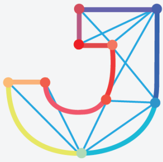
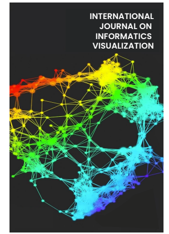
JOIV : Int. J. Inform. Visualization, 8(3-2): IT for Global Goals: Building a Sustainable Tomorrow - November 2024 1905-1915



INTERNATIONAL JOURNAL
ON INFORMATICS VISUALIZATION

journal homepage : www.joiv.org/index.php/joiv



Automated UML Class Diagram Generation from Textual Requirements Using NLP Techniques

Yang Meng [a], Ainita Ban [a]

[a] Department of Software Engineering and Information Systems, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Selangor, Malaysia

Corresponding author: *ainita@upm.edu.my

***Abstract*—Translating textual requirements into precise Unified Modeling Language (UML) class diagrams poses challenges due to the unstructured and often ambiguous nature of text, which can lead to inconsistencies and misunderstandings during the initial stages of software development. Current methods often struggle with effectively addressing these challenges due to limitations in handling diverse and complex textual requirements, which may result in incomplete or inaccurate UML diagrams. This study aims to propose a Natural Language Processing (NLP) model that analyzes and comprehends textual requirements to extract relevant information for generating UML class diagrams, ensuring accuracy and consistency between the diagrams and requirement descriptions. The research employs a four-step approach: preprocessing to handle text noise and redundancy, sentence classification to distinguish between "class" and "relationship" sentences, syntactic analysis to examine grammatical structures, and UML class diagram generation based on predefined rules. The results show that the model achieved a classification accuracy of 88.46% with a high Area Under the Curve (AUC) value of 0.9287, indicating robust performance in distinguishing between class definitions and relationships. This study highlights that existing methods may not fully address the nuances of translating complex textual requirements into accurate UML diagrams. This study successfully demonstrates an automated method for generating UML class diagrams from textual requirements and suggests that future research could expand datasets, optimize feature extraction, explore advanced models, and develop automated rule generation methods for further improvements.**

***Keywords*—Software engineering; UML class diagrams; Natural Language Processing (NLP); software development.**

Manuscript received 15 Apr. 2024; revised 9 Jul. 2024; accepted 12 Sep. 2024. Date of publication 30 Nov. 2024.
International Journal on Informatics Visualization is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

I. Introduction

Requirement gathering and the design phase are crucial stages in the software development lifecycle within the software development industry [1]. The duration of these phases significantly impacts the overall project timeline. Creating UML diagrams is a critical and time-intensive task spanning these software development phases. Diagrams, particularly class diagrams, bridge these phases [2]. Class diagrams are widely used in Object-Oriented analysis and design, forming the cornerstone from which other models are derived [3].

Translating textual requirements into precise UML class diagrams presents several challenges. Firstly, these requirements are often unstructured, with verbose, ambiguous, or incomplete descriptions, making it difficult to accurately extract information such as identifying classes, attributes, and their relationships [4]. Secondly, textual requirements can contain ambiguities, leading to inconsistencies and misunderstandings [5]. Manual analysis and mapping of textual requirements to UML class diagrams are time-consuming and error-prone, requiring significant time to interpret and understand the descriptions [6]. Existing tools have explored both semi-automatic and fully automated methods for generating UML class diagrams:

The semi-automated method refers to the need for human involvement or guidance, such as editing, validating, correcting, or supplementing the generated class diagrams.

The approaches presented in [7] propose an NLP-based framework to generate UML class diagrams from software requirements. It parses texts using a syntax parser and POS tagger, identifies linguistic elements, and employs semantic networks and word sense disambiguation to score and select candidate classes and relationships. The framework generates class diagrams and C# code templates. However, it has limitations with complex language structures and relies on

predefined semantic networks that may only be suitable for some domains. Utilized NLP techniques and heuristic rules but required users to adhere to grammar rules and necessitated manual intervention to validate and modify the generated class diagrams [8]. Proposed RAPID (Requirement analysis to Provide Instant Diagrams), a tool employing NLP and domain ontology techniques [9]. However, it might have limitations with complex sentence structures and require each sentence to conform to a specific structure. A method for extracting object-oriented elements through NLP was presented in [10]. However, refining class diagrams might necessitate developer involvement for complex problem statements.

The fully automated method refers to a process that doesn't require manual intervention or interaction. The approaches presented in [11] propose a system named Requirements Engineering Analysis and Design (READ) for generating UML class diagrams from informal natural language requirements using NLP and domain ontology techniques. The system preprocesses requirement text with sentence segmentation, tokenization, stop-word removal, stemming, and POS tagging using the NLTK library, then extracts UML concepts like class names, attributes, methods, and associations through heuristic rules. However, the method's rules may be incomplete, leading to potential errors in class name filtering. A study by [12] relied on syntactic parsing and GKP (Grammar Knowledge Patterns), using the Stanford parser for requirements. It needs help with complex language and parser accuracy dependency. In [13], AGUML was proposed for automated UML class diagram generation, utilizing text normalization, semantic analysis, and parsing. The system improves accuracy and reduces manual effort but lacks the adaptability to new language structures and detailed evaluation against existing tools. An NLP-based approach for automated UML class diagram generation from textual requirements, integrating text analysis and word vectorization, was proposed in [14]. This enhances efficiency and accuracy but may require further adaptation for complex texts. Table 1 evaluates prior research endeavors related to similar work in this domain.

