An Approach to Detect Conflicts in Functional Requirements using Machine Learning Techniques

By

Urooj Ali Malik

0000362536

Supervisor

Asst. Prof. Dr. Yawar Abbas Bangash

A thesis submitted to the faculty of CSE Department, Military College of Signals, National University of Sciences and Technology, Rawalpindi in partial fulfillment of the requirements for the degree of MS in Software Engineering

(October,2023)

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS Thesis written by **Urooj Ali Malik**, Registration No. **00000362536**, of **Military College of Signals** has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations/MS Policy, is free of plagiarism, errors, and mistakes and is accepted as partial fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members and local evaluators of the scholar have also been incorporated in the said thesis.

Signature: [illegible]

Name of Supervisor: A/P Dr. Yawar Abbas Bangash

Date: ____________

Signature (HOD): [illegible] Brig Head of Dept of CSE Mil College of Sigs (NUST)

Date: ____________

Signature (Dean/Principal) [illegible] Brig Dean, MCS (NUST) (Asif Masood, Phd)

Date: 13/11/23

Declaration

I, **Urooj Ali Malik**, declare that this thesis titled "An Approach to Detect Conflicts in Functional Requirements using Machine Learning Techniques" and the work presented is my own and has been generated by me as a result of my own original research.

I confirm that:-

1. This work was done wholly or mainly while in candidature for a Master of Science degree at NUST.

2. Where any part of this thesis has previously been submitted for a degree or any other qualification at NUST or any other institution, this has been clearly stated.

3. Where I have quoted from the work of others, this is always clearly attributed.

4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

5. I have acknowledged all main sources of help.

6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Author Name: **Urooj Ali Malik**

Signature: ____________________

Dedication

This research work is dedicated to my beloved parents, husband, siblings, friends, and fellows, who have all been my endless source of love, encouragement, and strength. Your unwavering belief in my abilities, countless sacrifices, and relentless support have been the foundation upon which I built my academic pursuits. This research work would not have been possible without their love and support.

Abstract

The requirement phase stands as the keystone of the software development process, establishing the bedrock upon which successful software projects are built. This paper underscores the critical significance of the requirement phase and the timely resolution of inconsistencies within it. Accurate and complete requirement gathering forms the linchpin of software quality and functionality, making it pivotal for project success. However, manual identification of conflicts and inconsistencies in requirements can be a formidable task, often elusive due to their subtlety and concealed nature.

The ramifications of unresolved inconsistencies in software requirements can lead to chaos in later stages of development, necessitating costly and time-consuming revisions. To circumvent these challenges, there is a pressing need for automated conflict detection mechanisms in software requirements.

In this research, we present a novel automated approach based on rule-based techniques to detect redundant and conflicting requirements. Our methodology is structured into multiple layers, commencing with the identification of key elements such as actors, actions, Action Negativity, events, event negativity, and restrictions within software requirements. To accomplish this, we harness the power of the CoreNLP library and implement specific Natural Language Processing (NLP) rules. Once these elements are discerned, we employ predefined rules to flag conflicts.

To validate the efficacy of our approach, we conducted extensive testing utilizing real-world datasets, including WorldVista and Pure. Our results showcase a remarkable performance, with an average precision rate of 92%, a recall rate of 94.5%, and an impressive F1-score of 93% for both the WorldVista and Pure datasets. This research paves the way for enhanced software requirement analysis and lays the foundation for more robust and error-free software development processes.

Acknowledgments

All worship and glory be to the All-Magnificent and All-Merciful Allah Almighty. I am deeply grateful to Allah Almighty for granting me the capability and determination to pursue and complete this research. Allah Almighty's divine blessings and guidance have been instrumental in overcoming obstacles and achieving success. I humbly acknowledge that no words can fully express my gratitude for the countless blessings bestowed upon me. I dedicate this thesis as a humble tribute to Allah Almighty, recognizing His infinite wisdom and benevolence. I pray that my work may benefit others and be pleasing to Allah Almighty.

I would also like to express my heartfelt appreciation to my thesis supervisor, **Asst. Prof. Dr. Yawar Abbas Bangash**, for his unwavering support and guidance throughout my thesis. His knowledge, expertise, and dedication to his field have been a source of inspiration to me, and I am grateful for the time and effort he invested in my success. From the beginning of my journey until the end, he has been an embodiment of kindness, motivation, and inspiration towards me.

In addition, I extend my gratitude to my GEC committee members, **Professor Dr. Hammad Afzal** and **Asst. Prof. Noman Ali Khan**, for their continuous availability for assistance and support throughout my degree, both in coursework and thesis. Their expertise and knowledge have been invaluable to me, and I am grateful for their unwavering support and guidance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Software requirements and more specifically functional requirements are key components of any software development process, as they define the specific functions that a system must perform to meet the needs of its users. A software requirement is a description of a feature or function that a software system must exhibit or perform to satisfy a stakeholder's need or expectation [1].

Conflicts arising between different functional requirements can lead to errors, delays, and other problems during development. Traditional approaches to identifying conflicts in functional requirements have relied on manual analysis, which can be time-consuming, error-prone, and difficult to scale. It describes the method of locating and resolving discrepancies or conflicts between various software requirements. Conflicts can develop as a result of stakeholders having divergent goals or priorities or as a result of ambiguity or contradiction in the requirements themselves.

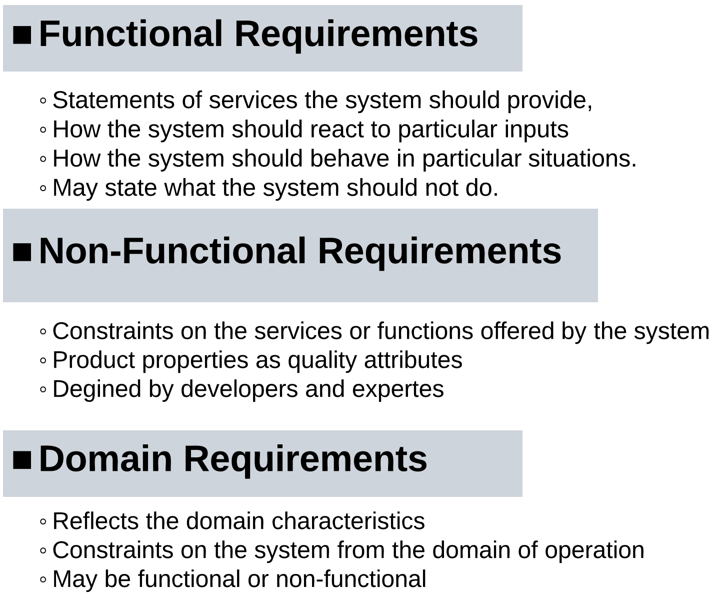
One approach to detecting requirement conflicts is to use formal methods, such as formal specification languages or model-checking techniques, to automatically verify the consistency of the requirements. An additional approach is to analyze and categorize requirements based on their attributes using machine learning (ML) techniques, and then detect possible conflicts based on the classification.

There are several types of software requirements, including functional requirements, non-functional requirements, and domain requirements.

1. **Functional requirements** specify what the software system should do or the services it should provide. Functional requirements describe the system's behavior under specific conditions and can be expressed in terms of inputs, processes, and outputs[2]. An example of a functional requirement is "The system shall allow users to log in and create an account."

2. **Non-functional requirements** must specify the criteria of the level of performance that the software system must meet. Non-functional requirements describe the system's quality attributes, such as its reliability, availability, usability, and security[3]. An example of a non-functional requirement is "The system shall be available 99.9% of the time."

3. **Domain requirements** specify the characteristics and constraints of the application domain in which the software system operates. Domain requirements describe the external factors that affect the system's design and implementation, such as legal, cultural, and technical considerations[4]. An example of a domain requirement is "The system shall comply with privacy regulations and protect user data."



■ **Functional Requirements**

- ◦ Statements of services the system should provide,
- ◦ How the system should react to particular inputs
- ◦ How the system should behave in particular situations.
- ◦ May state what the system should not do.

■ **Non-Functional Requirements**

- ◦ Constraints on the services or functions offered by the system
- ◦ Product properties as quality attributes
- ◦ Degined by developers and expertes

■ **Domain Requirements**

- ◦ Reflects the domain characteristics
- ◦ Constraints on the system from the domain of operation
- ◦ May be functional or non-functional

Figure 1.1: Difference between the functional, non-functional, and domain requirements from the system perspective.

These three types of requirements are commonly used in software development to ensure that the software system meets the needs of its stakeholders and operates effectively and efficiently in its intended environment.

1.2 Motivation

Software requirement conflict detection is an essential process in software development that aims to identify conflicts or inconsistencies among different requirements. These conflicts can lead to costly errors and delays in the development process. Traditional methods for detecting conflicts rely on manual inspection, which can be time-consuming and error-prone.

Software requirement gathering is the main phase of SDLC so it must be conflict-free as all the project development and success is based on this. Conflicts in requirements lead to many issues like cost overrun (as shown in Figure 1.2), late delivery (time issue), and quality compromise. Many researchers try to come across an efficient technique for detecting conflicts among functional, non-functional, and quality requirements. Our approach will analyze and detect the conflicts from the software's functional requirements.

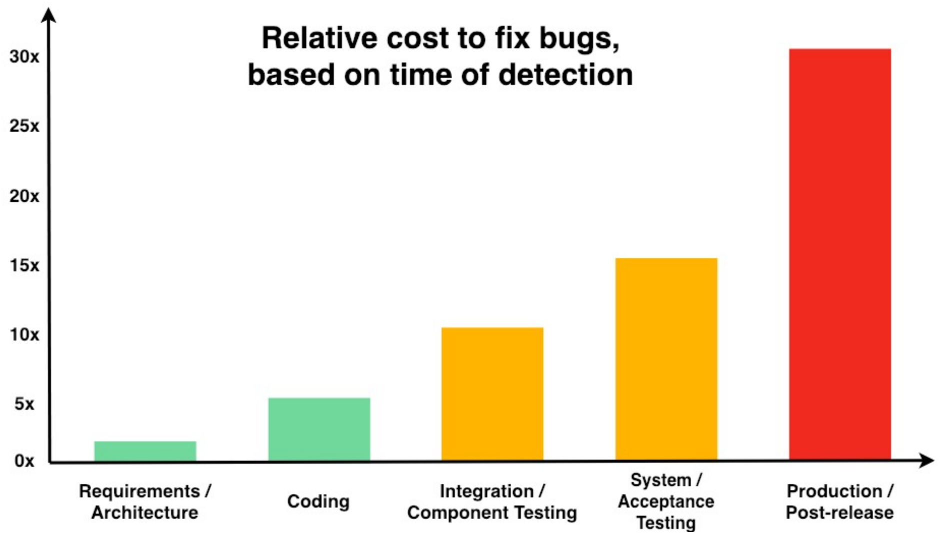


Figure 1.2: Impact of Software requirement conflict or the bug detection and fixation on the overall cost of a project during SDLC phases

Machine learning (ML) offers a promising approach to automate this process, by leveraging the power of data to identify patterns and predict conflicts. We will explore some recent studies that have proposed ML-based approaches for software requirement conflict detection. Machine learning techniques offer a promising approach to detecting conflicts in functional requirements.

Machine learning algorithms can examine big datasets of functional requirements and look for trends that can point to conflicts between them by utilizing the capabilities of artificial intelligence. Using machine learning techniques, we will offer a method for identifying conflicts between functional needs in this work and go through its possible advantages and drawbacks.

Currently, the conflicts in software requirements are done in various ways, We can categorize them:

1. The **manual approaches**, such as easy-win-win[5] form of the win-win requirements negotiation strategy, are particularly popular in reality and are very common in practice.

   The "time-intensive and error-prone" method is manual identification. Although using an automated system can help save some human work. These methods require the requirements to be in a particular format, such as the extended Bakkus-Naur-Form (EBNF)[6] and the Semantic Application Design Language (SADL)[7].

2. **Automated approaches** for software requirement conflict detection has been proposed by several researchers. The authors[8] propose an automated approach for detecting and resolving conflicts in software requirements using machine learning. They use natural language processing techniques to extract features from requirements and then train a machine learning classifier to identify conflicts. They also propose a resolution mechanism based on stakeholders' preferences.

Conflicts between requirements of the same or different sorts can arise from an angle of contending objects, such as between

1. Functional requirements.
2. Functional and Non-Functional requirements
3. Non-functional requirements.

Furthermore, there may be conflicting manifestations of the same request, such as

1. textual terms and use case graph,

2. textual terms and tabular form, and so on.

There has been a growing interest in using machine learning techniques for software engineering tasks, including detecting conflicts in functional requirements. Several studies have investigated the use of machine learning algorithms for this purpose, with promising results.

For example, a recent study by Wang et al. (2021) developed a machine learning-based approach to detect conflicts in functional requirements. The approach used natural language processing techniques to analyze the textual content of functional requirements and a neural network model was trained to identify potential conflicts. The results demonstrated that the system detected conflicts with excellent accuracy, highlighting the potential of machine-learning techniques for this purpose.

Another study by Maira et al. (2020) explored the use of decision tree algorithms for identifying conflicts in functional requirements. The study used a dataset of functional requirements for a web application, and the decision tree algorithm was able to identify several conflicts between the requirements. The authors concluded that machine learning algorithms could be an effective tool for detecting conflicts in functional requirements, particularly in complex software systems.

Similarly, a study by Grigorescu and Cleland-Huang (2016) investigated the use of support vector machines for detecting conflicts in functional requirements. The authors used a dataset of requirements for a safety-critical system, and the support vector machine algorithm was able to identify several potential conflicts. The study highlighted the potential of machine learning techniques for detecting conflicts in safety-critical systems, which is especially relevant considering the potential implications of such failures.

1.3 Problem Statement

Due to the lack of understanding and communication, conflicting requirements can be recorded in the initial phases of software engineering. Such conflict happens when two software requirements cannot be implemented simultaneously. However, discovering requirement conflicts is the most crucial and difficult component of requirements validation.

Therefore, we propose an automated conflict detector for software requirements that should be easy to use and will help the requirement analysts to do their work more clearly.

