Automated Checking of Conformance to Requirements Templates Using Natural Language Processing

Chetan Arora, Mehrdad Sabetzadeh, *Member, IEEE*, Lionel Briand, *Fellow, IEEE*, and Frank Zimmer, *Member, IEEE*

Abstract—Templates are effective tools for increasing the precision of natural language requirements and for avoiding ambiguities that may arise from the use of unrestricted natural language. When templates are applied, it is important to verify that the requirements are indeed written according to the templates. If done manually, checking conformance to templates is laborious, presenting a particular challenge when the task has to be repeated multiple times in response to changes in the requirements. In this article, using techniques from natural language processing (NLP), we develop an automated approach for checking conformance to templates. Specifically, we present a generalizable method for casting templates into NLP pattern matchers and reflect on our practical experience implementing automated checkers for two well-known templates in the requirements engineering community. We report on the application of our approach to four case studies. Our results indicate that: (1) our approach provides a robust and accurate basis for checking conformance to templates; and (2) the effectiveness of our approach is not compromised even when the requirements glossary terms are unknown. This makes our work particularly relevant to practice, as many industrial requirements documents have incomplete glossaries.

Index Terms—Requirements templates, natural language processing (NLP), case study research

1 Introduction

Natural language (NL) is arguably the most common way for specifying requirements. NL tends to be easier to understand for many stakeholders as it requires little training. NL further has the advantage of being universal, i.e., it can be used in any problem domain with the flexibility to accommodate arbitrary abstractions [1]. Despite these advantages, NL is prone to ambiguity and without enforcing restrictions, NL can be hard to analyze automatically.

Templates, also known as boilerplates, molds, or patterns [1], [2], provide an effective tool for reducing ambiguity in NL requirements and for making these requirements more amenable to automated analysis [3], [4], [5], [6]. A template organizes the syntactic structure of a requirements statement into a number of pre-defined slots. Fig. 1 shows one of the well-known requirements templates, due to Rupp [5]. The template envisages six slots: (1) an optional condition at the beginning; (2) the system name; (3) a modal (shall/should/will) specifying how important the requirement is; (4) the required processing functionality; this slot can admit three different forms based on the manner in which the functionality is to be rendered (explained later in Section 2.1); (5) the object for which the functionality is needed; and (6) optional additional details about the object.

We show in Fig. 2 three example requirements: R_1 and R_2 conform to Rupp's template; whereas R_3 does not. For R_1 and R_2, we show the different sentence segments and how they correspond to the slots in Rupp's template. The fixed elements of the template are written in capital letters.

When templates are used, an important quality assurance task is to verify that the requirements conform to the templates. If done manually, this task can be time-consuming and tedious [5]. Particularly, the task presents a challenge when it has to be repeated multiple times in response to changes in the requirements. When the requirements glossary terms (domain keywords) are known, checking conformance to templates can be automated with relative ease. For example, consider R_1 in Fig. 2 and assume that "Surveillance and Tracking module" (system component), "system administrator" (agent), "monitor" (domain verb), and "system configuration change" (event) have been already declared as glossary terms. In this situation, an automated tool can verify conformance by checking that R_1 is composed of a subset of the glossary terms (or terms with close syntactic resemblance to the glossary terms) and a subset of fixed template elements, put together in a sequence that is admissible by the template. The fixed elements may be leveraged for distinguishing different template slots. For example, whatever appears between `PROVIDE` and `WITH THE ABILITY TO` in R_1 has to correspond to the ⟨whom?⟩ (sub)slot.

This approach does not work when the glossary terms are unknown, because it can no longer distinguish sentence segments that correspond to the template slots. For example, in Rupp's template, ⟨process⟩, ⟨object⟩, and ⟨additional details⟩ come in a sequence without any fixed elements in

- *C. Arora, M. Sabetzadeh, and L. Briand are with the SnT Centre for Security, Reliability, and Trust, University of Luxembourg, Luxembourg L-2721. E-mail: {chetan.arora, mehrdad.sabetzadeh, lionel.briand}@uni.lu.*
- *F. Zimmer is with the SES TechCom, Luxembourg. E-mail: frank.zimmer@ses.com.*

Manuscript received 12 Nov. 2014; revised 24 Mar. 2015; accepted 25 Apr. 2015. Date of publication 30 Apr. 2015; date of current version 16 Oct. 2015. Recommended for acceptance by R.R. Lutz.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2428709

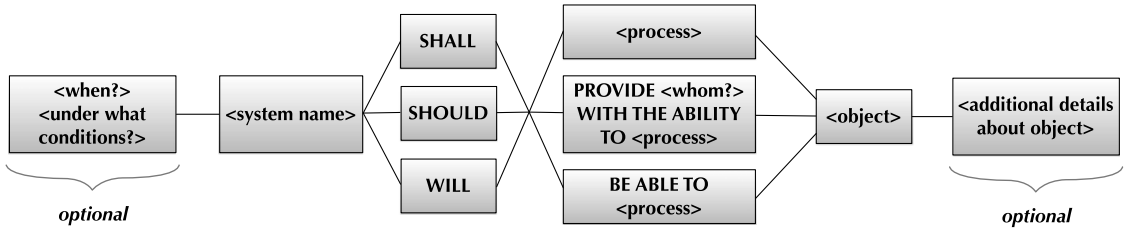

Fig. 1. Rupp's template [5].

between. For an automated tool to deem R_1 as conformant, it has to correctly distinguish these three slots in the following: "monitor system configuration changes posted to the database". A second important issue is that even when a slot falls in between fixed elements, e.g., ⟨whom?⟩, and is thus easy to delineate, there is no way to distinguish an acceptable filler for the slot from an unacceptable one, e.g., a grammatically-incorrect phrase. For instance, in R_1, we may accept "system administrator" as correctly filling ⟨whom?⟩, but we may be unwilling to accept a grammatically-incorrect phrase such as "system administer" for the slot.

1.1 Motivation and Contributions

While little empirical evidence exists on the efficacy and cost-effectiveness of requirements templates, there are credible indicators suggesting that requirements templates offer benefits in certain, if not most, industrial settings. Practitioner-oriented RE textbooks, e.g., [1], [7], recommend templates as one of the main strategies for reducing ambiguity and making NL requirements easier to analyze. Using templates is particularly common in safety-critical domains where having structured and unambiguous requirements is paramount. Several projects concerned with the development and certification of safety-critical systems, e.g., CESAR [8], OPENCOSS [9] and SAREMAN [10], recommend the use of templates for more precise specification of requirements.

Support for requirements templates now exists in some commercial requirements authoring and management tools [11], indicating a general industrial interest in templates and, naturally, the quality assurance activities surrounding templates. For example, the RQA tool [12], provides features for automatically verifying correct use of templates; however, the effectiveness of the automation is adversely affected when the glossary terms are left unspecified. While building a glossary is an essential activity in any requirements project, the glossary is not necessarily available at the time one wants to check conformance to templates. In fact, based on our experience [13], practitioners tend to identify and define the glossary terms *after* the requirements have sufficiently matured and are thus less likely to change. This strategy avoids wasted effort at the glossary construction stage, but it also means that the glossary will not be ready in time to support activities such as template conformance checking (TCC), which often take place *during* requirements writing. Furthermore, the literature suggests that glossaries may remain incomplete throughout the entire development process [14], thus providing only partial coverage of the glossary terms. This implies that automated techniques for checking conformance to templates would have limited effectiveness if such techniques rely heavily on the glossary terms being known a priori.

This article is motivated by the need to provide an automated and generalizable solution for template conformance checking *without* reliance on a glossary. To this end, we make the following three contributions:

- We propose an approach for the automation of TCC using natural language processing (NLP). The main enabling NLP technology used in our approach is *text chunking*, which identifies sentence segments (chunks) without performing expensive analysis over the chunks' internal structure, roles, or relationships [15]. These chunks, most notably noun phrases (NPs) and verb phrases (VPs), provide a suitable level of abstraction over natural language for characterizing template slots and performing TCC (Section 3). Our approach further utilizes NLP parsing when text chunking alone cannot conclusively determine template compliance, e.g., when requirements analysts elect to fill the template slots with complex noun phrases that include verb phrases in their makeup.
- We report on four case studies conducted in order to evaluate our approach. Two of these studies involve Rupp's template, discussed above, and the other two involve another well-known template, called EARS

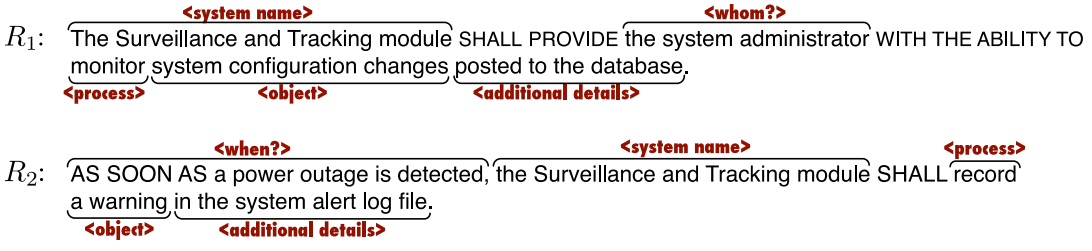

Fig. 2. Example requirements: R_1 and R_2 conform to Rupp's template but R_3 does not. The fixed elements of the template are written in capital letters.

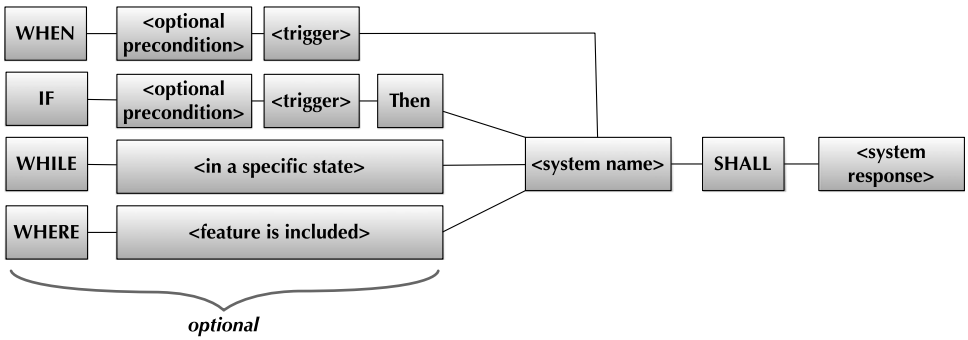

Fig. 3. The EARS template [16].

[16], [17], [18] and discussed in Section 2.1. In both of the studies that use Rupp's template, the requirements were written directly by professionals. As for the two studies using EARS, one—which is the largest case study reported in this article—was written directly by professionals as well, while the remaining study uses transformed requirements from one of our two case studies based on Rupp's template. The results from our case studies indicate firstly, that our approach provides an accurate basis for TCC; and secondly, that the effectiveness of our approach is not compromised even when the glossary terms are unspecified.

- We provide tool support for TCC. Our tool, named REquirements Template Analyzer (RETA) enables analysts to automatically check conformance to both Rupp's and the EARS templates, and to obtain diagnostics about potentially problematic syntactic constructs in requirements statements.

This article extends a previous conference paper [19] published at the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13) and a workshop paper [13] at the 4th International Workshop on Requirements Patterns (RePa'14). The article further reports on the tool we have developed in support of our approach. An earlier version of our tool (supporting only Rupp's template at the time) was demonstrated [20] at the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13).[1]

We present in this article a definitive treatment of our approach by bringing together and extending ideas from the above papers. The article further provides substantial new empirical evidence to support the effectiveness of our approach, complementing the single case study in our previous work [19] with three new case studies. Finally, we provide a more detailed examination of the alternative NLP solutions that one can use to realize our approach, with a four-fold increase in the number of alternatives considered when compared to our previous work.

1.2 Structure

The remainder of the article is structured as follows: Section 2 provides background information on requirements templates and NLP to the extent needed in our approach. Section 3 describes our approach for automation of TCC. Section 4 discusses tool support. Sections 5 presents the case studies we have conducted to evaluate our approach. Section 6 identifies the limitations and analyzes threats to validity. Section 7 compares our approach with related work. Section 8 concludes the article with a summary and directions for future work.

2 Background

2.1 Requirements Templates

When properly followed, templates serve as a simple and yet effective tool for increasing the quality of requirements by avoiding complex structures, ambiguity, and inconsistency in requirements. Templates further facilitate automated analysis by making NL requirements more easily transformable into analyzable artifacts, e.g., models [5].

Several templates have been proposed in the requirements engineering literature. While our approach can be tailored to work with a variety of existing templates, we ground our work in this article on two templates, namely Rupp's and the EARS templates [5], [16]. Our choice is motivated by the reported use of these templates in the industry [7], [18], [21] and the availability of practitioner guidelines for the templates, e.g., [5], [22]. These guidelines present an advantage for training purposes and introducing templates in production environments.

Rupp's template, shown in Fig. 1, was already introduced in Section 1 except for the different forms that the processing functionality slot (i.e., the template's fourth slot) can assume. Rupp's template distinguishes three types of processing functionality:

- *Autonomous requirements*, captured using the "⟨process⟩" form, state functionality that the system offers independently of interactions with users.
- *User interaction requirements*, captured using the "`PROVIDE` ⟨whom?⟩ `WITH THE ABILITY TO` ⟨process⟩" form, state functionality that the system provides to specific users.
- *Interface requirements*, captured using the "`BE ABLE TO` ⟨process⟩" form, state functionality that the system performs to react to trigger events from other systems.

The EARS[2] template, shown in Fig. 3, is made up of four slots: (1) an optional condition at the beginning; (2) the

1. We note that in these previous papers, we used the term *requirements boilerplate* instead of *requirements template*.

