Automated Analysis of Natural Language Textual Specifications

Conformance and Non-Conformance with Requirement Templates (RTs)

by

SHIVANI BALWANI
202111022

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

July, 2023

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

Shivani

Shivani Balwani

Certificate

This is to certify that the thesis work entitled **Automated Analysis of Natural Language Textual Specifications** has been carried out by **Shivani Balwani (202111022)** for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.

17/7/23

Dr. Saurabh Tiwari
Thesis Supervisor

Acknowledgments

I wish to express my sincere gratitude to all those who have contributed to the completion of this work. Firstly, I would like to thank my supervisor, Prof. Saurabh Tiwari, for his valuable guidance, constructive feedback and unwavering support throughout my research work. Without his continuous guidance and trust, this thesis would not have reached its conclusion. He was always there to lead me towards the solution whenever I got stuck.

Furthermore, I am grateful to the Indian Space Research Organisation (ISRO) for its funding of this research project. I would also like to acknowledge the Dhirubhai Ambani Institute of Information and Communication Technology for providing me with the resources, facilities, and opportunities necessary to carry out this research. Additionally, I would like to extend my gratitude to Prof. Sourish Dasgupta for his invaluable guidance during the experimentation and analysis phase of my thesis, as his expertise has been pivotal in shaping the direction of my research.

Finally, I would like to express my gratitude to my family and friends for their constant support, encouragement, and understanding throughout this journey.

Contents

Abstract v

List of Principal Symbols and Acronyms vi

List of Tables vii

List of Figures viii

1 Introduction 1
1.1 Natural Language Textual Specifications 1
1.2 Objective and Problem Description 2
1.3 Thesis contribution 2
1.3.1 Automated Conformance Checking with RTs 3
1.3.2 Automated Recommendations for Non-Conforming FRs . . 3
1.3.3 Automated Assessment of Testability for NFRs 3
1.3.4 Tool Support 3
1.4 Overall Analysis of the Requirement Document: Flow Graph 3
1.5 Organisation of the Thesis 5

2 Background of Requirement Templates 7
2.1 Analysis of RTs 7
2.2 EARS template 9
2.3 RUPPs template 10

3 Literature Review 13
3.1 NLP in Requirement Engineering 13
3.2 Identification of Ambiguities in NL text 14
3.3 Requirement Templates 15
3.4 Non-Functional Requirements 16

4 Automated Conformance Checking of Functional Requirements to RTs 17
4.1 NLP Pipeline to process FR 17
4.2 JAPE Rules for Conformance Checking 19

5 Automated Recommendation System for Non-Conformance Requirements 23
5.1 JAPE Rules for Providing Recommendations 23

6 Automated Analysis of Non-Functional Requirements 27
6.1 Testability of Non-Functional Requirements 27
6.2 Acceptance criteria 28
6.3 NLP Pipeline to process NFRs 29
6.4 JAPE Rules for detecting the Acceptance criteria in NFRs 31

7 Tool Support 33
7.1 Introduction 33
7.2 Architecture and Demonstration of the tool 33

8 Experimentation & Results 42
8.1 Selection of case studies 42
8.2 Manual examination of the case studies 42
8.2.1 Algorithm for Conformance Checking 42
8.2.2 Algorithm for Verifying Testability 43
8.3 Analysis & Results 44
8.3.1 Conformance FRs 44
8.3.2 Non-Conformance FRs 45
8.3.3 Recommendations 48
8.3.4 Testable/Non-Testable NFRs 51

9 Conclusion and Future work 54

References 55

Abstract

Natural Language (NL) is widely adopted as the primary method of expressing software requirements, although determining its superiority is challenging. Empirical evidence suggests that NL is the most commonly used notation in the industry for specifying requirements. One of the main advantages of NL is its accessibility to various stakeholders, requiring minimal training for understanding. Additionally, NL possesses universality, allowing its application across diverse problem domains. However, the unrestricted use of NL requirements can result in ambiguities. To address this issue and restrict the usage of NL requirements, Requirement Templates (RTs) are employed. RTs have a fixed syntactic structure and consist of predefined slots. When requirements are structured using RTs, ensuring they conform to the specified template is crucial.

Manually verifying the conformity of requirements to RTs becomes a tedious task due to the large size of industry requirement documents, and it also introduces the possibility of errors. Furthermore, rewriting requirements to conform to the template structure when they initially do not conform presents a significant challenge. To overcome these issues, we propose a tool-assisted approach that automatically verifies whether Functional Requirements (FRs) conform to RTs. It provides a recommendation for a Template Non-Conformance (TNC) requirement by generating a semantically identical requirement that Conforms to the template structure. Our study focused on two well-known RTs, namely, Easy Approach to Requirements Syntax (EARS) and RUPPs, for checking conformance and making recommendations. We utilized Natural Language Processing (NLP) techniques and applied our approach to industrial and publicly available case studies. Our results demonstrate that the tool-based approach facilitates requirement analysis and aids in recommending requirements based on their conformity with RTs. Furthermore, we have developed an approach to assess Non-Functional Requirements (NFRs) testability by analyzing the associated acceptance criteria. We evaluated the applicability of this approach by applying it to various case studies and determining the testability of the NFRs.

List of Principal Symbols and Acronyms

EARS Easy Approach to Requirements Syntax

FN False Negative

FP False Positive

FRs Functional Requirements

GATE General Architecture for Text Engineering

JAPE Java Annotation Patterns Engine

LHS Left-Hand Side

NFRs Non-Functional Requirements

NL Natural Language

NLP Natural Language Processing

POS Part-Of-speech

RHS Right-Hand Side

RTs Requirement Templates

SRS Software Requirements Specification

TC Template Conformance

TN True Negative

TNC Template Non-Conformance

TP True Positive

List of Tables

5.1 Non-Conformance requirements with recommendations generated by the approach 25

6.1 Acceptance-Criteria concepts description 29

8.1 Accuracy results for TC requirements 46
8.2 Accuracy results for TNC requirements 47
8.3 Accuracy results for Recommendations 49
8.4 Two iterations for partially correct recommendations 51
8.5 Accuracy results for NFR testability checking 52

List of Figures

1.1 Approach Flow Graph . . . 4

2.1 EARS Template [33] . . . 10
2.2 Example requirements showing Conformance/Non-Conformance to EARS . . . 11
2.3 RUPPs Template [43] . . . 12
2.4 Example requirements showing Conformance/Non-Conformance to RUPPs . . . 12

4.1 NLP Pipeline for FR . . . 18
4.2 Example requirements with the annotations generated after running the JAPE file . . . 21
4.3 JAPE Rule for EARS Ubiquitous requirement . . . 22

5.1 JAPE Rule to identify Conditional details . . . 26

6.1 NLP Pipeline for NFRs . . . 30
6.2 Example NFRs with the annotations generated after the processing of NLP Pipeline . . . 31
6.3 JAPE Rule for the 'Limit' Acceptance Criteria Class . . . 31

7.1 GATE components . . . 35
7.2 GATE Framework utilized for FR conformance checking . . . 36
7.3 GATE Framework utilized for NFRs testability checking . . . 37
7.4 Wrapper tool utilized for FR conformance checking . . . 39
7.5 Wrapper tool utilized for NFR testability checking . . . 40
7.6 Architecture of the tool . . . 40

8.1 Confusion Matrix for conformance requirements . . . 45
8.2 Confusion Matrix for Non-Conformance requirements . . . 46
8.3 Complex Noun identification problem . . . 50
8.4 Confusion matrix for measuring accuracy of testability checking . . . 52

Chapter 1
Introduction

1.1 Natural Language Textual Specifications

Ensuring high-quality software is a critical step in today's technology-driven world. An essential aspect of achieving software quality lies in establishing accurate and unambiguous software requirements. Conventionally, NL text serves as the predominant means of expressing software requirements due to its universal accessibility and familiarity [43]. The significance of NL in specifying requirements has been firmly established through extensive research [47][50]. Although it is challenging to definitively establish NL as the optimal choice, empirical evidence gathered over the years has consistently demonstrated that it is the most prevalent notation used for expressing requirements in industrial practice [36][31]. Its accessibility allows stakeholders from diverse backgrounds and expertise to contribute and comprehend the desired functionalities and features.

When individuals use NL text without following any specific rules or regulations, the inherent flexibility of NL can introduce unintended ambiguities into the requirements. These ambiguities have the potential to cause misunderstandings, misinterpretations, and misalignments throughout the software development lifecycle. Ambiguities can arise due to unclear words or phrases, inconsistent sentence structures, vague terminology, or misunderstandings of the context, making it challenging to comprehend and capture the precise specifications and objectives of a software system. Consequently, developers may unintentionally create software with incorrect features, resulting in poor performance, dissatisfied users, and even critical system failures. Bruijn et al. [21] conducted research on the impact of highly ambiguous requirements documents on project success. In order to minimize these risks and uphold the desired quality standards, it is crucial to implement measures that specifically target the ambiguities and uncertainties associated with NL requirement specifications.

1.2 Objective and Problem Description

Introducing a level of formality and precision to requirements can help eliminate or minimize potential ambiguities, thereby enhancing the clarity and accuracy of the software requirements. Several strategies to improve quality involve implementing rigorous techniques, such as formal language specification, domain-specific modeling languages, or structured frameworks for requirements documentation. One such structured framework that plays a pivotal role in ensuring consistency, clarity, and completeness during the capture and communication of software requirements is the use of RTs [42].

RTs are predefined structures employed to specify and document software requirements. These templates consist of various slots or fields that serve as designated spaces for capturing specific information related to the requirements [42]. Using templates, requirements become more amenable to automated analysis, such as semantic consistency checking and model transformation [18][57][43]. There is evidence of support for requirements templates in requirements authoring and management tools [49], which suggests a broader industrial interest in templates and the quality assurance activities associated with their use. Furthermore, using RTs eliminates the required training overhead.

When utilizing RTs, analysts must ensure that the templates are correctly implemented. Manual verification of Template Conformance (TC) for large sets of requirements can be time-consuming and prone to errors [43], especially when multiple stakeholders are involved. Moreover, the task becomes challenging when it needs to be performed repeatedly in response to requirement modifications. Similarly, manually transforming TNC requirements into TC is a laborious process. It requires rewriting the requirement document and repeatedly checking it for conformity to the template. This process is time-consuming and may involve examining multiple document versions before achieving the desired level of conformity.

To address these issues, we propose an approach that involves two primary tasks: the first task entails automatically checking TC, while the second task involves recommending a semantically identical TC version for a TNC requirement.

1.3 Thesis contribution

The major contributions of the thesis are:

1.3.1 Automated Conformance Checking with RTs

The proposed comprehensive approach enables the automatic classification of requirements based on their conformity to the EARS and RUPPs Template in a more efficient and streamlined manner. To assess the Conformance of FRs with specified Templates, We used an NLP Pipeline equipped with distinct NLP modules and a rule-based pattern matching technique known as 'Java Annotation Patterns Engine (JAPE)', which allows for an effective assessment of FRs.

1.3.2 Automated Recommendations for Non-Conforming FRs

Once the requirements are identified as non-conforming, we employ JAPE Rules to analyse the underlying causes of Non-Conformance for each requirement. Subsequently, utilizing JAPE, automatic recommendations are generated to resolve the identified Non-Conformance issues.

1.3.3 Automated Assessment of Testability for NFRs

The provided approach automatically identifies the acceptance criteria within the NFRs and classifies the requirements as either Testable or Non-Testable based on the presence of acceptance criteria. To accomplish this, an NLP Pipeline, in conjunction with JAPE Rules, has been employed to extract the pattern of acceptance criteria within the NFRs.

1.3.4 Tool Support

We have developed tool support for automated testability checking, conformance checking, and providing recommendations for the Non-Conformance requirements with RTs.

1.4 Overall Analysis of the Requirement Document: Flow Graph

Our proposed approach flow is shown in Figure 1.1, involves a systematic procedure for checking the Conformance of FRs to RTs. The approach involves the following steps:

1. **NLP Pipeline to process FR:** The FRs from the document undergo a sequential processing pipeline consisting of different NLP modules. The result

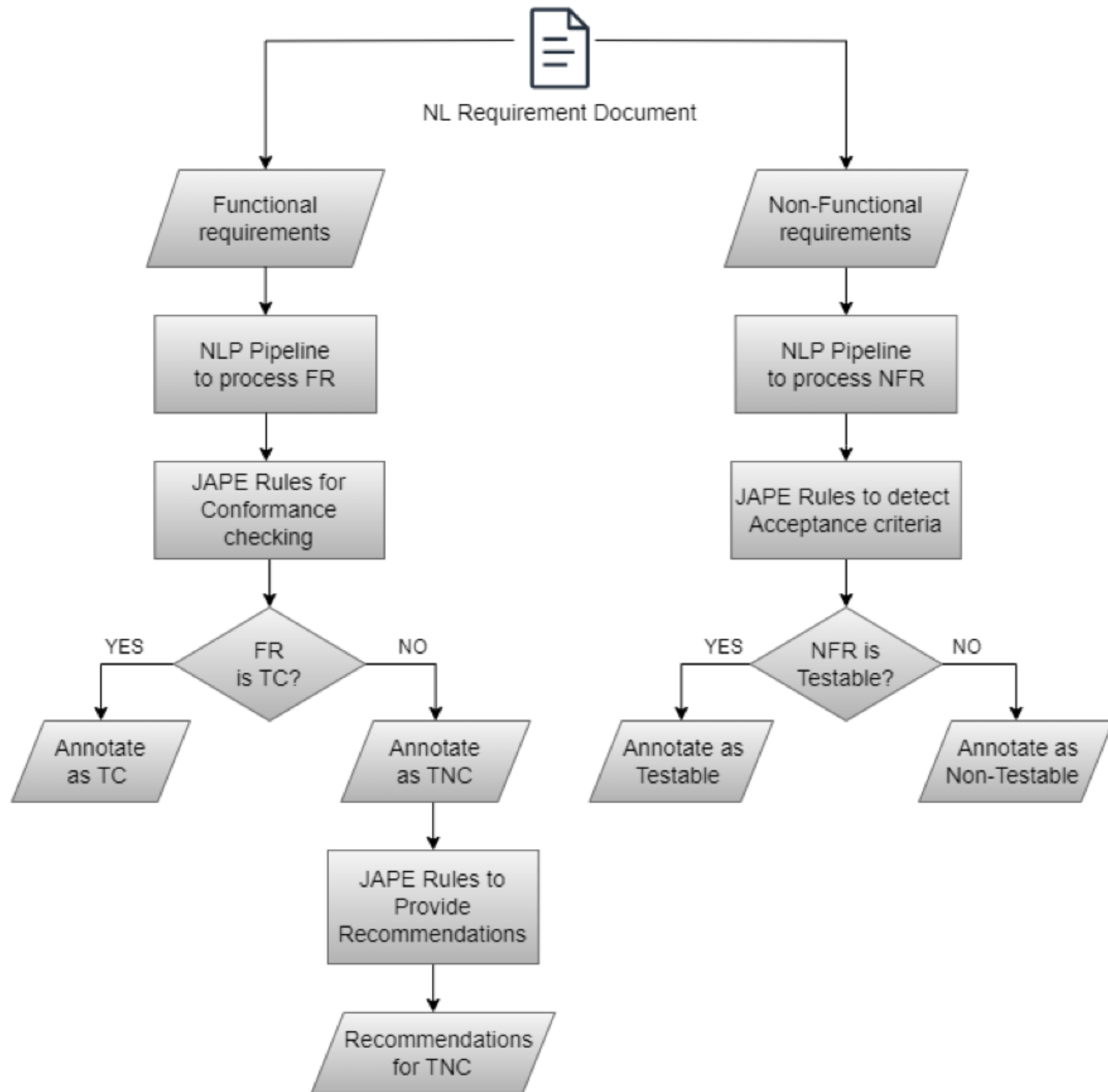

Figure 1.1: Approach Flow Graph

of this pipeline is the extraction of tokens, sentences, Part-Of-Speech (POS) tags, and other relevant linguistic information from the document.

