Automated and Quality-driven Requirements Engineering

Rolf Drechsler Mathias Soeken Robert Wille
Department of Mathematics and Computer Science, University of Bremen, Germany
Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{drechsle,msoeken,rwille}@cs.uni-bremen.de

***Abstract*—This tutorial paper summarizes selective research results from the field of automated requirement engineering. Automatization is achieved by employing natural language processing techniques. We show algorithms that work directly on the natural language text and algorithms that translate natural language text to formal models. To ensure quality, we further illustrate verification algorithms that can proof correctness of the extracted formal models.**

I. Introduction

The design flow for complex safety critical systems starts way before the implementation phase. A considerate amount of time is spent on extracting and organizing requirements from several documents that are provided by the stakeholders and customers. This process is called *requirements engineering* and usually carried out manually thus far. Some software tools, in particular IBM Rational DOORS,[1] are available that mainly focus on providing methods to organize requirements and link them to artifacts of the design flow. Besides extracting and organizing requirements, designers also link them to model elements, code blocks, and verification plans such they can be traced during the implementation phase.

In this tutorial, we present (automatic) methods which ease this flow, i.e. (i) techniques for checking pre-processing requirements in textual specifications, (ii) automatic approaches for the extraction of formal models from customer specifications, and (iii) verification methods that check the extracted formal models for consistency. *Natural language processing* (NLP) tools and formal methods are used for this purpose. Having a long history in the area of artificial intelligence, NLP has recently been described as a promising technique for electronic design automation [1], [2].

This paper summarizes the recently introduced techniques in a concise manner and gives pointers to their original references.

II. Background

The approaches reviewed in this tutorial make use of solutions from NLP and rely on descriptions provided in formal modeling languages such as UML or SysML. Background on these is given in this section.

[1]www-03.ibm.com/software/products/en/ratidoorfami

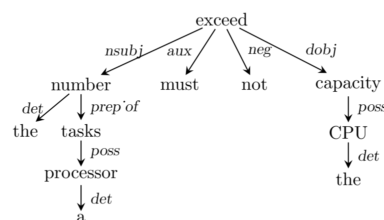

Fig. 1: Dependency graph

A. Natural Language Processing

A good overview on NLP techniques can be found in the literature [3], [4]. Since methods proposed in this paper particularly make use of *typed dependencies* [5], we explain them in more detail.

In order to represent dependencies between individual words, natural language processing techniques make use of dependency parses. For this purpose, binary relations describing syntax and semantic are extracted from a sentence. A dependency is given as a $r(g, d)$ with a relation r, a governor g, and a dependent d. As an example the relation *nsubj* binds a verb to its subject. Other relations are *nn* that groups compound nouns or *det* that assigns a noun to its determiner. Altogether, 48 relations have been arranged in a grammatical relation hierarchy. Given a sentence s, a dependency graph is an edge-labeled directed graph in which vertices represent words of s. There is an edge $g \xrightarrow{r} d$ between two different words g and d if and only if $r(g, d)$ is a dependency in s.

Example 1: The dependency graph for the sentence "The number of a processor's tasks must not exceed the CPU's capacity." is depicted in Fig. 1.

B. Modeling Languages

Modeling languages such as the *Unified Modeling Language* (UML, [6]) or the *Systems Modeling Language* (SysML, [7]) have been established to explicitly specify the design of systems prior to the implementation phase. They offer description means which are expressive enough to formally specify a complex system, but hide specific implementation details. In this tutorial, we focus on block definition diagrams, sequence diagrams, and requirement diagrams.

Block definition diagrams formally describe the structure of a system. They are composed of blocks which are organized by