2. EARS stands for the Easy Approach to Requirements Syntax.

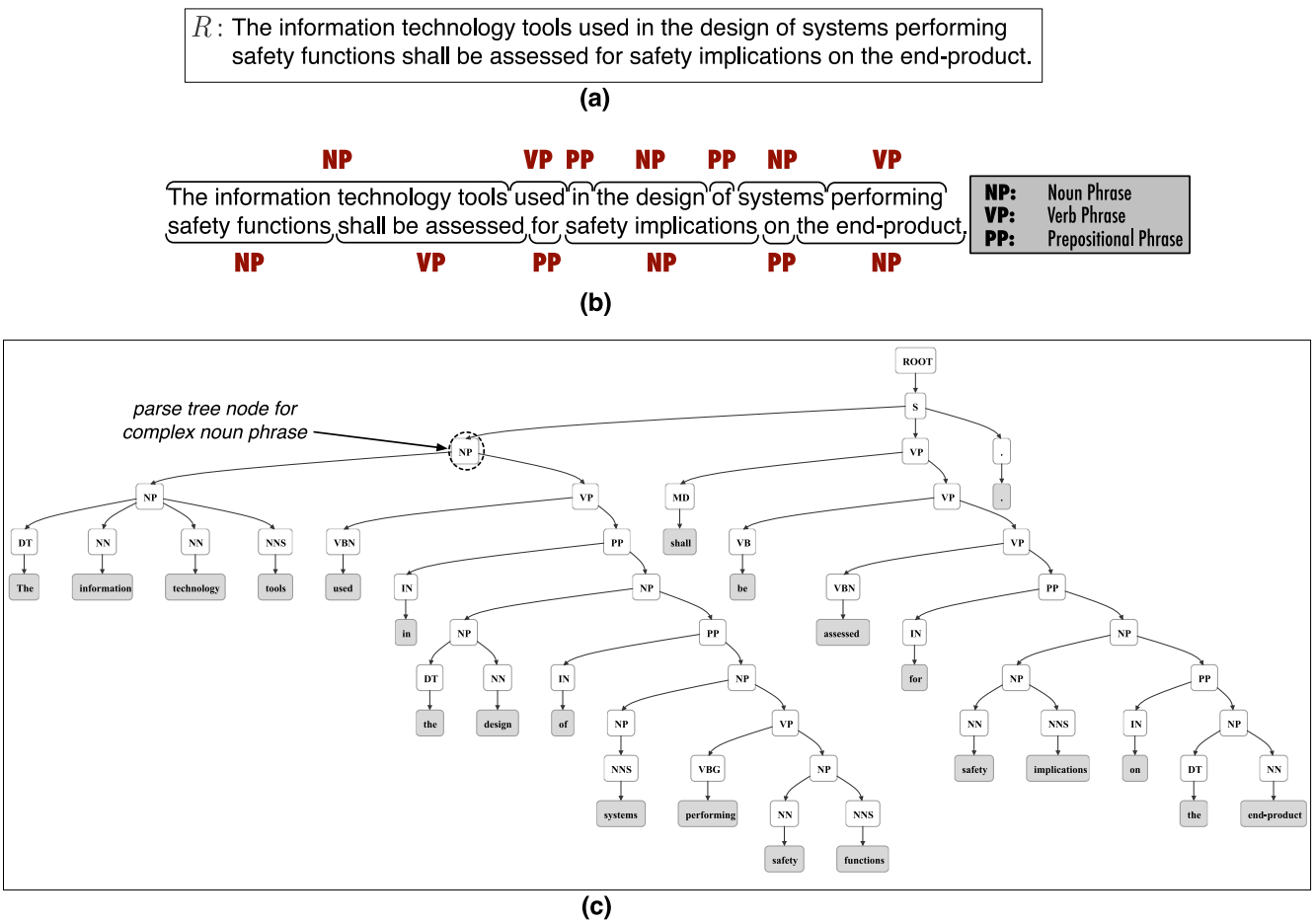

Fig. 4. (a) An example requirements statement, (b) its sentence chunks, and (c) its full parse tree.

system name; (3) a modal; and (4) the system response depicting the behavior of the system. EARS distinguishes five requirements types using five alternative structures for its first slot [16]:

- *Ubiquitous requirements* have no pre-condition, and are used for requirements that are always active.
- *Event-driven requirements* begin with `WHEN` and are used for requirements that are initiated by a trigger event.
- *Unwanted behavior requirements* begin with `IF` followed by `THEN` before the ⟨system name⟩. These requirements are usually used for expressing undesirable situations.
- *State-driven requirements* begin with `WHILE` and are used for requirements that are active in a definite state.
- *Optional feature requirements* begin with `WHERE` and are used for requirements that need to be fulfilled when certain optional features are present.
- *Complex requirements* use a combination of the above patterns, i.e., more than one type of condition.

Compared to Rupp's, the EARS template offers more advanced features for specifying conditions. In contrast, Rupp's template enforces more structure than EARS over the non-conditional parts of requirements sentences, particularly by requiring an object (denoted by the ⟨object⟩ slot) to always be present.

2.2 Text Chunking

Text chunking is the core NLP technology underlying our approach. Briefly, text chunking is the process of decomposing a sentence into non-overlapping segments [15]. The main segments of interest are noun phrases and verb phrases. A *noun phrase (NP)* is a segment that can be the subject or object of a verb. A *verb phrase (VP)*, sometimes also called a verb group, is a segment that contains a verb with any associated modal, auxiliary, and modifier (often an adverb). For example, a correct chunking of the requirements statement R in Fig. 4a would yield the segments shown in Fig. 4b. As seen from this figure, segments generated by text chunking have a flat structure. This is in contrast to segments in a parse tree—generated by a natural language parser such as the Stanford Parser [23]—which can have arbitrary depths, as shown in Fig. 4c.

When a parse tree is not required for analysis, text chunking offers two major advantages over parsing [24]: First, text chunking is computationally less expensive, having a complexity of $O(n)$, with n denoting the length of a sentence, versus $O(n^3)$ for (rule-based) parsing. This lower complexity makes text chunking more scalable. Scalability is an important consideration in our context where we need to deal with large requirements documents. Second, text chunking is more robust than parsing [25], in the sense that it produces results in the large majority of cases; whereas parsing may fail to generate a parse tree, particularly when faced with unfamiliar input. Similar to scalability,

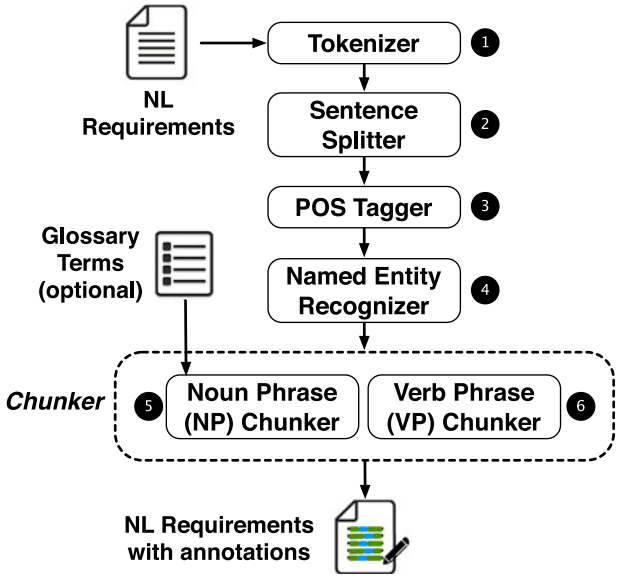

Fig. 5. NLP pipeline for text chunking.

robustness is an important consideration in our context, noting that requirements documents are highly technical and can deviate from commonly-used texts for training parsers, e.g., news articles.

While the above considerations make text chunking better suited to our context, the lack of a parse tree means that text chunking cannot reveal the complete semantics of requirements statements. In particular, text chunking cannot identify complex phrases, and would instead find only the atomic phrases that make up complex ones. For example, the sentence segment "The information technology tools used in the design of systems performing safety functions" in the example of Fig. 4 is a complex noun phrase. The chunks in Fig. 4b capture only the atomic phrases of this complex noun phrase; whereas, the nodes in the parse tree of Fig. 4c further capture the complex noun phrase in its entirety, as marked on the figure. We discuss the implications of complex phrases for TCC in Section 3.3. Below, we describe how a text chunker is implemented in an NLP environment.

A text chunker is a pipeline of NLP modules running in a sequence over an input document. The (generic) pipeline for chunking is shown in Fig. 5. As we explain in Section 5.4, this pipeline can be instantiated in many ways, as there are alternative implementations for each step in the pipeline. The first module, the Tokenizer, breaks up the input into tokens. A token can be a word, a number or a symbol. The Sentence Splitter divides the text into sentences. The POS Tagger annotates each token with a part-of-speech tag. These tags include among others, adjective, adverb, noun, verb. Most POS Taggers use the Penn Treebank tagset [26]. Next is the Name Entity Recognizer, where an attempt is made to identify named entities, e.g., organizations and locations. In a requirements document, the named entities can further include domain keywords and component names. The main and final step is the actual Chunker. Typically, but not always, NP and VP chunking are handled by separate modules, respectively tagging the noun phrases and verb phrases in the input. When a glossary of terms is available, one can instruct the

```
1. Phase: MarkConformantSegment
2. Input: Condition NP VP Token
3. Options: control = appelt
4.
5. Rule: MarkConformantSegment_RuppAutonomous
6.  (
7.     (({Condition}({NP}):system_with_condition) |
8.      ({NP}): system_without_condition)
9.     ({VP, VP.startsWithValidMD == "true",
10.     !VP contains {Token.string == "provide"}}):process
11.    ({NP}):object
12.  ):label
13. -->
14.  :label.Conformant_Segment =
15. {explanation = "Matched pattern: Autonomous"},
16. :system_with_condition.System_Name = {},
17. :system_without_condition.System_Name = {},
18. :process.Process = {},
19. :object.Object = {}
```

Fig. 6. JAPE script for Rupp's autonomous type.

NP Chunker to treat occurrences of the glossary terms in the input as named entities, thus reducing the likelihood of mistakes by the NP Chunker. To what extent the glossary is useful for TCC is the subject of RQ2 (see Section 5.6). Once processed by the pipeline of Fig. 5, a document will have annotations for tokens, sentences, parts-of-speech, named entities, noun phrases, and verb phrases. We use these annotations for automating TCC, as we explain in Section 3.

2.3 Pattern Matching in NLP

As we elaborate in Section 3, we represent templates as BNF grammars. This representation enables the definition of pattern matching rules for checking template conformance. For implementing a BNF grammar over NL statements, we use JAPE (Java Annotation Patterns Engine). JAPE is a regular-expression-based pattern matching language, available as part of the GATE NLP workbench [27]. Fig. 6 shows an example JAPE script, which checks conformance to Rupp's *Autonomous* requirements type.

Each JAPE script consists of a set of phases, with each phase made up of a set of rules. In the script of Fig. 6, we have a single phase, named *MarkConformantSegment* (line 1), which includes a single rule, named *MarkConformantSegment_RuppAutonomous* (line 5). The phase could be extended with rules for checking conformance to Rupp's *User Interaction* and *Interface* requirements types. Each rule in JAPE consists of a left hand side (LHS) and a right hand side (RHS), which are separated by –> (line 13). The LHS specifies the annotation pattern that needs to be matched (lines 6-12), and the RHS—the action to be taken when a match is found (lines 14-19). The LHS in the rule of Fig. 6 matches the pattern for Rupp's *Autonomous* requirements type. The corresponding action in the RHS annotates as `Conformant_Segment` the entire segment matching the pattern (lines 14-15). The RHS has further actions for delineating `System_Name`, `Process`, and `Object` (lines 16-19). Note that `System_Name` needs to be recognized both in the presence and absence of a condition. Two temporary annotations, namely `system_with_condition` (line 7) and `system_without_condition` (line 8) have been defined in the LHS to enable detection of `System_Name` in both cases. The RHS of a JAPE rule can optionally contain Java

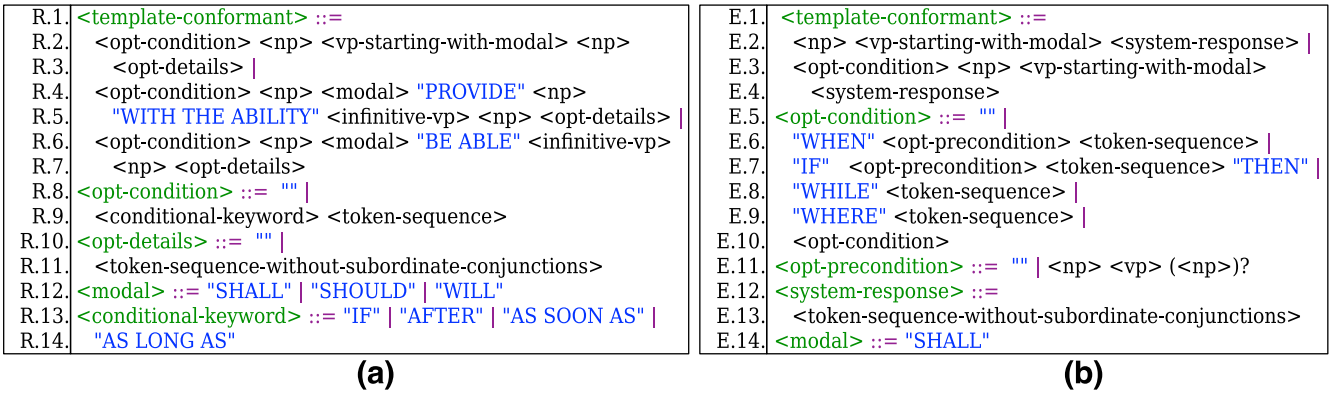

Fig. 7. BNF grammars for (a) Rupp's template and (b) the EARS template.

code for manipulating the annotations. An example of an RHS with embedded Java code can be seen in Fig. 10.

JAPE provides various options for controlling the results of annotations when multiple rules match the same section in text, or for controlling the text segment that is annotated on a match. These options are *brill*, *appelt*, *all*, *first*, and *once* [27]. In our work, we make use of *brill*, *appelt* and *first*. *Brill* means that when more than one rule matches the same region of text, all of the matching rules are fired, and the matching segment can have multiple annotations. This is useful, for example, while detecting potential ambiguities in a requirements sentence: if multiple ambiguities are present in a given text segment, all the ambiguities will be annotated. *Appelt* means that when multiple rules match the same region, the one with the maximum coverage (i.e., longest text segment) is fired. This can be used, for example, as shown in Fig. 6, where we want to assign a unique type to a requirement that is most appropriate. *First* means that when there is more than one rule matching, the first one matching is fired without trying to get the longest sequence. This is useful, for example, when delineating sentences. To delineate a sentence, we need to match a sequence of tokens followed by a full stop. Using *first* enables us to correctly match individual sentences; whereas using *appelt* would result in matching the longest possible sequence, i.e., entire paragraphs.

3 Approach

In this section, we describe how to use the annotations produced by (an instantiation of) the text chunking pipeline in Fig. 5 for automating TCC. Specifically, we show how Rupp's and the EARS templates can be represented as BNF grammars over the annotations resulting from text chunking and then verified automatically in an NLP framework. Rupp's and the EARS templates are the basis for the case studies in Section 5.

3.1 Expressing Templates as Grammars

In Figs. 7a and 7b, we provide the BNF grammars for Rupp's and the EARS templates in terms of the annotations generated by the pipeline of Fig. 5. For simplicity, in the grammars, we abstract from nested tags. For example, on line R.2, we use ⟨vp-starting-with-modal⟩ to denote a verb phrase (⟨vp⟩) that contains a modal at its starting offset. Similarly ⟨infinitive-vp⟩ denotes a ⟨vp⟩ starting with "to".

In both templates, requirements can start with an optional condition. Rupp's template does not provide any detailed syntax for the conditions, recommending only the use of the following conditional key-phrases: `IF` for logical conditions; and `AFTER`, `AS SOON AS`, and `AS LONG AS` for temporal conditions. EARS, in contrast, differentiates the types of requirements by the ⟨opt-condition⟩ rule in E.5—E.10. E.10 captures the *complex* requirements type (Section 2.1) in EARS by the use of recursion in the ⟨opt-condition⟩ slot.

A restriction that can be made in both templates is for the conditional segment to always end with a comma (","). This restriction may however be too constraining because commas can be easily forgotten or applied according to one's personal preferences for punctuation. To avoid relying exclusively on the presence of a comma, one can employ heuristics for identifying the conditional segment in a requirement. In particular, one can use the system name (an NP) followed by a modal (e.g., SHALL) as an anchor for identifying the conditional part. For example, consider the following requirement $R =$ "When a GSI component constraint changes STS SHALL deliver a warning message to the system operator". In Rupp's template, the heuristic capturing the syntax of R is ⟨conditional-keyword⟩ ⟨sequence-of-tokens⟩ ⟨np⟩⟨vp-starting-with-modal⟩⟨np⟩⟨opt-details⟩.

Template-specific keywords, e.g., the modals and the conditional keywords, are grouped into keyword lists, often called gazetteers [27] in NLP. These lists decouple TCC rules from template-specific keywords, thus avoiding the need to change the rules when the keywords change.

Lastly, for the optional details in Rupp's and the system response in the EARS template, we accept any sequence of tokens as long as the sequence does not include a subordinate conjunction (e.g., after, before, unless). The rationale here is that a subordinate conjunction is very likely to introduce additional conditions. Both Rupp's and the EARS templates envisage that such conditions must appear at the beginning and not the end of requirements statements. Checking conformance to the rules in Fig. 7 can be implemented using NLP pattern matching as we describe next.

