2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement

Automatic Checking of Conformance to Requirement Boilerplates via Text Chunking: An Industrial Case Study

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

Frank Zimmer, Raul Gnaga
SES TechCom
9, rue Pierre Werner, Betzdorf, Luxembourg
{*frank.zimmer,raul.gnaga*}*@ses.com*

Abstract—Context. **Boilerplates have long been used in Requirements Engineering (RE) to increase the precision of natural language requirements and to avoid ambiguity problems caused by unrestricted natural language. When boilerplates are used, an important quality assurance task is to verify that the requirements indeed conform to the boilerplates.**
Objective. **If done manually, checking conformance to boilerplates is laborious, presenting a particular challenge when the task has to be repeated multiple times in response to requirements changes. Our objective is to provide automation for checking conformance to boilerplates using a Natural Language Processing (NLP) technique, called *Text Chunking*, and to empirically validate the effectiveness of the automation.**
Method. **We use an exploratory case study, conducted in an industrial setting, as the basis for our empirical investigation.**
Results. **We present a generalizable and tool-supported approach for boilerplate conformance checking. We report on the application of our approach to the requirements document for a major software component in the satellite domain. We compare alternative text chunking solutions and argue about their effectiveness for boilerplate conformance checking.**
Conclusion. **Our results indicate that: (1) text chunking provides a robust and accurate basis for checking conformance to boilerplates; and (2) the effectiveness of boilerplate conformance checking based on text chunking 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.**

***Keywords*-Requirement Boilerplates; Natural Language Processing (NLP); Text Chunking; Case Study Research.**

I. Introduction

Natural Language (NL) is arguably the most common way for specifying requirements. NL tends to be easier to understand for many stakeholders since little training is required. 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.

Sentence boilerplates, also known as templates, molds, or patterns [2], [1], provide an effective tool for reducing ambiguity in NL requirements and for making them more amenable to automated analysis. A boilerplate organizes the syntactic structure of a requirement statement into a number of pre-defined slots. Figure 1 shows one of the well-known requirement boilerplates, due to Rupp [3]. The boilerplate 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 further in Section III); (5) the object for which the functionality is needed; and (6) optional additional details about the object.

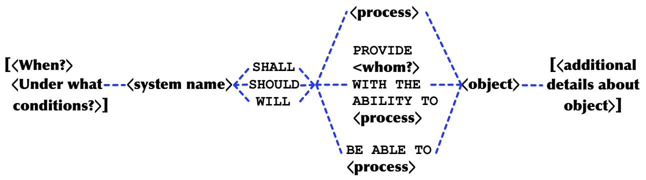


Figure 1. Rupp's Requirement Boilerplate [3]

We show in Figure 2 two example requirements: R_1, which conforms to Rupp's boilerplate, and R_2, which does not. For R_1, we show the different sentence segments and how they correspond to the slots in Rupp's boilerplate. The fixed elements of the boilerplate are written in capital letters.

$$R_1\text{:}\overbrace{\text{The Surveillance and Tracking module}}^{\text{<system name>}}\text{ SHALL PROVIDE }\overbrace{\text{the system administrator}}^{\text{<whom?>}}$$
$$\text{WITH THE ABILITY TO }\underbrace{\text{monitor}}_{\text{<process>}}\underbrace{\text{system configuration changes}}_{\text{<object>}}\underbrace{\text{posted to the database}}_{\text{<additional details>}}.$$

R_2: For each communication channel type, the system needs to maintain a configurable timeout parameter.

Figure 2. Example requirements: R_1 conforms to the boilerplate in Figure 1; R_2 does not. Fixed elements are written in capital letters.

When boilerplates are used, an important quality assurance task is to verify that the requirements conform to the boilerplates. If done manually, this task can be time-consuming and tedious [3]. 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 boilerplates can be automated with relative ease. For example, consider R_1 in Figure 2 and assume that `Surveillance and Tracking module` (system component), `system administrator` (agent), `monitor` (domain verb), and `system configuration change` (event) are already marked as glossary terms. In this situation, an automated tool can verify conformance by checking that R_1 is composed of a subset of glossary terms (or close derivatives thereof) and a subset of fixed boilerplate elements, put together in a sequence that is admissible by the boilerplate. The fixed elements may be leveraged for distinguishing different boilerplate slots. For example, whatever appears between `PROVIDE` and `WITH THE ABILITY TO` in R_1 has to correspond to the $\langle$**whom?**$\rangle$ (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 boilerplate slots. For example, in Rupp's boilerplate, $\langle$**process**$\rangle$, $\langle$**object**$\rangle$, and $\langle$**additional details**$\rangle$ come in a sequence without any fixed elements in 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 issue is that even when a slot falls in between fixed elements, e.g., $\langle$**whom?**$\rangle$, and is thus easily distinguishable, there is no way to verify that the content of the slot is acceptable. For example, in R_1, we may accept `system administrator` as correctly filling $\langle$**whom?**$\rangle$, but we may be unwilling to accept a complex phrase, e.g., `the person handling system administration`, for the slot.

