Journal of Theoretical and Applied Information Technology
31st March 2025. Vol.103. No.6

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A STRUCTURED TRACEABILITY APPROACH FOR TRANSFORMING REQUIREMENTS INTO CLASS DIAGRAMS

KELETSO J. LETSHOLO
Faculty of Computer Information Science, Higher Colleges of Technology, United Arab Emirates
E-mail: kletsholo@hct.ac.ae

ABSTRACT

In software engineering, requirement traceability is a critical factor in ensuring that software systems comply with user requirements, enabling effective management and verification throughout the development lifecycle. However, due to inherent ambiguities and inconsistencies in user-specified requirements, typically expressed in natural language, establishing accurate traceability remains a significant challenge. This study introduces a novel, automated approach to requirement traceability, addressing this challenge by transforming natural language requirement specifications (NLRs) into class diagrams while simultaneously generating traceability links. The proposed approach leverages Natural Language Processing (NLP) techniques and Semantic Object Models (SOMs), a structured and reusable pattern-based method that enhances the accuracy and consistency of traceability links. To validate this approach, the TRAM application was developed and evaluated against manually produced class diagrams from requirements engineering experts. Precision and recall metrics were used to assess the accuracy and completeness of the generated traceability links. Results indicate that TRAM achieves high recall (0.60–0.96), demonstrating its effectiveness in capturing relevant elements, although precision (0.25–0.51) remains a challenge due to the integration of predefined elements through SOM patterns. These findings highlight the contribution of this research in advancing automated traceability solutions, reducing manual effort, and improving consistency in requirements engineering. Additionally, the structured use of SOMs suggests that this methodology can be extended beyond class diagrams to other software artifacts, broadening its applicability in software engineering.

Keywords: *Class Diagram; Natural Language Processing; Requirements Engineering; Requirements Traceability; Semantic Object Model.*

1. INTRODUCTION

The requirements of a software system describe what the system should accomplish, the services it must provide, and the restrictions on its operation [1]. These requirements are derived from users seeking a system that meets specific needs and are usually specified in natural language. Natural language is highly expressive and intuitive for users, but it often introduces ambiguities and inconsistencies that can complicate the subsequent stages of the development process. Requirement traceability is crucial in this context, as it provides a systematic way to link and manage the relationships between the specification of natural language requirements (NLR) and its software artifacts.

Requirement traceability refers to *"the ability to describe and follow the life of a requirement, both in a forward and backward direction"* [2]. Requirement forward traceability refers to the ability to track requirements components through various stages, including analysis, design, or implementation. The backward traceability of requirements is the ability to trace the model elements back to their original NLRs. A comprehensive approach to ensure traceability, especially for complex computer-based systems, requires linking all system components back to NLRs. These components include hardware, software, human resources, manuals, policies, and procedures. To accomplish this goal, it is crucial to maintain traceability throughout all phases of the system development process, starting from the requirements specified by the user, through the analysis, design, implementation, and testing of the final product. Various stakeholders can use this traceability information to show that requirements have been met, validate the reasoning behind design choices, and set up change control and maintenance procedures. By establishing and maintaining these connections, traceability helps identify and resolve

ambiguities early, ensures alignment throughout the development life cycle [3], [4], [5], and supports accurate and reliable transformation of requirements into software artifacts [2], [6], [7]. This reduces errors, improves consistency, and ultimately leads to the development of software systems that satisfy user requirements. In addition, traceability promotes the simultaneous development of requirements and software, which in turn reduces the time and expenses associated with software maintenance and growth. [8], [9]. Despite progress in natural language processing (NLP) and model transformation methods, the semantic gap between NLRs and software artifacts continues to pose difficulties for requirement transformation techniques, resulting in lower accuracy of traceability links. This gap arises mainly from the intrinsic flexibility and ambiguity of the natural language, which contrasts starkly with the formal structure demanded by software artifacts.

The primary objective of this study is to develop a structured method for the automatic creation of class diagrams from NLRs, while also ensuring the establishment and preservation of requirements traceability connections. Building on the previous work established during the doctoral research of the author [10], this work addresses the ongoing issue of ensuring the accuracy and completeness of the derived software artifacts. This method employs NLP techniques for data preprocessing along with a collection of reusable patterns called Semantic Object Models (SOMs), which bridge the semantic gap between unstructured NLR and class diagrams. These patterns impose both a semantic and a structural framework on NLR, enabling a more structured approach to model construction and improving the accuracy of trace link generation. This study focuses on class diagrams for two main reasons: they document key elements in requirements engineering, namely, classes, attributes, and relationships, and are a foundational analysis tool used in numerous software development methodologies [11], [12]. Furthermore, the use of SOM patterns enables the generation of other software artifacts. This adaptability underscores the flexibility of the approach, allowing it to extend beyond class diagrams and accommodate various aspects of software modeling, thus increasing its applicability across different methodologies. This approach was implemented using a software tool called Textual Requirements to Analysis Models (TRAM). TRAM was first introduced in [13], focusing mainly on the transformation process involved in the TRAM method. This study presents an evaluation framework that compares class diagrams generated with those created by professionals in requirements engineering. By offering structured translation and evaluation, this study contributes significantly to the domain of natural language processing in requirements engineering, providing valuable solutions and insights that enhance both the precision and effectiveness of early system design activities.

The remainder of this document is structured as follows. Section 2 reviews the existing literature on requirement traceability and highlights research gaps. Section 3 explores the underlying theories and concepts that support the TRAM approach. Section 4 provides a high-level description of the TRAM components. Sections 5, 6, and 7 focus on the specific components of the TRAM approach and detail their roles and functionalities. Section 8 presents the method and results to evaluate the accuracy and completeness of the class diagrams produced by TRAM. Finally, Section 9 summarizes the findings and discusses future research and development directions.

2. RELATED WORK

Research on requirement traceability has been ongoing for more than four decades. One of the first notable tools in this area was reported in 1978 by Robert Pierce [14]. Since then, requirement traceability has become integral to software engineering, particularly requirements engineering. The previous research of the author [13] explored the initial aspects of the transformation process of the TRAM approach. This study expands on this foundation by integrating traceability link establishment and introducing detailed transformation rules, elements that have not been addressed earlier.

2.1 Traceability Approaches of Traditional Requirements

Spanoudakis et al. [6] developed a rule-based linguistic approach to establish traceability connections between use case specifications and UML analysis models. Building on this foundation, Jirapanthong and Zisman [15] introduced a reference model for traceability along with a rule-based method to link documents focused on features in the engineering of the product line. Almeida et al. [16] offered a systematic framework to associate requirements with multiple artifacts during the model-driven design process. Although

this framework is valuable for understanding the traceability of requirements in model-driven development, the proposed metamodel has a high level of abstraction and lacks representation of lower-level requirements and model elements. Cleland-Huang et al. [17] proposed an event notification-based traceability method suitable for heterogeneous and globally distributed development environments. This approach maintains a loose coupling between requirements and other artifacts by using an event service to inform stakeholders of any changes. A limitation, as noted in [6], is its dependence on preidentified relations, which lack support for the initial identification.

2.2 Tools for Requirements Traceability

The landscape of tools that support the traceability of requirements is diverse and includes both user-driven and system-defined approaches. Some tools enable users to establish traceability links based on their personal experience and intuition, although they often lack the ability to define the semantics of these links declaratively. An excellent example is IBM Rational DOORS [18], which is a platform tailored to monitor requirements and ensure traceability in the realms of complex systems and software development. The REquirements TRacing On-target (RETRO) tool [19] facilitates the automatic creation of requirement traceability matrices, which play a crucial role in the maintenance of software systems. It achieves traceability recovery for artifacts characterized by unstructured text, employing information retrieval and text mining methods to generate potential trace links. DesignTrack [20] is a prototype tool that supports requirement traceability (RT) by linking requirements with architectural design. It serves organizations by providing a unified environment for both requirement modeling and specification. This tool allows architects to effectively manage design requirements and related tasks. The object-oriented requirements traceability tool (TOOR) [21] offers a unique capability by allowing users not only to assert various types of traceability link, but also to define their semantics axiomatically. This capability allows for the connection of requirements to design documents, specifications, and code through significant and customizable relationships, thereby offering a more adaptable and thorough approach to traceability. Despite these advances, the integration of traceability tools across different platforms remains challenging. The diversity in methodologies and the lack of a common framework often led to compatibility issues, making it difficult to efficiently share traceability information across various tools and systems. Furthermore, there are no benchmarking models to assess the precision and completeness of traceability links generated by these tools. However, these tools remain essential for monitoring and maintaining connections between requirements and their related artifacts throughout the software development lifecycle.

2.3 Deep Learning Models

Advancements in deep learning have promoted the use of large language models (LLMs) such as bidirectional encoder representations from Transformers (BERT) [22] and generative pre-trained transformers (GPT) [23], to transform NLRs into various software artifacts.ifacts. These models offer superior contextual understanding and leverage a pre-training and fine-tuning paradigm that surpasses the traditional requirement transformation approaches. In the domain of requirement traceability, various approaches have been proposed to take advantage of advanced deep learning and NLP techniques to improve accuracy and efficiency. Guo et al. [24] presented a deep learning-based method that uses word embedding and repetitive neural network models, particularly a bidirectional gated recurrent unit (BI-GRU), to improve trace link accuracy by integrating artifact semantics and domain knowledge. Ali et al. [25] proposed an LLM-supported retrieval-augmented generation strategy to improve the traceability of class diagrams within code repositories. This approach employs keyword, vector, and graph indexing techniques, and capitalizes on code comments, dependency trees, and enhanced code summarization to effectively link high- level requirements with technical code constructs. Further contributions include the introduction of Trace BERT (T-BERT) [26], which leverages the bidirectional contextual capabilities of BERT to improve automated traceability in industrial settings. T- BERT is particularly advantageous for projects with limited data and scale to accommodate large-scale applications. Dai et al. [7] addressed the challenges in automated requirement code traceability (RCT) by introducing a deep neural network method. RCT effectively processes natural language by converting software requirements and source code documents into same-dimensional feature vectors and assessing linkages through a similarity analysis.

These studies collectively demonstrate the potential of advanced NLP and deep learning techniques to address traceability challenges by improving the precision of linking diverse software artifacts. Although LLMs have shown promising

results in the field [27], [28], their black-box nature presents a significant risk. The opaque decision-making processes in these AI models make it challenging to debug and refine systems, thus complicating trust and acceptance between stakeholders [29]. Furthermore, in requirements engineering, where documents are typically highly domain specific, the limited availability of labeled data hinders the ability of these models to learn effectively. Pauzi and Capiluppi [30] identified two main challenges in the application of NLP to the traceability of requirements. First, it is challenging to ensure that syntax and semantic similarities between different artifacts are adequately represented because effective traceability is based on the identification and linking of related components. Second, they highlight the need for scalable, adaptive, and transparent NLP models, because current solutions often operate as black boxes. These models must be explainable and efficient, particularly for requirement validation and regulatory tracing, while justifying any resource trade-off involved in their implementation.