2. **JAPE Rules for Conformance Checking:** JAPE Rules are employed to annotate distinct slots of EARS/RUPPs template structure within the requirement and verify conformance of requirements to the specified template structure. Based on these rules, if a requirement's slots align with the conformance pattern rule of the specified template, it is labelled as 'TC'. Otherwise, it is classified as 'TNC'.

3. **JAPE Rules to Provide Recommendations:** In cases where the requirement does not conform to the prescribed template structures, JAPE Rules are employed to generate a recommendation. This recommendation entails a mod-

ified version of the requirement that conforms with the template structure. The determination of the recommendation is based on the nature of Non-Conformance, which is assessed through the application of JAPE Rule patterns.

By employing this approach, the verification of Testability for NFRs can be achieved through the following steps:

1. **NLP Pipeline to process NFRs:** Initially, the NFRs undergo a sequential processing phase involving various NLP modules. These modules analyze the NFRs to extract linguistic information such as tokens, sentences, numbers, measurements, and other relevant elements.

2. **JAPE Rules for acceptance criteria detection:** The linguistic information derived from the aforementioned NLP Pipeline is subsequently utilized within the JAPE Rules patterns. These patterns enable the identification and detection of acceptance criteria within the NFRs. If the NFR includes acceptance criteria, it is categorized as Testable. Conversely, if the NFR lacks such acceptance criteria, it is classified as Non-Testable.

By executing these steps, the proposed approach facilitates the verification of the Testability aspect for NFRs. The NLP Pipeline aids in extracting linguistic features, while the application of JAPE Rules enables the identification of acceptance criteria embedded within the NFRs.

1.5 Organisation of the Thesis

The thesis is structured into distinct chapters, each focusing on a specific aspect. In Chapter 2, we explore the background of RTs by conducting an analysis of the existing available RTs. Additionally, we provide a detailed discussion of the specific RTs utilized in this study, namely the EARS and RUPPs templates. Chapter 3 delves into the literature review and related work, providing a comprehensive overview of existing research. In Chapter 4, we explain the approach for automating the conformance checking of FRs to RTs. Chapter 5 presents our approach for an automated recommendation generation system specifically designed to address TNC requirements. In Chapter 6, we discuss our approach for automating the testability checking of NFRs using the acceptance criteria. The features and architecture of the tool, along with a demonstration, are discussed in Chapter 7. In Chapter 8, we delve into the experiment conducted to obtain and analyze the

results that form the basis for evaluating the tool and the work presented in this thesis. This chapter offers valuable insights into the practicality and effectiveness of the tool. Lastly, Chapter 9 concludes the thesis, summarizing the key findings and contributions while discussing potential avenues for future research and development.

CHAPTER 2

Background of Requirement Templates

Requirement Templates (RTs) serve as valuable tools for requirement engineers in specifying various types of requirements, encompassing both Functional and Non-Functional aspects. By providing a structured framework for expressing requirements in NL, RTs help mitigate the ambiguities that often arise when using unrestricted NL [43]. In this chapter, we will delve into a discussion of the RTs that are already available, examining their applicability and benefits in addressing various requirements engineering challenges.

2.1 Analysis of RTs

Numerous RTs have been proposed by researchers, each offering distinct relevance and significance to specific domains of requirements. We begin our exploration with a template that focuses on the specification of general application FRs. One notable template in this category is the one developed by Chris Rupp [43]. RUPPs template adopts a unique approach, utilizing verbs to designate the functionalities and actions of the system. This approach enhances the clarity and precision in capturing the FRs of the application. Another important template in the realm of requirement engineering is the I-star template, introduced by author Eric S. K. Yu [59]. The I-star template serves as a modeling technique specifically applied during the early phases of requirement engineering. The I* framework utilizes a graph-based approach to construct visual strategic models.

In addition to specifying FRs using the aforementioned template, it is equally crucial to address the Non-Functional aspects of requirements. Non-Functional specifications are majorly needed in applications that require safety-critical and security-related aspects. In the existing literature, several notable templates have been proposed specifically for specifying requirements in the context of safety-critical applications. These templates have gained popularity and serve as valuable resources for effectively capturing the requirements associated with safety-

critical applications. The EARS template is widely recognized as a valuable tool for specifying requirements that adhere to safety criteria. It employs basic syntactic thumb rules to ensure clarity and consistency in the specification process. For instance, the use of "when" is employed to denote event-driven behaviour, "while" for state-driven behaviour, and "if-then" statements to address potential failures [33]. These conventions provide a structured framework for expressing safety-related requirements effectively. In addition to the EARS template, Antonino proposed another template [39] that introduces additional features for specifying parameterized safety requirements. While the EARS template serves well for specifying high-level stakeholder requirements, it falls short in ensuring the traceability of safety requirements from an architectural perspective. Antonino's template, on the other hand, tackles this issue by introducing a structured approach. Antonino's template emphasizes the significance of Top-Level Safety Requirements as a starting point, followed by templates for safety requirements at the functional level. Finally, it provides templates for specifying safety requirements at the technical level, encompassing both software and hardware artefacts [39]. This comprehensive approach enables a more systematic and granular representation of safety requirements, facilitating better traceability and integration within the overall architectural context.

In recent times, software system security has emerged as a critical concern, underscored by numerous reported incidents exploiting software vulnerabilities and posing threats to systems. These threats encompass activities such as unauthorized access to sensitive information, data manipulation, and the potential for denial-of-service attacks. Given the gravity of these risks, it is imperative to incorporate security considerations right from the initial phases of requirement modeling for system software. To address these security requirements effectively, several templates have been developed and utilized. We explored a selection of these templates, starting with the template proposed by Firesmith [24]. Firesmith's security template adopts a parameterized and reusable approach specifically designed to handle security parameters within the system. This template revolves around identifying valuable assets (such as data and servers), different attacker types and the associated threats to the assets. Another notable template proposed for specifying security requirements is the Riaz'16 template [46]. Riaz's template offers comprehensive coverage of essential security aspects by accounting for requirements related to confidentiality, integrity, availability, authentication, accountability, and privacy. In contrast, Kamalrudin's work on specifying security requirements introduces a unique approach [30]. Kamalrudin et al. de-

veloped a security requirements library named SecLab, which encompasses a comprehensive collection of predefined security properties such as confidentiality, integrity, and more. These properties can be seamlessly incorporated within a textual template structure, enabling efficient and accurate specification of security requirements. In his review paper titled "A Comparative Study of Proposals for Establishing Security Requirements for the Development of Secure Information Systems" [34], Mellado et al. present a comprehensive comparative analysis of eight security RTs. Their study assesses the templates based on multiple criteria, including their degree of agility, user-friendliness, degree of integration, help support, and overall contributions.

Considering the aforementioned RTs and their applicability, we have selected the EARS template and RUPPs template for the conformance checking of FRs. These templates demonstrate a correlated and straightforward syntax, rendering them well-suited for conformance checking to utilize NLP techniques. Furthermore, these templates have extensive industry usage, as indicated by various studies [54], and there are readily available practitioner guidelines for their utilization. This combination of factors positions the EARS and RUPPs templates as highly suitable options for automating the conformance checking of requirements and generating recommendations for TNC requirements.

2.2 EARS template

The EARS template, depicted in Figure 2.1 [33], encompasses a comprehensive framework for composing requirement sentences. Within this template, each sentence contains four distinct slots:

1. **An optional initial condition** serving as a starting point.
2. **The system name** specifies the particular system under consideration.
3. **A modal verb (SHALL)**
4. **The system response** details the intended behaviour or actions of the system.

EARS incorporates five diverse alternative structures within its first slot. These structures are employed to differentiate between various requirement types, thereby enhancing the versatility and applicability of the template.

1. **Ubiquitous requirements:** These requirements are forever active and do not rely on any preconditions.

2. **Event-driven requirements:** These requirements come into play when the trigger event occurs. They are activated by a specific event denoted by the term 'WHEN'.

3. **Unwanted behaviour requirements:** These requirements capture undesired scenarios or conditions. They begin with the conditional keyword 'IF'.

4. **State-driven requirements:** Designed for active requirements within a particular state. These requirements initiate with 'WHILE'.

5. **Optional feature requirements:** Referred to as requirements that must be fulfilled when specific optional features are present. These requirements begin with the word 'WHERE'.

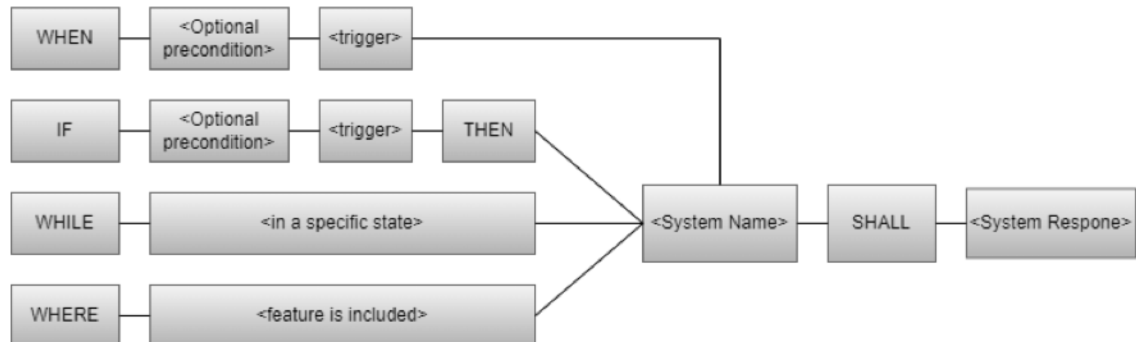

Figure 2.1: EARS Template [33]

In Figure 2.2, two requirements, R1 and R2, are presented. It is apparent that R1 and R2 share the same meaning, but R2 conforms to the 'Unwanted behavior requirement' type of EARS while R1 does not follow the EARS Template structure. Within R2, all the fixed components of the EARS Template are expressed in capital letters. The Non-Conformance of R1 to the template can be attributed to two reasons: 1) The usage of the term 'MUST' instead of the specified keyword 'SHALL' and 2) The condition fragment is not positioned at the beginning of the requirement.

2.3 RUPPs template

As illustrated in Figure 2.3, The RUPPs template [43] incorporates six distinct components within a single requirement sentence. These components are as follows:

R1: The system must display an error message indicating a login failure if the user attempts to log in with an incorrect password.

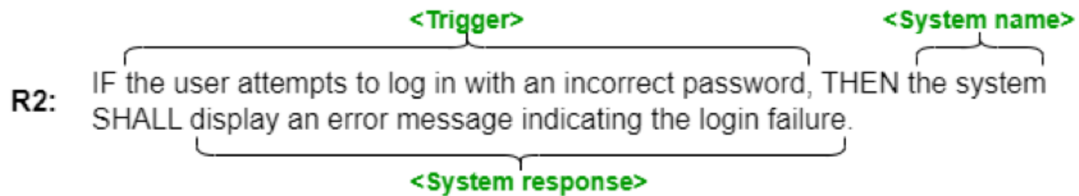

Figure 2.2: Example requirements showing Conformance/Non-Conformance to EARS

1. An optional condition, positioned at the beginning.

2. The name of the system.

3. A modal verb (such as "shall," "should," or "will") indicates the level of importance of the requirement. Here are examples of how the modal verbs can be used in the RUPPs template:

   - **Shall:** The system shall generate an automated monthly report for the finance department. (This requirement is mandatory and must be implemented.)

   - **Should:** The system should provide an option to save user preferences for future sessions. (This requirement is recommended but not mandatory.)

   - **Will:** The system will display real-time notifications to users when new messages arrive. (This requirement expresses a future action that the system will perform.)

4. The required processing functionality, which can take on three different forms depending on how the functionality is intended to be provided.

5. The object for which the functionality is being described.

6. Optional additional details about the object.

The three alternatives for the fourth component capture the following:

- **<process>:** This alternative is used for capturing functions that the system performs without user interactions. These functions are commonly referred to as autonomous requirements.

- **PROVIDE <whom?> WITH THE ABILITY TO <process>:** This alternative is employed to capture functions that the system provides to specific users, which are commonly referred to as user interaction requirements.
- **BE ABLE TO <process>:** This alternative is employed for interface requirements, which capture the functions performed by the system in response to trigger events initiated by other systems.

