Knowledge Extraction from German Automotive Software Requirements using NLP-Techniques and a Grammar-based Pattern Detection

Mathias Schraps
Software Development
Audi Electronics Venture GmbH
85080 Gaimersheim, Germany
e-mail: mathias.schraps@audi.de

Alexander Bosler
Fakultät für Informatik
Technische Universität München
85748 Garching, Germany
e-mail: alexander.bosler@tum.de

***Abstract*—In Requirements Engineering, natural language is often used to specify the system under development with textual requirements. Especially in the automotive industry it is used to specify the processing of signals and parameters, as well as the behavior of sensors or actuators. During the creation of a specification first executable software models were developed, which have to be implemented according to the corresponding requirements. Due to the asynchronous development of specifications and software models, inconsistencies and defects may occur. To overcome this issue, we developed an approach using Natural Language Processing (NLP) techniques and a formal grammar to match semantic patterns in order to extract knowledge of requirements and represent it in an ontology. This approach will be introduced based on an example of an automotive software requirement.**

Keywords-Natural Language Processing; Requirements; Ontology; Knowledge Representation; Semantic Annotated Grammar; Knowledge Extraction; Pattern Detection.

I. INTRODUCTION

The development of embedded software in the automotive domain is a challenging task. First, requirements regarding architecture, data communication and behavior of the system under development have to be elicited and documented. This involves several stakeholders like electronic engineers, software developers, architects and other domain engineers.

During the phase of Requirements Elicitation, first executable software models are created, so-called Rapid Prototyping [1]. Therein, a partial amount of these requirements are implemented so far. During the progress of the project more and more requirements will be specified and have to be covered by the model and later by the implementation. This procedure implies a high linkage between two project phases: Requirements Elicitation and Modelling. If the artifacts of these both phases were not updated permanently by the involved developers and stakeholders, defects, errors or inconsistencies may occur and could be propagated across the entire development process. The later these issues are detected and solved, the more cost-intensive is their removal [2]. Therefore, the artifacts created in early phases of a software development project have to be consistent as much as possible.

In the automotive industry, a software requirement specification consists of more than only one document. Even though these documents come from many authors with different background and interests, they share one thing in common to specify their requirements: a natural language. Unfortunately, these natural language requirements can be incomplete, ambiguous and error-prone [3]–[5], especially in early phases of development.

This paper presents a method of extracting knowledge from textual requirements formulated in German natural language using Natural Language Processing (NLP) techniques. According to the detected patterns within a single requirement, this knowledge will be transferred into a requirements ontology in order to be able to check consistency between several requirements and for reuse purposes.

The structure of this paper is as follows. Section II gives an overview about the annotation of textual requirements using NLP-techniques. This is illustrated on a sample requirement given at the end of this section. In Section III, the pattern detection and the mapping of the sample requirement into an ontology will be introduced. Section IV provides a conclusion and outlines possible future work. The sample requirement on which all illustrations in the following sections are based, is formulated as follows: *"Wenn die Klemme 50 eingeschalten ist und s_MTrig=p_MAn, dann ist der Motor zu starten (s_MStart=1)."* An English translation of this requirement would be: *"If clamp 50 is switched on and s_MTrig=p_Man, then the engine must be started (s_MStart=1)."*

II. REQUIREMENTS ANNOTATION USING NLP

The NLP annotation process is the concatenation of different NLP-tools in a pipeline-style manner, where each tool provides additional information about the processed requirement (cf. Fig 1). This process was inspired by Arora et al [6] and adds a possibility to map the knowledge within the requirement about the system under development into an ontology.

If needed, tools are able to query the underlying ontology, where the knowledge is stored. This knowledge is extracted from the requirements in later steps. At the beginning of the requirement acquisition, it is possible to initialize the underlying ontology with a priori knowledge about predefined signals, constants and parameters. Furthermore, it is possible to add an existing taxonomy, which in this case is extracted from an existing High Level

