Requirement Boilerplates: Transition from Manually-Enforced to Automatically-Verifiable Natural Language Patterns

Chetan Arora, Mehrdad Sabetzadeh, Lionel C. Briand
SnT Centre for Security, Reliability and Trust
University of Luxembourg, Luxembourg
{chetan.arora, mehrdad.sabetzadeh, lionel.briand}@uni.lu

Frank Zimmer
SES TechCom
9, rue Pierre Werner, Betzdorf, Luxembourg
{frank.zimmer}@ses.com

***Abstract*—By enforcing predefined linguistic patterns on requirements statements, boilerplates serve as an effective tool for mitigating ambiguities and making Natural Language requirements more amenable to automation. For a boilerplate to be effective, one needs to check whether the boilerplate has been properly applied. This should preferably be done automatically, as manual checking of conformance to a boilerplate can be laborious and error prone. In this paper, we present insights into building an automatic solution for checking conformance to requirement boilerplates using Natural Language Processing (NLP). We present a generalizable method for casting requirement boilerplates into automated NLP pattern matchers and reflect on our practical experience implementing automated checkers for two well-known boilerplates in the RE community. We further highlight the use of NLP for identification of several problematic syntactic constructs in requirements which can lead to ambiguities.**

***Index Terms*—Requirement Boilerplates; Natural Language Processing (NLP); Text Chunking; NLP Pattern Matching.**

I. Introduction

Requirement boilerplates, also known as requirement templates or patterns, have long been part of requirement writing best practice [1], [2]. A requirement boilerplate is essentially a Natural Language (NL) pattern that restricts the syntax of requirements sentences to pre-defined linguistic structures. By so doing, boilerplates provide an effective tool for ambiguity reduction and making natural language requirements more amenable for automated analysis [2], [3].

For a (requirement) boilerplate to be effective, it needs to be applied correctly by the requirements analysts. It is therefore important to check whether a given set of requirements indeed conforms to the boilerplate of interest. Conformance to boilerplates is typically verified through a requirements review [4]. Such a review can be time-consuming for large requirements documents. Furthermore, a manual review aimed at checking boilerplate conformance can be error-prone, as reading tens of similarly-structured requirements is a tedious task [4], particularly when the task has to be repeated several times due to requirements changes.

Despite boilerplates being commonly used by practitioners, little tool support exists for checking boilerplate conformance in an automated manner. We have been studying in our previous work [5], [6] the application of Natural Language Processing (NLP) as an enabler to automate the checking of conformance to boilerplates. The automation builds on two key observations: (1) boilerplate patterns are commonly expressed as a grammar, e.g., a BNF grammar and (2) the core linguistic units that constitute a boilerplated requirement can be automatically recognized through a lightweight Natural Language Processing technique known as *text chunking* [7].

Our previous work [5], [6] focused on Rupp's boilerplate [4]. In this paper, we investigate how one can generalize our earlier work and present practical guidelines on how to customize the underlying NLP technology for the implementation of new boilerplates. To illustrate the customization process, we develop a conformance checker for the EARS boilerplate [8]. EARS offers more advanced features than Rupp's boilerplate and has been applied with success in a number of recent industry projects [9], [10]. We further describe a range of generic requirements writing best practices, e.g., avoiding passive voice, that can be verified automatically using NLP.

The remainder of this paper is structured as follows: Section II provides background information on boilerplates and Natural Language Processing (NLP), and further compares our work against related work. Section III explains our approach for boilerplate conformance checking, and extends our approach for Rupp's boilerplate to the EARS boilerplate. Section IV outlines the tool support we have built for our approach. In Section V, we reflect on our experience implementing the approach and discuss its effectiveness. Section VI concludes the paper with a summary, and provides directions for future work.

II. Background

A. *Boilerplates*

Boilerplates provide patterns for the grammatical structure of requirement sentences. When properly followed, boilerplates provide a simple and yet effective way for increasing the quality of requirements by avoiding complex structures, ambiguity, and inconsistency in requirements. Boilerplates further facilitate automated analysis on NL requirements, e.g., transformation of NL requirements to models [3]. A variety of boilerplates have been proposed in the literature. Figure 1 shows two such well-known boilerplates, namely Rupp's boilerplate [3] and the EARS boilerplate [8].