3. THEORETICAL FOUNDATIONS

Software engineering recognizes the need for different models to depict a software system as it evolves from initial requirements to completed implementation. These models can illustrate various facets of the system (such as structural or behavioral) or reflect the system at different abstraction levels (such as an analysis model or a design model). Ensuring traceability is crucial to verify that source elements are accurately converted to target elements.

3.1 Classification of Traceability Links

Pinheiro [21] identified two categories of traceability links: functional and nonfunctional. Functional links are created through the transformation of one element into another by adhering to a defined set of rules. These links are deliberately generated along with the artifact due to the transformation, or they can be accurately recreated at any point by analyzing the initial artifact, the final artifact, and the transformation rules that were applied. The traces come from the syntactic and semantic relationships dictated by the models or notations used. Non-functional links are related to tracing software requirements aspects, including intentions, purposes, goals, responsibilities, and other abstract notions. This study focuses on functional links that can be further categorized into three different types.

- Links from text to model create a relationship between the concepts used in the requirements phase and the elements found in the analysis phase. For example, this connection can be seen between a requirements document and the associated software artifacts.
- Model-to-model connections link elements of different models across various levels of abstraction. For example, when an intermediate model is utilized in the transformation process, these connections relate the elements of the intermediate model to those of the final model.
- Inter-requirements or derived links connect different sections of artifacts. For example, when a class is linked to several requirements, it is related to that specific class.

3.2 Representation & Visualization of Links

Requirement tracking is essential for professionals in both management and technical domains, the task is driven by necessity. For instance, a technical aim may involve tracing the completeness of the software artifacts created. An analyst might review software artifacts alongside a requirement document to verify which parts are covered by the model and which are not. Management goals may include responding to change requests. When modifications affect certain aspects of the model, the manager should refer to the initial requirements before making any changes. User interfaces and visualization tools are vital to effectively explore and use traceability information. In [31], a set of requirements for visualization tools was identified, which outlined the ability to support the navigation and maintenance of traceability links and stipulated that such tools should:

- Provide users with various methods to investigate traceability links.
- Users can add, delete, and change the attributes of the existing links and their related artifacts, ensuring that changes are communicated effectively.
- Enable users to perform queries and filter traceability links.
- Connect with other software engineering applications (i.e., analysis tools).
- Assess and condense information related to the traceability process and its links.

Winkler and Pilgrim [8] classified visualization strategies into three main categories: matrices, cross

references, and graph-based techniques, as shown in Figure 1.

3.2.1 Traceability Matrix

A traceability matrix is a grid with two dimensions that shows the connections between two categories of artifacts, such as the NLR and associated software artifacts. The rows and columns correspond to the artifact elements, with marks at their intersections indicating links. Although this is equivalent to a graph representation, the matrix is visually less ordered. For example, in Figure 1(a), the black box indicates a direct link between the artifact on the left side and the artifact on the top side. Although traceability matrices are straightforward for simple tasks, such as noting a single link between artifacts [8], they are complex and difficult to manage in large-scale projects. Moreover, the two-dimensional nature of the grid recursively complicates the tracking of links across multiple artifacts.

3.2.2 Cross Referencing

A requirement specification document often contains numerous cross-references within the document and between different artifacts. A cross-reference serves as a guide within the text, which may be expressed in either casual everyday language or a structured specification. For ease of navigation, it is preferable for the cross-reference to appear as clickable hyperlinks. Figure 1(b) shows references stored as part of the metadata of an artifact, which is distinct from the artifact itself. Although readers can easily grasp these cross-references, traceability links offer a limited view, showing only the outgoing and incoming links of a single artifact. Unlike traceability matrices, this format provides a local perspective rather than comprehensive traceability artifacts. Moreover, it is almost impossible to sensibly visualize n-ary links using cross references [8].

3.2.3 Graph-based visualization

In visualizations that use graphs, artifacts are illustrated as nodes, while traceability links are shown as edges, allowing models to be displayed in a manner similar to that of other fields employing graph-based notation (Figure 1(c)). Unique identifiers were used as reference artifacts. This approach economizes space, provides a more comprehensive view of traceability information, and displays multiple artifacts, including their content and traceability links. Various link types, including n–ary links, can also be visualized; however, such models become large and complex, making it difficult for general users to understand them [8].

3.3 Semantic Object Models

Semantic Object Models (SOMs) [32] serve as basic abstraction frameworks that encapsulate the meaning of frequently utilized requirement concepts and their interconnections. They represent essential business transactions in terms of a network of cooperative objects that perform tasks to achieve a given goal. The premise of SOMs is that most business application systems are grounded in transactional domains that involve the orderly transfer and movement of resources (e.g., loans, trading, and banking). The reusable knowledge attached to the SOMs provides partial solutions that can be customized to satisfy the requirements of the new application problem. SOMs represent a collection of business transactions characterized by a set of shared business concepts. The agent denotes the assignment of responsibility to humans or machines, such as customers, cashiers, or

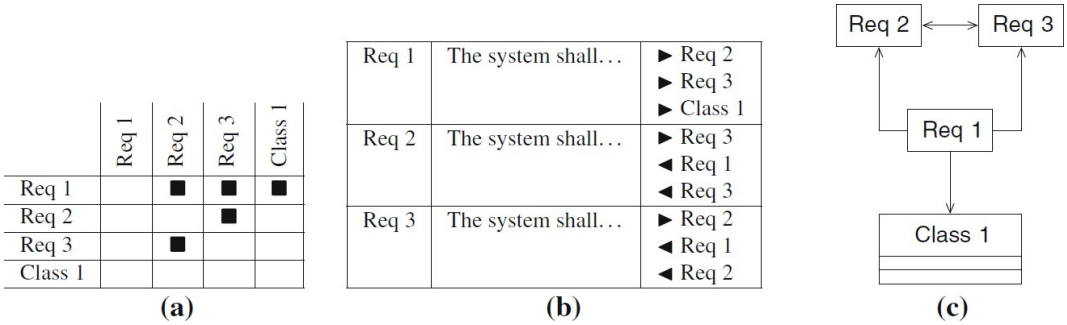

Figure 1. Visualization techniques: (a) Traceability matrix; (b) Cross-referencing; (c) Graph-based [8].

computers. An object refers to the subject matter involved in a business transaction that includes products or services. An **action** signifies the procedure of modifying an object to fulfill a particular objective; examples include ordering, purchasing, submitting, and detecting. A **state** illustrates the outcomes generated by actions performed on objects such as book-reserved or order-placed objects. A **container** refers to the storage or location where objects are stored or transferred, such as a warehouse or database. A problem or requirement is described by a minimal set of agents, objects, and actions that achieve a goal. A goal is a desired state of key objects, achieved through agent interactions and state transitions, and maintained by relationships between agents and key objects. State transitions are mapped to verb categories provided by WordNet. WordNet includes more than 21,000 forms of verbs (with more than 13,000 of them being unique strings) and around 8,400 meanings of words [33]. These verbs are divided into 14 categories, which correspond to the following semantic domains: verbs of bodily care and functions, change, cognition, communication, competition, consumption, contact, creation, emotion, emotion, motion, perception, possession, social interaction, and weather verbs. These categories form the basis for describing SOMs. SOMs are grounded in nine different types of actions from the following categories: possession, cognition, change, creation, motion, perception, communication, contact, and stative. Refer to Section 1 of the Supplementary Material for a visual representation of the SOMs.

- **Possession SOM:** This pertains to at least one source agent and one destination agent, focusing on the key object. This SOM can be utilized to illustrate various scenarios in which the ownership of the key object shifts among different agents, including purchasing goods and allocating resources.
- **Creation SOM:** This involves at least one agent, one key object, and the materials used to create a key object. This SOM captures the conditions under which a key object is formed from preexisting physical or conceptual elements, such as putting together a vehicle from ready-made components.
- **Change SOM:** This is related to at least one agent and one key object. Optionally, instruments may be used to carry out actions. This SOM represents scenarios where a key object is altered based on specific business rules, such as changing or canceling a purchase order.
- **Motion SOM:** This includes at least one agent, a key object, together with the source and destination containers. This SOM can depict a variety of situations in which the association of the key object with different containers is altered; for example, moving goods from a warehouse to a retail store.
- **Cognition SOM:** This is associated with at least one agent and a key object and may also include additional objects and containers. This SOM represents diverse cognitive tasks such as preparing a purchase order and identifying a supplier.
- **Perception SOM:** This relates to at least one agent and one key object. Tools may optionally be utilized to sense and report changes in the state of a key object. This SOM is applied in various contexts where monitoring the state changes of a key object is crucial, such as customers tracking their order status online.
- **Communication SOM:** This involves at least one source agent, one destination agent, and a key object that serves as the subject of communication. This SOM captures the exchanges between two agents: for example, an employee requesting a purchasing form from their department.
- **Stative SOM:** This is associated with at least one key object and an attribute. Attributes refer to the properties of things (elements or components). This SOM encapsulates the specifications that describe the characteristics of key objects. For example, a book contains a title and at least one author.
- **Contact SOM:** This involves at least one agent and a key object. Tools may optionally be used to perform tasks. This SOM can represent scenarios in which an action engages a key object, but its physical state remains unchanged.

4. OVERVIEW OF THE TRAM APPROACH

The previous work of the author laid the groundwork for the transformation process within TRAM. This study details the transformation rules and outlines the methodology used to validate the effectiveness of this approach in establishing robust traceability links. As shown in Figure 2, the TRAM approach facilitates the automatic generation of traceability links between NLRs and their

corresponding class diagrams. The transformation process begins with a domain expert providing an NLR specification, which is parsed by the **NLR Parser** into a structured format (XML). The parsed information is then utilized by the **SOM Constructor**, which employs a set of predefined rules to identify SOM patterns from the parsed NLR. The SOM Constructor then instantiates elements of the identified SOM pattern with constructs from the parsed NLR to produce SOM instances (SOMi) in XMI format. The SOMi is taken as input by the **SOM Instances Translator**, which transforms each SOMi into individual class diagrams. These individual class diagrams are assembled into UML class diagrams using the **Model Composer**. Furthermore, the approach incorporates a **Requirements Tracer** component, which creates and visualizes traceability links. This component produces a traceability report that facilitates traceability between NLRs, SOMs, and class diagram elements. These components will be discussed in detail in the following sections.