 ISBN: 978-1-61208-465-7

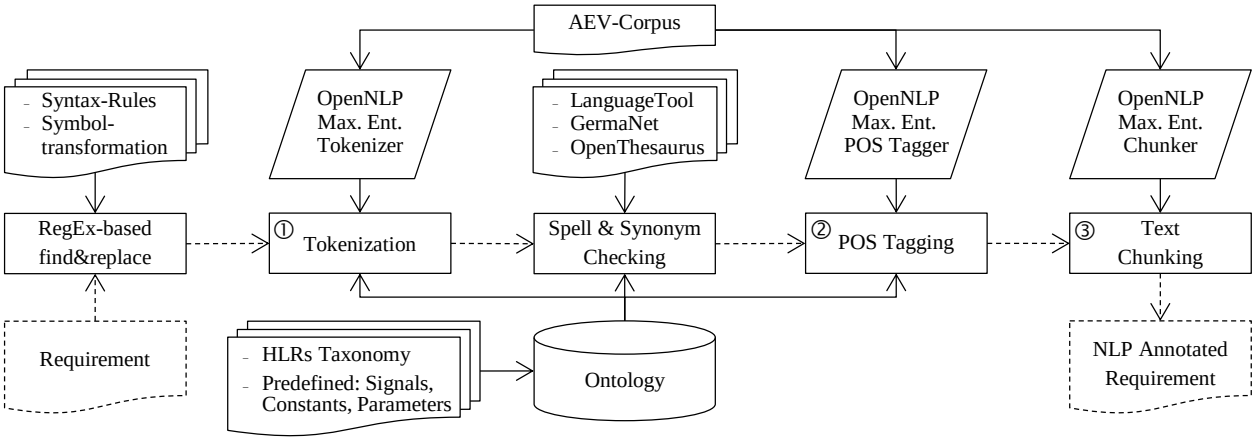

Figure 1. NLP based knowledge extraction process (part 1)

Requirements (HLRs) Specification [7]. To improve the results of the OpenNLP-tools [8], which are based on machine learning models the AEV-Corpus (internal Audi Electronics Venture Requirement based Corpus) was introduced (cf. ①, ②, ③ in Fig. 1). At the moment the AEV-Corpus consists of about 450 tokenized, POS-tagged and chunked automotive software requirements. In the following paragraphs the annotation of the requirement, which results in an "NLP Annotated Requirement" (cf. Fig. 1), is described in more detail.

The "RegEx-based search & replace" activity provides the possibility to define search and replace pairs, which are enforced at the start of the NLP-process in order to fix syntactical problems like missing or multiple whitespace characters and to standardize symbol usage.

During the "Tokenization" (cf. ① in Fig. 1), the OpenNLP Tokenizer splits the requirements according to a Maximum Entropy Model trained on the AEV-Corpus. The model achieves results similar to the default models provided by OpenNLP when tokenizing the natural language parts of a given requirement. The main advantage achieved by introducing the AEV-Corpus trained model, is the tokenization of very formal requirements: concepts like formal equations of signals and constants, C-Structs or Arrays are not part of the OpenNLP default models and thus, tokenization tends to fail. To improve the tokenization results, generated by the OpenNLP Tokenizer, the Named Entities (contained in the underlying ontology) are used to verify their correct tokenization of the currently processed requirement. The tokens, which are generated for the sample requirement (cf. Section I), are shown at ① in Fig. 2.

The "Spell & Synonym Checking" activity uses the Java version of the JLanguageTool [9] to detect misspelled tokens and provides a list of suggestions for each of them. Furthermore, a set of synonyms for each token is created using GermaNet [10] and OpenThesaurus [11]. The resulting set is used to query the underlying ontology to check if one or more of them are already included. After this, the synonyms provided and found at least once in the ontology, are added to their corresponding token.

In the "POS Tagging" activity (cf. ② in Fig. 1), each token (word, punctuation character and mathematical symbol) of the requirement is tagged with its corresponding Part Of Speech (POS) Tag. Since formal definitions are very common in automotive software requirements and common Tagsets only provide Part Of Speech Tags for natural language, we extended the STTS Tagset [12] by *$S* to tag mathematical symbols (=,<,>,≥,≤,+,-,…) and *$L* to tag listing symbols (:,->) to address this issue. The assignment of the POS-Tags to each token is done using the OpenNLP POS-Tagger based on a Maximum Entropy Model, which is trained on the AEV-Corpus. To improve the POS-Tagging results generated by the OpenNLP POS-Tagger, the Named Entities and Concepts (contained in the underlying ontology) are used to verify their correct tagging in the currently processed requirement. The POS-Tags for all tokens, generated during the "Tokenization" of the sample requirement (cf. Section I), are shown at ② in Fig. 2.

