Semantic Annotation of a Formal Grammar by SemanticPatterns

Mathias Schraps
Software Development
Audi Electronics Venture GmbH
Gaimersheim, Germany
mathias.schraps@audi.de

Maximilian Peters
Software Engineering Group
Leibniz Universität Hannover
Hannover, Germany

***Abstract*—The elicitation and formalization of natural language requirements are still a challenge of Requirements Engineering. The characteristics of such requirements are manifold: they need to be understandable to each of the stakeholders and they should be processable by computers, for example, to analyze or to trace requirements across many steps during the system development process. For this purpose, a controlled natural language (CNL) can be used to specify textual requirements. Thus, it is possible to define the allowed syntax of requirements, but this does not consider the semantic content. With semantic annotations of a formal grammar (so-called SemanticPatterns), it is possible to capture the semantic dependencies within textual requirements and to transform them into an ontology, and this ontology is used to check the consistency of whole requirement specifications and also between the specification and other design artifacts. This paper presents an approach using a formal grammar with semantic annotations to formalize textual requirements and to make it processable by computers. The formulation and consistency check of several automotive requirements are shown based on the prototypical tool GO-Editor.**

***Index Terms*—Grammar, textual requirements, specification language, natural language, SemanticPattern, semantic annotation, consistency, ontology.**

I. Introduction

The formalization of natural language requirements is still a challenge in Requirements Engineering. On the one hand, the requirements must be comprehensible, so that all stakeholders are able to understand them in order to avoid misunderstandings or imprecise interpretations. On the other hand, the requirements need to be formal, so that they can be analyzed and processed by computers in order to reuse them in further process steps. In this case, the quality of a requirement specification plays an essential role [1].

In order to ensure that all participants have a mutual understanding of a requirement specification, attention has to be paid to the usage of the language in this specification. The customer and the participating developers mostly come from different domains. For example, today's automotive engineering involves motor, electronic and chassis engineers and also software developers and architects. They do not always have the same knowledge about the system under development, especially at the beginning of a new project. A common understanding of the system is developed in initial conversations and during the creation of project documents such as requirements specifications and system models. These two development artifacts should be mutually consistent. By transferring requirements into different structure and behavior models, care has to be taken that each individual requirement is transformed correctly in these development artifacts. Therefore, the more formally the requirement has been formulated, the better it can be evaluated through computer assistance and it can then be checked against other developed models.

The use of natural language requirements is a common specification method during the elicitation process, because such requirements can be used without specific foreknowledge and they have a broad spectrum of individual formulation possibilities. However, the requirements can become ambiguous and it is possible that interpretation errors may arise [2, 3]. Against this backdrop, especially controlled natural languages (CNLs) are suitable as a means for describing textual requirements in specification documents, because they restrict the language space to a specific problem domain and they approve only useful formulations [3]. Hence they can define the syntactic structure of a requirement but not its semantic content. However, for the formalization of natural language requirements, this semantic aspect must be considered.

Currently, no approach exists in the automotive domain to establish how to store a textual requirement formulated with a CNL into a more formal representation than structured plain text in order to support the consistency of requirement specifications in the early phases of the system development process [4].

This paper presents a method for the semantic annotation of a grammar for natural language requirements. The method is not based on natural language processing, but rather on restricting the sentence structure via a formal grammar. The so-formulated requirements are transferred into a computer-internal model (ontology) with the help of so-called SemanticPatterns and can then be used in further development phases, e.g., for consistency checks.

The structure of this paper is as follows. Section II surveys related work. In section III, the grammar approach and the annotation by SemanticPatterns are presented with several example requirements in the automotive domain. Section IV gives details about the ontological mapping of

SemanticPatterns. In section 0, the tool support with the GO-Editor and the possibility of checking the consistency of the requirements specification are presented. Section VI provides a conclusion and outlines possible future work with this approach.

II. Related Work

To increase the quality of natural language, Rupp presented a syntactical requirements template, which restricts the structure of the sentences and should help requirements engineers to satisfy this predefined structure while formulating the textual requirements [2, 5]. These given templates help to formulate the sentence according to a corresponding syntax, e.g., to avoid defects in the documentation (deletion, generalization, distortion), but it cannot prevent semantic ambiguity and inconsistency. The risk that a reader may misinterpret such formulated requirements is still present. Nevertheless, this requirements template has a major influence on the approach described in this paper, because of its widespread acceptance in requirements and software engineering.

Holtmann and co-workers [6, 7] described the usage of many different domain-specific sentence patterns, which can cover a large amount of requirements. Each of these patterns provides the requirements engineer with a template containing predefined semantics and placeholders, which can be filled with domain-specific terms. The identified domain-specific sentence patterns cannot usually be adapted and applied in other application areas, since they are bound to a specific intended application purpose.

