CNL, the additional requirements are also specified in CNL. Therefore, the requirement pattern statements listed in Table II are added.

2) Transition from CNL to Requirements Models: Afterwards, this information can be transferred back again to the model-based representation of the function hierarchy. This detailed function hierarchy is depicted by the strong black parts of Fig. 5.

E. Detailing Iteration

Since a new subsystem level was added to the function hierarchy, a new process iteration is necessary. The detailing of the particular artifacts is described in the following.

1) Detail Context: Each subsystem has to be considered in a particular context model. Fig. 6 shows these two context models for the subsystems ControlBrakeLight and ControlTurnSignals. In each context model the other subsystem is depicted as a SysML actor. This is because the system ControlTurnSignals is considered as part of the context of ControlBrakeLight and vice versa.

2) Detail Goals: As the next step the goal model is reconsidered and possibly revised. In this example no changes are made to the goals.

3) Detail Scenarios: Next, the scenario depicted in Fig. 4 is detailed to represent the interaction sequences with the new subsystems ControlBrakeLight and ControlTurnSignals that were added to the function hierarchy in Step 4 of the last process iteration (cf. Sect. V-D). Fig. 7 shows the revised scenario split up for each of the two subsystems. Again, the respective other subsystem is depicted as an actor. Instead of the complete system BodyControlModule being in full charge of requesting the context functions based on the notification about the occurrence of an emergencyBrake (cf. Fig. 4), in the revised scenarios this functionality is partitioned onto the subsystems ControlBrakeLight and ControlTurnSignals. Note

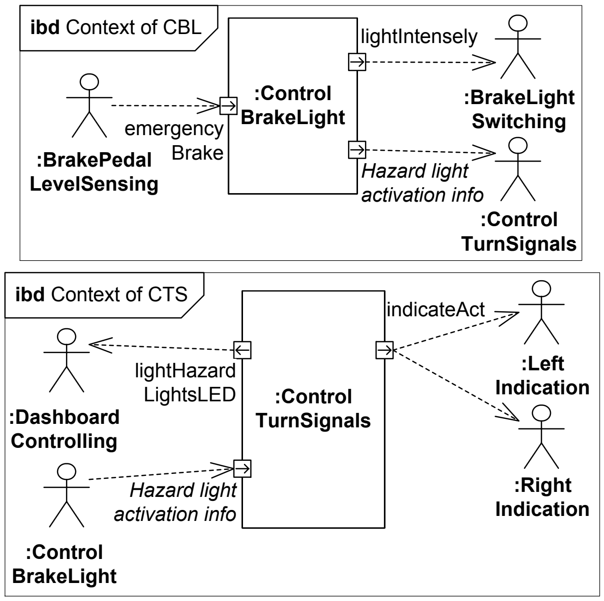


Fig. 6. Contexts of the subsystems

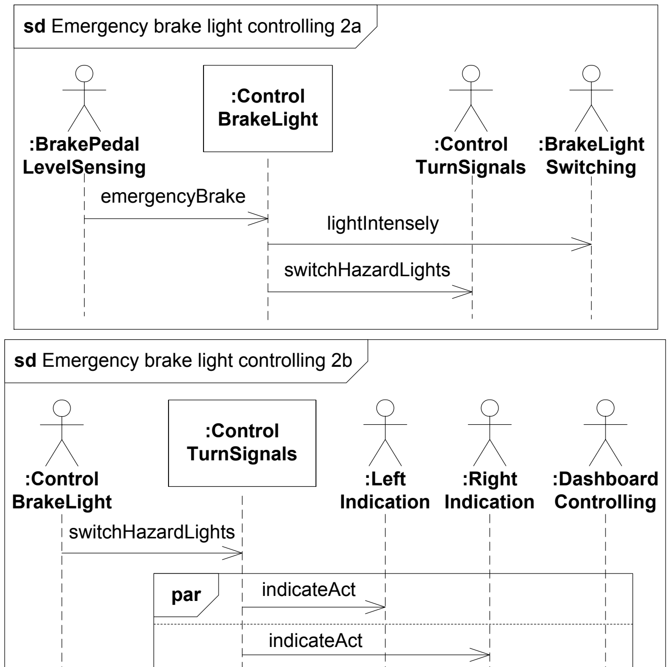


Fig. 7. Detailed scenarios for emergency braking

that this consideration of each particular subsystem in a dedicated scenario is not possible with the part decomposition concept of UML2 Sequence Diagrams [21].

