2011 Fourth International Conference on Software Testing, Verification and Validation Workshops

Automatic Validation and Correction of Formalized, Textual Requirements*

Jörg Holtmann**, Jan Meyer**, Markus von Detten
**s-lab – Software Quality Lab, Software Engineering Group,
Heinz Nixdorf Institute, University of Paderborn
Warburger Str. 100, 33098 Paderborn, Germany
[jholtmann|jmeyer]@s-lab.upb.de, mvdetten@upb.de

Abstract

Nowadays requirements are mostly specified in unrestricted natural language so that each stakeholder understands them. To ensure high quality and to avoid misunderstandings, the requirements have to be validated. Because of the ambiguity of natural language and the resulting absence of an automatic mechanism, this has to be done manually. Such manual validation techniques are time-consuming, error-prone, and repetitive because hundreds or thousands of requirements must be checked. With an automatic validation the requirements engineering process can be faster and can produce requirements of higher quality. To realize an automatism, we propose a controlled natural language (CNL) for the documentation of requirements. On basis of the CNL, a concept for an automatic requirements validation is developed for the identification of inconsistencies and incomplete requirements. Additionally, automated correction operations for such defective requirements are presented. The approach improves the quality of the requirements and therefore the quality of the whole development process.

1 Introduction

Today's technical products are characterized by a high number of functions which are developed out of new innovations or by combination of existing functionalities. Because of the high product complexity, the development of sub components are often delegated to suppliers. One domain in which this is done in an intense way is the automotive domain. In this industrial sector the automotive original equipment manufacturer (OEM) only produces 35% or less of the car himself [4]. The other parts are developed by suppliers, and the OEM integrates the produced parts to a car.

According to the process reference model Automotive SPICE [3], the OEM forwards *customer requirements* to the supplier, which designate the high-level functionality of the system to be developed. In response to that, the supplier—having the necessary technical background—analyses them and specifies *system requirements*, which propose a possible solution of the required system functionality. Among other things, the overall system requirement specifications have to be readable, complete, and consistent [9, 12] in order to serve as contractual basis and as basis for the further development process.

Multiple stakeholders from different departments with different educational backgrounds are involved in the requirements engineering (RE) and the further development process. Since natural language for the formulation of requirements is easily applicable because it does not require training or dedicated tools [12], system requirements are formulated textually in order to achieve that all stakeholders understand them.

The problem of unrestricted natural language lies in its informal character and ambiguity. Thus, requirements formulated in this way cannot be automatically processed without resolving the ambiguity by hand. Thus, all further tasks like the validation of the requirements and the transition to the model-based design have to be done manually, which is time-consuming, error-prone, and often repetitive. This particularly applies to manual validation methods for requirement specifications like reviews.

To overcome this problem, we propose a controlled natural language (CNL) approach for the specification of functional system requirements in the automotive domain. Requirements specified with this CNL describe functional requirements organized in a function hierarchy including required inputs and provided outputs for the functions. The CNL restricts the expressiveness of natural language and disambiguates it, enabling automatic processing of the re-

*This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES2020, 01IS08045H".

quirements while having textual requirements understandable for all stakeholders at the same time. This allows for checking automatically whether the formalized requirements violate certain rules, which consist of logical constraints and guidelines for high-quality specifications. Additionally, we present correction operations for defective requirements, which also support the requirements engineer in the correction process in an automated way. The automatic identification of defective requirements is what we call automatic requirements validation—in contrast to approaches that obtain executable models for simulation purposes, for example.

As an ongoing example we use a scenario from the automotive domain. However, since the division into high-level customer requirements specified by a principal and more detailed system requirements specified by an agent applies also to other domains, the general concept is also applicable to them if the domain-specific information is adapted.

In the following section, the CNL for the specification of the requirements is introduced. Section 3 presents the general concept of our automatic validation of the requirements, some of the constraints to be checked, and the correction operations. In Section 4 the related work is discussed, while the last section gives a summary.

2 Requirement Patterns

We extended a CNL for the specification of functional system requirements, which is already successfully used in the automotive industry [10]. The CNL consists of textual templates for requirements (*requirement patterns*) with static, variable, alternative, and optional parts. The syntax is similar to the one of regular expressions. Text fragments that are not surrounded by brackets resemble the static parts of a pattern and must not be changed. Text fragments within angle brackets are variable parts, which have to be filled with specific elements. A special form of these variable parts are lists, that allow to concatenate elements separated by a comma. Text fragments within parentheses contain alternatives that are separated by "|", while text fragments within square brackets are considered to be optional. Some of these requirement patterns, which are relevant for this paper, are listed below.

1. The system "<system>" consists of the following subsystem[s]: <subsystem list>.

2. The functionality of the system "<system>" consists of the following function[s]: <function list>.

3. The (system "<system>" | function "<function>") (processes | creates) the following signal[s]: <signal list>.

4. When the event <event> occurs within the system "<system>" [and the condition <condition> is fulfilled], then the function "<function>" is (activated | deactivated).