The usage of controlled natural languages (CNLs) is another way to capture knowledge about a system under development in order to process that knowledge by computers. Attempto Controlled English (ACE) is one of the first CNLs and was introduced by Fuchs and Schwitter [8]. ACE is for general purposes and is not restricted to a certain application domain. The syntax of ACE covers a defined subset of the English language. Therefore, ACE is solely based on English grammar. Sentences, which are formulated with ACE can be translated into discourse representation structures and first-order logic in order to process these sentences by computers. Schwitter gives a brief overview about ACE and other CNLs for knowledge representation in [9].

Hull, Jackson and Dick described the use of so-called "boilerplates", which are a sequence of syntactic elements and placeholders. By concatenation, these boilerplates can be formed into small sentence components or up to entire sentences [10]. By combining several such boilerplates, the result can be used to formulate more complex textual requirements. Farfeleder et al. [11, 12] presented an ontology-based application of boilerplates for requirements elicitation using the DODT tool. In this approach the boilerplates only define the syntax for formulating a requirement and the special placeholders can be filled out with terms from an ontology. Boilerplates can be combined in any way and thus determine the structural composition (syntax) of a textual requirement. However, the semantic content of each requirement also cannot be captured and processed with computer assistance.

Breitman and Leite [13] proposed an ontology construction process by using a language extended lexicon. The process focuses on ontology scoping for reuse issues by "… providing a separation between new terms to be implemented by the ontology and terms that could be 'borrowed' from existing ontology libraries" [13]. The authors aimed to construct ontologies (dictionaries or lexicons) in a systematic way and described how to relate the captured terms to each other (term level). This process seems to be a manual task for the user using an ontology editing tool. It does not provide any possibility of representing the structure of a textual requirement or of checking whether several requirements are in conflict with each other (requirement level).

Yue et al. [14] presented a systematic review of transformation approaches between user requirements and analysis models and examined its automation, efficiency, completeness and traceability capabilities. For this review, a conceptual framework was introduced in order to be able to compare 16 published approaches with each other. According to this framework, the transformation approach presented in our paper could be classified as ontology based and the requirements configuration would be characterized as follows:

- Domain-specific information: Glossary in the form of domain-specific terms (cf. Fig. 1).
- Representation: None, but according to the given specific grammar.
- Restricted Natural Language: Yes.

Vidhu Bhala et al. [15] considered the possibility of generating a conceptual model from functional specifications automatically, which are written in natural language. After several natural language processing steps, the analyzed requirements are transformed into a UML model by applying a number of presented rules. Among others, this methodology requires grammatically correct sentences and no negative statements and does not allow non-functional requirements in order to work properly. Furthermore, because of the inherent ambiguity in the English language and the partially occurring peculiar sentence constructions, an appropriate rule may not be applied.

III. The Grammar Approach

This section gives an overview of the grammar approach and introduces the specific grammar $\mathcal{G}$spec as the core of this approach and its extension by SemanticPatterns. Figure 1 illustrates the structure and dependencies of the various components.

A. The Core – A Formal Chomsky Grammar

The specific grammar $\mathcal{G}_{\text{spec}}$ corresponds to the definition of a context-free grammar (type 2 in the Chomsky hierarchy) [16]. With the help of this specific grammar, at the beginning of a project the requirements engineer defines what types of requirements are allowed to be used later for the documentation and also the formulation rules for textual requirements, e.g., to formulate system activities for functional requirements (with or

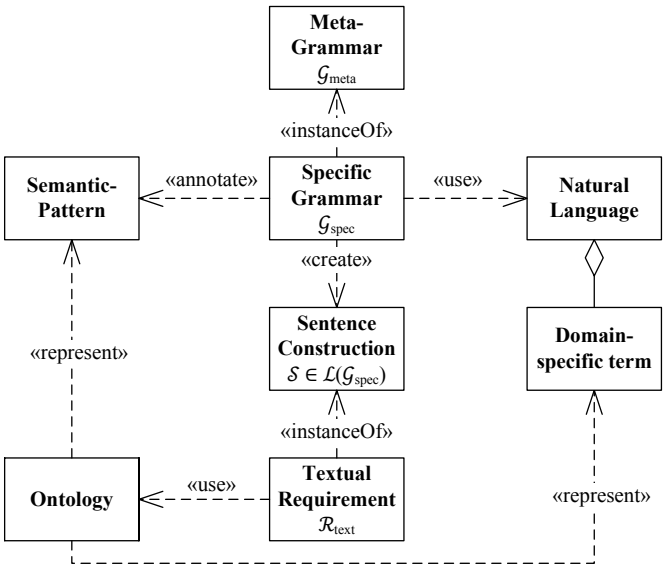


Fig. 1. The Grammar Approach

without temporal or logical conditions) or quality requirements. Similarly to boilerplates and requirement templates, the grammar approach also allows placeholders to be defined for domain-specific terms and process words (cf. section Related Work). The formulations for requirements that are allowed by the specific grammar are dependent on the natural standard language used in the project (e.g., German, English) and also the application domain (e.g., automotive, medical). In this way, the grammatical structures can be defined by the derivation rules (production rules according to Chomsky) for a specific and domain-dependent language (CNL).