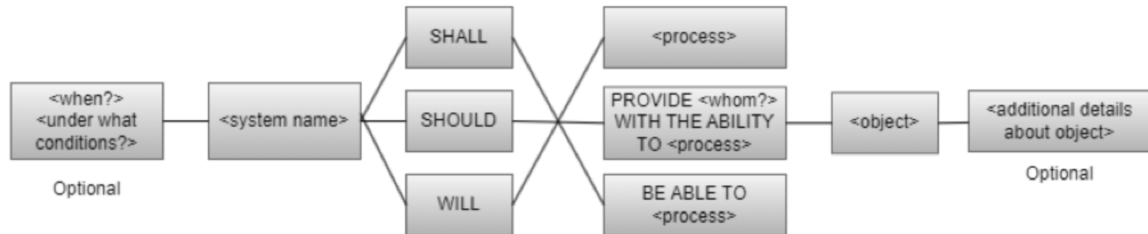

Figure 2.3: RUPPs Template [43]

In Figure 2.4, two requirements, R3 and R4, are displayed. It is evident that R3 conforms to the 'autonomous requirement' type of RUPPs, while R4 does not conform to the structure of the RUPPs Template. In R3, all the fixed components of the RUPPs Template are expressed in capital letters. The Non-Conformance of R4 to the template can be attributed to the fact that the condition fragment, "IF the User has no other option," is not placed at the beginning of the requirement.

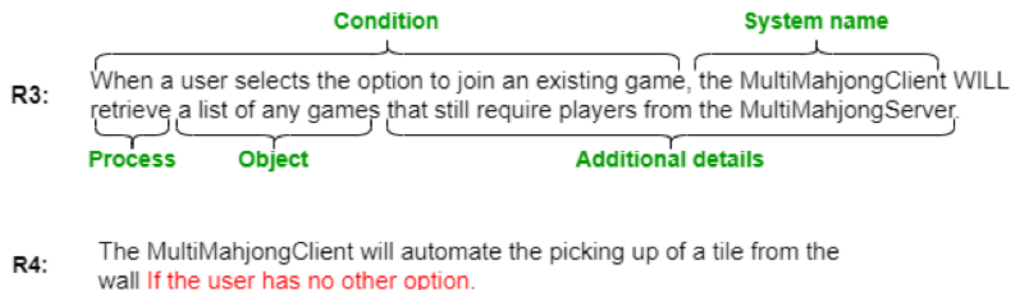

Figure 2.4: Example requirements showing Conformance/Non-Conformance to RUPPs

CHAPTER 3

Literature Review

3.1 NLP in Requirement Engineering

The field of NLP has a lengthy history of application in Requirements Engineering. It has been deduced that the utilization of NLP techniques and tools has proven to be greatly advantageous in expediting the Software Requirements Engineering process [38]. This can be attributed to the widespread use of NL in the outlining and definition of requirements. NLP can be effectively employed to analyze the initial software requirements with the aim of accomplishing objectives such as requirement prioritization and classification into categories of FRs and NFRs. Sharma et al. [51] developed an automated framework to identify NFRs in NL requirements. Their approach involves extracting multiple features, represented as pattern-based rules, and the subsequent identification of NFRs based on specific combinations or relationships among these features.

NLP techniques are also utilized in Quality Assurance (QA) procedures for requirements [20][40][29][25]. In their research, Genova et al. [25] presented multiple indicators aimed at assessing the quality of textual requirements, complemented by a fully automated tool [9] designed to calculate these quality measures. Fabbrini et al. [20] developed a Quality Model that can be applied to assess requirements and eradicate any ambiguities or incompleteness. The authors have developed a tool QuARS consisting of NLP modules like a lexical and syntax analyzer. Ormandjieva et al. [40] presented an automatic approach to assessing the quality of textual requirements using NLP in all the phases of the Software Development Life Cycle (SDLC). In NLARE [29], authors propose a set of guidelines for promoting a disciplined sentence structure for expressing requirements. By adhering to these guidelines, NLP techniques can be effectively leveraged to evaluate the quality of the requirements.

The conversion of software requirements from NL to a more formal specification has the potential to reduce their inherent ambiguity and incompleteness.

Numerous studies [8][37][23][55][28] have focused on utilizing NLP techniques to transform NL requirements into formal specifications. The authors Ibrahim et al. [8] have presented a method that utilizes NLP and Domain Ontology techniques to support the analysis of textual requirements and the extraction of class diagrams. More et al. [37] proposed an approach to extract Unified Modeling Language (UML) diagrams from textual requirements. The methodology employs various NLP technologies such as OpenNLP Parser, Word Net, Concepts Extraction Engine, and Domain Ontology. Hamza et al. [28] have devised a methodology that employs various NLP techniques, such as tokenization, stemming, POS tagging, and chunking, to generate a UML use case diagram from textual requirements. Few studies [23][55] have also investigated the use of automated approaches utilizing NLP techniques for transforming NL textual requirements into object-oriented (OO) specifications.

3.2 Identification of Ambiguities in NL text

For quite some time, requirements engineers have acknowledged the significant challenge posed by ambiguity when it comes to specifying and constructing software systems [48]. Ambiguity can give rise to various issues that impact the system, as it essentially becomes a bug if left undetected and unresolved during the early stages. According to the findings of Berry et al. [14], it has been found that "In reality, it is impossible to eliminate ambiguity entirely, and thus, we must develop the ability to identify it." Therefore, the identification and correction of ambiguous requirements should be regarded as the foremost priority. Extensive research has been conducted over the years to address the detection of ambiguities in software requirements. Zait et al. [60] presented an approach to identify lexical and Semantic ambiguities in requirements and provide all the possible interpretations of the requirements to the Requirement analysts. The authors Chantree et al. [15] introduced a novel and scalable technique that effectively notifies requirement authors about the existence of potentially risky ambiguities. Osama and co-authors presented an efficient automatic syntactic ambiguity detection technique for NL requirements [41]. Their technique utilizes the Stanford CoreNLP library to filter the potential scored interpretations of a given sentence. Furthermore, it offers users feedback by presenting the possible correct interpretations to address the identified ambiguity. In their work, Kiyavitskaya, Nadzeya, et al. [32] introduced a two-step tool-supported methodology that aims to detect ambiguities within NL requirements specifications. The first step applies a range of

ambiguity measures to the Requirements to identify sentences that potentially exhibit ambiguity. In the second step, the tool provides an analysis of the specific aspects that may contribute to ambiguity within each identified sentence. Yang et al. [58] introduced an approach aimed at automatically identifying harmful ambiguities that arise when readers interpret requirements differently. Their approach detects and addresses ambiguities that can lead to misunderstandings or conflicts among different stakeholders.

3.3 Requirement Templates

Several works of literature in the field of Requirement Engineering have extensively examined the utilization of RTs. In a research paper titled "On systematically building a controlled natural language for functional requirements" [56], authored by Alvaro Veizaga et al., a methodology is presented for defining Controlled Natural Languages (CNLs) that can effectively express FRs. As a result of this methodology, a tool called Rimay was developed. Rimay offers a systematic process for developing CNLs, referred to as RTs, enabling practitioners to create their own RTs using the tool. In particular, the paper "Automated Recommendation of Templates for Legal Requirements" [53] introduces a novel approach for automatic template recommendation for legal requirements. The approach hinges upon a qualitative study that establishes NLP rules to facilitate the recommendation process. Numerous works in the past literature have addressed the topic of software requirements conformance checking to RTs, as referenced in [22][25][12]. Among these, the DODT tool [22] and the RQA tool [9] rely on domain ontology to facilitate conformance checking with templates. However, Requirements Template Analyzer (RETA) [4] has been specifically developed to assess requirements conformance with two templates, namely EARS [33] and RUPPs [43]. An advantage of RETA is, it is independent of domain ontology and domain-specific terms during requirements analysis [12].

After a comprehensive review, we have come to the overall conclusion that none of the aforementioned discussions explicitly address the complete scope of the issue we tackle in this work: The automated conformance checking of requirements with RTs and The automated generation of recommendations for TNC requirements. Our approach stands out from RETA in several significant aspects. Firstly, our tool goes beyond RETA by providing recommendations for Non-Conforming requirements, which RETA lacks. Secondly, while RETA employs a text chunking process for automated conformance checking, our tool lever-

ages parsing techniques. The generation of recommendations in our tool relies exclusively on the effective results derived from the parsing-based conformance checking process. This distinction ensures the reliability of our recommendation generation process.

3.4 Non-Functional Requirements

Despite their inherent importance in software development, NFRs have historically received less attention compared to FRs. In many cases, they have been neglected or overlooked entirely [16]. Insufficient consideration or inadequate treatment of NFRs often results in software products that are not deemed acceptable. Therefore, it is equally crucial to analyze NFRs. NFRs that cannot be tested are often disregarded in the development of a system since there are no means to validate them. Consequently, it is essential to assess the testability of NFRs [19][26]. Various research studies have demonstrated the significance of evaluating the testability of NFRs [35][52][45]. In related research [52], the use of Scenario-Based representation for NFR testability analysis has been proposed. The authors introduced a template that employs vertical and horizontal dissection to analyze quality concerns, with the identified delimiters serving as a basis for assessing system testability. Metsa et al. [35] evaluate the usage of aspect orientation for testing NFRs in software systems. The mentioned approach identifies the system features that can be effectively tested using aspects and explores methods for deriving test objectives from NFRs. In the research for quality assurance of NFR [45], authors have used the fit criteria and applied Rule-based learning to check the testability. However, In this thesis, we have developed a tool-supported approach which performs testability checking of NFRs by evaluating the presence of acceptance criteria and by leveraging technologies like the 'General Architecture for Text Engineering (GATE) NLP Workbench' [17][27] and 'JAPE Rules'.

CHAPTER 4

Automated Conformance Checking of Functional Requirements to RTs

4.1 NLP Pipeline to process FR

Our methodology entails the utilization of an NLP Pipeline, illustrated in Figure 4.1, which encompasses seven modules that systematically process the FR document. The result of this sequential processing is an annotated requirement document. The following description provides a concise overview of the usage and functionality of each module.

1. **Tokenizer:** It is a crucial NLP module that effectively breaks down FRs into smaller units known as tokens, each representing a single word in the requirement.

2. **Sentence Splitter:** The goal of sentence splitting is to break down a larger piece of text into smaller, more manageable units called sentences. It is used in our approach for identifying the boundaries between requirement sentences.

3. **POS Tagger:** This module assigns a grammatical tag to each token in a requirement based on its syntactic function and context. The POS tagger provides us with valuable insights into the linguistic aspects of the requirements by identifying the POS tags such as nouns, verbs, adjectives, adverbs, and others within the requirements.

4. **Named Entity (NE) transducer:** With the objective of conducting comprehensive requirement analysis, our approach incorporates an NE transducer module. This module identifies and categorizes named entities within the FRs, including specific People, locations, organizations, products, and other entities that possess unique names or designations.

5. **Gazetteer:** It is a module that stores lists of words or phrases related to a particular concept or entity. By utilizing this module, our approach can effectively identify and classify text based on the presence of these predefined words or phrases. We have developed a dedicated gazetteer list tailored to identify Vague, Implicit, Conditional, and Quantifier words, as well as Adverbs within verb phrases, in the FRs.

6. **Morphological Analyser:** This module provides an in-depth analysis of the structure and grammatical properties of tokens within the FRs, such as voice, tense, root form etc. By incorporating this module into our approach, we want to accurately determine the root form of words. The root form represents the fundamental and uninflected version of a word, commonly used as the dictionary form or lemma.

7. **Parser:** The parser module plays a vital role in our approach by facilitating the construction of parse trees. These parse trees depict the hierarchical structure of the requirements and the relationships between their components. By utilizing the parser, we can identify complex noun phrases and verb phrases within the requirements.

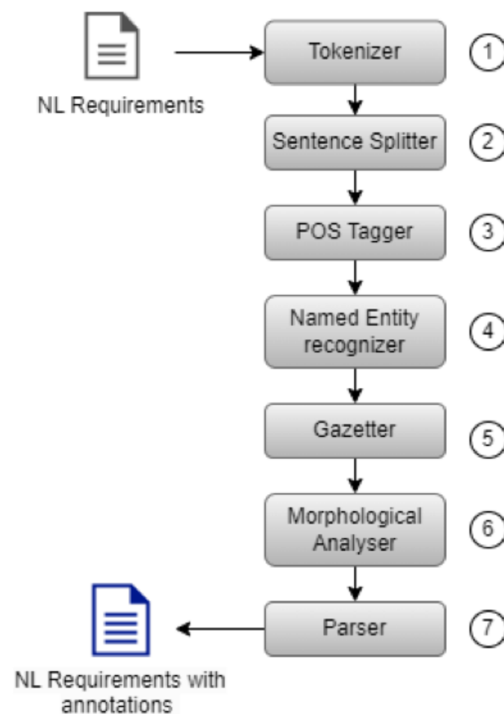

Figure 4.1: NLP Pipeline for FR

Deviating from the approach outlined in RETA [12], which employs Text Chunking for conformance checking, we embrace the Parsing technique. Unlike the Text Chunking process, Parsing possesses the capability to accurately identify Complex Nouns. Therefore, the adoption of Parsing is paramount in our work, as it enables us to achieve precise conformance checking results and facilitates the generation of accurate recommendations for TNC requirements.

4.2 JAPE Rules for Conformance Checking

After the successful execution of the entire NLP Pipeline on the FR document, an annotated document is generated. This annotated document contains various linguistic features, including annotations for Noun Phrases, Verb Phrases, Conditional words, and more. The JAPE file utilizes the acquired information to classify FRs into two distinct categories: Conformance requirements and Non-Conformance requirements.

The JAPE file comprises a collection of rules that serve to identify different slots of EARS and RUPPs templates within the FRs. Once all the slots have been appropriately identified and annotated, an additional rule is applied to evaluate the correct occurrence of these annotations within the requirement, as per the template structure. This rule facilitates the categorization of requirements into either Conformance or Non-Conformance. The following elucidation outlines the successive rules of the JAPE file.

1. **Annotate Condition** annotates the condition that appears at the beginning of a requirement and precedes the system name. Condition phrase is characterized by a conditional keyword, which is followed by a sequence of tokens. The tokens may consist of one or more words that describe the conditions under which the functionality of the system needs to be provided.

2. **Annotate System Name** refers to the rule for identifying and labelling the Noun Phrase that represents the system being referred to in a given requirement. The System Name is a Noun Phrase that appears before the Verb Phrase in the requirement and serves as the subject of the action being described.

3. **Annotate Valid Modal Verb Phrase** is a rule used for recognizing and marking the Verb Phrase that contains a valid modal auxiliary verb as per the templates, such as "will", "shall", or "should".

4. **Annotate Object** rule is used to annotate the Noun Phrase that appears after the Verb Phrase in a requirement. The Object typically represents the entity or thing that is affected by the action described in the Verb Phrase.