3.2 Conformance Checking via Pattern Matching

TCC starts with the text chunking pipeline, shown and discussed in Section 2.2. Text chunking identifies Tokens

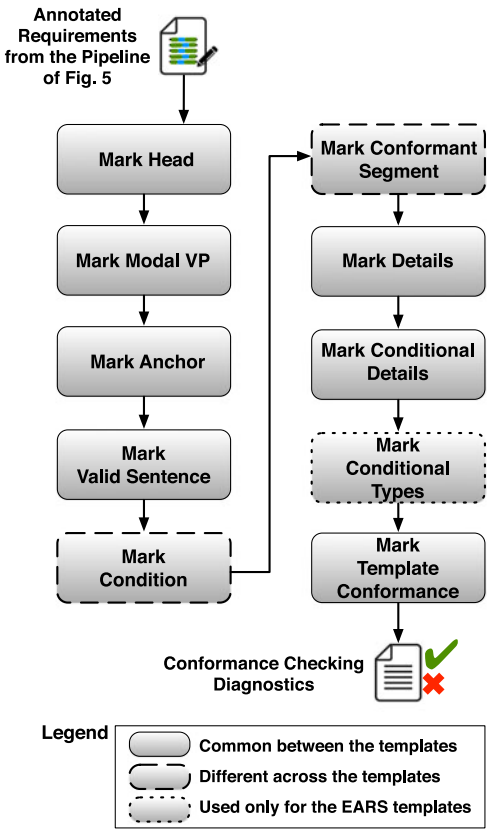

Fig. 8. Pipeline for template conformance checking.

(along with their parts of speech), NPs, VPs, and named entities. Following text chunking, another text processing pipeline is executed. This second pipeline, shown in Fig. 8, is composed of pattern matchers for recognizing template grammars. We use the JAPE language, introduced in Section 2.3, for implementing these pattern matchers.

Below, we outline each of the steps in the pipeline of Fig. 8, showing the annotations generated by each step over the requirements of Fig. 9 (same requirements as in Fig. 2). We further present the JAPE implementation of selected steps to illustrate the NLP machinery behind the approach:

1) *Mark Head* marks the starting word in a requirements sentence, denoted `Head` in the examples of Fig. 9.
2) *Mark Modal VP* marks the VP that starts with a modal. A requirements statement typically has only one modal. If more than one modal is found, the first modal is annotated and a warning is generated to bring to the user's attention the presence of multiple modals. The annotation resulting from this step is denoted `Modal_VP` in Fig. 9. Note that R_3 contains no valid modal VP, hence the requirement is not having a `Modal_VP` annotation.
3) *Mark Anchor* first tags as `System_Name` the NP that precedes the modal. Subsequently, the step tags as `Anchor` the system name and the VP (including the modal) that follows `System_Name`. The anchor is later used for delineating the conditional slot. In the examples of Fig. 9, R_3 has no `Anchor` marked in it because the `Modal_VP` annotation is absent from this requirement.
4) *Mark Valid Sentence* marks as `Valid_Sentence` all sentences containing the anchor described above, subject to the constraint that the anchor is either at the beginning of a sentence or preceded by a segment starting with a conditional keyword. Sentences that do not have an anchor or fail to meet the additional constraint are marked as `Invalid_Sentence`. In the examples of Fig. 9, R_1 and R_2 are valid sentences whereas R_3 is an invalid one.
5) *Mark Condition* marks the optional condition in those valid sentences that start with a conditional keyword. The text between the beginning of such a sentence up to the anchor is marked as being a `Condition`. The scripts for marking conditions are different for Rupp's and EARS, as the syntactic structure of the conditions differs across these templates.
6) *Mark Conformant Segment* marks as `Conformant_Segment` the segment of a valid sentence that complies with the template. Since conformance rules are different across the templates (R.1-R.7 and E.1-E.4 in Rupp's and EARS, respectively), the scripts for conformance checking are also different. Fig. 6, discussed earlier in Section 2.3, shows an excerpt of the JAPE script for marking conformant segments for the *Autonomous* requirements type in Rupp's template.

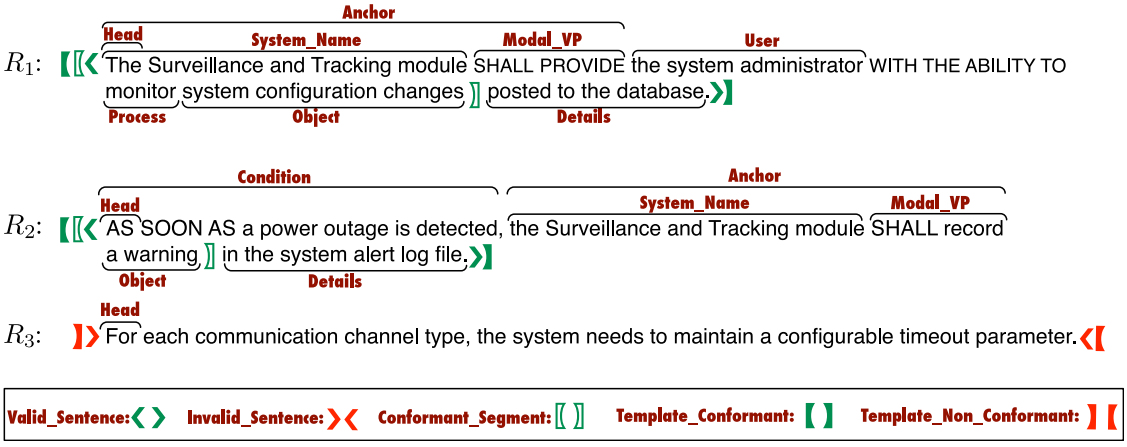

Fig. 9. Annotations generated by the pipeline of Fig. 8 over the example requirements of Fig. 2.

```
1. Phase: DoMarkDetails
2. Input: Sentence Conformant_Segment
3. Options: control = appelt
4.
5. Rule: MarkTemplateDetails
6.  (
7. {Sentence contains Conformant_Segment}
8. ):sentence
9. -->
10. {
11.   AnnotationSet sentenceAs =
12. (gate.AnnotationSet)bindings.get("sentence");
13.   AnnotationSet compliantAs =
14. inputAS.get("Conformant_Segment").getContained(
15.     sentenceAs.firstNode().getOffset(),
16.     sentenceAs.lastNode().getOffset());
17.   Node start = compliantAs.lastNode();
18.   Node end = sentenceAs.lastNode();
19.
20.   if (start == null || end == null)
21.     return;
22.
23.   FeatureMap features = Factory.newFeatureMap();
24.   outputAS.add(start, end, "Details", features);
25. }
```

Fig. 10. JAPE script for marking details.

7) *Mark Details* annotates as `Details` both the optional details envisaged by Rupp's template as well as the system response envisaged by EARS. Naturally, this annotation is relevant for only requirements sentences that contain a `Conformant_Segment`. The `Details` annotation applies to R_1 and R_2, as depicted in Fig. 9. Fig. 10 shows the JAPE script for generating the `Details` annotation. Lines 11-24 of the script are written directly in Java. The Java code computes the beginning and ending offsets for the annotation, that is, from the end of the `Conformant_Segment` to the end of the sentence.
8) *Mark Conditional Details* checks the details for terms that may imply additional conditions, notably, subordinate conjunctions, e.g., "whenever", "whereas", and "once". If such terms are detected in the details segment, the segment will be additionally marked as `Conditional_Details`. For example, if a phrase such as "whenever logging is enabled" is appended at the end of R_2 in Fig. 9, the requirement will be deemed as having conditional details. Both Rupp's and the EARS templates mandate that the conditional part should appear at the beginning of a requirements statement. Hence, conditional details should trigger non-conformance (see step 10). Fig. 11 shows the JAPE script for marking conditional details. The script recognizes any segment already marked as `Details` in which a conditional keyword, denoted `Conditional` is present. The `Conditional` annotation is produced via a customizable list (gazetteer) of conditional keywords.

```
1. Phase: DoMarkConditionalDetails
2. Input: Conditional Details
3. Options: control = appelt
4.
5. Rule: MarkConditionalDetails
6. (
7. {Details contains Conditional}
8. )
9. :label
10. -->
11. :label.Conditional_Details= {}
```

Fig. 11. JAPE script for marking conditional details.

```
1. Phase: DoMarkConditionalType
2. Input: Condition
3. Options: control = appelt
4.
5. Rule: MarkEventConditions
6. Priority: 20
7. (
8.   (
9.     {Condition.string ==~ "[Ww]hen(.)+" }
10.   )
11. ):label
12. -->
13. :label.ConditionType = {Type="Event-driven"}
```

Fig. 12. JAPE script for marking the event-driven conditional type in EARS.

9) *Mark Conditional Types*, exclusive to the EARS template, distinguishes the different sub-parts of the condition slot, e.g., trigger and specific states. The script subsequently infers the requirements type based on the type of the condition used. Fig. 12 shows an excerpt of the script, concerned with marking the condition type for event-driven requirements. For example, suppose that the conditional keyword of R_2 (Fig. 9) was `WHEN` instead of `AS SOON AS`. R_2 would then be deemed conformant to EARS with its conditional type being event-driven.
10) *Mark Template Conformance* marks as `Template_Conformant` any valid sentence that contains a conformant segment, excluding those that have conditional details (see step 8). Any requirement without a `Template_Conformant` segment will be marked as `Template_Non_Conformant`. For example, R_1 and R_2 in Fig. 9 are deemed conformant and R_3 is deemed non-conformant to Rupp's template.

3.3 Handling Complex Phrases

As we discussed in Section 2.2, text chunking cannot identify complex phrases. To illustrate how this affects TCC, consider the requirements statement R in Fig. 4a. This statement conforms to the EARS template if we elect to fill the ⟨system name⟩ slot of the template with the complex noun phrase "The information technology tools used in the design of systems performing safety functions". When a parse tree such as the one shown in Fig. 4c is available, we can, as we explain below, deduce that the above phrase is indeed a noun phrase. Text chunking, in contrast, identifies only the atomic noun phrases of this complex noun phrase. Consequently, the pipeline of Fig. 8 will mark R as non-conformant.

The absence of a parse tree seldom poses a problem for TCC. One of the main reasons why templates are used is to minimize the use of complex linguistic structures. Deeming as non-conformant a complex requirements statement such as that in the example of Fig. 4a may be desirable as a way to bring the complexity to the attention of the analysts. If the analysts however decide that such complexity does not warrant further investigation, they will need a mechanism

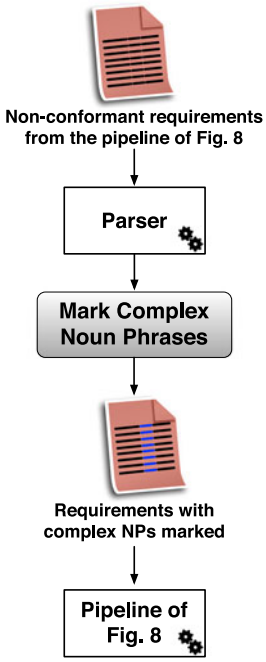

Fig. 13. Enhancing TCC with parsing.

for filtering non-conformance warnings that are exclusively due to the use of complex structures, thus narrowing the warnings to genuine deviations from the underlying template.

In Fig. 13, we show how one can enhance TCC with such a mechanism using natural language parsing. First, all non-conformant requirements statements from the pipeline of Fig. 8 are processed by a parser, with a parse tree such as the one in Fig. 4c generated for each statement. Equipped with a parse tree, we can recognize complex noun phrases, in turn enabling us to distinguish between non-conformance due to the use of complex sentence structures from non-conformance due to genuine deviations from the template of interest.

Fig. 14 shows a JAPE script that can recognize complex noun phrases. The script searches for parse tree node annotations, denoted `SyntaxTreeNode`, that have NP as their category (line 7). Using the *appelt* control option (line 3) ensures that when multiple noun phrases are detected in the same text region, only the one that is the longest is marked.[3] For instance, running this JAPE script over the example of Fig. 4a would mark the following two regions as NP: (1) "The information technology tools used in the design of systems performing safety functions", (2) "safety implications on the end-product".

Following the execution of the JAPE script in Fig. 14, the requirements originally marked as non-conformant are reprocessed by the pipeline of Fig. 13, but this time accounting for any new NP annotations generated for complex noun phrases. This second execution of the TCC pipeline filters out any non-conformance annotations caused by the limitation of text chunking in detecting complex phrases.

3. JAPE's different control options including *appelt* were discussed in Section 2.3.

```
1. Phase: DoMarkComplexNPs
2. Input: SyntaxTreeNode
3. Options: control = appelt
4.
5. Rule: MarkComplexNPs
6. (
7.    {SyntaxTreeNode.cat ==~ "NP"}
8. )
9. :label
10. -->
11. :label.NP= {}
```

Fig. 14. JAPE script for marking complex NPs using a parse tree.

3.4 Checking NL Best Practices

In addition to checking template conformance, we use NLP for detecting and warning about several potentially problematic constructs, also called requirements smells [28], that may be signs of vagueness or ambiguity in requirements statements. We build upon the requirements writing best practices by Berry et al. [2]. Table 1 lists and exemplifies several constructs that we detect automatically. The automation is done through JAPE in a manner similar to how template conformance is checked.

4 Tool Support

We have implemented our approach in a tool named REquirements Template Analyzer. RETA has been developed as an application for the GATE workbench (http://gate.ac.uk/).

Fig. 15 shows the overall architecture of RETA. Analysts may specify the requirements in the requirements authoring and management environment of their choice, e.g., Enterprise Architect (http://www.sparxsystems.com.au) or IBM DOORS (www.ibm.com/software/products/ca/en/ratidoor/). A glossary (if one exists) can optionally be provided to RETA to assist in the detection of noun phrases during the text chunking phase. The rules for checking conformance to templates and best practices are provided as scripts written in GATE's pattern matching language, JAPE. The various lists of terms used by these scripts, e.g., lists of template fixed terms, vague terms, conjunctions, modals and conditional words, are provided as customizable lists in GATE (gazetteers).

RETA uses GATE's user interface for showing diagnostics about template non-conformance and deviations from requirements writing best practices. A snapshot of GATE's interface after running the RETA application is shown in Fig. 16. The requirements in the snapshot are drawn from one of our case studies (Case-A) discussed in Section 5. These requirements have been slightly altered from their original form to protect confidentiality.

In this snapshot, the left panel displays the various GATE resources (applications, language resources, and processing resources). The center panel displays the contents of the resource selected—here, the exemplar requirements document. The right panel displays the annotation labels for the current document. When an annotation is selected from the list, all its occurrences are highlighted over the document.