A. Motivation and Contributions

Support for boilerplates already exists in commercial RE tools [4], indicating an industrial interest in boilerplates and naturally the automation of quality assurance activities surrounding boilerplates. For example, the RQA tool [5], provides features for automatically verifying correct use of boilerplates; 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 RE project, this activity often concludes after (rather than before) basic activities such as checking conformance to boilerplates. Further, the literature suggests that requirements glossaries may remain incomplete throughout the whole project [6], thus providing only partial coverage of the glossary terms. This implies that automated techniques for checking conformance to boilerplates would have limited effectiveness if they rely too heavily on glossary terms.

This paper is motivated by the need to provide an automated and generalizable solution for Boilerplate Conformance Checking (hereafter, BCC) *without* reliance on a glossary. To this end, we make the following three contributions:

(1) We propose a simple and effective approach for automation of BCC using an existing Natural Language Processing (NLP) technique, called *text chunking*. Text chunking provides a fast and robust alternative to full parsing for identifying sentence segments (chunks), without having to perform expensive analysis over the chunks' internal structure, roles, or relationships [7]. These chunks, most notably Noun Phrases (NPs) and Verb Phrases (VPs), provide a suitable level of abstraction over natural language for performing BCC. Figure 3(a) shows what a correct chunking of R_1 in Figure 2 would yield. NPs and VPs provide a straightforward way for characterizing boilerplate slots (Section III).

(2) We report on the design and execution of a case study in the satellite domain, as part of which we compare the effectiveness of several text chunking solutions for BCC. The comparison is based on the requirements document (composed of 380 requirements) for a major software component in a satellite system. The requirements were written by industry professionals using Rupp's boilerplate (Figure 1) after they underwent training on the boilerplate. Results indicate that: first, text chunking provides an accurate basis for BCC; and second, the effectiveness of text chunking is not compromised if the glossary terms are left unspecified.

(3) We provide tool support for BCC. Our tool, named RUBRIC (ReqUirements BoileRplate sanIty Checker), enables analysts to automatically check conformance to Rupp's boilerplate and to obtain diagnostics about potentially problematic syntactic constructs in requirements statements. The tool can be extended to accommodate additional boilerplates.

B. Research Questions (RQs)

Our case study aims to answer the following RQs:

RQ1. What text chunking solution leads to best BCC results? As we discuss in more detail throughout the paper, a text chunker is made up of a set of NLP modules, executed in a particular 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 best results. Precision, recall, and F-measure are used as metrics for assessing accuracy.

RQ2. How effective is BCC (based on text chunking) at identifying non-conformance defects? Our ultimate goal from BCC is to identify requirements that do not conform to a given boilerplate and take remedial action. With RQ1 establishing what text chunking solution works best, one has to further investigate if the best solution is worthwhile to be used by practitioners. This is what RQ2 aims to address.

RQ3. Is BCC (based on text chunking) scalable? In a realistic setting, one can be faced with hundreds and sometimes thousands of requirements. RQ3 aims to establish whether a BCC approach based on text chunking will run within reasonable time.

RQ4. Does the absence of a glossary have an impact on the effectiveness of BCC? At the time that BCC is performed, the glossary terms may be unknown or incomplete. RQ4 aims to investigate how the absence of a glossary affects BCC based on text chunking.

In Section VII, we provide answers to these RQs based on the results of our case study.

II. Background

A. Related Work

Requirement Boilerplates. Requirement boilerplates are one of the simplest and most effective ways for increasing the quality of requirements specifications. Withall [8] notes two key advantages for boilerplates: (1) being able to collect more complete requirements in less time and (2) increasing the level of consistency between requirements. Boilerplates are particularly commonplace in safety-critical applications due to the strong need to avoid ambiguities [3]. Further, boilerplates, and more generally restricted NL, are prerequisites for automatic transformation of NL requirements to formal specifications and models [9], [10]. Numerous boilerplates for NL requirements have been proposed over the years, e.g., by Rolland and Proix [11], Rupp [3], and Mavin [12]. Our work in this paper does not aim to develop new boilerplates, but rather to devise and empirically validate a solution for automatic boilerplate conformance checking.

Use of NLP in RE. NLP has a long history of application in RE due to the prevalent use of NL in the specification of requirements. Yilmaz & Yilmaz [13] provide an overview of how NLP is applied for improving various quality attributes in requirements, e.g., consistency and verifiability. Numerous threads exist in the RE literature addressing the detection and resolution of specific types of requirement defects using NLP. Kof et al [14] describe an NLP approach for detection of inconsistencies and omissions by extracting domain-specific abstractions and relations from NL requirements and building an ontology. Gervasi & Nuseibeh [15] and Gervasi & Zowghi [16] apply a combination of NLP and formal logic for detection of semantic conflicts and developing a general quality control process for NL requirements. Yang et al [17] propose an automatic framework for analyzing anaphoric ambiguities caused by disagreements over the interpretation of pronouns. None of these threads specifically address the problem that we tackle in this paper, namely automatic checking of conformance to boilerplates. In addition, these earlier threads do not explore the use of text chunking. In this sense, our work provides insights into an NLP technique that has not been adequately investigated in an RE context.