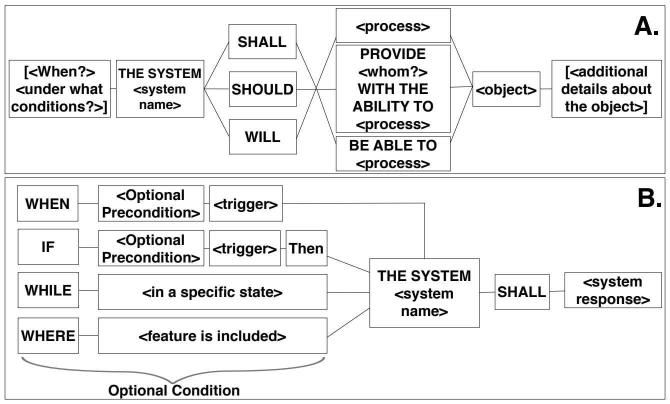


Fig. 1. (A) Rupp's Boilerplate [3] and (B) EARS Boilerplate [8]

Rupp's boilerplate is made up of 6 slots: (1) an optional condition at the beginning of the requirement; (2) the system name; (3) a modal (shall/should/will) specifying how important the requirement is; (4) the required processing functionality; this slot can be phrased in three different forms depending on how the intended functionality is to be rendered; (5) the object for which the functionality is needed; and (6) optional additional details about the object. The three alternatives for the 4th slot are used for capturing the following:

- ⟨**process**⟩: This alternative is used for *autonomous requirements* capturing functions that the system performs independently of interactions with users.
- **PROVIDE** ⟨**whom?**⟩ **WITH THE ABILITY TO** ⟨**process**⟩: This alternative is used for *user interaction requirements* capturing functions that the system provides to specific users.
- **BE ABLE TO** ⟨**process**⟩: This alternative is used for *interface requirements* capturing functions that the system performs to react to trigger events from other systems.

The EARS boilerplate is composed of 4 slots: (1) an optional condition at the beginning; (2) the system name; (3) a modal; and (4) the system response depicting the behavior of the system. The boilerplate provides 5 different requirement types, corresponding to five alternative structures for the first slot.

- *Ubiquitous requirements* have no pre-condition, and are used to format requirements that are always active.
- *Event-driven requirements* begin with **WHEN**, and are initiated only when a trigger is detected.
- *Unwanted behaviour requirements* begin with **IF** followed by a **THEN** before the ⟨**system name**⟩. These requirements are usually used for capturing 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.

B. Natural Language Processing (NLP)

Natural Language Processing (NLP) deals with automatic interpretation, processing and analysis of human-language text. For the purposes of this paper, we are interested specifically in an NLP technique known as *text chunking* [7]. Text chunking provides an efficient and robust alternative to deep (full) parsing for identifying NL constituent segments (chunks). Chunking, unlike deep parsing, does not require any extensive analysis to be performed over the internal structure, roles, or relationships of chunks [11]. Text chunking identifies sentence segments, such as, Noun Phrase (NP), Verb Phrase (VP), Prepositional Phrase (PP). Out of these, the most important segments for an abstract representation of a boilerplate structure are NPs and VPs. A Noun Phrase (NP) is a chunk that can be the subject or object for a verb, with a possible associated adjectival modifier. A Verb Phrase (VP), also known as a verb group, is a chunk that contains a verb with any associated modal, auxiliary, and modifier (often an adverb).

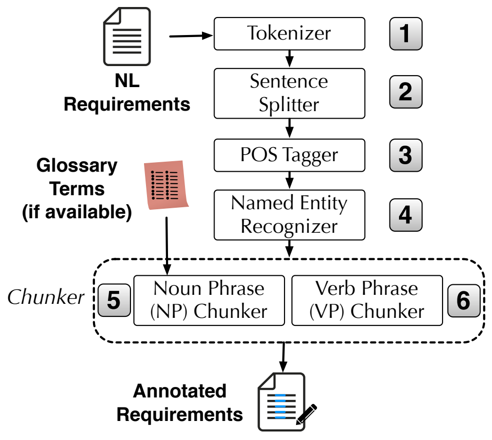


Fig. 2. Text Chunking Pipeline

A text chunker is implemented as a pipeline of various NLP modules, shown in Figure 2. The modules are executed in the sequential order shown in the figure. The first step in the pipeline, Tokenization, breaks up the sentence into units called tokens, such as words, numbers, or punctuation. The next component, Sentence Splitter, segments the text into sentences. The sentences can be split based on full stops or newline characters. Sentence Splitters can distinguish between the full stops for abbreviations and the sentence endings. The POS Tagger, assigns Part-Of-Speech (POS) tags to each token, such as adjectives, adverbs, nouns, verbs. The next step, Name Entity Recognition, attempts to classify text elements into certain pre-defined categories such as name of persons, organizations, locations, monetary values, or certain custom list of keywords specified in a predefined dictionary, commonly known in NLP as a gazetteer. In our case, the gazetteers contain, among other things, the custom list of terms specific to a boilerplate, the list of modals, and the list of potentially ambiguous terms whose use is discouraged in requirements statements. The Chunker partitions text into sequences of semantically related words based on the POS-tags. The NPs and VPs are usually extracted by separate modules, as depicted in the figure. The elements recognized through Named Entity Recognition assist the (NP) Chunker to more accurately detect the NPs, by instructing the chunker to treat these elements as atomic units. The output from the chunking pipeline is an annotated document, with annotations such as, Token, NP, VP required for the boilerplate conformance checking pipeline (Figure 6) discussed in Section III.