RETA consists of 30 JAPE scripts, containing in total approximately 800 lines of code. Out of these, 26 scripts are common between the implementation of Rupp's and the EARS templates, suggesting that a large fraction of our

TABLE 1
Potentially Problematic Constructs (from Berry et al. [2]) Detected Through NLP

Annotation	Potential Ambiguities	Example
`Warn_AND`	The "and" conjunction can imply several meanings, including temporal ordering of events, need for several conditions to be met, parallelism, etc.	The S&T module shall process the query data **and** send a confirmation to the database. *A temporal order is implied by the use of 'and'.*
`Warn_OR`	The "or" conjunction can imply "exclusive or", or "inclusive or".	The S&T module shall command the database to forward the configuration files **or** log the entries. *The inclusive or exclusive nature of 'or' is unclear.*
`Warn_Quantifier`	Terms used for quantification such as all, any, every can lead to ambiguity if not used properly.	All lights in the room are connected to a switch. (example borrowed from Berry et al. [2]) *Is there a single switch or multiple switches?*
`Warn_Pronoun`	Pronouns can lead to referential ambiguity.	The trucks shall treat the roads before **they** freeze. (example borrowed from Berry et al. [2]) *Does "they" refer to the trucks or the roads?*
`Warn_VagueTerms`	There are several vague terms that are commonly used in requirements documents. Examples include userfriendly, support, acceptable, up to, periodically. These terms should be avoided in requirements.	The S&T module shall **support up to** five configurable status parameters. *"support" is a vague term.* *It is unclear whether "up to" means "up to and including", or "up to and excluding".*
`Warn_PassiveVoice`	Passive voice blurs the actor of the requirement and must be avoided in requirements.	If the S&T module needs a local configuration file, it shall be created from the database system configuration data. *It is unclear whether the actor is S&T module, database, or another agent.*
`Warn_Complex_Sentence`	Using multiple conjunctions in the same requirements sentence make the sentence hard to read and are likely to cause ambiguity.	The S&T module shall notify the administrator visually and audibly in case of alarms and events. *The statement may be interpreted as visual notification only for alarms and audible notification only for events.*
`Warn_Plural_Noun`	Plural Nouns can potentially lead to ambiguous situations.	**The S&T components** shall be designed to allow 24/7 operation without interruption. *Does this mean that every individual component shall be designed to be 24 / 7 or is this a requirement to be satisfied by the S&T as a whole?*
`Warn_Adverb_in _Verb_Phrase`	Adverbial verb phrases are discouraged due to vagueness and the chances of important details remaining tacit in the adverb (e.g. frequencies, locations)	The S&T module **shall periodically poll** the database for EDTM CSI information. *The frequency of the periodic activity is unspecified.*
`Warn_Adj_followed_by _Conjunction`	The adjective followed by two nouns separated by a conjunction, can lead to ambiguity due to the possible relation of adjective with just first noun or both nouns.	**compliant** hardware and software *Whether only hardware or both hardware and software have to be compliant is unclear.*

implementation can be reused from one template to another. RETA further includes 10 lists (gazetteers) containing the keywords used by the scripts. Out of these, eight are common between the Rupp's and the EARS template implementations.

To enable using GATE in realistic requirements development settings, we have implemented plugins to automatically export requirements written in IBM DOORS and Enterprise Architect. The IBM DOORS plugin has been implemented using a simple script written in DOORS Extension Language (DXL); the plugin for Enterprise Architect has been implemented in C# and is approximately 1,000 lines of code. A demo version of RETA along with a screencast illustrating its use is available at:

http://sites.google.com/site/retanlp/

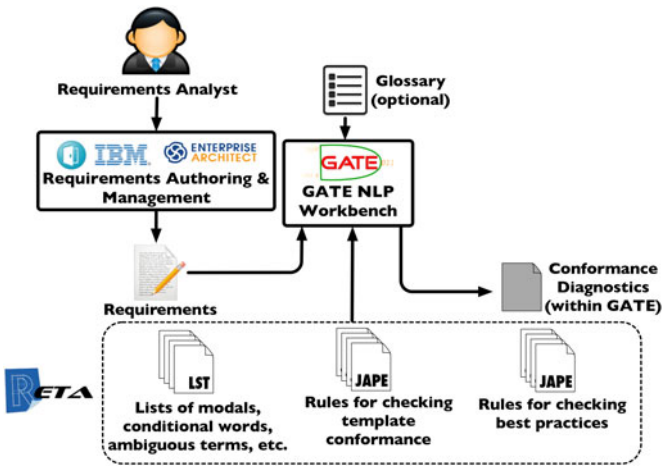

Fig. 15. RETA tool architecture.

5 Evaluation

We have conducted four case studies for evaluating our approach. The case studies have been selected to address and balance several considerations. Most notably, these considerations include: coverage of different industrial domains, coverage of different templates, and ensuring realistic scale in terms of the number of requirements statements. In the remainder of this section, we discuss the design, execution, and results of our case studies.

5.1 Research Questions (RQs)

Our case studies aim to answer the following RQs:

RQ1. What are optimal configurations for the NLP pipeline? Text chunking uses several NLP modules, executed in a sequence over an input document. For each stage in the sequence, one needs to choose from a number of alternative implementations. The aim of RQ1 is to establish which combination of implementations produces the most accurate results. Precision, recall, and F-measure are used as metrics for assessing accuracy.

RQ2. Does the absence of a glossary have an impact on the accuracy of our approach? The glossary terms may be unknown or incomplete at the time one needs to check conformance to a template. RQ2 aims to investigate how the absence of a glossary affects the accuracy of our approach.

RQ3. How effective is our approach at identifying non-conformance defects? Our ultimate goal is to make it easier for practitioners to identify requirements that do not conform to a given template. With RQ1 establishing the level of accuracy to be expected from our approach, RQ3 aims to determine whether the accuracy is likely to be good enough from a practical standpoint.

RQ4. Is our approach scalable? In a realistic setting, one can be faced with hundreds and sometimes thousands of requirements. RQ4 aims to establish whether our approach runs within reasonable time.

In Section 5.6, we answer these RQs based on the results of our case studies, and further provide preliminary insights into the benefits of our approach from a practitioner's perspective.

5.2 Description of Case Studies

The case studies in our evaluation are the following:

- *Case-A* concerns a software component in a satellite ground station. The requirements for the component

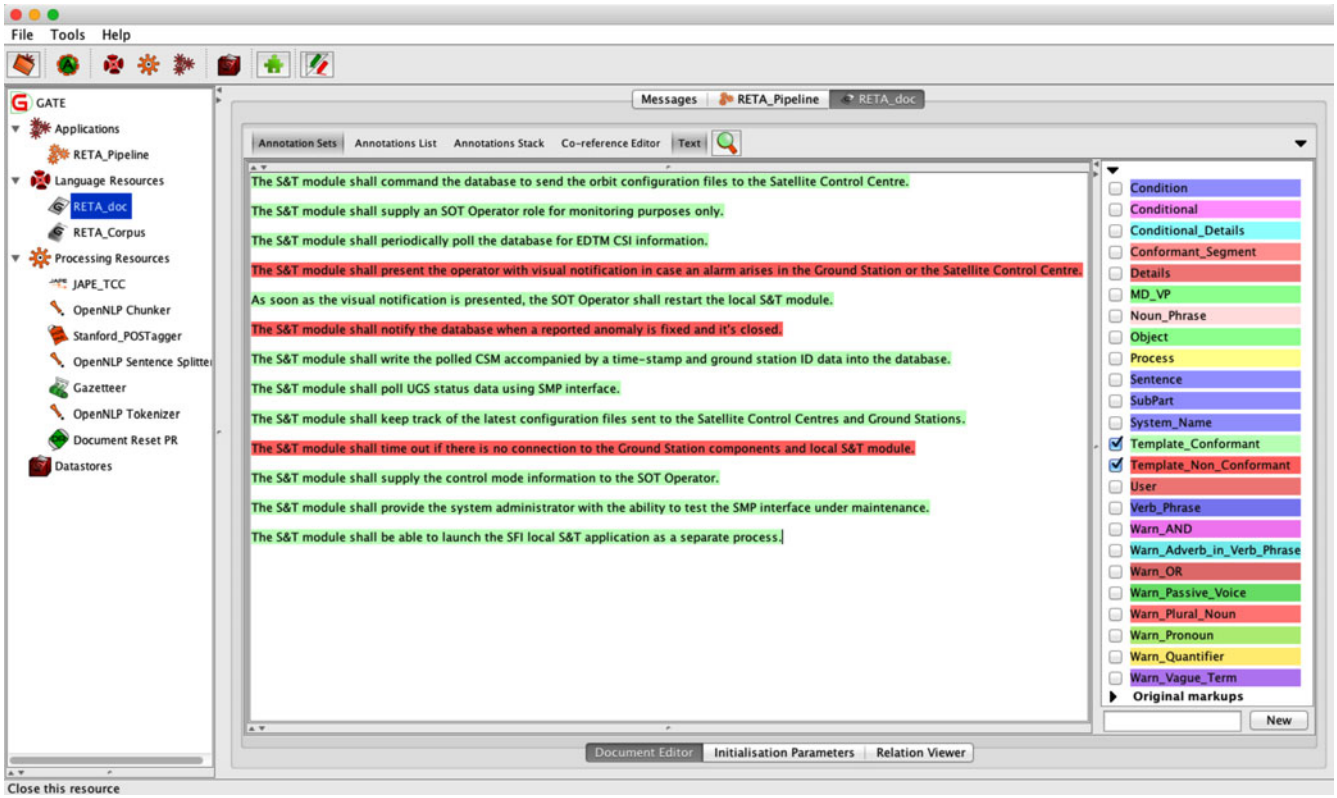

Fig. 16. Snapshot of the annotations generated by RETA.

TABLE 2
The Case Studies Used for Evaluation

Case	Description	Domain	Number of Requirements	Template Used
Case-A	Requirements for a software component in a satellite ground station	Satellites	380	Rupp's
Case-B	Requirements for a safety evidence management tool suite	Safety certification of embedded systems	110	Rupp's
Case-C	Requirements from Case-A restated in the EARS template.	Satellites	380	EARS
Case-D	Regulatory requirements for nuclear safety by the Finnish Radiation and Nuclear Safety Authority	Nuclear energy	890	EARS

were written by professionals in the satellite industry using Rupp's template. This case study, a preliminary version of which was reported in [19], was conducted in collaboration with SES TechCom—a satellite service provider. Case-A contains 380 requirements statements.

- *Case-B* concerns a tool suite for managing the safety information (safety evidence) used during the safety certification of embedded systems. The tool suite is currently under development in a European project named OPENCOSS (http://www.opencoss-project.eu). This case study was conducted in collaboration with requirements analysts from the OPENCOSS project. Similar to Case-A, Rupp's template was applied for writing the requirements. Case-B has 110 requirements statements.
- *Case-C* is a variant of Case-A, whereby all the 380 requirements in Case-A were transformed from Rupp's to the EARS template. Examples and details about the transformation are provided in Section 5.3.
- *Case-D* concerns safety requirements for nuclear facilities developed by the Finnish Radiation and Nuclear Safety Authority [29]. These requirements were written using the EARS template by requirements experts in collaboration with nuclear safety engineers. The experience from applying EARS to these requirements has been documented by the requirements authors [6]. Case-D contains 890 requirements statements.

Table 2 summarizes key information about the case studies by providing for each case study a brief description, the domain in which the study was conducted, the number of requirements involved, and the requirements template used.

5.3 Data Collection Procedure

Data collection was performed in two phases: (1) documentation of requirements statements and identification of glossary terms; and (2) inspection of the requirements resulting from the first phase in order to determine which requirements are conformant and which ones are not. In Cases A, B, and D, Phase 1 was carried out by industry experts, except that in Case-D the glossary terms were not specified. In Case-C, the requirements were rephrased (from Case-A) by the researchers, with the glossary terms from Case-A reused as is. Phase 2 in all case studies was carried out by the researchers following the completion of Phase 1. We elaborate each of the two phases below:

Phase 1. The requirements in Case-A were written following two half-day training sessions, in which the researchers familiarized the participating industry experts with Rupp's template and how to use it. The researchers had no control over how the requirements were specified after these training sessions. In Case-B, the requirements originated from varied sources in the project's consortium. These requirements were documented using Rupp's template by expert engineers within the consortium, without involvement from the researchers.

Case-C requirements were derived from those in Case-A through a transformation. To this end, a systematic process was followed: (1) All non-conformant requirements (to Rupp's template) were carried over verbatim without being rephrased; (2) From the conformant requirements, those that did not have a conditional part were carried over verbatim as well and classified as *ubiquitous* requirements under EARS. (3) Conditional requirements were mapped to different requirements types in EARS based on the nature of the condition. For instance, requirements with temporal conditions were rephrased so as to conform to the *event-driven* requirements type in EARS. As an example, consider the following requirement in Rupp's template: "As soon as an unplanned outage is detected the S&T shall inform the SMP interface". This requirement falls under the *event-driven* requirements type in EARS and is thus rephrased as follows in Case-C: "When an unplanned outage is detected the S&T shall inform the SMP interface".

Note that the above transformation is *not* conformance-preserving. In particular, non-conformant requirements under Rupp's template may be deemed conformant under EARS. This happens due to the fact that in Rupp's template, it is mandatory to have an ⟨object⟩ slot following the ⟨process⟩ slot; whereas, in EARS no specific constraint exists regarding the presence or the position of an object in a requirements statement. For example, the requirement "The S&T shall present to the SOT operator the EDTM anomalies." is not conformant to Rupp's template due to the object not being placed in the expected slot. This requirement is nevertheless conformant to EARS.

Finally, in Case-D, the requirements were written by experts with advanced training in the EARS template [6]. The researchers were not involved in requirements specification.

In Cases A and B, the glossary terms were provided by the experts. Two considerations about glossary term identification were highlighted to the experts ahead of time: First, that for the purposes of our study, we did not require the experts to define the glossary terms, but only to identify

them. Second, it was suggested to the experts that, when in doubt as to whether a particular term should be in the glossary, to include rather than exclude the term. These measures are important for RQ2, as the RQ aims to examine the effect of a glossary that is as complete as possible. Hence, we needed to mitigate the risk that the set of glossary terms would have major omissions due to the time pressure posed by having to define the terms. As noted earlier, for Case-C, we use the same glossary terms as Case-A. For Case-D, no glossary was provided as part of the requirements document and the researchers did not have access to the involved experts to elicit the glossary terms.

Phase 2. This phase is concerned with manually inspecting the requirements to determine which requirements (from Phase 1) are conformant to the underlying template. The data collected in this phase is used for calculating the effectiveness of automated conformance checking, as we discuss in Section 5.4. To increase the validity of the results, this phase was independently conducted by two different individuals (first and second authors). Subsequently, all discrepancies between the two inspectors were discussed and an agreement about conformance vs. non-conformance was reached in all cases. To ensure that the inspectors performed the inspection consistently with one another, an abstract inspection protocol, shown in Listing 1, was drawn up and followed by both inspectors.

Listing 1. Manual inspection protocol for template conformance.

1. Let R be the requirement being inspected for conformance to template T (either Rupp's or EARS).
2. Verify that R is a grammatically-correct sentence. Do *not* consider punctuation in determining correctness.
3. Verify that R uses an acceptable modal.
4. **if** R is conditional **then**
5. Verify that the conditions appear *only* at the beginning of R.
6. Verify that the conditions conform to the structure prescribed by T.
7. **end if**
8. **if** T is Rupp's template **then**
9. Verify that ⟨system name⟩, ⟨object⟩, and ⟨whom?⟩ (when applicable) are filled by noun phrases.
10. Verify that ⟨process⟩ is filled by a verb phrase.
11. **else if** T is EARS **then**
12. Verify that ⟨system name⟩ is filled by a noun phrase.
13. Verify that ⟨system response⟩ starts with a verb phrase.
14. **end if**
15. **if** all criteria are fulfilled **then**
16. R is conformant to T;
17. **else**
18. R is not conformant to T;
19. **end if**

An important factor during the inspection is whether the experts who wrote the requirements would like to admit only atomic noun phrases in the noun phrase slots (i.e., ⟨system name⟩, ⟨object⟩, and ⟨whom?⟩ for Rupp's template and ⟨system name⟩ for the EARS template), or to further admit complex noun phrases, as discussed in Section 3.3. In Cases A, B, and C, the inspection would yield identical results irrespective of whether we admit complex noun phrases or not—an indication that the experts used only atomic noun phrases in the relevant slots. In contrast, in Case-D, the experts indicated that a conscious choice had been made to allow complex noun phrases in the ⟨system name⟩ slot. Accordingly, during our inspection of the requirements in Case-D, we accepted (grammatically-correct) complex noun phrases in the ⟨system name⟩ slot.

5.4 Analysis Procedure

Our analysis involves the execution of different configurations of NLP modules for text chunking and measuring how effective each configuration is in distinguishing requirements that conform to a template from those that do not.

5.4.1 NLP Pipeline Configuration

To instantiate the pipeline of Fig. 5, one needs to choose, for each step in the pipeline, a specific implementation from the set of existing alternative implementations. We narrow our investigation to the set of implementations in GATE. This decision is based on two considerations: First, GATE brings together a large collection of mature NLP technologies and provides a unified mechanism for integrating them through a generic annotation infrastructure. These characteristics of GATE make it possible for us to experiment with several alternative solutions, beyond what would have been feasible in other existing NLP frameworks. Second, an important conclusion we would like to reach from our evaluation is how to build an accurate and at the same time practical tool. Focusing on a single NLP framework like GATE enables us to come up with concrete recommendations, without having to worry about the (likely) risk of interface incompatibilities between implementations that have not been already adapted to work together.

