Mathias Soeken, Rolf Drechsler (Editors)

Natural Language Processing for Electronic Design Automation

May 4, 2015

Springer

Contents

1

Automating Guideline Validation for Requirements based on Natural Language Processing

No Institute Given

Summary. Recently, many approaches for automatic information extraction from technical specifications in the area of electronic design automation have been proposed. For this purpose, techniques from natural language processing are used. In order to lower the bars for designers and customers, some approaches do not intend to restrict the natural language that is used to describe the specifications. However, ambiguity and vagueness in the language often cause wrong or bad results obtained from the algorithms. This work describes preliminary ideas for automatic approaches that assist in writing the specification and aim at increasing the quality of the text. Besides improved comprehensibility, better written specifications will also enhance the quality of the automatic extraction approaches in subsequent steps of the design flow.

1.1 Introduction

The quality of requirements in textual specification documents has a significant impact on the final product. Requirements of a low quality can lead to misunderstandings and therefore to errors in the design flow that are usually difficult to detect or detected too late. Consequently, deadlines must be postponed which results in an overall higher cost of production. Furthermore, badly written requirements also impede the application of automatic methods for requirement formalization. In this paper, we present two approaches that check the quality of requirements, the first one is based on static syntactic and semantic analysis whereas the second one is based on requirement guidelines.

The first approach makes use of syntactic and semantic properties of the sentence. Basis for the quality measure are e.g. the possible interpretations of a sentence which corresponds to the parses of a sentence and the possible meanings of a word in a sentence which can be determined using dictionaries such as WordNet [15].

The second approach considers requirement guidelines of which several exist and 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. Typical examples for rules defined in such guidelines are the avoidance of imprecise words such as “should” or “could”,

adjectives such as “high”, “robust”, or “low enough”, or the use of passive verb forms. If many of those rules have been defined, it becomes cumbersome to manually check whether all of them are followed. We present algorithms based on natural language processing techniques that for a given requirement can automatically determine whether a rule has been violated.

Besides leading to more comprehensive specifications, the proposed algorithms are also of significant interest to information extraction algorithms that have recently been proposed in the field of electronic design automation [6, 18, 17]. Algorithms for checking the quality of requirements have been proposed in the past. As one example, NASA developed the tool “*Requirements Assistant*” for internal requirements quality assurance [3]. It helps to ensure that natural language requirements are complete, consistent, feasible, and unambiguous. Similar tools have been presented in [13, 2], however, non of them are freely available. Other related work has also previously discussed metrics to determine the linguistic quality of texts, sentences, and words [12, 5].

Our algorithms are implemented using techniques from *Natural Language Processing* (NLP). To evaluate our approach we collected and manually annotated requirements that are used in industrial specifications.

1.2 Preliminaries

This section describes the NLP techniques that are used for the implementation of the proposed algorithms. A good overview on NLP techniques can be found in (e.g. [11, 9]).

1.2.1 Phrase Structure Trees

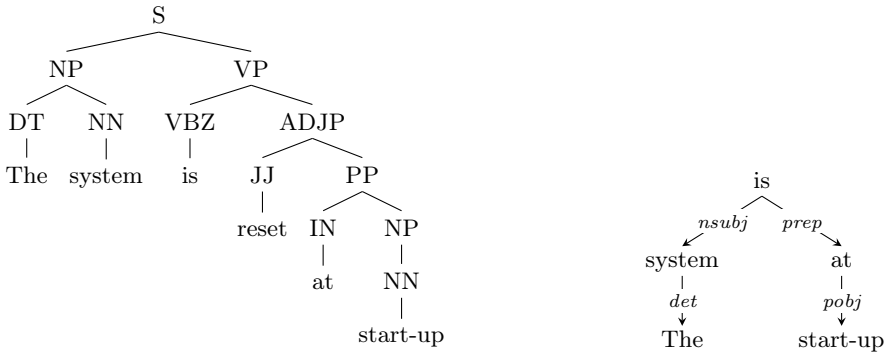
A *Phrase Structure Tree* (PST) is a tree containing structural information about a sentence. The root node represents the whole sentence. The non-terminal nodes represent the syntactic grammar structure in terms of constituents, while the terminal nodes are the atomic words of the sentence. The analysis of the sentence and annotation into a PST is performed by structural parsers such as the one contained in the Stanford CoreNLP natural language processing toolkit [14].