B. The Extension – Semantic Annotation via SemanticPatterns

Each of these derivation rules produces grammatical phrases (parts of sentences similar to boilerplates) and can be annotated with so-called SemanticPatterns, which capture the semantic information of the requirement, e.g., that a causal relationship exists between the terms used within the sentence construction $\mathcal{S}$. Sentence construction in this context means a fully derived word of the formal specific grammar with only the occurring placeholders to be filled out by the requirements engineer. Hence by using several derivation rules according to the specific grammar, this has two effects. First, it builds up different valid sentence constructions which correspond to the previously defined types of requirements that the requirements engineer is allowed to use. Second, the SemanticPatterns which are linked to the derivation rules will be combined with each other and a more complex structure builds up. This structure can hold the semantic content of the textual requirement $\mathcal{R}_{text}$ and represents a more formal model of this requirement. This representation is then converted by a tool-based support into appropriate ontology structures. If several textual requirements of a whole requirement specification are formulated in this way, the requirements engineer can be supported by semantic technologies, e.g.,

- Creation and management of a vocabulary (terminology) with domain-specific terms used in a project.

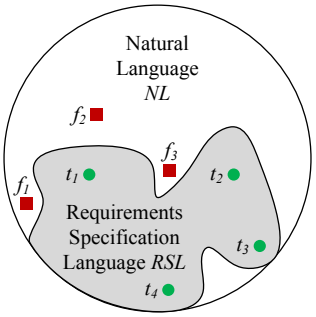


Fig. 2. Language Space of a RSL

- Analysis of semantic dependencies between domain-specific terms or single requirements.
- Consistency checks between single requirements or between requirements and other design artifacts.

Because SemanticPatterns are not a basic component of a Chomsky grammar, the described mechanism must be explicitly defined in a meta-grammar $\mathcal{G}_{meta}$. A detailed description of the meta-grammar is omitted here as it merely describes how the specific grammar has to be interpreted by a parser, and can be found elsewhere [17].

C. Advantages of the Grammar Approach

With the definition of a controlled natural language based on a formal grammar, the advantages of boilerplates and requirements templates are combined in a single approach. Hence it is possible to predefine an easy to use requirement syntax, e.g., to allow only the formulation of active requirement sentences in order to avoid defects of natural language. Furthermore, the grammar offers the possibility of adding special text fragments to the CNL to customize the specification language that is used to formulate textual requirements. This design choice depends on the specific context or circumstances of a project in order to describe the system under development appropriately. For example, if the project concerns a real-time or safety-critical system, it might be necessary to use special formulations.

Because the CNL is designed to formulate textual requirements, it can also be denoted a Requirement Specification Language (RSL), as depicted in fig. 2. Natural language formulations which should not appear in requirement specifications such as interrogative clauses or vague and incomplete statements (f_i) were explicitly disallowed ($f_i \in NL \mid i \in [1..3]$). Hence only well-formed sentence structures will be accepted by the grammar of the requirement specification language. Compared with boilerplates or requirement templates ($t_i \mid i \in [1..4]$), this requirement specification language offers a much wider language space and is not restricted to just a few formulations, but allows more application possibilities by adapting the derivation rules.

D. The Specific Grammar – a CNL for Textual Requirements

The production rules of the specific grammar are depicted in fig. 3. Each rule is in the form l→r, where "l" denotes a nonterminal symbol (starting with "/") and "r" represents the right side of the rule, which replaces "l" in a single derivation

```
1  /start→</must:1>.
2  /start→</dataRestriction:1>.
3  /start→</ifThen:1>.
4  /start→</isA:1>.
5  /must→The <i:a> shall <processWord:pw> <i:i_n>==>verb(pw,a,i_n)
6  /must→The <i:a> shall provide the ability to <processWord:pw>
     <i:i_n>==>verb(pw,a,i_n)
7  /must→The <i:a> shall be able to <processWord:pw> <i:i_n>
     ==>verb(pw,a,i_n)
8  /dataRestriction→the property '<p:prop>' of <i:a> shall be
     </restr(prop):1> <#:num>==>dataRestriction(prop,a,1,num)
9  /ifThen→If </condition:1>, then </action:2>==>ifThen(1,2)
10 /condition→<i:a> does <processWord:pw> <i:i_n>==>verb(pw,a,i_n)
11 /condition→the property '<p:prop>' of <i:a> is </restr(prop):1>
     <#:num>==>dataRestriction(prop,a,1,num)
12 /action→</must:1>
13 /action→</dataRestriction:1>
14 /restr(prop)→at least==>restrMin(prop)
15 /restr(prop)→at most==>restrMax(prop)
16 /restr(prop)→exactly==>restrExactly(prop)
17 /isA→<i:a> is an object of <c:class>==>indiv(a, class)
18 /isA→<c:sub> is a more specific term of <c:super>
     ==>concept(sub,super)
```

Fig. 3. Production rules of the Specific Grammar with semantic annotations

step. The right side "r" can be any combination of terminal and nonterminal symbols. Therefore, the specific grammar corresponds to the definition of a context-free Chomsky grammar [16]. Figure 3 depicts terminal symbols with italic font and the annotated SemanticPatterns are underlined.