By using the requirement patterns, the functionality of the system under development (SUD) is described. The patterns allow a decomposition of the overall system functionality into atomic functions across several abstraction levels. Besides the different functions, also the dependencies between them are of interest. To identify the dependencies, the input and output data required and provided by the different functions are analyzed and described by using the CNL. The approach is similar to the one of the Structured Analysis as presented in [13], for example. By following the methodology using the requirement patterns, a function hierarchy spanning a tree with functions as leafs is developed. In Figure 1 the different elements of the function hierarchy are shown.

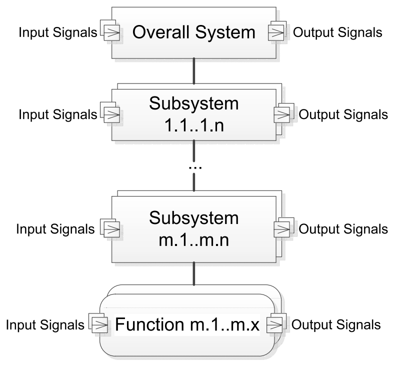

Figure 1. Structure of the function hierarchy described with requirement patterns

The element *system* in the requirement patterns is a functional unit (i.e., a carrier of functionality) and does not necessarily reflect a structural element in the subsequent process of system architecture design (e.g., a software component or a hardware device). For example, it is often the case that there is a one-to-one mapping, so one function is realized by one component. But it is also possible that a component realizes several functions or that concessions to the technical architecture have to be made (e.g., similar functions for actions in the front and rear car area are allocated to a hardware device in the front and rear area, respectively). Thus, the Overall System in Figure 1 stands for the functionality of the SUD, which is decomposed across the subsystems to atomic *functions*. These functions have a behavior by means of a relation between the input and output signals. The description of this behavior is not in scope of this paper and could be specified with free natural language, with formal models, or also with a CNL. The functions have to be mapped to the structural elements of the system (or software) architecture in the subsequent development process.

The element *signal* describes the input and output data of a function. All signals are defined in a central *data lexicon* and referenced by the requirement patterns. With a signal only logical values are specified to model the data flow between different functions (e.g., velocity). The logical values are more abstract than the real ones in the system architecture. In the system architecture the interfaces are described in more detail and are mapped to technical signals like bus signals. Hence, the input and output signals can be used to define interfaces in the system architecture.

The typical procedure while using the requirement patterns is that the requirements engineer first develops the function hierarchy in a top-down approach without considering the input/output data of the functions. Afterwards, the input/output data flows of the leaf functions are determined and propagated towards the overall system in a bottom-up manner. Eventually, the determined input/output data of the overall system can be compared to the interface description delivered by the OEM. If there are differences (e.g., required input data is not provided by the surrounding systems), these have to be resolved in cooperation with the OEM.

Furthermore, there are *events* that trigger the activation or deactivation of functions. Additionally, some *conditions* can be specified, which must hold when the event occurs. These events and conditions are described with the fourth requirement pattern. There are further templates that formalize the variables <event> and <condition> from requirement pattern no. 4. These are listed in Table 1.

Event	"<signal>" (increases above | decreases below | reaches) "<value>"
	"<signal>" is turned (on | off)
Condition	"<signal>" [is] [not] (greater than | lower than | equal to | unequal to | greater than or equal to | lower than or equal to) "<value>" [is]
	"<signal>" (< | > | == | <= | >= | <>) "<value>"

Table 1. Templates for events and conditions

It could be argued that a graphical modeling language for the description of the function hierarchy is better suited. But, as indicated in the introduction, such a dedicated formalism is difficult to apply wrt. to the comprehension by all stakeholders and the use within legally binding documents. Furthermore, the presented four requirement patterns are only an excerpt of over 100 patterns, which altogether describe more information than a simple function hierarchy. Besides these elements, the description of a system or a function comprises more information than can be expressed by using only the requirement patterns (e.g., unrestricted text, drawings, behavior description of the functions). Thus, the function hierarchy is divided into chapters, which contain this additional information.

In this paper, we focus only on the aspects that can be described by means of the requirement patterns. From these patterns, we chose the four presented ones to illustrate our concept of automated requirements validation. Therefore, a grammar for the patterns as well as a metamodel resembling the concepts presented above were specified. This requirements metamodel is depicted in Figure 2. On top of that, we developed a prototypical requirements editing environment consisting of a tabular editor for the data lexicon as well as a text editor for the requirements formulation with the requirement patterns. The text editor has features like error marking (e.g., in the case of text that does not correspond to the requirement patterns), syntax highlighting, auto completion, and an overview of the current function hierarchy. These features support the requirements engineer in a constructive manner while formulating requirements. The text is parsed into an abstract syntax graph (ASG) typed by the requirements metamodel. This ASG is the starting point for further automatic processing. Besides the possibility of an automatic identification of rule-violating requirements, the information of this ASG can also be transformed to graphical models based on, for example, the UML.

The metamodel is a different representation of the requirement patterns and is a prerequisite for the specification of the means for the identification and the correction of defective requirements as described in Section 3. The Model is the container for the top-level overallSystem as well as for all Signals. A System can contain further subSystems and functions. A System and a Function can have several input and output Signals. Finally, an Event activates or deactivates a Function if an eventCondition occurs and the additionalConditions are fulfilled. These conditions are specified by means of Value expressions wrt. to the Signals.