RE Tools with Syntax Checking Capabilities. To examine the level of automated support for BCC in existing RE tools, we conducted a tool review, guided by a recent RE tool survey [4] 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, boilerplate, syntax checking, linguistic analysis, and Natural Language Processing. We found nine tools that matched this criterion. Upon closer examination of these tools, we identified two, DODT [18] and RQA [5], offering automated support for BCC. Our work differs from DODT in its focus: they concentrate on requirements transformation to achieve boilerplate conformance, whereas we focus on non-conformance detection. Further, in contrast to our work, DODT strictly requires a (high-quality) domain ontology to be developed first, which as we argued earlier, is not always feasible. As for RQA, while an investigation of the underlying BCC 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 boilerplate conformance is an important activity [3], limited automated assistance exists for it. In particular, tool support for BCC is lacking for the setting where one has little control over the requirements authoring environments used by the analysts, e.g., when multiple organizations are involved in requirements writing. This means that little usable knowledge may exist about *how* the requirements were written; all that is available for analysis are the outcomes, i.e., the written requirements. Our tool can be used in such a setting as it relies on no particular requirement authoring environment and works directly on the requirements' textual content.

B. Text Chunking

Text chunking is the process of decomposing a sentence into non-overlapping segments [7]. 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 for 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 requirement statement R_1 in Figure 2 would yield the segments shown in Figure 3(a).

Chunking provides a more robust and efficient alternative to full parsing, where one does not require the parse tree for analysis. Further, since chunkers produce a flat structure, their output is easier to manipulate that parse trees which can have arbitrary depths, as shown in Figure 3(b) for R_1.

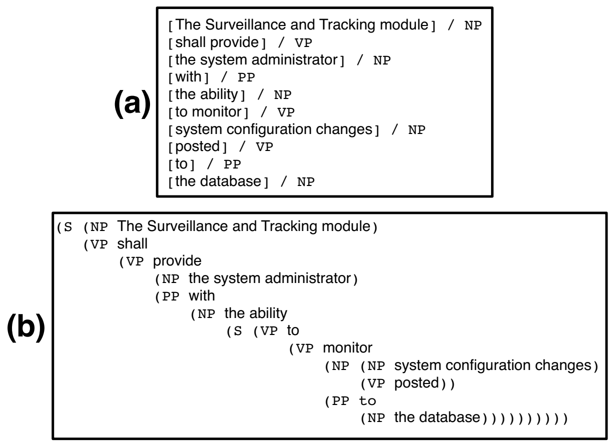

(a)
```
[The Surveillance and Tracking module] / NP
[shall provide] / VP
[the system administrator] / NP
[with] / PP
[the ability] / NP
[to monitor] / VP
[system configuration changes] / NP
[posted] / VP
[to] / PP
[the database] / NP
```
(b)
```
(S (NP The Surveillance and Tracking module)
   (VP shall
       (VP provide
           (NP the system administrator)
           (PP with
               (NP the ability
                   (S (VP to
                          (VP monitor
                              (NP (NP system configuration changes)
                                  (VP posted))
                              (PP to
                                  (NP the database)))))))))
```

Figure 3. Sentence chunks (a) versus full parse tree (b)

Similar to most NLP applications, a text chunker is a pipeline of NLP modules, running in a sequence over an input document. The (abstract) pipeline for chunking is shown in Figure 4. As we explain in Section V-C, 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, nouns, verb. Most POS Taggers use the Penn Treebank tagset [7]. 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 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 BCC is the subject of RQ4 (see Section VII). An input document after going through the pipeline of Figure 4 will have annotations for tokens, sentences, parts-of-speech, named entities, noun phrases and verb phrases. These annotations can be used for automating BCC, as we discuss in Section III.

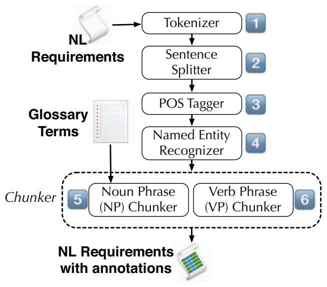


Figure 4. NLP pipeline for text chunking

III. BCC via Text Chunking

In this section, we describe how we use the annotations produced by (an instantiation of) the pipeline in Figure 4 for automating BCC. We ground our discussion on Rupp's boilerplate, which is also the boilerplate underlying our industrial case study. Rupp's boilerplate, shown earlier in Figure 1 and briefly discussed in Section I, distinguishes three types of requirements [3]:

Autonomous: Requirements that capture functionality that the system offers independently of interactions with users.

User interaction: Requirements that document functionality that the system provides to specific users.

Interface: Requirements that the system performs as reactions to trigger events from other systems.

The type of a requirement is reflected by the process slot in Figure 1. The three alternative syntactic structures for this slot, from top to bottom, are respectively used for expressing autonomous, user interaction, and interface requirements. In Figure 5, we provide a BNF grammar that interprets Rupp's boilerplate in terms of the tags generated by the pipeline of Figure 4. For simplicity, in the grammar, we abstract from nested tags. For example, on line 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".