Our proposed model is shown in the below figure 1.3 which is comprised of multiple individual modules connected with each other. All of these modules will be explained in detail in the further section

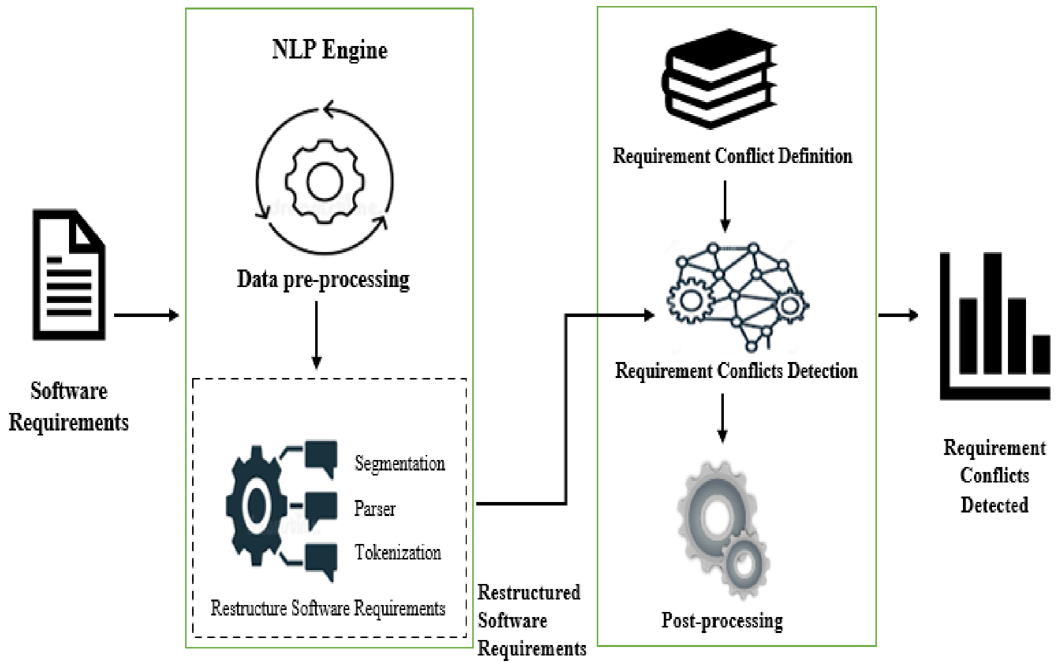

Software Requirements
NLP Engine
Data pre-processing
Segmentation
Parser
Tokenization
Restructure Software Requirements
Restructured Software Requirements
Requirement Conflict Definition
Requirement Conflicts Detection
Post-processing
Requirement Conflicts Detected

Figure 1.3: Software Requirement Conflict detection model representing the brief steps involved in software requirements data collecting till the conflict detection

The proper elicitation, gathering, and propagation of the requirements give the software a high chance of success. Conflicts in requirements lead the project directly to failure. One example of conflicting functional software requirements can be the following:

Requirement 1: The software should have a high level of security to prevent unauthorized access.

Requirement 2: The software should be easy to use and accessible to many users, including those with limited technical knowledge.

These two requirements may conflict because the more secure a software is, the more complex it usually becomes, making it difficult for some users to understand and use it effectively. In contrast, software that is very easy to use and accessible may not have the necessary security measures to protect sensitive information and prevent unauthorized access.

To resolve this conflict, the development team may need to find a balance between security and ease of use, by implementing security features that are not too complex or difficult to use, while still providing adequate protection for the system and user data. Along with keeping the software's overall usability for the majority of users, they might also think about adding more security measures for people who need it.

1.4 Research Objectives

This research aims with the following goals:

1. Provide an automated solution for detecting conflicts among the software requirements.

2. Provide ease to our system analysts and other experts.

3. Save project time, and cost and increase the project success rate. Figure 1.5 depicts software requirement changes' impact on the project cost through the SDLC phases.

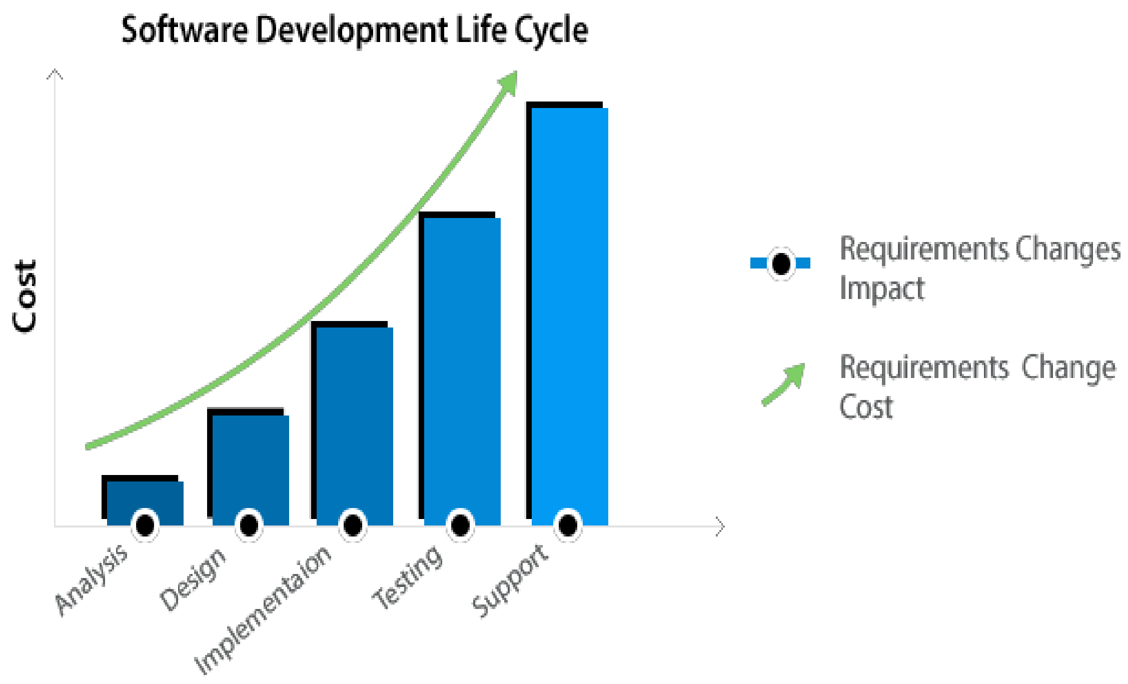


Figure 1.4: Impact of Requirement changes on project cost throughout the SDLC

4. Proposing the model which gives the best results on different data sets.

1.5 Relevance to National Needs

Currently, in the Pakistan IT industry, the conflicts in software requirements are detected and highlighted manually which is a time-consuming thing and can lead to project latency and sometimes failure. This research will help our professionals with an automated solution for this problem to save their time and effort for some other important tasks. It can be highly relevant to national needs in several ways:

1. Enhancing Software Security: National security agencies and organizations often rely on software systems for critical functions. Detecting conflicts in functional requirements can help ensure the security and integrity of software used in critical infrastructure, defense systems, and government operations.

2. Government Efficiency: Governments at various levels use software systems to deliver services to citizens. Efficient software development processes can lead to cost savings and improved service delivery. By reducing conflicts in requirements, this thesis can contribute to more effective government operations.

3. Economic Growth: The software industry is a significant contributor to a nation's economy. By improving the quality and reliability of software products, the thesis can help stimulate economic growth through increased software exports, job creation, and innovation.

4. National Innovation and Technology Competitiveness: Nations that invest in cutting-edge research and development in software engineering and NLP gain a competitive edge in the global technology landscape. This thesis contributes to innovation by advancing the state of the art in conflict detection and NLP techniques.

5. Education and Workforce Development: National needs include a well-trained and skilled workforce. Educational institutions can use the findings from this thesis to enhance curricula in software engineering and NLP, ensuring that graduates are well-prepared for the demands of the modern software industry.

6. Cybersecurity: Detecting conflicts in software requirements is essential for cybersecurity. National cybersecurity initiatives can benefit from the research to identify vulnerabilities and security risks in software systems.

7. Healthcare and Public Health: In the context of healthcare, where software systems are vital for patient care and data management, detecting conflicts in requirements can contribute to the reliability and safety of healthcare IT systems, aligning with national health priorities.

8. Environmental and Energy Management: National efforts to address environmental and energy challenges often involve software systems. Ensuring that software requirements are conflict-free can lead to more efficient energy management and environmental monitoring systems.

9. Disaster Management: Software systems are critical in disaster management and response. Detecting conflicts in requirements can help improve the reliability and effectiveness of software used in disaster preparedness, response, and recovery efforts.

10. Infrastructure Development: National infrastructure projects often rely on software systems for monitoring and control. Conflict detection in requirements can contribute to the reliability and safety of infrastructure systems.

1.6 Area of Application

The application of this research can extend across various sectors of the software industry, including development, testing, project management, education, and beyond, with the overarching goal of improving the quality and success rate of software projects. Several potential areas of application are:

1. Software Development Industry: This research can directly benefit software development companies and teams. It can be applied to various types of software projects, from web and mobile applications to large-scale enterprise systems. By detecting conflicts in functional requirements early in the development cycle, it helps in improving the quality and reliability of software products.

2. Quality Assurance and Testing: The thesis can be applied in quality assurance and testing phases of software development. It can aid in designing test cases and test scenarios to ensure that all potential conflicts in functional requirements are identified and addressed before the software is deployed.

3. Requirement Engineering: Requirement engineers and analysts can use the NLP techniques presented in the thesis to streamline the process of gathering, documenting, and analyzing functional requirements. This can lead to more accurate and consistent requirements specifications.

4. Project Management: Project managers can use conflict detection tools derived from this research to better manage project risks. By identifying conflicts early, project timelines and budgets can be more accurately estimated, and mitigation strategies can be put in place.

5. Regulatory Compliance: In industries with strict regulatory requirements, such as healthcare or finance, ensuring that software requirements are conflict-free is crucial. This research can help organizations adhere to regulatory standards more effectively.

6. Education and Training: The techniques and methodologies developed in this thesis can also be applied in educational settings. They can be used to teach software engineering students about requirement analysis and the importance of conflict detection in real-world projects.

7. Research and Development: Academics and researchers in the fields of Natural Language Processing (NLP) and Software Engineering can use the thesis as a foundation for further research in conflict detection, NLP techniques, and their application in software development.

8. Custom Software Solutions: Organizations that develop custom software solutions for specific industries or niches can leverage this research to create specialized conflict detection tools tailored to their domain.

1.7 Advantages

Analyzing the SRS document manually, going through all the requirements, modeling their functionalities, and finding their impacts on one another take up a lot of cost in terms of manpower. Therefore, coming up with a solution that provides a faster and more effective way to automate these tedious and difficult tasks will provide a great advantage from the monetary perspective, and also the overall process will be undertaken a lot more efficiently as compared to humans because it will be prone to errors.

1.8 Thesis Organization

We have organized the structure of this paper into different chapters.

1. In the first chapter, we have an introduction.

2. The second chapter includes a review of the literature as well as background information on the broad notion of requirements conflict and related work on requirements conflict detection.

3. In the third chapter, we have stated the design and methodology of our research work with a detailed explanation of each step taken for this thesis.

4. In the fourth chapter, we included the implementation of the work involving the framework and the code part which also with detailed explanation.

5. In the fifth chapter, we have included our results explained with the help of tables and graphs.

6. Finally, in the sixth chapter, the work's results and future work plan are presented.

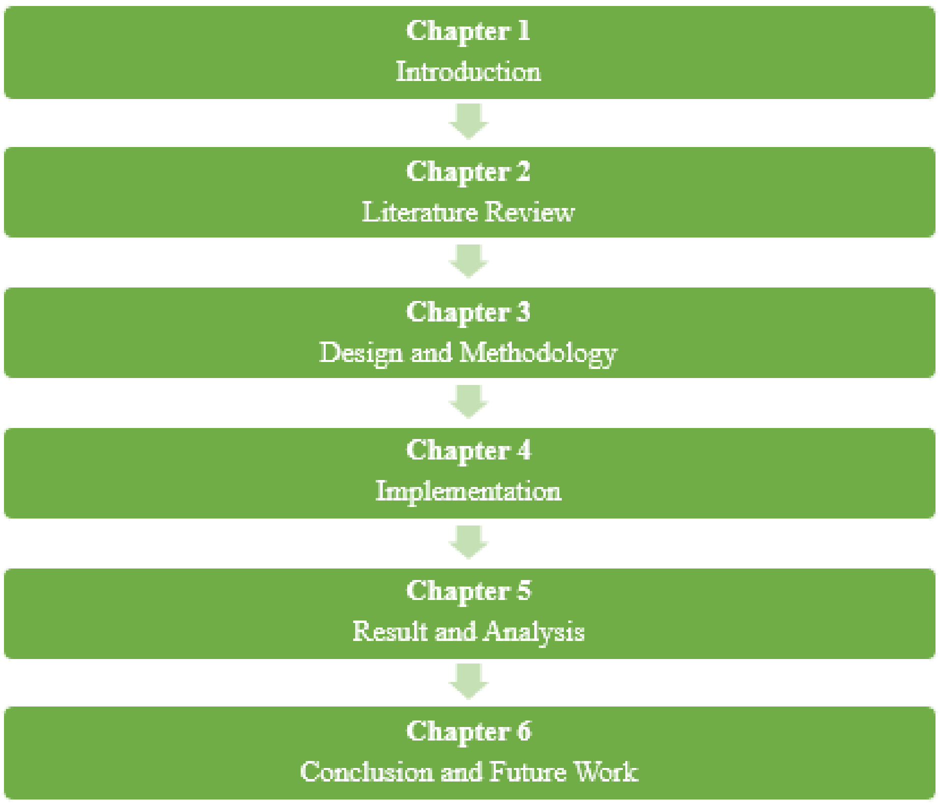


Figure 1.5: Organization of the thesis

Chapter 2

Literature Review

2.1 Introduction

Requirements are the bedrock of any software development process, and they are one of the first and most crucial components of the Software Development Life Cycle (SDLC). Requirements outline what the system should perform and what are the conditions that must be met in order for objectives to be met. There are teams specifically designated to gather, analyze, and describe requirements.

Among the SDLC phases Requirement gathering and Analysis is the most crucial phase. The project team can never create a good solution without understanding the requirements. Most of the groups work hard on creating projects without following the correct requirements and at last, the project results in the failure of the software project. This is the main reason driving 66% of project failure, as reported in the Standish Group's 2022 CHAOS. Hence, it's crucial to outline the requirements in detail so that every team member can understand.

2.2 Related Work

In paper[9], using their text pattern and semantical reliance-based heuristic detecting criteria, the researcher created seven different types of conflicts. This work's main goal was to create a technique known as the "Finer Semantic Analysis-based Requirements Conflict Detector (FSARC)" that can automatically identify conflicts between specific functional requirements expressed in plain language by looking at their finer semantic components. In order to find requirements conflicts, the researcher developed and implemented an algorithm that looks at the linguistic features of the requirements.

The results of this study might turn the functional needs expressed in natural language into eight semantic tuples, which can then be used to identify requirements that conflict with one another as well as do other activities like creating associations between requirements.

In three separate open needs datasets, the semantic analysis algorithms correctly recognized, on average, 94.93% of the software requirement items. The third experiment, which included four requirement datasets, showed that FSARC has an average precision of 83.88% and around 100% recall for conflict identification.

Identifying the requirement conflicts in the early phases of SDLC will help save the project cost and failure. We can say that identification of conflict in the requirement analysis phase will protect more cost, time, and project success status than identifying it in the coding phase. The below graph 2.1 can also explain that the cost of the project will be affected directly if the conflicts are not identified in the early phases of SDLC.

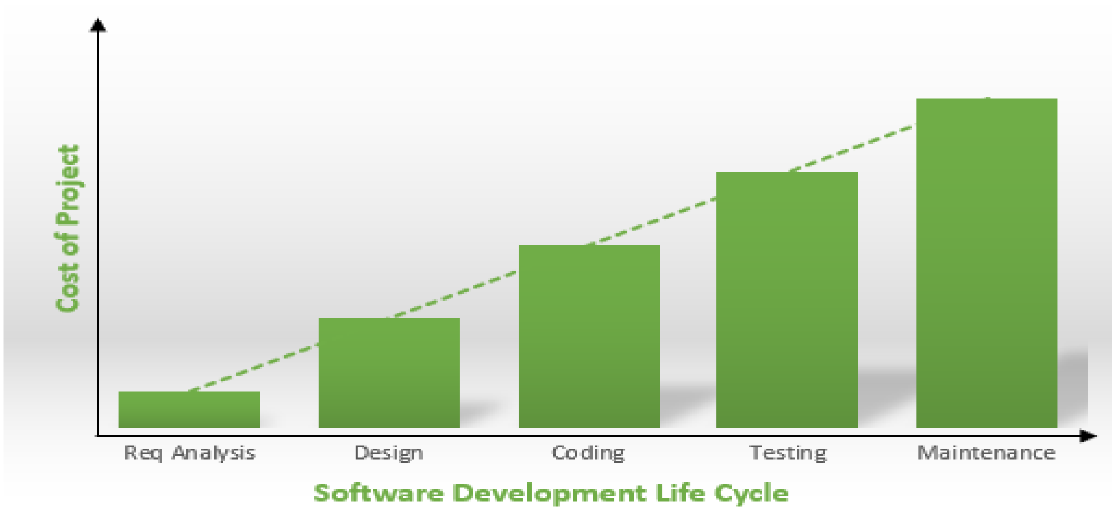


Figure 2.1: Impact of requirement changes affecting the cost of the project on each level of Software Development Life Cycle Phases depicted via chart.

