Leveraging NLP Techniques for Ambiguity Analysis: The AmbiTRUS Tool

THIS PAGE LEFT BLANK INTENTIONALLY

KEYWORDS

Agile Software Engineering

Ambiguity

Requirements Engineering

Requirements Quality

Software Engineering

User Stories

CONTENTS

THIS PAGE LEFT BLANK INTENTIONALLY

1. INTRODUCTION

This report details the technical aspects in the development of AmbiTRUS (Ambiguity Tracking and Resolution for User Stories) tool, specifically tailored to implement the AmbiTRUS framework. The AmbiTRUS tool employs Natural Language Processing (NLP) techniques for ambiguity analysis within a set of user stories, aligning with quality criteria established within the AmbiTRUS framework ("Technical Report: The AmbiTRUS Ambiguity Analysis Framework" 2024). The primary objective of this report is to offer comprehensive technical insights, focusing on the NLP techniques underpinning each quality criterion, and facilitating a clear understanding of the AmbiTRUS tool functionality.

Industries have recognized the value of translating quality guidelines into a Computer-Aided Software Engineering (CASE) tool. This not only enhances software development efficiency and resource management but also ensures system robustness and reliability (Alzayed and Al-Hunaiyyan 2021; Koh and Chua 2023; Luisa et al. 2004; Osama et al. 2020; Sonbol et al. 2022; Zhao et al. 2021). However, the study presented in (Fernández et al. 2017) revealed that only 16% of companies in Europe and the US utilized automated techniques for requirements analysis, despite the known inefficiencies and error-proneness of manual analysis processes (Dalpiaz et al. 2018; Koh and Chua 2023; Riaz et al. 2019).

On the other hand, the adoption of NLP-based techniques has proven effective in enhancing problem understanding and user expectations. These techniques analyze textual requirements documents to pinpoint linguistic issues (Ezzini et al. 2022; Fantechi et al. 2023), classify functional aspects of the requirements (Dalpiaz, Dell'Anna, et al. 2019; Kocerka et al. 2022), and identify similarities between them (Alhoshan et al. 2022; Dalpiaz, van der Schalk, et al. 2019).

In this context, the AmbiTRUS tool has been developed to address these challenges. By leveraging NLP techniques, the tool efficiently detects and resolves potential ambiguity in user stories by validating against the AmbiTRUS quality criteria. Recognizing the subjective nature of ambiguity, the tool offers semi-automatic analysis, empowering users to select relevant criteria and validate results before implementation.

Moreover, the tool incorporates a machine-learning-based recommendation system and an automatic calculation feature to enhance the effectiveness of ambiguity analysis. These features not only enhance the tool's performance but also facilitate an examination of the alignment between tool-generated recommendations and user opinions. Consequently, a thorough evaluation of how well the tool meets user expectations.

This report will discuss the methodology used in the development of the AmbiTRUS tool, NLP techniques that support the validation of the quality criteria, and the practical implementation of the AmbiTRUS in analyzing ambiguity in user stories.

2. SOFTWARE DEVELOPMENT METHODOLOGY

The tool development started with the careful selection of relevant NLP techniques. The selection process involved a thorough review of the literature on identifying user story ambiguity using NLP. The outcome of this review was translated into a set of tool functionalities that was then implemented within a Jupyter Notebook. These functionalities were tested using an open-source dataset sourced from **https://data.mendeley.com/datasets/7zbk8zsd8y/1, encompassing** over 1000 user stories from 23 domains (Dalpiaz, van der Schalk, et al. 2019). Based on the test results, the tool functionalities underwent iterative review to ensure alignment with the AmbiTRUS quality criteria.

Next, to enhance usability and encourage user adoption, we transformed the previously defined Jupyter Notebook tool functionalities into a more user-friendly web application. A web-based tool that facilitates a more comfortable analysis of user stories was developed utilizing the Django framework. Additionally, the tool functionality was expanded by incorporating a user feedback mechanism. Users were provided with the opportunity to express their opinions on the recommendations provided by the tool for rewriting user stories identified as potentially ambiguous. To assist users in this process, a glossary was integrated to assist users in the selection of standard terminology for rewriting user stories. This iterative approach aimed to ensure that the tool is not only effective but also user-friendly, increasing user satisfaction and engagement.

The development of the tool adopted a Design Science approach (Hevner et al. 2004), with iterative improvements based on feedback obtained during the empirical usability testing conducted between September and November 2023 (Figure 1).

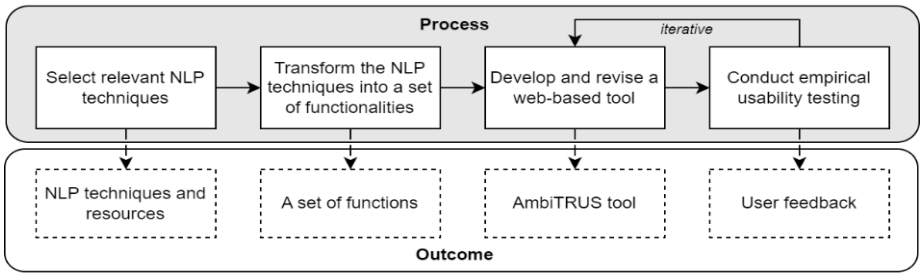

Figure 1. The development process of AmbiTRUS tool-based method

3. SOFTWARE REQUIREMENTS AND SPECIFICATIONS

AmbiTRUS tool is designed with a primary focus on analyzing ambiguity in a set of user stories. The tool's development is driven by the following objectives:

- Analyzing ambiguity in a set of user stories
- Refining user stories that are flagged as "potentially ambiguous"
- Evaluating level of agreement with the results generated by the tool
- Customizing quality criteria that users wish to analyze

These objectives have been broken down into software requirements, which further have been broken down into functional and non-functional requirements.

3.1 Functional Requirements

- The AmbiTRUS must capable of performing a thorough examination of ambiguity presents in user stories in accordance with the AmbiTRUS framework quality criteria.
- The AmbiTRUS must capable of providing recommendations for the purpose of user stories refinement for those that are identified as "potentially ambiguous".

3.2 Non-functional Requirements

- The AmbiTRUS tool should enable users to assess their level of agreement with the outcomes generated by the tool to evaluate the analysis and refinement results.
- The AmbiTRUS tool should provide flexibility and customization to select quality criteria that users wish to analyze.

4. SOFTWARE DESIGN

4.1 ARCHITECTURE AND TECHNOLOGY OF AMBITRUS TOOL

The AmbiTRUS tool has two versions. Version 1.0 focuses on identifying ambiguity issues in the *WHO* and *WHAT* segments while excluding the *WHY* segment. This version analyzes potential ambiguity and provides results based on the entire set of quality criteria outlined in the AmbiTRUS framework. Version 1.1, on the other hand, includes the *WHY* segment, recognizing that syntactic ambiguity in this segment could affect understanding (Wautelet et al. 2017). Version 1.1 also streamlines the analysis process and reduces the time users spend reviewing results by using a hierarchical method. This new approach decreases the notification of "potential ambiguity" in user stories from multiple notifications to a single notification, as it requires user stories to meet the earlier criteria before moving to the next level of analysis (Figure 2).

Regardless of the versions, the tool operates through four main stages: preprocessor, analyzer, enhancer, and report generator (Figure 3). The **preprocessor** prepares user stories for analysis by splitting them into *WHO*, *WHAT*, and *WHY* segments.

The **analyzer** initiates the process by allowing users to choose the quality criteria they wish to assess. Once the criteria are selected, the analyzer employs a variety of NLP tools and

techniques associated with the chosen criteria. This process identifies potential ambiguities in user stories and generates results accordingly.

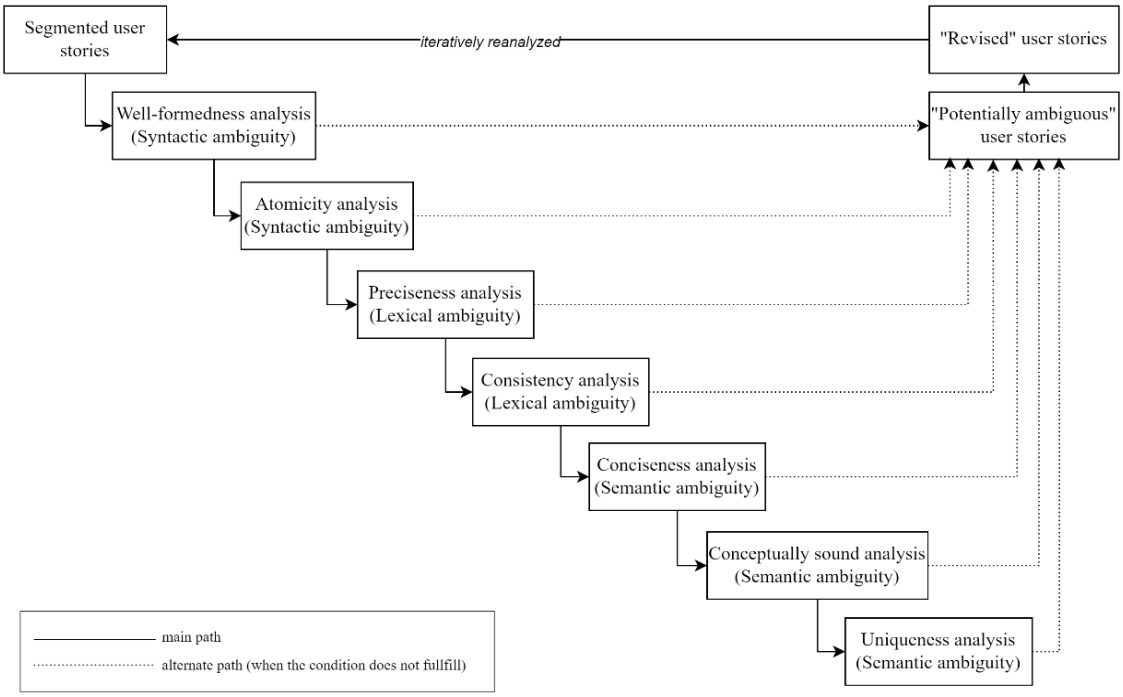

Figure 2. Hierarchical analysis procedure that is used in the AmbiTRUS tool Version 1.1.

The **enhancer** uses the analyzer results to aid users in enhancing user stories by leveraging recommendations provided by the tool to rephrase the user stories flagged as “potentially ambiguous”. The enhancer helps users improve their flagged user stories by providing recommendations for rephrasing user stories which are differently designed based on the specific criteria violated. The detailed improvement recommendations are presented in

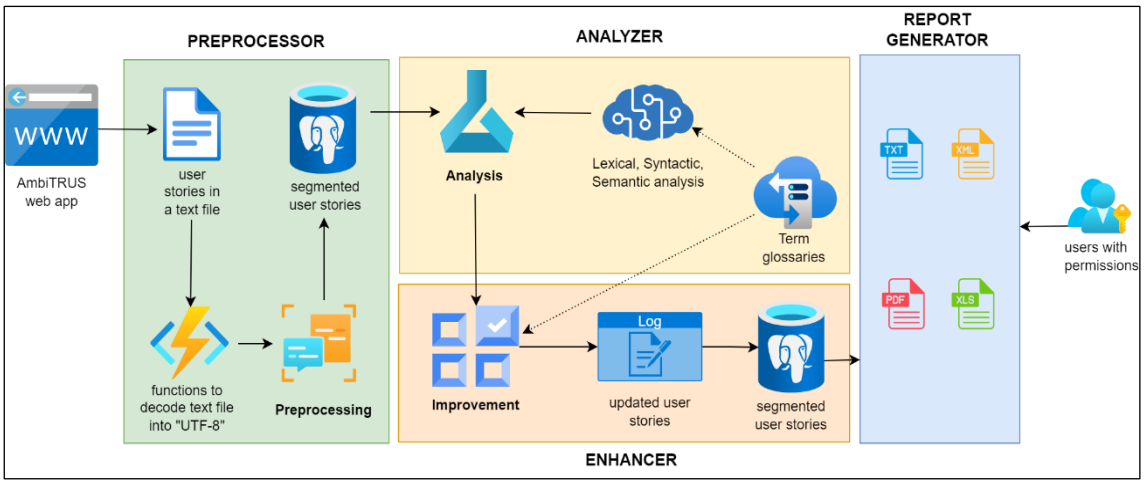

Figure 3. Components of the NLP-based tool-supported method tool: preprocessor, analyzer, enhancer, and report generator

Table 1. Finally, the **report generator** allows users to download analysis results and improvements in different file formats, including .PDF, .XLS, .CSV, and .TXT.