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

Figure 5. BNF grammar for Rupp's boilerplate

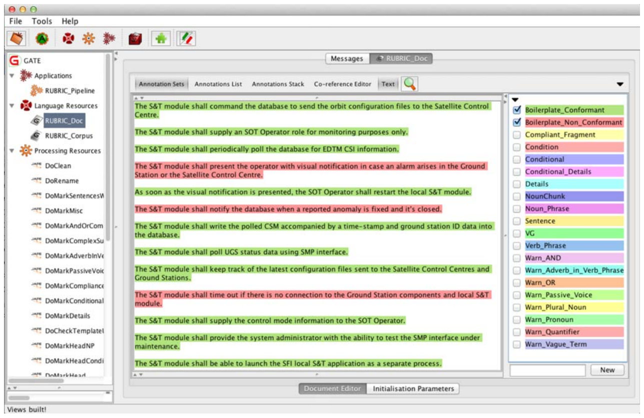
All requirements can start with an optional condition. The boilerplate does not provide a detailed syntax for the conditions, but recommends the use of the following conditional key-phrases: IF for logical conditions; and AFTER, AS SOON AS, and AS LONG AS for temporal conditions. In the grammar, one can assume to have a comma to mark the end of the conditional part. However, this can sometimes be constraining because it is easy to forget the comma and further, different people have different styles for punctuation. To avoid relying exclusively on the presence of a comma, one can employ heuristics to identify the conditional segment. For example, consider the following requirement $R =$ *"AS SOON AS the Surveillance and Tracking module receives a power outage signal the Surveillance and Tracking module SHALL record a warning in the system alert log file."* The heuristic capturing the syntax of R is ⟨conditional-keyword⟩⟨np⟩⟨vp⟩⟨np⟩⟨np⟩⟨vp-starting-with-modal⟩⟨np⟩⟨opt-details⟩. A similar heuristic can be used when the conditional part is expressed in passive voice. Note that a correct matching of R against the heuristic would require the automated processor to correctly distinguished two consecutive ⟨np⟩s in the absence of punctuation. Sophisticated text chunkers, such as the ones we use in our analysis have this capability.

Lastly, for the optional details, 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 requirement statements. Checking conformance to the syntactic rules in Figure 5 can be easily implemented using a regular expression recognizer tool. We use a regular expression language called JAPE [19] to implement the rules in Figure 5, as we discuss in Section V-C.

IV. Tool Support

We have built a tool named RUBRIC (ReqUirements BoileRplate sanIty Checker), enabling requirements analysts to verify conformance to requirement boilerplates. Our current implementation only covers Rupp's boilerplate, but it can be easily generalized to other boilerplates. The tool further generates warnings for several potentially problematic constructs in requirements statements, including use of vague terms, complex combinations of conjunctions, adverbs in verb phrases, quantifiers, pronouns, and passive voice. Berry et al [2] give a detailed treatment of these constructs and the issues they can give rise to in requirements.

RUBRIC has been developed as an application for the GATE workbench (http://gate.ac.uk/), which is an open-source NLP infrastructure. In addition to providing implementations for a large variety of NL processing resources, GATE offers an intuitive user interface for text markup. We use this interface for showing the diagnostics resulting from checking conformance to boilerplates and other requirements writing best practices. A snapshot of the interface after running the RUBRIC application on a small exemplar is shown in Figure 6. 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 examplar 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.



The S&T module shall command the database to send the orbit configuration files to the Satellite Control Centre.

The S&T module shall supply an SOT Operator role for monitoring purposes only.

The S&T module shall periodically poll the database for EDTM CSI information.

The S&T module shall present the operator with visual notification in case an alarm arises in the Ground Station or the Satellite Control Centre.

As soon as the visual notification is presented, the SOT Operator shall restart the local S&T module.

The S&T module shall notify the database when a reported anomaly is fixed and it's closed.

The S&T module shall write the polled CSM accompanied by a time-stamp and ground station ID data into the database.

The S&T module shall poll UGS status data using SMP interface.

The S&T module shall keep track of the latest configuration files sent to the Satellite Control Centres and Ground Stations.

The S&T module shall time out if there is no connection to the Ground Station components and local S&T module.

The S&T module shall supply the control mode information to the SOT Operator.

The S&T module shall provide the system administrator with the ability to test the SMP interface under maintenance.

The S&T module shall be able to launch the SFI local S&T application as a separate process.

Figure 6. Snapshot of markup generated by RUBRIC

To enable using GATE in a realistic RE setting, we have implemented a plugin to export requirements from the Enterprise Architect tool (http://www.sparxsystems.com.au), used by our industry partner for requirements and design specification. Similar plugins can be developed for other environments, e.g., IBM DOORS (www.ibm.com/software/products/ca/en/ratidoor/).

V. Case Study Design

Our case study is an attempt to analyze and gain insights about the fitness of particular technique (text chunking) for a particular purpose (checking conformance to sentence boilerplates) in a particular situation (software requirements development). Our case study can thus be viewed as an *exploratory case study* [20]. An important decision in our study concerns the boilerplate or set of boilerplates to use. We choose Rupp's boilerplate as the basis for our study due to the simplicity and flexibility of this boilerplate as well as the availability of good training resources for it [1], [3]. The case study was conducted at SES TechCom – a leading provider of satellite communications services worldwide. Below, we describe our case study design.

A. Case Selection

Our case selection was driven by three main criteria:

Cr1. Access to a system in early stages of requirements specification. We were interested in systems that had already undergone preliminary analysis, but whose technical requirements had not yet been formally documented. Rewriting existing specifications and bringing them into conformance with a boilerplate was deemed infeasible as any such rewriting attempt had to be carried out by the domain experts to avoid threats to validity. This needed time and resources beyond what the experts could spend (see Cr2).