5. NLR PARSER

The NLR Parser is designed to preprocess the NLRs, while the Stanford parser [34] is utilized to perform this function. The Stanford parser is a natural language parser that lacks lexical information, trained on data from the Wall Street Journal, achieving a total accuracy of 96.86% and an accuracy of 86.91% for words not previously encountered. A detailed explanation of the Stanford parser exceeds the limits of this study and is available in [34]. The NLR parser aims to accomplish the following goals.

- to identify the grammatical roles of words in the text and assign part-of-speech tags that reflect these roles.
- to generate grammatical relations among elements of a requirement statement.
- assign every word in the text a distinct identifier based on the sentence and token numbers.

5.1 Part-of-Speech Tagging

This step assigns a part-of-speech (POS) tag to each word in the text, which reflects its grammatical role; that is, nouns, verbs, adjectives, etc. Each sentence is parsed lexically to generate a parse tree using the Stanford Parser. The tagging was carried out in four steps. In the first stage, the text is divided into individual words and sentences. The words then receive POS tags using a predetermined lexicon and a specific set of rules. In the third stage, the preliminary POS tags are modified according to the contextual rules for POS assignments. Lastly, the likelihood of each possible

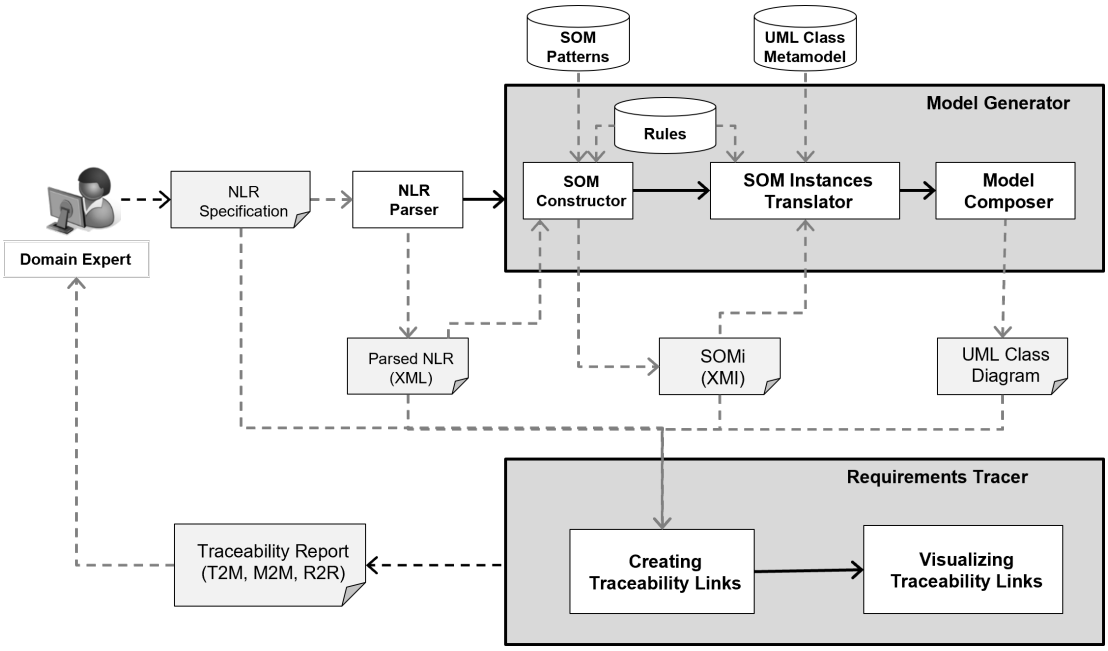

Figure 2. TRAM Architecture: A Structured Approach for Model Generation and Traceability

tag sequence is computed, and the sequence with the greatest probability is selected. The tagged requirement statements were then transformed into an eXtensible Markup Language (XML). Each word in the textual requirements is represented in XML according to the following definition:

<w id="ID" pos="POS"> word </w>

5.2 Generating Grammatical Relations

This step establishes grammatical relationships among words in a requirement statement, generating semantically valuable information. It uses the POS tags generated earlier as input to produce grammatical relationships, also known as typed dependencies. This process relies on the rules and patterns applied to both phrase structure and English sentences. Currently, the Stanford-typed dependencies manual contains approximately 53 grammatical relationships. Each grammatical relation involves a governor and is dependent. The following are examples of some typical grammatical relations; refer to [35] for a comprehensive list of definitions:

- root: *root* – The grammatical relationship of the root highlights the foundation of the sentence. The ROOT node is labeled as 0, while the indexing of actual words in the sentence starts at 1.
- dobj: direct object – The noun phrase that is the verb object is the verb's direct object.
- nsubj: nominal subject – A noun phrase that serves as the grammatical subject of a clause is known as a nominal subject.
- aux:*auxiliary* – An auxiliary within a clause refers to a nonprimary verb in that clause, such as a modal auxiliary, or a form of "be", "do", or "have" used in a periphrastic tense.
- neg:*negation modifier* – The negation modifier is the relation between a negation word and the word it modifies.
- agent: *agent* – An agent refers to the subject of a passive verb that is presented by the preposition "by" and performs the action.
- prep: *prepositional modifier* – A prepositional modifier associated with a verb function as a connection to the dobj relation; depending on the adjective, the noun involved in this relation may indicate either a source or a destination object.
- conj: *conjunct* – A conjunct refers to the relationship between two components linked by a coordinating conjunction, for example, "and" or "or."

A grammatical relation is written as the relation's name (governor, dependent), where governor and dependent are words in the sentence. In general, content words are preferred as heads and auxiliaries that depend on them. However, in the TRAM approach, verbs are chosen as the heads of sentences when determining dependencies. The dependencies create a web of connections that correspond to a directed graph model, where the words in the sentence serve as the nodes, and the grammatical relationships are represented by the edge labels. The grammatical relations are further represented in XML format and grouped by direct object dependency. By linking these dependencies, the subjects and objects of verbs in a sentence can be determined, leading to the identification of the Semantic Object Models discussed in Section 3.3.

5.3 Assigning identifiers to NLR Constructs

The NLR Parser divides NLRs into linguistic components, including words, punctuation marks, numbers, and mixed alphanumeric sequences through a process called tokenization. This process produces a series of tokens, spaces, and positions. In English, words are generally divided by spaces that act as separators. A token is a sequence of characters grouped into meaningful linguistic units for processing within text. The tokenizer moves through the streams of spaces and tokens to identify pairs of character offsets that indicate the starting and ending positions of tokens. Given that the required text is a linear string of characters, each token has a distinct beginning and end character offsets. An identifier is created using the start and end character offsets of a token, indicating its length. This data was stored in an XML (parsed NLR) document generated by the NLR parser. A token ID follows the format:

<beginCharacterOffset.endCharacterOffset>.

6. MODEL GENERATOR

6.1 SOM Constructor

The SOM Constructor plays a key role in translating the information derived from the NLR Parser into SOM patterns. It utilizes a collection of verbs gathered from the *dobj* dependencies in Parsed NLR documents to pinpoint the relevant SOM types. Additionally, the SOM Constructor employs word-sense disambiguation [36] to identify the meaning of the verb. Each verb v is associated with a set of possible senses Sv; Given a direct object dependency; dobj containing the verb

v, we determine which of the possible senses in Sv to assign to v in the context of dobj. In this study, the SOM Constructor chooses the primary sense of the verb from WordNet and associates it with a corresponding SOM pattern. The senses provided in WordNet are arranged in order of frequency, with the primary sense indicating the most frequently used meaning of the word. Nouns were classified in two stages. In the first stage, agents were tagged using WordNet derived dictionaries. Two separate dictionaries were created: one for human agents containing nouns representing individuals, and another for nonhuman agents containing nouns denoting groups. The dictionaries were developed by extracting person-related nouns for the former and group-related nouns for the latter. The second stage focuses on identifying key objects on the basis of their syntactic structures. Once each SOM pattern is identified, the SOM Constructor creates its corresponding concepts using elements from dependency relations. This component employs a series of translation rules to instantiate the SOM patterns. The result of this process is the SOM instances, which encapsulate the formalized textual requirements. The translation rules are as follows.

- **Rule 1:** A noun phrase that functions as the syntactic subject of a clause is seen as a potential candidate for the agent's role. If this noun phrase is not assigned to the agent role, it is classified as a non-agent. The distinction between agents and nonagents is that the latter cannot initiate or execute any actions.
- **Rule 2:** The external subject of an open clausal complement (the argument associated with a predicate) is recognized as a potential candidate for the agent's role.
- **Rule 3:** The direct object of a verb phrase, which is a noun phrase acting as the object, is designated as a key object.
- **Rule 4:** For a Communication SOM, the noun phrases introduced by the prepositions "for," "about," and "with" are assigned to the role of a key object.
- **Rule 5:** In the case of a passive verb, the complement introduced by the preposition "by" is considered a candidate for the agent role.
- **Rule 6:** A passive nominal subject, which serves as the syntactic subject of a passive clause, is assigned the role of key object.
- **Rule 7:** For Possession and Communication SOMs, the prepositional modifier of the verb is viewed as a candidate for the source or destination agent.
- **Rule 8:** Except for the rules mentioned above, all other verb prepositional modifiers are assigned roles, such as instruments, objects, containers, or materials, based on the specific SOM pattern that the verb triggers.
- **Rule 9:** When an SOM includes a non-agent, it is categorized as a Stative SOM. To assess this, TRAM calculates the likelihood that each concept is a physical object. A concept with a higher probability value is assigned to the agent role in the Stative SOM. Agents in Stative SOMs cannot initiate or perform any actions, and key objects are usually attributes of the agent.

Consider the following requirement statement:

"A library issues loan items to customers. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth."

When the NLR parser and SOM Constructor process the input, they produce an XMI document that includes SOM instances, as shown in Figure 3. Two SOM patterns were derived from this sentence using the translation rules. The first is the SOM of Possession, which is represented by these concepts: a possession action (issues), a source agent (library), a key object (loan items), and a destination agent (customers). The second pattern is the Stative SOM, defined by the following concepts: a Stative Action (keep), an unidentified agent currently designated as UNKNOWN, a key object (details), and object properties (name, address, and date).