Table 1. The expected improvement actions

Quality criteria	Action of improvement
Well-formed analysis	Rewrite user story using Connextra template, *"As a [role], I want [goal], so that [benefit]"*
Atomicity analysis	Split user story to remove conjunctions
Conciseness analysis	Rewrite user story to avoid subordinate clause
Preciseness analysis	Change the action in *WHAT/WHY* segment using the recommended terminology from the standard glossary
Consistent analysis	Change the role in in the *WHO* segment or the actions in the *WHAT/WHY* segments using the recommended terminology
Conceptually sound analysis	Consider rewriting the user story using the recommended terminology
Uniqueness analysis	Delete one user story that has been identified as duplicate

4.2 THE AMBIGUITY ANALYSIS PIPELINE

An ambiguity analysis begins with an examination of the structure of the user stories to ensure their atomicity and well-formedness, as issues at this level may lead to more intricate semantic problems (Wautelet et al. 2016, 2017). Next, the tool checks the terminology used in user stories to maintain clarity and consistency (Urbieta et al. 2022). Following these processes, semantic analysis is performed to ensure that user stories express clear, consistent, and unique ideas without potentially causing ambiguities. To achieve this objective, the tool employs various NLP techniques aligned with the

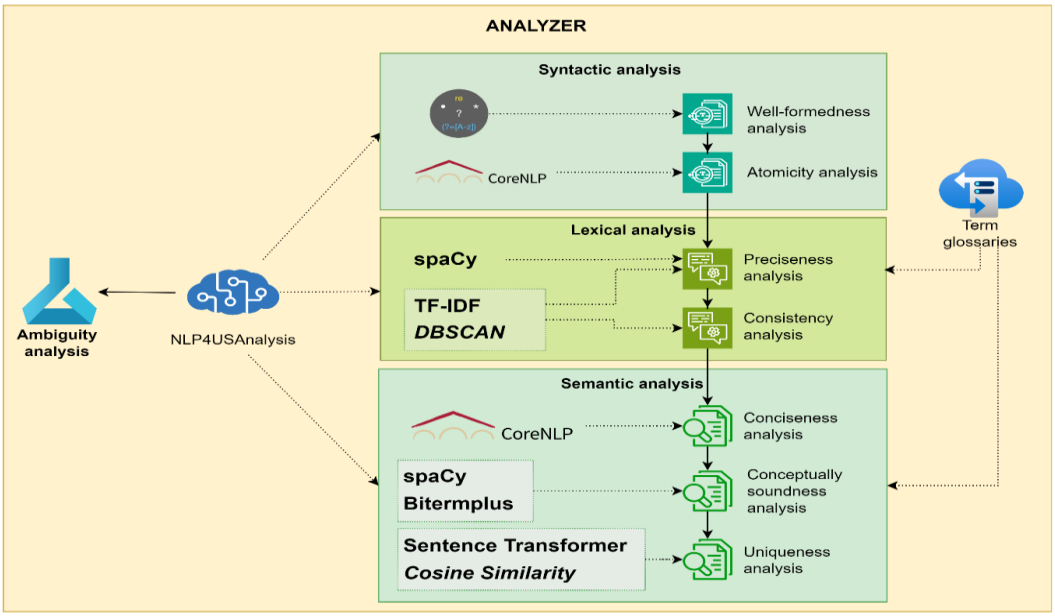

Figure 4. The pipeline of analyzer supporting AmbiTRUS tool

AmbiTRUS criteria. Figure 4 shows how the AmbiTRUS tool uses NLP technologies to identify ambiguity based on quality criteria outlined in the AmbiTRUS framework.
Our tool uses NLP techniques for basic tasks like breaking a user story sentence into tokens, analyzing the structure, and tagging Part-of-speech (POS) (Alzayed and Al-Hunaiyyan 2021; Zhao et al. 2021). We then enhanced these capabilities by integrating Machine Learning (ML) features, specifically Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering (Deng 2020) and Biterm Topic Models (Yan et al. 2013). As a result, the tool seamlessly combines NLP and ML techniques to check for the AmbiTRUS criteria compliance. A detailed list of the NLP technologies used in the AmbiTRUS tool is outlined in Table 2.

Table 2. The criteria analysis derived from AmbiTRUS framework involving NLP

Linguistic level	Quality criteria	Library/Toolkit	Algorithm/Techniques
Syntactic	Well-formed analysis	Regular Expression	Chunking
Syntactic	Atomicity analysis	Stanford CoreNLP parser, NLTK	Tokenization, POS tagging
Lexical	Preciseness analysis	SpaCy, Scikit-learn, DBSCAN, WordNet	Tokenization, Lemmatization, DBSCAN Clustering, POS tagging
Lexical	Consistency analysis	SpaCy, Scikit-learn, DBSCAN	Tokenization, Lemmatization, DBSCAN Clustering
Semantic	Conciseness analysis	Stanford CoreNLP parser, NLTK	Tokenization, POS tagging, Dependency parsing
Semantic	Conceptually soundness analysis	SpaCy, WordNet, Bitermplus	Tokenization, Text classification, Word extraction, Dependency parsing, Biterm Topic Model
Semantic	Uniqueness analysis	Sentence Transformer	Sentence embedding, Cosine similarity, Pairwise comparison

To facilitate the improvement process, the tool is equipped with a term glossary proposed by Müter et al. (Müter et al. 2019) as a foundation for terminology classification. This glossary dynamically evolves based on user input, leveraging both the terms identified as problematic during analysis and those selected by users during the user story improvement process. If users encounter an action keyword that is not present in the glossary, they have the flexibility to add it to ensure future usability.

4.2.1 Preprocessing

Preprocessing is the stage that prepares user stories for analysis. The stage consists of transforming user stories into "UTF-8" format, removing stop-words and special characters, and splitting user stories into their segments, *WHO, WHAT, WHY* (Figure 5). To remove stop-words, there are different libraries such as SpaCy ("Spacy.Io" n.d.), Gensim (Řehůřek et al. n.d.), and NLTK (NLTK team n.d.). This study chooses NLTK because it is easy to use.

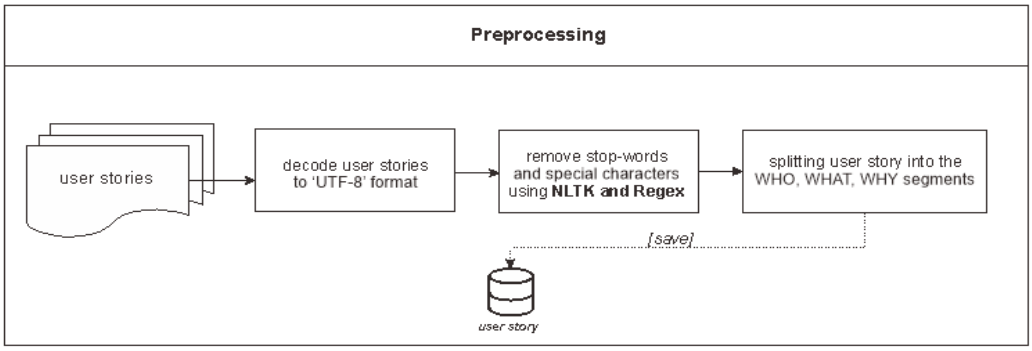

Figure 5. Activities performed in preprocessing stage.

The segmentation process focuses on splitting user stories into segments, which is known as chunking. The process iteratively reads each user story in the list using a regular expression, extracting identifiers and content related to *WHO*, *WHAT*, and *WHY* segments. The full process of the chunking is presented in Table 3.

Table 3. Algorithm to chunk user stories into user story segment

Input: User stories	
Output: List of user story segment	
1:	function chunk_user_stories(user_story_list):
2:	**user_story_segment ← empty list**
3:	**for each** userstory_id **in** user_story_list **do**
4:	who_identifier ← extract_who_identifier (userstory_id)
5:	who_content ← extract_ who_content (userstory_id)
6:	what_identifier ← extract_what_identifier (userstory_id)
7:	what_content ← extract_ what_content (userstory_id)
8:	why_identifier ← extract_why_identifier (userstory_id)
9:	why_content ← extract_ why_content(userstory_id)
10:	**segment_data** ← {
11:	"userstory_id": userstory_id,
12:	"who_identifier": who_identifier,
13:	"who_content": who_content,
14:	"what_identifier": what_identifier,
15:	"what_content": what_content,
16:	"why_identifier": why_identifier,
17:	"why_content": why_content,
18:	}
19:	user_story_segments.append(segment_data)
10:	**end for**
11:	**return** user_story_segments
12:	**end function**

As a result, a user story consists of six parts. The indicators are identified based on the list of indicators that is allowed in the Connextra template (Table 4).

Table 4. The list of supported identifiers for user story segments

Segment	Identifier
WHO	as an, as a, as
WHAT	I'm able to, I am able to, I want to, I want, I wish to, I can, I should be able to
WHY	so that, so, to, in order

4.2.2 Syntactic Analysis

The syntactic analysis involves two parts, well-formedness analysis and atomicity analysis. In well-formedness analysis, the focus is on ensuring that user story segments are complete. This examination involves the use of regular expressions and checking for any missing segments.

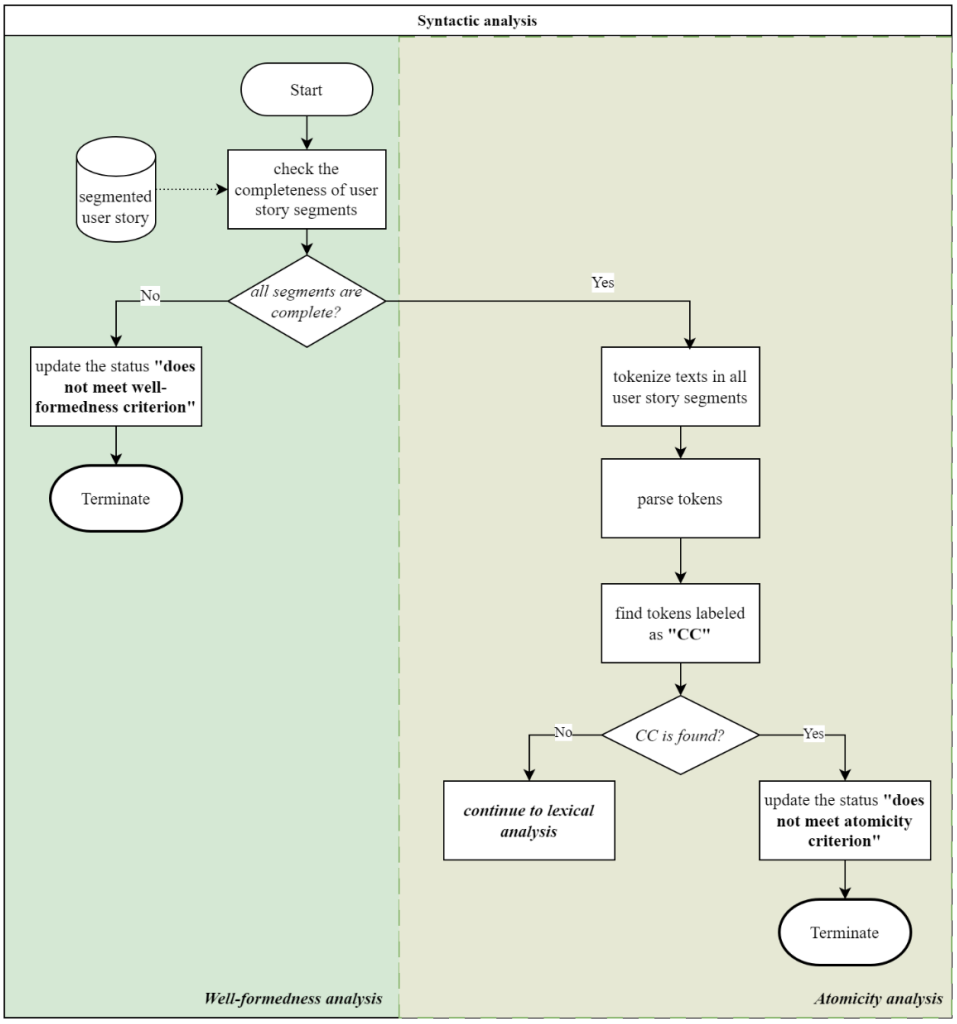

Figure 6. The outline of syntactic analysis

In atomicity analysis, the goal is to look for coordinating conjunctions (CC) in user stories. To do this, Part-of-Speech Tagging (POS-Tagging), chunking, and dependency parsing (Sonbol et al. 2022) are used. The flow of syntactic analysis is outlined in Figure 6.