TABLE I
EVALUATION OF EXISTING RESEARCH

Study	Input	Automation	Method / Technique used	Output and advantages	Limitations
[7]	Software Requirements Specification (SRS)	Semi-automatic	NLP + POS tagger	Produced UML class diagrams and C# code templates. The advantages lie in automated generation, enhancing efficiency in understanding requirements and generating code.	Need help to handle complex language structures and semantics, relying on pre-defined semantic networks and vocabulary.
[8]	NL textual requirements	Semi-automatic	NLP techniques + heuristic rules	Class diagram. The advantages lie in handling multiple elements and guiding users to standardize requirement documents, enhancing generation accuracy.	Requires adherence to syntax rules in document writing. Involves manual intervention for irrelevant class identification and diagram validation.
[9]	Informal NL requirements	Semi-automatic	NLP + domain ontology	Class diagram. Its strengths lie in extracting concepts using various technologies and offering an interactive interface.	Limits with complex sentence structures. Requires specific sentence structure compliance. It may restrict the accurate parsing of complex requirements. Limited applicability for intricate requirement documents.
[10]	NL problem statements	Semi-automatic	NLP + Relative extraction method + Dependency Graph	Automatic extraction of object-oriented elements from natural language text. Utilizes an intermediate representation (dependency graph). Conversion of dependency graph to UML class diagram. Simplified graphical representation allows user manipulation.	Accuracy reduction in complex problem statements. Utilizes an intermediate representation (dependency graph). - Developer involvement is needed for class diagram refinement.
[11]	Informal NL textual requirements	Automatic	NLP + domain ontology	Class diagram. The READ system features a user-friendly Tkinter interface, reduces over-generation issues by introducing strong and weak thresholds, and includes a refinement module to eliminate irrelevant elements.	The method's rules might be imperfect or unsuitable for all cases, potentially resulting in valid class names being incorrectly filtered out or invalid ones being mistakenly retained.
[12]	Informal NL textual requirements	Automatic	Syntactic dependency analysis + GKPs	UML class diagram's textual representation". Utilizes syntactic dependency analysis and grammatical knowledge patterns, avoiding the need to rewrite or annotate requirement statements and having no restrictions on input formats.	Depends on the accuracy of the parser, unable to handle anaphora resolution, ambiguity, and polysemy, cannot identify the multiplicity of relationships, and lacks direct generation of graphical class diagrams.
[13]	Informal NL textual requirements	Automatic	NLP + text normalization, syntactic and semantic analysis, parsing, information extraction	Class diagram. Reduces time and effort for manual creation. Improves accuracy of component recognition	Dependent on input text quality. Limited context understanding for complex scenarios. Relies on predefined rules

Study	Input	Automation	Method / Technique used	Output and advantages	Limitations
[14]	Textual requirements	Automatic	NLP	Class diagram. Increased software design efficiency by reducing manual effort. Enhanced quality and accuracy through automated consistency checks. Decreased human errors in diagram creation.	Dependency on input text quality and clarity. Challenges with complex domain-specific terminologies and ambiguous language structures.

Moreover, translating requirements into sequence and class diagrams also faces additional limitations [15]. The generation of sequence diagrams and class diagrams may encounter the following issues: the interpretation of requirements might be influenced by personal understanding, leading to inconsistencies in the models; complex business logic and system behaviors can be challenging to represent in sequence diagrams comprehensively; class diagrams and sequence diagrams have limitations in capturing dynamic system behaviors and interactions, which can affect the completeness and accuracy of the system design[16]. Therefore, existing methods often need help in handling these translations, impacting the accuracy and effectiveness of the generated models [17].

Despite advancements, existing methods require substantial manual involvement or face limitations with complex text and domain adaptability [18]. More effective approaches are needed that minimize human intervention while accurately generating UML class diagrams from textual requirements, handling ambiguities, and ensuring completeness [19].

The outlined objectives achieved throughout the research project encompass proposing an NLP model capable of analyzing and comprehending given textual requirements, extracting pertinent information related to software design such as classes, attributes, and methods. Implementing this NLP model to generate UML class diagrams that align with provided requirements automatically ensures consistency and accuracy between the diagrams and requirement descriptions. The research also aimed to validate the tool's accuracy and completeness in generating UML class diagrams through case studies or experimental validation.

The research will reduce the time and effort required to create UML class diagrams, improving accuracy and consistency in reflecting textual requirements. This has significant implications for enhancing efficiency in software design processes and addressing the gaps identified in existing methods.

II. Material and Method

A. General Framework

This study employs natural language processing techniques to automatically generate UML class diagrams from English text. The framework, illustrated in Fig. 1, includes four main steps: preprocessing, sentence classification, syntactic analysis, and UML class diagram generation. Through these steps, the study achieves the automatic conversion of natural language text into UML class diagram models, providing tool support for requirements modeling in software engineering and system design.

The goal of the preprocessing stage is to identify structured information from natural language text. The objective of the sentence classification stage is to distinguish between "class" and "relationship" types of sentences in the text, clarifying the type of each sentence so that specific parsing rules can be applied in subsequent steps. In the syntactic analysis stage, techniques such as part-of-speech tagging and dependency parsing are used to analyze sentences' grammatical and semantic information.

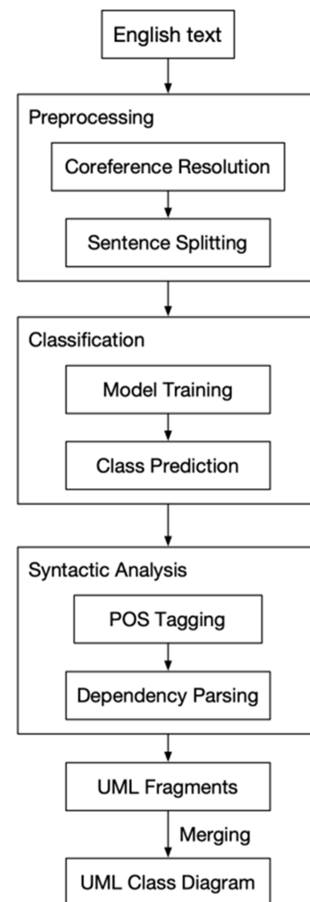


Fig. 1 Framework for Automatically Generating UML Class Diagrams from Textual Requirements Using Natural Language Processing Techniques

B. Preprocessing Stage

The English text is typically unstructured and may contain a lot of redundant, consistent, and clear information. To address these issues, this paper first preprocesses the English text, transforming the raw text into a more accessible format for analysis and processing. In this study, text preprocessing mainly includes two steps: coreference resolution and sentence segmentation.

Coreference Resolution: English text often contains numerous pronouns, nouns, and other lexical items that may refer to the same entity. While humans easily understand the referents of these terms in the context of surrounding sentences, computers often struggle to identify them. Therefore, the study performs coreference resolution in the input text to clarify the specific referents of pronouns and

nouns. This ensures that pronouns retain their meaning during sentence segmentation by correctly referencing their targets.

Sentence Segmentation: Sentence segmentation is breaking down text into individual sentences by identifying punctuation marks and specific sentence boundaries. This process effectively divides complex, multi-sentence texts into smaller units. It is crucial for text preprocessing in natural language processing, especially when dealing with intricate and multi-sentence inputs. Sentence segmentation reduces contextual dependencies, thereby enhancing the accuracy of subsequent parsing and processing tasks.