The "Text Chunking" activity (cf. ③ in Fig. 1) uses the OpenNLP Chunker to chunk the requirement according to a Maximum Entropy Model, which is trained on the AEV-Corpus. Each chunk consists of one or more tokens, tagged with the corresponding Chunk-Tag and can be considered as a "part of interest" of the processed requirement. The chunks, generated for the sample requirement (cf. Section I), are shown at ③ in Fig. 2. The meaning of the used Chunk-

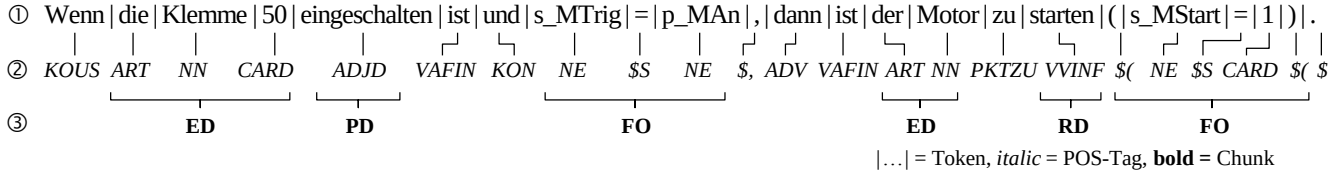

Figure 2. Sample requirement annotated by the NLP-process according to Fig. 1

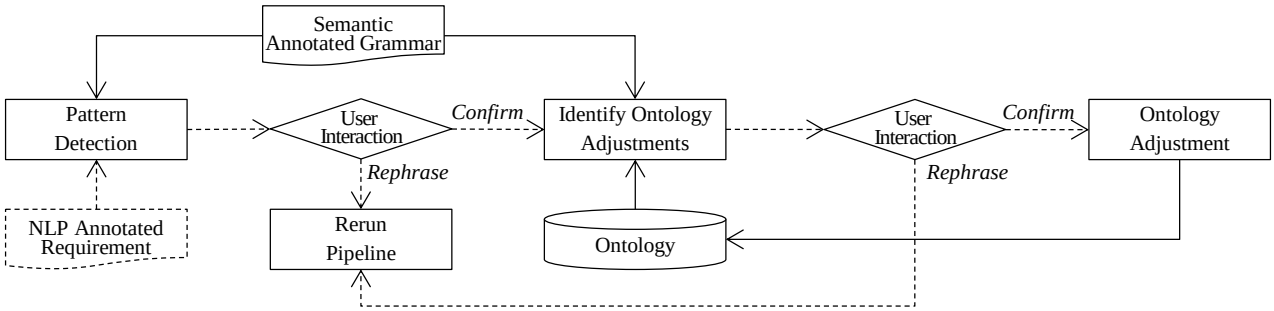

Figure 3. NLP-based knowledge extraction process (part 2)

Tags is as follows: Entity Definition (ED), State/Property Definition (PD), Action/Relation Definition (RD), Formula (FO).

III. PATTERN DETECTION AND ONTOLOGY ADJUSTMENT

Based on the NLP Annotated Requirement (cf. Fig. 2) and patterns in the form of a semantic annotated grammar (cf. Fig. 4), which is inspired by [13], the "Pattern Detection" and the "Ontology Adjustment" (which is split into "Identify Ontology Adjustments" and "Ontology Adjustment" to allow User Interaction), extracts the knowledge contained in the processed requirement and stores it into the underlying ontology (cf. Fig. 3). A more detailed view on the knowledge extraction process is given in the following paragraphs.

During the "Pattern Detection", the semantic aspect of the semantic annotated grammar is ignored since it does not provide any additional information for this activity. The first step in the Pattern Detection is the aggregation of tokens, POS-Tags and chunks to a list, which is referred to as "word". The "word" only contains elements, which are available in the NLP Annotated Requirement and also terminals of the grammar. According to this rule and the semantic annotated grammar, the following elements in the annotated sample requirement (cf. Fig. 2.) would be ignored: |ist|(VAFIN), |,|($,), |ist|(VAFIN), |zu|(PKTZU), |.|($.) and the "word" would be: "Wenn ED PD KON FO dann ED RD FO". To verify if the "word" can be expressed in the formal language defined by the semantic annotated grammar, a finite state machine based recognizer is being used. For the sample requirement the recognizer would tell us that our "word" can be expressed using the If-Then-Pattern of the semantic annotated grammar (cf. <If-Then-Pattern> in Fig. 4).

At the end of the "Pattern Detection" activity, the user is informed about the results of the "Spell & Synonym Checking" activity (cf. Section II) and whether a valid requirement pattern was found in the processed requirement or not. This allows the user to rephrase the requirement or to start the "Ontology Adjustment" for the processed requirement.