- **Well-formedness analysis**

Well-formedness analysis is a crucial step for ensuring consistency in user stories. The analysis focuses on examining key segments of user stories, *WHO*, *WHAT*, and *WHY* segments (Gralha et al. 2022a; Murtazina and Avdeenko 2019). This analysis relies solely on the *null checking* mechanism without employing NLP techniques. The process begins by extracting various components from the user story segments. These components are then evaluated against specific conditions to assess the completeness and structural integrity of the user stories. If any components are missing or if there are syntactical errors in the structure of the user stories, the system flags them as "potentially ambiguous". To address flagged issues, users are prompted to rewrite their user story following the Connextra template. For further reference, the algorithm of the analysis is provided in Table 5.

Table 5. Algorithm to identify violation of the well-formedness criterion in user stories

Input: List of user story segment	
Output: IsProblem	
1:	**function identify_ambiguity(user_story_segments):**
2:	who_identifier_text ← " "
3:	who_user_text ← " "
4:	who_full ← user_story_segment.who_full
5:	**if** who_full is not **None** then
6:	who_identifier_text ← lower(who_full.who_identifier) **if** who_full.who_identifier is not **None else** " "
7:	who_user_text ← lower(who_full.who_user) **if** who_full.who_ user **is not None else** " "
8:	**end if**
9:	what_identifier_text ← " "
10:	what_user_text ← " "
11:	what_full ← user_story_segment.what_full
12:	**if** what_full is not **None** then
13:	what_identifier_text ← lower(what_full.what_identifier) **if** what_full.what_identifier is not **None else** " "
14:	what_user_text ← lower(what_full.what_user) **if** what_full.what_ user **is not None else** " "
15:	**end if**
16:	why_identifier_text ← " "
17:	why_user_text ← " "
18:	why_full ← user_story_segment.why_full
19:	**if** why_full is not **None** then
20:	why_identifier_text ← lower(why_full.why_identifier) **if** why_full.why_identifier **is not None else** " "
21:	why_user_text ← lower(why_full.why_user) **if** why_full.why_ user **is not None else** " "
22:	**end if**
23:	**isProblem ← False**

```
24:     if (
25:         who_identifier_text is empty or who_user_text is empty
26:         what_identifier_text is empty or what_user_text is empty
27:         why_identifier_text is empty or why_user_text is empty
28:     ) then
29:         isProblem ← True
30:     end if
31:     return isProblem
32: end function
```

Figure 7 illustrates the results of the well-formedness analysis, showing which user stories meet the well-formedness criterion based on the presence and the contents of their segments.

```
Full text: as an administrator, i want to have a connection with magicdraw, so that implementation details can easily be conn
ected to business rules.
Well-formed criteria is achieved!

Full text: as an administrator, i want to manage the format and layout of reports, so that the technical details are my respo
nsibility.
Well-formed criteria is achieved!

Full text: i want to separate projects, so that people cannot traverse each others work.
Problem: Well-formed is not achieved ! WHO segment is not not found !
Solution: Rewrite user story in Connextra format: *As a <role>, I want <action>, so that <goal>*

Full text: administrator, i want to have branching and merging, so that i can easily make releases from the rulebase.
Problem: Well-formed is achieved ! WHO segment is not complete. WHO identifier does not found !
Solution: Rewrite user story in Connextra format: *As a <role>, I want <action>, so that <goal>*
```

Figure 7. Illustration for the well-formedness analysis

- **Atomicity analysis**

Atomicity analysis ensures that each user story addresses one specific feature or function to maintain its clarity (Liskin et al. 2014; Lombriser et al. 2016). This analysis examines coordinating conjunctions (CC) in both the *WHAT* and *WHY* segments to determine if the *WHAT* segment introduces multiple functionalities needed to achieve the benefits described in the *WHY* segment (Berry and Kamsties 2004; Elallaoui et al. 2018; Gilson and Irwin 2018; Gralha et al. 2022b).

The analysis initiates by tokenizing sentences within user story segments and scrutinizing the syntactic structure of the tokens in the *WHAT* and *WHY* segments using the *Stanford CoreNLP Parser* (Group n.d.). To identify CC in these segments, the tokens undergo iterative examination until either the end of the text in the segments is reached or a CC is found (Manning et al. 2014). When a CC is found, the examination stops, shifting focus to the verb phrases preceding and following the CC. If verb phrases are identified, the atomicity analysis concludes, and the user stories are labelled as “potentially non-atomic”.

In cases where verb phrases are solely identified either before or after CC, an additional examination of the CC’s position is conducted. If CCs are present in both the *WHAT* and *WHY* segments, the user stories are labelled as “potentially non-atomic”. However, user stories are classified as “potentially atomic” when the CC is found only in either the *WHAT*

or *WHY* segments and verb phrases are identified only before or after the CC. Details on this algorithm are presented in Table 6.

Table 6. Algorithm for detecting complex sentences based on the presence of conjunctions in user stories segments

Input: List of *WHO*, *WHAT,* and *WHY* segments	
Output: List of segments containing CC and the verb after CC	
1:	**function** find_cc_with_verb(who_segments, what_segments, why_segments):
2:	cc_with_verb_segments ← empty list
3:	isProblem ← false
4:	**for** who_segment in who_segments:
5:	who_text ← who_segment[“text”]
6:	cc_and_verb ← find_cc_and_verb(who_text)
7:	**if** cc_and_verb:
8:	cc_with_verb_segments.append({
9:	“segment_type”: “who”,
10:	“segment_text”: who_text,
11:	“cc_text”: cc_and_verb[“cc_text”],
12:	“verb_after_cc”: cc_and_verb[“verb_after_cc”]
13:	})
14:	**for** what_segment in what_segments:
15:	what_text ← what_segment[“text”]
16:	cc_and_verb_what ← find_cc_and_verb(what_text)
17:	**if** cc_and_verb_what:
18:	cc_with_verb_segments.append({
19:	“segment_type”: “what”,
20:	“segment_text”: what_text,
21:	“cc_text”: cc_and_verb_what[“cc_text”],
22:	“verb_after_cc”: cc_and_verb_what[“verb_after_cc”]
23:	})
24:	**for** why_segment in why_segments:
25:	why_text ← why_segment[“text”]
26:	cc_and_verb_why ← find_cc_and_verb(why_text)
27:	**if** cc_and_verb_why:
28:	cc_with_verb_segments.append({
29:	“segment_type”: “why”,
30:	“segment_text”: why_text,
31:	“cc_text”: cc_and_verb_why[“cc_text”],
32:	“verb_after_cc”: cc_and_verb_why[“verb_after_cc”]
33:	})
34:	**if** (
35:	(cc_and_verb_why[“verb_after_cc”] **is not None**) **and**
36:	(cc_and_verb_why[“cc_text”] == cc_and_verb_what[“cc_text”])
37:	):

```
38:                         isProblem ← True
39:  return cc_with_verb_segment, isProblem

40:  function find_cc_and_verb(segment_text):
41:      tokens ← tokenize(segment_text)
42:      parsed_sentence ← parse_with_corenlp(tokens)
43:      for subtree in parsed_sentence.subtrees():
44:          if subtree.label() == "CC":
45:              vc_text ← subtree.leaves()[0]
46:              verb_after_cc ← find_verb_after_cc(subtree, parsed_sentence)
47:              return {"cc_text": cc_text, "verb_after_cc": verb_after_cc}
48:  return None

49:  function find_verb_after_cc(cc_subtree, parsed_sentence):
50:      for idx, sibling in enumerate (cc_subtree.rights()):
51:          if sibling.label().startswith("VB"):
52:              return sibling.leaves()[0]
53:          elif sibling.label() == "CC":
54:              Continue
55:          else:
56:              return search_for_verb_below(sibling, parsed_sentence)
57:  return None

58:  function find_verb_below(subtree, parsed_sentence):
59:      for child in subtree:
60:          if child.label().startswith("VB"):
61:          return child.leaves()[0]
62:          elif len(list(child.subtrees())) > 0:
63:              return search_for_verb_below(sibling, parsed_sentence)
64:  return None
```

Figure 8 presents the results of the atomicity analysis, indicating the status of user stories in relation to the atomicity criterion. The figure also specifies the coordinating conjunctions that may cause the potential problem.

```
Story # 64 :  as an admin, i want to have a pricing plan and billing system, so that i can charge users and make my platform sustainable.
Coordinating conjunction that could trigger syntactic ambiguity: ** and **

Status: User story does not meet atomicity criterion. User story is potentially ambiguous ! It is recommended to split user story !
==============================

Story # 65 :  as a publisher, i want to know if this site has a pricing plan and what the prices are, so that i can work out what this will cost me in the future and have a sense that these guys are sustainable.
Coordinating conjunction that could trigger syntactic ambiguity: ** and **

Status: User story does not meet atomicity criterion. User story is potentially ambiguous ! It is recommended to split user story !
==============================
```

Figure 8. Illustration for the atomicity analysis

4.2.3 Lexical Analysis

Lexical analysis consists of preciseness and consistency analysis. Although both use similar techniques, the goals are different. Preciseness analysis aims to detect any "noise" in user stories that could lead to "potential impreciseness", while consistency analysis ensures that terms used in user stories are classified as "within cluster".

The analysis uses TF-IDF to convert the *WHO* segment into vectors before employing DBSCAN clustering (Deng 2020). As a result of the clustering process, role values and the associated cluster labels are determined. Notably, default settings of scikit-learn are applied for DBSCAN hyperparameters, including an epsilon (ε) default value of 0.5, and the minimum number of points (minPts) is set to 2.