C. Pattern Matching in NLP

As we elaborate further in Section III, we use a BNF grammar to represent the structure of a boilerplate. This abstract representation enables the definition of pattern matching rules for checking conformance to the boilerplate. We use the pattern language of the GATE NLP workbench [12] for checking boilerplate conformance. This language, called Java Annotation Patterns Engine (JAPE), is a regular-expression-based pattern matching language. Figure 3 shows an example JAPE script. This script checks conformance to Rupp's Autonomous Requirement type.

```
 1.Phase: MarkConformantSegment
 2.Input: Condition NP VP Token
 3.Options: control = Appelt
 4.
 5.Rule: MarkConformantSegment_RuppAutonomous
 6.Priority: 20
 7. (
 8.    (({Condition}{NP}) |
 9.     ({NP}))
10.    ({VP, VP.startsWithValidMD == "true",
11.     !VP contains {Token.string == "provide"}})
12.    ({NP})
13. ):label --> :label.Conformant_Segment =
14.{explanation = "Matched pattern: Autonomous"}
```

Fig. 3. JAPE script for Rupp's Autonomous Requirement type

JAPE consists of a set of phases, each of which further consist of a set of pattern rules. Figure 3 shows a JAPE phase named "MarkConformantSegment" (line 1), including a single pattern rule "MarkConformantSegment_RuppAutonomous" (line 5). This phase could further be extended to have other pattern rules for user interaction, and interface requirement types in Rupp's boilerplate. The patterns would then be fired based on a priority index (line 6) if they match the same segment of text.

JAPE provides various control 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* [13]. 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 requirement 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 Figure 3, where we want to assign a unique type to a requirement that is most appropriate. *First* means that all individual instances of a match are annotated. This is particularly useful for recognizing sequences of named entities, where we are interested in obtaining the entities individually instead of having all of them merged into one annotation.

D. Related Work

This paper is an extension of our previous work, where we presented an empirical evaluation of the effectiveness of automated boilerplate conformance checking [5], and described our tool support for this activity [6]. Our previous work did not provide a detailed treatment of the underlying NLP technologies but rather focused on covering end-user functions and demonstrating the usefulness of the approach through empirically-rigorous means. In this paper, we complement our previous work with an in-depth explanation of how we use different NLP technologies to achieve our objective. Furthermore, our previous work was centered around Rupp's boilerplate. In this paper, we consider a second boilerplate (EARS) and investigate the extent to which our automation framework is generic and can be made independent of the specific boilerplate being addressed. This is of practical importance as we would like to achieve significant levels of reuse across boilerplates. By analyzing the commonalities and differences between Rupp's and the EARS boilerplates, we further provide insights into the cost of tailoring our automation framework to other boilerplates and the level of reuse that is achieved.

To compare our work with other related strands of work, we previously reviewed several RE tools, guided by a recent RE tool survey [14]. We identified only two other tools directly addressing boilerplates: DODT [15] and RQA [16]. DODT focuses on using requirements transformation for achieving conformance to boilerplates. The tool nevertheless has a strong prerequisite: the availability of a high-quality domain ontology for the system to be developed. In our case studies and arguably in many other industrial projects, developing such an ontology is too costly especially at the initial stages of development when boilerplate conformance checking is most useful. RQA is closer in nature to our work. It provides features for defining boilerplates and for automatically checking conformance. RQA is however most accurate when a domain glossary has been already defined. Unfortunately, a glossary may be non-existing or may have significant omissions at the early stages of development, as suggested in the literature [17]. Our approach, in contrast and as we demonstrate in [5], is highly accurate even in the absence of a glossary.

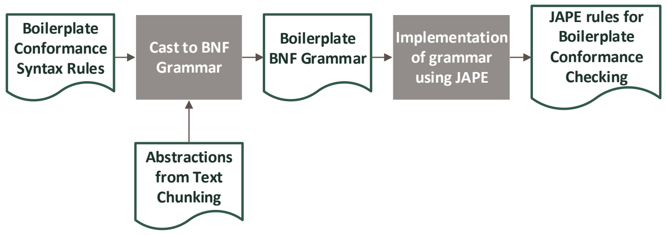

Boilerplate Conformance Syntax Rules
Cast to BNF Grammar
Boilerplate BNF Grammar
Implementation of grammar using JAPE
JAPE rules for Boilerplate Conformance Checking
Abstractions from Text Chunking

Fig. 4. Approach Overview

III. Approach

A. Implementation Details

Figure 4 shows an overview of our approach for developing automated boilerplate conformance checkers. In the first step, boilerplate conformance rules are cast into a BNF grammar, using the main NLP abstractions, namely NPs and VPs from text chunking. Subsequently, conformance rules are implemented using JAPE. In the rest of this section, we elaborate these two steps as well as their inputs and outputs.