6.2 SOM Instances Translator

This component converts each SOMi into a distinct class diagram. This process involves several steps.

```
<?xml version="1.0" encoding="ASCII"?>
<som:FunctionalSpec xmi:version="2.0 name="Library">
 <hasRequirement id="1" name="issue item">
    <hasSOM type="Possession">
      <hasAction name="issues" id="10.16" pos="VBZ" cardinality="1" lemma="issue"/>
      <haskeyObject name="items" id="22.27" pos="NNS" cardinality="*" lemma="item"/>
      <hasdstAgent name="customers" id="31.40" pos="NNS" cardinality="*" lemma="customer"/>
      <hassrcAgent name="library" id="2.9" pos="NN" cardinality="1" lemma="library"/>
    </hasSOM>
  </hasRequirement>
  <hasRequirement id="2" name="keep detail">
    <hasSOM type="Stative">
      <hasAgent name="UNKNOWN" id="" pos="" cardinality="" lemma=""/>
      <hasAction name="kept" id="72.76" pos="VBN" cardinality="1" lemma="keep"/>
      <haskeyObject name="Details" id="42.49" pos="NNS" cardinality="*"   lemma="detail"/>
      <hasObject name="name" id="87.91" pos="NN" cardinality="1" lemma="name"/>
      <hasObject name="address" id="93.100" pos="NN" cardinality="1" lemma="address"/>
      <hasObject name="date" id="106.110" pos="NN" cardinality="1" lemma="date"/>
    </hasSOM>
  </hasRequirement>
</som:FunctionalSpec>
```

Figure 3. Example of Possession and Stative SOM Instances [10]

Step 1. Loading SOM instances: A collection of SOMi is derived from the SOM Constructor component, created by instantiating SOM patterns that include NL requirement constructs. Each SOMi represents a requirement statement and is formatted in XMI. The XMI file shown in Figure 3 was uploaded to the SOMi Translator. The SOMi Translator employs an XMI parser to analyze separate SOM instances alongside their associated concepts, ultimately producing a parse tree. Following this, the XMI document is formatted for better human readability, allowing the modeler to explore SOMi and its associated concepts effectively.

Step 2. Identifying Missing Concepts: This step is optional since TRAM can generate preliminary class diagrams, even if some concepts are absent. SOMi is deemed incomplete if TRAM has difficulty extracting certain concepts from a sentence due to co-reference challenges that are typical in NLP techniques. For example, requirements that use passive constructions or include pronouns (it, she, he, they, etc.) and acronyms can hinder coreference resolution. Accurate interpretation of the text or evaluation of the significance of different topics, pronouns, and other referential expressions presents computational difficulties. In the TRAM approach, if an SOMi cannot be fully instantiated using requirement constructs, it is designated UNKNOWN. The resolution of missing concepts was facilitated through an intuitive interface that allowed a human modeler to select a concept from a drop-down list or introduce a new one. When a missing concept is detected in the selected SOMi, a dedicated window prompts the user to address it directly.

Step 3. Converting SOMi to Class Diagram: This process consists of three substeps for mapping SOM instances to class diagrams: (a) transforming each SOMi concept into a UML class element through the application of translation rules, (b) establishing UML relationships based on the concept associations defined in the SOM patterns, and (c) developing a UML class diagram to illustrate the SOMi.

In **Step 3(a),** suitable translation rules were used to convert the SOMi concepts into the respective components of the UML class. These rules define the correlations between SOMi concepts and UML Class elements. Implementation occurs within a transformational context that converts the source model (SOMi) into the target model (UML Class Diagram), using the specific transformation rules presented in Table 1.

Table 1. Rules for converting SOMi into UML Class elements.

Target Element	Rule Description
Class	All THING concepts (e.g., AGENT, KEY OBJECT, MATERIAL, CONTAINER, INSTRUMENT, OBJECT) are represented as class elements such that the class name is a THING name.
Attribute	If a THING has a PROPERTY, then the PROPERTY is an attribute of the THING class, such that, the attribute name and type are derived from a PROPERTY concept.
Operation	If an AGENT performs an ACTION and an ACTION affects a KEY OBJECT then an ACTION is an operation of the AGENT class, such that, operation name is the ACTION name and return type is the KEY OBJECT class.
Association I	If an AGENT performs an ACTION and an ACTION affects a KEY OBJECT, then the relationship is an association such that the memberEnd class is an AGENT, ownedEnd class is a KEY OBJECT and the association label is the ACTION name.
Association II	If an OBJECT modifies a KEY OBJECT, then the relationship is an association such that a memberEnd class is an OBJECT and ownedEnd class is KEY OBJECT.
Composition	If a MATERIAL makes a KEY OBJECT, then the relation is composition, such that the composite class is a KEY OBJECT and a part class is MATERIAL.
Aggregation I	If a CONTAINER contains a KEY OBJECT, then the relation is an aggregation, such that an aggregate class is a CONTAINER and part class is a KEY OBJECT.
Aggregation II	If a THING has an OBJECT, then the relation is an aggregation such that an aggregate class is a THING and a part class is an OBJECT.
Dependency I	If an AGENT uses an INSTRUMENT, then the relationship is a dependency, such that the client class is a AGENT and the supplier class is an INSTRUMENT.
Dependency II	If there is an ACTION that involves two AGENTs and one AGENT depends on another to perform the ACTION, then the relationship is a dependency.
Generalization	If a THING concept is a specialization of a generic concept THING, then the relationship is generalized, such that a specialized THING is a subclass and a generalized THING a superclass.

Step 3(b) converts the relationships between concepts derived from the SOM definitions into UML relationships. Each SOM pattern follows a predefined structure based on the SOM metamodel. This internal framework specifies relevant concepts and their interconnections. For instance, a Possession SOM is linked with at least one source agent that possesses a key object assigned to a destination agent. In this case, association rule 1 translates the 'owns' and 'allocate' relationships into UML associations. A Stative SOM can be associated with both an agent and a key object. Here, aggregation rule 2 converts the 'has' relationship into a UML aggregation association. TRAM extracts the multiplicity of conceptual relationships from part-of-speech tags produced by the NLP parser. The nature of noun tags, whether singular or plural, dictates how many instances of a concept can exist within a relationship. When a noun phrase is singular, the multiplicity is defined as one; conversely, if it is plural, the multiplicity is considered unlimited (*).

To conclude the SOM Instances Translator phase, in Step 3(c), class diagrams representing each SOMi were created, as shown in Figure 4. The element that was formerly labeled as UNKNOWN in the Stative SOMi has now been recognized as a "library."

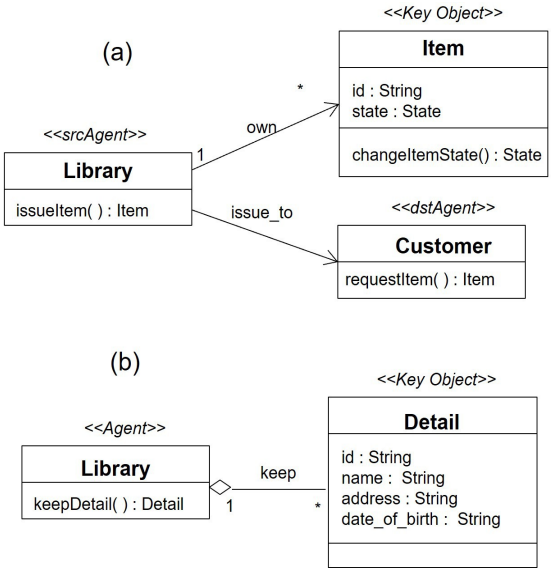

Figure 4. UML Class Diagrams representing (a) Possession SOMi and (b) Stative SOMi

6.3 Model Composer

The Model Composer component was designed to combine class diagrams that represent individual SOMi into a cohesive class diagram. The integration process involves two primary steps. The first step, known as **matching**, involves identifying elements that depict the same concepts in different class diagrams. After the matching elements are identified, a merging process takes place where the matched elements are combined to create new model elements that offer a cohesive representation of concepts.

Matching: A crucial step in creating the model elements is recognizing similarities or differences in the input models to decide what should be combined. This method relies on a signature-based composition technique that circumvents the conflicts typically found in name-based matching. A signature is composed of some or all properties linked to an element within the UML Class metamodel. Properties defining a signature can range from the name of the element to an extensive list of all associated properties. For example, the signature of an attribute is determined by both its name and type. The signature of a class element is established by its name and type, where the type is derived from the SOM metamodel, encompassing categories such as Agent, Key Object, and Container. Usually, the default signature includes only the name of the model element. In signature-based matching, elements that belong to the same type offer various points of view on a single concept. The semantics involved in the matching process depend on the specific domain. Recognizing model elements that signify the same concept is contingent on details pertinent to the interpretation of the model.

Merging: The merging operator creates a new model by combining two pre-existing models. It combines elements that match the criteria established by the matching operator and produces new elements in the resulting model. This operator analyzes all elements that correspond to the two input models. When these elements fulfill the merging criteria, it generates a new element in the output model. The merging process follows the specified rules.

MR 1. *When the elements of the model (such as classes) in the input models share identical signatures, they are consolidated into a single element.*

MR 2. *If the properties represented by model elements (such as class attributes) have corresponding signatures, they are represented only once in the resulting merged element.*

MR 3. *If one property exists in one matching element but not in the other, it will be included in the composed model element.*

MR 4. *If the relationships have matching ends, they merge, as shown in Figure 5.*

MR 5. *In cases where matching relationships have different multiplicities at their respective ends, the greater multiplicity is applied to the end of the merged association in the composite model.*

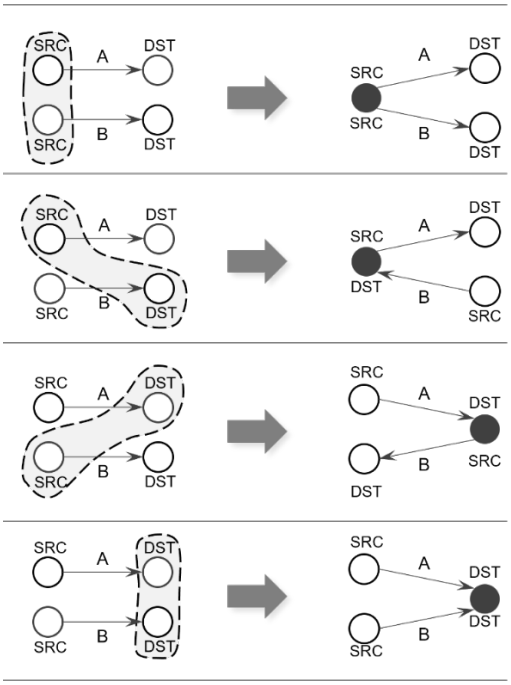

Figure 5. Illustration of matching and merging relationship ends. Relationships are unified when their ends match, ensuring consistent connections within the model.