The "Identify Ontology Adjustments" activity performs two major tasks. At first, it creates a temporary knowledge representation for the processed requirement, based on the semantic annotated grammar (cf. Fig. 4, Fig. 5 and Fig. 6). Secondly, it checks whether the insertion of the temporary knowledge representation can be performed to the

```
<ED-Chunk> ::= <optional-Article> <Entity>
<optional-Article> ::= " " | ART
<Entity> ::=
  NN          => createIndividual(valueOf(NN),valueOf(NN))    |
  NN CARD     => createIndividual(valueOf(NN),
                 concat(valueOf(NN)," ",valueOf(CARD)))

<If-Then-Pattern> ::=
  Wenn <Condition> dann <Action> => createIfThenReq(
                                    individualOf(<Condition>),
                                    individualOf(<Action>))
<Condition> ::=
  FO                    => createCondition(individualOf(FO))  |
  ED PD KON <Condition> => createCondition(individualOf(ED),
                           individualOf(PD),
                           addSubCondition(<Condition>))
<Action> ::=
  ED RD       => createAction(individualOf(ED),individualOf(RD)) |
  ED RD FO    => createAction(individualOf(ED),individualOf(RD),
                 individualOf(FO))
```

Figure 4. Simplified Semantic Annotated Grammar for Pattern Detection and Ontology Adjustment in BNF-Style

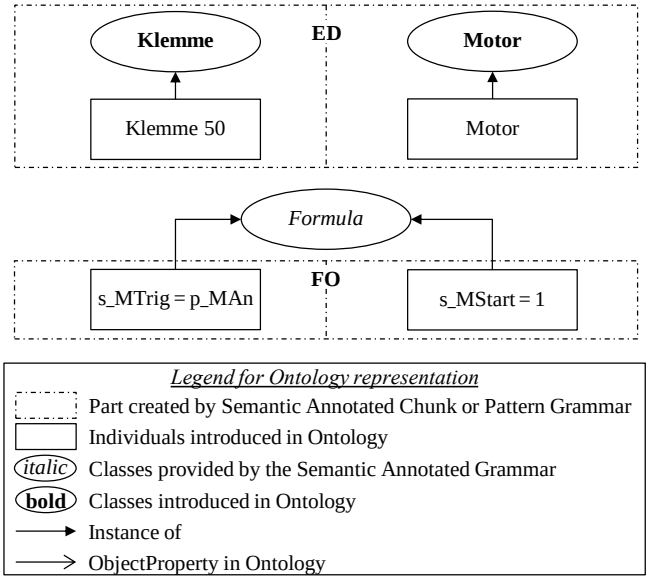

Figure 5. Ontology representation of ED and FO chunks

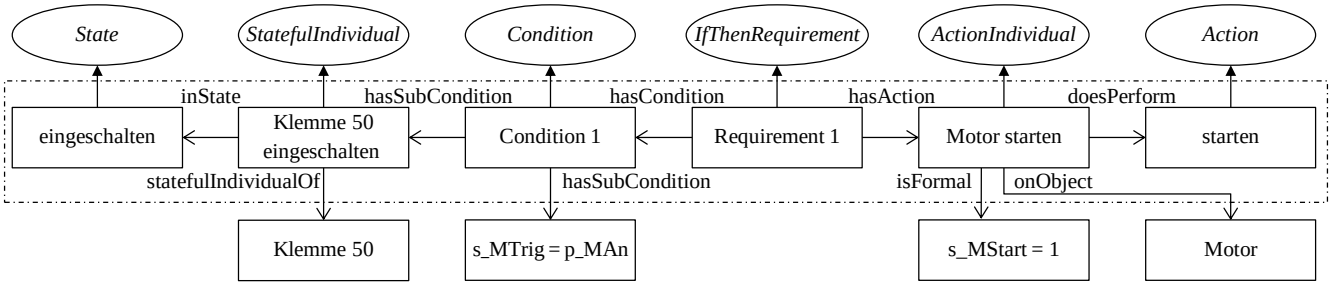

Figure 6. Knowledge representation of the sample requirement

underlying ontology without the violation of existent axioms or not. To create the temporary knowledge representation for the entire processed requirement, a knowledge representation is created for each chunk according to the semantic annotations of its Chunk-Grammar (cf. <ED-Chunk> in Fig. 4). The temporary knowledge representation, created for the ED and FO chunks of the sample requirement are different, since FO is an instance of a generic *Formula* Class and ED, as defined in the Semantic Annotated Grammar (cf. Fig. 4), creates both, the Class itself and an Individual as an instance of the Class (cf. Fig. 5). After the temporary knowledge representation for the chunks has been build, the contained individuals are connected and enhanced with new knowledge according to the semantic annotations of the Pattern-Grammar (cf. <If-Then-Pattern> in Fig. 4), determined during the "Pattern Detection" activity. The temporary knowledge representation, which is created for the sample requirement, is given in Fig. 6. Otherwise, if there is no matching pattern found during the "Pattern Detection" activity and the user confirmed the formulated requirement in the previous activity, the temporary knowledge representation of each single chunk will be linked to a temporary DefaultRequirementIndividual, which represents a lean requirement structure within the ontology. Finally, the "Identify Ontology Adjustments" activity checks whether it would be possible or not to insert the temporary knowledge representation into the underlying ontology without violating existent axioms of previous inserted requirements, which may lead to an inconsistent ontology. During this step, the underlying ontology is not updated or modified but queried to detect axiom violations.