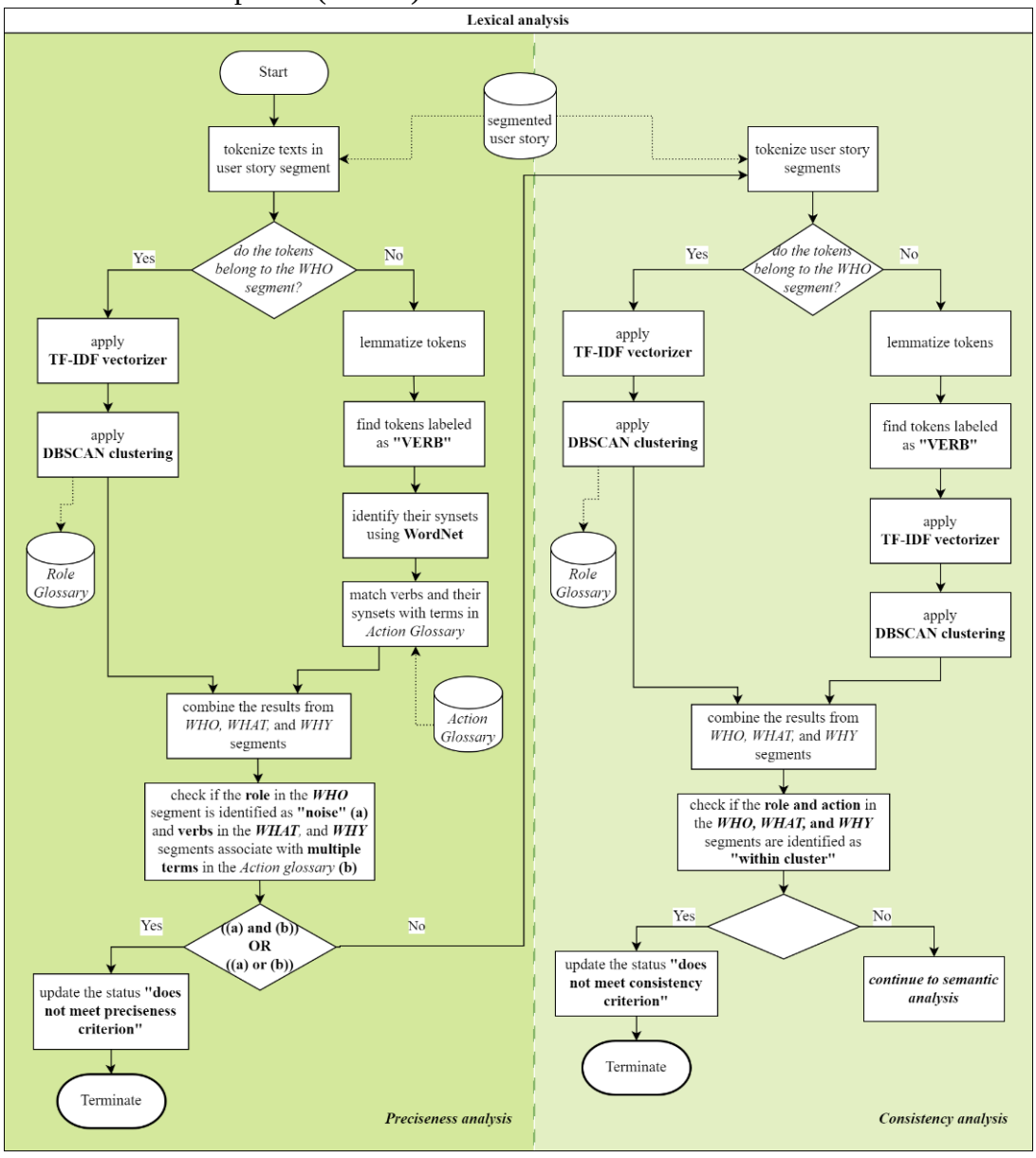

Figure 9. Illustration for preciseness and consistency analysis

In the *WHAT* and *WHY* segments, the SpaCy library identifies morphological features and *verbs*. However, these verbs are treated differently according to the analysis that has been conducted. Detailed procedures for preciseness and consistency analysis are illustrated in Figure 9.

- **Preciseness analysis**

Preciseness analysis is conducted to ensure clarity in user stories (Müter et al. 2019; Urbieta et al. 2022). This analysis involves examining terminology within user stories. The WHO segment uses unique domain context for actor description, while the WHAT and WHY segments use standard terms from the glossary to represent actions. This approach prevents different interpretations of action execution by identifying frequent *nouns* in the WHO segment and evaluating similarities between *verbs* in the WHAT and WHY segments and the predefined terms in the standard glossary. This process aims to associate verbs with a singular term class in the glossary.

The analysis starts with tokenizing texts in user story segments, followed by different processes according to the segments. In the *WHO* segment, the tokens are vectorized using TF-IDF (Das et al. 2021; Liu et al. 2022) before being clustered using DBSCAN clustering (Murugesan et al. 2021). Instances that are identified as "noisy" by the DBSCAN clustering are classified as the candidates of "potentially imprecise" user stories. This procedure is presented in Table 7 as follows.

Table 7. Algorithm to check the preciseness of the *WHO* segment

Input: well_formed_data	
Output: dic_sub	
1:	**function** actor_precise():
2:	role_s_values ← empty list
3:	role_s_text ← empty list
4:	userstory_values ← empty list
5:	**for** each item in well_formed_data **do**
6:	role_s_values.append(item.actor.who_action)
7:	role_s_text.append(item.userstory)
8:	userstory_values.append(item.userstory_obj)
9:	vectorizer ← create_tfidf_vectorizer(role_s_values)
10:	labels ← apply_dbscan_clustering(vectorizer)
11:	dic_sub ← group_and_print_clusters(role_s_values, role_s_text, userstory_values, labels)
12:	**return** dic_sub

In the *WHAT* segment, tokens undergo an iterative process. Initially, they are lemmatized to obtain the base form of the text. Subsequently, the lemmas are analyzed to determine their labels using the *SpaCy* library (Table 8). If a token is identified as a *verb*, both the verb and its lemma are added to their respective lists (Table 9). To precisely associate the

verb with a specific action, it is mapped with the word class terminology from the **action glossary**. When the word class is found, the verb along with its lemma is then linked to its corresponding keywords in the glossary (Table 10). If no word class is identified, the lemma is compared with synsets from WordNet (Miller 1995) (Table 11-12), and similar processes are followed for classification and mapping.

Table 8. Algorithm to lemmatize tokens in the *WHAT* segment

Input: token, prob_act, keyword_words, sentence_class, keyword_to_sentence_class	
Output: None (modifies prob_act, keyword_words, sentence_class, keyword_to_sentence_class)	
1:	**function** process_verb_token(token, prob_act, keyword_words, sentence_class, keyword_to_sentence_class):
2:	tok_verb ← token.text
3:	tok_process ←token.lemma_
4:	prob_act.append(tok_verb)
5:	word_class ← get_word_class(tok_process)
6:	**if** word_class:
7:	process_word_class(word_class, keyword_words, sentence_class, keyword_to_sentence_class)
8:	**else:**
9:	process_synsets(tok_process, keyword_words, sentence_class, keyword_to_sentence_class)
10:	**end if**
11:	**end function**

Table 9. Algorithm to identify verb in the *WHAT* segment

Input: token, prob_act, keyword_words, sentence_class, keyword_to_sentence_class	
Output: None (modifies prob_act, keyword_words, sentence_class, keyword_to_sentence_class)	
1:	**function** process_verb_token(token, prob_act, keyword_words, sentence_class, keyword_to_sentence_class):
2:	tok_verb ← token.text
3:	tok_process ←token.lemma_
4:	prob_act.append(tok_verb)
5:	word_class ← get_word_class(tok_process)
6:	**if** word_class:
7:	process_word_class(word_class, keyword_words, sentence_class, keyword_to_sentence_class)
8:	**else:**
9:	process_synsets(tok_process, keyword_words, sentence_class, keyword_to_sentence_class)
10:	**end if**
11:	**end function**

Table 10. Algorithm to map identified verbs into corresponding action glossary

Input: word_class, keyword_words, sentence_class, keyword_to_sentence_class	
Output: None (modifies keyword_words, sentence_class, keyword_to_sentence_class)	
1:	**function** process_word_class(word_class, keyword_words, sentence_class, keyword_to_sentence_class):
2:	sentence_class.add(word_class.keyword)
3:	keyword_words.add(word_class.text)
4:	update_keyword_to_sentence_class(word_class.text, keyword_to_sentence_class, word_class.keyword)
5:	**end function**

Table 11. Algorithm to classify unidentified word class into action glossary

Input: tok_process, keyword_words, sentence_class, keyword_to_sentence_class	
Output: None (modifies keyword_words, sentence_class, keyword_to_sentence_class)	
1:	**function** process_synsets(tok_process, keyword_words, sentence_class, keyword_to_sentence_class):
2:	synsets ← get_synsets(tok_process)
3:	**for** each synset **in** synsets **do**
4:	synset_class ← get_synset_class(synsets)
5:	**if** synset_class:
6:	sentence_class.add(synset_class.keyword)
7:	keyword_words.add(tok_process)
8:	update_keyword_to_sentence_class(tok_process, keyword_to_sentence_class, synset_class.keyword)
9:	**end if**
10:	**end for**
11:	**end function**

Table 12. The list of the predefined terminology representing the class of actions, later referred to as the "action glossary" (Müter et al. 2019).

Keyword class	Keyword Item
Create	add, insert, create, make, build, develop, establish, generate, construct, invite
Read	view, read, display, show, retrieve, get, access, examine, browse
Update	modify, edit, change, update, revise, alter, adjust, adapt, refine, fix, improve, renew, replace
Delete	remove, delete, erase, clear, eliminate, exclude, discard, purge, drop
Merge	bind, export, integrate, link, list, offer
Validate	check, evaluate, test, verify
Search	investigate, inquire, research, search

The assessment of user story preciseness relies on the *WHO*, *WHAT*, and *WHY* segments. A user story is considered satisfactory if the *noun* or *noun phrase* in the *WHO* segment is not identified as noise in cluster analysis, and if each *verb* in both *WHAT* and *WHY* segments is associated with a single action from the **action glossary**. Otherwise, a user story is labelled "potentially imprecise" if it fails to meet both conditions. The results are illustrated in Figure 10.

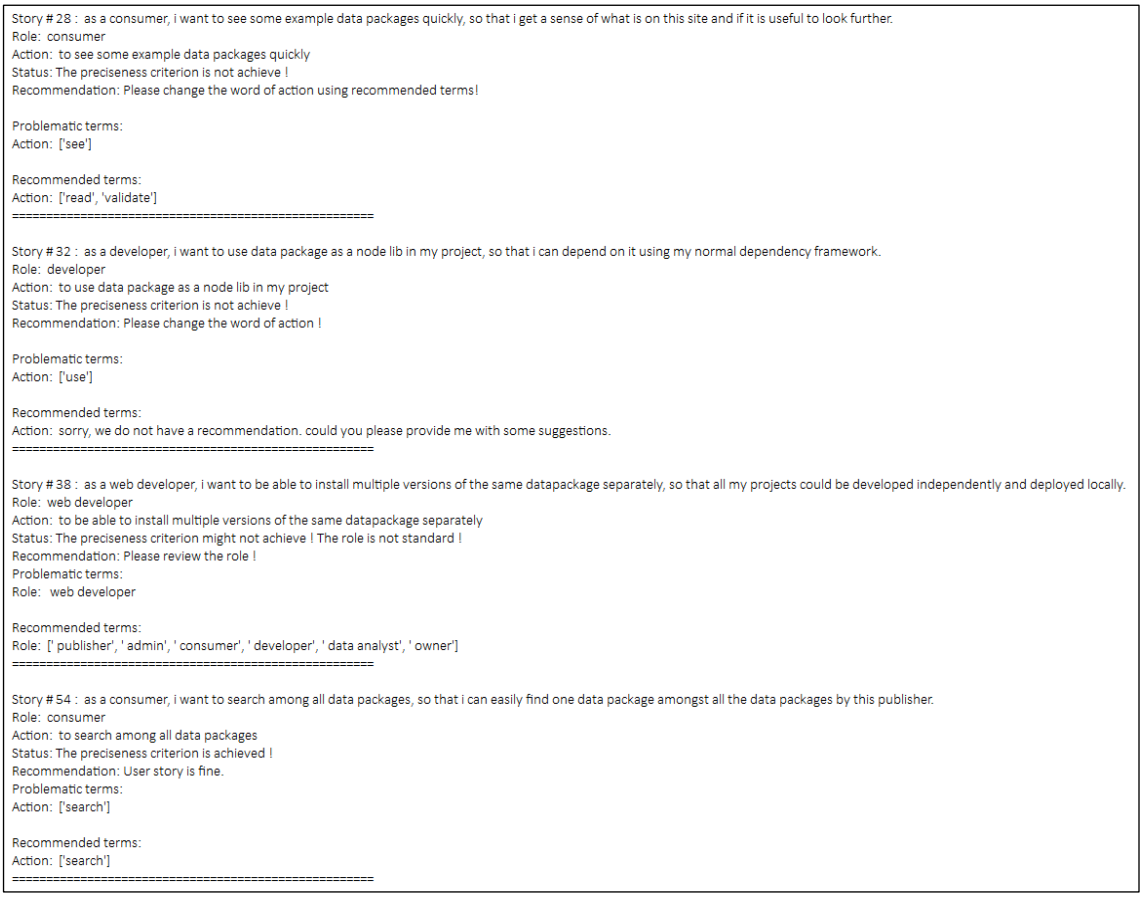

Figure 10. Illustration for preciseness analysis

- **Consistency analysis**

Consistency analysis is performed to ensure that a user story uses consistent terms to describe the same actor or similar behaviour in the same set of user stories. According to the Agile Requirements Verification Framework, it is essential to maintain project-specific glossaries and abbreviations to ensure consistency and correctness in agile requirements artifacts (Heck and Zaidman 2014). This finding is similar to other studies that recommend the use of standard glossaries to describe actors and behavioral responses (Müter et al. 2019; Urbieta et al. 2022). However, although consistency analysis and preciseness analysis follow similar processes, this analysis uses the results from the DBSCAN clustering in a different way.