7. REQUIREMENT TRACER

A traceability link represents an abstraction that arises from converting one model element into another. These transformations are naturally directional, creating from-to-relations between the model elements. Each link is defined by a type that reflects its place in a series of transformations, influenced by the kind of model element involved and the transformation rule applied. TRAM can generate three types of traceability links: (1) linking

requirement constructs with model elements, known as Text2Model; (2) connecting model elements across various levels of abstraction, referred to as Model2Model; and (3) associating concepts across multiple requirement statements, called inter-requirements. The Requirement Tracer component has a dual-step process that first generates traceability links, forming connections among related entities to clearly delineate dependencies and relationships. Second, it renders these traceability links visually, offering a graphical representation that facilitates the comprehension and analysis of the interrelations among various components.

7.1 Creating Traceability Links

As discussed in Section 5.3, a distinct identifier was assigned to each requirement element. Instances of SOM are derived from these requirement constructs (or terms). Therefore, the same identifier is assigned to an element within the SOMi that relates to a specific NLR construct. These identifiers are incorporated into the XMI document that contains the SOMi, as demonstrated in Figure 3. Each element within the SOMi corresponds to a specific requirement it represents. The connection between model elements and requirement constructs generally demonstrates a many-to-many relationship, where multiple model elements can reference the same requirement construct, and a single requirement construct can be represented by various model elements. TRAM defines three types of traceability links: Text2Model (T2M), Model2Model (M2M), and Inter-Requirements (R2R). A traceability link instance is created when a mapping rule establishes the connection between the source and target elements during the transformation process.

T2M Links: These links are created using transformation rules that connect the Parsed NLR constructs with SOMi elements (intermediate model). Once the relationship between a Parsed NLR construct and an SOMi element is established, a T2M link is created. Each T2M link is represented as a tuple: *Type(source_element, target_element),* where the source element comes from the requirements, and the target element is an SOMi element.

M2M Links: These links are created using mapping rules that connect elements of models at different levels of abstraction, such as those present in SOMi and UML. For example, an Agent or Key Object is associated with a class element; an action aligns with a class operation and is represented through an association between classes, while properties of the agent are related to attributes. M2M links are represented as a tuple: *Type(source_element, target_element)* in which the source element comes from SOMi and the target element belongs to a UML class model.

Each element within a class diagram is assigned to a unique identifier that is generated automatically. This functionality is facilitated by the MetaObject Facility (MOF) and Eclipse Modeling Framework (EMF), which utilize XMI tags and attributes. The MOF is designed to provide a framework for defining and managing metadata and interoperability between different modeling tools. It describes how models are represented in XML Metadata Interchange, a standard that allows different tools to exchange metadata information common to UML tools. Throughout the model transformation process, a unique xmi.id is created for every UML element. This automatic creation of distinct identifiers supports consistency and guarantees traceability.

R2R Links: Connections among concepts found in various requirement statements. R2R links are recognized during the model composition phase. To ensure traceability of requirements, a composed model ***M*** must meet the following criteria:

- Each component of the input must correspond to a related component within ***M***.
- The input components are aligned with an identical component in ***M*** only if they are equivalent, and this equality matching must demonstrate transitivity.
- In ***M*** there should be no elements outside those included in the input models.

R2R links associate a model element with one or more requirement statements. For example, if an agent handles several tasks, these tasks are referenced in multiple requirement statements. During the composition process, elements that reference the same concept are combined into one single element. Consequently, an R2R link encompasses references where a merged element is indicated in the requirements document. An R2R link is formatted as an element (reqID, reqID, reqID, etc.), where element denotes a merged element from the resulting UML class diagram, and reqID reflects the requirement statement that defines the element. When intermediate models are utilized in the transformation, additional traceability links can be formulated between the initial and final target models. R2R links can be likened to the derived traceability links because they create implicit associations between the

original NLRs and the final target model (UML class diagram). For example, knowing that the SOMi element ***S*** contributes to the fulfillment of requirements *R1* and *R2*, and that the class diagram element ***C*** is derived from ***S***, we can infer that the class diagram element ***C*** is also related to requirements *R1* and *R2*. The traceability links derived facilitate a deeper understanding of the connections between the source and target models.

7.2 Visualizing Links

The TRAM approach incorporates both a traceability matrix and graph-based visualization techniques to address some of the known challenges associated with these methods. In this study, the traceability matrix is presented in a table format that is accessible to users of various backgrounds. Typically, an element in the matrix is identified by a reference number and a pop-up window is used to display the metadata or content of that element. Conventional traceability matrices only include references (IDs) to the elements or markers at each intersection to indicate the existence of a link, without revealing the model elements themselves. On the contrary, TRAM presents the elements at each intersection in the matrix instead of the identifiers, facilitating easier navigation through the links among the various elements. Furthermore, TRAM includes the ability to emphasize related elements within models at different abstraction levels. For example, the modeler can visualize the traceability connections between T2M and M2M. When an SOMi is selected, its associated textual requirements are highlighted, as illustrated in the lower left panel of Figure 6, which shows the T2M links. When an SOMi is selected, the corresponding UML class diagram is shown, as demonstrated in the right panel of Figure 6, which illustrates the M2M link.

Table 2 presents an illustrative traceability report or matrix generated using the TRAM approach. Examples of T2M links include *T2M (item, KeyObject.item)* and *T2M (library, Agent.library)*, while examples of M2M links are *M2M (KeyObject.item, Class.Item)* and *M2M (Agent.library, Class.Library)*. An example of an R2R link is library (1, 7, 25), indicating that the library element is derived from requirements 1, 7, and 25.

Traceability data may be incorporated within the models they reference, manifested as model element attributes like tags and properties, or kept separate from these models in another distinct model. The former approach encounters various issues [37], as each model possesses its own representation and semantics. Including such information directly within a model can lead to pollution. A directed link that exists solely in the

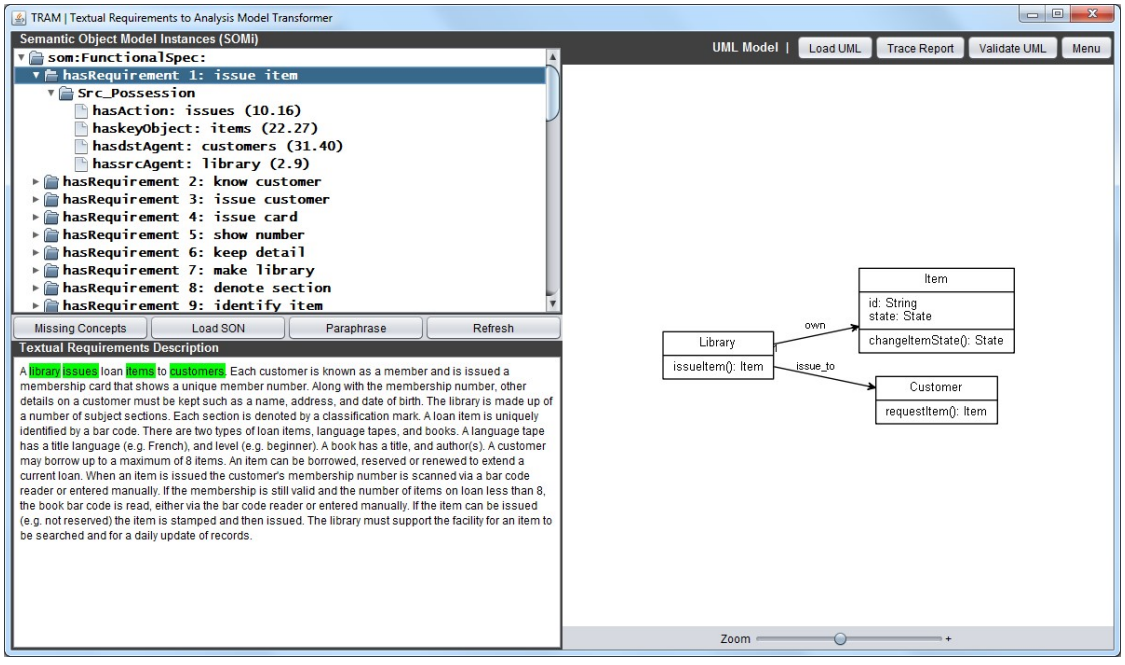

Figure 6. Visual Representation of Traceability Connections

source model would not be observable in the target model. However, if the linking data is maintained in both the source and target models, synchronization with each change would be necessary, increasing the effort needed to maintain consistency. The latter approach of external storage is beneficial because it keeps both the source and target models clean by storing link information in a separate model. Moreover, the maintenance required for linking information upon modifications is significantly reduced. Thus, TRAM opts to store linking information outside the models they pertain to, generating a traceability report in XML format that encapsulates all elements and properties of the links. A key requirement for externally storing traceability links is that model elements possess unique and persistent identifiers, allowing traceability links to be resolved clearly and unambiguously.

8. EVALUATION OF TRAM

This section presents the method and results to evaluate the accuracy and completeness of class diagrams produced by TRAM. This method compares the class diagrams generated by TRAM with those created manually by requirements engineering experts. Professionals and experts in the requirement engineering domain should participate in the evaluation of the automatically generated analysis model [38], [4] and compare it with a manual model developed by human experts to determine how closely the automated analysis model matches the expert solution. The models created by human experts are precise; therefore, they act as standards for comparison. The use of human experts to create benchmark class diagrams is important because it ensures a high-quality and accurate representation of the required classes, relationships, and attributes. Their expertise established a solid standard for evaluating

Table 2. Illustrative Traceability Report generated by TRAM.