978-1-4799-6278-5/14/$31.00 ©2014 IEEE

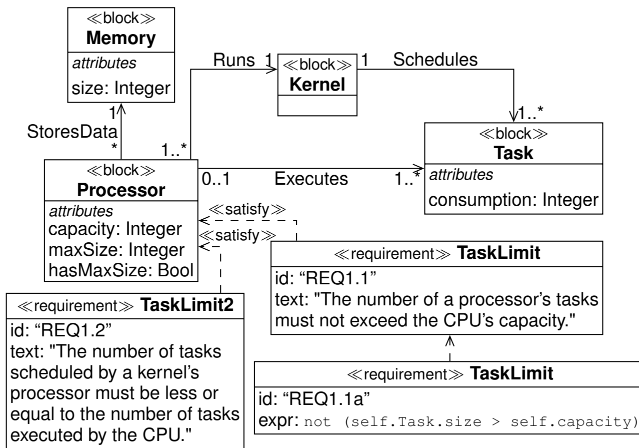

Fig. 2: Specification of a system using SysML

compartments to store different information such as attributes, operations, or relations to other blocks. Attributes describe the data elements of the block, whereas operations are used to modify them. The dynamic flow caused by operation calls can be visualized by *sequence diagrams*. Here, instances of the blocks are extended by life lines that express the time of creation and destruction in the scenario. Arrows indicate operations that are called on an instance and are drawn from the caller to the callee. Finally, *requirement diagrams* support the organization of requirements which allow for a representation of dependencies to themselves as well as to other model elements such as blocks. Requirements may be provided in natural language, particularly in early phases of a project, but also as a formal description. For the latter, the *Object Constraint Language* (OCL, [8]) may be used.

Example 2: Fig. 2 illustrates the specification of a simple computer architecture in SysML in terms of a block definition diagram. The structure of the system is defined by means of four blocks, namely a '*Kernel*', a '*Processor*', a '*Task*', and a '*Memory*'. Attributes such as '*capacity*' provide further details on the respective components (e.g. its maximal capacity). By means of a requirement diagram (at the bottom of Fig. 2 and related to the block '*Processor*') two (informal) requirements are specified that constrain the number of a processor's task. For the first requirement REQ 1.1 also a formal representation in terms of OCL is provided.

III. Natural Language Processing for Requirements Engineering

Two problems for natural language processing in requirements engineering, namely *automatic guideline checking and fixing* and *requirements classification*, are presented in this section.

A. Automatic Guideline Checking and Fixing

Several guidelines exist which aid designers in writing good requirements. These guidelines are either provided globally to a large audience e.g. by means of books or they are used internally as an agreement between employees of a company

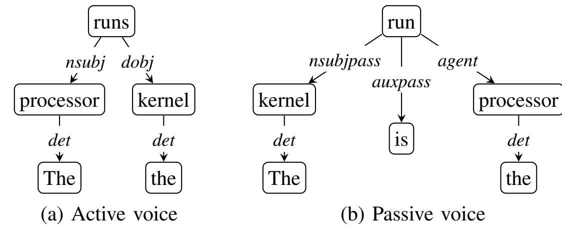

Fig. 3: Active and passive voice

and their customers. Typical examples for rules defined in such guidelines are the avoidance of imprecise words such as "should" or "could" or the use of passive verb forms. However, checking whether all these rules are followed is a cumbersome task.

Problem 1 (Guideline checking and fixing): Given a set of rules from guidelines how to write requirements and a natural language requirement R, the *guideline checking* problem asks whether R adheres to the rules. The *guideline fixing* problem also asks whether such a requirement R that violates a rule can automatically be rewritten into a requirement R' that does not violate the rule anymore.

In order to address this problem algorithms based on natural language processing techniques have been proposed that, for a given requirement, (i) can automatically determine whether a rule has been violated and, if possible, (ii) offers the possibility to automatically fix such a requirement.

Example 3: The problem is further illustrated by means of an example. A rule from some guideline document may be "*Avoid passive voice.*". By making use of typed dependencies it can easily be checked whether a sentence is given in active or passive voice since different relations are found in the corresponding typed dependency graphs. While the subject is indicated as the dependent of a '*nsubj*' relation in a sentence in active voice the relation will be '*nsubjpass*' when using passive voice (cf. Fig. 3). But it cannot only be checked rather easy whether the rule is violated by inspecting whether such dependency relations occur in the sentence; passive sentences can also be translated automatically using NLP techniques.[2]

B. Classification of Requirements

Specifications of technical systems consists of many requirements written in natural language, usually English. These requirements are typically expressed by means of one sentence and can be categorized into *low-level requirements* and *high-level requirements*. Low-level requirements can easily be translated into formal counterparts, because they are very precise and usually refer to specific signals of the design and concrete values. High-level requirements on the other hand need further investigation of the context that can be obtained from reading the specification and hence a direct translation is often not possible.