The special syntactic structure of one production rule is defined in the meta-grammar, so that an appropriate implemented parser can process the rules. The notation of symbols occurring in the specific grammar is as follows (cf. fig. 3):

- /nts(p) – The nonterminal symbol "nts" with a parameter "p", where "nts" and "p" are denotations in place of concrete notations in fig. 3. These symbols were replaced successively by production rules in several steps until a valid sentence construction was derived. The parameter "p" can pass an entered value through another production rule to a specific SemanticPattern, e.g., the value in the parameter "prop" (fig. 3, line 11) is forwarded to one of the restriction options and different SemanticPatterns in lines 14–16.
- </nts:1> – A placeholder for a nonterminal symbol with the parameter "1", which has to be replaced by any production rule. The numeric parameters act here to combine several SemanticPatterns with each other and thus to capture the whole semantics of an entire textual requirement.
- <i:a> – A placeholder for an individual "a", e.g., an object or subject of a sentence. These placeholders can be replaced with domain-specific terms from a terminology (or ontology) while formulating the textual requirement.
- <#:num> – A placeholder for a selectable number "num".
- <processWord:pw> – A placeholder for a selectable process word "pw". With process words, the intended functionality or process can be defined which is specified in the textual requirement.
- <p:prop> – A placeholder for a property / attribute "prop" of an individual.
- <c:class> – A placeholder for a concept "class", e.g., a system or a kind of data. This rule is to classify an individual and to build up the project's terminology.
- ==>semPat(parameter-list) – With this notation, the semantic annotation "semPat" will be added to a production rule, and according to the type of pattern it passes a list of parameters which contains the entered values (e.g., individuals, process words, numbers).

E. Using the Grammar for Automotive Requirements

To prove the concept of the grammar approach, this section will show that a number of automotive requirements can be formulated using the developed grammar. The production rules which are depicted in fig. 3 already cover a wide range of applications. The following example introduces eight different textual requirements from an automotive project (cf. TABLE I). A distinction is made between several different types of requirements. Types 1, 2 and 3, which are widely used in requirements engineering, are intended to specify functional requirements [2, 5]. Additionally, the specific grammar supports the possibility of specifying quality requirements (type 4).

- Type 1 – Independent system action. This type is used to specify the functionality that the system under development shall perform without any user interaction.
- Type 2 – User interaction. This type of requirement is

TABLE I. EXAMPLES OF TEXTUAL AUTOMOTIVE REQUIREMENTS

ID	Requirement description	Type	# args	Used patterns
R1	The Online-Application shall receive TEC-messages.	1	2	Verb
R2	The Navigation-Map shall display Speed-And-Flow-Data.	1	2	Verb
R3	If the property active of Route-Guidance is exactly 1, then the Online-Application shall be able to request TEC-messages from content provider.	3	3	IfThen, RestrExactly, Verb
R4	If the property count-valid-license of Online-Application is at least 1, then the Online-Application shall activate the traffic license.	1	2	IfThen, DataRestr, RestrMin, Verb
R5	The Online-Application shall protect traffic-data against unauthorized access.	1	3	Verb
R6	The property feature-accessibility of Online-Application shall be at least 99%	4	-	DataRestr, RestrMin
R7	The Online-Application shall provide the ability to configure map contents via control buttons.	2	3	Verb
R8	The property latency of a traffic request shall be at most 10s.	4	-	DataRestr, RestrMax

used to specify the system functionality with a user interface. At this stage of development, the presented specific grammar does not support to whom the functionality is available.

- Type 3 – Interface requirement. This type of requirement is used to specify an interface to another system on which it depends in order to perform the desired functionality.
- Type 4 – Data restriction. This type of requirement is used to specify quality aspects of the system under development, especially performance characteristics.

With the developed specific grammar (cf. fig. 3), it is also possible to formulate conditions for the requirement (for examples, see R3 and R4 in TABLE I).

The column "#args" indicates the number of obligatory arguments of a verb and of a process word in the context of requirements engineering. The number of arguments indicates the degree of transitivity of process words:

- #1 – Intransitive verbs. Verbs which require only one argument (the subject) and do not require any object [18]. This type of verb is not useful for formulating functional requirements and is not considered in this approach.
- #2 – Transitive verbs. Verbs which require two arguments: the subject and the (direct) object. For example, in R1 the Online-Application receives TEC-messages, where "Online-Application" is the subject and "TEC-messages" is the object (cf. TABLE I).
- #3 – Ditransitive verbs. Verbs which takes three arguments: a subject, a direct object and an indirect object [19]. For example, in R7 the Online-Application must be able to configure map contents via control buttons. Here the verb "configure" takes the indirect object "control buttons" with the preposition "via".