5. **Annotate Object Details** This rule is responsible for identifying and annotating the sequence of tokens that appears after the Object in a requirement. It is important to note that this part is optional and may not be present in every requirement. This rule excludes the requirements that contain conditional keywords after the Object Name.

6. **Annotate TC requirements** is the rule for recognizing and annotating all FRs that conform to either RUPPs or EARS template structure. The objective of this rule is to ensure that the FRs conform to the specific template structure and include all the necessary components. This rule is performed after the previous annotations have been completed, as it relies on the information gathered from those annotations.

7. **Annotate TNC requirements** is a rule that involves identifying and annotating all requirements that do not conform to any of the template structures, RUPPs or EARS; and have not been previously annotated as TC requirements.

In Figure 4.2, example requirements are presented along with the annotations generated after executing the NLP Pipeline and applying the JAPE File. Requirements R1 and R2 are annotated as TC, while requirement R3 is annotated as TNC.

In requirement R1, the phrase "If the statistical report is selected" is annotated as a Condition since it begins with the conditional keyword 'If' and is followed by a series of tokens. The phrase "the THEMS system" is annotated as System Name because it represents a Noun Phrase occurring before the Verb Phrase. Next, "SHALL present" is annotated as a Valid Modal Verb Phrase because it constitutes a Verb Phrase commencing with the valid modal 'SHALL'. Lastly, "a list of available months" is a Noun Phrase which is following the Verb Phrase, and it is therefore annotated as an Object.

The annotation sequence for requirement R1 is as follows: Condition -> System Name -> Valid Modal Verb Phrase -> Object. It conforms to the Autonomous requirement type specified in RUPPs, leading to its annotation as a TC requirement.

Requirement R3 is annotated as a TNC requirement because it deviates from the expected structure outlined in both the RUPPs and EARS templates. Accord-

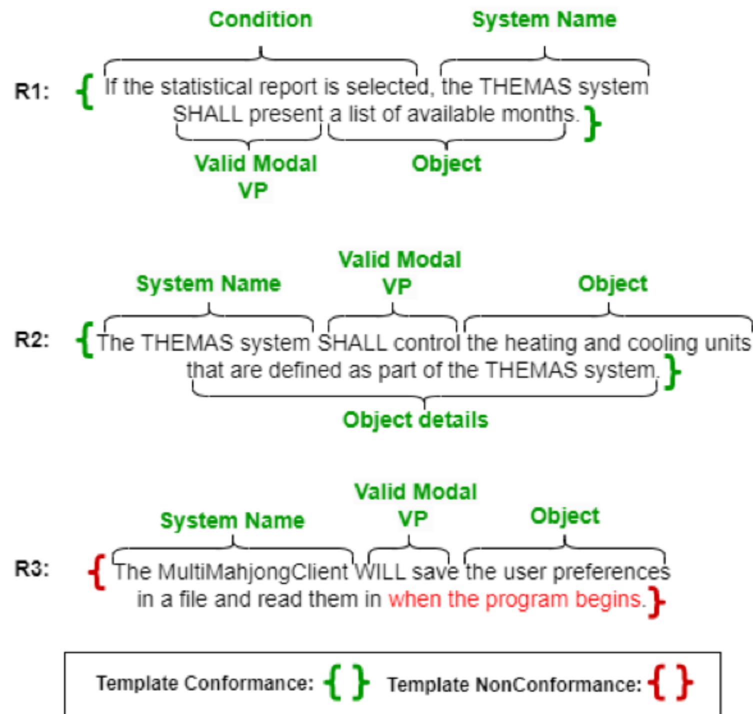

Figure 4.2: Example requirements with the annotations generated after running the JAPE file

ing to these templates, the condition is expected to appear at the beginning of the requirement. However, in this particular requirement, the condition appears after the Object Name, which violates the specified template structure.

The example JAPE Rule in Figure 4.3 demonstrates the identification and annotation of the Ubiquitous requirement type of EARS. The rule consists of two sides: the Left-Hand Side (LHS) and the Right-Hand Side (RHS), separated by the symbol '–>'. The LHS is employed to match a specific pattern in the text, while the RHS processes and annotates that text.

In this particular rule, the LHS is used to identify the pattern for the EARS Ubiquitous requirement type. It begins with the {*SystemName,!Warn_Pronoun, !Warn_implicity*} annotation, followed by {*ModalEARS*}, and concludes with the occurrence of a certain number of tokens denoted by {*Token,!Conditional*}+. To determine the end of the requirement sentence, the {*Split*} is matched after this pattern. This JAPE Rule takes input from previous JAPE Rules, which produce annotations such as *SystemName, ModalEARS, Conditional, Warn_Pronoun, and Warn_-implicity*. The {*SystemName*} annotation is used to annotate the subject of the requirement, while {*ModalEARS*} annotates the SHALL keyword that follows

the subject. To prevent the matching of texts with ambiguous terms as their SystemName, {*!Warn_Pronoun, !Warn_implicity*} are utilized. These restrictions ensure the rule does not match texts containing such ambiguous terms. The {Token,!Conditional}+ suggests that the System Response should not contain any 'Conditional' tokens. Here, 'Conditional' is a dictionary created in the Gazetteer, which includes conditional keywords such as 'if', 'while', 'when', and so on. Once this entire requirement text is identified, it is annotated using the RHS of the RULE.

```
Phase: EARS
Input: Token Split Warn_Pronoun ModalEARS SystemName Conditional Warn_implicity
Options: control = first

Rule Ubiquitous
(
    (({SystemName,!Warn_Pronoun,!Warn_implicity}) : System
    ({ModalEARS}) : Modal
    (({Token,!Conditional})+
    ({Split})) : SystemRes) : TemplateConformance

):ann
-->
{
    AnnotationSet con = bindings.get("TemplateConformance");
    Annotation conformanceAnn = conformance.iterator().next();
    FeatureMap features = Factory.newFeatureMap();
    features.put("explanation", "Following Ubiquitous requirement type of EARS");
    outputAS.add(con.firstNode(), con.lastNode(),"TemplateConformance",features);
}
```

Figure 4.3: JAPE Rule for EARS Ubiquitous requirement

CHAPTER 5

Automated Recommendation System for Non-Conformance Requirements

5.1 JAPE Rules for Providing Recommendations

After annotating the requirements as TNC, it is crucial to determine the reason behind their Non-Conformance to the template structure and provide modified requirements that maintain the same meaning as the original requirement and Conform to either of the templates, EARS or RUPPs. This task is accomplished through the JAPE Rules. To achieve this, the LHS of each JAPE Rule contains specific pattern that indicates the reason for Non-Conformance, while the RHS of the each JAPE Rule provides recommendation based on the identified error.

When it comes to identifying TNC in requirements, there are five potential cases to consider. To address each of these cases, we have developed distinct JAPE rules that facilitate the precise identification and categorization of instances of Non-Conformance.

1. **Missing System Name:** If the System Name specified in the FR is an implicit or pronoun term, there is a potential for referential ambiguity within the requirement, and it deviates from the expected EARS and RUPPs template structure. This JAPE rule identifies such requirements and provides appropriate recommendations by replacing the pronoun/implicit term with the phrase {*The System*}. Alternatively, if explicit glossary terms for these requirements are provided, referential ambiguity can be resolved by substituting the ambiguous terms with the specified System Name mentioned in the Requirement glossary.

2. **Missing/Incorrect Modal:** This rule addresses cases in which a modal verb other than those specified in the EARS and RUPPs templates (i.e., "will," "shall," or "should") is used. In such instances, the rule suggests a revised

version of the requirement in which the incorrect modal is replaced with the appropriate modal phrase *Will/Shall/Should*. However, due to the rule-based nature of the system, offering a specific modal as a suggestion poses a challenge.

3. **Passive Voice:** When a requirement is expressed in the passive voice, it suggests that the System name of the requirement is either implicit or described after the functionality. However, both the EARS and RUPPs templates specify that the subject or System name should precede the functionality or Verb Phrase. This JAPE Rule identifies such cases, facilitates the automatic conversion from the passive voice to active voice, and generates a TC version of the requirement.

4. **Conditional details:** When the condition is positioned after the object in the requirement sentence, such requirements fall into this category. To ensure the correct TC recommendation, the entire conditional fragment is relocated to the beginning of the requirement, specifically before the System Name.

5. **Redundant Data:** This category comprises requirements that contain data other than a conditional fragment or system name at the beginning of the requirement. To address this issue, a JAPE rule is employed to provide a recommendation that involves removing all tokens preceding the system name/condition and relocating them after the object details fragment. This adjustment ensures that these tokens are placed at the end of the requirement, resulting in improved clarity and conciseness.

The example JAPE Rule depicted in Figure 5.1 showcases the process of identifying and annotating the Conditional details type of Non-Conformance. The LHS of this rule commences with *({Condition})?*, indicating that there may or may not be a condition present at the beginning of the requirement. Following that, {*SystemName, !Warn_Pronoun, !Warn_implicity*} examines whether the subsequent component of the requirement corresponds to a System Name. Next, it verifies if any of the valid Modals from RUPPs and EARS are used in the next component of the requirement using the Syntax *({ModalForRuppInterface} | {ModalEARS} | {ModalForRuppAutonomous} | {ModalForRuppUI})*. The last part of the rule is responsible for determining if the 'Conditional' keyword appears anywhere in the System response or details section of the requirement. If this pattern is matched, the rule proceeds to annotate the identified portion and provides a recommendation for that particular requirement through the RHS of the rule.

Table 5.1: Non-Conformance requirements with recommendations generated by the approach

Requirement	**Reason for Non-Conformance**	**Recommendation**
The MultiMahjongClient will inform the user if another player is fishing.	Conditional details	If another player is fishing, The MultiMahjongClient will inform the user.
For each event that is generated, the THEMAS system shall identify each event and generate the appropriate event data.	Redundant data	The THEMAS system shall identify each event and generate the appropriate event data, For each event that is generated.
The MultiMahjongClient must only allow players to make moves according to the Chinese rules of Mahjong.	Incorrect Modal	The MultiMahjongClient will/shall/should only allow players to make moves according to the Chinese rules of Mahjong.
The processing for any Computer Opponents will be done by the MultiMahjongClient program.	The sentence is in passive voice	The MultiMahjongClient program will do The processing for any Computer Opponents.
When the THEMAS system is initialized, it shall first turn off all the heating and cooling units.	Referencial ambiguity, System name is missing	When the THEMAS system is initialized, {The system} shall first turn off all the heating and cooling units.

```
Phase: Nonconformance
Input: Token Split Warn_Pronoun ModalEARS SystemName Conditional Warn_implicity
Condition ModalForRuppInterface ModalForRuppAutonomous ModalForRuppUI
Options: control = first

Rule: ConditionalDetails
(
    (({Condition})? : Con
    ({SystemName,!Warn_Pronoun,!Warn_implicity}) : SN
    ({ModalForRuppInterface} | {ModalEARS} |
        {ModalForRuppAutonomous} | {ModalForRuppUI}) : Modal
    (({Token,!Split})* {Conditional} ({Token})+ {Split}) : Details ) : ConditionalDetails
):ann
-->
{
    String recommendation = "";
    \\code to generate the recommendation

    AnnotationSet AnnSet = bindings.get("ConditionalDetails");
    Annotation Ann = AnnSet.iterator().next();
    FeatureMap newFeatures = Factory.newFeatureMap();
    newFeatures.put("Reason", "Conditional details");
    newFeatures.put("Recommendation", recommendation);
    outputAS.add(AnnSet.firstNode(), AnnSet.lastNode(), "TemplateNonConformance",newFeatures);
}
```

Figure 5.1: JAPE Rule to identify Conditional details

Table 5.1 presents several examples of TNC requirements, accompanied by the corresponding reasons for Non-Conformance and the recommended corrections for achieving conformance.

CHAPTER 6
Automated Analysis of Non-Functional Requirements

6.1 Testability of Non-Functional Requirements

NFRs have historically been overlooked in software development. Despite being among the most expensive and difficult requirements, NFRs are often neglected or forgotten, as functionality takes priority over quality concerns. It is universally acknowledged that NFRs are crucial for software acceptability, yet the industry has treated them casually for a long time. Failure to satisfy these requirements results in low acceptability, which goes against the product due to increasing competition and critical system failures. The importance of addressing NFRs has become evident, and the industry must focus on these requirements to deliver successful software products.

According to the IEEE format, a Software Requirements Specification (SRS) should adhere to several criteria, including being correct, unambiguous, complete, consistent, ranked for importance and/or testability, verifiable, modifiable, and traceable, as outlined in [13][44][19]. NFRs are particularly crucial in this regard since they typically arise from the quality concerns of stakeholders, which can be inherently ambiguous and vague. Therefore, NFRs must be specified objectively and be Testable to ensure they are addressed effectively. Measurable and Testable requirements are essential for testing NFRs, as highlighted by the performance testing model [19][26], where such requirements form the first step towards testing. NFRs that cannot be tested are usually disregarded during software development since there is no way to validate them. Consequently, NFRs need to be evaluated for their testability. However, with NL requirements, determining testability has traditionally been done manually, leading to significant time and cost implications.

A requirement is considered Testable when it has been analyzed and broken

down to a level where it is specific, clear, unambiguous, and not capable of being divided into lower-level requirements. When NFRs are not Testable or quantifiable, they are likely to be ambiguous, incomplete, or incorrect.

Below are a few examples of NFRs categorized as Non-Testable and Testable:

1. Non-Testable: "The system shall be fast." The specified requirement is not quantifiable, and the interpretation of "fast" can differ from one individual to another. A revised Testable requirement could be: The system shall respond to user requests within two seconds.

2. Non-Testable: "The system shall be scalable." This requirement lacks precision and is not quantifiable, rendering it Non-Testable. One possible way to rephrase the requirement to make it Testable could be: The system shall be able to handle a 20% increase in user load without a decrease in performance.

3. Non-Testable: "The system shall be reliable." The term "reliable", as used in the requirement, lacks clarity. A revised Testable version of the requirement may read as follows: The system shall have a mean time of at least 10,000 hours between failures.

The objective of this work is to enhance the clarity and testability of NFRs, particularly those that are Non-Testable, by identifying them and presenting them to stakeholders for further refinement.