NLR Construct	SOMI Element	UML Element	Requirement ID
mark	STATIVE_INV.hasnonAgent.mark	CLASS.Mark	8
card	COMMUNICATION.haskeyObject.card	ATTRIBUTE.Section.mark, CLASS.Card	4, 5
section	STATIVE_IN.haskeyObject.section	CLASS.Section	8
customer	COMMUNICATION.hasobject.customer	CLASS.Customer	2, 3
item	COMMUNICATION.haskeyObject.item, POSSESSION.haskeyObject.item, CREATION.haskeyObject.item, MOTION.haskeyObject.item	CLASS.Item	1, 9, 14, 15, 16, 17, 18, 22, 23, 24
extend	CHANGE.hasAction.extend	METHOD.Item.extendLoan()	17
address	STATIVE.haskeyObject.address	CLASS.Address	2
code	STATIVE_INV.haskeyObject.code, COGNITION.haskeyObject.code, MOTION.haskeyObject.code	ATTRIBUTE.Item.code, CLASS.Code	9, 20, 21
library	COMMUNICATION.hasrcAgent.library,	CLASS.Library	1, 7, 25
loan	CHANGE.haskeyObject.loan	CLASS.Loan	17
read	COGNITION.hasAction.read	METHOD.BarcodeReader.readCode(), ASSOCIATION.BarcodeReader-to-Code	26
reserve	POSSESSION.hasAction.reserve	METHOD.Library.reserveItem()	15
have	STATIVE.haskeyObject.have	CLASS.Name	10, 11, 12, 13
author	POSSESSION.haskeyObject.author	ATTRIBUTE.Book.author	13
issue	COMMUNICATION.hasAction.issue	METHOD.Library.issueItem(), METHOD.Library.issueCard(), METHOD.Library.issueNumber()	1, 3, 4, 18, 19, 22, 24
customer	COMMUNICATION.hasdtAgent.customer, CREATION.hasnonAgent.customer, MOTION.hasAgent.customer	CLASS.Customer	1, 4, 14, 15, 16, 18, 19, 21, 22, 24
librarian	MOTION.hasAgent.librarian	CLASS.Librarian	23
borrow	POSSESSION.hasAction.borrow	METHOD.Library.borrowItem()	14, 27

automated tools. Experts effectively handle real-world complexities and interpret requirements and constraints, leading to more adaptable solutions. Manually created benchmarks also help identify the limitations of automated tools that lead to improvements in their performance.

8.1 Selection of Critique Criteria

The effectiveness of the TRAM tool was evaluated using a set of criteria chosen to reflect the core goals of the study: ensuring accurate and comprehensive traceability between NLRs and class diagrams. The selected criteria are as follows:

Precision is an important metric in assessing how accurately the tool generates relevant elements within class diagrams. When human experts create software artifacts from NLRs, they establish connections between the requirement constructs and elements of the class diagram. For each element of the created class diagram, a corresponding link exists for each construct in the NLRs. To evaluate the accuracy of the traceability links produced by TRAM in comparison to those crafted manually by experts, a precision metric was used. This metric evaluates the accuracy and relevance of elements within a TRAM-constructed class diagram, highlighting how closely the model aligns with the real world or domain. The precision was calculated using the following formula:

$$Precision = \frac{Number\ of\ correctly\ identified\ elements}{Total\ number\ of\ elements\ retrieved} \quad (1)$$

Recall measures the completeness of the traceability process. The completeness criterion evaluates the ability of TRAM to identify all model elements that align with those created by human experts, thus confirming its traceability links. The recall metric was used to assess the completeness of the class diagrams generated by TRAM. Recall evaluates the proportion of accurately identified relevant items compared to the total count of relevant items in the benchmark. Specifically, it focuses on how well the generated model captures all relevant information defined in the requirements. The recall measure was expressed as follows:

$$Recall = \frac{Number\ of\ correctly\ identified\ elements}{Total\ number\ of\ elements\ in\ the\ Benchmark} \quad (2)$$

Both **precision** and **recall** are well-established metrics in information retrieval systems and have been widely adopted in software engineering to evaluate tools that generate software artifacts from natural language requirement specifications (NLRs) [39], [40], [41]. Given the importance of balancing accuracy and completeness, the F1 score is used, a metric that combines precision and recall into a single value. The F1 score provides the harmonic mean of precision and recall, offering a comprehensive measure of the model's overall performance. This score is particularly valuable when it is necessary to consider both the accuracy of the retrieved elements and the completeness of the generated software artifacts, ensuring a robust evaluation of the tool's effectiveness.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

These criteria—precision, recall, and the F1 score—are widely adopted in the software engineering domain for evaluating the effectiveness of traceability tools, ensuring a thorough assessment of TRAM's performance in generating reliable traceability links.

8.2 Constructing Class Diagrams

The evaluation used a dataset comprising five NLRs sourced from peer-reviewed studies, which are well-regarded for assessing requirement engineering tools. A full description of the data set is provided in Supplementary Material Section 2.

- **NLRs-1**: originated from the domain of library information systems and was discussed and used in [41].
- **NLRs-2**: provides a high-level overview of the library system, as used in [42] and [43].
- **NLRs-3:** The site centers on an online footwear retailer that offers shoes to customers through a website.
- **NLRs-4**: outlines the interaction between a fictional salesperson and a sales order system. This specification was published and referenced in [44].
- **NLRs-5**: describes a theoretical specification related to an in-flight missile control system. This specification was introduced and applied in [45].

Two experts in requirements engineering were involved in developing benchmark class diagrams for the five NLRs. Expert 1 (E1) had more than 25

years of experience in requirements engineering, education and research, while Expert 2 (E2) had more than 15 years of experience in the same fields. The two experts received the same data set and the following instructions.

Create a class diagram or identify components (e.g., classes, properties, methods, and associations) required to develop a class diagram based on each NLR specification. A class element can be textually represented as Class [Attribute] (Operation), and a relationship can be represented as a class –verb phrase– class.

Table 3 presents the total number of elements (including classes, operations, and relationships) identified by experts for each NLR specification. This analysis was performed by counting the individual classes, operations, and binary relationships within the benchmark diagrams produced by the experts. For verification purposes, the benchmark class diagram elements are presented in Appendix A and Appendix B to allow the reader to independently review the number of elements. Table 4 shows the total number of elements identified or constructed using the TRAM approach of the five NLRs. The generated TRAM class diagrams are presented in Supplementary Material, Section 3.

8.3 Comparison Strategy

In the class diagram generated by TRAM from an identical NLR specification, each component was aligned with a corresponding element in the benchmark. When a match is identified, the counter for the number of relevant elements correctly identified by TRAM increases by one, thus categorizing an element as relevant if it corresponds accurately to a benchmark element. Due to limited resources, the comparison was executed by the author and independently corroborated by two graduate students. The author instructed the students on how to implement the comparison strategy.

The matching process was carried out in two stages according to the methodology described in [41]. Initially, the names of the elements to be matched must exhibit a sufficient level of similarity; for example, *loan items* can appropriately match *items* but not with *loans*. The context in which each element was used was then evaluated to verify that it complied with the meanings associated with its name. **Classes** are matched using approximate name matching, and contextual relevance is evaluated by analyzing their attributes, operations, and interrelationships. Although attributes are not quantitatively assessed, they play a role in the contextual evaluation of class elements. In cases where a single key class could potentially correspond to two response classes, or conversely, the matching score remains constant, irrespective of the chosen configuration. The matching of **operations** was solely based on name similarity; consequently, parameters, return types, and data types were excluded from the analysis. It is essential for an operation to have a name that matches exactly that of the benchmark and to exist within the correct class. The **relationships** were evaluated through contextual comparisons, focusing on binary relationships. A relationship is considered *relevant* between classes A and B if it exists in their respective benchmark classes. In this study, associations, aggregations, compositions, and generalizations are counted as relationships.

8.4 Results

Although experts used the same data set and instructions, they identified different numbers of elements. This shows that a single benchmark class diagram for any NLR specification is not definitive, because different experts often produce varying diagrams. The same NLR can be represented in various ways by different experts, or even by the same expert at different moments. This element of subjectivity implies that class diagrams cannot be classified as right or wrong, but rather as adequate or inadequate. Consequently, TRAM performance was evaluated against the benchmark diagrams of both experts. This method recognizes the intrinsic variability in how humans interpret requirements and underscores the need for adaptability when evaluating automated tools. By incorporating diverse expert points of view, the assessment becomes more resilient, facilitating a more thorough understanding of the strengths of TRAM and potential areas for improvement.

Table 3. Number of elements identified by experts for each NLR (Benchmarks).

		NRL-1	NRL-2	NRL-3	NRL-4	NRL-5
Expert 1	Classes	10	10	6	6	9
	Operations	12	10	16	17	12
	Relations	5	5	5	4	7
	Total	**27**	**25**	**27**	**27**	**28**
Expert 2	Classes	8	8	7	7	8
	Operations	11	12	16	18	15
	Relations	5	6	6	6	7
	Total	**24**	**26**	**29**	**31**	**30**

Table 4. Number of elements produced by TRAM for each NLR.

	NRL-1	NRL-2	NRL-3	NRL-4	NRL-5
Classes	18	10	17	18	22
Operations	23	21	39	45	22
Relations	17	16	20	23	25
Total	**58**	**47**	**76**	**86**	**69**

Table 5. TRAM's performance results.

		NRL-1	NRL-2	NRL-3	NRL-4	NRL-5
Expert 1	Precision	0.33	0.49	0.25	0.30	0.35
	Recall	0.70	0.92	0.70	0.96	0.86
	F1-score	0.45	0.64	0.37	0.46	0.49
Expert 2	Precision	0.29	0.51	0.24	0.27	0.32
	Recall	0.71	0.92	0.62	0.74	0.73
	F1-score	0.41	0.66	0.34	0.39	0.44

As presented in Table 5, TRAM has a strong recall in all NLRs, consistently capturing most relevant elements, with values remaining above 0.60 and peaking at 0.96 for NLRs-4. However, the precision is significantly lower, falling between 0.25 and 0.51, which indicates that a considerable number of irrelevant or inaccurate elements are also included. A contributing factor to this outcome is the use of SOM patterns by TRAM that contain predefined elements. This significantly increases the number of elements identified, affecting precision by including many elements that may not be as relevant as per human experts. F1 scores, which balance both precision and recall, reflect moderate achievement and range from 0.34 to 0.66. These results imply that while TRAM effectively maintains coverage, there is a need to improve its accuracy in recognizing relevant elements. Enhancing precision would result in better overall performance, as evidenced by higher F1 scores.

8.5 Threats to Validity

Regarding ***internal validity***, it is important to note that the approach used to compare TRAM-generated class diagrams with benchmark diagrams may be biased. To address this concern, a comparison strategy was explicitly defined. This strategy was previously used by other scholars in the domain and was applied systematically throughout this study. The comparison was conducted by the author and independently verified by two post-graduate students. Students enlisted by the author were provided with guidelines to execute the comparison strategy. Additionally, there is potential for subjective bias by human experts during the manual creation of benchmark diagrams, as these may reflect individual expert opinions rather than actual system requirements. To mitigate this, the experts involved in generating the benchmark diagrams for this study have substantial experience and specialization in requirements engineering.

Regarding external validity, the data set employed may not encapsulate the entire spectrum of complexity and variability characteristics of real-world software engineering projects, thereby constraining the generalizability of the results. To address this limitation, the data set used in this study comprised five NLR specifications from various problem domains derived from peer-reviewed publications and renowned for their use in assessing NLP4RE tools.