C. Binary Sentence Classification

The purpose of binary sentence classification is to categorize input natural language sentences into two types: "class definition" or "relationship description." In the dataset, sentences are labeled as either "class definition" or "relationship description," and our goal is to determine whether an input sentence belongs to the "class definition" category or the "relationship description" category.

D. Syntactic Analysis

The process of syntactic analysis includes two steps: part-of-speech tagging and dependency parsing. Part-of-speech tagging's primary task is to assign an appropriate grammatical tag to each word in a sentence. This process identifies the grammatical category of each word, including nouns, verbs, adjectives, etc., enhancing understanding of sentence structure and laying the groundwork for subsequent information extraction.

In software requirements text, UML class diagrams involve numerous dependency relationships. For instance, there are subject-verb relationships between classes and their own attributes and between classes and relationships. Dependency parsing helps us identify these elements and their relationships. The primary task of dependency parsing is to determine the dependency relationships between components within a sentence, such as subject-verb relationships (SBV), verb-object relationships (VOB), and coordination relationships (COO).

E. UML Class Diagram Generation

A series of class and relationship rules are defined to handle different categories of sentences to accurately extract the various components of a UML class diagram from natural language text. These rules extract the necessary structures for constructing the UML class diagram from syntactically analyzed text. By parsing the sentence structure, these rules identify classes, attributes, and their relationships, mapping these elements to the UML model to generate UML fragments.

After extracting multiple independent UML fragments from the text, including information such as classes, attributes, and relationships, the next step is concatenating these fragments into a complete UML class diagram. Issues like attribute and class name conflicts may arise during the fragment concatenation process. To ensure the consistency and correctness of the model, conflict detection and resolution are performed, addressing conflicts such as attribute names conflicting with class names or relationship names. When necessary, attributes or classes are renamed based on context to maintain model consistency and integrity.

F. Dataset Overview

To develop and evaluate the automated UML class diagram generation system, a suitable dataset was created for training and testing machine learning classifiers. This dataset was compiled from software requirement documents across various domains and complexities, including 600 high-quality UML class diagrams. From these, 100 diagrams were meticulously selected for detailed processing and broken down into independent class and relationship units. The dataset includes UML class diagrams from various domains, specifically healthcare, finance, and education. The healthcare domain encompasses electronic medical record systems and medical management systems; the finance domain includes banking management systems and securities trading systems; and the education domain involves learning management systems and online education platforms. In total, there are three different types of domains. This domain diversity helps validate the model's performance and robustness across different application scenarios.

After the steps above, the extracted and annotated dataset consists of 756 rows and three columns. Each row represents a UML fragment. Table II provides descriptions for columns in the dataset.

TABLE II
DATASET COLUMN DESCRIPTIONS

Column Name	Description	DataType
fragments_id	Unique identifier for the UML fragment, used to trace the original UML dataset information before segmentation.	String
English	Contains the English description of each UML fragment, explaining the content of the fragment.	String
kind	Indicates the category of the UML fragment, possible values include "class" (for classes) and "rel" (for relationships).	Enum ('class', 'rel')

For example, one of the Relationship UML fragments from the class diagram named GWPNV0 in our dataset (Fig. 2) has a fragments_id of 50, and its annotation results are shown in Table III.

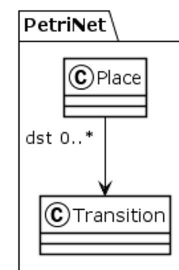


Fig. 2 Example of a UML Fragment Depicting a Relationship

TABLE III
ANNOTATION RESULTS FOR THE UML FRAGMENTS EXAMPLE

fragments_id	English	kind
50	In a Petri Net a Place may be the destination of a Transition	rel

The study first analyzed the distribution of categories for the dataset. The analysis results are shown in Fig. 3, indicating that the numbers of "rel" and "class" categories are similar, with no imbalance between categories.

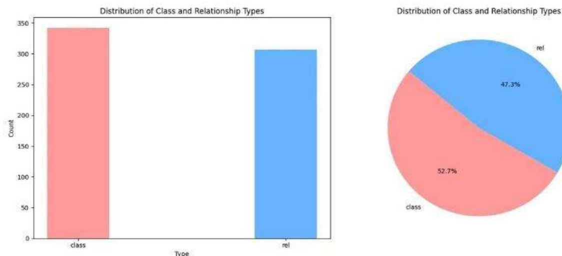


Fig. 3 Distribution of Class and Relationship Types

Furthermore, considering the presence of numerous long texts in the dataset and significant differences in text lengths between the two types, introducing more sophisticated feature extraction methods, such as TF-IDF, is being considered. This approach aims to better capture subtle relationships within the texts, enhancing the model's ability to process and learn from detailed textual information in the dataset.

G. Data Preprocessing

This study primarily utilized the Coreferee plugin from the spaCy library for coreference resolution. Coreferee is an advanced coreference resolution tool that utilizes pre-trained neural network models to identify and resolve coreference chains in text. Coreferee achieves an accuracy of 81% for general English text.

Firstly, the study used spaCy for the initial processing of the text. SpaCy can recognize entities and pronouns in the text, thereby establishing preliminary coreference relationships. Subsequently, the Coreferee plugin for coreference resolution takes over the processing. Building upon spaCy's recognition of entities and pronouns, Coreferee further analyzes the connections between these coreferences, weighing various factors such as semantic consistency and syntactic structure, to determine the most appropriate resolution strategy. Fig. 4 illustrates the process of Coreference Resolution.

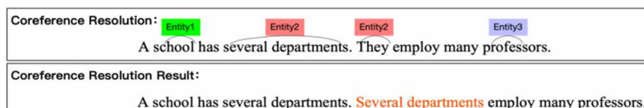


Fig. 4 The process of Coreference Resolution

After coreference resolution, the next step is sentence segmentation, which involves breaking down the text into independent sentences. In this study, the SpaCy library was utilized for sentence segmentation. Firstly, the study used SpaCy's language model to parse the text and generate a document object (doc) containing all tokens with their attributes. Each token in this document object includes its position in the original text, text content, and other linguistic features.

Next, the study iterated through this document object, processing each token sequentially and assigning it to its corresponding sentence based on specific rules. To handle substitutions and carry-over replacements across sentences, a substitutions dictionary and a carry-over dictionary were introduced. The substitutions dictionary stored tokens that needed replacement along with their corresponding replacement content, while the carry-over dictionary managed replacements that spanned across sentences.