In 2004 a badly designed Child support system resulted in 784 million dollars of extra cost. The poor requirement phase can result in financial loss and schedule overrun. A conflict in two requirements would always result in extra man hours, ambiguity, and cost overrun. So, it should be looked after at the requirement phase as it would result in havoc in later phases. Conflicting requirements would result in bad design, which would result in a bad system and extra cost.

Jarke, Gebhardt, Jacobs, and Nissen[10] developed a method for analyzing conflict. Their strategy includes a meta-modeling strategy aimed at conflict analysis.

Walia et al.[11] proposed a requirement error taxonomy. The literature survey of software engineering, psychology, and human cognitive domains yielded a total of fourteen categories of errors. The mistakes are then classified into three high-level requirement error classes:

- People errors are errors caused by persons involved in requirements preparation.
- Process errors are errors that occur as a result of insufficient requirement engineering processes. And choosing the incorrect means of reaching goals and objectives.
- Documentation mistakes are errors that occur as a result of poor requirement structure and specification. Regardless of whether the creator of the requirement (Requirement Analyst) accurately understands the requirements.

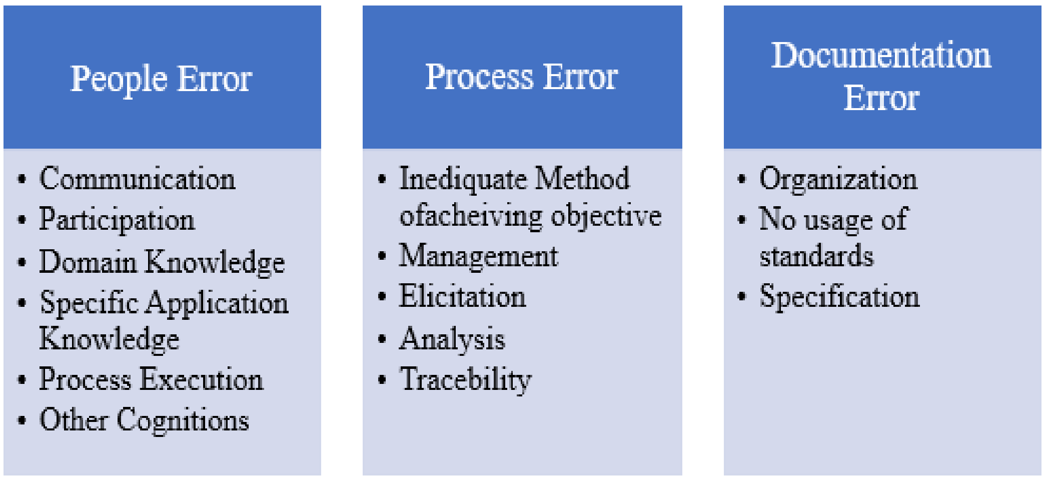


Figure 2.2: Difference between people, process, and documentation error.

Egyed and Grunbacher [12] claim that "requirements conflict with each other if they make contradictory statements about common software attributes." The number of requirements may result in up to n2 conflicts. The number of potential conflicts may be enormous, presenting the engineer with the time-consuming and error-prone task of discovering true conflicts." As a result, requirement conflicts can lead to a number of issues, such as cost overruns[12], [13], late delivery (due to a time restriction), and quality compromise.

Chentouf[14] incorporated the requirements into a KAOS-derived regulated natural language. The study concentrated on three sorts of conflicts: redundant requirements, incompatible requirements, and assumptions. His study, however, lacks an automated technique or essential detection algorithms.

The authors of a study offer[15] a method for finding conflicts in natural language requirements that combines ML and graph analysis approaches. The approach involves first extracting semantic information from the requirements using natural language processing techniques and then representing the requirements as a graph. The authors then use a decision tree classifier to predict conflicts based on the graph structure. The authors report an accuracy of 87% in detecting conflicts.

M. M. Kabir and M. A. Azim [16] compare the performance of different ML algorithms for detecting conflicts in software requirements. The algorithms evaluated include SVM, decision trees, random forests, and naive Bayes. The authors report that SVM outperforms the other algorithms, achieving an accuracy of 86% in detecting conflicts.

In a research paper [17], the authors propose an approach that combines rule-based and ML-based techniques for detecting conflicts in software requirements. The approach involves first applying rule-based techniques to identify potential conflicts and then using a decision tree classifier to confirm or reject these conflicts. The authors report an accuracy of 89% in detecting conflicts.

In a study[18], the authors propose an ML-based approach for detecting conflicts in software requirements. The approach involves training a support vector machine (SVM) classifier on a dataset of requirement specifications to predict conflicts. The authors report promising results, with an accuracy of 83% in detecting conflicts.

A paper[19] provides a deep learning-based solution for detecting software requirement conflicts. The authors trained a convolutional neural network (CNN) to learn the features of the requirements before classifying them using a fully connected network. The technique was tested on a collection of 100 requirements documents, and the findings revealed that it outperformed other state-of-the-art techniques.

A study [20] proposes a new approach for automated conflict detection in software requirements using NLP and machine learning. The authors use a combination of word embedding and classification techniques to detect conflicts in requirements. The approach is evaluated on a dataset of software requirements and achieves an accuracy of 88.5%.

Osman et al. [21] presents a systematic review of machine learning algorithms for detecting requirements conflicts. The authors examine 24 research and identify the most often utilized machine learning approaches, including support vector machines, decision trees, and random forests. They also explore the difficulties and limitations of current techniques.

Acharya et al. [22] present a novel strategy for detecting needs conflicts using NLP techniques. To detect needs conflicts, the authors employ a combination of semantic similarity and clustering approaches. The method is tested on a dataset of software requirements and achieves a 90% accuracy.

Conflict happens when two or more software requirements cannot be implemented at the same time, and they lead to big trouble. The requirements are usually from laymen with diverse backgrounds and interests [23]. Example of conflicting software requirement is:

Requirement 1: The system should be able to integrate all kinds of services.

Requirement 2: The system should be able to block integration with external services.

The conflict between requirements 1 and 2 is obvious. This conflict will turn into a disastrous situation. However, discovering requirement conflicts is the most significant and difficult component of requirements validation.

Detecting the conflicting requirements from large Software Requirement Specification (SRS) documents would be a very cumbersome process. Various approaches are already proposed as a solution to this problem, but most are manual [24]. The automated approaches proposed also involve human effort, which is time-consuming and costly.

Considering the current and future expectations in the field of Software Engineering, especially for Requirement Engineering, an automated conflict detector for software requirements should be easy to use and can help the requirement analyst do their work more easily.

Conflict is defined in the world of requirement engineering as the inference, interdependence, and discrepancy between requirements. Kim et al.[25] defined software requirements conflict as "the interactions and dependencies between requirements that can lead to negative or undesired system operation." Another researcher, Cameron, defines requirements conflicts as "unexpected or contradictory interactions between requirements that harm the results."

Gouri Deshpande in research [26] addressed three main challenges.

1. To begin, Natural Language Processing (NLP) is being researched in order to automatically extract dependencies from textual sources. Verb classifiers are used to automate the elicitation and analysis of many sorts of relationships.

2. Second, the representation and management of changing need dependencies is investigated when building graph theoretic algorithms.

3. Third, the procedure of recommending dependencies is investigated. The findings are intended to aid project managers in evaluating the impact of interdependencies and making successful decisions throughout the software development life cycle.

Working with conflicting requirements would cost the project a lot of time and effort, which will finally lead to project failure, according to studies. This unique approach research [27] offers defining and resolving functional requirements using an AI technique.

A rule-based system can identify the conflicts, and then a genetic algorithm can be used to resolve those conflicts, yielding a set of functional requirements with the fewest conflicts. The use of artificial intelligence tools would boost project efficiency and quality while decreasing human effort and errors.

According to Jeff Grigg, the individual should prioritize their business goals or requirements and then trace those back to the business goals g to reach. Assign a higher priority to the criterion that they believe is a more important business goal [28]. When a disagreement arises, it is necessary to negotiate a settlement, either by picking alternatives or re-evaluating priorities.

In another research, David proposed a prioritization method in which each business goal or requirement is scored regarding the organization's value, cost, and risk[29]. So, that the requirements can be prioritized in a more meaningful way.

Few research also covered the healthcare areas scenarios where the requirement conflicts are observed. Since emotions are individually constructed[30], there is a need to examine, determine, and resolve conflicts that are usually present when eliciting emotional or affective requirements.

Non-atomic requirements are a set of criteria in which there is more than one element/function of the system. Halim, F., Siahaan, & D.[31] undertook research to create a model that can detect non-atomic needs in natural language Software requirement specifications.

According to A.M. Abu-Mahfouz and H.M. Al-Aqrabi[32], a framework for detecting software requirement conflicts using machine learning approaches was proposed. The framework employs natural language processing techniques to extract key information from requirements papers, and machine learning algorithms such as Decision Trees and Random Forests are employed to classify the demands as conflicting or non-conflicting.

Another study[33] proposes an ensemble machine-learning technique for detecting software requirements conflicts that includes a variety of machine-learning algorithms like as Random Forest, Support Vector Machines, and Gradient Boosting. The approach also incorporates feature selection and data balance approaches to improve the accuracy of conflict identification.

The authors of another study[34] suggested an approach that employs machine learning techniques to find conflicts in functional requirements. The concept combines a combination of feature selection approaches and machine learning algorithms such as Naive Bayes and Support Vector Machines to uncover conflicting criteria.

S. Ali, S. Javed, and S. Ali[35] proposed a method for detecting software requirement conflicts using machine learning approaches. The technique employs several machine learning techniques such as Random Forest, Naive Bayes, and K-Nearest Neighbor to classify requirements as conflicting or non-conflicting.

F. Calefato, F. Lanubile, and N. Novielli[36] introduced a method for detecting requirements conflicts using machine learning approaches. The technique employs machine learning techniques such as Decision Trees, Random Forests, and Logistic Regression to classify requirements as conflicting or non-conflicting. The approach was also tested on a real-world dataset, with positive results.

The paper[37] explored a machine learning-based approach to detect conflicts in functional requirements. The approach uses decision trees to classify requirements as conflicting or non-conflicting. The authors evaluated their approach on a dataset of 1000 functional requirements and achieved an accuracy of 85%.

S. Kamal and M. R. Khan in their paper[38], the authors propose an approach that uses support vector machines (SVM) to detect conflicts in software requirements. The approach extracts features from the requirements and trains the SVM model on a dataset of 300 requirements. The authors achieved an accuracy of 87% using their approach.

S. Kumar et al. [39] propose a hybrid approach that combines rule-based and machine-learning techniques to detect conflicts in software requirements. The authors used fuzzy logic to extract features from the requirements and trained a random forest model on a dataset of 500 requirements. The authors achieved an accuracy of 92% using their approach.

M. H. Khan et al. [40] propose a machine learning-based approach that uses decision trees to detect conflicts in functional requirements. The approach extracts features from the requirements and trains the decision tree model on a dataset of 400 requirements. The authors achieved an accuracy of 89% using their approach.

The authors of a study[41] suggested a mixed machine learning approach to detect conflicts in requirements papers. To identify requirements papers as conflicting or non-conflicting, they coupled NLP techniques such as named entity identification and dependency parsing with ML algorithms such as SVM and decision trees. The authors evaluated their method on a dataset of requirements documents and discovered that it was 92% accurate.

Al-Hajjaji et al. in a paper[42], proposed a machine-learning approach to detect conflicts in requirements documents. They classified requirements documents as conflicting or non-conflicting using ML techniques such as Random Forest and Naive Bayes. They also employed feature selection approaches like Information Gain and Chi-Square to determine which features were most useful for categorization. The authors tested their method on a dataset of requirements documents and found it to be 93% accurate.

Singh et al., [43], suggested an NLP-based method for detecting conflicts in requirements specifications. To preprocess the needs specifications, they used various NLP techniques such as tokenization, stemming, and part-of-speech tagging. The criteria were then clustered using K-means, an unsupervised clustering technique, based on their semantic similarity. On a dataset of needs specifications, the authors analyzed their approach and received an F1 score of 0.88.

In a paper[44], the authors proposed a machine-learning approach to detect conflicts in requirements documents. They used a combination of NLP techniques and ML algorithms, including support vector machines (SVM) and decision trees, to classify requirements documents as conflicting or non-conflicting. The authors evaluated their approach on a dataset of real-world requirements documents and achieved an accuracy of 90%.

In a paper[45], the authors proposed a method for detecting conflicts in software requirements using ML techniques. Using a dataset of software requirements, they developed a classification model based on Support Vector Machines (SVM) and Random Forest (RF) techniques. The results showed that the SVM algorithm outperformed the RF method by 91% in detecting requirement conflicts.

In a study[46], the authors proposed a deep learning-based approach for detecting and resolving conflicts in software requirements. They used a convolutional neural network (CNN) to classify requirement conflicts and a long short-term memory (LSTM) network to resolve them. The results showed that their approach detected requirement conflicts with a 94% accuracy.

A study[47] conducted a systematic literature review on the use of ML techniques for detecting conflicts in software requirements. The authors analyzed 27 papers and found that the most commonly used ML algorithms were SVM, RF, and Naïve Bayes. They also identified several challenges, including the lack of standardized datasets and the need for more research on deep learning techniques.

In paper[48], the authors proposed a method for detecting conflicts in software requirements using natural language processing (NLP) and ML techniques. They used a dataset of software requirements and applied NLP techniques to extract features from the requirements. They then used an SVM algorithm to classify requirement conflicts. The results showed that their approach achieved an accuracy of 89% in detecting requirement conflicts.