We present the functional analysis of an electronic comfort control unit as a running example. Such a comfort control unit has the task to offer so-called comfort functions to the vehicle passengers like interior light control and dimming, central locking, window lifting, and so on. In order to accomplish this task, it synchronizes the functions of other control units like the electronic control unit (ECU) of the central locking system and the ECU for the interior light control, for example. Such a comfort control unit is a typical example of functionality that is created by interconnecting already existing functionalities.

The system requirements listed in Table 2 are an excerpt of the functional analysis with the requirement patterns. The data lexicon containing referenced logical signals is given in Table 3. The requirements are only complete regarding the door locking functionality of the system. There

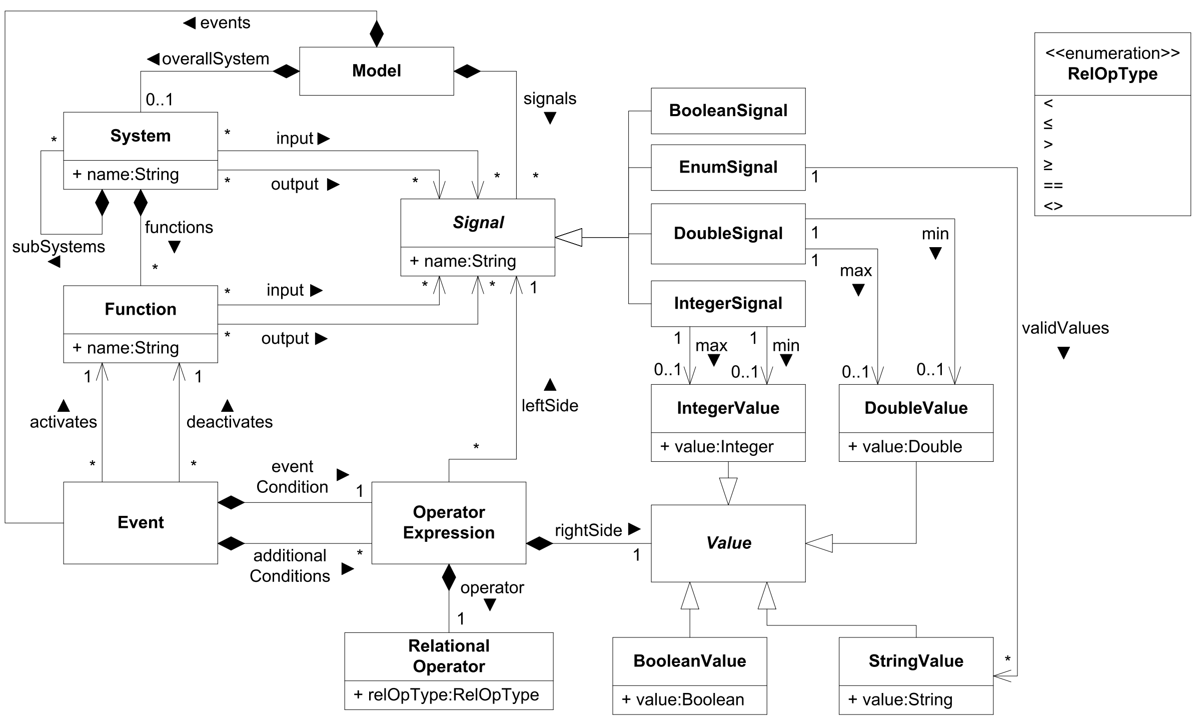

Figure 2. Metamodel of the requirements ASG

are further systems mentioned like the door unlocking, interior light control, and the window lift control. These systems are not fully specified by the requirements patterns.

Note that the requirements already contain some defects. Some of these defects can maybe be found easily within this small excerpt. But this is not possible for realistic specifications with hundreds or thousands of requirements, which are additionally distributed across several chapters, documents, or DOORS[1] modules. Other defects may not be found even in the excerpt, since they contradict guidelines that were created from experiences with the procedure for the creation of the function hierarchy. As these guidelines are only known to experts, such defects cannot be found by unexperienced writers or readers.

Figure 3 depicts an excerpt of the ASG generated from the requirements. It represents the requirements R05, R07, R09, R11, R12 partially, R13, and R14.

3 Automatic Requirements Validation

Indeed, the use of the requirement patterns and the requirements editing environment already supports the requirements engineer in a constructive manner. Thus, only formulations that conform to the requirement patterns are allowed, leading to the presented function hierarchy. Furthermore, the tool prohibits references to not existing elements, typing errors, multiple assignments of unique identifiers to different elements, and constrains the use of synonyms. But the function hierarchy as a whole can be incomplete or can contain inconsistencies. Such defects contradict the static semantics of the requirement patterns language and cannot be eliminated just by applying the presented procedure for the creation of the function hierarchy and using the editing environment. To identify such defects, we propose an automated requirements validation technique that can be executed repeatedly on the function hierarchy.

Together with our industrial partner, we developed guidelines for the application of the procedure of creating the functional hierarchy using the requirement patterns. We also analyzed requirement specifications that were created by using the procedure and the requirement patterns and determined typical defects like inconsistent, incomplete, or awkward function hierarchies that contradict the guidelines.

These studies led to rules that have to be fulfilled to achieve a high quality function hierarchy. The rules address the readability, completeness, and consistency of the function hierarchy. These are *quality criteria* [12] for software requirement specifications demanded by the IEEE Standard 830 [9] and other literature on RE (e.g., [12]).