During the iteration of the document object, if the current token was found in the substitutions dictionary, it was replaced with the corresponding content and added to the result of the current sentence. If the token was in the carry-over dictionary, it was replaced with the carry-over replacement content. Additionally, the study needed to check if the current token marked the end of a sentence (is_sent_end). If it did, it indicated that the current sentence processing was complete, and the study moved on to the next sentence.

Throughout this process, the study used a sentence identifier (sent_id) to track the current sentence being processed and stored the processed sentence content in a result dictionary (result). Fig. 5 depicts the process of sentence segmentation.

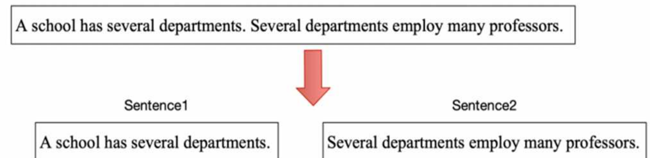


Fig. 5 Sentence segmentation

H. Design and Implementation of Classification Algorithms

Text feature extraction involves converting input text into numerical features that machine learning models can process. This forms the foundation for subsequent machine learning classification tasks. This study utilized two common feature extraction methods: TF-IDF (Term Frequency-Inverse Document Frequency) and Count Vectorization.

Firstly, the study utilized the Count Vectorization method, which converts text into a frequency matrix by calculating the occurrence frequency of each word in the text. Count Vectorization is straightforward and intuitive, effectively capturing basic information from the text. However, it may encounter information loss when dealing with high-frequency or low-frequency words. To overcome this drawback, the study also experimented with the TF-IDF method.

The TF-IDF (Term Frequency-Inverse Document Frequency) method measures the importance of each word by combining its term frequency (TF) and inverse document frequency (IDF). Specifically, term frequency (TF) indicates how frequently a word appears in a document, while inverse document frequency (IDF) represents the reciprocal of how often the word appears across all documents. By multiplying these two values together, TF-IDF reduces the weight of common words (high-frequency) and increases the weight of rare words (low-frequency), thereby better capturing key information in the text.

During the TF-IDF analysis process, the study extracted the most representative keywords for each category and calculated the average TF-IDF scores of these words within their respective categories. Additionally, word cloud visualizations were created to display these keywords intuitively. The word cloud's font sizes reflect each word's importance in its corresponding category; larger fonts indicate higher importance in that category. Fig. 6 and Fig. 7 are the word cloud visualizations generated for each category.

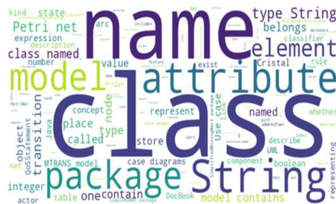


Fig. 6 Word Cloud for "Class"

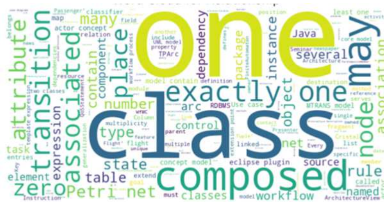


Fig. 7 Word Cloud for "rel"

After completing text feature extraction, the study conducted experiments using multiple machine learning classification algorithms to find the best algorithm for binary classification of input text into "class" and "relat". The following classic classification algorithms were selected for comparison and analysis: Bernoulli Naive Bayes, Multinomial Naive Bayes, k-Nearest Neighbors, Linear SVC, SVC (Support Vector Classifier), Gaussian Naive Bayes, AdaBoost, Random Forest, and Logistic Regression.

To ensure the reliability and scientific rigor of the experimental results, the study utilized the `train_test_split` function from the `sklearn.model_selection` library to split the dataset into training and testing sets with an 80:20 ratio.

During the model training stage, various classifiers from the sklearn library, including naive_bayes, neighbors, gaussian_process, and others, were utilized to train the data. The applied machine learning methods included Bernoulli Naive Bayes, Multinomial Naive Bayes, k-Nearest Neighbors, Linear SVC, SVC, Gaussian Process, AdaBoost, Random Forest, and Logistic Regression.

After completing model training, a test function was defined to evaluate the model's performance. This function uses the trained model to make predictions on the test set and generates a classification report using the classification_report function from the sklearn.metrics library. The report includes metrics such as accuracy, precision, recall, and F1-score, which comprehensively assess the model's classification performance.

The TF-IDF vectorized Bernoulli Naive Bayes model was chosen as the final classifier. The Bernoulli Naive Bayes model demonstrated high accuracy, and due to its simple structure and fast execution speed, it showed stable performance across different training experiments and text datasets, with satisfactory accuracy.

I. Syntactic Analysis

This study utilized the en_core_web_sm model from the spaCy library for Part-of-Speech Tagging (POS Tagging). This is a small English language pre-trained model widely used in natural language processing, particularly for dependency parsing tasks. POS tagging is automatically conducted when performing dependency parsing with the en_core_web_sm model in spaCy. The model serves as a full-fledged language processing pipeline, initially assigning POS tags to each word in the text, which are then utilized for dependency parsing. This means that before constructing the syntactic dependency tree, each word has already been assigned a POS tag, such as a noun, verb, adjective, etc.

Here is a specific example: for the text "A school has several departments.", the Part-of-Speech (POS) tagging results are as shown in Fig. 8 below:

```
A: DET
school: NOUN
has: VERB
several: ADJ
departments: NOUN
.: PUNCT
```

Fig. 8 Part-of-Speech Tagging

Building upon its part-of-speech tagging capabilities, this model analyzes the syntactic dependencies between words to generate a dependency tree. This tree clarifies the hierarchical relationships among the components of a sentence, such as subject-verb relationships, verb-object relationships, relative clauses, and more.

Specifically, the process of dependency parsing involves the following steps:

1) Load the Model: Load the pretrained en_core_web_sm model using the spaCy library.

2) Parse the Text: Pass the input text data to the loaded model. The model processes the text, performing tokenization and part-of-speech tagging (POS tagging).

3) Dependency Relation Identification: Analyze the syntactic dependency relationships between words in the sentence. Dependency relations help determine which words modify others (such as adjectives modifying nouns) and which words serve as objects of verbs.