After the notification about an emergencyBrake, the system ControlBrakeLight sends the request to light up the brake lights intensely to the context function BrakeLightSwitching. Additionally, the hazard lights shall be activated. This makes

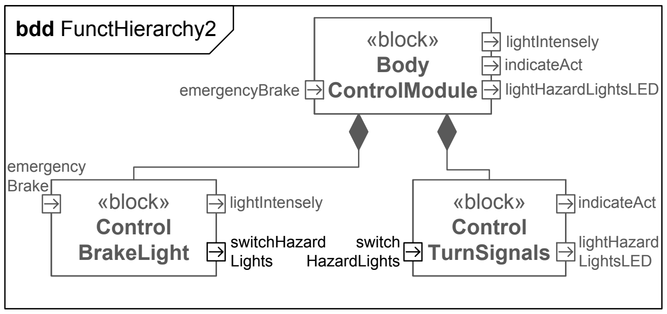


Fig. 8. Detailed function hierarchy

necessary to add the new signal switchHazardLights, which requests the subsystem ControlTurnSignals to activate the hazard lights. In the other scenario, triggered by that new signal, ControlTurnSignals sends a signal to start flashing the turn signal lamps in parallel and another signal to activate the hazard lights LED on the dashboard to the corresponding context functions. This shows how a detailed scenario also introduces new internal communication between subsystems.

After the scenarios have been detailed, the function hierarchy has to be updated accordingly. In this case, the two ports switchHazardLights in Fig. 8 had to be added.

4) Detail Function Hierarchy: Afterwards, the function hierarchy can again be decomposed further, either in its model-based or its NL-based representation. Such a decomposition would then cause further iterations of the RE process until the subsystems are sufficiently trivial to implement. Such further detailing iterations are shown in [4]. If we assume that the RE process is finished in this stage, we now would use the NL representation for a final review and as contractual basis to hand over to the customer. The model-based notation would be given to the system architect as a basis to develop the logical architecture of the BCM. The leaf functions of the function hierarchy would be allocated to those parts of the logical architecture that realize their functionality to foster traceability from the requirements to the implementation [9].

VI. Transformation between Function Hierarchy Model and CNL

In [9], we presented the model transformation between the CNL-based requirements and the function hierarchy model only in an abstract way as part of a whole development process. In this section, we focus on this bidirectional transformation and explain in detail the relations between both documentation formats throughout intermediate models.

Fig. 9 shows an excerpt of the used model elements in abstract syntax. It depicts part of the example from the previous section. The faded out part of Fig. 9 concerns the transformation from the textual representation of requirement R03 from Table II to its model-based representation. This leads to the next abstraction layer of the function hierarchy (the strong part of Fig. 5 except the ports). The strong part of Fig. 9 concerns the transformation in the opposite direction after the ports switchHazardLights were added to the function hierarchy