6.2 Acceptance criteria

To identify Non-Testable NFRs, our automated approach utilizes acceptance criteria as a key metric. By incorporating acceptance criteria into our analysis, we can differentiate between Testable and Non-Testable requirements. Requirements that include acceptance criteria are recognized as Testable, whereas those lacking acceptance criteria are categorized as Non-Testable. This approach enables efficient identification and classification of NFRs based on their potential for testing.

The concept of acceptance criteria consists of two distinct subclasses: Unit and Quantity. The Quantity subclass relates to the numerical value preceding a specific unit of measurement. Meanwhile, the Unit subclass encompasses six subclasses that represent different unit categories such as Time, Percentage, Limit, Speed, Frequency, and Distance, which are detailed in Table 6.1.

Table 6.1: Acceptance-Criteria concepts description

Concept	Description	Example
Time	The time units	12 seconds, two days, 8 PM
Percentage	The percentage of the measured object	80% of the users
Limit	The measured entities that do not belong to other fit-criteria	5 movies
Speed	The data transfer connection speed	15 mbps
Frequency	The occurrence number of events per time interval	2 times per day
Distance	The distance between two objects	1.4 miles

In order to identify the presence of acceptance criteria, we utilized a similar technology that employs an NLP Pipeline and JAPE rules, similar to our approach for checking the conformance of FRs to RTs. However, it is important to note that the NLP Pipeline and JAPE rules implemented for evaluating testability differ from those used for conformance checking.

6.3 NLP Pipeline to process NFRs

The NLP Pipeline processing of NFRs aids in the categorization of requirements into two distinct groups based on the presence of acceptance criteria: 1) Testable and 2) Non-Testable.

1. **Tokenizer:** A tokenizer is a module that breaks down NFRs into separate words or tokens.

2. **Sentence Splitter:** This module plays a crucial role in dividing extensive collections of NFRs into distinct sentences, enabling further analysis of the NFRs on a sentence-by-sentence basis.

3. **POS Tagger:** This module assigns a precise label, such as a noun, verb, adjective, and more, to each token within an NFR sentence.

4. **Gazetter:** This module manages dictionaries containing words that associate to specific concepts or entities. We have curated different lists to identify Vague, Implicit, Conditional, and Quantifier words within NFRs. Furthermore, we have constructed a dictionary to handle keywords associated with limits, including terms like "within," "minimum," "maximum," "more

than," and others. This dictionary contributes to the identification of acceptance criteria.

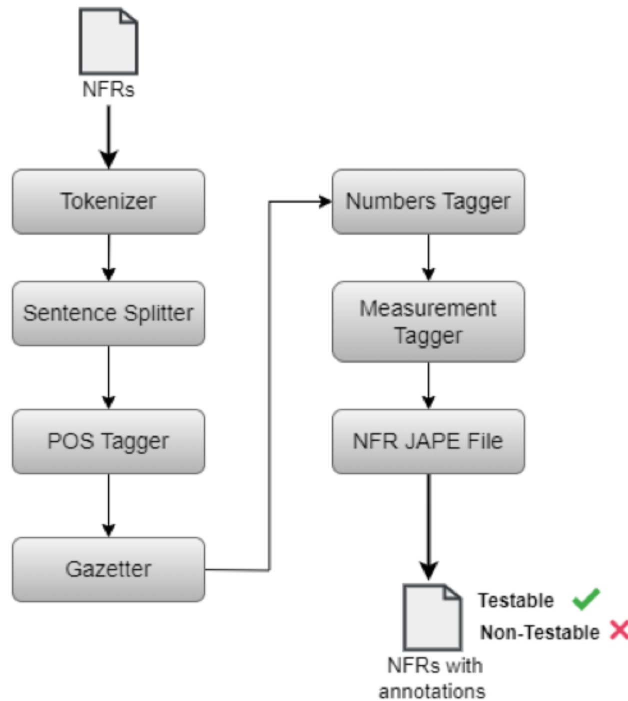

Figure 6.1: NLP Pipeline for NFRs

5. **Numbers Tagger:** This module identifies and extracts numerical data, which includes integers, decimals, fractions, percentages, and currencies. It proficiently detects these numeric values, thereby contributing to the identification of the Quantity subclass of acceptance criteria.

6. **Measurement Tagger:** This specialized module recognizes and extracts measurements, such as lengths, weights, temperatures, time, and more, from textual data. By precisely identifying these measurements, the module plays a crucial role in identifying the Unit subclass of acceptance criteria.

7. **JAPE File:** Once the Unit and Quantity subclasses annotations are obtained from the preceding NLP modules, the JAPE file employs these annotations within the RULES to classify requirements as either Testable or Non-Testable.

As depicted in Figure 6.2, after the completion of pipeline processing, R1 and R2 are classified as Testable, while R3 is categorized as Non-Testable. R1 features acceptance criteria specifying "1 minute," wherein the term "minute" belongs to the **Time** subclass. Conversely, R2 falls within the **Limit** subclass and presents acceptance criteria indicating "10 simultaneous games."

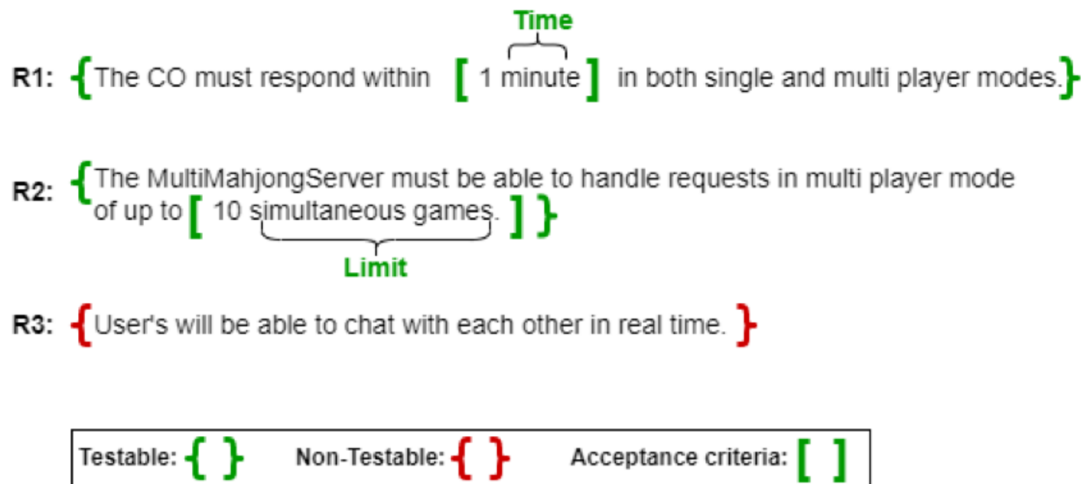

Figure 6.2: Example NFRs with the annotations generated after the processing of NLP Pipeline

6.4 JAPE Rules for detecting the Acceptance criteria in NFRs

Each class in the Acceptance criteria is accompanied by a JAPE Rule specifically designed to identify and annotate the corresponding pattern within the NFRs.

```
Phase: AcceptanceCriteria
Input: Limit_Keywords Measurement Token Number
Options: control = applet

Rule: Limit
(
    ({Limit_Keywords})?
    (({Measurement}) |
    ({Token.kind == number} {Token.category == JJ} {Token.category == NNS}) |
    ({Token.kind == number} {Token.category == NNS}) |
    ({Number} {Token.category == NNS}))
)
:ann
-->
:ann. Quantification =
{ type = "Quantification" , string = :ann@string , majorType="units"}
```

Figure 6.3: JAPE Rule for the 'Limit' Acceptance Criteria Class

For instance, the example JAPE Rule for the 'Limit' class is illustrated in Figure 6.3. This rule utilizes the annotations generated by previous modules of the NLP Pipeline to match the desired pattern. In this particular rule, the {*Limit_Keywords*} refers to a dictionary containing various limit-related words such as 'within', 'min-

imum', 'maximum', and so on. The Measurement Tagger module generates the {*Measurement*} annotation, while the Numbers Tagger generates {*Number*} annotation. These annotations are employed by the rule to effectively match the pattern and annotate the relevant information.

CHAPTER 7

Tool Support

7.1 Introduction

We have developed a comprehensive tool that effectively implements our approach. This tool has been specifically designed to analyze both the FRs and NFRs, effectively reducing any ambiguities that are present. It offers a range of valuable features, including:

Functional Requirements Conformance Checking: The provided tool is a valuable resource for verifying the conformity of FRs with two RTs: EARS and RUPPs.
Recommendation generation for TNC Requirements: It assists in identifying Non-Conformance reasons and recommends semantically identical requirements that conform to the specified templates. By doing so, the tool ensures that all FRs align with the predefined standards.
Testability Checking of Non-Functional Requirements : In addition to FR conformance checking, the tool also performs comprehensive testability verification for NFRs based on well-defined acceptance criteria.

7.2 Architecture and Demonstration of the tool

Our tool has been developed utilizing two distinct technologies. Firstly, we have integrated the **GATE Framework** [17], which incorporates all the NLP modules and JAPE files. This framework is an open-source framework developed in 1995 at Sheffield University. Its purpose is to help developers, students, educators, users, and scientists in resolving text-processing challenges. GATE is built on Java and offers a user-friendly interface known as GATE Developer [27], along with a comprehensive set of libraries accessible through its Application Programming Interface (API).

In addition to GATE, our tool employs **JavaServer Pages (JSP)**, a Java-based framework, to encapsulate the functionalities of the GATE Framework. This integration enables us to leverage the capabilities of GATE within a web application context, thereby enhancing the overall architecture of our tool. Through this wrapping process, we ensure a seamless integration that allows the functionalities of the GATE Framework to interact smoothly with the users, thereby enhancing their experience with our tool. To encapsulate or embed GATE processing, we utilize 'GATE Embedded' in our Java code, which is the API provided by the GATE Framework.

GATE Main characteristics: The GATE Framework operates on a component-based architecture, which emphasizes the separation of data and application functionalities.

1. **Language Resources** include a diverse range of items for processing, such as documents, corpora, annotation schemas, and ontologies.

2. **Processing Resources** consist of tools and plugins that execute specific text analysis functions, such as parsers and tokenizers.

3. **Applications** involve pipelines of Processing Resources designed to process data from Language Resources systematically.

4. **Data Store** acts as a repository for processed Language Resources, ensuring their accessibility and preservation.

NLP Pipeline configuration used for FR: In Figure 7.1, we can observe the various components of GATE. Specifically, under the Applications tab, we have saved a pipeline called 'Pipelinefr', which represents the NLP Pipeline utilized for processing the FRs. Moving on to the Language Resources tab, we can see a corpus and a document labelled as 'FR', signifying the FR document that requires processing. Lastly, within the Processing Resources tab, there are eight NLP resources displayed. These NLP modules are employed in our NLP Pipeline to process the data provided in the FR document file. The Processing Resources flow through the FR document one by one and generate results that involve the classification of FRs according to their conformance to RTs. Furthermore, the system offers recommendations for non-conforming requirements.

Excluding the ANNIE Gazetteer [11] and GATE Morphological analyzer [2], we have thoroughly explored all available alternatives within the GATE framework for the remaining five modules. Our focus has been exclusively on exploring alternatives within the GATE framework without considering any options outside

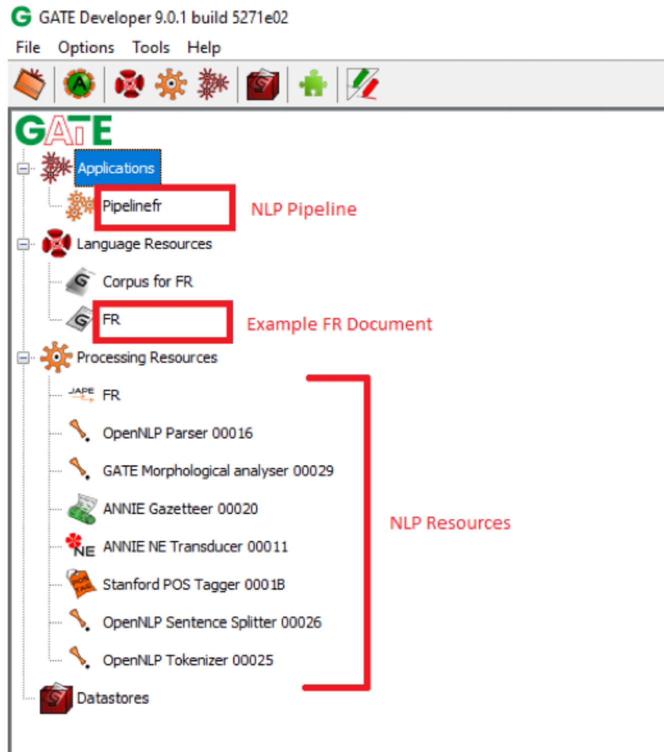

Figure 7.1: GATE components

of it. In the following list, we present the diverse alternatives that were considered for each of the five modules:

1. **Tokenizer:** ANNIE English Tokeniser [11], OpenNLP Tokenizer [10], Stanford PTB Tokenizer [7]

2. **Sentence Splitter:** ANNIE Sentence Splitter [11], OpenNLP Sentence Splitter [10]

3. **Named Entity Recognizer:** ANNIE NE Transducer [11], OpenNLP NER [10], Stanford NER [5]

4. **POS Tagger:** ANNIE POS Tagger [11], Stanford POS Tagger [8], OpenNLP POS Tagger [10]

5. **Parser:** OpenNLP Parser [10], StanfordParser [6]

As a result, we have a total of 108 different NLP Pipeline configurations (3 x 2 x 3 x 3 x 2) to compare. After carefully observing the results, we have considered the following alternatives for the given modules: OpenNLP Tokenizer, OpenNLP Sentence Splitter, ANNIE NE Transducer, Stanford POS Tagger, and OpenNLP Parser.

NLP Pipeline configuration used for NFRs: In the NFRs NLP Pipeline, we have opted for the same alternatives as in the FR NLP Pipeline. Specifically, we have utilized the OpenNLP Tokenizer, OpenNLP Sentence Splitter, ANNIE Gazetteer, and Stanford POS Tagger. Additionally, we have included two new NLP modules, namely the Numbers Tagger [3] and the Measurement Tagger. A clear visual representation of all the modules included in the NFRs Pipeline can be found in Figure 7.3, which is located within the Processing Resources tab.

We offer two ways for FR conformance checking and NFR testability checking:

1. **Directly accessing the GATE Framework:** Users can open the GATE Framework and execute the NLP Pipeline on the document of interest. This method allows for direct interaction with the GATE NLP Workbench.