If the "Identify Ontology Adjustments" activity determines, that it is not possible to insert the temporary knowledge representation into the ontology without violating existent axioms, the user is asked whether he/she wants to rephrase or refine the requirement or continue with the next process step according to Fig. 3 by confirming the detected issue.

"Ontology Adjustment" is the final step in the knowledge extraction process. It updates the underlying ontology according to the temporary knowledge representation of the sample requirement (cf. Fig. 6), which was created by the "Identify Ontology Adjustments" activity. If the ontology can't be updated with the temporary knowledge acquired during the previous activity without violating existent axioms (as determined by the "Identify Ontology Adjustments" activity) and the user confirmed the issue after the "Identify Ontology Adjustments" activity, every element of the temporary knowledge representation, that violates an existing axiom is removed from the remaining temporary knowledge representation and the therein remaining elements are inserted into the ontology and marked to be partial.

IV. CONCLUSION AND OUTLOOK

In this paper, we presented an approach to annotate automotive software requirements formulated in German natural language using NLP-techniques. The pattern detection matched predefined patterns and transforms the tagged and chunked parts of a requirement according to its semantic into a requirements ontology in order to represent the knowledge of the entire requirement.

In our next work, we will support a mapping of the developed requirements ontology to block-elements of a software model created with MATLAB Simulink. This will provide the ability to trace the semantic of requirements between the phases Requirements Elicitation and Modelling within the embedded software development process. In further stages, this approach will be evaluated by a prototypical tool with a graphical user interface to let the user write requirements and check the consistency to the corresponding software model.

REFERENCES

[1] J. Schäuffele and T. Zurawka, Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen, 5th ed. Wiesbaden: Springer Fachmedien Wiesbaden, 2013.

[2] S. McConnell, Code complete: A practical handbook of software construction, 2nd ed. Redmond, Washington: Microsoft Press, 2004.

[3] E. Hull, K. Jackson, and J. Dick, Requirements Engineering, 3rd ed. London: Springer Verlag London Limited, 2011.

[4] C. Rupp and SOPHIST GROUP, Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis, 5th ed. München, Wien: Hanser, 2009.

[5] K. Pohl, Requirements Engineering: Grundlagen, Prinzipien, Techniken, 2nd ed. Heidelberg: dpunkt-Verlag, 2008.

[6] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga, "RUBRIC: A Flexible Tool for Automated Checking of Conformance to Requirement Boilerplates," Proceedings of

the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), New York, NY: Association for Computing Machinery, 2013, pp. 599–602, doi:10.1145/2491411.2494591.

[7] M. Ringsquandl and M. Schraps, “Taxonomy Extraction from Automotive Natural Language Requirements Using Unsupervised Learning,” International Journal on Natural Language Computing (IJNLC), vol. 3, no. 4, pp. 41–51, 2014.

[8] Apache Software Foundation, “Apache OpenNLP,” [Online]. Available: https://opennlp.apache.org/. Accessed: Nov. 11, 2015.

[9] D. Naber, “LanguageTool,” [Online]. Available: https://languagetool.org/. Accessed: Nov. 11, 2015.

[10] University of Tübingen, Tübingen, Germany, “GermaNet - An Introduction,” [Online]. Available: http://www.sfs.uni-tuebingen.de/GermaNet/. Accessed: Nov. 11, 2015.

[11] D. Naber, “openthesaurus.de,” [Online]. Available: https://www.openthesaurus.de/. Accessed: Nov. 11, 2015.

[12] Universität Stuttgart, Institute for Natural Language Processing, Stuttgart, Germany, “STTS Tag Table (1995/1999),” [Online]. Available: http://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html. Accessed: Nov. 16, 2015.

[13] M. Schraps and M. Peters, “Semantic Annotation of a Formal Grammar by SemanticPatterns,” 2014 IEEE 4th International Workshop on Requirements Patterns (RePa), 2014, pp. 9–16, doi:10.1109/RePa.2014.6894838.

 ISBN: 978-1-61208-465-7