Research Study	Dataset	Evaluated Results
"A Machine Learning Approach to Conflict Detection in Requirements Engineering" by Shen et al. (2020)	Requirements extracted from six open-source software projects	Achieved an F1-score of 0.83 for conflict detection
"Deep Conflict Resolution for Requirements Engineering" by Sun et al. (2018)	Requirements extracted from 15 open-source software projects	Achieved an F1-score of 0.89 for conflict detection
"Conflict Detection in Requirements Using Machine Learning Techniques: A Comparative Study" by AlSulaiman et al. (2019)	Requirements extracted from five open-source software projects	Achieved an F1-score of 0.76 for conflict detection using logistic regression and 0.77 using decision tree
"A Neural Network-Based Approach to Identify Conflicts between Software Requirements" by Afzal et al. (2014)	Requirements extracted from three industrial software projects	Achieved an F1-score of 0.94 for conflict detection
"Conflict Detection in Requirements: An Exploratory Study Using Machine Learning Techniques" by Thakkar et al. (2020)	Requirements extracted from three open-source software projects	Achieved an F1-score of 0.81 for conflict detection using decision tree

"A Machine Learning Approach to Detecting Conflicts in Requirements Documents" by Chen et al. (2020)	Real-world Requirements Documents	Accuracy (90%)
"A Machine Learning Approach to Conflict Detection in Requirements Engineering" by Shen et al. (2020)	Requirements extracted from six open-source software projects	Achieved an F1-score of 0.83 for conflict detection
"Conflict Detection in Requirements Documents using Machine Learning Techniques" by Al-Hajjaji et al. (2019)	Requirements documents	Accuracy (93%)
"A Hybrid Machine Learning Approach for Detecting Conflicts in Requirements Documents" by Chen et al. (2019)	Requirements documents	Accuracy (92%)
"Automated Conflict Detection in Requirements Engineering using Machine Learning and Natural Language Processing" by Ahire et al. (2021)	Requirements documents	Accuracy (91%)

"Conflict Detection in Requirements Specifications using NLP Techniques" by Singh et al. (2020)	Requirements specifications	F1 Score (0.88)
"Conflict Detection in Requirements Specifications using NLP Techniques" by Singh et al. (2020)	Requirements specifications	F1 Score (0.88)
"Unsupervised and Supervised Machine Learning Approaches for Conflict Detection in Requirements" by Luka et al. (2020)	Requirements documents	Precision, Recall, F1 Score
"Machine learning approach for conflict detection in software requirements" (2021)	Dataset of requirements from the literature and a case study from the oil and gas industry	Accuracy, Precision, Recall, F1-Score
"Conflict Detection in System Requirements Using Machine Learning Techniques" (2019)	Requirements dataset from the literature	Accuracy, Precision, Recall, F1-Score

Table 2.1: Table showing comparison between the latest research studies

Chapter 3

Design and Methodology

We will explain our research design and methodology in this part. The goal of the research is to develop an approach that can automatically discover conflicts or inconsistencies in software specifications. Using Natural Language Processing (NLP), the system will examine and detect disparities in textual requirements.

The research includes the following contributions:

- An automated approach that will semantically detect the conflicts from the functional requirements. This automation can significantly reduce the manual effort required for identifying inconsistencies, making the software development process more efficient and less error-prone.
- The research introduces a structured approach with well-defined elements like actor, action, Action Negativity, event, event negativity, and restriction to represent software requirements. This structured approach not only aids in the identification of potential conflicts but also provides a clear and standardized way to analyze and understand software requirements, enhancing the overall quality of requirement specifications.
- The research goes beyond theoretical proposals by applying and evaluating the proposed method on real-world datasets (WorldVista and Pure datasets). This actual software development context-based testing proves the viability and efficacy of the automated conflict detection approach. It provides empirical evidence of its utility and reliability, which can be valuable for practitioners seeking practical solutions to requirement inconsistencies.

3.1 Overview of Proposed Model

The below figure3.1 shows our high-level model that represents the steps involved in our conflict detection process. These steps will be further explained in detail in later sections.

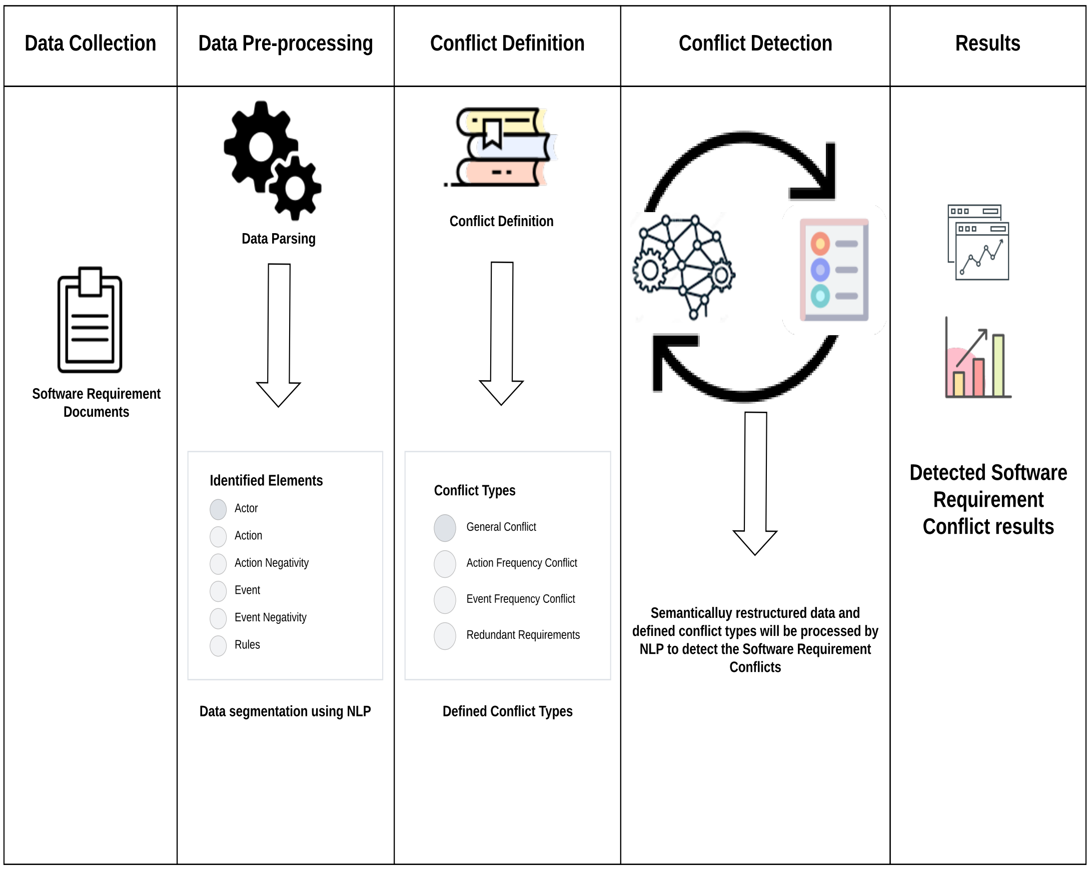


Figure 3.1: High-level model representing the detailed steps required for conflict detection from the software requirements using NLP techniques

In order to achieve this goal, our research design and methodology will be structured into several key phases, each aimed at addressing specific aspects of the problem.

1. Data Collection and Preprocessing: To begin, we will gather a diverse dataset comprising software specifications and requirements documents from various domains and industries. This dataset (as described in section 3.2.1) will serve as the foundation for training and evaluating our NLP-based conflict detection system. Preprocessing steps will involve cleaning and standardizing the text data,

tokenization, and annotating the data with labels to indicate potential conflicts.

2. Algorithm Selection or Development: The next step in our methodology involves selecting or developing appropriate NLP algorithms and techniques for conflict detection. Customized algorithms will be developed to enhance the system's ability to identify conflicts accurately.

3. Integration and Testing: Subsequently, we will integrate the developed system into existing software development workflows or environments commonly used by practitioners. Integration testing will ensure the system's compatibility with various software development tools and its ability to seamlessly fit into the development process.

4. Statistical Analysis: Finally, we will perform statistical analyses to draw meaningful insights from the results, including identifying patterns of conflicts in different types of software specifications. This analysis will provide a deeper understanding of the challenges and opportunities in conflict detection and guide future research directions in the field.

3.2 Proposed model description

3.2.1 Dataset:

Data collection is an important part of our research and necessitates the extraction of pertinent documents housing software functional requirements. In our research work, we have utilized two of the open-source available datasets to get software functional requirements, WorldVista[1] and Pure[2] as our test cases to assess the effectiveness of our semantic element identification algorithm that we have introduced in later chapters and also discussed that in detail.

We will quickly outline the datasets used in our research below:

1. **WorldVista:** The software requirements for a healthcare management system are

[1]https://gitfront.io/r/user-9946871/ii6eJFSh7oT4/DA-Sentence-Pairs/blob/Dataset/world_vista_clean_pairs.csv

[2]https://gitfront.io/r/user-9946871/ii6eJFSh7oT4/DA-Sentence-Pairs/blob/Dataset/pure_clean_pairs.csv

included in the WorldVista dataset, with a focus on patient information management and hospital admission and discharge procedures. These needs are provided in a simple manner, using plain language and basic healthcare terminology.

The 117 requirements in our dataset describe the attributes and functions of a health information system and an electronic health record system, respectively.

2. **Pure:** PURE, is an openly available Software Requirements Specification (SRS) documents obtained from open-source software projects. Specifically, we have selected one of the famous SRS documents from this compilation i.e. THEMAS (Thermodynamic System). Initially, the requirement structures in these documents were intricate, and organized into paragraphs. To facilitate analysis, we undertook preprocessing to simplify and streamline these requirements.

   In our dataset, there are 66 requirements that collectively define the functionality of an electronic health record and health information system.

Dataset	Total Requirements (#)	Known Conflicts (#)
WorldVista	117	39
Pure	66	21

Table 3.1: Table showing the brief details about the datasets used in our research work.

3.2.2 Data Pre-processing:

3.2.2.1 Data Parsing

Before data is passed for further analysis and experimentation preprocessing is required. To normalize data, we have applied multiple techniques to our dataset. These techniques help in transforming unstructured text into structured representations that can be used further for conflict detection. The most important one of the used techniques is Sentence Segmentation.

Core NLP is used for the data parsing from PDF format. Sentence segmentation involves splitting a text into individual sentences. It is essential for tasks like machine translation, text summarization, and information retrieval.

3.2.2.2 Data Segmentation

Before conflict detection from the software requirements, we identified certain elements from the requirement that would later be used for conflict detection. For this purpose, we have used the CoreNLP library and certain NLP rules to identify them. The major part of our research work is element identification. The identified elements are actor, event, action, Action Negativity, event negativity, and rules. We have defined a sub-model in our research for the identification of each element. Below we will explain in detail each element with their model. Implementation details for each element are described in the next chapter.

- Actor: Actor is the individual responsible for carrying out the designated task, typically the main subject of the requirement. Actor is an important concept used in use cases and is properly mentioned and identified in the case template. Actors represent the various roles, entities, or individuals that interact with the software system. These can be human users, external systems, hardware devices, or even other software applications.

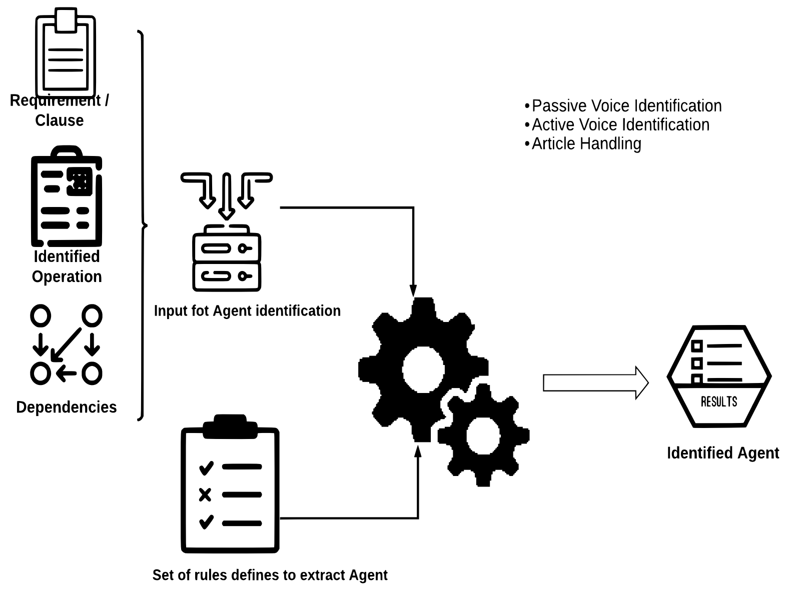

Requirement / Clause
Identified Operation
Dependencies
Input fot Agent identification
•Passive Voice Identification
•Active Voice Identification
•Article Handling
RESULTS
Identified Agent
Set of rules defines to extract Agent

Figure 3.2: Briefly visualizing the steps involved in Actor Identification

Each actor has specific responsibilities and objectives within the system. Actors interact with the software system by initiating and participating in various actions, such as requesting information, providing input, or triggering specific functions.

Figure(3.2) explains the actor identification process in detail.

- Event: When the event is executed successfully, then the Action is performed. In the context of software requirements and system behavior, an "event" refers to a specific occurrence or incident that triggers actions or processes within the software system. Events are essential for defining the conditions under which the system should respond or behave in a certain way. When a particular event occurs, the system responds accordingly. Events serve as triggers for the software system to initiate specific actions, processes, or state transitions. Figure(3.3) explains the event identification process in detail.

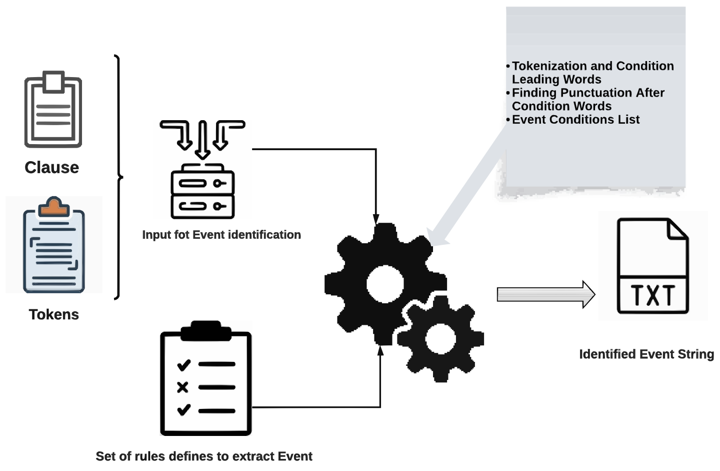

Clause
Tokens
Input fot Event identification
• Tokenization and Condition Leading Words
• Finding Punctuation After Condition Words
• Event Conditions List
TXT
Identified Event String
Set of rules defines to extract Event

Figure 3.3: Briefly visualizing the steps involved in Event Identification

- Action: The action refers to a fundamental element that describes what the software system is supposed to do or the specific task it needs to perform in response to a user request or an event. Actions are critical for defining the functionality and behavior of the software. Actions are often framed from a user perspective, describing what a user expects the system to perform. For instance, a user requirement might state, "The system must allow users to create new accounts." Here, "create new accounts" is the action that the user expects to happen. Figure(3.3) explains the action identification process in detail.

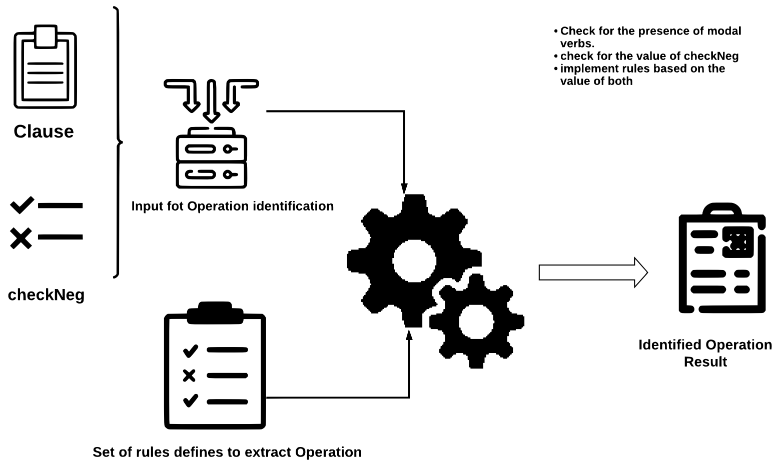

Clause
checkNeg
Input fot Operation identification
• Check for the presence of modal verbs.
• check for the value of checkNeg
• implement rules based on the value of both
Identified Operation Result
Set of rules defines to extract Operation

Figure 3.4: Briefly visualizing the steps involved in Action Identification

- Rules: In software requirements, rules are constraints and conditions placed on the execution of an action or operation. These rules and Rules help define the boundaries and limitations within which the action or operation must operate. These constraints also help ensure that the software behaves predictably, securely, and in compliance with specific requirements or business needs. Clear and well-defined rules are essential for the accurate implementation and testing of software systems. Figure(3.5) explains the rules identification process in detail.