   Figure 7.2 shows the results of the conformance checking approach after running the NLP Pipeline on the document by directly opening the GATE Framework. Here the 'TemplateConformance' checkbox highlights all sentences in green that conform to either the template structure of EARS or RUPPs, and the 'TemplateNonConformance' checkbox highlights all sentences in red that fail to conform to the template structures, EARS, and RUPPs. By hovering the mouse over a specific 'TemplateNonConformance' requirement, users gain access to the recommendation and reason for the Non-Conformance, presented in the box below it.

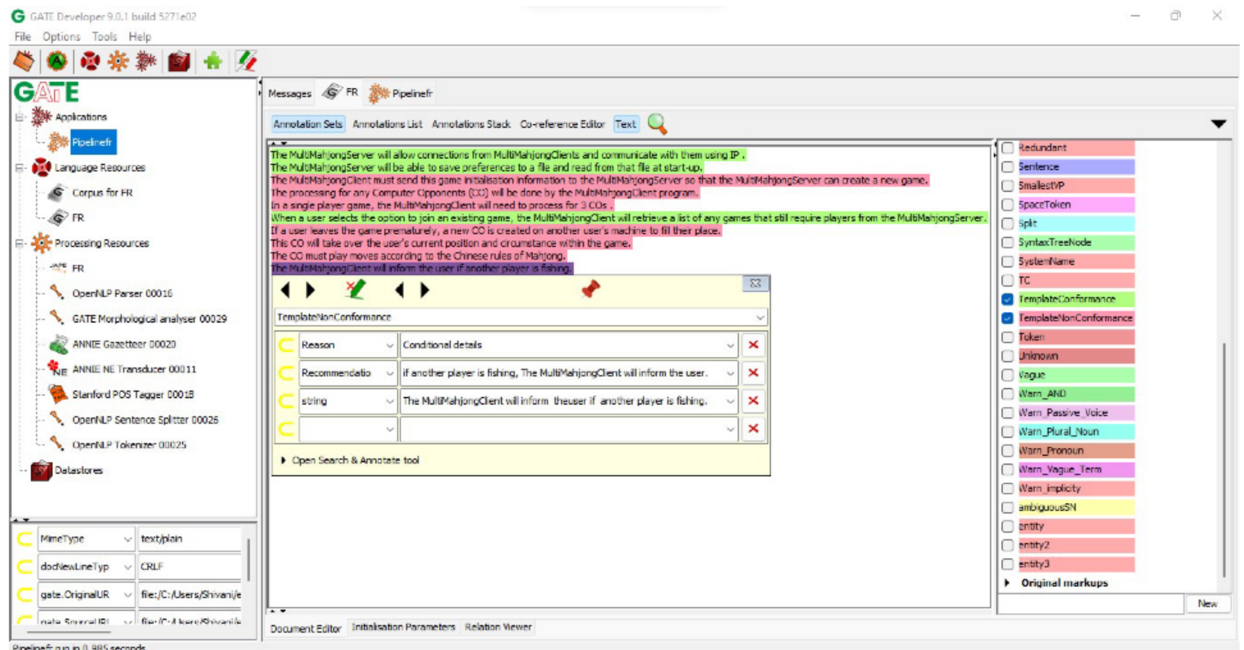

Figure 7.2: GATE Framework utilized for FR conformance checking

Users have the option to access the GATE Framework to execute the NLP

Pipeline on the desired NFR document. Figure 7.3 visually demonstrates the outcomes of the testability checking approach by running the NFRs NLP Pipeline on the NFR document, using the GATE Framework. In this scenario, the 'Testable' checkbox emphasizes all sentences containing Quantification/ acceptance criteria by highlighting them in green. Conversely, the 'Non-Testable' checkbox highlights sentences that lack any Quantification, presenting them in red.

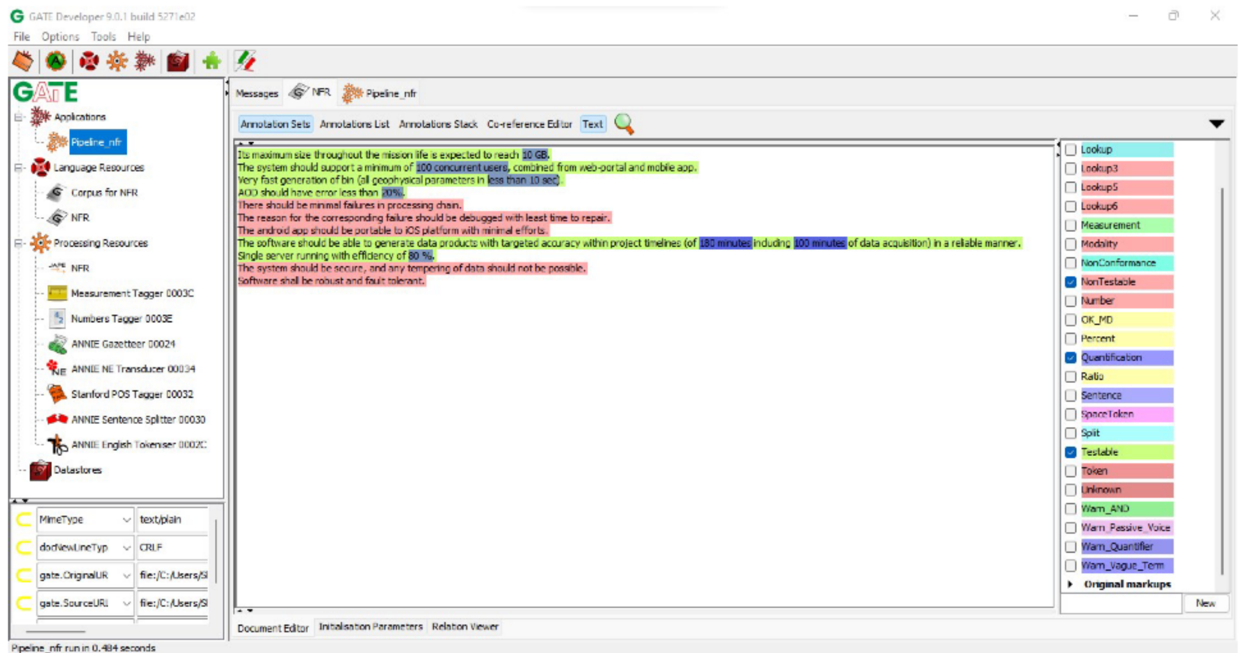

Figure 7.3: GATE Framework utilized for NFRs testability checking

2. **Utilizing our wrapper tool:** Our Wrapper tool combines various web technologies, including HTML and JSP, to facilitate the execution of the GATE NLP Workbench in the background. Users can leverage this tool to perform FR conformance checking and NFR testability checking.

   Figure 7.4 illustrates the outcomes of the conformance classification after executing our wrapper tool on the document. The tool presents distinct tables to display TC and TNC FRs. For each TNC requirement, the tool showcases the reason for Non-Conformance and the recommended TC requirement. Users can conveniently download both tables in Excel format by using the labelled buttons "Download Conformance Excel" and "Download Non-Conformance Excel". Additionally, the annotated FR document can be easily downloaded through the "Download PDF" button. With its seamless user interface, the tool greatly facilitates thorough requirement analysis. Importantly, when the document is submitted for analysis, the JSP file em-

ploys GATE Embedded code, effectively utilizing the GATE Framework in the background to generate conformance classification results.

The results of the testability classification, obtained by running our wrapper tool on the NFR document, are depicted in Figure 7.5. This tool efficiently presents the outcomes in a comprehensive table, clearly distinguishing between Testable and Non-Testable NFRs. In addition to the classification, the tool also showcases the acceptance criteria for each Testable requirement, enabling users to gain valuable insights. In the background, the tool leverages the powerful GATE Framework to generate these results, ensuring an effective and robust analysis. To enhance convenience, the tool offers labelled buttons for downloading the table in Excel and PDF formats.

Architecture: The architecture of the tool, as illustrated in Figure 7.6, contains five sequential steps.

1. The requirement analyst utilizes the user interface (UI) of the Wrapper tool to upload the FR/NFR PDF document. Additionally, they make a selection between checking the "Testability of NFRs" or "Conformance of FRs". The information provided through the UI is then passed to the JSP file, which acts as the backend of the tool and facilitates its functionality.

2. The code within the JSP file converts the FR/NFR document from PDF to a text file and opens the GATE Framework. This is achieved by utilizing the relevant libraries from GATE EMBEDDED.

3. The JSP code proceeds to open either an FR.XGAPP/NFR.XGAPP file within the GATE Framework. These XGAPP files contain the NLP pipeline specific to the FR/NFR. Upon opening the file, the NLP pipeline is loaded under the 'Applications' tab, which in turn loads all the NLP modules under the 'Processing Resources' tab, including the JAPE file. Simultaneously, the converted text file from the previous step is loaded under the 'Language Resources' tab. Subsequently, the NLP pipeline is executed on the FR/NFR text file, resulting in the generation of annotations.

4. The annotations generated through the GATE Framework are sent back to the JSP file using the GATE EMBEDDED code. Afterwards, the JSP file closes the XGAPP file and proceeds to delete all the resources associated with the GATE Framework. Finally, the JSP file ensures the proper termination of the GATE by effectively closing the process.

Template Conformance

Requirement
The MultiMahjongServer will allow connections from MultiMahjongClients and communicate with them using IP .
The MultiMahjongServer will be able to save preferences to a file and read from that file at start-up.
When a user selects the option to join an existing game, the MultiMahjongClient will retrieve a list of any games that still require players from the MultiMahjongServer.

Template NonConformance

Requirement	Reason	Recommendation
The MultiMahjongClient must send this game initialisation information to the MultiMahjongServer so that the MultiMahjongServer can create a new game.	Modality missing	The MultiMahjongClient will/shall/should send this game initialisation information to the MultiMahjongServer so that the MultiMahjongServer can create a new game.
The processing for any Computer Opponents (CO) will be done by the MultiMahjongClient program.	The sentence is in passive voice	the MultiMahjongClient program will do The processing for any Computer Opponents (CO) .
In a single player game, the MultiMahjongClient will need to process for 3COs.	Requirement contains Redundant data before System Name	the MultiMahjongClient will need to process for 3COs. In a single player game,.
If a user leaves the game prematurely, a new CO is created on another user's machine to fill their place.	The sentence is in passive voice	If a user leaves the game prematurely, {The system} will/shall/should create a new CO on another user's machine to fill their place.
This CO will take over the user 's current position and circumstance within the game.	Referencial ambiguity, System's name is missing; Specify name of tool,module,subsystem or system	{The system} / {Specify the name of CO} will take over the user 's current position and circumstance within the game.
The CO must play moves according to the Chinese rules of Mahjong.	Modality missing	The CO will/shall/should play moves according to the Chinese rules of Mahjong.
The MultiMahjongClient will inform theuser if another player is fishing.	Conditional details	if another player is fishing, The MultiMahjongClient will inform theuser .

Download Conformance Excel | Download Non-conformance Excel | Download PDF

Figure 7.4: Wrapper tool utilized for FR conformance checking

Non-Functional requirements

Requirement	Acceptance-criteria	Testable
Its maximum size throughout the mission life is expected to reach 10 GB.	10 GB	Yes
The system should support a minimum of 100 concurrent users , combined from web-portal and mobile app.	100 concurrent users	Yes
Very fast generation of bin (all geophysical parameters in less than 10 sec).	less than 10 sec	Yes
AOD should have error less than 20%.	20%	Yes
The software should be able to generate data products with targeted accuracy within project timelines (of 180 minutes including 100 minutes of data acquisition) in a reliable manner.	180 minutes	Yes
Single server running with efficiency of 80 %.	80 %	Yes
There should be minimal failures in processing chain.	NA	No
The reason for the corresponding failure should be debugged with least time to repair.	NA	No
The android app should be portable to iOS platform with minimal efforts.	NA	No
The system should be secure, and any tempering of data should not be possible.	NA	No
Software shall be robust and fault tolerant.	NA	No

Download Excel Download PDF

Figure 7.5: Wrapper tool utilized for NFR testability checking

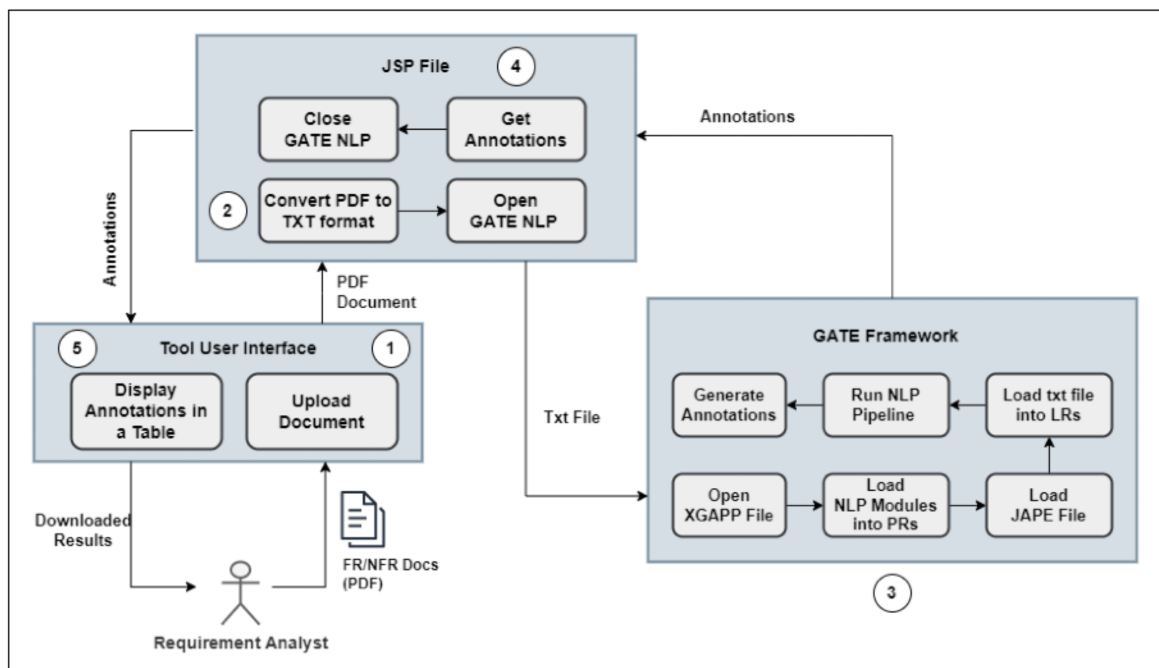

Figure 7.6: Architecture of the tool

5. All the annotations are displayed in a tabular format within the UI of the tool. This allows the requirement analyst to conveniently access and review the results, facilitating analysis and evaluation of the generated annotations.