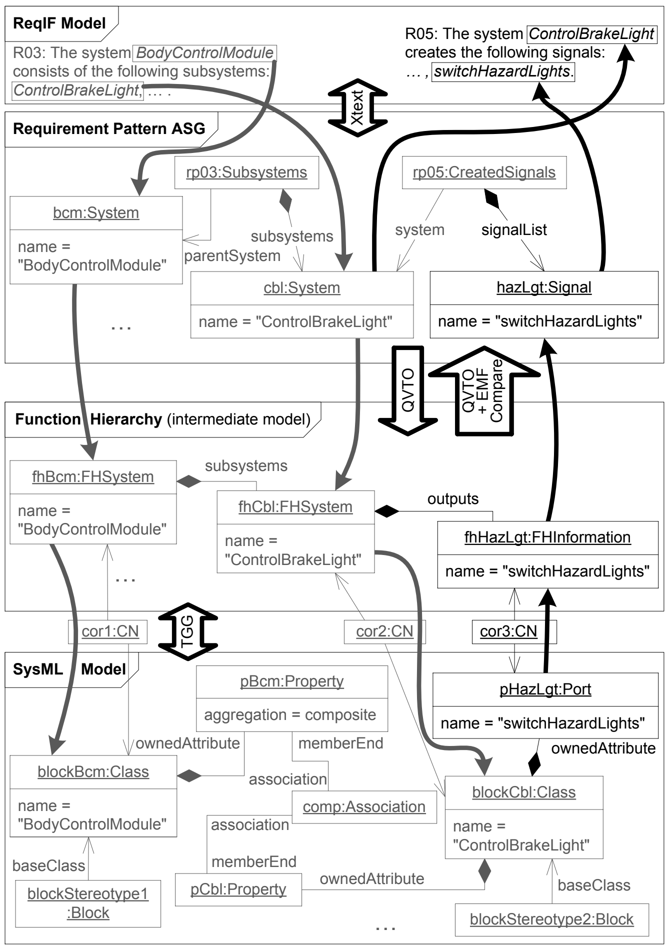


Fig. 9. Transformation from textual requirements to SysML model and back

(strong part of Fig. 8). This leads to the addition of the signal *switchHazardLights* to requirement R05 from Table II and a new requirement "The system *ControlTurnSignals* processes the following signal: *switchHazardLights*." (not shown in Fig. 9). The requirement IDs are propagated through the model elements to ensure traceability but omitted from the figure.

The textual representation of the requirements is edited in Eclipse RMF/ProR[1]. ProR is a tabular text editor for the requirements, and the underlying Requirements Modeling Framework (RMF) stores them in a ReqIF [20] model (cf. ReqIF Model in Fig. 9).

The requirements formulated by requirement patterns are extracted from that model and transformed transparently into an abstract syntax graph (cf. Requirement Pattern ASG in Fig. 9). This transformation uses the text modeling framework Xtext[2]. It allows to define the textual grammar of requirement patterns as well as a metamodel by means of the Eclipe Modeling Framework (EMF)[3] describing their ASG representation. Based on this information, the transformation (in either direction) is performed automatically. Each requirement

[1]http://www.eclipse.org/rmf
[2]http://www.eclipse.org/Xtext
[3]http://www.eclipse.org/emf

pattern is represented by its own metamodel element (e.g., Subsystems) that references its variable parts (i.e., System and Signal). The requirement pattern ASG therefore is a directed acyclic graph built up of requirement pattern instances referencing their variable parts.

The requirement pattern used in the requirement R03 is represented by the class Subsystems. Thus, the requirement R03 is represented by the object rp03 that is an instance of that class. It references the parent system bcm representing the *BodyControlModule* and its subsystem cbl representing the system *ControlBrakeLight*. The requirement R03 in Table II also lists the subsystem *ControlTurnSignals*, which is omitted from Fig. 9 to keep it simple (indicated by "...").

The structure of the Requirement Pattern ASG and the SysML function hierarchy (cf. SysML Model in Fig. 9) differ to a great degree. Therefore, we perform a transformation into an intermediate model (cf. Function Hierarchy in Fig. 9) that contains only the dense information about the function hierarchy. It intentionally contains no objects representing sentences or complex SysML constructs. For instance, in the ASG, systems do not reference their subsystems directly. They are indirectly connected via the object representing the requirement pattern instance (e.g., rp03:Subsystems). The intermediate function hierarchy model is generated from the requirement pattern ASG using the model transformation technique QVT Operational (QVTO, [19]) by means of the Eclipse QVTO implementation[4].