Construct validity concerns can arise from definitions and perceptions of precision and completeness in the evaluation of class diagrams. Definitions that are too narrow or too vague may fail to capture all relevant aspects, potentially leading to a partial evaluation of class-diagram quality. In addition, differing interpretations can cause inconsistencies, which undermines the validity of the results. Such definitions must align with real-world demands to ensure that the evaluation mirrors practical applications. The consistent application of these definitions throughout the evaluations is crucial to avoid misleading comparisons and to accurately represent the potential of the tool in this study. The definitions used in this study came from literature and were initially developed to assess information retrieval systems, which are now widely used to assess the performance of NLP4RE tools.

9. CONCLUSION

This study introduces a structured traceability approach to transform NLRs into class diagrams, enhancing requirement traceability in the software development process. Using NLP and SOM patterns, the TRAM tool automates class diagram generation, facilitating comprehensive traceability between NLRs and class diagrams. The evaluation results demonstrate TRAM's high recall in capturing relevant elements; however, its precision is affected by the inclusion of predefined elements within SOM patterns, leading to irrelevant details. Despite this, the results highlight TRAM's robustness in achieving complete traceability and suggest further opportunities to improve accuracy.

From my perspective, TRAM represents a meaningful step forward in bridging the gap between natural language requirements and formal software design models. While the tool effectively establishes traceability links, refining its precision remains an important challenge. In my view, this trade-off between recall and precision reflects a broader challenge in requirements engineering, where automation must balance completeness with correctness. I believe that integrating advanced NLP techniques and machine learning models could significantly mitigate these limitations, potentially leading to more adaptive and intelligent traceability solutions.

Furthermore, I recognize the broader implications of this approach. The structured nature of SOM patterns suggests that TRAM's methodology can be extended beyond class diagram generation to other software artifacts, such as sequence diagrams or even architectural models. This adaptability opens new avenues for research, reinforcing my belief that structured traceability techniques can play a transformative role in software engineering.

Future research should focus on refining the implementation of SOM patterns and exploring strategies to optimize the trade-off between recall and precision. Additionally, incorporating Large Language Models (LLMs) presents an exciting opportunity to enhance the transformation process by enabling a deeper contextual understanding of NLRs. Leveraging LLMs will significantly improve TRAM's ability to generate accurate and comprehensive traceability links, thereby broadening its impact within the field.

Overall, while TRAM demonstrates promising results, further refinement is necessary to enhance its precision and adaptability. Nevertheless, my contributions lay a strong foundation for future advancements in automated requirements traceability, ultimately driving more efficient and intelligent software development practices.

REFERENCES

[1] I. Sommerville, Software Engineering, 9th ed. Boston, Massachusetts, USA: Pearson Education Inc, 2011.

[2] O. Gotel and C. Finkelstein, ''An analysis of the requirements traceability problem,'' in Proceedings of IEEE International Conference on Requirements Engineering. Colorado Springs, CO, USA: IEEE, 1994, pp. 94–101. https://doi.org/10.1109/ICRE.1994.292398

[3] I. Galvao and A. Goknil, ''Survey of traceability approaches in model-driven engineering,'' in 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007). Annapolis, MD, USA: IEEE, 2007, pp. 313–324. https://doi.org/10.1109/EDOC.2007.42

[4] T. Yue, L. Briand, and Y. Labiche, ''A systematic review of transformation approaches between user requirements and analysis models,'' Requirements Engineering, vol. 16, pp. 75–99, 2011. https://doi.org/10.1007/s00766-010-0111-y

[5] D. Siahaan, R. Fauzan, A. Widyadhana, D. B. Firmawan, R. R. Putri, Y. Desnelita, Gustientiedina, and R. N. Putrian, ''A scoping review of auto-generating transformation between software development artifacts,'' Frontiers in Computer Science, vol. 5, p. 1306064, 2024. https://doi.org/10.3389/fcomp.2023.1306064

[6] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause, ''Rule-based generation of requirements traceability relations,'' Journal of Systems and Software, vol. 72, no. 2, pp. 105–127, 2004. https://www.sciencedirect.com/science/article/pii/S0164121203002425

[7] P. Dai, L. Yang, Y. Wang, D. Jin, and Y. Gong, ''Constructing traceability links between software requirements and source code based on neural networks,'' Mathematics, vol. 11, no. 2, p. 315, 2023. https://www.mdpi.com/2227-7390/11/2/315

[8] S. Winkler and J. von Pilgrim, ''A survey of traceability in requirements engineering and model-driven development,'' Software & Systems Modeling, vol. 9, pp. 529–565, 2010. https://doi.org/10.1007/s10270-009-0145-0

[9] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. A. Raja, and K. Kamran, ''Requirements traceability: A systematic review and industry case study,'' International Journal of Software Engineering and Knowledge Engineering, vol. 22, no. 03, pp. 385–433, 2012. https://doi.org/10.1142/S021819401250009X

[10] K. J. Letsholo, ''TRAM: Transforming textual requirements to support the earliest stage of model driven development,'' Ph.D. dissertation, The University of Manchester, United Kingdom, 2014. https://pure.manchester.ac.uk/ws/files/156331363/FULL_TEXT.PDF

[11] C. Larman, Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd ed. USA: Prentice Hall, 2005.

[12] M. Fowler, UML Distilled: A brief guide to the standard object modeling language, 3rd ed. USA: Addison-Wesley Professional, 2004.

[13] K. J. Letsholo, L. Zhao, and E. V. Chioasca, ''TRAM: A tool for transforming textual requirements into analysis models,'' in Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. Silicon Valley, CA, USA: IEEE, 2013, pp. 738–741. https://doi.org/10.1109/ASE.2013.6693146

[14] R. A. Pierce, ''A requirements tracing tool,'' SIGMETRICS Perform. Eval. Rev., vol. 7, no. 3–4, p. 53–60, 1978. https://doi.org/10.1145/1007775.811100

[15] W. Jirapanthong and A. Zisman, ''Xtraque: traceability for product line systems,'' Software & Systems Modeling, vol. 8, pp. 117–144, 2009. https://doi.org/10.1007/s10270-007-0066-8

[16] J. P. A. Almeida, M.-E. Iacob, and P. Van Eck, ''Requirements traceability in model-driven development: Applying model and transformation conformance,'' Information Systems Frontiers, vol. 9, no. 4, pp. 327–342, 2007. https://doi.org/10.1007/s10796-007-9038-3

[17] J. Cleland-Huang, C. Chang, and M. Christensen, ''Event-based traceability for managing evolutionary change,'' IEEE Transactions on Software Engineering, vol. 29, no. 9, pp. 796–810, 2003. https://doi.org/10.1109/TSE.2003.1232285

[18] IBM, ''IBM engineering requirements management doors documentation,'' 2024. https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors

[19] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, E. A. Holbrook, S. Vadlamudi, and A. April,

''Requirements tracing on target (retro): improving software maintenance through traceability recovery,'' Innovations in Systems and Software Engineering, vol. 3, pp. 193–202, 2007. https://doi.org/10.1007/s11334-007-0024-1

[20] I. Ozkaya and Ö. Akin, ''Tool support for computer-aided requirement traceability in architectural design: The case of designtrack,'' Automation in Construction, vol. 16, no. 5, pp. 674–684, 2007. https://doi.org/10.1016/j.autcon.2006.11.006

[21] F. Pinheiro and J. Goguen, ''An object-oriented tool for tracing requirements,'' IEEE Software, vol. 13, no. 2, pp. 52–64, 1996. https://doi.org/10.1109/52.506462

[22] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, ''Bert: Pre-training of deep bidirectional transformers for language understanding,'' in North American Chapter of the Association for Computational Linguistics. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. https://api.semanticscholar.org/CorpusID:52967399

[23] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, ''Improving language understanding by generative pre-training,'' OpenAI, San Francisco, CA, USA, Tech. Rep., 2018. https://www.mikecaptain.com/resources/pdf/GPT-1.pdf

[24] J. Guo, J. Cheng, and J. Cleland-Huang, ''Semantically enhanced software traceability using deep learning techniques,'' in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Buenos Aires, Argentina: IEEE, 2017, pp. 3–14. https://doi.org/10.1109/ICSE.2017.9

[25] S. J. Ali, V. Naganathan, and D. Bork, ''Establishing traceability between natural language requirements and software artifacts by combining rag and llms,'' in Conceptual Modeling. "Cham": Springer Nature Switzerland, 2024, pp. 295–314. https://doi.org/10.1007/978-3-031-75872-0_16

[26] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, ''Traceability transformed: Generating more accurate links with pre-trained bert models,'' in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Madrid, ES: IEEE, 2021, pp. 324–335. https://doi.org/10.1109/ICSE43902.2021.00040

[27] N. Marques, R. R. Silva, and J. Bernardino, ''Using chatgpt in software requirements engineering: A comprehensive review,'' Future Internet, vol. 16, no. 6, p. 180, 2024. https://doi.org/10.3390/fi16060180

[28] I. Ozkaya, ''Application of large language models to software engineering tasks: Opportunities, risks, and implications,'' IEEE Software, vol. 40, no. 3, pp. 4–8, 2023. https://doi.org/10.1109/MS.2023.3248401

[29] S.-C. Necula, F. Dumitriu, and V. Greavu-S, erban, ''A systematic literature review on using natural language processing in software requirements engineering,'' MDPI, Electronics, vol. 13, no. 11, p. 2055, 2024. https://www.mdpi.com/2079-9292/13/11/2055

[30] Z. Pauzi and A. Capiluppi, ''Applications of natural language processing in software traceability: A systematic mapping study,'' Journal of Systems and Software, vol. 198, p. 111616, 2023. https://www.sciencedirect.com/science/article/pii/S0164121223000110

[31] A. Marcus, X. Xie, and D. Poshyvanyk, ''When and how to visualize traceability links?'' in Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, ser. TEFSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 56–61. https://doi.org/10.1145/1107656.1107669

[32] E.-v. Chioasca, K. J. Letsholo, and L. Zhao, ''Transforming natural language requirement descriptions into analysis models,'' Oct. 13 2016, uS Patent App. 15/035,682.

[33] C. Fellbaum, ''English verbs as a semantic net,'' International journal of Lexicography, vol. 3, no. 4, pp. 278–301, 1990. https://doi.org/10.1093/ijl/3.4.278

[34] D. Klein and C. Manning, ''Accurate unlexicalized parsing,'' in Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1. Sapporo, Japan: Association for Computational Linguistics, 2003, pp. 423–430.