In order to enable an automated requirements validation, we specify formal patterns that violate the rules. The overall concept is illustrated in Figure 4. We use the *structural*

[1]`http://www.ibm.com/software/awdtools/doors`

ID	Requirement text
R01	The system “Comfort Control” consists of the following subsystems: Central Locking, Interior Light Control, Window Lift Control.
R02	The system “Central Locking” processes the following signals: Request_Doors_Lock, Request_Doors_Unlock, Request_Trunk_Open.
R03	The system “Central Locking” creates the following signals: Doors_Unlocked.
R04	The system “Central Locking” consists of the following subsystems: Door Locking, Door Unlocking.
R05	The system “Door Locking” consists of the following subsystem: Safety Locking.
R06	The system “Safety Locking” creates the following signal: Doors_Locked.
R07	The functionality of the system “Door Locking” consists of the following functions: Lock Door L, Lock Door R.
R08	The function “Lock Door L” processes the following signal: Request_Doors_Lock.
R09	The function “Lock Door L” creates the following signal: Doors_Locked.
R10	The function “Lock Door R” processes the following signal: Request_Doors_Lock.
R11	The function “Lock Door R” creates the following signal: Doors_Locked.
R12	The functionality of the system “Safety Locking” consists of the following functions: Safety Lock Door L, Safety Lock Door R.
R13	The function “Safety Lock Door L” creates the following signal: Doors_Locked.
R14	When the event “Velocity increases above 250” occurs, the function Safety Lock Door L is activated.

Table 2. Example requirements specified with requirement patterns

patterns of the Reclipse Tool Suite[2] [17] for this purpose. These structural patterns are based on a graph transformation formalism and defined in terms of the requirements metamodel. The function hierarchy is represented by an abstract syntax graph (ASG), which is generated from the textual requirements and is based on the same metamodel. Instances of the structural patterns are searched within this ASG during the *structural analysis*. These pattern instances indicate the violation of the rules and therefore are candidates for poorly readable, incomplete, or inconsistent (i.e., defective) parts of the function hierarchy.

[2]http://www.fujaba.de/reclipse

Name	Type	Range
Request_Doors_Lock	Boolean	
Request_Doors_Unlock	Boolean	
Request_Trunk_Open	Boolean	
Doors_Locked	Boolean	
Doors_Unlocked	Boolean	
Velocity	Integer	[-20..230]

Table 3. The data lexicon excerpt of the comfort control unit example

In addition to the structural analysis, the concept of the structural patterns of Reclipse allows us to specify correction operations for defective requirements with the same formalism (i.e., graph transformations). Such a correction operation can be triggered by the requirements engineer and resembles the quick fix feature of modern integrated development environments like Eclipse[3]. In this way, an defective requirement representation within the ASG can be transformed automatically into a correct one. Afterwards, the ASG can be transformed into textual requirements shaped by our CNL as presented in the last section using code generation techniques.

In the next subsection we present some of the structural patterns for the detection of rule-violating requirements. Subsection 3.2 introduces the correction operations for defective requirements.

3.1 Structural patterns for rule-violating requirements

In this subsection we present some of the structural patterns for the identification of requirements that violate the rules for the function hierarchy. Due to space limitations, this is only an excerpt to illustrate the approach. Among other things, further structural patterns exist for the identification of incomplete systems, unused signals, incomplete functions, unnecessary complex interfaces, and overlapping activation and deactivation events for the same function.

Of course, it is not really possible to ensure a semantically complete specification since the intentions of a requirement engineer are neither known nor predictable. But if requirements are identified that violate the rules, it is a hint that the overall specification is incomplete in some way. For example, if functions or signals are not referenced, either references on them are missing or they are superfluous.

The presentation of the structural patterns is organized as follows. The first part relates the aspects addressed by the structural pattern to the quality criteria readability, completeness, and consistency of [9, 12]. The second part ex-

[3]http://www.eclipse.org

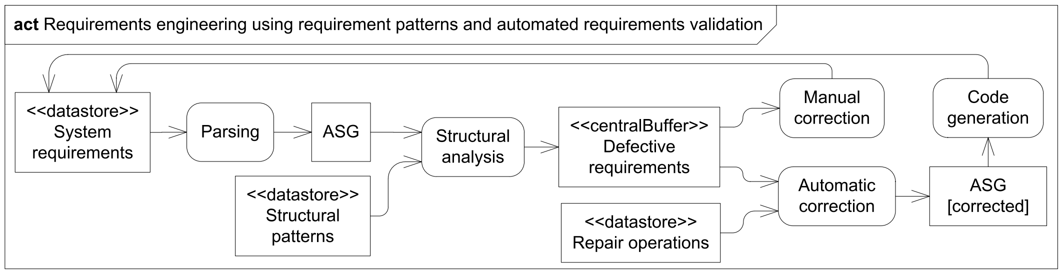
act Requirements engineering using requirement patterns and automated requirements validation
<<datastore>> System requirements
Parsing
ASG
Structural analysis
<<datastore>> Structural patterns
<<centralBuffer>> Defective requirements
Manual correction
Automatic correction
<<datastore>> Repair operations
Code generation
ASG [corrected]