IV. Ontological Representation of SemanticPatterns

This section describes the possibility of structuring several SemanticPatterns in order to formulate more complex requirements. In addition, the representation as ontology constructs of the project's terminology and the semantics of the textual requirements is explained.

A. Structure of Combined SemanticPatterns

The annotation of the single production rules depicted in fig. 3 has the result that several SemanticPatterns can occur in a fully derived sentence construction. This effect is also shown in TABLE I. The column "used patterns" lists all the SemanticPatterns that are used in the formulation of the corresponding requirement. Figure 4 shows the structure of the SemanticPattern "verb" which is applied in the example requirement R7.

If there is a condition in a particular requirement, several SemanticPatterns can also be combined and structured hierarchically (e.g., requirements R3 and R4 in TABLE I). This is indicated by dashed lines in fig. 4. Here, the SemanticPattern "verb" is substituted by the numeric parameter "2" of the ifThen-Pattern (cf. fig. 3 and fig. 4).

The internal structure of the SemanticPatterns remains hidden to the user of the grammar. It is only used so as to be able to represent the semantics contained in textual requirements as an appropriate data structure, in this approach as an OWL-ontology.

B. Ontological Mapping of the Project's Terminology

Ontologies are suitable for representing knowledge about a certain domain which can be modeled in an ontology. The aforementioned types of placeholders, <i>, <#>, <processWord>, <p> and <c>, are used to pass concrete parameters to the SemanticPatterns. Depending on the implementation of the SemanticPattern, these parameters can be mapped into different structures of an ontology. Only the placeholders for individuals and concepts have to be considered in order to map domain-specific terms into a corresponding structure.

Because a glossary of terms is an essential part of a requirement specification, the terms used in the requirements are represented as taxonomy in the ontology; cf. fig. 5, where the prefix "c:" or "i:" denotes whether it is a concept or individual; it is not a statement about different namespaces of this ontology.

For processing and manipulation of the ontology, the Java-based OWL API is used in this approach [20]. Via this interface, among other things, different axioms can be added into the ontology, for example:

- The assignment of a new individual "Online-Application" to a concept "System" via OWL-ClassAssertionAxiom, depicted as "hasIndividual" in fig. 5.
- The refinement of concepts via OWL-SubClassOfAxiom, depicted as "isA" in fig. 5.

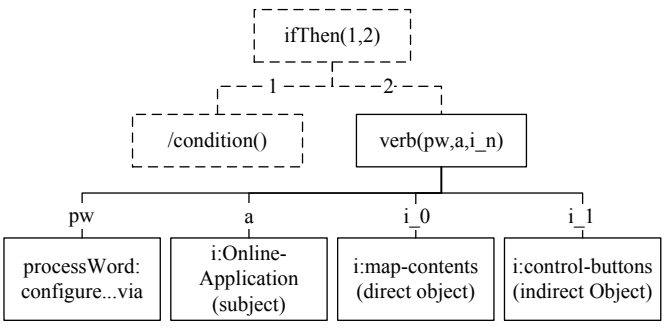


Fig. 4. SemanticPattern structure of R7

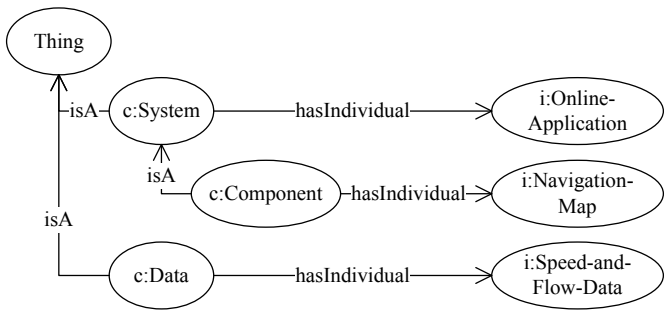


Fig. 5. Taxonomy

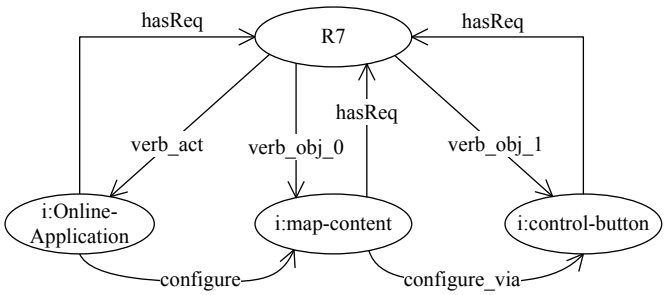


Fig. 6. Representation of requirement R7 in an ontology-structure

C. Ontological Mapping of Textual Requirements

In addition to the project's terminology, the semantics of the actual textual requirement are mapped into the ontology. While formulating a certain requirement, the structure of the ontology will be created automatically because of a defined mapping of each SemanticPattern.