Due to ambiguities of natural language, there are in fact many possible PSTs for one given sentence. How the PSTs are generated depends on the structural grammar the parser uses. The parser of our choice uses a *Probabilistic Context-Free Grammar* (PCFG, [11]) as back-end grammar for the parse tree computation. Any possible PST of a sentence is assigned a score indicating the probability to be the correct parse. The PST with the highest PCFG score is the one which is most likely to be correct.

Example 1. A phrase structure tree for the sentence “The system is reset at start-up.” is depicted in Fig. 1.3(a).

1.2.2 Dependency Graphs

In order to represent dependencies between individual words, NLP techniques make use of dependency parses [4]. For this purpose, binary relations describing syntax



(a) Phrase structure tree (b) Dependency graph

Fig. 1.1. Common data structures in natural language processing

and semantic are extracted from a sentence. A dependency is given as $r(g, d)$ 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. In [4], 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 2. The dependency graph for the sentence "The system is reset at start-up." is depicted in Fig. 1.3(b).

1.2.3 WordNet

WordNet [15], developed by linguists and computer scientists at Princeton University, is a large lexical database of English that is designed for use under program control. It groups nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms called *synsets*, each representing a lexicalized concept. Each word in the database can have several *senses* that describe different meanings of the word. In total, WordNet consists of more than 90,000 different word senses, and more than 166,000 pairs that connect different senses with a semantic meaning.

Further, each sense is assigned a short description text which makes the precise meaning of the word in that context obvious. Frequency counts provide an indication of how often the word is used in common practice. The database does not only distinguish between the word forms noun, verb, adjective, and adverb, but further categorizes each word into sub-domains. Those categories are e.g. *artifact*, *person*, or *quantity* for a noun.

1.3 Guideline Validation

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 and their customers. We aim at providing solutions to the following problem:

Problem 1 (Guideline checking). 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.

We propose to solve the problem using natural language processing techniques. For our experimental evaluation we have composed a set of such rules that we extracted from several guideline documents [1, 8, 19]:

R1. Define one requirement at a time.
R2. Avoid conjunctions (and, or, with, also) that make multiple requirements.
R3. Use simple direct sentences.
R4. Each requirement must contain a subject and a predicate.
R5. Avoid let-out clauses (unless, except, if necessary, but, when, unless, although).
R6. Avoid expressing suggestions or possibilities (might, may, could, ought, should, could, perhaps, probably).
R7. Avoid weak phrases and undefined terms (adequate, as a minimum, as applicable, easy, as appropriate, be able to, be capable, but not limited to, capability of, capability to, effective, if practical, normal, provide for, timely, tbd, user-friendly, versatile, robust, approximately, minimal impact, etc., and so on, flexible, to the maximum extent, as much as possible, minimal impact, in one whack, different, various, many, some of, diverse)
R8. Do not speculate (usually, generally, often, normally, typically).
R9. Avoid wishful thinking (100% reliable, safe, handle all failures, fully upgradeable, run on all platforms).
R10. Define verifiable criteria.

We have taken the rules as they were written in the original documents. It is debatable whether all these rules make sense in each context, but it can clearly be seen, that most of the rules were not postulated with having automatic approaches in mind. As an example rule R10 "*Define verifiable criteria.*" is very difficult to be checked automatically.