Figure 4. Concept of the automated requirements validation

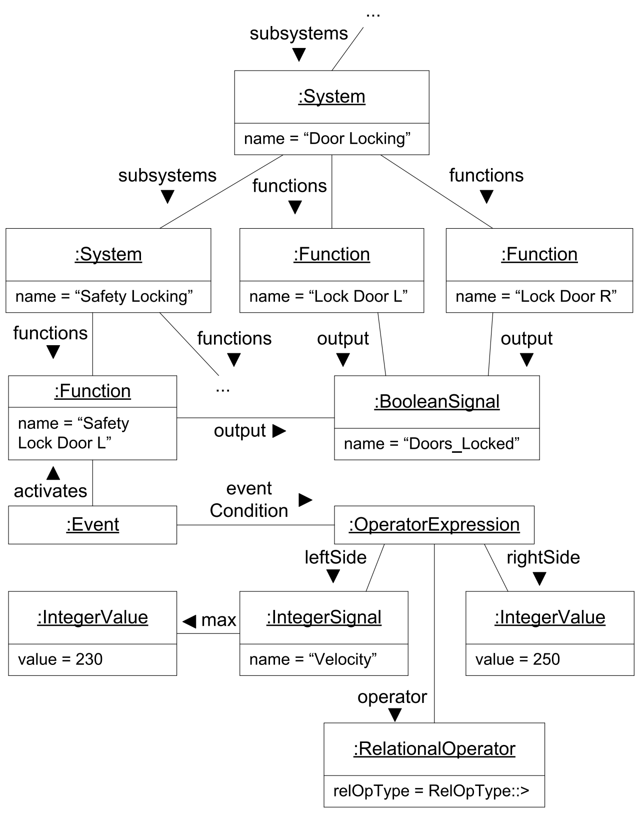

Figure 3. Example excerpt of the ASG

plains the rationale behind the corresponding rule. The structural pattern itself is described by the third part, while the last part illustrates the application of the pattern on the running example.

Structural pattern 1: System with subsystems and functions

Quality criteria Readability

Rationale One of the guidelines developed with our industrial partner, as well as several further ones, addresses the overall structure of the function hierarchy. A system shall be either completely defined by its functions, or its functionality shall be decomposed into subfunctionalities. That is, a system consists either of functions or of subsystems, exclusively.

This stems from readability reasons. As explained in Section 2, the function hierarchy is divided into chapters. If the behavior descriptions of the functions are spread across these chapters, it is difficult to find them in the overall specification. The specification is easier to read, if the functions are always contained by the leaf subsystems (cf. Figure 1).

Description Figure 5 shows the structural pattern TooComplexSystem. The black elements (i.e., *object variables* as well as *links*) specify the object structure to be detected in the ASG. Like the ASG, the structural patterns and thus the object structures are defined in terms of the requirements metamodel shown in Figure 2. If an object of a variable's type is detected in the ASG and the object's references to other objects match the links of the variable, the variable is *bound* to the object. If all variables are bound, the whole specified object structure is detected in the ASG and the structural pattern matches.

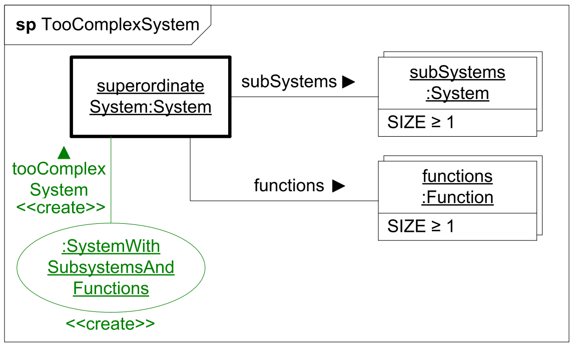

Figure 5. Structural pattern 1: System with subsystems and functions

The structural pattern specifies an object of the type System that has both a link typed by the association subSystems to at least one object subSystem of the type System and a functions-link to at least one function object of the type Function.

The double border of the object variables subSystems and functions denotes an *object set*. Such a set can be bound to an arbitrary amount of objects of the corresponding types attached to superordinateSystem via the corresponding links. The constraint SIZE $\geq$ 1 requires the sets to contain at least one object.

The bold border of the object variable superordinateSystem:System denotes a *trigger*. Such a trigger variable marks an entry point for the structural analysis. When the ASG is searched for a structural pattern, all objects of the type of the trigger object are the starting point for the matching of the pattern.

The green elliptical node and the green link with the stereotype ≪create≫ specify an *annotation* :SystemWithSubsystemsAndFunctions, which is created when the specified object structure is matched in the ASG. All object structures annotated in this way are presented to the user as candidates for defective requirements.

Example This structural pattern would identify the requirements R05 and R07, whose ASG representation is also shown in Figure 3. Functions and subsystems are mixed up for the system "Door Locking". A possible solution would be the outsourcing of the functions to a new subsystem.

Structural pattern 2: Subordinate element uses not provided signal

Quality criteria Completeness, consistency

Rationale To ensure the completeness and consistency of the system data flows and the function data flows across the function hierarchy, there are structural patterns which check locally whether a subordinate element of a system (i.e., a subsystem or a function) has input and output data that are also owned by the superordinate system. If no element is identified that violates this local rule, the rule holds globally. That is, the overall system owns the union of all input and output data flows of all functions. This is important to determine differences between the actually required input/output data and the interface description given by the OEM.