While our approach depends mainly on the annotations produced by the text chunking modules, i.e., Steps 5 and 6 in Fig. 5, these two steps rely on the annotations produced in previous steps, i.e., Steps 1-4; therefore, the performance of Steps 5 and 6 ultimately relates to that of their previous steps. For this reason, we consider in our analysis not only different instantiations of Steps 5 and 6 but also different instantiations of Steps 1-4. Below, we list the different alternatives considered for each of the steps in Fig. 5:

Step 1 (2 alternatives): ANNIE English Tokenizer [30], OpenNLP Tokenizer [31].

Step 2 (2 alternatives): ANNIE Sentence Splitter [30], OpenNLP Sentence Splitter [31].

Step 3 (3 alternatives): OpenNLP POS Tagger [31], Stanford POS Tagger [32], ANNIE POS Tagger [33].

Step 4 (2 alternatives): ANNIE Named Entity (NE) Transducer [30], OpenNLP Name Finder [31].

Step 5 (3 alternatives): OpenNLP (NP) Chunker [31], Multilingual Noun Phrase Extractor (MuNPEx) [34], ANNIE Noun Phrase Chunker (an extension of Ramshaw & Marcus (RM) Noun Phrase Chunker [35]).

Step 6 (2 alternatives): ANNIE Verb Group Chunker [30], OpenNLP (VP) Chunker [31].

Most of the above modules are based on machine learning. For all modules requiring training, we use the default

		Predicted (Automatic)	
		Conformant	Non-Conformant
Actual (Manual)	Conformant	*True Negative (TN)*	*False Positive (FP)*
	Non-Conformant	*False Negative (FN)*	*True Positive (TP)*

Fig. 17. Confusion matrix for measuring accuracy.

training data (for English) that is distributed with GATE Release 8.0 [27].

The noun and verb phrase tags produced in Steps 5-6 are the basis for checking template conformance, as we described in Section 3. For Step 5, there is a choice to be made as to whether to include the glossary terms as an input. Therefore, we have a total of $2 \times (2 \times 2 \times 3 \times 2 \times 3 \times 2) = 288$ different configurations to compare for cases A through C. For Case-D, since the glossary terms are unknown; we have only half the number of configurations, i.e., 144.

The annotations produced by each configuration is fed to the JAPE pipeline of Fig. 8. For Case-D, since we would like to further admit complex noun phrases into the slots of the underlying template (as discussed in Section 5.3), the periphery pipeline of Fig. 13 needs to be executed as well. We use the Stanford Parser [23] that is integrated into GATE 8.0 for the Parser step of the pipeline in Fig. 13. The accuracy of TCC is subsequently analyzed using the accuracy measures discussed next.

5.4.2 Metrics for Measuring Accuracy

Our analysis of accuracy is based on *precision* and *recall*. These classification accuracy metrics are widely used in many areas, e.g., information retrieval (IR) [36], where one needs to measure the ability of an approach to correctly classify a set of objects into classes with certain properties. In our case, we are concerned with two classes: (1) template conformant and (2) template non-conformant. The simple confusion matrix shown in Fig. 17 captures the possible errors that an automated conformance checker can make in the classification.

Precision is a metric for quality (low number of false positives) and is defined as $\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$. Recall is a metric for coverage (low number of false negatives) and is defined as $\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$. In most classification problems, including ours, an increase in precision comes at the cost of a decrease in recall and vice versa [37]. To compare different NLP pipeline configurations while simultaneously accounting for both precision and recall, we use a metric, called F-measure [36], which computes the weighted harmonic mean of precision and recall. Depending on the context, one may want to place more emphasis on either precision or recall. In our study, recall is the primary factor as it is easier for analysts to rule out a small number of false positives than to go through a large document in search of false negatives. Hence, we use a definition of F-measure, known as F_2-measure, which gives more weight to recall than precision. F_2-measure is defined as: $3 \times \mathrm{Precision} \times \mathrm{Recall}/(2 \times \mathrm{Precision} + \mathrm{Recall})$.

We note that other metrics and weights could have been used for combining precision and recall. Nevertheless, F_2-measure is standard when recall needs to be weighted higher than precision. We further note that we use classification accuracy metrics only for evaluation purposes. The end-users of our approach are not exposed to these metrics and do not need to make decisions based on them. The practical implications of these metrics for end-users are addressed in RQ3.

5.5 Results

This section describes the results of our case studies.

5.5.1 Requirements Inspection and Glossary Elicitation

In Table 3, we provide various statistics about the case studies: the level of template conformance as established by the manual inspection protocol in Section 5.3, the number of glossary terms elicited from the experts, and the distribution of conformant requirements across the different requirements types in the underlying template. The table further shows the inter-rater agreement, expressed as

TABLE 3
General Statistics for the Case Studies

Case	Template Conformant	Template Non-Conformant	Number of Glossary Terms	Requirements Types		Inter-rater Agreement (Cohen's Kappa)
Case-A	243 (64%)	137 (36%)	127	Autonomous	206	0.943 (almost perfect agreement)
				User Interaction	35	
				Interface	2	
Case-B	98 (89%)	12 (11%)	51	Autonomous	44	1.0 (perfect agreement)
				User Interaction	43	
				Interface	11	
Case-C	297 (78%)	83 (22%)	127 (reused from Case-A)	Ubiquitous	290	0.946 (almost perfect agreement)
				Event-Driven	5	
				Unwanted Behavior	2	
				State-Driven	0	
				Optional Feature	0	
				Complex	0	
Case-D	857 (96%)	33 (4 %)	0	Ubiquitous	546	0.786 (substantial agreement)
				Event-Driven	41	
				Unwanted Behavior	72	
				State-Driven	22	
				Optional Feature	150	
				Complex	26	

TABLE 4
Results from the Analysis of Non-Conformant Requirements

Non Conformance Type	Explanation	Example	Case	Percentage of Non Conformances
Minor Deviations	Requirement deviates only slightly from template's prescribed structure, e.g., by missing some of the fixed elements.	The S&T component shall provide the user with a function to view the network status.	**Case-A**	2 / 137 = 1.5%
			Case-B	0
			Case-C	0
			Case-D	0
Enumerations	Requirement concerns more than one object or functionality.	The state of the S&T module can be standby, active, or degraded.	**Case-A**	14 / 137 = 10.2%
			Case-B	2 / 12 = 16.7%
			Case-C	14 / 83 = 16.9%
			Case-D	5 / 33 = 15.2%
Missing or Misplaced Object	Exclusive to Rupp's template: The object slot is missing or placed at a non-prescribed position.	The OPENCOSS platform shall provide users with the ability to use in an assurance project evidence types that have been defined in another project.	**Case-A**	54 / 137 = 39.4%
			Case-B	6 / 12 = 50%
			Case-C	0
			Case-D	0
Incorrect Conditional Keyword	The conditional keyword is not among the ones prescribed by the template, e.g., WITHIN and AFTER.	After underwriting1 is complete, if necessary the Insurance Officer shall perform underwriting2.	**Case-A**	2 / 137 = 1.5%
			Case-B	0
			Case-C	2 / 83 = 2.4%
			Case-D	0
Misplaced Conditions	Conditions appear in positions other than prescribed (which is the requirement's beginning).	The S&T module shall load a new configuration from the database as soon as the module receives a reloading request.	**Case-A**	61/137 = 44.5%
			Case-B	4 / 12 = 33.3%
			Case-C	61/ 83 = 73.5%
			Case-D	18 / 33 = 54.5%
Ill-formed Requirement	Requirement is grammatically incorrect or has some ill-formed slot.	The S&T shall periodically check the of the network elements by using the ping command.	**Case-A**	4 / 137 = 3%
			Case-B	0
			Case-C	4 / 83 = 4.8%
			Case-D	2 / 33 = 6.1%
Incorrect or missing modal	Requirement has an incorrect or missing modal in its verb phrase.	The approval of the final safety analysis report is a precondition for the endorsement of the application for an operating license.	**Case-A**	0
			Case-B	0
			Case-C	2 / 83 = 2.4%
			Case-D	8 / 33 = 24.2%

Cohen's Kappa [38], for the independent inspections conducted by the first two authors. The conformance statistics in Table 3 are based on the mutually agreed-upon inspection results, after the differences between the two inspectors were discussed and resolved.

The requirements in Case-C, as mentioned previously, were derived from those in Case-A. The significant majority of (conformant) requirements in Case-C fall under the ubiquitous category, which is the simplest requirements type in EARS. This is because the requirements in Case-A were written without considering the EARS template. Hence, no conscious effort was made in Case-A to use the range of requirements types available in EARS. All cases, except Case-C, cover all the requirements types in their underlying template. Further, Case-C, as shown in Table 3, has a higher percentage of conformant requirements than Case-A (78 percent in Case-C versus 64 percent in Case-A). The reason for this increase in the rate of conformance is the absence of a mandatory ⟨object⟩ slot in the EARS template. Specifically, the requirements that are deemed non-conformant to Rupp's template in Case-A because of a missing or misplaced object are deemed conformant to EARS in Case-C.

With regards to reasons for non-conformance, the majority of issues are explained by the factors listed in Table 4. As shown in the table, the most frequent reasons for non-conformance are misplaced or missing objects in Rupp's template, and misplaced conditions in both Rupp's and the EARS templates. These two factors collectively account for approximately 80 percent of non-conformance issues. The other factors are minor deviations, enumerations, incorrect conditional keywords, ill-formed sentences, and incorrect or missing modals.

For requirements containing enumerated objects, e.g., the example given for enumerations in Table 4, we did not advise the experts to take any remedial action to address non-conformance. This is because enumerated objects appear to be more naturally expressed using a sentence structure similar to our example, as opposed to the structure prescribed by the Rupp's and the EARS templates.

In the case of misplaced conditions, a closer examination was conducted to determine if one could naturally fit the affected requirements into the respective template, either by revising the sentence structures or by decomposing the affected requirements into finer-grained ones. Here, we observed what appears to be a limitation in the set of

TABLE 5
Best Pipelines in Terms of F_2-Measure

Case	Tokenizer	Splitter	POS Tagger	Name Finder	Glossary?	NP Chunker	VP Chunker	Time (secs)	Precision	Recall	F_2-measure
Case-A	--	--	Stanford	--	YES	MUNPEX	ANNIE	4 s (avg)	0.91	0.99	0.96
	OpenNLP	--	Stanford	--	NO	MUNPEX	ANNIE	3.5s (avg)	0.91	0.99	0.96
Case-B	--	--	OpenNLP	--	--	ANNIE	--	2s (avg)	1	1	1
	--	--	OpenNLP	--	--	OpenNLP	--	2.5s (avg)	1	1	1
Case-C	OpenNLP	--	OpenNLP	--	--	ANNIE	ANNIE	4.2s (avg)	0.94	0.98	0.97
Case-D	OpenNLP	ANNIE	OpenNLP	--	NO	OpenNLP	--	73.2s (avg)	0.94	1	0.98

Note: A "--" in the table means that the results are not sensitive to the choice of alternative used for that particular NLP module. See Section 5.4.1 for the set of alternatives for each module.

conditional keywords recommended by both Rupp's and the EARS template. Specifically, to be able to express a performance constraint, one often needs to use `WITHIN`, e.g., "WITHIN 2 seconds after a critical failure is detected, the S&T module shall trigger the sound alarm on the main control panel". During the inspection, a requirement like the above would be deemed non-conformant because the conditional keyword is not among the ones prescribed. We therefore propose that the conditional keywords in the two templates be extended with `WITHIN`. Our accuracy results, presented next in Section 5.5.2, are nevertheless calculated based on our original inspection results, i.e., as reported in Table 3, and irrespective of our own observation about conditional keywords.

5.5.2 Accuracy and Execution Time

The NLP pipelines discussed in Section 5.4.1 were executed for Cases A through D. For each pipeline, the classification accuracy metrics were computed as well as the time it took to execute the pipeline. In Table 5, we show the most accurate pipelines (best F_2-measure) for each case study. Descriptive statistics for precision, recall, F_2-measure, and execution time across *all* pipelines are given in the form of box plots in Fig. 18. For Case-D, the accuracy results and execution times in Table 5 are inclusive of the periphery process for handling complex noun phrases. The accuracy gain from this periphery process is shown in Fig. 19, where we contrast, through box plots, the accuracy with and without applying the periphery process. All experiments were conducted on a 2.3 GHz Intel Core i7 CPU with 8 Gb of memory.

In Case-A and Case-C, there are 24 outliers in precision (< 0.7), leading to 24 outliers in F_2-measure. All outliers have three features in common: the use of ANNIE Tokenizer, MuNPEx NP Chunker and the absence of a glossary. In 12 of the outliers, where precision is very low (in the 0.3-0.4 range), the poor outcome is due to MuNPEx NP Chunker being misled by incorrect tokenization produced by ANNIE Tokenizer. The result is that MuNPEx NP Chunker cannot correctly recognize the ⟨system name⟩ slot. In the remaining 12 outliers, the problem remains the same, except that OpenNLP NE Transducer corrects some of ANNIE Tokenizer's mistakes, by recognizing the system name as an (atomic) named entity. Despite this, precision remains fairly low (in the 0.5-0.7 range for Case-A and 0.5-0.6 range in Case-C).

In Case-B and Case-D, there are no outliers, but there is significant variation in F_2-measure, brought about by the variation in precision. The variation in precision is largely explained by the low number of non-conformant requirements. In other words, even a small number of false positives can have a considerable impact on precision and in turn on F_2-measure.

5.6 Discussion

RQ1. The best (i.e., most accurate) text chunking pipelines for Cases A through D were shown earlier in Table 5. Since the best choice differs across the case studies, one cannot recommend a single pipeline per se for use over a new (and unknown) requirements document. The results in Table 5 thus do not allow us to draw conclusions about the optimal NLP pipeline for text chunking. To come up with a general recommendation for the NLP pipeline, we need to pay attention to the impact that a particular NLP module can have on the outcome across all the pipelines it appears in. For example, a module that does not appear in the best pipeline but performs consistently well across all the pipelines it appears in may be preferred over a module that does appear in the best pipeline but also appears in some pipelines with poor results.

In more precise terms, we need to determine what modules cause the most variation in the accuracy metrics and avoid modules that cause a large degree of uncertainty. This analysis of variation is best conducted using a regression tree [39]. A regression tree is built by partitioning a data set, e.g., NLP module combinations here, in a stepwise manner to obtain partitions that are as consistent as possible with respect to a certain criterion, e.g., F_2-measure in our context.