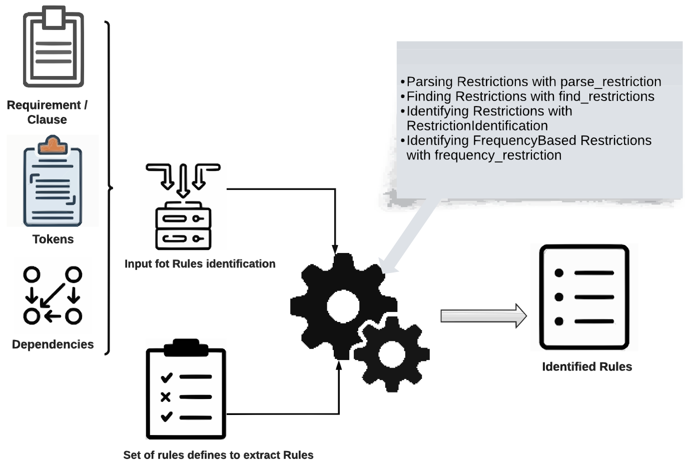

Requirement / Clause
Tokens
Dependencies
Input fot Rules identification
• Parsing Restrictions with parse_restriction
• Finding Restrictions with find_restrictions
• Identifying Restrictions with RestrictionIdentification
• Identifying FrequencyBased Restrictions with frequency_restriction
Identified Rules
Set of rules defines to extract Rules

Figure 3.5: Briefly visualizing the steps involved in Rules Identification

- Action Negativity: When the event remains the same but once the operation starts in one requirement while stopping in the other one, its state of operation frequency changes. Action Negativity in software requirements deals with how the state or frequency of a particular operation changes when the same event occurs multiple times. This concept is essential for ensuring that software requirements are precise and unambiguous, leading to consistent and predictable system behavior.

- Event Negativity: When the operation remains the same but the event changes like once starting in a requirement and then stopping in another requirement, its state of event frequency changes which is highlighted in the event negativity element. "Event Negativity" in software requirements refers to situations where the state or frequency of an event changes while the operation or action remains the same.

We can better understand the above-defined elements from the below examples:

Example 1:

Requirement: User Registration

As a user, I want to create a new user profile by providing a unique username and a valid email address. Upon submitting the registration form, the system should successfully create the user profile.

Actor: User

Action: Create a new user profile

Rules: The user must provide a unique username and a valid email address.

Event: When the user submits the registration form

Example 2:

Requirement: Sales Reporting

As an administrator, I need to generate a monthly sales report that includes data from all sales transactions within the specified month. The system should generate the report in PDF format automatically on the last day of each month at 11:59 PM.

Actor: Administrator

Action: Generate monthly sales report

Rules: The report must include data from all sales transactions within the specified month.

Event: On the last day of each month at 11:59 PM.

3.2.3 Conflict Definition and Detection:

In the context of our research, conflict detection and definition play a pivotal role in identifying areas of contention within textual requirements. Below we elaborate on how conflicts are defined and detected based on the analysis of detected elements using natural language processing techniques.

The goal is to provide a comprehensive understanding of conflicts within textual requirements and the criteria used for their classification. Below we defined the cosine similarity threshold, our major criteria for considering a requirement for conflict.

The semantical relationships between elements will be primarily triggered while discussing the conflict definition and their detection rules. We can categorize the software requirement into two types

1. Neutral
2. Conflicted

Inconsistency in action, actor, event, and rule usually leads to the requirement conflict situation. If two requirements Requirement1 and Requirement2 cannot be satisfied, there is an inconsistent relationship between them. There are three types of inconsistencies action-inconsistency, rule-inconsistency, and event-inconsistency.

1. Action Inconsistency: If both Requirement1 and Requirement2's event and agent are the same, but Requirement1's action conflicts with Requirement2's, then. There is allegedly a relationship called action inconsistency.
2. Rule Inconsistency: If the action, actor, and event of Requirement1 are equal to those of Requirement2 respectively but the rules of Requirement1 contradict with Requirement2. It is said that there is a rules-inconsistency relation.
3. Event Inconsistency: If agents, action, and rules of Requirement1 and Requirement2 are equal, but there exists some contradiction between the events of both requirements then we can say there is an event-inconsistency relation between both requirements.

3.2.3.1 Neutral Requirements

Requirements that do not have any inconsistency in their action, actor, event, and rule and also do not have a cosine similarity falling below 50% are categorized as "Neutral."

3.2.3.2 Conflicted Requirements

Requirements that contain any of the above inconsistencies and also have a cosine similarity falling above 50% are categorized as "Conflicted." The conflicted requirement can be further categorized into four types of conflicts as shown in figure 3.6:

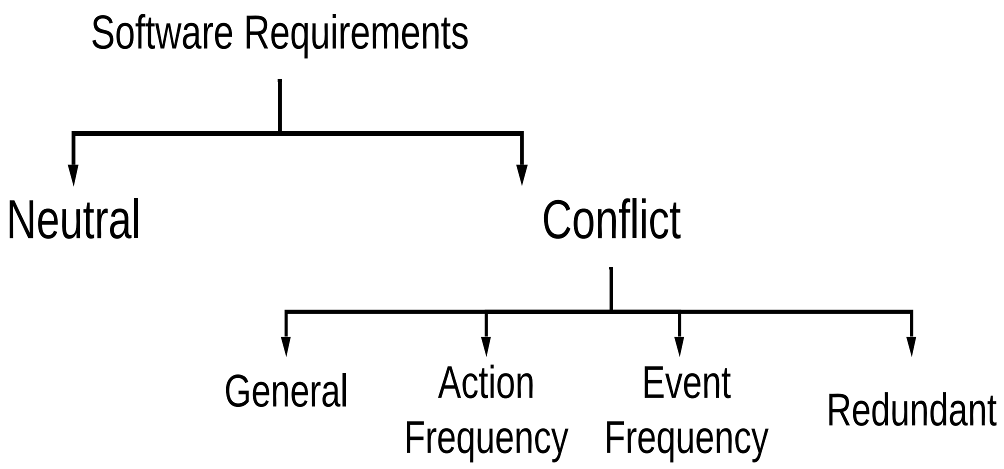


Figure 3.6: Rule-based defined Requirements Conflict type

1. General Conflict: If two requirements share the same "Action", "Agent", "Event", "Rules," "Action Negativity", and "Event Negativity," they are classified as having a general conflict with no specific conflict type.

2. Action Frequency Conflict: When two requirements match in "Action", "Agent", "Event", "Rules," and "Event Negativity" but differ in "Action Negativity", they are identified as exhibiting an "Action Frequency Conflict."

3. Event Frequency Conflict: Requirements sharing the same "Action", "Agent", and "Rules" but differing in "Event" and "Event Negativity" are considered to exhibit an "Event Frequency Conflict."

4. Redundant Requirements: Requirements sharing identical values for "Action", "Agent", "Event", "Rules", "Action Negativity", and "Event Negativity" are regarded as having a "Redundant Conflict."

Chapter 4

Implementation

4.1 Dataset:

As discussed in the previous chapter, data collection is a pivotal aspect of our research and necessitates the extraction of pertinent documents housing software functional requirements. This extraction process involves techniques such as web scraping and the availability of the selected data sources. It is imperative that the collected documents exhibit diversity across software projects, domains, and industries, ensuring the creation of a comprehensive and representative dataset.

In our research work, we have utilized two of the open-source available datasets to get software functional requirements, WorldVista and Pure as our test cases to assess the effectiveness of our semantic element identification algorithm.

Dataset is being collected from https://gitfront.io/r/user-9946871/ii6eJFSh7oT4/DA-Sentence-Pairs/

We'll give a brief overview of the datasets we used below:

1. **WorldVista:** The features and capabilities of an electronic health record system and a health information system are covered by the 117 requirements in the WorldVista dataset.
2. **Pure:** Our dataset consists of a total of 66 requirements, delineating the functionalities of a THEMAS system.

We initiated a process of requirement adjustment to ensure that all requirements meet the prerequisites for natural language (NL) requirements, facilitating our automated processing. This adjustment procedure involves the incorporation of modal verbs, the inclusion of conditional adverbial clause keywords such as "if" and "when," and the replacement of pronouns with their corresponding nouns. We are interested in quantifying the ratio of requirements necessitating these adjustments.

4.2 Data extraction & pre-processing:

4.2.1 PDF data extraction

The data set will be in pdf form and we will be extracting the data from the pdf for further processing. Below is the code of how we read the data from the pdf file.

```
def readPdf(pdf_path):
    sentences_with_page = []
    with pdfplumber.open(pdf_path) as pdf:
        for page_number, page in enumerate(pdf.pages, start=1):
            page_text = page.extract_text()
            sentences = re.split(r'(?<=[.!?])\s+(?=[A-Z][^.]*\.)', page_text)
            for sentence in sentences:
                sentences_with_page.append((page_number,sentence))
    return sentences_with_page
```

4.2.2 Text Splitting

The data that is extracted from pdf format is now split into sentences. This will separate the all pdf text into sentences for further processing.

```
def SentenceSplit(pdf_text):
    pdf_text = re.sub(r'\n+', ' ', pdf_text)
    pdf_text = re.sub(r'\x0c', ' ', pdf_text) # Page break character

    # Split sentences based on periods, exclamation marks, or question marks
        followed by spaces
    sentences = re.split(r'(?<=[.!?])\s+(?=[A-Z])', pdf_text)

    return sentences
```

4.2.3 Part-of-Speech Tagging

We will do POS tagging on the data in the final step after pre-processing the dataset. Through the process of POS tagging, each word in the text is assigned a part of speech (such as a noun, verb, or adjective). Understanding the grammatical structure of the demand and recognizing verbs, which frequently denote activities, are made easier with the use of POS tagging.

4.2.4 Dependency Parsing

A essential activity in natural language processing (NLP), dependency parsing examines the relationships between words in a phrase to assess its grammatical structure. To develop a dependency tree, it includes examining the syntactic connections between words in a sentence. It aids in comprehending the relationships between words in terms of their grammatical functions.

```
# Create a CoreNLP parser
parser = CoreNLPParser()
parsed = next(parser.raw_parse(requirement))
dependency_parser = CoreNLPDependencyParser()
dependency_tree = next(dependency_parser.raw_parse(requirement))
# Extract and format dependencies
dependencies = []
for triple in dependency_tree.triples():
    dep = {
        "type": triple[1],
        "start": triple[0][0],
        "end": triple[2][0]
    }
    dependencies.append(dep)
# return dependencies
return formatted_dependencies
```

This code demonstrates how to perform dependency parsing using the Stanford CoreNLP library and extract the grammatical relationships between words in a sentence. Dependency parsing is useful for various NLP tasks, including syntactic analysis, information extraction, and machine translation. Here we are using it for information extraction.

4.2.5 Tokenization

Tokenization is a crucial step in natural language processing (NLP), where a text is divided into individual words or tokens. Many NLP procedures, such as part-of-speech tagging, named entity recognition, and syntactic parsing, are built upon it.

```python
for sentence in parsed_response["sentences"]:
    for dep in sentence["basicDependencies"]:
        formatted_dep = {
            "dep": dep["dep"],
            "governor": dep["governor"],
            "governorGloss": sentence["tokens"][dep["governor"] -
                1]["word"],
            "dependent": dep["dependent"],
            "dependentGloss": sentence["tokens"][dep["dependent"] -
                1]["word"]
        }
        formatted_dependencies.append(formatted_dep)
    for token in sentence["tokens"]:
        formatted_token = {
            "index": token["index"],
            "word": token["word"],
            "originalText": token.get("originalText", ""),
            "lemma": token.get("lemma", "N/A"),
            "characterOffsetBegin": token.get("characterOffsetBegin", -1),
            "characterOffsetEnd": token.get("characterOffsetEnd", -1),
            "pos": token.get("pos", ""),
            "before": token.get("before", ""),
            "after": token.get("after", "")
        }
        formatted_tokens.append(formatted_token)
return formatted_tokens
```

This code demonstrates how to tokenize text using the Stanford CoreNLP library and extract the individual tokens for further NLP analysis or processing.

4.2.6 Text Normalization

Text normalization is an essential preprocessing step in NLP that aims to standardize and clean text data to make it more suitable for analysis and modeling. In the provided method, we have a simple text normalization function called normalize.

```python
def normalize(clause):
    index = len(clause) - 1
    while clause[index] in ['!', '?', ';', '\n', ' ', '.']:
        index -= 1
    return clause[: index + 1] + '.'
```

4.2.7 Text Preprocess

Text preprocessing is an important stage in natural language processing (NLP) that entails cleaning and transforming raw text input into a format appropriate for analysis and machine learning activities.

```python
def preprocess(text):
    text = remove_stopwords(text)
    text = remove_instances(text)
    text = change_ables(text)
    return find_restrictions(text)
```

The above code demonstrates a set of text pre-processing techniques commonly used in NLP. These techniques aim to remove noise, standardize certain phrases, and simplify the text, making it more amenable for downstream NLP tasks i.e. information retrieval.

4.3 Elements identification:

As defined in the above section we have four tuples/elements i.e. Actor, Event, Action, Action Negativity, Event Negativity, and Rules extracted from a text requirement based on semantics. So here we will be identifying each element separately:

4.3.1 Identifying the Action

The action identification from a text requirement refers to the process of understanding and extracting the specific actions that need to be performed or allowed based on the information provided in the requirement text.

Natural Language Processing (NLP) involves using computational techniques to analyze the text, identify the relevant actions, and possibly categorize or label them accordingly.

We have defined some rules/ conditions based on which the action will be extracted from the text requirements. We will pass all the required data as input to the method that will then process the further things based on the defined code. The following data will be passed to the action function as Input

1. clause (a single requirement entry)
2. checkNeg (check whether word before operation is negative or not)

This will involve rules that capture specific sentence structures or verb-object relationships that indicate actions.