The textual requirements contain the knowledge about a system under development, i.e., that the domain-specific terms used are associated by corresponding process words. The textual requirement R7 from TABLE I, which was annotated with the SemanticPattern "verb", is depicted as an example in fig. 6. The relationships between the individuals (in an ontology called "properties") are represented as owl:objectProperty and are each inserted into the ontology via an OWL-ObjectPropertyAssertion with the parameters property-name, source-individual and target-individual. By inserting several such axioms into the ontology, a structure is built as shown in fig. 6. This illustration shows several kinds of such semantic properties:

- hasReq – connects all subjects and objects of the textual requirement with an additional requirement-individual. This is to link all domain-specific terms with the unique requirement-ID and to establish whether a term is used within any textual requirement of a specification.
- verb_act, verb_obj_[n] – connects the requirement-individual and denotes whether the linked individual is the actor (subject) or one of the objects. The numbers indicate whether the object is used in a direct or indirect sense of transitivity.
- configure, configure_via – these OWL object properties represent the process word used in the textual requirement. As in this example, the transitive verb is split into two OWL object properties, "configure" and "configure_via", which connect the subject ("i:Online-Application") with the direct object ("i:map-content") and the direct object with the indirect object ("i:control-button").

Hence with this semantic mapping of a SemanticPattern into ontology-structures, the knowledge of many textual requirements can be represented in an ontology.

V. Grammar- and Ontology based Requirements Editor

To show the process of formulating textual requirements with a semantically annotated grammar and to perform consistency checks of a requirement specification, this section presents a prototypical tool called "GO-Editor" [21].

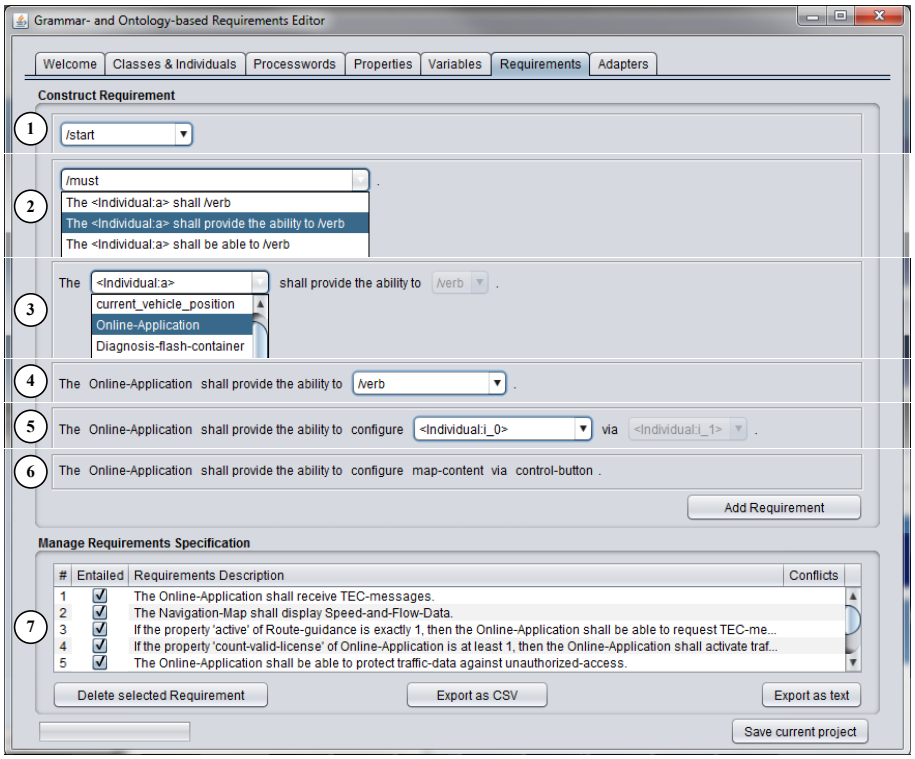


Fig. 7. Formulating a textual requirement R7 with the GO-Editor

A. Writing Textual Requirements with the GO-Editor

Figure 7 shows several tabs with which the requirements engineer can choose between the features of this tool, e.g., browsing or managing the concepts and individuals of the project's requirement specification. By selecting the "Requirements" tab, the user can formulate the textual requirements:

1) Begin with the start symbol. Choose the first production rule from a list of possible sentence constructions, e.g., "must" for a functional requirement (cf. fig. 3, lines 1-4). While choosing several derivation rules, the sentence construction will be annotated with SemanticPatterns in the background and the structure shown in fig. 4 is built up.

2) Select the type of requirement. By choosing one of the possible sentence constructions, the requirements engineer can define its type (cf. fig. 3, lines 5-7).