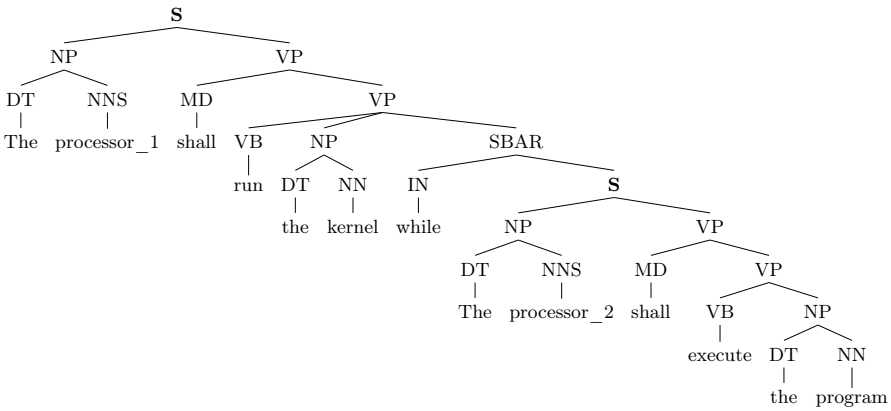
In order to handle this vagueness, the decision of the algorithms is given in terms of a tri-state value. This value distinguishes the cases of whether a rule is clearly violated or not, or whether no confident result could be computed.

1.4 Implementation

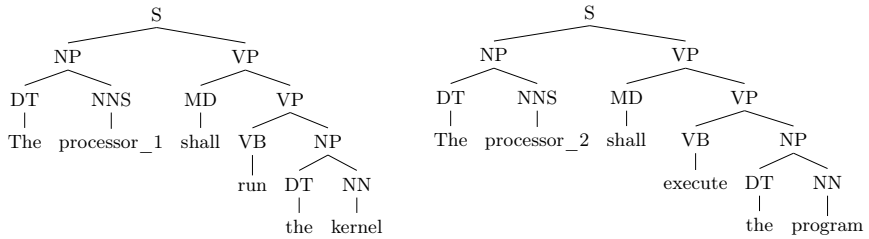
text to add

1.4.1 Rule 1: Define one requirement at a time

A requirement should be clear and targeting one point. When it contains multiple sub-requirements, it arises ambiguities and misunderstandings. In the following, we list the cases that lead to violate this rule:



(a) Phrase structure tree for S1



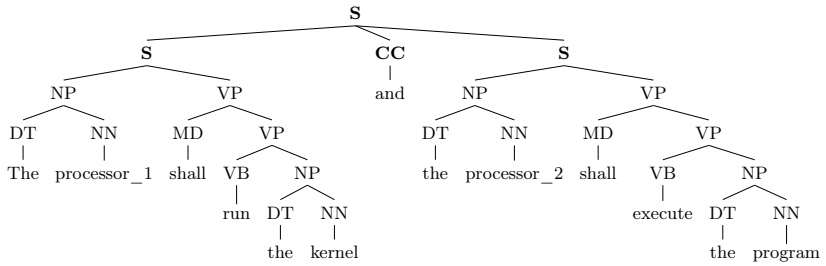
(b) Phrase structure tree for S2

Fig. 1.2. Phrase structure trees violating rule 1

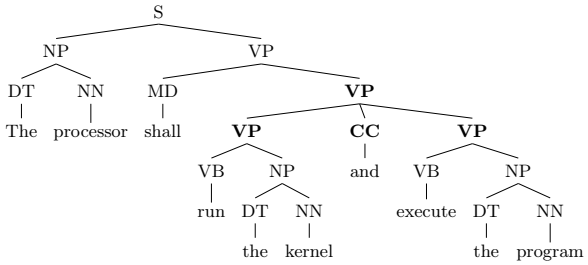
1. Joining two sub-sentences in one sentence.
   Example S1: `The processor_1 shall run the kernel while The processor_2 shall execute the program.`
2. Having two or more sentences.
   Example S2: `The processor_1 shall run the kernel. The processor_2 shall execute the program.`

The check for the existence of multiple sub-sentences in a requirement is done by exploring its phrase structure tree. We can determine that the rule has been violated when the tree has two or more phrases, i.e., two nodes labelled by "S". As it is depicted in Fig. 1.2a, the phrase structure tree of the requirement given by S1 contains two nodes with a label "S".

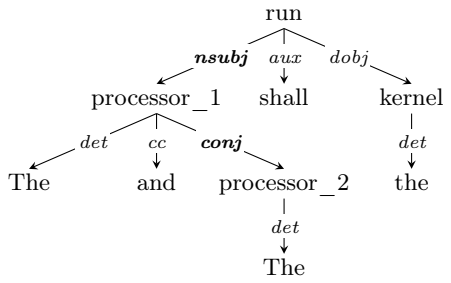
Consider now the case two, where the requirement contains two or more sentences. Here to check that, we count the number of sentences on the requirement. The rule is respected when the number of sentences is equal to 1.