4) Construct Dependency Tree: Based on the identified dependency relationships, construct the dependency tree of the sentence. This tree structure is crucial for subsequent relationship extraction and UML class diagram generation.

Fig. 9 depicts the dependency parsing results for the text "A school has several departments."

```
A -> det -> school
school -> nsubj -> has
has -> ROOT -> has
several -> amod -> departments
departments -> dobj -> has
. -> punct -> has
```

Fig. 9 Dependency Parsing Results for "A school has several departments."

Fig. 10 shows the dependency tree for the sentence "A school has several departments."

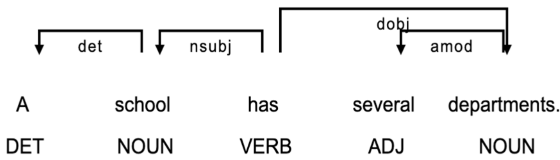


Fig. 10 Dependency Tree for "A school has several departments."

J. Definition and Application of UML Rules

This project uses a rule-based approach to extract elements of UML models from natural language text [20]. The core of this method involves parsing and transforming textual descriptions using predefined semantic rules to generate corresponding UML structures [21]. Nine class rules and six relationship rules have been defined based on standard software engineering terminology and expressions [22].

The following nine class rules have been defined.

1) Rule 1: Simple Copula. Pattern: "The ... is a class ...". Functionality: Matches simple sentences where the subject is followed by the verb "be" and the noun "class", extracting the subject as the class name. Example: Input: "A class named Job." Output: Generates a class named "Job"[23].

2) Rule 2: There is or Exists. Pattern: "There is a class called ...". Functionality: Handles sentences that start with "There is" or "There exists", indicating the existence of a class. Example: Input: "There is a class called Employee." Output: Generates a class named "Employee".

3) Rule 3: Compound. Pattern: "Employee has a Job.". Functionality: Handles sentences containing compound nouns, typically not directly used for generating UML, and requires further contextual understanding. Example: Input: "Employee has a Job." Output: This may not directly generate UML as it requires a contextual understanding of relationships [24].

4) Rule 4: Compound Class Explicit. Pattern: "The Job class has a title.". Functionality: Explicitly identifies a class and describes its attributes. Example: Input: "The Job class has a title." Output: Generates a class named "Job" with an attribute named "title".

5) Rule 5: To Have. Pattern: "An Employee has a Job.". Functionality: Parses sentences expressing ownership relationships, extracting the subject as a class and the object as an attribute. Example: Input: "An Employee has a Job." Output: Generates a class named "Employee" with an attribute named "Job" of unspecified type [25].

6) Rule 6: Class Named. Pattern: "The class named Job.". Functionality: Directly specifies the class name. Example: Input: "The class named Job." Output: Generates a class named "Job"[26].

7) Rule 7: Component of Package. Pattern: "Job is a component of the Employee package.". Functionality: Parses sentences indicating that a component belongs to a specific package. Example: Input: "Job is a component of the Employee package." Output: Generates a class named "Job" and indicates it is part of the "Employee package".

8) Rule 8: 3 Component and Clause. Pattern: "Employee has a Job and a Salary and a Department." Functionality: Parses sentences involving multiple components and clauses. Example: Input: "Employee has a Job and a Salary and a Department." Output: Generates a class named "Employee" with properties "Job", "Salary", and "Department"[27].

9) Rule 9: 2 Component and Clause. Pattern: "Employee has a Job and a Department." Functionality: Parses sentences involving two components and clauses. Example: Input: "Employee has a Job and a Department." Output: Generates a class named "Employee" with properties "Job" and "Department".

The following six relationship rules have been defined.

1) Rule 1: To Have Multiplicity. Pattern: "An Employee can have multiple Jobs." Function: Parses relationships indicating that a class can have multiple instances and specifies the quantity relationship. Example: Input: "An Employee can have multiple Jobs." Output: Establishes a relationship between the "Employee" class and the "Jobs" class, specifying that there can be multiple instances of "Jobs"[28].

2) Rule 2: Passive Voice. Pattern: "The Employee performs the Job." Function: Handles sentences in passive voice to determine the subject and object of an action. Example: Input: "The Employee performs the Job." Output: Establishes a relationship named "performs" between "Employee" and "Job"[29].

3) Rule 3: Composed. Pattern: "A Department is composed of many Employees." Function: Parses composition relationships, expressing that one class is composed of multiple instances of another class. Example: Input: "A Department is composed of many Employees." Output: Establishes a composition relationship between "Department" and "Employees".

4) Rule 4: Active Voice. Pattern: "The Employee performs a Job." Function: Handles sentences in active voice to determine the subject and action. Example: Input: "The Employee performs a Job." Output: Establishes a relationship named "performs" between "Employee" and "Job".

5) Rule 5: Noun With. Pattern: "An Employee with a Job." Function: Parses structures containing "with" to indicate an additional or inclusive relationship. Example: Input: "An Employee with a Job." Output: Establishes an associative relationship between "Employee" and "Job"[30].

6) Rule 6: Copula Rel. Pattern: "The Job is part of the Employee's responsibilities." Function: Parses structures like "is part of," determining ownership or composition relationships. Example: Input: "The Job is part of the Employee's responsibilities." Output: Establishes a "part of" relationship between "Job" and "Employee's responsibilities".

K. Concatenation of UML Fragments

UML fragment concatenation combines multiple independent UML classes and relationship fragments into a complete UML class diagram. This study adopts a greedy strategy to merge UML fragments into the evolving UML model sequentially. The most suitable fragment is selected for integration at each merger step until all fragments are merged. Conflicts such as attributes and class name conflicts may arise

during the merging process. Conflict detection and resolution are necessary to ensure the consistency and correctness of the final model. Common conflicts include between attribute and class names, as well as between attribute and relationship names. In such cases, attributes or classes may need to be renamed based on the context to maintain model consistency. The specific merging and conflict resolution strategies are as follows:

1) Merging Classes: Each class from every fragment is sequentially added to the UML model. If classes with the same name are encountered, their attributes and methods are merged.

2) Merging Relationships: Relationships from each fragment are sequentially added to the UML model. If identical relationships are found, they are skipped; otherwise, new relationships are added.

3) Conflict Resolution: During the merging process, conflicts involving attribute names, class names, and relationship names are detected and addressed in real-time. Conflicts are resolved through renaming or merging actions.