3) Select the acting individual. Choose the acting individual (subject) of this requirement from the project's terminology. During the process of formulating the requirement, the user can see the built-up sentence construction and fills out the placeholders with individuals stored in the ontology.

4) Define the process word. By choosing the process word of this requirement, the user can specify the functionality of the acting individual.

5) Define the object(s). Depending on the selected process word, the requirements engineer gets a preview of the resulting sentence construction. Figure 7 shows that a transitive verb has been selected and so the tool adds two placeholders for a direct object and an indirect object.

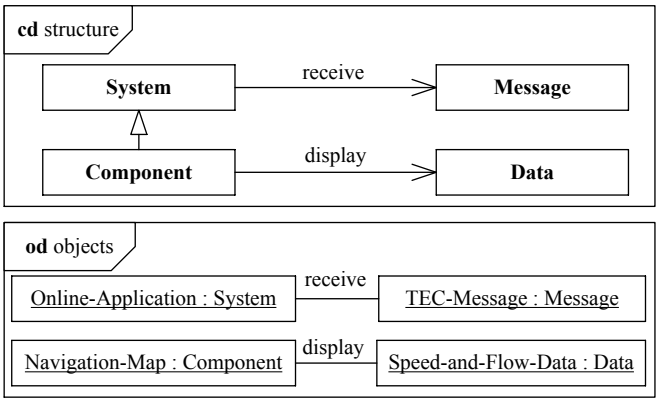
6) Check the constructed textual requirement. After the requirements engineer has constructed the sentence according to the derivation rules shown in fig. 3 and has filled out the placeholders for individuals and process words, the formulation of this textual requirement is completed. Now it can be inserted into the requirements specification by clicking the "Add Requirement" button, and the structure of this requirement will be transformed into the ontological representation (cf. fig. 4 and fig. 6).

7) View the requirements specification. In this step, the user can manage the requirements specification, e.g., to delete some requirements or to export them into another file-format.

B. Checking the Internal Consistency

After formulating the requirements by the above steps and by checking or unchecking the box in the column "Entailed", the corresponding requirement can be inserted in or removed from the requirements ontology. The column "Conflicts" indicates the requirement-IDs to the user, whether or which of the requirements are in conflict with each other. Thus the user can resolve the indicated conflicts and entail the revised requirement to achieve a consistent specification.

The evaluation of this consistency check is performed by a semantic reasoner, which analyses the requirements ontology to establish whether all entailed axioms are fulfilled and consistent. The GO-Editor uses the HermiT-reasoner because of its performance and OWL 2 support [22].



Requirements Verification

#	Requirements Description	Fulfilled ...
1	The Online-Application shall receive TEC-messages.	true
2	The Navigation-Map shall display Speed-and-Flow-Data.	true
3	The Online-Application shall be able to protect traffic-data against ...	false
4	The Online-Application shall provide the ability to configure map-c...	false

Fig. 8. Check of external consistency with a UML-Diagram

C. Checking the External Consistency

Once the textual requirements of the system under development have been elicited via the grammar approach and the corresponding knowledge has been formalized and stored in the requirements ontology, it is possible to check whether design artifacts, such as UML-models, fulfill this requirement specification, e.g., for verification purposes. Therefore, the GO-Editor supports a flexible interface to load several file formats of design artifacts via adapters. Each of these adapters implements its own translation from the file contents into an ontological representation of the design artifact. For example, the UML-adapter maps, among others, UML classes, associations and objects to OWL concepts, properties and individuals. Figure 8 shows a UML class diagram (cd) and an object diagram (od) of the modelled requirements R1 and R2 (cf. TABLE I). Each of the textual requirements, which are depicted below the UML diagrams in fig. 8, will be checked for whether it is fulfilled in the design artifact's ontology. In this example, R1 and R2 are modelled correctly but R3 and R4 were not modelled at all.

VI. Conclusion and Outlook

This paper has shown how a (formal) specific grammar can be semantically annotated by SemanticPatterns. Thereby, it is possible to transform the so formulated textual requirements into an ontology in order to process them in some further steps of the development process. The practical application of this specification language, especially in the automotive domain, and the use of an ontology for checking consistency between individual requirements and also to verify textual requirements against a structural model (and UML-Model), have been demonstrated using the example of the prototypical tool GO-Editor.

With the help of the developed specification language, the quality of extensive textual requirement specifications is increased, because of the predefined semantics in the grammar

and the ontology support. The so-formulated requirements can be mechanically analyzed and processed in early stages of the development process. This contributes to the build-up of a common understanding (the overall picture) with which all of the stakeholders agree.

In conclusion, the grammar approach is promising but should be improved. Among other things, in further stages the development of the grammar will be focused upon. In detail, the flexibility of the CNL and the set of SemanticPatterns should be increased in order to support more possibilities for formulating requirements. Because the current state of development of the presented specific grammar is mainly based on Rupp's requirements pattern [2], which is general for many other domains, it is possible to adopt this approach for these domains also. In future work we shall investigate more automotive-specific patterns and develop the presented grammar further.