1) Boilerplate Grammar: Figure 5 shows BNF grammars for Rupp's and EARS boilerplates. These grammars are interpretations of the boilerplates in terms of NPs (⟨`np`⟩) and VPs (⟨`vp`⟩). We use ⟨`vp-starting-with-modal`⟩ to denote a VP that contains a modal at its starting offset. The valid modals are shown on line R.12 for Rupp's and E.15 for the EARS boilerplate.

In both boilerplates, a requirement can start with an optional condition. There are no detailed syntax guidelines for writing the conditions in Rupp's boilerplate, and only a few recommendations for using key-phrases: `IF` for logical conditions; and `AFTER`, `AS SOON AS`, and `AS LONG AS` for temporal conditions. The EARS boilerplate, on the other hand, differentiates the types of requirements by the ⟨`opt-condition`⟩ rule in E.5 - E.9. In both grammars, one could assume to have a comma to mark the end of the conditional part of the requirement. However, punctuation is commonly forgotten and different people have different styles for punctuation. There may further be multiple commas in the conditional part for separating different conditions. To avoid relying exclusively on the presence of a comma, one can employ heuristics for identifying the conditional segment in a requirement. We identify the system name (an NP) followed by a modal (e.g., SHALL) as the anchor to correctly identify 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 boilerplate, the heuristic capturing the syntax of R is ⟨`conditional-keyword`⟩⟨`sequence-of-tokens`⟩ ⟨`np`⟩⟨`vp-starting-with-modal`⟩⟨`np`⟩⟨`opt-details`⟩. This heuristic assumes that the automated processor correctly delineates the ⟨`np`⟩ corresponding to system name, and subsequent ⟨`vp-starting-with-modal`⟩. Heuristics such as this can be easily defined using JAPE to tag as a condition the sequence of tokens between a conditional keyword and the system name.

The different requirement types in Rupp's boilerplate are distinguished using the ⟨`boilerplate-conformant`⟩ rule in R.1 – R.7 (Figure 5).

For the optional details in Rupp's and the system response in EARS boilerplate, 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 and the boilerplate envisages that such conditions must appear at the beginning and not the end of a requirements statement.

Boilerplate specific keywords, e.g., the modals and the conditional keywords, are entered into a gazetteer. This enables writing more abstract JAPE rules for pattern matching. In particular, future changes to the keywords will not impact the JAPE rules.

2) JAPE Rules for Conformance Checking: To check boilerplate conformance, we execute the pipeline of JAPE scripts shown in Figure 6. The execution of this pipeline follows that of the pipeline in Figure 2. In other words, Tokens (along with their parts of speech), NPs, VPs, and named entities have been already identified and annotated at the time the JAPE scripts in Figure 6 are executed.

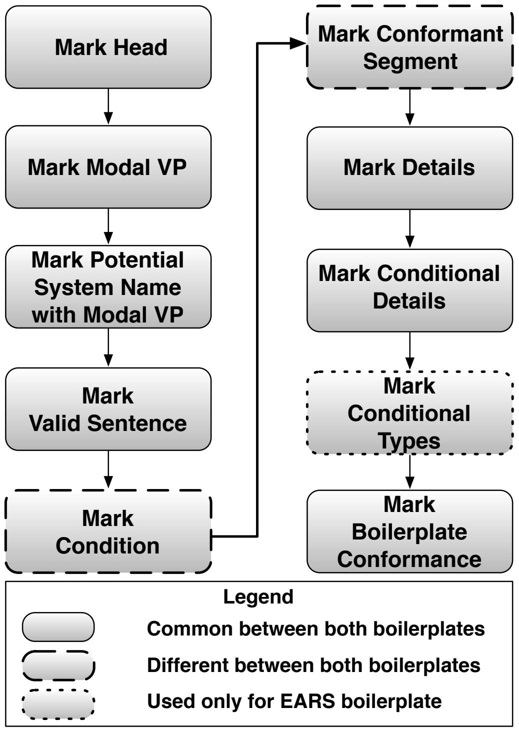

Mark Head
Mark Modal VP
Mark Potential System Name with Modal VP
Mark Valid Sentence
Mark Condition
Mark Conformant Segment
Mark Details
Mark Conditional Details
Mark Conditional Types
Mark Boilerplate Conformance
Legend
Common between both boilerplates
Different between both boilerplates
Used only for EARS boilerplate

Fig. 6. JAPE Pipeline for Boilerplate Conformance Checking

The sequence of JAPE scripts in Figure 6 are as follows:

1) **Mark Head** marks the starting of a requirement sentence.
2) **Mark Modal VP** marks the VP that starts with a modal. A requirements statement typically has only one

```
R.1. <boilerplate-conformant> ::=
R.2.    <opt-condition> <np> <vp-starting-with-modal> <np>
R.3.       <opt-details> |
R.4.    <opt-condition> <np> <modal> "PROVIDE" <np>
R.5.       "WITH THE ABILITY" <infinitive-vp> <np> <opt-details> |
R.6.    <opt-condition> <np> <modal> "BE ABLE" <infinitive-vp>
R.7.       <np> <opt-details>
R.8. <opt-condition> ::=  "" |
R.9.    <conditional-keyword> <non-punctuation-token>* ","
R.10. <opt-details> ::=  "" |
R.11.    <token-sequence-without-subordinate-conjunctions>
R.12. <modal> ::= "SHALL" | "SHOULD" | "WILL"
R.13. <conditional-keyword> ::= "IF" | "AFTER" | "AS SOON AS" |
R.14.    "AS LONG AS"
R.15.
```

```
E.1. <boilerplate-conformant> ::=
E.2.    <np> <vp-starting-with-modal> <system-response> |
E.3.    <opt-condition> <np> <vp-starting-with-modal> <system-
E.4. response>
E.5. <opt-condition> ::=  "" |
E.6.    "WHEN" <opt-precondition> <trigger> |
E.7.    "IF"   <opt-precondition> <trigger> "THEN" |
E.8.    "WHILE" <specific-state> |
E.9.    "WHERE" <feature-included>
E.10. <opt-precondition> | <trigger>|<specific-state>|<feature-
E.11. included> ::=
E.12.    <np><vp><non-punctuation-token>* ","
E.13. <system-response> ::=
E.14.    <token-sequence-without-subordinate-conjunctions>
E.15. <modal> ::= "SHALL"
```

Fig. 5. BNF grammar for Rupp's boilerplate (left), and EARS boilerplate (right)

modal. If more than one modal is found, the first modal is annotated and a warning is generated to bring the presence of multiple modals to the user's attention.

3) **Mark Potential System Name with Modal VP** tags the NP preceding the modal as the potential system name, and annotates both the NP and the VP containing the modal as an anchor for delineating the conditional slot.
4) **Mark Valid Sentences** annotates all the requirement sentences containing the above anchor. The anchor either has to be at the beginning of a sentence or must be preceded with a segment that starts with a conditional keyword. Sentences that meet this constraint are marked as "Valid".
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 JAPE scripts for marking conditions are different for Rupp's and the EARS boilerplate, as the syntactic structure of the conditions differs from one boilerplate to the other.
6) **Mark Conformant Segment** marks as "Conformant Segment" the segment of a sentence that complies with the boilerplate in a Valid requirement sentence. Since the conformance rules, i.e., R.1–R.7 and E.1–E.4, are different across the two boilerplates, the JAPE scripts for conformance checking are also different.
7) **Mark Details** annotates the optional details in Rupp's boilerplate as well as the system response in the EARS boilerplate, using a shared annotation, named "Details".
8) **Mark Conditional Details** checks the Details segments for any subordinate conjunctions. If a subordinate conjunction is detected, the requirement is marked as "Non-Conformant". This is because both boilerplates mandate that the conditional part should appear at the beginning and not in the middle or at the end.
9) **Mark Conditional Types** is exclusive to the EARS boilerplate for delineating the different sub parts of the Condition slot, such as, trigger and specific states. This script further annotates the requirement type based on the type of the condition used.
10) **Mark Boilerplate Conformance** marks as "Conformant" any Valid sentences that contain a conformant segment, excluding those sentences that have been deemed Non-Conformant (due to the presence of conditionals at positions other than the beginning). Any requirement without a Conformant or a Non-Conformant annotation after this stage will be marked as "Non-Conformant".

B. *Checking NL Best Practices*

In addition to checking boilerplate conformance, we utilize NLP for detecting and warning about several potentially problematic constructs that have been discouraged by requirements writing best practices. We build upon the best practices by Berry et al [1]. Figure 7 lists and exemplifies potentially problematic constructs that we can detect automatically. The automation is done through JAPE in a manner very similar to how boilerplate conformance is checked. To illustrate, we show in Figure 8 the JAPE script that generates warnings about pronouns in requirements statements ("Warn_Pronoun" annotation in Figure 7). If not used properly, pronouns can lead to referential ambiguity [1]. The JAPE rule in the figure matches the POS tags of the Tokens in a given document against Penn Treebank's [18] POS tags for pronouns. The matched tags are: personal pronoun (PRP), possessive pronoun (PRP$), wh-pronoun (WP), or possessive wh-pronoun (WP$). If matched, the Tokens will be labeled with the "Warn_Pronoun" annotation, and can be reviewed at a later stage by the requirement analysts for possible ambiguities.

```
Rule: MarkPronouns
(
 {Token.category == "PRP"} |
 {Token.category == "PRP$"} |
 {Token.category == "WP"} |
 {Token.category == "WP$"}
):label --> :label.Warn_Pronoun =
{explanation = "Pronouns can cause ambiguity."}
```