The analysis starts by tokenizing the *WHO, WHAT,* and *WHY* segments using SpaCy. Subsequently, tokens in the *WHO* segment are converted into vectors using TF-IDF vectorization (Liu et al. 2022). The vectors are then clustered using DBSCAN clustering (Murugesan et al. 2021), resulting in role values and associated cluster labels (Table 13).

Table 13. Algorithm to identify the consistency in the *WHO* segment

```
Input: Well-formed data
Output: List of dictionaries containing consistent WHO segment
function who_consistency(well_formed_data):
    txt = []            # List to store user story texts
    r_txt = []          # List to store 'who' segment
    dic_role = []       # List to store consistent 'who' segment
    userstory_list = []   # List to store user story objects
    for item in well_formed_data:
        text = item["userstory"]
        userstory = item["userstory_obj"]
        role = item["actor"].Who_action
        txt.append(text)
        r_txt.append(role)
        userstory_list.append(userstory)
    end for
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(r_txt)
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(X)
    grouped_role_s = defaultdict(list)
    role_s_lists = defaultdict(list)
    total_role_members = defaultdict(int)
    for role, text, label, userstory in zip(r_txt, txt, labels, userstory_list):
        grouped_role_s[label].append((role, text, userstory))
    end for
    for label, role_s_text_list in grouped_role_s.items():
        role_s_list = []
        for role, text, userstory in role_s_text_list:
            if role not in role_s_list:
                role_s_list.append(role)
            end if
        end for
        role_s_lists[label] = role_s_list
        total_role_members[label] = len(role_s_list)
        if label == -1:
            for role, text, userstory in role_s_text_list:
                dic_role.append({"userstory": userstory, "text": text, "actor": role, "role_cluster_label": label})
            end for
        else:
            for role, text, userstory in role_s_text_list:
                dic_role.append({"userstory": userstory, "text": text, "actor": role, "role_cluster_label": label})
```

40:	**end for**
41:	**end if**
42:	**end for**
43:	**return** dic_role, role_s_lists, total_role_members

In the *WHAT* and *WHY* segments, tokens are iteratively lemmatized and inspected to identify *verbs*. The verbs obtained are then vectorized using TF-IDF and clustered to evaluate whether the verbs belong to any cluster labels (Table 14).

Table 14. Algorithm to check the consistency in the *WHAT* and *WHY* segments

Input: Well-formed data	
Output: List of dictionaries containing consistent *WHAT* and *WHY* segments	
1:	**function** who_consistency(well_formed_data):
2:	txt = [] # List to store user story texts
3:	a_txt = [] # List to store 'what' segment
4:	m_txt = [] # List to store 'why' segment
5:	dic = [] # List to store consistent 'what' and 'why' segments
6:	userstory_list = [] # List to store user story objects
7:	**for** item **in** well_formed_data:
8:	text = item["userstory"]
9:	userstory = item["userstory_obj"]
10:	action = item["action"].Why_action
11:	motive = item["motive"].Why_action
12:	action = action.translate(str.maketrans("", "", string.punctuation))
13:	motive = motive.translate(str.maketrans("", "", string.punctuation))
14:	words_act = word_tokenize(action)
15:	words_mot = word_tokenize(motive)
	lemmatized_stc_act = " ".join(lemmatizer.lemmatize(word) for word in words_act)
16:	lemmatized_stc_mot = " ".join(lemmatizer.lemmatize(word) for word in words_mot)
17:	vectorizer = TfidfVectorizer()
18:	X _act = vectorizer.fit_transform([lemmatized_stc_act])
19:	X _mot = vectorizer.fit_transform([lemmatized_stc_mot])
20:	dbscan = DBSCAN(eps=eps, min_samples=min_samples)
21:	labels_act = dbscan.fit_predict(X_act)
22:	labels_mot = dbscan.fit_predict(X_mot)
23:	status = "consistent" **if** labels_act [0] != -1 **and** labels_mot [0] != -1 **else** "inconsistent"
24:	dic.append({"userstory": userstory, "text": text, "action": lemmatized_stc_act, "motive": lemmatized_stc_mot, "act_cluster_label": labels_act[0], "mot_cluster_label": labels_mot[0], "status": status})
25:	act_s_lists = defaultdict(list)
26:	why_s_lists = defaultdict(list)
27:	total_act_members = defaultdict(int)
28:	total_why_members = defaultdict(int)
29:	**for** item **in** dic:

30:	act_cluster_label = item["act_cluster_label"]
31:	why_cluster_label = item["why_cluster_label"]
32:	act_s_lists[act_cluster_label] += role_s_lists[act_cluster_label]
33:	why_s_lists[why_cluster_label] += role_s_lists[why_cluster_label]
34:	total_act_members[act_cluster_label] = len(act_s_lists[act_cluster_label])
35:	total_why_members[why_cluster_label] = len(why_s_lists[why_cluster_label])
36:	item["isProblem"] = is_problem(total_act_members[act_cluster_label], total_why_members[why_cluster_label])
37:	**end for**
38:	**end for**
39:	**return** dic, what_s_lists, what_members, why_s_lists, total_why_members

The consistency analysis yields the results by considering the results of DBSCAN clustering performed in the entire user stories segments (i.e., *WHO*, *WHAT*, and *WHY*). User stories are considered non-ambiguous when the *WHO*, *WHAT*, and *WHY* segments are identified as "within cluster". Otherwise, they are considered "potentially inconsistent". The algorithm outlining this analysis is presented in Table 15, and the anticipated results are illustrated in Figure 11.

Table 15. Algorithm to determine the consistency of user stories

Input: total_role_members, total_act_members, total_why_members	
Output: None	
1:	**function** is_problem(total_role_members, total_act_members, total_why_members):
2:	**if** total_role_members > 1 **and** total_act_members > 1 **and** total_why_members > 1:
3:	**return** True
4:	**else:**
5:	**return** False
6:	**end if**
7:	**end function**

```
Story # 11 :  as an hrd manager, i want to select date and month and year, so that i can see who take days off and how long.
Role:  hrd manager
Action: to select date and month and year
Status: Consistency criterion is not achieved. User role is not consistent.
User story is potentially ambiguous!
Recommendation: Change the user role using the same terminology.
Problematic terms:   hrd manager
Recommendation terms:
Terms for role: {0: [' developer'], 1: [' business user'], 2: [' ruleauthor'], 3: [], 4: [' terminator'], 5: [' ruleanalys
t'], 6: [' administrator', 'administrator, ']}

--------------------

Story # 12 :  as a ruleauthor, i want to filter the list per rulebase with a new or existing filter, so that i can quickly fi
nd what i am looking for.
Role:  ruleauthor
Action: to filter the list per rulebase with a new or existing filter
Status: Consistency criterion is achieved. User story is good.
Recommendation: pass
```

Figure 11. Illustration for the consistency analysis

4.2.4 Semantic Analysis

Semantic analysis focuses on comprehending user stories' meanings. The analysis can be examined through the consistency of user stories when they are transformed into conceptual models and scrutinizing the user story description to deliver contextual information (**Author citation** 2022).

The analysis encompasses three key aspects, which are conciseness analysis, conceptual soundness, and uniqueness analysis. Conciseness analysis aims to identify implicit meanings within user stories by examining the presence of subordinate conjunctions. While often considered as a syntactic criterion (Heck and Zaidman 2014; Lucassen et al. 2016), issues arising from the appearance of subordinate conjunction (SBAR) extend into semantics as this may prevent the establishment of clear contexts (De Araujo and Siqueira 2016; Berry and Kamsties 2004; Wautelet et al. 2017).

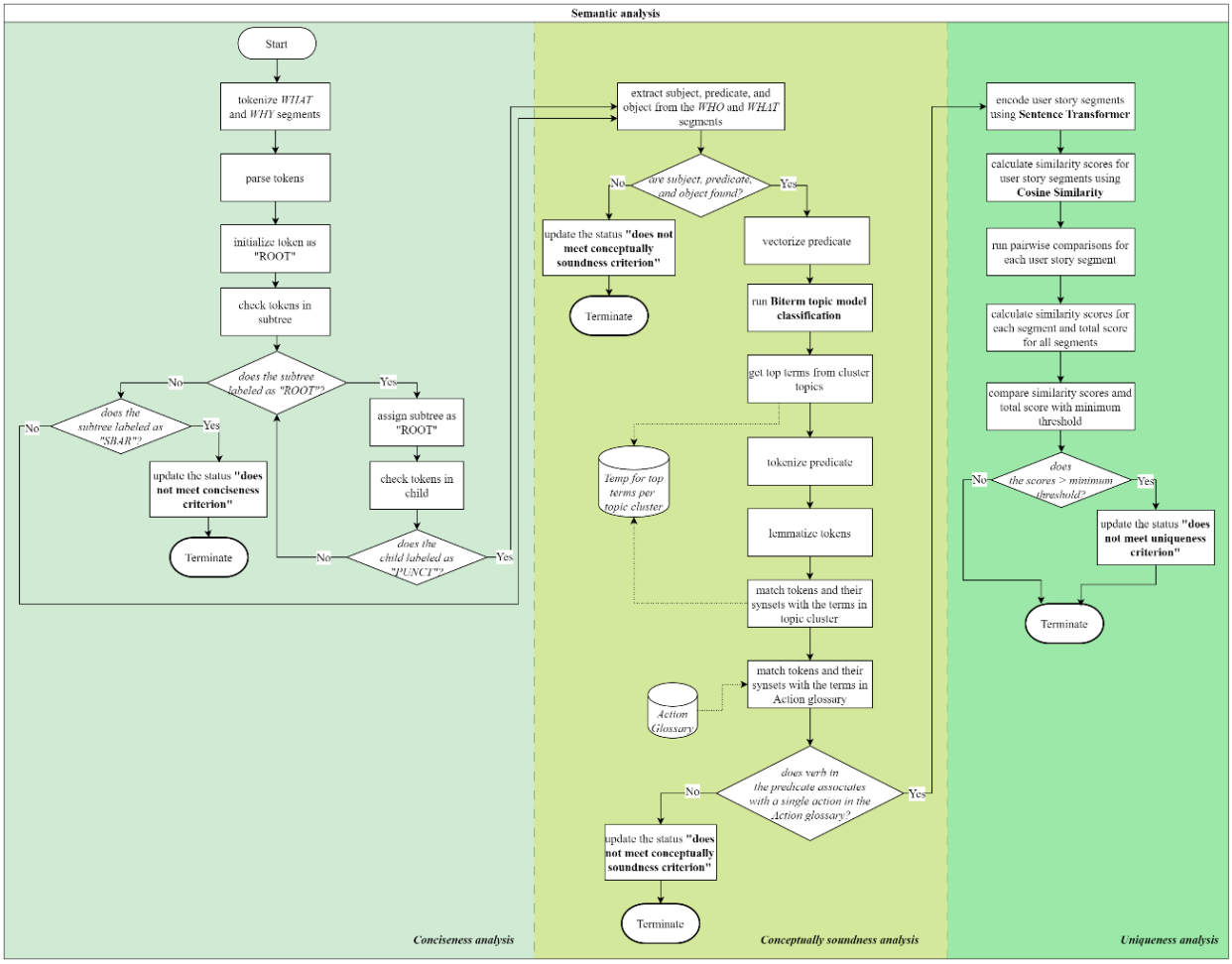

Figure 12. The outline of semantic analysis

Conceptual soundness involves categorizing user stories into particular topics to ensure clarity and prevent user stories from being underspecified or missing (Tiwari et al. 2020).

This analysis is facilitated by the BiTerm topic model which is well-suited for short text analysis (Yan et al. 2013).

Uniqueness analysis focuses on identifying similarities between user stories to mitigate duplication. This analysis employs Sentence Transformer[1] and Cosine similarities to identify similar user stories by extracting linguistics annotations from user story segments. The flow of the analysis is outlined in Figure 12.