Conditions or the rules to identify the actions are as follows:

1. Check for the presence of modal verbs (e.g., "shall," "must," "can," "may," "should," "will") in the text. If a modal verb is found, proceed with the following checks:
    (a) If checkNeg is False, meaning negations are not being checked:
        i. If "able to" is found after the modal verb, extract the words following "able to" until the end of the sentence or another preposition is encountered (e.g., "in," "on," "at"). Return these words as the operation and the preposition that follows "able to."
        ii. If "be" is found after the modal verb and "by" is also present in the text, extract the words following "be" until the end of the sentence or another preposition is encountered. Return these words as the operation and the preposition that follows "be."
        iii. If neither "able to" nor "be" is found, extract the words following the modal verb until the end of the sentence or another preposition is encountered. Return these words as the operation and the preposition that follows the modal verb.

```
def OperationParse(clause,checkNeg):
    tokens = clause.split()
    operation_tokens = []
    check = ["be", "have", "do"]
    stoppers = ["in", "on", "at", "with", "by", "for", "among",
        "if", "between", "from", ":", ",",";"]
    modals = ["shall", "must", "can", "may", "should", "will"]
    for modal in modals:
        if modal in tokens:
            index = tokens.index(modal)
            if "only" in tokens[index+1]:
                indexs=tokens.index("only")
                for token in tokens[indexs + 1:]:
                    if token in stoppers:
                        break
                    operation_tokens.append(token)
                return (" ".join(operation_tokens), tokens[indexs
                    + 1])
            if checkNeg == False:
                if "able" in tokens and "to" in tokens:
                    to_index = tokens.index("to")
                    for token in tokens[to_index + 1:]:
                        if token in stoppers:
                            break
                        operation_tokens.append(token)
                    return (" ".join(operation_tokens),
                        tokens[to_index + 1])
                elif "be" in tokens:
                    be_index = tokens.index("be")
                    if index + 2 < len(tokens) and "by" in tokens:
                        for token in tokens[index + 2:]:
                            if token in stoppers:
                                break
                            operation_tokens.append(token)
                        return (" ".join(operation_tokens),
                            tokens[index + 2])
                for token in tokens[index + 1:]:
```

```
                if token in stoppers:
                    break
                operation_tokens.append(token)
            return (" ".join(operation_tokens), tokens[index +
                1])
```

(b) If checkNeg is True, meaning negations are being checked:

i. Extract the words following the modal verb until the end of the sentence or another preposition is encountered. Return these words as the operation and the word two positions after the modal verb (to account for negations like "can't").

```
    elif checkNeg == True:
        for token in tokens[index + 2:]:
            if token in stoppers:
                break
            operation_tokens.append(token)
        return (" ".join(operation_tokens), tokens[index + 2])
```

2. If no modal verb is found in the text, return an empty string ("").

The extracted operation and preposition are returned as a tuple, making it easier to process the information.

4.3.2 Identifying the Actor

Using computational methods to examine and extract data from the text is necessary for NLP-based actor identification.

The following data will be passed to the function as Input data:

1. clause (a single requirement entry)
2. Operation (result returned from operation identification)
3. Dependencies (dependencies parsed in data extraction and pre-processing step)

We will pass all the required data as input to the method that will then process the further things based on the defined code. The steps involved in identifying or extracting the actor from text requirement will be:

1. **Passive Voice Identification:**

   (a) It first checks if the clause is in passive voice. If the dependency analysis indicates that the word "by" is present and is the last word in a dependency path, it identifies the actor.

   (b) It also considers articles ("a," "an," "the," etc.) that might precede the actor and includes them in the actor's identification if present.

```
def Agent_Identification(clause, operation, dependencies):
    agents = ''
    for dep in dependencies:
        if dep["type"] in ("dep", "case") and dep["end"] == "by":
            agents= dep["start"]
            if agents:
                article = clause.split(agents, 1)[0].strip().split()[-1]
                if article in ["a", "an", "the", "A", "An", "The", "This",
                    "this"]:
                    agents = f"{article} {agents}"
                elif article not in ["a", "an", "the", "A", "An", "The",
                    "This", "this"]:
                    agents = agents
            else:
                agents =''
```

2. **Active Voice Identification:**

   (a) If the clause is not in passive voice, it looks for a nominal subject (nsubj) dependency with the starting word being either the extracted operation or "able." This is an indicator of the actor in an active voice construction.

   (b) Like in the passive voice identification, it also considers articles preceding the actor.

```
elif dep["type"] == "nsubj" and dep["start"] == operation or dep["start"]
    == "able":
            agents = dep["end"]
            if agents:
                article = clause.split(agents, 1)[0].strip().split()[-1]
                if article in ["a", "an", "the", "A", "An", "The", "This",
                    "this"]:
                    agents = f"{article} {agents}"
                elif article not in ["a", "an", "the", "A", "An", "The",
                    "This", "this"]:
                    agents = agents
            else:
                agents = ''
```

3. **Fallback:**

   (a) If neither passive nor active voice constructions are identified, the code checks for the presence of modal verbs (e.g., "shall," "must," "can," etc.) in the clause.

   (b) If a modal verb is found, it identifies the word immediately preceding the modal verb as the actor.

```
if agents == '':
    for modal in modals:
        if modal in words:
            index = words.index(modal)
            agents = words[index - 1]
            if "by" in words:
                keyword = "by" if "by" in words else ""
                idx = words.index(keyword)
                agents = words[idx + 1]
            if agents:
                article = clause.split(agents, 1)[0].strip().split()[-1]
                if article in articles:
                    agents = f"{article} {agents}"
                elif article not in articles:
                    agents = agents
    return agents
```

4. **Article Handling:**

   (a) The code handles articles in a way that if an article is found before the identified actor, it is included in the actor's identification.

5. **Return the Identified actor:**

   The identified actor is returned as a string.

4.3.3 Identifying the Event

Using NLP, event identification from text requirements entails extracting and comprehending particular events, actions, or tasks based on textual descriptions or needs.

This process utilizes NLP techniques to analyze and parse the text, enabling the system to identify and categorize various events or actions that are mentioned.

The following data will be passed to the event identification function as Input

1. clause (a single requirement entry)
2. tokens (extracted in token parse)

We will pass all the required data as input to the method that will then process the further things based on the defined code. The steps involved in identifying or extracting the event from the text requirement will be:

1. **Tokenization and Condition Leading Words:**

   (a) The input tokens and clause are used to tokenize the clause into words.

   (b) The find_condition_words function is called to identify the positions (indexes) of condition-leading words (e.g., "when," "if," "while") within the tokens.

```
def find_condition_words(tokens):
    condition_leading_words = ['when', 'if', 'While']
    return find_in_tokens(tokens, condition_leading_words, start=1, end=0)
```

2. **Finding Punctuation After Condition Words:**

   (a) For each identified condition-leading word, the code searches for the positions of specific punctuations (, and .) that occur after the condition word.

(b) If punctuations are found, the start and end positions of the condition are determined based on the token indexes.

(c) The condition text is extracted by joining the words between the start and end positions in the clause_words.

```
def Event_Parse(tokens, clause):
    clause_words = clause.split()
    words_indexes = find_condition_words(tokens)
    conditions_list = []
    for words_index in words_indexes:
        start_pos = tokens[words_index]['index'] - 1
        punc_index = find_in_tokens(tokens, [',', '.'], words_index + 1,
            len(tokens))
        if len(punc_index) > 0:
            end_pos = tokens[punc_index[0]]['index']-1
            condition = ' '.join(clause_words[start_pos:end_pos])
            conditions_list.append(condition)
    return conditions_list if conditions_list else " "
```

3. **Event Conditions List:**

   Extracted conditions are added to the conditions_list.

4. **Return the extracted event:** The function returns the list of extracted event conditions. If no conditions are found, it returns "None."

4.3.4 Identifying the Rules

In NLP, determining the rules from text requirements is a crucial problem, especially in fields like software engineering, compliance, or legal document analysis. These limitations or regulations frequently list prerequisites, limitations, or instructions that must be followed.

The following data will be passed to the event identification function as Input

1. clause (a single requirement entry)
2. tokens (extracted in token parse)

3. dependency (extracted in dependency parse)

We will pass all the required data as input to the method that will then process the further things based on the defined code. The steps involved in identifying or extracting the rules from text requirements will be:

1. **Parsing Restrictions with parse_restriction:**

   (a) This function is the entry point for extracting restrictions.

   (b) It first calls RulesIdentification to identify restrictions based on dependency parsing.

   (c) It then extends the identified restrictions with those found using frequency_restriction.

   (d) If there's only one empty string in the restriction list, it is removed to ensure the result is clean.

   (e) Finally, it returns the list of restrictions.

```
def parse_rule(clause, dependencies, tokens):
    restriction = RulesIdentification(clause, dependencies, tokens)
    restriction.extend(frequency_restriction(clause, dependencies,
        tokens))
    if len(restriction) == 1 and restriction[0] == '':
        restriction = None
    return restriction if restriction else " "
```

2. **Finding Restrictions with find_restrictions:**

   (a) This function identifies restrictions or conditions based on specific keywords and phrases defined in the dictionary. These include words like "without," "within," "via," "as long as," and "until."

   (b) It extracts these phrases from the text and appends them to the restrictions list.

   (c) The extracted restrictions are removed from the original text to prevent double counting.

   (d) The modified text and the list of restrictions are returned.

```
def find_rules(text):
    restrictions = []
    dictionary = {
        'before condition': ['when', 'if'],
        'before instance': ['such as', 'for example', 'for instance',
            'including', 'etc'],
        'before restriction': ['without', 'within', 'via', 'as long
            as', 'until'],
        'stop words': ['then ', 'in turn ', '_', '"'],
        'modal verbs': ['must', 'shall', 'should', 'can', 'may'],
        'before adj clause': ['that', 'which', 'who', 'whom'],
        'able words': ['provide the means to ', 'provide means to ',
             'provide a way to ', 'provide the way to ','provide the
                 ability to ']
    }
    for w in ['without', 'within', 'via', 'as long as', 'until']:
        if (l_index := text.find(w)) >= 0:
            right = text.find(',', l_index)
            res = text[l_index: right].replace('.', '')
            if res != '':
                restrictions.append(res)
            text = text[:l_index] + text[right:]
    return text,restrictions
```

3. **Identifying Restrictions with RulesIdentification:**

   (a) This function identifies restrictions based on the grammatical dependencies between words in the text.

   (b) It looks for words that serve as adverbial modifiers (advmod) and appends them to the restriction list if they are not common adverbs like "when" or "then."

   (c) Additionally, it identifies restrictions related to time and appends them to the restriction list based on specific dependencies (nmod:tmod, nmod:per, nmod:npmod).

```
def RulesIdentification(clause, dependencies, tokens):
    restriction = []
    for dep in dependencies:
        if dep['dep'] == 'advmod' and dep['dependentGloss'].lower()
            not in ['when', 'then']:
            restriction.append(dep['dependentGloss'].lower())
    for dep in [dep for dep in dependencies if dep['dep'][:4] ==
        'nmod']:
        end_index = int(dep['dependent'])
        if tokens[end_index]['pos'] in ('NNP') or
            tokens[end_index]['lemma'] == 'time':
            for d in [d for d in dependencies if d['dep'] == 'case'
                and d['governor'] == end_index]:
                restriction.append(token2text(tokens, d['dependent'],
                    end_index + 1))
                break
    return restriction
```

4. **Identifying FrequencyBased Restrictions with frequency_restriction:**

   (a) This function identifies frequency-based restrictions, such as "every time," "everyday," or "N time1 per/a time2."

   (b) It looks for specific dependencies (nmod:tmod, nmod:per, nmod:npmod) and the use of "everyday" as a keyword.

   (c) The identified frequency-based restrictions are appended to the restriction list.

```
def frequency_restriction(clause, dependencies, tokens):
    restriction = []
    for dep in [dep for dep in dependencies if dep['dep'] ==
        'nmod:tmod']:
        end_index = dep['dependent']
        for d in dependencies:
            if d['dep'] == 'det' and d['governor'] == end_index and
                d['dependentGloss'] == 'every':
                restriction.append(token2text(tokens, d['dependent'],
                    end_index + 1))
                break
    for t in tokens:
        if t['lemma'] == 'everyday':
            restriction.append('everyday')
    for dep in [dep for dep in dependencies if dep['dep'] ==
        'nmod:per' or dep['dep'] == 'nmod:npmod']:
        time1 = dep['governor']
        time2 = dep['dependent']
        for d in [d for d in dependencies if d['dep'] == 'nummod' and
            d['governor'] == time1]:
            restriction.append(token2text(tokens, d['dependent'],
                time2 + 1))
            break
    return restriction
```

4.4 Element Identification Results:

After all the elements are identified in the above section, now we need to store all extracted results in an Excel file.

```
def Main(requirement, id, groupId):
    formatted_dependencies, formatted_tokens, dependencies =
        CoreNLPParse(requirement)
    restrictions = []
    requirement1 = normalize(requirement)
    requirement2, restrictions = preprocess(requirement1)
    operation_Negativity = check_not(requirement2,'op')
    operation_string, operation =
        OperationParse(requirement2,operation_Negativity)
    agent = Agent_Identification(requirement2, operation, dependencies)
    event = Event_Parse(formatted_tokens, requirement2)
    eventString=" ".join(event)
    event_Negativity = check_not(eventString,'eve')
    if restrictions == []:
        restriction = parse_rule(requirement2, formatted_dependencies,
            formatted_tokens)
    else:
        restriction = restrictions
    return [
        requirement,
        operation,
        operation_string,
        operation_Negativity,
        agent,
        eventString,
        event_Negativity,
        ", ".join(restriction)
    ]
file_path1 = "worldvista dataset.pdf"
file_path2 = "pureclean dataset.pdf"

saveDataIntoExcel(file_path1,"WorldVista")
saveDataIntoExcel(file_path2,"Pure")
```

The returned results from all the element identification methods will be passed to an Excel file via the Main function.

Here we have used, openpyxl library. It collects data from the processing of text requirements and saves it row by row in an Excel worksheet. In Table 4.1, our rules-based element detection approach achieved varying levels of accuracy for different types of elements.

Element	**WorldVista**	**Pure**
Actor	90%	90%
Action	100%	100%
Action Negativity	100%	100%
Event	98%	100%
Event Negativity	100%	100%
Restriction	85%	100%

Table 4.1: Element detection accuracy table showing the percentage of correctly detected elements by our Rules-based element detection approach from the text requirements.

4.5 Conflict Definition & Detection:

Through the incorporation of Natural Language Processing (NLP) methods, the field of software requirement conflict identification has made great strides in recent years. The goal of NLP, a branch of artificial intelligence (AI), is to make it possible for computers to comprehend, interpret, and produce text in human language. Utilizing NLP for requirement conflict detection introduces automation and scalability, improving the efficiency of processing huge and complicated information.

We have used two most advantageous features of NLP i.e. semantic analysis and textual similarity in our research work. Semantic Analysis allows the analysis of the requirement documents in a semantical context. By calculating similarity scores, it becomes possible to identify requirements that are closely related or potentially conflicting based on their wording and content.

Dataset	**Requirement1**	**Requirement2**	**Result**
WorldVista	The system shall allow physician offices to only use social security numbers to identify patients.	The system's pilot program shall use a smart card to digitally sign medication orders.	Neutral
Pure	The THEMAS system should record each event by a description of that event.	The themas system shall only process the current temperature.	Neutral
WorldVista	The system shall retain notifications for a predetermined amount of time (up to 30 days).	The system shall retain notifications for a predetermined amount of time (more than 30 days).	Conflict
Pure	When an alarm is requested for each time, an alarm event shall be recorded.	If an alarm is requested for the first time, an alarm event shall be recorded.	Conflict

Table 4.2: Sample pairs of texts taken from datasets. The relationship between Texts 1 and 2 is depicted in the 'Result' column.