Cr2. Securing adequate level of participation from the experts. The case study involved training and regular interaction with the experts. Further, since the chosen boilerplate was being applied by the experts for the first time, there was a learning curve which could prolong requirements writing. Hence, the availability of the experts throughout the case study was an important factor.

Cr3. Ensuring reasonable scale and syntactic diversity in requirements. Since the case had to be chosen early into or prior to the requirements writing phase (see Cr1), we had to rely on the judgment of the experts regarding scale and diversity. To this end, we needed measures to convey to the experts what was desirable for a case. For scale, the (expected) number of requirements provides a good measure. For diversity, numerous factors can play a role, including the types of requirements, the number and background of the stakeholders and requirements analysts, and the (expected) number of human and software agents in the system. Regarding requirement types, since we planned to use Rupp's boilerplate, it seemed logical to define the types according to this boilerplate, i.e., autonomous, user interaction, and interface, as explained in Section III. We presented these considerations to our industry partner. For size, a desired minimum of 100 requirements was stated. For the diversity measures, the researchers conveyed no specific thresholds, other than stating a desire to maximize the number of stakeholders and achieve coverage (to the extent that could be foreseen) of all the requirements types in Rupp's boilerplates.

The timing of our case study coincided with the development of a new suite of software applications for a ground-to-satellite communication project. Within this project, our industry partner chose for the case study an application whose function is to control and provide real-time fault diagnosis for software, hardware, and network components.

Cr1 and Cr2 above could be ascertained in advance. The degree to which Cr3 was met could only be established post-mortem in our analysis, explained in Section VI.

B. Data Collection Procedure

Data collection was performed in two phases: (1) requirements writing and identification of glossary terms; and (2) inspection of the requirements resulting from the first phase in order to identify the ones that do not conform to Rupp's boilerplate. Phase 1 was carried out entirely by the experts at the partner company following the training they received from the researchers. Phase 2 was carried out entirely by the researchers following the completion of phase 1.

Phase 1. The output from this phase is a set of requirements that must conform to Rupp's boilerplate along with a set of terms for the requirements glossary. This phase was preceded by two half-day training sessions. In the first session, the experts were presented with best practices for mitigating ambiguity and vagueness in NL requirements. In particular, this session provided a detailed treatment of Rupp's boilerplate. Further, the experts were presented with heuristics for extracting domain concepts from requirements documents and guidelines for building glossaries and domain models. The second training session was hands-on. During this session, the experts reviewed a selection of 30 requirements from a past project and restated them using Rupp's boilerplate. The experts further participated in a collaborative exercise with the researchers where they read an excerpt of an existing system description and extracted domain concepts from the excerpt.

Following training, the researchers had no control over how the experts applied Rupp's boilerplate to the case study system and how they identified the glossary terms. Interaction between the experts and researchers was limited to clarifications about the training material. With regards to glossary terms, two considerations were specifically highlighted to the experts: 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 RQ4, 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.

Phase 2. This phase is concerned with manually inspecting the requirements to identify which requirements (written in phase 1) conform to Rupp's Boilerplate and which ones do not. The data collected in this phase is used for calculating the effectiveness of automated conformance checking (see Section V-C). To increase the validity of the results, this phase was independently conducted by two different inspectors (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.

C. 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 Rupp's boilerplate from those that do not.

1) NLP Pipeline Configuration: To instantiate the pipeline of Figure 4, one needs to choose for each step in the pipeline a specific implementation from the set of existing alternative implementations. For our study, we concentrate on the implementations available in the GATE workbench (discussed earlier in Section IV). This decision is motivated by two factors: First, GATE brings together a large collection of well-established NLP resources via plugins, thus enabling us to experiment with several alternative solutions. Second, the plugins in GATE can be easily integrated with one another through the unified text annotation mechanism offered through the GATE infrastructure. This significantly reduces the risk that poor results would follow from an NLP pipeline due to interface incompatibilities between its components.

While our BCC approach depends primarily on the text chunking modules, i.e., Steps 5 and 6 in Figure 4, these two steps rely on the annotations produced in Steps 3 and 4; therefore, the performance of Steps 5 and 6 ultimately relates to that of Steps 3 and 4. For this reason, we consider in our analysis not only different instantiation of Steps 5 and 6 but also different instantiation of Steps 3 and 4. Similarly, Steps 1 and 2 impact Steps 3 and 4; however, we choose fixed implementations for the first two steps, since these steps are simpler and significant variations between different implementations are unlikely. Below, we list the different alternatives considered for each of the steps in Figure 4:

Step 1 (fixed): ANNIE English Tokenizer [21]

Step 2 (fixed): ANNIE Sentence Splitter [21]

Step 3 (3 alternatives): Stanford POS Tagger [22], OpenNLP POS Tagger [23], ANNIE POS Tagger [24]

Step 4 (2 alternatives): ANNIE Named Entity (NE) Transducer [21], OpenNLP Name Finder [21]

Step 5 (3 alternatives): Multilingual Noun Phrase Extractor (MuNPEx) [25], OpenNLP (NP) Chunker [23], Ramshaw & Marcus (RM) Noun Phrase Chunker [26]

Step 6 (2 alternatives): OpenNLP (VP) Chunker [23], ANNIE Verb Group Chunker [21]

Note that most of the above modules are based on machine learning. For all modules requiring training, we use the default training data (for English) that is distributed with GATE Release 7.1 [19].