Fig. 11 provides the PlantUML syntax description of the merged result of the two sentences. Fig. 12 shows the translation of the PlantUML syntax into a UML class diagram.

```
@startuml
!theme plain
class School {
  departments
}
class Departments {
  professors
}
School "1" -- "1..*" Departments : contains >
Departments "1..*" -- "0..*" Professors : employs >
@enduml
```

Fig. 11 PlantUML syntax description of the merged result of the two sentences.

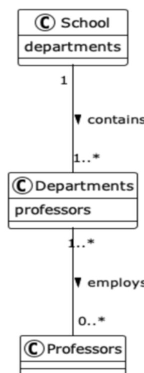


Fig. 12 Translation of PlantUML syntax into a UML class diagram.

III. Results and Discussion

A. Experimental Validation

This chapter outlines the experimental validation and analysis phase. Through comprehensive testing and validation of the model, its accuracy and completeness in generating UML class diagrams are confirmed. Accuracy refers to the consistency between the automatically generated UML class diagrams and the content of the original textual requirements. It measures whether the generated diagrams correctly reflect the classes, attributes, methods, and relationships described in the requirements. Completeness refers to the extent to which the automatically generated UML class diagrams cover all relevant information in the textual requirements. It evaluates whether the diagrams include all the classes, attributes, methods, and relationships explicitly or implicitly indicated in the requirements.

The TF-IDF Bernoulli Naive Bayes model will be optimized and analyzed. Additional data will be annotated to evaluate the model's performance, and the trained model will be used for classification to observe its effectiveness and performance. Using an automatic tool, a comprehensive test and analysis were conducted by manually annotating an additional 130 data entries. These entries included 64 class definitions and 66 relationship descriptions, maintaining consistency with the original dataset distribution to validate the model in a similar environment. Subsequently, UML class diagrams were generated using the tool.

Three specialists, each with extensive industry experience and academic backgrounds, were invited to evaluate the generated UML class diagrams for expert review. They come from software engineering, system design, and natural language processing and compared these diagrams with those manually created by themselves.

Two parameters can be tuned to explore the optimal performance of the Bernoulli Naive Bayes model: Alpha (α) and Binarize.

Alpha (α): This is the smoothing parameter used to prevent overfitting. The default value is 1.0, and different values, such as 0.1, 0.5, 1, 2, 10, etc., can be tried to determine the best performance.

Binarize: This parameter sets the threshold for binarizing input values. If set to None, no binarization is performed.

The relationship between the Alpha parameter and accuracy is depicted in Fig. 13.

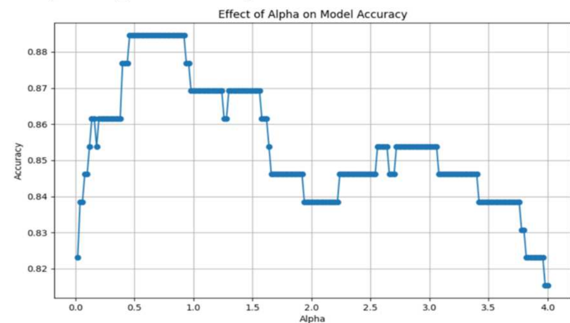


Fig. 13 The relationship between the Alpha parameter and model accuracy

The model achieves the highest accuracy of 88.66% when the Alpha parameter is set between 0.46 and 0.9. The classification report for Alpha values of 0.5 and 1 is shown in Fig. 14 and Fig. 15. With Alpha set to 0.5, the accuracy improves by 1.35% compared to the default Alpha of 1, reaching 87.31%. Additionally, the model shows improvements in recall and f1-score values for both categories.

```
              precision    recall  f1-score   support

       class     0.8551    0.9219    0.8872        64
         rel     0.9180    0.8485    0.8819        66

    accuracy                         0.8846       130
   macro avg     0.8866    0.8852    0.8846       130
weighted avg     0.8870    0.8846    0.8845       130
```

Fig. 14 Classification_report for Default Alpha=0.5

```
              precision    recall  f1-score   support

       class     0.8310    0.9219    0.8741        64
         rel     0.9153    0.8182    0.8640        66

    accuracy                         0.8692       130
   macro avg     0.8731    0.8700    0.8690       130
weighted avg     0.8738    0.8692    0.8690       130
```

Fig. 15 Classification_report for Optimized Alpha=1

The relationship between the Binarize parameter and accuracy is depicted in Fig. 16.

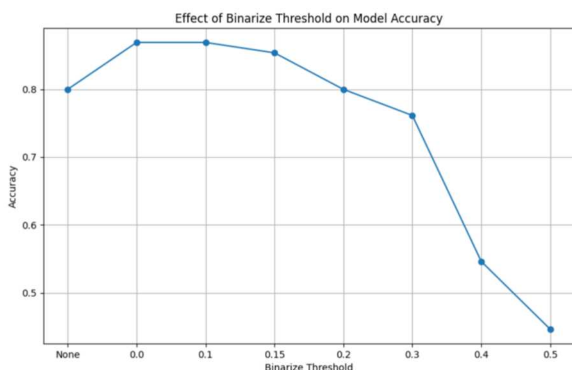


Fig. 16 The Relationship between the Binarize Parameter and Accuracy

As observed, the model achieves the highest classification accuracy with the default Binarize value of 0 and reaches optimal performance with this setting, indicating no need for parameter adjustment.

B. Result Analysis

Fig. 17 and 18 show that the optimized parameters Alpha=0.5 and Binarize=0.0 were selected for generating heatmaps and ROC curves. The heatmap results indicate that out of 64 samples labeled as "class," the model correctly classified 59 and misclassified 5. For the 66 samples labeled as "rel," the model correctly classified 56 and misclassified 10. This suggests that the model accurately distinguishes between "class" and "rel." However, the classification performance of "rel" samples is relatively poorer than that of "class" samples.