- **Conciseness analysis**

Conciseness analysis is a process for ensuring that user stories convey the expected features/objects/functionalities clearly and concisely to prevent unintentionally omitting features or functionalities (Berry 2021; Ribeiro and Berry 2020). This process examines the presence of subordinating conjunctions (SBAR) that are sometimes added to provide additional information to clarify the connections among comprehension of logical relationships, general vocabulary knowledge, and reading comprehension, but which could make information in user stories less concise (Fraser et al. 2021; Lucassen et al. 2016).

The analysis is conducted specifically to user stories meeting the consistent terminology criterion. Unlike other analyses, conciseness analysis is employed in the *WHAT* and *WHY* segments of user stories, where this information is mainly presented. The analysis starts with tokenization and is followed by parsing the tokens to identify the root of the parsed *WHAT* and *WHY* segments and to identify the presence of SBAR in the sentence. These processes are performed using the *Stanford CoreNLP Parser*.

The adherence of user stories to the conciseness criterion is determined by the presence of SBAR in the *WHAT* and *WHY* segments. User stories with the SBAR in both segments are considered as "potentially ambiguous" and "do not meet the conciseness criterion", while those having SBAR in the *WHAT* segment are considered as "does not potentially ambiguous" even though it "does not meet conciseness criterion". This decision is motivated by the explanation of the *minimal* criterion in the QUS framework, which primarily requires the *WHO* and *WHAT* segments to be concise (Lucassen et al. 2016). The algorithm constructing conciseness criterion analysis is presented in Table 16, and the illustration of the expected results is presented in Figure 13.

Table 16. Algorithm to identify conciseness analysis in the *WHAT* and *WHY* segments

Input: action_what_text, goal_why_text	
Output: SBAR_text_what, SBAR_text_why, is_Problem, userstory	
1:	**function** find_sbar(action_what_text, goal_why_text):
2:	isProblem ← False
3:	act_sbar ← empty list
4:	goal_sbar ← empty list
5:	dependency_pairs_what ← empty list
6:	sbar_text_what ← empty list

[1] https://www.sbert.net/

```
7:      dependency_pairs_why ← empty list
8:      sbar_text_why ← empty list
9:      for each well_formed in well_formed_data do
10:         index ← 0
11:         action ← well_formed["action"]
12:         goal ← well_formed["goal"]
13:         text ← well_formed["userstory"]
14:         for each segment in ["action_what_text", "goal_why_text"] do
15:             if segment == "action_what_text" then
16:                 tokens_action ← word_tokenize(action_what_text)
17:                 parsed_sentence_action ← next(self.corenlp_parser.parse(tokens_action))
18:                 root_what = None
19:                 for each subtree in parsed_sentence_action.subtrees() do
20:                     if subtree.label() == "ROOT" then
21:                         root_what ← subtree
22:                     else if subtree.label() =="SBAR" then
23:                         sbar_text_what ← subtree.leaves()
24:                     end if
25:                 end for
26:                 for each child in root_what do
27:                     if isinstance(child, Tree) and child.label() != "PUNCT" then
28:                         dependency_pairs_what.append(child[0], child.label(), root_what[0])
                        else
29:                         dependency_pairs.append(child, "PUNCT", root_what[0])
30:                 end for
31:                 new_text_what ← action_what_text
32:                 if sbar_text_what then
33:                     sbar_text_joined_what ← " ".join(sbar_text_what)
34:                     sbar_start_index_what ← action_what_text.find(sbar_text_joined_what)
35:                     if sbar_start_index_what > = 0 then
36:                         sbar_end_index_what ← sbar_start_index_what + len(sbar_text_joined_what)
37:                         if sbar_start_index_what == len(action_what_text) then
38:                             new_text_what ← action_what_text[:sbar_start_index_what].strip()
39:                         else
40:                             new_text_what ← action_what_text[:sbar_start_index_what] + action_what_text[sbar_end_index_what:].strip()
41:                         end if
42:                     end if
43:                 end if
```

```
44:                 match_what ← re.search(r"I\s+want\s+to|I\s+want", new_text_what, re.IGNORECASE)
45:                 captured_text_what ← ""
46:                 if match_what then
47:                     start_index_what ← match_what.end ()
48:                     comma_index_what ← new_text_what. find (",", start_index_what)
49:                     if comma_index_what != -1 then
50:                         captured_text_what <-- new_text_what[start_index_what:comma_index_what]
51:                     else
52:                         captured_text_what <-- new_text_what[start_index_what:]
53:                     end if
54:                 end if
55:             else if segment == “goal_why_text” then
56:                 tokens_action ← word_tokenize(action_what_text)
57:                 parsed_sentence_action ← next(self.corenlp_parser.parse(tokens_action))
58:                 root_what = None
59:                 for each subtree in parsed_sentence_action.subtrees() do
60:                     if subtree.label() == “ROOT” then
61:                         root_what ← subtree
62:                     else if subtree.label() ==”SBAR” then
63:                         sbar_text_what ← subtree.leaves()
64:                     end if
65:                 end for
66:                 for each child in root_what do
67:                     if isinstance(child, Tree) and child.label() != “PUNCT” then
68:                     dependency_pairs_what.append(child[0], child.label(), root_what[0])
69:                     else
70:                         dependency_pairs.append(child, “PUNCT”, root_what[0])
71:                     end if
72:                 end for
73:                 new_text_what ← action_what_text
74:                 if sbar_text_what then
75:                     sbar_text_joined_what ← “ “.join(sbar_text_what)
76:                     sbar_start_index_what ← action_what_text.find(sbar_text_joined_what)
77:                     if sbar_start_index_what > = 0 then
78:                         sbar_end_index_what ← sbar_start_index_what + len(sbar_text_joined_what)
79:                         if sbar_start_index_what == len(action_what_text) then
80:                         new_text_what ← action_what_text[:sbar_start_index_what].strip()
81:                         else
```

```
                                    new_text_what ← action_what_text[:sbar_start_index_what] + action_what_text[sbar_end_index_what:].strip()
                                end if
                            end if
                        end if
                        match_what ← re.search(r"I\s+want\s+to|I\s+want", new_text_what, re.IGNORECASE)
                        captured_text_what ← ""
                        if match_what then
                            start_index_what ← match_what.end ()
                            comma_index_what ← new_text_what. find (",", start_index_what)
                            if comma_index_what != -1 then
                                captured_text_what <-- new_text_what[start_index_what:comma_index_what]
                            else
                                captured_text_what <-- new_text_what[start_index_what:]
                            end if
                        end if
                    end if
                end for
                if sbar_text__what or sbar_text__why then
                    isProblem ←True
                end if
                return sbar_text__what, sbar_text__why, action_what_text, goal_why_text, isProblem
```

```
Story # 31 :  as a ruleanalyst, i want to have an overview of all rulebase parts that have been reused, so that i can see wha
t the external dependencies are.

Subordinate conjunction that could trigger semantic ambiguity: ** that have been reused **
Status: User story does not meet conciseness criterion.
User story is potentially ambiguous ! It is recommended to remove subordinate conjunction !
==============================
Story # 33 :  as a ruleanalyst, i want to have an impact report, so that i can estimate the impact of a future change.

Status: User story meet conciseness criterion.
User story is not potentially ambiguous !
==============================
Story # 34 :  as a ruleanalyst, i want to see per business rule in what form and how much of it has been implemented, so that
is in known what is implemented from the rulebase.

Subordinate conjunction that could trigger semantic ambiguity: ** how much of it has been implemented **
Status: User story does not meet conciseness criterion.
User story is potentially ambiguous ! It is recommended to remove subordinate conjunction and split the user story !
==============================
```

Figure 13. Illustration of the conciseness analysis

- **Conceptual soundness analysis**

A thorough analysis is performed to assess whether user stories effectively communicate the necessary actions for delivering requested objects, features, or functionalities. This involves identifying *verb* and *noun phrases* in the *WHO* and *WHAT* segments, classifying

user stories, and ensuring each user story expresses a single action towards one object/feature/functionality (Table 17).

Table 17. Algorithm to extract subject (role), object (feature/functionality), and predicate (action) in user stories

Input: well_formed_data	
Output: sentence_dependency	
1:	**function** extract_subject_object_predicate(well_formed_data):
2:	sentence_dependency ← empty list
3:	dict_sent ← empty dictionary
4:	**for** each item **in** well_formed_data **do**
5:	userstory ←item[“userstory_obj”]
6:	text ←item[“userstory”]
7:	who ← item[“actor”].who_action **if** item[“actor”] **else** None
8:	what ← item[“actor”].what_action **if** item[“action”] **else** None
9:	why ← item[“actor”].why_action **if** item[“motive”] **else** None
10:	index ← item[“index”]
11:	doc ← nlp(action)
12:	subject ← None
13:	predicate ← None
14:	obj ← None
15:	skip_next_token ← False
16:	skip_next_verb ← False
17:	**for** each token in doc **do**
18:	**if** skip_next_token **then**
19:	skip_next_token ← False
20:	**continue**
21:	**end if**
22:	**if** lowercase(token.text) **is** "want to" **or** lowercase(token.text) **is** "want" **then**
23:	skip_next_token ← True
24:	**continue**
25:	**end if**
26:	**if** skip_next_verb **and** token.pos_ **is** "VERB" **then**
27:	skip_next_verb ← False
28:	**continue**
29:	**end if**
30:	**if** "subj" **is in** token.dep_ **then**
31:	subject ← token.text
32:	**else if** "obj" **is in** token.dep_ **and** (token.pos_ **is** "PROPN" **or** token.pos_ **is** "NOUN") **then**
33:	obj ← token.text
34:	**end if**
35:	**if** list(doc[token.left_edge.i : token.right_edge.i + 1].noun_chunks) **then**

36:	obj ← concatenate text **for** each chunk **in** doc[token.left_edge.i : token.right_edge.i + 1].noun_chunks
37:	**else if** token.pos_ **is** "VERB" **then**
38:	**if** predicate **is** None **then**
39:	predicate ← token.text
40:	**else**
41:	predicate ← concatenate predicate **and** token.text
42:	**end if**
43:	skip_next_verb ← True
44:	**end if**
45:	**if** token.subtree **then**
46:	predicate ← concatenate text **for** each t **in** token.subtree
47:	**end if**
48:	**end for**
49:	**end for**
50:	**return** sentence_dependency

The SpaCy library was employed for component extraction, focusing on extracting *verb*/*verb phrases* and *noun/noun phrases* in *WHAT* segments representing predicates and objects. The extracted predicate was then vectorized and classified using the Biterm topic model (Yan et al. 2013) (Table 18). To ensure each user story represents a single action delivering one feature/functionality, the *verb/verb phrases* and their synsets in user story predicates were examined through sentence matching using the **action glossary**. This sentence-matching process follows similar steps to the one performed in the preciseness analysis (Table 10-11).

Table 18. Algorithm to perform Biterm topic model

Input: sentence_dependency	
Output: topic_btm	
1:	**function** classify_topic(sentence_dependency):
2:	topic_btm = []
3:	new_docs = []
4:	**for** each dic_sent **in** sentence_dependency:
5:	index1 = dic_sent["index"]
6:	sent = dic_sent["sentence"]
7:	subject = dic_sent["subject"]
8:	predicate = dic_sent["predicate"]
9:	obj = dic_sent["object"]
10:	**if** predicate **is not** None:
11:	doc = nlp(predicate)
12:	filtered_predicate = **concatenate** token.text **for** each token **in** doc **if not** token.is_stop
13:	cleaned_filtered_predicate = **remove** non-alphanumeric characters **from** filtered_predicate

```
14:                 new_docs.append(cleaned_filtered_predicate)
15:             end if
16:         end for
17:         X, vocabulary, vocab_dict = btm.get_words_freqs(new_docs)
18:         docs_vec = btm.get_vectorized_docs(new_docs, vocabulary)
19:         biterms = btm.get_biterms(docs_vec)
20:         model = btm.BTM(X, vocabulary, T=8, M=20, alpha=50/7, beta=0.01)
21:         model.fit(biterms, iterations=100)
22:         p_zd = model.fit_transform(docs_vec, biterms, infer_type=u'sum_b', iterations=100)
23:         result = btm.get_docs_top_topic(new_docs, p_zd)
24:         for each index, doc_topic_dist in enumerate(p_zd):
25:             text = new_docs[index]
26:             cluster_topic = index of the maximum value in doc_topic_dist
27:             if text is not None and cluster_topic is not None:
28:                 matching_predicates = list of dic_sent where predicate contains text
29:                 for each dic_sent in matching_predicates:
30:                     top_words = btm.get_top_topic_words(model, words_num=10,
                        topics_idx=[cluster_topic])
31:                     word_column = top_words.columns[0]
32:                     cluster_words = top_words[word_column].tolist()
33:                     shuffle(cluster_words)
34:                     sentence_length = minimum of 5 and length of cluster_words
35:                     sentence = randomly select sentence_length words from cluster_words
36:                     cluster_sentence = concatenate words in sentence
37:                 end for
38:             end if
39:         end for
40:         return topic_btm
```

User stories are considered as “potentially ambiguous” and do not meet the conceptual soundness criterion when their *verb/verb phrase* is not associated with a single action from the **action glossary**. However, this problem can be resolved by referring to other user stories classified by the same topic to predict the intended actions that need to be addressed by “potentially underspecified” user stories. The illustration of the conceptual soundness analysis is depicted in Figure 14.