The System objects bcm and cbl are transformed into instances of FHSystem in the intermediate model. Their subsystem-relation, which is indirectly encoded in the references from the requirement pattern instance rp03, is transformed into a direct composition association subsystems.

In the last step, the intermediate model is transformed into the final SysML Model. For this purpose, we apply the bidirectional model transformation technique Triple Graph Grammars (TGGs) [25], which feature the preservation of the consistency between initially transformed models. This enables to easily propagate changes between the intermediate model and the SysML model. As concrete implementation, we use the so-called TGG-interpreter[5].

The two FHSystems fhBcm and fhCbl are transformed into the UML classes blockBcm and blockCbl with an applied SysML block stereotype. The subsystem-relation subsystems is transformed into a UML composition comprised of the association comp with two property-ends pBcm and pCbl that are part of the two classes.

Once the SysML model has initially been generated, TGGs allow keeping the two models consistent automatically. So-called correspondence nodes (CN objects in Fig. 9) save the mappings between elements of the intermediate model and the SysML model. They are used to detect changes in one model that have to be transferred to the other. When changes are made to the textual requirements, all models except the SysML model are newly generated from the text. The TGG-interpreter detects the changes in the intermediate model (via the correspondence nodes) and updates the SysML model accordingly.

We use Eclipse Papyrus[6] to edit the SysML model. When the port pHazLgt for the outgoing information *switchHazardLights* is added to the block blockCbl of the SysML model, the intermediate model is automatically updated by the TGG-interpreter. The new port initially is not connected to any correspondence node and therefore has to be added to the intermediate model as the object fhHazLgt connected to the system fhCbl. The updated intermediate model is used to generate a new requirement pattern ASG via QVTO. This newly generated ASG is then compared to the existing ASG via EMF Compare[7] to identify the changes and update only those textual requirements that are affected.

The object rp05 originates from the requirement pattern instance of requirement R05 from Table II to specify *ControlBrakeLight*'s output *lightIntensely* (not shown in Fig. 9). So this object is not new. The object hazLgt represents the output signal *switchHazardLights* and therefore is identified as a new ASG element by EMF Compare. The textual requirement R05 is thus replaced by a new version including the added signal (via Xtext serialization).

VII. Performance Evaluation

To evaluate the applicability of the transformation approach described in Sect. VI, we measured the execution time of transformation runs in different use cases. In this section, we describe the test setup and the results.

Our tool chain is completely integrated in the Eclipse environment and consists of the tools mentioned in the previous section. We tested the transformation tool chain with function hierarchies of different sizes (number of contained systems) and measured the execution time for each transformation step. In this paper we only show the results of the combined execution times due to space limitations.

For the test runs, we generated random function hierarchies without inputs and outputs. The randomly chosen variables are the number of complete systems (root nodes), the depth of the hierarchy, and the number of subsystems per system. Depending on the transformation direction under test the function hierarchies were either generated as textual requirement pattern instances in RMF or as SysML model in Papyrus.

We tested four use cases: (1) Generating a new ReqIF model (containing the textual requirements formulated by requirement patterns) from a SysML model (Generate SysML $\rightarrow$ ReqIF), (2) generating a new SysML model from a ReqIF model (Generate ReqIF $\rightarrow$ SysML), (3) updating a ReqIF model after deletion of a system from the SysML model (Update SysML $\rightarrow$ ReqIF), and (4) updating a SysML model after addition of a new system to the textual requirements in the ReqIF model (Update ReqIF $\rightarrow$ SysML). We did not