(a) Joined sub-sentences



(b) Joined verbs



(c) Joined subjects

Fig. 1.3. Common data structures in natural language processing

1.4.2 Rule 2: Avoid conjunctions that make multiple requirements

Conjunctions including **{and, but, or, yet, for, nor, so}** lead to join multiple requirements in one sentence. We have three different situations that may cause the combination of many requirements in one using a conjunction:

1. Two or more sub-sentences.
   Example S3: `The processor shall run the kernel and the processor_2 shall execute the program.`
2. Two or more verbs related to the subject.
   Example S4: `The processor shall run the kernel and execute the program.`
3. Two or more subjects.
   Example S5: `The processor_1 and processor_2 shall run the kernel.`

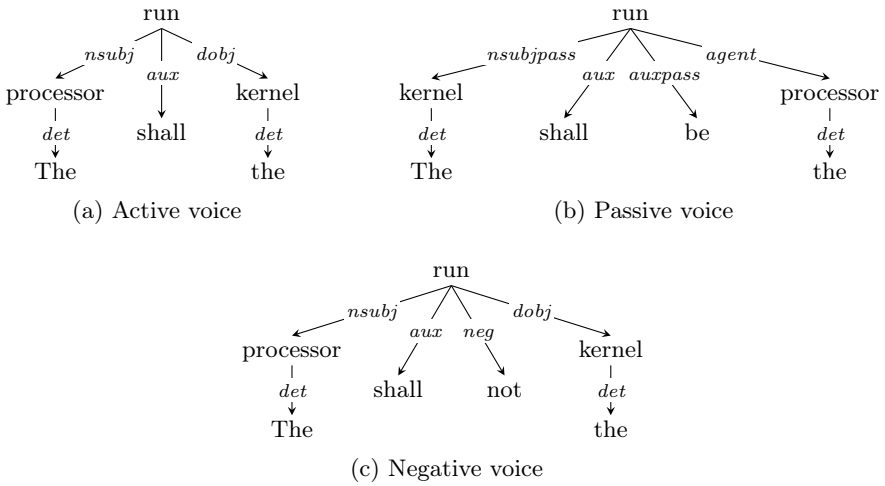


Fig. 1.4. Active, passive, and negative voice

To check whether case one or two are respected or not, one can look through the phrase structure tree of the sentence. If it contains a coordination ("CC") which has two siblings and a parent of type verb ("S*") or sentence ("V*"), respectively (Fig. 1.3a and 1.3a). However this does not work with case three since this tree doesn't give the dependency of a noun phrase to its parent to decide if it is a subject or not. Fortunately, This could be done by inspecting the dependency graph of a requirement. To do so, one can check if a subject is related to another noun phrase by a conjunction or not. Consider the dependency graph of the requirement given in S3, the two relations nsubj(run, processor_1) and conj(processor_1, processor_2) indicate that the requirement has two subjects. hence the rule is violated. Hence this rule is respected when the dependent of the relation '*nsubj*' is not a governor of a '*conj*' relation.

1.4.3 Rule 3: Use simple direct sentences

Here, one heuristic to use is to check for active, passive, and negative voice in sentence. If the sentence is given in passive or negative voice, we can determine that the rule has been violated. Note that the contrary is not necessarily true. By making use of typed dependencies it can easily be checked whether a sentence is given in active, passive, or negative voice since different relations are found in the corresponding typed dependency graphs. While the subject is indicated as the dependent of an '*nsubj*' relation in a sentence in active voice (cf. Fig. 1.4a), the relation will be '*nsubjpass*' when using passive voice (cf. Fig. 1.4b). On the other hand when the sentence is in negative voice, there is a relation '*neg*' (cf. Fig. 1.4c).

But it cannot only be checked rather easy whether the rule is violated by inspecting if such dependency relations occur in the sentence. −> I don't understand that !!!