[2]This is e.g. being illustrated using the *Voice Conjugator* widget at www.contextors.com

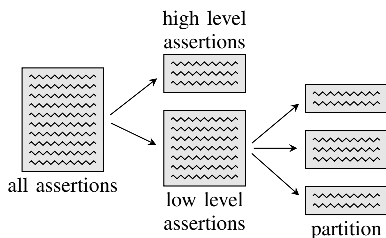

Fig. 4: Flow for requirements classification

Problem 2 (Requirements classification [9]*):* Given a set of requirements in natural language text $R = \{R_1, \dots, R_n\}$, the *requirements classification* problem asks to find a partition (L, H) of S such that L consists of all low-level requirements and H consists of all high-level requirements. The low-level requirements are further partitioned into *clusters* $\{C_1, \dots, C_k\}$ that contain similar sentences. That means that all sentences in one cluster can be automatically translated into a canonical form that is described by a *representative* of the cluster. This representative also consists of variables and hence can be used for describing the automatic translation of the low-level requirements of the cluster into formal assertions.

The problem is further illustrated by means of Fig. 4 which shows the flow of an algorithm that solves the problem. The starting point is a set of natural language requirements given in terms of English sentences (indicated in the figure by zigzag lines). These assertions are automatically partitioned into subsets of high abstraction level and low abstraction level assertions in the first step. For this purpose, the sentence is checked for special syntactic or semantic properties. One possible criteria for classification can e.g. be the occurrence of signal names in the text. In the second step the determined low level assertions are partitioned into clusters of similar sentences. For this purpose, a metric for sentence similarity is defined based on the grammatical structure of the sentence and the semantics of some words. Each cluster is represented by a graph structure that represents the general structure of the sentences and stores the words which are variable for all sentences in the cluster. For this purpose, typed dependency graphs are used and canonicized using several transformation rules.

It can easily be seen that the quality of implementations for the requirements classification problem can be enhanced when pre-processing the step using guideline checking.

IV. Extracting Formal Models from Textual Specifications

After all natural language specifications and requirements have been preprocessed using the techniques described in the previous section, the next step in the design flow is to extract formal representations from them. Thus far, transforming a textual description into a formal model is usually performed manually. But recent achievements to (semi-)automatically map informal descriptions into formal ones have been made (e.g. [10]). For this purpose, techniques for natural language processing as briefly summarized in Section II-A are utilized. We illustrate these accomplishments by means of two problems and their corresponding solutions. We show that already simple grammatical analyses enable the derivation of a formal representation of the structure and the behavior as well as the requirements of the system to be implemented.

A. Extracting Structure and Behavior

Specifications provided in natural language obviously include precise descriptions what a system is supposed to do. This leads to the following problem which we want to address in the proposed design flow.

Problem 3 (Structure extraction [10]*):* Given a natural language text, the *structure extraction* problem asks to find a formal model (e.g. a block definition diagram) that represents the components, their attributes and operations, and their relations to each other which are being described by the text.

Analyzing the natural language specifications with respect to their grammatical structure often allows to extract implications for the formal model to be generated from it. For example, (i) basic components of a system can often be derived from nouns in a sentence, (ii) their functions can often be derived from verbs in a sentence, or (iii) attributes can often be derived from adjectives in a sentence. In particular, the analysis of *test cases* in a specification are suited for those analyses as they inherently provide a compact and concise summary of the system's structure and behavior.

Example 4: Consider the following test case from some specification describing how a user is placing a telephone call:

A caller picks up the receiver from a telephone.
The caller dials the number 6-345-789.
The telephone places a call.

Fig. 5 illustrates that already from these three sentences a significant amount of structural information can be extracted: Since '*telephone*' and '*receiver*' are object nouns, it can be concluded that they represent components of the considered system (to be represented by blocks). Preceded adjectives (such as '*wireless*') substantiate objects and, thus, shall be added as attributes to the corresponding block. Verbs correlate to operations which can be invoked by components or actors. Prepositions help to determine relations between blocks. For example, '*the receiver from a telephone*' does not only imply a relation due to the preposition '*from*' but also indicates that a telephone can only have one receiver due to the definite article '*the*'.