The noun and verb phrase tags produced in Steps 5-6 are the basis for checking conformance to Rupp's boilerplate, as described in Section III. For Step 5, there is a choice to be made as to whether to include the glossary as an input. Therefore, we have a total of $2 \times (3 \times 2 \times 3 \times 2) = 72$ different configurations to compare.

The annotations produced by each configuration is fed to a pattern matcher, implemented in GATE's JAPE language [19]. The matcher implements the grammar in Figure 5 along with the two additional heuristics, described in Section III that deal with the conditional part in the absence of punctuation. The results are subsequently analyzed based on the effectiveness measures discussed next.

2) Metrics for Measuring Effectiveness: Our analysis of effectiveness is based on *precision* and *recall*. These classification accuracy metrics are widely used in many areas, e.g., information retrieval [27], 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: Conformant to Rupp's boilerplate and Non-Conformant to Rupp's boilerplate. The simple confusion matrix shown in Figure 7 captures the possible errors that an automated conformance checker can make in the classification.

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

Figure 7. Confusion Matrix

Precision is a metric for quality (low number of false positives) and is defined as $\text{TP}/(\text{TP}+\text{FP})$. Recall is a metric for coverage (low number of false negatives) and is defined as $\text{TP}/(\text{TP}+\text{FN})$. Depending on the context, one may want to place more emphasis on either precision or recall. In our study, we expect recall to be the primary factor, as it is easier for analysts to rule out a small number of false positives rather than have to go through a large document in search of the false negatives. To be able to compare the precision and recall values across different NLP component configurations, we use a third metric called F-measure [27] which computes the harmonic mean of recall and precision. We use a definition of F-measure, known as F_2-measure, which gives more weight to recall than precision, and is defined as: $3 \times \text{Precision} \times \text{Recall}/(2 \times \text{Precision} + \text{Recall})$.

VI. Results

This section describes the case study results. The example requirements we use in this section for illustration are based on the actual requirements in the study but have been altered from their original form to protect confidentiality.

A. Requirements and Glossary

The experts at the industry partner delivered 380 requirements for the case study system, accompanied by 127 glossary terms. The manual inspection procedure, described in Section V, deemed 243 of the requirements to be conformant to Rupp's boilerplate (64%) and the remaining 137 to be non-conformant (36%). The conformant requirements covered all the three requirement types in Rupp's boilerplate (see Cr3 in Section V-A), although coverage was not balanced. Specifically, of the 243 conformant requirements, 206 were of the autonomous type; 35 were of the user interaction type; and 2 of the interface type. As for the 137 cases of non-conformance, the significant majority were explained by the following causes:

Minor deviations, e.g., *"The S&T component shall provide the user with a function to view the network status."*

Enumerations, e.g., *"The state of the S&T module can be standby, active, or degraded."*

Non-conformant conditions, e.g., *"The S&T module shall load a new configuration from the database as soon as the module receives a reloading request."*, where the condition appears at the end, as opposed to the beginning.

Language problems and grammatical mistakes accounted for a small fraction of non-conformances (approx. 2%). With regards to the most recurring non-conformance classes described above, one can easily address minor deviations and bring the affected requirements into conformance. Requirements concerning enumerations are more naturally expressed using a different sentence structure, similar to the example given above. We therefore did not recommend the experts to bring enumeration requirements into conformance with Rupp's boilerplate. In the case of non-conformant conditions, a closer examination was conducted to determine if one could naturally fit the affected requirements into Rupp's boilerplate, 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 conditional key-phrases recommended by Rupp's boilerplate. 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 key-phrase is not among the ones recommended by the boilerplate. In light of this observation, we proposed to the industry partner to extend the conditional key-phrases with `WITHIN`. The inspection results were left intact for analytical purposes. These results are used for computing the classification accuracy metrics for different BCC solutions, as we describe next.

B. Classification Accuracy Metrics and Execution Time

The 72 NLP pipelines discussed in Section V-C were each executed on the 380 requirements in the case study. For each pipeline, the classification accuracy metrics were computed as well as the time it took to execute the pipeline. In Table I, we show the best five pipelines in terms of F_2-measure. Descriptive statistics for precision, recall, F_2-measure, and execution time are given in the form of box plots in Figure 8.

Table I
TOP 5 PIPELINES FOR F_2-MEASURE

Row	POS Tagger	Name Finder	Glossary?	NP Chunker	VP Chunker	Time (seconds)	Precision	Recall	F_2measure
1	Stanford POS Tagger	ANNIE NE Transducer	Yes	MuNPEx	ANNIE VG Chunker	10.48	0.91	0.99	0.96
2	Stanford POS Tagger	OpenNLP Name Finder	Yes	MuNPEx	ANNIE VG Chunker	10.7	0.91	0.99	0.96
3	Stanford POS Tagger	ANNIE NE Transducer	Yes	RM NP Chunker	ANNIE VG Chunker	16.81	0.92	0.96	0.95
4	Stanford POS Tagger	OpenNLP Name Finder	Yes	RM NP Chunker	ANNIE VG Chunker	16.46	0.92	0.96	0.95
5	Stanford POS Tagger	ANNIE NE Transducer	No	RM NP Chunker	ANNIE VG Chunker	17.67	0.92	0.96	0.95