Fig. 8. JAPE script for detecting pronouns

Tag	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. `Note:` 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. `Note:` The inclusive or exclusive nature of 'or' is not clear.
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. `Note:` 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. `Note:` 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. `Note:` Support: providing the operator with the ability to define? `Note:` up to: 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. `Note:` It is not clear whether the actor is S&T module, database, or another agent.
Warn_Complex_Sentence	Using multiple conjunctions in the same requirement 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. `Note:` 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 . . . `Note:` 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. `Note:` The adverb periodically is ambiguous.
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 `Note:` This is unclear without appropriate domain knowledge.

Fig. 7. List of potentially problematic constructs detected through NLP

IV. Tool Support

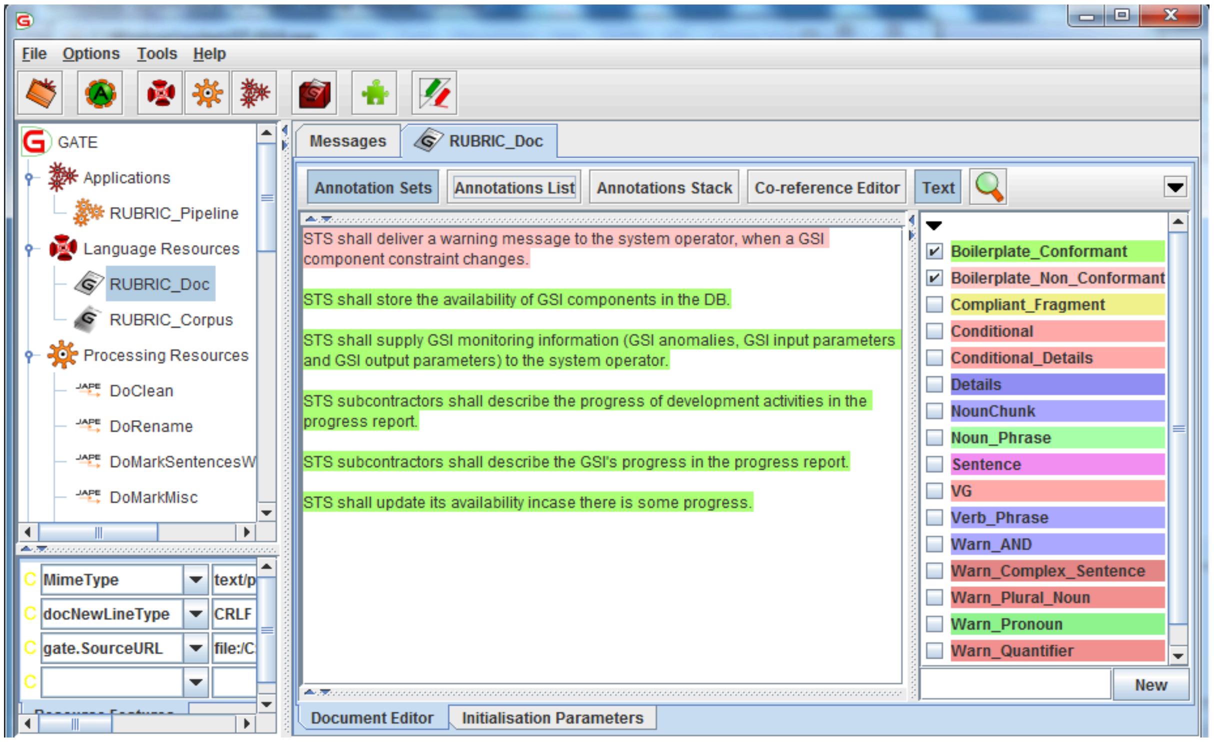
We have developed a tool named RUBRIC (ReqUirements BoileRplate sanIty Checker), within the GATE NLP framework [12]. RUBRIC enables analysts to automatically check conformance to requirement boilerplates, and to detect potentially problematic linguistic constructs. The initial version of RUBRIC [6] covered only Rupp's boilerplate. The tool has since been extended to cover the EARS boilerplate as well. The core of RUBRIC consists of 30 JAPE scripts with approximately 800 lines of code. Out of these, 26 scripts are common between the implementation of Rupp's and the EARS boilerplates, suggesting that a large fraction of our implementation can be reused from one boilerplate to another. RUBRIC further includes 10 customizable lists containing keywords used by the scripts (e.g., boilerplate fixed terms, vague terms, conjunctions, and modals). Out of these, 8 are common between Rupp's and the EARS boilerplate implementations. To facilitate using RUBRIC in a production environment, we have implemented two plugins, one for IBM DOORS (http://www.ibm.com/software/products/en/ratidoor) and the other for Enterprise Architect (EA) (http://www.sparxsystems.com.au). These plugins automatically export the requirements written in DOORS and EA and execute the RUBRIC NLP pipeline over the exported requirements. The IBM DOORS plugin has been implemented using a simple script in DOORS Extension Language (DXL), and the Enterprise Architect plugin has been implemented in C# (approximately 1000 lines of code). A demo version of RUBRIC along with a screencast illustrating its use over Rupp's boilerplate are available at:

sites.google.com/site/rubricnlp/

V. Discussion

In this section, we discuss some important practical considerations regarding the implementation of our approach.