Algorithm 1 Conflict Detection Algorithm (Part 1)

```
1: Input: Detected elements of Requirements $R$ saved in Excel file with semantical
   annotation, where Requirements are $r_1, r_2, \ldots, r_n$
2: Output: Result(result, conflict_type), ground truth value
3: for all Requirements $r \in R$ do
4:     for all pair of requirements $r(i, j)$ do
5:         if $i = j$ then
6:             skip $r(i, j)$ from $R$
7:         else
8:             $S \leftarrow$ CosineSimilarity($TF - IDF(i), TF - IDF(j)$)
9:             if $S \geq 40$ then            ▷ /* Assume that pair $r(i, j)$ is conflicting */
10:                if action or event or rules-inconsistency $r(i, j)$ found then
11:                    if action frequency conflict condition satisfied for $r(i, j)$ then
12:                        Return Result as Conflict, conflict_type as Action Frequency
13:                    else if event frequency conflict condition is satisfied for $r(i, j)$ then
14:                        Return Result as Conflict, conflict_type as Event Frequency
15:                    else if redundant requirement condition is satisfied for $r(i, j)$ then
16:                        Return Result as Conflict, conflict_type as Redundant
17:                    else
18:                        Return Result as Conflict, conflict_type as General
19:                    end if
20:                else if not action or event or rules-inconsistency $r(i, j)$ then
21:                    Return Result as Neutral, conflict_type as Nil
22:                end if
23:            else
24:                if any condition met for $r(i, j)$ then
25:                    Return Result as Conflict, conflict_type as met condition type
26:                else
27:                    Return Result as Neutral, conflict_type as Nil
28:                end if
29:            end if
30:        end if
31:    end for
32: end for
```

Algorithm 2 Conflict Detection Algorithm (Part 2)

```
1: /* Check for the ground truth values for precision, recall, and f1-score */
2: Match the detected result with the known conflict list known_Conflict[]
3: if pair $r(i,j)$ is conflict and matched with list item then
4:     True Positive
5: end if
6: if pair $r(i,j)$ is conflict and not matched with list item then
7:     False Positive
8: end if
9: if pair $r(i,j)$ is Neutral and is not in list items then
10:     True Negative
11: end if
12: if pair $r(i,j)$ is Neutral and is matched with any list items then
13:     False Negative
14: end if
15: Return Result(result, conflict_type)
```

The "Conflict Detection Algorithm" described in Algorithm 1 and 2 serves the critical function of identifying conflicts and evaluating their nature within a set of detected requirements. Here's an overview of its key functionalities:

1. Input: The algorithm takes as input a collection of requirements denoted as R, which are stored in an Excel file with semantical annotation. These requirements are labeled as $r_1, r_2, \ldots, r_n$. The algorithm starts by preprocessing this input data.

2. Conflict Detection: The primary function of the algorithm is to detect conflicts within the set of requirements. It systematically compares pairs of requirements $r(i,j)$ to determine whether they exhibit conflict or not. To optimize efficiency, it avoids comparing a requirement with itself ($i = j$) since such comparisons are inherently non-conflicting.

3. Similarity Calculation: The algorithm determines the cosine similarity (S) between the ith and jth requirements' TF-IDF (Term Frequency-Inverse Document Frequency) representations. This similarity score is a crucial indicator of probable

disputes. The program then analyzes the nature of the conflict if the similarity score is more than or equal to a predetermined threshold (in this case, 40).

4. Conflict Types: The algorithm categorizes conflicts into three distinct types:
   (a) Action-Inconsistency: This type of conflict pertains to inconsistencies related to actions specified in the requirements.
   (b) Event-Inconsistency: It deals with conflicts arising from inconsistencies related to events described in the requirements.
   (c) Rules-Inconsistency: Conflicts related to rule specifications are classified under this category.
5. Conflict Conditions: For each conflict type, the algorithm evaluates specific conditions to determine the precise nature of the conflict. These conditions include action frequency conflict, event frequency conflict, and redundant requirement conditions. If these conditions are met, the algorithm categorizes the conflict type accordingly. If none of the specific conditions apply, the conflict is categorized as "General."
6. Neutral Pairs: If the similarity score falls below the threshold, the algorithm assumes that the pair of requirements ($r(i, j)$) is neutral. It then checks for any other conflict conditions that might apply. If such conditions are met, it categorizes the conflict as "Neutral" with the corresponding conflict type.
7. Performance Evaluation: After processing all requirements and identifying conflicts, the algorithm assesses its performance against ground truth values. It matches the detected conflicts with a known conflict list called "known_Conflict." This evaluation helps determine the algorithm's precision, recall, and F1score, providing insights into the accuracy of conflict detection.
8. Output: The algorithm produces the final result in the form of "Result(result, conflict_type)" and provides the ground truth value. The result indicates whether conflicts were detected and their specific conflict type.

In summary, the Conflict Detection Algorithm plays a crucial role in analyzing and categorizing conflicts within semantically annotated requirements. It leverages cosine similarity and specific conflict conditions to classify conflicts accurately. Additionally, it evaluates its performance against known ground truth values, offering valuable insights into the quality of conflict detection. This functionality is essential in various domains, such as software engineering and natural language processing, where conflict resolution and requirement analysis are critical tasks.

The below Conflict Detection Algorithm analyzes requirements for conflicts, categorizes them, and evaluates their performance against ground truth values, providing insights into the accuracy of conflict detection in semantically annotated requirement sets.

We manually reviewed each of the discovered findings for our WorldVista and Pure databases, are assessed and noted whether or not they were correct. Then also implemented conditions in our code to check for the detected conflict accuracy as True Positive or False Positive based on our provided known conflict list.

Chapter 5

Results & Analysis:

Overall, the two datasets produced by our work performed extremely well in terms of precision and recall. The precision, recall, and f1-score for the requirement datasets (Pure and WorldVista) are 84% and 100%, 94% and 95%, and 89% and 97%, respectively.

The below table 5.1 offers insights into the precision and recall metrics, providing a nuanced understanding of how well our work identifies and classifies conflicts within the two datasets, Pure and WorldVista.

	WorldVista	**Pure**
Total requirements (#)	117	66
Known conflicts (#)	39	21
Detected conflicts (#)	44	21
Correct detected conflicts (#)	37	20
Incorrect detected conflicts (#)	7	1
Precision (%)	84%	100%
Recall (%)	94%	95%
F1 Score (%)	89%	97%

Table 5.1: Conflict detection result table

In the WorldVista dataset, the system identified 44 potential conflicts. From the detected conflicts 37 were correct meaning True Positive and only 7 were not correct meaning False Positive where non-conflict elements were incorrectly classified as conflicts. This resulted in a precision of 84% and recall of 94%. This indicates that most of the potential conflicts identified were indeed real conflicts in the WorldVista dataset.

Ground Truth	**Predicted Class**	
	Positive (P)	**Negative (N)**
Positive (P)	37	7
Negative (N)	0	2

Table 5.2: True Positive and False Negative Table for WorldVista Dataset

The above evaluation demonstrates the system's ability to accurately pinpoint actual conflicts within the dataset. However, it is worth noting that there were 7 instances where the system incorrectly classified non-conflict elements as conflicts, constituting false positives.

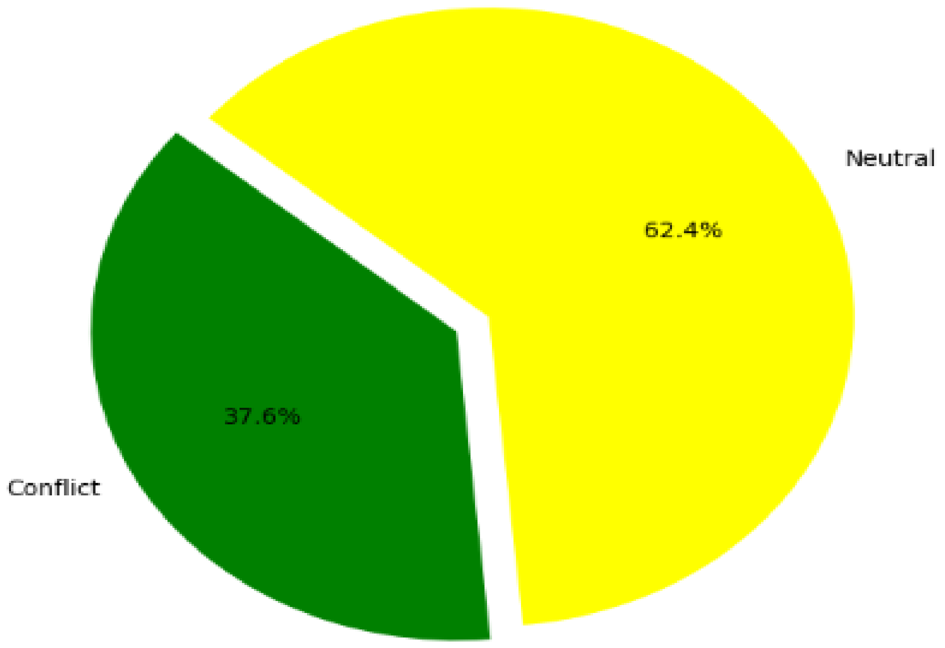


Figure 5.1: Ratio of Conflicted and Neutral Requirements in WorldVista dataset shown by piechart

This performance translates into a precision of 84% and the f1-score of 89% of the potential conflicts identified by the system were genuine conflicts in the WorldVista dataset. The high precision indicates that the system exercises caution in labeling elements as conflicts, reducing the likelihood of false alarms.

In the Pure dataset, the conflict detection system identified a total of 21 potential conflicts. Out of these, 20 were true positives, indicating that the system correctly detected 100% of the actual conflicts in this dataset. However, there was only 1 false positive, where non-conflict elements were incorrectly classified as conflicts.

Ground Truth	**Predicted Class**	
	Positive (P)	**Negative (N)**
Positive (P)	20	0
Negative (N)	0	1

Table 5.3: True Positive and False Negative Table for Pure Dataset

The Pure dataset showcases an even more remarkable performance by the conflict detection system. Out of the 21 potential conflicts identified, an impressive 20 were true positives, signifying that the system achieved a precision rate of 100% and the f1-score of 97%. In other words, every potential conflict identified in the Pure dataset was indeed a genuine conflict.

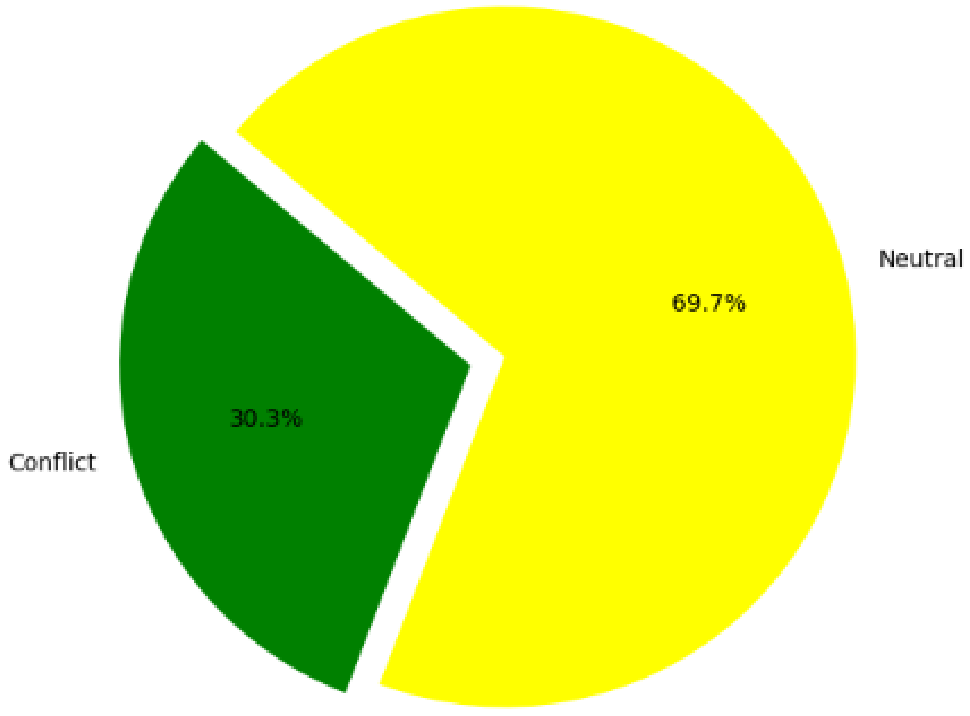


Figure 5.2: Ratio of Conflicted and Neutral Requirements in Pure dataset shown by piechart

Furthermore, there was only one instance of a false positive, where the system incorrectly classified a non-conflict element as a conflict. This exceptional precision highlights the system's high degree of accuracy in differentiating between conflicts and non-conflict elements within the Pure dataset.

We performed an extensive examination of the precise sorts of conflicts found in the WorldVista and Pure datasets to acquire a more thorough knowledge of the performance of our conflict detection method. As shown in the below table 5.4, this nuanced evaluation allows us to discern which categories of conflicts were successfully identified and where potential improvements or refinements may be needed.

Datasets	**Conflict Types (Correctly Detected)**			
	General	**Action Frequency**	**Event Frequency**	**Redundancy**
WorldVista	35	1	1	0
Pure	19	0	1	0

Table 5.4: The effectiveness of our research in identifying conflict types in WorldVista and Pure requirements

In the WorldVista dataset, our conflict detection system correctly identified 35 conflicts of the "General" type, 1 conflict related to "Action Frequency," and 1 conflict associated with "Event Frequency." Remarkably, there were no instances where the system incorrectly labeled conflicts of the "Redundancy" type.

This performance breakdown highlights the system's proficiency in detecting various types of conflicts within the WorldVista dataset. The majority of detected conflicts fall into the "General" category, demonstrating the system's effectiveness in identifying conflicts of a diverse nature. The detection of action and event frequency conflicts, albeit in smaller numbers, indicates the system's ability to handle specific conflict sub-types as well.

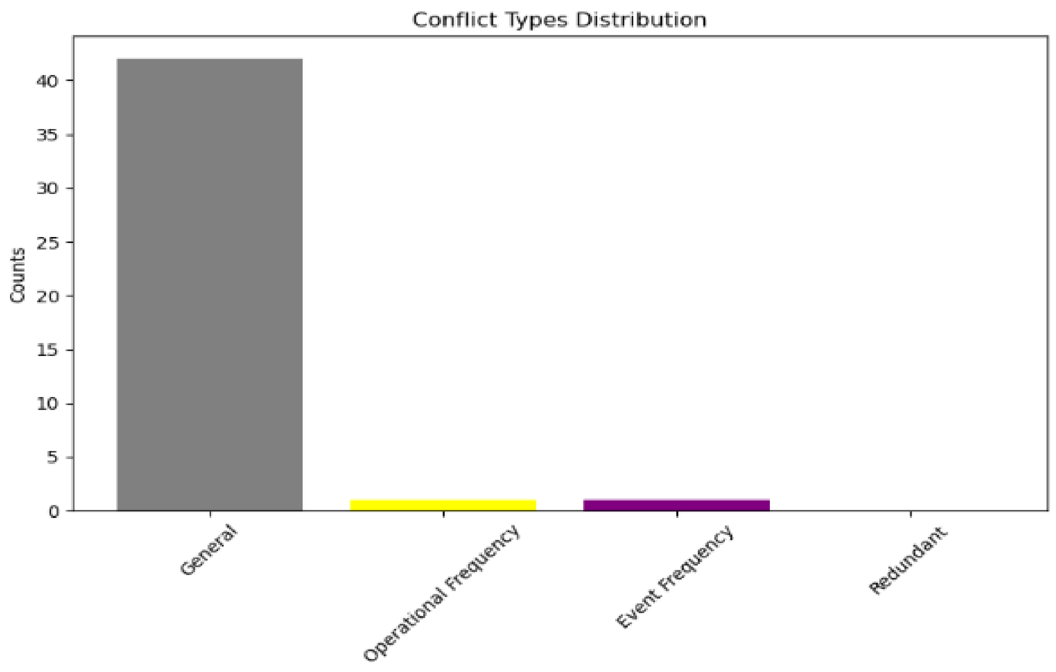


Figure 5.3: Conflict type distribution for WorldVista dataset shown in bar chart

In the Pure dataset, the conflict detection system successfully identified 19 conflicts of the "General" type and 1 conflict related to "Event Frequency." Similar to the WorldVista dataset, there was no instance incorrectly marked as "Redundancy."