Description Figure 6 shows the structural pattern InputSignalUsedBySubsystemButNotProvidedBySystem. This pattern checks, whether an input signal used by the interface of a subsystem is consistently provided by the superordinate system. For this purpose, the objects superordinateSystem and subSystem connected by a subSystems link are searched, where subSystem has an input link to an object signal. The pattern only matches completely if superordinateSystem has no input link to signal, which is indicated by the crossed-out input link between them. If an instance of this structural pattern is found, the corresponding object structure is marked by the annotation :SystemUsesNotProvidedInputSignal.

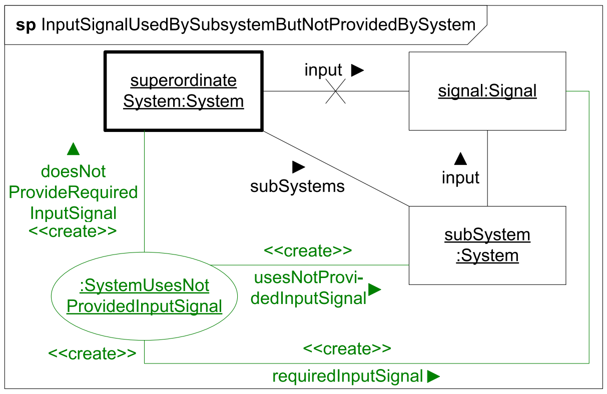
sp InputSignalUsedBySubsystemButNotProvidedBySystem
superordinate System:System
input ▶
signal:Signal
▲ doesNot ProvideRequired InputSignal <<create>>
▶ subSystems
▲ input
subSystem :System
<<create>>
usesNotProvidedInputSignal ▶
:SystemUsesNot ProvidedInputSignal
<<create>>
<<create>>
requiredInputSignal ▶

Figure 6. Structural pattern 2: Input signal used by subsystem but not provided by superordinate system

There are similar structural patterns, which consider functions as subordinate elements and output instead of input signals.

Example This structural pattern would determine that the signal "Doors_Locked" used by the functions "Lock Door L" and "Lock Door R" is not provided by the superordinate system "Door Locking". Thus, it would identify the requirements R09 and R11 as defective.

Structural pattern 3: Expression values out of range

Quality criteria Consistency

Rationale In order to formulate requirements on activation or deactivation of functions, relational operator expressions within event conditions are specified wrt. signal types. In the automotive domain numerical signals are typically constrained in the range of the allowed values. If the conditions are specified, it must be ensured that the values are within the range of the corresponding signal.

Description Figure 7 depicts the structural pattern ExpressionValuesOutOfRange for the identification of the above mentioned expressions. First of all, a trigger object opExp:OperatorExpression is searched that is attached to an object sig:IntegerSignal via a link leftSide and to an object v:IntegerValue via a link rightSide. This means that opExp evaluates the value expression v wrt. signal type sig. The allowed range for the signal sig is defined as a closed interval between the IntegerValues min and max which are connected to sig by links of the same name. Since the intervals do not have to be bound, min and max are declared as *optional* indicated by the dashed line. Even if the object for such an optional object variable does not exist, the structural pattern can still be successfully matched.

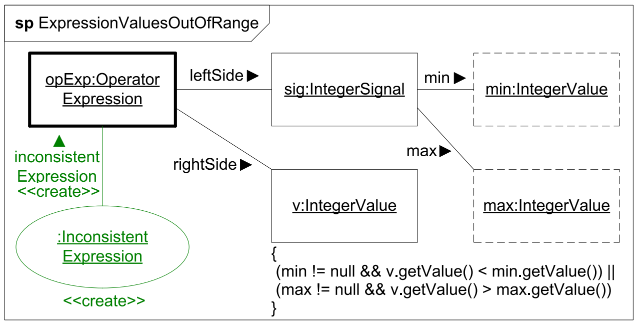
sp ExpressionValuesOutOfRange
opExp:Operator Expression
leftSide ▶
sig:IntegerSignal
min ▶
min:IntegerValue
inconsistent Expression <<create>>
rightSide ▶
max ▶
v:IntegerValue
max:IntegerValue
:Inconsistent Expression
<<create>>
{
(min != null && v.getValue() < min.getValue()) ||
(max != null && v.getValue() > max.getValue())
}

Figure 7. Structural pattern 3: Condition expression out of range

Besides these structural properties, the value assignment of v must be checked wrt. to the interval defined by min and max. This is done by the constraint within the curly brackets. The constraint can access methods and attributes of bound objects and can relate these to methods and attributes of other objects or to concrete values. The constraint in the structural pattern expresses that the matching is successful iff the expression value (v.getValue()) is below the minimum (min.getValue()) or above the maximum (max.getValue()) allowed values defined by the corresponding signal. In this case, the specified value is outside of the signal range and an inconsistent expression is detected. Since min and max are optional, there is a null check on them: If they are not bound, their values are not queried.

Example This structural pattern would identify the last requirement R14, whose representation is also depicted in Figure 3. Obviously, there is a typing error for the specification of the minimum velocity for the activation of the function "Safety Lock Door L".