Choice of NLP Framework. The first consideration concerns the choice of the NLP framework to build on. Choosing GATE as the basis for our implementation was motivated by



STS shall deliver a warning message to the system operator, when a GSI component constraint changes.

STS shall store the availability of GSI components in the DB.

STS shall supply GSI monitoring information (GSI anomalies, GSI input parameters and GSI output parameters) to the system operator.

STS subcontractors shall describe the progress of development activities in the progress report.

STS subcontractors shall describe the GSI's progress in the progress report.

STS shall update its availability incase there is some progress.

MimeType	text/p
docNewLineType	CRLF
gate.SourceURL	file:/C

- ✓ Boilerplate_Conformant
- ✓ Boilerplate_Non_Conformant
- Compliant_Fragment
- Conditional
- Conditional_Details
- Details
- NounChunk
- Noun_Phrase
- Sentence
- VG
- Verb_Phrase
- Warn_AND
- Warn_Complex_Sentence
- Warn_Plural_Noun
- Warn_Pronoun
- Warn_Quantifier

Fig. 9. Snapshot of RUBRIC

(1) the availability of several alternative implementations for each NLP module in GATE; (2) GATE's unified annotation scheme, enabling one to seamlessly combine NLP modules built independently of one another and by different research groups; and (3) GATE's extensive and well-documented API providing the flexibility to manipulate GATE at a fine level of granularity and outside GATE's own user interface. This API is also crucial for integrating GATE into external tools such as IBM DOORS and Enterprise Architect.

Choice of NLP Modules. As noted above, GATE offers multiple alternative implementations for each NLP module. For example, consider the pipeline of Figure 2 for text chunking. The users have at their disposal several concrete implementations to choose from for each step in the pipeline. For example, there are numerous alternatives for Step 3 (POS Tagger) in GATE, including Stanford POS Tagger [19], OpenNLP POS Tagger [20], and ANNIE POS Tagger [21]. Similarly there are alternative implementations available for Step 4 (Named Entity Recognizer), including ANNIE Named Entity (NE) Transducer [22] and OpenNLP Name Finder [22]. For Step 5 (NP Chunker), one can choose from the Multilingual Noun Phrase Extractor (MuNPEx) [23], OpenNLP (NP) Chunker [20], andRamshaw & Marcus (RM) NP Chunker [7]. Finally, for Step 6 (VP Chunker), one can choose between the OpenNLP (VP) Chunker [20] and ANNIE Verb Group Chunker [22].

An important consideration when choosing from the alternatives is which combination of alternatives yields the most accurate results. In our previous work [5], we evaluated the accuracy of different combinations for boilerplate conformance checking. Using the best combination of alternatives, our approach yields, over an industrial case study with 380 requirements and *without* a glossary, a precision of 92% and a recall of 96%. This combination, however, might comprise of alternatives that are overly sensitive to the particular configuration in that combination. In other words, an alternative may produce excellent results when used alongside other specific alternatives in other steps, but poor results otherwise. Such alternatives are not desired and we may instead prefer reliable alternatives that perform consistently well in all combinations. To mitigate this risk, we performed a sensitivity analysis to identify the most reliable alternatives. For example, for Step 5 in the pipeline of Figure 2, OpenNLP NP Chunker [20] was deemed to be the most reliable alternative. A detailed explanation of the most reliable alternative(s) for all the steps is given in [5].

We have further evaluated our approach for conformance checking against the EARS boilerplate using 72 example requirements that we found in various online EARS tutorials [8], [9], [10]. Since all the requirements conformed to the EARS boilerplate, we could only check for possible false positives (i.e., requirements that conform to the boilerplate but are found to be non-conformant by our tool) but *not* false negatives (i.e., requirements that do not conform to the boilerplate but are found to be conformant by our tool). We ran the most reliable configuration from our previous work on the EARS-conformant requirements. All the requirements were deemed conformant by our tool and no false positives were seen.

Reliance on glossary. Avoiding reliance on a glossary during boilerplate conformance checking was one of the primary motives for the development of our approach. We observed (in our empirical evaluation of [5]) that the accuracy of our approach is reduced only by a slight margin when the glossary is missing. This was due to the advanced NP chunkers that accurately detect complex NPs without needing a glossary. We further observed that the extracted NPs provide a good basis for recommendation of glossary terms [24].