Other usability improvements could be considered concerning the GO-Editor in order to raise the acceptance, e.g., an autocomplete-function or a project-independent ontology for more general verbs and terms for reuse purposes across several projects. Moreover, the mapping from UML to OWL elements of the UML-adapter is still a proprietary implementation. The Object Management Group defined a mapping between UML and OWL [23], and an example of this mapping by Eclipse ATL has been reported [24]. The implementation of this standard and the development of other adapters (e.g., for MATLAB/Simulink) will be considered in future work.

REFERENCES

[1] *IEEE Recommended Practice for Software Requirements Specification*, Std 830, 1998.

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

[3] K. Pohl, *Requirements Engineering: Grundlagen, Prinzipien, Techniken,* 2nd ed. Heidelberg: dpunkt-Verl, 2008.

[4] M. Schraps and C. Allmann, "Ontologiebasierte Entwicklung von Anforderungsspezifikationen im Automotive-Umfeld," *Softwaretechnik-Trends*, vol. 32, no. 4, 2012.

[5] C. Rupp, *Requirements Templates - The Blueprint of your Requirement*. Nürnberg, 2004.

[6] J. Holtmann, "Mit Satzmustern von textuellen Anforderungen zu Modellen," *OBJEKTspektrum*, no. RE/2010 (Online Themenspecial Requirements Engineering), pp. 1–5, 2010.

[7] J. Holtmann, J. Meyer, and M. von Detten, "Automatic Validation and Correction of Formalized, Textual Requirements," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW11)*, Piscataway, NJ: IEEE Computer Society, 2011, pp. 486–495.

[8] N. E. Fuchs and R. Schwitter, "Attempto Controlled English (ACE)," in *Proceedings of the First International Workshop on Controlled Language Applications (CLAW 96)*, 1996.

[9] R. Schwitter, "Controlled Natural Language for Knowledge Representation," in *Proceedings of COLING 2010*, 2010, pp. 1113–1121.

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

[11] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, I. Omoronyia, and H. Zojer, "Ontology-Driven Guidance for Requirements Elicitation," in *Lecture Notes in Computer Science, The Semanic Web: Research and Applications*, G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. de Leenheer, and J. Pan, Eds.: Springer Berlin / Heidelberg, 2011, pp. 212–226.

[12] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, H. Zojer, and C. Panis, "DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development," in *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2011, pp. 271–274.

[13] K. K. Breitman and J. C. S. d. P. Leite, "Lexicon Based Ontology Construction," in *Lecture Notes in Computer Science, Software Engineering for Multi-Agent Systems II*, G. Goos, J. Hartmanis, J. Leeuwen, C. Lucena, A. Garcia, A. Romanovsky, J. Castro, and P. S. C. Alencar, Eds, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 19–34.

[14] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Eng*, vol. 16, no. 2, pp. 75–99, 2011.

[15] R. V. S. Vidhu Bhala and S. Abirami, "Conceptual modeling of natural language functional requirements," *Journal of Systems and Software*, vol. 88, pp. 25–41, 2014.

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* 2nd ed. Boston, Massachusetts: Addison-Wesley, 2001.

[17] M. Peters, "Concept to capture Relations between Project Documents using Ontologies," Masterarbeit, Fakultät für Elektrotechnik und Informatik, Leibniz Universität Hannover, Hannover, 2013.

[18] P. J. Hopper and S. A. Thompson, "Transitivity in Grammar and Discourse," *Language*, vol. 56, no. 2, pp. 251–299, 1980.

[19] A. Malchukov, M. Haspelmath, and B. Comrie, "Ditransitive constructions: A typological overview," in *Studies in Ditransitive Constructions: A Comparative Handbook*, A. Malchukov, M. Haspelmath, and B. Comrie, Eds, Berlin/New York: Walter de Gruyter, 2010, pp. 1–64.

[20] *The OWL API.* Available: http://owlapi.sourceforge.net/ (2014, Feb. 27).

[21] M. Schraps and M. Peters, "Spezifikation und Verifikation von Anforderungen mit dem GO-Editor," *OBJEKTspektrum*, no. RE/2013 (Online Themenspecial Requirements Engineering), pp. 1–5, 2013.

[22] University of Oxford, Information Systems Group, *HermiT OWL Reasoner: The New Kid on the OWL Block.* Available: http://hermit-reasoner.com/ (2014, Feb. 28).

[23] Object Management Group, *Ontology Definition Metamodel (ODM)*. Version 1.0. Available: http://www.omg.org/spec/ODM/1.0 (May 2009).

[24] G. Hillairet, *ATL Use Case - ODM Implementation (Bridging UML and OWL).* Available: http://www.eclipse.org/atl/usecases/ODMImplementation/ (2007).