[4]http://www.eclipse.org/mmt/?project=qvto

[5]http://www.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter

[6]http://www.eclipse.org/papyrus

[7]http://www.eclipse.org/emf/compare

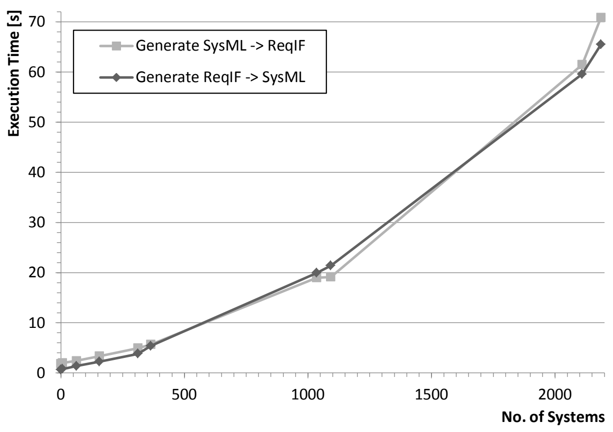


Fig. 10. Comparison of execution times for initial generation

include the generation of SysML diagrams showing the model elements nor any automated layouting algorithms. We used a typical office computer[8] for all test runs.

Fig. 10 shows the results of the two use cases (1) and (2) generating new artifacts. Function hierarchies with less than 500 systems take less than ten seconds to transform in either direction. Function hierarchies with about 2,200 systems take approx. 70 seconds to transform. Requirements specifications in the embedded systems domain can contain 10,000 requirements (e.g., [16]), but these also encompass very detailed, non-functional, and regulatory requirements. Therefore, we argue that the pure function hierarchy part describing only the SUD's functional decomposition would be unusually complex if it contains 2,200 systems. Thus, these performance results indicate the applicability of our approach with a realistic amount of requirements.

From the performance perspective it makes no difference whether one starts with textual requirements or the model based representation. The initial generation of the respectively other representation takes about the same time (generating the SysML model is approx. one second faster). The execution time of both directions increases approx. quadratic in the number of systems.

Fig. 11 shows the results of the four use cases for two exemplary function hierarchies. One function hierarchy that consists only of one system (the complete system) and one that consists of 364 systems (1 complete system, depth 6, 3 subsystems per system).

Deleting a system from the SysML model and updating the textual representation takes less time than the initial generation in the same direction. This fits our expectation, as the TGG correspondences that were set up during the initial generation just have to be checked for dangling links that indicate elements that have to be removed.

Adding a system to the textual representation and updating the SysML model takes approx. twice as long as the initial

[8]Intel Core 2 Duo P8600 2.40 GHz, 8 GB DDR3 1066 MHz, 128 GB SSD, Windows 8 Pro 64 bit, JDK 7u24, Eclipse 4.3.0 (-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=512m), RMF 0.8.0, Xtext 2.4.3, QVTO plug-in 3.3.0, TGG-Interpreter 0.5.0, Papyrus 0.10.0

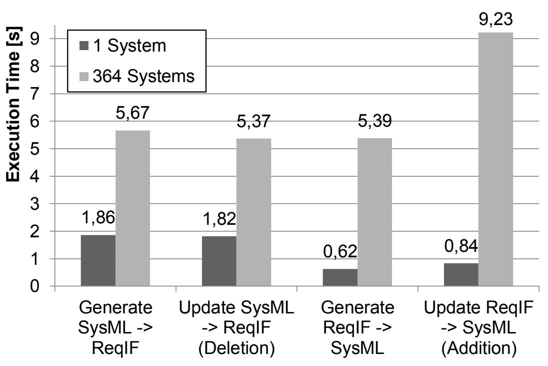


Fig. 11. Comparison of execution times of the different use cases

generation in the same direction. This not what we expected. As in the deletion use case, it should take less time because the TGG correspondences were saved previously. The models just have to be checked for systems that are not connected to any correspondence node and thus have to be added on the other side. This issue might be caused by a performance leak in the TGG-Interpreter or inappropriately modeled TGG rules. We will investigate this issue in the near future.

In conclusion, the performance results indicate that the approach is applicable to realistic sizes of requirement specifications describing function hierarchies.