New boilerplates and reuse. Extending our work to the EARS boilerplate provides evidence about the extent to which our approach and automation framework can be easily tailored to other boilerplates, through the extensive reuse of implementation resources, such as JAPE scripts. While extending our implementation, the foremost task was to understand the commonalities between the EARS and Rupp's boilerplates, particularly in relation to the conditions, functionality types, and requirement details. As we argued in Section IV, in our implementation of the EARS boilerplate, we reused a significant fraction of the implementation of Rupp's boilerplate. This reuse saved substantial effort from the implementation standpoint. For example, determining the system name and the modal verb anchor (discussed in Section III-A2) is common between Rupp's and the EARS boilerplate, hence making it possible for us to reuse the original JAPE scripts. Differences between boilerplate-specific keywords were abstracted away using gazetteers, with minimal impact on the pattern matching rules. While further validation is required to demonstrate generalizability, the above observation makes us optimistic about the possibility of reuse across different boilerplates and the possibility to have an automatic transition from a boilerplate grammar to pattern matching scripts for boilerplate conformance checking.

VI. Conclusion

We presented our solution for automatically checking conformance to requirement boilerplates and other requirements writing best practices. Our exposition in this paper focused on how one can bring together different NLP technologies, particularly text chunking and NLP pattern matching, to make such automation possible. We described the technical steps for implementing an automated conformance checker and reported on our experience extending our existing solution to accommodate a new boilerplate, reusing existing implementation resources to the maximum extent possible. We highlight the implementation abstractions enabling such a high level of reuse across boilerplates, and also characterize the effort saved through reuse. For future work, we would like to automate the transformation of boilerplate grammars to JAPE rules, which is currently done in a systematic manner but manually. We further need to investigate the practical utility of our approach by conducting larger case studies with a diverse set of boilerplates, and user studies to compare against manual identification of boilerplate conformance. Additionally, we plan to use the structure enforced by boilerplates as a basis for extraction of semantic information and for automated change analysis in an evolving set of requirements.

References

[1] D. Berry, E. Kamsties, and M. Krieger, *From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity, A Handbook*, 2003. [Online]. Available: http://se.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf
[2] K. Pohl, *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010.
[3] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 1st ed. Rocky Nook, 2011.
[4] C. Rupp and die SOPHISTen, *Requirements-Engineering und -Management: professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser, 2009.
[5] 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 *ESEM'13*, 2013.
[6] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga, "RUBRIC: A flexible tool for automated checking of conformance to requirement boilerplates," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013.
[7] L. Ramshaw and M. Marcus, "Text chunking using transformation-based learning," in *3rd ACL Workshop on Very Large Corpora*, 1995.
[8] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 317–322.
[9] A. Mavin and P. Wilkinson, "Big ears (the return of" easy approach to requirements engineering")," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010, pp. 277–282.
[10] S. Gregory, "Easy EARS: Rapid application of the easy approach to requirements syntax," 2011.
[11] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Prentice Hall, 2000.
[12] "GATE NLP Workbench," http://gate.ac.uk/.
[13] Cunningham et al, "Developing Language Processing Components with GATE Version 7 (a User Guide)." [Online]. Available: http://gate.ac.uk/sale/tao/tao.pdf
[14] J. C. de Gea, J. Nicolás, J. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno, "Requirements engineering tools: Capabilities, survey and assessment," *Information & Software Technology*, vol. 54, no. 10, 2012.
[15] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, H. Zojer, and C. Panis, "DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development," in *14th IEEE Intl. Symp. on Design and Diagnostics of Electronic Circuits Systems*, 2011.
[16] "RQA: The Requirements Quality Analyzer Tool." [Online]. Available: http://www.reusecompany.com/rqa
[17] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: a study of term-based enhancement methods," *Empirical Softw. Engg.*, vol. 15, no. 2, 2010.
[18] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.
[19] "Stanford Log-linear Part-Of-Speech Tagger." [Online]. Available: http://nlp.stanford.edu/software/tagger.shtml
[20] "Apache's OpenNLP." [Online]. Available: http://opennlp.apache.org
[21] M. Hepple, "Independence and commitment: assumptions for rapid training and execution of rule-based POS taggers," in *38th Annual Meeting on Association for Computational Linguistics*, 2000.
[22] "GATE's ANNIE: a Nearly-New Information Extraction System." [Online]. Available: http://gate.ac.uk/sale/tao/splitch6.html
[23] "Multilingual Noun Phrase Extractor (MuNPEx)." [Online]. Available: http://www.semanticsoftware.info/munpex
[24] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Improving requirements glossary construction via clustering: Approach and industrial case studies," in *ESEM'14*, 2014.