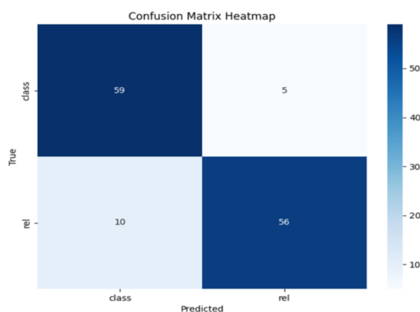


Fig. 17 Confusion Matrix Heatmap

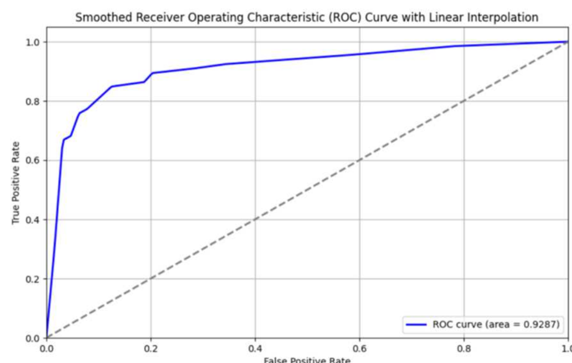


Fig. 18 Smoothed ROC Curve with Linear Interpolation

The ROC curve shows that with parameters Alpha=0.5 and Binarize=0.0, the model achieves an AUC value of 0.9287, close to 1. This indicates that the model can distinguish between positive and negative classes. Additionally, the curve is positioned near the top-left corner, suggesting a high True Positive Rate (TPR) and low False Positive Rate (FPR), further confirming the excellent classification performance of the model.

Accuracy Test Results: The overall accuracy indicates the model's general performance in class definition and relationship description tasks. High accuracy validates the model’s stability and effectiveness in handling various types of textual data, though it may have limitations with specific requirements, such as complex relationship descriptions. Table IV displays the detailed accuracy test results.

TABLE IV
ACCURACY TEST RESULTS

Data Type	**Number of Correctly Classified**	**Total Number of Classifications**	**Accuracy**
Class Definitions	59	64	92.19%
Relationship Descriptions	56	66	84.85%
Overall Accuracy	115	130	88.46%

Completeness Test Results: The overall requirement coverage rate demonstrates the tool's capability to encompass most requirements during the processing and generation of UML class diagrams, showcasing its strong performance in maintaining completeness. Refer to Table V for detailed completeness test results.

TABLE V
COMPLETENESS TEST RESULTS

Data Type	**Total Number of Requirements**	**Number of Covered Requirements**	**Requirement Coverage Rate**
Class Definitions	64	60	93.75%
Relationship Descriptions	66	61	92.42%
Overall	130	121	93.08%

IV. CONCLUSION

This study achieved the complete process of automatically generating UML class diagrams from English text using natural language processing techniques. Text preprocessing: Structured information was extracted from unstructured text

through coreference resolution and sentence segmentation. Coreference resolution improved the accuracy of identifying referents for pronouns and noun phrases, while sentence segmentation helped break down the complex text into smaller units for easier subsequent processing.

Sentence Classification: Using a machine learning classifier, sentences were classified into "class definition" and "relationship description." Through feature extraction and vectorization, the model accurately categorized input sentences. Test results indicated that the combination of TF-IDF vectorization and the Bernoulli Bayes classifier performed the best, achieving a classification accuracy of 88.66%.

Syntactic Analysis: Sentence syntax and semantic structure were identified and analyzed through part-of-speech tagging and dependency parsing. Part-of-speech tagging helped determine each word's grammatical category, while dependency parsing further identified the syntactic relationships between words, laying the foundation for generating UML fragments.

UML Class Diagram Generation: A series of class and relationship rules were defined and applied to map semantic information to UML model elements, successfully generating UML fragments. Subsequently, these fragments were merged using a combination algorithm to resolve conflicts such as attribute and class name clashes, creating a complete UML class diagram.

Experimental Validation: Optimizing the Bernoulli Bayes model further enhanced the classification performance. With optimal parameter settings, the model demonstrated outstanding accuracy and AUC values. Analysis of heatmaps and ROC curves indicated that the model exhibited high accuracy and robustness in distinguishing between "class definition" and "relationship description" sentences.

Future research could consider introducing more complex and powerful models, such as neural networks and deep learning models, for text classification and syntactic analysis, aiming to enhance the model's generalization capability and adaptability. Additionally, exploring data-driven approaches that combine supervised and unsupervised learning could automate the generation and optimization of rules, thereby improving the accuracy and robustness of UML class diagram generation.

References

[1] Y. Rigou and I. Khriss, “A Deep Learning Approach to UML Class Diagrams Discovery from Textual Specifications of Software Systems,” 2023, pp. 706–725. doi: 10.1007/978-3-031-16078-3_49.

[2] M. Jahan, Z. S. H. Abad, and B. Far, “Generating Sequence Diagram from Natural Language Requirements,” in *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, IEEE, Sep. 2021, pp. 39–48. doi:10.1109/rew53955.2021.00012.

[3] O. S. Dawood Omer and S. Eltyeb, “Towards an Automatic Generation of UML Class Diagrams from Textual Requirements using Case-based Reasoning Approach,” in *2022 4th International Conference on Applied Automation and Industrial Diagnostics (ICAAID)*, IEEE, Mar. 2022, pp. 1–5. doi: 10.1109/icaaid51067.2022.9799502.

[4] M. A. Ahmed, I. Ahsan, U. Qamar, and W. H. Butt, “A Novel Natural Language Processing approach to automatically Visualize Entity-Relationship Model from Initial Software Requirements,” in *2021 International Conference on Communication Technologies (ComTech)*, IEEE, Sep. 2021, pp. 39–43. doi:10.1109/ComTech52583.2021.9616949.

[5] A. Abdalazeim and F. Meziane, “A review of the generation of requirements specification in natural language using objects UML models and domain ontology,” *Procedia Comput Sci*, vol. 189, pp. 328–334, 2021, doi: 10.1016/j.procs.2021.05.102.

[6] J. Shivamurthy, T. Uppal, and D. Vidyarthi, “NLP-based Auto Generation of Graph Database from Textual Requirements,” in *2024 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, IEEE, Jul. 2024, pp. 1–6. doi: 10.1109/CONECCT62155.2024.10677144.

[7] Fatma Alharbia, Shadi R .Masadeh, and Faiz Alshrouf, “A Framework for the Generation of Class Diagram from Text Requirements using Natural Language Processing,” *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 10, no. 1, pp. 25–31, Feb. 2021, doi: 10.30534/ijatcse/2021/041012021.

[8] E. A. Abdelnabi, A. M. Maatuk, T. M. Abdelaziz, and S. M. Elakeili, “Generating UML Class Diagram using NLP Techniques and Heuristic Rules,” in *2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, IEEE, Dec. 2020, pp. 277–282. doi:10.1109/STA50679.2020.9329301.