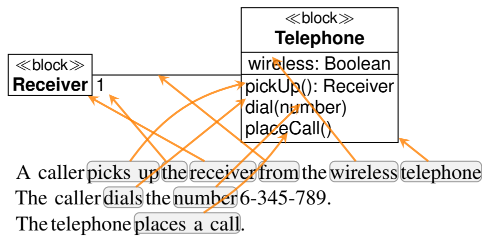

Fig. 5: Extracting structure

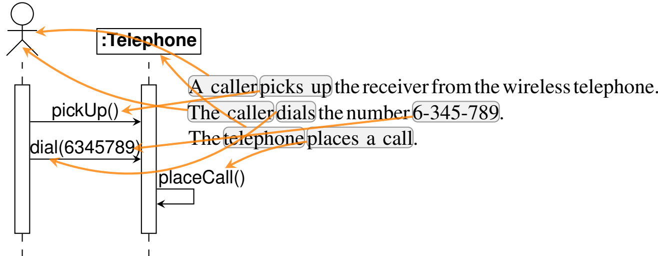

Fig. 6: Extracting behavior

Moreover, from these sentences also information on the behavior of a system, i.e. a sequence of actions to be conducted by the considered design, can be obtained: Each verb in a sentence can be mapped to a corresponding operation call. The caller and a possible callee can be determined by the subject and the object, respectively. Structural information of the system derived beforehand are exploited for this purpose. From this information, a sequence diagram as shown in Fig. 6 can be extracted which formally represents this particular scenario.

Obviously, the syntactical and grammatical information alone is not sufficient to fully automatically extract the desired formal models from a textual specification — manual interaction might still be required. But schemes as sketched above assist the designer in this process.

B. Extracting Formal Expressions

Grammatical analyses can also be exploited in order to extract formal expressions provided in natural language e.g. by means of requirement diagrams as shown in Fig. 2.

Problem 4 (Expression extraction): Given a natural language requirement and a formal representation of the system's structure, the *expression extraction* problem asks for a formal expression for the requirement. The formal expression must be consistent with the formal representation of the system.

To solve this problem, the description of a sentence (i.e. the requirement) in terms of a dependency graph as reviewed in Section II-A can be utilized. In fact, it has been observed that the grammatical structure represented by dependency graphs shows similarities to abstract syntax trees of a formal expression. This can be exploited in order to translate a natural language requirement into its formal equivalent.

Example 5: Consider the informal requirement "The number of a processor's tasks must not exceed the CPU's capacity" and its formal counterpart from Fig. 2. A direct mapping of

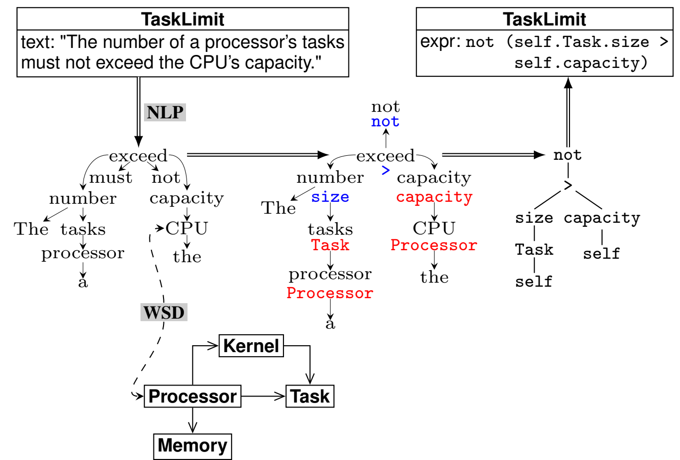

Fig. 7: Extracting a formal expression

these two descriptions (cf. top of Fig. 7) is not straightforward. But their corresponding dependency graph and abstract syntax tree (cf. bottom of Fig. 2) obviously share the same structure. In fact, "only" the natural language identifiers have to be substituted by the corresponding formal identifiers from the given model (as e.g. in case of *processor* to '*Processor*') or the underlying OCL language (as e.g. in case of *exceed* to $>$). While this is trivial in many cases, obstacles occur when e.g. synonyms are applied (as e.g. in case of *CPU* to '*Processor*'). Here, methods such as word sense disambiguation (e.g. [3]) can be applied to determine normal forms and synonyms of identifiers.

Based on the example, possible cases for extraction can be illustrated. If all relevant vertices of the dependency graph can uniquely be assigned a model element or OCL operator, the extraction of an OCL expression can be performed automatically. However, if to some vertex v more than one model element or OCL operator can be found, the sentence is ambiguous and needs to be resolved by the designer. Moreover, the cause of the conflict can easily be determined from v. The sentence cannot automatically be processed if at least one vertex cannot be assigned a model element or OCL operator. Furthermore, some sentences may result in an abstract syntax tree that does not describe a valid OCL expression. But even in these cases, the proposed direction allows for an assistance of the designer. The quality of the results can be enhanced by applying requirements classification as described in Section III-B in a pre-processing step.