3.2 Correction Operations

We specify correction operations with Story Diagrams [7], a graph rewriting language that combines graph transformations with control flow elements from UML Activity Diagrams. The graph transformations are the same formalism that is used in the structural patterns. Thus, the annotation created in the pattern detection process can be reused as input for the repair operations.

Figure 8 shows an example of a correction operation, which describes one possibility to correct defective requirements detected with the structural pattern from Figure 5. This pattern detects systems that contain subsystems as well as functions. The correction operation creates a new subsystem and outsources the functions into this system.

The signature at the top indicates that the operation receives the parameter tooComplexSystem of the type SystemWithSubsystemsAndFunctions; that is an annotation of said type that was created during the pattern detection. The first action of the correction operation, "Create temporary subsystem", specifies that, starting from the tooComplexSystem annotation, the superordinateSystem object (annotated during the pattern detection) shall be bound. Next, a new System object with name "temporarySystemName" is created as a subsystem of superordinateSystem.

The next action "Move function to temporary subsystem" is a *for each action* as signified by the double border. It represents a loop whose body is executed for each Function object that can be matched via the functions link from superordinateSystem. In that case, the action moves the function by destroying the functions link between superordinateSystem and function and creating a new link between newSubsystem (created in the first action) and function. This is repeated until no more Function objects can be bound for superordinateSystem. In this case the control flow labeled end is traversed. Upon reaching the last action, all functions have been moved to the temporary subsystem. Thus, the defect has been removed and the annotation can be deleted by using the ≪destroy≫ stereotype for the corresponding object variable and link. This concludes the correction operation.

4 Related Work

There are some other RE approaches that use a CNL for requirements validation. Similar to our approach, Denger et al. [6] propose a CNL as well as a requirements metamodel representing the contents of the CNL for the embedded systems domain. But since their goal is not an automatic processing but the support of the requirements engineer to write precise and unambiguous requirements, the approach presents neither a parsing technique nor an automated validation. Thus, our approach can be seen as a continuation of this idea by filling in these gaps and providing validation and correction techniques for a requirements specification. Similarly, in [14] more general textual templates for the formulation of precise requirements are pro-

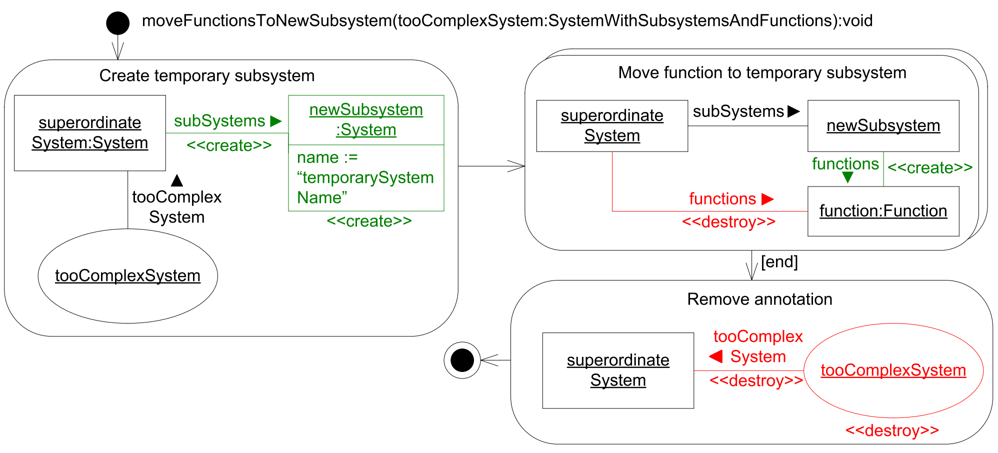

Figure 8. Repair Operation: Move functions to new subsystem

posed. These templates constrain only some syntactical elements of a requirement and define not its semantics. In order to apply our approach to such templates they have to be more constrained (e.g., process words must be defined in a dictionary) and their contents have to be mapped to a metamodel like the one presented in this paper. Then, structural patterns for the validation as well as correction operations could be specified in terms of this metamodel.

The CNLs Attempto Controlled English (ACE) [8] and Processable English (PENG) [15] enable a translation to first-order logic notation, which can be used for automated reasoning. In [5] a CNL for the formulation of use cases is presented that is translated into a formal model of the process algebra Communicating Sequential Processes (CSP). This model serves as input for a CSP model checker. Since the underlying semantics of these three languages differ, completely different facts are formulated with such languages than in our CNL, and correction operations are not possible.

Ambriola and Gervasi [1] propose the sophisticated parser framework CIRCE. The resulting model can be checked for the satisfaction of certain rules, similar to our approach. Additionally, graphical multi-view models are build from the specification for the manual validation by the requirements engineer. But CIRCE does not offer correction operations like our approach. Furthermore, we believe that the use of a CNL can constructively guide the requirements engineer to write precise specifications, like it is the philosophy of the CNL approaches mentioned above.

Wilson et al. [18] present objectively measurable quality indicators for requirement specifications in natural language. These metrics contribute positively or negatively to the overall quality of a specification and encompass the use of imperatives like "must" or weak expressions like "appropriate", for example. Like in our approach, the metrics are assigned to the quality criteria of the IEEE Standard 830. An automatic analysis of a specification wrt. to the indicators determines the overall quality. But the focus is not on the identification of individual defective requirements like in our approach.