[9] P. More and R. Phalnikar, “Generating UML Diagrams from Natural Language Specifications,” *Int J Appl Inf Syst*, vol. 1, no. 8, pp. 19–23, Apr. 2012, doi: 10.5120/ijais12-450222.

[10] H. Krishnan and P. Samuel, “Relative Extraction Methodology for class diagram generation using dependency graph,” in *2010 International Conference on Communication Control and Computing Technologies*, IEEE, Oct. 2010, pp. 815–820. doi:10.1109/ICCCCT.2010.5670730.

[11] N. Bashir, M. Bilal, M. Liaqat, M. Marjani, N. Malik, and M. Ali, “Modeling Class Diagram using NLP in Object-Oriented Designing,” in *2021 National Computing Colleges Conference (NCCC)*, IEEE, Mar. 2021, pp. 1–6. doi: 10.1109/nccc49330.2021.9428817.

[12] R. Sharma, P. K. Srivastava, and K. K. Biswas, “From natural language requirements to UML class diagrams,” in *2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, IEEE, Aug. 2015, pp. 1–8. doi:10.1109/aire.2015.7337625.

[13] A. Gupta, G. Poels, and P. Bera, “Generating multiple conceptual models from behavior-driven development scenarios,” *Data Knowl Eng*, vol. 145, p. 102141, May 2023, doi:10.1016/j.datak.2023.102141.

[14] S. Yang and H. Sahraoui, “Towards automatically extracting UML class diagrams from natural language specifications,” in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, New York, NY, USA: ACM, Oct. 2022, pp. 396–403. doi: 10.1145/3550356.3561592.

[15] Z. Babaalla, E. M. Bouziane, A. Jakimi, and M. Oualla, “From text-based system specifications to UML diagrams: A bridge between words and models,” in *2024 International Conference on Circuit, Systems and Communication (ICCSC)*, IEEE, Jun. 2024, pp. 1–6. doi:10.1109/iccsc62074.2024.10616686.

[16] A. Ferrari, S. Abualhaija and C. Arora, "Model Generation with LLMs: From Requirements to UML Sequence Diagrams," in 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW), Reykjavik, Iceland, 2024, pp. 291-300, doi:10.1109/rew61692.2024.00044.

[17] E. A. Abdelnabi, A. M. Maatuk, and M. Hagal, “Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques,” in *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, IEEE, May 2021, pp. 288–293. doi: 10.1109/mi-sta52233.2021.9464433.

[18] Z. Babaalla, A. Jakimi, M. Oualla, R. Saadane, and A. Chehri, “Towards an Automatic Extracting UML Class Diagram from System’s Textual Specification,” in *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security*, New York, NY, USA: ACM, Apr. 2024, pp. 1–5. doi:10.1145/3659677.3659742.

[19] Z. Babaalla, H. Abdelmalek, A. Jakimi, and M. Oualla, “Extraction of UML class diagrams using deep learning: Comparative study and critical analysis,” *Procedia Comput Sci*, vol. 236, pp. 452–459, 2024, doi: 10.1016/j.procs.2024.05.053.

[20] M. A. Umar and K. Lano, “Advances in automated support for requirements engineering: a systematic literature review,” *Requir Eng*, vol. 29, no. 2, pp. 177–207, Jun. 2024, doi: 10.1007/s00766-023-00411-0.

[21] S. Zhong, A. Scarinci, and A. Cicirello, “Natural Language Processing for systems engineering: Automatic generation of Systems Modelling Language diagrams,” *Knowl Based Syst*, vol. 259, p. 110071, Jan. 2023, doi: 10.1016/j.knosys.2022.110071.

[22] S. M. Cheema, S. Tariq, and I. M. Pires, “A natural language interface for automatic generation of data flow diagram using web extraction techniques,” *Journal of King Saud University - Computer and Information Sciences*, vol. 35, no. 2, pp. 626–640, Feb. 2023, doi:10.1016/j.jksuci.2023.01.006.

[23] S. Kumar, Aryaman, Aryan, and D. Yadav, “Natural Language Processing based Automatic Making of Use Case Diagram,” in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*, IEEE, Aug. 2023, pp. 1026–1032. doi:10.1109/icirca57980.2023.10220849.

[24] A. A. Almazroi, L. Abualigah, M. A. Alqarni, E. H. Houssein, A. Q. M. AlHamad, and M. A. Elaziz, “Class Diagram Generation from Text Requirements: An Application of Natural Language Processing,” 2021, pp. 55–79. doi: 10.1007/978-3-030-79778-2_4.

[25] A. Akundi, J. Ontiveros, and S. Luna, “Text-to-Model Transformation: Natural Language-Based Model Generation Framework,” *Systems*, vol. 12, no. 9, p. 369, Sep. 2024, doi: 10.3390/systems12090369.

[26] D. Peral-García, J. Cruz-Benito, and F. J. García-Peñalvo, “Using Quantum Natural Language Processing for Sentiment Classification and Next-Word Prediction in Sentences Without Fixed Syntactic Structure,” 2024, pp. 235–243. doi: 10.1007/978-3-031-48981-5_19.

[27] J. Chen, B. Hu, W. Diao, and Y. Huang, “Automatic generation of SysML requirement models based on Chinese natural language requirements,” in *Proceedings of the 2022 6th International Conference on Electronic Information Technology and Computer Engineering*, New York, NY, USA: ACM, Oct. 2022, pp. 242–248. doi: 10.1145/3573428.3573470.

[28] R. Bougacha, R. Laleau, S. Collart-Dutilleul, and R. Ben Ayed, “Extending SysML with Refinement and Decomposition Mechanisms to Generate Event-B Specifications,” 2022, pp. 256–273. doi:10.1007/978-3-031-10363-6_18.

[29] R. Saini, G. Mussbacher, J. L. C. Guo, and J. Kienzle, “Automated, interactive, and traceable domain modelling empowered by artificial intelligence,” *Softw Syst Model*, vol. 21, no. 3, pp. 1015–1045, Jun. 2022, doi: 10.1007/s10270-021-00942-6.

[30] V. Danylyk, V. Lytvyn, and S. Mushasta, “Information system of identification of terms and abbreviations in text documents,” *Herald of Khmelnytskyi National University. Technical sciences*, vol. 319, no. 2, pp. 81–87, Apr. 2023, doi: 10.31891/2307-5732-2023-319-1-81-83.