[35] M. C. de Marneffe, C. D. Manning, J. Nivre, and D. Zeman, ''Universal dependencies,'' Computational Linguistics, vol. 47, no. 2, pp. 255–308, 07 2021. https://doi.org/10.1162/coli_a_00402

[36] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller, ''Introduction to wordnet: An on-line lexical database*,'' International Journal of Lexicography, vol. 3, no. 4, pp. 235–244, 12 1990. https://doi.org/10.1093/ijl/3.4.235

[37] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, ''Model traceability,'' IBM Systems Journal, vol. 45, no. 3, pp. 515–526, 2006. https://doi.org/10.1147/sj.453.0515

[38] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E. V. Chioasca, and R. T. Batista-Navarro, ''Natural language processing for requirements engineering: A systematic mapping study,'' ACM Comput. Surv., vol. 54, no. 3, Apr. 2022. https://doi.org/10.1145/3444689

[39] V. Sagar, V. B. R, and S. Abirami, ''Conceptual modeling of natural language functional requirements,'' Journal of Systems and Software, vol. 88, pp. 25–41, 2014. https://doi.org/10.1016/j.jss.2013.08.036

[40] M. Elbendak, P. Vickers, and N. Rossiter, ''Parsed use case descriptions as a basis for object-oriented class model generation,'' Journal of Systems and Software, vol. 84, no. 7, pp. 1209–1223, 2011. https://doi.org/10.1016/j.jss.2011.02.025

[41] H. Harmain and R. Gaizauskas, ''Cm-builder: A natural language-based case tool for object-oriented analysis,'' Automated Software Engineering, vol. 10, no. 2, pp. 157–181, 2003.

[42] M. Ibrahim and R. Ahmad, ''Class diagram extraction from textual requirements using natural language processing (nlp) techniques,'' in Computer Research and Development, 2010 Second International Conference on. Kuala Lumpur, Malaysia: IEEE, 2010, pp. 200–204. https://doi.org/10.1109/ICCRD.2010.71

[43] S. K. Shinde, V. Bhojane, and P. Mahajan, ''Nlp based object oriented analysis and design from requirement specification,'' International Journal of Computer Applications, vol. 47, no. 21, pp. 30–34, June 2012. https://doi.org/10.5120/7475-0574

[44] P. Harmon and M. Watson, Understanding UML: The Developer's Guide: with a Web-based Application in Java. Massachusetts, United States: Morgan Kaufmann Publishers Inc., 1997.

[45] V. Ambriola and V. Gervasi, ''On the systematic analysis of natural language requirements with circe,'' Automated Software Engineering, vol. 13, no. 1, pp. 107–167, 2006.

Journal of Theoretical and Applied Information Technology
31st March 2025. Vol.103. No.6

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

APPENDIX A - EXPERT 1 BENCHMARK CLASS DIAGRAM ELEMENTS

NLRs 1: Library Information Systems I

Class [Attribute](Operation):

Library [] (issuesItem(), issuesCard())
Item [barCode, name] (reserveItem(), renewItem(), returnItem())
Customer [name, address, dateOfBirth] (login(), borrowItem(),returnItem())
MembershipCard [memberNo] ()
Subject_Section [classificationMark] ()
Book [title, author]()
Language_Tape [title, level] ()
BarCode_Reader [title, author] (scanBarCode(), enterBarCode())
Current_Loan [] (extendLoan())
Record [] (updateRecords())

Relationships:
Library -issues- Loan Items
Customer -issued- Membership Card
Library -made of- Subject Sections
Customer -borrow- Items
Membership card -scanned by- BarCode_Reader

Time spent: 16 mins

NLRs 2: Library Information Systems II

Class[Attribute](Operation):

Library_System [] (searchItem())
Student [] (borrowItem(), returnItem())
Faculty [] (borrowItem(), returnItem())
Item [status] (getStatus())
Book [isbn, title, author, year, publisher] ()
Journal [title, author] ()
Librarian [] (issueItem())
Deputy_Librarian [] (receiveItem())
Accountant [] (calculateFine())
Fine [] (checkoverdueItems())

Relationships:
Students -can borrow- Books
Faculty -can borrow- Books & Journals
Librarian -issues- Items
Deputy Librarian ---receives- Items
Accountant -charges- Fines

Time spent: 10 mins

NLRs 3: Online Shoe Company

Class [Attribute] (Operation):

Customer [name, address, payment_details, shoe_size, gender] (register(), login(), makeOrder(), trackOrder(), cancelOrder()),
OddShoeCompany [] (checkStock(), deliverOrder(), dispatchOrder(), sendStatement()),
Order [customer, status, shoe, date] (getOrderStatus())
Shoe [picture, price, stock_level] (findShoe(), getShoeList(), updateStockLevel())
Weekly_Report [Customers, ShoeSizes, Orders, StockLevels, cancelledOrders] (generateWeeklyReport())
Statement [] (getOffers(), sendEmail())

Relationships:
Customer -makes- Order
Customer -registers with- OddShoeCompany
OddShoeCompany -creates- Weekly_Report
OddShoeCompany -sends- Statement
Order -contains- Shoes

Time spent: 17 mins

NLRs 4: Sales Report Application

Class [Attribute] (Operation):

Salesperson [name,employee_number,ID] (login(), submitCustInfo(), checkCustInfo(), printOrder(), checkOrder(), submitOrder())
Order [availability, price, shipping, tax] (checkSalesPerson(), checkAvailability(), checkPricing())
Customer [name, address, status] (checkCreditLimit())
Item [name, price] (addItem())
Inventory [date, price, shipping, tax](approvePrice(), addShipping(), addTaxes(), totalOrder())
Accounting [] (approveCustInfo(), supplyCustomerCreditLimit()),

Relationships:
Salesperson ---submits--- Order
Order ---includes--- Items
Inventory ---checks--- Order
Accounting ---approves--- Order

Time spent: 23 mins

NLRs 5: Missile Control System

Class [Attribute] (Operation):

Missile [mode] ()
FMCS [command, height, position] (enterMode(), setMainEngineMode(), readHeight(), readHeading(), readPosition(), sendNavigationCommand(), computeCommand())
Main_Engine [mode: on, off] ()
Navigation_Engine [command] ()
Gyroscope [] (getHeading())
Altimeter [] (getHeight())
GPS [](getPosition())
Command [token] (getCommand())
Launch_Control_Center [] (setTarget())

Relationships:
Launch_Control_Center -manages- Missiles
FMCS_Unit -controls- Missile
Missile -has- Main_Engine
Missile -has- Navigation_Engine
Missile -has- Gyroscope
Missile -has- GPS
Missile -has- Altimeter

Time spent: 25 mins

Journal of Theoretical and Applied Information Technology
31st March 2025. Vol.103. No.6

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

APPENDIX B - EXPERT 2 BENCHMARK CLASS DIAGRAM ELEMENTS

NLRs 1: Library Information Systems I

Class [Attribute](Operation):
Customer [name, address, dateOfBirth] (login(), borrowItem(),returnItem())
Library [] (searchItem(), updateRecords())
Section [classificationMark]()
LoanItem [barCode, name] (issueItem(), reserveItem(), renewItem(), extendLoan())
types of LoanItem: Book [title, author]();
LanguageTape[title, level]()
MembershipCard [memberNo] ()
BarCodeReader [title, author] (scanBarCode(); enterBarCode())

Relationships:
Customer -issued- MembershipCard
Library -made up- Section
Customer -borrows- LoanItem
BarCodeReader -scans- MembershipCard
Section -contain- LoanItem

Time spent: 18 mins

NLRs 2: Library Information Systems II

Class [Attribute](Operation):
Book [isbn, title, author, year] (updateStatus())
Journal [title, author, year](updateStatus())
Student [id, name, major] (borrowBook(), returnBook())
Faculty [id, name, position] (borrowBook(), borrowJournal())
Librarian [id, name] (issueBook(), issueJournal())
DeputyLibrarian[id, name](receiveReturns())
Accountant [id, name] (receivePayment())
Fine [amount, date] (payAmount(), calculateAmount())

Relationships:
Books -issued to- Students.
Books -issued to- Faculty.
Journals -issued to- Faculty.
Librarian -issues- Books.
Librarian -issues- Journals.
Accountant -receives- Fines

Time spent: 14 mins

NLRs 3: Online Shoe Company

Class [Attribute] (Operation):
Customer [cust_number, name, address, payment_details, shoe_sizes, gender, age] (register(), placeOrder(), trackOrder(), cancelOrder())
Order [order_id, status] (getStatus(), updateStatus())
Stock/Shoe [shoe_id, size, price, quantity, picture, status] (checkStatus(), updateQuantity())
Report [cust_number, order_id, shoe-id] (generateReport(), sendReport())
Supplier [name, address] (fulfilOrder())
ShoeCompanySys [] (createPurchaseOrder(), receiveGoods(), cancelPurchaseOrder())
SpecialOffer [shoe_id, price, shoe_size] (getOffers(), sendOffers())

Relationships:
Customer -makes- Order
Order -contains- Stock/Shoes
Supplier -supplies- Stock/Shoes
ShoeCompanySys -order from- Supplier
ShoeCompanySys -generates- Report
Stock/Shoe -have- SpecialOffer

Time spent: 25 mins

NLRs 4: Sales Report Application

Class [Attribute] (Operation):
Salesperson [emp_number, name, ID] (login(), createOrder(), printOrder(), submitOrder())
SaleWebProgram [] (checkLoginDetails(), checkCustInfo())
Customer [name, address, status](checkCustStatus ())
Accounting [] (approveCustInfo(), approveOrder())
SalesOrder [id, date, totalPrice] (addItem(), checkItemAvailabity (), calculateTotalPrice())
Inventory [] (checkItemAvailabity(), checkItemPricing(), checkDeliveryDate(), calculateShipping(), updateInventory())
Item [item_id, name, price] (searchItem())

Relationships:
Saleperson -uses- SaleWebProgram
SaleWebProgram -creates- SalesOrder
SalesOrder -includes- Items
Inventory -contains- Items
SalesWebProgram -checks- Customer
Accounting -approves- SalesOrder

Time spent: 28 mins

NLRs 5: Missile Control System

Class [Attribute] (Operation):
LaunchControlCenter [] (setTarget())
Missile [status] ()
FMCS_Unit [height, heading, tartgetPosition] (receiveCommand(), updateStatus(), turnOnEngine(), readAltimeter(), readGyroscope(), turnOffEngine(), readGPS()), computeNavCommand())
MainEngine [mode] (receiveCommand(), updateMode())
NavigationEngine [currentPosition, targetPosition] (receiveNavCommand())
Gyroscope [] (getHeading())
GPS_Device [] (getCurrentPosition())
Altimeter [] (getHeight())

Relationships:
LaunchControlCenter -manages- Missiles
FMCS_Unit -controls- Missile
Missile -has- MainEngine
Missile -has- NavigationEngine
Missile -has- Gyroscope
Missile -has- GPS_Device
Missile -has- Altimeter

Time spent: 21 mins