CHAPTER 8

Experimentation & Results

8.1 Selection of case studies

To evaluate the effectiveness of our approach, we conducted evaluations on a set of 14 diverse SRS documents. Out of these, eight documents were sourced from publicly available repositories [1], while our industry partner, SAC-ISRO, contributed an additional six documents. Three fundamental criteria that guided the selection of these eight documents are:

1. Ensuring coverage across diverse industrial domains.
2. Inclusion of requirements from both EARS and RUPPs templates.
3. Maintaining practical scalability in terms of the number of requirement statements.

8.2 Manual examination of the case studies

This section discusses our process of examining the case studies to develop the ground truth for analysis. First, we manually checked the FRs for Conformance, Non-Conformance, recommendations, and NFRs testability. The thesis author did the step and reviewed it with the supervisor for finalisation. Next, we used the developed dataset to analyse the proposed approaches.

8.2.1 Algorithm for Conformance Checking

To perform the Manual TC Checking, we utilized an algorithm outlined in Algorithm 1.

The algorithm used for manually verifying automatic recommendations remains consistent with the conformance checking process. When evaluating the

recommended requirement using this algorithm, if it is determined to be TC, it is deemed a correct recommendation. However, an additional factor is taken into account: ensuring that the tool has not modified the meaning of the requirement during the recommendation generation process. Conversely, if the recommended requirement is classified as TNC by this algorithm OR the meaning of the requirement is altered in the recommended sentence, it is considered an incorrect recommendation.

Algorithm 1 Manual inspection protocol for conformance checking [12]

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

8.2.2 Algorithm for Verifying Testability

The manual verification of NFR testability involves two steps (as shown in Algorithm 2):

1) If an NFR sentence includes a number followed by any unit from the specified six classes (Time, Limit, Percentage, Speed, Frequency, Distance), then that NFR is considered to have acceptance criteria.

2) NFRs that contain acceptance criteria are classified as Testable, while all other NFRs are categorized as Non-Testable NFRs.

Algorithm 2 Manual inspection protocol for verifying Testability

```
1: Let R be the NFR being inspected for testability.
2: Verify that R is a grammatically-correct sentence. Do not consider punctuation in
   determining correctness.
3: if R includes a numeric value followed by a valid unit then
4:     The acceptance criteria are present.
5: end if
6: if R contains Acceptance Criteria then
7:     R is Testable;
8: else
9:     R is Non-Testable;
10: end if
```

8.3 Analysis & Results

This section presents our analysis results and findings.

8.3.1 Conformance FRs

Figure 8.1 depicts the Confusion matrix utilized to assess the performance of the approach concerning a given set of Conformance requirements. The matrix comprises a 2x2 table representing the two prediction classes, TC and TNC. This table is structured into two dimensions, with one dimension representing the predicted values generated by the tool and the other dimension representing the actual values derived from manual inspection. In cases where the FRs are actually in Conformance with the template but are inaccurately predicted as Non-Conformance by the tool, these instances are categorized as False Negative (FN) cases. Conversely, if the tool predicts FRs as Conformance, but in reality, they are Non-Conformance, these instances are identified as False Positive (FP). True Positive (TP) cases arise when the tool correctly identifies a requirement as Conformance. Similarly, when the tool correctly identifies a requirement as Non-Conformance, it falls into the True Negative (TN) category.

We have computed and derived three distinct parameters by utilizing the values of TP, FP, TN and FN. These parameters provide valuable insights into performance evaluation and analysis.

1. **Precision**, calculated as TP/(TP+FP), determines the proportion of TP predictions made by the tool out of all positive classes.

2. **Recall**, calculated as TP/(TP+FN), measures the proportion of TP predictions out of the total number of actual positive classes.

3. **Accuracy**, calculated as (TP+TN)/(TP+TN+FP+FN), evaluates the overall correctness of the tool's predictions by considering TP and TN cases in relation to all classes. It provides an assessment of the tool's performance in predicting both positive and negative instances.

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

Figure 8.1: Confusion Matrix for conformance requirements

Table 8.1 presents the accuracy results for conformance requirements. Out of a total of 1139 FRs, the tool predicts 355 requirements as TC. However, during the manual inspection, 376 requirements are identified as TC. The table highlights that the disparity in results is due to 21 FN predictions made by the tool. With the exception of documents involving FN results, the Recall value is 1 for all other documents. Notably, no FP predictions are observed, resulting in a precision value of 1 for each document. For all the TC requirements, the tool yields an average precision value of 1, indicating that all the positive predictions made by the tool are accurate. The average recall value stands at 0.94, implying that the tool successfully identifies 94% of the TC requirements out of the total actual TC instances. Additionally, the average accuracy achieved by the tool for TC requirements reaches 0.98, reflecting a high level of correctness in its predictions.

8.3.2 Non-Conformance FRs

The performance evaluation of the approach in relation to a set of Non-Conformance requirement is represented by Figure 8.2. This figure illustrates the Confusion matrix, which consists of a 2x2 table that captures the two prediction classes, TC and TNC. Instances classified as FN occur when the tool incorrectly predicts FRs as Conformance, despite them being Non-Conformance with the template. On the other hand, instances labelled as FP arise when the tool predicts FRs as Non-Conformance, but they are actually Conformance. TP cases emerge when the tool correctly identifies a requirement as Non-Conformance. Conversely, when

Table 8.1: Accuracy results for TC requirements

No.	Document Name	No. of Reqs.	TC Reqs.	FP	TP	FN	TN	Rec.	Prec.	Acc.
1	THEMAS	94	36	0	36	3	55	0.923	1	0.968
2	Home 1.3	39	23	0	23	2	14	0.92	1	0.949
3	Evla corr	16	10	0	10	0	6	1	1	1
4	Blit draft	23	9	0	9	0	14	1	1	1
5	RLCS	105	62	0	62	1	42	0.984	1	0.990
6	Hats	249	116	0	116	10	123	0.920	1	0.96
7	ESA	14	5	0	5	0	9	1	1	1
8	MultiMahjong	96	38	0	38	3	55	0.927	1	0.969
9	DPGS	31	14	0	14	0	17	1	1	1
10	EOS06_SCAT_SRS_V1.1	24	4	0	4	1	19	0.8	1	0.958
11	DAT_v1.1_20may19	75	5	0	5	0	70	1	1	1
12	O3OCM3_V2.1_DRAFT	59	20	0	20	0	39	1	1	1
13	solar-calc-india	28	4	0	4	0	24	1	1	1
14	SGL_IGiS_V11	286	9	0	9	1	276	0.9	1	0.996

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

Figure 8.2: Confusion Matrix for Non-Conformance requirements

the tool correctly identifies a requirement as Conformance, it falls under the TN category.

The assessment of Non-Conformance requirements involves calculating three key parameters: precision, recall, and accuracy. These parameters are determined using the same formula for evaluating Conformance requirements.

Table 8.2 provides valuable insights into the accuracy assessment of TNC requirements, showcasing the tool's precision, recall, and overall accuracy in identifying TNC requirements with minimal discrepancies. Out of a total of 1139 FRs, the tool identifies 784 requirements as TNC. However, upon manual inspection, it is determined that 763 of these requirements are indeed TNC, indicating a slight disparity between the tool's predictions and the manual assessment.

Table 8.2: Accuracy results for TNC requirements

No.	Document Name	No. of Reqs.	TNC Reqs.	FN	TN	FP	TP	Prec.	Rec.	Acc.
1	THEMAS	94	58	0	36	3	55	0.948	1	0.968
2	Home 1.3	39	16	0	23	2	14	0.875	1	0.949
3	Evla corr	16	6	0	10	0	6	1	1	1
4	Blit draft	23	14	0	9	0	14	1	1	1
5	RLCS	105	43	0	62	1	42	0.976	1	0.990
6	Hats	249	133	0	116	10	123	0.925	1	0.96
7	ESA	14	9	0	5	0	9	1	1	1
8	MultiMahjong	96	58	0	38	3	55	0.948	1	0.969
9	DPGS	31	17	0	14	0	17	1	1	1
10	EOS06_SCAT_SRS_V1.1	24	20	0	4	1	19	0.95	1	0.958
11	DAT_ v1.1_20may19	75	70	0	5	0	70	1	1	1
12	O3OCM3_V2.1_DRAFT	59	39	0	20	0	39	1	1	1
13	solar-calc-india	28	24	0	4	0	24	1	1	1
14	SGL_IGiS_V11	286	277	0	9	1	276	0.996	1	0.996

The average precision value achieved is 0.97, indicating that the tool correctly identifies the majority of Non-Conformance requirements. Similarly, the average recall value stands at 1, implying that the tool successfully captures all TNC requirements without missing any. Furthermore, the average accuracy achieved by the tool is **0.98**, reflecting its overall correctness in predicting the conformity of requirements. This high accuracy score demonstrates the tool's ability to distinguish between Conformance and Non-Conformance requirements effectively. There are 21 cases classified as FP, which correspond to 21 FRs that actually conform to the template structure and are TC. However, the tool erroneously identifies them as TNC, resulting in incorrect labelling of these requirements as Non-Conformance. This discrepancy in the predictions introduces a minor inconsistency in the assessment results. There are two reasons for FP results:

1. **POS Tagger Problem:** Out of the 21 cases where FP results were observed, seven instances were attributed to inaccurate token tagging performed by the POS tagger. In these cases, the FP classification arose due to errors in the process of assigning appropriate POS tags to the tokens.

2. **Parser Problem:** This contributes to the remaining 14 cases of FP. In these instances, the FP occurrence can be attributed to the parser's limitations in accurately identifying complex noun phrases that represent the system name.

The parser struggles to correctly parse and comprehend the intricate structure and context of these noun phrases, resulting in misclassifications and FP predictions.

For example, in the requirement, "Display windows opened by the system shall have buttons for closing the windows." The phrase **Display windows opened by the system** can be described as a complex noun phrase or the name of the system. However, due to limitations in the parser's capabilities, it fails to recognize this and instead identifies **the system** as the system name. As a result, the JAPE Rules categorize this requirement as TNC because it considers the preceding phrase "Display windows opened by" as redundant information appearing before the system name. Consequently, it provides recommendations based on this categorization.

8.3.3 Recommendations

The accuracy results for recommendations are shown in Table 8.3. We have performed evaluations of recommendations provided by the approach on a total of 302 TNC requirements. Among these requirements, the tool successfully provided correct recommendations for 239 of them, while generating 18 partially correct recommendations and 45 incorrect recommendations. This translates to an average accuracy of 83.99%, showcasing the tool's effectiveness in delivering the recommendations.

Reason for missing, incorrect and partially correct recommendations:

1. **Missing recommendations:** In order for the rule-based system to generate recommendations, it is essential that the requirements are written in the specified format defined by the RUPPs and EARS Template. Major deviations from this format make it impossible for the system to generate accurate recommendations. To ensure the integrity of our recommendation evaluations, we have excluded these requirements from our analysis. Additionally, we have removed all 21 FP cases to obtain more precise and reliable accuracy results.

2. **Incorrect recommendations:** Based on our observations and analysis of the generated results, we have identified three potential reasons for incorrect recommendations.

Table 8.3: Accuracy results for Recommendations

No.	Document Name	Recs.	Correct Recs.	Partially Correct Recs.	Accuracy
1	THEMAS	47	42	2	0.94
2	Home 1.3	14	11	0	0.78
3	Evla corr	6	2	3	0.83
4	Blit draft	14	13	1	1
5	RLCS	39	30	1	0.79
6	Hats	112	90	4	0.84
7	ESA	9	7	0	0.78
8	MultiMahjong	44	32	7	0.89
9	DPGS	17	12	0	0.70

(a) **Complex Noun identification problem:** Within the context of Non-Conformance requirements, an incorrect recommendation occurs when the rule pattern specified for recommendation is applied exclusively to a portion of the requirement, rather than the entire requirement sentence because of the issue in the results generated by the parser. As depicted in Figure 8.3, in requirement R1, the intended object is "the number of levels of children added to the displayed graph." However, due to limitations in the parser, the tool fails to identify this complex noun phrase accurately and mistakenly annotates "the displayed graph" as the object. Consequently, only a portion of the sentence is matched by the JAPE Rule for recommendation generation, leading to an erroneous recommendation being generated.

Matched Pattern for Recommendation: displayed graph shall be determined by the application configuration.
Incorrect Recommendation: The application configuration shall determine the displayed graph.

(b) **More than one Verb Phrase in a single requirement:** When a requirement contains multiple Verb Phrases, there is a possibility for the JAPE Rules to erroneously match an incorrect pattern due to this confusion. An example from this category is illustrated below:

Requirement: All files in the application directory, including all subdirectories, **shall be copied** to the new directory, and the new directory

shall become the currently selected application.
Incorrect Recommendation: The new directory shall become the currently selected application, All files in the application directory, including all subdirectories, shall be copied to the new directory, and.
The reason for the incorrect recommendation is that the rule incorrectly identifies the entire highlighted phrase in red as redundant data, which is the portion preceding the Noun Phrase, that is, "The new directory". Due to its limitation in understanding the correct context of the requirement, the rule-based system mistakenly identifies this Noun Phrase as the System name, which appears before the Verb Phrase "shall become". But, the actual Verb Phrase for the requirement is "shall be copied".
The correct recommendation for the requirement is: {The System} shall copy all files in the application directory, including all subdirectories to the new directory, and the new directory shall become the currently selected application.