V. Verification of the Resulting Formal Models

Once a formal representation of the specification has been derived, the structure, the behavior, and the requirements of the represented system are available in a formal description. However, whether the resulting model indeed satisfies the intended functionality and requirements remains unclear. Also the system description might include inconsistencies or contradictions, making a direct implementation impossible. The interest of the designer is to be aware of all these possible

problems before the actual implementation process is started. In this section, we briefly summarize possible verification problems which can already be addressed at this stage and refer to respective solutions for them.

A. Verification of Structural Aspects

Having a formal representation of the design does not necessarily imply that a working implementation can be generated from it. In fact, the formal model may inherit constraints or requirements which contradict each other. As a result, no valid instantiation would be possible and any implementation would be erroneous from scratch.

Problem 5 (Model finding, e.g. [11]*):* Given a formal model with OCL constraints, the *model finding* problem asks whether there exists a non-empty instance of the model (e.g. in terms of an object diagram) that satisfies all OCL constraints.

Approaches introduced e.g. in [11], [12], [13], [14] can be utilized for this purpose. They take the obtained formal model (representing the structure) together with the requirements (which are encoded as OCL constraints) and automatically perform the above described *consistency checks*. Besides enumerative methods [14], also elaborated formal approaches have been proposed in the recent past [11]. Considering the abstract description of the models (usually, no complex data-structures are applied), particularly the latter approaches are applicable to quite significantly complex designs.

B. Verification of Behavioral Aspects

The dynamic behavior of formal models can be verified when it is e.g. specified by means of so-called pre- and post-conditions of operations. They enable a descriptive representation of the behavior, without giving a precise implementation. A pre-condition describes in which states an operation can be called, while the post-condition describes the effect an operation has on that system state. These conditions may be specified directly from the designer or are determined e.g. by invariant elimination as described in [15].

Problem 6 (Behavioral model finding [16]*):* Given a model with OCL constraints and operations that are specified in terms of pre- and post-conditions, an initial state, and a verification task, the *behavioral model finding* problem asks whether a sequence of operation calls exists starting from the initial state which satisfies the given verification task.

The problem can e.g. be solved by translating it into an instance of the *Bounded Model Checking* (BMC) [17] and, therefore, allows for addressing certain verification tasks which are usually expressed in terms of reachability obligations. In fact, similar to verification at the implementation level, operation sequences can be determined that lead e.g. to bad states, good states, live locks, or dead locks [16]. Utilizing these techniques, again, errors can be detected before any code is written.

VI. Conclusions

In this tutorial paper, we have summarized several problems and sketched addressing (i) pre-processing at the specification level, (ii) extracting formal models from textual specifications, as well as (iii) formal verification of the extracted formal models. As can be seen by utilizing natural language processing techniques together with formal methods, an interesting design flow for requirements engineering can be found that aids with automated techniques the entry point to today's design flows.

Acknowledgments

The authors wish to thank Nabila Abdessaied for her support. This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001 as well as by the German Research Foundation (DFG) within a *Reinhart Koselleck* project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1.

References

[1] R. Drechsler, "Quality-driven design of embedded systems based on specification in natural language," in *EUROMICRO Symp. on Digital System Design*, 2011.

[2] I. G. Harris, "Extracting design information from natural language specifications," in *Design Automation Conference*, 2012, pp. 1256–1257.

[3] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Pearson Prentice Hall, 2008.

[4] N. Indurkhya and F. J. Damerau, *Handbook of Natural Language Processing*, 2nd ed. Chapman & Hall/CRC, 2010.

[5] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses," in *Conf. on Language Resources and Evaluation*, May 2006, pp. 449–454.

[6] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman, Jan. 1999.

[7] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Feb. 2008.

[8] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman, Mar. 1999.

[9] M. Soeken, C. B. Harris, N. Abdessaied, I. G. Harris, and R. Drechsler, "Automating the translation of assertions using natural language processing techniques," in *Forum on Specification & Design Languages*, 2014.

[10] M. Soeken, R. Wille, and R. Drechsler, "Assisted Behavior Driven Development Using Natural Language Processing," in *Int'l. Conf. on Objects, Models, Components, Patterns*, May 2012.

[11] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, Mar. 2010, pp. 1341–1344.

[12] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, Apr. 2006.

[13] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop*, Apr. 2008, pp. 73–80.

[14] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proofs*. Springer, Jul. 2009, pp. 90–104.

[15] M. Soeken, R. Wille, and R. Drechsler, "Eliminating Invariants in UML/OCL Models," in *Design, Automation and Test in Europe*, Mar. 2012.

[16] ——, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, Mar. 2011, pp. 1077–1082.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.