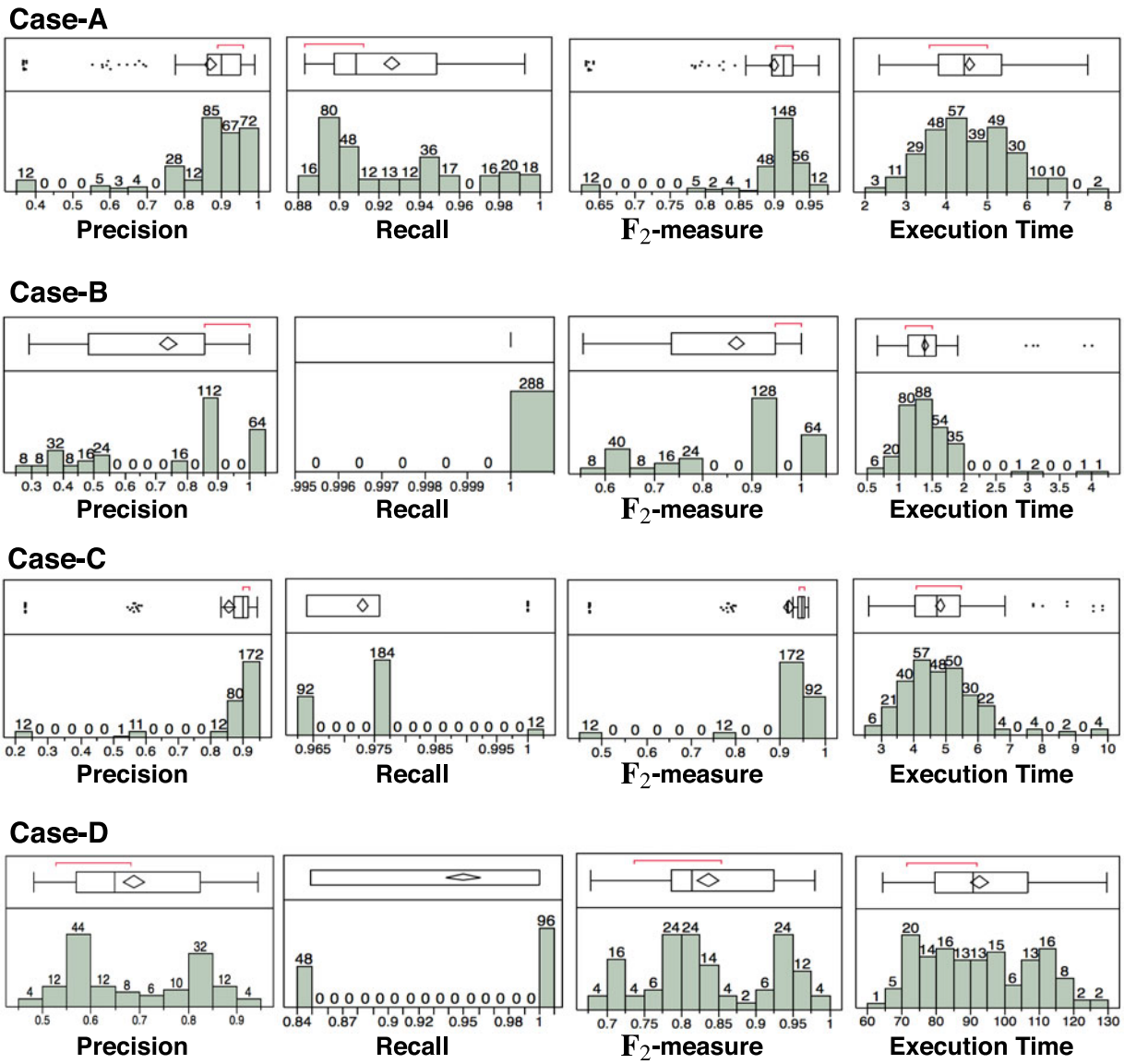

Fig. 18. Box plots for classification accuracy metrics and execution times.

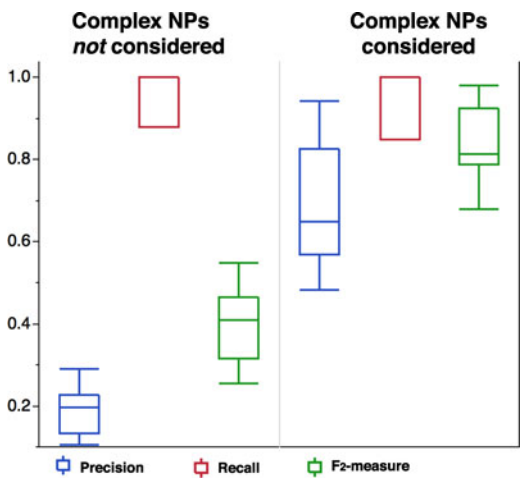

Fig. 19. Accuracy results for Case-D with and without considering complex noun phrases.

In Fig. 20, we show the regression trees for F_2-measure for our case studies. At each level in the tree, one NLP module is picked out and the pipelines are partitioned according to whether they use that module or not. The criterion for selection is to choose the module that would minimize the standard deviation across the branches that result after partitioning. In other words, the module that is most influential in explaining the variance in F_2-measure is selected. In each node of the tree, we show the count (number of pipelines), the mean and standard deviation for F_2-measure, and the difference between the smallest and largest F_2-measure observed in the partition. We note that the regression tree for Case-D has been constructed *without* applying the periphery process for handling complex noun phrases (see Fig. 13). This is because the regression tree is meant for analyzing the sensitivity of accuracy to the text chunking NLP modules. The inclusion of the periphery process can undesirably alter the sensitivity results, because the process not only identifies complex noun phrases but also any atomic noun phrases that might have been missed by text chunking, thus masking text chunking errors.

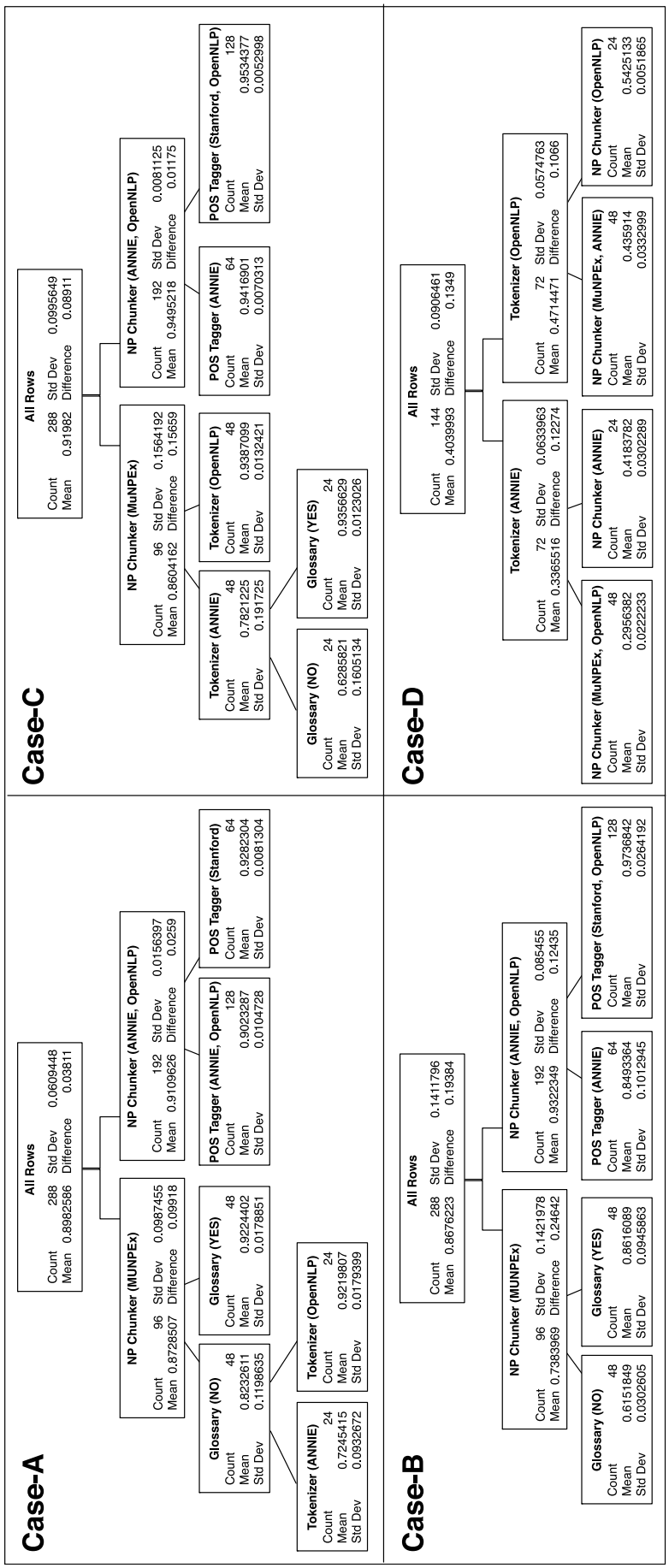

Fig. 20. Regression trees for F_2-measure.

TABLE 6
Average Accuracy for Recommended Pipeline Configurations

Case	With Glossary			Without Glossary		
	Precision	Recall	F_2-measure	Precision	Recall	F_2-measure
Case-A	0.94	0.91	0.92	0.94	0.91	0.92
Case-B	0.93	1	0.98	0.93	1	0.98
Case-C	0.91	0.98	0.96	0.91	0.98	0.96
Case-D	-	-	-	0.85	0.96	0.92

As shown by Fig. 20, in three out of the four cases, namely Cases A through C, the most critical decision concerns the choice of the NP Chunker module. In these three case studies, MuNPEx NP Chunker performs well when the glossary terms are provided but does poorly on average otherwise. In Case-C (but not Case-A and Case-B), MuNPEx NP chunker performs well in the absence of glossary terms, as long as OpenNLP Tokenizer is used for tokenization. Compared to MuNPEx Chunker, ANNIE and OpenNLP NP Chunkers are less sensitive to the implementation choices for other NLP modules and thus introduce less variation. Within the ANNIE / OpenNLP Chunker branch in Cases A through C, the most critical decision concerns the POS Tagger, with Stanford and OpenNLP taggers achieving higher means.

The accuracy in Case-D is most sensitive to the choice of the Tokenizer, with OpenNLP leading to more accurate results than the ANNIE tokenizer. At the next level in regression tree, the most critical choice is that of the NP Chunker, with MuNPEx chunker yielding less accuracy than ANNIE and OpenNLP chunkers—consistent with Cases A through C.

The change in the most critical component being the Tokenizer in Case-D, as opposed to the NP Chunker in Cases A through C, is largely explained by the different tokenization behaviors of ANNIE and OpenNLP over certain patterns that were seen frequently in Case-D, most notably the use of cross-references in the requirements statements. In the case of the ANNIE tokenizer, the components of the numeric part of a cross-reference would be treated as separate tokens. For example, the ANNIE tokenizer would tokenize the cross-reference "Article 3.3.2.1" into eight tokens, one for the term "Article" and seven for the numeric part. OpenNLP would instead tokenize the cross-reference into two tokens, one for "Article" and one for the numeric part. Thus, in our context, misleading tokenization by the ANNIE tokenizer affects the behavior of the NP Chunker, making it the primary reason for wrongly-identified NPs, and eventually an incorrect delineation of the ⟨system name⟩ slot in the requirements statements. Subsequently, accuracy in Case-D is less affected by the choice of the NP Chunker than the Tokenizer.

Despite the above discrepancy between Case-D and Cases A through C, there is no inconsistency between the four case studies, in the sense that there are combinations of NLP modules that work well across all the case studies. Based on our analysis of the full regression trees for Cases A through D, we recommend the following modules for instantiating the text chunking pipeline:

- *Tokenizer.* OpenNLP Tokenizer.
- *Sentence splitter.* ANNIE Sentence Splitter OR OpenNLP Sentence Splitter.
- *POS Tagger.* OpenNLP POS Tagger OR Stanford POS Tagger.
- *NE Transducer.* ANNIE NE Transducer OR OpenNLP Name Finder.
- *NP Chunker.* ANNIE (RM) NP Chunker OR OpenNLP NP Chunker.
- *VP Chunker.* ANNIE VG Chunker OR OpenNLP VP Chunker.

The above choices lead to the best overall results with little variation. Other alternatives may produce good results but may turn out to be too sensitive to the presence of certain other modules or conditions. For example, the MuNPEx NP Chunker should be used only in the presence of a (good) glossary.

If we narrow the text chunking pipeline configurations to those made up of the modules recommended above, we obtain 32 configurations that can be executed with or without a glossary. In Table 6, we show for each case study, the average accuracy of these 32 configurations. Comparing these accuracy levels against the best possible accuracy levels shown earlier in Table 5, we see an accuracy reduction of 4, 2, 1, and 6 percent, respectively for Cases A through D.[4] We believe this reduction is small enough to be tolerated in exchange for the high stability of the results generated by our recommended configurations. In RQ3, we discuss the practical implications of these accuracy levels.

RQ2. As indicated by Table 6, as long as one uses the NLP modules recommended in RQ1, the presence of a glossary does not lead to accuracy gains. We therefore expect our approach to work with high accuracy even when the glossary terms are unknown.

RQ3. Table 7 shows for each of our case studies the expected number of false positives and false negatives, based on the number of requirements statements in the study and the average accuracy levels in Table 6 for our recommended pipelines.

The expected number of false positives is small across all case studies both in absolute numbers, as suggested by Table 7, and also as a percentage of the total number of nonconformances (Table 3). Specifically, this percentage is: $8/137 = 5.8\%$ for Case-A, $1/12 = 8.3\%$ for Case-B, $8/83 = 9.6\%$ for Case-C, and $6/33 = 18.2\%$ for Case-D. We thus anticipate that excluding false positives would comparatively take little effort.

4. As a technical remark, we note that the pipeline configurations in our study are deterministic, i.e., they produce the same results across different runs over the same input. In other words, we obtain a single accuracy value, as opposed to an accuracy distribution, for a given pipeline configuration over a given input. Due to the absence of random variation, we do not need statistical significance testing for comparing the accuracy results.

TABLE 7
Expected Number of False Positives and False Negatives Based on Average Accuracy Levels (Table 6)

Case	Number of Requirements	False Positives	False Negatives
Case-A	380	8	12
Case-B	110	1	0
Case-C	380	8	2
Case-D	890	6	1

The remaining question is how much of a quality problem false negatives pose. Ideally, one would like to avoid false negatives completely; however, this is practically infeasible as doing so will come at the expense of introducing a large number of false positives. Table 7 suggests that false negatives are few in absolute numbers. Further, when viewed as a percentage of the total number of requirements in each case study, false negatives constitute a small fraction: $12/380 = 3.2\%$ for Case-A, $0/110 = 0\%$ for Case-B, $2/380 = 0.5\%$ for Case-C, and $1/890 = 0.1\%$ for Case-D. For a large set of requirements in a real project, a manual inspection conducted under time pressure is unlikely to produce perfect results. Hence, such small fractions of false negatives are unlikely to outweigh the automation benefits of our approach, when compared to manual inspections.

RQ4. The main aspect of scalability that needs to be investigated in our context is whether our approach works within reasonable time over a large collection of requirements. Ideally, as the number of requirements grows, we expect the execution time to grow linearly as well. To analyze how execution times grow, we first combined the requirements sets written using the same template, i.e., we combined Case-A with Case-B requirements, and combined Case-C with Case-D requirements. The combination of Case-A and Case-B (Rupp's template) yields $380 + 110 = 490$ requirements, and that of Case-C and Case-D (the EARS template) – $380 + 890 = 1{,}270$ requirements. We then randomized the order of the combined sets. Let *Case-AB* and *Case-CD* denote these sets after randomization.

From Case-AB, we built five requirements sets of increasing sizes: the first 98 (i.e., $490/5$) requirements of Case-AB, the first $2 * 98$ requirements in Case-AB and so on. We built five similar sets for Case-CD: the first 254 (i.e., $1{,}270/5$) requirements of Case-CD, the first $2 * 254$ requirements of Case-CD, and so on until the last one with $5 * 254 = 1{,}270$ requirements. We then examined the execution time for TCC using different text chunking pipelines and against the growing number of requirements. In Fig. 21, we show the execution time plots for each template. The results show a linear growth pattern in both plots, thus providing evidence that automatic TCC based on text chunking will scale linearly. Given such linear relation and the fact that processing the entire set of Case-AB and Case-CD requirements takes only a few seconds, we anticipate that our approach should be practical for much larger sets of requirements.

We further evaluated the scalability of the additional process for handling complex noun phrases in Case-D. To do so, we took the non-conformant requirements resulting from the execution of the least precise (i.e., worst) text chunking pipeline, followed by the JAPE pipeline of Fig. 8. The rationale for using the least precise text chunking pipeline is to obtain the largest set of requirements that contain either complex noun phrases or potentially confusing segments for text chunking.