(a) Normal sentence (b) Requirement sentence

Fig. 1.5. Requirement format

Passive sentences can also be translated automatically using NLP techniques.[1]

1.4.4 Rule 4: Each requirement has a subject and a predicate

Each requirement provides the specifics of a desired end goal or result. For that reason it must respect the flowing form:

```
<subject> shall/must/will <predicate>
```

- *Subject*: a user or the system under discussion.
- *Predicate*: a condition, action, or intended result.
- *Verb*: {shall, will, must} to show mandatory nature or {may, should} to show optionality.

Consider the typed dependencies of the requirements in Fig. 1.5a and 1.5b, where the first is violating the rule and the second is respecting it. As it is clearly seen, when the requirement has the form specified above there exist a relation '*aux*' between the main verb (run) and an auxiliary verb (shall). Besides that, the dependant of this relation (the auxiliary verb) belongs to this set {shall, will, must, may, should}.

1.4.5 Rule 5: Avoid let-out clauses

The let-out clauses generate ambiguities and misunderstanding, then errors in the final products. In order to avoid having these complications, it is important to not use such clauses. To check the violation of a requirement against this rule, we have transformed each requirement as well as the set of let-out clauses to lower-case form. Then we search the existence of each clause in the requirement. The rule is violated once a clause is found.

Since rules 6, 7, 8, and 9 have the same type as the current rule, they are implemented the same way as this rule.

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

Table 1.1. Experimental results

Rules	Manual Class.		Automatic Class.		Classifier Evaluation					
	T	F	T	F	SA	TP	TN	FP	FN	Acc.
R1	88	15	61	42	70	58	12	30	3	67.96%
R2	84	19	89	14	80	75	5	9	14	77.67%
R3	90	13	89	14	86	81	5	9	8	83.50%
R4	102	1	92	11	93	92	1	10	0	90.29%
R5	94	9	95	8	102	94	8	0	1	99.03%
R6	92	11	85	18	92	83	9	9	2	89.32%
R7	102	1	103	0	102	102	0	0	1	99.03%
R8	103	0	103	0	103	103	0	0	0	100.00%
R9	9	17	18	8	13	7	6	2	11	50.00%
Total	826	127	772	181	811	728	83	98	44	82.48%

1.4.6 Rule 10: Define verifiable criteria

In this rule, we have considered three annotations:

1. *Respected*: when a requirement does contain a numeric value or has an element of the set {"high", "low", "immediately"}, then the rule is respected.
2. *None*: when a requirement does contain a numeric value and an element of the flowing set {"minimum", "minimal", "maximum", "maximal", "approximately", "different", "equal", "big", "bigger", "small", "smaller", "bigger", "less", "cheaper", "lower", "more", "much"}, then we can not say weather it is verifiable or not therefore we assign it to none.
3. *Violated*: When both of the two condition above are not satisfied, the rule is considered as it is violated.

To check for the existence of a numeric value into the requirement, we defined a regular expression representing a number and search for its existence in the requirement, while to find an element from one of the two sets, we have used the same mechanism as in rule 5.

1.5 Experimental Evaluation

The proposed approach has been implemented in Scala based on the Stanford CoreNLP library. We evaluated our approach using the same set of requirements as for the static approach in the previous section and manually annotated them according to the given rules. The manual annotations were then compared to the results of the classifier.

1.5.1 Evaluation of a Random Set of Requirements

We applied the algorithm to a test set of requirements that have been extracted from various specifications [7, 10, 16]. The test set contains 103 different requirements.

Table 1.2 summarizes the obtained results for the conducted experiments. The first column gives the rules as defined in this section. In the following columns, the