VIII. Conclusions and Outlook

In this paper, we presented a RE approach combining requirements models and a CNL as documentation formats. We use requirements models in a model-driven manner for the elicitation, documentation, and negotiation of requirements, i.e., for the SUD's context, goals, scenarios, and particularly a function hierarchy. The function hierarchy can be transformed into a CNL representation that is used for the final requirements specification as well as intermediate reviews. Requirement changes from such reviews are automatically synchronized with the model-based representation. On top of both documentation formats, we conceived a systematic, model-driven RE methodology supporting the specification and detailing of the requirements artifacts across several abstraction layers. The methodology is based on two previously conceived, independent RE methods (i.e., a model-based and a CNL-based approach). We extended the model-based approach to comply with the functional decomposition of the SUD proposed by the CNL-based approach. This extension allows arbitrarily many iterations across several abstraction layers of the SUD's context, scenarios, and the function hierarchy. To enable the transition between both documentation formats, we apply a bidirectional, multi-step model transformation that makes use of several technologies.

By combining models and natural language to document requirements, it is possible to harness the advantages of both documentation formats: On the one hand, the usage of requirements models for requirements elicitation, documentation, and negotiation satisfies the desire of requirements engineers to use models during the RE process [26] and enables all advantages of model-based RE like facilitating requirements understanding [18] and fostering automatic analysis techniques. On the other hand, the usage of NL enables—particularly in the domain of embedded systems—the inclusion into legally binding documents [26] and satisfies the need of document-oriented requirements specifications for stakeholder reviews [10]. We restrict the expressiveness of NL by using a CNL, which allows an easy automatic processing enabling the automatic switching between model-based and NL-based representation. The underlying model transformation automatically enables traceability and ensures consistency between both documentation formats. The performance evaluation yields that the model transformation is applicable in realistic scenarios. The systematic RE methodology provides guidance in specifying the particular artifacts across several abstraction layers and in prescribing the adequate phases in which switches between the documentation formats should occur. The application of the methodology has been illustrated by means of excerpts from an example from the automotive industry [4].

Up to now, we applied the methodology to an automotive example. In the future, we plan to evaluate the approach more extensively w.r.t. its acceptance in industrial settings. Furthermore, we want to investigate the applicability of the approach in similar embedded systems sectors like avionics, in which similar reasons as in the automotive sector impede the usage of requirements models [26]. Furthermore, we want to improve the tool chain to reduce its complexity and make the requirements traceability more explicit to the user. Last, a possible extension of the approach could be the use of formal models for scenarios across several abstraction levels like [14] in order to enable automatic analysis techniques like simulative validation and formal verification for consistency.

Acknowledgments

This research is funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL) and is managed by the Project Management Agency Karlsruhe (PTKA). We thank Marian Daun and Bastian Tenbergen for sharing their expertise on the model-based RE approach and for ongoing discussions about this paper.

References

[1] V. Ambriola and V. Gervasi. On the systematic analysis of natural language requirements with CIRCE. *Automated Software Engineering*, 13(1):107–167, 2006.

[2] M. Broy, W. Damm, S. Henkler, K. Pohl, A. Vogelsang, and T. Weyer. Introduction to the SPES modeling framework. In *Model-Based Engineering of Embedded Systems*, chapter 3, pages 31–49. Springer, 2012.

[3] B. Cheng and J. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, FOSE '07. IEEE, 2007.

[4] M. Daun, M. Fockel, J. Holtmann, and B. Tenbergen. Goal-scenario-oriented requirements engineering for functional decomposition with bidirectional transformation to controlled natural language: Case study body control module. Technical Report ICB-Research Report No. 55, University of Duisburg-Essen, 2013.

[5] M. Daun, B. Tenbergen, and T. Weyer. Requirements viewpoint. In *Model-Based Engineering of Embedded Systems*, chapter 4, pages 51–68. Springer, 2012.

[6] D. Deeptimahanti and M. Babar. An automated tool for generating UML models from natural language requirements. In *Proc. of the IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '09)*. IEEE, 2009.