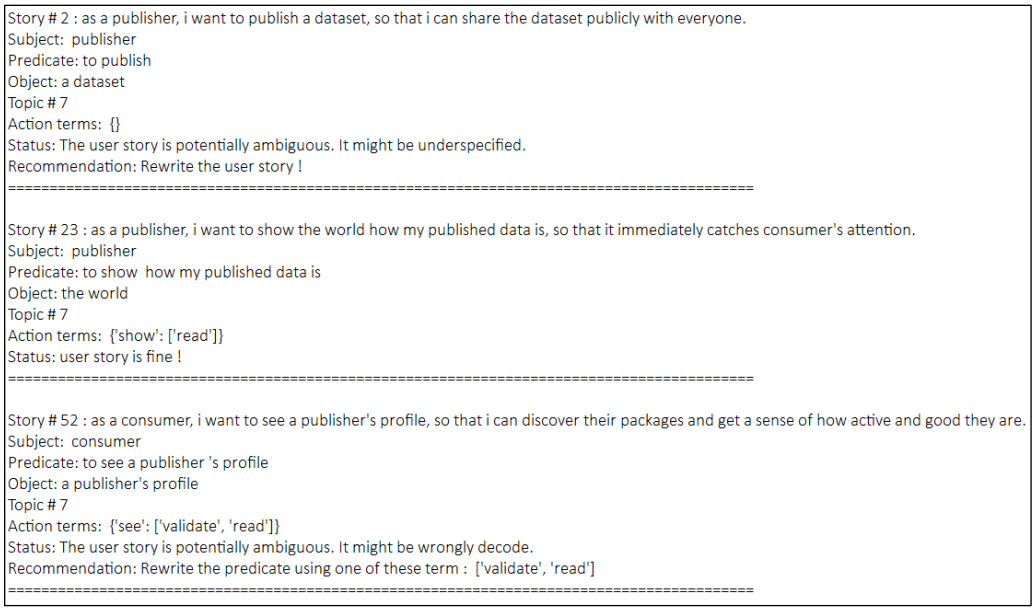

Figure 14. Illustration of the conceptual soundness analysis

- **Uniqueness analysis**

This analysis is performed to detect potential similarities among user stories. The analysis focused on examining *WHO*, *WHAT*, and *WHY* segments in user stories. The *Sentence Transformer* model, specifically "all-MiniLM-L6-v2" (Hugging Face n.d.) is employed to encode these segments, and *Cosine similarity* (Han et al. 2012) is used to compute similarity scores.

The cosine similarity metric measures the similarity between user story segments. Iterative pairwise comparisons are performed for each combination of user stories, distinctively for the three segments, resulting in similarity scores that reflect the degree of similarity between them. It is noteworthy to mention that this study adopts a uniform minimum threshold of 0.6 for all user story segments.

The chosen threshold of 0.6 serves as a critical determinant to assess adherence to the uniqueness criterion. User stories with similarity scores above this threshold is flagged as potentially ambiguous. Importantly, users have the flexibility to independently modify this threshold in the tool, offering adaptability to specific project requirements.

Figure 15 represents the outcome of the uniqueness analysis, providing a clear depiction of the relationships between user stories. A detailed understanding of the algorithm employed in this analysis can be found in Table 19.

Table 19. Algorithm to calculate similarities between user stories

Input: well_formed_data
Output: userstory, isProblem

```
1:  function uniqueness_criterion(well_formed_data):
2:      pair_role = []
3:      pair_action = []
4:      pair_goal = []
5:      tot_score = []
6:      isProblem = False
7:      role_user = []
8:      action_user = []
9:      goal_user = []
10:     for each item in self.well_formed_data do
11:         append item["actor"].Who_action to role_user if item["actor"] is not None else append None
12:         append item["action"].What_action to action_user if item["action"] is not None else append None
13:         append item["goal"].Why_action to goal_user if item["goal"] is not None else append None
14:         append item["userstory_obj"] to userstory_list
15:     end for
16:     role_embeddings = settings.MODEL_ST.encode(role_user)
17:     action_embeddings = settings.MODEL_ST.encode(action_user)
18:     goal_embeddings = settings.MODEL_ST.encode(goal_user)
19:     score_role = util.cos_sim(role_embeddings, role_embeddings)
20:     score_action = util.cos_sim(action_embeddings, action_embeddings)
21:     score_goal = util.cos_sim(goal_embeddings, goal_embeddings)
22:     for each i, j in itertools.combinations(range(len(self.well_formed_data)), 2) do
23:         append {'index': [i, j], 'sim_score_role': score_role[i][j]} to pair_role
24:     end for
25:     for each i, j in itertools.combinations(range(len(self.well_formed_data)), 2) do
26:         append {'index': [i, j], 'sim_score_action': score_action[i][j]} to pair_action
27:     end for
28:     for each i, j in itertools.combinations(range(len(self.well_formed_data)), 2) do
29:         append {'index': [i, j], 'sim_score_goal': score_goal[i][j]} to pair_goal
30:     end for
31:     for each i, j in itertools.combinations(range(len(self.well_formed_data)), 2) do
32:         append {'index': [i, j], 'sim_score_tot': (score_role[i][j] + score_action[i][j] + score_goal[i][j]) / 3} to tot_score
33:     end for
34:     result = zip(pair_role, pair_action, pair_goal, tot_score)
35:     result = sort result by key=lambda x: x[3]['sim_score_tot'] in descending order
36:     isProblem = all(x['sim_score_role'] >= 0.6 and x['sim_score_action'] >= 0.6 and x['sim_score_goal'] >= 0.6 for x in result)
37:     return isProblem
```

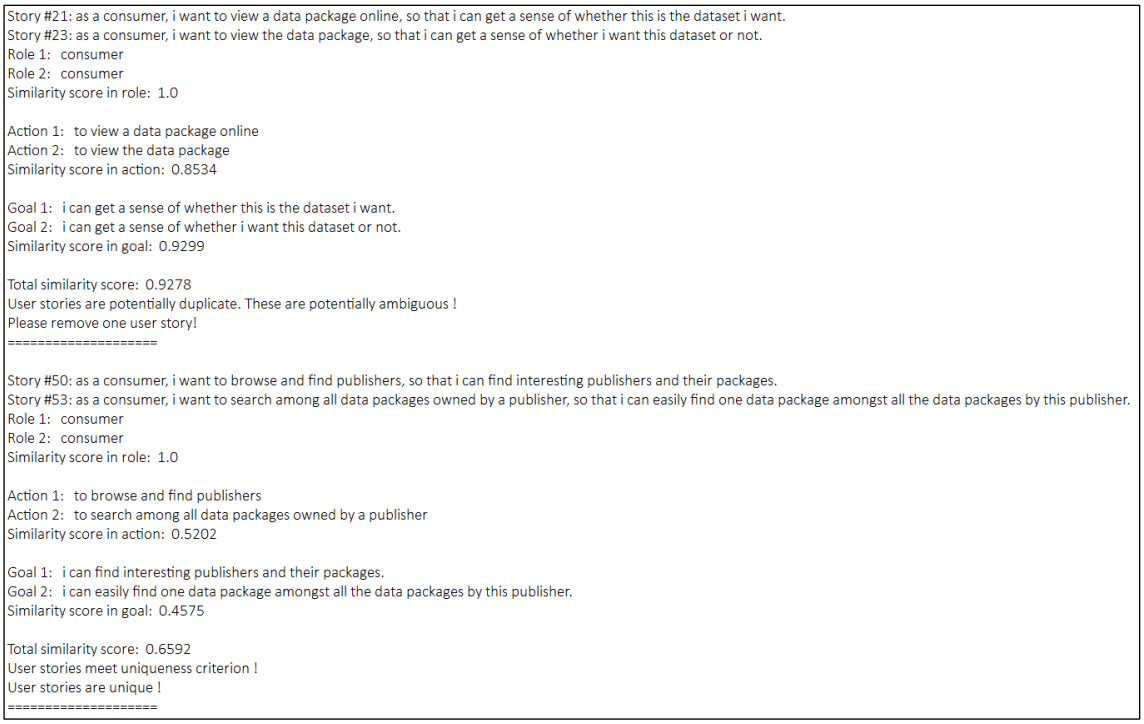

Figure 15. Illustration of the uniqueness analysis

5. CONCLUSIONS

The development of the AmbiTRUS tool represents a significant advancement as an operationalization of the AmbiTRUS framework. It was designed to improve the framework's effectiveness and usability, and the productivity of its users, by transitioning from a manual review of user story quality criteria to a semi-automated approach. This was accomplished through the integration of various NLP technologies.

However, the AmbiTRUS tool does have limitations. Notably, the tool is restricted to a single user story template, namely Connextra template. These limitations constrain its applicability and functionality, suggesting areas for future development and improvement.

6. FUTURE WORK

Our study confirms the significant impact of the AmbiTRUS tool on enhancing user performance and efficiency in identifying potential ambiguity in user stories. However, it also sheds light on certain limitations of the tool, particularly in its effectiveness, and areas for improvement.

To establish a benchmark for the AmbiTRUS tool, it is essential to evaluate a version of the tool that includes an analysis of the *WHY* segment (i.e., version 1.1). This evaluation will provide more comprehensive results regarding the effectiveness of the tool in analyzing potential ambiguity in a set of user stories. Integrating the manual glossary with

an LLM such as ChatGPT could offer more improvement recommendation options. Additionally, simplifying the user interface (UI) by incorporating colour-coded highlighting can significantly improve the visualization of the review process and enhance the overall user experience.

It is essential to explore the tool's potential to improve awareness of potential ambiguity in user stories among novice users, including requirements engineers and business analysts. Employing eye-tracking technology can offer valuable insights into the cognitive process of the tools' users during user story ambiguity analysis. Integrating the tool into training or certification programs can help assess its impact on participant understanding and skills. Furthermore, sharing insights from our study can enrich education in requirements engineering classes, fostering a deeper understanding of ambiguity issues and practical tools for addressing them.

In conclusion, addressing the identified limitations and enhancing the AmbiTRUS tool are critical steps towards improving user efficiency, satisfaction, and awareness. By implementing these recommendations, we can further advance the capabilities and impact of the tool in addressing potential ambiguity in user stories.

7. REFERENCES

Alhoshan, W., Zhao, L., Ferrari, A., and Letsholo, K. J. 2022. "A Zero-Shot Learning Approach to Classifying Requirements: A Preliminary Study," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 13216 LNCS), Springer International Publishing. (https://doi.org/10.1007/978-3-030-98464-9_5).

Alzayed, A., and Al-Hunaiyyan, A. 2021. "A Bird's Eye View of Natural Language Processing and Requirements Engineering," *International Journal of Advanced Computer Science and Applications* (12:5), pp. 81–90. (https://doi.org/10.14569/IJACSA.2021.0120512).

Author citation 2022. "Blinded for Review".

De Araujo, L. B., and Siqueira, F. L. 2016. "Using I∗ with Scrum: An Initial Proposal," in *CEUR Workshop Proceedings* (Vol. 1674), pp. 19–24.

Berry, D. M. 2021. "Empirical Evaluation of Tools for Hairy Requirements Engineering Tasks," in *Empirical Software Engineering* (Vol. 26), Empirical Software Engineering. (https://doi.org/10.1007/s10664-021-09986-0).