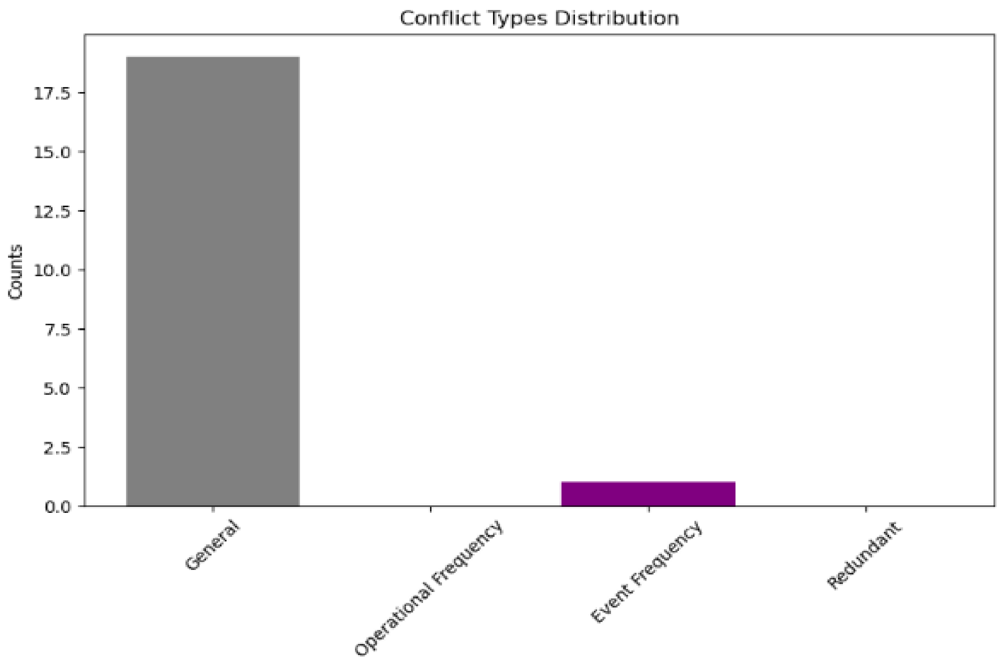


Figure 5.4: Conflict type distribution for Pure dataset shown in bar chart

This outcome in the Pure dataset further underscores the system's competence in detecting conflicts of the "General" type, which often encompasses a wide range of conflicts. Additionally, the identification of an "Event Frequency" conflict is indicative of the system's ability to pinpoint specific conflict sub-types when they occur.

The performance analysis of conflict type detection in both the WorldVista and Pure datasets reaffirms the effectiveness of our conflict detection system in identifying conflicts of various categories. The system's high accuracy in detecting "General" conflicts, which tend to be the most prevalent and encompassing category, is particularly noteworthy. Furthermore, the detection of specific sub-types such as "Operational Frequency" and "Event Frequency" conflicts demonstrates the system's versatility in addressing nuanced conflicts within requirements.

Chapter 6

Conclusion

6.1 Discussion

The conflict detection results presented in Table 5.1 provide valuable insights into the performance of conflict detection mechanisms for two different systems, WorldVista and Pure. These results demonstrate that the Pure system achieved a perfect precision rate of 100%, indicating that all the conflicts it detected were indeed genuine conflicts. However, it's important to note that Pure detected fewer conflicts overall, which may suggest that it may have missed some conflicts that WorldVista was able to identify.

On the other hand, WorldVista achieved a respectable precision rate of 84%, indicating that the majority of conflicts it detected were accurate. However, it also had a higher rate of incorrect detection compared to Pure. This suggests that while WorldVista may have identified more conflicts, it also introduced a higher risk of false alarms, which could lead to unnecessary disruptions or interventions.

In terms of recall and F1 score, both systems performed fairly well, with WorldVista showing a slightly higher recall rate of 94% compared to Pure's 95%. The F1 score, which balances precision and recall, was 89% for WorldVista and 97% for Pure. These scores indicate that both systems strike a reasonable balance between identifying conflicts and minimizing false alarms, with Pure having a slight edge in overall performance.

Building upon the discussion of precision, recall, and F1 score, it is crucial to delve deeper into the implications of these results for real-world applications. The perfect precision rate achieved by the Pure system is undoubtedly an impressive feat, as it ensures that every identified conflict warrants attention. However, the trade-off in this case is the possibility of missed conflicts. In contexts where the cost of missing a conflict is exceptionally high, such as in safety-critical systems, Pure's approach may prove to be the preferred choice. Nevertheless, it is essential to consider the potential consequences of false negatives, which may outweigh the benefits of a perfect precision rate in certain scenarios.

On the other hand, the WorldVista system's higher rate of false alarms implies a greater need for human intervention to validate and resolve conflicts. While this may increase the operational workload, it could also serve as a safety net, ensuring that conflicts are not overlooked. This characteristic may be more suitable for applications where a high level of caution is necessary, even if it comes at the cost of occasional false alarms. The choice between these systems ultimately depends on the specific requirements and risk tolerance of the application.

Furthermore, it is worth exploring the factors that may have contributed to the differences in performance between the two systems. One possible explanation for Pure's superior precision could be its reliance on a more conservative conflict detection algorithm, which results in fewer false alarms. Conversely, WorldVista may employ a more sensitive approach, detecting a broader range of potential conflicts but at the expense of higher false positives. Investigating the specific algorithms and parameters used by both systems could provide valuable insights into their respective strengths and weaknesses.

Additionally, the dataset used for this evaluation warrants consideration. The composition of the dataset, including the types of conflicts and their frequency, could have influenced the results. Future research may involve experimenting with different datasets to assess how the systems' performance varies under various conditions. Moreover, it would be insightful to explore the impact of varying thresholds for conflict detection on precision, recall, and the F1 score for both systems, as this could provide a means of optimizing their performance for specific applications.

In conclusion, the conflict detection results presented in this study shed light on the trade-offs between precision, recall, and false positives in conflict detection mechanisms. The choice between a system with perfect precision and one with a higher recall rate, but more false alarms, ultimately hinges on the specific requirements and risk tolerance of the application. Further investigation into the algorithms, parameters, and datasets used in these systems, as well as the implications of varying detection thresholds, could pave the way for more tailored and effective conflict detection strategies in the future.

The results presented in Table 5.1 represent a significant step forward in harnessing NLP techniques for conflict detection in software functional requirements. By achieving a perfect precision rate of 100% with the Pure system, our research demonstrates the potential of NLP-driven approaches to minimize the risk of false alarms in conflict detection. This finding has profound implications for industries where false alarms can lead to costly disruptions or where safety and reliability are paramount.

However, it is vital to recognize that the high precision rate of the Pure system comes at the cost of potentially missing some conflicts. This trade-off raises important questions about how to strike the right balance between precision and recall, especially when dealing with complex and evolving software systems. our thesis not only provides valuable insights into this balance but also offers a roadmap for future research in optimizing NLP-driven conflict detection systems.

Furthermore, our thesis contributes to the growing body of literature on the application of NLP techniques to software engineering. The successful implementation of NLP for conflict detection underscores the versatility of NLP in addressing various challenges in the software development life cycle. It highlights the potential for NLP to assist in requirements analysis, quality assurance, and even automated code generation by understanding and processing natural language specifications.

Moreover, our thesis can inspire further exploration into the interpretability of NLP-driven conflict detection models. Understanding how these models arrive at their decisions can be critical in gaining trust from stakeholders and making informed decisions in software development projects. Discussing interpretability techniques or potential future research directions in this area would be valuable.

6.2 Future Work

Investigating the ability of models to generalize across different domains or datasets can be beneficial. Transfer learning and domain adaptation techniques can be explored to leverage knowledge learned from one domain and apply it effectively to another, potentially reducing the need for large amounts of domain-specific data.

In our current research, we have concentrated on two specific datasets, and our work has demonstrated commendable performance for these particular instances of requirement conflicts. However, to further enhance the robustness and generalizability of our findings, future investigations could expand the scope by incorporating a more extensive range of datasets. These datasets could encompass diverse domains, industries, and application areas, providing a comprehensive view of requirement conflicts in various contexts. By doing so, we can gain deeper insights into the commonalities and distinctions in conflict patterns across different domains, enabling the development of more adaptable and universally applicable conflict resolution techniques. Additionally, exploring different perspectives of requirement conflicts, such as temporal changes, evolving stakeholder preferences, and dynamic project environments, could offer valuable insights into the adaptability and responsiveness of conflict resolution approaches. This multifaceted approach will not only strengthen the practical relevance of our research but also contribute to a more comprehensive understanding of the challenges associated with requirements engineering in complex systems.

Incorporating user feedback and preferences into the work can lead to more user-friendly and effective results.

Chapter 7

Bibliography

[1] B. H. C. Cheng, J. M. Atlee, and T. Chau, "Research directions in requirements engineering," in *International Conference on Software Engineering, 2009. ICSE 2009*, IEEE, 2009.

[2] S. Biffl, J. Kroll, and D. Winkler, "A machine learning approach for detecting conflicts in functional requirements," *Journal of Systems and Software*, vol. 95, pp. 66–80, 2014.

[3] P. Clements and L. Northrop, *Software architecture: perspectives on an emerging discipline.* Prentice Hall, 2001.

[4] R. Wieringa, *Requirements engineering: framework for understanding.* Wiley Publishing, 2014.

[5] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 2nd. Rocky Nook, 2015.

[6] T. Moser, D. Winkler, H. Matthias, and S. Biß, "Requirements management with semantic technology: An empirical study on automated requirements categorization and conflict analysis," in *International Conference on Advanced Information Systems Engineering, 2011*, 2011.

[7] A. Moitra, K. Siu, A. Crapo, *et al.*, "Towards development of complete and conflict-free requirements," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, 2018.

[8] A. Almazyad and S. Ambreen, "Automated conflict detection and resolution in requirements engineering using machine learning," 2021.

[9] W. Guo, L. Zhang, and X. Lian, “Automatically detecting the conflicts between software requirements based on finer semantic analysis,” 2021.

[10] G. S. Walia and J. C. Carver, “A systematic literature review to identify and classify software requirement errors,” *Inform. Softw. Technol.*, vol. 51, pp. 1087–1109, 2009.

[11] M. Jarke, M. Gebhardt, S. Jacobs, and H. W. Nissen, “Conflict analysis across heterogeneous viewpoints: Formalization and visualization,” in *I14 1996*, 1996.

[12] D. Yang, Q. Wang, M. Li, Y. Yang, K. Ye, and J. Du, “A survey on software cost estimation in the chinese software industry,” in *Proceedings of the Second ACM-I14 International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, ACM, 2008, pp. 253–262.

[13] M. Urbieta, M. J. Escalona Cuaresma, E. Robles Luna, and G. Rossi, “Detecting conflicts and inconsistencies in web application requirements,” in *Current Trends in Web Engineering - Workshops, Doctoral Symposium, and Tutorials, Held at ICWE 2011*, 2011.

[14] M. Kim, S. Park, V. Sugumaran, and H. Yang, “Managing requirements conflicts in software product lines: A goal and scenario-based approach,” *Data Knowledge Engineering*, vol. 61, no. 3, pp. 417–432, 2007.

[15] S. A. Saboor and H. M. Sajjad, “Conflict detection in natural language requirements using machine learning and graph analysis,” 2019.

[16] M. M. Kabir and M. A. Azim, “A comparative study of machine learning algorithms for software requirement conflict detection,” 2019.

[17] Y. Hu and X. Wang, “An approach to software requirement conflict detection based on machine learning,” 2020.

[18] P. Mader and R. Koschke, “Using machine learning to detect conflicts in software requirements,” 2018.

[19] W. et al., “A deep learning-based approach for conflict detection in software requirements,” 2018.

[20] A. et al., “Automated requirements conflict detection using natural language processing and machine learning,” 2020.

[21] O. et al., "Requirements conflict detection using machine learning techniques: A systematic literature review," 2020.

[22] A. et al., "Detecting conflicts in requirements using natural language processing," 2018.

[23] K. Pohl, *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[24] M. Aldekhail, A. Chikh, and D. Ziani, "Software requirements conflict identification: Review and recommendations," *IJACSA*, vol. 7, no. 10, pp. 336–345, 2017.

[25] D. Zowghi, D. Mairiza, and N. Nurmuliani, "Managing conflicts among nonfunctional requirements," 2009.

[26] G. Deshpande, "Sreyantra: Automated software requirement inter-dependencies elicitation, analysis, and learning," in *2019 I14/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019.

[27] M. Aldekhail and D. Ziani, "Intelligent method for software requirement conflicts identification and removal: Proposed framework and analysis," 2017.

[28] "Conflicting requirements." (), [Online]. Available: http://c2.com/cgi/wiki?ConflictingRequirements.

[29] "How do you manage conflicting stakeholder demands?" (), [Online]. Available: http://pm.stackexchange.com/questions/1399/how-do-you-manage-conflicting-stakeholder-demands.

[30] K. Taveter, L. Sterling, S. Pedell, R. Burrows, and E. Taveter, "A method for eliciting and representing emotional requirements: Two case studies in ehealthcare," in *2019 I14 27th International Requirements Engineering Conference Workshops (REW)*, 2019, pp. 100–105.

[31] F. Halim and D. Siahaan, "Detecting non-atomic requirements in software requirements specifications using classification methods," in *2019 1st International Conference on Cybernetics and Intelligent System (ICORIS)*, 2019.

[32] A. Abu-Mahfouz and H. Al-Aqrabi, "A conflict detection framework for software requirements using machine learning," 2019.

[33] S. Javed, S. Ali, and S. Ali, "An ensemble machine learning approach for conflict detection in software requirements," 2020.

[34] D. Tan and L. My, “An approach for detecting conflicts in functional requirements using machine learning techniques,” 2018.

[35] S. Ali, S. Javed, and S. Ali, “Using machine learning techniques for detecting conflicts in software requirements,” 2019.

[36] F. Calefato, F. Lanubile, and N. Novielli, “Detecting conflicts in requirements: An approach based on machine learning techniques,” 2019.

[37] S. T. e. a. Gao, “A machine learning approach for detecting conflicts in functional requirements,” 2017.

[38] S. Kamal and M. R. Khan, “Detecting conflicts in software requirements using machine learning techniques,” 2018.

[39] S. e. a. Kumar, “A hybrid machine learning approach for conflict detection in software requirements,” 2019.

[40] M. H. e. a. Khan, “A machine learning-based approach for conflict detection in functional requirements,” 2020.

[41] C. et al., “A hybrid machine learning approach for detecting conflicts in requirements documents,” 2019.

[42] A.-H. et al., “Conflict detection in requirements documents using machine learning techniques,” 2019.

[43] S. et al., “Conflict detection in requirements specifications using nlp techniques,” 2020.

[44] C. et al., “A machine learning approach to detecting conflicts in requirements documents,” 2020.

[45] R. Rosas and A. Garcia-Sanchez, “Using machine learning techniques to detect conflicts in software requirements,” 2021.

[46] M. A. e. a. Khalid, “Detecting and resolving conflicts in software requirements using deep learning,” 2021.

[47] S. R. S. e. a. Fernandes, “Conflict detection in software requirements using machine learning techniques: A systematic literature review,” 2020.

[48] P. et al., “Detecting conflicts in software requirements using natural language processing and machine learning,” 2020.