Table 1.2. Experimental results

Rules	Manual Class.		Automatic Class.		Classifier Evaluation					
	T	F	T	F	SA	TP	TN	FP	FN	Acc.
R1	52	32	54	30	76	49	27	3	5	90.48%
R2	68	16	35	49	45	32	13	36	3	53.57%
R3	31	53	19	65	62	14	48	17	5	73.81%
R4	76	8	76	8	80	74	6	2	2	95.24%
R5	77	7	75	9	80	74	6	3	1	95.24%
R6	65	19	61	23	78	60	18	5	1	92.86%
R7	82	2	80	4	82	80	2	2	0	97.62%
R8	84	0	84	0	84	84	0	0	0	100.00%
R9	26	8	27	7	19	19	0	7	8	55.88%
Total	589	201	541	249	680	510	170	79	31	84.28%

respective annotations for the manual classification (Manual Class.), the results for the automated classification (Auto. Class.), and the concluded results for the classifier evaluation are presented. T, F, SA, TP, FP, TN, and FN refer to the number of *true annotated*, *false annotated*, *same annotated*, *true positives*, *false positives*, *true negatives*, and *false negatives*, respectively. The last column represents the accuracy (Acc.) computed by $\frac{TP+TN}{TP+TN+FP+FN}$.

The accuracy reaches more than 80% for the majority of the rules and 99% for *R5*, *R8*, and *R9*. Our rule based tool has a significant performance in general since it has an average accuracy of 82.48%.

1.5.2 Evaluation of Infenion's Set of Requirements

We applied the algorithm to a test set of requirements that have been extracted from Infenion's component specifications. The selected requirements represent real product requirements provided by a stakeholder, used for the concept and the design of a system. The requirements are expressed in natural language, in English. They are not respecting any specific recommendations for writing requirements in natural language. This is an intentional choice so that it is possible to evaluate the algorithm capability to check both compliance and non-compliance with a given rule.

The majority of the requirements are functional requirements, stating how the system must behave:

- What inputs the system requires for its functionality?
- What outputs the system shall produce?
- What data the system should store?
- What computation the system should perform?
- What are the timing and synchronization characteristics.

The analysis was performed on approximately 80 requirements, which are not respecting the majority of the rules. The requirements are carefully chosen so that there is a balance between the number of requirements that respect a given rule and the ones that do not. Next, there are presented some examples of requirements and some typical issues.

Table 1.2 summarizes the obtained results for the conducted experiments. The accuracy of most of the rules is higher than 90% and reaches 97% and 100% for *R7* and *R8*, respectively. Our rule based tool has a significant performance in general since it has an average accuracy of 84.28%.

Remarks and Observations

One common issue is the atomicity of the requirements. The initial requirements identification was not done based on their atomicity. In the majority of cases, multiple functional characteristics were grouped together and considered only as one requirement. This translates into more than one atomic requirement, for 70% out of the entire set of requirements. Let us take the following example:

> **The chip shall be able to enter in test mode if the Pin15 pin is set to high and thereafter 3 SPI_in commands are received in the correct sequence. If the sequence is not respected, the test mode will be inhibited and locked for the rest of the ignition cycle. If the Pin15 pin is set to low all test modes are immediately disabled.**

It has been tagged as one requirement expressing the functionality of the chip for the test mode situation. By splitting this into atomic requirements, we can identify three main requirements, based on their meaning as well as on the paragraph structure, which includes 3 phrases, each one representing a separate requirement. The number of phrases from a paragraph is an indication that one phrase can mean one requirement, but this is not compulsory or mandatory for all situations.

The information expressed in a requirement needs to be correct and precise. In natural language this translates into avoiding constructions that can lead to misinterpretations, wrong assumptions or contradictions. This includes let-out-clauses, expression that suggest possibility, undefined, unquantifiable or speculative terms, as described by rules 5 to 9. The majority of the requirements are respecting these rules.

A specific problem was identified while analysing let-out clauses that use conjunctions such as *"unless"*, *"when"*, and *"although"*. A clause is a sentence-like construction contained within a sentence, which may represent an added condition or context to the given requirement. However, in the majority of situations, *when* was not used for introducing a separate requirement, but to introduce a condition which is specific for the situation described in the requirement. For example consider the following requirement:

> **The module shall become active when the following conditions are true: Vbat is supplied.**

"when" is used instead of an *"if"* condition. This double use of the conjunctions makes the automatic detection of problems even more difficult.