Berry, D. M., and Kamsties, E. 2004. "Ambiguity in Requirements Specification," *Perspectives on Software Requirements*, pp. 7–44. (https://doi.org/10.1007/978-1-4615-0465-8_2).

Dalpiaz, F., Dell'Anna, D., Aydemir, F. B., and Çevikol, S. 2019. "Requirements Classification with Interpretable Machine Learning and Dependency Parsing,"

Proceedings of the IEEE International Conference on Requirements Engineering (2019-Septe), pp. 142–152. (https://doi.org/10.1109/RE.2019.00025).

Dalpiaz, F., van der Schalk, I., Brinkkemper, S., Aydemir, F. B., and Lucassen, G. 2019. "Detecting Terminological Ambiguity in User Stories: Tool and Experimentation," *Information and Software Technology* (110), pp. 3–16. (https://doi.org/10.1016/j.infsof.2018.12.007).

Dalpiaz, F., van der Schalk, I., and Lucassen, G. 2018. "Pinpointing Ambiguity and Incompleteness in Requirements Engineering via Information Visualization and NLP," in *International Working Conference on Requirements Engineering: Foundation for Software Quality* (Vol. 10753 LNCS), Springer, pp. 119–135. (https://doi.org/10.1007/978-3-319-77243-1_8).

Das, M., Kamalanathan, S., and Alphonse, P. 2021. "A Comparative Study on TF-IDF Feature Weighting Method and Its Analysis Using Unstructured Dataset," *CEUR Workshop Proceedings* (2870), pp. 98–107.

Deng, D. 2020. "DBSCAN Clustering Algorithm Based on Density," *Proceedings - 2020 7th International Forum on Electrical Engineering and Automation, IFEEA 2020*, pp. 949–953. (https://doi.org/10.1109/IFEEA51475.2020.00199).

Elallaoui, M., Nafil, K., and Touahni, R. 2018. "Automatic Transformation of User Stories into UML Use Case Diagrams Using NLP Techniques," in *Procedia Computer Science* (Vol. 130), Elsevier B.V., pp. 42–49. (https://doi.org/10.1016/j.procs.2018.04.010).

Ezzini, S., Abualhaija, S., Arora, C., and Sabetzadeh, M. 2022. "Automated Handling of Anaphoric Ambiguity in Requirements: A Multi-Solution Study," *Proceedings - International Conference on Software Engineering* (2022-May), pp. 187–199. (https://doi.org/10.1145/3510003.3510157).

Fantechi, A., Gnesi, S., and Semini, L. 2023. "VIBE: Looking for Variability In Ambiguous Requirements," *Journal of Systems and Software* (195), Elsevier Inc., p. 111540. (https://doi.org/10.1016/j.jss.2022.111540).

Fernández, D. M., Wagner, S., Kalinowski, M., Felderer, M., Mafra, P., Vetrò, A., Conte, T., Christiansson, M.-T., Greer, D., Lassenius, C., Männistö, T., Nayabi, M., Oivo, M., Penzenstadler, B., Pfahl, D., Prikladnicki, R., Ruhe, G., Schekelmann, A., Sen, S., Spinola, R., Tuzcu, A., de la Vara, J. L., and Wieringa, R. 2017. "Naming the Pain in Requirements Engineering," *Empirical Software Engineering* (22:5), pp. 2298–2338. (https://doi.org/10.1007/s10664-016-9451-7).

Fraser, C., Pasquarella, A., Geva, E., Gottardo, A., and Biemiller, A. 2021. "English Language Learners' Comprehension of Logical Relationships in Expository Texts: Evidence for the Confluence of General Vocabulary and Text-Connecting Functions," *Language Learning* (71:3), pp. 872–906. (https://doi.org/10.1111/lang.12453).

Gilson, F., and Irwin, C. 2018. "From User Stories to Use Case Scenarios towards a Generative Approach," in *Proceedings - 25th Australasian Software Engineering Conference, ASWEC 2018*, Institute of Electrical and Electronics Engineers Inc., December 24, pp. 61–65. (https://doi.org/10.1109/ASWEC.2018.00016).

Gralha, C., Pereira, R., Goulão, M., and Araujo, J. 2022a. "Assessing User Stories: The Influence of Template Differences and Gender-Related Problem-Solving Styles," *Requirements Engineering* (27:4), Springer Science and Business Media Deutschland GmbH, pp. 521–544. (https://doi.org/10.1007/s00766-022-00389-1).

Gralha, C., Pereira, R., Goulão, M., and Araujo, J. 2022b. "Assessing User Stories: The Influence of Template Differences and Gender-Related Problem-Solving Styles," *Requirements Engineering* (27:4), Springer Science and Business Media Deutschland GmbH, pp. 521–544. (https://doi.org/10.1007/s00766-022-00389-1).

Group, S. N. (n.d.). "StanfordCoreNLP." (https://stanfordnlp.github.io/CoreNLP/, accessed January 23, 2024).

Han, J., Kamber, M., and Pei, J. 2012. "Getting to Know Your Data," *Data Mining*. (https://doi.org/10.1016/b978-0-12-381479-1.00002-2).

Heck, P., and Zaidman, A. 2014. *A Quality Framework for Agile Requirements: A Practitioner's Perspective*. (http://arxiv.org/abs/1406.4692).

Hevner, A. R., March, S. T., Park, J., and Ram, S. 2004. "Design Science in Information Systems Research," *MIS Quarterly: Management Information Systems* (28:1), pp. 75–105. (https://doi.org/10.2307/25148625).

Hugging Face. (n.d.). "Sentence Transformer." (https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, accessed January 23, 2024).

Kocerka, J., Krześlak, M., and Gałuszka, A. 2022. "Ontology Extraction from Software Requirements Using Named-Entity Recognition," *Advances in Science and Technology Research Journal* (16:3), pp. 207–212. (https://doi.org/10.12913/22998624/149941).

Koh, S. J., and Chua, F. F. 2023. "ReqGo: A Semi-Automated Requirements Management Tool," *International Journal of Technology* (14:4), pp. 713–723. (https://doi.org/10.14716/ijtech.v14i4.5631).

Liskin, O., Pham, R., Kiesling, S., and Schneider, K. 2014. "Why We Need a Granularity Concept for User Stories," in *Lecture Notes in Business Information Processing* (Vol. 179 LNBIP), pp. 110–125. (https://doi.org/10.1007/978-3-319-06862-6_8).

Liu, C., Sheng, Y., Wei, Z., and Yang, Y.-Q. 2022. "Research of Text Classification Based on Improved TF-IDF Algorithm," in *2018 the International Conference of Intelligent Robotic and Control Engineering* (Vol. 2171), IEEE, pp. 218–222. (https://doi.org/978-1-5386-7416-1/18/$31.00).

Lombriser, P., Dalpiaz, F., Lucassen, G., and Brinkkemper, S. 2016. "Gamified Requirements Engineering: Model and Experimentation," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pp. 171–187. (https://doi.org/10.1007/978-3-319-30282-9_12).

Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., and Brinkkemper, S. 2016. "Improving Agile Requirements: The Quality User Story Framework and Tool," *Requirements Engineering* (21:3), Springer London, pp. 383–403. (https://doi.org/10.1007/s00766-016-0250-x).

Luisa, M., Mariangela, F., and Pierluigi, N. I. 2004. "Market Research for Requirements Analysis Using Linguistic Tools," *Requirements Engineering* (9:1), pp. 40–56. (https://doi.org/10.1007/s00766-003-0179-8).

Manning, C. D., Bauer, J., Finkel, J., and Bethard, S. J. 2014. "The Stanford CoreNLP Natural Language Processing Toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60. (http://macopolo.cn/mkpl/products.asp).

Miller, G. A. 1995. "WordNet: A Lexical Database for English," *Communications of the ACM* (38:11), pp. 39–41. (https://doi.org/10.1145/219717.219748).

Murtazina, M. S., and Avdeenko, T. V. 2019. "An Ontology-Based Approach to Support for Requirements Traceability in Agile Development," *Procedia Computer Science* (150), Elsevier B.V., pp. 628–635. (https://doi.org/10.1016/j.procs.2019.02.044).

Murugesan, N., Cho, I., and Tortora, C. 2021. "Benchmarking in Cluster Analysis: A Study on Spectral Clustering, DBSCAN, and K-Means," *Studies in Classification, Data Analysis, and Knowledge Organization* (Vol. 5), Springer International Publishing. (https://doi.org/10.1007/978-3-030-60104-1_20).

Müter, L., Deoskar, T., Mathijssen, M., Brinkkemper, S., and Dalpiaz, F. 2019. "Refinement of User Stories into Backlog Items: Linguistic Structure and Action Verbs," in *International Working Conference on Requirements Engineering: Foundation for Software Quality* (Vol. 11412 LNCS), pp. 109–116. (https://doi.org/10.1007/978-3-030-15538-4_7).

NLTK team. (n.d.). "NLTK." (https://www.nltk.org/, accessed February 5, 2024).

Osama, M., Zaki-Ismail, A., Abdelrazek, M., Grundy, J., and Ibrahim, A. 2020. "Score-Based Automatic Detection and Resolution of Syntactic Ambiguity in Natural Language Requirements," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 651–661. (https://doi.org/10.1109/ICSME46990.2020.00067).

Řehůřek, R., Mohr, G., Penkov, M., and Menshikh, I. (n.d.). "Gensim Topic Modelling for Humans." (https://radimrehurek.com/gensim/intro.html, accessed February 6, 2024).

Riaz, M. Q., Butt, W. H., and Rehman, S. 2019. "Automatic Detection of Ambiguous

Software Requirements: An Insight," *5th International Conference on Information Management, ICIM 2019*, IEEE, pp. 1–6. (https://doi.org/10.1109/INFOMAN.2019.8714682).

Ribeiro, C., and Berry, D. 2020. "The Prevalence and Severity of Persistent Ambiguity in Software Requirements Specifications: Is a Special Effort Needed to Find Them?," *Science of Computer Programming* (195), Elsevier B.V., p. 102472. (https://doi.org/10.1016/j.scico.2020.102472).

Sonbol, R., Rebdawi, G., and Ghneim, N. 2022. "The Use of NLP-Based Text Representation Techniques to Support Requirement Engineering Tasks: A Systematic Mapping Review," *IEEE Access* (10), IEEE, pp. 62811–62830. (https://doi.org/10.1109/ACCESS.2022.3182372).

"Spacy.Io." (n.d.). (https://spacy.io/api/doc, accessed January 23, 2024).

"Technical Report: The AmbiTRUS Analysis Framework." 2024.

Tiwari, S., Rathore, S. S., Sagar, S., and Mirani, Y. 2020. "Identifying Use Case Elements from Textual Specification: A Preliminary Study," *Proceedings of the IEEE International Conference on Requirements Engineering* (2020-Augus), pp. 410–411. (https://doi.org/10.1109/RE48521.2020.00059).

Urbieta, M., Antonelli, L., Guerra, J., and Rossi, G. 2022. "Tracing User Stories and Source Code Using the Language Extended Lexicon," in *Information Systems and Technologies*, A. Rocha, H. Adeli, G. Dzemyda, and F. Moreira (eds.), Cham: Springer International Publishing, pp. 413–429.

Wautelet, Y., Heng, S., and Kolp, M. 2017. "Perspectives on User Story Based Visual Transformations," in *CEUR Workshop Proceedings* (Vol. 1796).

Wautelet, Y., Heng, S., Kolp, M., Mirbel, I., and Poelmans, S. 2016. "Building a Rationale Diagram for Evaluating User Story Sets," in *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–12.

Yan, X., Guo, J., Lan, Y., and Cheng, X. 2013. "A Biterm Topic Model for Short Texts," *WWW 2013 - Proceedings of the 22nd International Conference on World Wide Web*, pp. 1445–1455. (https://doi.org/10.1145/2488388.2488514).

Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E.-V., and Batista-Navarro, R. T. 2021. "Natural Language Processing for Requirements Engineering: A Systematic Mapping Study," *ACM Computing Surveys* (54:3), New York, NY, USA: Association for Computing Machinery. (https://doi.org/10.1145/3444689).