Instead of specifying structural patterns in a negative way for the detection of ASG fragments that violate our domain-specific rules, it is also possible to formalize the rules in a positive way by using the Object Constraint Language (OCL) [11] to check that the rules are satisfied. The technique based on graph transformation used in this paper was already successfully applied on the analysis and repair of MATLAB/Simulink models within the MATE project (e.g., [16, 2]), in which some deficits of OCL were identified. The most important point is that OCL does not support correction operations [16] as provided by our approach. These are in turn—in the form of quick fix support—offered by the parser framework Xtext[4], which is also part of our requirements editing environment. Xtext provides another OCL-like constraint language and additionally allows Java-based checks, while the quick fix operations have to be coded in Java. In contrast, we can specify rule-violating structural patterns as well as correction operations with the same declarative, model-based formalism. Furthermore, the specification of complex rules in terms of a metamodel with OCL is not well-suited, since path navigation expressions combined with set operations and operational if-then-else-statements have to be formulated [2]. However, our approach provides purely declarative and visual patterns that can be combined with complex constraints.

5 Conclusion and Outlook

We presented an approach for the automated validation and correction of textual, functional system requirements

[4]`http://www.eclipse.org/Xtext`

describing a function hierarchy in the automotive domain. To enable the automatic processing of the textual requirements, we use a slightly extended CNL that is already used in the automotive industry. By using a grammar for the formalization of the CNL, the CNL requirements are parsed into an ASG typed by a requirements metamodel, which is a different representation of the CNL contents. On the basis of this metamodel we specify structural patterns based on a graph transformation formalism for the detection of requirements that violate rules for the use of the CNL. Additionally, we propose correction operations specified with the same formalism, which allow to automatically transform a rule-violating ASG into a correct one. Afterwards, the corrected ASG can be translated to textual requirements shaped by the CNL using code generation techniques.

The use of a CNL enables the automated processing of the textual requirements, which are understandable for all stakeholders at the same time since they are based on natural language. The validation takes care of unwanted properties of the overall requirements specification that cannot be eliminated by the requirements editor such as typing errors and wrong references. By automatically searching for simple structural patterns within the ASG generated from the requirements specification, manual validation techniques, which are time-consuming, error-prone, and often monotonous, are avoided. Furthermore, the automated validation can be simply repeated after the change of some requirements, which happens often within the automotive domain. If defective requirements are found, the requirements engineer can be further guided by correction operations for the adjustment of the requirements to achieve a high quality specification.

While the tool environment for formulating the CNL requirements and parsing them into the ASG is completed, we are currently working on the specification of the structural patterns and the surrounding framework. Furthermore, the possibility of the automated requirements processing of the CNL can enable the automatic transition to the model-based design by transferring a part of the information to an initial design model using, for example, model transformations. Last, we believe that the overall approach is also feasible for other domains, if the artifacts containing domain-specific information (i.e., the CNL, the requirements metamodel, the structural patterns, and the correction operations) are adapted.

References

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

[2] C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer. Checking and enforcement of modeling guidelines with graph transformations. In *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 313–328. Springer, 2008.

[3] Automotive Special Interest Group (SIG). Automotive SPICE: Process reference model. Release v4.5, 2010.

[4] H. Becker. *High Noon in the Automotive Industry*. Springer, 2006.

[5] G. Cabral and A. Sampaio. Formal specification generation from requirement documents. *Electronic Notes in Theoretical Computer Science*, 195:171–188, 2008.

[6] C. Denger, D. M. Berry, and E. Kamsties. Higher quality requirements specifications through natural language patterns. In *SWSTE '03: Proc. of the IEEE International Conference on Software-Science, Technology & Engineering*, pages 80–91. IEEE, 2003.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In *Theory and Application of Graph Transformations: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 296–309. Springer, 2000.

[8] N. Fuchs, K. Kaljurand, and T. Kuhn. Attempto controlled english for knowledge representation. In *Reasoning Web*, volume 5224 of *LNCS*, pages 104–124. Springer, 2008.

[9] Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for Software Requirements Specifications*, volume 830-1998 of *IEEE Std.* IEEE, 1998.

[10] R. Kapeller and S. Krause. So natürlich wie Sprechen - Embedded Systeme modellieren. *Design & Elektronik*, 2006.

[11] Object Management Group. Object constraint language: Version 2.2, OMG document number formal/2010-02-01, 2010.

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

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

[14] C. Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser, München, 2004.

[15] R. Schwitter. English as a formal specification language. In *Proc. of the 13th International Workshop on Database and Expert Systems Applications*, pages 228–232, 2002.

[16] D. Travkin and I. Stürmer. Tool supported quality assessment and improvement in MATLAB Simulink and Stateflow models. In *Proc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, volume 236 of *HNI-Verlagsschriftenreihe*, pages 61–68. Heinz Nixdorf Institute, 2008.

[17] M. von Detten, M. Meyer, and D. Travkin. Reverse engineering with the Reclipse tool suite. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, 2010.

[18] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated analysis of requirement specifications. In *Proc. of the 19th International Conference on Software Engineering*, ICSE '97, pages 161–171. ACM, 1997.