From the above process, we obtained a total of 260 non-conformant requirements statements. These statements were divided into five incremental sets, with cardinalities of $260/5 = 52$, $2 * 260/5 = 104$,..., and 260. We then subjected these sets to the process of Fig. 13. In Fig. 22, we show the execution time growth for the Parser step of the process, implemented via the Stanford Parser as discussed in Section 5.4.1. The "Mark Complex Noun Phrases" step of

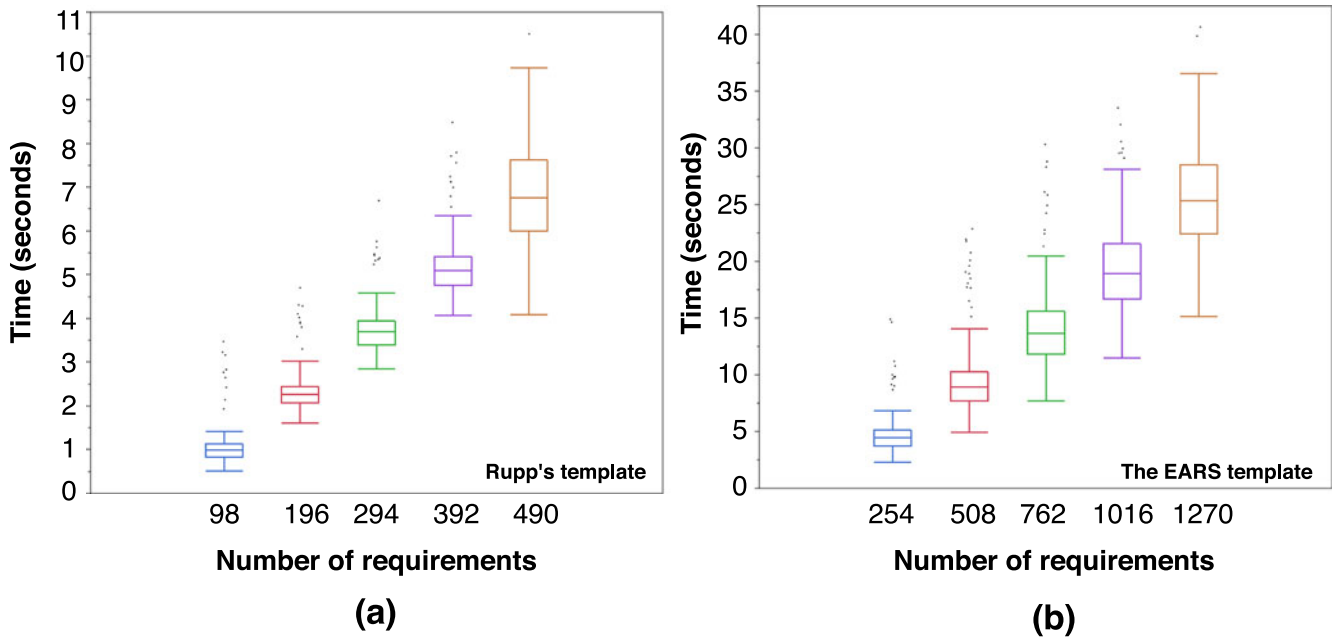

Fig. 21. Execution time growth for checking conformance (a) to Rupp's template and (b) to the EARS template.

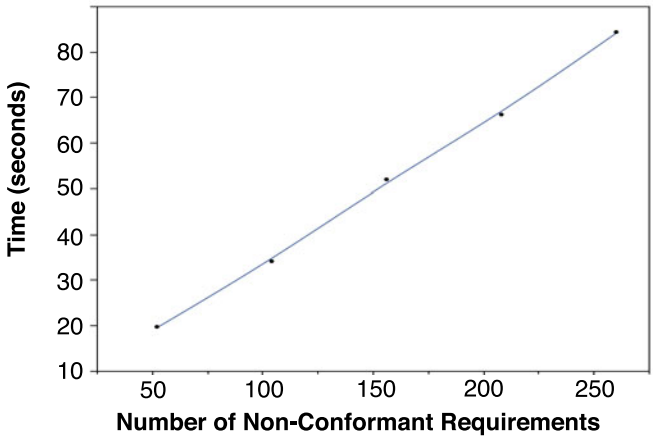

Fig. 22. Execution time growth for the Parser step envisaged by the process of Fig. 13.

the process takes negligible time; the execution time growth for the second run of the pipeline of Fig. 8, i.e., after the detection of complex noun phrases, is consistent with the execution times reported previously in Fig. 21 and hence not shown. As suggested by Fig. 22, while the Parser is computationally more expensive than text chunking, its execution time grows linearly as the number of requirements statements increases.

Benefits from a practitioner's perspective. Unless practitioners find our approach useful, they are unlikely to adopt it. It is therefore important to investigate practitioners' perceptions of the benefits of our approach. To do so, we draw on the qualitative reflections of a group of software and systems engineers at SES TechCom, with whom we have been collaborating on the research presented in this article. The reflections are based on the observations the engineers made throughout their interactions with the researchers and, in the case of the last author, his hands-on experience applying Rupp's template and our tool.

The engineers' primary reason for interest in requirements templates and requirements writing best practices was to reduce ambiguity and vagueness as much as possible. They knew from experience that requirements containing vague terms or expressed using complex sentences were more likely to be the subject of clarification requests during formal requirements reviews. They believed that our tool would help reduce cost and overhead by enabling them to easily identify and address a sizable fraction of readability and vagueness issues before formal reviews took place. In a similar vein, the engineers found our tool to be useful as a training aid for requirements writing and, in the longer term, for establishing harmonized and industry-accepted requirements writing guidelines for their application domain.

Another important benefit the engineers noted with regards to templates is the flexibility that templates provide for requirements clustering. For example, conformance to Rupp's template would enable one to cluster the requirements based on the system name, the process verb being used, the object being processed, and so on. Such clustering facilitates the development of lower-stream artifacts, e.g., when one wants to orient the design or the test cases around requirements clusters. Since our approach automatically delineates different template slots in requirements, it can further automatically cluster the requirements based on the contents of different slots. In this respect, the engineers saw opportunities for cost and effort savings by applying our approach.

Given the limited scale to which our tool has been applied by our industry partner so far and the fact that we have not yet undertaken rigorous user studies, the benefits highlighted above are only suggestive but not conclusive. This being said, the positive experience reported by our industry partner and the fact that our tool could be successfully applied in real projects is promising and makes our approach worthy of future study.

6 Limitations and Threats to Validity

Limitations. Our approach tackles only structural conformance checking. This means that, as long as the syntax of a requirements statement follows what is prescribed by a given template, the statement will be deemed conformant. One cannot detect semantic mismatches using our approach, i.e., the situation where the analyst's choice of requirements type (and hence syntax) does not match the analyst's intent. For example, an analyst may incorrectly frame an interface requirement as a user interaction requirement in Rupp's template or confuse the event-driven and unwanted behavior types in the EARS template. Our approach is unable to detect such problems.

Internal validity. We tried to mitigate all foreseeable factors that could cause confounding effects. Particularly, learning effects from the tool were considered in the case study planning. The requirements experts had no exposure to the tool before finishing requirements specification and glossary terms elicitation. The use of case study data as test data for tool development was strictly avoided.

Another potential threat to internal validity is that the gold standard for evaluation was developed by the researchers. We took several mitigation actions to counter bias during the construction of the gold standard. Our tool was not used during the development of the gold standard to minimize influences on the reasoning of the researchers. A detailed protocol was drawn up and followed by the researchers for classifying requirements into conformant and non-conformant (see Section 5.3). Finally, the gold standard was constructed independently by two researchers, and the differences were then reconciled. Our inter-rater agreement analysis shows substantial or better agreement across our case studies (see Table 3), thus providing confidence about the quality of the gold standard that underlies our evaluation.

Construct validity. The main consideration about construct validity has to do with what it means to conform to a template. The guidelines accompanying generic templates such as Rupp's and EARS are intentionally abstract to ensure wide applicability. Subsequently, a certain degree of interpretation is required when one attempts to operationalize the process for conformance checking.

In our work, we opted for a *relaxed* definition of conformance, merely enforcing proper use of noun phrases and verb phrases. This is the most fundamental and yet the most complex aspect to handle for an automated tool. More restrictions can be considered for conformance, e.g., ensuring absence of vague terms in the requirements. Our tool

indeed already reports many such issues in the form of warnings (see Section 4). However, since such restrictions are easy to enforce automatically, i.e., with a precision and recall of 100 percent, incorporating the restrictions into the definition of conformance provides "easy targets" for an automated tool to deem requirements as non-conformant. This can potentially conceal the mistakes that a tool might make in more complex operations, notably the detection of noun phrases and verb phrases. By having our definition of conformance focused on the most basic criteria, we thus provide conservative estimates about the effectiveness of our approach. Therefore, even better results can be expected when the conformance is more constrained.

External validity. We applied our approach to four case studies from different domains and using two different templates. We have further tried to remain as generic as possible in our treatment of template conformance. We believe that the consistency seen across the results of our case studies provides reasonable confidence about the generalizability of our approach. Further experimentation with requirements documents from other domains with even larger sizes would nevertheless be useful for improving external validity.

With regards to the benefits of our approach for practitioners (Section 5.6), the reflections presented in this article are those of a small number of domain experts working in a single application domain. Broader and more systematic user studies and surveys are necessary in the future for obtaining a more generalizable picture of the benefits and potential drawbacks of our approach.

7 Related Work

As we explained earlier (in Section 1), this article extends our previous work [13], [19], [20] and further provides a comprehensive exposition of our overall approach. In the remainder of this section, we compare with several strands of related work on requirements templates and applications of NLP to requirements.

7.1 Requirements Templates

Numerous requirements templates have been proposed over the years, e.g., by Rolland and Proix [40], by Rupp and Pohl [5], and by Mavin et al. [16]. These templates have been used both in academia and in industrial settings for specifying the requirements of complex systems [21], [41], [42], [43]. Our work in this article does not propose any new templates, but rather devises and empirically validates a scalable, flexible solution for automated checking of conformance to existing templates.

To examine the level of support for templates in existing tools, we conducted a tool review, guided by a recent requirements tool survey [11] and a direct examination of the information sources that this survey builds upon. Specifically, we selected tools whose publicly-available feature descriptions include one or a combination of the following keywords: template, mold, template, syntax checking, linguistic analysis, and natural language processing. We found nine tools that matched this criterion, namely, ARM [44], Cradle [45], DODT [46], LEXIOR [47], QuARS [48], RQA [12], TIGER Pro [49], inteGREAT [50], and visibleThread [51].

Upon closer examination of these tools, we identified two, DODT [46] and RQA [12], offering automated support for checking template conformance. Our work differs from DODT in its focus: DODT concentrates on requirements transformation to achieve template conformance, whereas we focus on non-conformance detection. Further, in contrast to our work, DODT requires a high-quality domain ontology to be developed first, which as we argued earlier, is not a realistic expectation in most industrial development contexts.

As for RQA, while an investigation of the underlying conformance checking algorithm was not possible due to the tool's proprietary nature, we observed that the tool's ability to identify sentence segments was impacted when the glossary terms were left unspecified. Our overall conclusion from the tool review is that although verifying template conformance is an important activity [5], limited automated assistance exists for it. In particular, tool support is lacking for settings where one has little control over the requirements authoring environments used by the analysts, e.g., when multiple organizations are involved in requirements writing. Having multiple organizations is also a factor that contributes to the difficulty of developing a glossary beforehand, thus making existing tools difficult to use. Our tool, in contrast, can be used in settings where little usable knowledge exists about how the requirements were written, and where all that is available for analysis are the requirements statements themselves.

7.2 NLP in Requirements Engineering

NLP has a long history of use in Requirements Engineering due to the prevalent use of natural language in the specification of requirements [52].

Quality assurance processes such as consistency checking represent one of the earliest areas where NLP has been applied to requirements. Gervasi and Nuseibeh [53] use domain-based parsing to enable checking of various formal properties over requirements. Gervasi and Zowghi [54] use part-of-speech tagging and parsing to automatically transform requirements into propositional logic and identify inconsistencies through logical reasoning. Kof et al. [55] develop an ontology-based technique for requirements validation using lemmatization and part-of-speech tagging.

NLP is further a cornerstone of automated requirements traceability, both for identifying inter-dependencies between requirements and for tracing requirements to lower-stream development artifacts such as design and source code. For example, Zou et al. [14] use phrase detection and similarity measures to enhance the requirements trace detection process. Duan and Huang [56] combine similarity measures with clustering-based techniques to group candidate requirements trace results. Sundaram et al. [57] utilize similarity measures in a manner analogous to the above to improve requirements traceability detection. Sultanov and Hayes [58] use tokenization and stemming for establishing traceability between requirements specifications at different levels of abstraction. Torkar et al. [59] and Cleland-Huang et al. [60] provide detailed reviews of the state-of-the-art on requirements traceability, including the range of NLP-based techniques used in this area.

Another area where NLP is commonly used is requirements ambiguity detection. Chantree et al. [61] use morphological analysis for detecting nocuous requirements ambiguities. Kiyavitskaya et al. [62] use parsing for finding various potential syntactic and semantic ambiguities in requirements. Yang et al. [63] focus specifically on anaphoric ambiguities in requirements and develop a heuristics-based approach based on parsing for detecting this class of ambiguities. Femmer et al. [28] use part-of-speech tagging, morphological analysis, and customized dictionaries for detecting patterns that are signs of potential quality defects in requirements.

Transforming NL requirements to models constitutes yet another important application of NLP in Requirements Engineering. Yue et al. [64] present a systematic literature review on the approaches for transforming NL requirements into analysis models, further providing insights about how NLP is utilized in this context.

In addition to the general areas outlined above, where NLP is pervasive, there are various other Requirements Engineering tasks in which the use of NLP has been explored. For example, Zachos and Maiden [65] use text chunking for matching requirements to web-service descriptions. Kiyavitskaya et al. [66] use part-of-speech tagging and parsing for generating various kinds of markup information over regulatory requirements. Holbrook et al. [67] utilize text chunking for assessing requirements satisfaction against different artifacts, such as design. Güldali et al. [68] detect redundancies and implicit relationships between requirements using parsing and similarity measures. Falessi et al. [69] apply and empirically compare different NLP-based strategies for identifying equivalent requirements. Guzman and Maalej [70] employ part-of-speech tagging, stemming and sentiment analysis for synthesizing user opinions about the requirements of mobile applications. Adedjouma et al. [71] use tokenization and NLP pattern matching for detection and resolution of cross-references in legal texts. Arora et al. [13] combine text chunking, similarity measures and clustering for extracting and grouping together requirements glossary terms.

None of the threads outlined above specifically address the problem that we tackle in this article, namely automatic checking of conformance to requirements templates. In addition, text chunking, which is the primary enabling technology for our approach has not yet been exploited widely in Requirements Engineering. Subsequently, limited empirical evidence exists about the effectiveness of text chunking over requirements documents. The empirical evaluation we report in this article provides insights into the effectiveness of various alternative implementations of text chunking, thus paving the way for the wider future application of text chunking in Requirements Engineering.

8 Conclusion

We presented an automated and tool-supported approach for checking conformance to requirements templates. The approach builds on a mature natural language processing technique, known as text chunking. We reported on the application of the approach to four case studies from different domains, using two different templates—Rupp's [5] and EARS [16]. In this context, we evaluated and compared several text chunking solutions in terms of effectiveness and scalability for checking template conformance. The case studies' results indicate that text chunking provides an accurate and scalable basis for template conformance checking. The study further shows that, within the range of alternatives considered, there exist several text chunking solutions with little sensitivity to the presence or absence of a requirements glossary. This makes it possible to automatically check and enforce the correct use of templates even in the (common) case where the glossary is partial or missing.