Another issue with the *"if"* clauses is that they usually require an *"else"* or *"otherwise"* clause which is not always specified. In this case, a default behaviour is assumed by the reader of the requirements. However, the default behaviour can

be assumed to be different by different people depending on their knowledge, background, context, etc., hence it leads to potential errors in the implementation. For example, consider the requirement in the following:

> **If the Pin15 pin is set to low all test modes are immediately disabled.**

It is clearly stated what should happen in the mentioned situation. What is not stated is what happens if the pin is set to high (which is the complementary case). Either the situation is mentioned in another requirement (in the best situation), otherwise if nothing is mentioned it is assumed that nothing changes from the current situation.

The selected requirements are part of a safety-related mechanism of a system. Because of this, internal characteristics describe a specific mode for the system, which is named safe-mode. Unfortunately for this situation, the word safe is considered as a *"wishful thinking"* for rule nine. This is the reason why the system identifies the requirements describing the safe-mode as incorrectly expressed, despite their correctness.

1.6 Conclusions

We have presented two automatic approaches that assist the designer in writing better requirements in specifications by (i) checking syntactic and semantic properties of the requirement, and (ii) validating the requirement with respect to rules from a guideline specification. For a selected set of typical rules it has been shown that our methods work effectively. For future work a thorough case study should further evaluate the practicability of our approaches. Also, it should be investigated which rules are suitable for automatic fixing.

References

1. Ian F Alexander and Richard Stevens. *Writing better requirements*. Pearson Education, 2002.
2. ClearSpecs. Tekchecker. Available at http://clearspecs.com.
3. Sunny Hills Consultancy. Requirements assistant. Available at http://www.requirementsassistant.nl/.
4. Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *Conf. on Language Resources and Evaluation*, pages 449–454, 2006.
5. Arthur C. Graesser, Danielle S. Mcnamara, and Jonna M. Kulikowich. Coh-metrix: Providing multilevel analysis of text characteristics. In *Educational Researcher*, volume 40, pages 223–234, 2011.
6. Ian G. Harris. Extracting design information from natural language specifications. In *DAC*, pages 1256–1257, 2012.
7. Peter R. Harvey. (NASA) flight software requirements, 2010. Available at ftp://apollo.ssl.berkeley.edu/pub/RBSP/1.1.%20Management/5%20 Meetings/PhaseB_080902_IPDR/Documents/ RBSP_EFW_FSW_002_Requirements.pdf.
8. IBM. Get it right the first time: Writing better requirements, 2009. Available at http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help .download.doors.doc/pdf92/get_it_right_the_first_time.pdf.
9. Nitin Indurkhya and Fred J. Damerau. *Handbook of Natural Language Processing*. Chapman & Hall/CRC, 2nd edition, 2010.
10. Intel. Intel active management technology (Intel AMT) 7.0 release : Fw & sw product requirements document (PRD), 2010. Available at http://www.intel.de/content/dam/www/public/us/en/documents/product-specifications/amt-7-0-release-fw-sw-prd.pdf.
11. Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Pearson Prentice Hall, 2008.
12. Nadzeya Kiyavitskaya, Nicola Zeni, Luisa Mich, and Daniel M Berry. Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements Engineering*, 13(3):207–239, 2008.
13. Giuseppe Lami. Quars: A tool for analyzing requirements. Technical report, DTIC Document, 2005.

14. Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL: System Demonstrations*, pages 55–60, 2014.
15. George A. Miller. WordNet: a lexical database for English. *Commun. ACM*, 38(11):39–41, 1995.
16. Tom Morgan. Requirements and functional specification : Evla correlator backend, 2003. Taken from National Radio Astronomy Observatory, available at http://www.aoc.nrao.edu/evla/techdocs/computer/workdocs/BE_rfs_1.pdf.
17. Mathias Soeken, Christopher B. Harris, Nabila Abdessaied, Ian G. Harris, and Rolf Drechsler. Automating the translation of assertions using natural language processing techniques. In *Forum on Specification & Design Languages*, 2014.
18. Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *TOOLS*, pages 269–287, 2012.
19. William Wilson. Writing effective requirements specifications. In *Software Technology Conference*, 1997. Available at NASA Software Assurance Technology Center (SATC) http://www.csc.kth.se/utbildning/kth/kurser/DD1363/NASARequirements.html.