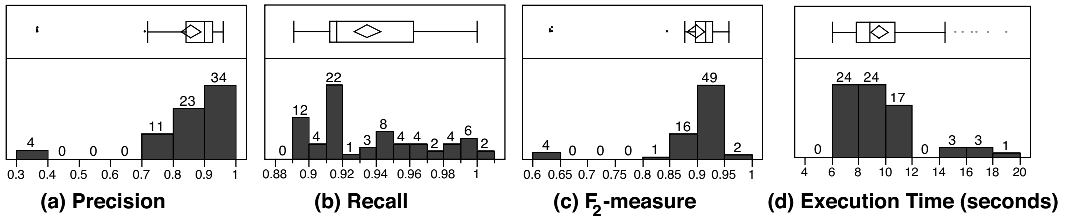


Figure 8. Box plots for classification accuracy metrics and execution time

All experiments were conducted on a 2.3 GHz Intel Core i7 with 8Gb of memory.

There are five outliers in precision, leading to five outliers in F_2-measure. All outliers have two features in common: the use of MuNPEx NP Chunker and the absence of a glossary. In four of the cases, 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 POS tags produced by either ANNIE POS Tagger or OpenNLP POS Tagger. The result is that MuNPEx NP Chunker cannot correctly recognize the ⟨**system name**⟩ slot. The fifth case is very close to being within 1.5 IQR of the lower quartile (and thus is barely an outlier). In this case, MuNPEx NP Chunker is combined with the Stanford POS Tagger. The same mistake is not observed in the POS tags; however, the overall performance is still low. In Section VII, we discuss the behavior of MuNPEx NP Chunker when addressing RQ1.

VII. DISCUSSION

RQ1. The text chunking solutions that lead to best BCC results in the presence and absence of a glossary are respectively given by rows 1 and 5 of Table I. In choosing a solution, one further has 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 [28]. 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 case. In Figure 9, we show the regression tree for F_2-measure in our case study. 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 on classification accuracy 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.

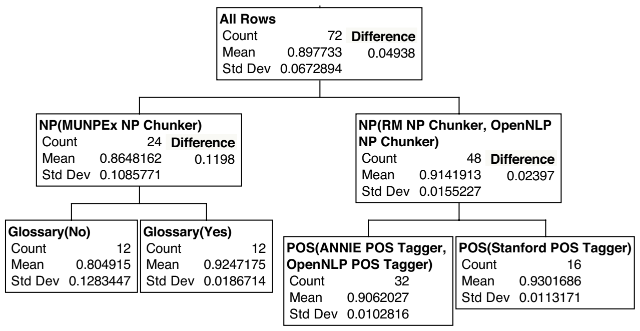


Figure 9. Regression tree for F_2-measure

As shown by the tree, the most critical decision concerns the choice of the NP Chunker module. MuNPEx NP Chunker performs well in the presence of a (good) glossary but does poorly on average where a glossary is absent. In contrast, RM NP Chunker and OpenNLP chunker introduce much less variation if chosen as the NP Chunker. Within the RM NP Chunker / OpenNLP Chunker branch, the most critical decision concerns the POS Tagger, with the Stanford POS Tagger achieving a higher mean. Based on our analysis, we therefore suggest that the MuNPEx NP Chunker should not be used in the absence of a good glossary. Combinations of Stanford POS Tagger with either RM NP Chunker or Open NLP Chunker produce the best overall results with little variation. The top five F_2-measure results in Table I indicate a preference for RM NP Chunker. Lastly, we note

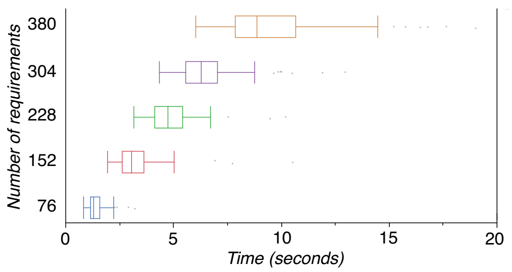


Figure 10. Execution time growth

that F_2-measure shows little sensitivity to the selected VP Chunker used and both OpenNLP VP Chunker and ANNIE VG Chunker lead to good results.

RQ2. One can approach this RQ quantitatively by developing a cost-benefit model; however, the development of such a model requires data that we cannot have from a single case study. Instead, we answer this question qualitatively by arguing about the trade-off between the following: (1) the level of effort that it takes to do an ideal (i.e. fully accurate) manual inspection; and (2) the consequences posed by the false positives and false negatives in automated BCC. As demonstrated by the analysis for RQ1, a suitable NLP configuration can be expected to produce a small percentage of false positives and false negatives. In our case study, the best pipeline with glossary (row 1 of Table I) produces 14 false positives and 2 false negatives. For the best outcome without a glossary (row 5), there are 12 false positives and 5 false negatives. Given the small number of false positives and the ease at which they can be excluded, the question therefore is whether the false negatives can be tolerated. We expect this to be so, as such a small fraction of false negatives is unlikely to introduce major quality concerns. Further, for a large set of requirements in a real project, a manual inspection conducted under time pressure is unlikely to produce perfect results; hence, the false negatives seem a reasonable price to pay for automation.