[7] D. Drusinsky. From UML activity diagrams to specification requirements. In *IEEE Int. Conf. on System of Systems Engineering 2008 (SoSE '08)*, pages 1–5. IEEE, 2008.

[8] M. Fockel, P. Heidl, J. Holtmann, W. Horn, J. Höfflinger, H. Hönninger, J. Meyer, M. Meyer, and J. Schäuffele. Application and evaluation in the automotive domain. In *Model-Based Engineering of Embedded Systems*, chapter 12, pages 157–175. Springer, 2012.

[9] M. Fockel, J. Holtmann, and J. Meyer. Semi-automatic establishment and maintenance of valid traceability in automotive development processes. In *2nd International Workshop on Software Engineering for Embedded Systems (SEES)*. IEEE, 2012.

[10] R. Goldsmith. *Discovering Real Business Requirements for Software Project Success*. Artech House, Boston, 2004.

[11] H. Harmain and R. Gaizauskas. CM-Builder: A natural language-based CASE tool for object-oriented analysis. *Automated Software Engineering*, 10(2), 2003.

[12] J. Holtmann, J. Meyer, and M. Meyer. A seamless model-based development process for automotive systems. In *Software Engineering 2011*, volume P-184 of *LNI*, pages 79–88. Bonner Köllen Verlag, 2011.

[13] J. Holtmann, J. Meyer, and M. von Detten. Automatic validation and correction of formalized, textual requirements. In *Proc. IEEE 4^{th} Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011.

[14] J. Holtmann and M. Meyer. Play-out for hierarchical component architectures. In *Proc. 11^{th} Workshop Automotive Software Engineering*, volume P-220 of *Lecture Notes in Informatics*, pages 2458–2472, 2013.

[15] M. Ilieva and O. Ormandjieva. Models derived from automatically analyzed textual user requirements. In *4^{th} Int. Conf. on Software Engineering Research, Management and Applications*. IEEE, 2006.

[16] J. Leuser and D. Ott. Tackling semi-automatic trace recovery for large specifications. In *Requirements Engineering: Foundation for Software Quality*, volume 6182 of *LNCS*, pages 203–217. Springer, 2010.

[17] F. Meziane, N. Athanasakis, and S. Ananiadou. Generating natural language specifications from UML class diagrams. *Requirements Engineering*, 13(1):1–18, 2008.

[18] J. Nicolás and A. Toval. On the generation of requirements specifications from software engineering models: A systematic literature review. *Information and Software Technology*, 51(9):1291–1307, 2009.

[19] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification: Version 1.1, OMG document number: formal/2011-01-01, 2011.

[20] Object Management Group. OMG Requirements Interchange Format (ReqIF): Version 1.0.1, OMG document number: formal/2011-04-02, 2011.

[21] Object Management Group. OMG Unified Modeling Language (OMG UML) superstructure: Version 2.4.1, OMG document number: formal/2011-08-06, 2011.

[22] Object Management Group. OMG Systems Modeling Language (OMG SysML): Version 1.3, OMG document number: formal/2012-06-01, 2012.

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

[24] D. Ross and K. Schoman. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, 1977.

[25] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*. Springer, 1995.

[26] E. Sikora, B. Tenbergen, and K. Pohl. Industry needs and research directions in requirements engineering for embedded systems. *Requirements Engineering*, 17(1), 2012.

[27] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models and Software Specifications*. John Wiley & Sons, 2009.

[28] A. Vogelsang, S. Eder, M. Feilkas, and D. Ratiu. Functional viewpoint. In *Model-Based Engineering of Embedded Systems*, chapter 5, pages 69–83. Springer, 2012.

[29] Wyner et al. On controlled natural languages: Properties and prospects. In *Controlled Natural Language*, volume 5972 of *LNCS*. Springer, 2010.

[30] T. Yue, L. Briand, and Y. Labiche. A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, 16(2):75–99, 2011.