For future work, we would like to conduct empirical investigations of the impact of templates on the quality of semantic extraction and consistency checking for natural language requirements. Additionally, we plan to investigate whether it is feasible to automatically transform template grammars to NLP conformance checking pipelines. Automating this transformation enables one to directly derive a conformance checker from the design of a template, without the need to understand the NLP technology that underlies conformance checking. Another aspect of our future work is to leverage templates to enable more accurate and automated change analysis in an evolving set of requirements.

Acknowledgments

Funding was provided by the National Research Fund-Luxembourg (FNR/P10/03-Verification and Validation Laboratory and AFR grant FNR-6911386). The authors are grateful to Eero Uuisitalo, Jose Luis de la Vara, and Sunil Nair for their contributions towards our case studies. Authors would further like to thank Arda Goknil, Alistair Mavin, Domenico Bianculli, and Shiva Nejati for their feedback and comments on earlier versions of this article.

References

[1] K. Pohl, *Requirements Engineering-Fundamentals, Principles, and Techniques*, New York, NY, USA: Springer, 2010.

[2] D. Berry, E. Kamsties, and M. Krieger. (2003). From contract drafting to software specification: Linguistic sources of ambiguity, a handbook. [Online]. Available: http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf

[3] C. Denger, J. Dörr, and E. Kamsties, QUASAR: A survey on approaches for writing precise natural language requirements. [Online]. Available: http://publica.fraunhofer.de/eprints/urn:nbn:de:0011-n-77930.pdf, 2001.

[4] S. Withall, *Software Requirement Patterns (Best Practices)*, 1st ed, Redmond, WA, USA: Microsoft, 2007.

[5] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 1st ed, Rocky Nook, Santa Barbara, CA 93103, 2011.

[6] E. Uusitalo, M. Raatikainen, T. Mannisto, and T. Tommila, "Structured natural language requirements in nuclear energy domain towards improving regulatory guidelines," in *Proc. 4th Int. Workshop Requirements Eng. Law*, pp. 67–73, 2011.

[7] C. Rupp and die SOPHISTen, *Requirements-Engineering und-Management: professionelle, iterative Anforderungsanalyse für die Praxis*, Hanser Verlag, München, D-81631, pp. 225–251, 2009.

[8] CESAR: Cost-efficient methods and processes for safety relevant embedded systems. [Online]. Available: http://www.cesarproject.eu, 2012.

[9] OPENCOSS: Open Platform for EvolutioNary Certification Of Safety-critical Systems. [Online]. Available: http://www.opencoss-project.eu, 2012.

[10] The SAREMAN Project: Controlled natural language requirements in the design and analysis of safety critical I&C systems. [Online]. Available: http://www2.vtt.fi/inf/julkaisut/muut/2014/VTT-R-01067-14.pdf, 2014.

[11] J. Carrillo-de-Gea, J. Nicolás, J. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno, "Requirements engineering tools: Capabilities, survey and assessment," *Inform. Softw. Technol.*, vol. 54, no. 10, pp. 1142–1157, 2012.

[12] RQA: The Requirements Quality Analyzer Tool. [Online]. Available: http://www.reusecompany.com/rqa, 2012.

[13] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns," in *Proc. 4th Int. Workshop Requirements Patterns*, 2014, pp. 1–8.

[14] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: a study of term-based enhancement methods," *Empirical Softw. Eng.*, vol. 15, no. 2, pp. 119–146, 2010.

[15] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 2000.

[16] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Proc. 17th IEEE Int. Requirements Eng. Conf.*, 2009, pp. 317–322.

[17] A. Mavin and P. Wilkinson, "Big EARS (the return of "Easy Approach to Requirements Engineering")," in *Proc. 18th IEEE Int. Requirements Eng. Conf.*, 2010, pp. 277–282.

[18] S. Gregory, "Easy EARS: Rapid application of the easy approach to requirements syntax," in *Proc. 19th IEEE Int. Requirements Eng. Conf.*, 2011, pp. 1–2.

[19] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga, "Automatic checking of conformance to requirement boilerplates via text chunking: An industrial case study," in *Proc. 7th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2013, pp. 35–44.

[20] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga, "RUBRIC: A flexible tool for automated checking of conformance to requirement boilerplates," in *Proc. 9th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 599–602.

[21] J. Terzakis, "Reducing requirements defect density by using mentoring to supplement training," *Int. J. Adv. Intell. Syst.*, vol. 6, no. 1-2, pp. 102–111, 2013.

[22] A. Mavin, "Listen, then use EARS," *IEEE Softw.*, vol. 29, no. 2, pp. 17–18 Mar./Apr. 2012.

[23] The Stanford Parser: A statistical parser. [Online]. Available: http://nlp.stanford.edu/software/lex-parser.shtml, 2014.

[24] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, O'Reilly, Sebastopol, CA 95472, 2009.

[25] M. Song, I. Song, and K. Lee, "Automatic extraction for creating a lexical repository of abbreviations in the biomedical literature," in *Proc. 8th Int. Conf. Data Warehousing Knowl. Discovery*, 2006, vol. 4081, pp. 384–393.

[26] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: The Penn Treebank," *Comput. Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[27] H. Cunningham, D. Maynard, K. Bontcheva , V. Tablan, C. Ursu, M. Dimitrov, M. Dowman, and N. Aswani. (2014). Developing Language Processing Components with GATE Version 8 (a User Guide). [Online]. Available: http://gate.ac.uk/sale/tao/tao.pdf

[28] H. Femmer, D.Méndez Fernández, E. Juergens, M. Klose, I. Zimmer, and J. Zimmer, "Rapid requirements checks with requirements smells: Two case studies," in *Proc. 1st Int. Workshop Rapid Continuous Softw. Eng.*, 2014, pp. 10–19.

[29] List of regulatory guides on nuclear safety (YVL). [Online]. Available: http://plus.edilex.fi/stuklex/en/lainsaadanto/luettelo/ydinvoimalaitoso hjeet/, 2013.

[30] GATE ANNIE: A Nearly-New Information Extraction System. [Online]. Available: http://gate.ac.uk/sale/tao/splitch6.html, 2014.

[31] Apache OpenNLP. [Online]. Available: http://opennlp.apache.org, 2013.

[32] Stanford Log-linear Part-Of-Speech Tagger. [Online]. Available: http://nlp.stanford.edu/software/tagger.shtml, 2003.

[33] M. Hepple, "Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers," in *Proc. 38th Annu. Meeting Assoc. Comput. Linguistics*, 2000, pp. 278–277.

[34] Multilingual Noun Phrase Extractor (MuNPEx). [Online]. Available: http://www.semanticsoftware.info/munpex, 2012.

[35] L. Ramshaw and M. Marcus, "Text chunking using transformation-based learning," in *Proc. 3rd ACL Workshop Very Large Corpora*, 1995, pp. 82–94.

[36] M. McGill and G. Salton, *Introduction to Modern Information Retrieval*, New York, NY, USA: McGraw-Hill, 1983.

[37] M. K. Buckland and F. C. Gey, "The relationship between recall and precision," *J. Am. Soc. Inf. Sci.*, vol. 45, no. 1, pp. 12–19, 1994.

[38] J. Cohen, "A coefficient of agreement for nominal scales," *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, 1960.

[39] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*, Belmont, CA, USA: Wadsworth, 1984.

[40] C. Rolland and C. Proix, "A natural language approach for requirements engineering," in *Seminal Contributions to Information Systems Engineering*, J. Bubenko, J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. Sølvberg, Eds. New York, NY, USA: Springer, 2013, pp. 35–55.

[41] O. Daramola, G. Sindre, and T. Stalhane, "Pattern-based security requirements specification using ontologies and boilerplates," in *Proc. 2nd Int. Workshop Requirements Patterns*, Sep. 2012, pp. 54–59.

[42] S. Joseph, M. Schumm, O. Rummel, A. Soska, M. Reschke, J. Mottok, M. Niemetz, and I. Schroll-Decker, "Teaching finite state machines with case method and role play," in *Proc. IEEE Global Eng. Educ. Conf.*, 2013, pp. 1305–1312.

[43] D. Slipper, A. McEwan, and W. Ifill, "A framework for specification of arming system safety functions," in *Proc. 8th IET Int. Syst. Safety Cyber Security Conf.*, 2013, pp. 1–7.

[44] W. Wilson, L. Rosenberg, and L. Hyatt, "Automated analysis of requirement specifications," in *Proc. 19th Int. Conf. Softw. Eng.*, 1997, pp. 161–171.

[45] The Cradle Tool. [Online]. Available: http://www.threesl.com/, 2012.

[46] 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 *Proc. 14th IEEE Int. Symp. Des. Diagnostics Electron. Circuits*, 2011, pp. 271–274.

[47] The LEXIOR Tool. [Online]. Available: http://www.cortim.com/lexior/staticdemo/, 2004.

[48] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami, "The linguistic approach to the natural language requirements quality: Benefit of the use of an automatic tool," in *Proc. 26th Annu. NASA Goddard Softw. Eng. Workshop*, 2001, pp. 97–105.

[49] Tiger Pro: Tool to ingest and elucidate requirements. [Online]. Available: http://www.therightrequirement.com/TigerPro/TigerPro.html, 2008.

[50] The inteGREAT Tool. [Online]. Available: http://www.modernrequirements.com, 2012.

[51] The Visible Thread Tool. [Online]. Available: http://www.visiblethread.com/, 2012.

[52] A. Egemen Yilmaz and I. Berk Yilmaz, "Natural language processing techniques in requirements engineering," in *Proc. Knowl. Eng. Softw. Develop. Life Cycles*, 2011, pp. 21–33.

[53] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements," *Softw.: Prac. Exp.*, vol. 32, no. 2, pp. 113–133, 2002.

[54] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 277–330, 2005.

[55] L. Kof, R. Gacitua, M. Rouncefield, and P. Sawyer, "Ontology and model alignment as a means for requirements validation," in *Proc. 4th IEEE Int. Conf. Semantic Comput.*, 2010, pp. 46–51.

[56] C. Duan and J. Cleland-Huang, "Clustering support for automated tracing," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 244–253.

[57] S. Sundaram, J. Hayes, A. Dekhtyar, and E. Holbrook, "Assessing traceability of software engineering artifacts," *Requirements Eng.*, vol. 15, no. 3, pp. 313–335, 2010.

[58] H. Sultanov and J. Hayes, "Application of swarm techniques to requirements engineering: Requirements tracing," in *Proc. 18th IEEE Int. Requirements Eng. Conf.*, 2010, pp. 211–220.

[59] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. Raja, and K. Kamran, "Requirements traceability: A systematic review and industry case study," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 22, no. 3, pp. 385–433, 2012.

[60] J. Cleland-Huang, O. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proc. Future Softw. Eng.*, 2014, pp. 55–69.

[61] F. Chantree, B. Nuseibeh, A. De Roeck, and A. Willis, "Identifying nocuous ambiguities in natural language requirements," in *Proc. 14th IEEE Int. Requirements Eng. Conf.*, 2006, pp. 59–68.
[62] N. Kiyavitskaya, N. Zeni, L. Mich, and D. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements Eng.*, vol. 13, no. 3, pp. 207–239, 2008.
[63] H. Yang, A. De Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Analysing anaphoric ambiguity in natural language requirements," *Requirements Eng.*, vol. 16, no. 3, pp. 163–189, 2011.
[64] T. Yue, L. 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.
[65] K. Zachos and N. Maiden, "Inventing requirements from software: An empirical investigation with web services," in *Proc. 16th IEEE Int. Requirements Eng. Conf.*, 2008, pp. 145–154.
[66] N. Kiyavitskaya, N. Zeni, T. Breaux, A. Antón, J. Cordy, L. Mich, and J. Mylopoulos, "Automating the extraction of rights and obligations for regulatory compliance," in *Proc. 27th Int. Conf. Conceptual Modeling*, 2008, pp. 154–168.
[67] E. Holbrook, J. Hayes, and A. Dekhtyar, "Toward automating requirements satisfaction assessment," in *Proc. 17th IEEE Int. Requirements Eng. Conf.*, 2009, pp. 149–158.
[68] B. Güldali, S. Sauer, G. Engels, H. Funke, and M. Jahnich, "Semi-automated test planning for e-ID systems by using requirements clustering," in *Proc. 24th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 29–39.
[69] D. Falessi, G. Cantone, and G. Canfora, "Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 18–44 Jan. 2013.
[70] E. Guzman and W. Maalej, "How do users like this feature? a fine grained sentiment analysis of app reviews," in *Proc. 22nd IEEE Int. Requirements Eng. Conf.*, 2014, pp. 153–162.
[71] M. Adedjouma, M. Sabetzadeh, and L. Briand, "Automated detection and resolution of legal cross references: Approach and a study of Luxembourg's legislation," in *Proc. 22nd IEEE Int. Requirements Eng. Conf.*, 2014, pp. 63–72.

Chetan Arora received the MS degree in computer science (software engineering) from the Technical University in Kaiserslautern. He is working towards the PhD degree at the SnT Centre for Security, Reliability, and Trust, the University of Luxembourg. His research interests include requirements engineering, empirical software engineering, and change management in software systems. Prior to that, he worked as a software engineer at the Computer Sciences Corporation (CSC), India. More details can be found at: http://people.svv.lu/arora/

Mehrdad Sabetzadeh received the PhD degree from the University of Toronto, Canada in 2008. He is currently a senior research scientist at the SnT Centre for Security, Reliability, and Trust, the University of Luxembourg, Luxembourg. From 2009 to 2012, he was a member of the research staff at Simula Research Laboratory, Norway. In 2009, he was a visiting researcher at University College London, UK. His research interests are in software engineering with an emphasis on requirements engineering, model-based development, software dependability, and empirical software engineering. He has co-authored more than 40 journal and conference papers, and regularly serves on the program committees of international conferences in the area of software engineering. He has eight years of experience conducting applied research in collaboration with industry partners. His experience spans several industry sectors, including public service, telecommunication, maritime, energy, railways, and automotive. He is a member of the IEEE and IEEE Computer Society. More details can be found at: http://people.svv.lu/sabetzadeh/

Lionel Briand is a professor and FNR PEARL chair in software verification and validation at the SnT Centre for Security, Reliability, and Trust, the University of Luxembourg. He also acts as a vice-director of the centre. He started his career as a software engineer in France (CS Communications & Systems) and has conducted applied research in collaboration with industry for more than 20 years. Until moving to Luxembourg in January 2012, he was heading the Certus center for software verification and validation at Simula Research Laboratory, where he was leading applied research projects in collaboration with industrial partners. Before that, he was on the faculty of the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was a full professor and held the Canada Research Chair (Tier I) in Software Quality Engineering. He has also been the Software Quality Engineering Department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He was recently granted the IEEE Computer Society Harlan Mills award and the IEEE Reliability Society engineer-of-the-year award for his work on model-based verification and testing. His research interests include: software testing and verification, model-driven software development, search-based software engineering, and empirical software engineering. He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the co-editor-in-chief of *Empirical Software Engineering* (Springer) and is a member of the editorial boards of *Systems and Software Modeling* (Springer) and *Software Testing, Verification, and Reliability* (Wiley). He was elevated to the grade of the IEEE fellow for his work on the testing of object-oriented systems. More details can be found at: http://people.svv.lu/briand/

Frank Zimmer received the Diploma Ing. (FH) degree in electrical engineering from the University of Applied Sciences in Kaiserslautern and the DEA Diploma in signal theory and communications from the University of Vigo. He is a senior manager in SES TechCom S.A. heading the system engineering section. His research interests include autonomous agents and multi-agent systems, embedded systems, software radio technologies and systems engineering. He is a member of the IEEE and IEEE Computer Society, and the ACM. Contact him at frank.zimmer@ses.com.