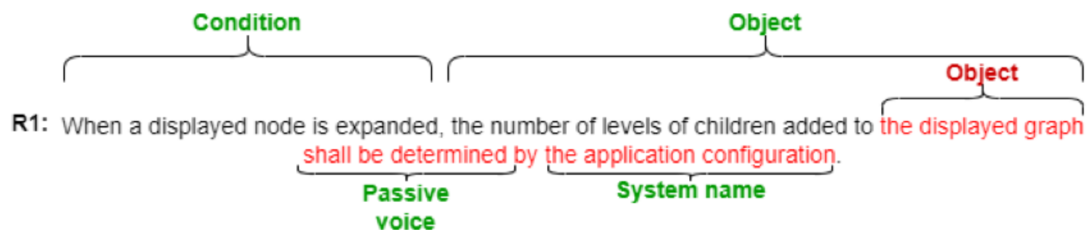

Figure 8.3: Complex Noun identification problem

3. **Partially correct recommendations:** A situation of partially correct recommendation arises when a requirement fails to conform to the template due to any two underlying reasons. In such cases, during the initial run of the tool, one of the reasons is successfully identified, and a recommendation is provided based on that particular reason. However, despite this recommendation, the second reason remains unchanged and unaddressed. One such instance from this category is exemplified below:

   Requirement: The user's screen is then updated to display these changes. This Requirement is not Conformance to the template, and the reasons for Non-Conformance are 'Incorrect Modal' and 'Passive voice'. The recommended requirement after the first iteration of the tool run is:
   Partially correct Recommendation: The user's screen will/shall/should be

Table 8.4: Two iterations for partially correct recommendations

Requirement	First iteration	Second iteration
The MultiMahjongClient must inform the user if another player is fishing.	The MultiMahjongClient shall inform the user if another player is fishing.	If another player is fishing, The MultiMahjongClient shall inform the user.
If there is no line after the current line, the cursor is not moved.	If there is no line after the current line, the cursor will be not moved.	If there is no line after the current line, {The system} will not move the cursor.
In order to provide an operational history and statistical reports, this process shall generate an event .	This process shall generate an event In order to provide an operational history and statistical reports.	{The system} / {Specify the name of process} shall generate an event In order to provide an operational history and statistical reports.
The system shall generate a standard confirmation message after saving data and warning messages after the cancel or close button is clicked.	After the cancel or close button is clicked, The system shall generate a standard confirmation message after saving data and warning messages.	After the cancel or close button is clicked, after saving data and warning messages, The system shall generate a standard confirmation message.

then updated to display these changes.

In this case, the recommendation is partially correct as it addresses the incorrect modality error but fails to rectify the passive voice issue. The second iteration of the tool run on this requirement would give the correct recommendation:

Correct Recommendation: {The system} will then update the user's screen to display these changes.

Out of the 18 partially correct recommendations, we conducted a rerun of all the requirements in the tool and obtained the correct recommendations thereafter. This indicates that two iterations of the tool run yield improved results. Table 8.4 showcases a few examples of these requirements.

8.3.4 Testable/Non-Testable NFRs

Figure 8.4 presents the confusion matrix utilized to assess the accuracy of testability results. This matrix consists of a 2x2 table that represents two potential prediction classes: Testable and Non-Testable. One axis of the table corresponds to the predicted values generated by the tool, while the other axis represents values

derived from the manual inspection of NFRs.

FN instances arise when certain NFRs, which actually contain the acceptance criteria and are Testable, are incorrectly labelled as Non-Testable by the tool. On the other hand, FP instances occur when NFRs that are truly Non-Testable, are erroneously identified as Testable by the tool. TP instances emerge when the tool correctly predicts NFRs as Testable when they are indeed Testable in reality. Finally, TN instances occur when the tool accurately identifies Non-Testable NFRs as Non-Testable.

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

Figure 8.4: Confusion matrix for measuring accuracy of testability checking

Table 8.5 provides detailed information about the precision, recall, and overall accuracy achieved by the tool in the classification of NFRs.

Table 8.5: Accuracy results for NFR testability checking

Document Name	NFR	FP	TP	FN	TN	Prec.	Rec.	Acc.
EOS06_SCAT_SRS_V1.1	46	0	4	0	42	1	1	1
DAT_v1.1_20may19	17	0	1	0	16	1	1	1
SGL_IGiS_V11	18	0	0	0	18	1	1	1
O3OCM3_V2.1_DRAFT	74	0	13	3	58	1	0.81	0.96
solar-calc-india	4	0	1	0	3	1	1	1
DPGS	87	0	6	4	77	1	0.6	0.95
Total	**246**	**0**	**25**	**7**	**214**	**1**	**0.90**	**0.98**

During the analysis of 246 NFRs, the tool identifies 221 requirements as Non-Testable. However, upon manual examination, it is discovered that out of these 221 requirements, only 214 are indeed Non-Testable. The results demonstrate an absence of FP instances, indicating that the tool consistently identifies Non-Testable NFRs accurately, never mislabeling them as Testable. Every NFR clas-

sified as Testable by the tool is indeed Testable. Conversely, there are seven instances of FN, signifying that the tool occasionally fails to recognise requirements as Testable when, in reality, they are Testable.

The tool shows good performance with an average precision score of 1, indicating the absence of any FP instances. Moreover, it achieves an average recall value of 0.90, underscoring its ability to identify a significant portion of Testable requirements accurately. The overall average accuracy of 98% further suggests the tool's reliability in accurately categorising NFRs.

CHAPTER 9

Conclusion and Future work

In this thesis, we have introduced a tool-assisted approach for checking the conformance of FRs to RTs and providing recommendations for Non-Conforming FRs. Our evaluation included 1,139 FRs from fourteen SRS documents to assess the accuracy of the conformance checking results. We also examined 302 requirements flagged as TNC to evaluate the accuracy of the recommendation results. The results demonstrate an average accuracy of 98% for conformance classification, confirming the tool's reliability in effectively classifying FRs. Furthermore, the generated recommendations showcased an average accuracy of 83.99%, highlighting the tool's effectiveness in guiding Non-Conforming requirements.

In addition, the tool includes a feature for verifying the testability of NFRs by assessing the presence of acceptance criteria. To evaluate the testability results, we examined 246 NFRs across six documents. Remarkably, the tool achieved an average accuracy of 98% in accurately classifying NFRs based on their testability.

In terms of future work, our plans involve enhancing the tool's accuracy by incorporating advanced Machine Learning techniques into its framework. By leveraging the power of Machine Learning, we aim to refine the tool's capabilities and ensure even more precise results in the recommendation generation process. Additionally, we intend to expand the work's scope by introducing additional RTs. This expansion will involve integrating specific JAPE rules and modifying the NLP Pipeline used for analysis. These updates will allow the tool to handle a wider range of requirements, domains, and contexts, enhancing its effectiveness.

References

[1] Dataset. [online]. available:. https://zenodo.org/record/1414117#.Y8JznuxBw_W/.

[2] Gate morphological analyzer. https://gate.ac.uk/sale/tao/splitch23.html#x28-50100023.10.

[3] Numbers tagger. https://gate.ac.uk/sale/tao/splitch23.html#sec:misc-creole:numbers:numbers.

[4] Reta: Requirements template analyzer. http://sites.google.com/site/retanlp/.

[5] Stanford named entity recognizer (ner). [online]. available:. https://nlp.stanford.edu/software/CRF-NER.shtml.

[6] Stanford parser. [online]. available:. https://nlp.stanford.edu/software/lex-parser.shtml.

[7] Stanford ptb tokenizer. [online]. available:. https://nlp.stanford.edu/software/tokenizer.shtm.

[8] Stanford log-linear part-of-speech tagger [online]. available:. https://nlp.stanford.edu/software/tagger.shtml, 2003.

[9] Rqa: The requirements quality analyzer tool. [online]. available. http://www.reusecompany.com/rqa, 2012.

[10] Apache opennlp. [online]. available:. https://opennlp.apache.org/, 2013.

[11] Gate annie: A nearly-new information extraction system. [online]. available:. https://gate.ac.uk/sale/tao/splitch6.html, 2014.

[12] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer. Automated checking of conformance to requirements templates using natural language processing. *IEEE Transactions on Software Engineering*, 41(10):944–968, 2015.

[13] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[14] D. M. Berry. *Ambiguity in natural language requirements documents.* Springer Berlin Heidelberg, 2008.

[15] F. Chantree, B. Nuseibeh, A. De Roeck, and A. Willis. Identifying nocuous ambiguities in natural language requirements. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 59–68. IEEE, 2006.

[16] L. Chung and J. C. S. do Prado Leite. On non-functional requirements in software engineering. *Conceptual modeling: Foundations and applications: Essays in honor of john mylopoulos*, pages 363–379, 2009.

[17] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. Gate: A framework and graphical development environment for robust nlp tools and applications. 07 2002.

[18] C. Denger, D. Jörg, and E. Kamsties. Quasar: A survey on approaches for writing precise natural language requirements. *Fraunhofer IESE*, 2001.

[19] S. Desikan and G. Ramesh. *Software testing: principles and practice*. Pearson Education India, 2006.

[20] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An automatic quality evaluation for natural language requirements. 1, 01 2001.

[21] H. L. D. Fabian, de Bruijn. *Ambiguity in natural language software requirements: A case study*. Springer Berlin Heidelberg, 2010.

[22] 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 International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 271–274, 2011.

[23] A. Fatwanto. Software requirements specification analysis using natural language processing technique. pages 105–110, 06 2013.

[24] D. Firesmith. Analyzing and specifying reusable security requirements. *Journal of Object Technology - JOT*, 01 2003.

[25] G. Génova, J. M. Fuentes, J. Llorens, O. Hurtado, and V. Moreno. A framework to measure and improve the quality of textual requirements. *Requirements engineering*, 18:25–41, 2013.

[26] D. Graham, E. Veenendaal, I. Evans, and R. Black. Foundation of software testing, 2007.

[27] D. M. Hamish Cunningham. Developing language processing components with gate version 9 (a user guide). https://gate.ac.uk/sale/tao/tao.pdf, 2011.

[28] Z. A. Hamza and M. Hammad. Generating uml use case models from software requirements using natural language processing. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 1–6. IEEE, 2019.

[29] C. Huertas and R. Juárez-Ramírez. Nlare, a natural language processing tool for automatic requirements evaluation. pages 371–378, 09 2012.

[30] M. Kamalrudin, N. Mustafa, and S. Sidek. *A Template for Writing Security Requirements*, pages 73–86. 01 2018.

[31] M. Kassab, C. Neill, and P. Laplante. State of practice in requirements engineering: Contemporary data. *Innovations in Systems and Software Engineering: A NASA Journal*, 12 2014.

[32] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry. Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements engineering*, 13:207–239, 2008.

[33] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy approach to requirements syntax (ears). pages 317 – 322, 10 2009.

[34] D. Mellado, E. Fernández-Medina, and M. Piattini. A comparative study of proposals for establishing security requirements for the development of secure information systems. pages 1044–1053, 05 2006.

[35] J. Metsa, M. Katara, and T. Mikkonen. Testing non-functional requirements with aspects: An industrial case study. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 5–14, 2007.

[36] L. Mich, M. Franch, and P. L. Novi Inverardi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9:40–56, 01 2004.

[37] P. More and R. Phalnikar. Generating uml diagrams from natural language specifications. *International Journal of Applied Information Systems*, 1:19–23, 04 2012.

[38] F. Nazir, W. H. Butt, M. W. Anwar, and M. A. Khan Khattak. The applications of natural language processing (nlp) for software requirement engineering-a systematic literature review. *Information Science and Applications 2017: ICISA 2017 8*, pages 485–493, 2017.

[39] P. Oliveira Antonino, M. Trapp, P. Barbosa, and L. Sousa. The parameterized safety requirements templates. 06 2015.

[40] O. Ormandjieva, I. Hussain, and L. Kosseim. Toward a text classification system for the quality assessment of software requirements written in natural language. pages 39–45, 09 2007.

[41] M. Osama, A. Zaki-Ismail, M. Abdelrazek, J. Grundy, and A. Ibrahim. Score-based automatic detection and resolution of syntactic ambiguity in natural language requirements. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 651–661, 2020.

[42] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. 01 2010.

[43] K. Pohl and C. Rupp. Requirement engineering fundamentals. https://www.bbau.ac.in/dept/dit/TM/requirementsengi.pdf, 2011.

[44] R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[45] A. Rashwan. *Automated quality assurance of non-functional requirements for testability*. PhD thesis, Concordia University, 2015.

[46] M. Riaz, J. King, J. Slankas, L. Williams, F. Massacci, C. Quesada-López, and M. Jenkins. Identifying the implied: Findings from three differentiated replications on the use of security requirements templates. *Empirical Software Engineering*, 22:1–52, 08 2017.

[47] C. Rolland and C. Proix. A natural language approach for requirements engineering. *Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE*, pages 35–55, 2013.

[48] Roman. A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23, 1985.

[49] C. Rupp et al. Requirements engineering und–management: Professionelle, iterative anforderungsanalyse für die praxis. 5. *Aufl., München/Wien: Hanser*, 2009.

[50] K. Ryan. The role of natural language in requirements engineering. 02 1970.

[51] V. S. Sharma, R. Ramnani, and S. Sengupta. A framework for identifying and analyzing non-functional requirements from text. *4th International Workshop on the Twin Peaks of Requirements and Architecture, TwinPeaks 2014 - Proceedings*, 06 2014.

[52] P. Singh. Treating nfr as first grade for its testability. *Journal of Software Engineering and Applications*, 05:991–1000, 01 2012.

[53] A. Sleimi, M. Ceci, M. Sabetzadeh, L. Briand, and J. Dann. Automated recommendation of templates for legal requirements. pages 158–168, 08 2020.

[54] T. Tahvonen and E. Uusitalo. Easy approach to requirements syntax in nuclear power plant safety design. pages 1–2, 08 2018.

[55] A. Tripathy, A. Agrawal, and S. K. Rath. Requirement analysis using natural language processing. In *Fifth International Conference on Advances in Computer Engineering*, volume 26, page 27, 2014.

[56] A. Veizaga, E. Alférez Salinas, D. Torre, M. Sabetzadeh, and L. Briand. On systematically building a controlled natural language for functional requirements. *Empirical Software Engineering*, 26, 07 2021.

[57] S. Withall. *Software requirement patterns*. Pearson Education, 2007.

[58] H. Yang, A. de Roeck, V. Gervasi, A. Willis, and B. Nuseibeh. Extending nocuous ambiguity analysis for anaphora in natural language requirements. In *2010 18th IEEE International Requirements Engineering Conference*, pages 25–34, 2010.

[59] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*, pages 226–235, 1997.

[60] F. Zait and N. Zarour. Addressing lexical and semantic ambiguity in natural language requirements. In *2018 Fifth International Symposium on Innovation in Information and Communication Technology (ISIICT)*, pages 1–7, 2018.