RQ3. Scalability involves mainly to show that text chunking can be performed 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 randomized the order of the 380 requirements in the case study and then created five requirement sets: the first 76 requirements, the first 2*76 requirements and so on. We examined the execution time for each pipeline against the growing number of requirements. In Figure 10, we show the execution time plots for each of these requirement sets. The result show a linear growth pattern, thus providing confidence that automatic BCC based on text chunking will scale linearly. Hence, for much larger requirements sets, given such linear relation and the fact that 380 requirements take only a few seconds to process, the approach should still be practical.

RQ4. As shown in Figure 9, within the RM NP Chunker / OpenNLP Chunker branch, the presence or absence of a glossary does not contribute significantly to the uncertainty of results. In other words, knowing the glossary terms does not lead to major accuracy gains in automated BCC.

VIII. Threats to Validity

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 industry experts had no exposure to the tool until they finished requirements writing. The use of case study data as test data for tool development was strictly avoided. The tool was not used in the manual inspection of the requirements to mitigate influences on the reasoning of the inspectors. Finally, as also stated earlier, to ensure the quality of the manual inspection results, the inspection process was first carried out independently by two individuals, and then the discrepancies were discussed and resolved.

Construct Validity. The main remark about construct validity has to do with what it means to conform to a boilerplate. The guidelines accompanying generic boilerplates such as Rupp's, 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 IV). However, since such restrictions are easy to enforce automatically (with a precision and recall of 100%), 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 conformance is more constrained.

External Validity. Generalizability is always a concern in any singular case study, including ours. While we have tried to remain as generic as possible in our treatment of boilerplate conformance and have based our work on a reasonably large and diverse case, more research is required to ensure that our approach remains effective in other application domains and for other boilerplates.

IX. Conclusion

We presented an automated and tool-supported approach for checking conformance to requirement boilerplates. The approach builds on a mature Natural Language Processing technique, known as text chunking. We reported on the application of the approach to an industrial case study in the satellite domain. In this context, we evaluated and compared several text chunking solutions in terms of effectiveness and scalability for checking boilerplate conformance. The case study indicates that text chunking provides an accurate and scalable basis for boilerplate conformance checking. The study further shows that, within the range of alternatives considered, the majority of text chunking solutions have little sensitivity to the presence or absence of a requirements glossary. This makes it possible to automatically check and enforce the correct use of boilerplates even when the glossary is partial or missing.

For future work, we would like to conduct empirical investigations of the impact of boilerplates on the quality of semantic extraction and consistency checking for natural language requirements. Additionally, we plan to investigate the robustness of our approach in a setting where no conscious attempt has been made to conform to a given boilerplate. Another aspect of our future work is to leverage boilerplates to enable more accurate and automated change analysis in an evolving set of requirements.

Acknowledgments. We gratefully acknowledge funding from the National Research Fund - Luxembourg (FNR/P10/03 - Verification and Validation Laboratory). We thank Domenico Bianculli and Shiva Nejati for their comments on a previous draft of this paper.

References

[1] K. Pohl, *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010.

[2] 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

[3] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 1st ed. Rocky Nook, 2011.

[4] 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.

[5] "RQA: The Requirements Quality Analyzer Tool." [Online]. Available: http://www.reusecompany.com/rqa

[6] 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.

[7] 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.

[8] S. Withall, *Software Requirement Patterns (Best Practices)*, 1st ed. Microsoft, 2007.

[9] 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

[10] T. Yue, L. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requir. Eng.*, vol. 16, no. 2, 2011.

[11] C. Rolland and C. Proix, "A natural language approach for requirements engineering," in *4th Conf. on Advanced Information Systems Engineering*, 1992.

[12] A. Mavin, "Listen, then use EARS," *IEEE Software*, vol. 29, no. 2, 2012.

[13] A. E. Yilmaz and I. B. Yilmaz, "Natural language processing techniques in requirements engineering," in *Knowledge Engineering for Software Development Life Cycles*, M. Ramachandran, Ed. IGI Global, 2011.

[14] L. Kof, R. Gacitua, M. Rouncefield, and P. Sawyer, "Ontology and model alignment as a means for requirements validation," in *4th IEEE Intl. Conf. on Semantic Computing*, 2010.

[15] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements," *Software: Practice and Experience*, vol. 32, no. 2, 2002.

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

[17] H. Yang, A. De Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Analysing anaphoric ambiguity in natural language requirements," *Requir. Eng.*, vol. 16, no. 3, 2011.

[18] 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.

[19] Cunningham et al, "Developing Language Processing Components with GATE Version 7 (a User Guide)." [Online]. Available: http://gate.ac.uk/sale/tao/tao.pdf

[20] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, 2009.

[21] "GATE's ANNIE: a Nearly-New Information Extraction System." [Online]. Available: http://gate.ac.uk/sale/tao/splitch6.html

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

[23] "Apache's OpenNLP." [Online]. Available: http://opennlp.apache.org

[24] 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.

[25] "Multilingual Noun Phrase Extractor (MuNPEx)." [Online]. Available: http://www.semanticsoftware.info/munpex

[26] L. Ramshaw and M. Marcus, "Text chunking using transformation-based learning," in *3rd ACL Workshop on Very Large Corpora*, 1995.

[27] M. McGill and G. Salton, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[28] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Wadsworth and Brooks, 1984.